

Matematyka stosowana

Grafika komputerowa I

Przemysław Kiciak
przemek@mimuw.edu.pl
<http://www.mimuw.edu.pl/~przemek>

Uniwersytet Warszawski, 2011



Streszczenie. Treścią wykładu są podstawy teoretyczne grafiki komputerowej oraz wykorzystywane w niej algorytmy i struktury danych. Kolejnymi tematami są algorytmy rasteryzacji odcinków, krzywych i wielokątów, obcinanie odcinków i wielokątów, elementy geometrii afinicznej, rzutowanie, elementy modelowania geometrycznego (w tym krzywe i powierzchnie Béziera i B-sklejane), algorytmy widoczności, podstawowe modele oświetlenia i cieniowania i metoda śledzenia promieni. Wykład uzupełniają kursy języka PostScript i programowania w OpenGL.

Wersja internetowa wykładu:

<http://mst.mimuw.edu.pl/lecture.php?lecture=gk1>

(może zawierać dodatkowe materiały)



Niniejsze materiały są dostępne na [licencji Creative Commons 3.0 Polska](#):
Uznanie autorstwa — Użycie niekomercyjne — Bez utworów zależnych.

Copyright © P.Kiciak, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 2011. Niniejszy plik PDF został utworzony 5 października 2011.



Projekt współfinansowany przez Unię Europejską w ramach
Europejskiego Funduszu Społecznego.



Skład w systemie L^AT_EX, z wykorzystaniem m.in. pakietów beamer oraz listings. Szablony podręcznika i prezentacji: Piotr Krzyżanowski; koncept: Robert Dąbrowski.

Spis treści

1. Wiadomości wstępne	7
1.1. Przegląd zastosowań grafiki komputerowej	7
1.2. Komputerowa wizualizacja przestrzeni	8
1.2.1. Wprowadzenie	8
1.2.2. Perspektywa geometryczna	9
1.2.3. Widoczność	9
1.2.4. Perspektywa powietrzna	9
1.2.5. Głębia ostrości	9
1.2.6. Oświetlenie	11
1.2.7. Cienie	11
1.2.8. Stereoscopia	11
1.2.9. Ruch	12
1.2.10. Współdziałanie innych zmysłów	12
1.3. Grafika interakcyjna	12
1.3.1. Działanie programu interakcyjnego	15
2. Podstawowe algorytmy grafiki rastrowej	20
2.1. Rysowanie odcinka	20
2.2. Rysowanie okręgu	23
2.3. Rysowanie elips	24
2.4. Wypełnianie wielokątów	26
2.5. Wypełnianie obszaru przez zalewanie	27
2.6. Algorytm z pływającym horyzontem	28
3. Obcinanie linii i wielokątów	32
3.1. Obcinanie odcinków i prostych	32
3.1.1. Wyznaczanie punktu przecięcia odcinka i prostej	32
3.1.2. Algorytm Sutherlanda-Cohana	32
3.1.3. Algorytm Lianga-Barsky'ego	34
3.1.4. Obcinanie prostych	35
3.2. Obcinanie wielokątów	35
3.2.1. Algorytm Sutherlanda-Hodgmana	35
3.2.2. Algorytm Weilera-Athertona	37
4. Elementy geometrii afinicznej	39
4.1. Przestrzenie afiniczne i euklidesowe	39
4.1.1. Określenie przestrzeni afinicznej	39
4.1.2. Iloczyn skalarny	39
4.2. Układy współrzędnych	40
4.2.1. Współrzędne kartezjańskie	40
4.2.2. Współrzędne barycentryczne	41
4.2.3. Współrzędne jednorodne	43
4.3. Przekształcenia afiniczne	45
4.3.1. Definicja i własności	45
4.3.2. Jednorodna reprezentacja przekształceń afinicznych	46
4.3.3. Przekształcanie wektora normalnego	46
4.3.4. Zmiana układu współrzędnych	47
4.3.5. Szczególne przypadki przekształceń afinicznych	48

4.3.6.	Składanie przekształceń w zastosowaniach graficznych	52
4.3.7.	Rozkładanie przekształceń	53
4.3.8.	Obroty, liczby zespolone i kwaterniony	54
5.	Rzutowanie równoległe, perspektywiczne i inne	60
5.1.	Rzutowanie równoległe	60
5.2.	Rzutowanie perspektywiczne	61
5.3.	Algorytm rzutowania	62
5.4.	Stereoskopia	65
5.5.	Panorama i inne rzuty	66
6.	Elementy modelowania geometrycznego	68
6.1.	Krzywe Béziera	68
6.1.1.	Określenie krzywych Béziera	68
6.1.2.	Algorytm de Casteljau	68
6.1.3.	Własności krzywych Béziera	70
6.2.	Krzywe B-sklejane	72
6.2.1.	Określenie krzywych B-sklejanych	72
6.2.2.	Algorytm de Boora	73
6.2.3.	Podstawowe własności krzywych B-sklejanych	74
6.2.4.	Wstawianie węzła	76
6.2.5.	Krzywe B-sklejane z węzłami równoodległymi	77
6.3.	Powierzchnie Béziera i B-sklejane	78
6.3.1.	Płaty tensorowe	78
6.3.2.	Płaty trójkątne	80
6.3.3.	Metody siatek	81
6.4.	Krzywe i powierzchnie wymierne	82
6.5.	Modelowanie powierzchni i brył	84
6.5.1.	Zakreślanie	84
6.5.2.	Powierzchnie rozpinane	85
6.5.3.	Powierzchnie zadane w postaci niejawnej	86
7.	Reprezentacje scen trójwymiarowych	90
7.1.	Prymitywy i obiekty złożone	90
7.2.	Reprezentowanie brył wielościennych	90
7.3.	Konstrukcyjna geometria brył	92
7.3.1.	Wyznaczanie przecięcia wielościanów	94
7.4.	Drzewa i grafy scen	95
8.	Drzewa binarne, czwórkowe i ósemkowe	98
8.1.	Drzewa czwórkowe	98
8.1.1.	Konstrukcyjna geometria figur płaskich	99
8.1.2.	Algorytm widoczności	100
8.1.3.	Transmisja obrazów i MIP-mapping	100
8.2.	Drzewa ósemkowe	101
8.3.	Drzewa binarne	102
8.4.	Binarny podział przestrzeni	102
9.	Algorytmy widoczności	104
9.1.	Rodzaje algorytmów widoczności	104
9.2.	Algorytmy przestrzeni danych	105
9.2.1.	Odrzucanie ścian „tylnych”	105
9.2.2.	Obliczanie punktów przecięcia odcinków w przestrzeni	106
9.2.3.	Algorytm Ricciiego	107
9.2.4.	Algorytm Weilera-Athertona	108
9.2.5.	Algorytm Appela	109
9.2.6.	Algorytm Hornunga	110

9.3.	Algorytmy przestrzeni obrazu	110
9.3.1.	Algorytm malarza	110
9.3.2.	Algorytm binarnego podziału przestrzeni	111
9.3.3.	Algorytm BSP wyznaczania cieni	112
9.3.4.	Algorytm z buforem głębokości	112
9.3.5.	Algorytm przeglądania liniami poziomymi	113
9.3.6.	Algorytm cieni z buforem głębokości	115
10.	Podstawowe modele oświetlenia i metody cieniowania	116
10.1.	Oświetlenie	116
10.1.1.	Oświetlenie bezkierunkowe	116
10.1.2.	Odbicie lambertowskie	116
10.1.3.	Model Phong	117
10.2.	Cieniowanie	118
10.2.1.	Cieniowanie stałe	118
10.2.2.	Cieniowanie Gourauda	119
10.2.3.	Cieniowanie Phong	119
10.2.4.	Tekstura odkształceń	120
11.	Śledzenie promieni	121
11.1.	Zasada działania algorytmu	121
11.2.	Wyznaczanie przecięć promieni z obiektami	122
	Przykład 1	123
	Przykład 2	124
	Przykład 3	125
11.3.	Techniki przyspieszające	126
11.3.1.	Bryły otaczające i hierarchia sceny	126
11.3.2.	Drzewa ósemkowe	127
11.3.3.	Połączenie śledzenia promieni z z-buforem	128
11.4.	Śledzenie promieni i konstrukcyjna geometria brył	129
11.5.	Antyaliasing	129
11.5.1.	Antyaliasing przestrzenny	130
11.5.2.	Antyaliasing czasowy	131
11.6.	Symulacja głębi ostrości	131
11.7.	Układy cząsteczek	133
11.8.	Implementacja programu do śledzenia promieni	134
12.	Przestrzeń barw i jej układy współrzędnych	136
12.1.	Podstawy kolorymetrii	136
12.2.	Diagram CIE	137
12.3.	Współrzędne <i>RGB</i> i <i>YIQ</i>	140
12.4.	Współrzędne <i>CMY</i> i <i>CMYK</i>	141
12.5.	Współrzędne <i>HSV</i> i <i>HLS</i>	142
13.	Język PostScript	145
13.1.	Wprowadzenie	145
13.2.	Przykład wstępny	146
13.3.	Operatory PostScriptu (wybrane dość arbitralnie)	148
13.3.1.	Operatory arytmetyczne	148
13.3.2.	Operacje na stosie argumentów	148
13.3.3.	Operatory relacyjne i logiczne	149
13.3.4.	Operatory sterujące	149
13.3.5.	Operatory konstrukcji ścieżki	150
13.3.6.	Operatory rysowania	150
13.3.7.	Operatory związane ze stanem grafiki	150
13.4.	Napisy i tworzenie obrazu tekstu	151
13.5.	Słowniki	154
13.6.	Stosy interpretera	156

13.7. Operatory konwersji	156
13.8. Przekształcenia afiniczne	157
13.9. Operacje na tablicach	161
13.10. Obrazy rastrowe	162
13.11. Programowanie L-systemów	163
13.12. PostScript obudowany	170
14. OpenGL — wprowadzenie	172
14.1. Informacje ogólne	172
14.1.1. Biblioteki procedur	173
14.1.2. Reguły nazewnictwa procedur	173
14.1.3. OpenGL jako automat stanów	174
14.2. Podstawowe procedury rysowania	174
14.2.1. Wyświetlanie obiektów	176
14.3. Przekształcenia	177
14.3.1. Macierze przekształceń i ich stosy	177
14.3.2. Rzutowanie	177
14.4. Działanie GLUTa	180
14.4.1. Schemat aplikacji GLUTa	180
14.4.2. Przegląd procedur GLUTa	182
14.4.3. Współpraca okien z OpenGL-em	184
14.4.4. Figury geometryczne dostępne w GLUCie	184
14.5. Określanie wyglądu obiektów na obrazie	185
14.5.1. Oświetlenie	185
14.5.2. Własności powierzchni obiektów	186
14.5.3. Powierzchnie przezroczyste	187
14.5.4. Mgła	188
14.6. Ewaluatory	188
14.6.1. GL — krzywe i powierzchnie Béziera	188
14.6.2. GLU — krzywe i powierzchnie B-sklejane	189
14.7. Bufor akumulacji i jego zastosowania	191
14.7.1. Obsługa bufora akumulacji	191
14.7.2. Antyaliasing przestrzenny	192
14.7.3. Symulacja głębi ostrości	193
14.8. Nakładanie tekstury	193
14.8.1. Tekstury dwuwymiarowe	193
14.8.2. Mipmapping	194
14.8.3. Tekstury jednowymiarowe	195
14.8.4. Tekstury trójwymiarowe	195
14.8.5. Współrzędne tekstury	195
14.9. Listy obrazowe	196
14.9.1. Wiadomości ogólne	196
14.9.2. Rysowanie tekstu	197
14.10. Implementacja algorytmu wyznaczania cieni	198
Literatura	205

1. Wiadomości wstępne

1.1. Przegląd zastosowań grafiki komputerowej

Można podać tylko dolne oszacowanie liczby zastosowań grafiki komputerowej. Nawet gdyby poniższa lista przez chwilę była pełna (nie należy mieć takich złudzeń), po jej wydrukowaniu pojawiły się nowe.

- Projektowanie wspomagane komputerowo (CAD, od ang. *Computer Aided Design*).
 - Rysowanie urządzeń i ich części; planowanie ich rozmieszczenia.
 - Sprawdzanie, czy części dobrze współpracują i czy urządzenie daje się zmontować.
 - Wizualizacja wyników badań modeli komputerowych (np. wykresy naprężeń konstrukcji, temperatury, odkształceń itp.).
 - Kontrola ergonomii i estetyki.
 - Wspomaganie projektowania procesów produkcyjnych (CAM, ang. *Computer Aided Manufacturing*), np. planowanie i symulacja obróbki na frezarkach sterowanych numerycznie.
 - Dobieranie barwy lakieru (np. nadwozi samochodowych).
 - Sporządzanie planów architektonicznych, połączone z możliwością nieograniczonego oglądania budynków z każdej strony, w różnych warunkach oświetlenia (zależnych od pory dnia, pogody, pory roku i związanego z nią wyglądu okolicznych zarośli).
 - Projektowanie tkanin i haftów (produkowanych następnie przez urządzenia sterowane przez komputer).
 - Wytwarzanie materiałów prezentacyjnych (oferty, reklamy, instrukcje obsługi).
- Grafika prezentacyjna.
 - Wykresy danych statystycznych (statystyka, ekonomia, wyniki eksperymentów fizycznych, biologicznych, medycznych, badań socjologicznych itp.).
 - Geograficzne bazy danych (systemy informacji przestrzennej, mapy fizyczne, geologiczne, gospodarcze, administracyjne, nie mówiąc już o wojskowych).
 - Prognozy pogody.
 - Prognozy finansowe.
 - Wizualizacja przebiegu wypadków i katastrof.
- Wizualizacja naukowa.
 - Wykresy funkcji (oczywiście, były one rysowane na długo przed wynalezieniem komputerów, a nawet w ogromnym stopniu do tego wynalazku się przyczyniły; przyczyniły się też do wielu innych odkryć i wynalazków, także wtedy, gdy ich rysowaniem zajęły się komputery).
 - Płaskie i przestrzenne wykresy pól wektorowych, stałych lub zmiennych w czasie, otrzymanych z pomiarów lub z obliczeń wykonanych też przez komputer.
 - Obrazy pochodzące z symulacji zjawisk fizycznych.
- Wizualizacja medyczna.
 - Obrazy choroby wykonane na podstawie badań z użyciem tomografu, gamma-kamery, ultrasonografu, urządzeń mierzących rezonans magnetyczny (NMR) itd.
 - Planowanie operacji.

- Trening przed operacją (chirurg wykonuje czynności takie jak podczas operacji i ogląda mniej więcej to, co później zobaczy tnąc i zszywając prawdziwego pacjenta).
- Zdalna asysta przy operacji (chirurg wykonuje operację wspomagany radami albo pod dyktando bardziej doświadczonego lekarza, który jest oddalony od sali operacyjnej).
- Edukacja.
 - Multimedialne podręczniki i zbiory zadań.
 - Multimedialne encyklopedie.
 - Symulatory lotu, jazdy samochodem, pociągiem.
- Kryminalistyka.
 - Wykonywanie portretów pamięciowych.
- Sztuka.
 - Plastyka.
 - Sztuka użytkowa.
 - Komputerowy skład tekstu (ang. *desktop publishing*, *DTP*).
- Rozrywka i zabawa
 - Wytwarzanie filmów (animowanych, ale także „zwykłych”).
 - Gry komputerowe.
- Interfejs człowiek–komputer
 - Systemy okienkowe (kiedyś to się nazywało WIMPS (ang. *widg*ets, *icons*, *menus*, *pointers*), potem (zapewne w ramach politycznej poprawności) zmieniło nazwę na GUI (ang. *graphical user interface*). Niektóre wihajstry (*widg*ets) są bardzo wymyślne.
 - Komputery już mogą przyjmować polecenia wydawane głosem i odpowiadać za pomocą syntezatora mowy. Miłym uzupełnieniem jest obraz ludzkiej głowy, która mówi. Taki interfejs użytkownika może mieć wiele zastosowań w edukacji, a także w komunikacji z użytkownikami niepełnosprawnymi.

1.2. Komputerowa wizualizacja przestrzeni

1.2.1. Wprowadzenie

Grafika jest w znacznej części sztuką wykonywania obrazów płaskich, których oglądanie powoduje powstanie u widza *złudzenia* oglądania przedmiotów trójwymiarowych. W szczególności dotyczy to grafiki komputerowej.

Cechy obrazów, które przyczyniają się do powstania takiego złudzenia, są ściśle związane z cechami ludzkiego zmysłu wzroku. Istnieją zwierzęta wyposażone w zmysł wzroku, u których obrazy (przez ludzi postrzegane jako obrazy obiektów trójwymiarowych) nie są odbierane w taki sposób.

Najważniejszym ludzkim narządem biorącym udział w procesie widzenia jest **mózg**. Uczenie się polega z grubsza na tworzeniu w mózgu pewnych wyobrażeń na temat rozmaitych przedmiotów. Podczas ponownego oglądania przedmiotu można go **rozpoznać**, co polega na „dopasowaniu” do informacji napływającej z oczu któregoś z wyobrażeń. Za powyższą hipotezą przemawia fakt, że ilość informacji zbieranej przez receptory wzrokowe jest rzędu 10^9 bitów na sekundę, ale nerwy przewodzące sygnały z oczu do mózgu mają znacznie mniejszą przepustowość. Jeszcze więcej informacji mózg eliminuje podczas przetwarzania sygnału dochodzącego przez te nerwy. To, że widzowi się wydaje, że odbiera więcej informacji, jest złudzeniem. Mózg uzupełnia napływający strumień informacji o informacje zapamiętane wcześniej. Dzięki temu m.in. nie zdajemy sobie sprawy z istnienia plamki ślepej, tj. otworu w siatkówce oka, przez który przechodzi nerw wzrokowy (w tym miejscu nie ma receptorów, tj. czopków ani pręcików).

Powyższe spostrzeżenie ma dwie ważne konsekwencje. Po pierwsze, rozpoznawanie obrazów odbywa się metodą „przyrostową” dzięki czemu mózg jest w stanie przetwarzać napływające informacje „w czasie rzeczywistym”, dopasowując do otrzymywanych danych pamiętane wyobrażenia stopniowo.

Po drugie, obraz nie musi doskonale odpowiadać jakiemuś przedmiotowi. Jeśli nie odpowiada żadnemu dotychczas widzianemu przedmiotowi, to widz może utworzyć nowe wyobrażenie (tj. nauczyć się czegoś nowego). Jeśli odpowiada znanemu przedmiotowi w sposób niedokładny, to widz może „naciągnąć” dopasowanie otrzymywanej informacji do wyobrażenia tego przedmiotu i myśleć, że ogląda rzecz znaną. W rezultacie widz może być przekonany, że np. szkic (wykonany długopisem) i fotografia jakiegoś przedmiotu przedstawiają tę samą rzecz, mimo, że to są ewidentnie różne obrazy.

Wpływ na postrzeganie obrazów jako wizerunków scen trójwymiarowych mają

- Perspektywa geometryczna,
- Widoczność,
- Oświetlenie,
- Cienie,
- Perspektywa powietrzna,
- Głębia ostrości,
- Stereoskopia,
- Ruch,
- Współdziałanie innych zmysłów.

1.2.2. Perspektywa geometryczna

Perspektywa geometryczna jest związana z odpowiednim rzutem przestrzeni na płaszczyznę. Jej podstawy były stopniowo odkrywane przez artystów malarzy; najszybszy rozwój wiedzy na ten temat miał miejsce w okresie Renesansu. Obecnie ludzie mają masowy kontakt z fotografiami, przez co są przyzwyczajeni do obrazów otrzymanych przez rzutowanie perspektywiczne. Rzuty perspektywiczne są łatwe do oprogramowania na komputerze; łatwość ta jest nieco złudna; łatwo jest z perspektywą „przesadzić”.

1.2.3. Widoczność

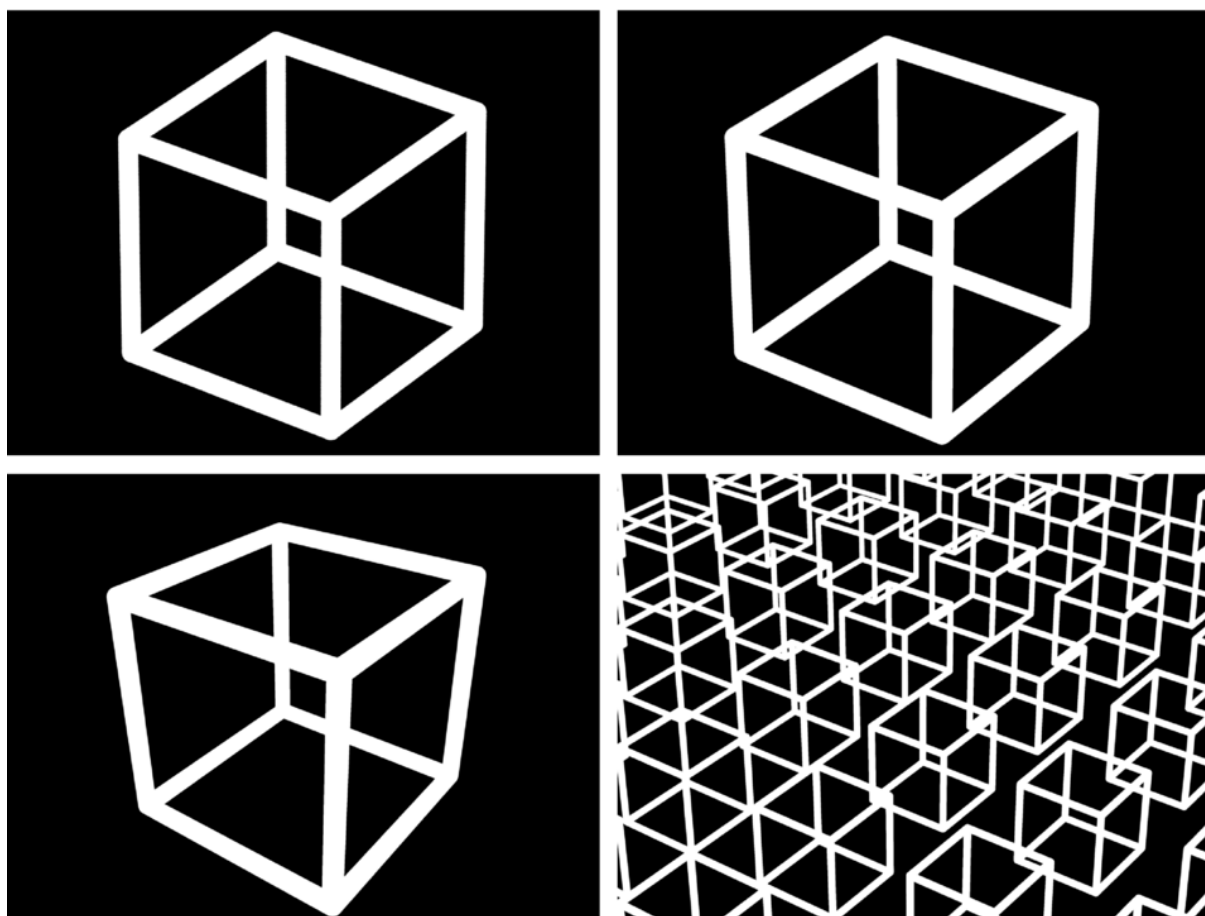
Symulowanie widoczności jest konieczne do tego, aby oglądanie obrazu mogło dostarczyć podobnych wrażeń jak oglądanie „prawdziwych” przedmiotów. Gdyby na obrazie narysować „wszystko”, to nie byłoby na nim widać niczego.

1.2.4. Perspektywa powietrzna

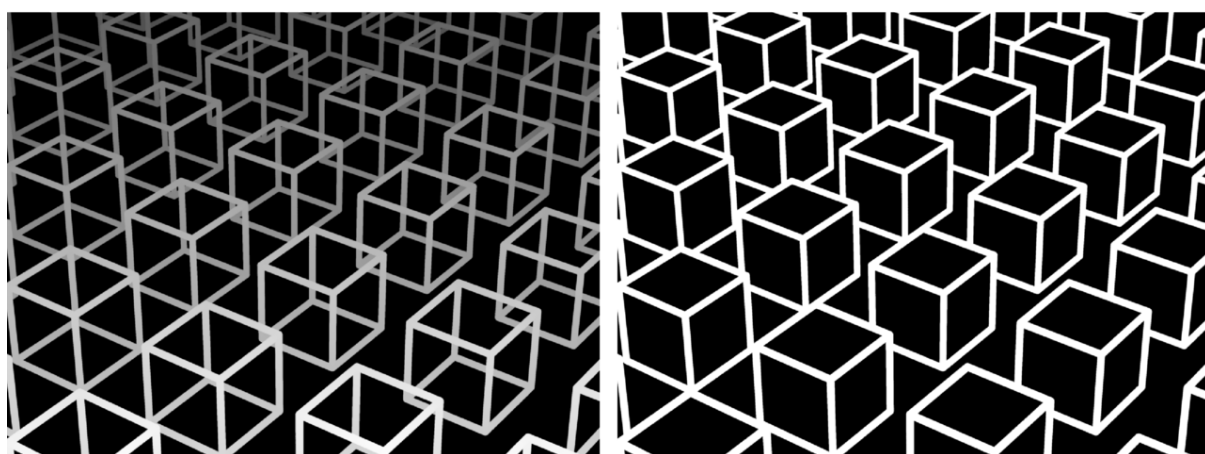
Perspektywa powietrzna jest to dostrzegalny wpływ odległości obserwatora od przedmiotu na jego (tj. przedmiotu) wygląd na obrazie. Może to być skutek obecności mgły, której warstwa między obserwatorem i przedmiotem tym bardziej zmienia wygląd przedmiotu im jest grubsza. Warto zwrócić uwagę, że „bezpośrednia” informacja o odległości obserwatora od punktu przedstawionego na obrazie jest słabo przyswajana przez osobę oglądającą ten obraz, tj. osoba ta raczej nie ma wrażenia, że widzi kształt przedmiotu.

1.2.5. Głębia ostrości

Pojęcie głębi ostrości zostało gruntownie zbadane w związku z rozwojem optyki i fotografią. Zarówno ludzkie oko, jak i aparat fotograficzny lub kamera, mają obiektywy (soczewki) o pewnej regulowanej średnicy. Taki obiektyw skupia w jednym punkcie rzutni (siatkówki oka, filmu,



Rysunek 1.1. Rzut równoległy i rzut perspektywiczny.

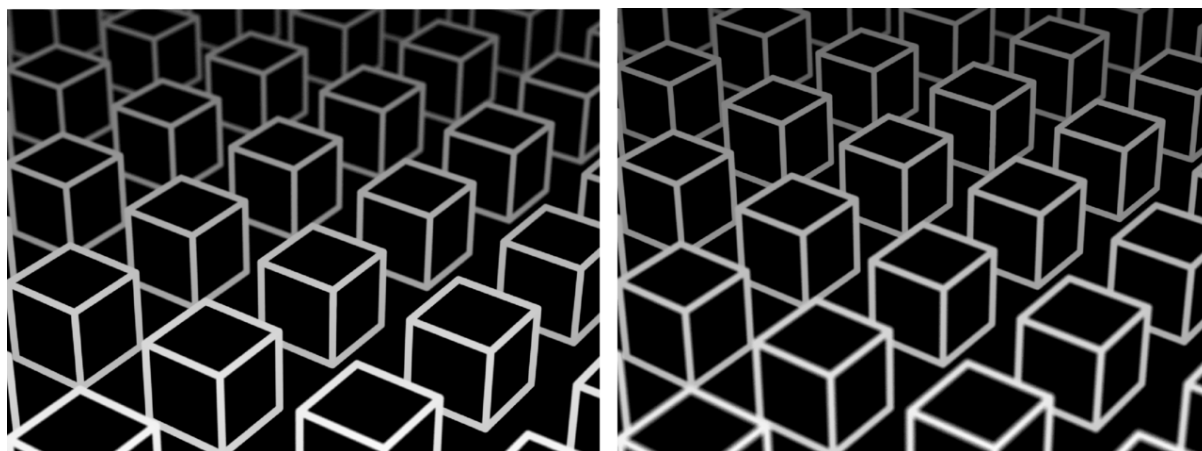


Rysunek 1.2. Uwzględnienie perspektywy powietrznej i widoczności.

matrycy CCD lub co tam jeszcze będzie wynalezione) światło wychodzące z punktu położonego w pewnej ustalonej odległości od obiektywu i dlatego trzeba nastawiać aparat (lub wytyęgać wzrok) na wybraną odległość. Obrazy punktów położonych bliżej lub dalej są nieostre.

Ograniczona głębia ostrości aparatów fotograficznych to *nie jest* kwestia ich niedoskonałości technicznej (choć czasem bywa tak postrzegana). Jej istnienie wiąże się ze świadomym wybiera-

niem zasadniczego tematu zdjęcia przez jego autora. Obrazy, na których wszystkie przedmioty są ostre niezależnie od ich odległości, sprawiają często nienaturalne wrażenie i drażnią widzów.



Rysunek 1.3. Głębia ostrości — różne nastawienia odległości.

1.2.6. Oświetlenie

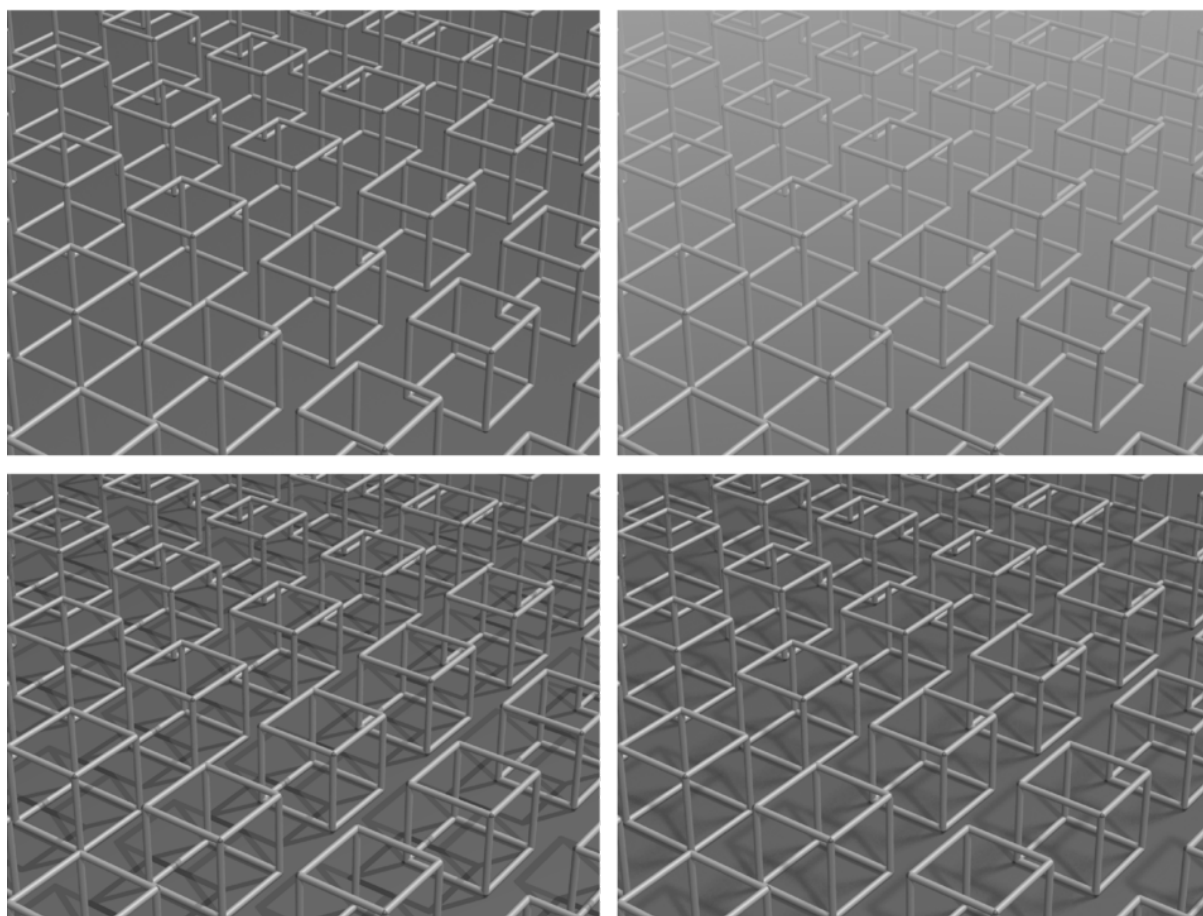
Najważniejszym po widoczności czynnikiem wpływającym na wrażenie trójwymiarowości narysowanych przedmiotów jest oświetlenie. Fascynującym zjawiskiem jest ogromna zdolność ludzi do rozpoznawania kształtu przedmiotów niezależnie od kierunku padania światła i od własności optycznych powierzchni tych przedmiotów.

1.2.7. Cienie

Wykonanie obrazu z uwzględnieniem oświetlenia światłem padającym z ustalonego kierunku nie wystarczy do określenia położenia obiektów przedstawionych na obrazie względem siebie. Bardzo pomaga tu uwzględnienie cieni, które te obiekty rzucają. Dla wrażenia trójwymiarowości wystarczy, aby cienie były „ostre”, tj. takie jak gdyby źródła światła były punktowe (co jest dosyć łatwe do osiągnięcia w grafice komputerowej). Ponieważ w rzeczywistości takich źródeł światła prawie nie ma, więc na bardziej realistycznych obrazach granice cieni będą nieostre, tj. pojawiają się półcienie. Otrzymanie tego efektu w grafice komputerowej jest trudniejsze i bardziej pracochłonne, ale możliwe.

1.2.8. Stereoskopia

Stereoskopia to oglądanie przedmiotów dwójkiem oczu, z których każde widzi nieco inny obraz. Ludzie dokonali wielu wynalazków, które umożliwiają oglądanie każdym okiem innego obrazu, w celu wytworzenia wrażenia stereoskopowego, zaczynając od fotoplastykonu i anaglifów (obrazów dwubarwnych oglądanych przez okulary z kolorowymi szybami) do kasków z wbudowanymi miniaturowymi monitorami i okularów ciekłokrystalicznych. Dla widzenia stereoskopowego podstawowe znaczenie ma fakt, że ludzkie oczy mają dość dużą głębię ostrości i dla mózgu od ostrości obrazów większe znaczenie mają drobne różnice w wyglądzie przedmiotu widzianego z różnych miejsc.



Rysunek 1.4. Oświetlenie, perspektywa powietrzna i cienie ostre i rozmyte.

1.2.9. Ruch

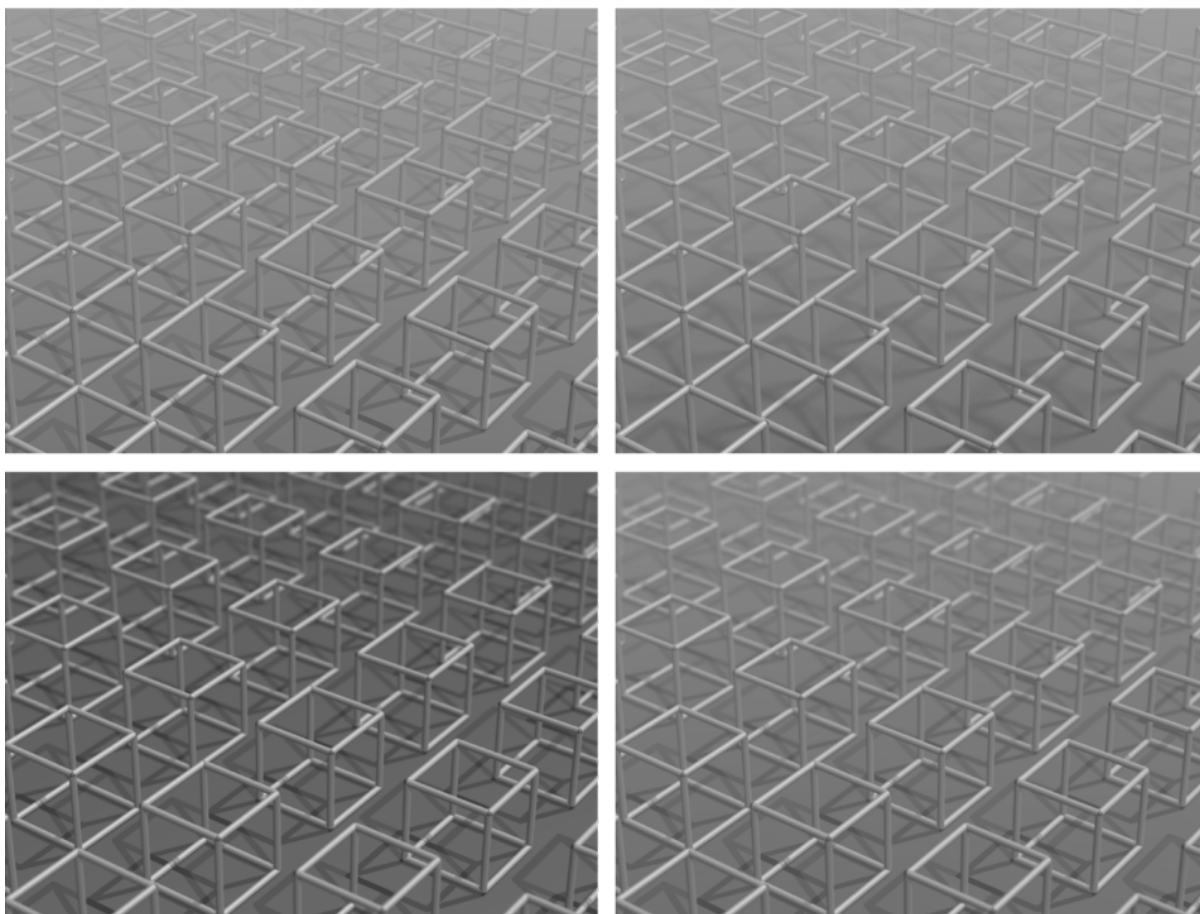
Ruch obserwatora stwarza okazję do obejrzenia przedmiotu z różnych miejsc. Rozszerza to stereoskopię, ponieważ „widzenie” przedmiotu jest w istocie utworzeniem przez mózg obserwatora pewnego modelu (albo wyobrażenia) tego przedmiotu na podstawie wielu kolejno zgromadzonych informacji na jego temat.

1.2.10. Współdziałanie innych zmysłów

Inne zmysły dostarczają człowiekowi w sumie czterokrotnie mniej informacji niż wzrok, jeśli jednak informacje te są zgodne ze wzrokowymi, to całościowe wrażenie może być bardzo silne. Najważniejsze zmysły współpracujące ze wzrokiem w oglądaniu przedmiotów to słuch, dotyk, i zmysł równowagi. Są podejmowane próby syntetyzowania odpowiednich bodźców dla ludzi (nazywa się to **sztuczną rzeczywistością**), ale zreferowanie tego wykracza poza temat tego kursu i również poza moje kompetencje.

1.3. Grafika interakcyjna

Każdy program komputerowy (nie tylko „graficzny”) ma pewien określony sposób obsługi. Kolejne możliwości są coraz bardziej pracochłonne dla programisty, ale coraz atrakcyjniejsze dla użytkownika.



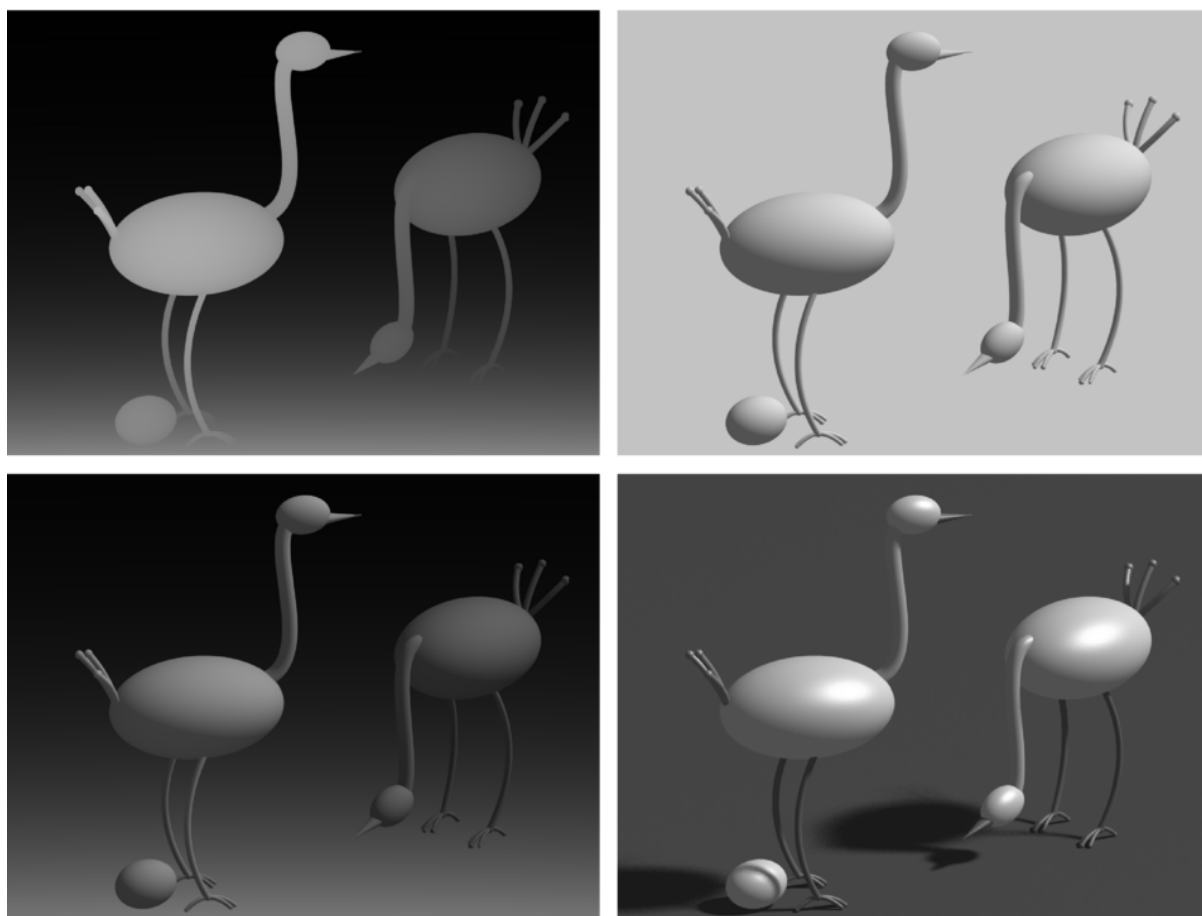
Rysunek 1.5. Dołączenie głębi ostrości.

Tryb wsadowy. Nazwa pochodzi z czasów, gdy użytkownicy komputerów, w tym programiści, pisali program lub dane na arkuszach (tzw. „szytkach”) i zostawiali je wykwalifikowanemu personelowi, który przepisywał zawartość arkuszy na klawiaturze perforatora kart. Po sprawdzeniu (przez użytkownika) pliku (stąd nazwa „plik”) podziurkowanych kart, uzupełnionego o karty sterujące wykonaniem zadania przez system operacyjny, był on (tj. plik) *wsadzany* do czytnika komputera (przez wykwalifikowany personel). Następnego dnia można było odebrać tzw. wydruk i przystąpić do poprawiania programu¹.

Tryb wsadowy polega na tym, że w trakcie działania programu nie można ingerować w jego przebieg (z wyjątkiem, być może, przerwania go). Obecnie w grafice komputerowej tryb ten jest wciąż stosowany. Na przykład, po przygotowaniu publikacji do druku, system DTP produkuje plik zawierający opis stron do wydrukowania. Opis ten jest programem (najczęściej w języku PostScript lub PDF) wykonywanym właśnie w trybie wsadowym przez tzw. RIP (ang. *raster image processor*), tj. specjalizowany komputer sterujący urządzeniem drukującym lub naświetlarką.

Inny ważny przykład, to program wykonujący obraz sceny trójwymiarowej (albo wiele obrazów — klatek filmu) metodą śledzenia promieni, lub obliczający oświetlenie sceny metodą bilansu energetycznego. Obliczenia te są dość długotrwałe (rzędu minut lub godzin). Poza ustawieniem niewielkiej liczby parametrów podczas uruchamiania tego programu, osoba obsługująca nie ma wpływu na jego wykonanie.

¹ Proszę się nie śmiać. Ja to opisałem z szacunkiem.



Rysunek 1.6. Inny przykład: perspektywa powietrzna, oświetlenie, oba elementy i cienie.

Tryb interakcji biernej. Program po uruchomieniu wykonuje swoje instrukcje, z których niektóre są instrukcjami czytania danych lub poleceń. Po ich podaniu (np. wpisaniu i naciśnięciu klawisza <Enter>) użytkownik nie ma wpływu na to, kiedy program da mu kolejną okazję do sterowania przebiegiem obliczeń.

W tym trybie działają najczęściej programy, które wykonują czasochłonne obliczenia zwieńczone wykonaniem rysunku. Możliwość ingerencji w działanie programu w takiej sytuacji bywa nieistotna. Często też tryb taki przydaje się podczas uruchamiania procedur, które mają być następnie wbudowane w program umożliwiający czynną interakcję. Program pracujący w trybie wsadowym lub w trybie interakcji biernej jest znacznie prostszy do poprawiania przy użyciu debuggera, a ponadto szczególne dane wejściowe dla uruchamianej procedury, dla których chcemy się przyjrzeć obliczeniom, mogą być „zaszyte” w programie na stałe, dzięki czemu możemy powtarzać eksperymenty dla tych samych danych wielokrotnie. Generując dane wejściowe za pomocą myszy w trybie interakcji czynnej, nie jesteśmy zwykle w stanie powtórzyć wszystkich czynności dokładnie co do piksela.

Tryb interakcji czynnej. W każdej chwili działania programu użytkownik może spowodować zdarzenie, na które program niezwłocznie reaguje. Program w różnych chwilach swojego działania może udostępniać różne zestawy możliwych reakcji. Użytkownik może wykonywać akcje nieprzewidywalne, a program ma na nie reagować sensownie i nie dać się wywrócić ani zawiesić.

1.3.1. Działanie programu interakcyjnego

Tryb interakcji czynnej jest ściśle związany z paradygmatem *programowania obiektowego*. Jego istotą jest specyficzny sposób traktowania danych. W „zwykłym” programowaniu *imperatywnym* mamy do czynienia z procedurami, które otrzymują dane jako parametry. Procedura wykonuje obliczenie, ewentualnie przypisuje danym nowe wartości i zwraca sterowanie, po czym może być ponownie wywołana z nowymi danymi. Dane w programowaniu obiektowym to są *obiekty*, które oprócz wartości mają określone sposoby reakcji na *zdarzenia*. Procedura, która realizuje reakcję na zdarzenie, jest częścią obiektu. W tej terminologii wywołanie takiej procedury (zwanej *metodą*) nazywa się *wysłaniem komunikatu*. Metoda może zmienić stan obiektu i wysłać komunikaty do innych obiektów (a także do swojego), czyli wywołać ich metody.

Warto mieć na uwadze, że wybór języka jest w programowaniu rzeczą drugorzędną wobec podstawowych decyzji projektowych. Istnieją języki (na czele z C++), które zawierają „konstrukcje obiektowe”, a także biblioteki obiektów (w sensie podanym wyżej) zrealizowanych w sposób podany wyżej i gotowych do użycia w programach. Można jednak pisać programy „nieobektowe” w C++, jak również realizować programy z obiektami w „nieobektowym” języku takim jak Pascal lub C². Wybór języka ma przede wszystkim wpływ na wygodę programisty i jest podyktowany przez środowisko, w jakim on pracuje oraz dostępne biblioteki procedur. W realnym świecie często bywa narzucany przez klienta, który zamawia program.

W programowaniu obiektowym może istotnie pomóc pojęcie **automatu skończonego**. Automaty skończone są często stosowane w rozpoznawaniu zdań języków regularnych, czym tu się nie zajmujemy, natomiast użyjemy tego pojęcia do implementacji graficznego dialogu z użytkownikiem. Formalna definicja jest następująca: Automat skończony jest piątką elementów, (V, S, S_k, s_0, f) , w której symbol V oznacza skończony zbiór zwany **alfabetem**, S oznacza **zbiór stanów**, również skończony, podzbiór S_k zbioru S jest **zbiorem stanów końcowych**, element s_0 zbioru $S \setminus S_k$ jest **stanem początkowym**, zaś funkcja $f: S \setminus S_k \times V \rightarrow S$ jest zwana **funkcją przejścia**. Taki automat zaczyna działanie w stanie s_0 . Działanie to polega na pobieraniu i przetwarzaniu kolejnych symboli alfabetu. Automat, który w stanie s_i otrzymuje symbol c_k , przechodzi do stanu $s_j = f(s_i, c_k)$, przy czym jeśli jest to jeden ze stanów końcowych, to automat kończy działanie.

Przypuśćmy, że naszym obiektem jest okno programu, w którym wyświetlony jest pewien rysunek. W obszarze okna są wyróżnione pewne punkty, które użytkownik może „chwycić” wskazując je kursorem i naciskając przycisk, a następnie „przemieszczać” do nowego położenia i „puszczać”. W najprostszych przypadkach stworzymy automat o trzech stanach: stan początkowy „NIC”, stan „PRZESUWA” i stan trzeci, do którego przejście polega na zatrzymaniu programu.

Alfabet zdarzeń przełączających między stanami składa się z czterech elementów. Pierwszym jest naciśnięcie guzika myszy, pod warunkiem, że kursor znajduje się w pobliżu (w granicach określonej tolerancji) od jednego z wyróżnionych punktów. Jeśli automat jest w stanie „NIC”, to automat przechodzi do stanu „PRZESUWA”. Drugie zdarzenie to przesunięcie myszy (i zmiana położenia kursora). Z każdego ze stanów automat przechodzi do tego samego stanu, przy czym jeśli bieżącym stanem jest „PRZESUWA”, to program wybranemu punktowi przypisuje nowe współrzędne, obliczone na podstawie nowej pozycji kursora, a następnie uaktualnia rysunek w oknie. Trzecie zdarzenie to zwolnienie guzika — automat ze stanu „PRZESUWA” przechodzi do stanu „NIC”. Zdarzenie czwarte, czyli zatrzymanie programu, może być spowodowane np. przez naciśnięcie klawisza.

Uwaga: Nie musimy, choć możemy, za element alfabetu uznać zdarzenia polegającego po pro-

² Podobnie wygląda sprawa używania rekurencji w języku Fortran, który nie dopuszcza rekurencyjnych wywołań procedur i funkcji

stu na naciśnięciu przycisku. Możemy zresztą określić alfabet, którego elementami są różne zdarzenia, z których każde polega na naciśnięciu przycisku gdy kursor wskazuje inny punkt.

Dodatkową komplikację programów działających w środowiskach okienkowych powoduje fakt, że zwykle użytkownik pracuje jednocześnie z wieloma programami, z których każdy może wyświetlać wiele okien. Powoduje to dwa utrudnienia: okno może nie być widoczne w całości (albo wcale), przez co program zwykle nie może wyświetlać rysunków bezpośrednio na ekranie, a ponadto potrzeba wyświetlenia rysunku może być spowodowana zdarzeniami związanymi z innym programem (np. gdy okno innego programu zostało przesunięte lub zamknięte). Są różne rozwiązania tego problemu, przy czym żadne z nich nie jest uniwersalne. Program dla każdego okna powinien utworzyć **reprezentację rysunku**, która może mieć postać obrazu rastrowego³, lub tzw. **listy obrazowej** — struktury danych umożliwiającej szybkie wyświetlenie przechowywanych w niej figur geometrycznych. Pierwsza z metod ma tę zaletę, że ukrywa pośrednie etapy rysowania (np. skasowane tło), które objawiają się jako migotanie.

W odpowiedzi na zdarzenia, które powinny spowodować zmianę rysunku w oknie (np. stan „PRZESUWA” może być wyróżniony zmianą koloru punktu wybranego do przesuwania, a po każdej zmianie położenia mamy nowy rysunek) program powinien utworzyć nową reprezentację rysunku. Następnie, jeśli system okienkowy to umożliwia, program od razu wyświetla ten rysunek. Podany niżej przykładowy program pracuje w ten sposób. Istnieją też systemy okien, w których aby zmienić rysunek w oknie, należy zawiadomić system, że zawartość pewnego obszaru w oknie jest „nieważna”. W odpowiedzi system wyznacza część tego obszaru, która nie jest zasłonięta przez inne okna i wysyła do programu komunikat, który jest poleceniem „odtworzenia” rysunku na ekranie w wyznaczonej części. Każdy system wysyła do programu taki komunikat również w przypadku odsłonięcia części okna w wyniku zmiany położenia okien na ekranie.

Zbadajmy przykład kompletnego programu działającego zgodnie z opisanymi wyżej zasadami. Program ten jest napisany w języku C i korzysta z biblioteki Xlib, tj. najbardziej „niskopoziomowego” interfejsu programisty w systemie XWindows.

```
#include <stdlib.h>
#include <X11/Xlib.h>

int ekran;
Window okno;
GC kontgraf;
XEvent zdarzenie;
unsigned int front, tlo;

#define SZER 420
#define WYS 300

#define NIC 0
#define PRZESUWA 1
#define KONIEC 2
int stan = NIC;

#define LICZBA_PUNKTOW 5
XPoint punkt[LICZBA_PUNKTOW] =
    {{10,150},{110,150},{210,150},{310,150},{410,150}};
int np;
```

³ Brzydko nazywanego, cytuję, „mapą bitową”.


```

void Przygotuj ( void )
{
    stacja = XOpenDisplay ( "" );
    ekran = DefaultScreen ( stacja );
    front = WhitePixel ( stacja, ekran );
    tlo = BlackPixel ( stacja, ekran );
    okno = XCreateSimpleWindow ( stacja,
                                DefaultRootWindow(stacja),
                                100, 100, SZER, WYS, 7, front, tlo );
    kontgraf = XCreateGC ( stacja, okno, 0, 0 );
    XSetBackground ( stacja, kontgraf, tlo );
    XSetForeground ( stacja, kontgraf, front );
    XMapRaised ( stacja, okno );
    XSelectInput ( stacja, okno,
                  ButtonPressMask | ButtonReleaseMask |
                  PointerMotionMask | KeyPressMask |
                  ExposureMask );
} /*Przygotuj*/

int ZnajdzNajblizszyPunkt ( int x, int y )
{
    int d, e;
    int i, k;

    d = 10; k = -1;
    for ( i = 0; i < LICZBA_PUNKTOW; i++ ) {
        e = abs(x-punkt[i].x) + abs(y-punkt[i].y);
        if ( e < d )
            { d = e; k = i; }
    }
    return k;
} /*ZnajdzNajblizszyPunkt*/

void GdzieKursor ( int *x, int *y )
{
    int xr, yr;
    unsigned int maska;
    Window rw, cw;

    XQueryPointer ( stacja, okno, &rw, &cw, &xr, &yr,
                    x, y, &maska );
} /*GdzieKursor*/

void Narysuj ( void )
{
    XSetForeground ( stacja, kontgraf, tlo );
    XFillRectangle ( stacja, okno, kontgraf,
                    0, 0, SZER, WYS );
    XSetForeground ( stacja, kontgraf, front );
    XDrawLines ( stacja, okno, kontgraf,
                punkt, LICZBA_PUNKTOW, CoordModeOrigin );
} /*Narysuj*/

void MetodaOkna ( void )
{
    int x, y;

```

```

    switch ( zdarzenie.type ) {
case Expose:
    if ( zdarzenie.xexpose.count == 0 )
        Narysuj ();
    break;

case KeyPress:
    stan = KONIEC;
    break;

case ButtonPress:
    GdzieKursor ( &x, &y );
    np = ZnajdzNajblizszyPunkt ( x, y );
    if ( np >= 0 )
        stan = PRZESUWA;
    break;

case ButtonRelease:
    stan = NIC;
    break;

case MotionNotify:
    if ( stan == PRZESUWA ) {
        GdzieKursor ( &x, &y );
        punkt[np].x = x; punkt[np].y = y;
        Narysuj ();
    }
    break;

default:
    break;
    }
} /*MetodaOkna*/

int main ( int argc, char **argv )
{
    Przygotuj ();
    while ( stan != KONIEC ) {
        XNextEvent ( stacja, &zdarzenie );
        MetodaOkna ();
    }
    exit ( 0 );
} /*main*/

```

Program jest na tyle krótki, że nie powinno być zbyt trudnym ćwiczeniem rozpoznanie w nim przedstawionych wcześniej elementów, tj. implementacji automatu skończonego i listy obrazowej. Procedura Przygotuj zawiera preliminaria, na które składa się ustalenie komunikacji z systemem, utworzenie okna i umieszczenie go na ekranie oraz przygotowanie tzw. kontekstu graficznego, czyli struktury danych niezbędnych do rysowania (zawierającej informacje takie jak bieżący kolor, grubość linii, wzorzec wypełniania wielokątów itd.).

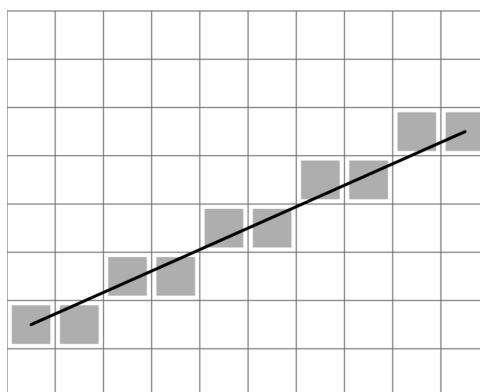
Procedura główna (main) po przygotowaniu wykonuje tzw. **pętlę komunikatów**. Po wywołaniu procedury XNextEvent program czeka na zdarzenie. Gdy ono nastąpi, następuje powrót z tej procedury i program może zareagować na zdarzenie, którym (w przykładzie wyżej) może być

naciśnięcie lub zwolnienie guzika (komunikaty `ButtonPress` i `ButtonRelease`), przesunięcie myszy (`MotionNotify`), naciśnięcie klawisza (`KeyPress`), a także zawiadomienie przez system o konieczności odtworzenia zawartości okna (komunikat `Expose`).

Aby interakcja była wygodna dla użytkownika, obsługa każdego komunikatu musi być wykonana w krótkim czasie — co najwyżej ok. 1/10s, tak aby nie było zauważalnych opóźnień. W razie konieczności przeprowadzenia bardziej długotrwałego obliczenia najlepiej wykonywać je za pomocą osobnego wątku obliczeniowego. Nie będę tu rozwijać tego wątku.

2. Podstawowe algorytmy grafiki rastrowej

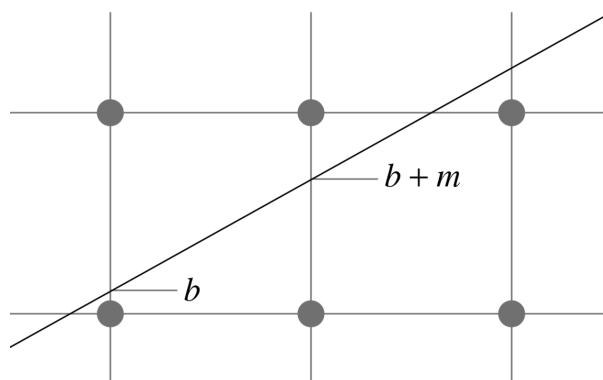
2.1. Rysowanie odcinka



Rysunek 2.1. Odcinek rastrowy.

Współrzędne punktów końcowych odcinka są liczbami całkowitymi; zakładamy, że $x_2 > x_1$, $y_2 \geq y_1$ oraz $y_2 - y_1 \leq x_2 - x_1$. Chcemy „narysować odcinek”, czyli wyznaczyć piksele najbliższe tego odcinka, i nadać im odpowiedni kolor. Pierwsza przymiarka procedury rysowania odcinka, czyli **algorytm I**, wygląda tak:

```
 $\Delta x = x_2 - x_1; \Delta y = y_2 - y_1;$   
 $m = \Delta y / \Delta x;$   
for (  $x = x_1, y = y_1; x \leq x_2; x++$  ) {  
    SetPixel (  $x, \text{round}(y)$  );  
     $y += m;$   
}
```



Rysunek 2.2. Wybór mniejszego błędu.

Zmienne y i m przyjmują wartości ułamkowe, a zatem muszą być typu **float**; występuje konieczność zaokrąglania współrzędnych, a ponadto błędy zaokrągleń w dodawaniu $y + m$ mogą się kumulować. Zauważmy, że w każdej kolumnie rastra o współrzędnych x między x_1 i x_2 rysujemy jeden piksel; idziemy zawsze w bok i czasem do góry — wtedy gdy spowoduje to wybranie piksela bliżej odcinka, czyli dającego *mniej* błąd. **Algorytm II** jawnie wykorzystuje to spostrzeżenie:

```

b = 0;
Δx = x2 - x1; Δy = y2 - y1;
m = Δy/Δx;
for ( x = x1, y = y1; x <= x2; x++ ) {
    SetPixel ( x, y );
    b += m;
    if ( b > 1/2 ) { y++; b -= 1; }
}

```

W algorytmie II nadal używamy zmiennych b i m typu *real*, ale przyjmują one zawsze wartości wymierne, które można sprowadzić do wspólnego mianownika Δx (a także $2\Delta x$); zatem niech $c = 2\Delta x \cdot b - \Delta x$. Zamiast podstawiać $b += m$ weźmy $c += 2\Delta y$; zamiast sprawdzać warunek $b > 1/2$ można sprawdzać równoważny warunek $c > 0$. Otrzymujemy w ten sposób **algorytm III**, w którym wszystkie rachunki są wykonywane na liczbach całkowitych, bez potrzeby zaokrąglania:

```

Δx = x2 - x1; Δy = y2 - y1;
c = -Δx;
for ( x = x1, y = y1; x <= x2; x++ ) {
    SetPixel ( x, y );
    c += 2Δy;
    if ( c > 0 ) { y++; c -= 2Δx; }
}

```

Powyższy algorytm rysowania odcinka nazywa się **algorytmem Bresenhama**. Obecnie jest on powszechnie implementowany w sprzęcie, tj. procesory stosowane w sterownikach („kartach”) graficznych zawierają odpowiednie podukłady, które obliczają kolejne piksele rysowanych odcinków właśnie w ten sposób. Mimo to nieraz zdarza się potrzeba użycia odpowiedniego podprogramu, jeśli zadanie nie polega po prostu na wyświetleniu pikseli.

Aby narysować odcinek, którego końce nie spełniają warunków określonych na początku, trzeba odpowiednio zamienić współrzędne x i y rolami lub zamienić znaki przyrostów współrzędnych. Procedurę rysowania odcinka odpowiednią w każdym przypadku możemy zrealizować tak:

```

void DrawLine ( int x1, int y1, int x2, int y2 )
{
    int deltax, deltay, g, h, c;

    deltax = x2-x1;
    if ( deltax > 0 ) g = +1; else g = -1;
    deltax = abs(deltax);
    deltay = y2-y1;
    if ( deltay > 0 ) h = +1; else h = -1;
    deltay = abs(deltay);
    if ( deltax > deltay ) {
        c = -deltax;

```

```

while ( x1 != x2 ) {
    SetPixel ( x1, y1 );
    c += 2*deltay;
    if ( c > 0 ) { y1 += h; c -= 2*deltax; }
    x1 += g;
}
}
else {
    c = -deltax;
    while ( y1 != y2 ) {
        SetPixel ( x1, y1 );
        c += 2*deltax;
        if ( c > 0 ) { x1 += g; c -= 2*deltay; }
        y1 += h;
    }
}
} /*DrawLine*/

```

Zaletą tej procedury jest fakt, że odcinek zawsze jest rysowany od pierwszego podanego końca (którego współrzędne są początkowymi wartościami zmiennych x_1 i y_1) do drugiego. Drugi koniec (tj. ostatni piksel obrazu odcinka) nie jest rysowany, co łatwo jest uzupełnić, dopisując na końcu wywołanie procedury `SetPixel ( x2, y2 )`, ale jest to zbędne (a nawet niepożądane) jeśli procedura rysująca odcinek jest używana do narysowania łamanej.

Przyjrzyjmy się jeszcze zawartości poszczególnych wierszy, narysowanej przez algorytm III: w wierszu y_1 mamy $\lceil \frac{\Delta x}{2\Delta y} \rceil$ pikseli, w kolejnych albo $\lfloor \frac{\Delta x}{\Delta y} \rfloor$ pikseli, albo o 1 więcej, a w ostatnim — resztę. W wielu przypadkach narysowanie jednocześnie kilku sąsiednich pikseli można zrealizować sprawniej niż poprzez rysowanie każdego z nich osobno. Jeśli weźmiemy $\tilde{x}_1 = x_1$, $\tilde{y}_1 = y_1$, $\tilde{x}_2 = x_2 - \lfloor \frac{\Delta x}{\Delta y} \rfloor \Delta y$ i $\tilde{y}_2 = y_2$, to mamy odcinek, dla którego $\tilde{y}_2 > \tilde{y}_1$, $\tilde{x}_2 \geq \tilde{x}_1$ i $\tilde{x}_2 - \tilde{x}_1 \leq \tilde{y}_2 - \tilde{y}_1$. Rasteryzacja tego odcinka wymaga wyznaczenia $y_2 - y_1 + 1$ pikseli, czyli na ogół znacznie mniej niż odcinka wyjściowego. Przypuśćmy, że dysponujemy taką procedurą, wywoływaną przez instrukcję `SetHLine ( x1, x2, y )`, która rysuje $x_2 - x_1$ pikseli w linii poziomej y , zaczynając od x_1 (czyli bez x_2). Możemy jej użyć w algorytmie Bresenhama dla odcinka o końcach $(\tilde{x}_1, \tilde{y}_1)$ i $(\tilde{x}_2, \tilde{y}_2)$. Odpowiedni fragment programu ma postać:

```

if ( y2 = y1 ) SetHLine ( x1, x2 + 1, y1 );
else {
    Δx = x2 - x1; Δy = y2 - y1;
    m = Δx / Δy; Δx̃ = Δx - m*Δy;
    c = -Δy;
    x3 = x1 - m / 2; x4 = x3 + m;
    SetHLine ( x1, x4, y1 );
    for ( y = y1 + 1; y < y2 ) {
        c = c + 2Δx̃;
        x3 = x4;
        if ( c > 0 ) { x4 = x4 + m + 1; c := c - 2Δy; }
        else x4 = x4 + m;
        SetHLine ( x3, x4, y );
    }
    SetHLine ( x3, x2, y2 );
}
}

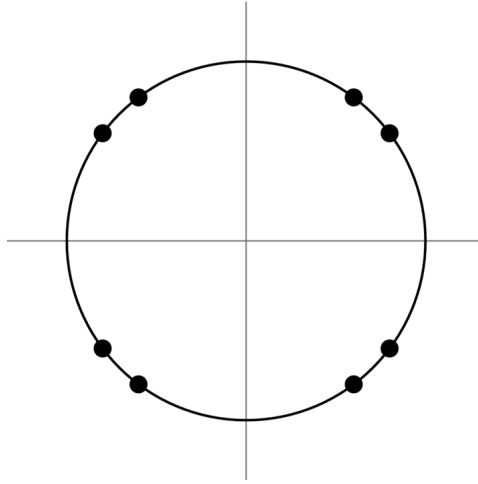
```

Oprócz zmniejszenia liczby obliczeń błędów dla odcinków nachylonych pod małymi kątami, do szybkości działania tej procedury przyczynia się fakt, że przypisanie koloru wielu pikselom

położonym obok siebie w poziomej linii rastra (przez procedurę `SetHLine`) może zająć znacznie mniej czasu niż przypisywanie koloru tym pikselom po kolei, m.in. dzięki uproszczeniu obliczania adresów (w pamięci obrazu) sąsiednich pikseli.

2.2. Rysowanie okręgu

Rysując okrąg warto wykorzystać ośmiokrotną symetrię jego obrazu rastrowego; jeśli ma on środek $(0,0)$ i zawiera piksel (x,y) , to zawiera on również piksele $(-x,y)$, $(-x,-y)$, $(x,-y)$, (y,x) , $(-y,x)$, $(-y,-x)$, $(y,-x)$. Do narysowania wszystkich tych pikseli możemy zastosować procedurę o nazwie np. `Set8Pixels`, która może też dodać do ich współrzędnych współrzędne środka okręgu. Wystarczy więc wyznaczyć piksele, które tworzą obraz jednej ósmej okręgu. Zasada działania algorytmu Bresenhama rysowania okręgu jest ta sama co w przypadku odcinka: wybieramy kolejne piksele starając się zminimalizować błąd, tj. odległość piksela od przedstawianej na obrazie figury.



Rysunek 2.3. Ośmiokrotna symetria okręgu.

Aby wyprowadzić algorytm rysowania okręgu, rozważmy dwie tożsamości:

$$\sum_{i=0}^x (2i+1) = (x+1)^2,$$

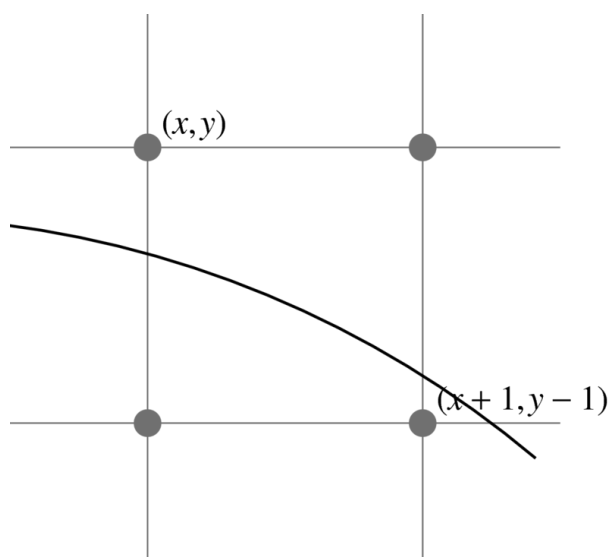
$$\sum_{i=y}^r (2i-1) = r^2 - (y-1)^2.$$

Wynika z nich, że funkcja

$$f(x,y) = \sum_{i=0}^x (2i+1) - \sum_{i=y}^r (2i-1) =: (x+1)^2 + (y-1)^2 - r^2$$

ma wartość 0 jeśli punkt $(x+1, y-1)$ leży na okręgu, jest dodatnia jeśli na zewnątrz i ujemna jeśli leży wewnątrz. Przypuśćmy, że ostatnio narysowany piksel to (x,y) i znamy wartość $c = f(x,y)$. Będziemy rysowali od góry, zaczynając w punkcie $(0,r)$. Rysując łuk przesuwamy się zawsze o 1 piksel w prawo i czasem o 1 w dół — wtedy, gdy da to mniejszy błąd, czyli wtedy gdy

$$|f(x,y)| < |f(x,y+1)|.$$



Rysunek 2.4. Wybór następnego piksela.

Uwaga: To nie zapewnia wyboru piksela bliżej okręgu, tylko piksela, dla którego funkcja f ma mniejszą wartość bezwzględną. Różnica jest tak mała, że w praktyce jest nieistotna.

Mamy $f(x, y) = c$, $f(x, y + 1) = c + 2y - 1$, a także (przyda się to za chwilę) $f(x + 1, y) = c + 2x + 3$ oraz $f(x + 1, y - 1) = f(x + 1, y) - 2y + 3$. Ponieważ rysujemy tak, aby zawsze wybierać między dwoma pikselami które leżą po przeciwnych stronach okręgu, więc $f(x, y) \leq 0 \leq f(x, y + 1)$. Stąd, zamiast porównywać $|f(x, y)|$ i $|f(x, y + 1)|$ można sprawdzać, czy $-c < c + 2y - 1$, czyli $2c > 1 - 2y$. Mamy więc następującą procedurę:

```

x = 0;
y = r;
c = 2(1 - r);
while ( x ≤ y ) {
    Set8Pixels ( x, y );
    if ( 2c > 1 - 2y ) { /* czasem w dół */
        y --;
        c -= 2y - 1;
    }
    x++; /* zawsze w bok */
    c += 2x + 1;
}

```

2.3. Rysowanie elips

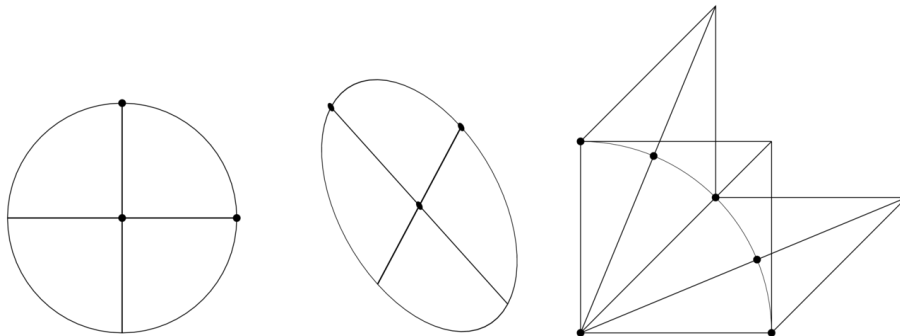
Jeśli współczynnik **aspekt** rastra, czyli iloraz szerokości i wysokości piksela (tj. odległości środka piksela od środków pikseli sąsiednich z boku i z góry) jest różny od 1, to zamiast okręgu otrzymamy elipsę; wtedy aby otrzymać obraz okręgu trzeba narysować elipsę, której oś pionowa (mierzona liczbą pikseli) jest aspekt razy dłuższa niż oś pozioma (mierzona też w pikselach). Ponadto czasem potrzebujemy narysować elipsę osiach o dowolnych długościach. Oznaczmy długość półosi poziomej literą a , a pionowej — b . Metoda pierwsza polega na narysowaniu (za pomocą algorytmu Bresenhama opisanego w poprzednim punkcie) okręgu o promieniu

równym długości dłuższej półosi, z procedurą Set8Pixels zmienioną w ten sposób, aby zamiast SetPixel (x, y); wywoływała SetPixel ($(x*a)/b, y$); (tu jest założenie, że $b > a$) itd. Należy przy tym uważać na nadmiar (mógłby on nas zaskoczyć, gdybyśmy używali arytmetyki szesnastobitowej).

Metoda druga, sporo trudniejsza, polega na narysowaniu okręgu o promieniu $r = ab$, za pomocą algorytmu opartego na tej samej zasadzie co algorytm Bresenhama. W bok należy poruszać się z krokiem b , a w dół z krokiem a pikseli. Rastrowy obraz elipsy ma tylko czterokrotną symetrię, więc dla każdego kolejnego punktu rysujemy tylko 4 piksele, a nie 8; należy przy tym wyznaczyć ćwiartkę elipsy, od pewnego miejsca poruszając się zawsze w dół i czasem w bok. Również ten algorytm wymaga użycia arytmetyki co najmniej 32-bitowej dla uniknięcia nadmiaru.

Opisane wyżej metody mają na celu rysowanie elips, których jedna oś jest pozioma, a druga pionowa. Jeśli elipsa, którą trzeba narysować nie spełnia tego warunku (tj. jest *w położeniu ogólnym*), to można zastosować metodę odpowiednią dla dowolnej krzywej ciągłej: wyznaczyć dostatecznie dużo punktów i narysować łamaną.

Aby narysować elipsę (lub dowolny inny obiekt), należy mieć jej odpowiednią reprezentację. Wygodne jest użycie **średnic sprzężonych**. Jak wiadomo, elipsa jest obrazem okręgu w pewnym przekształceniu afinicznym. Średnice sprzężone elipsy są obrazem pary prostopadłych średnic tego okręgu w tym samym przekształceniu afinicznym. Mając taką reprezentację możemy zastosować strategię „dziel i zdobywaj”. Mając dwa punkty końcowe łuku elipsy możemy zbadać, czy leżą one dostatecznie blisko. Jeśli tak, to narysujemy odcinek. Jeśli nie, to wyznaczmy punkt „środkowy” tego łuku, a następnie dwa łuki otrzymane z podziału łuku danego w wyznaczonym punkcie narysujemy stosując tę procedurę rekurencyjnie.



Rysunek 2.5. Średnice sprzężone elipsy i sposób ich użycia do rysowania.

Rozważmy łuk okręgu jednostkowego o środku $[0,0]^T$, którego końce są wyznaczone przez wektory $\mathbf{v}_1 = [1,0]^T$ i $\mathbf{v}_2 = [0,1]^T$. Wektor $\mathbf{v}_3$, wyznaczający punkt środkowy tego łuku, jest równy $(\mathbf{v}_1 + \mathbf{v}_2)a_1$, gdzie $a_1 = 1/\sqrt{2}$. Dalej możemy obliczyć wektory $\mathbf{v}_4 = (\mathbf{v}_1 + \mathbf{v}_3)a_2$ oraz $\mathbf{v}_5 = (\mathbf{v}_2 + \mathbf{v}_3)a_2$. Ogólnie, na k -tym poziomie rekurencyjnego podziału łuku okręgu, sumę wektorów wyznaczających końce łuku mnożymy przez $a_k = 1/(2 \cos(\pi/2^{k+1}))$. „Całą” procedurę rysowania elipsy za pomocą rekurencyjnego podziału możemy zapisać tak:

```
void r_Elipsa ( k, c, v1, v2 )
{
    if ( dostatecznie blisko ( v1, v2 ) )
        rysuj_odcinek ( c + v1, c + v2 );
    else {
        v3 = ak*(v1 + v2);
        r_Elipsa ( k + 1, c, v1, v3 );
        r_Elipsa ( k + 1, c, v3, v2 );
    }
}
```

```

}
} /*r_Elipse*/

```

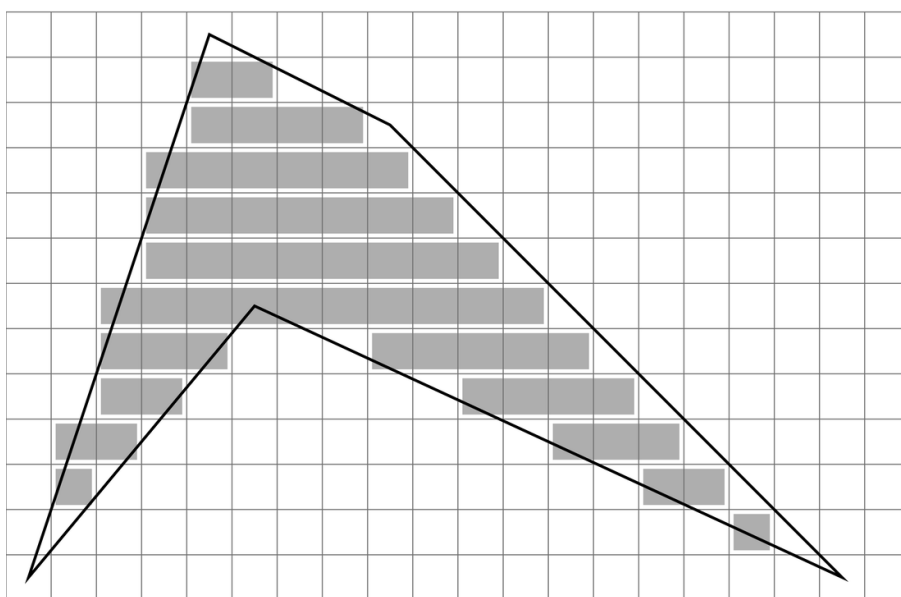
Parametry wywołania procedury przez program główny to $k = 1$, środek elipsy c i wektory v_1 i v_2 określające połówki średnic sprzężonych. W przypadku, gdy są one prostopadłe i mają równą długość, procedura narysuje ćwiartkę okręgu.

Współczynniki a_k najlepiej jest wcześniej obliczyć i przechowywać w tablicy. Zauważmy, że ograniczając głębokość rekurencji do 10, możemy narysować przybliżenie elipsy w postaci afinicznego obrazu 4096-kąta foremnego, co jest wystarczające w większości zastosowań.

2.4. Wypełnianie wielokątów

Dany jest n -kąt, reprezentowany przez n par (x, y) liczb całkowitych, określających wierzchołki. Należy zamalować piksele w jego wnętrzu.

W dobrze zaprojektowanych pakietach graficznych jest przyjęta i konsekwentnie przestrzegana umowa dotycząca rozstrzygania, czy piksel, którego środek leży na brzegu wielokąta, należy do niego, czy nie. Na przykład:



Rysunek 2.6. Piksele należące do wielokąta.

1. Jeśli środek piksela leży na krawędzi aktywnej, to piksel jest zamalowywany wtedy, gdy wnętrze wielokąta jest z prawej strony krawędzi;
2. Piksele leżące na krawędzi poziomej są zamalowywane wtedy gdy wnętrze wielokąta leży poniżej tej krawędzi.

Dzięki takiej umowie, jeśli mamy wielokąty o wspólnych krawędziach, to każdy piksel na takiej krawędzi należy do dokładnie jednego wielokąta. Ma to szczególne znaczenie w rysowaniu z przypisywaniem pikselom wartości zależnej od wartości poprzedniej (np. w trybie xor itd.).

Do wykonania zadania posłuży nam następująca **reguła parzystości**: punkt leży wewnątrz wielokąta, jeśli dowolna półprosta, która z niego wychodzi przecina brzeg wielokąta nieparzystą liczbę razy.

Algorytm przeglądania liniami poziomymi składa się z następujących kroków:

```

Utwórz tablicę krawędzi (par kolejnych wierzchołków, w tym  $(x_n, y_n), (x_1, y_1)$ );
Dla każdej krawędzi, jeśli współrzędna  $y$  jej drugiego końca jest mniejsza,
    zamień końce; krawędzie poziome usuń z tablicy;
Posortuj tablicę w kolejności rosnących współrzędnych  $y$  pierwszego końca;
Utwórz początkowo pustą tablicę krawędzi aktywnych (t.k.a.), czyli przeciętych kolejną linią poziomą;
 $y$  = współrzędna  $y$  pierwszej krawędzi w tablicy;
do {
    Wstaw do t.k.a. krawędzie, których pierwszy koniec jest na linii  $y$ ;
    Oblicz dla każdej krawędzi w t.k.a. współrzędną  $x$  punktu przecięcia z linią poziomą  $y$ ;
    Posortuj t.k.a. w kolejności rosnących współrzędnych  $x$  punktów przecięcia;
    Dla kolejnych par krawędzi aktywnych rysuj odcinek poziomy na linii  $y$ ,
        między ich punktami przecięcia z linią  $y$ ;
     $y$  ++;
    Usuń z t.k.a. krawędzie, których drugi koniec jest na linii  $y$ ;
} while ( t.k.a jest niepusta );

```

Uwaga: Tablica krawędzi aktywnych po każdym uaktualnieniu zawiera parzystą liczbę elementów.

2.5. Wypełnianie obszaru przez zalewanie

Przypuśćmy, że należy zamalować obszar, którego brzeg został narysowany wcześniej i obszar jest określony przez dany na początku obraz. Mamy tu więc problem z pogranicza grafiki i przetwarzania obrazów. Aby rozwiązać takie zadanie, należy je nieco uściślić, przez wprowadzenie dodatkowych pojęć.

Figura rastrowa jest **czterospójna**, jeśli za sąsiadów dowolnego piksela uznajemy cztery inne piksele (dwa po bokach i po jednym powyżej i poniżej; innymi słowy, sąsiadami piksela (x, y) są $(x - 1, y)$, $(x + 1, y)$, $(x, y - 1)$ i $(x, y + 1)$) i dla dowolnych dwóch pikseli należących do tej figury istnieje droga złożona z pikseli należących do niej, z których każde dwa kolejne są sąsiadami w podanym wyżej sensie.

Figura jest **ośmiospójna**, jeśli za sąsiadów piksela uznajemy oprócz podanych wyżej jeszcze cztery piksele, które mają wspólny narożnik z pikselem danym i dla dowolnych dwóch pikseli należących do tej figury istnieje droga złożona z ... itd.

Reguły spójności odgrywają dużą rolę w rozpoznawaniu obrazów, które polega na identyfikowaniu linii i innych figur na obrazie; tymczasem zauważmy następującą regułę: *brzeg obszaru ośmiospójnego jest czterospójny; brzeg obszaru czterospójnego jest ośmiospójny*.

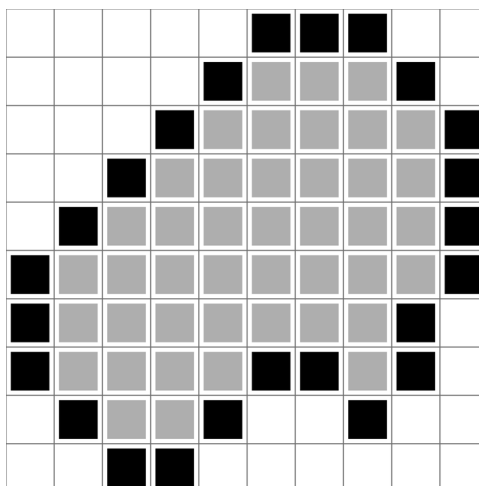
Mając określone sąsiedztwo pikseli w rastrze, dowolną figurę możemy reprezentować za pomocą **grafu sąsiedztwa pikseli**; jego wierzchołkami są piksele należące do figury; jego krawędziami są *wszystkie* krawędzie łączące wierzchołki, które są sąsiednie (w sensie jednej z powyższych definicji).

Algorytm **wypełniania przez zalewanie** (ang. *flood fill*) polega na przeszukaniu grafu sąsiedztwa pikseli obszaru, którego reprezentacją jest początkowy obraz rastrowy. Oprócz obrazu należy podać tzw. ziarno, czyli jeden piksel, który należy do obszaru, ale nie do jego brzegu. W wielu książkach jest podana procedura rekurencyjna, która stosuje metodę przeszukiwania grafu w głąb (ang. *depth-first search*, *DFS*); w wersji dla obszaru czterospójnego wygląda ona następująco:

```

void r_FloodFill ( x, y )
{
    if ( niezamalowany ( x, y ) ) {

```



Rysunek 2.7. Obszar rastrowy (czterospójny) i jego brzeg.

```

SetPixel ( x, y );
r_FloodFill ( x + 1, y );
r_FloodFill ( x - 1, y );
r_FloodFill ( x, y + 1 );
r_FloodFill ( x, y - 1 );
}
} /*r_FloodFill*/

```

Wadą przeszukiwania w głąb jest ogromne zapotrzebowanie na pamięć (w tym przypadku stos rekurencyjnych wywołań procedury). Trzeba się liczyć z koniecznością przechowania na nim rekordów aktywacji procedury dla wszystkich pikseli w obszarze (czyli np. rzędu 10^6 , jeśli obszar jest tak duży jak ekran).

Znacznie lepiej działa przeszukiwanie wszerz (ang. *breadth-first search*, *BFS*), które polega na użyciu kolejki. Ziarno wstawiamy do pustej kolejki, a następnie, dopóki kolejka nie jest pusta, wyjmujemy z niej piksel, i jeśli jest niezamalowany, to zamalowujemy go i wstawiamy jego sąsiadów do kolejki. Potrzebna pojemność kolejki jest rzędu liczby pikseli na brzegu obszaru.

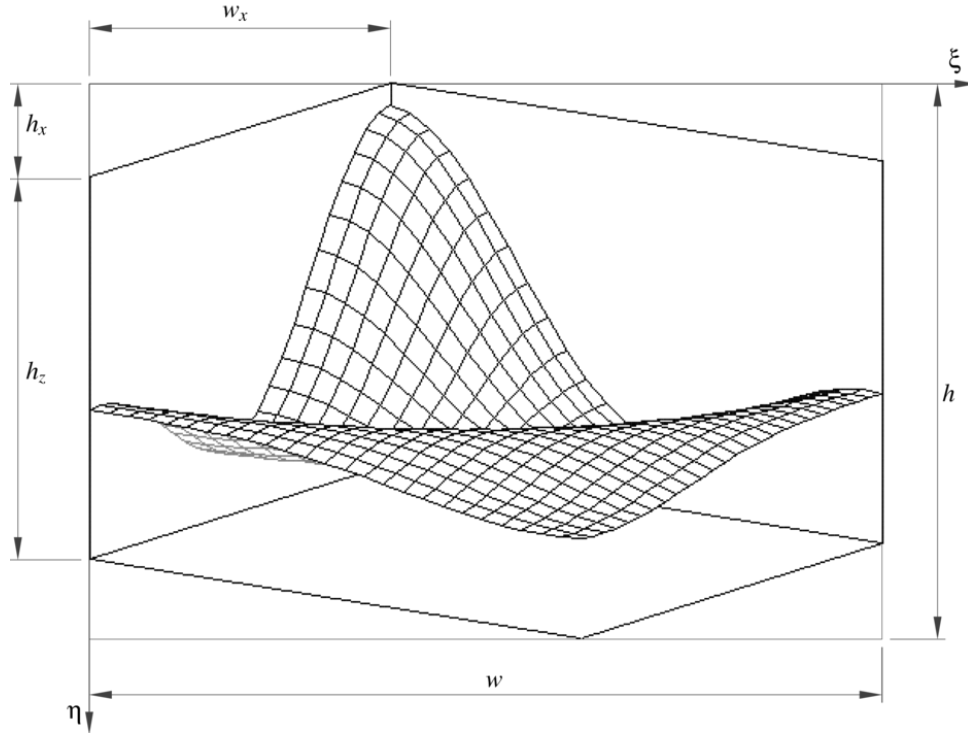
Jeszcze sprawniejsza procedura wyznacza **graf sąsiedztwa linii**; jego wierzchołkami są poziome odcinki (o maksymalnej długości), złożone z pikseli należących do obszaru. Na razie tego tematu nie rozwijam.

2.6. Algorytm z pływającym horyzontem

Jednym z efektownych i użytecznych zastosowań algorytmu Bresenhama jest rysowanie wykresów funkcji dwóch zmiennych, tj. obrazów powierzchni $z = f(x, y)$, z uwzględnieniem widoczności. Właśnie w tym algorytmie potrzebna jest procedura, która obliczy piksele pewnych odcinków i dla każdego z nich wywoła procedury, które rysowanie uzupełniają dodatkowym obliczeniem. W takim zastosowaniu implementacja algorytmu Bresenhama obecna w sprzęcie (w układzie grafiki komputera) jest bezużyteczna.

Algorytm rysuje wykres funkcji w danym prostokącie $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. Zakładamy, że funkcja f w tym prostokącie jest ciągła i przyjmuje wartości z pewnego przedziału $[z_{\min}, z_{\max}]$.

Wykres funkcji mieści się więc w kostce $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}]$, której obraz powinien być wpisany w prostokąt o szerokości w i wysokości h pikseli. Aby jednoznacznie



Rysunek 2.8. Wykres funkcji dwóch zmiennych wykonany przy użyciu algorytmu z pływającym horyzontem.

określić odwzorowanie przestrzeni trójwymiarowej na płaszczyznę ekranu, trzeba jeszcze podać wymiary w_x , h_x i h_z , zaznaczone na rysunku 2.8. Punktowi (x, y, z) w przestrzeni trójwymiarowej odpowiada na obrazie punkt (ξ, η) , którego współrzędne są równe

$$\begin{aligned}\xi &= ax + by + cz + d, \\ \eta &= ex + fy + gz + h.\end{aligned}$$

Współczynniki $a, \dots, h$, określone na podstawie podanych wymiarów obrazu dla przedstawionej na rysunku orientacji osi układu współrzędnych na ekranie, można obliczyć na podstawie wzorów

$$\begin{aligned}a &= \frac{w_x}{x_{\min} - x_{\max}}, & b &= \frac{w - w_x}{y_{\max} - y_{\min}}, & c &= 0, & d &= -ax_{\max} - by_{\min}, \\ e &= \frac{h_x}{x_{\max} - x_{\min}}, & f &= \frac{h - h_x - h_z}{y_{\max} - y_{\min}}, & g &= \frac{h_z}{z_{\min} - z_{\max}}, & h &= -ex_{\min} - fy_{\min} - gz_{\max}.\end{aligned}$$

Istotne w określeniu tego przekształcenia dla algorytmu z pływającym horyzontem jest to, że obrazy punktów, które mają identyczne współrzędne x i y , mają tę samą współrzędną ξ . Jest to zapewnione przez przyjęcie współczynnika $c = 0$.

Rysowanie wykresu następuje „od przodu do tyłu”, przy czym wcześniej narysowane odcinki ograniczają obszar „zasłonięty”, w którym nie wolno rysować. W każdej kolumnie pikseli obszar ten jest odcinkiem, którego końcami są najwyższy i najniższy piksel narysowany wcześniej w tej kolumnie. Dla każdej kolumny pikseli potrzebujemy zatem dwóch zmiennych całkowitych do zapamiętania współrzędnych η tych pikseli. Zmienne te są elementami dwóch tablic, zwanych horyzontami. Horyzont dolny, który ogranicza obszar zasłonięty od dołu, w trakcie rysowania „obniża się” (tj. wartości każdego elementu tej tablicy rosną). Podobnie horyzont górny

„podwyższa się”, przy czym początkową wartością wszystkich elementów tych dwóch tablic jest odpowiednio -1 i h .

Aby wykonać wykres obliczamy wartości funkcji f w węzłach regularnej siatki wypełniającej prostokąt $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$, obliczamy (i wpisujemy do tablicy) obrazy punktów $(x, y, f(x, y))$ (po zaokrągleniu współrzędnych do liczb całkowitych), inicjalizujemy horyzonty i rysujemy odpowiednie odcinki za pomocą algorytmu Bresenhama. Dla każdego piksela wyznaczonego przez ten algorytm sprawdzamy, czy jest on powyżej górnego lub poniżej dolnego horyzontu i jeśli tak, to przypisujemy mu odpowiedni kolor. Następnie uaktualniamy horyzonty, jako że narysowanie piksela w danej kolumnie oznacza rozszerzenie obszaru zasłoniętego. Ale zrobienie tego natychmiast prowadzi do błędów. Następne piksele rysowanego odcinka znajdujące się w tej samej kolumnie też powinny zostać narysowane, a uaktualnienie horyzontu po narysowaniu poprzedniego piksela może spowodować ich pominięcie.

Rozwiązaniem tego problemu jest dwukrotne wywołanie algorytmu Bresenhama dla każdego odcinka, z różnymi procedurami wywoływanymi w celu przetworzenia pikseli. Za pierwszym razem wykonujemy test widoczności i rysowanie, a za drugim razem uaktualniamy horyzonty. Procedura DrawLine, realizująca algorytm Bresenhama, otrzymuje procedurę przetwarzającą piksele jako parametr, zatem jej nagłówek musi być taki:

```
void DrawLine ( int x1, int y1, int x2, int y2, void (*SetPixel)(int x, int y) );
```

Jako ostatni parametr będziemy przekazywać dwie różne procedury. Procedura SetPixel1 sprawdza widoczność i rysuje (wywołując „prawdziwą” procedurę rysowania piksela), zaś SetPixel2 uaktualnia horyzonty.

Zmienne wdt i hgh opisują wymiary obrazu (w pikselach). W tablicy ftab są przechowywane końce odcinków do narysowania (w tym kodzie jest to tablica jednowymiarowa, o długości $(densx+1)*(densy+1)$, gdzie parametry densx i densy określają gęstość siatki). Tablice fhup i fhdn opisują horyzont górny i dolny (rezerwowanie i zwalnianie pamięci na te tablice zostało pominięte). Procedury SetPixel1 i SetPixel2 są dodatkowo wywoływane poza procedurą DrawLine po to, aby uzupełnić brak ich wywołania dla ostatniego piksela rysowanych odcinków (w całym obrazie trzeba w ten sposób uzupełnić tylko jeden piksel). Podany niżej fragment programu w C rysuje odcinki parami.

```
short *fhup, *fhdn;

void SetPixel1 ( int x, int y )
{
    if ( y <= fhdn[x] && y >= fhup[x] )
        return;
    SetPixel ( x, y );
} /*SetPixel1*/

void SetPixel2 ( int x, int y )
{
    if ( y > fhdn[x] ) fhdn[x] = y;
    if ( y < fhup[x] ) fhup[x] = y;
} /*SetPixel2*/

...
for ( i = 0; i < wdt; i++ )
    { fhup[i] = hgh; fhdn[i] = -1; }
for ( j = k = 0; j < densy; j++, k++ ) {
    pa = ftab[k]; pb = ftab[k+1];
    DrawLine ( pa.x, pa.y, pb.x, pb.y, &SetPixel1 );
```

```
    DrawLine ( pa.x, pa.y, pb.x, pb.y, &SetPixel2 );
}
SetPixel1 ( pb.x, pb.y ); SetPixel2 ( pb.x, pb.y );
for ( i = 1, k = densy+1; i <= densx; i++, k++ ) {
    pa = ftab[k]; pb = ftab[k-densy-1];
    DrawLine ( pa.x, pa.y, pb.x, pb.y, &SetPixel1 );
    DrawLine ( pa.x, pa.y, pb.x, pb.y, &SetPixel2 );
    for ( j = 0; j < densy; j++, k++ ) {
        pb = pa; pa = ftab[k+1]; pc = ftab[k-densy];
        DrawLine ( pb.x, pb.y, pa.x, pa.y, &SetPixel1 );
        DrawLine ( pa.x, pa.y, pc.x, pc.y, &SetPixel1 );
        DrawLine ( pb.x, pb.y, pa.x, pa.y, &SetPixel2 );
        DrawLine ( pa.x, pa.y, pc.x, pc.y, &SetPixel2 );
    }
}
...

```

Pozostaje mi zaproponować uzupełnienie tego kodu do pełnego programu, uruchomienie, eksperymenty i dorabianie bajerów, takich jak obliczanie kolorów pikseli zależnie od wartości funkcji f , albo rozszerzenie algorytmu umożliwiające rysowanie wykresów „z dziurami”.

3. Obcinanie linii i wielokątów

3.1. Obcinanie odcinków i prostych

Zadanie obcinania polega na

- wyznaczeniu fragmentu odcinka lub prostej, który leży wewnątrz okna na ekranie (ogólniej: dowolnego wielokąta), lub
- wyznaczeniu fragmentu odcinka lub prostej, który leży wewnątrz ustalonej bryły wielościennej.

Rozwiązanie jednego z tych zadań może być potrzebne w celu narysowania pewnej części odcinka, a także w konstrukcyjnej geometrii brył, w algorytmach widoczności i w śledzeniu promieni, o czym będzie mowa dalej.

Jeśli problem do rozwiązania polega na wyznaczeniu części wspólnej odcinka (lub prostej) i wielokąta (lub wielościanu) wypukłego, to możemy ten wielokąt (wielościan) przedstawić jako przecięcie pewnej liczby półpłaszczyzn (półprzestrzeni) i sprowadzić zadanie do obcinania półpłaszczyznami (półprzestrzeniami).

3.1.1. Wyznaczanie punktu przecięcia odcinka i prostej

Dane są punkty końcowe odcinka, $\mathbf{p}_1 = (x_1, y_1)$ i $\mathbf{p}_2 = (x_2, y_2)$, oraz współczynniki a, b, c równania prostej $ax + by = c$. Mamy znaleźć (jeśli istnieje) punkt wspólny tego odcinka i prostej.

Przedstawienie parametryczne odcinka, $\mathbf{p} = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1)$, czyli

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + t \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix} \quad \text{dla } t \in [0, 1], \quad (3.1)$$

wstawiamy do równania prostej, otrzymując równanie

$$a(x_1 + t(x_2 - x_1)) + b(y_1 + t(y_2 - y_1)) = c,$$

którego rozwiązaniem jest

$$t = \frac{c - ax_1 - by_1}{a(x_2 - x_1) + b(y_2 - y_1)}.$$

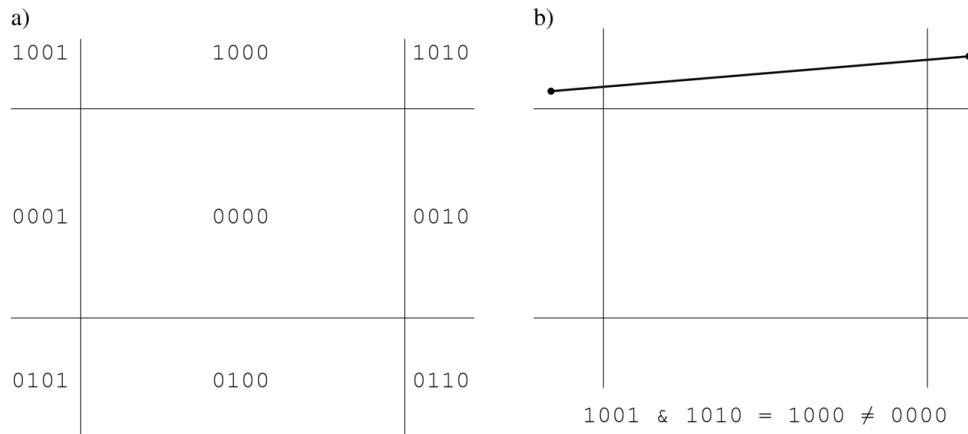
Jeśli $t \notin [0, 1]$, to prosta i odcinek są rozłączne. W przeciwnym razie możemy obliczyć współrzędne x, y punktu wspólnego. Wzór jest szczególnie prosty w przypadku, gdy równanie prostej obcinającej ma postać $x = c$ (prosta jest wtedy pionowa), lub $y = c$ (prosta jest pozioma).

Prawie identyczne jest wyprowadzenie odpowiednich wzorów w przestrzeni trójwymiarowej, dla danych punktów końcowych (x_1, y_1, z_1) , (x_2, y_2, z_2) i równania płaszczyzny obcinającej $ax + by + cz = d$.

3.1.2. Algorytm Sutherlanda-Cohena

Omówimy najbardziej popularną wersję obcinania do okna prostokątnego. Dla dowolnego wielokąta/wielościanu wypukłego zasada działania algorytmu jest identyczna.

Dane są punkty końcowe odcinka i prostokątne okno. Proste, na których leżą krawędzie okna, dzielą płaszczyznę na 9 obszarów. Przyporządkujemy im czterobitowe kody przedstawione na rysunku 3.1a.



Rysunek 3.1. Algorytm Sutherlanda-Cohana: podział płaszczyzny na obszary a) podział płaszczyzny na obszary, b) odrzucanie odcinka.

Kody obszarów, do których należą końce odcinka, możemy wyznaczyć na podstawie ich współrzędnych. Zauważmy, że jeśli oba kody na dowolnej pozycji mają jedynkę, to cały odcinek leży poza oknem (rys. 3.1b). Jeśli oba punkty końcowe mają kod 0, to cały odcinek leży wewnątrz okna. Jeśli kody są różne od 0, ale nie mają jedynki jednocześnie na żadnej pozycji, to odcinek może, ale *nie musi* mieć części wewnątrz okna.

Dla dowolnego wielokąta lub wielościanu określa się kody o liczbie bitów równej liczbie krawędzi albo ścian. Algorytm wyznacza punkt przecięcia odcinka z krawędzią, której odpowiada 1 w którymś kodzie, a następnie zastępuje jeden z punktów (ten, w którego kodzie występuje ta jedynka) przez punkt przecięcia. Nie grozi przy tym dzielenie przez 0 (dlaczego?).

```
#define NIC 0
#define CAŁY 1
#define CZĘŚĆ 2

char SC_clip ( punkt &p1, punkt &p2 )
{
    char wynik, c1, c2;

    wynik = CAŁY;
    c1 = KodPunktu ( p1 );
    c2 = KodPunktu ( p2 );
    while ( c1 || c2 ) {
        if ( c1 & c2 )
            return NIC;
        if ( !c1 ) {
            zamień ( p1, p2 );
            zamień ( c1, c2 );
        }
        switch ( c1 ) {
            case 0001: case 0101: case 1001: *p1 = punkt przecięcia z prostą  $x = x_{\min}$ ; break;
            case 0010: case 0110: case 1010: *p1 = punkt przecięcia z prostą  $x = x_{\max}$ ; break;
            case 0100: *p1 = punkt przecięcia z prostą  $y = y_{\min}$ ; break;
            case 1000: *p1 = punkt przecięcia z prostą  $y = y_{\max}$ ;
        }
    }
}
```

```

    }
    c1 = KodPunktu ( p1 );
    *kod = CZĘŚĆ;
}
return wynik;
} /*SC_clip*/

```

3.1.3. Algorytm Lianga-Barsky'ego

Wadą algorytmu Sutherlanda-Cohena jest to, że obliczane są punkty, które mogą być następnie odrzucone (ponieważ leżą poza oknem). Zabiera to czas, a ponadto wprowadza większe błędy zaokrągleń. Aby otrzymać algorytm pozbawiony tych wad, skorzystamy z parametrycznego przedstawienia odcinka (3.1). Oznaczmy $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$. Aby dokonać obcięcia odcinka, zawężymy przedział zmienności parametru t do przedziału odpowiadającego części wspólnej odcinka z oknem, a dopiero potem wyznaczymy punkty końcowe. Zauważmy, że w ogólności możemy przyjąć na początku inny przedział niż $[0, 1]$, co umożliwi obcinanie odcinka który leży na prostej przechodzącej przez punkty (x_1, y_1) i (x_2, y_2) , ale o innych końcach. Możemy nawet przyjąć początkowy przedział $[-\infty, +\infty]$, który reprezentuje całą prostą (ale wtedy procedura obcinania musi wyprowadzić wynik w postaci odpowiednich wartości parametru prostej).

```

float t1, t2;

char Test ( float p, float q )
{
    float r;

    if ( p < 0 ) {
        r = q/p;
        if ( r > t2 ) return 0;
        else if ( r > t1 ) t1 = r;
    }
    else if ( p > 0 ) {
        r = q/p;
        if ( r < t1 ) return 0;
        else if ( r < t2 ) t2 = r;
    }
    else if ( q < 0 ) return 0;
    return 1;
} /*Test*/

char LB_clip ( punkt *p1, punkt *p2 )
{
    float dx, dy;

    t1 = 0; t2 = 1; dx = p2->x - p1->x;
    if ( Test ( -dx, p1->x - xmin ) )
        if ( Test ( dx, xmax - p1->x ) ) {
            dy = p2->y - p1->y;
            if ( Test ( -dy, p1->y - ymin ) )
                if ( ( Test ( dy, ymax - p1->y ) ) ) {
                    if ( t2 != 1 ) { p2->x += t2*dx; p2->y += t2*dy; }
                    if ( t1 != 0 ) { p1->x += t1*dx; p1->y += t1*dy; }
                    return 1;
                }
            }
        }
}

```

```

    }
}
return 0;
} /*LB_clip*/

```

Wartością funkcji jest 1 jeśli przynajmniej część odcinka jest widoczna i 0 w przeciwnym razie. Zasada działania algorytmu wiąże się z interpretacją prostych, na których leżą krawędzie okna, jako zbiorów algebraicznych. Funkcja $f(x, y) = ax + by - c$, której zbiorem miejsc zerowych jest taka prosta, jest dodatnia w półpłaszczyźnie zawierającej okno (to jest zapewnione przez odpowiedni wybór znaku współczynników a , b , c). Parametr q funkcji Test otrzymuje wartość tej funkcji w punkcie p_1 kolejno dla każdej krawędzi okna. Po wykryciu, że przedział zmienności parametru t jest pusty, funkcja ma wartość 0. W przeciwnym razie, po obcięciu wszystkimi krawędziami okna, procedura oblicza punkty końcowe części odcinka widocznej w oknie (uwaga na kolejność obliczania tych punktów; w pewnej książce wykryłem błąd, polegający na zmianie kolejności).

Algorytm Lianga-Barsky’ego, podobnie jak algorytm Sutherlanda-Cohena, może być łatwo zmodyfikowany w celu obcinania odcinków do dowolnych wielokątów lub wielościanów wypukłych.

3.1.4. Obcinanie prostych

Jeśli mamy wyznaczyć część wspólną danej prostej i prostokątnego okna, to możemy użyć algorytmu Lianga-Barsky’ego. Opierając się na przedstawieniu prostej w postaci niejawnej, za pomocą równania liniowego $ax + by = c$, możemy to zadanie wykonać jeszcze trochę szybciej. W tym celu trzeba obliczyć wartości funkcji $f(x, y) = ax + by - c$ w wierzchołkach okna, a następnie, badając znaki wartości funkcji f w tych punktach, wyznaczyć punkty przecięcia prostej tylko z tymi krawędziami okna, które są przez prostą przecięte. Zysk z takiego podejścia to kilka zaoszczędzonych działań arytmetycznych, co (w przypadku średnim) przekłada się na prawie o połowę krótszy czas działania. Ponieważ procedury obcinania należą do oprogramowania podstawowego w bibliotekach graficznych (często są one implementowane sprzętowo) i wywoływane wiele razy, więc każde ich przyspieszenie jest istotne.

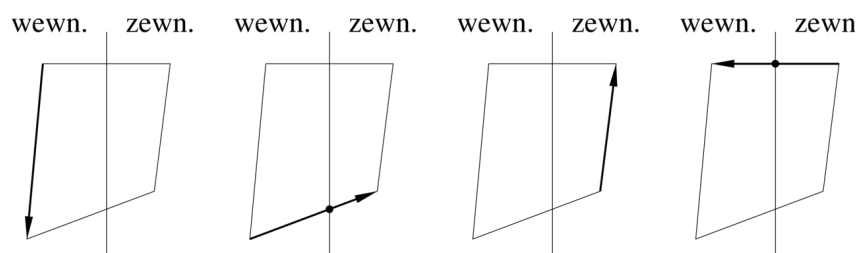
3.2. Obcinanie wielokątów

3.2.1. Algorytm Sutherlanda-Hodgmana

Zajmiemy się wyznaczaniem części wspólnej wielokąta i półpłaszczyzny. Wyznaczanie przecięcia dwóch wielokątów, z których przynajmniej jeden jest wypukły (np. jest prostokątnym oknem) można wykonać za pomocą kilkakrotnego obcinania do półpłaszczyzny.

Założymy, że brzeg wielokąta jest jedną łamaną zamkniętą. Jeśli tak nie jest (wielokąt ma dziury i łamanych jest więcej), to każdą łamaną trzeba obciąć osobno. Brzeg wielokąta obciętego przez opisany niżej algorytm Sutherlanda-Hodgmana też jest łamaną zamkniętą, zorientowaną zgodnie z brzegiem wielokąta danego.

W algorytmie Sutherlanda-Hodgmana kolejno przetwarzamy boki wielokąta (odcinki łamanej zamkniętej). Zmienna s reprezentuje punkt końcowy poprzedniego odcinka. Wywoływana przez poniższą procedurę funkcja `Inside` ma wartość **true** jeśli punkt podany jako argument leży we „właściwej” półpłaszczyźnie. Funkcja `Intersect` oblicza punkt przecięcia odcinka (o końcach podanych jako parametry). Procedura `Output` ma za zadanie wyprowadzić kolejny wierzchołek obciętego wielokąta, co może polegać na wstawieniu go do tablicy.



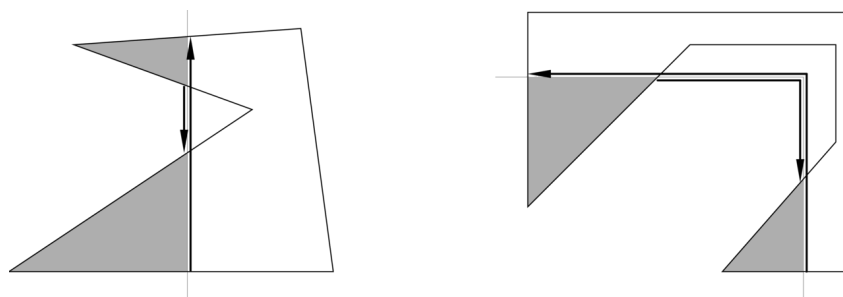
Rysunek 3.2. Algorytm Sutherlanda-Hodgmana.

```

void SH_clip ( int n, punkt w[] );
{
    punkt p, q, s;
    char is, ip;

    s = w[n-1];
    is = Inside ( s );
    for ( i = 0; i < n; i++ ) {
        p = w[i]; ip := Inside ( p );
        if ( is ) {
            if ( ip ) Output ( p );
            else {
                q = Intersect ( s, p );
                Output ( q );
            }
        }
        else if ( ip ) {
            q = Intersect ( s, p );
            Output ( q );
            Output ( p );
        }
        s = p;
        is = ip;
    }
} /*SH_clip*/

```



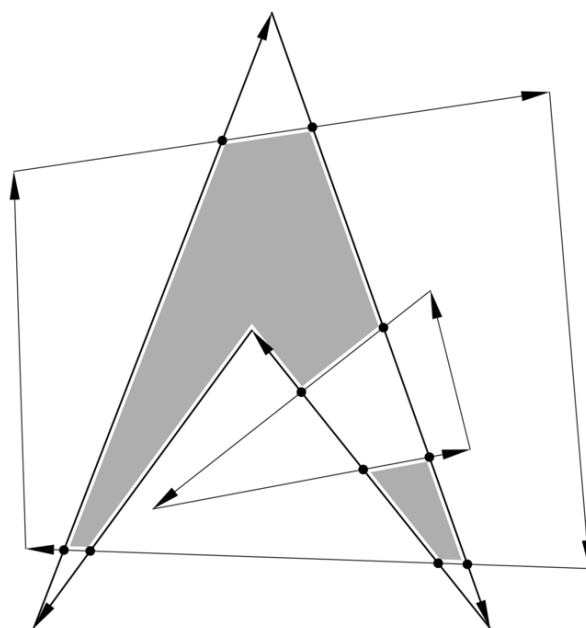
Rysunek 3.3. Wyniki obcinania wielokątów niewypukłych.

Zauważmy, że jeśli wielokąt nie jest wypukły, to jego część wspólna z półpłaszczyzną (albo oknem) może nie być spójna. Dostajemy wtedy na wyjściu łamaną, której krawędzie mają wspólne części (i części te są przeciwnie zorientowane). Jeśli po obcięciu wypełniamy wielokąt

algorytmem przeglądania liniami poziomymi, to obecność tych „falszywych krawędzi” nie ma znaczenia, choć jeśli brzeg półpłaszczyzny jest ukośny (tj. nie jest prostą pionową ani poziomą), to po zaokrągleniu współrzędnych wierzchołków do liczb całkowitych błędy mogą się ukazać (w postaci błędnie zamalowanych pikseli).

3.2.2. Algorytm Weilera-Athertona

W różnych zastosowaniach zdarza się potrzeba wyznaczenia części wspólnej, sumy lub różnicy dwóch wielokątów dowolnych, tj. niekoniecznie wypukłych. Umożliwia to **algorytm Weilera-Athertona**, który dopuszcza jako dane wielokąty, których brzegi mogą składać się z wielu zamkniętych łamanych; wielokąty takie mogą być niejednostopne, tj. z dziurami, albo nawet niespójne.



Rysunek 3.4. Algorytm Weilera-Athertona.

Podstawowe wymaganie, konieczne aby wynik był dobrze określony, to właściwe zorientowanie brzegu. Dla ustalenia uwagi przyjmiemy umowę, że każda krawędź jest zorientowana w ten sposób, że poruszając się wzdłuż niej zgodnie z orientacją mamy wewnątrz wielokąta po prawej stronie (rys. 3.4). Aby wyznaczyć przecięcie wielokątów, kolejno

1. Konstruujemy reprezentacje dwóch grafów zorientowanych; wierzchołkami i krawędziami każdego z nich są wierzchołki i zorientowane krawędzie jednego z wielokątów.
2. Znajdujemy wszystkie punkty przecięcia krawędzi jednego wielokąta z krawędziami drugiego. Dzielimy krawędzie, na których leżą te punkty, na kawałki; dołączamy w ten sposób nowe wierzchołki do grafu, zachowując uporządkowanie punktów podziału krawędzi w grafie. Wprowadzamy dodatkowe powiązanie między wierzchołkami grafów, które odpowiadają temu samemu punktowi (przecięcia). Tworzy się w ten sposób jeden graf, którego krawędzie reprezentują kawałki brzegu jednego lub drugiego wielokąta.
3. Znajdujemy wierzchołek grafu, który jest wierzchołkiem przecięcia wielokątów. Może to być wierzchołek odpowiadający punktowi przecięcia brzegów wielokątów, a jeśli takiego nie ma, to możemy sprawdzić, czy jakiś wierzchołek leży wewnątrz drugiego wielokąta, korzystając z reguły parzystości. Wybieramy wychodzącą ze znajdującego się wewnątrz wierzchołka krawędź, która jest krawędzią przecięcia (wyboru dokonujemy na podstawie orientacji brzegów).

Zaczynając od znalezionej wierzchołka, obchodzimy graf, aż do trafienia ponownie na wierzchołek, z którego wyszliśmy. W każdym wierzchołku, który odpowiada przecięciu brzegów, wybieramy krawędź wielokąta innego niż ten, po którego krawędzi poruszaliśmy się. Odwiedzone wierzchołki wyprowadzamy; ich ciąg reprezentuje odpowiednio zorientowany fragment (zamkniętą łamaną) przecięcia wielokątów danych.

Algorytm kończy działanie po stwierdzeniu braku nieodwiedzonego wierzchołka należącego do brzegu wielokątów.

Mając algorytm, który wyznacza przecięcie wielokątów, łatwo możemy otrzymać algorytm wyznaczania sumy lub różnicy; dany brzeg określa wielokąt, ale także jego dopełnienie — wystarczy tylko odwrócić orientację wszystkich krawędzi. W ten sposób sumę wielokątów otrzymamy jako dopełnienie przecięcia ich dopełnień.

Ważne dla poprawnej implementacji algorytmu jest uwzględnienie przypadków, gdy wielokąty mają wspólne wierzchołki lub gdy ich krawędzie mają wspólne odcinki. Pewne punkty mogą też być końcami więcej niż dwóch krawędzi wielokąta. Istotna jest możliwość wykonywania działań na wielokątach otrzymanych za pomocą tego algorytmu, i wtedy takie sytuacje zdarzają się często.

4. Elementy geometrii afinicznej

4.1. Przestrzenie afiniczne i euklidesowe

Figury geometryczne, których obrazy są tworzone w grafice komputerowej, mogą leżeć w różnych przestrzeniach (np. nieeuklidesowych), ale ostateczna reprezentacja powstaje w afinicznej przestrzeni euklidesowej, a w każdym razie z tą przestrzenią są związane sposoby reprezentowania obrazów przez różne urządzenia. Warto więc przypomnieć, co to jest.

4.1.1. Określenie przestrzeni afinicznej

Aksjomatyczne określenie przestrzeni afinicznej jest następujące: jeśli mamy pewien zbiór E i przestrzeń liniową V , oraz działanie **odejmowania punktów**, które parze punktów p i $q \in E$ przyporządkowuje pewien wektor $v = p - q \in V$, takie że

- dla każdego $q \in E$ i $v \in V$ istnieje dokładnie jeden punkt $p \in E$ spełniający równanie $v = p - q$, oraz
- dla dowolnych punktów $p, q, r \in E$ zachodzi tak zwana **równość trójkąta**: $r - p = (r - q) + (q - p)$,

to zbiór E nazywa się **przestrzenią afiniczną**, a przestrzeń V jej **przestrzenią wektorów swobodnych**. **Wymiar** przestrzeni afinicznej jest równy wymiarowi przestrzeni V . Związki między punktami przestrzeni E i wektorami swobodnymi są takie:

$$\begin{aligned} v &= p - q \in V \text{ — różnica punktów jest wektorem,} \\ p &= q + v \in E \text{ — suma punktu i wektora jest punktem.} \end{aligned}$$

Przestrzeń afiniczna jest **rzeczywista**, jeśli jej przestrzeń wektorów swobodnych jest przestrzenią liniową nad ciałem liczb rzeczywistych $\mathbb{R}$ i w grafice tylko takie przestrzenie się rozpatruje, głównie dwu- i trójwymiarową (nie wykluczam zastosowania przestrzeni nad innymi ciałami w jakichś bardzo specyficznych zastosowaniach).

4.1.2. Iloczyn skalarny

W przestrzeni wektorów swobodnych V możemy określić **iloczyn skalarny**, czyli funkcję $\langle \cdot, \cdot \rangle: V \times V \rightarrow \mathbb{R}$, która jest symetryczna,

$$\forall x, y \in V \langle x, y \rangle = \langle y, x \rangle,$$

liniowa ze względu na pierwszy (a ze względu na symetrię także drugi) argument,

$$\forall x, y, z \in V, a, b \in \mathbb{R} \langle ax + by, z \rangle = a \langle x, z \rangle + b \langle y, z \rangle,$$

i dodatnio określona,

$$\forall x \in V, x \neq 0 \langle x, x \rangle > 0.$$

Przestrzeń afiniczna, której przestrzeń wektorów swobodnych jest wyposażona w iloczyn skalarny, nazywa się **przestrzenią euklidesową**. W takiej przestrzeni możemy mierzyć odległości punktów, wzorem

$$\rho(\mathbf{p}, \mathbf{q}) = \sqrt{\langle \mathbf{p} - \mathbf{q}, \mathbf{p} - \mathbf{q} \rangle},$$

a także kąty między prostymi, np. jeśli prosta ℓ_1 przechodzi przez dwa różne punkty $\mathbf{p}$ i $\mathbf{q}$, a ℓ_2 przez $\mathbf{s}$ i $\mathbf{t}$, to kąt α między tymi prostymi spełnia równość

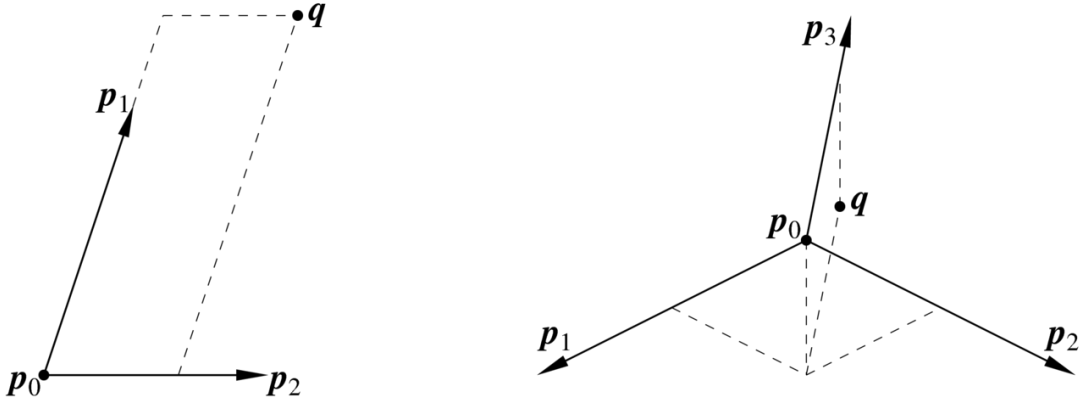
$$\cos \alpha = \left| \frac{\langle \mathbf{p} - \mathbf{q}, \mathbf{s} - \mathbf{t} \rangle}{\rho(\mathbf{p}, \mathbf{q})\rho(\mathbf{s}, \mathbf{t})} \right|.$$

Mając pojęcia odległości i kąta, możemy określić miary, takie jak pole powierzchni i objętość. Po to, by to wszystko obliczać, potrzebny jest jakiś układ współrzędnych.

Istotne jest, że w dowolnej przestrzeni liniowej (oprócz zerowymiarowej) istnieje wiele różnych iloczynów skalarnych. Każdy z nich określa kąty i odległości inaczej. Możemy wybrać jeden z nich, wyróżniając pewien układ współrzędnych kartezjańskich i postulując, że wersory osi właśnie tego układu mają długość 1 i są wzajemnie do siebie prostopadłe (co określa się stwierdzeniem **tworzą układ ortonormalny**).

4.2. Układy współrzędnych

4.2.1. Współrzędne kartezjańskie



Rysunek 4.1. Punkty w układach odniesienia układów współrzędnych kartezjańskich.

Ustalmy dowolne punkty $\mathbf{p}_0, \dots, \mathbf{p}_n$ w n -wymiarowej przestrzeni afinicznej E . Jeśli układ wektorów $\mathbf{v}_1 = \mathbf{p}_1 - \mathbf{p}_0, \dots, \mathbf{v}_n = \mathbf{p}_n - \mathbf{p}_0$ jest liniowo niezależny (jest bazą przestrzeni wektorów swobodnych), to można go użyć do określenia układu współrzędnych w E ; dla dowolnego punktu $\mathbf{q} \in E$ istnieje dokładnie jeden ciąg liczb $x_1, \dots, x_n$, taki że

$$\mathbf{q} = \mathbf{p}_0 + \sum_{i=1}^n x_i(\mathbf{p}_i - \mathbf{p}_0). \quad (4.1)$$

Liczby $x_1, \dots, x_n$ to **współrzędne kartezjańskie** punktu $\mathbf{q}$.

Punkt $\mathbf{p}_0$ jest **początkiem układu** i razem z wektorami $\mathbf{v}_1, \dots, \mathbf{v}_n$ (zwanymi **wersorami osi**) tworzy **układ odniesienia** rozpatrywanego układu współrzędnych kartezjańskich.

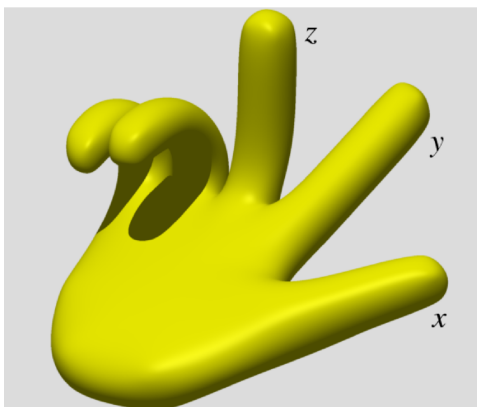
Ciąg liczb będących współrzędnymi (nie tylko kartezjańskimi) punktu wygodnie jest przedstawiać w postaci macierzy. Może to być macierz kolumnowa (tu będzie stosowana ta konwencja) lub wierszowa (spotykana często w literaturze i w różnych pakietach oprogramowania). Użycie macierzy umożliwia przedstawienie przekształceń za pomocą mnożenia macierzy; powyższe dwie konwencje różnią się wtedy kolejnością zapisu czynników.

Pewien układ współrzędnych w przestrzeni trójwymiarowej wyróżnimy i nazwiemy **układem globalnym**, albo **układem świata**. W tym układzie będziemy ustawiać rozmaite przedmioty, z których składa się scena do przedstawienia na obrazie, oraz „kamerę”, czyli obiekt określający odwzorowanie przestrzeni trójwymiarowej na płaszczyznę obrazu. Przyjmiemy, że iloczyn skalarny w tym układzie jest dany wzorem

$$\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^T \mathbf{b},$$

przy czym utożsamiliśmy tu wektory (swobodne) z ich macierzami (kolumnowymi) współrzędnych w układzie globalnym. W konsekwencji, układ odniesienia globalnego układu współrzędnych składa się z wektorów o długości 1, wzajemnie do siebie prostopadłych — wektory układu odniesienia stanowią bazę ortonormalną przestrzeni $\mathbb{R}^3$. W wielu innych układach współrzędnych iloczyn skalarny jest określony tym samym wzorem; układ odniesienia każdego takiego układu współrzędnych, który będziemy nazywać **układem prostokątnym**, jest też bazą ortonormalną. Pozostałe układy współrzędnych będziemy nazywać **układami ukośnymi**.

Dowolny układ współrzędnych jest **prawoskrętny** albo **lewoskrętny**; przynależność do jednej z tych klas nazywa się **orientacją**. Nazwa jest kwestią umowy; przyjmujemy, że układ globalny jest prawoskrętny i prawoskrętny jest też układ wektorów wyznaczonych przez kciuk, palec wskazujący i palec środkowy prawej ręki (rys. 4.2). Orientacja jest związana z kolejnością współrzędnych; przestawienie dowolnych dwóch współrzędnych (czyli przestawienie dowolnych dwóch wektorów układu odniesienia) powoduje zmianę orientacji na tę drugą.



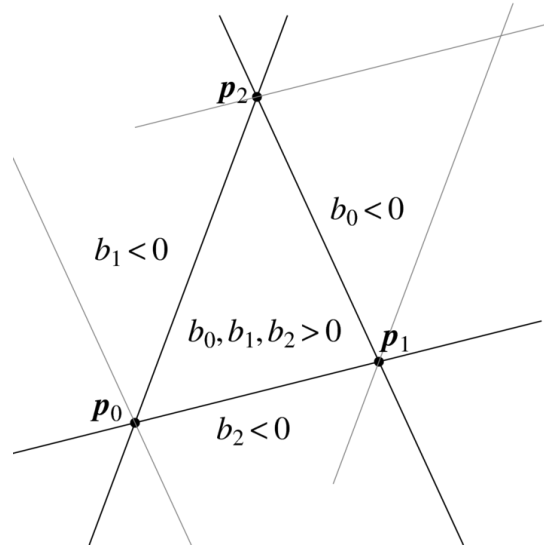
Rysunek 4.2. Określenie układu prawoskrętnego.

4.2.2. Współrzędne barycentryczne

Równość (4.1) można zapisać w bardziej symetrycznej postaci,

$$\mathbf{q} = \sum_{i=0}^n b_i \mathbf{p}_i,$$

w której $b_0 = 1 - \sum_{i=1}^n x_i$ oraz $b_i = x_i$ dla $i = 1, \dots, n$. Liczby b_i nazywają się **współrzędnymi barycentrycznymi** punktu $\mathbf{q}$ w układzie odniesienia $\mathbf{p}_0, \dots, \mathbf{p}_n$.



Rysunek 4.3. Znaki współrzędnych barycentrycznych na płaszczyźnie.

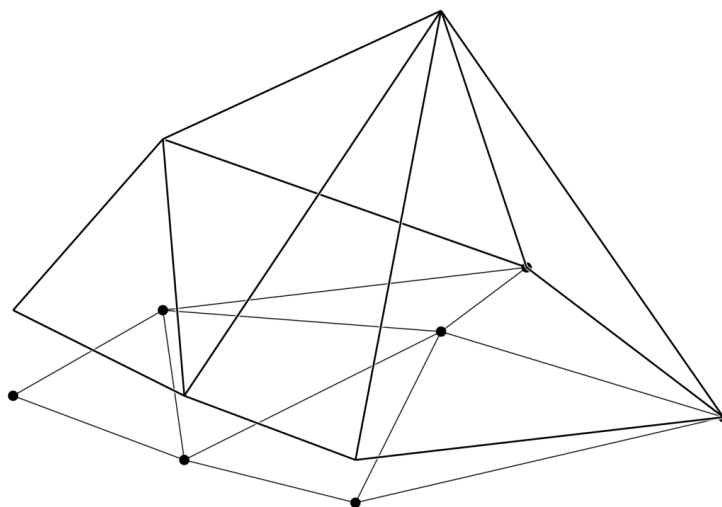
Fizyczna interpretacja współrzędnych barycentrycznych jest następująca: niech b_i oznacza masę odważnika umieszczonego w punkcie p_i . Poszczególne odważniki mogą mieć masy dodatnie, ujemne, a także równe 0, ale zakładamy, że ich suma jest równa 1. Wtedy punkt q , którego współrzędnymi barycentrycznymi w układzie odniesienia $p_0, \dots, p_n$ są liczby b_i , jest środkiem ciężkości układu punktów p_i (stąd i z greki wzięła się nazwa tych współrzędnych). Na płaszczyźnie punkty p_0, p_1, p_2 , które stanowią układ odniesienia układu współrzędnych barycentrycznych, są wierzchołkami trójkąta. Współrzędne barycentryczne dowolnego punktu wewnątrz tego trójkąta są dodatnie; punkty każdej z trzech prostych, na których leżą boki trójkąta, mają jedną ze współrzędnych barycentrycznych równą 0, a punkty na zewnątrz trójkąta mają jedną lub dwie współrzędne barycentryczne ujemne.

Przykład zastosowania: przypuśćmy, że dana jest funkcja ciągła f , której dziedziną jest wielokąt, składającym się z trójkątów o rozłącznych wnętrzach. Wykres tej funkcji w każdym trójkącie zawiera się w płaszczyźnie (czyli też jest trójkątem, zobacz rysunek (4.4)), a więc jeśli wprowadzimy układ współrzędnych *kartezjańskich* na płaszczyźnie zawierającej dziedzinę, to we wspomnianych trójkątnych fragmentach dziedziny funkcja f jest wielomianem pierwszego stopnia tych współrzędnych. Należy obliczyć wartość funkcji f w dowolnym punkcie q na podstawie wartości tej funkcji w wierzchołkach trójkąta zawierającego punkt q .

W grafice komputerowej powyższe zadanie rozwiązuje się podczas **cieniowania** trójkątów. Najprostsza (i powszechnie stosowana, a w szczególności implementowana w sprzęcie, tj. w procesorach graficznych) metoda polega na nadaniu każdemu pikselowi należącemu do trójkąta na obrazie wartości (koloru) otrzymanej przez interpolację wartości podanych w wierzchołkach (jest to tzw. cieniowanie Gourauda).

Załóżmy, że mamy obliczyć wartość funkcji f w (pojedynczym) punkcie q (którego współrzędne kartezjańskie są dane) i znamy wszystkie trójkąty, tj. współrzędne kartezjańskie ich wierzchołków i wartości funkcji f w tych punktach. Aby obliczyć wartość funkcji $f(q)$, należy

1. znaleźć wierzchołki p_i, p_j, p_k trójkąta zawierającego punkt q ,



Rysunek 4.4. Wykres funkcji kawałkami pierwszego stopnia.

2. obliczyć współrzędne barycentryczne b_i, b_j, b_k punktu $\mathbf{q}$ w układzie odniesienia $\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k$, rozwiązując układ równań liniowych

$$\begin{bmatrix} x_i & x_j & x_k \\ y_i & y_j & y_k \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_i \\ b_j \\ b_k \end{bmatrix} = \begin{bmatrix} x_q \\ y_q \\ 1 \end{bmatrix},$$

którego współczynnikami są współrzędne kartezjańskie odpowiednich punktów,

3. obliczyć $f(\mathbf{q}) = b_i f(\mathbf{p}_i) + b_j f(\mathbf{p}_j) + b_k f(\mathbf{p}_k)$.

Sposób obliczania współrzędnych barycentrycznych punktów w przestrzeni trójwymiarowej jest taki sam; układ odniesienia składa się z wierzchołków dowolnego czworościanu (tj. z dowolnych czterech punktów nie leżących w jednej płaszczyźnie). Ze współrzędnych kartezjańskich tych punktów i punktu $\mathbf{q}$, którego współrzędne barycentryczne chcemy obliczyć, tworzymy układ równań liniowych

$$\begin{bmatrix} x_i & x_j & x_k & x_l \\ y_i & y_j & y_k & y_l \\ z_i & z_j & z_k & z_l \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_i \\ b_j \\ b_k \\ b_l \end{bmatrix} = \begin{bmatrix} x_q \\ y_q \\ z_q \\ 1 \end{bmatrix},$$

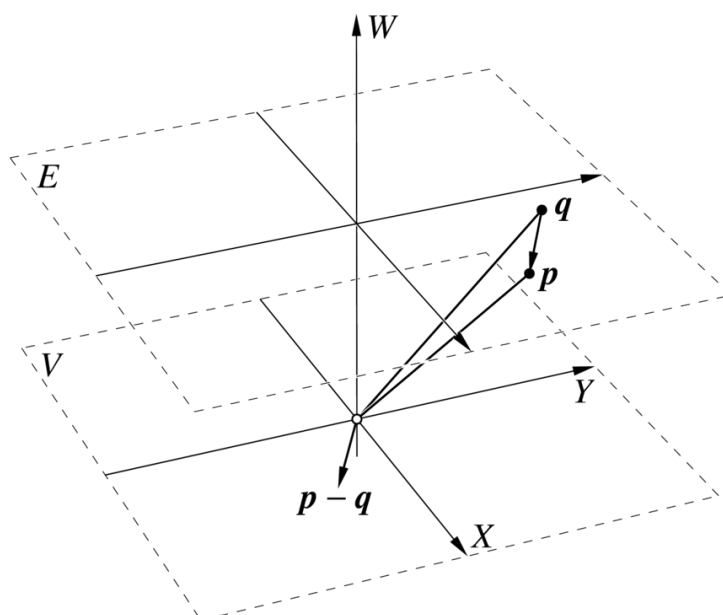
i rozwiązujemy.

4.2.3. Współrzędne jednorodne

Dowolnemu punktowi w n -wymiarowej przestrzeni afinicznej z ustalonym układem współrzędnych kartezjańskich możemy przyporządkować **współrzędne jednorodne**; jest ich $n + 1$, przy czym ostatnia z tych współrzędnych jest różna od zera. Współrzędne kartezjańskie otrzymamy dzieląc pierwsze n współrzędnych jednorodnych przez ostatnią, np. w przestrzeni trójwymiarowej punkt, którego współrzędnymi jednorodnymi są liczby X, Y, Z, W , ma współrzędne kartezjańskie $x = \frac{X}{W}, y = \frac{Y}{W}, z = \frac{Z}{W}$. Najprostszy sposób otrzymania współrzędnych jednorodnych to dołączenie jedynki do współrzędnych kartezjańskich danego punktu. Ostatnią współrzędną jednorodną będziemy nazywać **współrzędną wagową**.

Jest oczywiste, że pomnożenie *wszystkich* współrzędnych jednorodnych przez dowolną liczbę inną niż 0 daje w wyniku współrzędne jednorodne tego samego punktu. Przymiotnik „jednorodne” oznacza w matematyce właśnie tę cechę różnych obiektów. Współrzędne jednorodne wydają

się być „nieoszczędne”, jeśli chodzi o ilość zajmowanego miejsca i czas przetwarzania punktów, ale dają liczne i istotne korzyści w zastosowaniach.



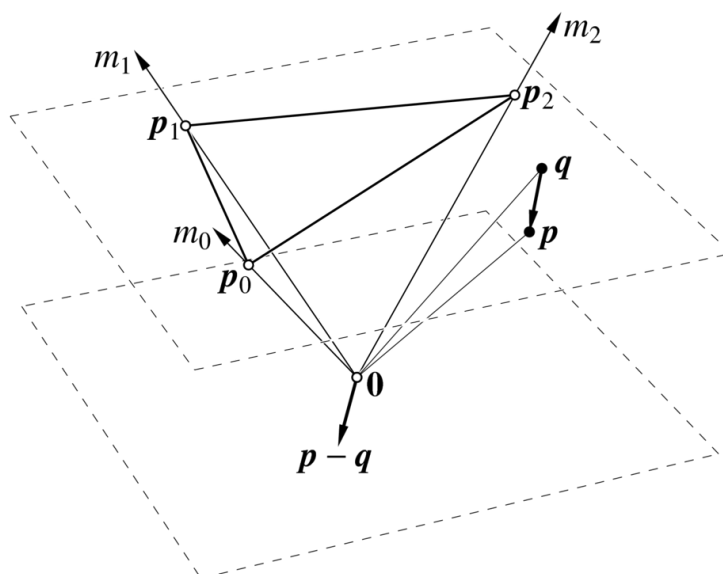
Rysunek 4.5. Model płaszczyzny afinicznej i jej przestrzeni wektorów swobodnych.

Rozważmy **przestrzeń współrzędnych jednorodnych** $\mathbb{R}^3$. Zbadamy jej związek z dwuwymiarową przestrzenią afiniczną E i jej przestrzenią wektorów swobodnych V . Przestrzeń E możemy utożsamić z warstwą $W = 1$ przestrzeni współrzędnych jednorodnych, a przestrzeń V z warstwą $W = 0$ (która jest podprzestrzenią liniową).

Jeśli $W \neq 0$, to wektor $[X, Y, W]^T$ reprezentuje punkt $[\frac{X}{W}, \frac{Y}{W}, 1]^T$. Wszystkie takie wektory reprezentują pewne punkty przestrzeni E , natomiast pozostałe reprezentują *kierunki wektorów* w przestrzeni wektorów swobodnych V . Jeśli uznamy, że nie interesują nas różnice między punktami przestrzeni E i kierunkami wektorów w V (które są nazywane **punktami niewłaściwymi**), to otrzymamy **przestrzeń rzutową** i okaże się, że zajmujemy się geometrią rzutową.

Jeśli chcemy wykonywać rachunki na punktach i wektorach reprezentowanych przez wektory współrzędnych jednorodnych, dopuszczając różne wagi (czyli różne wartości ostatniej współrzędnej jednorodnej), to musimy „uzgodnić” reprezentacje. Rozważmy przykład — wyznaczanie środka odcinka, którego końce są reprezentowane przez macierze $[0, 0, 1]^T$ i $[-2, 0, -2]^T$. Jeśli obliczymy macierz $\frac{1}{2}([0, 0, 1]^T + [-2, 0, -2]^T) = [-1, 0, -\frac{1}{2}]^T$, to otrzymamy punkt który nie leży nawet na naszym odcinku (współrzędne kartezjańskie końców tego odcinka to $[0, 0]^T$ i $[1, 0]^T$, a otrzymana macierz reprezentuje punkt $[2, 0]^T$). Aby dostać poprawny wynik, należy pomnożyć macierz współrzędnych jednorodnych jednego punktu przez taki czynnik, aby ostatnia współrzędna obu argumentów była taka sama. Nieco większy kłopot sprawiają wektory swobodne; macierz współrzędnych jednorodnych reprezentuje tylko kierunek takiego wektora i trzeba „z zewnątrz” dostarczyć informację, jakiej współrzędnej wagowej punktów odpowiada dana macierz współrzędnych jednorodnych.

Podobnie jak współrzędne kartezjańskie, możemy „ujednorodnić” także współrzędne barycentryczne; wystarczy opuścić założenie, że ich suma jest równa 1. Podanie dla ustalonego układu współrzędnych barycentrycznych ciągu dowolnych $n + 1$ liczb $m_0, \dots, m_n$, których suma jest różna od 0, określa punkt, którego współrzędne barycentryczne otrzymamy dzieląc te liczby przez ich sumę. Oczywiście, suma jednorodnych współrzędnych barycentrycznych, które interpretujemy jako masy odważników, jest współrzędną wagową punktu. Jeśli suma jednorod-



Rysunek 4.6. Barycentryczne współrzędne jednorodne.

nych współrzędnych barycentrycznych jest równa 0, to określają one pewien kierunek wektorów swobodnych, czyli punkt niewłaściwy.

4.3. Przekształcenia afiniczne

4.3.1. Definicja i własności

Przekształcenie $f: E \rightarrow E$ jest **afiniczne**, jeśli dla każdego układu punktów $p_0, \dots, p_m$ i liczb $b_0, \dots, b_m$, takich że $\sum_{i=0}^m b_i = 1$ jest spełniony warunek

$$f\left(\sum_{i=0}^m b_i p_i\right) = \sum_{i=0}^m b_i f(p_i).$$

To oznacza w szczególności, że obrazem dowolnej prostej jest prosta (prosta to zbiór $\{q: q = tp_0 + (1-t)p_1, t \in \mathbb{R}\}$ dla ustalonych punktów $p_0 \neq p_1$), albo punkt. Obrazem prostych równoległych są punkty albo proste równoległe (bo jeśli proste $\ell_1 = \{q: q = tp_0 + (1-t)p_1, t \in \mathbb{R}\}$ i $\ell_2 = \{q: q = tp_2 + (1-t)p_3, t \in \mathbb{R}\}$ są równoległe, czyli wektory $p_1 - p_0$ i $p_3 - p_2$ są liniowo zależne, to łatwo jest sprawdzić, że ich obrazy w przekształceniu f są punktami albo spełniają tę samą definicję równoległości).

Własności figur, zachowywane przez przekształcenia afiniczne, nazywają się **niezmiennikami afinicznymi**. Dalsze ich przykłady to

- współliniowość i współpłaszczyznowość punktów,
- równoległość prostych i płaszczyzn,
- wypukłość figury,
- proporcje odległości punktów współliniowych (ten i następne przykłady dotyczą przekształceń różnowartościowych),
- bycie trójkątem,
- bycie równoległobokiem,
- bycie elipsą.

Jeśli mamy ustalony układ współrzędnych (kartezjański), to obliczenia punktów sprowadzamy do rachunków na wektorach współrzędnych. Ogólnie, przekształcenie afiniczne można zapisać w postaci

$$f(\mathbf{p}) = A\mathbf{p} + \mathbf{t},$$

w której macierz A reprezentuje **część liniową** przekształcenia f , a wektor $\mathbf{t}$ — **przesunięcie**.

Przekształcenie jest różnowartościowe wtedy gdy macierz A jest nieosobliwa. Macierz ta opisuje związane z przekształceniem f przekształcenie liniowe przestrzeni wektorów swobodnych:

$$f(\mathbf{p}) - f(\mathbf{q}) = (A\mathbf{p} + \mathbf{t}) - (A\mathbf{q} + \mathbf{t}) = A(\mathbf{p} - \mathbf{q}).$$

Złożenie przekształceń afinicznych jest przekształceniem afinicznym. Odwrotność (jeśli istnieje) też.

4.3.2. Jednorodna reprezentacja przekształceń afinicznych

Często trzeba wyznaczyć złożenie przekształceń afinicznych; możemy je opisać wzorem

$$f_2(f_1(\mathbf{p})) = A_2(A_1\mathbf{p} + \mathbf{t}_1) + \mathbf{t}_2 = (A_2A_1)\mathbf{p} + (A_2\mathbf{t}_1 + \mathbf{t}_2),$$

który jest niewygodny (i jeszcze mniej wygodne wzory dostaniemy chcąc opisać złożenie większej liczby przekształceń). Jeśli do współrzędnych kartezjańskich x, y, z punktu $\mathbf{p}$ dopiszemy 1, to możemy napisać

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & & & \\ & A & & \mathbf{t} \\ & & & \\ 0 & 0 & 0 & 1 \end{array} \right] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

W ten sposób pojawiają się współrzędne jednorodne, dzięki którym przekształcenie afiniczne możemy przedstawić w wygodny sposób, za pomocą jednego mnożenia macierzy.

Macierze A i $\mathbf{t}$ są blokami macierzy jednorodnej przekształcenia f (to jest jeden z dwóch głównych powodów, dla których działania na macierzach 4×4 i 4×1 są często realizowane przez sprzęt). Składanie przekształceń afinicznych reprezentowanych w takiej postaci polega na mnożeniu macierzy reprezentujących poszczególne przekształcenia.

Możemy też jednolicie traktować punkty i wektory swobodne. Istotnie, wystarczy do współrzędnych wektora swobodnego dopisać 0 i dalej przekształcenie wektora za pomocą mnożenia macierzy jest wykonywane poprawnie (otrzymujemy wynik działania części liniowej przekształcenia f na wektor).

4.3.3. Przekształcanie wektora normalnego

Rozważmy płaszczyznę w przestrzeni trójwymiarowej,

$$\pi = \{ \mathbf{x} : \langle \mathbf{x} - \mathbf{p}, \mathbf{n} \rangle = 0 \},$$

daną za pomocą punktu $\mathbf{p}$ i wektora normalnego $\mathbf{n}$. Załóżmy, że iloczyn skalarny jest dany wzorem $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{y}^T \mathbf{x}$, a zatem równanie płaszczyzny π (które jest spełnione przez wszystkie punkty $\mathbf{x}$ tej płaszczyzny i przez żadne inne) ma postać

$$\mathbf{n}^T(\mathbf{x} - \mathbf{p}) = 0.$$

Niech f oznacza różnowartościowe przekształcenie afiniczne, którego część liniowa jest reprezentowana przez macierz A . Obraz płaszczyzny π w przekształceniu f jest płaszczyzną zawierającą punkt $f(\mathbf{p})$, który łatwo jest znaleźć, a jej wektor normalny też jest łatwy do znalezienia. Możemy mianowicie napisać

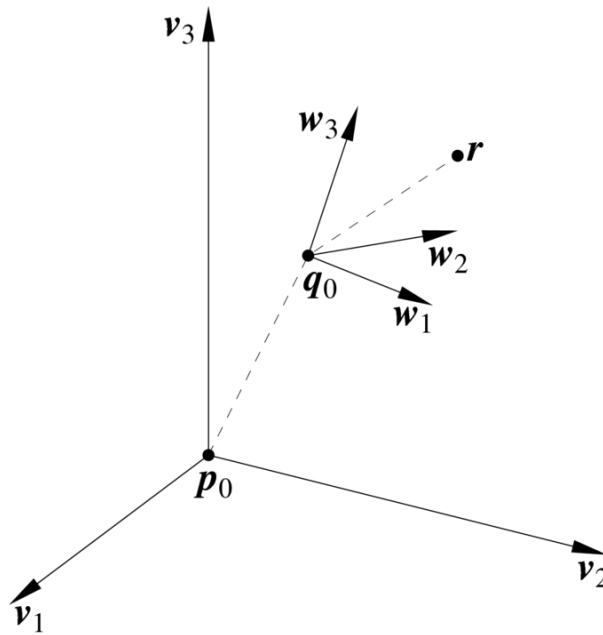
$$0 = \mathbf{n}^T A^{-1} A(\mathbf{x} - \mathbf{p}) = \mathbf{n}^T A^{-1} (f(\mathbf{x}) - f(\mathbf{p})),$$

skąd wynika, że tym wektorem jest $(A^{-1})^T \mathbf{n}$ (zamiast $(A^{-1})^T$ możemy w skrócie pisać A^{-T}). Warto o tym pamiętać z uwagi na rolę, jaką spełnia wektor normalny we wszystkich modelach oświetlenia powierzchni.

4.3.4. Zmiana układu współrzędnych

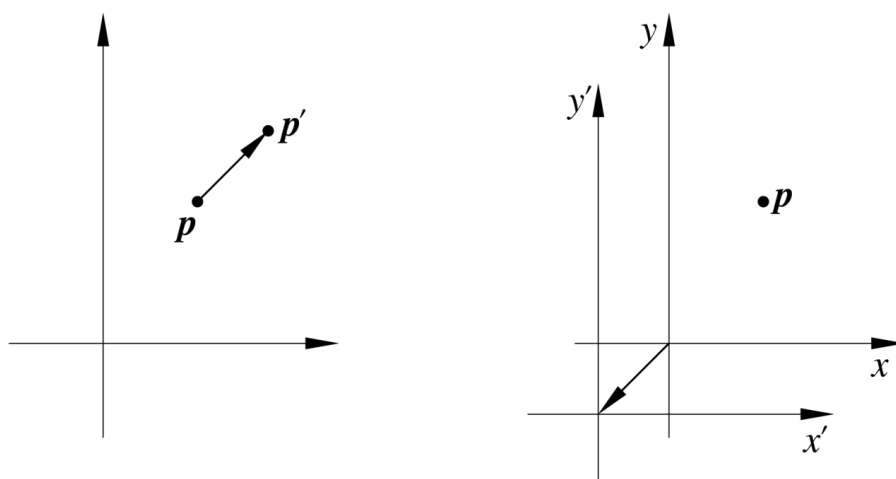
Przypuśćmy, że punkt $\mathbf{r}$ ma w układzie współrzędnych określonym przez układ odniesienia $(\mathbf{q}_0; \mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3)$ współrzędne jednorodne $[x, y, z, 1]^T$, a z kolei punkt $\mathbf{q}_0 = [x_q, y_q, z_q, 1]^T$ i wektory $\mathbf{w}_i = [x_i, y_i, z_i, 0]^T$ znamy mając ich współrzędne jednorodne w układzie $(\mathbf{p}_0; \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$. Wtedy współrzędne punktu $\mathbf{r}$ w tym drugim układzie spełniają równość

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_q \\ y_1 & y_2 & y_3 & y_q \\ z_1 & z_2 & z_3 & z_q \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$



Rysunek 4.7. Zmiana układu współrzędnych.

Zatem, wzór opisujący przekształcenie afiniczne (mnożenie wektora współrzędnych przez macierz) można też interpretować jako przejście do nowego układu współrzędnych. Często w grafice komputerowej modelujemy sceny złożone z wielu obiektów, z których każdy jest opisywany osobno w wygodnym dla opisu tego obiektu układzie. Składanie sceny z takich obiektów jest równoważne określeniu sposobu przejścia od ich układów współrzędnych do układu całej sceny, co polega na podaniu odpowiednich macierzy.



Rysunek 4.8. Dualne interpretacje przesunięcia: przekształcenie punktu i zmiana układu współrzędnych.

Często należy znaleźć macierz przekształcenia, które jest określone przy użyciu innego układu odniesienia niż układ dany. Aby to zrobić, należy dokonać odpowiedniego przejścia. Na przykład, niech macierz R opisuje (we współrzędnych jednorodnych) obrót wokół prostej ℓ przechodzącej przez początek układu. Niech T oznacza macierz przesunięcia, która początek p_0 wyjściowego układu przekształca na punkt q_0 . Aby znaleźć obraz punktu x w obrocie o ten sam kąt wokół prostej równoległej do ℓ i przechodzącej przez punkt q_0 , należy kolejno

1. obliczyć współrzędne punktu x w układzie przesuniętym (którego początkiem jest punkt q_0 ; macierz zmiany układu jest równa T^{-1} ,
2. dokonać obrotu,
3. obliczyć współrzędne obrazu w wyjściowym układzie; odpowiednią macierzą jest macierz T . Zatem poszukiwana macierz jest równa iloczynowi TRT^{-1} . Podobne wyrażenia opisują wszelkie przekształcenia określone w innym układzie niż układ wyjściowy.

4.3.5. Szczególne przypadki przekształceń afinicznych

Przesunięcia Część liniowa przesunięcia jest opisana przez macierz jednostkową, a zatem przekształcenia te nie zmieniają kierunku żadnej prostej, ani długości żadnego odcinka.

Skalowania Skalowanie to przekształcenie, którego część liniowa jest w pewnym układzie reprezentowana przez macierz diagonalną. Podobnie jak przesunięcie o wektor t można interpretować jako przesunięcie układu odniesienia o $-t$, zaś obrót o kąt φ jako obrót układu o kąt $-\varphi$, skalowanie ze współczynnikami s_x, s_y, s_z (na diagonalu macierzy) może być traktowane jak przejście do układu z jednostkami osi różniącymi się od dotychczasowych o czynniki $1/s_x, 1/s_y$ i $1/s_z$.

Skalowanie oczywiście nie jest różnowartościowe, jeśli któryś ze współczynników, s_x, s_y lub s_z jest równy 0; w przeciwnym razie istnieje przekształcenie odwrotne, które jest skalowaniem ze współczynnikami $1/s_x, 1/s_y$ i $1/s_z$.

Niezmiennikami skalowań różnowartościowych są kierunki prostych równoległych do osi układu. Klasa skalowań wymieniona wyżej jest bardzo obszerna i dlatego w praktyce często przez skalowanie rozumie się przekształcenie, którego część liniowa jest diagonalna a przesunięcie zerowe w układzie współrzędnych określonym przez układ odniesienia o wzajemnie prostopadłych wersorach osi. Jeśli osie układu równań, w którym część liniowa skalowania jest diagonalna,

są wzajemnie prostopadłe, to macierz skalowania w każdym układzie współrzędnych o wzajemnie prostopadłych osiach jest **symetryczna** (współczynniki skalowania osi są wartościami własnymi, a wektory własne wyznaczają wzajemnie prostopadłe kierunki skalowania).

Jeśli wszystkie współczynniki są równe 0 albo 1, to mamy do czynienia z **rzutem**. Jeśli natomiast $s_x, s_y, s_z \in \{-1, 1\}$, to mamy **odbicie symetryczne**. W przestrzeni trójwymiarowej mamy takie możliwości rzutów i odbić (zakładamy we wzorach, że $\|\mathbf{v}\|_2 = 1$)

- $\mathbf{p}' = \mathbf{0}$ — rzut na początek układu.
- $\mathbf{p}' = \mathbf{v}\langle\mathbf{v}, \mathbf{p}\rangle$ — rzut na prostą o kierunku wektora $\mathbf{v}$.
- $\mathbf{p}' = \mathbf{p} - \mathbf{v}\langle\mathbf{v}, \mathbf{p}\rangle$ — rzut na płaszczyznę prostopadłą do $\mathbf{v}$.
- $\mathbf{p}' = \mathbf{p}$ — rzut na całą przestrzeń, a także odbicie względem całej przestrzeni (czyli przekształcenie tożsamościowe).
- $\mathbf{p}' = \mathbf{p} - 2\mathbf{v}\langle\mathbf{v}, \mathbf{p}\rangle$ — odbicie względem płaszczyzny prostopadłej do $\mathbf{v}$.
- $\mathbf{p}' = 2\mathbf{v}\langle\mathbf{v}, \mathbf{p}\rangle - \mathbf{p}$ — odbicie względem prostej o kierunku $\mathbf{v}$.
- $\mathbf{p}' = -\mathbf{p}$ — odbicie symetryczne względem początku układu.

Powyższe rzuty i odbicia są określone za pomocą pojęcia prostopadłości. Nie jest ono potrzebne do określania **rzutów równoległych**. Takie przekształcenie jest określone za pomocą wektora $\mathbf{v} \neq \mathbf{0}$ i płaszczyzny zwanej **rzutnią**, nie zawierającej tego wektora. Aby dokonać rzutowania, należy

1. wybrać dowolny punkt $\mathbf{p}_0$ rzutni oraz dwa liniowo niezależne wektory $\mathbf{w}_1$ i $\mathbf{w}_2$ równoległe do rzutni,
2. przejść do układu współrzędnych o początku $\mathbf{p}_0$ i wersorach osi $\mathbf{v}$, $\mathbf{w}_1$ i $\mathbf{w}_2$,
3. dokonać rzutowania; macierz rzutu w tym układzie ma na diagonalu współczynniki 0, 1, 1,
4. wrócić do układu wyjściowego — wcześniej były opisane wszystkie potrzebne szczegóły.

Obroty Obroty w płaszczyźnie $\mathbb{E}^2$ są jednoznacznie określone przez środek obrotu i kąt. Macierz części liniowej obrotu ma postać

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix},$$

gdzie $c = \cos \varphi$, $s = \sin \varphi$.

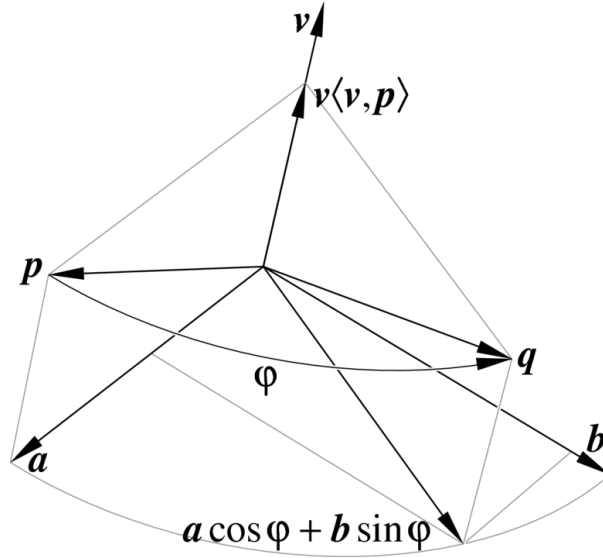
Macierz części liniowej obrotu jest ortogonalna, o wyznaczniku równym 1. W przestrzeni trójwymiarowej dla każdej takiej macierzy istnieje oś oraz kąt, takie że pomnożenie dowolnego wektora przez tę macierz jest równoważne obróceniu tego wektora wokół tej osi o ten kąt. Zamiast mówić „obrót wokół osi”, można też mówić „obrót w płaszczyźnie”, mając na myśli płaszczyznę prostopadłą do osi obrotu.

Obroty wokół osi x , y , z , czyli obroty w płaszczyznach odpowiednio yz , zx i xy , są opisane przez macierze, których części liniowe są następujące:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix}, \quad \begin{bmatrix} c & 0 & s \\ 0 & 1 & 0 \\ -s & 0 & c \end{bmatrix}, \quad \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Obrót wokół osi o kierunku dowolnego wektora $\mathbf{v}$ można przedstawić jako złożenie pięciu obrotów reprezentowanych przez powyższe macierze (dwa obroty przekształcają oś obrotu na oś np. z układu współrzędnych, następnie należy wykonać obrót wokół tej osi, a na końcu wrócić do wyjściowego układu współrzędnych), ale znacznie wygodniejszą metodą jest użycie bezpośredniego wzoru opisującego macierz takiego obrotu. Wyprowadzimy go.

Niech $\|\mathbf{v}\|_2 = 1$. Poddawany przekształceniu wektor $\mathbf{p}$ możemy rozłożyć na dwa, wzajemnie prostopadłe składniki: $\mathbf{v}\langle\mathbf{v}, \mathbf{p}\rangle$ — obraz $\mathbf{p}$ w rzucie prostopadłym na kierunek wektora $\mathbf{v}$ i $\mathbf{a} = \mathbf{p} - \mathbf{v}\langle\mathbf{v}, \mathbf{p}\rangle$ — obraz $\mathbf{p}$ w rzucie na płaszczyznę obrotu (prostopadłą do $\mathbf{v}$). Oznaczmy $\mathbf{b} =$

Rysunek 4.9. Obrót wokół osi o kierunku wektora v .

$v \wedge a = v \wedge p$. Obraz p' punktu p , obróconego w płaszczyźnie prostopadłej do v o kąt φ , jest równy

$$q = v\langle v, p \rangle + a \cos \varphi + b \sin \varphi.$$

Macierz obrotu jest sumą trzech macierzy opisujących przekształcenia liniowe, dzięki którym otrzymaliśmy powyższe składniki. Zatem,

$$\begin{aligned} v\langle v, p \rangle &= (vv^T)p, \\ a &= p - v\langle v, p \rangle = (I_3 - vv^T)p, \\ b &= v \wedge p = (v \wedge I_3)p. \end{aligned}$$

Kolumny macierzy $v \wedge I_3$ są iloczynami wektorowymi wektora v i odpowiednich kolumn macierzy jednostkowej 3×3 . Stąd macierz obrotu jest równa

$$vv^T + (I_3 - vv^T) \cos \varphi + v \wedge I_3 \sin \varphi.$$

Złożenie dwóch obrotów w przestrzeni trójwymiarowej jest obrotem; mając dane wektory jednostkowe v_1 i v_2 osi obrotu i kąty φ_1 i φ_2 tych obrotów możemy znaleźć oś i kąt obrotu, który jest złożeniem tych dwóch. Wektor v wyznaczający kierunek osi i kąt φ tego obrotu spełniają równości

$$\begin{aligned} \cos \frac{\varphi}{2} &= \cos \frac{\varphi_1}{2} \cos \frac{\varphi_2}{2} - \langle v_1, v_2 \rangle \sin \frac{\varphi_1}{2} \sin \frac{\varphi_2}{2}, \\ \sin \frac{\varphi}{2} &= \|w\|_2, \\ v &= \frac{w}{\|w\|_2}, \end{aligned}$$

w których występuje wektor w określony wzorem

$$w = v_2 \sin \frac{\varphi_2}{2} \cos \frac{\varphi_1}{2} + v_1 \sin \frac{\varphi_1}{2} \cos \frac{\varphi_2}{2} + v_2 \wedge v_1 \sin \frac{\varphi_1}{2} \sin \frac{\varphi_2}{2}.$$

Wyprowadzenie powyższych wzorów jest łatwe przy użyciu kwaternionów, o których będzie mowa dalej.

Konstrukcja obrotu do ustalonego położenia Podczas modelowania sceny trójwymiarowej możemy napotkać następujący problem: mamy dane trzy punkty niewspółliniowe, $\mathbf{p}_0$, $\mathbf{p}_1$ i $\mathbf{p}_2$, które jednoznacznie określają położenie pewnego obiektu (bryły sztywnej). Chcemy ten obiekt umieścić tak, aby te punkty znalazły się we wskazanej płaszczyźnie. W tym celu musimy obiekt odpowiednio obrócić i przesunąć. Skonstruujemy macierz reprezentującą potrzebny obrót (konstrukcja odpowiedniego przesunięcia jest prostym ćwiczeniem).

Uściślijmy zadanie: płaszczyzna, w której mają znaleźć się dane punkty jest określona przez podanie punktów $\mathbf{q}_1$, $\mathbf{q}_2$ i $\mathbf{q}_3$, też niewspółliniowych. Chcemy, aby dla skonstruowanego obrotu, reprezentowanego przez macierz R ,

- obraz wektora $\mathbf{w}_1 = \mathbf{p}_1 - \mathbf{p}_0$ był równoległy do wektora $\mathbf{v}_1 = \mathbf{q}_1 - \mathbf{q}_0$ i miał ten sam zwrot,
- obraz wektora $\mathbf{w}_2 = \mathbf{p}_2 - \mathbf{p}_0$ był kombinacją liniową wektorów $\mathbf{v}_1$ i $\mathbf{v}_2 = \mathbf{q}_2 - \mathbf{q}_1$, a ponadto ma być $\langle R\mathbf{w}_2, \mathbf{v}_2 \rangle > 0$.

Macierz R przekształcenia, które jest obrotem, musi być ortogonalna i jej wyznacznik musi być równy 1. Rozwiązanie tak postawionego zadania jest jednoznaczne.

Niech $\mathbf{w}_3 = \mathbf{w}_1 \wedge \mathbf{w}_2$. Rozważmy macierz $A = [\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3]$. Wyznacznik tej macierzy jest dodatni. Macierz ta opisuje przejście od układu współrzędnych o układzie odniesienia $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ do układu współrzędnych, którego wersorami osi są wektory $\mathbf{e}_1 = [1, 0, 0]^T$, $\mathbf{e}_2 = [0, 1, 0]^T$ i $\mathbf{e}_3 = [0, 0, 1]^T$. Istnieją (jednoznacznie określone) macierze Q_A i R_A , takie że $A = Q_A R_A$, macierz Q_A jest ortogonalna (tj. $Q_A^T = Q_A^{-1}$) i $\det Q_A = 1$, a macierz R_A jest trójkątna górna i współczynniki na jej diagonalu są dodatnie (mamy $\det R_A = \det A$; wyznacznik ten jest równy iloczynowi współczynników diagonalnych macierzy R_A).

Możemy zauważyć, że kolumna $\mathbf{q}_{A1}$ macierzy Q_A ma kierunek i zwrot wektora $\mathbf{w}_1$, zaś kolumna $\mathbf{q}_{A2}$ tej macierzy leży w przestrzeni rozpiętej przez $\mathbf{w}_1$ i $\mathbf{w}_2$, a ponadto $\langle \mathbf{q}_{A2}, \mathbf{w}_2 \rangle > 0$.

Macierz Q_A jest więc macierzą obrotu, który przekształca wersory $\mathbf{e}_1$, $\mathbf{e}_2$ i $\mathbf{e}_3$ na $\mathbf{q}_{A1}$, $\mathbf{q}_{A2}$ i $\mathbf{q}_{A3}$; macierz $Q_A^{-1} = Q_A^T$ dokonuje przekształcenia odwrotnego. Obraz wektora $\mathbf{w}_1$ w tym przekształceniu ma kierunek i zwrot wektora $\mathbf{e}_1$, zaś obraz wektora $\mathbf{w}_2$ leży w płaszczyźnie rozpiętej przez $\mathbf{e}_1$ i $\mathbf{e}_2$.

Podobnie możemy postąpić z macierzą $B = [\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3] = Q_B R_B$, której ostatnia kolumna $\mathbf{v}_3 = \mathbf{v}_1 \wedge \mathbf{v}_2$. Poszukiwana macierz R obrotu, który spełnia postawione warunki, jest równa $Q_B Q_A^T$. Macierze Q_A i Q_B możemy obliczyć na różne sposoby, na przykład dokonując ortogonalizacji Grama-Schmidta kolumn macierzy A i B .

Przykład składania przekształceń Przypuśćmy, że chcemy otrzymać macierz obrotu na płaszczyźnie, o kąt 30° wokół punktu $[200, 100]^T$. Przekształcenie takie otrzymujemy w następujących krokach:

1. Zmieniamy układ współrzędnych tak, aby środek obrotu był początkiem układu. Odpowiednia macierz ma postać

$$\begin{bmatrix} 1 & 0 & -200 \\ 0 & 1 & -100 \\ 0 & 0 & 1 \end{bmatrix}.$$

2. Wykonujemy obrót o 30° wokół początku układu. Macierz obrotu ma postać

$$\begin{bmatrix} \sqrt{3}/2 & -1/2 & 0 \\ 1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

3. Wracamy do wyjściowego układu, co opisuje macierz

$$\begin{bmatrix} 1 & 0 & 200 \\ 0 & 1 & 100 \\ 0 & 0 & 1 \end{bmatrix}.$$

Macierz całego przekształcenia jest iloczynem powyższych trzech:

$$\begin{bmatrix} 1 & 0 & 200 \\ 0 & 1 & 100 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \sqrt{3}/2 & -1/2 & 0 \\ 1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -200 \\ 0 & 1 & -100 \\ 0 & 0 & 1 \end{bmatrix} = \\ \begin{bmatrix} \sqrt{3}/2 & -1/2 & 250 - 100\sqrt{3} \\ 1/2 & \sqrt{3}/2 & -50\sqrt{3} \\ 0 & 0 & 1 \end{bmatrix}.$$

4.3.6. Składanie przekształceń w zastosowaniach graficznych

Zbadamy teraz sposoby składania przekształceń w trzech typowych sytuacjach, z jakimi mamy do czynienia w grafice. Interesuje nas kolejność, w jakiej trzeba mnożyć macierze reprezentujące składane przekształcenia.

Ustawianie obiektu w scenie Przypuśćmy, że pewien obiekt, O , został określony przy użyciu wprowadzonego w tym celu („lokalnego”) układu współrzędnych kartezjańskich. Mamy też drugi układ, „pośredni”, w którym chcemy „ustawić” obiekt A . Układ odniesienia (tj. początek układu i wersory osi) układu lokalnego jest obrazem układu odniesienia „pośredniego” układu współrzędnych w przekształceniu afinicznym f_1 reprezentowanym (w postaci jednorodnej) przez macierz A_1 .

Mamy też trzeci, „globalny” układ współrzędnych. Układ odniesienia układu „pośredniego” jest obrazem układu odniesienia „globalnego” układu współrzędnych w przekształceniu f_2 , którego macierzą jest A_2 . Jeśli wektor v składa się ze współrzędnych jednorodnych w „lokalnym” układzie pewnego punktu obiektu O , to wektorem współrzędnych jednorodnych tego punktu w układzie „pośrednim” jest wektor $A_1 p$, zaś w układzie „globalnym” wektor $A_2 A_1 p$.

Przypuśćmy, że obiekt O jest „generowany” przez pewną procedurę (nazwijmy ją O). Dokładniej, procedura ta „wytwarza” pewne punkty (np. wierzchołki trójkątów), obliczając ich współrzędne w układzie „lokalnym”. Procedura O „nie wie” niczego o innych układach współrzędnych. Inna procedura, P , wywołuje O , poprzedzając to określeniem przekształcenia układu, w którym O podaje współrzędne punktów, do swojego („pośredniego”) układu współrzędnych. Procedura P jest z kolei wywoływana przez pewną procedurę G , która przed wywołaniem P określa przejście od jej układu współrzędnych do układu „globalnego”. „Określenie” przejścia polega na nadaniu odpowiedniej wartości pewnej macierzy (która może być przechowywana w ustalonej tablicy w programie, lub w rejestrach urządzenia graficznego). Jeśli początkowa macierz przejścia (od układu „globalnego” do „globalnego”) jest jednostkowa, to procedura G przed wywołaniem P pomnoży ją przez A_2 (i przypisze macierzy przejścia iloczyn). Dalej, procedura P przed wywołaniem O pomnoży tę macierz przez A_1 . Jak widać, każda kolejna macierz musi być „domnożona” **z prawej strony**.

Typowe dla tej sytuacji jest użycie **stosu macierzy przekształceń**. Jeśli procedura P ustawia więcej niż jeden obiekt, przy czym każdy z tych obiektów jest określony w układzie, z którego przejście do układu „pośredniego” jest inne, to procedura P powinna zapamiętać (na stosie) bieżącą macierz przekształcenia (od swojego „pośredniego” układu do „globalnego”). Następnie powinna dla każdego obiektu pomnożyć kopię tej macierzy przez macierz przejścia

od „lokalnego” układu tego obiektu do układu pośredniego. Po zakończeniu działania procedury wprowadzającej obiekt procedura P powinna odtworzyć macierz przejścia taką, jaka była w chwili jej wywołania.

Zarówno w języku PostScript, jak i w bibliotece OpenGL jest to podstawowy sposób określania przekształceń złożonych. W obu tych przypadkach są dostępne odpowiednie stopy.

Wykonywanie kolejnych przekształceń Druga sytuacja jest typowa dla animacji: pewien obiekt O jest określony w swoim, „lokalnym” układzie współrzędnych. Układ ten początkowo pokrywa się z układem „globalnym”. Chcemy uzyskać serię obrazów, w których każdy przedstawia obiekt O w kolejnym położeniu.

Jeśli macierz przekształcenia od położenia p_{n-1} do p_n , reprezentowanego w „globalnym” układzie współrzędnych, oznaczmy A_n , to oczywiście macierz, która „przestawia” obiekt z położenia p_0 do p_n jest równa $A_n A_{n-1} \dots A_2 A_1$. W tym przypadku macierz każdego kolejnego przekształcenia jest czynnikiem **z lewej strony**.

Grafika żółwia Mamy sytuację podobną do poprzedniej: chcemy poddać pewien obiekt serii przekształceń (po to, aby otrzymać jego kolejne położenia), ale tym razem określimy każde kolejne przekształcenie w układzie „lokalnym”, związanym z aktualnym położeniem obiektu. Rozważmy dwa takie przekształcenia. Niech A_1 oznacza macierz pierwszego z nich; ponieważ początkowo „lokalny” układ pokrywa się z „globalnym”, więc jedno (pierwsze) przekształcenie wykonujemy jak poprzednio. Drugie przekształcenie jest reprezentowane przez macierz A_2 w układzie współrzędnych, którego układ odniesienia jest obrazem układu odniesienia układu globalnego w pierwszym przekształceniu.

Macierz drugiego przekształcenia w układzie globalnym jest równa $A_1 A_2 A_1^{-1}$. Jeśli zatem zastosujemy przepis na składanie kolejno wykonywanych przekształceń określonych w układzie „globalnym”, to otrzymamy macierz $A_1 A_2 A_1^{-1} A_1 = A_1 A_2$. Zatem, jeśli kolejne przekształcenie mamy określone w układzie związanym z bieżącym położeniem obiektu, to odpowiednia macierz jest czynnikiem **z prawej strony**.

Opisana sytuacja jest najczęściej związana z tzw. **grafiką żółwia**. Wytresowany żółw porusza się po płaszczyźnie lub w przestrzeni i wykonuje kolejne polecenia takie jak „idź jeden krok do przodu” lub „obróć się w lewo”. Przemieszczając się żółw zostawia ślad (np. rysuje odcinki). Ta technika wykonywania obrazów jest niezastąpiona np. podczas rysowania roślin.

4.3.7. Rozkładanie przekształceń

Każde przekształcenie afiniczne jest złożeniem obrotów, skalowań i przesunięć. Interesującym problemem, którego rozwiązanie bywa potrzebne w praktyce, jest znalezienie tych przekształceń na podstawie danej macierzy reprezentującej pewne przekształcenie afiniczne.

Dowolne przekształcenie afiniczne jest złożeniem przekształcenia liniowego i przesunięcia. Współrzędne wektora przesunięcia są dane w czwartej kolumnie macierzy (reprezentacji jednorodnej). Pozostaje niebanalny problem rozłożenia macierzy 3×3 , opisującej część liniową przekształcenia afinicznego, na macierze obrotów i skalowań. W tym celu przypomnijmy, że każda macierz rzeczywista jest iloczynem trzech macierzy:

$$A = U \Sigma V^T,$$

takich że U i V są ortogonalne, a macierz Σ jest diagonalna, o nieujemnych współczynnikach na diagonalu. Jest to tak zwany **rozkład SVD** (albo **rozkład względem wartości szczególnych**). W naszym przypadku wszystkie te macierze mają wymiary 3×3 . Zmieniając odpowiednio znaki wierszy macierzy U i kolumn macierzy V^T oraz współczynników diagonalnych

macierzy Σ , możemy otrzymać macierze U i V^T , których wyznaczniki są równe 1. Rozważymy jeszcze jeden rozkład, tzw. **rozkład biegunowy**:

$$A = QS,$$

w którym $Q = UV^T$ oraz $S = V\Sigma V^T$. Macierz Q jest ortogonalna, a macierz S jest symetryczna. Jeśli macierz A jest nieosobliwa, to rozkład biegunowy możemy łatwo znaleźć za pomocą **algorytmu Highama**. Przyjmujemy $Q_0 = A$, a następnie obliczamy kolejne macierze

$$Q_k = \frac{1}{2}(Q_{k-1} + Q_{k-1}^{-T}), \quad k = 1, 2, \dots$$

Otrzymany ciąg macierzy bardzo szybko zbiega do macierzy Q ; w praktyce często wystarczy wykonać kilka iteracji. Macierz S otrzymamy, obliczając iloczyn $S = Q^T A$.

Mając macierz S , możemy znaleźć macierze V i Σ przez rozwiązanie algebraicznego zagadnienia własnego. Macierz diagonalna Σ reprezentuje skalowanie. Macierz S też reprezentuje skalowanie, w kierunkach pewnych trzech prostych wzajemnie prostopadłych; kolumny macierzy V mają te kierunki. Oczywiście, współczynnikami skalowania są wartości własne macierzy S i Σ . Znając macierz V , możemy obliczyć macierz $U = QV$.

Macierz ortogonalna 3×3 ma wyznacznik równy $+1$ albo -1 . W pierwszym przypadku jest to macierz obrotu; jej wartościami własnymi są liczby zespolone 1 , (c, s) i $(c, -s)$. Oś obrotu ma kierunek wektora własnego przynależnego do wartości własnej 1 . Liczby c i s to kosinus i sinus kąta obrotu. Oś obrotu reprezentowanego przez macierz U możemy znaleźć (z dokładnością wystarczającą w grafice), wybierając dwa niezależne liniowo wiersze macierzy $U - I_3$ i obliczając ich iloczyn wektorowy.

W drugim przypadku wartościami własnymi (macierzy ortogonalnej 3×3) są liczby -1 , (c, s) i $(c, -s)$. Macierz taka reprezentuje złożenie obrotu z odbiciem symetrycznym względem płaszczyzny prostopadłej do osi obrotu. W obu przypadkach, jeśli $s = 0$, to mamy do czynienia z odbiciem symetrycznym względem punktu, prostej lub płaszczyzny.

Podsumowując: dowolne przekształcenie afiniczne jest złożeniem obrotu, skalowania osi układu xyz , drugiego obrotu i przesunięcia (to jest interpretacja związana z rozkładem SVD), albo złożeniem skalowania pewnych trzech osi prostopadłych, obrotu i przesunięcia (ta interpretacja jest związana z rozkładem biegunowym).

W pewnych zastosowaniach (np. w animacji i rejestrowaniu ruchu) mamy do czynienia z macierzami ortogonalnymi, które są znane niedokładnie, z powodu błędów zaokrągleń. Algorytm Highama jest najprostszym sposobem wyeliminowania tych błędów, tj. znalezienia macierzy ortogonalnej najbliższej macierzy danej.

4.3.8. Obroty, liczby zespolone i kwaterniony

Liczba zespolona jest parą liczb rzeczywistych: $z = (a, b)$; na zbiorze takich obiektów określa się dodawanie, tak jakby to były wektory w $\mathbb{R}^2$, i mnożenie, wzorem $(a_1, b_1)(a_2, b_2) = (a_1a_2 - b_1b_2, a_1b_2 + b_1a_2)$. Liczba zespolona $z = (c, s)$, taka że $c^2 + s^2 = 1$ nazywa się jednostkowa; pomnożenie dowolnej liczby zespolonej p przez z daje w wyniku liczbę, której obie części, rzeczywista i urojona, są równe współrzędnym obrazu wektora p w obrocie o kąt φ , takim że $c = \cos \varphi$, $s = \sin \varphi$.

Liczbie zespolonej $z = (a, b)$ przyporządkowujemy macierz

$$Z = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}.$$

Pierwsza jej kolumna to liczba z . Mnożenie i dodawanie liczb zespolonych to działania równoważne dodawaniu i mnożeniu takich macierzy, a w przypadku, gdy liczba z jest jednostkowa, macierz Z jest macierzą obrotu w $\mathbb{R}^2$.

Dość podobną reprezentację obrotów w przestrzeni trójwymiarowej stanowią **kwaterniony**. Są to wektory w $\mathbb{R}^4$, z odpowiednio określonymi działaniami. Dodawanie i odejmowanie kwaternionów jest dodawaniem i odejmowaniem wektorów w $\mathbb{R}^4$.

Aby określić mnożenie i dzielenie, zamiast macierzy kolumnowej $q = [a, x, y, z]^T$, będziemy używać notacji $q = (a, \mathbf{b})$, w której wyróżniamy **część skalarną** $a \in \mathbb{R}$ (pierwszą współrzędną) i **część wektorową** $\mathbf{b} = [x, y, z]^T \in \mathbb{R}^3$. Wzór opisujący mnożenie kwaternionów ma postać

$$(a_1, \mathbf{b}_1) \cdot (a_2, \mathbf{b}_2) = (a_1 a_2 - \langle \mathbf{b}_1, \mathbf{b}_2 \rangle, a_1 \mathbf{b}_2 + \mathbf{b}_1 a_2 + \mathbf{b}_1 \wedge \mathbf{b}_2).$$

Jak widać, wzór opisujący iloczyn kwaternionów bardzo przypomina wzór na iloczyn liczb zespolonych; najbardziej widoczna różnica to składnik $\mathbf{b}_1 \wedge \mathbf{b}_2$ w części wektorowej. Z powodu tego składnika mnożenie kwaternionów jest nieprzemienne, ale jest ono łączne i rozdzielne względem dodawania (ćwiczenie: sprawdź to).

Aby dokładniej zbadać własności kwaternionów, kwaternionowi $q = [a, x, y, z]^T = (a, \mathbf{b})$ przyporządkujemy macierz

$$Q = \begin{bmatrix} a & -x & -y & -z \\ x & a & -z & y \\ y & z & a & -x \\ z & -y & x & a \end{bmatrix} = \begin{bmatrix} a & -\mathbf{b}^T \\ \mathbf{b} & aI_3 + \mathbf{b} \wedge I_3 \end{bmatrix}.$$

Oczywiście, każdej macierzy utworzonej z czterech liczb zgodnie z tym schematem odpowiada pewien kwaternion. Wykonując odpowiednie rachunki, możemy sprawdzić, że sumie kwaternionów q_1 i q_2 odpowiada suma przyporządkowanych im macierzy, $Q_1 + Q_2$, a ponadto $q_1 \cdot q_2 = Q_1 Q_2$, skąd dalej wynika, że macierz $Q_1 Q_2$ odpowiada iloczynowi kwaternionów $q_1 \cdot q_2$.

Dalej przyda się kilka określeń: **kwaternion sprzężony** do $q = (a, \mathbf{b})$ to $\bar{q} = (a, -\mathbf{b})$; **wartość bezwzględna** kwaternionu $q = (a, \mathbf{b})$ jest liczbą rzeczywistą $|q| = \sqrt{a^2 + \langle \mathbf{b}, \mathbf{b} \rangle}$. Wartość bezwzględna kwaternionu jest więc pierwiastkiem z sumy kwadratów wszystkich czterech współrzędnych i jedyny kwaternion, którego wartość bezwzględna jest równa 0 to **zero**, czyli $(0, \mathbf{0})$.

Zobaczmy, jak to wygląda w notacji macierzowej. Jeśli kwaternion q jest związany z macierzą Q , to kwaternionowi sprzężonemu $\bar{q}$ odpowiada macierz transponowana Q^T . Iloczynowi $q \cdot \bar{q}$ odpowiada macierz $Q Q^T = |q|^2 I_4$. Macierz Q jest więc iloczynem pewnej macierzy ortogonalnej i liczby $|q|$. Zachodzi równość $\det Q = |q|^4$ (można pokazać, że wyznacznik macierzy Q jest nieujemny), a z niej (i z twierdzenia Cauchy'ego) wynika, że dla dowolnych kwaternionów q_1, q_2 zachodzi równość $|q_1 \cdot q_2| = |q_1| |q_2|$.

Kwaternion niemy ma część wektorową równą $\mathbf{0}$ (odpowiadająca mu macierz jest diagonalna). Zauważmy, że mnożenie kwaternionów niemych daje w wyniku kwaternion niemy, o części skalarnej równej iloczynowi części skalnych czynników; można więc utożsamić zbiór kwaternionów niemych ze zbiorem liczb rzeczywistych i dodawanie oraz mnożenie w obu zbiorach są wykonywane tak samo. Zauważmy jeszcze dwie rzeczy: jeśli dowolny argument mnożenia jest kwaternionem niemy, to kolejność tych argumentów można zmienić, a ponadto wzór opisujący wartość bezwzględną dowolnego kwaternionu możemy teraz zapisać w postaci $|q| = \sqrt{q \cdot \bar{q}}$.

Jedynka kwaternionowa to kwaternion $(1, \mathbf{0})$ (odpowiada jej macierz jednostkowa I_4). Jest ona elementem neutralnym mnożenia (i jest to jedyny taki element). Dla skrótu kwaternion zerowy i jedynkę można zapisywać symbolami 0 i 1, pamiętając, że to kwaterniony.

Kwaternion odwrotny do q to q^{-1} , taki że $q \cdot q^{-1} = q^{-1} \cdot q = (1, \mathbf{0})$. Odwrotność jest jednoznaczna i wyraża się wzorem $q^{-1} = \bar{q}/|q|^2$ (przypomina on wzór na odwrotność liczby

zespolonej). Każdy kwaternion różny od zera ma odwrotność. W notacji macierzowej kwaternionowi q^{-1} odpowiada macierz $Q^{-1} = \frac{1}{|q|^2} Q^T$.

Mając pojęcie odwrotności, można określić **dzielenie** kwaternionów, a właściwie dwa dzielenia: $q_1/q_2 = q_1 \cdot q_2^{-1}$ i $q_2 \backslash q_1 = q_2^{-1} \cdot q_1$. **Uwaga:** przypominam, że na ogół $q_1 \cdot q_2^{-1} \neq q_2^{-1} \cdot q_1$ (równość zachodzi wtedy, gdy części wektorowe obu kwaternionów są liniowo zależne). Dlatego nie będziemy pisać kwaternionowych wyrażeń z kreską ułamkową, chyba że mianownik jest liczbą rzeczywistą (albo kwaternionem niemym).

Ostatnie dwa pojęcia, których będziemy potrzebować, to **kwaternion czysty**, którego część skalarna jest równa 0 i **kwaternion jednostkowy** (nie mylić z jedynką), którego wartość bezwzględna jest równa 1. Ponieważ wartość bezwzględna iloczynu kwaternionów jest iloczynem ich wartości bezwzględnych, zbiór kwaternionów jednostkowych jest zamknięty ze względu na mnożenie. Co więcej, odwrotnością kwaternionu jednostkowego jest jego kwaternion sprzężony. Kwaternionom jednostkowym odpowiadają macierze ortogonalne 4×4 .

Z punktu widzenia algebry zbiór kwaternionów z opisanymi wyżej działaniami jest **ciałem nieprzemienne**. Tradycyjnie oznacza się je symbolem $\mathbb{H}$, dla uczczenia Williama R. Hamiltona, który 16 października 1843 r. odkrył je w Dublinie¹.

Obroty w przestrzeni $\mathbb{R}^3$ są reprezentowane przez kwaterniony jednostkowe². Weźmy dowolny wektor jednostkowy $\mathbf{v} \in \mathbb{R}^3$ i liczbę φ . Obrotowi o kąt φ wokół prostej o kierunku $\mathbf{v}$ przyporządkujemy kwaternion $q = (\cos \frac{\varphi}{2}, \mathbf{v} \sin \frac{\varphi}{2})$. Jest on oczywiście jednostkowy. Wektorowi $\mathbf{p} \in \mathbb{R}^3$ przyporządkujemy kwaternion czysty $p = (0, \mathbf{p})$. Udowodnimy, że

$$q \cdot p \cdot q^{-1} = p',$$

gdzie p' jest kwaternionem czystym, $p' = (0, \mathbf{p}')$, takim że wektor $\mathbf{p}'$ jest obrazem wektora $\mathbf{p}$ w rozpatrywanym obrocie.

Oznaczmy $s = \sin \frac{\varphi}{2}$ i $c = \cos \frac{\varphi}{2}$. Liczymy

$$\begin{aligned} q \cdot p \cdot q^{-1} &= (c, s\mathbf{v}) \cdot (0, \mathbf{p}) \cdot (c, -s\mathbf{v}) = (-s\langle \mathbf{v}, \mathbf{p} \rangle, c\mathbf{p} + s\mathbf{v} \wedge \mathbf{p}) \cdot (c, -s\mathbf{v}) = \\ &= (-cs\langle \mathbf{v}, \mathbf{p} \rangle + cs\langle \mathbf{p}, \mathbf{v} \rangle + s\langle \mathbf{v} \wedge \mathbf{p}, \mathbf{v} \rangle, s^2\langle \mathbf{v}, \mathbf{p} \rangle \mathbf{v} + c^2\mathbf{p} + cs\mathbf{v} \wedge \mathbf{p} - cs\mathbf{p} \wedge \mathbf{v} - s^2(\mathbf{v} \wedge \mathbf{p}) \wedge \mathbf{v}) \\ &= (0, \mathbf{p}'). \end{aligned}$$

Część skalarna iloczynu jest zgodnie z zapowiedzią równa 0, część wektorową, $\mathbf{p}'$, trzeba jeszcze obliczyć. Zauważmy, że $s^2 = 1 - c^2$; stąd mamy

$$s^2\langle \mathbf{v}, \mathbf{p} \rangle \mathbf{v} + c^2\mathbf{p} = \langle \mathbf{v}, \mathbf{p} \rangle \mathbf{v} + c^2(\mathbf{p} - \langle \mathbf{v}, \mathbf{p} \rangle \mathbf{v}).$$

Ponadto $\mathbf{v} \wedge \mathbf{p} = \mathbf{b}$ (rysunek 4.9), oraz $\mathbf{b} \wedge \mathbf{v} = \mathbf{a} = \mathbf{p} - \langle \mathbf{v}, \mathbf{p} \rangle \mathbf{v}$. Na podstawie wzorów $2cs = \sin \varphi$, $c^2 - s^2 = \cos \varphi$ otrzymujemy

$$\mathbf{p}' = \mathbf{v}\langle \mathbf{v}, \mathbf{p} \rangle + (\mathbf{p} - \langle \mathbf{v}, \mathbf{p} \rangle \mathbf{v}) \cos \varphi + \mathbf{v} \wedge \mathbf{p} \sin \varphi,$$

czyli wcześniej wyprowadzony wzór opisujący obraz wektora $\mathbf{p}$ w zadanym obrocie, co kończy dowód. $\square$

¹ Hamilton wymyślił wtedy sposób mnożenia czwórek liczb rzeczywistych, który spełnia wszystkie z wyjątkiem przemienności warunki potrzebne do otrzymania ciała (ze „zwykłym” działaniem dodawania). Kwintesencją tego mnożenia jest wzór

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{i} \cdot \mathbf{j} \cdot \mathbf{k} = -1,$$

w którym są użyte symbole $\mathbf{i} = [0, 1, 0, 0]^T$, $\mathbf{j} = [0, 0, 1, 0]^T$, $\mathbf{k} = [0, 0, 0, 1]^T$ i $-1 = [-1, 0, 0, 0]^T$.

² Dla porządku odnotujmy, że dowolny obrót w $\mathbb{R}^3$ może być reprezentowany za pomocą dwóch kwaternionów jednostkowych, choć w grafice to ma niewielkie znaczenie.

Zauważmy, że reprezentacja kwaternionowa obrotu nie jest jednoznaczna: kwaternion $-q$ reprezentuje ten sam obrót co q . Mamy bowiem

$$-q = \left(-\cos \frac{\varphi}{2}, -\mathbf{v} \sin \frac{\varphi}{2}\right) = \left(\cos \frac{2\pi - \varphi}{2}, -\mathbf{v} \sin \frac{2\pi - \varphi}{2}\right),$$

czyli reprezentację obrotu o kąt $2\pi - \varphi$ w drugą stronę, tj. wokół osi zorientowanej odwrotnie. Ponadto, jedynka kwaternionowa reprezentuje przekształcenie tożsamościowe, czyli obrót o kąt 0 wokół osi, której kierunek nie jest (i nie musi być) określony.

Bezpośrednie obliczanie iloczynu trzech kwaternionów nie jest zbyt tanie; trzeba wykonać przy tym 28 mnożeń liczb rzeczywistych, podczas gdy licząc na podstawie wzoru, do którego rzecz doprowadziliśmy, wykonamy tylko 16 mnożeń. Dysponując kwaternionami, mamy jednak możliwość stosunkowo łatwego dokonania interpolacji położenia kątownych bryły w ruchu kulistym, zadanych w wybranych chwilach. W tym celu trzeba skonstruować krzywą położoną na sferze jednostkowej w $\mathbb{R}^4$. Krzywa ta przechodzi przez podane punkty (odpowiadające kolejno zadany położeniom kątownym bryły w ruchu kulistym) i określa jednoznacznie położenia bryły w innych chwilach. Konstruowanie skomplikowanych krzywych, których punkty są kwaternionami jednostkowymi, zostawimy na później, a tymczasem skonstruujemy najkrótszą krzywą o zadanych końcach.

Przypuśćmy, że dwa kwaterniony, q_0 i q_1 , reprezentują pewne obroty, które wyznaczają położenia kątowe dowolnego obiektu w chwilach 0 i 1. Chcielibyśmy interpolować te obroty, tj. dla dowolnego $t \in [0, 1]$ znaleźć obrót (czyli odpowiedni kwaternion jednostkowy q_t), który wyznacza położenie „pośrednie” obiektu.

Jedno z możliwych podejść polega na użyciu operacji **potęgowania kwaternionów**. Możemy zauważyć, że dla dowolnej liczby całkowitej n i kwaternionu $q = |q|(\cos \frac{\varphi}{2}, \mathbf{v} \sin \frac{\varphi}{2})$ zachodzi równość

$$q^n = |q|^n \left(\cos \frac{n\varphi}{2}, \mathbf{v} \sin \frac{n\varphi}{2}\right).$$

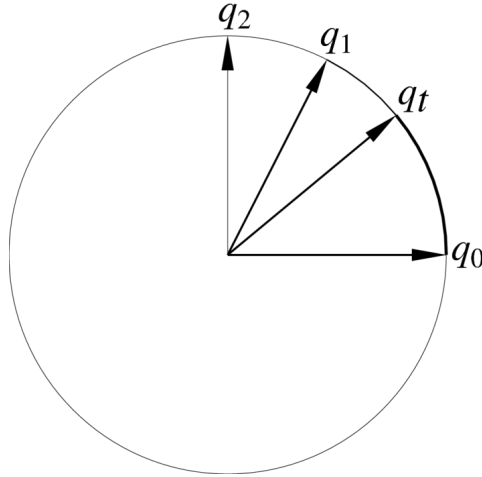
Możemy rozszerzyć potęgowanie tak, aby dopuścić dowolny wykładnik rzeczywisty t ; dla kwaternionu jednostkowego $q = (\cos \frac{\varphi}{2}, \mathbf{v} \sin \frac{\varphi}{2})$ mamy zatem

$$q^t = \left(\cos \frac{t\varphi}{2}, \mathbf{v} \sin \frac{t\varphi}{2}\right).$$

Obrót odpowiadający chwili t moglibyśmy określić przy użyciu jednego z kwaternionów określonych wzorami $q_t' = q_0^{1-t} \cdot q_1^t$ albo $q_t'' = q_1^t \cdot q_0^{1-t}$, do których należałoby podstawić odpowiednie potęgi kwaternionów q_0 i q_1 określone wyżej. W obu przypadkach dla $t = 0$ otrzymamy kwaternion q_0 , a dla $t = 1$ kwaternion q_1 . Dla $t \notin \{0, 1\}$ brak przemienności mnożenia kwaternionów daje różne wyniki, dlatego oba podane tu wzory *nie są poprawne*. Poprawny wzór wyprowadzimy za chwilę.

Poprawny sposób polega na dokonaniu **interpolacji łukowej kwaternionów**. Kwaterniony jednostkowe q_0 i q_1 , takie że $q_0 \neq q_1$ i $q_0 \neq -q_1$ jednoznacznie określają najkrótszy łuk na sferze jednostkowej, którego te kwaterniony są końcami. Dla chwili $t \in [0, 1]$ przyjmujemy obrót reprezentowany przez kwaternion q_t , który dzieli ten łuk w proporcji $t : 1 - t$. Na rysunku 4.10 jest pokazany przekrój przez sferę jednostkową w $\mathbb{R}^4$ płaszczyzną zawierającą kwaterniony q_0 i q_1 ; przekrój ten jest oczywiście okręgiem jednostkowym. Kąt ψ między kwaternionami q_0 i q_1 możemy znaleźć, traktując te kwaterniony jak wektory w $\mathbb{R}^4$ i obliczając ich iloczyn skalarny; jest on równy $\cos \psi$. Kąt między kwaternionami q_0 i q_t jest równy $t\psi$. Niech q_2 oznacza leżący na rozważanym okręgu kwaternion prostopadły do q_0 . Wtedy mamy

$$\begin{aligned} q_1 &= q_0 \cos \psi + q_2 \sin \psi, \\ q_t &= q_0 \cos t\psi + q_2 \sin t\psi. \end{aligned}$$



Rysunek 4.10. Interpolacja łukowa kwaternionów jednostkowych.

Wyznaczając q_2 na podstawie pierwszego równania i wstawiając do drugiego, po uporządkowaniu otrzymamy wzór

$$q_t = \frac{q_0 \sin(1-t)\psi + q_1 \sin t\psi}{\sin \psi}.$$

Stosowanie tego wzoru wymaga użycia funkcji trygonometrycznych, ale w szczególnym przypadku mamy

$$q_{1/2} = \frac{q_0 + q_1}{|q_0 + q_1|},$$

dzięki czemu łuk łączący kwaterniony q_0 i q_1 możemy dzielić rekurencyjnie na połowy, ćwiartki itd. za pomocą dodawań, mnożeń i pierwiastka kwadratowego.

Chcąc interpolować położenia kątowe obiektu, reprezentowane przez kwaterniony q_0 i q_1 , możemy dzielić w odpowiednich proporcjach łuk o końcach q_0 i q_1 lub łuk o końcach q_0 i $-q_1$. Zwykle wybieramy łuk krótszy, tj. jeśli kosinus kąta ψ między q_0 i q_1 jest ujemny, to wybieramy drugi z tych dwóch łuków (ruch określony przez dłuższy łuk jest obracaniem w drugą stronę, trzeba wtedy obrócić obiekt o kąt większy niż π).

Interpretację interpolacji łukowej, a także poprawny zapis tej interpolacji przy użyciu potęgowania, otrzymamy, rozpatrując przyporządkowane kwaternionom macierze. Jak wiemy, macierz Q , odpowiadająca dowolnemu kwaternionowi jednostkowemu q , jest ortogonalna. Macierz ta reprezentuje pewien obrót przestrzeni czterowymiarowej. Łuk, którego końcami są kwaterniony q_0 i q_1 , obrócimy tak, aby przekształcić q_0 na jedynekę kwaternionową. Obrazem punktu q_t na tym łuku jest kwaternion $\tilde{q}_t = Q_0^{-1}q_t$, a w szczególności koniec q_1 przejdzie na $\tilde{q}_1 = Q_0^{-1}q_1 = q_0^{-1}q_1$. Zauważmy, że $\tilde{q}_t = \tilde{q}_1^t$. Stąd otrzymujemy

$$q_t = Q_0 \tilde{q}_t = q_0 \cdot (q_0^{-1} \cdot q_1)^t.$$

Stosując interpolację łukową, np. w animacji, otrzymamy ciąg położenia kątowych przedmiotu, który obraca się (ze stałą prędkością, jeśli parametr t zmieniamy ze stałą prędkością) wokół ustalonej osi w przestrzeni.

Interpolacja łukowa może być wykorzystana do skonstruowania gładkiej krzywej interpolacyjnej położonej na sferze jednostkowej w $\mathbb{R}^4$ (czyli złożonej z kwaternionów jednostkowych), przechodzącej przez punkty dane na sferze. Konstrukcja tych krzywych przypomina określenie

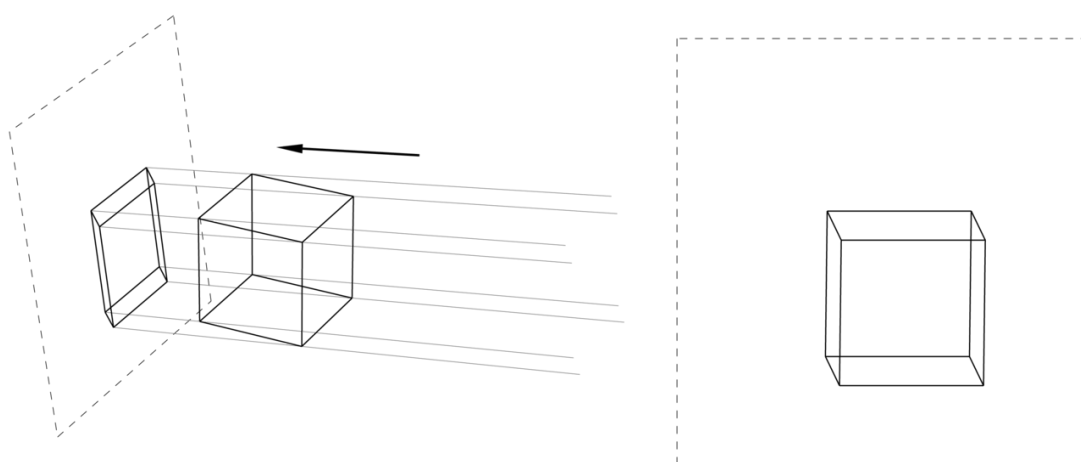
krzywych Béziera, o których będzie mowa dalej — interpolacja afiniczna używana w algorytmie de Casteljau zostaje zastąpiona przez interpolację łukową.

Reprezentacja obrotów przy użyciu kwaternionów ma pewną cechę, która może być wadą albo zaletą, zależnie od sytuacji. Otóż ruch, którego opis konstruujemy w obróconym układzie współrzędnych, jest identyczny jak w układzie wyjściowym. Metoda ta nie bierze pod uwagę czegoś takiego jak kierunek „pionowy”, przez co otrzymany ruch może być nienaturalny w danym zastosowaniu.

5. Rzutowanie równoległe, perspektywiczne i inne

5.1. Rzutowanie równoległe

Obrazem dowolnego punktu p w rzucie równoległym jest taki punkt p' położony na rzutni, że kierunek wektora $p - p'$ (o ile nie jest to wektor zerowy) jest kierunkiem rzutowania.



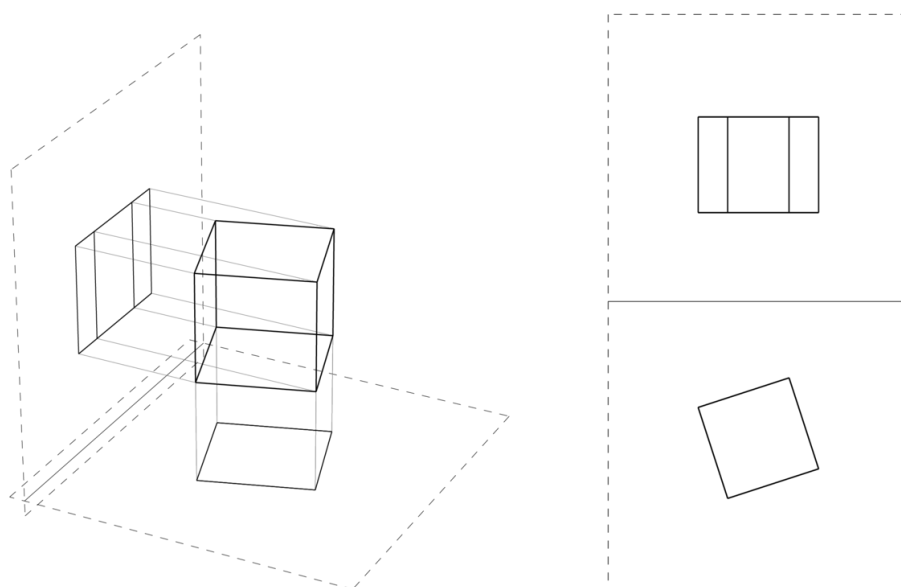
Rysunek 5.1. Rzutowanie równoległe.

Takie rzutowanie równoległe jest powszechnie używane w rysunku technicznym, gdzie jednym z celów jest umożliwienie odtworzenia (restytucji) punktu w przestrzeni na podstawie rysunku. Inna potrzeba, do pewnego czasu dominująca w projektowaniu przemysłowym, to dokonywanie konstrukcji geometrycznych w przestrzeni trójwymiarowej na płaskich rysunkach. Na podstawie tych potrzeb rozwinęła się **geometria wykreślna**, która obecnie wydaje się być sztuką w zaniku — użycie komputerów zwalnia od wysiłków i coraz częściej przedmiot ten jest wykładany już tylko po to, aby rozwijać wyobraźnię przestrzenną u studentów architektury (tego, niestety, czy na szczęście, komputer nie zastąpi). Istnieją dwa podstawowe podejścia do stosowania rzutowania równoległego w rysunku technicznym: **metoda Monge’a**¹, czyli wykonywanie konstrukcji trójwymiarowych na dwóch obrazach będących rzutami prostopadłymi na dwie, wzajemnie prostopadłe rzutnie, oraz **aksonometria**.

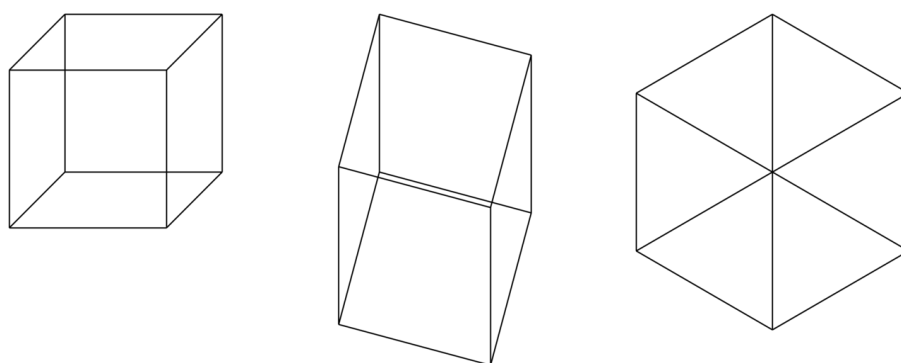
Podstawowe znaczenie dla aksonometrii ma **twierdzenie Pohlkego**: *dobierając odpowiednio rzutnię i kierunek rzutowania można odwzorować wierzchołki danego czworościanu na figurę podobną do zbioru dowolnych czterech punktów, z których żadne trzy nie leżą na jednej prostej*.

Na podstawie tego twierdzenia możemy określić rzut aksonometryczny, wybierając obraz początku układu współrzędnych i trzy wektory (z których żadne dwa nie są liniowo zależne) — obrazy wersorów osi. W ten sposób postąpiliśmy w celu narysowania wykresu funkcji przy użyciu algorytmu z pływającym horyzontem, określając rzut aksonometryczny. Rzuty stosowane najczęściej w rysunku technicznym są pokazane rysunku 5.3

¹ Gaspard Monge, 1746–1818



Rysunek 5.2. Rzuty Monge'a.



Rysunek 5.3. Obrazy sześciangu w aksonometrii kawalerskiej, wojskowej i izometrycznej.

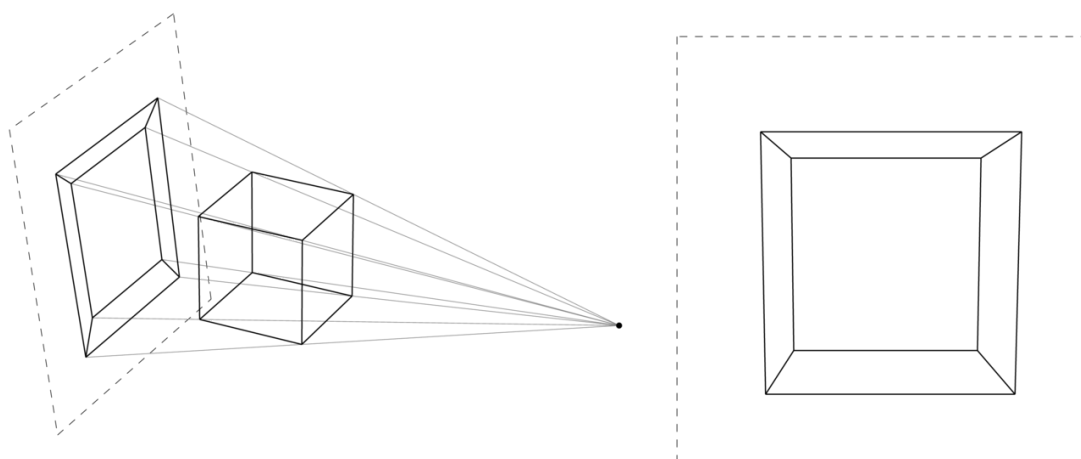
5.2. Rzutowanie perspektywiczne

Obrazem punktu p w rzucie perspektywicznym jest punkt p' , który jest przecięciem rzutni i prostej przechodzącej przez p i **środek rzutowania** (czyli punkt położenia obserwatora).

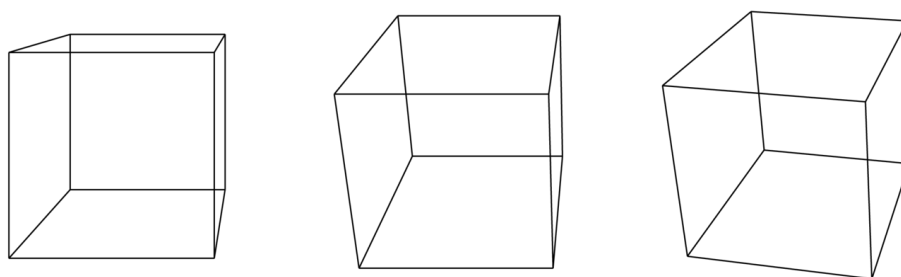
Rzut równoległy można interpretować jako graniczny przypadek rzutu perspektywicznego, gdy obserwator znajduje się w bardzo dużej odległości.

W rysunkach technicznych części maszyn rzut perspektywiczny bywa używany bardzo rzadko, głównie w rysunkach poglądowych, takich jak schematy montażowe, natomiast często stosuje się go w architekturze. Opracowanie konstrukcji geometrycznych związanych z tym sposobem rzutowania wiąże się z historią malarstwa, którą tu się nie zajmujemy.

Zależnie od położenia rzutni względem głównego obiektu na obrazie, mówimy o perspektywie **jednopunktowej**, **dwupunktowej** i **trypunktowej**. Z obiektem związane są trzy kierunki osi, jakoś powiązane z obiektem (najbardziej oczywisty związek jest wtedy, gdy obiekt jest prostopadłościanem). Poszczególne przypadki otrzymamy wybierając rzutnię tak, aby jeden, dwa lub trzy spośród tych kierunków nie były równoległe do rzutni.



Rysunek 5.4. Rzutowanie perspektywiczne.



Rysunek 5.5. Perspektywa jedno-, dwu- i trzypunktowa.

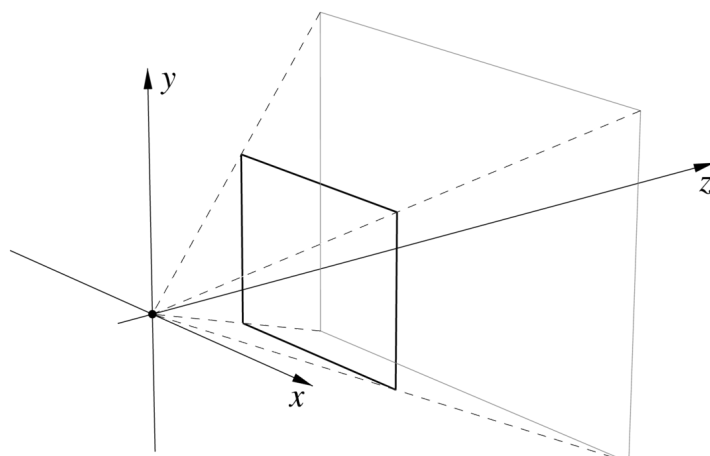
5.3. Algorytm rzutowania

Przekształcenie rzutowe jest to przekształcenie przestrzeni rzutowej (tj. afinicznej uzupełnionej o punkty niewłaściwe, czyli kierunki), któremu odpowiada różnowartościowe (czyli o nieosobliwej macierzy) przekształcenie liniowe przestrzeni jednorodnej. Macierz reprezentująca przekształcenie afiniczne we współrzędnych jednorodnych ma ostatni wiersz o postaci $[0, 0, 0, 1]$ (lub ogólniej $[0, 0, 0, a]$ dla dowolnego $a \neq 0$). Przekształcenia rzutowe otrzymamy dopuszczając dowolne liczby w ostatnim wierszu (pod warunkiem zachowania nieosobliwości macierzy; wymaganie to bierze się stąd, że punkt $[0, 0, 0, 0]^T$ przestrzeni współrzędnych jednorodnych, który jest obrazem pewnych niezerowych wektorów jeśli macierz jest osobliwa, nie reprezentuje żadnego punktu przestrzeni rzutowej).

Dowolne przekształcenia rzutowe są nieco rzadziej niż afiniczne stosowane w modelowaniu obiektów, ale przydają się do określenia rzutowania perspektywicznego, a zatem ich implementacja jest zawsze potrzebna w grafice „trójwymiarowej” i często realizowana w spręście.

Rzutowanie, tj. odwzorowanie przestrzeni trójwymiarowej na płaszczyznę ekranu, zwykle określa się opisując za pomocą odpowiednich parametrów tzw. **wirtualną kamerę**, przy czym w różnych pakietach graficznych szczegóły tego postępowania mogą być różne. Wirtualna kamera związana jest z kartezjańskim **układem współrzędnych kamery** i pierwszy krok rzutowania punktu polega na obliczeniu jego współrzędnych w tym układzie.

Przypuśćmy, że początek układu współrzędnych kamery jest położeniem obserwatora (albo środkiem obiektywu) i że oś z tego układu jest osią optyczną obiektywu. Rzutnia jest płaszczyzną prostopadłą do tej osi, tj. jest równoległa do płaszczyzny xy układu kamery i leży w odległości f od tej płaszczyzny. Wyróżnimy **klatkę**, czyli prostokąt leżący w rzutni, w którym obraz



Rysunek 5.6. Wirtualna kamera i bryła (ostrośłup) widzenia.

będzie np. wyświetlony na ekranie (boki klatki są równoległe do osi x i y). Kierunek rzutowania lub położenie obserwatora i klatka określają tzw. **bryłę widzenia**, będącą zbiorem punktów, których rzuty należą do klatki.

Rzutowanie równoległe polega na zignorowaniu współrzędnej z (oś z wyznacza kierunek rzutowania). Obraz punktu $[x, y, z]^T$ w rzucie perspektywicznym ma współrzędne fx/z , fy/z i f , przy czym tę ostatnią współrzędną zignorujemy. Zauważmy, że współrzędne x , y , z w układzie kamery są (z dokładnością do stałej f) współrzędnymi jednorodnymi obrazu rozpatrywanego punktu na klatce, przy czym współrzędna z jest współrzędną wagową. Przypuśćmy, że zamiast współrzędnych kartezjańskich x , y , z , do reprezentowania punktu używamy współrzędnych jednorodnych X , Y , Z , W , takich że $x = \frac{X}{W}$, $y = \frac{Y}{W}$, $z = \frac{Z}{W}$. Wtedy wektor $[X, Y, Z, W]^T$ możemy poddać przekształceniu, które w przypadku rzutowania równoległego i perspektywicznego polega na pomnożeniu go odpowiednio przez macierz

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{albo} \quad \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Wynikiem mnożenia jest wektor $[X, Y, W]^T$ albo $[fX, fY, Z]^T$, a zatem w każdym z tych przypadków otrzymujemy współrzędne jednorodne odpowiedniego obrazu rzutowanego punktu na płaszczyźnie rzutni.

Po obliczeniu współrzędnych jednorodnych wykonuje się dwa dzielenia, a następnie dokonuje jeszcze jednego przekształcenia afinicznego, którego celem jest przejście do współrzędnych urządzenia (jednostką w tym układzie jest szerokość lub wysokość piksela). Współrzędne w układzie urządzenia można następnie zaokrąglić i przystąpić do rysowania.

Zatrzymajmy się jeszcze na kroku poprzednim. Współrzędna z w układzie kamery jest **głębookością** punktu i jest potrzebna do rozstrzygania widoczności w algorytmach linii i powierzchni zasłoniętej. Z dwóch punktów, których obrazem w rzucie jest ten sam punkt, i które znajdują się *przed obserwatorem*, widoczny jest punkt o mniejszej głębokości. Dokonując rzutowania będziemy chcieli otrzymać informację o głębokości, przy czym ze względów oszczędnościowych (aby dostać tę informację w jednej operacji mnożenia macierzy 4×4 i 4×1) chcemy, aby to była współrzędna jednorodna, taka że głębokość (lub informacja jej równoważna) jest ilorzem tej współrzędnej i współrzędnej wagowej.

Dokonując rzutowania równoległego możemy użyć macierzy jednostkowej; wtedy otrzymamy wynik $[X, Y, Z, W]^T$ i obliczymy współrzędne obrazu punktu na rzutni $x = X/W$, $y = Y/W$ i głębokość $z = Z/W$. Zauważmy, że rozstrzyganie widoczności jest też możliwe jeśli zamiast współrzędnej z w układzie kamery znamy dla każdego punktu odpowiadającą mu liczbę $c_0 - c_1 W/Z$ (dla dowolnej stałej c_0 oraz $c_1 \neq 0$). Dzięki temu podczas rzutowania perspektywicznego możemy wektor współrzędnych jednorodnych danego punktu w układzie kamery pomnożyć przez macierz

$$\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & c_0 & -c_1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Macierz ta jest nieosobliwa, a iloczynem jej i wektora $[X, Y, Z, W]^T$ jest wektor $[fX, fY, c_0Z - c_1W, Z]^T$. Możemy dalej obliczyć współrzędne obrazu punktu na rzutni $x = fX/Z$, $y = fY/Z$ i głębokość $d = c_0 - c_1W/Z$. Jeśli $c_1 > 0$, to z dwóch punktów, które mają ten sam obraz w rzucie, bliższy obserwatora jest punkt, któremu odpowiada mniejsza liczba d .

Określając rzutowanie w programach korzystających z różnych pakietów oprogramowania, należy podać zakres głębokości, tj. granice przedziału $[z_{\min}, z_{\max}]$, do którego należą współrzędne punktów rzutowanej sceny. Jest to potrzebne dlatego, że informacja o głębokości jest (po przeskalowaniu) zaokrąglana do liczby całkowitej (np. 24- lub 32-bitowej) i w testach widoczności dane są przetwarzane w tej postaci. Przypuśćmy, że punktom o głębokościach z przedziału $[z_{\min}, z_{\max}]$ mają odpowiadać liczby d z przedziału $[0, 1]$. Mamy stąd, dla rzutu perspektywicznego, układ dwóch równań liniowych

$$\begin{cases} z_{\min}c_0 - c_1 = 0, \\ z_{\max}c_0 - c_1 = 1, \end{cases}$$

na podstawie którego możemy obliczyć c_0 i c_1 (oczywiście, musi być spełniony warunek $z_{\max} > z_{\min} > 0$). Możemy wprowadzić analogiczne współczynniki do macierzy rzutowania równoległego i określić je w podobny sposób.

Wiemy już dostatecznie dużo, aby rozszyfrować sposób tworzenia macierzy rzutowania wykorzystywany przez standard OpenGL. W standardzie tym bryła widzenia jest poddawana takiemu przekształceniu, aby jej obraz był kostką $[-1, 1] \times [-1, 1] \times [0, 1]$. Ma to na celu umożliwienie wygodnego obcinania rysowanych obiektów (odcinków i wielokątów) do takiej kostki, a następnie zastosowania algorytmu z buforem głębokości do rozstrzygania widoczności (współrzędne z punktów w bryle widzenia, należące do przedziału $[0, 1]$, są dalej mnożone przez liczbę N zależną od implementacji OpenGL'a, może być np. $N = 2^{16} - 1$ lub $N = 2^{32} - 1$, i zaokrąglane do liczby całkowitej — to jest już ukryte przed autorem programu korzystającego z biblioteki OpenGL). Dlatego macierz rzutowania perspektywicznego, tworzona przez procedurę `glFrustum`, ma postać

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}.$$

Symbole r, l, t, b, n i f są współrzędnymi punktów określających bryłę widzenia: liczby dodatnie n i f (ang. *near* i *far*) określają przednią i tylną płaszczyznę obcinania; współrzędne z (w układzie kamery) punktów bryły widzenia leżą w przedziale między nimi. Ściana bryły widzenia równoległa do rzutni położona bliżej środka rzutowania (w płaszczyźnie $z = n$) jest prostokątem,

którego wierzchołkami są punkty $[l, t, n]^T$, $[r, t, n]^T$, $[r, b, n]^T$ i $[l, b, n]^T$. Wyświetlane punkty, po przejściu przez test widoczności i obliczeniu koloru, są następnie poddawane przekształceniu, które kwadrat $[-1, 1] \times [-1, 1]$ odwzorowuje na odpowiedni prostokąt (np. okno) na ekranie. Szczegóły są opisane w dodatku B.

Macierz tworzona przez procedurę `glOrtho` realizuje rzutowanie równoległe; bryła widzenia w układzie kamery jest prostopadłościanem, który zostaje odwzorowany na kostkę $[-1, 1] \times [-1, 1] \times [0, 1]$. Macierz ta ma postać

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Obcinanie, rozstrzyganie widoczności i dalsze etapy rysowania przebiegają tak samo jak w przypadku rzutowania perspektywicznego.

5.4. Stereoskopia

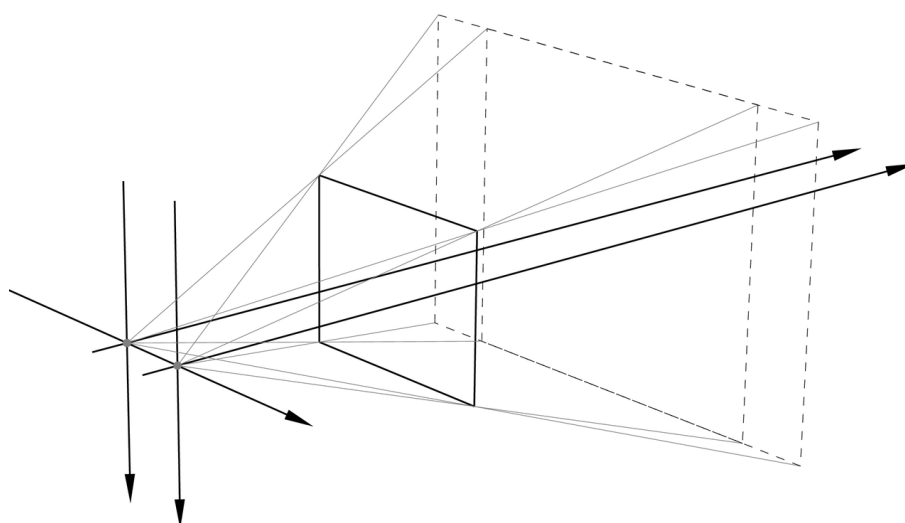
Wrażenie widzenia przestrzennego, czyli postrzeganie głębi obrazu, powstaje w mózgu obserwatora na podstawie dwóch obrazów na siatkówkach dwojga oczu. W świecie zwierząt jest to dość rzadkie zjawisko; zwróćmy uwagę na własności ludzkiego narządu wzroku, które to umożliwiają:

- Pola widzenia obojga oczu prawie w całości pokrywają się.
- Rozstaw oczu jest ustalony, sporo większy od ich wielkości i porównywalny z wielkością oglądanych obiektów (jeśli oglądamy duże obiekty, np. budynki, to wrażenie trójwymiarowości powstaje na podstawie poprzednich doświadczeń i oglądania obiektu z różnych stron; odległości położenia obserwatora są wtedy porównywalne z wielkością budynku).
- Ludzkie oko ma bardzo dużą głębię ostrości (tj. szeroki zakres odległości, w których jednocześnie jest w stanie „widzieć ostro”).

Dzięki powyższemu zbiegowi okoliczności wystarczy zapewnić, aby każde oko widziało odpowiedni obraz, którego odległość od oka jest raczej nieistotna. Techniki oglądania „obrazów przestrzennych” są następujące:

- Użycie kasku z wbudowanymi miniaturowymi monitorami,
- Zastosowanie okularów z ciekłymi kryształami; prawa i lewa szyba stają się na przemian nieprzezroczyste, a na ekranie są wtedy wyświetlane obrazy lewy i prawy.
Ta technika umożliwia oglądanie obrazów „trójwymiarowych” nie tylko na ekranie monitora lub telewizora, ale także na dużych ekranach, które są ścianami pomieszczenia. W takim pomieszczeniu może przebywać nawet duża grupa osób (z których każda ma swoje okulary), które odbierają poprawnie wrażenia widzenia przestrzennego. Potwierdza to fakt, że stosunkowo łatwo jest wywołać takie wrażenia u ludzi, którzy dobrze tolerują zniekształcenia wywołane wyświetlaniem obrazów otrzymanych z innego niż rzeczywiste położenia obserwatora.
- Można użyć kolorowych okularów, np. z czerwoną i zieloną szybą i wyświetlać każdy obraz w innym kolorze (takie rysunki są znane od dawna pod nazwą **anaglify**).
- Istnieją też obrazy „magiczne oko”, złożone z pozornie chaotycznych plam, wśród których można dostrzec obiekty trójwymiarowe.

Aby otrzymać odpowiednie dwa obrazy należy określić dwa rzuty perspektywiczne (po jednym dla każdego oka). Aby to zrobić poprawnie, trzeba ustalić odpowiedni układ współrzędnych



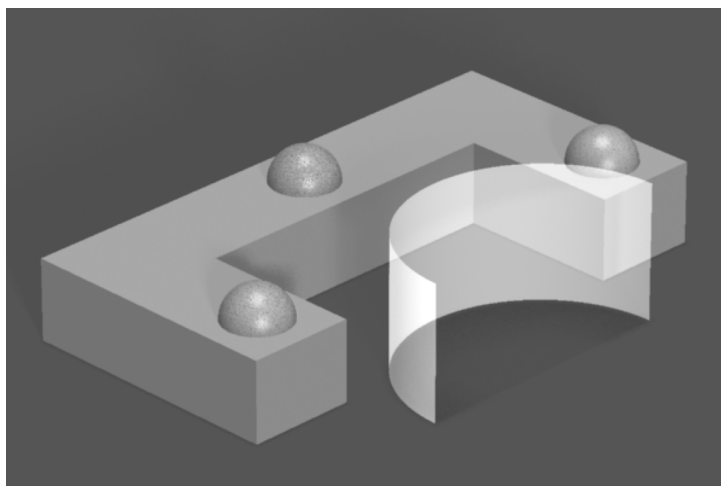
Rysunek 5.7. Wirtualne kamery dla obrazów stereoskopowych.

— prostokątny, o wersorach osi o identycznej długości, która jest jednostką odległości potrzebną dalej. Przypuśćmy, że jednostką tą jest 1 cal (1"). Przy użyciu wybranej jednostki należy zmierzyć wymiary ekranu (np. od 11.2" × 8.4" do 16.8" × 12.6"), a także odległość widza od płaszczyzny ekranu (np. od 25" do 40", siedzenie bliżej monitora jest niezdrowe) i odległość źrenic jego oczu (zwykle 2.64"). Na podstawie tych wymiarów wybieramy środki rzutowania i klatkę (rys. 5.7). Dla uzyskania dobrego efektu rysowane obiekty powinny mieć wymiary rzędu wielkości obrazu i znajdować się w odległości od obserwatora mniej więcej takiej, jak ekran (np. od połowy do dwóch odległości ekranu od obserwatora). Z moich doświadczeń wynika, że wspomniane wymiary można mierzyć niezbyt dokładnie, np. obrazy anaglifowe utworzone w celu wyświetlenia na monitorze czternastocalowym dają się bez problemu oglądać na monitorze siedemnastocalowym.

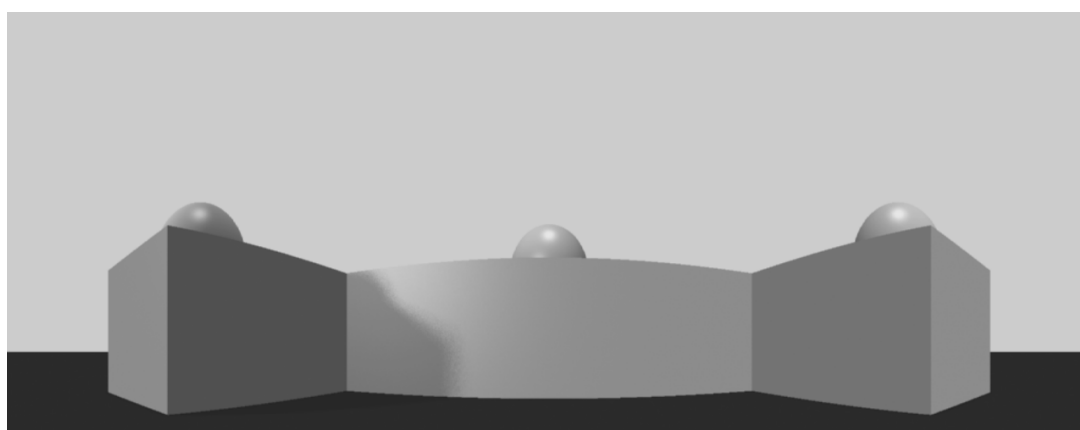
5.5. Panorama i inne rzuty

Istnieje wiele różnych stosowanych w praktyce odwzorowań przestrzeni w płaszczyznę; mogą one być określone przy użyciu rzutowania na powierzchnie krzywoliniowe, które następnie są przekształcane w płaszczyznę. Warto zauważyć, że obrazem odcinka w takim rzucie może być łuk krzywej, co stanowi utrudnienie podczas wykonywania obrazów.

Teoretycznie najprostsze jest przekształcenie w płaszczyznę powierzchni rozwijalnych, ale rzeczywiście łatwe do zrealizowania jest rozwinięcie tylko powierzchni walcowej. Możemy przyjąć jako rzutnię powierzchnię boczną walca (lub jej fragment), wybrać środek rzutowania w dowolnym punkcie jego osi, a następnie odwzorować rzutnię w płaszczyznę. Takie odwzorowanie nazywa się **panoramą** i bywa najczęściej stosowane w rysunkach architektonicznych i w wizualizacji danych. Rzadziej są stosowane rzuty na powierzchnie nierozwijalne, np. **olorama**, czyli rzut na sferę. Odwzorowanie obrazu na sferze (lub innej powierzchni nierozwijalnej) w płaszczyznę wprowadza dodatkowe zniekształcenia, które utrudniają jego oglądanie.



Rysunek 5.8. Scena i rzutnia walcowa.



Rysunek 5.9. Panorama.

6. Elementy modelowania geometrycznego

6.1. Krzywe Béziera

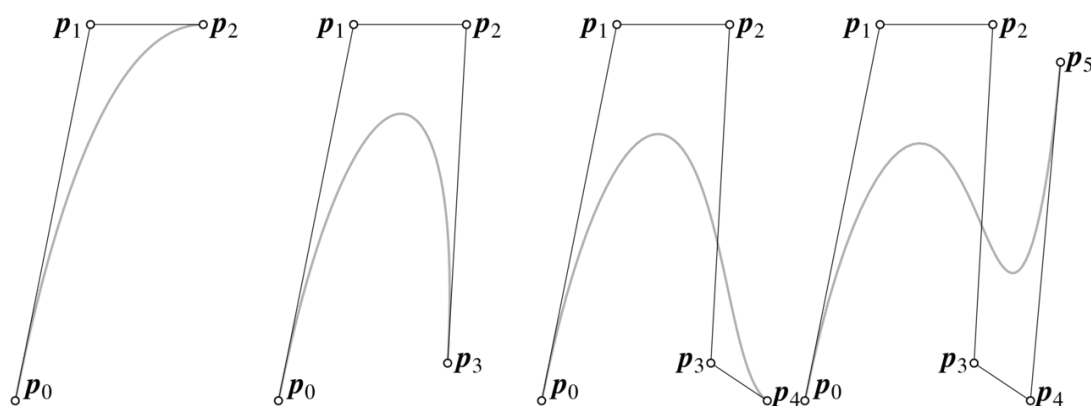
6.1.1. Określenie krzywych Béziera

Nazwa „krzywa Béziera” stopnia n oznacza reprezentację krzywej parametrycznej w postaci ciągu punktów $\mathbf{p}_0, \dots, \mathbf{p}_n$, tzw. **punktów kontrolnych**, które należy podstawić do wzoru

$$\mathbf{p}(t) = \sum_{i=0}^n \mathbf{p}_i B_i^n(t).$$

Występujące w nim funkcje B_i^n to **wielomiany Bernsteina** stopnia n , określone wzorem

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}.$$



Rysunek 6.1. Krzywe Béziera stopni 2, 3, 4 i 5 (dla $t \in [0, 1]$).

Powyższą reprezentację krzywych i stanowiącą jej rozwinięcie reprezentację płatów powierzchni we wczesnych latach tysiąc dziewięćset sześćdziesiątych opracowali niezależnie Pierre Bézier i Paul de Faget de Casteljau, na potrzeby firm Renault i Citroën. Dzięki swoim zaletom, takim jak łatwość interakcyjnego kształtowania i istnienie sprawnych algorytmów przetwarzania, reprezentacje te obecnie używane powszechnie nie tylko w inżynierskich systemach projektowania wspomagane komputerem, ale także w wielu innych zastosowaniach graficznych (np. w projektowaniu czcionek).

6.1.2. Algorytm de Casteljau

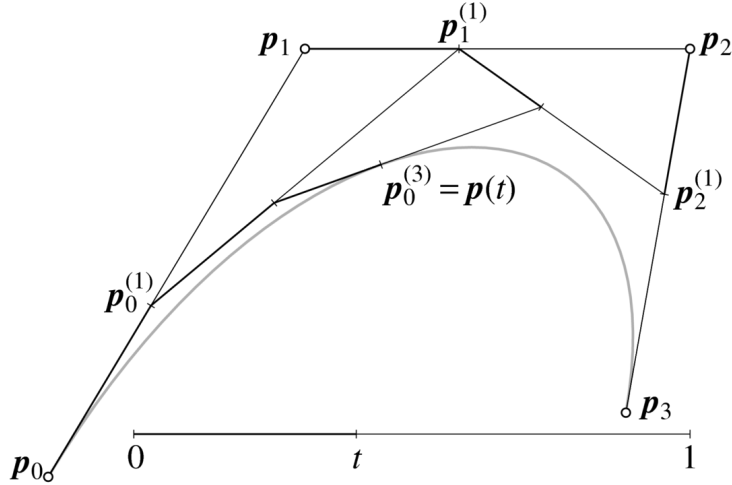
Dla dowolnego $t \in \mathbb{R}$ punkt $\mathbf{p}(t)$ można obliczyć za pomocą **algorytmu de Casteljau**, realizowanego przez następujący podprogram:

Listing.

```

{  $p_i^{(0)} = p_i$  dla  $i = 0, \dots, n$ . }
for  $j := 1$  to  $n$  do
  for  $i := 0$  to  $n - j$  do
     $p_i^{(j)} := (1 - t)p_i^{(j-1)} + tp_{i+1}^{(j-1)}$ ;
  {  $p(t) = p_0^{(n)}$ . }

```



Rysunek 6.2. Algorytm de Casteljau.

Udowodnimy, że punkt $p(t)$ jest punktem $p_0^{(n)}$ obliczonym przez powyższą procedurę. Jeśli $n = 0$, to krzywa jest zdegenerowana do jednego punktu, czyli $p(t) = p_0 B_0^0(t) = p_0 = p_0^{(0)}$, ponieważ $B_0^0(t) = 1$ dla każdego $t \in \mathbb{R}$. Załóżmy, że $n > 0$. Najpierw obliczymy

$$(1 - t)B_0^{n-1}(t) = (1 - t)^n = B_0^n(t), \quad tB_{n-1}^{n-1}(t) = t^n = B_n^n(t),$$

oraz dla $i \in \{1, \dots, n - 1\}$

$$(1 - t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) = \binom{n-1}{i} t^i (1 - t)^{n-i} + \binom{n-1}{i-1} t^i (1 - t)^{n-i} = B_i^n(t).$$

Następnie przyjmujemy założenie indukcyjne, że punkty $p_0^{(n-1)}$ i $p_1^{(n-1)}$ są odpowiadającymi danej wartości parametru t punktami krzywych Béziera stopnia $n - 1$, reprezentowanych przez punkty kontrolne odpowiednio $p_0, \dots, p_{n-1}$ i $p_1, \dots, p_n$. Na podstawie pokazanego wyżej rekurencyjnego związku między wielomianami Bernsteina stopni $n - 1$ i n , mamy

$$\begin{aligned}
p_0^{(n)} &= (1 - t) \sum_{i=0}^{n-1} p_i B_i^{n-1}(t) + t \sum_{i=1}^n p_i B_{i-1}^{n-1}(t) = \\
&= p_0 (1 - t) B_0^{n-1}(t) + \sum_{i=1}^{n-1} p_i ((1 - t) B_i^{n-1}(t) + t B_{i-1}^{n-1}(t)) + p_n t B_{n-1}^{n-1}(t) = \\
&= \sum_{i=0}^n p_i B_i^n(t).
\end{aligned}$$

Wielu podanych dalej własności krzywych Béziera i B-sklejanych dowodzi się w podobny sposób.

Algorytm de Casteljau ma oprócz obliczania punktów krzywej wiele innych zastosowań, o których będzie dalej.

6.1.3. Własności krzywych Béziera

Krzywe Béziera mają następujące własności:

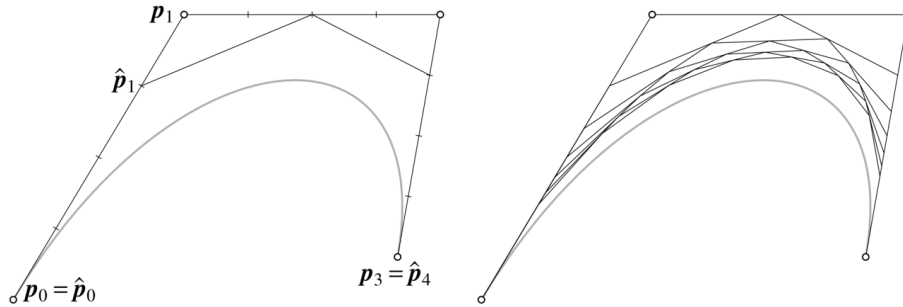
- **Nieziemniczość afiniczna reprezentacji.** Suma wielomianów Bernsteina stopnia n jest równa 1, a zatem dla dowolnego $t \in \mathbb{R}$ punkt $\mathbf{p}(t)$ jest kombinacją afiniczną punktów kontrolnych. Ponieważ przekształcenia afiniczne zachowują kombinacje afiniczne, więc dla ustalonego przekształcenia afinicznego f i dla każdego t odpowiedni punkt krzywej Béziera reprezentowanej przez punkty kontrolne $f(\mathbf{p}_0), \dots, f(\mathbf{p}_n)$ jest równy $f(\mathbf{p}(t))$. Innymi słowy, aby otrzymać obraz krzywej Béziera w dowolnym przekształceniu afinicznym, wystarczy poddać temu przekształceniu jej punkty kontrolne.
- **Własność otoczki wypukłej.** Dla $t \in [0, 1]$ punkt $\mathbf{p}(t)$ jest kombinacją wypukłą punktów kontrolnych (wielomiany Bernsteina są w przedziale $[0, 1]$ nieujemne), a więc należy do otoczki wypukłej ich zbioru.
- Zachodzi **interpolacja skrajnych punktów kontrolnych:**

$$\mathbf{p}(0) = \mathbf{p}_0, \quad \mathbf{p}(1) = \mathbf{p}_n.$$

- Dla $t \in [0, 1]$ krzywa Béziera nie ma z żadną prostą (na płaszczyźnie) albo płaszczyzną (w przestrzeni) większej liczby punktów przecięcia niż jej łamana kontrolna (to się nazywa **własnością zmniejszania wariacji**).
- Istnieje możliwość **podwyższenia stopnia**, czyli znalezienia reprezentacji stopnia $n + 1$. Związek obu reprezentacji wyraża się wzorami

$$\mathbf{p}(t) = \sum_{i=0}^n \mathbf{p}_i B_i^n(t) = \sum_{i=0}^{n+1} \hat{\mathbf{p}}_i B_i^{n+1}(t),$$

$$\hat{\mathbf{p}}_i = \frac{i}{n+1} \mathbf{p}_{i-1} + \frac{n+1-i}{n+1} \mathbf{p}_i.$$



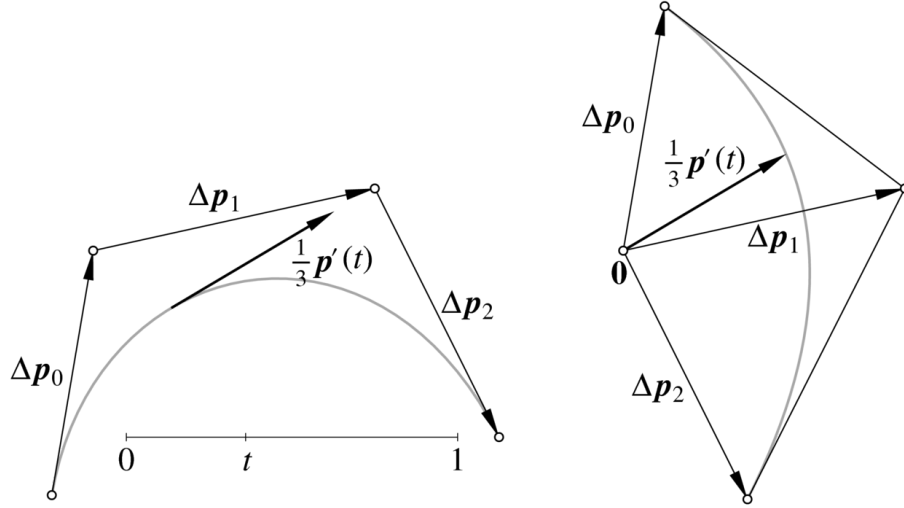
Rysunek 6.3. Podwyższenie stopnia krzywej Béziera.

Podwyższanie stopnia możemy iterować, dostając reprezentacje coraz wyższych stopni. Ciąg łamanych kontrolnych otrzymanych w ten sposób zbiega jednostajnie do krzywej dla $t \in [0, 1]$. Zbieżność tego ciągu jest jednak zbyt wolna, aby miała praktyczne znaczenie.

- Jeśli kolejne punkty kontrolne leżą na prostej, w kolejności indeksów i w równych odstępach, to krzywa Béziera jest odcinkiem sparametryzowanym ze stałą prędkością. Najłatwiej jest udowodnić to rozpatrując reprezentację odcinka w postaci krzywej Béziera stopnia 1 i jego reprezentacje otrzymane przez podwyższanie stopnia.
- Pochodna krzywej Béziera o punktach kontrolnych $\mathbf{p}_0, \dots, \mathbf{p}_n$ wyraża się wzorem

$$\mathbf{p}'(t) = \sum_{i=0}^{n-1} n(\mathbf{p}_{i+1} - \mathbf{p}_i) B_i^{n-1}(t) = \sum_{i=0}^{n-1} n \Delta \mathbf{p}_i B_i^{n-1}(t).$$

Na podstawie własności otoczki wypukłej mamy więc **własność hodografu**, według której kierunek wektora $\mathbf{p}'(t)$ dla $t \in [0, 1]$ jest zawarty w stożku rozpiętym przez wektory $\Delta \mathbf{P}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$ dla $i = 0, \dots, n-1$. Ponadto zachodzą równości $\mathbf{p}'(0) = n(\mathbf{p}_1 - \mathbf{p}_0)$ oraz $\mathbf{p}'(1) = n(\mathbf{p}_n - \mathbf{p}_{n-1})$.



Rysunek 6.4. Pochodna krzywej Béziera.

- **Podział krzywej.** Punkty $\mathbf{p}_0^{(0)}, \dots, \mathbf{p}_0^{(n)}$ oraz $\mathbf{p}_0^{(n)}, \dots, \mathbf{p}_n^{(0)}$, otrzymane w trakcie wykonywania algorytmu de Casteljau, są punktami kontrolnymi tej samej krzywej, w innych parametryzacjach. Dokładniej, dla dowolnego $s \in \mathbb{R}$ zachodzą równości

$$\mathbf{p}(s) = \sum_{i=0}^n \mathbf{p}_i B_i^n(s) = \sum_{i=0}^n \mathbf{p}_0^{(i)} B_i^n\left(\frac{s}{t}\right) = \sum_{i=0}^n \mathbf{p}_i^{(n-i)} B_i^n\left(\frac{s-t}{1-t}\right).$$

Aby narysować krzywą, możemy dzielić ją na „dostatecznie krótkie” łuki i rysować zamiast nich odcinki. Możemy użyć w tym celu procedury rekurencyjnej (porównaj ją z przedstawioną wcześniej procedurą rysowania elipsy):

Listing.

```

procedure r_Krzywa ( n, p );
begin
  if dostatecznie blisko ( p[0], p[n] )
  then rysuj odcinek ( p[0], p[n] );
  else begin
    q[0] := p[0];
    for j := 1 to n do begin
      for i := 0 to n - j do
        p[i] := 1/2 (p[i] + p[i + 1]);
      q[j] := p[0]
    end;
    r_Krzywa ( n, q );
    r_Krzywa ( n, p )
  end
end { r_Krzywa };

```

Otoczka wypukła łamanej kontrolnej „całej” krzywej jest z reguły znacznie większa niż suma otoczek łamanych kontrolnych kilku jej fragmentów, a zatem przez podział możemy uzyskiwać znacznie dokładniejsze oszacowania położenia krzywej.

- Algorytm szybkiego obliczania punktu $\mathbf{p}(t)$ (o koszcie $O(n)$ zamiast $O(n^2)$, jak w przypadku algorytmu de Casteljau) możemy uzyskać, adaptując schemat Hornera. Podstawiając $s = 1 - t$, otrzymujemy

$$\begin{aligned} \mathbf{p}(t) = & \mathbf{p}_0 \binom{n}{0} s^n + \mathbf{p}_1 \binom{n}{1} t s^{n-1} + \cdots + \mathbf{p}_{n-1} \binom{n}{n-1} t^{n-1} s + \mathbf{p}_n \binom{n}{n} t^n = \\ & (\cdots (\mathbf{p}_0 \binom{n}{0} s + \mathbf{p}_1 \binom{n}{1} t) s + \cdots + \mathbf{p}_{n-1} \binom{n}{n-1} t^{n-1}) s + \mathbf{p}_n \binom{n}{n} t^n. \end{aligned}$$

Schemat Hornera ze względu na s trzeba tylko uzupełnić o obliczanie wektorów $\mathbf{p}_0 \binom{n}{0}$, $\mathbf{p}_1 \binom{n}{1} t$, $\dots$, $\mathbf{p}_n \binom{n}{n} t^n$. Ponieważ $\binom{n}{0} = 1$ i $\binom{n}{i+1} = \frac{n-i}{i+1} \binom{n}{i}$, więc mamy stąd algorytm

Listing.

```

s := 1 - t;
p := p0;
d := t;
b := n;
for i := 1 to n do begin
    p := sp + b*d*p_i;
    d := d*t;
    b := b*(n - i)/(i + 1)
end;
{ p(t) = p }

```

6.2. Krzywe B-sklejane

6.2.1. Określenie krzywych B-sklejanych

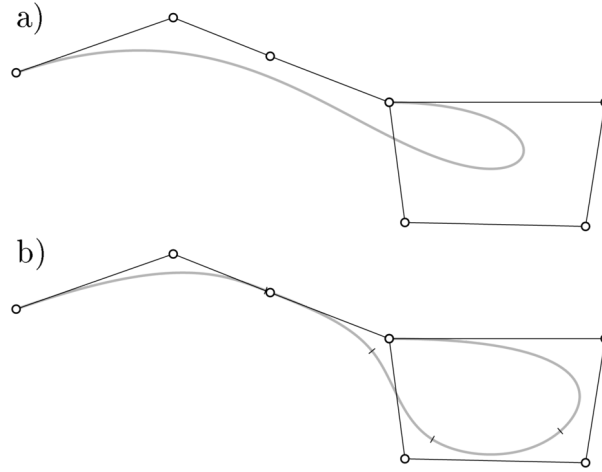
Modelowanie figur o skomplikowanym kształcie wymagałoby użycia krzywych Béziera wysokiego stopnia, co oprócz niewygody (z punktu widzenia użytkownika programu interakcyjnego) sprawiałoby różne kłopoty implementacyjne (bardzo duże wartości współczynników dwumianowych, wysoki koszt algorytmów obliczania punktu). Dlatego często stosuje się krzywe kawałkami wielomianowe w reprezentacji B-sklejanej; jest ona uogólnieniem reprezentacji Béziera krzywych wielomianowych.

Krzywa B-sklejana jest określona przez podanie: **stopnia** n , ciągu $N + 1$ węzłów $u_0, \dots, u_N$ (ciąg ten powinien być niemalejący, a ponadto N powinno być większe od $2n$), oraz $N - n$ **punktów kontrolnych** $\mathbf{d}_0, \dots, \mathbf{d}_{N-n-1}$. Wzór, który jest definicją krzywej B-sklejanej ma postać

$$\mathbf{s}(t) = \sum_{i=0}^{N-n-1} \mathbf{d}_i N_i^n(t), \quad t \in [u_n, u_{N-n}].$$

We wzorze tym występują **funkcje B-sklejane** N_i^n stopnia n , które są określone przez ustalony ciąg węzłów.

Istnieje kilka definicji funkcji B-sklejanych, które różnią się stopniem skomplikowania, a także trudnością dowodzenia na podstawie takiej definicji różnych własności tych funkcji (w zasadzie



Rysunek 6.5. Porównanie krzywej Béziera z krzywą B-sklejaną.

więc wychodzi na jedno, której definicji użyjemy, jeśli chcemy dowodzić twierdzenia, to trudności nie da się uniknąć). Ponieważ w tym wykładzie ograniczamy się do praktycznych aspektów zagadnienia, więc przytoczę rekurencyjny **wzór Mansfielda-de Boora-Coxa**, który w książkach o grafice chyba najczęściej pełni rolę definicji:

$$N_i^0(t) = \begin{cases} 1 & \text{dla } t \in [u_i, u_{i+1}), \\ 0 & \text{w przeciwnym razie,} \end{cases} \quad (6.1)$$

$$N_i^n(t) = \frac{t - u_i}{u_{i+n} - u_i} N_i^{n-1}(t) + \frac{u_{i+n+1} - t}{u_{i+n+1} - u_{i+1}} N_{i+1}^{n-1}(t) \quad \text{dla } n > 0. \quad (6.2)$$

Wzór ten jest uogólnieniem rozważanego wcześniej wzoru wiążącego wielomiany Bernsteina stopni $n - 1$ i n .

6.2.2. Algorytm de Boora

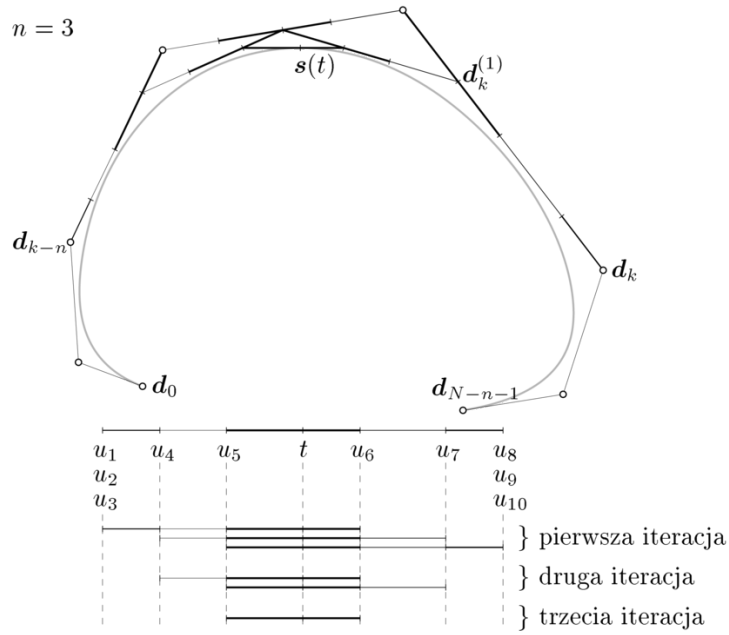
Na podstawie wzoru Mansfielda-de Boora-Coxa łatwo jest otrzymać **algorytm de Boora** obliczania punktu $s(t)$ na podstawie danych opisujących krzywą s i parametru t . Algorytm ten jest uogólnieniem algorytmu de Casteljau i jest realizowany przez następujące instrukcje:

Listing.

```

{  $d_i^{(0)} = d_i$  dla  $i = k - n, \dots, k$ . }
 $k := N - n - 1$ ;
while  $t < u_k$  do  $k := k - 1$ ;
 $r := 0$ ;
while  $(r < n)$  and  $(t = u_{k-r})$  do  $r := r + 1$ ;
for  $j := 1$  to  $n - r$  do
  for  $i := k - n + j$  to  $k - r$  do begin
     $\alpha_i^{(j)} := (t - u_i) / (u_{i+n+1-j} - u_i)$ ;
     $d_i^{(j)} := (1 - \alpha_i^{(j)}) d_{i-1}^{(j-1)} + \alpha_i^{(j)} d_i^{(j-1)}$ 
  end;
{  $s(t) = d_{k-r}^{(n-r)}$  }

```

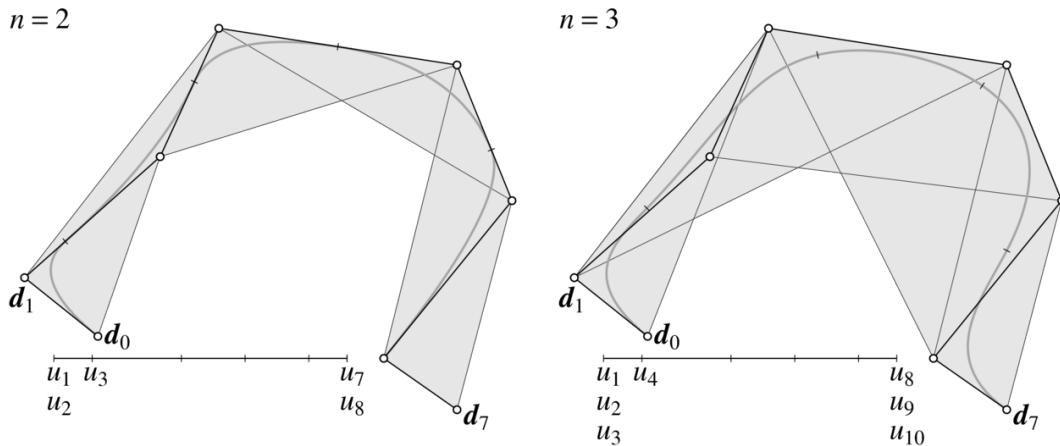


Rysunek 6.6. Algorytm de Boora.

6.2.3. Podstawowe własności krzywych B-sklejanych

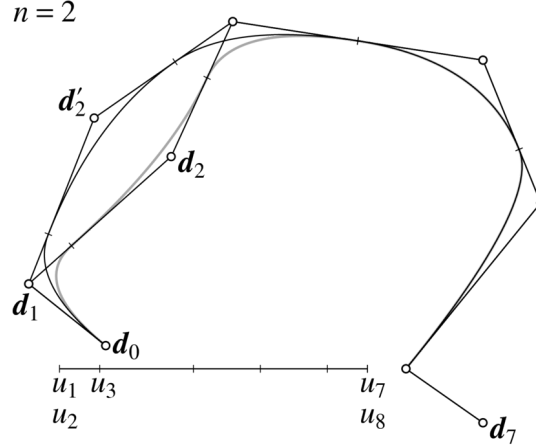
Własności krzywych B-sklejanych najbardziej istotne w zastosowaniach związanych z grafiką komputerową, są takie:

- Jeśli wszystkie węzły od u_n do u_{N-n} są różne (tworzą ciąg rosnący), to krzywa składa się z $N - 2n$ łuków wielomianowych. W przeciwnym razie (jeśli występują tzw. węzły krotne), to liczba łuków jest mniejsza.
- Algorytm de Boora dokonuje liniowej interpolacji kolejno otrzymywanych punktów (liczby $\alpha_i^{(j)}$ należą do przedziału $[0, 1]$); stąd wynika **silna własność otoczki wypukłej**: wszystkie punkty łuku dla $t \in [u_k, u_{k+1}]$ leżą w otoczce wypukłej punktów $d_{k-n}, \dots, d_k$. Mamy też **afiniczną niezmienniczość** tej reprezentacji krzywej; aby otrzymać jej obraz w dowolnym przekształceniu afinicznym, wystarczy zastosować to przekształcenie do punktów kontrolnych $d_0, \dots, d_{N-n-1}$.



Rysunek 6.7. Silna własność otoczki wypukłej.

- **Lokalna kontrola kształtu.** Ponieważ punkt $s(t)$ dla $t \in [u_k, u_{k+1})$ zależy tylko od punktów $d_{n-k}, \dots, d_k$, więc zmiana punktu d_i powoduje zmianę fragmentu krzywej dla $t \in [u_i, u_{i+n+1}]$.

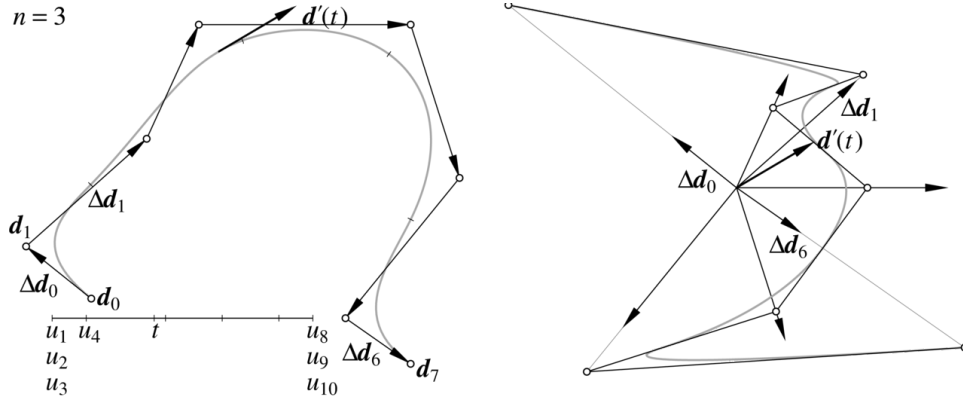


Rysunek 6.8. Lokalna kontrola kształtu.

- Pochodna krzywej B-sklejanej stopnia n jest krzywą stopnia $n - 1$:

$$s'(t) = \sum_{i=0}^{N-n-2} \frac{n}{u_{i+n+1} - u_{i+1}} (d_{i+1} - d_i) N_{i+1}^{n-1}(t).$$

Funkcje B-sklejane N_i^{n-1} występujące w powyższym wzorze są określone dla tego samego ciągu węzłów, co funkcje N_i^n w określeniu krzywej s . Zastosowanie (silnej) własności otoczki wypukłej do pochodnej daje (**silną**) **własność hodografu** krzywych B-sklejanych.



Rysunek 6.9. Pochodna krzywej B-sklejanej.

- W otoczeniu węzła o krotności r krzywa jest klasy C^{n-r} (dowód na podstawie wzoru Mansfielda-de Boora-Coxa, który przyjęliśmy tu za definicję, jest pracochłonny, ale reguła jest prosta, więc warto ją zapamiętać).
- Jeśli dwa sąsiednie węzły są n -krotne, to łuk krzywej między nimi jest krzywą Béziera; dokładniej, jeśli $u_{k-n+1} = \dots = u_k < u_{k+1} = \dots = u_{k+n}$, to dla $t \in [u_k, u_{k+1}]$ mamy

$$s(t) = \sum_{i=0}^n d_{k-n+i} B_i^n \left(\frac{t - u_k}{u_{k+1} - u_k} \right).$$

6.2.4. Wstawianie węzła

Wstawianie węzła jest procedurą, która zmienia reprezentację krzywej, nie zmieniając samej krzywej. W wyniku zastosowania tej procedury otrzymujemy reprezentację o większej liczbie węzłów i punktów kontrolnych. Idea postępowania, które prowadzi do otrzymania tej reprezentacji może być przedstawiona w następujących krokach

1. Określamy tak zwane **współrzędne Greville’a**:

$$\xi_i = \frac{1}{n}(u_{i+1} + \dots + u_{i+n}).$$

Będziemy (w myśli) przekształcać łamaną o wierzchołkach $\mathbf{f}_i = [\xi_i, \mathbf{d}_i]$.

2. Dołączamy liczbę $t \in [u_n, u_{N-n}]$ (wstawiany węzeł) do wyjściowego ciągu węzłów, z zachowaniem uporządkowania.
3. Obliczamy nowe współrzędne Greville’a ξ_i^t , odpowiadające ciągowi węzłów z dołączoną liczbą t .
4. Znajdujemy punkty $\mathbf{f}_i^t = [\xi_i^t, \mathbf{d}_i^t]$ na łamanej. Punkty $\mathbf{d}_i^t$ są punktami kontrolnymi nowej reprezentacji krzywej.

Praktyczna implementacja procedury wstawiania węzła nie musi obliczać współrzędnych Greville’a, które pełnią tu rolę pomocniczą. Równoważny skutek daje następujący podprogram:

Listing.

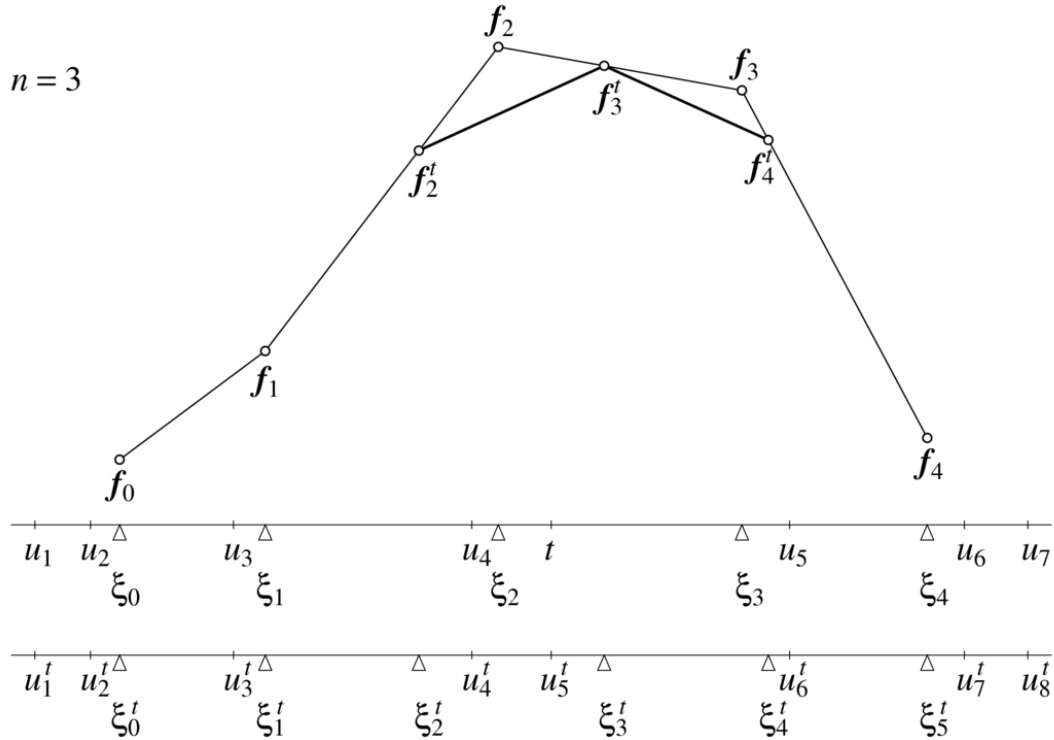
```

{  $u[i] = u_i$  dla  $i = 1, \dots, N-1$ ,  $d[i] = d_i$  dla  $i = 0, \dots, N-n-1$ , }
{  $t \in [u_n, u_{N-n}]$  }
 $k := N-1$ ;
while  $t < u[k]$  do
     $k := k-1$ ;
 $r := 0$ ;  $i := k$ ;
while  $(i \geq 1)$  and  $(t = u[i])$  do
    begin  $i := i-1$ ;  $r := r+1$  end;
for  $i := N-n-1$  downto  $k-r$  do
     $d[i+1] := d[i]$ ;
for  $i := k-r$  downto  $k-n+1$  do
     $d[i] := ((u[i+n] - t)*d[i-1] + (t - u[i])*d[i]) / (u[i+n] - u[i]);$ 
for  $i := N-1$  downto  $k+1$  do
     $u[i+1] := u[i]$ ;
 $u[k+1] := t$ ;
 $N := N+1$ ;
{ zmienna  $N$  oraz tablice  $u$  i  $d$  zawierają wynik wstawiania węzła. }

```

Własności tego przekształcenia reprezentacji krzywej są następujące:

- Po wstawieniu węzła liczba punktów kontrolnych jest większa o 1.
- Wynik wstawienia kilku węzłów nie zależy od kolejności ich wstawiania.
- Krzywa reprezentowana przez nowy ciąg węzłów i nowy ciąg punktów kontrolnych jest identyczna z krzywą wyjściową.
- Algorytm de Boora jest procedurą wstawiania węzła, powtórzoną tyle razy, aby ostatecznie otrzymać n -krotny węzeł t .
- Po wstawieniu dostatecznie wielu węzłów (tak, aby ich odległości stały się dostatecznie małe), otrzymujemy łamaną kontrolną, która jest dowolnie bliska krzywej. Odległość odpowiednio sparametryzowanej łamanej kontrolnej od reprezentowanej przez nią krzywej B-sklejanej jest proporcjonalna do h^2 , gdzie h oznacza maksymalną odległość sąsiednich węzłów. Dlatego po wstawieniu nawet niezbyt dużej liczby węzłów możemy otrzymać łamaną, którą można narysować w celu uzyskania dość dokładnego obrazu krzywej.



Rysunek 6.10. Zasada wstawiania węzła.

Procedurę wstawiania węzła możemy wykorzystać tak:

1. wybieramy początkowo małą liczbę węzłów i punktów kontrolnych w celu zgrubnego ukształtowania krzywej,
2. wstawiamy pewną liczbę dodatkowych węzłów,
3. poprawiamy krzywą w celu wymodelowania szczegółów.

Inne zastosowanie to wstawienie węzłów tak, aby krotność wszystkich węzłów była równa $n + 1$; łamana kontrolna składa się wtedy z łamanych kontrolnych Béziera poszczególnych łuków wielomianowych, które można narysować za pomocą jakiejś szybkiej procedury, np. opartej na schemacie Hornera. Procedury rysowania krzywych Béziera mogą być zaimplementowane w sprzęcie. Mając procedurę wstawiania węzłów, możemy użyć takiego sprzętu do rysowania krzywych B-sklejanych.

6.2.5. Krzywe B-sklejane z węzłami równoodległymi

Narzucenie węzłów równoodległych ogranicza klasę krzywych B-sklejanych, ale jednocześnie upraszcza wzory i umożliwia stosowanie specjalnych algorytmów.

Przypuśćmy (bez straty ogólności), że dana krzywa jest reprezentowana przez nieskończony ciąg węzłów, składający się z wszystkich liczb całkowitych, i nieskończony ciąg punktów kontrolnych c_i ; mamy zatem

$$s(t) = \sum_{i \in \mathbb{Z}} c_i N_i^n(t),$$

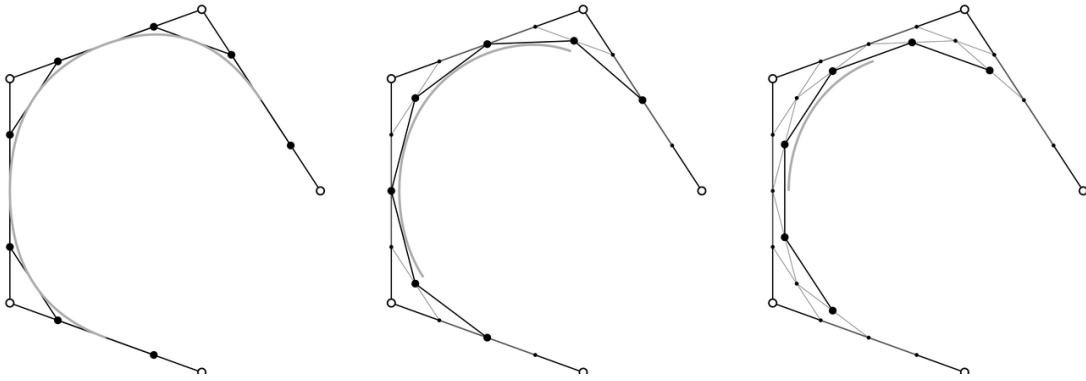
gdzie każda funkcja B-sklejana N_i^n stopnia n przyjmuje wartości różne od zera tylko w przedziale $[i, i + n + 1)$. Ponadto dla każdego $i \in \mathbb{Z}$ oraz $t \in \mathbb{R}$ mamy $N_i^n(t) = N_0^n(t - i)$.

Rozważmy teraz nieskończony ciąg węzłów, składający się z wszystkich całkowitych wielokrotności liczby $\frac{1}{2}$. Niech M_i^n oznacza funkcję B-sklejaną opartą na tym ciągu węzłów i przyjmującą niezerowe wartości w przedziale $[\frac{i}{2}, \frac{i+n+1}{2})$. Krzywą s możemy przedstawić w postaci

$$s(t) = \sum_{i \in \mathbb{Z}_s} d_i M_i^n(t).$$

Na podstawie danych punktów c_i należy znaleźć punkty d_i .

Algorytm Lane’a-Riesenfelda, który rozwiązuje to zadanie, podaję bez dowodu. Algorytm ten składa się z dwóch etapów. Pierwszy z nich to **podwajanie**: konstruujemy punkty $d_{2i}^{(0)} = d_{2i+1}^{(0)} = c_i$. W drugim etapie wykonujemy n -krotnie operację **uśredniania**: obliczamy punkty $d_i^{(j)} = \frac{1}{2}(d_{i-1}^{(j-1)} + d_i^{(j-1)})$. Mając daną początkową reprezentację krzywej s w postaci łamanej o wierzchołkach $c_0, \dots, c_N$, w kolejnych krokach uśredniania otrzymamy łamane o wierzchołkach $d_j^{(j)}, \dots, d_{2N+1-j}^{(j)}$. Wynikiem obliczenia są punkty $d_i = d_i^{(n)}$ dla każdego $i \in \mathbb{Z}$.



Rysunek 6.11. Algorytm Lane’a-Riesenfelda dla krzywych stopnia 2, 3 i 4.

Wyniki działania algorytmu dla krzywych stopnia 2, 3 i 4 są pokazane na rysunku 6.11. Otrzymaną łamaną można następnie użyć jako dane dla algorytmu Lane’a-Riesenfelda i otrzymać łamaną reprezentującą krzywą s przy użyciu ciągu węzłów $(\frac{i}{4})_{i \in \mathbb{Z}_s}$ itd. Określony w ten sposób nieskończony ciąg łamanych, z których każda jest reprezentacją krzywej związanej z odpowiednim ciągiem węzłów równoodległych, jest szybko zbieżny do krzywej, można zatem narysować jedną z tych łamanych zamiast krzywej.

6.3. Powierzchnie Béziera i B-sklejane

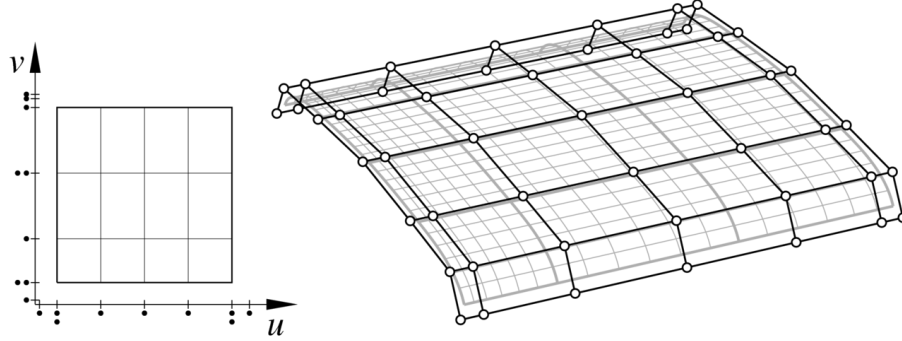
6.3.1. Płaty tensorowe

Do określenia powierzchni potrzebne są funkcje dwóch zmiennych. Najczęściej wykorzystuje się iloczyn tensorowy przestrzeni funkcji jednej zmiennej. Użycie go prowadzi do wzorów

$$p(u, v) = \sum_{i=0}^n \sum_{j=0}^m p_{ij} B_i^n(u) B_j^m(v),$$

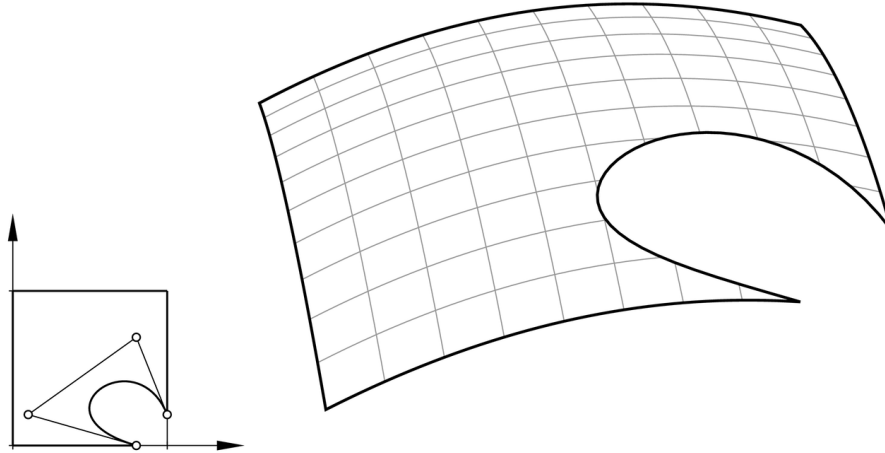
$$s(u, v) = \sum_{i=0}^{N-n-1} \sum_{j=0}^{M-m-1} d_{ij} N_i^n(u) N_j^m(v),$$

które opisują odpowiednio płat powierzchni Béziera i płat powierzchni B-sklejanej stopnia (n, m) . W przypadku płata B-sklejanego, nawet jeśli stopień ze względu na każdy parametr jest taki sam, można podać inny ciąg węzłów określających funkcje bazowe.



Rysunek 6.12. Płat B-sklejany.

Dziedziną płata Béziera jest zwykle kwadrat jednostkowy. Dziedziną płata B-sklejanego jest prostokąt $[u_n, u_{N-n}] \times [v_m, v_{M-m}]$. Często dziedzinę otrzymuje się przez odrzucenie fragmentów takiego prostokąta; mamy wtedy **płat obcięty** (rys. 6.13).

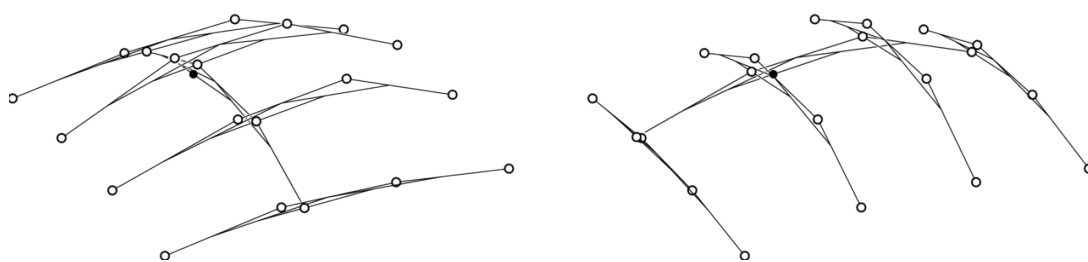


Rysunek 6.13. Obcięty płat Béziera.

Punkty kontrolne płata każdego z tych rodzajów dla wygody kształtowania przedstawia się w postaci **siatki**. Wyróżniamy w niej **wiersze** i **kolumny**. Wyznaczenie punktu na powierzchni, dla ustalonych parametrów u, v , można sprowadzić do wyznaczania punktów na krzywych (Béziera albo B-sklejanych), np.

$$p(u, v) = \sum_{i=0}^n \underbrace{\left(\sum_{j=0}^m p_{ij} B_j^m(v) \right)}_{q_i} B_i^n(u) = \sum_{i=0}^n q_i B_i^n(u).$$

Wszystkie działania wykonujemy na kolumnach siatki, traktując je tak, jakby to były łamane kontrolne krzywych. Punkty tych krzywych, odpowiadające ustalonemu v , są punktami kontrolnymi krzywej stałego parametru u leżącymi na płacie. Można też postąpić w odwrotnej kolejności i najpierw przetwarzać wiersze, a potem kolumnę otrzymanych punktów.



Rysunek 6.14. Wyznaczanie punktu na płacie Béziera.

Zasada przetwarzania reprezentacji płata w celu podwyższenia stopnia, podziału na kawałki, wstawienia węzła i obliczenia pochodnych cząstkowych jest identyczna.

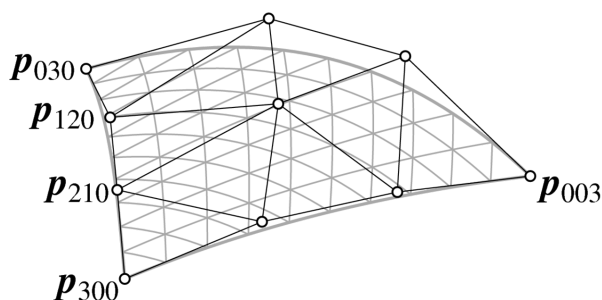
6.3.2. Płaty trójkątne

Dziedziną **trójkątnego płata Béziera** jest zwykle trójkąt, którego wierzchołki stanowią układ odniesienia układu współrzędnych barycentrycznych r , s , t . Suma tych współrzędnych jest równa 1, wewnątrz trójkąta mają one wartości dodatnie.

Płat jest określony wzorem

$$p(r, s, t) = \sum_{\substack{i, j, k \geq 0 \\ i+j+k=n}} p_{ijk} B_{ijk}^n(r, s, t),$$

w którym występują **wielomiany Bernsteina trzech zmiennych** stopnia n i **punkty kontrolne** p_{ijk} , będące wierzchołkami **siatki kontrolnej** płata trójkątnego.



Rysunek 6.15. Trójkątny płat Béziera i jego siatka kontrolna.

Algorytm wyznaczania punktu (algorytm de Casteljau) jest następujący:

Listing.

```

{  $p_{ijk}^{(0)} = p_{ijk}$  dla  $i, j, k \geq 0, i + j + k = n$  }
for  $l := 1$  to  $n$  do
  for  $i, j, k \geq 0, i + j + k = n - l$  do
     $p_{ijk}^{(l)} := r p_{i+1, j, k}^{(l-1)} + s p_{i, j+1, k}^{(l-1)} + t p_{i, j, k+1}^{(l-1)}$ ;
  {  $p(r, s, t) = p_{000}^{(n)}$  }

```

Podobnie jak w przypadku krzywych Béziera, algorytm de Casteljau pozwala dokonać podziału płata. Punkty kontrolne jego fragmentów to odpowiednio $p_{0jk}^{(i)}$, $p_{i0k}^{(j)}$, $p_{ij0}^{(k)}$. Jeśli współrzędna r , s lub t jest równa 0 (punkt, któremu odpowiada obliczony punkt na płacie, leży na

boku trójkąta, który jest dziedziną), to mamy możliwość podziału płata na dwa fragmenty. Algorytm de Casteljau działa wtedy w taki sposób, jakby poszczególne wiersze siatki kontrolnej były łamanymi kontrolnymi krzywymi Béziera stopni $0, 1, \dots, n$.

6.3.3. Metody siatek

Algorytm Lane'a-Riesenfelda dla powierzchni B-sklejanych stopnia (n, n) jest dwuwymiarową wersją algorytmu przedstawionego wcześniej dla krzywej. Punkty c_{ik} , przy użyciu których powierzchnia sklejana jest określona wzorem

$$s(u, v) = \sum_{i \in \mathbb{Z}s} \sum_{j \in \mathbb{Z}s} c_{ik} N_i^n(u) N_k^n(v),$$

z funkcjami B-sklejanymi stopnia n opartymi na (tym samym) ciągu węzłów całkowitych, posłużą do obliczenia punktów d_{ik} , takich że

$$s(u, v) = \sum_{i \in \mathbb{Z}s} \sum_{j \in \mathbb{Z}s} d_{ij} M_i^n(u) M_j^n(v),$$

gdzie funkcje M_i^n są określone przy użyciu ciągu węzłów $(\frac{i}{2})_{i \in \mathbb{Z}s}$. Algorytm składa się z kroku podwajania i n kroków uśredniania, przy czym

1. W kroku podwajania przyjmujemy

$$d_{2i,2k}^{(0)} = d_{2i,2k+1}^{(0)} = d_{2i+1,2k}^{(0)} = d_{2i+1,2k+1}^{(0)} = c_{ik}.$$

2. W kroku uśredniania dla $j = 1, \dots, n$ przyjmujemy

$$d_{ik}^{(j)} = \frac{1}{4} (d_{i-1,k-1}^{(j-1)} + d_{i,k-1}^{(j-1)} + d_{i-1,k}^{(j-1)} + d_{ik}^{(j-1)}).$$

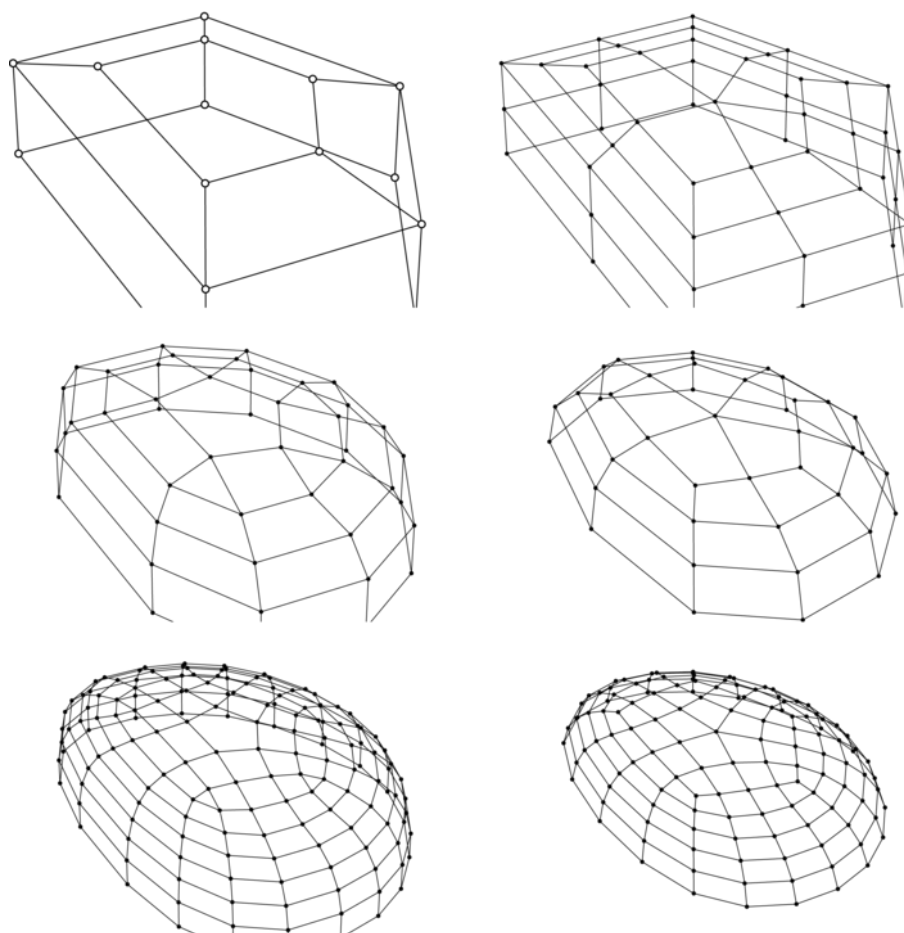
W ten sposób otrzymujemy nową siatkę kontrolną; powtarzając ten algorytm, otrzymamy ciąg siatek szybko zbieżny do powierzchni s . Dowolną z tych siatek (otrzymaną np. po kilku, najwyżej kilkunastu iteracjach) możemy uznać za dostatecznie dokładne przybliżenie powierzchni i narysować zamiast niej.

Okazuje się, że opisane wyżej postępowanie daje się uogólnić na siatki nieregularne, tj. takie, których wierzchołków nie można ustawić w prostokątną tablicę. Rozważmy siatkę jako graf; wierzchołki siatki są wierzchołkami grafu, odcinki siatki są krawędziami grafu, możemy też określić ściany grafu, jako łamane zamknięte złożone z krawędzi, takie że odrzucenie wierzchołków i krawędzi łamanej nie likwiduje spójności grafu. Zakładamy, że każda krawędź należy do jednej lub do dwóch ścian.

Siatka jest **regularna**, jeśli wszystkie ściany mają 4 krawędzie i każdy wierzchołek nie leżący na brzegu siatki, tzw. wewnętrzny, jest stopnia 4 (tj. jest incydentny z czterema krawędziami). Każda ściana nie-czworokątna i każdy wierzchołek wewnętrzny stopnia różnego od 4 jest tzw. **elementem specjalnym** siatki.

Dla siatki, która zawiera elementy specjalne, możemy określić operację **podwajania** w ten sposób: każdy wierzchołek stopnia k zastępujemy przez ścianę (zdegenerowaną do punktu), która ma k wierzchołków i krawędzi. Każdą krawędź zastępujemy przez ścianę czworokątną, zdegenerowaną do odcinka; ściany dotychczasowe pozostawiamy (oczywiście, w nowej siatce zmieni się numeracja wierzchołków każdej ściany). Jeśli siatka jest regularna, to wynik jest taki sam jak wynik kroku podwajania w algorytmie Lane'a-Riesenfelda dla powierzchni.

Operację **uśredniania** określamy tak: konstruujemy graf dualny do siatki danej. Dla każdej ściany określamy wierzchołek, jako środek ciężkości wierzchołków tej ściany. Dla każdej krawędzi wspólnej dla dwóch ścian wprowadzamy krawędź łączącą wierzchołki skonstruowane dla tych



Rysunek 6.16. Siatki przetwarzane przez algorytmy Doo-Sabina i Catmulla-Clarka.

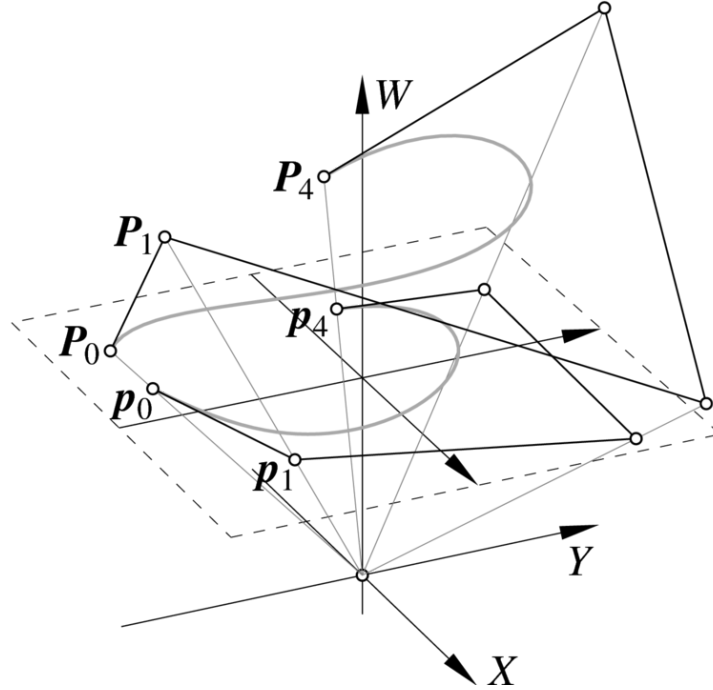
ścian. Ściany nowej siatki odpowiadają wierzchołkom wewnętrznym siatki danej. Również taka operacja uśredniania jest dla siatki regularnej identyczna z uśrednianiem wykonywanym w algorytmie Lane'a-Riesenfelda. Zauważmy, że choć liczba wierzchołków, krawędzi i ścian rośnie podczas podwajania, ale żadna z opisanych operacji nie może powiększyć liczby elementów specjalnych w siatce.

Możemy teraz wykonać podwajanie, a następnie n kroków uśredniania. Dwa najczęściej stosowane przypadki, dla $n = 2$ oraz $n = 3$, są znane odpowiednio jako algorytmy Doo-Sabina i Catmulla-Clarka. Iterując dowolny z tych algorytmów, otrzymamy ciąg siatek zbieżny do powierzchni granicznej. Powierzchnia ta prawie w całości składa się z kawałków wielomianowych stopnia (n, n) ; wyjątkiem jest otoczenie punktów, do których zbiegają elementy specjalne siatki.

6.4. Krzywe i powierzchnie wymierne

Krzywe i powierzchnie Béziera i B-sklejane są wielomianowe lub kawałkami wielomianowe, tj. ich parametryzacje są opisane za pomocą wielomianów. Nakłada to spore ograniczenia na możliwe do przedstawienia w tej postaci kształty, na przykład nie istnieje wielomianowa parametryzacja okręgu. Znacznie szersze możliwości modelowania udostępniają **krzywe i powierzchnie wymierne**. W zasadzie każdą reprezentację krzywych wielomianowych lub kawałkami wielomianowych (sklejanych) można uogólnić tak, aby otrzymać krzywe wymierne.

Aby to zrobić, wystarczy określić krzywą (albo powierzchnię) wielomianową w przestrzeni, której wymiar jest o 1 większy niż wymiar przestrzeni docelowej, tj. w przestrzeni współrzędnych jednorodnych. Mając współrzędne np. X, Y, Z i W punktu $\mathbf{P}(t)$ takiej krzywej jednorodnej, współrzędne punktu $\mathbf{p}(t)$ krzywej wymiernej otrzymamy ze wzorów $x = X/W, y = Y/W, z = Z/W$.



Rysunek 6.17. Wymierna krzywa Béziera i jej jednorodna reprezentacja.

Krzywą $\mathbf{P}$ w przestrzeni współrzędnych jednorodnych nazywamy **krzywą jednorodną**. Pewien kłopot w jej konstruowaniu sprawia fakt, że jeśli jest to np. krzywa Béziera, to jej punkty kontrolne są wektorami w przestrzeni współrzędnych jednorodnych; trudno byłoby nimi manipulować, jako że leżą w innej przestrzeni niż krzywa wymierna (i użytkownik programu do modelowania takiej krzywej oraz obraz krzywej utworzony przez ten program). Dlatego punkt kontrolny (albo inne wektory w przestrzeni współrzędnych jednorodnych, które służą do reprezentowania krzywej) najwygodniej jest określić za pomocą punktu (albo wektora) w przestrzeni, w której leży krzywa, oraz **wagi**, czyli liczbowego współczynnika dodatkowo określającego wpływ punktu na kształt krzywej. Na przykład, jeśli przyjmujemy punkty kontrolne $\mathbf{p}_i = [x_i, y_i, z_i]^T$ oraz wagi w_i , to krzywa Béziera, której punktami kontrolnymi są wektory

$$\mathbf{P}_i = \begin{bmatrix} w_i x_i \\ w_i y_i \\ w_i z_i \\ w_i \end{bmatrix},$$

jest jednorodną reprezentacją **wymiernej krzywej Béziera**, danej wzorem

$$\mathbf{p}(t) = \frac{\sum_{i=0}^n w_i \mathbf{p}_i B_i^n(t)}{\sum_{i=0}^n w_i B_i^n(t)}.$$

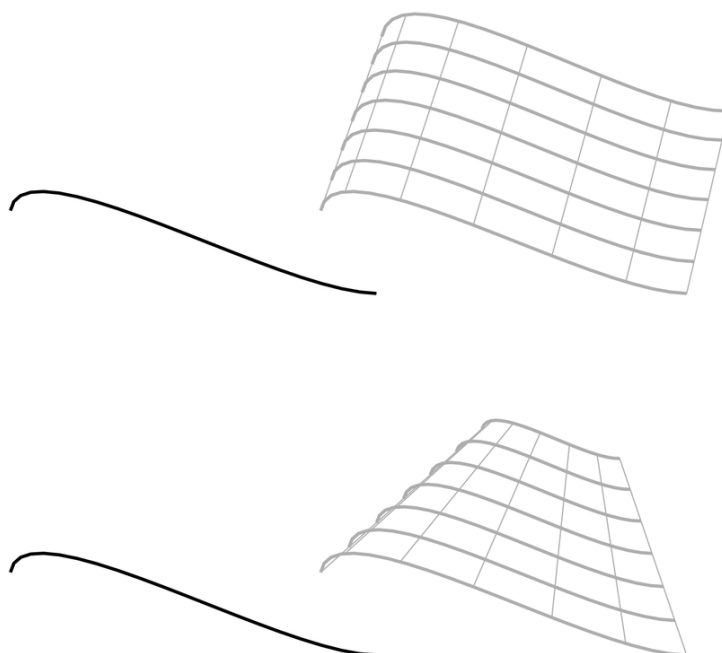
W podobny sposób określa się **wymierne krzywe B-sklejane**, a także **wymierne płaty powierzchni Béziera i B-sklejane**.

Wymierne krzywe i powierzchnie sklejane znane są pod nazwą **NURBS**, która jest skrótem angielskiego określenia *non-uniform rational B-splines*. Słowa *non-uniform* (czyli nierównomierne) dotyczą dopuszczalnych ciągów węzłów w definicji takiej krzywej — węzły te nie muszą być równoodległe.

6.5. Modelowanie powierzchni i brył

6.5.1. Zakreślanie

Zakreślanie (ang. *sweeping*) jest sposobem określania powierzchni, w najprostszym przypadku za pomocą krzywej, tzw. **przekroju** i odcinka, tzw. **prowadnicy**. Zasada jest przedstawiona na rysunku.



Rysunek 6.18. Zakreślanie.

Konstrukcję tę można uogólniać na wiele sposobów; jednym z nich jest przekształcanie przekroju podczas „przesuwania” go wzdłuż prowadnicy. Jeśli przekształcenie to jest jednokładnością o ustalonym środku, to zamiast powierzchni walcowej otrzymujemy powierzchnię stożkową. Dalsze możliwości uogólnienia tej konstrukcji są następujące:

- Prowadnica może być krzywą;
- Każdemu punktowi prowadnicy odpowiada inne przekształcenie afiniczne, któremu będzie poddany przekrój;
- Można też dopuścić zmiany kształtu przekroju podczas „przesuwania” go.

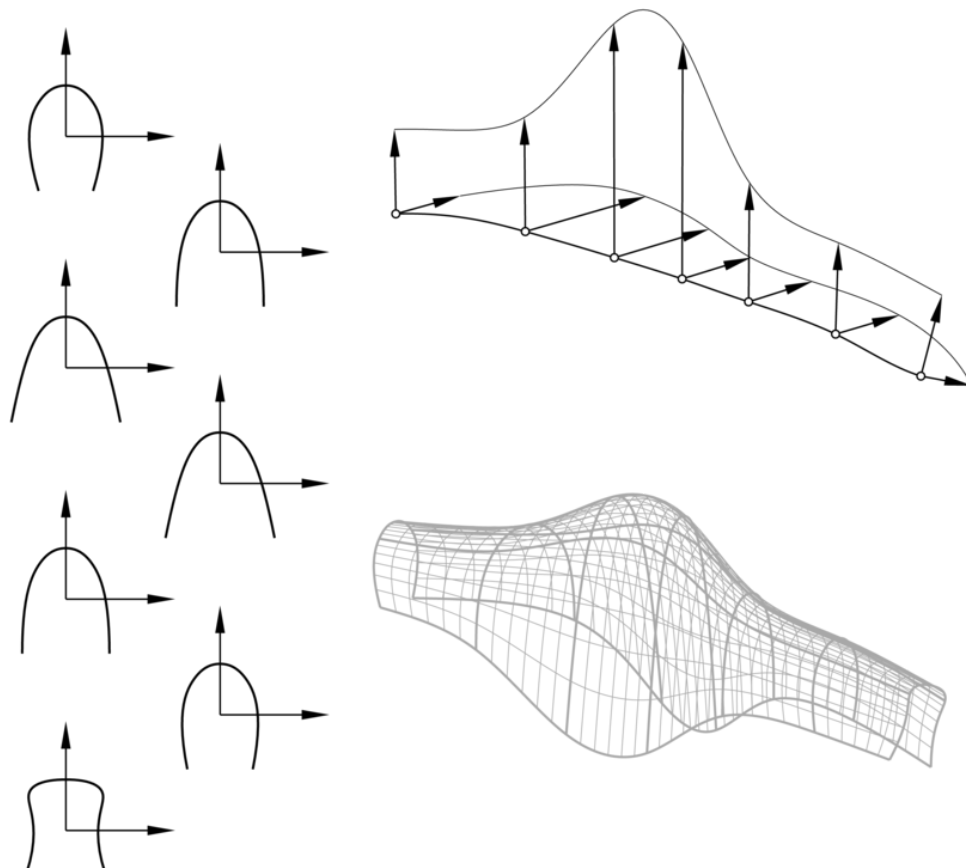
Dopuszczenie wszystkich tych możliwości wymaga opisu powierzchni zakreślanej wzorem

$$\mathbf{s}(u, v) = \mathbf{p}(u) + \mathbf{x}_2(u)x_q(u, v) + \mathbf{x}_3(u)y_q(u, v) + \mathbf{x}_1(u)z_q(u, v).$$

W tym wzorze występują krzywe:

- prowadnica, $\mathbf{p}$,
- tzw. kierownice, $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$,
- przekrój, $\mathbf{q}$, który jest w najogólniejszym przypadku jednoparametrową rodziną krzywych.

Zmienna v jest parametrem przekroju; zmienna u , która jest parametrem prowadnicy i kierunku, jest też parametrem rodziny krzywych opisujących przekrój. We wzorze są widoczne funkcje x_q , y_q i z_q , które opisują współrzędne punktu przekroju.



Rysunek 6.19. Uogólniona konstrukcja powierzchni zakreślanej.

Zauważmy, że dla ustalonego u wzór opisujący powierzchnię zakreślaną opisuje przekształcenie afiniczne, którego część liniowa jest określona przez macierz $[\mathbf{x}_2(u), \mathbf{x}_3(u), \mathbf{x}_1(u)]$, a wektor przesunięcia jest równy $\mathbf{p}(u)$.

Jeśli wszystkie krzywe są krzywymi B-sklejany i przekrój nie zależy od u , to mając łamane kontrolne tych krzywych można dość łatwo skonstruować siatkę kontrolną powierzchni zakreślonej. W przypadku zmieniającego się przekroju jest to trudniejsze i dlatego zwykle konstruuje się przybliżenie takiej powierzchni. Są dwa sposoby otrzymania takiego przybliżenia:

1. Tablicujemy wzór określający powierzchnię; otrzymujemy w ten sposób prostokątną tablicę punktów na powierzchni. Na podstawie tych punktów generujemy trójkąty, które jeśli jest ich dość dużo, przybliżają powierzchnię dostatecznie dokładnie.
2. Konstruujemy powierzchnię rozpinaną. W tym celu należy wyznaczyć krzywe B-sklejane opisujące przekrój dla wybranych wartości parametru u , a następnie wyznaczyć powierzchnię interpolacyjną dla tych krzywych.

6.5.2. Powierzchnie rozpinane

Przypuśćmy, że dane są liczby (węzły) $u_n, \dots, u_{N-n}$ i krzywe B-sklejane $\mathbf{x}_i(v)$ dla $i = n, \dots, N-n$. **Powierzchnia rozpinana** s spełnia warunek $s(u_i, v) = \mathbf{x}_i(v)$ dla każdego v .

Założymy, że stopień i ciąg węzłów użyty do określenia wszystkich krzywych $\mathbf{x}_i$ jest identyczny. Punktem wyjścia do konstrukcji reprezentacji B-sklejanej powierzchni rozpinanej jest konstrukcja B-sklejanej krzywej interpolacyjnej, określonej przez podanie punktów $\mathbf{x}_i$. Mamy obliczyć punkty kontrolne krzywej $\mathbf{s}$, takiej że $\mathbf{s}(u_i) = \mathbf{x}_i$, $i = n, \dots, N - n$.

Konstrukcja dla $n = 3$ wygląda następująco: przyjmujemy $u_1 = u_2 = u_3$, $u_{N-3} = u_{N-2} = u_{N-1}$. Punkty krzywej $\mathbf{s}$ otrzymamy, rozwiązując układ równań liniowych

$$\begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & N_1^3(u_4) & N_2^3(u_4) & N_3^3(u_4) & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & N_{N-7}^3(u_{N-4}) & N_{N-6}^3(u_{N-4}) & N_{N-5}^3(u_{N-4}) & & \\ & & & & & 1 & & \\ & & & & & & 1 & \end{bmatrix} \begin{bmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \mathbf{d}_2 \\ \vdots \\ \mathbf{d}_{N-6} \\ \mathbf{d}_{N-5} \\ \mathbf{d}_{N-4} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_3 \\ \mathbf{d}_1 \\ \mathbf{x}_4 \\ \vdots \\ \mathbf{x}_{N-4} \\ \mathbf{d}_{N-5} \\ \mathbf{x}_{N-3} \end{bmatrix}.$$

Punkty $\mathbf{d}_1$ i $\mathbf{d}_{N-5}$ można wybrać dowolnie (podanie tych punktów określa tzw. **warunki brzegowe** krzywej). Współczynniki macierzy można obliczyć ze wzorów (otrzymanych na podstawie wzoru Mansfielda-de Boora-Coxa (6.1) i (6.2))

$$\begin{aligned} N_{k-3}^3(u_k) &= \frac{(u_{k+1} - u_k)^2}{(u_{k+1} - u_{k-2})(u_{k+1} - u_{k-1})}, \\ N_{k-2}^3(u_k) &= \frac{u_k - u_{k-2}}{u_{k+1} - u_{k-2}} \frac{u_{k+1} - u_k}{u_{k+1} - u_{k-1}} + \frac{u_{k+2} - u_k}{u_{k+2} - u_{k-1}} \frac{u_k - u_{k-1}}{u_{k+1} - u_{k-1}}, \\ N_{k-1}^3(u_k) &= \frac{(u_k - u_{k-1})^2}{(u_{k+2} - u_{k-1})(u_{k+1} - u_{k-1})}. \end{aligned}$$

Aby skonstruować B-sklejaną powierzchnię interpolacyjną (powierzchnię rozpinaną), wystarczy zamiast punktów $\mathbf{x}_i$ podstawić „punkty” — łamane kontrolne krzywych $\mathbf{x}_i$. Liczba współrzędnych każdego takiego punktu jest równa 3 razy liczba punktów kontrolnych łamanej. Obliczone „punkty” $\mathbf{d}_i$ w przestrzeni o tym samym wymiarze w podobny sposób reprezentują kolumny siatki kontrolnej powierzchni rozpinanej. Wybór kolumn numer 1 i $N - 5$, określających warunki brzegowe, jest dowolny.

6.5.3. Powierzchnie zadane w postaci niejawnej

Najczęściej stosowane w grafice powierzchnie zadane w postaci niejawnej to powierzchnie algebraiczne i kawałkami algebraiczne, czyli zbiory miejsc zerowych wielomianów trzech zmiennych. Najbardziej znana reprezentacja płaszczyzny, za pomocą dowolnego punktu $[x_0, y_0, z_0]^T$ i wektora normalnego $[a, b, c]$, jest w istocie niejawna:

$$\{[x, y, z]^T : a(x - x_0) + b(y - y_0) + c(z - z_0) = 0\}.$$

Podobnie, współrzędne środka $[x_0, y_0, z_0]$ i promień r są współczynnikami równania sfery:

$$\{[x, y, z]^T : (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0\}.$$

Dość często spotyka się w praktyce powierzchnie wyższego stopnia (np. torus, powierzchnia stopnia 4), ale rzadko stopień ten jest większy niż kilka.

Określenie prostokątnego płata Béziera opiera się na pojęciu iloczynu tensorowego dwóch przestrzeni funkcji jednej zmiennej. Możemy rozpatrywać iloczyny tensorowe większej liczby

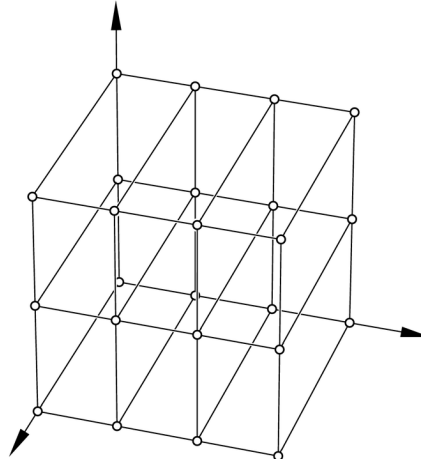
przestrzeni. Jeśli są to przestrzenie wielomianów stopnia n, m, l , to bazą ich iloczynu tensorowego jest np. baza

$$\{B_i^n(u)B_j^m(v)B_k^l(w) : i = 0, \dots, n, j = 0, \dots, m, k = 0, \dots, l\}.$$

Dowolny wielomian trzech zmiennych, stopnia (n, m, l) , można przedstawić w tej bazie. W kostce $[0, 1]^3$ określmy punkty $\mathbf{p}_{ijk} = [i/n, j/m, k/l]^T$. Określmy wielomian

$$a(x, y, z) = \sum_{i=0}^n \sum_{j=0}^m \sum_{k=0}^l a_{ijk} B_i^n(x) B_j^m(y) B_k^l(z).$$

Współczynnik a_{ijk} przyporządkujemy punktowi $\mathbf{p}_{ijk}$. Wielomian $B_i^n(x)B_j^m(y)B_k^l(z)$ w tym punkcie kostki przyjmuje maksymalną wartość.



Rysunek 6.20. Siatka reprezentacji wielomianu trzech zmiennych.

Powierzchnia określona za pomocą takiego wielomianu trzech zmiennych jest zbiorem jego miejsc zerowych wewnątrz kostki. Wyznaczanie punktów takiej powierzchni lub jej przybliżenia (np. za pomocą trójkątów) wymaga obliczania wartości funkcji a , najczęściej podczas rozwiązywania równania nieliniowego, które powstaje przez podstawienie parametrycznej reprezentacji pewnej prostej:

$$a(x_0 + tx_v, y_0 + ty_v, z_0 + tz_v) = 0.$$

Znając rozwiązanie t tego równania możemy obliczyć współrzędne punktu na powierzchni na podstawie reprezentacji prostej. Metody, w których jest stosowane to podejście, będą omówione później; jedną z nich jest **śledzenie promieni**, a drugą to **metoda maszerujących sześciątów**. Zwróćmy uwagę, że na ogół nie warto wyznaczać współczynników wielomianu f zmiennej t , który występuje po lewej stronie równania. Stopień tego wielomianu jest równy $n + m + l$, czyli jest duży. Zamiast tego, wygodniej jest obliczać wartość wielomianu a sposobem podobnym do wyznaczania punktu płata Béziera. Mamy

$$a(x, y, z) = \sum_{i=0}^n \underbrace{\left(\sum_{j=0}^m \underbrace{\left(\sum_{k=0}^l a_{ijk} B_k^l(z) \right)}_{b_{ij}} B_j^m(y) \right)}_{c_i} B_i^n(x),$$

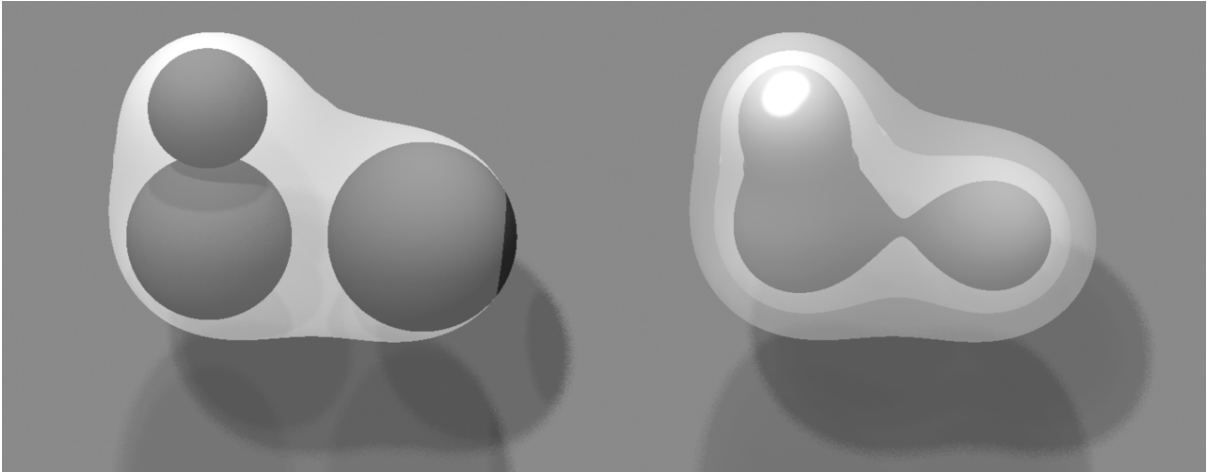
co umożliwia zastosowanie algorytmu de Casteljau albo schematu Hornera.

Algorytm de Casteljau umożliwia podział kostki, w której jest określona powierzchnia, na prostopadłościany. Możemy to wykorzystać do otrzymania obszaru o dowolnie małej objętości, w którym znajduje się powierzchnia. Wystarczy dzielić rekurencyjnie kostkę i odrzucać te jej fragmenty, w których współczynniki lokalnej reprezentacji wielomianu a mają stały znak. To jest oczywiście zastosowanie własności otoczki wypukłej opisanej tu reprezentacji wielomianu trzech zmiennych.

W grafice komputerowej bardzo popularne są powierzchnie określane słowem *blob* (to angielskie słowo nie doczekało się polskiego odpowiednika). Powierzchnia taka może być zbiorem miejsc zerowych funkcji f określonej wzorami

$$\begin{aligned}\Phi(x, y, z) &= e^{(1-x^2-y^2-z^2)}, \\ f_i(x, y, z) &= \Phi((x-x_i)/r_i, (y-y_i)/r_i, (z-z_i)/r_i), \\ f(x, y, z) &= \sum_{i=1}^n s_i f_i(x, y, z) - c.\end{aligned}$$

Zamiast funkcji przestępnej e^x często do określenia blobu wykorzystuje się też wielomiany.



Rysunek 6.21. Blob.

Aby określić taką powierzchnię, należy określić tzw. szkielet, czyli zbiór punktów $[x_i, y_i, z_i]^T$ oraz parametry r_i i s_i dla każdego z tych punktów i współczynnik c . Zbiór rozwiązań równania $s_i f_i(x, y, z) - c = 0$ jest wprawdzie sferą, ale jeśli punktów szkieletu jest więcej niż 1, to otrzymujemy gładkie połączenia nieco zdeformowanych sfer. Zwiększanie parametru r_i powoduje powiększanie sfery (lub „rozdmuchiwanie” odpowiedniego fragmentu powierzchni), zaś parametr s_i odpowiada za długość gradientu funkcji f_i ; im większy, tym „sztywniejsza” jest powierzchnia ze względu na zmiany parametru c . Na lewym rysunku 6.21 jest pokazana taka powierzchnia razem z trzema kulami, których środki tworzą jej szkielet, z prawej zaś strony są trzy takie powierzchnie, które różnią się tylko wartościami parametru c .

Powierzchnie zadane w sposób niejawni mają to do siebie, że funkcja występująca w definicji umożliwia rozróżnienie trzech podzbiorów przestrzeni: powierzchnia jest zbiorem miejsc zerowych. W punktach należących do wnętrza bryły, której brzegiem jest rozpatrywana powierzchnia, funkcja ma wartości dodatnie, a punkty na zewnątrz odpowiadają ujemnym wartościom funkcji (można też traktować znaki funkcji odwrotnie). Ta informacja jest niesłychanie ważna podczas znajdowania punktów powierzchni, a także w konstrukcyjnej geometrii brył i dlatego nie

należy uważać funkcji $f^2(x, y, z)$ albo $|f(x, y, z)|$ za pełnowartościową reprezentację powierzchni, mimo że funkcje te mają zbiór miejsc zerowych taki sam jak funkcja f .

7. Reprezentacje scen trójwymiarowych

7.1. Prymitywy i obiekty złożone

Obiekty trójwymiarowe dzielimy na

- obiekty z zamkniętą objętością (bryły),
- obiekty z otwartą objętością (powierzchnie, krzywe),
- są też obiekty specjalne, np. chmury, płomienie lub futro, reprezentowane przez pola skalarne lub wektorowe, których wygląd na obrazach powinien ukazywać ich budowę przestrzenną.

Obiekt z zamkniętą objętością jest zwykle wizualizowany przez narysowanie obrazu powierzchni stanowiącej jego brzeg. Dlatego często są one reprezentowane za pomocą powierzchni brzegowej (spotykany skrót B-rep oznacza reprezentację bryły za pomocą brzegu). Od obiektów z otwartą objętością bryły odróżniają się tym, że nie można „obejrzeć” obu stron powierzchni takiej figury bez przeniknięcia przez nią.

Oprócz podziału podanego wyżej, obiekty można klasyfikować według sposobu ich określenia i reprezentowania. Obiekt, który jest określony za pomocą innych obiektów (np. stanowiących jego części) jest to tzw. **obiekt złożony**. Obiekt określony bez odwoływania się do innych, prostszych obiektów to tzw. **prymityw**¹. Te określenia nie są zbyt precyzyjne, ponieważ uznanie obiektu za złożony lub za prymityw zależy od zastosowania, a nawet kontekstu. Na przykład za prymityw możemy uznać wielościan opisany za pomocą list wierzchołków, krawędzi i ścian, jeśli figur tych nie rozpatrujemy w oderwaniu od wielościanu. Natomiast z punktu widzenia OpenGL-a prymitywem geometrycznym jest tylko punkt, odcinek lub wielokąt wypukły, który można narysować bezpośrednio.

Jest oczywiste, że reprezentacja obiektu może być inna dla potrzeb wykonywania obrazu niż dla potrzeb przetwarzania danych. Na przykład wyświetlenie obrazu powierzchni złożonej z wielu płatów B-sklejanych polega na wyznaczeniu dostatecznie dużej liczby dostatecznie małych trójkątów, które są następnie rzutowane, rasteryzowane i teksturowane. W tym przypadku odpowiednią reprezentacją powierzchni jest zbiór trójkątów. Natomiast interakcyjne kształtowanie takiej powierzchni (np. w systemie CAD) polega na dobieraniu węzłów i punktów kontrolnych, przy czym spełnienie warunków nałożonych na płaty (takich jak gładkość połączeń na wspólnych brzegach) może być zapewnione przez użycie specjalnej struktury danych, której najprostszą częścią są tablice węzłów i punktów kontrolnych płatów. Zaryzykowałbym stwierdzenie, że zaprojektowanie takiej struktury danych jest najtrudniejszym zadaniem do rozwiązania podczas tworzenia programów.

7.2. Reprezentowanie brył wielościennych

Przypomnijmy, że

- Brzeg bryły wielościennej jest zbiorem płaskich wielokątów, tzw. **ścian**.
- Brzeg ściany jest zbiorem odcinków, tzw. **krawędzi**.
- Brzeg krawędzi składa się z dwóch punktów, tzw. **wierzchołków**.

¹ W tym określeniu nie ma niczego pejoratywnego.

Widać tu pewną hierarchię, oraz analogie między poszczególnymi poziomami tej hierarchii. Reprezentacja bryły może składać się z tablicy punktów, krawędzi i ścian; każda z tych tablic może zawierać odnośniki do pozostałych tablic.

Reprezentacja punktu w przestrzeni trójwymiarowej zawiera oczywiście 3 współrzędne kartezjańskie lub 4 jednorodne. Może ona także zawierać listę (indeksów do tablicy) krawędzi, których końcem jest dany punkt oraz listę (indeksów do tablicy) ścian, dla których dany punkt jest wierzchołkiem. Dodatkowymi informacjami związanymi z punktem są np. **wektor normalny**, potrzebny w obliczeniach oświetlenia i w cieniowaniu powierzchni, współrzędne tego punktu w przestrzeni tekstury i inne.

Reprezentacja krawędzi najczęściej zawiera wskaźniki (indeksy) punktów końcowych i wskaźniki ścian, których dana krawędź jest wspólnym brzegiem. Reprezentacja krawędzi może też zawierać informację, czy na obrazie krawędź ta ma być wygładzona (jeśli gładka powierzchnia krzywoliniowa jest przybliżana przez powierzchnię złożoną z trójkątów, to raczej tego chcemy).

Zamiast krawędzi wygodne może być używanie **półkrawędzi**: krawędź wspólna dwóch ścian jest reprezentowana za pomocą dwóch półkrawędzi, z których każda jest częścią reprezentacji jednej z tych ścian. Każda półkrawędź ma wskaźnik do drugiej półkrawędzi. Zaletą takiej reprezentacji jest uproszczenie implementacji różnych algorytmów.

Reprezentacja ściany może zawierać listę krawędzi, albo listę list krawędzi (ta ostatnia reprezentacja przydaje się, jeśli ściana nie jest spójna lub jednospójna). Może też zawierać listę wierzchołków, a także obie listy: krawędzi i wierzchołków. Aby przyspieszyć wyświetlanie ściany, można ją podzielić na trójkąty (albo dokonać innego wygodnego podziału; np. biblioteka GL w standardzie OpenGL, której procedury bezpośrednio współpracują ze sprzętem, umożliwia wyświetlanie tylko wielokątów wypukłych). Lista tych trójkątów w reprezentacji ściany może znacznie przyspieszyć wykonywanie obrazu.

Zyski czasowe z utworzenia takich list trójkątów należy rozpatrywać w dwojakim kontekście. Po pierwsze, w programie interakcyjnym (takim jak system CAD albo gra komputerowa), ta sama ściana bywa wyświetlana wielokrotnie, np. z różnych punktów położenia obserwatora. Biblioteka GLU w OpenGL-u umożliwia wyświetlanie wielokątów niewypukłych, jednak wtedy ta sama praca (dzielenia wielokąta na wypukłe kawałki) jest wykonywana wielokrotnie². Jeśli takich ścian jest dużo, to częstość odświeżania obrazu (podczas animacji w czasie rzeczywistym) może zmniejszyć się, znacznie obniżając komfort pracy użytkownika programu. Po drugie, programy wykonujące obraz fotorealistyczny, np. metodą śledzenia promieni lub bilansu energetycznego, muszą obliczać przecięcia promieni (pewnych półprostych w przestrzeni) ze ścianami. Zastąpienie skomplikowanych ścian odpowiednimi listami trójkątów może przyspieszyć działanie programu o kilka procent, co przy czasie obliczeń rzędu minut lub nawet godzin też ma znaczenie.

Dodatkowe możliwe atrybuty ściany, to wektor normalny (zwróćmy uwagę, że wektor normalny może być też związany z wierzchołkami ściany), reprezentacja układu odniesienia współrzędnych tekstury, identyfikator rodzaju powierzchni i jej własności optyczne (przezroczystość, współczynnik załamania światła, kolor itd.).

Jak widać, taka reprezentacja zawiera sporo informacji redundantnej. Jest to spowodowane przez

- potrzebę używania takiej informacji bez wielokrotnego wyznaczania jej na podstawie minimalnego zbioru danych wystarczającego do tego,
- potrzebę sprawdzania poprawności; na przykład, dla bryły, której powierzchnia jest homeomorficzna ze sferą, prosty test poprawności polega na sprawdzeniu wzoru Eulera:

$$V - E + F = 2,$$

² Chyba, że posługujemy się listami obrazowymi OpenGL-a

w którym V oznacza liczbę wierzchołków, E — krawędzi, a F — ścian. Informacja redundantna może zarówno ułatwić, jak i utrudnić sprawdzanie poprawności danych, które jest konieczne zwłaszcza podczas wymiany danych między różnymi programami.

Przykład: Rozwiązanie przyjęte w moim pakiecie BStools reprezentuje siatkę, która może opisywać brzeg wielościanu, a także powierzchnię siatkową (zobacz p. ??) za pomocą sześciu tablic. Zdefiniowane są następujące struktury w C:

```
typedef struct {
    char degree, tag;
    int firsthalfedge;
} BSMfacet, BSMvertex;

typedef struct {
    int v0, v1;
    int facetnum;
    int otherhalf;
} BSMhalfedge;
```

Pierwsza struktura reprezentuje ściany i wierzchołki, dlatego ma dwie nazwy. Mamy dwie tablice takich struktur, elementy jednej reprezentują ściany, a drugiej wierzchołki. Pole `degree`, tj. stopień, oznacza liczbę (pół)krawędzi (i wierzchołków) ściany albo liczbę półkrawędzi wychodzących z danego wierzchołka. W dodatkowych dwóch tablicach liczb całkowitych przechowuje się indeksy półkrawędzi — dzięki takiemu rozwiązaniu nie ma ciasnego ograniczenia stopni ścian i wierzchołków i nie marnuje się miejsca. Długość tych tablic jest równa liczbie półkrawędzi, która jest sumą stopni wszystkich wierzchołków i sumą stopni wszystkich ścian. Pole `firsthalfedge` struktury `BSMfacet` lub `BSMvertex` oznacza indeks pierwszej półkrawędzi dla danej ściany lub wierzchołka. Pole `tag` jest używane przez różne procedury przetwarzania siatek.

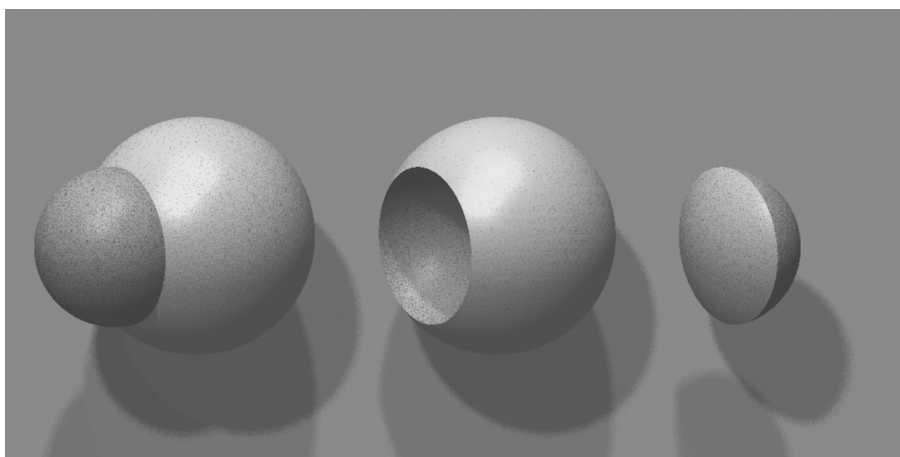
Piąta tablica, struktur `BSMhalfedge`, reprezentuje półkrawędzie. Półkrawędź jest opisana przez podanie indeksów wierzchołków `v0` i `v1`, indeksu ściany `facetnum`, do której brzegu należy półkrawędź i indeksu `otherhalf` drugiej półkrawędzi o tych samych wierzchołkach końcowych (ale o przeciwnej orientacji). Jeśli krawędź jest brzegowa, to pole `otherhalf` ma wartość `-1`.

Szósta tablica służy do przechowywania współrzędnych poszczególnych wierzchołków. Dzięki rozdzieleniu informacji topologicznej na temat wierzchołka od jego współrzędnych, te same struktury mogą opisywać siatki płaskie i przestrzenne, a także można podawać współrzędne jednorodne. W dodatkowych tablicach, zależnie od potrzeb, można przechowywać wektory normalne dla ścian lub wierzchołków, współrzędne tekstury, kolory itd.

Opisana wyżej reprezentacja wielościanu stanowi pewnego rodzaju „program maksimum”. Bywa ona potrzebna w niektórych algorytmach widoczności i w konstrukcyjnej geometrii brył, ale często wiele jej elementów jest zbędnych. Dla kontrastu, przypomnijmy „minimalną” reprezentację powierzchni wielościennej, stosowaną w komunikacji między programem a sprzętem, jaką jest **taśma trójkątowa**, czyli ciąg punktów, w którym każde trzy kolejne punkty są wierzchołkami trójkąta.

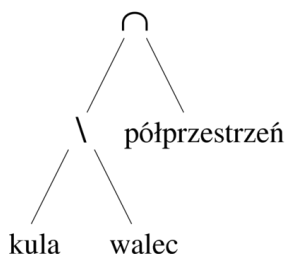
7.3. Konstrukcyjna geometria brył

Obiekty, które definiujemy „w jednym kawałku”, są nazywane prymitywami; mogą to być np. wielościany, albo bryły, których brzegiem jest powierzchnia algebraiczna. Z tych obiektów można budować obiekty bardziej złożone, przy czym zwykle dodawanie obiektów nie wystarczy. Czasem interesuje nas bryła, która jest różnicą (mnożościową) brył danych; jeśli chcemy np. pograć w bilard, to trzeba usunąć kawałki stołu (wywiercić otwory na bile).



Rysunek 7.1. Suma, różnica i przecięcie brył.

W zasadzie **konstrukcyjna geometria brył** (ang. *constructive solid geometry*, *CSG*³) jest zastosowaniem algebry zbiorów. Mając do dyspozycji prymitywy i bryły z nich zbudowane, można określić ich sumę, przecięcie, różnicę i dopełnienie. Bryła o skomplikowanym kształcie jest opisana przez pewne wyrażenie mnogościowe, które wygodnie jest przedstawić w postaci drzewa (tzw. **drzewa CSG**).

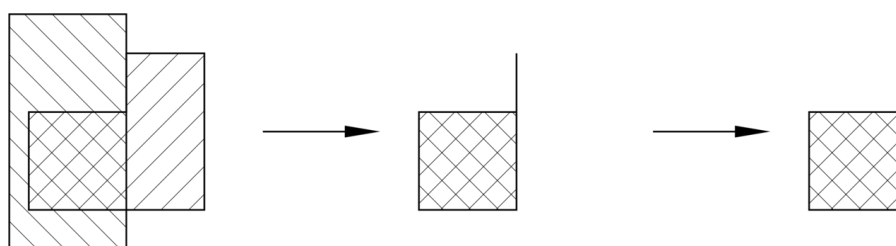


Rysunek 7.2. Przykład drzewa i bryły CSG.

Słowa „w zasadzie” oznaczają pewne dodatkowe przekształcenie, jakiemu jest poddawany wynik każdej operacji mnogościowej. Jest nim tzw. **regularyzacja**. Zbiór regularny jest domknięciem swojego wnętrza. Brzeg takiego zbioru należy do niego i w otoczeniu każdego punktu brzegu leżą punkty z wnętrza — a zatem np. każda ściana wielościanu regularnego może być widoczna z jednej strony (czyli, powtarzając wcześniejsze uwagi, bryły CSG są obiektami o zamkniętej objętości).

Niezależnie od algorytmów wyznaczania reprezentacji brył CSG, określenie takich brył jest źródłem kłopotów, spowodowanych błędami zaokrągleń (małe zaburzenia argumentów mogą istotnie zmienić wynik operacji). Ponadto nawet dla bryły wielościennej znalezienie reprezentacji brzegu jest skomplikowane z powodu mnóstwa kłopotliwych przypadków szczególnych, które algorytm rozwiązujący to zadanie musi uwzględniać. Mimo to często wyznacza się reprezenta-

³ Nie będziemy wprowadzali skrótu polskiej nazwy.



Rysunek 7.3. Regularyzacja przecięcia figur.

cję wielościennych brył CSG, ponieważ jest to jedyna reprezentacja, która może posłużyć do utworzenia obrazu za pomocą wydajnych i powszechnie dostępnych algorytmów widoczności. W przypadku brył ograniczonych powierzchniami algebraicznymi albo stosuje się przybliżenie tych powierzchni złożone z trójkątów (i wtedy można stosować algorytmy tworzenia obrazu odpowiednie dla wielościanów), albo też jedyną reprezentacją bryły CSG jest samo drzewo CSG; niektóre algorytmy widoczności (algorytm śledzenia promieni i odpowiednie warianty algorytmu przeglądania liniami poziomymi) są w stanie utworzyć obraz bryły CSG bez wyznaczania jawnej reprezentacji brzegu tej bryły.

7.3.1. Wyznaczanie przecięcia wielościanów

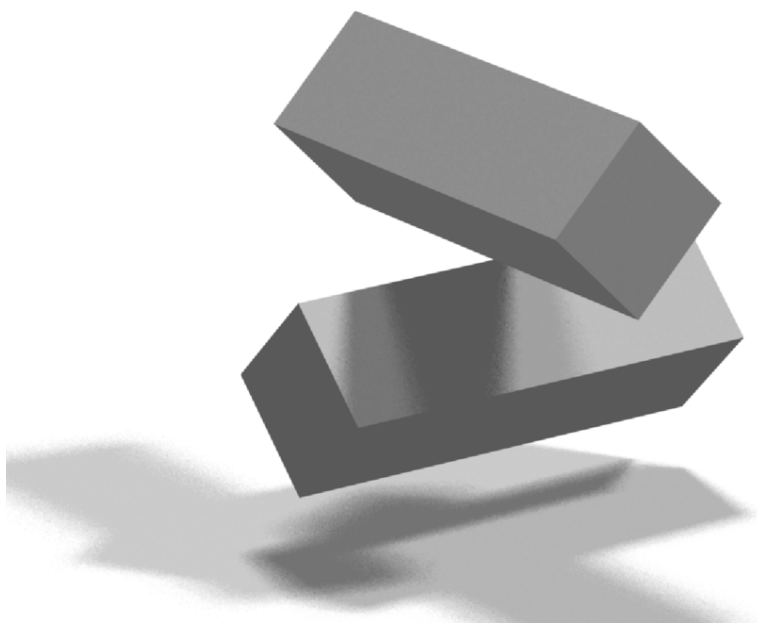
Mając dwa wielościany reprezentowane za pomocą ich brzegów (czyli listy wierzchołków, krawędzi i ścian), możemy wyznaczyć brzeg wielościanu, który jest ich regularyzowanym przecięciem. Przypuśćmy, że reprezentacja ma tę własność, że dla każdej ściany wektor normalny jest zorientowany „na zewnątrz” bryły. Aby wyznaczyć jej dopełnienie, wystarczy zmienić zwrot wektora normalnego każdej ściany na przeciwny. Operacje wyznaczania przecięcia i dopełnienia umożliwiają otrzymanie sumy i różnicy brył.

Aby znaleźć brzeg wielościanu, który jest regularyzowanym przecięciem dwóch regularnych wielościanów o znanych brzegach, można wykonać algorytm naszkicowany niżej:

1. Dla każdej pary ścian, z których jedna jest częścią brzegu jednej, a druga częścią brzegu drugiej bryły, wyznaczamy przecięcia tych ścian. Przecięcie to może być zbiorem pustym, punktem, odcinkiem lub wielokątem. Jeśli przecięcie ścian jest wielokątem, to znajdujemy odcinki będące przecięciami krawędzi jednej ściany z drugą (to wymaga obcinania odcinka do wielokąta w ogólności niewypukłego).
 2. Odcinkami wyznaczonymi w poprzednim kroku dzielimy ściany wielokątów na spójne fragmenty. Wnętrze każdego takiego fragmentu ściany bryły w całości leży wewnątrz drugiej bryły, na zewnątrz, lub na jej brzegu. Określamy graf sąsiedztwa fragmentów ścian, którego wierzchołkami są fragmenty, a krawędziami są krawędzie wielościanów lub odcinki otrzymane w pierwszym kroku.
 3. Wybieramy nieodwiedzony fragment ściany (wierzchołek grafu sąsiedztwa ścian) i sprawdzamy (np. na podstawie reguły parzystości), czy należy on do brzegu przecięcia brył.
 4. Metodą DFS lub (lepiej) BFS przeszukujemy graf sąsiedztwa ścian, przy czym na krawędzi przecięcia zatrzymujemy przeszukiwanie. Jeśli fragment ściany, od którego zaczęliśmy przeszukiwanie, należy do przecięcia, to wszystkie odwiedzone fragmenty też. Fragmenty takie dołączamy do reprezentacji przecięcia brył, a pozostałe (nie należące do przecięcia) zaznaczamy jako odwiedzone.
- Kroki 3 i 4 powtarzamy tak długo, aż zostaną odwiedzone wszystkie fragmenty ścian brył wyjściowych.

5. Wynikiem przeszukiwania jest reprezentacja brzegu przecięcia brył w postaci grafu sąsiedztwa ścian, w którym brakuje krawędzi przecięcia ścian znalezionych w pierwszym kroku). Krawędzie te dołączamy teraz do grafu.

Dzielenie ścian na fragmenty można uzupełnić o triangulację lub o podział na fragmenty wypukłe, dzięki czemu będziemy mieli dane o postaci ułatwiające dalsze ich przetwarzanie (np. wykonywanie dalszych operacji CSG, albo wyświetlanie). Procedura opisana wyżej jest dość prosta (choć niektóre jej fragmenty są nietrywialne), jednak największy kłopot polega na uodpornieniu jej na szczególne przypadki danych, np. taki jak na rysunku 7.4.



Rysunek 7.4. Prymitywy sprawiające kłopot w konstrukcyjnej geometrii brył.

Opisana wyżej procedura jest potrzebna, jeśli obrazy brył CSG mamy otrzymywać za pomocą algorytmów widoczności, które wymagają jawnej reprezentacji brzegu (np. w postaci listy wielokątów do wyświetlenia). Takim algorytmem jest np. algorytm z buforem głębokości. Istnieją jednak algorytmy widoczności dopuszczające opis brył CSG w postaci reprezentacji prymitywów i drzew CSG (np. można zrealizować w ten sposób śledzenie promieni). Co więcej, taki opis jest jedynym możliwym „dokładnym” opisem wielu brył CSG utworzonych z prymitywów będących bryłami krzywoliniowymi. Aby wyświetlić taką bryłę za pomocą algorytmu z buforem głębokości, trzeba najpierw znaleźć wielościany będące przybliżeniami prymitywów, a następnie wielościany reprezentujące bryłę CSG.

7.4. Drzewa i grafy scen

Ważną cechą scen trójwymiarowych, która powinna być uwzględniona w reprezentacji, jest **hierarchia obiektów**. Zwykle nie określa się sceny jako liniowej listy prymitywów do wyświetlenia, ale raczej komponuje się ją z obiektów, z których każdy jest złożony z jeszcze prostszych obiektów itd. Prymitywy są na samym dole tej hierarchii.

Opisana wyżej hierarchia może być odzwierciedlona za pomocą **drzewa hierarchii sceny**. Korzeń drzewa reprezentuje całą scenę. Każdy wierzchołek drzewa reprezentuje pewien

prymityw lub wskazuje wierzchołki będące korzeniami swoich poddrzew (np. zawiera wskaźnik początku listy tych wierzchołków). Ponadto każdy wierzchołek zawiera dodatkowe atrybuty, które są elementami określenia sceny, albo są potrzebne do usprawnienia procesu wykonywania obrazu. Atrybuty mogą być następujące:

Przekształcenie geometryczne. Opisuje ono przejście między układem współrzędnych związanym z wierzchołkiem drzewa położonym wyżej w hierarchii, a układem, w którym jest określony obiekt reprezentowany przez wierzchołek dany. Na przykład jeśli reprezentowana scena jest umeblowanym pomieszczeniem, to każdy mebel jest opisany w pewnym swoim układzie, a jego przekształcenie określa ustawienie go w pomieszczeniu. Wazon stojący na stole lub książki na półce mają przekształcenia opisujące ich ustawienie względem odpowiedniego mebla.

Najczęściej przekształcenie obiektu jest afiniczne lub rzutowe, a jego reprezentacją w wierzchołku drzewa hierarchii jest odpowiednia macierz.

Bryła otaczająca. Może nią być prostopadłościan o krawędziach równoległych do osi lokalnego układu współrzędnych (wówczas wystarczy podać sześć liczb; zwróćmy uwagę, że zwykle łatwo jest je znaleźć, np. dla powierzchni Béziera wystarczy znaleźć najmniejsze i największe współrzędne punktów kontrolnych). Bryła taka umożliwia pominięcie czasochłonnego przetwarzania poddrzewa, jeśli program wykryje, że jest ona w całości niewidoczna (np. zasłonięta przez inny obiekt). Inne zastosowanie brył otaczających, które może znacznie skrócić czas obliczeń, to wykrywanie kolizji obiektów w animacji.

Uproszczona reprezentacja obiektu. Jeśli obiekt na obrazie jest mały (co można zbadać wyznaczając rzut bryły otaczającej), to nie należy tracić czasu na rysowanie go ze wszystkimi szczegółami. Można narysować nawet bardzo uproszczony obraz, np. odpowiednio poteksturowany prostopadłościan zamiast domu, samochodu lub drzewa w rozległym pejzażu, co oczywiście zajmie mniej czasu niż rysowanie wszystkich dachówek, wentyli w kołach lub liści, które trzeba by narysować na obrazie danego obiektu widzianego z bliska.

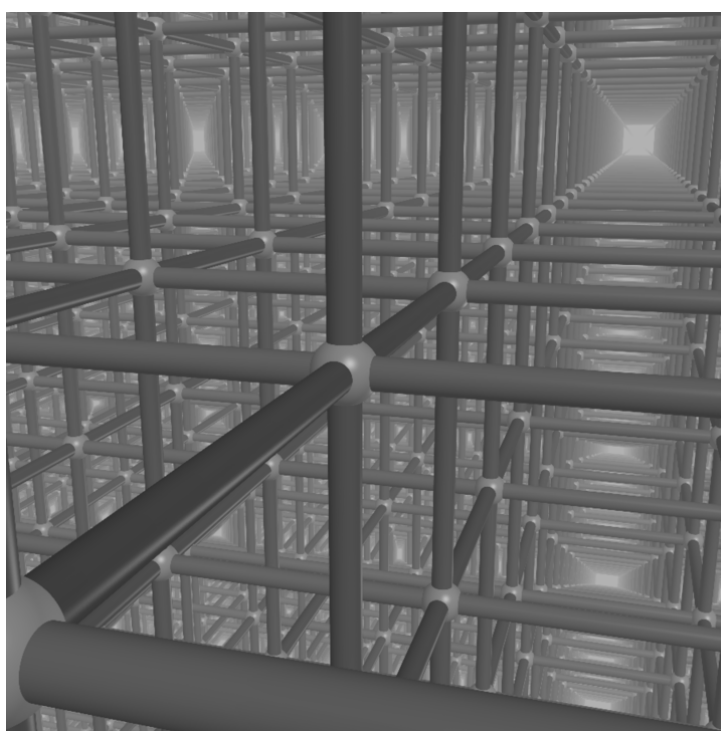
Rysowanie uproszczonych obiektów, jeśli są one małe na obrazie, nazywa się **elizją**. Termin ten pochodzi z językoznawstwa i oznacza zanikanie pewnych głosek w płynnej mowie, czyli w pewnym sensie pomijanie mało istotnych szczegółów wymawianych słów (co nie ma niczego wspólnego z niestaranną wymową).

Działanie CSG. Konstrukcyjna geometria brył w naturalny sposób wiąże się z hierarchicznym opisem sceny. Możemy zatem zrealizować jednolitą hierarchiczną reprezentację sceny z bryłami CSG. Oczywiście, albo algorytm tworzenia obrazu takiej sceny musi dopuszczać takie dane, albo też reprezentacja musi zostać poddana wstępnemu przetwarzaniu, w wyniku którego powstanie reprezentacja nadająca się do wyświetlenia sceny za pomocą algorytmu, który ma być użyty.

Często zdarza się, że pewne obiekty pojawiają się w scenie „w wielu egzemplarzach”. Na przykład samochody mają najczęściej cztery jednakowe koła. Byłoby rzeczą niestosowną każde z kół opisywać osobno, a jeszcze bardziej niestosowne byłoby tworzenie osobnych reprezentacji wszystkich szprych w tych kołach. Dlatego zamiast drzewa bardziej odpowiednią strukturą danych reprezentującą hierarchię sceny bywa **bezcyklowy graf skierowany** (ang. *directed acyclic graph*, *DAG*). Podobnie jak drzewo ma on **korzeń**, tj. wierzchołek od którego zaczyna się przeszukiwanie grafu, ale do niektórych wierzchołków można dojść od korzenia na więcej niż jeden sposób. Każdy wierzchołek ma określony **poziom**, który określimy jako długość *najdłuższej* drogi od korzenia do tego wierzchołka (w drzewie jest to długość jedynej drogi). Każdy wierzchołek bezcyklowego grafu skierowanego ma poziom skończony, co oznacza, że w grafie nie ma cykli, tj. nie można, idąc wzdłuż krawędzi zgodnie z ich orientacją, dojść ponownie do wierzchołka, z którego się wyszło. Krawędzie takiego grafu określają częściowy porządek wierzchołków; idąc wzdłuż krawędzi zgodnie z ich orientacją, przechodzimy przez wierzchołki

o coraz większych poziomach. Wierzchołki, z których nie wychodzą żadne krawędzie, podobnie jak w drzewie, nazywamy **liśćmi**.

Wyświetlenie sceny reprezentowanej przez bezcyklowy graf skierowany można wykonać tak samo jak wyświetlanie sceny reprezentowanej przez drzewo; należy przeszukać graf metodą DFS (oczywiście nie zaznaczając odwiedzonych wierzchołków). W związku z tym, że pewne obiekty mogą być wyświetlane wielokrotnie, listę atrybutów wierzchołków można rozszerzyć o **atrybuty modyfikujące** interpretację poddrzew. Na przykład możemy mieć scenę, w której jest kilka jednakowych samochodów pomalowanych na różne kolory. Graf będzie zawierał opis tylko jednego samochodu, ale kolor jego karoserii (nie opon!) będzie ustalony na drodze od korzenia całego grafu do korzenia podgrafu reprezentującego samochód (podobnie jak przekształcenie określające położenie tego samochodu). Można wprowadzić wiele parametrów modyfikujących, na przykład kąt skręcenia przednich kół, uchylenia drzwi, itd.



Rysunek 7.5. Obraz sceny złożonej z dużej liczby obiektów.

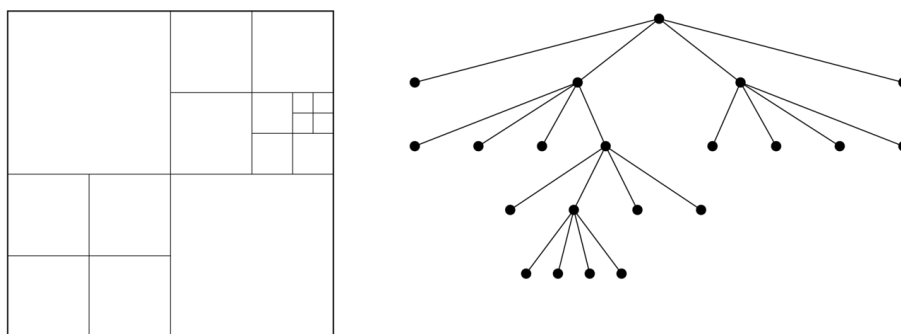
Na rysunku 7.5 jest przedstawiony obraz (inspirowany jedną z grafik M. C. Eschera) sceny złożonej z 2^{16} kul i $3 \cdot 2^{11}$ walców. Bezcyklowy graf skierowany reprezentujący tę scenę miał 93 wierzchołki, przy czym wszystkie wierzchołki oprócz korzenia i liści miały wskaźniki do dwóch wierzchołków o większym poziomie.

8. Drzewa binarne, czwórkowe i ósemkowe

Wiele zadań związanych z przetwarzaniem figur geometrycznych można rozwiązać lub uprościć dokonując podziału tych figur. Podział może być wykonany w sposób zależny od figury (np. triangulacja wielokąta polega na dzieleniu wzdłuż odcinków, których końcami są wierzchołki wielokąta), albo w sposób arbitralny.

Zajmiemy się teraz rekurencyjnym podziałem kostki d -wymiarowej. Otrzymane fragmenty kostki nazwiemy **boksami**. Każdy boks powstaje przez podział większego boksu na równe części. Wspólne ściany tych części są równoległe do ścian kostki. Podział taki możemy wykonywać stosownie do potrzeb i stosownie do potrzeb możemy wybierać metodę podziału.

8.1. Drzewa czwórkowe



Rysunek 8.1. Podział kwadratu na boksy i odpowiadające mu drzewo czwórkowe.

Zacniemy od przypadku najłatwiejszego do narysowania, a zatem najbardziej odpowiedniego do przedstawienia zasady. Przyjmiemy, że $d = 2$, a zatem kostka jest kwadratem (albo, ogólniej, prostokątem). Cała kostka jest reprezentowana przez korzeń drzewa. Kostkę i każdy boks dzielimy na cztery przystające kwadraty (lub prostokąty). Każdy wierzchołek drzewa, który nie jest liściem, ma cztery poddrzewa.

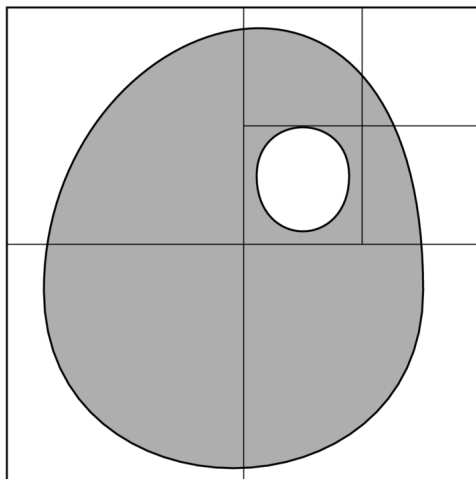
Drzewa czwórkowe można zastosować w naturalny sposób do rozwiązywania różnych zadań dwuwymiarowych, takich jak

- Przeszukiwanie obszarów, na przykład w geograficznej bazie danych,
- Konstrukcyjna geometria figur płaskich,
- Algorytmy widoczności,
- Transmisja obrazów,
- Wykrywanie kolizji w ruchu płaskim, np. w animacji.

Omówimy reprezentację płaskiej figury F , mieszczącej się w prostokącie. Z każdym wierzchołkiem drzewa zwiążemy atrybut, zwany umownie kolorem. Wierzchołek jest

- **czarny**, jeśli wnętrze boksu reprezentowanego przez ten wierzchołek jest zawarte w figurze F ,
- **biały**, jeśli wnętrze boksu jest rozłączne z figurą F ,
- **szary**, jeśli przecięcie wnętrza boksu i brzegu figury F jest niepuste.

Nie ma powodu, aby dzielić wierzchołki czarne i białe, zatem są one zwykle liśćmi drzewa, a wszystkie wierzchołki wewnętrzne są szare. Istnieją też na ogół szare liście, co wynika z ograniczenia wysokości drzewa, które określa osiągalną dokładność reprezentowania figur. Można jednak umieścić w szarych liściach dodatkową informację, która pozwala na przykład zbadać, czy dany punkt, należący do boku reprezentowanego przez szary liść, należy do F , czy nie. Informacja taka może być prostsza niż reprezentacja całej figury F .



Rysunek 8.2. Podział kwadratu na boksy dla pewnego wielokąta krzywoliniowego.

Przykład: Niech figura F będzie wielokątem krzywoliniowym, którego brzeg składa się z kilku krzywych sklepanych. Podział boksów wykonujemy do chwili, gdy w każdym boksie, który zawiera fragment brzegu, fragment taki składa się z co najwyżej dwóch połączonych łuków wielomianowych. Sprawdzenie, czy dany punkt p należy do F polega na odnalezieniu w drzewie liścia, który zawiera ten punkt, i jeśli ten liść jest szary, to zbadanie, po której stronie brzegu leży punkt p , jest dosyć łatwe.

8.1.1. Konstrukcyjna geometria figur płaskich

Drzewo reprezentujące dopełnienie figury reprezentowanej przez dane drzewo czwórkowe jest bardzo łatwe do otrzymania; wystarczy zamienić kolor każdego wierzchołka na przeciwny (tj. czarny na biały, biały na czarny, i szary na złoty). Przetwarzanie dodatkowej informacji o brzegu (jeśli taka jest) może polegać na zmianie orientacji krzywej brzegowej. Jeśli krzywa ta jest krzywą Béziera, to wystarczy w tym celu ustawić jej punkty kontrolne w odwrotnej kolejności.

Mając drzewa reprezentujące dwie figury płaskie, zawarte w tej samej kostce, możemy skonstruować drzewo, które reprezentuje sumę, przecięcie lub różnicę tych figur. Dysponując procedurą wyznaczającą dopełnienie figury, każdą z tych operacji możemy zrealizować za pomocą np. procedury wyznaczania przecięcia. Procedura ta opiera się na fakcie, że każdy boks może być reprezentowany tylko przez wierzchołki znajdujące się w identycznej pozycji w obu drzewach i w obu drzewach może być podzielony tylko w ten sam sposób.

Dzięki powyższej własności drzew, jest możliwe jednoczesne obejście metodą DFS wierzchołków reprezentujących te same boksy. Przetwarzając wierzchołki, którym odpowiada ten sam boks, procedura sprawdza, czy to liście i bada ich kolory. Zależnie od wyniku tego badania, procedura tworzy wierzchołek drzewa reprezentującego przecięcie i nadaje mu kolor, lub wykonuje działanie takie jak w poniższej tabelce:

$\downarrow$ 1 2 $\rightarrow$	czarny liść	biały liść	szary liść	poddrzewo
czarny liść	czarny liść	biały liść	szary liść 2	kopia poddrzewa 2
biały liść	biały liść	biały liść	biały liść	biały liść
szary liść	szary liść 1	biały liść	A	B
poddrzewo	kopia poddrzewa 1	biały liść	B	C

Procedura A: jeśli szare liście zawierają informację, która umożliwia dokładne odtworzenie przecięć boksów z figurami, to należy zbadać, czy te przecięcia są rozłączne. Jeśli tak, to procedura tworzy biały liść. W przeciwnym razie powstaje szary wierzchołek, który może być liściem albo korzeniem poddrzewa, jeśli informacja o brzegu przecięcia jest zbyt skomplikowana, aby można ją było przechowywać w liściu. Jeśli drzewa nie zawierają dodatkowej informacji, to procedura tworzy szary liść (albo biały); kolor tego liścia może być błędny.

Procedura B: jeśli nie ma dodatkowej informacji o przecięciu figur z boksem, to procedura tworzy szary liść. W przeciwnym razie szary liść jednego lub drugiego drzewa jest zamieniany na korzeń poddrzewa (zostaje ono rozbudowane przez dodanie czterech liści reprezentujących ćwiartki boks); wierzchołki obu drzew, reprezentujące te ćwiartki, są następnie przetwarzane rekurencyjnie, za pomocą procedury C.

Procedura C: oba wierzchołki reprezentujące dany boks są wewnętrzne, mają więc wskaźniki do poddrzew reprezentujących ćwiartki boks. Ćwiartki te przetwarzamy wywołując rekurencyjnie procedurę wyznaczania przecięcia.

Po wykonaniu działań zgodnie z powyższą tabelką i opisem, należy jeszcze uprościć wynik, jeśli się da. Jeśli wierzchołek reprezentujący przecięcie figur w boksie jest korzeniem poddrzewa i wszystkie jego poddrzewa są białymi (albo czarnymi) liśćmi, to usuwamy je i zamieniamy bieżący wierzchołek na biały (albo czarny) liść.

Przykład zastosowania: Przypuśćmy, że jedna z figur przedstawia obszar zalesiony, a druga obszar zabagniony. Ostoję puszczy (tzw. macecznik) możemy zlokalizować wyznaczając drzewo reprezentujące część wspólną tych obszarów.

8.1.2. Algorytm widoczności

Drzewo czwórkowe ma zastosowanie w następującym algorytmie widoczności (algorytmie Warnocka). Przypuśćmy, że mamy scenę trójwymiarową składającą się z płaskich wielokątów o co najwyżej wspólnych krawędziach. Algorytm jest następujący:

1. Rzutujemy krawędzie wielokątów na płaszczyznę obrazu; dostajemy zbiór rzutów krawędzi.
2. Tworzymy drzewo czwórkowe, którego korzeń reprezentuje cały obraz. Boks dzielimy na mniejsze wtedy, gdy jest większy niż jeden piksel, a w jego wnętrzu leży rzut jakiejś krawędzi (jest też wariant: więcej niż jednej krawędzi).
3. Dla każdego liścia znajdujemy środek boksów i rozstrzygamy widoczność w tym punkcie, tj. znajdujemy ścianę, której przecięcie z półprostą, której rzutem jest ten punkt, jest najbliższy obserwatora. Cały boks wypełniamy kolorem tej ściany. W wariantcie dopuszczającym obecność rzutu jednej krawędzi wewnątrz boksów reprezentowanego przez liść widoczność rozstrzygamy w dwóch punktach, leżących po przeciwnych stronach tej krawędzi i nadajemy kolory dwóm wielokątom powstałym z podziału boksów przez tę krawędź. Wariant ten działa szybciej, ponieważ wymaga przeszukiwania drzew o znacznie mniejszej wysokości.

8.1.3. Transmisja obrazów i MIP-mapping

Przypuśćmy, że należy przesłać pewien obraz w ten sposób, aby odbiorca mógł go niezbyt dokładnie wyświetlić po otrzymaniu niewielkiej ilości danych. Może wtedy przerwać transmisję przed końcem jeśli obraz mu się nie podoba i nie chce płacić za przesyłanie całości.

Metoda opisana niżej powoduje pewien wzrost objętości danych (o $1/3$), ale ponieważ wiele algorytmów kompresji (bezstratnej) działa „po kolei” (tj. odtwarza zakodowany ciąg w kolejności uporządkowania jego elementów), więc to nie jest bardzo ważne.

1. Tworzymy drzewo pełne, którego liście to piksele, a każdy węzeł wewnętrzny ma kolor o wartości średniej arytmetycznej kolorów korzeni poddrzew (kolor korzenia obliczamy na końcu),
2. Przesyłamy kolory wierzchołków drzewa w kolejności przeszukiwania drzewa algorytmem BFS, tj. najpierw korzeń, potem jego 4 poddrzewa, potem 16 ich poddrzew itd. Wyświetlanie polega na wypełnianiu stałym kolorem coraz mniejszych kwadratów.

Powyższy algorytm, jak łatwo zauważyć, polega na przesyłaniu kolejnych obrazów o rozdzielczości 1×1 , 2×2 itd., aż do oryginalnego obrazu na końcu (z pikselami poprzestawianymi w pewien sposób). Stąd bierze się wspomniany wzrost objętości danych. Zastanówmy się, jak go uniknąć.

Wydawało by się, że znając średnią arytmetyczną czterech liczb i trzy z nich, można obliczyć czwartą, a zatem można by nie przysyłać koloru jednego (np. ostatniego) z czterech poddrzew każdego wierzchołka. Tak jednak nie jest z powodu błędów zaokrągleń. Jeśli jednak zamiast średniej arytmetycznej przyjmiemy kolor korzenia poddrzewa równy kolorowi jednego z wierzchołków jego poddrzew (np. pierwszego), to możemy go później nie przysyłać i błędów zaokrągleń (ani żadnych obliczeń numerycznych) tu nie ma. Jest za to pewne pogorszenie jakości przybliżenia obrazu przez początkowo przesłane dane, przez co decyzja o przesłaniu go do końca lub nie, może być błędna.

Identyczna zasada tworzenia obrazów o niższej rozdzielczości ma zastosowanie w tzw. MIP-mappingu, który jest zastosowaniem elizji w nakładaniu tekstury. Będzie o tym mowa później.

8.2. Drzewa ósemkowe

Drzewa ósemkowe spełniają w zastosowaniu do figur przestrzennych te same role, co drzewa czwórkowe dla figur płaskich. Korzeń drzewa ósemkowego reprezentuje kostkę trójwymiarową (prostopadłościan lub w szczególności sześcian), a jego wierzchołki reprezentują boksy, będące prostopadłościanami podobnymi do całej kostki. Drzewo ósemkowe może być podstawową reprezentacją figury przestrzennej, a także pomocniczą strukturą danych, która ma na celu przyspieszenie pewnych obliczeń.

Przykład: Opisana wcześniej konstrukcyjna geometria brył wielościennych wymaga wyznaczenia wszystkich krawędzi przecięcia ścian danych brył. Jeśli liczby ścian brył oznaczmy symbolami n_1 i n_2 , to „bezpośrednia” procedura

Listing.

```

for i := 1 to  $n_1$  do
  for j := 1 to  $n_2$  do
    sprawdź, czy ściany  $f_{1i}$  i  $f_{2j}$  przecinają się
    i jeśli tak, to wyznacz ich krawędź przecięcia;

```

ma koszt rzędu $n_1 n_2$, nawet jeśli liczba znalezionych krawędzi jest znacznie mniejsza. Można jednak dla każdej ściany pierwszej bryły znaleźć kostkę, w której ta ściana jest zawarta, a następnie zbudować drzewo ósemkowe, którego każdy wierzchołek zawiera listę ścian pierwszej bryły, mających niepuste przecięcie z odpowiednim boksem. Następnie, dla każdej ściany drugiej bryły wyszukujemy boksy, z którymi ta ściana przecina się i sprawdzamy, czy istnieją wspólne krawędzie ze ścianami pierwszej bryły znajdującymi się w odpowiednich listach. Ta metoda jest opłacalna zwłaszcza wtedy, gdy brzegi brył składają się z dużej liczby małych ścian.

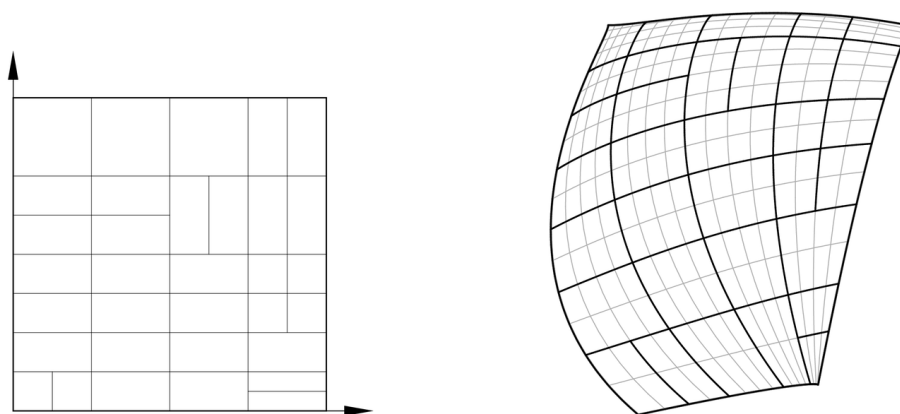
Kolejne przykłady zastosowania takich drzew, to wykrywanie kolizji obiektów poruszających się w scenie, która nie zmienia się w czasie oraz wyznaczanie przecięć promieni z obiektami w metodzie śledzenia promieni. W obu tych przypadkach oszczędności czasowe mogą być bardzo duże.

8.3. Drzewa binarne

Wadą drzew czwórkowych i ósemkowych może być bardzo duża liczba poddrzew każdego wierzchołka; jeśli liście mają taką samą reprezentację jak wierzchołki wewnętrzne, to mają cztery lub osiem pustych wskaźników.

Druga cecha, która może być zaletą lub wadą (zależnie od sytuacji) to brak adaptacji kierunkowej; na wszystkich poziomach rekurencyjnego podziału boksy są podobne do wyjściowej kostki; nie można w związku z tym dostosować się do „wydłużonych” obiektów.

Aby to umożliwić, można zamiast drzewa czwórkowego lub ósemkowego zastosować drzewo binarne, w którym boksy dzieli się na dwie części — prostokąty albo prostopadłościany, i można przy tym dowolnie (na przykład kilka razy kolejno tak samo) wybrać kierunek podziału.



Rysunek 8.3. Przykład zastosowania drzewa binarnego.

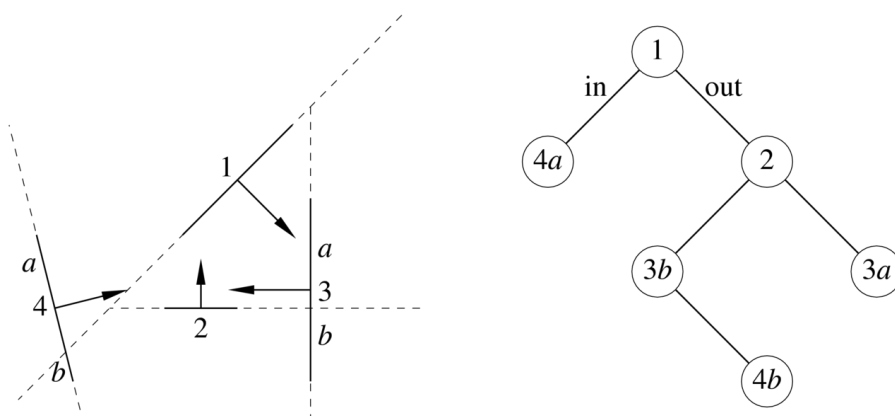
Przedstawione na rysunku drzewo binarne powstało z zastosowaniem adaptacyjnego wyboru kierunku podziału boksów. Pokazany płat był dzielony rekurencyjnie na kawałki za pomocą algorytmu de Casteljau. Kierunek podziału był za każdym razem wybierany tak, aby otrzymać kawałki o jak najmniejszej średnicy.

8.4. Binarny podział przestrzeni

W odróżnieniu od kostki będziemy teraz dzielić całą przestrzeń, czyli zbiór nieograniczony. Idea binarnego podziału przestrzeni (ang. *binary space partition*, *BSP*) polega na uzależnieniu go od danych, które wyznaczają pewne hiperpłaszczyzny. Na przykład przestrzeń trójwymiarową będziemy dzielić płaszczyznami, w których leżą dane wielokąty płaskie. Na rysunkach niżej są odcinki, które wyznaczają proste, którymi można podzielić płaszczyznę, bo łatwiej to narysować, a zasada podziału jest identyczna.

Dla każdej ściany znamy jej płaszczyznę, która jest określona przez podanie dowolnego punktu i wektora normalnego. Zwrotu wektora normalnego użyjemy do rozróżnienia dwóch półprzestrzeni rozgraniczonych przez ścianę; wektor ten jest zorientowany w stronę półprzestrzeni

„out”, a druga półprzestrzeń jest oznaczona „in”. Dla ścian przetwarzanych w kolejności zgodnej z oznaczeniami na rysunku 8.4 powstanie drzewo przedstawione obok na tym rysunku.



Rysunek 8.4. Zasada tworzenia drzewa binarnego podziału przestrzeni.

Korzeń drzewa reprezentuje całą przestrzeń. Po wstawieniu pierwszej ściany mamy dwa poddrzewa, które reprezentują półprzestrzenie. Wstawiając każdą następną ścianę przeszukujemy drzewo algorytmem DFS, wybierając za każdym razem to poddrzewo, które zawiera wstawianą ścianę. Jeśli ściana przecina płaszczyznę podziału przestrzeni, to dzielimy ją na dwie części (np. za pomocą algorytmu Sutherlanda-Hodgmana) i wstawiamy je odpowiednio do lewego i prawego poddrzewa. Ściany położone w płaszczyźnie innej ściany, wstawionej wcześniej, możemy dołączyć do odpowiedniego wierzchołka drzewa (którego atrybutem powinna być wtedy lista ścian).

Zauważmy, że

- Koszt budowy drzewa jest nie mniejszy niż $O(n \log n)$ operacji (jeśli drzewo jest idealnie zrównoważone i żadnej ściany nie trzeba dzielić), i nie większy niż $O(n^3)$ operacji (ten najgorszy przypadek ma miejsce wtedy, gdy każda ściana przecina się z wszystkimi pozostałymi; wtedy wskutek podziału, niezależnie od uporządkowania otrzymamy $\frac{1}{2}(n^2 + n)$ fragmentów ścian.).
- Jeśli ściany są ścianami wielościanu wypukłego, to drzewo BSP ma wysokość równą liczbie ścian, każdy wierzchołek oprócz ostatniego ma tylko jedno niepuste poddrzewo, a koszt jego utworzenia jest $O(n^2)$ operacji.
- Ściany płaskie (wielokąty) można reprezentować za pomocą drzew BSP, które opisują podział płaszczyzn ścian przez proste, na których leżą krawędzie ścian. Można więc wprowadzić hierarchię drzew BSP (i czasem tak się robi).
- Budując drzewo BSP warto obniżać jego koszt. Wybierając ścianę, która ma być wstawiona w następnej kolejności, warto wybrać taką ścianę, która
 - spowoduje podzielenie najmniejszej liczby pozostałych (jeszcze nie wstawionych) ścian, a najlepiej żadnych,
 - rozdzieli pozostałe ściany w przybliżeniu na równoliczne podzbiory (co zmniejsza wysokość drzewa). Aby osiągnąć ten cel, czasem wprowadza się dodatkowe płaszczyzny podziału przestrzeni (bez ścian) — to jest jedyna metoda skuteczna dla zbioru ścian bryły wypukłej.

Drzewa BSP mają zastosowanie w algorytmach widoczności i wyznaczania cieni, a także w różnych zadaniach, w których ich celem jest obniżenie kosztu algorytmu, dzięki zmniejszeniu ilości wykonywanych obliczeń.

9. Algorytmy widoczności

9.1. Rodzaje algorytmów widoczności

Dana jest scena trójwymiarowa, tj. pewien zbiór figur w przestrzeni, i położenie obserwatora. Zadanie polega na znalezieniu (i ewentualnym wykonaniu obrazu) fragmentów figur widocznych dla obserwatora. Algorytmów rozwiązujących to zadanie jest dużo i warto je poklasyfikować, dzięki czemu łatwiej będzie wybrać algorytm odpowiedni do potrzeb.

Z problemem rozstrzygania widoczności wiąże się zadanie wyznaczenia cieni dla punktowych źródeł światła. Części figur sceny niewidoczne z punktu położenia źródła światła leżą w cieniu. Dlatego wyznaczanie cieni jest rozstrzyganiem widoczności. Nie wszystkie algorytmy rozstrzygania widoczności są jednak odpowiednie do wyznaczania cieni.

Podstawowy podział algorytmów widoczności wyróżnia

Algorytmy przestrzeni danych, które wyznaczają reprezentację obszaru widocznego, na podstawie której można wykonać wiele obrazów, o dowolnej rozdzielczości (w tym obrazy otrzymane po zmianie rzutni, przy ustalonym położeniu obserwatora),

Algorytmy przestrzeni obrazu — wynikiem takiego algorytmu jest obraz, czyli tablica odpowiednio pokolorowanych pikseli. Mogą też być dodatkowe informacje związane z każdym pikselem, ale zmiana rozdzielczości obrazu wymaga ponownego wykonania algorytmu widoczności.

Inne cechy dowolnego algorytmu widoczności to

Klasa danych, dla których algorytm może działać. Algorytm może dopuszczać

- tylko powierzchnie płaskie (np. wielokąty), albo także powierzchnie zakrzywione,
- zbiory powierzchni, które mogą się przecinać, albo nie,
- obiekty z otwartą objętością (powierzchnie dwustronne), albo tylko z zamkniętą objętością,
- bryły o jawnej reprezentacji, albo drzewa CSG, a także obiekty zdeformowane reprezentowane za pomocą niezdeformowanego prymitywu i odpowiedniego przekształcenia,
- tylko powierzchnie, albo również dane objętościowe (reprezentujące np. chmurę, w której są widoczne zanurzone obiekty).

Rodzaj obrazu; tu można wyróżnić

- algorytmy powierzchni zasłoniętej i
- algorytmy linii zasłoniętej.

Stopień skomplikowania algorytmu i jego struktur danych, możliwość zrównoleglenia obliczeń, a także możliwość użycia sprzętu (procesora w karcie graficznej) do wykonania wszystkich lub niektórych kroków algorytmu.

Inne szczegóły, np. obecność preprocesingu, w wyniku którego rozstrzyganie widoczności dla wielu kolejno otrzymywanych obrazów, np. podczas ruchu obserwatora względem sceny, jest tańsze.

Można zaobserwować ogólną zasadę, że im mniej informacji ma dostarczyć algorytm (tj. im mniej dokładna reprezentacja obszaru widocznego jest potrzebna — np. tylko pokolorowane piksele) tym ogólniejsze dane można dopuścić. Daje to przewagę w zastosowaniach algorytmom przestrzeni obrazu. Ważna jest też możliwość implementacji tych algorytmów w sprzęcie. Co prawda, wtedy nie zawsze jest zrealizowana możliwość przetwarzania najbardziej skom-

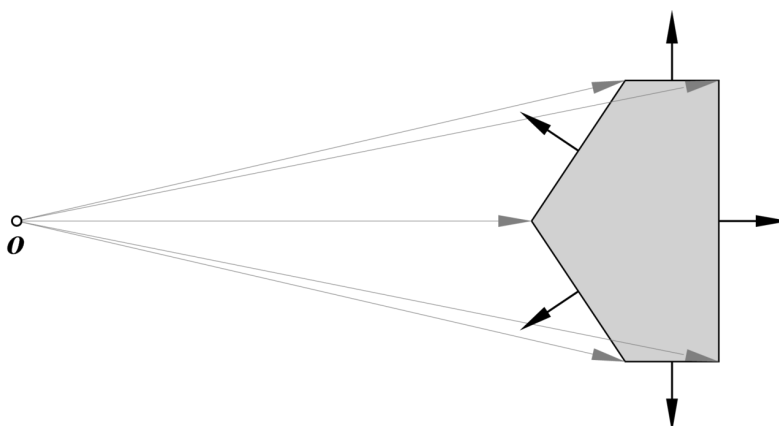
plikowanych danych, jakie dla danego algorytmu są teoretycznie dopuszczalne. Na przykład algorytm z buforem głębokości teoretycznie dopuszcza bardzo wiele różnych rodzajów danych (np. gładkie powierzchnie zakrzywione), ale jego implementacja sprzętowa, która jest jednym z najważniejszych elementów OpenGL-a, umożliwia rozstrzyganie widoczności tylko płaskich wielokątów i odcinków. Można napisać własną implementację, programową, ale z wyjątkiem bardzo specjalnych sytuacji nie ma to sensu. Ze względu na szybkość działania i dodatkowe możliwości OpenGL-a (np. oświetlenie, teksturowanie) lepiej jest przybliżać powierzchnie za pomocą odpowiedniej liczby płaskich trójkątów.

9.2. Algorytmy przestrzeni danych

Skupimy się na algorytmach rozstrzygania widoczności w scenach złożonych z płaskich wielokątów. Widoczna część każdego z nich też jest płaskim wielokątem.

9.2.1. Odrzucanie ścian „tylnych”

Jeśli wielokąty są ścianami brył (obiektów z zamkniętą objętością), to korzystając z umowy, że wektor normalny płaszczyzny każdej ściany jest zorientowany „na zewnątrz bryły”, można odrzucić (uznać za niewidoczne w całości) ściany „odwrócone tyłem” do obserwatora. Ten wstępny test widoczności sprowadza się do zbadania znaku iloczynu skalarnego: jeśli punkt p jest dowolnym wierzchołkiem ściany, punkt o jest położeniem obserwatora, zaś n jest wektorem normalnym ściany to z warunku $\langle p - o, n \rangle \geq 0$ wynika, że ściana jest niewidoczna.



Rysunek 9.1. Zasada odrzucania ścian „odwróconych tyłem”.

Jeśli scena składa się z jednego wielościanu wypukłego, to wszystkie pozostałe ściany są w całości widoczne. Opisany test jest często stosowany jako wstępny etap innych algorytmów widoczności, zarówno działających w przestrzeni danych, jaki i obrazu. Przed zastosowaniem tego algorytmu trzeba się upewnić, że rysujemy bryły i położenie obserwatora nie jest wewnątrz jednej z nich.

W OpenGL-u można uaktywnić taki test, wywołując przed wyświetlaniem sceny procedury

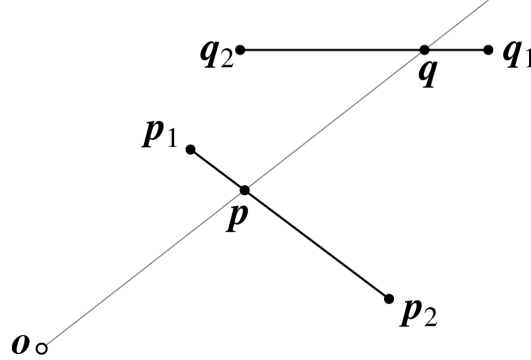
```
glEnable ( GL_CULL_FACE );
glCullFace ( GL_BACK );
```

Wyświetlane wielokąty muszą być wypukłe, a ich brzegi odpowiednio zorientowane; jeśli v_0 , v_1 i v_2 są pierwszymi trzema wierzchołkami wielokąta (nie mogą one być współliniowe), to wektor

$(\mathbf{v}_1 - \mathbf{v}_0) \wedge (\mathbf{v}_2 - \mathbf{v}_0)$ (który jest wektorem normalnym płaszczyzny ściany) musi być zorientowany na zewnątrz bryły.

9.2.2. Obliczanie punktów przecięcia odcinków w przestrzeni

Przypuśćmy, że mamy dane dwa odcinki w przestrzeni trójwymiarowej i punkt $\mathbf{o}$, który jest położeniem obserwatora. Odcinki są określone za pomocą punktów końcowych, odpowiednio $\mathbf{p}_1, \mathbf{p}_2$ i $\mathbf{q}_1, \mathbf{q}_2$. Chcemy obliczyć (jeśli istnieją) punkty $\mathbf{p}$ i $\mathbf{q}$, które leżą na odcinkach i na prostej przechodzącej przez punkt $\mathbf{o}$.



Rysunek 9.2. Odcinki w przestrzeni, których obrazy przecinają się.

Aby rozwiązać zadanie, przedstawimy poszukiwane punkty w taki sposób:

$$\begin{aligned}\mathbf{p} &= \mathbf{p}_1 + s(\mathbf{p}_2 - \mathbf{p}_1), \\ \mathbf{q} &= \mathbf{q}_1 + t(\mathbf{q}_2 - \mathbf{q}_1).\end{aligned}$$

Warunkiem współliniowości punktów $\mathbf{o}$, $\mathbf{p}$ i $\mathbf{q}$ jest istnienie liczby rzeczywistej u , takiej że $\mathbf{p} - \mathbf{o} = u(\mathbf{q} - \mathbf{o})$. Mamy tu układ trzech równań z trzema niewiadomymi, s , t , u :

$$\mathbf{p}_1 + s(\mathbf{p}_2 - \mathbf{p}_1) - \mathbf{o} = u(\mathbf{q}_1 + t(\mathbf{q}_2 - \mathbf{q}_1) - \mathbf{o}).$$

Po przekształceniu otrzymujemy układ równań liniowych

$$[\mathbf{p}_2 - \mathbf{p}_1, \mathbf{o} - \mathbf{q}_1, \mathbf{q}_1 - \mathbf{q}_2]\mathbf{x} = \mathbf{o} - \mathbf{p}_1,$$

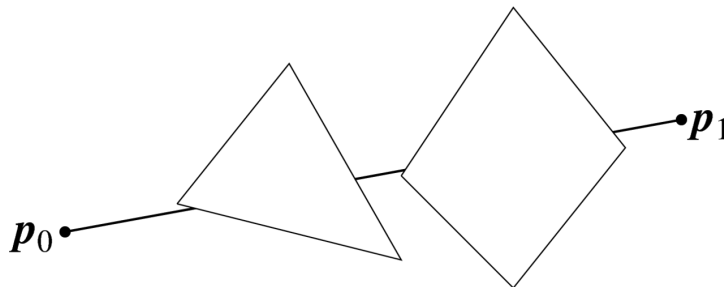
z niewiadomym wektorem $\mathbf{x} = [s, u, t]^T$. Po jego rozwiązaniu możemy obliczyć punkty $\mathbf{p}$ i $\mathbf{q}$. Dla celów rozstrzygnięcia zasłaniania trzeba jednak zinterpretować możliwe wyniki.

- Jeśli macierz układu ma rząd 0, to oba dane odcinki są zdegenerowane do punktów, a ponadto jeden z nich pokrywa się z położeniem obserwatora. Tego rodzaju dane są zwykle eliminowane przed przystąpieniem do rozstrzygnięcia widoczności.
- Rząd 1 macierzy może wystąpić wtedy, gdy oba odcinki są równoległe i jeden z nich leży na prostej przechodzącej przez punkt $\mathbf{o}$. Również ten przypadek nie wystąpi, jeśli odcinki są krawędziami ścian „odwróconych przodem” do obserwatora (test opisany przed chwilą odrzuca każdą ścianę, której płaszczyzna zawiera położenie obserwatora).
- Jeśli macierz ma rząd 2, to albo nie ma rozwiązań, czyli poszukiwane punkty $\mathbf{p}$ i $\mathbf{q}$ nie istnieją, albo rozwiązań jest nieskończenie wiele — oba odcinki leżą w płaszczyźnie zawierającej punkt $\mathbf{o}$ i mogą się zasłaniać, albo nie. Dokończenie dyskusji tego przypadku polecam jako ćwiczenie, przyda się ono podczas pisania procedur realizujących różne algorytmy widoczności (wskazówka: trzeba zbadać cztery rozwiązania odpowiadające ustalonym wartościom s i t , którym nadajemy kolejno wartości 0 i 1).

- Macierz rzędu 3 oznacza istnienie jednoznacznego rozwiązania. Po jego obliczeniu należy sprawdzić, czy $s, t \in [0, 1]$; punkty p i q istnieją tylko w takim przypadku. Jeśli $u < 0$, to punkty te znajdują się po przeciwnych stronach położenia obserwatora. Jeśli $u \in (0, 1)$, to bliżej obserwatora jest punkt p , a jeśli $u > 1$, to punkt q . Ta informacja jest potrzebna w testach widoczności.

9.2.3. Algorytm Ricciego

Algorytm Ricciego jest algorytmem linii zasłoniętej dla sceny złożonej z wielokątów (np. ścian wielościanów), które nie przenikają się. Dane składają się ze zbioru ścian (ewentualnie tylko „odwróconych przodem” do obserwatora), zbioru ich krawędzi i położenia obserwatora. Przyjęte jest też założenie upraszczające — wszystkie ściany są wielokątami wypukłymi (w tym celu można je podzielić na wielokąty wypukłe, krawędzie wprowadzone podczas tego dzielenia nie są dołączane do zbioru krawędzi, których fragmenty mogą być widoczne).



Rysunek 9.3. Krawędź częściowo zasłonięta przez ściany.

Wykonując algorytm, kolejno dla każdej krawędzi k wykonujemy następujące czynności:

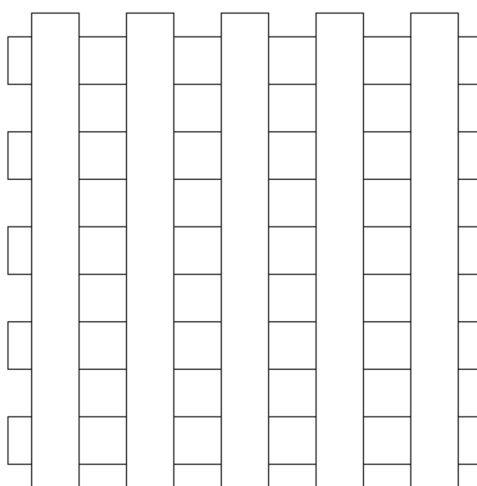
- Tworzymy reprezentację widocznych części krawędzi, w postaci listy par liczb. Liczby te są wartościami parametru, które odpowiadają punktom końcowym niezasłoniętym częściom krawędzi. Na początku każda taka lista zawiera tylko jedną parę liczb, $[0, 1]$, która reprezentuje całą krawędź.
- Kolejno dla każdej ściany s wykonujemy (specjalnie dostosowany) algorytm Lianga-Barsky’ego obcinania odcinka, czyli
 - wyznaczamy liczby a, b , które odpowiadają punktom przecięcia rzutów krawędzi k z krawędziami ściany s (jeśli takie punkty istnieją, to najwyżej dwa, ponieważ ściany są wypukłe).
 - Jeśli ściana s zasłania krawędź k , to od przedziałów parametrów reprezentujących niezasłoniętą część krawędzi k odejmujemy przedział $[a, b]$, przetwarzając odpowiednio listę (można to nazwać „konstrukcyjną geometrią odcinków”).

Proces kończymy po przetworzeniu ostatniej ściany, albo po otrzymaniu pustej listy przedziałów (która oznacza, że krawędź k jest w całości niewidoczna).

- Kolejno dla każdego elementu listy obliczamy punkty na krawędzi k i wyprowadzamy (np. rysujemy) odcinek.

Koszt tego algorytmu jest proporcjonalny do kwadratu liczby krawędzi, czyli jest dość wysoki. Wśród algorytmów przestrzeni danych może to jednak być algorytm optymalny, ponieważ wynik (liczba widocznych fragmentów krawędzi albo ścian) może mieć wielkość tego rzędu. Przykład jest na rysunku 9.4.

Takie dane jak na rysunku zdarzają się jednak rzadko, w związku z czym implementując algorytm warto sięgnąć po techniki przyspieszające (obniżające średnią złożoność obliczeniową algorytmu). Najprostszy taki sposób polega na



Rysunek 9.4. Scena, w której liczba widocznych kawałków krawędzi jest proporcjonalna do kwadratu liczby krawędzi.

- zrzutowaniu wierzchołków na płaszczyznę obrazu,
- posortowaniu krawędzi i ścian w kolejności rosnących najmniejszych współrzędnych x końców krawędzi lub wierzchołków ścian,
- utworzeniu (początkowo pustej) listy ścian aktywnych,
- następnie, kolejno dla każdej krawędzi k (w kolejności otrzymanej w wyniku sortowania)
 - wyrzucamy z listy ścian wszystkie ściany, których największa współrzędna x wierzchołka jest mniejsza od mniejszej współrzędnej x końca krawędzi k ,
 - wstawiamy do listy wszystkie ściany, których najmniejsza współrzędna x wierzchołka jest mniejsza niż większa współrzędna x końca krawędzi k ,
 - rozstrzygamy widoczność krawędzi k ze ścianami obecnymi w liście.

Na skuteczność tego sposobu ma wpływ scena oraz sposób jej zrzutowania. Metoda działa tym lepiej, im krótsze w stosunku do wielkości sceny są krawędzie i im mniejsze są ściany.

Jeszcze jedna uwaga: dla każdej krawędzi trzeba znać ściany, które do niej przylegają. Testując widoczność pomijamy je, bo żadna ściana płaska nie zasłania swoich krawędzi, a błędy zaokrągleń mogłyby doprowadzić do innego wyniku.

9.2.4. Algorytm Weilera-Athertona

Algorytm Weilera-Athertona powierzchni zasłoniętej może być użyty do znalezienia fragmentów ścian widocznych z danego punktu, który może być położeniem obserwatora lub źródła światła; jest to więc także algorytm wyznaczania cieni. Dane dla tego algorytmu reprezentują scenę złożoną z płaskich wielokątnych ścian, które mogą mieć otwory, być niespójne itd. Założymy, że nie ma „zapętlania” relacji zasłaniania ścian, tj. nie występuje para ścian, z których każda zasłania część drugiej (w przeciwnym razie należy podzielić jedną ze ścian wzdłuż prostej wspólnej płaszczyzn ścian).

W algorytmie tym sprawdza się widoczność kolejno dla wszystkich par ścian w scenie. Jedną ze ścian testowanej pary rzutuje się na płaszczyznę drugiej ściany, a następnie wykonuje się algorytm Weilera-Athertona w celu wyznaczenia różnicy tych wielokątów. Argument odejmowany odpowiada ścianie leżącej bliżej obserwatora (test widoczności można przeprowadzić dla jednej, dowolnej pary punktów, z których jeden zasłania drugi).

Znaleziony wielokąt odpowiada widocznej części ściany. Od wielokąta tego odejmuje się rzu-

ty wszystkich innych ścian, które ścianę daną zasłaniają. Przetwarzanie ściany kończymy po wyczerpaniu ścian, które mogą ją zasłaniać, albo po otrzymaniu zbioru pustego.

Opisany algorytm jest dość kosztowny (ponieważ wykonuje obliczenia dla każdej pary ścian), ale często można obniżyć jego koszt za pomocą podobnej metody, jak dla algorytmu Ricciego.

9.2.5. Algorytm Appela

Przypuśćmy, że ściany nie przenikają się. Wtedy dowolna krawędź, która nie jest widoczna w całości, ma punkty zmiany widoczności; punkty te są zasłaniane przez punkty leżące na **krawędziach konturowych**, tj. takich krawędziach, do których przylegają ściany, z których jedna jest „obrócona przodem” do obserwatora, a druga jest „obrócona tyłem”.

Krawędzi konturowych jest często znacznie mniej niż wszystkich krawędzi w scenie, co pozwala otrzymać algorytm linii zasłoniętej o mniejszej złożoności obliczeniowej niż ma algorytm Ricciego.

W tym celu określamy **graf krawędziowy**, czyli graf skierowany, którego wierzchołkami są wierzchołki ścian, a krawędziami są krawędzie tych ścian. Orientacja krawędzi jest obojętna, ale musi być określona. Po utworzeniu tego grafu będziemy wprowadzać dodatkowe wierzchołki, dokonując podziału krawędzi w punktach wyznaczonych za pomocą algorytmu opisanego w p. 9.2.2.

Dodatkowo określimy **stopień zasłonięcia**, który jest funkcją w przestrzeni. Wartość tej funkcji w dowolnym punkcie p jest równa liczbie ścian, które znajdują się między punktem p i obserwatorem. Punkty widoczne to te, których stopień zasłonięcia jest równy 0.

Kolejne kroki algorytmu są następujące:

1. Wyznaczamy graf krawędziowy (na ogół jest on częścią reprezentacji sceny).
2. Kolejno dla każdej krawędzi wyznaczamy jej punkty przecięcia w rzucie z krawędziami konturowymi i badamy każdy taki punkt, czy jest on bliżej, czy dalej od obserwatora niż odpowiedni punkt krawędzi konturowej. Jeśli krawędź konturowa zasłania ten punkt, to dzielimy krawędź, na której on leży na fragmenty i wprowadzamy dodatkowy wierzchołek grafu krawędziowego (dzieląc odpowiednią jego krawędź). Każdemu takiemu wierzchołkowi przypisujemy liczbę $+1$ albo -1 , która odpowiada zmianie stopnia zasłonięcia krawędzi (znak tej zmiany zależy od położenia ściany, której krawędź konturową przetwarzaliśmy i od orientacji krawędzi, na której wprowadziliśmy dodatkowy punkt).
3. Wyznaczamy stopień zasłonięcia dowolnego wierzchołka grafu, a następnie, zaczynając od tego wierzchołka, przeszukujemy graf krawędziowy. W każdym wierzchołku, w którym widoczność zmienia się, dodajemy lub odejmujemy (to zależy od orientacji krawędzi i od kierunku „poruszania się” po niej) zmianę stopnia widoczności do lub od wartości stopnia widoczności fragmentu krawędzi przetwarzanego wcześniej. W ten sposób otrzymujemy stopień widoczności wszystkich krawędzi grafu (przeszukujemy w ten sposób wszystkie jego składowe spójne). Wyprowadzamy lub zaznaczamy jako widoczne krawędzie, których stopień zasłonięcia jest równy 0.

Stosując ten algorytm, trzeba uważać, bo sceny złożone z wielokątnych ścian bywają dość skomplikowane i pełna implementacja algorytmu musi być w stanie radzić sobie z wszystkimi przypadkami szczególnymi.

Kłopoty zdarzają się we wspólnych wierzchołkach wielu krawędzi, które mają różne stopnie zasłonięcia. Przeszukując graf, doszedłszy do takiego wierzchołka, najbezpieczniej (w sensie odporności na błędy) jest bezpośrednio obliczyć stopień zasłonięcia krawędzi wychodzących z takiego wierzchołka.

W algorytmie Appela jest możliwa pewna kontrola poprawności; dochodząc do dowolnego odwiedzonego wcześniej wierzchołka grafu możemy porównać jego stopień zasłonięcia obliczony

wcześniej ze stopniem zasłonięcia obliczonym „po drodze” do tego wierzchołka. Muszą one być równe.

9.2.6. Algorytm Hornunga

W algorytmie Hornunga, którego dokładnie nie omówię z powodów, które (mam nadzieję) staną się za chwilę jasne, testy widoczności są ograniczone do par krawędzi konturowych. Jeśli liczba krawędzi konturowych jest dużo mniejsza niż wszystkich, to złożoność tego algorytmu jest jeszcze mniejsza niż złożoność algorytmu Appela. Jest to jednak algorytm skomplikowany. Implementacja algorytmu powierzchni zasłoniętej, opartego na tym algorytmie, miała ponad 5000 linii kodu źródłowego, a napisanie jej zajęło mi w swoim czasie ponad półtora roku. Nie obliczyłem, ile obrazków mógłbym otrzymać przez ten czas, gdybym zajął się innym, wolniejszym algorytmem, ale niewątpliwie dużo. Ponadto algorytm ten jest bardzo wrażliwy na błędy zaokrągleń, które mogą doprowadzić do załamania obliczeń lub całkowicie błędnych wyników.

9.3. Algorytmy przestrzeni obrazu

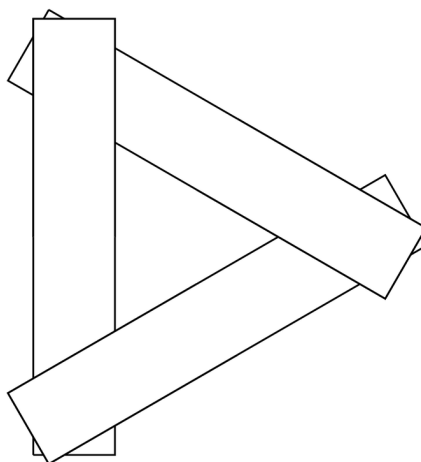
W odróżnieniu od algorytmów opisanych wcześniej, algorytmy przestrzeni obrazu dają w wyniku obraz rastrowy, na którym każdy piksel ma przypisany kolor punktu sceny widocznego w tym miejscu. Jeśli chcemy otrzymać obraz o innej rozdzielczości, albo ewentualnie zmienić kierunek, w którym patrzy obserwator, to musimy wykonać obliczenie jeszcze raz. Z drugiej strony, algorytmy przestrzeni obrazu mają tę przewagę nad algorytmami przestrzeni danych, że dla pewnych klas danych nie istnieje dokładna, obliczalna reprezentacja widocznej części sceny (jest tak dla większości powierzchni zakrzywionych).

9.3.1. Algorytm malarza

Zasada działania algorytmu malarza jest następująca:

1. Posortuj ściany w kolejności od najdalszej do najbliższej obserwatora,
2. Namaluj je w tej kolejności.

Prosta idea niech nie przesłania faktu, że mogą istnieć konflikty widoczności, w których ściana położona w zasadzie dalej zasłania część ściany bliższej obserwatora. Dlatego po posortowaniu (np. ze względu na odległość od obserwatora najbliższego punktu każdej ściany) konieczne jest sprawdzenie i ewentualna modyfikacja uporządkowania ścian.

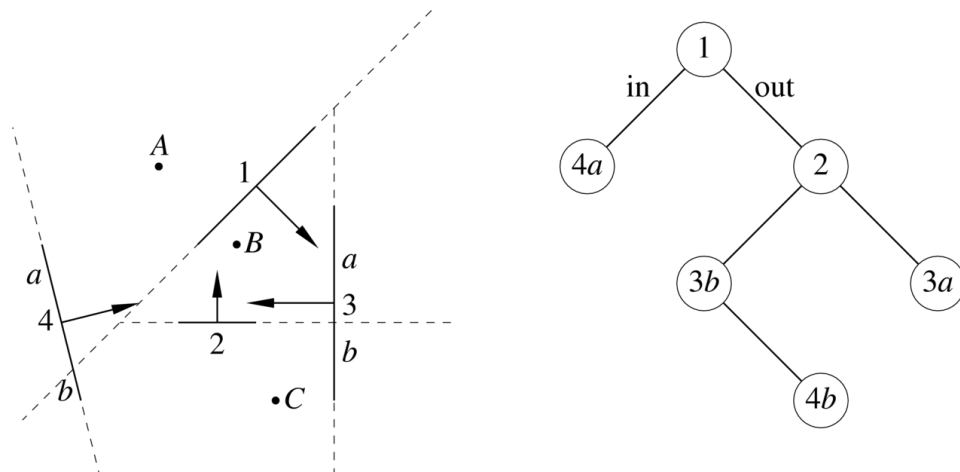


Rysunek 9.5. Zakleszczenie: tych ścian nie można poprawnie narysować w żadnej kolejności.

Ponadto mogą wystąpić konflikty nierozstrzygalne, tj. może nie istnieć uporządkowanie ścian gwarantujące otrzymanie poprawnego obrazu. Klasyczny przykład takiego konfliktu jest na rys. 9.5. Jedyne wyjście w tej sytuacji polega na podzieleniu co najmniej jednej ściany na kawałki i wyświetlenie tych kawałków we właściwej kolejności. Dodatkowa wada algorytmu malarza i jego wariantów (o których będzie mowa dalej) to wielokrotne przemalowywanie pikseli. Jeśli wyznaczenie koloru z uwzględnieniem oświetlenia i tekstury jest czasochłonne, a piksel ostatecznie otrzyma kolor innej ściany, to cała praca wykonana w celu obliczenia koloru piksela, który na ostatecznym obrazie będzie inny, pójdzie na marne.

9.3.2. Algorytm binarnego podziału przestrzeni

Algorytm widoczności z użyciem drzewa BSP jest dość trudny do jednoznacznego zaklasyfikowania; nie otrzymujemy w nim reprezentacji widocznych części ścian, a tylko obraz, utworzony podobnie jak w algorytmie malarza. Natomiast zasadniczy koszt obliczeń związanych z rozstrzyganiem zasłaniania wiąże się z budową drzewa, które nie zależy od rozdzielczości obrazu, ani nawet od położenia obserwatora i rzutni.



Rysunek 9.6. Algorytm widoczności BSP.

Zasadę działania algorytmu przedstawia płaski przypadek na rysunku 9.6, który pokazuje również drzewo BSP zbudowane dla danych czterech ścian. Spostrzeżenie, które leży u podstaw algorytmu jest następujące: jeśli obserwator znajduje się po pewnej stronie dowolnej płaskiej ściany, to ściana ta nie zasłania żadnych obiektów, które znajdują się po tej samej stronie. Zatem aby otrzymać obraz widocznych części ścian, obchodzimy drzewo algorytmem DFS. Dla każdego wierzchołka drzewa sprawdzamy, po której stronie jest obserwator. Jeśli jest on po stronie „in”, to rekurencyjnie wywołujemy procedurę obchodzenia poddrzewa „out”. Następnie rzutujemy i wyświetlamy (zamalowujemy odpowiednie piksele) ścianę w bieżącym wierzchołku i obchodzimy poddrzewo „in”. Jeśli obserwator znajduje się po stronie „out”, to poddrzewa danego wierzchołka przetwarzamy w odwrotnej kolejności. Jeśli położenie obserwatora zawiera się w płaszczyźnie ściany, to kolejność przeszukiwania poddrzew jest dowolna.

Dla położen A , B , C obserwatora pokazanych na rysunku algorytm namaluje kolejno

A : $3a$, $4b$, 2 , $3b$, 1 , $4a$,

B : $4a$, 1 , $3a$, $4b$, 2 , $3b$,

C : $4a$, 1 , $3b$, 2 , $3a$, $4b$.

Budowa drzewa BSP przejmie rolę sortowania, rozstrzygania konfliktów widoczności i dzielenia ścian z algorytmu malarza. Przyjemną cechą tego algorytmu jest fakt, że drzewo wystarczy zbudować tylko raz (chyba, że scena ulega zmianie) i można wykonać wiele obrazów sceny, widzianej z różnych punktów.

9.3.3. Algorytm BSP wyznaczania cieni

Odpowiedni podział przestrzeni, poprzez konstrukcję drzew BSP, może być użyty do wyznaczenia fragmentów ścian oświetlonych przez ustalone punktowe źródło światła. Drzewa BSP w tym algorytmie spełniają dwie różne funkcje:

- Służą do rozstrzygania widoczności w scenie (w taki sposób jak w algorytmie opisanym wcześniej),
- Służą do reprezentowania tzw. bryły cienia, która jest zbiorem punktów zasłoniętych przez co najmniej jedną ścianę. Bryłę tę reprezentuje się w pewnym uproszczeniu, które jest możliwe dzięki odpowiedniemu uporządkowaniu ścian.

Przyjmijmy założenie, że wszystkie ściany są wypukłe, co upraszcza implementację (w razie czego można np. dokonać triangulacji ścian). Pierwszy etap algorytmu, niezależny od punktów, w których znajduje się obserwator i źródła światła, polega na zbudowaniu zwykłego drzewa BSP.

Etap drugi polega na budowaniu reprezentacji bryły cienia, z jednoczesnym wyznaczaniem oświetlonych części ścian. W etapie tym przeszukujemy drzewo BSP sceny w kolejności odwrotnej niż podczas rysowania, przyjmując położenie obserwatora w punkcie położenia źródła światła. W ten sposób otrzymujemy najpierw ścianę lub fragment ściany, który jest cały oświetlony, a następnie ściany, które mogą być zasłonięte od światła tylko przez fragmenty wyprowadzone wcześniej. Każdy taki fragment ściany jest wielokątem wypukłym, bo jest on częścią wspólną wypukłej ściany i wypukłej komórki binarnego podziału przestrzeni.

Drzewo BSP bryły cienia jest początkowo puste. Procedura, która je tworzy, przetwarza kolejne wypukłe wielokąty. Każdy taki wielokąt jest podstawą nieograniczonego ściętego ostrosłupa, o wierzchołku w punkcie położenia źródła światła. Płaszczyzny, w których leżą brzegi komórek podziału, są wyznaczone przez punkt położenia źródła światła i odcinki — krawędzie bieżącego fragmentu ściany (a więc jeśli jest on np. trójkątem, to wstawimy trzy ściany, po jednej dla każdego boku).

Przed rozbudowaniem drzewa BSP bryły cienia możemy podzielić bieżący fragment ściany na część oświetloną i zacienioną, w ten sposób, że dzielimy go płaszczyznami podziału odpowiednich komórek (tak, jakbyśmy wstawiali fragment do drzewa). Dla każdej komórki dysponujemy informacją, czy jej punkty są oświetlone, czy nie.

9.3.4. Algorytm z buforem głębokości

Algorytm z buforem głębokości (tzw. *z-buforem*), powszechnie implementowany w sprzęcie, to tzw. algorytm brutalnej siły, która w tym przypadku polega na użyciu dużej ilości pamięci. W algorytmie tym mamy prostokątną tablicę liczb, o wymiarach równych wymiarom szerokości i wysokości obrazu w pikselach. Liczby te są rzeczywiste, ale najczęściej, dla przyspieszenia obliczeń są one reprezentowane w postaci stałopozycyjnej, zwykle 16-, 24- lub 32-bitowej.

Algorytm:

1. Przypisz wszystkim elementom *z*-bufora wartość, która reprezentuje nieskończoność, albo największą dopuszczalną odległość od obserwatora.
2. Kolejno dla każdej ściany, narysuj jej rzut (tj. dokonaj jego rasteryzacji za pomocą algorytmu przeglądania liniami poziomymi). Dla każdego piksela, który należy do rzutu ściany, należy przy tym obliczyć współrzędną *z* (tj. głębokość) punktu w układzie obserwatora. Kolor danej

ściany przypisujemy pikselowi tylko wtedy, gdy obliczona współrzędna z jest mniejsza niż aktualna zawartość bufora głębokości. Jednocześnie z przypisaniem koloru należy uaktualnić zawartość bufora.

Zaletami algorytmu są prostota, w tym łatwość implementowania algorytmu w sprzęcie, i duża niezawodność (odporność na błędy np. zaokrągleń — skutki takich błędów to pojedyncze piksele o złym kolorze, podczas gdy w algorytmach wyrafinowanych taki błąd może spowodować całkowite załamanie obliczeń). Niezawodność tego algorytmu jest być może ważniejsza od szybkości jego działania, jeśli rozpatrujemy przyczyny szerokiego rozpowszechnienia sprzętowych implementacji tego algorytmu.

Do wad tego algorytmu należy zaliczyć duże zapotrzebowanie na pamięć (np. 600kB dla obrazka o wymiarach 640×480 pikseli, przy 16 bitach bufora głębokości na każdy piksel). Wada ta skutecznie powstrzymywała stosowanie tego algorytmu na pecetach w czasach, gdy w powszechnym użyciu był system DOS, ale teraz jest mało istotna. Inne wady to: trudności w walce ze zjawiskiem intermodulacji (tzw. aliasu), niezbyt łatwa implementacja w celu wyświetlania powierzchni krzywoliniowych i niemożność tworzenia obrazów brył CSG na podstawie drzewa CSG (tj. bez jawnego wyznaczenia reprezentacji takiej bryły). Ponadto pewne typy sprzętu (akceleratorów graficznych) ukrywają zawartość z -bufora (nie można jej odczytać po wykonaniu obrazu, a informacja ta bywa nadzwyczaj cenna, np. do wyznaczenia cieni), a prawie żaden sprzęt nie jest w stanie wyświetlać powierzchni krzywoliniowych (zamiast tego wyświetla się zwykle duże ilości trójkątów).

Algorytm widoczności dla powierzchni Béziera. Aby narysować obraz powierzchni Béziera można (podobnie jak inne powierzchnie zakrzywione) przybliżyć ją dostatecznie dużą liczbą trójkątów i wyświetlić je; metoda opisana niżej prowadzi do otrzymania obrazu, którego dokładność zależy tylko od jego rozdzielczości (a nie od parametrów takich jak liczba trójkątów).

Mając siatkę kontrolną płata Béziera

1. Dokonaj rzutowania punktów kontrolnych,
2. Jeśli największa odległość obrazów tych punktów jest nie większa niż średnica piksela, to oblicz dowolny punkt płata i wektor normalny w tym punkcie, a następnie zbadaj widoczność w z -buforze, oblicz kolor i przypisz go pikselowi,
3. W przeciwnym razie podziel płat na mniejsze fragmenty (za pomocą algorytmu de Casteljau) i wywołaj procedurę rekurencyjnie.

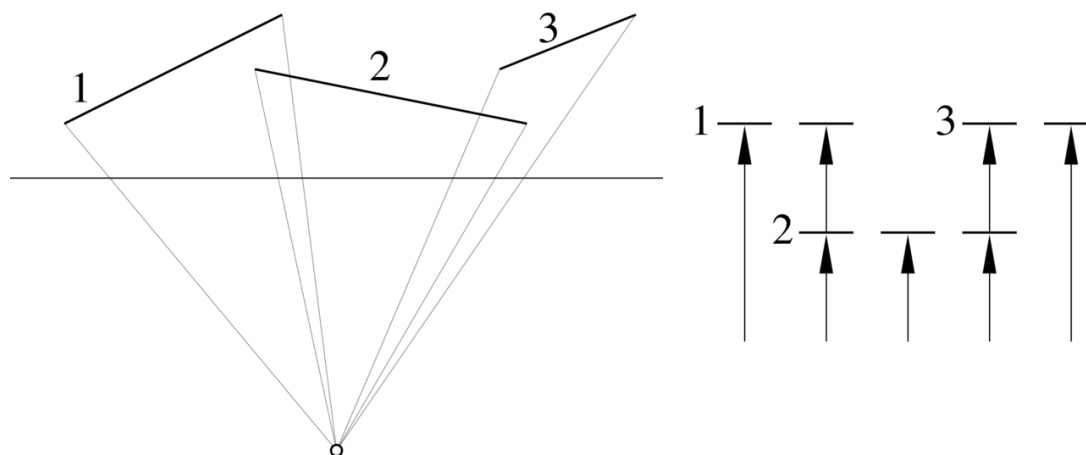
Głębokość podziału płata w tym algorytmie adaptacyjnie dopasowuje się do rozdzielczości obrazu. Okazuje się, że podczas działania tego algorytmu łatwo może się zdarzyć, że w wyniku błędów zaokrągleń na obrazie pozostają dziurki (to jest spowodowane zaokrągleniem współrzędnych obrazu punktu do całkowitych współrzędnych piksela). Ponadto zdarzają się błędy rozstrzygnięcia widoczności, spowodowane arbitralnym wyborem punktu płata, którego głębokość jest badana w teście widoczności. Uodpornienie algorytmu na takie błędy jest nietrywialne, ale możliwe.

9.3.5. Algorytm przeglądania liniami poziomymi

Główną motywacją tego algorytmu była chęć obniżenia kosztu pamięciowego algorytmu z buforem głębokości (ta motywacja ma już znaczenie tylko historyczne). Algorytm przeglądania liniami poziomymi ma wiele wariantów, które umożliwiają m.in. tworzenie obrazów cieni oraz obrazów brył CSG bez wyznaczania jawnej ich reprezentacji. Poniżej jest opisana wersja podstawowa. Jest w niej przyjęte założenie, że ściany są płaskie i nie przenikają się.

1. Zrzutuj wierzchołki ścian na obraz; dla każdej krawędzi znajdź wierzchołek, którego obraz ma mniejszą współrzędną y , a ponadto, zbadaj, czy obraz wnętrza ściany leży po lewej, czy po prawej stronie (krawędzie, których obraz jest poziomy, pomijamy).
2. Posortuj tablicę krawędzi w kolejności rosnących mniejszych współrzędnych y .

3. Utwórz początkowo pustą tablicę krawędzi aktywnych.
4. Dla kolejnych wartości y odpowiadających poziomym liniom rastra obrazu, wykonaj następujące czynności:
 - a) Wyrzuć z tablicy krawędzi aktywnych te, których rzut jest rozłączny z bieżącą poziomą linią rastra.
 - b) Dołącz do tablicy krawędzi aktywnych te, których obraz przecina się z bieżącą linią poziomą.
 - c) Oblicz współrzędną x (ekranową) i z (głębokość) punktu przecięcia każdej krawędzi aktywnej z bieżącą poziomą linią rastra i posortuj tablicę krawędzi aktywnych w kolejności rosnących x .
 - d) Utwórz (początkowo pustą) listę aktywnych ścian; wstaw do niej ściany, których lewa krawędź ma najmniejszą współrzędną x , porządkując je w kolejności rosnących głębokości.
 - e) Dla pierwszej w liście ściany wyznacz jej prawą krawędź aktywną i współrzędną x tej krawędzi (a), lub ścianę nieaktywną o najmniejszej współrzędnej x lewej krawędzi (b) (zależnie od tego, co jest mniejsze).
 - f) Narysuj poziomy odcinek w kolorze pierwszej w liście aktywnej ściany; w przypadku (a) usuń ścianę z listy.
 - g) Usuń z listy wszystkie ściany, których prawa krawędź ma współrzędną x mniejszą lub równą wsp. x końca narysowanego odcinka.
 - h) Jeśli lista jest niepusta, to dołącz do listy ścian aktywnych ściany, których współrzędną x lewej krawędzi jest mniejsza lub równa współrzędnej x ostatnio narysowanego odcinka. W przeciwnym razie wstaw do listy nieprzetworzone jeszcze ściany, których lewe krawędzie mają najmniejszą współrzędną x .
 - i) Uporządkuj listę i idź do kroku 4.5, chyba, że lista jest pusta.



Rysunek 9.7. Działanie algorytmu przeglądania liniami poziomymi.

Wariant algorytmu dla konstrukcyjnej geometrii brył korzysta z uproszczenia zadania przez obniżenie wymiaru; przecięcia płaszczyzny wyznaczonej przez położenie obserwatora i bieżącą przeglądaną linią rastra z wielościennymi prymitywami to wielokąty. Zamiast wykonywać działania CSG na wielościannach, można zatem wyznaczyć przecięcia, sumy, różnice lub dopełnienia wielokątów, a następnie poddać je regularyzacji. Jest to oczywiście zadanie prostsze (ale takie zadanie trzeba rozwiązać dla każdej poziomej linii rastra).

Podobne uproszczenie dotyczy wyznaczania cieni; po wyznaczeniu widocznych odcinków można znaleźć ich części oświetlone. Również to zadanie jest prostsze od wyznaczania oświetlonych części wielokątów, na których leżą te odcinki.

9.3.6. Algorytm cieni z buforem głębokości

Algorytm z buforem głębokości umożliwia narysowanie obrazu z cieniami. Jest to wprowadzić dość kłopotliwe, ale można to zrobić za pomocą sprzętu (np. działającego w standardzie OpenGL, konieczne są pewne rozszerzenia) i wtedy możliwa jest animacja w czasie rzeczywistym, tj. wyświetlanie kilkudziesięciu obrazów na sekundę, przedstawiających poruszające się obiekty. Są jednak pewne ograniczenia: źródło światła musi być reflektorem, emitującym wiązkę promieni świetlnych w pewnym stożku (nie może to być np. nieosłonięta świeca w pokoju), ponadto implementacja tego algorytmu może „zawłaszczyć” mechanizm teksturowania (ta trudność jest do ominięcia, jeśli istnieje możliwość bezpośredniego programowania karty graficznej np. w języku GLSL).

Algorytm działa tak: w pierwszym kroku rysujemy scenę, umieszczając obserwatora w punkcie położenia źródła światła; rzutnia jest prostopadłą do osi stożka światła. Po tym etapie nie jest potrzebny obraz, tylko zawartość buforu głębokości.

W etapie drugim zawartość buforu głębokości otrzymaną w pierwszym etapie wykorzystujemy do określenia tekstury trójwymiarowej; dla dowolnego punktu w przestrzeni możemy określić, czy jest on oświetlony, porównując jego współrzędną z w układzie źródła światła (tj. w układzie kamery wykorzystywanym podczas pierwszego etapu) z zawartością buforu głębokości. Wartość tej tekstury, jedna z dwóch, umożliwia wybranie właściwego koloru piksela.

W implementacji konieczne jest przeciwdziałanie skutkom błędów zaokrągleń, które mogą spowodować zacienienie obiektu przez samego siebie. Dodatkową wadą jest ograniczona rozdzielczość obrazu wykonanego w pierwszym etapie. Brzeg cienia jest w efekcie „schodkowy” — na końcowym obrazie są widoczne piksele obrazu z położenia źródła światła. Przykładowa implementacja jest w dodatku B.

10. Podstawowe modele oświetlenia i metody cieniowania

*Aby słońce nie odbijało się
w przyrządach celowniczych, należy je
przedtem okopcić (przyrządy, nie słońce).*

ANNA

10.1. Oświetlenie

Modele oświetlenia opisane w tym miejscu są heurystyczne. Zamiast na podstawach fizycznych, opierają się one na wzorach dobranych tak, aby otrzymane obrazy „dobrze wyglądały”. Do pełnego realizmu (takiego jak na fotografii) modele te są niewystarczające, za to umożliwiają przeprowadzanie obliczeń w krótkim czasie.

10.1.1. Oświetlenie bezkierunkowe

Najprostszy model oświetlenia opiera się na założeniu, że przestrzeń, w której znajdują się rysowane obiekty, jest wypełniona światłem, które dochodzi ze wszystkich kierunków z taką samą intensywnością. Założenie to jest nieodległe od prawdy, jeśli scena jest umieszczona w pomieszczeniu o jasnych (np. białych) matowych ścianach i obiekty te są oświetlone przede wszystkim światłem odbitym od ścian. Wzór, który służy do obliczenia intensywności światła odbitego w stronę obserwatora (czyli do obliczenia koloru pikseli) ma postać

$$I = I_{\text{amb}} a.$$

Czynnik I_{amb} opisuje intensywność światła rozproszonego w otoczeniu i jest funkcją długości fali świetlnej. Czynniki a to *albedo* powierzchni przedmiotu, które zależy również od długości fali, a także od punktu powierzchni, która nie musi być jednobarwna. Często funkcje długości fali, takie jak I , I_{amb} i a , są reprezentowane za pomocą trzech wartości odpowiadających barwie czerwonej, zielonej i niebieskiej. W niektórych zastosowaniach to nie wystarczy i trzeba użyć większej liczby próbek, tj. wartości funkcji opisującej oświetlenie dla więcej niż trzech długości fali świetlnej (nawet jeśli potem na monitorze obraz jest wyświetlany za pomocą luminoforów o trzech barwach).

Obrazy wykonane z użyciem tego modelu oświetlenia są całkowicie pozbawione plastyki; jeśli obiekty są jednobarwne (bez tekstury) to wyglądają jak plamy o stałym kolorze.

10.1.2. Odbicie lambertowskie

Wzór

$$I = \left(I_{\text{amb}} + \sum_i I_{\text{di}} v_i \max(0, \cos \angle(\mathbf{n}, \mathbf{L}_i)) \right) a$$

opisuje tzw. lambertowski model odbicia światła, nazwany tak od J.H. Lamberta, który zajmował się nim w 1760r. Oprócz światła rozproszonego w otoczeniu, o intensywności I_{amb} , mamy tu pewną liczbę punktowych źródeł światła. Intensywność oświetlenia powierzchni przez i -te źródło jest opisana przez funkcję I_{di} , zależną od długości fali świetlnej, a także od odległości źródła światła od oświetlanego punktu, jeśli ta odległość jest skończona. Czynniki v_i jest równy 1 lub 0, co zależy od tego, czy między rozpatrywanym punktem powierzchni i źródłem światła jest jakiś obiekt rzucający cień, czy nie.

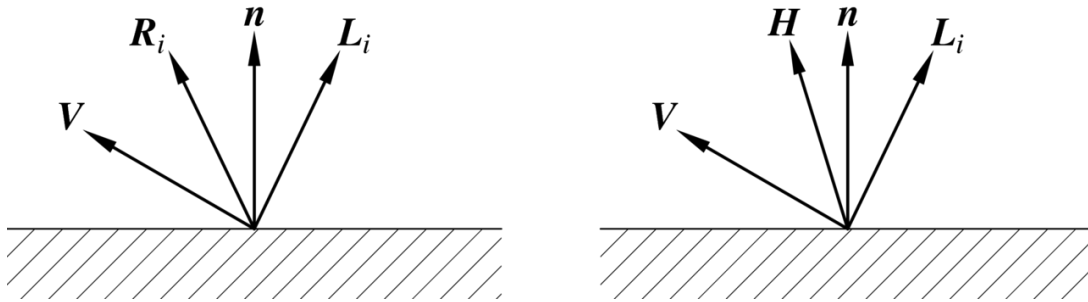
Wektor $\mathbf{n}$ jest wektorem normalnym powierzchni, a wektor $\mathbf{L}_i$ określa kierunek do źródła światła. Kosinus kąta między tymi wektorami jest obliczany przez obliczenie iloczynu skalarnego.

Lambertowski model oświetlenia opiera się na założeniu, że obiekty są idealnie matowe, a zatem światło odbite od powierzchni rozchodzi się we wszystkich kierunkach nad powierzchnią z taką samą intensywnością. Nie ma w tym modelu możliwości otrzymania na obrazie odbłasków, które można zaobserwować na powierzchniach błyszczących.

10.1.3. Model Phong

Odblaski są możliwe do uzyskania w modelu opracowanym w 1975r. przez Phong Buoi-Tuonga. Wzór używany do obliczenia intensywności światła odbitego od powierzchni w kierunku obserwatora w tym modelu ma postać

$$I = \left(I_{\text{amb}} + \sum_i I_{di} v_i \max(0, \cos \angle(\mathbf{n}, \mathbf{L}_i)) \right) a + \sum_i I_{di} v_i W(\angle(\mathbf{n}, \mathbf{L}_i)) \cos^m \angle(\mathbf{R}_i, \mathbf{v}).$$



Rysunek 10.1. Wektory występujące w modelu odbicia światła Phong.

Od wzoru opisującego model lambertowski wzór ten różni się obecnością składnika opisującego odbłaski. Funkcja W uwzględnia zależność intensywności odbłasku od kąta padania światła na powierzchnię i często przyjmowana jest tu funkcja stała. Intensywność światła w odbłasku zależy od kąta między wektorem $\mathbf{v}$ opisującym kierunek do obserwatora i wektorem $\mathbf{R}_i$, którego kierunek odpowiada odbiciu światła padającego z kierunku wektora $\mathbf{L}_i$ w idealnym lustrze. Wektor $\mathbf{R}_i$ można obliczyć na podstawie wzoru

$$\mathbf{R}_i = 2\langle \mathbf{n}, \mathbf{L}_i \rangle \mathbf{n} - \mathbf{L}_i,$$

do którego należy podstawić wektor *jednostkowy* $\mathbf{n}$. Kosinus kąta między wektorami $\mathbf{v}$ i $\mathbf{R}_i$ jest podnoszony do potęgi m . Liczba m zależy od „stopnia wypolerowania” powierzchni i jest tym większa im bardziej błyszcząca ma być ta powierzchnia. Przyjmowane w praktyce wartości parametru m są w granicach od kilku do kilkuset. Na ogół m jest liczbą całkowitą, ponieważ wtedy m -tą potęgę można obliczyć kosztem $O(\log_2 m)$ mnożeń.

Warto zwrócić uwagę na to, że albedo powierzchni (opisane przez funkcję a) nie wpływa na barwę światła w odbłaskach. Rzeczywiście, jeśli przyjrzymy się odbłaskom na powierzchni fizycznie istniejących przedmiotów, to najczęściej mają one barwę światła padającego na powierzchnię.

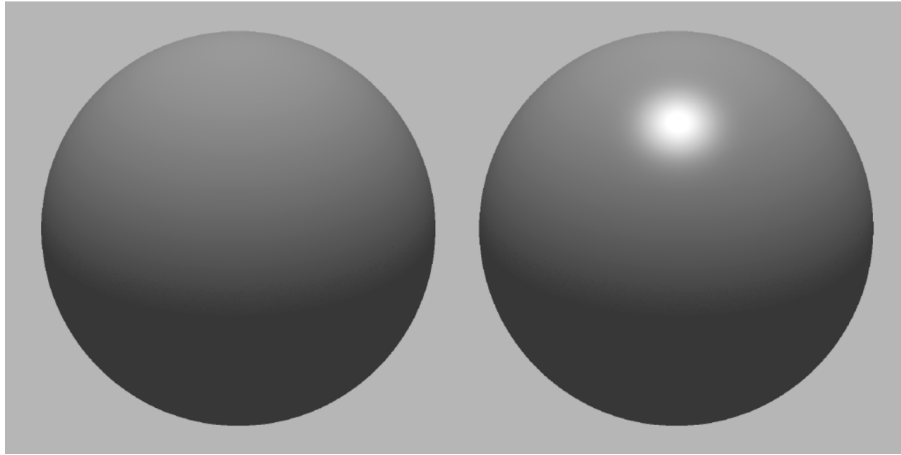
Zamiast obliczać wektor $\mathbf{R}_i$, a następnie $\cos \angle(\mathbf{R}_i, \mathbf{v}) = \langle \mathbf{R}_i, \mathbf{v} \rangle / (\|\mathbf{R}_i\|_2 \|\mathbf{v}\|_2)$, możemy wykonać rachunek przybliżony. Mamy $\angle(\mathbf{R}_i, \mathbf{v}) \approx 2\angle(\mathbf{n}, \mathbf{H})$, gdzie

$$\mathbf{H} = \frac{\mathbf{L}_i + \mathbf{v}}{\|\mathbf{L}_i + \mathbf{v}\|_2}$$

(wektory $\mathbf{L}_i$ oraz $\mathbf{v}$ mają długość 1). Równość zachodzi w przypadku, gdy wektory $\mathbf{L}_i$, $\mathbf{n}$ i $\mathbf{v}$ leżą w jednej płaszczyźnie. Następnie mamy przybliżoną równość

$$\cos^m \angle(\mathbf{R}_i, \mathbf{v}) \approx \langle \mathbf{n}, \mathbf{H} \rangle^{2m}.$$

W implementacjach często stosuje się tablicowane z dostatecznie małym krokiem wartości funkcji $f(t) = t^{2m}$, co znacznie przyspiesza obliczenia (kosztem pamięci).



Rysunek 10.2. Porównanie odbicia światła Lamberta i Phong'a.

Należy podkreślić, że model Phong'a jest czysto empiryczny, tj. nie ma on podstaw fizycznych, ale dobierając odpowiednio jego parametry, można osiągnąć dosyć dobre przybliżenie wyglądu różnych materiałów na obrazie. Najczęściej obiekty narysowane za pomocą tego modelu mają wygląd nieco „plastikowy”. Jest to spowodowane przez dość dobre przybliżenie skutków odbicia światła od przedmiotów wykonanych z plastiku.

10.2. Cieniowanie

Cieniowanie jest sposobem obliczania barwy pikseli należących do zrasteryzowanego wielokąta (najczęściej trójkąta) na podstawie informacji związanej z jego wierzchołkami. Może ono mieć na celu stworzenie złudzenia, że obraz zbioru trójkątów przedstawia gładką powierzchnię krzywoliniową.

10.2.1. Cieniowanie stałe

Najprostsza metoda cieniowania polega na obliczeniu barwy jednego piksela i nadaniu tej samej wartości wszystkim pikselom, które należą do obrazu np. danego wielokąta. Metoda ta

jest najprostsza i najszybsza, ale daje obrazy z wyraźnie widocznymi wspólnymi krawędziami ścian, co w sytuacji gdy te ściany stanowią przybliżenie gładkiej powierzchni zakrzywionej jest niepożądane (to jest tzw. efekt Macha).

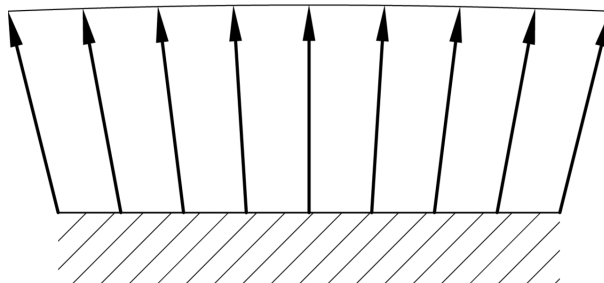
10.2.2. Cieniowanie Gourauda

Metoda cieniowania Gourauda jest stosowana najczęściej do trójkątów — inne wielokąty są w celu cieniowania triangulowane. Metoda ta polega na obliczeniu (przy użyciu dowolnego modelu oświetlenia) barw wierzchołków trójkąta, a następnie barwa każdego piksela jest obliczana w drodze interpolacji liniowej barw wierzchołków (wygodnie jest to opisać przy użyciu współrzędnych barycentrycznych na płaszczyźnie).

Zastosowanie cieniowania metodą Gourauda powoduje usunięcie lub znaczne osłabienie efektu Macha — barwa pikseli w pobliżu wspólnej krawędzi trójkątów zmienia się w sposób ciągły — ale zawodzi w przypadku, gdy zastosowany model oświetlenia uwzględnia odbicie zwierciadlane (tak jak model Phong). Można bowiem zgubić odbłaski, które leżą wewnątrz obrazu trójkąta, a także otrzymać kanciaste odbłaski na powierzchni, która ma wyglądać jak gładka.

10.2.3. Cieniowanie Phong

W metodzie cieniowania opracowanej przez Phong Bui-Tuonga określamy we wszystkich wierzchołkach trójkątnych ścian wektor normalny powierzchni, którą chcemy przybliżać za pomocą tych ścian. Następnie, dla każdego piksela, obliczamy „wektor normalny” powierzchni w drodze interpolacji wektorów normalnych w narożnikach i po unormowaniu podstawiamy go do wzoru odpowiadającego wybranemu modelowi oświetlenia. Metoda ta jest bardzo skuteczna, ale wymaga znacznie więcej obliczeń niż cieniowanie Gourauda.

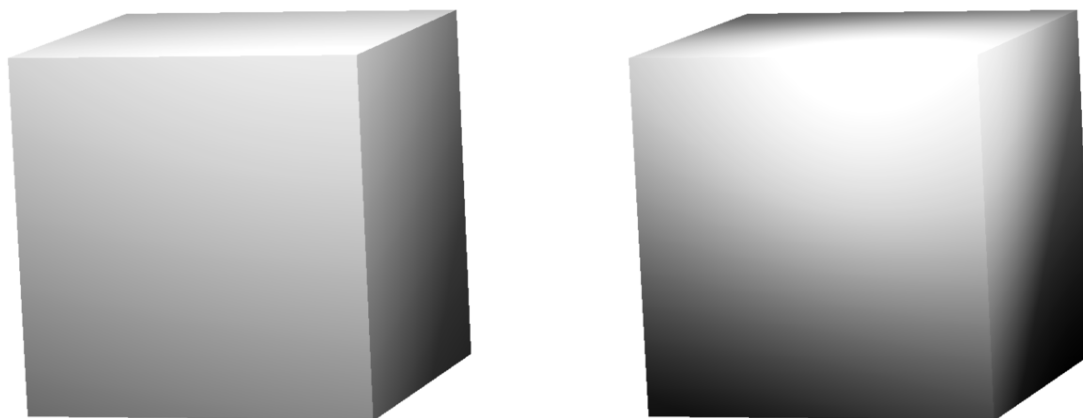


Rysunek 10.3. Interpolacja wektora normalnego w cieniowaniu Phong.

„Wektor normalny” powierzchni w wierzchołkach trójkąta może być przyjmowany na różne sposoby. Jeśli trójkąty przybliżają powierzchnię znaną (np. płat Béziera), to można obliczyć wektor normalny tej powierzchni w punkcie, który odpowiada wierzchołkowi. Jeśli trójkąty przybliżają powierzchnię daną w postaci niejawnej, tj. równaniem $f(x, y, z) = 0$, to najlepiej jest (jeśli to możliwe) obliczyć gradient funkcji f . Jeśli dysponujemy tylko trójkątami, to można przyjąć wektor, który jest średnią arytmetyczną wektorów normalnych wszystkich trójkątów, do których należy ten wierzchołek. Zwróćmy uwagę, że wyświetlając wielokąt w OpenGL-u, możemy dla każdego wierzchołka podać inny „wektor normalny” (o dowolnym kierunku).

Cieniowanie jest najczęściej realizowane za pomocą sprzętu, np. podsystemu graficznego stacji roboczej, albo akceleratora w pececie. Metoda cieniowania jest wtedy ukryta przed programem i różna (Gouraud albo Phong) w różnych urządzeniach realizujących ten sam interfejs programisty (np. OpenGL). W tanim sprzęcie do użytku domowego cieniowanie Phong'a to jeszcze rzadkość, choć sytuacja zmienia się. Nowe akceleratory graficzne mogą wykonywać programy wprowadzone przez aplikację. W zamierzeniu umożliwia to stosowanie różnych sposobów

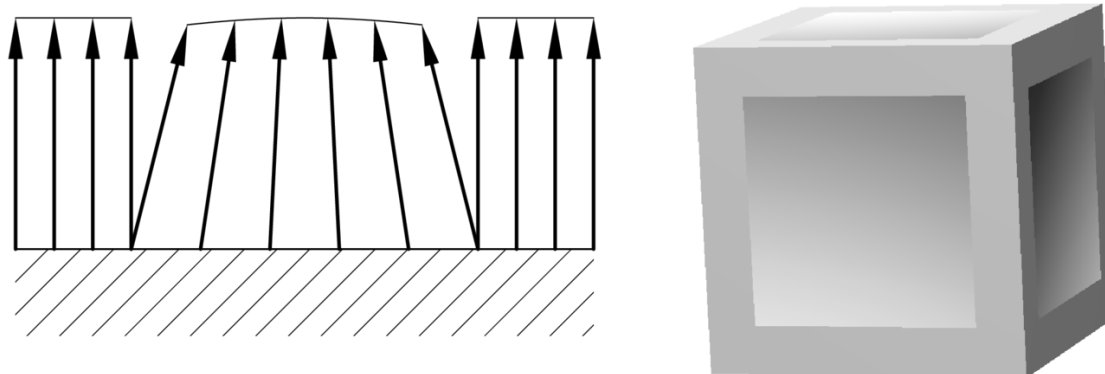
przetwarzania tekstury nakładanej na wyświetlane powierzchnie. W szczególności można używać własnych procedur cieniowania, a także nakładać opisane dalej tekstury odkształceń.



Rysunek 10.4. Sześcian cieniowany metodą Phongą.

10.2.4. Tekstura odkształceń

W każdym modelu oświetlenia uwzględniającym kierunek padania światła na powierzchnię zasadniczą rolę odgrywa wektor normalny, i jak wynika z rozważań nad cieniowaniem Phongą, kolorując obraz powierzchni można trochę oszukiwać zmysł wzroku, wybierając wektor niezupełnie normalny. Stąd już tylko krok do określania wektora normalnego za pomocą arbitralnej funkcji, dzięki czemu daje się symulować niewielkie odkształcenia i chropowatość powierzchni.



Rysunek 10.5. Tekstura odkształceń.

11. Śledzenie promieni

Śledzenie promieni (ang. *ray tracing*) jest pierwszą metodą wizualizacji obiektów trójwymiarowych, od której ludzie zaczęli mówić o „fotorealistycznych” obrazach utworzonych przez komputer. Można ją scharakteryzować za pomocą następujących uwag:

- Metoda ta jest algorytmem widoczności (przestrzeni obrazu), co prawda niezbyt sprawnym i dlatego często bywa on wspomagany innymi algorytmami widoczności.
- Jest to również algorytm wyznaczania cieni.
- Najbardziej charakterystyczna jest możliwość symulowania wielokrotnych odbić zwierciadlanych i załamania światła. Bardziej ograniczone są możliwości otrzymywania półcieni.
- Istnieje możliwość stosunkowo łatwego połączenia metody śledzenia promieni z konstrukcyjną geometrią brył i z wizualizacją objętościową (można więc tworzyć obrazy dymu, mgły itd.). Klasa dopuszczalnych obiektów, które można wizualizować tą metodą jest bardzo obszerna.
- Łatwo można przeciwdziałać zjawisku aliasu, możliwe są też efekty specjalne, takie jak symulowanie głębi ostrości obiektywu i inne.

Ceną za to wszystko jest duża ilość obliczeń wykonywanych w tym algorytmie, czyli długi (rzędu kilku sekund do nawet godzin) czas tworzenia obrazu. Nie jest to więc algorytm nadający się do wykorzystania w programach interakcyjnych jako główna metoda wizualizacji trójwymiarowych scen.

11.1. Zasada działania algorytmu

W metodzie śledzenia promieni kolejno dla każdego piksela należy obliczyć jego barwę, tj. intensywność światła, która dochodzi do obserwatora „przez ten piksel”. W tym celu należy przeszukać tzw. **drzewo promieni**. Korzeniem tego drzewa jest tzw. **promień pierwotny**, czyli półprosta o początku w położeniu obserwatora, „przechodząca przez piksel”. Dla danego promienia należy wyznaczyć punkt przecięcia z powierzchnią obiektu, najbliższy obserwatora. Punkt ten jest początkiem tzw. **promieni wtórnych**, których może być kilka (zwykle co najmniej 1, ale bywa też kilkanaście lub nawet kilkadziesiąt).

Dla każdego z promieni wtórnych wykonuje się podobne obliczenie jak dla promieni pierwotnych. Oczywiście, istnieją ograniczenia w generowaniu promieni wtórnych, które mają na celu zapewnienie własności stopu algorytmu. Dla każdego promienia oblicza się intensywność światła niesionego przez ten promień do jego punktu początkowego. Następnie w punkcie początkowym promieni wtórnych stosuje się odpowiedni model oświetlenia w celu obliczenia intensywności światła niesionego w kierunku początku promienia, który określił ten punkt początkowy promieni wtórnych. Intensywność światła niesionego przez promień pierwotny jest używana do obliczenia barwy piksela.

Dróg, po których porusza się światło jest niewyobrażalnie dużo i dlatego należy wybrać możliwie niewielki zbiór promieni wtórnych, które będą wystarczające do obliczenia intensywności światła z dostateczną dokładnością. Wśród promieni wtórnych wyróżniamy:

promienie odbite, których kierunek wynika z odbicia promienia pierwotnego lub wtórnego niższego rzędu w sposób zwierciadlany:

$$\begin{aligned} \mathbf{v}' &= \mathbf{v} - 2\langle \mathbf{v}, \mathbf{n} \rangle \mathbf{n} && \text{jeśli } \|\mathbf{n}\|_2 = 1, \\ \mathbf{v}' &= \mathbf{v} - \frac{2}{\|\mathbf{n}\|_2^2} \langle \mathbf{v}, \mathbf{n} \rangle \mathbf{n} && \text{jeśli } \|\mathbf{n}\|_2 \neq 0. \end{aligned}$$

Uwaga 1: Można zaburzać wektor $\mathbf{n}$, nakładając na powierzchnię teksturę odkształceń.

Uwaga 2: Czasem wykonuje się też śledzenie promieni odbitych w inny sposób.

promienie załamane — dla obiektów przezroczystych. Prawo załamania światła jest opisane wzorem

$$\frac{\sin \alpha}{\sin \beta} = \frac{n_2}{n_1},$$

w którym n_1 i n_2 są współczynnikami załamania światła w poszczególnych ośrodkach. Nawiasem mówiąc, istnieją ośrodki, których współczynnik załamania światła zmienia się w sposób ciągły, np. powietrze o różnej temperaturze w różnych miejscach, lub roztwór soli o niejednorodnym stężeniu. Światło w takich ośrodkach porusza się po krzywych, wskutek czego powstają zjawiska takie jak fatamorgana.

promienie bezpośredniego oświetlenia — ich kierunki są określone przez położenia źródeł światła. Promienie te służą do uwzględnienia w modelu oświetlenia punktu, który jest ich początkiem, światła dochodzącego bezpośrednio od źródeł. Śledzenie tych promieni ma na celu wyznaczenie odległości od źródeł światła oraz wykrycie ewentualnych obiektów rzucających cień.

Program przeszukuje drzewo promieni metodą DFS. Promienie wtórne są generowane wtedy, gdy są spełnione następujące warunki:

- Przypuśćmy, że w obliczeniach intensywności światła niesionego przez promień jest obliczana zgodnie z modelem oświetlenia Phong. Dla każdego promienia obliczamy jego wagę, czyli współczynnik, który określa wpływ intensywności światła niesionego przez ten promień na ostateczną barwę piksela. Dla promieni pierwotnych współczynnik ten jest równy 1. Jeśli na przykład stosujemy model oświetlenia Phong, to generując promień wtórny, którego kierunek jest określony przez wektor L_i , obliczamy jego wagę mnożąc wagę promienia niższego rzędu przez czynnik

$$\left(a \max(0, \cos \angle(\mathbf{n}, \mathbf{L}_i)) + W(\angle(\mathbf{n}, \mathbf{L}_i)) \cos^m \angle(\mathbf{v}, \mathbf{R}_i) \right).$$

Jeśli otrzymana waga promienia jest mniejsza niż ustalona wartość progowa, to nie tracimy czasu na śledzenie tego promienia. Dla przykładu, w programie RayShade domyślna wartość progowa (odpowiedni parametr nazywa się `cutoff`) jest równa 0.002.

- Program śledzenia promieni może też ustalić arbitralne ograniczenie wysokości drzewa promieni (np. w RayShade mamy parametr `maxdepth` o domyślnej wartości 15). Ten drugi sposób gwarantuje własność stopu algorytmu, natomiast pierwszy ma większe znaczenie dla szybkości obliczeń.

11.2. Wyznaczanie przecięć promieni z obiektami

Najwięcej czasu w śledzeniu promieni, do 95%, zajmuje obliczanie punktów przecięcia promieni z powierzchniami obiektów. Wyznaczanie przecięć jest rozwiązywaniem równań (na ogół nieliniowych) i składa się z dwóch zasadniczych etapów: lokalizacji rozwiązań i obliczania ich

z dużą dokładnością. Równania te mają postać zależną od reprezentacji obiektu. Zakładamy, że promień jest reprezentowany w postaci parametrycznej:

$$\mathbf{p} = \mathbf{p}_0 + t\mathbf{v}, \quad t \geq 0.$$

— Jeśli obiekt jest powierzchnią parametryczną,

$$\mathbf{p} = \mathbf{p}(u, v), \quad (u, v) \in A,$$

(litera A oznacza dziedzinę powierzchni $\mathbf{p}$), to punkt $\mathbf{p}$, wspólny dla promienia i płata, spełnia równanie

$$\mathbf{p}(u, v) - \mathbf{p}_0 - t\mathbf{v} = \mathbf{0}.$$

Powyższe równanie jest układem trzech równań z trzema niewiadomymi, u , v oraz t . Warto zauważyć, że ze względu na niewiadomą t wszystkie te równania są liniowe, co umożliwia wyeliminowanie tej niewiadomej, tj. otrzymanie dwóch równań, w których niewiadoma ta nie występuje. Daje to możliwość opracowania szybszego i *pewniejszego* algorytmu numerycznego rozwiązywania tego układu (pamiętajmy, że mamy do czynienia z układami równań nieliniowych, dla rozwiązywania których sprawą zasadniczą jest znalezienie przybliżeń początkowych rozwiązań, oraz rozstrzygnięcie, czy *wszystkie* rozwiązania zostały znalezione).

— Jeśli powierzchnię reprezentujemy w postaci niejawnej:

$$F(\mathbf{p}) = 0 \quad (F(x, y, z) = 0),$$

to po wstawieniu parametrycznej reprezentacji promienia dostajemy równanie skalarne z jedną niewiadomą t :

$$F(\mathbf{p}_0 + t\mathbf{v}) = 0.$$

Zauważmy, że równania liniowe otrzymamy wtedy, gdy rozpatrywana powierzchnia jest płaszczyzną, daną w postaci

$$\mathbf{p} = \mathbf{p}_1 + u\mathbf{v}_1 + v\mathbf{v}_2, \quad \text{albo} \quad ax + by + cz + d = 0.$$

Wysiłek mający na celu przyspieszenie działania programu do śledzenia promieni może mieć na celu

- zmniejszenie kosztu wyznaczania punktu przecięcia promienia z obiektem, albo
- zmniejszenie liczby par promień/obiekt, dla których poszukuje się przecięć (zwróćmy uwagę, że jeśli scena składa się z wielu obiektów, to każdy promień przecina tylko nieliczne z nich).

Ilość obliczeń w śledzeniu promieni jest taka, że niezależnie od mocy obliczeniowej komputera (która rośnie w zdumiewającym, ale i tak zbyt wolnym tempie¹), do realizacji pierwszego z powyższych celów wciąż jeszcze ludzie sięgają po assembler i optymalizują na tym poziomie procedury jak mało które. Przedtem lepiej jest wybrać algorytm, który wykonuje jak najmniej działań, bo assembler to ostateczność.

Przykład 1

Porównamy dwie metody obliczania punktu przecięcia promienia ze sferą. Zakładamy, że $\|\mathbf{v}\|_2 = 1$, a ponadto reprezentacja sfery zawiera obliczony kwadrat promienia r (aby nie liczyć go ponownie dla każdego promienia).

¹ 97th rule of acquisition: "Enough... is never enough."

Metoda 1: Równania promienia i sfery mają postać

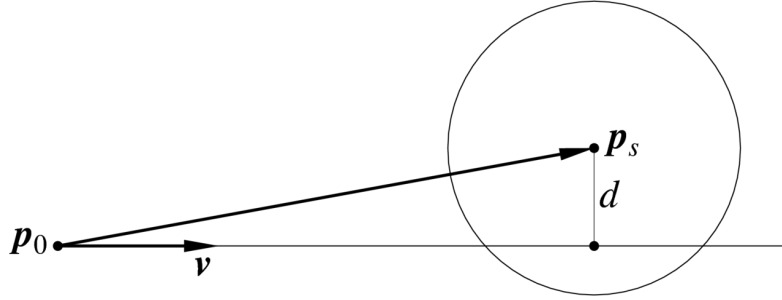
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + t \begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix},$$

$$(x - x_s)^2 + (y - y_s)^2 + (z - z_s)^2 - r^2 = 0.$$

Po podstawieniu otrzymujemy równanie $at^2 + 2bt + c = 0$, w którym

$$\begin{aligned} a &= x_v^2 + y_v^2 + z_v^2 = 1, \\ b &= x_v(x_0 - x_s) + y_v(y_0 - y_s) + z_v(z_0 - z_s), \\ c &= (x_0 - x_s)^2 + (y_0 - y_s)^2 + (z_0 - z_s)^2 - r^2, \end{aligned}$$

dalej możemy obliczyć $\Delta = b^2 - c$ i jeśli $\Delta \geq 0$ to $t_{1,2} = -b \pm \sqrt{\Delta}$. Przed sprawdzeniem, czy istnieją rozwiązania, trzeba wykonać 7 mnożeń i 9 dodawań, a potem obliczyć 1 pierwiastek i wykonać 2 dodawania.



Rysunek 11.1. Obliczanie punktów przecięcia promienia ze sferą.

Metoda 2: Jeśli $\|v\|_2 = 1$, to możemy obliczyć

$$d = \|p_s - p_0\|_2 \sin \varphi = \|(p_s - p_0) \wedge v\|_2$$

(liczymy $x = (p_s - p_0) \wedge v$, a następnie $d^2 = \langle x, x \rangle$ i nie wyciągamy pierwiastka). Przed sprawdzeniem, czy istnieją rozwiązania (czyli porównaniem d^2 i r^2) wystarczy wykonać 9 mnożeń i 8 dodawań, dalej liczymy

$$e = \langle p_s - p_0, v \rangle, \quad f = \sqrt{r^2 - d^2}, \quad t_{1,2} = e \pm f.$$

Jeśli rozwiązania istnieją, to dokończenie ich obliczania wymaga jeszcze 3 mnożeń i 3 dodawań i jednego pierwiastkowania. Często jednak interesuje nas tylko szybkie zbadanie, czy promień przecina się ze sferą, jeśli wykorzystujemy sfery jako bryły otaczające bardziej skomplikowane obiekty.

Przykład 2

Należy znaleźć przecięcie promienia z trójkątem (lub ogólniej, z płaskim wielokątem). Aby znaleźć punkt wspólny promienia z płaszczyzną trójkąta, wystarczy rozwiązać układ równań liniowych, ale znacznie trudniejszym problemem jest rozstrzygnięcie, czy punkt wspólny promienia i płaszczyzny należy do trójkąta.

Dla trójkąta można użyć następującego algorytmu. W pierwszym kroku dokonujemy takiej zmiany układu współrzędnych, aby promień pokrywał się z dodatnią lub ujemną półosią z .

W tym celu do wierzchołków trójkąta dodajemy odpowiednie przesunięcie, a następnie konstruujemy odbicie Householdera, które przekształci wektor kierunkowy promienia na wersor osi z lub wektor do niego przeciwny (zakładamy, że wektor kierunkowy promienia zawsze jest jednostkowy).

W drugim kroku rozwiązujemy układ równań z macierzą utworzoną ze współrzędnych w nowym układzie wierzchołków trójkąta:

$$\begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

W wyniku otrzymujemy współrzędne barycentryczne a_0, a_1, a_2 punktu przecięcia prostej, na której leży promień z płaszczyzną trójkąta. Punkt przecięcia należy do trójkąta, jeśli wszystkie te współrzędne są nieujemne.

Punkt przecięcia w przestrzeni możemy obliczyć ze wzoru $\mathbf{p} = a_0\mathbf{p}_0 + a_1\mathbf{p}_1 + a_2\mathbf{p}_2$. Następnie obliczamy parametr promienia $t = \langle \mathbf{v}, \mathbf{p} - \mathbf{p}_o \rangle$ i badamy, czy $t > 0$.

Współrzędne barycentryczne możemy wykorzystać do cieniowania (np. metodą Phongą) lub do obliczenia współrzędnych tekstury nakładanej na trójkąt.

Aby obliczyć przecięcie promienia z dowolnym wielokątem płaskim, możemy w podobny sposób sprowadzić zadanie do problemu dwuwymiarowego. Ze względów praktycznych (m.in. dla wygody cieniowania) wielokąty dowolne warto zastąpić odpowiednimi trójkątami przed przystąpieniem do śledzenia promieni i skorzystać z procedury dla trójkątów.

Przykład 3

Jedną z największych zalet śledzenia promieni jest możliwość „dorabiania” obsługi nietypowych obiektów. Możemy określić „rurkę Béziera”; jest to powierzchnia składająca się z punktów położonych w danej odległości r od ustalonej krzywej Béziera $\mathbf{p}$. Nie znam „gotowego” programu, który dopuszcza takie obiekty, a potrzebowałem przedstawić je na obrazkach. Obrazy takich powierzchni są na rysunku 1.6. Do ich utworzenia posłużyła mi procedura, którą tu opiszę.

Krzywa stanowiąca „oś” rurki jest stopnia n i jej reprezentacją jest ciąg punktów kontrolnych $\mathbf{p}_0, \dots, \mathbf{p}_n$. Promień jest dany za pomocą początku $\mathbf{q}$ i jednostkowego wektora $\mathbf{v}$. Przypuśćmy, że punkt $\mathbf{q}$ jest początkiem układu współrzędnych, zaś wektor $\mathbf{v}$ jest wersorem osi z . Wtedy mamy wyznaczyć zbiór liczb s leżących w przedziale $[0, 1]$, takich że punkt $\mathbf{p}(s)$ leży w odległości r od osi z . Jeśli funkcje $x(s)$, $y(s)$ i $z(s)$ są wielomianami opisującymi współrzędne krzywej $\mathbf{p}$, to mamy do wyznaczenia zbiór rozwiązań równania

$$x^2(s) + y^2(s) - r^2 = 0.$$

Jest to równanie algebraiczne stopnia $2n$; współczynniki wielomianu po lewej stronie w bazie Bernsteina są dosyć łatwe do obliczenia (procedura mnożenia wielomianów danych w tej bazie jest opisana w książce *Podstawy modelowania krzywych i powierzchni*). Po znalezieniu tych współczynników można zastosować procedurę numeryczną opartą na rekurencyjnym połowieniu dziedziny krzywej oraz na metodzie Newtona. Pozostają dwa dodatkowe problemy: na ogół promień nie spełnia przyjętych założeń. Ponadto należy wyznaczyć punkt przecięcia i wektor normalny (na podstawie którego będą wyznaczane promienie odbite i oświetlenie).

Aby rozwiązać pierwszy problem wystarczy przedstawić krzywą $\mathbf{p}$ w takim układzie współrzędnych, w którym promień pokrywa się z dodatnią lub ujemną półosią z . Wystarczy dokonać przesunięcia, które przekształci punkt $\mathbf{q}$ na początek układu, a następnie odbicia Householdera, które przekształci wektor $\mathbf{v}$ na wektor $\pm \mathbf{e}_3$. Przekształceniom tym poddajemy punkty kontrolne $\mathbf{p}_0, \dots, \mathbf{p}_n$.

Obliczenie punktu przecięcia i wektora normalnego polega na drobnym oszustwie. Mając rozwiązanie s równania możemy obliczyć punkt $\mathbf{p}(s)$ i wektor $\mathbf{p}'(s)$. Użyjemy ich do określenia walca o promieniu r , którego oś jest określona przez ten punkt i wektor. Przyjmujemy, że punkt przecięcia promienia z rurką Béziera i wektor normalny są tożsame z punktem przecięcia i wektorem normalnym przecięcia promienia z tym walcem. Dla potrzeb wykonania obrazu takie przybliżenie jest wystarczające. Oczywiście, sprawdzamy przy tym, czy parametr promienia jest dodatni (bo promień jest półprostą) i jeśli mamy dwa punkty wspólne promienia i walca, to wybieramy punkt położony w odległości r od punktu $\mathbf{p}(s)$.

11.3. Techniki przyspieszające

11.3.1. Bryły otaczające i hierarchia sceny

Obliczanie przecięć ma na celu znalezienie punktów przecięcia promienia z obiektem, albo stwierdzenie, że ich nie ma. O ile to pierwsze może wymagać dość kosztownych obliczeń, których nie da się uprościć, istnieje szybki test, który pozwala wykryć większość par promień/obiekt, które nie mają punktów wspólnych. Polega on na wprowadzeniu **brył otaczających** każdy obiekt w scenie. Jeśli obiekt jest powierzchnią algebraiczną lub bryłą wielościenną, rozwiązywanie równań, prowadzące do stwierdzenia, że rozwiązań nie ma, zajmuje zwykle dość sporo czasu (zależnie od stopnia powierzchni lub liczby ścian). Jeśli dla takiego obiektu wskażemy kulę, wewnątrz której ten obiekt jest zawarty, to możemy najpierw sprawdzić, czy promień przecina się ze sferą — brzegiem kuli i wykonywać dalsze, bardziej kosztowne obliczenia tylko w przypadku uzyskania odpowiedzi twierdzącej.

Można wprowadzać inne bryły otaczające, np. kostki (tj. prostopadłościany o ścianach równoległych do płaszczyzn układu), albo elipsoidy (przydatne wtedy, gdy obiekt ma wydłużony kształt). Aby dowiedzieć się, czy promień przecina prostopadłościan, możemy zastosować algorytm Lianga-Barsky'ego (opisany w p. 3.1.3). Poniżej jest przykład realizacji tego algorytmu w zastosowaniu do tego testu. Promień jest reprezentowany przez punkt początkowy $\mathbf{p}$ i wektor $\mathbf{v}$, a parametr b reprezentuje prostopadłościan.

Listing.

```
function TestBoxRay (  $\mathbf{p}$  : punkt;  $\mathbf{v}$  : wektor;  $b$  : boks ) : boolean;
  var  $t_1, t_2$  : real;
  function Test (  $p, q$  : real ) : boolean;
    ... Ta funkcja jest identyczna jak na str. 34
  end { Test };
begin { TestBoxRay }
   $t_1 := 0$ ;  $t_2 := \infty$ ;
  if Test (  $-\mathbf{v}.x, \mathbf{p}.x - b.x_{\min}$  ) then
    if Test (  $\mathbf{v}.x, b.x_{\max} - \mathbf{p}.x$  ) then
      if Test (  $-\mathbf{v}.y, \mathbf{p}.y - b.y_{\min}$  ) then
        if Test (  $\mathbf{v}.y, b.y_{\max} - \mathbf{p}.y$  ) then
          if Test (  $-\mathbf{v}.z, \mathbf{p}.z - b.z_{\min}$  ) then
            if Test (  $\mathbf{v}.z, b.z_{\max} - \mathbf{p}.z$  ) then
              return true;
            return false
          end { TestBoxRay };
        end { TestBoxRay };
      end { TestBoxRay };
    end { TestBoxRay };
  end { TestBoxRay };
```

Wprowadzenie bryły otaczającej dla każdego obiektu zmniejsza średni koszt wyznaczania przecięć (jeśli promienie są rozłączne z większością obiektów), ale nie obniża złożoności obliczeniowej algorytmu, bo nadal trzeba przetworzyć tyle samo par promień/obiekt. Można jednak

wprowadzić **hierarchię obiektów**, której reprezentacja ma postać drzewa; korzeniem drzewa jest cała scena, a poddrzewa opisują zbiory obiektów znajdujących się blisko siebie. W każdym wierzchołku tworzymy reprezentację bryły otaczającej wszystkie obiekty reprezentowane przez ten wierzchołek i jego poddrzewa. Podczas śledzenia promieni przeszukujemy drzewo hierarchii sceny metodą DFS, pomijając każde poddrzewo, z którego bryłą otaczającą promień jest rozłączny.

Przykładowa scena przedstawiona na rysunku 7.5 została opisana w postaci hierarchicznej — sąsiednie obiekty (kule i walce) były łączone w pary, pary w czwórki itd., czyli drzewo wprowadzonej hierarchii było binarne. Dla każdego wierzchołka drzewa (oprócz liści) została znaleziona kostka otaczająca. Utworzenie obrazu w rozdzielczości 200×200 metodą, w której stwierdzenie, że promień nie przecina się z kostką powodowało pominięcie testów przecinania promienia z obiektami w odpowiednim poddrzewie, trwało 56,6s. Pominięcie tych testów i sprawdzanie dla każdego promienia po kolei istnienia przecięć z wszystkimi obiektami trwało (nieistotnie, na jakim sprzęcie, ważne, że na tym samym) 5480s., tj. ok. 100 razy dłużej.

Jeśli nie ma „gotowej” hierarchii sceny, jest tylko lista obiektów, z których scena się składa, to można taką hierarchię wprowadzić „sztucznie”. Praktyczny sposób polega na znalezieniu bryły otaczającej (np. prostopadłościanu) dla każdego obiektu, a następnie na zbudowaniu drzewa binarnego, metodą łączenia wierzchołków w pary. Obiekty geometryczne, z których składa się scena, zostają liśćmi drzewa. Dla każdego (nowego, wewnętrznego) wierzchołka znajdujemy bryłę otaczającą sumę obiektów w poddrzewach. Dwa wierzchołki do połączenia wybieramy tak, aby otrzymać nowy wierzchołek, którego bryła otaczająca jest najmniejsza (np. ma najmniejszą średnicę; to jest algorytm zachłanny, który może nie dać optymalnego drzewa, ale jest stosunkowo prosty i skuteczny). W tym celu można użyć kolejki priorytetowej, np. w postaci kopca. Dla scen złożonych z dużej liczby obiektów takie postępowanie jest dość kosztowne, ale jest ono opłacalne dzięki oszczędności czasu podczas właściwego śledzenia promieni.

11.3.2. Drzewa ósemkowe

Niezależnie od hierarchii sceny, wynikającej ze sposobu jej modelowania, można zmniejszyć złożoność obliczeniową śledzenia promieni za pomocą drzewa ósemkowego, które wprowadza hierarchię wynikającą z rozmieszczenia poszczególnych obiektów w przestrzeni. Drzewo należy utworzyć przed przystąpieniem do śledzenia promieni, dokonując adaptacyjnego rekurencyjnego podziału kostki zawierającej scenę. Adaptacja ta ma na celu skojarzenie z każdym węzłem drzewa możliwie krótkiej listy obiektów, które mają niepuste przecięcie z boksem reprezentowanym przez ten węzeł. Trudność, którą trzeba przy tym pokonać polega na tym, że należy unikać powielania wskaźników do tego samego obiektu w wielu takich listach.

Reguły adaptacyjnego podziału są następujące: kostkę dzielimy na mniejsze, jeśli istnieje obiekt, który

- jest zawarty w jednej z kostek, które powstałyby z podziału kostki danej, lub
- ma średnicę istotnie mniejszą (np. 4 razy) niż średnica kostki (nawet, jeśli ma on niepuste przecięcia z wszystkimi kostkami mniejszymi).

Można arbitralnie ograniczyć wysokość drzewa (tj. głębokość podziału), a ponadto nie dzielić kostki wtedy, gdy nie spowoduje to otrzymania istotnie krótszych list obiektów.

Mając zbudowane drzewo można przystąpić do śledzenia promieni. Dla ustalonego promienia wybieramy pewien punkt p (np. początek promienia). Znając jego współrzędne, możemy (zaczynając od korzenia) znaleźć wszystkie węzły, które reprezentują kostki zawierające punkt p . Wyznaczamy punkty przecięcia promienia z obiektami umieszczonymi w listach obiektów tych kostek. Następnie wyznaczamy punkt przecięcia promienia i ściany kostki i znajdujemy kostkę, do której promień „wchodzi” w tym punkcie. Wyznaczamy przecięcia promienia z obiektami znajdującymi się w kostkach zawierających ten punkt (w drodze od korzenia do liścia) itd.

Procedura kończy działanie po znalezieniu punktu przecięcia promienia z obiektem — jeśli znajdziemy więcej niż jeden taki punkt (dla obiektów w liście pewnej kostki), to wybieramy punkt najbliższy początku promienia. Możemy też zakończyć procedurę w chwili wyjścia poza całą kostkę (bez znalezienia punktu przecięcia).

Ponieważ pewne obiekty mogą być obecne w listach obiektów więcej niż jednej kostki, więc warto zapobiec wykonywaniu pełnej procedury wyznaczania przecięć danego promienia z tym samym obiektem, jeśli występuje on w liście po raz drugi. Skuteczny sposób jest następujący: opis obiektu występuje „w jednym egzemplarzu”, a element listy w kostce składa się ze wskaźnika do tego opisu i wskaźnika do następnego elementu. W opisie obiektu występuje pomocniczy atrybut, będący liczbą całkowitą, o początkowej wartości 0. Kolejno przetwarzanym promieniom nadajemy kolejne numery $1, 2, \dots$. Przed wyznaczaniem przecięcia promienia z obiektem sprawdzamy, czy numer promienia różni się od atrybutu w obiekcie. Jeśli tak, to wykonujemy pełną procedurę wyznaczania przecięcia i przypisujemy atrybutowi obiektu numer bieżącego promienia. W przeciwnym razie parę tę już sprawdzaliśmy. Zakres liczb całkowitych 32-bitowych jest na tyle duży, aby można było prześledzić dostateczną liczbę promieni nawet dla obrazów o bardzo dobrej jakości.

Duże puste obszary są reprezentowane przez stosunkowo duże puste kostki. Stosując drzewa ósemkowe, najwięcej czasu zużywa się na wyznaczanie sąsiada, tj. najmniejszej kostki, której ściana przylega do ściany kostki bieżącej i do której promień „wchodzi”. Przeszukiwanie drzewa można zacząć

- od korzenia — wtedy czas znajdowania sąsiada jest proporcjonalny do poziomu w drzewie poszukiwanej kostki, albo
- od kostki bieżącej (idziemy w górę, a następnie w dół) — wtedy czas zależy od poziomu „najmniejszego wspólnego przodka” kostki bieżącej i sąsiada.

Czas znajdowania sąsiada można zmniejszyć, jeśli zamiast drzewa ósemkowego zastosujemy **równomierny podział** kostki otaczającej scenę. W metodzie tej każdy bok kostki dzieli się na n równych części, w wyniku czego powstaje n^3 „podkostek”. Są one elementami trójwymiarowej tablicy i mają tylko jeden atrybut: wskaźnik do listy obiektów. Mając współrzędne x, y, z dowolnego punktu można obliczyć indeksy do tablicy w czasie stałym. Program, który korzysta z tej metody może być znacznie prostszy; dodatkowy zysk, to oszczędność pamięci (nie trzeba przechowywać wskaźników do poddrzew), choć to oszczędność pozorna. Opisane uproszczenie ma swoją cenę, którą jest brak adaptacji. W rezultacie

- zbyt gruby podział może być nieskuteczny (tj. listy obiektów są zbyt długie),
- zbyt drobny podział powoduje, że większość elementów tablicy zawiera wskaźniki puste (czyli pamięć jest marnowana) i tablica zajmuje dużo miejsca,
- duże puste obszary są drobno „poszatkowane” i czas przechodzenia przez taki obszar jest długotrwały,
- duże obiekty występują w dużej liczbie list.

W praktyce przyjmuje się n rzędu kilkunastu.

11.3.3. Połączenie śledzenia promieni z z -buforem

Większość obiektów w typowych scenach jest matowa, w związku z czym nie obserwujemy w takich scenach wielokrotnych odbić światła i nawet zastosowanie algorytmu z z -buforem, połączonego z odpowiednim modelem oświetlenia daje duży stopień „realizmu” obrazu. W takiej sytuacji można użyć algorytmu z z -buforem do znacznego przyspieszenia śledzenia promieni. Metoda jest następująca:

1. Narysuj scenę za pomocą algorytmu z z -buforem. Ustaw cieniowanie płaskie i nadaj każdemu obiektowi (nie uwzględniając oświetlenia) kolor reprezentowany przez liczbę całkowitą, będącą numerem obiektu (osobnym obiektem w tym przypadku jest np. każda ściana). Cie-

niowanie Gourauda w przypadku, gdy wszystkie wierzchołki wielokąta mają ten sam kolor, powoduje nadanie tego samego koloru wszystkim zamalowanym pikselom.

2. Kolejno dla każdego piksela na obrazie, odczytaj jego kolor — określa on numer obiektu widocznego w tym pikselu. Z *z*-bufora odczytaj głębokość i na jej podstawie (oraz na podstawie współrzędnych *x*, *y* piksela) oblicz współrzędne punktu przecięcia z obiektem promienia pierwotnego, który odpowiada temu pikselowi. Następnie wykonaj śledzenie promieni wtórnych. Jeśli obiekt jest matowy, to wystarczy zbadać, czy dany punkt jest w cieniu. Jeśli obiekt jest zwierciadlany lub przezroczysty, to rekurencyjne śledzenie promieni jest potrzebne w celu otrzymania obrazu obiektów odbitych lub położonych za tym obiektem.

11.4. Śledzenie promieni i konstrukcyjna geometria brył

Metoda śledzenia promieni może być zastosowana do otrzymania obrazu sceny złożonej z brył CSG, bez wyznaczania jawnej reprezentacji takiej bryły. Wystarczy, aby mając dany promień, wyznaczyć punkt przecięcia (i wektor normalny w tym punkcie) powierzchni prymitywu, która jest brzegiem bryły CSG.

Rozważmy dwie metody. **Metoda pierwsza:** Wyznaczamy punkty przecięcia promienia z wszystkimi prymitywami bryły CSG i porządkujemy je w kolejności rosnącej odległości od początku promienia. Dla każdego prymitywu jego przecięcie z promieniem składa się z odcinków (dla brył wypukłych jest to najwyżej jeden odcinek). Następnie wykonujemy operacje CSG (tj. regularyzowane działania teoriomnogościowe) na tych odcinkach, co jest bez porównania łatwiejsze niż działania na bryłach trójwymiarowych. Na końcu wystarczy wybrać koniec odcinka najbliższego obserwatora.

Metodę tę można usprawnić; wykonując działania na listach odcinków, jeśli jeden z argumentów różnicy lub przecięcia jest zbiorem pustym, to nie trzeba wyznaczać drugiego argumentu.

Metoda druga: W metodzie pierwszej trzeba wyznaczyć wszystkie punkty przecięcia promienia z wszystkimi prymitywami. Metoda druga opiera się na fakcie, że stosując drzewo ósemkowe (albo równomierny podział kostki), otrzymujemy poszczególne punkty przecięcia promienia i powierzchni obiektów w kolejności zbliżonej do uporządkowania ich wzdłuż promienia. Rozważmy drzewo CSG, w którego liściach umieścimy zamiast prymitywów ich części wspólne z promieniem (tj. sumy odcinków leżących na promieniu). Początkowo wszystkie liście drzewa oznaczamy jako „nieznane”. Po wyznaczeniu przecięcia promienia z dowolnym prymitywem, podstawiamy odpowiednie odcinki do drzewa CSG i próbujemy określić wyrażenia, których to są argumenty. Jeśli uda się dojść w ten sposób do korzenia, to znajdziemy w nim punkt najbliższy początku promienia, jak poprzednio.

11.5. Antyaliasing

Zjawisko **intermodulacji** (ang. *alias*) jest skutkiem reprezentowania sygnału (w tym przypadku obrazu) za pomocą skończonej liczby próbek. Objawia się ono w postaci

- „ząbkowanych” krawędzi obiektów na obrazie,
- znikania lub zniekształcania małych przedmiotów,
- zniekształcania wyglądu tekstur (to jest najbardziej widoczny i najtrudniejszy do przeciwdziałania artefakt),
- skokowego ruchu i efektów stroboskopowych w animacji.

11.5.1. Antyaliasing przestrzenny

Dokładniejszy opis zjawiska intermodulacji i metod radzenia sobie z nim będzie później, a na razie zajmiemy się metodami związanymi ze śledzeniem promieni.

Nadpróbkowanie (ang. *supersampling*). Wykonujemy obraz o n razy większej rozdzielczości (mamy w ten sposób n^2 razy więcej promieni pierwotnych). Barwa przypisywana pikselowi jest średnią arytmetyczną otrzymanych w ten sposób barw „podpikseli”, czyli pikseli rastra o większej rozdzielczości. W praktyce przyjmuje się $n = 2, 3, 4$ (rzadko więcej).

Nadpróbkowanie adaptacyjne. Widoczne na obrazie artefakty zajmują zwykle nie więcej niż pewną małą część powierzchni obrazu (chyba, że dotyczą tekstury). Dlatego często robi się tak:

1. Wykonujemy obraz w niskiej rozdzielczości (jeden promień pierwotny na piksel),
2. Kolejno dla każdego piksela sprawdzamy, czy jego barwa różni się od barw sąsiednich pikseli o więcej niż określony próg. Jeśli tak, to generujemy dodatkowe promienie pierwotne (dla pewnej liczby podpikseli danego piksela) i obliczamy ostateczną barwę jako średnią barw podpikseli. Postępowanie to bywa czasem stosowane rekurencyjnie (tj. jeśli barwy podpikseli za bardzo się różnią, to podpiksele te podlegają dalszemu podziałowi i śledzi się jeszcze więcej promieni pierwotnych).

W pewnych sytuacjach nadpróbkowanie adaptacyjne może nie wykryć miejsc, w których należałoby śledzić dodatkowe promienie pierwotne. Często np. szczegóły wyglądu tekstury mogą być mniejsze niż piksel. Nie ma prostego rozwiązania tego problemu, zwłaszcza w przypadku obrazów, w których są widoczne odbicia obiektów w powierzchniach zakrzywionych.

Jittering. W przypadku nadpróbkowania artefakty w postaci „ząbkowanych” krawędzi obiektów są zastępowane przez mniej widoczne artefakty w postaci brzegu ostrego/nieostrego (okresowo na przemian). Lepsze efekty można osiągnąć zaburzając punkty, które określają kierunki promieni pierwotnych. Zamiast przez środek podpiksela, promień pierwotny jest określony przez wylosowany punkt należący do podpiksela. Takie postępowanie wprowadza do obrazu szum, który dla ludzkich oczu jest znacznie mniej widoczny i denerwujący.

Filtrowanie. Oprócz obliczania barwy jako średniej arytmetycznej próbek, można obliczyć średnią ważoną. Średnia taka jest przybliżeniem pewnej całki, która opisuje splot funkcji opisującej obraz z ustaloną funkcją zwaną filtrem. Funkcja taka jest dobierana zależnie od specyfiki urządzenia wyjściowego, np. może być inna dla monitorów z kineskopem i z wyświetlaczem ciekłokrystalicznym. W ogólności mamy dwa wzory; pierwszy opisuje to, co chcemy uzyskać:

$$s(x, y) = \int_{\xi_0}^{\xi_1} \int_{\eta_0}^{\eta_1} f(x - \xi, y - \eta) w(\xi, \eta) d\xi d\eta,$$

a drugi realizację (zamiast całki obliczamy kwadraturę):

$$s(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k f_{ij} w(x + i\Delta x, y + j\Delta y).$$

Filtr powinien być funkcją symetryczną, tj. spełniać warunki

$$f(x, y) = f(-x, y) = f(x, -y) = f(-x, -y),$$

unormowaną (tj. $\iint f(\xi, \eta) d\xi d\eta = 1$, albo $\sum_{i,j} f_{ij} = 1$, o jednym maksimum w punkcie $(0, 0)$).

Na przykład nadpróbkowanie i obliczanie średniej arytmetycznej jest zastosowaniem filtru prostokątnego. Inne, często stosowane filtry to filtr stożkowy, Gaussowski i inne. Szczególnie interesujący teoretycznie, choć rzadko stosowany w praktyce jest filtr opisany przez funkcję sinc (łac. *sinus cardinalis*), $\text{sinc } x = \frac{\sin x}{x}$.

11.5.2. Antyaliasing czasowy

Wyświetlanie ciągu obrazów przedstawiających kolejne fazy ruchu daje złudzenie ruchu, co znalazło praktyczne zastosowania co najmniej od czasów braci Lumière.

Zasada działania kamery filmowej jest następująca. Migawka jest tarczą z wyciętym pewnym segmentem i podczas filmowania obraca się ze stałą prędkością. W czasie, gdy dowolna część lub całe okienko o wymiarach klatki jest odsłonięte, film jest nieruchomy (i następuje naświetlenie klatki). W czasie, gdy okienko jest zasłonięte, następuje przesuwanie taśmy. Typowy kąt wyciętego segmentu migawki ma od 90° do 120° , co oznacza, że czas naświetlania (dla 24 klatek na sekundę, tak jak w kinie) jest od $1/96$ do $1/72$ sekundy. Jeśli filmowane obiekty poruszają się, to na poszczególnych klatkach są poruszone, tj. nieostre z wyróżnieniem kierunku ruchu. Gdyby były one idealnie ostre, to obserwator widziałby serię wyraźnie widocznych położów obiektów.

Dodatkowy skutek aliasu czasowego to tzw. zjawisko stroboskopowe. Jeśli mamy obiekt obracający się z dużą prędkością (np. śmigło, szprychy koła w dyliżansie), to możemy otrzymać wrażenie ruchu z inną (znacznie mniejszą) prędkością, być może w przeciwną stronę.

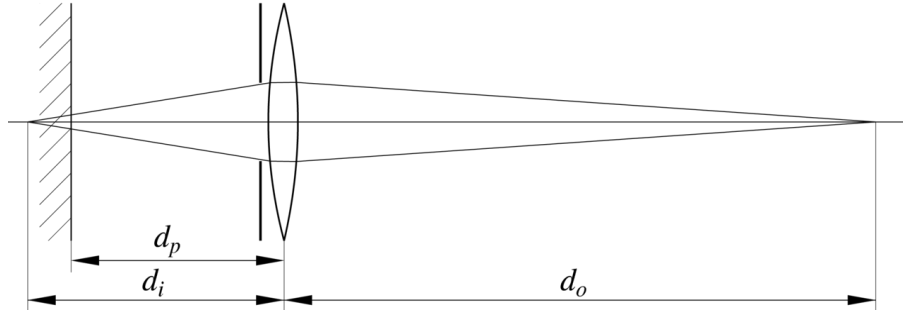
Aby otrzymać na klatkach produkowanego przez komputer filmu rozmyte w kierunku ruchu obrazy obiektów, można dokonać nadpróbkowania i nieraz tak się robi podczas stosowania algorytmu z z -buforem. Efekt jest jednak dość marny, po prostu widać (mniej wyraźnie, ale jednak) gęstszy ciąg (nieco błędnych) położów obiektów. Ponieważ jednak w algorytmie śledzenia promieni mamy możliwość przetwarzania każdego promienia osobno, więc możemy stosować jittering czasowy: dla ustalonego promienia pierwotnego, śledzonego w celu otrzymania klatki filmu w chwili t , losujemy zaburzenie Δt , o wartości bezwzględnej mniejszej niż połowa odstęp czasu między kolejnymi klatkami. Następnie obliczamy położenia wszystkich obiektów sceny w czasie $t + \Delta t$ i obliczamy wartość próbki (intensywność światła przyniesionego przez promień) w zwykły sposób. Połączenie tej metody z jitteringiem przestrzennym pozwala zachować niezbyt dużą liczbę (nie więcej niż kilkanaście) promieni na piksel. Obrazy bardzo wysokiej jakości wymagają śledzenia do kilkudziesięciu promieni pierwotnych na piksel.

Aby zmniejszyć koszt obliczeń, można stosować techniki przyśpieszające, takie jak drzewo ósemkowe. W takiej sytuacji bryła ograniczająca obiekt musi być otoczką wszystkich położów obiektu w pewnym przedziale czasowym (który zawiera chwile $t + \Delta t$ dla wszystkich wylosowanych zaburzeń Δt). Ponieważ liczba obiektów trafionych przez promień pierwotny i wszystkie jego promienie wtórne jest zwykle znacznie mniejsza niż liczba wszystkich obiektów, więc opłaca się stosować „leniwe wartościowanie”: położenie obiektu w chwili $t + \Delta t$ obliczamy po trafieniu przez promień jego bryły otaczającej.

11.6. Symulacja głębi ostrości

Oglądając scenę trójwymiarową, widz skupia uwagę na głównym przedmiocie swojego zainteresowania, nie zwracając uwagi na jego otoczenie i tło. Oko dostosowuje się do odległości od tego przedmiotu, wskutek czego bliższe i dalsze przedmioty są nieostre. Wykonując fotografię, trzeba również odpowiednio nastawić ostrość i nie jest to tylko kwestia niedoskonałości zasady działania obiektywu, ale przede wszystkim świadomego wyboru tematu zdjęcia. Obrazy, na których wszystko byłoby idealnie ostre, są trudniejsze w oglądaniu (i sprawiają nienaturalne wrażenie).

Obraz punktu utworzony przez obiektyw jest plamką o pewnej średnicy, która zależy od długości ogniskowej obiektywu, jego odległości od obiektu i od błony filmowej (albo macierzy CCD) oraz średnicy otworu przysłony. Inne czynniki, które mają na to wpływ (np. dokładne



Rysunek 11.2. Symulacja głębi ostrości — charakterystyczne wymiary.

ustawienie soczewek, zjawiska związane z dyfrakcją) pominiemy, przedstawiając obiektyw jako idealną soczewkę, która spełnia równanie

$$\frac{1}{d_o} + \frac{1}{d_i} = \frac{1}{f}.$$

W równaniu tym d_o oznacza odległość punktu od obiektywu, d_i to odległość ostrego obrazu tego punktu od obiektywu (po drugiej jego stronie), a f jest długością ogniskowej. Średnicę otworu s przysłony najczęściej określa się za pomocą bezwymiarowego współczynnika $n = f/s$ (z dobrego powodu — gęstość mocy światła padającego na film jest proporcjonalna do n^2 , a zatem nastawianie przysłony według światłomierza nie zależy od długości ogniskowej). Na podstawie schematycznego rysunku łatwo jest wyprowadzić wzór opisujący średnicę plamki, która jest obrazem punktu:

$$c_d = \left| 1 - \frac{d_p}{d_i} \right| \frac{f}{n}.$$

Mamy dwa sposoby otrzymania obrazu, który charakteryzowałby się głębią ostrości. Sposób pierwszy polega na postprocesingu; dla każdego piksela musimy znać odległość punktu powierzchni widocznej w tym pikselu od obserwatora (przypominam, że położenie obserwatora jest w tym przypadku tożsame ze środkiem soczewki). Informację taką możemy uzyskać zarówno jako wynik śledzenia promienia pierwotnego, jak i sięgając do z-bufora. Znając tę odległość możemy obliczyć średnicę plamki, a następnie dokonać filtrowania (obliczyć spłot), z zastosowaniem filtru dobrane do obliczonej średnicy plamki. Metoda ta nie najlepiej spisuje się w pobliżu ostrych krawędzi przedmiotów położonych na znacznie bardziej oddalonym tle (które musi być bardzo nieostre). Ponadto otrzymany efekt nie dotyczy obrazów oglądanych w lustrze, a powinien (odległość przedmiotu od lustra jest istotna i nie należy jej pomijać).

Metoda druga polega na zaburzaniu położenia obserwatora i „klatki”, tj. prostokąta położonego na rzutni, na którym powstaje obraz. Przypuśćmy, że chcemy ostrość ustawić na odległość d_o . W układzie obserwatora przesuniemy w kierunku osi x (równoległe do rzutni) obserwatora na odległość r_v , a rzutnię na odległość r_r . Odległość d_i obiektywu od klatki jest równa $d_o f / (d_o - f)$. Na podstawie rysunku 11.3 wnioskujemy, że musi być

$$r_r = r_v \left(1 - \frac{d_i}{d_o} \right) = r_v \left(1 - \frac{f}{d_o - f} \right).$$

Rozważmy teraz punkt położony w odległości ∞ od obiektywu. Ostry obraz tego punktu powstanie w odległości f od obiektywu. Plamka, która jest obrazem tego punktu w odległości d_i , ma średnicę

$$c_d = \left| 1 - \frac{f}{d_i} \right| \frac{f}{n} = \frac{f^2}{d_o n}.$$

1. Określamy pole wektorowe, które opisuje prędkości unoszenia cząsteczek przez wiatr, „cug” z komina itp. W zasadzie takie pole spełnia pewien układ równań różniczkowych cząstkowych, ale często wystarczy wygenerować je „ręcznie”.
2. W określonym miejscu (np. wylocie z komina) losujemy położenia początkowe cząsteczek, z określoną szybkością (np. kilkadziesiąt na jedną klatkę filmu).
3. Dalsze położenia cząsteczek obliczamy całkując pole wektorowe unoszenia. Dokładniej, następne położenie cząsteczki otrzymamy, dodając do poprzedniego położenia wektor prędkości unoszenia w tym miejscu razy przyrost czasu (odległość w czasie między wyświetlaniem kolejnych klatek) i przemieszczenie opisujące indywidualny ruch (ruchy Browna, opadanie z powodu grawitacji itd.).

Cząsteczki, które zostały uniesione poza ustalony obszar „znikają”. Zwalniamy zajmowane przez nie miejsca w tablicy cząsteczek i możemy w to miejsce wpisać położenia nowych cząsteczek, wylosowane w późniejszych chwilach.

4. Wykonujemy śledzenie promieni. Ponieważ średnica cząsteczek jest równa 0, więc szansa trafienia cząsteczki jest równa 0, ale traktujemy promień jak walec o ustalonym promieniu i liczymy cząsteczki, które znajdują się w tym walcu między początkiem promienia i punktem przecięcia promienia z najbliższą powierzchnią. Każda cząsteczka tłumi (częściowo pochłania) światło. Zależnie od liczby cząsteczek, które wpadły do walca, korygujemy intensywność światła niesionego przez promień.

Zauważmy, że postępowanie to dla promieni wtórnych pozwala otrzymać obraz cienia rzucałego przez dym na inne przedmioty.

Aby uzyskać zadowalający efekt, zwykle wystarczy symulować ruch od 10 do 100 tysięcy cząsteczek.

11.8. Implementacja programu do śledzenia promieni

Program wykonujący obrazy metodą śledzenia promieni² może (i powinien) składać się z modułów w znacznym stopniu niezależnych od siebie. Moduły te to

Translator reprezentacji sceny. W pakietach „użytkowych” moduł ten zawiera procesor tekstu, którego zadaniem jest utworzenie wewnętrznej reprezentacji sceny na podstawie skryptu (dostarczonego przez użytkownika programu). Procesor ten wywołuje procedury konstruujące obiekty reprezentujące poszczególne prymitywy i wierzchołki drzewa lub grafu hierarchii sceny (w tym np. wierzchołki reprezentujące operacje CSG) i buduje ten graf. Ponadto rozpoznaje i odpowiednio przetwarza opisy źródeł światła i położenia obserwatora. Przed napisaniem translatora należy określić język opisu sceny, tj. pewien język formalny, którego zdaniem są dopuszczalne teksty skryptów. Translator składa się z analizatora leksykalnego (który może być wyposażony w makrogenerator), analizatora syntaktycznego (który rozpoznaje opisy obiektów i np. kojarzy rozpoznane opisy tekstur z rozpoznanymi opisami figur geometrycznych) i generatora obiektów, który tworzy obiekty, nadaje wartości ich atrybutom i buduje z nich drzewo lub graf hierarchii.

Możliwym rozwiązaniem zastępczym jest użycie w charakterze takiego translatora kompilatora języka programowania, w którym realizujemy cały program (np. Pascala, C, C++). Wtedy zamiast skryptu opisującego scenę użytkownik będzie musiał napisać i dołączyć do programu procedurę, która wywołuje (dostępne w postaci biblioteki) odpowiednie procedury konstruujące „wewnętrzną” reprezentację sceny. Po wywołaniu tej procedury program może przystąpić do syntezy obrazu.

² Moje próby związętego spolszczenia określenia „*ray tracer*” zakończyły się po tym, jak pomyślałem o „śledzi promieni”. Mimo poprawności rybnej konotacji (słowo *ray* oznacza po angielsku również płaszczkę) nie uważam tego pomysłu za udany i więcej prób nie podejmę.

Zaletą tego rozwiązania jest możliwość wykorzystania dowolnych konstrukcji językowych w celu utworzenia sceny, a także łatwość rozszerzania możliwości programu (np. dodawania nowych rodzajów prymitywów) bez potrzeby zmieniania języka skryptów. Wadą jest przyjęcie założenia, że użytkownik programu umie pisać przynajmniej proste programy i umie obsługiwać kompilator. W praktyce często kompilator pełni rolę translatora reprezentacji sceny podczas uruchamiania modułu syntezy obrazu. Są również gotowe pakiety, które oferują (czasem opcjonalnie) taki interfejs użytkownika.

Biblioteka obiektów. Każdy element reprezentacji sceny (np. prymityw lub obiekt złożony) jest obiektem, który powinien udostępniać metody stanowiące interfejs sceny z procedurą śledzenia promieni opisaną dalej. W najprostszym przypadku wystarczą dwie metody. Pierwsza z nich wyznacza punkty przecięcia promienia danego jako parametr z obiektem i wektory normalne powierzchni obiektu w tych punktach. Druga metoda dla ustalonego punktu oblicza wartość tekstury w tym punkcie, czyli parametry (związane z powierzchnią) występujące na przykład w modelu oświetlenia Phonga. Może też być trzecia metoda, której zadaniem jest narysowanie obiektu przy użyciu algorytmu z buforem głębokości, w celu przyspieszenia przetwarzania promieni pierwotnych sposobem opisanym w p. 11.3.3.

Bardzo wygodne jest zrealizowanie tych obiektów w języku „obiekowym”, na przykład w C++. Wspomniane metody będą oczywiście wirtualne. W klasie „figura” (której obiektami są wszystkie elementy reprezentacji sceny) metody będą puste, natomiast w podklasach będą odpowiednio zdefiniowane. Dzięki temu procedura śledzenia promieni może wywoływać odpowiednią metodę „nie wiedząc”, czy należy ona do obiektu opisującego sferę, płat B-sklejany, czy też wewnętrzny wierzchołek drzewa CSG. W ostatnim przypadku obiekt zawiera wskaźniki do poddrzew reprezentujących argumenty operacji CSG; metoda obiektu wywołuje metody znajdowania przecięć promienia z tymi argumentami („nie wiedząc” jakiego one są rodzaju), a następnie obliczy punkt przecięcia promienia z bryłą CSG na podstawie wyników obliczeń dokonanych przez metody argumentów.

Procedura śledzenia promieni. Procedura ta otrzymuje promień jako argument. Jej zadaniem jest znalezienie odpowiedniego punktu przecięcia z którymś z obiektów (w tym celu procedura wywołuje metody obiektów), a następnie wygenerowanie promieni wtórnych, prześledzenie ich (za pomocą rekurencyjnego wywołania) i obliczenie światła niesionego przez promień do jego początku.

Jedynie ta procedura realizuje określony algorytm śledzenia, tj. tylko w niej jest określona liczba generowanych promieni wtórnych i kryterium zatrzymania rekurencyjnych wywołań. Procedura śledzenia promieni może posługiwać się strukturą danych taką jak drzewo ósemkowe w celu usprawnienia obliczeń. Alternatywnie, zadanie zmniejszania złożoności obliczeniowej wyznaczania przecięć może spoczywać na metodach obiektów, jeśli obiekty te są zorganizowane hierarchicznie. Na przykład procedura wywołuje metody tylko jednego lub najwyżej kilku obiektów, z których każdy jest korzeniem drzewa opisującego pewną część sceny. Działanie metody znajdowania przecięć promienia z obiektem zaczyna się od sprawdzania położenia promienia względem bryły otaczającej (i najczęściej na tym się kończy).

Generator promieni pierwotnych. To jest procedura, która oblicza kolor kolejnych pikseli na obrazie (wywołując procedurę śledzenia promieni), przy czym szczegóły obliczeń takie jak liczba promieni pierwotnych na piksel i rodzaj zastosowanego filtru w antyaliasingu są poza tą procedurą niewidoczne (z jednym wyjątkiem — w antyaliasingu czasowym procedura ta musi nadawać odpowiednie wartości globalnemu parametrowi czasu, na podstawie którego metody obiektów określają położenie tych obiektów w chwili, w której trafia je promień).

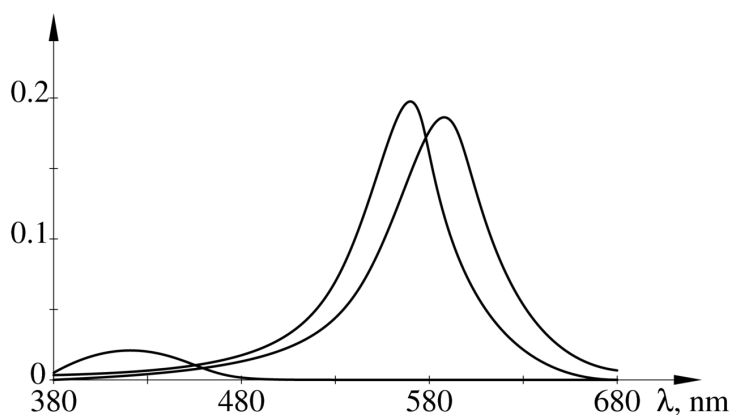
Biblioteka procedur wyjściowych, których zadaniem jest utworzenie odpowiedniego pliku w ustalonym formacie lub wyświetlenie obrazu na ekranie. Zmiana formatu pliku wymaga dokonania zmian tylko w tym module.

12. Przestrzeń barw i jej układy współrzędnych

12.1. Podstawy kolorymetrii

Światło jest promieniowaniem elektromagnetycznym o długości fali 380–780nm. Barwa światła jest określona przez jego widmo, czyli funkcję opisującą rozkład energii niesionej przez światło w zależności od długości fali.

Natura wyposażyła ludzi w receptory światła (tzw. czopki) trzech różnych rodzajów. Każdy z nich ma maksimum czułości odpowiadające innej długości fali światła¹. Dlatego przestrzeń wrażeń wywoływanych u ludzi przez światło (tj. sygnałów przesyłanych przez receptory do mózgu) jest trójwymiarowa². Inaczej mówiąc, można otrzymać dowolne wrażenie barwne przez zmieszanie światła w trzech tzw. barwach podstawowych (z zastrzeżeniem, o którym dalej).



Rysunek 12.1. Wykres czułości czopków na światło.

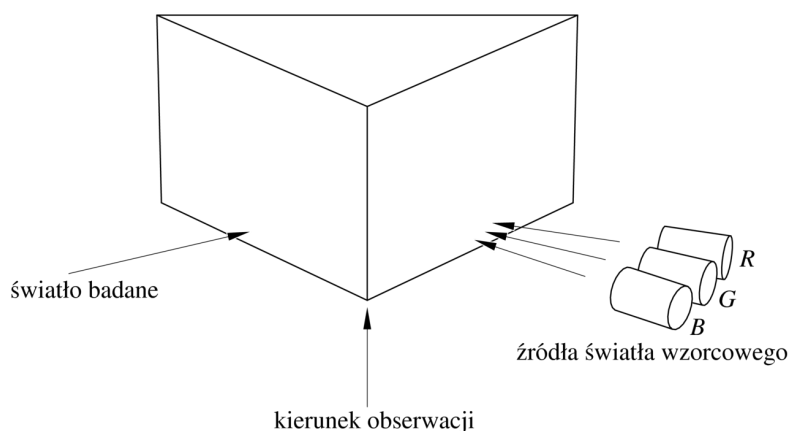
Dla poszczególnych rodzajów czopków można znaleźć wykresy ich czułości na światło zależnie od długości fali. Energia światła zaabsorbowanego przez receptor (czyli „intensywność wrażenia wzrokowego”) jest całką z iloczynu widma padającego nań światła i jego funkcji czułości. Mając źródła światła o znanych widmach (np. czerwony, zielony i niebieski luminofor w ekranie monitora) możemy, dobierając odpowiednio moc ich świecenia, tak pobudzać receptory, aby ich reakcja była taka sama jak na światło pochodzące z danego źródła.

Jeśli ustalimy moc (intensywność) światła i będziemy zmieniać widmo, to dla ustalonych trzech barw podstawowych możemy wykonać rysunek dwuwymiarowy. Barwy podstawowe znajdują się w wierzchołkach trójkąta; ich intensywności (o stałej sumie, unormowanej do 1) są współrzędnymi barycentrycznymi punktu odpowiadającego danemu światłu. Wypełniając trójkąt barwami (wykonując cieniowanie metodą Gourauda), otrzymamy obrazek zawierający wszystkie barwy możliwe do otrzymania przez zmieszanie barw podstawowych. Barwy repre-

¹ Oprócz czopków mamy też tzw. pręciki, które są czulsze od czopków, ale nie rozróżniają barw. Wskutek tego w słabym oświetleniu (np. o zmroku) oglądane przedmioty są bezbarwne.

² U innych gatunków bywa inaczej, np. przestrzeń wrażeń barwnych rozróżnianych przez krowę jest tylko dwuwymiarowa, natomiast morskie skorupiaki z gatunku rawka wieszczka (*Squilla mantis*) mają dziesięć różnych typów receptorów światła (w tym receptory czule na polaryzację).

zentowane przez punkty na zewnątrz trójkąta mają określone współrzędne, ale otrzymanie ich przez zmieszanie barw podstawowych jest fizycznie niewykonalne.



Rysunek 12.2. Zasada działania kolorymetru.

Pomiar współrzędnych barwy danego światła przeprowadza się za pomocą kolorymetru. Zawiera on trzy źródła światła wzorcowego; lampę czerwoną, czyli żarówkę z filtrem, oraz dwie lampy rtęciowe, wytwarzające światło zielone i niebieskie. Osoba dokonująca pomiaru tak ustawia przysłony źródeł światła wzorcowego, aby ściany białego klina były (jej zdaniem) identycznie oświetlone przez światło badane i wzorcowe. Współrzędne barwy odczytuje się z podziałek na przysłonach. Aby umożliwić pomiar ujemnych współrzędnych barwy (dla światła, którego nie można otrzymać, mieszając światła wzorcowe), kolorymetr jest wyposażony w źródła światła wzorcowego z przysłonami także od strony okienka, przez które wpada światło badane.

12.2. Diagram CIE

Dla danego widma może istnieć tzw. **dominująca długość fali**. Gdyby światło obserwowane przez oko było mieszaniną (w odpowiednich proporcjach) światła o tej długości fali ze światłem białym³, to wrażenia byłyby identyczne jak wrażenia wywołane przez światło o danym widmie. Światło białe jest rozszczepiane przez pryzmat na tęczę. Poszczególne barwy tęczy odpowiadają różnym dominującym długościom fali.

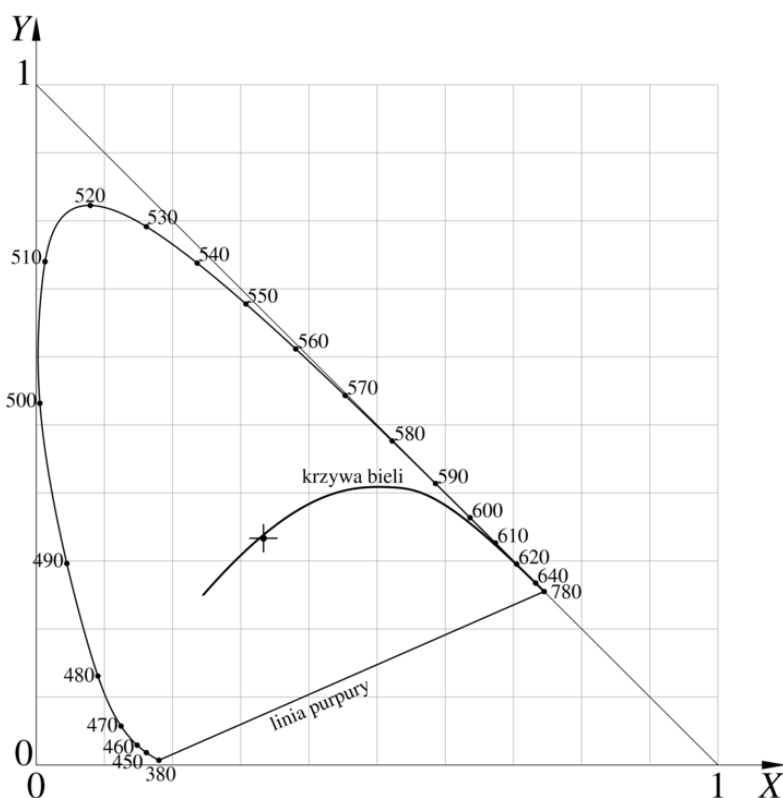
Jedną z podstaw postępu technicznego jest standaryzacja. W zastosowaniach technicznych podstawą do opracowywania standardów jest powszechnie używany **diagram CIE**, ustalony w 1931r. przez Międzynarodową Komisję Oświetleniową (*Commission Internationale de l'Eclairage*), przedstawiony na rysunku 12.3. Układ współrzędnych barw, w którym jest przedstawiany ten diagram, będący także podstawą do określania innych układów współrzędnych w przestrzeni barw, nazywa się układem *XYZ*. Obszar płaszczyzny, który reprezentuje barwy widzialne, ma kształt podkowy⁴ wpisanej w trójkąt, którego wierzchołki stanowią układ odniesienia. Dzięki temu współrzędne wszystkich widzialnych barw w układzie *XYZ* są nieujemne.

Punkty na krzywej ograniczającej diagram od góry odpowiadają barwom światła monochromatycznego (w którym występują tylko fale o jednej długości). Na krzywej tej leżą kolejne barwy

³ Określenie „światło białe” jest niejednoznaczne, przez co jest źródłem wielu nieporozumień. Można umawiać się, że jest to światło o stałym widmie w całym zakresie długości fali albo światło wysyłane przez „ciało doskonale czarne” o ustalonej temperaturze. Chwilowo stosuję pierwszą umowę.

⁴ Z moich obserwacji wynika, że końska podkowa ma inny kształt, ale wszystkie opisy diagramu CIE, a więc i ten, zawierają wzmiankę o podkowie.

tęczy. Odcinek ograniczający diagram od dołu to tzw. **linia purpury**. Światło reprezentowane przez punkty tego odcinka jest mieszaniną światła fioletowego i czerwonego i nie można dla niego wskazać dominującej długości fali. Punkty wewnątrz diagramu reprezentują światło „złamane”, z domieszką bieli.



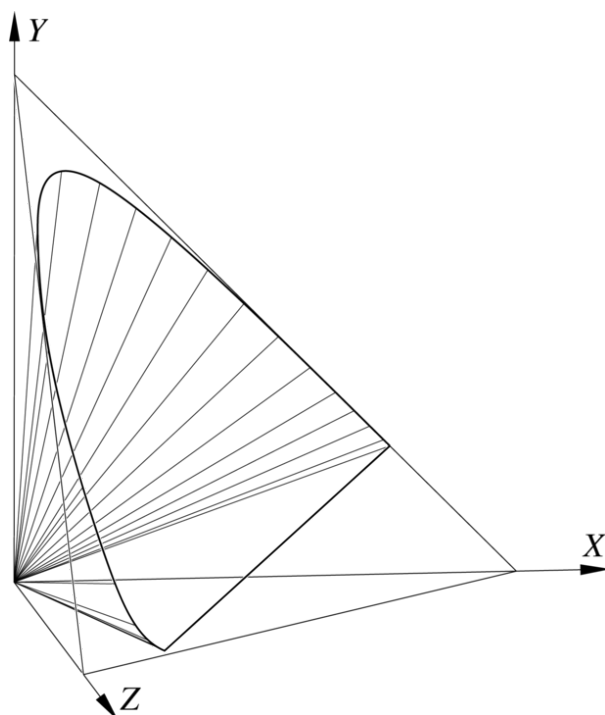
Rysunek 12.3. Diagram CIE (przybliżony).

Uwaga: Światło o barwie reprezentowanej przez punkty spoza obszaru ograniczonego krzywą tęczy i linią purpury nie istnieje. Także barwy określające układ odniesienia są tylko umowne, wbrew temu, co można przeczytać w niektórych publikacjach. Ponieważ obszar reprezentujący barwy światła widzialnego nie jest wielokątem, więc nie istnieje żaden skończony zbiór źródeł światła, którego mieszanie pozwoliłoby otrzymać wszystkie widzialne barwy (ale jest oczywiste, dlaczego obszar ten jest wypukły).

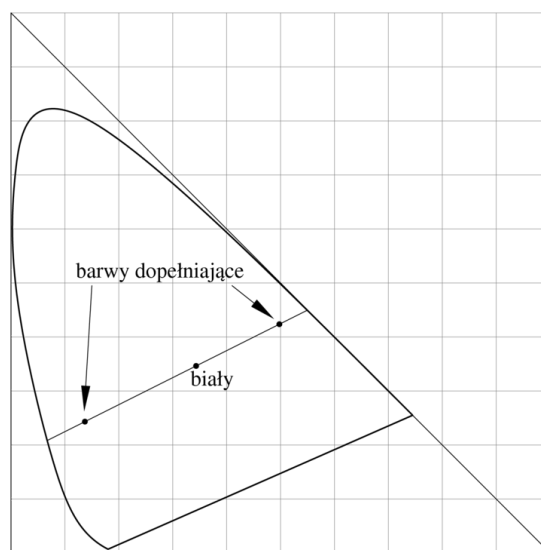
Płaski obrazek umożliwia przedstawienie pewnego przekroju trójwymiarowego zbioru wrażeń rozróżnianych przez ludzki narząd wzroku; diagram CIE przedstawia światło o stałej mocy. Bryła, której punkty reprezentują światło widzialne o ograniczonej mocy całkowitej, przedstawiona w układzie XYZ , jest pokazana na rysunku 12.4.

Pojęcie **bieli** jest umowne, ponieważ oko przyzwyczaja się do różnych warunków oświetlenia i odbiera jako „białe” światło o różnych barwach. Na ogół za białe przyjmuje się światło wysyłane przez ciało doskonale czarne (np. pogrzebacz, żarówka, Słońce), ogrzane do odpowiedniej temperatury (np. pogrzebacz — 700K, żarówka — 3000K, Słońce — 6000K, oświetlenie przez promienie słoneczne rozproszone w atmosferze — 9000K). Na diagramie CIE barwy światła białego o różnych temperaturach leżą na pewnej krzywej (uwidocznionej na rysunku 12.3).

Mając ustaloną barwę białą można wprowadzić pojęcia **nasycenia barwy** i **barwy dopełniających**. Barwa jest nasyciona wtedy, gdy reprezentujący ją punkt leży na brzegu obszaru barw widzialnych. Nasycenie jest równe 0 wtedy, gdy światło jest białe i 1 jeśli odpowiedni punkt



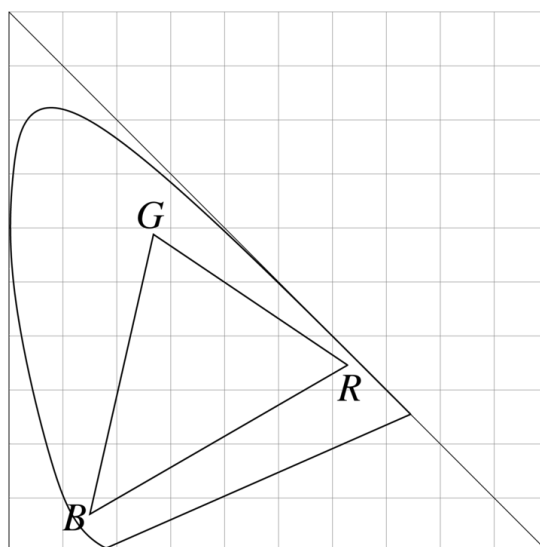
Rysunek 12.4. Bryła barw widzialnych w układzie CIE XYZ.



Rysunek 12.5. Barwy dopełniające na diagramie CIE.

znajduje się na brzegu obszaru barw widzialnych. Dla ustalonej barwy można narysować prostą przechodzącą przez reprezentujący ją punkt i punkt bieli. Nasycenie można odczytać mierząc proporcję podziału odcinka łączącego punkt bieli z brzegiem obszaru, przez punkt reprezentujący barwę daną. Barwa dopełniająca jest reprezentowana przez pewien punkt na rozpatrywanej prostej po przeciwnej stronie punktu bieli. Ma ona to samo nasycenie i moc, co barwa przez nią dopełniana.

Różne urządzenia mają różne zbiory barw, które mogą reprodukować. Na przykład telewizory i monitory komputerowe z lampą kineskopową mają trzy rodzaje luminoforów, które



Rysunek 12.6. Barwy możliwe do zrealizowania na ekranie monitora.

świecąc z różną intensywnością, zawsze nieujemną, odwzorowują barwy reprezentowane przez punkty należące do trójkąta, którego wierzchołki odpowiadają barwom światła wysyłanego przez poszczególne luminofory. Drukarki kolorowe mają co najmniej trzy różne atramenty, co teoretycznie umożliwia reprodukcję barwy o dowolnym odcieniu. Częściej jednak atramentów jest więcej (np. sześć), ponieważ dla otrzymania wydruku o dobrej jakości potrzebne jest uzyskanie barw o wysokim nasyceniu, a to jest trudne dla barw otrzymanych przez zmieszanie różnych atramentów, których barwy są odległe od barw reprodukowanych. Zwykle zbiór barw możliwych do wydrukowania nie zawiera, ani nie jest zawarty w zbiorze barw możliwych do otrzymania na monitorze. Oczywiście, zadrukowany papier nie jest źródłem światła, tylko obiektem odbijającym światło, a zatem reprodukcja barw przez urządzenia drukujące jest zależna od światła, w jakim obraz oglądany. Zwróćmy jednak uwagę, że wzrok osoby oglądającej wydrukowany obrazek przyzwyczaja się do światła w otoczeniu (przez co uważamy, że „biała” kartka papieru wygląda tak samo w świetle dziennym i sztucznym).

Tworzenie i przesyłanie obrazu wiąże się z błędami reprezentacji barw. Konieczne jest zatem zbadanie następującego problemu: na jakie zniekształcenia barw ludzki wzrok jest najbardziej i najmniej wyczulony?

Odpowiedź na to pytanie ma znaczenie dla opracowania metod transmisji (w tym kompresji) obrazów i dla reprodukcji obrazu na urządzeniu, które nie może oddać wszystkich barw obecnych w obrazie danym. W dużym uproszczeniu można stwierdzić, że najmniej dostrzegalne są zmiany nasycenia i jasności całego obrazu, a najbardziej widoczne są różne zmiany odcienia (czyli długości fali dominującej) oraz różne zmiany jasności sąsiednich obszarów. Z tego powodu, aby wydrukować obraz na drukarce, która nie odtwarza takich barw jak monitor, trzeba dokonać tzw. **desaturacji obrazu** — wszystkie punkty reprezentujące barwy na obrazie będą „przesunięte” w stronę bieli.

12.3. Współrzędne RGB i YIQ

W praktyce (telewizyjno-komputerowej) zamiast współrzędnych XYZ stosuje się współrzędne RGB , które mają związek z barwami światła emitowanego przez luminofory w kineskopach.

W tabeli niżej są współrzędne X , Y punktów odniesienia układu RGB w standardach NTSC i CIE, a także barwy luminoforów w typowym monitorze.

	NTSC	CIE	monitor
R	0.67, 0.33	0.735, 0.265	0.628, 0.346
G	0.21, 0.71	0.274, 0.717	0.268, 0.588
B	0.14, 0.08	0.167, 0.009	0.15, 0.07

Do transmisji obrazów (w telewizji) stosuje się układ YIQ , w którym podaje się wartość tzw. **luminancji** Y i dwie wartości **chrominancji**, I , Q . Sygnał luminancji wystarczy do utworzenia obrazu na ekranie telewizora czarno-białego.

Aby obliczyć współrzędne YIQ znając RGB i odwrotnie, stosuje się następujące wzory (tu jest NTSC RGB):

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ 0.596 & -0.275 & 0.321 \\ 0.212 & -0.528 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.62 \\ 1 & -0.272 & -0.647 \\ 1 & -1.108 & 1.705 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}.$$

Okazuje się, że w obrazach telewizyjnych do przesyłania sygnału Y wystarczy pasmo o szerokości 4MHz, a dla sygnałów I oraz Q odpowiednio 1.5MHz i 0.6MHz. Wynikające stąd zniekształcenia obrazów są dostatecznie małe w telewizji⁵. Również wiele algorytmów stratnej kompresji obrazu (np. JPEG) reprezentuje obraz we współrzędnych YIQ i koduje każdą składową osobno, dopuszczając większe zniekształcenia dla składowych chrominancji i osiągając dzięki temu większy współczynnik kompresji przy zachowaniu subiektywnie dobrej jakości obrazu.

12.4. Współrzędne CMY i CMYK

Współrzędne XYZ , a także YIQ i RGB , są związane z modelem **addytywnego** mieszania światła. W drukarstwie i fotografii ma zastosowanie też model **subtraktywny**, z którym związane są układy współrzędnych CMY i $CMYK$. Symbole współrzędnych pochodzą od angielskich nazw barw podstawowych (*Cyan* — niebieskozielony, *Magenta* — purpurowy, *Yellow* — żółty). Zasada subtraktywnego mieszania barw jest taka, że światło białe przechodzi przez barwne filtry, które je tłumią. Jeśli więc $C = M = Y = 0$, to mamy światło białe, jeśli $C = M = 0$, $Y = 1$ to światło żółte (żółty filtr, Y , nie przepuszcza światła niebieskiego), $C = 0$, $M = Y = 1$ opisuje światło czerwone (filtr purpurowy, M , zatrzymuje światło zielone) i dla $C = M = Y = 1$ otrzymujemy czerń.

Ponieważ barwniki, którymi drukuje się obrazy, nie są idealnymi filtrami, więc nawet przy ich maksymalnym stężeniu w danym miejscu pewna ilość światła przechodzi przez warstwę farby lub atramentu (dwukrotnie, w międzyczasie odbijając się od białego papieru) i nie otrzymujemy idealnej czerni (tylko jakiś bury kolor). Dlatego stosuje się model $CMYK$, w którym czwarta

⁵ Ale utworzenie obrazu na ekranie monitora komputerowego o wysokiej rozdzielczości wymaga wyświetlania pikseli w tempie od 100 do 250MHz; w przeciwnym razie małe literki byłyby nieczytelne. Także tzw. telewizja HDTV wymaga przesyłania sygnałów o znacznie szerszych pasmach. W tym celu sygnał w postaci cyfrowej poddaje się kompresji stratnej, dzięki czemu nie trzeba wykorzystywać aż tak szerokich pasm, niedostępnych z powodów technicznych.

współrzędna, K (ang. *black*) opisuje stężenie czarnej farby. Teoretycznie, mając współrzędne R, G, B , obliczamy

$$K = 1 - \min\{R, G, B\},$$

$$C = 1 - R - K, \quad M = 1 - G - K, \quad Y = 1 - B - K.$$

Jedna ze współrzędnych C, M, Y otrzyma wartość 0.

W praktyce często przyjmuje się mniejszą wartość K , co poprawia końcowy efekt, ponieważ m.in. domieszanie niewielkiej ilości czarnej farby do pozostałych „zabrudziłoby” barwę wydrukowanego obrazu. Ponadto, z uwagi na nieuniknione przesunięcia wydruków w poszczególnych barwach, czarny nadruk na kolorowym tle może „nie trafić” w odpowiedni biały obszar otoczony tłem i widać białą „obwódke” dookoła czarnego obszaru. Aby temu przeciwdziałać należy tło wydrukować bez „otworu” na czarną farbę, której naniesienie nieco obok właściwego miejsca będzie dzięki temu niedostrzegalne. Z tego przykładu wynika, że przejście od współrzędnych RGB do $CMYK$ w celu otrzymania wyciągów barwnych do drukowania jest trywialne tylko pozornie.

Warto pamiętać, że zagadnienia związane z mieszaniem farb drukarskich lub atramentów na papierze są bardzo skomplikowane i dlatego przygotowanie obrazu do druku w celu otrzymania takiej jak trzeba ilości farby na papierze jest sztuką trudną. Stanowi ona przedmiot intensywnych badań firm produkujących maszyny drukarskie i drukarki. Firmy te starannie chronią swoje tajemnice przemysłowe. Ponieważ współrzędne $CMYK$ są ściśle związane z określoną technologią (tj. na przykład z przygotowaniem filmów do naświetlania form drukarskich), więc ich praktyczna przydatność dla użytkowników grafiki (z wyjątkiem osób przygotowujących ilustracje do druku) jest raczej niewielka (ale warto wiedzieć, o co może chodzić pracownikowi firmy poligraficznej, z którym będziemy współpracować).

12.5. Współrzędne HSV i HLS

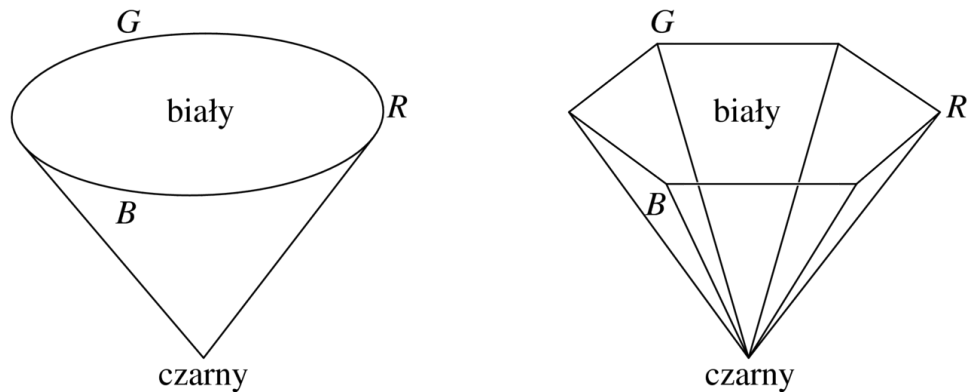
Współrzędne RGB są wygodne w technice (w konstrukcji monitorów), ale dla potrzeb interakcyjnego dobierania barw są one mało intuicyjne. Dla użytkowników programów graficznych (zwłaszcza dla osób, których wykształcenie techniczne nie dorównuje plastycznemu) znacznie wygodniejszy jest układ współrzędnych HSV (nazwy współrzędnych są wzięte z angielskiego: *Hue* oznacza odcień, *Saturation* — nasycenie⁶, *Value* — wartość).

Zbiór barw możliwych do osiągnięcia na monitorze można przedstawić jako sześciian; współrzędne R, G, B zmieniają się od 0 do 1 wzdłuż jego krawędzi. Tę **bryłę barw** można poddać nieliniowemu przekształceniu, w wyniku którego powstaje stożek. Czasem też przedstawia się tę bryłę jako ostrosłup sześciokątny, być może dlatego, że łatwiej jest go narysować.

Wierzchołek stożka ma współrzędną $V = 0$ i reprezentuje barwę czarną ($R = G = B = 0$). Współrzędne S i H są w tym punkcie nieokreślone.

Środek podstawy stożka ma współrzędne $V = 1, S = 0$ i reprezentuje barwę białą. Punkty na osi stożka reprezentują różne poziomy szarości; z wyjątkiem wierzchołka stożka współrzędna S jest równa 0, a H jest nieokreślona. Oddalając się od osi stożka zwiększamy nasycenie S do maksymalnej wartości 1 na powierzchni bocznej stożka. Współrzędna H odpowiada kątowi obrotu wokół osi stożka. Rysunek 12.7 przedstawia rozmieszczenie barw o poszczególnych odcieniach.

⁶ Nasycenie w tym układzie jest określone względem bryły barw fizycznie realizowalnych. Wartość 1 nasycenia odpowiada więc barwom o maksymalnym nasyceniu możliwym do uzyskania na danym urządzeniu, mimo że te barwy leżą wewnątrz obszaru barw widzialnych przedstawionego za pomocą diagramu CIE.



Rysunek 12.7. Układ współrzędnych i bryła HSV.

Do przejścia między układami RGB i HSV służą poniższe procedury, napisane przy założeniu, że wszystkie współrzędne przebiegają przedział $[0, 1]$. Czasem współrzędną H podaje się w stopniach, od 0 do 360.

Listing.

```

procedure RGBtoHSV (  $r, g, b, h, s, v$  );
begin
   $\max := \max(r, g, b)$ ;
   $\min := \min(r, g, b)$ ;
   $\text{delta} := \max - \min$ ;
   $v := \max$ ;
  if  $\max \neq 0$  then  $s := \text{delta} / \max$  else  $s := 0$ ;
  if  $s \neq 0$  then begin
    if  $r = \max$  then  $h := (g - b) / \text{delta}$ 
    else if  $g = \max$  then  $h := 2 + (b - r) / \text{delta}$ 
    else  $h := 4 + (r - g) / \text{delta}$ ;
    if  $h < 0$  then  $h := h + 6$ ;
     $h := h / 6$ 
  end
end {RGBtoHSV};

```

Listing.

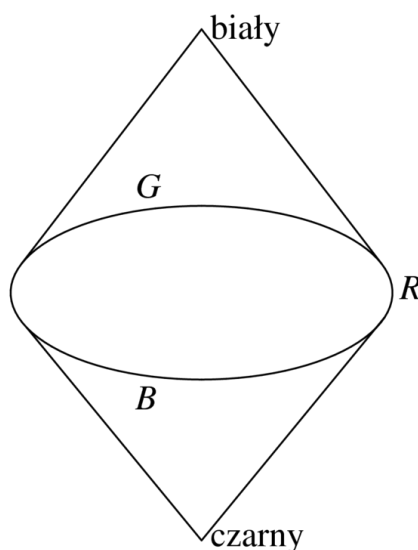
```

procedure HSVtoRGB (  $h, s, v, r, g, b$  );
begin
  if  $s = 0$  then begin  $r := v$ ;  $g := v$ ;  $b := v$  end
  else begin
    if  $h = 1$  then  $h := 0$ ;
     $h := 6h$ ;  $i := \lfloor h \rfloor$ ;  $f := h - i$ ;
     $a := v(1 - s)$ ;  $b := v(1 - s * f)$ ;  $c := v(1 - s(1 - f))$ ;
    case  $i$  of
      0: begin  $r := v$ ;  $g := c$ ;  $b := a$  end;
      1: begin  $r := b$ ;  $g := v$ ;  $b := a$  end;
      2: begin  $r := a$ ;  $g := v$ ;  $b := c$  end;
      3: begin  $r := a$ ;  $g := b$ ;  $b := v$  end;
      4: begin  $r := c$ ;  $g := a$ ;  $b := v$  end;
      5: begin  $r := v$ ;  $g := a$ ;  $b := b$  end
    end
  end
end

```

`end {HSVtoRGB};`

Współrzędna V w układzie HSV jest taka sama dla np. najjaśniejszej barwy niebieskiej, jak i dla białej. Ponieważ w tym drugim przypadku barwa jest związana z większą mocą światła, więc bardziej intuicyjny może wydawać się układ HLS , w którym bryła barw składa się z dwóch stożków zetkniętych podstawami. Współrzędna L (ang. *Light*), która zastępuje współrzędną V w tym układzie, dla światła białego ma wartość 2, natomiast barwy „czyste” o maksymalnym nasyceniu (które w bryle barw HSV leżą na brzegu podstawy stożka) mają współrzędną $L = 1$. Wprawdzie światło żółte ($R = G = 1, B = 0$) wydaje się znacznie jaśniejsze niż np. niebieskie ($R = G = 0, B = 1$; luminancja światła żółtego jest równa 0.866, a niebieskiego 0.144, czyli jest 6 razy mniejsza), ale takie rozwiązanie okazało się użyteczne.



Rysunek 12.8. Bryła barw HLS.

Wybór układu współrzędnych i związanej z nim bryły barw jest ważny nie tylko ze względu na wygodę wybierania pojedynczych punktów (tj. barw). Znacznie istotniejsze jest **mieszanie barw** oraz **interpolacja** między wskazanymi punktami. Mieszanie barw podczas filtrowania (zmiany rozdzielczości, antyaliasing) i interpolacja w cieniowaniu Gourauda najczęściej wiążą się z addytywnymi układami współrzędnych, natomiast w interakcyjnym dobieraniu barw często celem jest otrzymanie barw reprezentowanych przez punkty wskazanego odcinka. Zauważmy, że interpolacja w bryle barw RGB polega na interpolacji współrzędnych, natomiast interpolacja w bryłach HSV i HLS wymaga przejścia do pomocniczego układu współrzędnych kartezjańskich i interpolacji tych współrzędnych.

13. Język PostScript

13.1. Wprowadzenie

Język PostScript został opracowany przez firmę Adobe Systems Inc. w 1985r. Jest to tzw. język opisu strony; plik postscriptowy jest programem, który jest interpretowany przez drukarkę lub inne urządzenie, w celu utworzenia obrazu np. do wydrukowania. Dodatkowo, jest to prawdziwy język programowania (nawet dosyć „wysokopoziomowy”), w którym można pisać programy wykonujące skomplikowane obliczenia. Możliwości graficzne można wtedy zignorować lub wykorzystać do wyprowadzenia wyników.

Podstawowa zasada systemu grafiki związanego z językiem PostScript to niezależność opisu strony od urządzenia, które ma utworzyć obraz; wiadomo, że jest to urządzenie rastrowe, ale można i warto używać PostScriptu w oderwaniu od sprzętu; interpreter języka w dowolnym urządzeniu ma za zadanie przedstawić obraz o najlepszej jakości osiągalnej z tym urządzeniem.

Praktyczny przykład tej filozofii: piszemy `g setgray`, gdzie `g` jest liczbą rzeczywistą z przedziału $[0, 1]$. Polecenie to ustawia poziom szarości (0 to kolor czarny, 1 — biały). Rozwiązanie, w którym poziom szarości byłby określany przez podanie liczby całkowitej z przedziału od 0 do 255 nosiłoby piętno zależności sprzętowej (prawdopodobnie od liczby bitów w rejestrach przetwornika cyfrowo-analogowego sterownika graficznego). Tymczasem dzięki możliwości podania liczby rzeczywistej

- nie ma ograniczenia tylko do 256 poziomów szarości (istnieją, co prawda rzadko spotykane, sterowniki z dziesięcio- lub dwunastobitowymi przetwornikami, więc to rozwiązanie umożliwia pełne wykorzystanie ich możliwości),
- nawet jeśli jasność jest ostatecznie przeliczana na liczbę całkowitą od 0 do 255 (która będzie przypisana pikselom), może to być przekształcenie nieliniowe, dopasowane do specyfiki urządzenia (inne dla drukarki, inne dla monitora).

Można pisać programy zależne od docelowego urządzenia, warto jednak robić to tylko wtedy, gdy domyślne ustawienie tego urządzenia nie pasuje do specyfiki zastosowania (ale zdarza się to bardzo, *bardzo* rzadko).

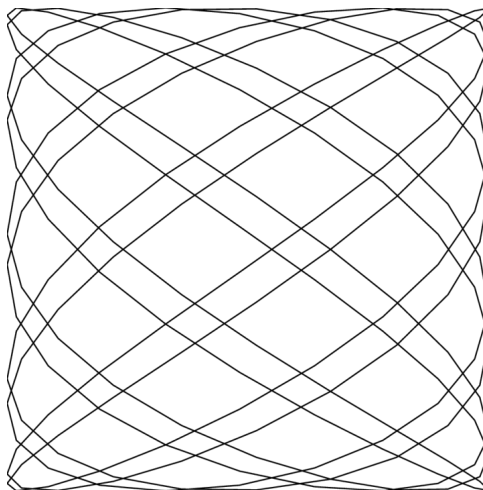
Program GhostScript jest interpreterem języka PostScript, opracowanym przez firmę Aladdin Software. Może on się przydać jako przeglądarka ekranowa, albo sterownik drukarki nie-postscriptowej, który czyni z niej drukarkę postscriptową. W odróżnieniu od większości produktów firmy Adobe, jest dostępny za darmo.

W ostatnim czasie PostScript traci nieco na popularności na rzecz języka PDF (ang. *portable document format*), też opracowanego przez firmę Adobe. Pliki PDF są binarne (w związku z czym zajmują mniej miejsca) i pozwalają na tworzenie hipertekstu, co przydaje się w pracy z dokumentami elektronicznymi. Do oglądania plików PDF można użyć programu Adobe Acrobat Reader (jest za darmo), ale również GhostScriptu.

Ostatnia sprawa — nazwa. Wzięła się ona od notacji przyrostkowej (ang. *postfix*), czyli odwrotnej notacji polskiej Łukasiewicza. Notacja ta pozwala na beznawiasowy zapis wyrażeń arytmetycznych. Interpreter PostScriptu jest maszyną stosową której zadaniem jest przetwarzanie kolejnych symboli takich wyrażeń.

13.2. Przykład wstępny

Podany niżej program tworzy pokazany obrazek.



Listing.

```

1: %!
2: /nx 10 def
3: /ny 7 def
4: /phi 20 def
5: /size 100 def
6: /transx 20 def
7: /transy 150 def
8: /steps 200 def
9: /cpoint {
10:   steps div 360 mul dup
11:   phi add nx mul sin 1 add size mul transx add
12:   exch
13:   ny mul cos 1 add size mul transy add
14: } def
15: newpath
16: 0 cpoint moveto
17: 1 1 steps 1 sub { cpoint lineto } for
18: closepath
19: stroke
20: showpage

```

Powyższy program służy do narysowania łamanej przybliżającej pewną krzywą Lissajous; zmieniając stałe w programie można otrzymywać różne krzywe. Liczby z dwukropkami są numerami linii i nie należy ich pisać w pliku postscriptowym.

Znak % (z wyjątkiem, gdy należy do napisu, o czym dalej) oznacza komentarz — zaczynając od niego do końca linii wszystkie znaki są ignorowane przez interpreter. Ludzie komentarze w programach powinni pisać i czytać. Dwa pierwsze znaki w pliku, %!, oznaczają, że jest to plik postscriptowy. Bez nich próba wydrukowania zakończyłaby się otrzymaniem tekstu pliku, zamiast odpowiedniego obrazka.

W kolejnych liniach jest ciąg symboli; część z nich to symbole *literalne*, a pozostałe są *wykonywalne*. Interpreter wstawia na stos symbole literalne, natomiast przetwarzanie symbolu wykonywalnego polega na wykonaniu odpowiedniej procedury. Procedura ta może mieć pewną

liczbę parametrów — to są obiekty obecne na stosie. Procedura może zdjąć ze stosu pewną liczbę obiektów i wstawić inne.

Na przykład, w linii 2 napis `nx` to jest symbol literalny, który jest nazwą (w terminologii PostScriptu — kluczem); następnie mamy symbol literalny `10`, który reprezentuje liczbę całkowitą. Symbol wykonywalny **def** powoduje wywołanie procedury przypisania, która spodziewa się znaleźć na stosie dwa parametry: nazwę i obiekt, który ma być skojarzony z nazwą. W Pascalu to samo zapisuje się w postaci `nx := 10;`.

W liniach 9–14 mamy tekst procedury, która, za pomocą operatora **def** (w linii 14) będzie przypisana nazwie `cpoint`. Umieszczenie na stosie symbolu `{` powoduje, że kolejne symbole będą traktowane jak literalne, aż do pojawienia się (do pary) klamry zamykającej `}`. Jest ona symbolem wykonywalnym i powoduje utworzenie obiektu, który jest procedurą. Obiekt ten jest umieszczony na stosie i operator **def** w linii 14 znowu znajduje dwa parametry zamiast zdjętych ze stosu symboli między nawiasami klamrowymi: nazwę `cpoint` i procedurę, która zostaje przypisana tej nazwie.

Linia 15: operator (wykonywalny; nazwy wykonywalne nie mają znaku `/` na początku) **newpath** zapoczątkowuje nową tzw. *ścieżkę*; wyznaczy ona w tym przypadku krzywą do narysowania (ale może też wyznaczyć brzeg obszaru do zamalowania, albo brzeg obszaru, poza którym malowanie będzie zabronione).

Zbadajmy teraz, co robi procedura `cpoint`. Znajduje ona na stosie 1 parametr, który powinien być liczbą z przedziału $0 \dots \text{steps}-1$. Oznaczmy go literą *i*; procedura `cpoint` ma obliczyć

$$\begin{aligned}x &= (\sin((i/\text{steps} * 360 + \text{phi}) * \text{nx}) + 1) * \text{size} + \text{transx}, \\y &= (\cos((i/\text{steps} * 360) * \text{ny}) + 1) * \text{size} + \text{transy},\end{aligned}$$

i zostawić na stosie liczby *x* i *y*. W linii 10 mamy kolejno: dzielenie *i* przez `steps` (operator **div**), mnożenie wyniku przez 360 (operator **mul**) i wstawienie na stos dodatkowej kopii tego ostatniego wyniku (operator **dup**). Dalej — dodanie `phi` (**add**), mnożenie przez `nx`, obliczenie sinusa (operator **sin**, kąt jest podawany w stopniach) itd. Po wykonaniu ostatniego **add** w linii 11 mamy wartość *x* na wierzchołku stosu.

Operator **exch** zamienia miejscami *x* z obiektem „pod spodem”, po czym (w linii 13) następuje obliczenie *y*.

W linii 16 mamy przykład wywołania procedury: `0` (umieszczone na stosie) jest parametrem procedury `cpoint`, po wykonaniu której na stosie są 2 argumenty *x* i *y* operatora **moveto**. Umieszcza on bieżącą pozycję (którą można sobie wyobrażać jako coś w rodzaju pisaka) w punkcie (*x*, *y*); ścieżka zaczyna się w tym punkcie, a parametry operatora **moveto** zostają usunięte ze stosu.

Kolejne 199 (tj. `steps-1`) punktów — wierzchołków łamanej, która ma być ścieżką, jest otrzymywane za pomocą operatora **for**, który realizuje pętlę. Pierwsze trzy jego parametry to wartość początkowa zmiennej sterującej (coś jak `i` w Pascalowym `for i := 1 to steps-1 do ...`) oraz przyrost i wartość końcową. Mogą to być liczby rzeczywiste. Czwarty argument to procedura (utworzona za pomocą klamer); operator **for** wywoła ją odpowiednią liczbą razy, za każdym razem wstawiając uprzednio na stos wartość zmiennej sterującej. W naszym przykładzie będzie ona parametrem procedury `cpoint`. Operator **lineto** wydłuża ścieżkę do punktu o współrzędnych *x*, *y* zdjętych ze stosu (coś jakby przesuwiał pisak). Zauważmy, że procedura wywoływana przez **for** w końcowym efekcie czyści stos ze zmiennej sterującej (i powinna to robić).

Operator **closepath** łączy koniec ścieżki z jej początkiem. Operator **stroke** wykonuje rysowanie łamanej. Nie bierze on argumentów ze stosu, ale przed jego wywołaniem musi być przygotowana stosowna ścieżka. Operator **showpage** powoduje wydrukowanie strony i przygotowanie interpretera do rysowania na następnej.

13.3. Operatory PostScriptu (wybrane dość arbitralnie)

Język PostScript zawiera wszystkie operatory arytmetyczne, logiczne i inne, jakich można się spodziewać w języku programowania. Operatory te są realizowane przez procedury, które pobierają argumenty ze stosu i pozostawiają na nim wyniki.

Poniższa lista zawiera prawie wszystkie operatory udostępniane przez interpreter, które przydają się w codziennej pracy z PostScriptem. Opis pozostałych można znaleźć w licznych podręcznikach poświęconych wyłącznie temu językowi i w dokumentacji firmowej, której przepisywanie nie byłoby wskazane.

Zgodnie z przyjętym zwyczajem, który jest bardzo wygodny, operator przedstawia się w ten sposób, że przed nim są wymienione argumenty (w kolejności wstawiania na stos), a po nim argumenty, które dany operator na stosie zostawia.

13.3.1. Operatory arytmetyczne

Litera n oznacza, że argument może być liczbą całkowitą lub rzeczywistą; litera i oznacza, że dopuszczalna jest tylko liczba całkowita. Typ wyniku zależy od wykonanej operacji i od typu argumentów, w sposób zgodny z intuicją.

n_1	n_2	add	n	(suma)
n_1	n_2	div	n	(iloraz n_1/n_2)
i_1	i_2	idiv	i	(część całkowita ilorazu i_1/i_2)
i_1	i_2	mod	i	(reszta ilorazu i_1/i_2)
n_1	n_2	mul	n	(iloczyn)
n_1	n_2	sub	n	(różnica $n_1 - n_2$)
	n	abs	n	(wartość bezwzględna)
	n	neg	n	(zmiana znaku)
	n	ceiling	n	(zaokrąglanie w górę)
	n	floor	n	(zaokrąglanie w dół)
	n	round	i	(zaokrąglenie)
	n	truncate	i	(obcięcie)
	n	sqrt	n	(pierwiastek kwadratowy)
n_1	n_2	atan	n	($\arctan n_1/n_2$, w stopniach)
	n	cos	n	(kosinus n)
	n	sin	n	(sinus n)
n_1	n_2	exp	n	($n_1^{n_2}$)
	n	ln	n	(logarytm naturalny)
	n	log	n	(logarytm dziesiętny)
	—	rand	i	(liczba losowa)
	i	srand	—	(inicjalizacja generatora liczb losowych)
	—	rrand	i	(wartość ziarna generatora l. losowych)

Generator liczb losowych wytwarza oczywiście liczby pseudolosowe, tj. elementy okresowego ciągu liczb o bardzo długim okresie. Jeśli program inicjalizuje ziarno generatora (tj. zmienną liczbową, która określa miejsce kolejnego elementu ciągu, który ma podać), to program może wygenerować „losowy” obrazek, który za każdym razem będzie identyczny. Liczby generowane przez operator **rand** są całkowite, z przedziału od 0 do $2^{31} - 1$.

13.3.2. Operacje na stosie argumentów

Ze względu na rolę jaką pełni stos argumentów, operatory obsługujące ten stos są używane często. Litera d niżej oznacza argument dowolnego typu.

d	pop	—	(usuwa obiekt ze stosu)
$d_1 d_2$	exch	$d_2 d_1$	(zamienia)
d	dup	$d d$	(podwaja)
$d_1 \dots d_k k$	copy	$d_1 \dots d_k d_1 \dots d_k$	(kopiuje k obiektów)
$d_{k-1} \dots d_0 k i$	roll	$d_{i-1} \dots d_0 d_{k-1} \dots d_i$	(przestawia)

13.3.3. Operatory relacyjne i logiczne

Operatory relacyjne służą do badania warunków i można ich użyć w celu sterowania przebiegiem obliczeń (warunkowe wykonanie podprogramu, zakończenie pętli itd.). Operatory logiczne realizują koniunkcję, alternatywę itp. Te same operatory zastosowane do liczb całkowitych realizują odpowiednie operacje na poszczególnych bitach argumentów, przy czym 0 jest uważane za fałsz, a 1 za prawdę.

Litera b oznacza obiekt boolowski, o możliwych wartościach **true** lub **false**. Litera s oznacza napis, porównania napisów są leksykograficzne.

d_1	d_2	eq	b	(test równości)
d_1	d_2	ne	b	(test nierówności)
n/s_1	n/s_2	ge	b	} (relacje między liczbami albo napisami)
n/s_1	n/s_2	gt	b	
n/s_1	n/s_2	le	b	
n/s_1	n/s_2	lt	b	
b/i_1	b/i_2	and	b/i	} (operacje boolowskie i bitowe)
b/i_1	b/i_2	or	b/i	
b/i_1	b/i_2	xor	b/i	
	b/i	not	b/i	
—		true	b	
—		false	b	

13.3.4. Operatory sterujące

Operatory sterujące służą do warunkowego lub wielokrotnego wykonywania różnych części programu. Realizują one pełny repertuar „instrukcji strukturalnych”, które umożliwiają sterowanie przebiegiem programu, wykonywanie obliczeń iteracyjnych itd. Zapis tych konstrukcji jest oczywiście przyrostkowy, w polskiej notacji odwrotnej. Napis **proc** oznacza procedurę, która jest przez podane niżej operatory wykonywana w określonych warunkach.

b	proc	if	—	(np. $1\ 2\ \mathbf{eq}\ \{ \dots \}\ \mathbf{if}$)
$b\ \mathit{proc}_1\ \mathit{proc}_2$		ifelse	—	
$i_1\ i_2\ i_3\ \mathit{proc}$		for	—	(i_1 jest wartością początkową, i_2 przyrostem, a i_3 wartością końcową zmiennej sterującej pętli)
$i\ \mathit{proc}$		repeat	—	(pętla powtarzana i razy)
proc		loop	—	(pętla „bez końca”)
—		exit	—	(wyjście z pętli)
—		quit	—	(wyjście z interpretera)

Procedura będąca argumentem operatora **if** jest wykonywana wtedy, gdy warunek b jest prawdziwy. Operator **ifelse** wykonuje procedurę proc_1 jeśli b albo proc_2 w przeciwnym razie.

Operator **for** zdejmuje ze stosu swoje cztery argumenty, a następnie wykonuje procedurę proc w pętli; za każdym razem przed wywołaniem procedury wstawia na stos wartość zmiennej sterującej; jej wartość zmienia się od i_1 z krokiem i_2 ; koniec pętli następuje jeśli wartość zmiennej sterującej jest większa niż i_3 (jeśli krok jest dodatni) albo jeśli jest mniejsza niż i_3 (jeśli krok jest ujemny). Procedura powinna (ale nie musi) usunąć ze stosu argumentów wartość zmiennej sterującej.

Operator **repeat** zdejmuje ze stosu swoje dwa argumenty, a następnie wykonuje procedurę *proc* *i* razy. Operator **loop** wykonuje procedurę wielokrotnie; zakończenie tej iteracji może nastąpić tylko wskutek wykonania operacji **exit** albo **quit**; inne pętle też mogą być przerwane w ten sposób. Operator **quit** kończy w ogóle działanie interpretera.

13.3.5. Operatory konstrukcji ścieżki

Dotychczas opisane operatory odpowiadają za konstrukcje dostępne w dowolnym języku programowania. Obecnie pora na grafikę; większość procedur rysowania wiąże się z tworzeniem i przetwarzaniem ścieżek, które są w ogólności łamanymi krzywoliniowymi.

—	newpath	—	(inicjalizacja pustej ścieżki)
<i>x y</i>	moveto	—	(ustawienie punktu początkowego)
<i>x y</i>	rmoveto	—	(ustawienie punktu początkowego względem bieżącej pozycji)
<i>x y</i>	lineto	—	(przedłużenie ścieżki o odcinek)
<i>x y</i>	rlineto	—	(przedłużenie o odcinek o końcu określonym względem bieżącej pozycji)
<i>x y r a₁ a₂</i>	arc	—	(przedłużenie o łuk okręgu)
<i>x₁ y₁ x₂ y₂ x₃ y₃</i>	curveto	—	(krzywa Béziera trzeciego stopnia)
—	closepath	—	(zamknięcie ścieżki)
—	currentpoint	<i>x y</i>	(zapisanie na stosie współrzędnych bieżącej pozycji)

Operatory konstrukcji ścieżki służą do określania krzywych złożonych z odcinków, łuków okręgów i krzywych Béziera. Ścieżka może następnie być narysowana jako linia, może być też wypełniona lub posłużyć do obcinania (podczas rysowania interpreter nie zmienia pikseli poza obszarem, którego brzegiem jest aktualna ścieżka obcinania).

13.3.6. Operatory rysowania

Operatory rysowania to te, których interpretacja powoduje przypisanie pikselom obrazu wartości. Operatory te (poza **erasepage** i **show**) wymagają wcześniejszego przygotowania ścieżki. Operator **show** tworzy ścieżkę opisującą odpowiednie litery, a następnie wypełnia ją, wywołując **fill**.

—	erasepage	—	(czyszczenie strony)
—	fill	—	(wypełnianie ścieżki)
—	eofill	—	(wypełnianie ścieżki z parzystością)
—	stroke	—	(rysowanie ścieżki jako linii)
<i>s</i>	show	—	(rysowanie liter napisu)

13.3.7. Operatory związane ze stanem grafiki

Stan grafiki to struktura danych zawierająca informacje takie jak bieżący kolor, grubość linii, wzorec linii przerywanych i wiele innych. Poniżej są wymienione tylko najważniejsze operatory związane ze stanem grafiki.

<i>n</i>	setlinewidth	—	(ustawianie grubości kreski)
<i>n</i>	setgray	—	(ustawianie poziomu szarości)
<i>r g b</i>	setrgbcolor	—	(ustawianie koloru)
—	gsave	—	(zachowanie stanu grafiki)
—	grestore	—	(przywrócenie stanu grafiki)

13.4. Napisy i tworzenie obrazu tekstu

Jednym z najważniejszych zastosowań języka PostScript jest tworzenie obrazów tekstu; wiele wydrukowanych stron zawiera tylko tekst. Aby otrzymać obraz tekstu należy utworzyć odpowiednie napisy (literały napisowe), rozmieścić je na stronie (tym najczęściej zajmują się systemy składu), wybrać odpowiednie kroje i wielkości czcionek i spowodować utworzenie obrazów tych czcionek.

Literał napisowy jest to ciąg znaków, umieszczony w nawiasach okrągłych, np. (napis). Może on zawierać dowolne, połączone w pary nawiasy okrągłe, które są wtedy przetwarzane bez problemów. Jeśli trzeba narysować nawias bez pary, pisze się \ (albo \). Inne zastosowania znaku \ to opisywanie znaków specjalnych, trudno dostępnych lub niedostępnych w kodzie ASCII.

\n	—	znak końca linii (LF, ASCII 10),
\r	—	cofnięcie karetki (CR, ASCII 13),
\t	—	tabulator,
\b	—	cofnięcie,
\f	—	wysuw strony,
\\	—	znak „\”,
\ddd	—	trzy cyfry ósemkowe, mogą określać dowolny znak od \000 ₈ do \377 ₈ .

Napis może być podany w kilku liniach i wtedy *zawiera* znaki końca linii, chyba że ostatni znak w linii to \ (pojedynczy znak \ jest przy tym ignorowany). Dla porządku wspomnę, że są jeszcze inne sposoby zapisywania napisów; ciąg (o parzystej długości) cyfr szesnastkowych w nawiasach <> (np. <1c3F>) jest często stosowany do reprezentowania obrazów rastrowych (kolejne dwie cyfry dają kod szesnastkowy kolejnego bajtu). Jest jeszcze inny sposób, który pozwala „pakować” dane (4 znaki napisu zakodowane w pięciu znakach „drukowalnych”), ale to zostawmy.

Aby wykonać napis na tworzonej stronie, trzeba najpierw wybrać krój i wielkość pisma. Przykład:

napis

Listing.

```
/Times–Roman findfont 32 scalefont setfont
100 100 moveto
(napis) show
```

Nazwa literalna /Times–Roman oznacza krój pisma o nazwie Times New Roman. Jest to antykwa szeryfowa dwuelementowa, będąca dwudziestowieczną wersją tzw. antykiw renesansowej. Została ona zaprojektowana w 1931r. dla dziennika The Times przez zespół pracujący pod kierunkiem Stanleya Morisona. Jej cechą charakterystyczną to wąskie litery, umożliwiające zmieszczenie dużej ilości tekstu na stronie.

Inne kroje pisma dostępne zawsze w PostScriptcie, to np. /Palatino (krój Palatino, zaprojektował go Hermann Zapf w 1948 r.), /Helvetica (Helvetica, antykwa bezszeryfowa jednoelementowa, Max Miedinger, 1956 r.), /Courier (Courier, krój pisma „maszynowego”, w którym każdy znak ma tę samą szerokość). Istnieją wersje pogrubione (np. /Times–Bold) i pochyłe (kursywy, np. /Times–Italic, /Times–BoldItalic), a także zestawy znaków specjalnych (/Symbol i /ZapfDingbats).

Ponadto istnieją tysiące krojów dostępnych za darmo i (zwłaszcza) komercyjnych, którymi można składać teksty i opisywać rysunki. Jednak dodatkowe zestawy znaków trzeba albo specjalnie doinstalować, albo umieścić w programie PostScriptowym (zwykle na początku).

Operator **findfont** wyszukuje krój o podanej nazwie i umieszcza na stosie obiekt (dokładniej: słownik, o słownikach będzie dalej) reprezentujący ten krój.

Operator **scalefont** skaluje czcionki w podanej proporcji. Domyślnie mają one wysokość 1 punktu (1/72 cala), czyli bez lupy są nieczytelne. Dokładniej — jest to „wysokość projektowa”, mają ją na przykład znaki nawiasów. Obiekt reprezentujący zestaw przeskalowanych znaków pozostaje na stosie, operator **setfont** zdejmuje go ze stosu i ustawia pisanie tymi znakami w bieżącym stanie grafiki.

Polecenie 100 100 **moveto** w przykładzie ustawia początek napisu, który jest następnie malowany przez operator **show**.

Nieco większy przykład:



Listing.

```
%!
shadeshow {
  /s exch def
  /y exch def
  /x exch def
  /g 1 def
  40 {
    /g g 0.025 sub dup setgray def
    x y moveto
    s show
    /x x 0.5 add def
    /y y 0.5 sub def
  } repeat
  1 setgray
  x y moveto
  s show
} def
/Times-Roman findfont 32 scalefont setfont
100 200 (napis) shadeshow
showpage
```

Procedura shadeshow w przykładzie otrzymuje za pośrednictwem stosu 3 parametry: napis i współrzędne jego początku. Przykład pokazuje, jak zdjąć je ze stosu, przypisując ich wartości nazwanym zmiennym. Ponieważ wszystkie pozostałe elementy były podane już wcześniej, proponuję **ćwiczenie**, polegające na takim przerobieniu przykładu, aby zamiast operatora **repeat** był użyty operator **for**, ze zmienną sterującą odpowiadającą poziomowi szarości (zamiast zmiennej g).

Dygresja na temat polskich liter: problem jest zwykle dosyć trudny, ale nie beznadziejny. Jego rozwiązanie zależy od konkretnego zestawu znaków i sposobu ich kodowania. W standardowym kodowaniu (Adobe Standard Encoding) mamy znaki:

```
\350 — Ł,
\370 — ł,
\302 — ´ (akcent do ć, ń, ś, ó, ź, Č, Ň, Š, Ó, Ž),
\316 — fi (ogonek do ą, ę, Ą, ě),
\307 — Ω (kropka do ż i Ž).
```

Ponieważ zestawy znaków można przekodowywać (tj. inaczej wiązać znaki z kodami, tj. wartościami bajtów w napisie), więc znaki te mogą być dostępne pod innymi kodami, albo niedostępne.

Położenie kropki do „ż” i kreski do „ć” jest odpowiednie dla małych liter; dla wielkich liter znaki diakrytyczne muszą być odpowiednio podniesione. Aby wypisać cały alfabet, możemy użyć programu

Listing.

```
/Times–Roman findfont 30 scalefont setfont
100 100 moveto
(P) show currentpoint (\302) show moveto (ojd) show
currentpoint (\302) show moveto (z) show currentpoint (\307) show moveto (ze, ki) show
currentpoint (\302) show moveto (n t) currentpoint (\316) show moveto (e chmurno) show
currentpoint (\302) show moveto (s) show currentpoint (\302) moveto (c w g\370) show
currentpoint (\316) show moveto (ab flaszyl) show
```

Procedura **currentpoint** zapisuje na stosie bieżącą pozycję przed narysowaniem znaku diakrytycznego, a procedura **moveto** ją przywraca, dzięki czemu np. litera „o” jest rysowana na właściwym miejscu, pod kreską, tworząc „ó”.

Wracając do poprzedniego przykładu; po ostatniej linijce procedury (s **show**) dopiszmy jeszcze

Listing.

```
0.5 setlinewidth
0 setgray
newpath x y moveto s false charpath stroke
```



Operator **charpath** otrzymuje dwa parametry. Pierwszy to napis, a drugi parametr jest boolowski, **false** albo **true**. Wynikiem działania operatora **charpath** jest utworzenie zarysu liter i dołączenie ich do bieżącej ścieżki. Ścieżkę tę w przykładzie wykreślił operator **stroke**. Drugi parametr powinien mieć wartość **false** wtedy, gdy ścieżkę chcemy wykreślić (tak jak w tym

przykładzie). Wartość **true** przygotowuje ścieżkę do wypełniania/obcinania (wersja GhostScriptu, z którą sprawdzałem te przykłady, nie daje widocznych różnic, ale dla innych interpreterów języka PostScript może to mieć istotne znaczenie).

Zmieńmy teraz ostatnią linię procedury na

Listing.

```
newpath x y moveto s true charpath clip
```

a po wywołaniu procedury **charpath** (przed **showpage**) dopiszmy

Listing.

```
300 -5 150 {  
  newpath 0 exch moveto 500 0 rlineto stroke  
} for
```



Jak widać, tylko kreski wewnątrz liter są narysowane; cokolwiek innego byśmy chcieli dalej narysować, ukaże się tylko część wspólna tego czegoś i liter napisu.

Częścią *stanu grafiki* jest tzw. ścieżka obcinania; początkowo jest ona brzegiem strony. Można utworzyć dowolną zamkniętą ścieżkę i za pomocą operatora **clip** ograniczyć rysowanie do obszaru, który jest częścią wspólną obszaru ograniczonego poprzednio ustawionymi ścieżkami i obszaru, którego brzeg stanowi ścieżka właśnie utworzona. W ten sposób można rysowanie uniemożliwić całkowicie; jeśli chcemy przywrócić możliwość rysowania poza obszarem ograniczonym dawną ścieżką, to powinniśmy *przed* wywołaniem operatora **clip** napisać **gsave**; późniejsze wywołanie operatora **grestore** przywróci stan grafiki (cały) sprzed wywołania **gsave**, łącznie ze ścieżką obcinania.

13.5. Słowniki

Zmienne w procedurze nie są lokalne; możemy mieć lokalne zmienne, tworząc *słowniki*. Przykład:

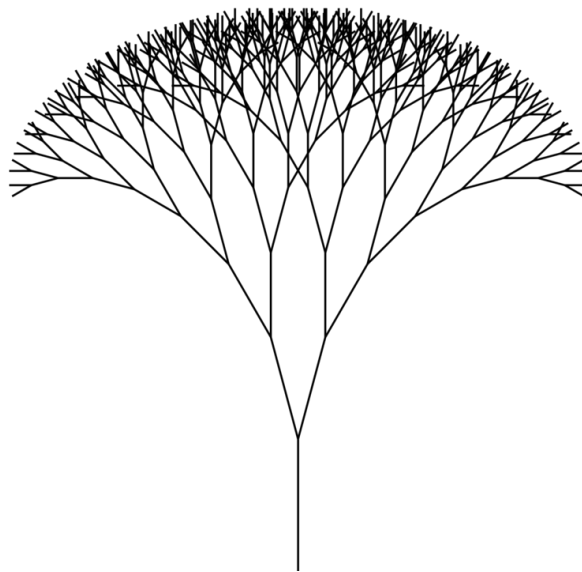
Listing.

```
%!  
/tree {  
  4 dict begin  
    /a exch def  
    /l exch def  
    /y exch def  
    /x exch def  
    l 10 ge {
```

```

newpath
x y moveto
a cos | mul a sin | mul rlneto
currentpoint
stroke
| 0.8 mul 3 copy
a 15 add tree
a 15 sub tree
} if
end
} def
200 100 70 90 tree
showpage

```



Mamy tu rekurencyjną procedurę, która ma lokalne zmienne (x , y , l , a) i przypisuje im wartości parametrów zdjętych ze stosu. Nie można ich przypisywać zmiennym globalnym, bo rekurencyjne wywołania zniszczą ich wartości. Dlatego procedura tworzy słownik, czyli wykaz par nazwa/skojarzony z nią obiekt. Słownik ten zostaje umieszczony na stosie słowników; ma on pojemność 4 obiektów i to w nim operator **def** wywołany w procedurze tworzy klucze i przypisuje im znaczenie.

Operator **dict** tworzy obiekt — słownik, którego pojemność jest określona za pomocą parametru, i umieszcza go na stosie argumentów. Operator **begin** zdejmuje obiekt ze stosu argumentów; powinien to być słownik. Słownik ten zostaje umieszczony na stosie słowników i otwarty do czytania i pisania. Operator **end** usuwa go ze stosu słowników.

Można utworzyć dowolnie dużo słowników, ponazywać je i trzymać w nich różne zestawy informacji. Na przykład, program

Listing.

```
/słowik 20 dict def
```

tworzy słownik o pojemności 20 miejsc i przypisuje go nazwie słowik. Później można napisać

Listing.

```
słowik begin
```

co spowoduje umieszczenie tego słownika na stosie słowników i możliwość czytania i pisania w nim. Operator **def** zmienia zawartość słownika na szczycie stosu; jeśli natomiast w programie pojawi się nazwa wykonywalna, to słowniki są przeszukiwane kolejno, zaczynając od wierzchołka stosu, aż do znalezienia obiektu skojarzonego z tą nazwą. Na początku działania interpretera na stosie są

- systemdict** — słownik tylko do czytania, zawiera nazwy wszystkich operatorów wbudowanych w interpreter PostScriptu i standardowe fonty,
- globaldict** — słownik do czytania/pisania w tzw. globalnej pamięci wirtualnej (nie będziemy w to wnikać),
- userdict** — słownik do czytania/pisania w tzw. lokalnej pamięci wirtualnej; to w nim są tworzone obiekty przez **def**, jeśli nie został utworzony inny słownik.

Oprócz tego istnieją słowniki opisujące kroje pisma, wzorce tworzenia półtonów i inne, ale nie są one na stosie — można je tam umieścić, posługując się nazwami obecnymi w słowniku **systemdict**.

13.6. Stosy interpretera

Interpreter przetwarza cztery stosy; **stos argumentów** (na którym są umieszczane kolejne symbole literalne programu), **stos słowników**, opisany w poprzednim punkcie, **stos stanów grafiki** (obsługiwany za pomocą operatorów **gsave** i **grestore** i **stos wywołanych procedur**, w którym przechowuje się adresy powrotne. Wszystkie cztery stosy działają niezależnie, tj. można wstawiać na każdy z nich i zdejmować obiekty bez związku z kolejnością działań na pozostałych stosach.

13.7. Operatory konwersji

- | | | | |
|-----------|-------------|-----|---|
| n/s | cvi | i | (konwersja liczby rzeczywistej albo napisu na l. całkowitą) |
| n/s | cvr | n | (konwersja liczby lub napisu na l. rzeczywistą) |
| $n\ s$ | cvs | s | (konwersja w układzie dziesiętnym) |
| $n\ r\ s$ | cvrs | s | (konwersja w układzie o podstawie r) |

Argument s jest napisem, czyli tablicą znaków, którą trzeba wcześniej utworzyć. Dla operatora **cvi** powinien napis ten powinien składać się z samych cyfr (z ewentualnym znakiem na początku); dla **cvr** może zawierać mnożnik, który jest potęgą 10, np. 3.14e-5. Operatory **cvs** i **cvrs** wymagają podania liczby n poddawanej konwersji, podstawy r (np. 10 — tylko ten ostatni) i tablicy, w której mają być umieszczone znaki (głównie cyfry) napisu reprezentującego liczbę n . Do utworzenia takiej tablicy służy operator **string**, na przykład fragment programu

Listing.

```
/temp 12 string def
```

tworzy napis o długości 12 znaków. Początkowo otrzymują one wartość 0.

Pierwszy argument operatora **cvs** nie musi być liczbą; jeśli jest to obiekt reprezentujący wartość boolowską, to **cvs** utworzy napis **true** albo **false**; jeśli argument jest nazwą operatora, to otrzymamy napis – nazwę. W pozostałych przypadkach (np. słownik, tablica, procedura) wystąpi błąd.

13.8. Przekształcenia afiniczne

Współrzędne punktów we wszystkich dotychczasowych przykładach były podawane w układzie, którego początek pokrywa się z lewym dolnym rogiem strony, oś x jest pozioma, oś y — pionowa, a jednostką długości jest 1 punkt, czyli $1/72$ cala (obecna definicja to $1\text{cal} = 25,4\text{mm}$, do roku 1959 obowiązywał nieco większy cal, taki że $1\text{cm} = 0.3937\text{cala}$).

Jeśli ktoś chciałby umieścić początek układu w innym punkcie, to może rysowanie opisać wyłącznie za pomocą komend „względnych”, np. **rlneto** i wtedy wystarczy zmienić tylko punkt startowy. Ale:

1. to załatwia tylko przesunięcia,
2. może być niewygodne,
3. może być niewykonalne, jeśli gotowy obrazek postscriptowy chcemy wkomponować w inny obrazek.

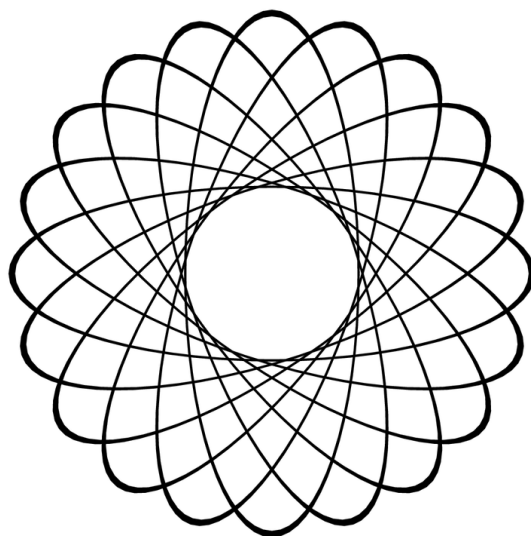
Operator **translate** otrzymuje dwa parametry, które opisują współrzędne (w dotychczasowym układzie) początku nowego układu, który będzie odtąd używany. Kierunki osi i jednostki długości obu układów są takie same.

Dwuargumentowy operator **scale** służy do zmiany jednostek długości; układ współrzędnych, który obowiązuje po jego zastosowaniu ma ten sam początek i kierunki osi; pierwszy argument określa skalowanie osi x , a drugi y . Rysunek wykonany po poleceniach **2 dup scale** jest 2 razy większy niż byłby bez tego. Podając różne współczynniki skalowania, np. **2 3 scale**, możemy spowodować, że polecenie rysowania okręgu spowoduje narysowanie elipsy.

Jednoargumentowy operator **rotate** pozwala rysunek obrócić; argument określa kąt obrotu w stopniach, w kierunku przeciwnym do zegara. Operatory **scale** i **rotate** mają punkt stały, który jest początkiem dotychczasowego układu, określonego przez poprzednio wykonane przekształcenia. To działa tak, że jeśli mamy fragment programu w PostScriptcie, który coś rysuje, to cokolwiek w nim byśmy przekształcali (z wyjątkiem, o którym później), jeśli *poprzedzimy* go pewnym przekształceniem, to odpowiednio przekształcimy ten rysunek w całości. Dzięki temu program, który umieszcza rysunek postscriptowy na stronie (w odpowiednim położeniu względem tekstu), może go poprzedzić przekształceniami, które ustalają odpowiednią wielkość i pozycję.

Dodatkowo, taki program okłada kod opisujący rysunek poleceniami **gsave** i **grestore**; może też ustawić ścieżkę obcinania (aby kod rysunku nie mógł mazać po tekście), utworzyć nowy słownik dla rysunku (aby wszystkie skutki działania operatora **def** w kodzie rysunku zlikwidować za końcem rysunku) i w słowniku tym wykonuje polecenie **/showpage {} def**, dzięki czemu polecenie **showpage** w pliku z obrazkiem nie spowoduje wydrukowania niekompletnej strony.

Przykład:



Listing.

```
%!
/ell {
  10 {
    1 3 scale
    newpath
    0 0 80 0 360 arc stroke
    1 1 3 div scale
    18 rotate
  } repeat
} def
2 setlinewidth
297 421 translate
ell
showpage
```

Skalowanie zostało wykorzystane do otrzymania elipsy o półosiach o długościach 80 i 240; po narysowaniu elipsy wracamy do nieprzeskalowanych jednostek. Grubość linii, którymi elipsy są narysowane, zmienia się, co jest spowodowane tym, że operator **stroke** zamienia ścieżkę opisującą elipsę na dwie krzywe, między którymi jest obszar zamalowywany na czarno. Krzywe te są równoodległe w bieżącym układzie współrzędnych, o różnych jednostkach długości osi w tym przypadku.

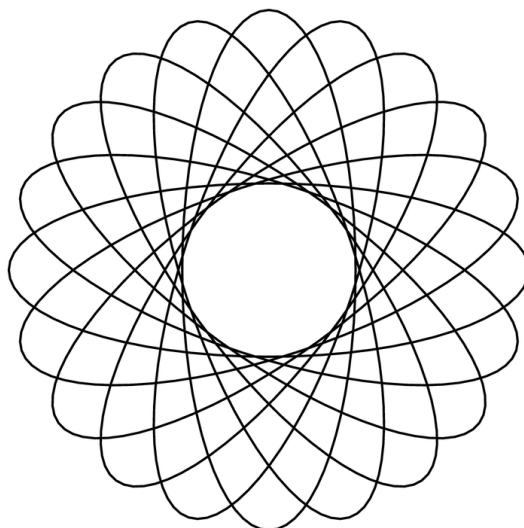
Pisząc powyższy przykład zrobiłem błąd, który jest wart obejrzenia. Zapisalem procedurę tak:

Listing.

```
/ell {
  1 3 scale
  10 {
    newpath
    0 0 80 0 360 arc stroke
    18 rotate
  } repeat
} def
```

Jaki był skutek i dlaczego? (proszę odpowiedzieć bez pomocy komputera).

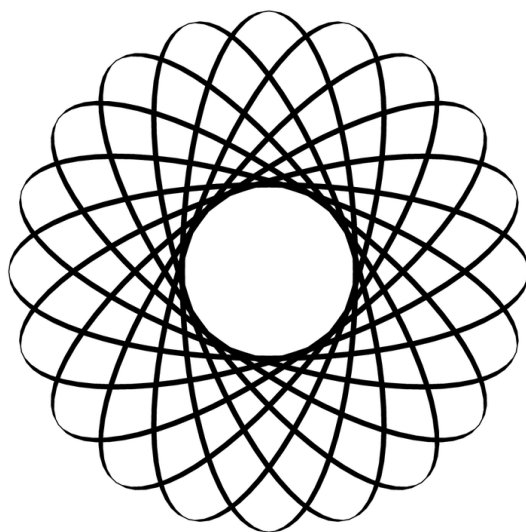
Teraz modyfikacja:



Listing.

```
%!
/ell {
  newpath
  10 {
    1 3 scale
    80 0 moveto
    0 0 80 0 360 arc
    1 1 3 div scale
    18 rotate
  } repeat
  stroke
} def
2 setlinewidth
297 421 translate
ell
showpage
```

Linie mają teraz grubość stałą, bo operator **scale** działa w układzie, którego jednostki osi mają tę samą długość. Polecam jako ćwiczenie zastanowienie się, jak narysować takie coś jak na rysunku poniżej.



Zamieńmy w ostatnim przykładzie **stroke** na **fill** lub **eofill** i obejrzymy skutki.

W powyższych przykładach zmiany układu współrzędnych są zrobione w sposób dość niedołążny. Chodzi o parę **1 3 scale** i **1 1 3 div scale**. Po pierwsze, tę samą stałą powtórzyłem w dwóch miejscach, a po drugie, wskutek błędów zaokrągleń nie przywracamy dokładnie stanu poprzedniego (w przykładzie na rysunku tego nie widać, ale błędy mogą wyleźć w poważniejszych zastosowaniach). Nie można w celu przywrócenia poprzedniego układu użyć pary **gsave** – **grestore**, bo to by zniszczyło konstruowaną ścieżkę. Możliwe jest takie rozwiązanie:

Listing.

```
/ell {
  newpath
  10 {
    [ 0 0 0 0 0 0 ] currentmatrix
    1 3 scale
    80 0 moveto
    0 0 80 0 360 arc closepath
    setmatrix
    18 rotate
  } repeat
  eofill
} def
```

Bieżący układ współrzędnych, a właściwie tzw. CTM (ang. *current transformation matrix*), czyli macierz przekształcenia używanego w danej chwili do obliczania punktów w układzie urządzenia, jest reprezentowana w postaci tablicy o 6 elementach. Macierz ta jest częścią stanu grafiki. Operator **currentmatrix** ma 1 argument — obiekt, który jest tablicą; operator ten wpisuje do niej współczynniki bieżącego przekształcenia i zostawia tablicę na stosie. Operator **setmatrix** przypisuje macierzy CTM współczynniki z tablicy podanej jako argument (w przykładzie — pozostawionej na stosie przez **currentmatrix**).

Samą macierz utworzyłem tu w sposób najbardziej „jawny” — przez podanie odpowiedniej liczby współczynników w nawiasach kwadratowych. Ich wartości w przykładzie są nieistotne, bo **currentmatrix** zaraz je zamaże. Można też napisać **6 array** albo **matrix**; pierwszy z tych operatorów tworzy tablicę o długości określonej przez parametr, a drugi tablicę o długości 6. Operator **matrix** dodatkowo przypisuje współczynnikom macierzy wartości reprezentujące przekształcenie tożsamościowe.

Oczywiście, aby odwoływać się do tablicy wielokrotnie, można ją nazwać, mogą być więc takie fragmenty programu, jak

Listing.

```
/tab 6 array currentmatrix def
```

Do celów specjalnych (!) służy operator **initmatrix**, który przypisuje CTM jej wartość początkową, niwecząc w ten sposób skutki wszystkich wcześniejszych operacji **translate**, **scale**, **rotate** i **setmatrix**. Z tego powodu obrazek, który został umieszczony na stronie przez program do składu, pojawi się zawsze w tym samym miejscu, jeśli na jego początku jest wywołanie **initmatrix**.

13.9. Operacje na tablicach

Jak wspomniałem, operator **[** zaczyna konstrukcję tablicy, a **]** liczy operatory na stosie, rezerwuje odpowiednie miejsce i przypisuje obiekty ze stosu elementom tablicy. Zakres indeksów tablicy zaczyna się od 0 (tak, jak w języku C). Aby „wydłubać” element tablicy, stosujemy operator **get**, np. po wykonaniu kodu **[10 21 32] 1 get** na stosie zostaje 21.

Zamiast tablicy, argumentem operatora **get** może być napis i wtedy na stosie zostaje umieszczona liczba całkowita, która jest kodem odpowiedniego znaku, np. po wykonaniu **(abcd) 1 get** zostaje liczba 98, czyli kod znaku b.

Pierwszym argumentem **get** może być też słownik; zamiast indeksu liczbowego podaje się wtedy nazwę (klucz) obiektu w słowniku, np.

Listing.

```
/mykey (napis) def
currentdict /mykey get
```

Po wykonaniu powyższego kodu na stosie zostaje (napis).

Przypisanie wartości elementowi tablicy wykonuje się za pomocą operatora **put**; ma on 3 argumenty: tablicę, indeks i obiekt, który ma być przypisany. Należy podkreślić, że tablica może zawierać obiekty różnych typów, np. liczby, napisy, tablice itd. Jeśli zamiast tablicy pierwszym argumentem **put** jest napis, to trzeci argument, czyli obiekt przypisywany, musi być liczbą całkowitą; na odpowiedniej pozycji napisu pojawi się znak, którego kodem jest ta liczba.

Zamiast tablicy lub napisu i indeksu liczbowego, można podać słownik i klucz, a więc operator **put** może być użyty do kojarzenia wartości z kluczami w dowolnym słowniku, niekoniecznie umieszczonym na stosie słowników.

Są też operatory **getinterval** i **putinterval**, które „wyjmują” i „wkładają” do tablicy lub napisu podciąg wartości:

```
a i c  getinterval  a
s i c  getinterval  s
```

Po wykonaniu operacji, na stosie pozostaje obiekt, który jest „podtablicą” lub „podnapisem” o długości c, którego pierwszym elementem jest obiekt lub znak na i-tej pozycji w pierwszym argumencie. Uwaga: to nie jest kopia odpowiednich elementów, tylko obiekt, który wskazuje elementy w podanej tablicy. Aby utworzyć kopię, należy użyć operatora **putinterval**:

```
a1 i a2  putinterval  —
s1 i s2  putinterval  —
```

Na przykład:

Listing.

```

/s1 (0123456789) def
/s2 (aaaaaaaaa) def
s2 4
s1 2 4 getinterval % wyciągnij 4 znaki z s1
putinterval % wstaw do s2, od miejsca nr 4
               % teraz s2 = (aaaa2345aa)

```

Wreszcie, istnieje operator **forall**, który pozwala wykonać pewną procedurę na wszystkich elementach tablicy, wszystkich znakach napisu, albo na wszystkich kluczach w słowniku. Pierwszym jego argumentem jest tablica/napis/słownik, drugim procedura. Jeśli pierwszy argument jest tablicą, to operator **forall** przed każdym wywołaniem procedury wstawia na stos kolejny element. Jeśli to napis, to będą to liczby całkowite od 0 do długości napisu–1. Jeśli argumentem jest słownik, to operator wstawia na stos kolejne pary klucz/wartość. W przypadku słownika kolejność kluczy jest przypadkowa.

Jeśli procedura nie usunie obiektów wstawianych na stos, to zostają tam one, co może być celowe. Wykonanie operatora **exit** w procedurze powoduje zakończenie działania jej i operatora **forall**.

13.10. Obrazy rastrowe

Często zdarza się potrzeba narysowania obrazu rastrowego, dostarczonego z zewnątrz (może to być zeskanowana fotografia lub obraz wygenerowany na przykład przez program śledzenia promieni). Rozdzielczość takiego obrazu na ogół nie ma związku z rozdzielczością rastra urządzenia, dla którego interpreter PostScriptu tworzy obraz. Tworzenie takiego obrazu, oprócz zmiany rozdzielczości obejmuje przekształcanie skali szarości i barw, co tu pominiemy, zastępując to stwierdzeniem, że jest to zwykle robione dobrze.

Do odwzorowania obrazu rastrowego służy operator **image**, który ma następujące argumenty:

w h b m p **image** —

Liczby całkowite *w* i *h* określają wysokość i szerokość obrazu (w pikselach „oryginalnych”). Liczba *b* ma wartość 1, 2, 4, 8 lub 12 i określa liczbę bitów na piksel. Macierz *m* określa wymiary i położenie obrazu na stronie utworzonej przez interpreter PostScriptu; jeśli reprezentuje ona przekształcenie tożsamościowe, to każdy piksel obrazu oryginalnego jest (w bieżącym układzie współrzędnych) kwadratem o boku o długości 1 punkt. Zadaniem procedury *p* jest dostarczanie danych (czyli wartości kolejnych pikseli); najczęściej procedura ta czyta dane z pliku, ale może również generować je na podstawie jakichś obliczeń (co między innymi umożliwia korzystanie z kompresji danych). Przykład:



Listing.

```

%!
/picstr 1 string def
/displayimage {
  /h exch def
  /w exch def
  w h 8 [ 1 0 0 -1 0 h ]
  { currentfile picstr readhexstring pop }
  image
} def
200 100 translate
40 dup scale
6 4 displayimage
00ff44ff88ff
44ffffff88ff
88ffffff44ff
ccffcc884400
showpage

```

Współczynniki macierzy m w przykładzie powodują zmianę zwrotu osi y (to jest to -1) i odpowiednie przesunięcie (o h) do góry. Dzięki temu kolejne wiersze danych reprezentują rzędy pikseli „od góry do dołu”.

Operator **currentfile** wstawia na stos obiekt reprezentujący plik bieżąco przetwarzany przez interpreter. Następnie **readhexstring** czyta z niego cyfry szesnastkowe i wpisuje odpowiednie kody (liczby całkowite od 0 do 255) do bufora, którym jest tu napis **picstr**. Napis ten zostaje na stosie (skąd konsumuje go operator **image**), ale nad nim jest jeszcze obiekt boolowski (**false** jeśli wystąpił koniec pliku), który trzeba usunąć za pomocą **pop**. Ze względu na prędkość lepszy byłby dłuższy bufor (np. o długości równej szerokości obrazka), ale nie jest to aż tak ważne.

Dla obrazów kolorowych mamy operator **colorimage**; jeden ze sposobów użycia go jest następujący:

```
w h b m p false 3} & \lstPScolorimage+ —
```

Parametry w , h , b i m mają takie samo znaczenie jak dla operatora **image**; argument p jest procedurą dostarczającą dane. Argument boolowski **false** oznacza, że jest tylko jedna taka procedura, która dostarcza wszystkie składowe koloru. Ostatni argument, 3, oznacza, że składowych tych jest 3 — czerwona, zielona i niebieska. Wartość 4 oznaczałaby składowe CMYK (ang. *Cyan*, *Magenta*, *Yellow* i *black*, czyli niebieskozielona, purpurowa, żółta i czarna).

W języku PostScript poziomu drugiego (ang. *Level 2*) operator **image** jest bardziej rozbudowany i w szczególności może służyć do odtwarzania obrazów kolorowych.

13.11. Programowanie L-systemów

Systemy Lindenmayera, albo L-systemy są pewnego rodzaju językami formalnymi, czyli zbiorami napisów możliwymi do otrzymania wskutek stosowania określonych reguł. Największe zastosowanie znalazły one w modelowaniu roślin; A. Lindenmayer był biologiem; wspólnie z informatykiem P. Prusinkiewiczem opracował wspomniane reguły właśnie w tym celu. L-systemami zajmiemy się w drugim semestrze bardziej szczegółowo; tymczasem spróbujemy wykorzystać interpreter PostScriptu do symulacji generatora i interpretera L-systemów i obejrzymy trochę obrazków.

Na początek formalności. Bezkontekstowy, deterministyczny L-system (tzw. D0L-system) jest trójką obiektów: $G = (V, \omega, P)$, gdzie

- V — alfabet (pewien ustalony, skończony zbiór symboli),
- $\omega \in V^+$ — aksjomat (pewien niepusty napis nad alfabetem V),
- $P \subset V \times V^*$ — skończony zbiór tzw. produkcji. Każdą produkcję można zapisać w postaci $p_i: a_i \rightarrow b_i$. Symbol a_i jest tu znakiem alfabetu V , a b_i oznacza pewien (być może pusty) napis. Każdemu symbolowi alfabetu w D0L-systemie odpowiada jedna produkcja, a więc zbiór produkcji i alfabet są równoliczne.

Produkcja jest regułą zastępowania symboli w przetwarzanych napisach. Interpretacja L-systemu polega na przetwarzaniu kolejnych napisów; pierwszy z nich to aksjomat; każdy następny napis powstaje z poprzedniego przez zastąpienie każdego symbolu przez ciąg symboli po prawej stronie odpowiedniej produkcji (uwaga: to jest istotna różnica między L-systemami i językami formalnymi Chomsky’ego; obecność w językach Chomsky’ego i brak w L-systemach rozróżnienia symboli tzw. terminalnych i nieterminalnych to różnica nieistotna).

Jeden z najprostszych L-systemów wygląda następująco:

$$\begin{aligned} V &= \{F, +, -\}, \\ \omega &= F--F--F--, \\ p_1: F &\rightarrow F+F--F+F, \\ p_2: + &\rightarrow +, \\ p_3: - &\rightarrow -. \end{aligned}$$

W pierwszych dwóch iteracjach otrzymamy kolejno napisy

$$\begin{aligned} &F+F--F+F--F+F--F+F--F+F--F+F-- \\ &F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F-- \dots \end{aligned}$$

Każdy taki napis możemy potraktować jak program, wykonując odpowiednią procedurę dla każdego znaku. Do otrzymania rysunku figury geometrycznej przydaje się tzw. grafika żółwia, nazwana tak, zdaje się, przez twórców skądinąd pożytecznego języka LOGO. Żółw jest obiektem, który w każdej chwili ma określone położenie (punkt na płaszczyźnie, w którym się znajduje) i orientację, czyli kierunek i zwrot drogi, w której się porusza (chyba, że przed wydaniem polecenia ruchu zmienimy tę orientację). Procedury w PostScriptcie, realizujące grafikę żółwia, można napisać w taki sposób:

Listing.

```
/TF {
  newpath
  x y moveto
  dist alpha cos mul dup x add /x exch def
  dist alpha sin mul dup y add /y exch def
  rlineto stroke
} def

/TPlus {
  /alpha alpha dalpha add def
} def

/TMinus {
  /alpha alpha dalpha sub def
} def
```

Procedura TF realizuje ruch żółwia od bieżącej pozycji, o współrzędnych x , y , na odległość $dist$, w kierunku określonym przez kąt α . Procedury TPlus i TMinus zmieniają orientację, tj. dodają lub odejmują ustalony przyrost $d\alpha$ do lub od kąta α .

Powyższe procedury zwiążemy odpowiednio z symbolami F , $+$ i $-$ w napisie otrzymanym w ostatniej iteracji; łatwo to zrobić, pisząc rekurencyjne procedury F, Plus i Minus, które realizują produkcje, a po dojściu do określonego poziomu rekurencji sterują żółwiem. Można to zrobić tak:

Listing.

```

%!
... % tu wstawiamy procedury TF, TPlus, TMinus

/F {
  /iter iter 1 add def
  iter itn eq { TF }
  {
    F Plus F Minus Minus F Plus F
  } ifelse
  /iter iter 1 sub def
} def

/Plus { TPlus } def
/Minus { TMinus } def

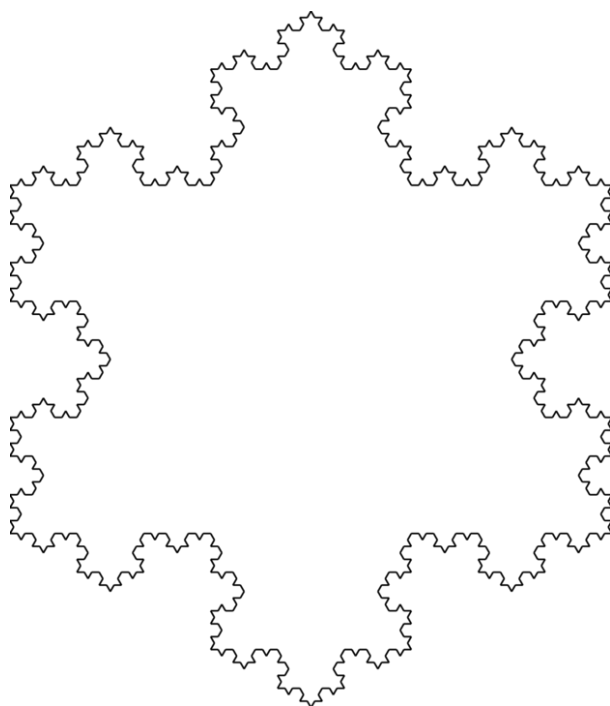
/dist 5 def
/dalpha 60 def

/iter 0 def
/itn 5 def
/x 100 def
/y 500 def
/alpha 0 def

F Minus Minus F Minus Minus F Minus Minus
showpage

```

Kolejne symbole napisu są reprezentowane przez wywołania procedur na odpowiednim poziomie rekurencji. Procedury Plus i Minus opisują produkcje, które zastępują symbol $+$ lub $-$ nim samym i dlatego mogą od razu wywołać procedury geometrycznej interpretacji tych symboli, bez rekurencji. Natomiast w procedurze F poziom rekurencji, przechowywany w zmiennej $iter$, decyduje o tym, czy generować symbole kolejnego napisu, czy też dokonać interpretacji geometrycznej — w tym przypadku ruchu żółwia, który kreśli. Łatwo w tym programie dostrzec prawą stronę produkcji dla symbolu F , a także aksjomat.



Otrzymany rysunek przedstawia przybliżenie znanej krzywej fraktalowej, który po raz pierwszy badał Helge von Koch w 1904r. Inny przykład zastosowania L-systemu do generacji figury fraktalowej mamy poniżej.

$$\begin{aligned} V &= \{F, +, -, L, R\}, \\ \omega &= L, \\ L &\rightarrow +RF - LFL - FR+, \\ R &\rightarrow -LF + RFR + FL - . \end{aligned}$$

Pominięte są tu produkcje dla symboli F , $+$, $-$, ponieważ powodują one przepisanie tych symboli bez zmiany i szkoda miejsca. Zauważmy, że tu symbol F „zostaje” we wszystkich następnych napisach i nie powoduje dokładania żadnych nowych symboli; tę rolę spełniają dwa symbole, L i R , które nie oddziałują na żółwia bezpośrednio.

Listing.

```

%!
... % tu procedury TF, TPlus, TMinus
/L {
  /iter iter 1 add def
  iter itn ne {
    Plus R F Minus L F L Minus F R Plus
  } if
  /iter iter 1 sub def
} def

/R {
  /iter iter 1 add def
  iter itn ne {
    Minus L F Plus R F R Plus F L Minus
  } if
  /iter iter 1 sub def
}

```

```

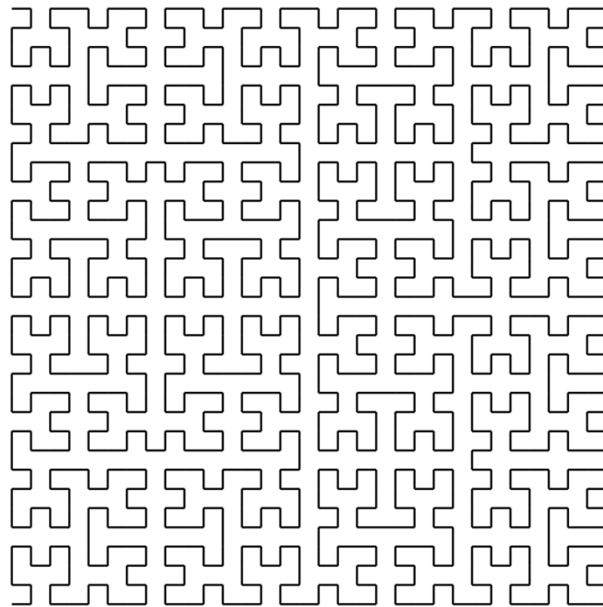
} def
/F { TF } def
/Plus { TPlus } def
/Minus { TMinus } def

/dist 10 def
/dalpha 90 def

/iter 0 def
/itn 6 def
/x 140 def
/y 200 def
/alpha 90 def

R
showpage

```



Aby narysować roślinkę, trzeba umieć wytwarzać rozgałęzienia krzywych. Przydają się do tego symbole tradycyjnie oznaczane nawiasami kwadratowymi (ponieważ symbole `[ i ]` są w PostScriptcie zarezerwowane dla innych celów, więc użyjemy nazw `TLBrack` i `TRBrack`), pierwszy z nich powoduje zapamiętanie bieżącego położenia i orientacji żółwia, a drugi — przywrócenie ich. W PostScriptcie moglibyśmy wykorzystać do tego stos argumentów, ale to by utrudniło korzystanie z niego w innym celu; dlatego lepiej zadeklarować odpowiednią tablicę i użyć jej w charakterze stosu.

Listing.

```

/TInitStack {
  /MaxTStack 120 def
  /TStack MaxTStack array def
  /TSP 0 def
} def

/TLBrack {
  TSP 3 add MaxTStack le {

```

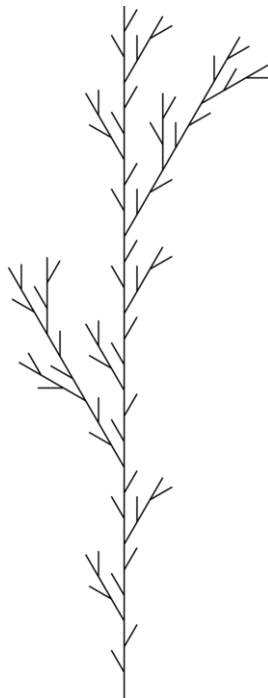
```

TStack TSP x put /TSP TSP 1 add def
TStack TSP y put /TSP TSP 1 add def
TStack TSP alpha put /TSP TSP 1 add def
} if
} def

/TRBrack {
  TSP 3 ge {
    /TSP TSP 1 sub def /alpha TStack TSP get def
    /TSP TSP 1 sub def /y TStack TSP get def
    /TSP TSP 1 sub def /x TStack TSP get def
  } if
} def

```

Użyjemy tych procedur w programie (od razu ćwiczenie: proszę odtworzyć opis L-systemu, tj. alfabet, aksjomat i produkcje, realizowanego przez ten program):



Listing.

```

%!
... % tu procedury obsługi żółwia
/F {
  /iter iter 1 add def
  iter itn eq { TF }
  {
    F LBrack Plus F RBrack F LBrack Minus F RBrack F
  } ifelse
  /iter iter 1 sub def
} def

/Plus { TPlus } def
/Minus { TMinus } def
/LBrack { TLBrack } def

```



```

/RBrack { TRBrack } def

/dist 20 def
/dalpha 30 def

/iter 0 def
/itn 4 def
/x 200 def
/y 100 def
/alpha 90 def

TInitStack
F
showpage

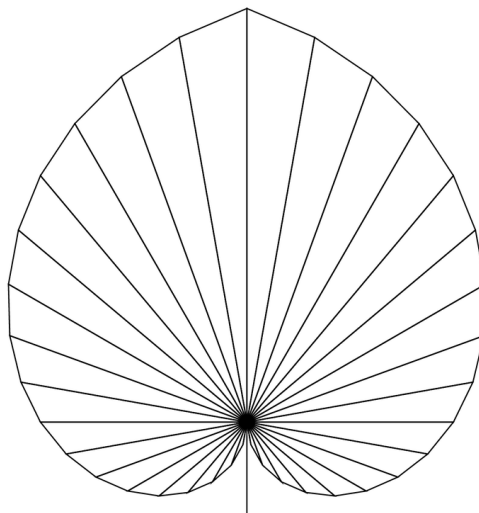
```

Symbol F może być interpretowany jako polecenie wygenerowania krawędzi wielokąta i wtedy można wprowadzić symbole $\{ i \}$, czyli klamry, które określają początek i koniec generowania wielokąta. Można też generować wierzchołki wielokąta — niech to będzie skutkiem interpretacji symbolu $.$; wierzchołek pojawi się w bieżącym punkcie położenia żółwia, który przemieszczany podczas przetwarzania symbolu f nie rysuje kresek.

Zaprogramujemy L-system

$$\begin{aligned}
 V &= \{f, +, -, [,], \{, \}, ., A, B, C\}, \\
 \omega &= FFFF[A][B], \\
 A &\rightarrow [+A\{.\}C.], \\
 B &\rightarrow [-B\{.\}C.], \\
 C &\rightarrow fC,
 \end{aligned}$$

który umożliwia wygenerowanie pokazanego na obrazku liścia.



Listing.

```

%!
... % procedury TF, TPlus, TMinus, TRBrack, TLBrack
    % jak poprzednio, a Tf proszę samemu napisać
/TLBrace { newpath /empty true def } def
/TRBrace { closepath stroke } def

```

```

/TDot {
  empty { x y moveto } { x y lineto } ifelse
  /empty false def
} def

/A {
  /iter iter 1 add def
  iter itn ne
  { LBrack Plus A LBrace Dot RBrack Dot C Dot
    RBrace } if
  /iter iter 1 sub def
} def

/B {
  /iter iter 1 add def
  iter itn ne
  { LBrack Minus B LBrace Dot RBrack Dot C Dot
    RBrace } if
  /iter iter 1 sub def
} def

/C {
  /iter iter 1 add def
  iter itn ne
  { f C } if
  /iter iter 1 sub def
} def

/F { TF } def
/f { Tf } def
/Plus { TPlus } def
/Minus { TMinus } def
/LBrack { TLBrack } def
/RBrack { TRBrack } def
/LBrace { TLBrace } def
/RBrace { TRBrace } def
/Dot { TDot } def

/dist 20 def
/dalpha 10 def
/iter 0 def
/itn 20 def
/x 300 def
/y 200 def
/alpha 90 def

TInitStack
F F F LBrack A RBrack LBrack B RBrack
showpage

```

13.12. PostScript obudowany

Zgodnie z podaną wcześniej informacją, aby system operacyjny uznał plik tekstowy za program źródłowy w PostScriptcie, pierwszymi dwoma znakami w tym pliku powinny być `%!` . Wysyłając taki plik na drukarkę, otrzymamy odpowiedni obrazek, a nie treść pliku. Taka minimalna

informacja często jednak nie wystarczy. Jeśli chcemy wygenerować obrazek, który ma być ilustracją tekstu (złożonego np. za pomocą T_EX-a), to trzeba dać dwie linie, o postaci

Listing.

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: x1 y1 x2 y2
```

Pierwsza z tych linii musi być na początku pliku. Informuje ona program, który ten plik przetwarza, że jest to tzw. PostScript obudowany, czyli program opisujący obrazek przeznaczony do umieszczenia w większej całości. Druga linia (może być zaraz po pierwszej lub na końcu pliku) zawiera informacje o prostokącie, w którym obrazek się mieści. Program T_EX po przeczytaniu tej informacji zostawi na stronie odpowiedni obszar na obrazek; cztery liczby całkowite są współrzędnymi dolnego lewego i górnego prawego narożnika, prostokąta. Jednostka długości jest równa 1/72".

Program w PostScriptcie obudowanym nie powinien zawierać instrukcji niszczących, takich jak kasowanie strony lub zdejmowanie ze stosu obiektów, których tam nie włożył. Nie należy też bezpośrednio przypisywać wartości CTM; to spowodowałoby umieszczenie obrazka w ustalonym miejscu na stronie, a nie w miejscu wyznaczonym przez program dokonujący składu, który ten obrazek wciąga na ilustrację. Najlepiej, aby program utworzył własny słownik, tylko z niego korzystał, a na końcu po sobie posprzątał.

14. OpenGL — wprowadzenie

14.1. Informacje ogólne

Dawno, dawno temu firma Silicon Graphics opracowała dla produkowanego przez siebie sprzętu bibliotekę graficzną zwaną IRIS GL. Związany z nią interfejs programisty był zależny od sprzętu i od systemu operacyjnego. Rozwój kolejnych wersji sprzętu (w tym uproszczonego, w którym procedury rysowania musiały być realizowane programowo) wymusił kolejne zmiany w kierunku niezależności sprzętowej. Uniezależnienie sposobu programowania grafiki od sprzętu i środowiska, w jakim działa program, zostało zrealizowane pod nazwą OpenGL. Umożliwiło to opracowanie implementacji współpracujących z zupełnie różnymi systemami okienkowymi, np. XWindow, Apple, OS/2 Presentation Manager i inne. Pierwsza specyfikacja standardu OpenGL, 1.0, została opublikowana w 1992r. O jakości tego projektu świadczy mała liczba rewizji (do tej pory 10); kolejne wersje, oznaczone numerami 1.1–1.5, pojawiły się w latach 1992–2003. Specyfikacje o numerach 2.0, i 2.1 ukazały się w latach 2004 i 2006. Specyfikacja 3.0 jest datowana na rok 2008, w chwili pisania tego tekstu najnowsza jest specyfikacja 4.1 (lipiec 2010).

Prawdopodobnie największa część sprzętu spotykanego obecnie jest zgodna ze specyfikacją 2.0 lub 2.1. Dlatego ten opis jest zgodny z tymi specyfikacjami. Niestety, przejście do specyfikacji o głównym numerze 3 i następnych wiąże się z uznaniem większości opisanych tu procedur za przestarzałe (jak wiadomo, lepsze jest najgorszym wrogiem dobrego, oczywiście wszystko jest usprawiedliwiane dążeniem do usprawnienia wielu rzeczy) i w przyszłości to się niestety zdezaktualizuje (należy się liczyć z tym, że opisane tu procedury zostaną ze standardu usunięte).

Opublikowanie standardu umożliwiło powstanie niezależnych implementacji. Firmy inne niż Silicon Graphics produkują sprzęt („karty graficzne”), który kosztuje niewiele i udostępnia OpenGL-a na pecetach. Producenci sprzętu dostarczają odpowiednie sterowniki pracujące w najpopularniejszych systemach operacyjnych. Oprócz tego istnieją implementacje niezależne. Jedną z nich, zwaną Mesa, której autorem jest Brian Paul, była czysto programowa (dzięki czemu jej używanie nie wymaga posiadania specjalnego sprzętu), ale mogła współpracować z odpowiednim sprzętem, jeśli taki był zainstalowany (dzięki czemu ten sprzęt się nie marnował i grafika była wyświetlana szybko). W trakcie prac nad implementacją XFree86 systemu XWindow Mesa została zintegrowana z tym systemem, dzięki czemu OpenGL jest powszechnie dostępny także na zasadach Wolnego Oprogramowania. Dobrą wiadomością jest też fakt, że wielu producentów kart graficznych do pecetów dostarcza sterowniki do swojego sprzętu dla systemu XFree86 (gorszą wiadomością jest to, że producenci nie publikują kodu źródłowego tych sterowników, co wywołuje irytację osób traktujących wolność oprogramowania jak najważniejszą filozofię życiową).

OpenGL ma m.in. następujące możliwości:

- Wykonywanie rysunków kreskowych, z cieniowaniem głębokości (ang. *depth cueing*), obcinaniem głębokości (*depth culling*) i antyaliasingiem,
- Wyświetlanie scen zbudowanych z wielokątów z uwzględnieniem widoczności i oświetlenia, nakładaniem tekstury i efektami specjalnymi, takimi jak mgła, głębia ostrości, jest też pewne wspomaganie wyznaczania cieni,
- Ułatwienia w programowaniu animacji — można skorzystać z podwójnego buforowania obrazu, a także otrzymać rozmycie obiektów w ruchu (ang. *motion blur*),

- Obsługa list obrazowych, wspomaganie wyszukiwania obiektów w scenie,
- Programowanie bezpośrednio sprzętu graficznego; obecnie porządne karty graficzne zawierają procesory o mocy kilkakrotnie większej niż główny procesor komputera; jest to osiągnięte przez zwielokrotnienie procesora, który wykonuje obliczenia masywnie równoległe (np. w trybie SIMD — ang. *single instruction, multiple data*), co jest naturalne podczas tekstuowania. Ten kierunek rozwoju standardu obecnie dominuje.
- Praca w sieci, w trybie klient/serwer.

Specyfikacja 2.0 i późniejsze określają język programowania sprzętu (zwany GLSL i podobny do języka C); napisane w nim procedury¹ są wykonywane przez procesor graficzny i służą do nietypowego przetwarzania danych geometrycznych i tekstur nakładanych na wyświetlane figury. Niniejszy opis OpenGL-a traktuje jednak o podstawach. Najważniejsze, to zacząć pisać programy; apetyt przychodzi (albo przechodzi) w miarę jedzenia.

14.1.1. Biblioteki procedur

Programista aplikacji ma możliwość użycia następujących bibliotek:

- GL — procedury „niskiego poziomu”, które mogą być realizowane przez sprzęt. Figury geometryczne przetwarzane przez te procedury, to punkty, odcinki i wielokąty wypukłe. Nazwy wszystkich procedur z tej biblioteki zaczynają się od liter gl. Plik nagłówkowy procedur z tej biblioteki można włączyć do programu pisząc `#include <GL/gl.h>`.
- GLU — procedury „wyższego poziomu”, w tym dające możliwości określania częściej spotykanych brył (sześciąt, kula, stożek, krzywe i powierzchnie Béziera i NURBS). Nazwy procedur zaczynają się od glu, a plik nagłówkowy jest włączany do programu poleceniem `#include <GL/glu.h>`.
- GLX, AGL, PGL, WGL — biblioteki współpracujące z systemem okien, specyficzne dla różnych systemów. Procedury w tych bibliotekach mają nazwy zaczynające się od glx, pgl, wgl. Aby umieścić w programie odpowiednie deklaracje dla biblioteki GLX, współpracującej z systemem XWindow, należy napisać

```
#include <X11/Xlib.h>
#include <GL/glx.h>
```

- GLUT — biblioteki całkowicie ukrywające szczegóły współpracy programu z systemem okienkowym (za cenę pewnego ograniczenia możliwości). Poszczególne wersje współpracują z procedurami z biblioteki GLX, PGL lub WGL, ale udostępniają taki sam interfejs programisty. Procedury realizujące ten interfejs mają na początku nazwy przedrostek glut, a ich nagłówki umieszczamy w programie, pisząc `#include <GL/glut.h>`. Nie musimy wtedy pisać dyrektyw `#include` dla plików `gl.h` ani `glu.h`, bo włączenie pliku `glut.h` spowoduje włączenie pozostałych dwóch. Nie powinniśmy również włączać plików `glx.h` lub pozostałych, bo używamy GLUTa aby mieć program niezależny od środowiska, w którym ma działać².

14.1.2. Reguły nazewnictwa procedur

Nazwy procedur są zwięzłe, ale raczej znaczące. Przedrostek gl, glu, glx, glut zależy od biblioteki, natomiast przyrostek określa typ argumentu lub argumentów. Wiele procedur ma kilka wariantów, które wykonują to samo zadanie, ale różnią się typem i sposobem przekazywania argumentów. Przyrostki nazw procedur i odpowiadające im typy argumentów są następujące:

¹ które już się niestety doczekały „polskiej” nazwy „szadery”!?

² Oryginalny projekt GLUT nie jest już rozwijany, niemniej będziemy korzystać z tej biblioteki. Istnieje niezależna implementacja, zwana FreeGLUT, której można używać zamiast GLUTa.

b	GLbyte	l. całkowita 8-bitowa,
ub	GLubyte, GLboolean	l. 8-bitowa bez znaku,
s	GLshort	l. całkowita 16-bitowa,
us	GLushort	l. 16-bitowa bez znaku,
i	GLint, GLsizei	l. całkowita 32-bitowa,
ui	GLuint, GLenum, GLbitfield	l. 32-bitowa bez znaku,
f	GLfloat, GLclampf	l. zmiennopozycyjna 32-bitowa,
d	GLdouble, GLclampd	l. zmiennopozycyjna 64-bitowa.

Typ GLenum jest zbiorem obiektów zdefiniowanych przez podanie listy identyfikatorów. Typy GLclampf i GLclampd są zbiorami liczb zmiennopozycyjnych z przedziału [0, 1].

Przed oznaczeniem typu bywa cyfra, która oznacza liczbę argumentów. Na samym końcu nazwy procedury może też być litera v, której obecność oznacza, że argumenty mają być umieszczone w tablicy, a parametr wywołania procedury jest wskaźnikiem (adresem) tej tablicy.

Powyższe reguły nazewnictwa możemy prześledzić na przykładzie procedur wyprowadzania punktu i koloru:

```
glVertex2i ( 1, 2 );
glVertex3f ( 1.0, 2.71, 3.14 );
glColor3f ( r, g, b );
glColor3fv ( kolor );
```

Procedura glVertex* wprowadza do systemu rysującego punkt; procedura z końcówką nazwy 2i otrzymuje dwie współrzędne, które są liczbami całkowitymi; końcówka 3f oznacza wymaganie podania trzech liczb zmiennopozycyjnych o pojedynczej precyzji (32-bitowych). Sposób przetwarzania punktów wprowadzonych za pomocą każdej z tych procedur jest zależny tylko od kontekstu wywołania, a nie od tego, która z nich była użyta. Punkty wprowadzane przez glVertex* mogą mieć 2, 3 lub 4 współrzędne, ale to są współrzędne jednorodne. Podanie mniej niż czterech współrzędnych powoduje przyjęcie domyślnej wartości ostatniej współrzędnej (wagowej) równej 1. Podanie tylko dwóch współrzędnych daje punkt o trzeciej współrzędnej równej 0.

Procedura glColor* może być wywołana w celu określenia koloru, za pomocą trzech liczb z przedziału [0, 1] (typu GLclampf). Można je podać jako osobne parametry procedury glColor3f, albo umieścić w tablicy, np.

```
GLclampf kolor[3] = { 0.5, 1.0, 0.2 };
```

i wywołać procedurę glColor3fv (tak jak w podanym wcześniej przykładzie).

14.1.3. OpenGL jako automat stanów

Istnieje pewien zestaw „wewnętrznych” zmiennych przetwarzanych przez biblioteki, które określają stan (tryb pracy), w jakim system się znajduje. Zmienne te mają określone wartości domyślne, które można zmieniać przez wywołanie procedur glEnable (...); i glDisable (...);, a także poznawać ich wartości — jest tu cała menażeria procedur o nazwach glGet..., glIsEnabled, glPush..., glPop.... W kategoriach stanu automatu można też rozpatrywać zmienne określające kolor, źródła światła, sposób rzutowania itd.

14.2. Podstawowe procedury rysowania

Rysowanie czegokolwiek zaczyna się od wywołania procedury glBegin, która ma jeden parametr typu GLenum. Parametr ten musi mieć jedną z dziesięciu opisanych niżej wartości:

GL_POINTS — kolejne punkty są traktowane indywidualnie, na przykład rzuty tych punktów są rysowane w postaci kropek.

GL_LINES — każde kolejne dwa punkty są końcami odcinków.

GL_LINE_STRIP — kolejne punkty są wierzchołkami łamanej otwartej.

GL_LINE_LOOP — kolejne punkty są wierzchołkami łamanej zamkniętej.

GL_TRIANGLES — kolejne trójki punktów są wierzchołkami trójkątów.

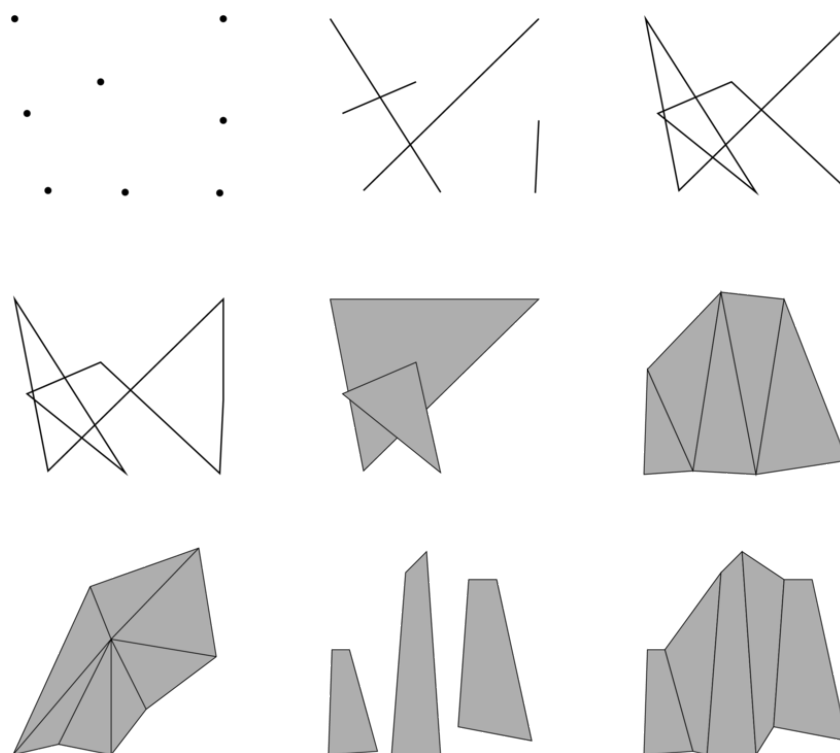
GL_TRIANGLE_STRIP — tak zwana taśma trójkątowa; po wprowadzeniu dwóch pierwszych punktów, każdy kolejny punkt powoduje wygenerowanie trójkąta, którego wierzchołkami są: ten punkt i dwa ostatnie wprowadzone wcześniej.

GL_TRIANGLE_FAN — wprowadzane punkty spowodują wygenerowanie trójkątów. Jednym z wierzchołków wszystkich tych trójkątów jest pierwszy punkt, dwa pozostałe to ostatni i przedostatni wprowadzony punkt. Liczba trójkątów, podobnie jak poprzednio, jest o 2 mniejsza od liczby punktów.

GL_QUADS — czworokąty, wyznaczone przez kolejne czwórki punktów. Uwaga: każda taka czwórka punktów musi być współpłaszczyznowa.

GL_QUAD_STRIP — taśma z czworokątami. Każda kolejna para punktów, z wyjątkiem pierwszej, powoduje wygenerowanie czworokąta, którego wierzchołkami są ostatnie cztery wprowadzone punkty.

GL_POLYGON — wielokąt wypukły. Rysując czworokąty jak i wielokąt należy zadbać o to, aby wszystkie wierzchołki leżały w jednej płaszczyźnie.



Rysunek 14.1. Interpretacja wierzchołków wprowadzanych w różnych trybach..

Koniec wprowadzania punktów w danym trybie sygnalizuje się przez wywołanie procedury `glEnd ()`. Przykład:

```
glBegin (GL_POLYGON);
glVertex2f ( x1, y1 );
...
```

```
glVertex2f ( xn, yn );
glEnd ();
```

Między wywołaniami procedur `glBegin` i `glEnd` dopuszczalne jest wywoływanie następujących procedur z biblioteki GL (gwiazdka w nazwie powinna być zastąpiona odpowiednią końcówką, zgodnie z regułami w p. 14.1.2):

<code>glVertex*(...);</code>	— punkt o podanych współrzędnych,
<code>glColor*(...);</code>	— kolor,
<code>glIndex*(...);</code>	— kolor w trybie z paletą,
<code>glNormal*(...);</code>	— wektor normalny,
<code>glTexCoord*(...);</code>	— współrzędne tekstury,
<code>glEdgeFlag*(...);</code>	— określenie, czy krawędź leży na brzegu wielokąta,
<code>glMaterial*(...);</code>	— określenie właściwości materiału,
<code>glArrayElement(...);</code>	— współrzędne punktu, kolor itd. z zarejestrowanej wcześniej tablicy,
<code>glEvalCoord(...);</code>	
<code>glEvalPoint*(...);</code>	— punkt na krzywej lub powierzchni Béziera,
<code>glCallList();</code>	
<code>glCallLists();</code>	— wyprowadzenie zawartości listy obrazowej.

Lista obrazowa jest strukturą danych, w której są przechowywane polecenia równoważne wywołaniom procedur OpenGL-a z odpowiednimi parametrami. Lista, której zawartość jest wyprowadzana między wywołaniami `glBegin` i `glEnd` musi zawierać tylko polecenia dozwolone w tym momencie.

Jeśli wyprowadzając punkt, podamy mniej niż cztery współrzędne, (np. dwie lub trzy), to domyślna wartość trzeciej współrzędnej jest równa 0, a czwartej (wagowej) 1. Możemy więc podawać współrzędne kartezjańskie lub jednorodne.

Zasada wyświetlania punktów (wierzchołków) wyprowadzanych za pomocą `glVertex*` jest taka, że właściwości obiektu w punkcie, który wprowadzamy wywołując `glVertex*`, na przykład kolor, należy określić *wcześniej*. Zanim cokolwiek wyświetlimy, należy określić odpowiednie rzutowanie, o czym będzie mowa dalej.

14.2.1. Wyświetlanie obiektów

Dla ustalenia uwagi, opis dotyczy rysowania z użyciem bufora głębokości. Aby go uaktywnić, podczas inicjalizacji wywołujemy

```
glEnable ( GL_DEPTH_TEST );
```

Przed wyświetlaniem należy ustawić odpowiednie przekształcenia określające rzut przestrzeni trójwymiarowej na ekran, co jest opisane dalej. Rysowanie powinno zacząć się od wyczyszczenia tła i inicjalizacji bufora głębokości. W tym celu należy wywołać

```
glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

Następnie wyświetlamy obiekty — to na najniższym poziomie jest wykonywane przez procedury biblioteki GL wywoływane między `glBegin` i `glEnd` (to bywa ukryte w procedurach „wyższego poziomu”, np. rysujących sześcian, sferę itp.). Na zakończenie rysowania obrazka należy wywołać `glFlush()`; albo, jeśli trzeba, `glFinish()`. Pierwsza z tych procedur powoduje rozpoczęcie wykonania wszystkich komend GL-a, które mogą czekać w kolejce. Zakończenie tej procedury nie oznacza, że obrazek w oknie jest gotowy. Druga procedura czeka na potwierdzenie zakończenia ostatniej komendy, co może trwać, zwłaszcza w sieci, ale bywa konieczne, np. wtedy, gdy chcemy odczytać obraz z ekranu (w celu zapisania go do pliku, albo utworzenia z niego tekstury).

14.3. Przekształcenia

14.3.1. Macierze przekształceń i ich stosy

OpenGL przetwarza macierze 4×4 i 4×1 . Reprezentują one przekształcenia rzutowe lub afiniczne przestrzeni trójwymiarowej i punkty, za pomocą współrzędnych jednorodnych. Macierz 4×4 jest przechowywana w tablicy jednowymiarowej o 16 elementach; tablica ta zawiera kolejne kolumny. Istnieją trzy wewnętrzne stosy, na których są przechowywane macierze przekształceń spełniających ustalone role w działaniu systemu. Operacje na macierzach dotyczą stosu wybranego przez wywołanie procedury `glMatrixMode`, z jednym argumentem, który może być równy

- `GL_MODELVIEW` — przekształcenia opisują przejście między układem, w którym są podawane współrzędne punktów i wektorów, a układem, w którym następuje rzutowanie perspektywiczne lub równoległe. W każdej implementacji OpenGL-a stos ten ma pojemność co najmniej 32.
- `GL_PROJECTION` — przekształcenie reprezentowane przez macierz na tym stosie opisuje rzut perspektywiczny lub równoległy. Ten stos ma pojemność co najmniej 2. Tyle wystarczy, bo w intencji twórców standardu ten stos jest potrzebny tylko w sytuacji awaryjnej — w razie błędu można zachować przekształcenie używane do rzutowania sceny, określić inne, odpowiednie dla potrzeb wyświetlenia komunikatu o błędach, a następnie przywrócić pierwotne przekształcenie podczas likwidacji komunikatu.
- `GL_TEXTURE` — przekształcenia na tym stosie opisują odwzorowanie tekstury. Jego pojemność nie jest mniejsza od 2.

Procedury obsługi macierzy, działające na wierzchołku wybranego stosu:

```
void glLoadIdentity ();    — przypisanie macierzy jednostkowej,
void glLoadMatrix* (m);    — przypisanie macierzy podanej jako parametr,
void glMultMatrix* (m);    — mnożenie przez macierz podaną jako parametr,
void glTranslate* (x,y,z); — mnożenie przez macierz przesunięcia,
void glRotate* (a,x,y,z);  — mnożenie przez macierz obrotu o kąt a wokół osi
                           o kierunku wektora  $[x, y, z]^T$ ,
void glScale* (x,y,z);     — mnożenie przez macierz skalowania.
```

W mnożeniu macierzy argument podany w wywołaniu procedury jest z *prawej* strony. Przekształcenia są przez to składane w kolejności odwrotnej do wykonywania obliczeń. Skutek tego jest taki sam, jak w PostScriptcie. Jeśli więc mamy procedurę, która rysuje (tj. wprowadza do systemu) jakieś obiekty określone w pewnym układzie współrzędnych, to aby umieścić je w globalnym układzie, należy wywołanie tej procedury poprzedzić wywołaniem procedur opisanych wyżej, które odpowiednio przekształcą te obiekty.

Operacje stosowe są wykonywane przez następujące procedury:

```
void glPushMatrix ();      — umieszcza na stosie dodatkową kopię macierzy, która dotychczas
                           była na wierzchołku,
void glPopMatrix ();       — usuwa element (macierz) z wierzchołka stosu.
```

14.3.2. Rzutowanie

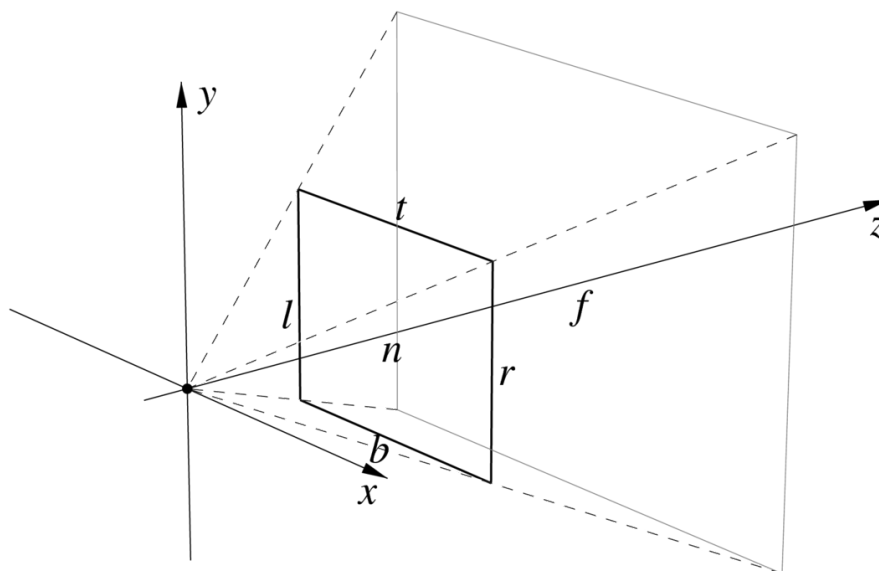
Odwzorowanie przestrzeni trójwymiarowej na ekran w OpenGL-u składa się z trzech (a właściwie czterech) kroków. Pierwszym jest przejście do układu współrzędnych (jednorodnych) obserwatora; polega ono na pomnożeniu macierzy współrzędnych jednorodnych punktu przez macierz znajdującą się na wierzchołku stosu `GL_MODELVIEW`.

Drugi krok to rzutowanie. Otrzymana w pierwszym kroku macierz współrzędnych jednorodnych jest mnożona przez macierz znajdującą się na wierzchołku stosu `GL_PROJECTION`, a następnie pierwsze trzy współczynniki iloczynu są dzielone przez czwarty (a więc jest to przejście

od współrzędnych jednorodnych do kartezjańskich). Obrazem bryły widzenia po tym kroku jest kostka jednostkowa.

Krok trzeci to odwzorowanie kostki na ekran. Do pierwszych dwóch współrzędnych, pomnożonych odpowiednio przez szerokość i wysokość klatki (w pikselach) są dodawane współrzędne piksela w *dolnym lewym* rogu klatki. Ostatni krok, za który jest odpowiedzialny system okien, to odwzorowanie klatki na ekran, zależne od położenia okna, w którym ma być wyświetlony obraz. Może się to wiązać ze zmianą zwrotu osi y , np. w systemie GLUT. Zauważmy, że klatka nie musi wypełniać całego okna.

Jeśli mamy współrzędne piksela na przykład wskazanego przez kursor, podane przez GLUTa (albo system okien, z którym program pracuje bez pośrednictwa GLUTa, np. XWindow), to należy przeliczyć współrzędną y , z układu określonego przez system okien do układu OpenGL-a. Wystarczy użyć wzoru $y' = h - y - 1$ (h jest wysokością okna w pikselach). Ten sam wzór służy również do konwersji w drugą stronę.



Rysunek 14.2. Parametry ostrosłupa widzenia w OpenGL-u.

Przykład poniżej przedstawia procedurę `reshape`, przystosowaną do współpracy z aplikacją GLUTa. Procedura ta będzie wywoływana po utworzeniu okna i po każdej zmianie jego wielkości (spowodowanej przez użytkownika, który może sobie okno rozciągać i zmniejszać myszą); jej parametrami są wymiary (wysokość i szerokość) okna w pikselach. Procedura ta umieszcza na stosie `GL_PROJECTION` macierz rzutowania perspektywicznego, skonstruowaną przez procedurę `glFrustum`. Przekształcenie to odwzorowuje bryłę widzenia na sześcian jednostkowy. Wywołanie macierzy `glViewport` określa przekształcenie odpowiedniej ściany tego sześcianu na wskazany prostokąt na ekranie.

```
void reshape ( int w, int h )
{
    glViewport ( 0, 0, w, h );
    glMatrixMode ( GL_PROJECTION );
    glLoadIdentity ( );
    glFrustum ( -1.0, 1.0, -1.0, 1.0, 1.5, 20.0 );
    glMatrixMode ( GL_MODELVIEW );
    glLoadIdentity ( );
} /*reshape*/
```

Procedura `glViewport` określa trzeci krok rzutowania, tj. przekształcenie kostki jednostkowej w okno. Jej dwa pierwsze parametry określają współrzędne lewego dolnego narożnika w pikselach, w układzie, którego początek znajduje się w lewym dolnym rogu okna. Kolejne dwa parametry to odpowiednio szerokość i wysokość okna w pikselach.

Pierwsze wywołanie procedury `glMatrixMode`, zgodnie z wcześniejszą informacją, wybiera do dalszych działań na przekształceniach stos macierzy rzutowania. Procedura `glLoadIdentity` inicjalizuje macierz na wierzchołku tego stosu; wywołana następnie procedura `glFrustum` oblicza współczynniki macierzy R przekształcenia rzutowego, które opisuje rzutowanie perspektywiczne, i zastępuje macierz na tym stosie przez iloczyn jej i macierzy R .

Parametry procedury `glFrustum` określają kształt i wielkość ostrosłupa widzenia. Znaczenie kolejnych parametrów, l , r , b , t , n , f jest na rysunku. Zwróćmy uwagę, że ostrosłup ten nie musi być symetryczny, a poza tym wymiary jego podstawy nie są skorelowane z wielkością okna, co może prowadzić do zniekształceń (nierównomiernego skalowania obrazu w pionie i poziomie). Dlatego trzeba samemu zadbać o uniknięcie takich zniekształceń; powinno być $(r - l) : (t - b) = w : h$, gdzie w , h to wartości parametrów w i h procedury `glViewport` (w przykładowej procedurze `reshape` podanej wyżej tak nie jest). Konstrukcja macierzy rzutowania jest opisana w wykładzie 5.3.

Łatwiejsza w użyciu jest procedura `gluPerspective`, która ma 4 parametry: `fovy`, `aspect`, n i f . Dwa ostatnie są takie jak n i f w `glFrustum`. Parametr `fovy` jest kątem (w stopniach; w OpenGL-u wszystkie kąty mierzy się, niestety, w stopniach) między płaszczyznami górnej i dolnej ściany ostrosłupa, który jest symetryczny. Parametr `aspect` odpowiada proporcjom wymiarów klatki na ekranie; jeśli piksele są kwadratowe (tj. o jednakowej wysokości i szerokości), to `aspect` powinien być równy w/h .

Domyślne położenie obserwatora to punkt $[0, 0, 0]^T$, patrzy on w kierunku osi z , w stronę punktu $[0, 0, -1]$ i oś y układu globalnego ma na obrazie kierunek pionowy. Jeśli chcemy umieścić obserwatora w innym punkcie, to możemy wywołać procedurę `glLookAt`. Ma ona 9 parametrów; pierwsze trzy, to współrzędne x , y , z punktu położenia obserwatora. Następne trzy to współrzędne punktu, który znajduje się przed obserwatorem i którego rzut leży na środku obrazu. Ostatnie trzy parametry to współrzędne wektora określającego kierunek „do góry”.

Procedurę `glLookAt`, która wywołuje procedury określające odpowiednie przesunięcia i obroty, wywołuje się na początku procesu ustawiania *obiektów*, który zaczyna się od wywołania `glMatrixMode ( GL_MODELVIEW );` i zaraz potem `glLoadIdentity ( );`.

Aby określić **rzutowanie równoległe**, można wywołać procedurę `glOrtho` lub `gluOrtho2D`. Pierwsza z tych procedur ma 6 parametrów, o podobnym znaczeniu jak `glFrustum`. Bryła widoczności jest prostopadłościanem, o ścianach równoległych do płaszczyzn układu, którego wierzchołkami są punkty $[l, b, n]^T$ i $[r, t, f]^T$. Procedura `gluOrtho2D` ma tylko 4 parametry — domyślnie przyjęte są wartości $n = -1$ i $f = +1$.

Gdybyśmy chcieli określić rzutowanie tak, aby współrzędne x i y punktów podawane w czasie rysowania były współrzędnymi w oknie, z punktem $[0, 0]^T$ w górnym lewym rogu i z osią y skierowaną do dołu, to procedura `reshape` powinna mieć postać

```
void reshape ( int w, int h )
{
    glViewport ( 0, 0, w, h );
    glMatrixMode ( GL_PROJECTION );
    glLoadIdentity ();
    gluOrtho2D ( 0.0, w, h, 0.0 );
    glMatrixMode ( GL_MODELVIEW );
}
```

```
glLoadIdentity ();
} /*reshape*/
```

Rzutowanie jest zwykle wykonywane przez sprzęt, ale zdarza się potrzeba obliczenia współrzędnych obrazu danego punktu w przestrzeni, albo przeciwobrazu punktu na ekranie. Umożliwiają to procedury

```
int gluProject ( x, y, z, mm, pm, vp, wx, wy, wz );
oraz
int gluUnProject ( wx, wy, wz, mm, pm, vp, x, y, z );
```

Parametry x , y i z określają współrzędne kartezjańskie punktu w przestrzeni. Parametry wx , wy i wz współrzędne „w oknie”. Dla punktu położonego między płaszczyznami obcinającymi z przodu i z tyłu jest $0 \leq wz \leq 1$. W wywołaniu `gluUnProject` parametr wz jest konieczny, aby wynik był jednoznacznie określony — pamiętamy, że to jest czynność odwrotna do rzutowania, które nie jest przekształceniem różnowartościowym.

Parametry mm i pm to odpowiednio macierz przekształcenia sceny i rzutowania. Współczynniki tych macierzy można „wyciągnąć” z systemu wywołując

```
glGetDoublev ( GL_MODELVIEW_MATRIX, mm );
glGetDoublev ( GL_PROJECTION_MATRIX, pm );
```

(mm i pm powinny tu być tablicami liczb typu **double**, o długości 16) natomiast parametr vp jest tablicą, która zawiera wymiary okna w pikselach. Można je uzyskać przez wywołanie

```
glGetIntegerv ( GL_VIEWPORT, vp );
```

z parametrem vp , który jest tablicą czterech liczb całkowitych — procedura wpisuje do niej parametry ostatniego wywołania procedury `glViewport`.

14.4. Działanie GLUTa

Zadaniem biblioteki GLUT jest ukrycie przed aplikacją wszystkich szczegółów interfejsu programowego systemu okien. W tym celu GLUT definiuje własny interfejs, zaprojektowany w duchu obiektowym. Korzystanie z GLUTa daje tę korzyść, że aplikacja może być przeniesiona do innego systemu (np. z Unixa do OS/2) bez żadnej zmiany kodu źródłowego, a ponadto GLUT jest niezwykle prosty w użyciu. Za tę przyjemność płacimy brakiem dostępu do obiektów zdefiniowanych w systemie, np. XWindow, takich jak boksy dialogowe i wihajstry (guziki, suwaki itp.). Można je utworzyć i obsługiwać wyłącznie za pomocą GLUTa i bibliotek GL i GLU, co jest bardziej pracochłonne.

14.4.1. Schemat aplikacji GLUTa

Aplikacja GLUTa składa się z kilku procedur bezpośrednio współpracujących z systemem. Podczas inicjalizacji (na ogół w procedurze `main`) należy określić pewne szczegóły korzystania z systemu i zarejestrować procedury obsługi zdarzeń, które będą następnie wywoływane przez system.

Szkielet programu — aplikacji GLUTa jest następujący:

```
... /* różne dyrektywy #include */
#include <GL/glut.h>
```

```

... /* różne procedury */
void reshape ( int w, int h ) { ... }
void display ( ) { ... }
void mouse ( int button, int state, int x, int y )
{ ... }
void motion ( int x, int y ) { ... }
void keyboard ( unsigned char key, int x, int y )
{ ... }
void idle ( ) { ... }
int main ( int argc, char **argv)
{
    glutInit ( &argc, argv );
    glutInitDisplayMode ( ... );
    glutInitWindowSize ( w, h );
    glutInitWindowPosition ( x, y );
    glutCreateWindow ( argv[0] );
    init ( ); /* tu inicjalizacja danych programu */
    glutDisplayFunc ( display );
    glutReshapeFunc ( reshape );
    glutKeyboardFunc ( keyboard );
    glutMouseFunc ( mouse );
    glutMotionFunc ( mousemove );
    glutIdleFunc ( idle );
    glutMainLoop ( );
    exit ( 0 );
} /*main*/

```

Procedury użyte w powyższym programie wykonują zadania opisane niżej, w kolejności wywoływania.

```
glutInit ( int *argc, char **argv );
```

Procedura `glutInit` dokonuje inicjalizacji biblioteki. Jako parametry są przekazywane parametry wywołania programu, wśród których mogą być opcje dla systemu okien, określające np. terminal, na którym program ma wyświetlać obrazki.

```
glutInitDisplayMode ( unsigned int mode );
```

Ta procedura określa sposób działania GL-a w tej aplikacji, w tym wykorzystywane zasoby. Parametr jest polem bitowym, np.

```
GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL | GLUT_ACCUM, albo
GLUT_DOUBLE | GLUT_INDEX
```

Poszczególne wyrazy są maskami bitowymi. `GLUT_SINGLE` oznacza używanie jednego bufora obrazów, `GLUT_DOUBLE` — dwóch, które są potrzebne do płynnej animacji. `GLUT_DEPTH` deklaruje chęć używania bufora głębokości (do rysowania z uwzględnieniem widoczności). `GLUT_STENCIL` oznacza bufor maski, do wyłączania rysowania w pewnych obszarach okna. `GLUT_ACCUM` oznacza bufor akumulacji, przydatny w antyaliasingu.

```
glutInitWindowSize i glutInitWindowPosition
```

Procedury określają początkowe wymiary i położenie okna (w pikselach, w układzie, w którym (0,0) jest górnym lewym narożnikiem ekranu). Użytkownik może je zmieniać gdy program już działa.

```
int glutCreateWindow ( char *tytuł );
```

Procedura `glutCreateWindow` tworzy okno (ale nie wyświetla go od razu). Parametr jest napisem, który system okien umieści na ramce, w przykładzie jest to nazwa programu z linii komend.

```
glutDisplayFunc ( void (*func)(void) );
```

Procedura dokonuje rejestracji w GLUCie procedury, która będzie wywoływana za każdym razem, gdy nastąpi konieczność odtworzenia (narysowania) zawartości okna. Może to nastąpić po odsłonięciu fragmentu okna (bo inne okno zostało przesunięte lub zlikwidowane), a także na wniosek aplikacji, który jest zgłaszany przez wywołanie `glutPostRedisplay()`.

```
glutReshapeFunc ( void (*func)( int w, int h ) );
```

Procedura rejestruje procedurę, która jest odpowiedzialna za przeliczenie macierzy rzutowania, stosownie do wymiarów okna, przekazywanych jako parametry. Przykłady takich procedur były podane wcześniej.

```
glutMouseFunc, glutMotionFunc i glutKeyboard
```

Powyższe procedury rejestrują procedury obsługi komunikatów o zdarzeniach spowodowanych przez użytkownika. Są to odpowiednio naciśnięcie lub zwolnienie guzika, przesunięcie myszy i naciśnięcie klawisza na klawiaturze. Obsługując taki komunikat, aplikacja może zmienić dane, a następnie wywołać `glutPostRedisplay()`; w celu spowodowania narysowania nowego obrazka w oknie. Nie powinno tu być bezpośrednich wywołań procedur rysujących.

Jest też procedura `glutIdleFunc` rejestruje procedurę, która będzie wywoływana za każdym razem, gdy komputer nie ma nic innego do roboty. Procedura taka powinna wykonywać krótkie obliczenie (które może być fragmentem długiego obliczenia) i wrócić; obliczenie będzie kontynuowane po następnym wywołaniu.

Procedury rejestrujące mogą być użyte do zmiany lub „wyłączenia” procedury obsługi komunikatu w trakcie działania programu. W tym celu należy wywołać taką procedurę, podając jako parametr nową procedurę lub wskaźnik pustej (`NULL`).

```
glutMainLoop ( void );
```

To jest procedura obsługi pętli komunikatów, która nigdy nie zwraca sterowania (czyli instrukcja `exit(0)`; w przykładzie jest wyrazem pewnej przesady). W tej procedurze następuje translacja komunikatów otrzymywanych z systemu (np. `XWindow` lub innego) na wywołania odpowiednich procedur zarejestrowanych w GLUCie. Zatrzymanie programu następuje przez wywołanie procedury `exit` w ramach obsługi komunikatu, który oznacza, że użytkownik wydał polecenie zatrzymania programu (np. przez naciśnięcie klawisza `<Esc>`).

14.4.2. Przegląd procedur GLUTa

Choć możliwości tworzenia menu udostępniane przez bibliotekę GLUT wydają się skromne, jednak mamy możliwość tworzenia wielu okien, z których każde może mieć inną zawartość, a także podokien, czyli prostokątnych części okien, w których możemy rysować cokolwiek. Dzięki temu do utworzenia wihajstrów obsługujących dialog z użytkownikiem możemy użyć wszystkich możliwości OpenGLa. Opis poniżej jest w zasadzie przewodnikiem po pliku nagłówkowym `glut.h`. Dlatego nie ma w nim zbyt dokładnego przedstawienia list parametrów.

Aby utworzyć **okno**, należy wywołać procedurę `glutCreateWindow`. Jej wartością jest liczba całkowita, która jest identyfikatorem okna w GLUCie (identyfikatory tworzone przez system

XWindow lub inny są przed aplikacją GLUTa ukryte). Początkowe wymiary i położenie okna określa się wywołując *wcześniej* procedury `glutInitWindowSize` i `glutInitWindowPosition`.

Aplikacja, która tworzy tylko jedno okno, może zignorować wartość funkcji `glutCreateWindow`. Inne aplikacje powinny ją zapamiętać. Jeśli utworzymy drugie okno, to potrzebujemy móc określić na przykład w którym oknie chcemy rysować. W danej chwili tylko jedno okno jest aktywne; możemy wywołać procedurę `glutGetWindow` aby otrzymać jego identyfikator. Okno jest aktywne natychmiast po utworzeniu i właśnie aktywnego okna dotyczą wywołania procedur `glutMouseFunc` itd., rejestrujące procedury obsługi komunikatów okna. Okno jest też aktywne w chwili wywołania jego procedury obsługi komunikatu. Jeśli chcemy spowodować odrysowanie zawartości tylko tego okna, to po prostu wywołujemy procedurę `glutPostWindowRedisplay` (wywołanie `glutPostRedisplay` powoduje odrysowanie wszystkich okien). Jeśli chcemy odrysowania innego okna, to powinniśmy wcześniej je uaktywnić, wywołując `glutSetWindow ( identyfikator_okna )`. Podobnie trzeba postąpić, aby w trakcie działania programu zmienić lub zlikwidować procedurę obsługi komunikatu (aby zlikwidować należy przekazać zamiast procedury wskaźnik `NULL`).

Podokno jest prostokątnym fragmentem okna, w którym można rysować niezależnie od tego okna i innych jego podokien. Aby utworzyć podokno, wywołujemy procedurę

```
glutCreateSubWindow ( win, x, y, w, h );
```

Wartością tej procedury jest identyfikator podokna. Identyfikatory okien i podokien tworzą wspólną przestrzeń, tj. identyfikatory wszystkich okien i podokien są różne. W ten sposób procedury `glutSetWindow` i procedury rejestracji procedur obsługi komunikatów działają tak samo na oknach jak i podoknach.

Pierwszym parametrem procedury `glutCreateSubWindow` jest identyfikator okna (lub podokna), którego to jest część. Cztery pozostałe parametry określają położenie i wymiary podokna, względem górnego lewego rogu okna.

Do zlikwidowania okna lub podokna służy procedura `glutDestroyWindow`. Aby zmienić położenie lub wymiary okna lub podokna, należy uczynić je aktywnym (przez wywołanie `glutSetWindow`), a następnie wywołać procedurę `glutPositionWindow` lub `glutReshapeWindow`. Jeśli okno jest podzielone na kilka podokien, to możemy zmieniać ich wymiary w procedurze obsługi komunikatu o zmianie wielkości okna głównego; użytkownik zmienia wielkość okna za pomocą myszy, a procedura ta oblicza wielkości i położenia podokien tak, aby dostosować je do zmniejszonego okna głównego.

Są jeszcze następujące procedury „zarządzania oknami”:

`glutSetWindowTitle` — ustawia tytuł okna na ramce utworzonej przez system (to chyba nie dotyczy podokien).

`glutSetIconTitle` — okno może być wyświetlone w postaci ikony; procedura określa podpis tej ikony na taką okoliczność.

`glutIconifyWindow` — wyświetla ikonę symbolizującą okno.

`glutHideWindow` i `glutShowWindow` — likwidują i przywracają obraz okna na ekranie.

`glutFullScreen` — po wywołaniu tej procedury aktywne okno zajmuje cały ekran (nie w każdej implementacji GLUTa to jest dostępne).

`glutPopWindow`, `glutPushWindow` — zmieniają kolejność wyświetlania okien, co ma wpływ na to, które jest widoczne, jeśli się nakładają.

Poza tym jest funkcja `glutGet`, która udostępnia różne informacje. Ma ona jeden parametr, któremu możemy nadać następujące wartości:

`GLUT_WINDOW_X`, `GLUT_WINDOW_Y` — wartością funkcji `glutGet` jest odpowiednia współrzędna górnego lewego narożnika okna w układzie okna nadrzędnego (w przypadku okna głównego — na ekranie),

`GLUT_WINDOW_WIDTH`, `GLUT_WINDOW_HEIGHT` — szerokość lub wysokość okna,

GLUT_WINDOW_PARENT — identyfikator okna, którego to jest podokno,
 GLUT_WINDOW_NUM_CHILDREN — liczba podokien,
 GLUT_WINDOW_DOUBLEBUFFER — informacja, czy jest podwójny bufor obrazu,
 GLUT_BUFFER_SIZE — liczba bitów reprezentujących kolor piksela,
 GLUT_WINDOW_RGBA — informacja, czy wartość piksela jest bezpośrednią reprezentacją koloru (jeśli 0, to jest tryb z paletą),
 GLUT_DEPTH_SIZE — liczba bitów piksela w buforze głębokości,
 GLUT_HAS_KEYBOARD, GLUT_HAS_MOUSE, GLUT_HAS_SPACEBALL itp. — informacja o obecności różnych urządzeń
 i wiele innych, o które na razie mniejsza.

14.4.3. Współpraca okien z OpenGL-em

GLUT tworzy osobny kontekst OpenGL-a dla każdego okna i podokna (ale konteksty te mają wspólne listy obrazowe, o których będzie mowa dalej). Nie jest to takie ważne, jeśli program jest napisany tak, aby wyświetlanie zawartości każdego okna było niezależne od tego, co robiliśmy z OpenGL-em poprzednio. Z moich skromnych doświadczeń wynika, że taki styl pisania programów opłaca się, nawet jeśli wiąże się to ze spadkiem sprawności programu, który za każdym razem ustawia te same parametry. Spadek ten jest zresztą niezauważalny.

Aby to urzeczywistnić, powinniśmy określać parametry rzutowania dla każdego okna w procedurze wyświetlania zawartości okna (tej rejestrowanej przez `glutDisplayFunc`). Wtedy zbędne są procedury obsługi zmiany wielkości okna (rejestrowane przez `glutReshapeFunc`), które informują OpenGL-a o wielkości okna (przez wywołanie `glViewport`) i obliczają macierz rzutowania. Zatem w programie może być tylko jedna procedura obsługi zmiany kształtu okna — ta związana z oknem głównym, bo ona ma zmienić kształt podokien.

14.4.4. Figury geometryczne dostępne w GLUCie

Rysowanie za pomocą ciągów wywołań `glVertex*` między `glBegin` i `glEnd` jest dość uciążliwe, ale to jest zwykła rzecz na niskim poziomie abstrakcji realizowanym w sprzęcie. Również procedury „wyższego poziomu” dostępne w bibliotece GLU są nie najprostsze w użyciu. Natomiast w bibliotece GLUT mamy proste w użyciu procedury

`glutSolidCube`, `glutWireCube` — rysuje sześcian, za pomocą wielokątów albo krawędzi. Parametr określa długość krawędzi. Dowolny prostopadłościan możemy zrobić poddając sześcian odpowiedniemu skalowaniu,
`glutSolidTetrahedron`, `glutWireTetrahedron` — rysuje czworościan (nie ma parametrów),
`glutSolidOctahedron`, `glutWireOctahedron` — rysuje ośmiościan,
`glutSolidDodecahedron`, `glutWireDodecahedron` — rysuje dwunastościan,
`glutSolidIcosahedron`, `glutWireIcosahedron` — rysuje dwudziestościan,
`glutSolidSphere`, `glutWireSphere` — rysuje przybliżenie sfery; kolejne parametry to promień i dwie liczby większe od 2, określające z ilu czworokątów składa się to przybliżenie (wystarczy rzędu kilku do kilkunastu),
`glutSolidTorus`, `glutWireTorus` — rysuje przybliżenie torusa; pierwsze dwa parametry to promień wewnętrzny i zewnętrzny, dwa następne określają dokładność przybliżenia (przez podanie liczby ścian),
`glutSolidCone`, `glutWireCone` — rysuje stożek o promieniu podstawy i wysokości określonych przez pierwsze dwa parametry. Dwa następne określają liczbę ścianek (a zatem dokładność) przybliżenia,
`glutSolidTeapot`, `glutWireTeapot` — rysuje czajnik z Utah, którego wielkość jest określona przez parametr.

Wszystkie powyższe procedury zawierają odpowiednie wywołania `glBegin` i `glEnd`, a także `glNormal` (tylko te ze słowem `Solid` w nazwie). Oczywiście, nie wystarczą one do narysowania np. sześcianu, którego ściany mają różne kolory (chyba, że na sześcian ten nałożymy teksturę).

14.5. Określanie wyglądu obiektów na obrazie

14.5.1. Oświetlenie

OpenGL umożliwia określenie kilku źródeł światła; mają one wpływ na wygląd rysowanych obiektów na obrazie. Oprócz oświetlenia, na wygląd obiektów wpływają własności materiału, z którego są „zrobione” obiekty, tekstura, a także ustawienia różnych parametrów OpenGL-a. Zacznijmy od opisu sposobu określania źródeł światła.

Źródła te są punktowe. Każda implementacja OpenGL-a obsługuje co najmniej 8 źródeł światła, są one identyfikowane przez stałe symboliczne `GL_LIGHT0 ... \ GL_LIGHT7`. Przykład określenia własności źródła światła:

```
glLightfv ( GL_LIGHT0, GL_AMBIENT, amb0 );
glLightfv ( GL_LIGHT0, GL_DIFFUSE, diff0 );
glLightfv ( GL_LIGHT0, GL_SPECULAR, spec0 );
glLightfv ( GL_LIGHT0, GL_POSITION, pos0 );
glLightf ( GL_LIGHT0, GL_CONSTANT_ATTENUATION, catt0 );
glLightf ( GL_LIGHT0, GL_LINEAR_ATTENUATION, latt0 );
glLightf ( GL_LIGHT0, GL_QUADRATIC_ATTENUATION, qatt0 );
glLightf ( GL_LIGHT0, GL_SPOT_CUTOFF, spco0 );
glLightfv ( GL_LIGHT0, GL_SPOT_DIRECTION, spdir0 );
glLightfv ( GL_LIGHT0, GL_SPOT_EXPONENT, spexp0 );
glEnable ( GL_LIGHT0 );
```

Pora na wyjaśnienie. Mamy tu ciąg wywołań procedur określających własności źródła światła 0, a na końcu wywołanie procedury `glEnable`, które ma na celu „włączenie” tego światła. Procedura `glLightf` określa własność źródła opisaną przez *jeden* parametr, natomiast procedura `glLightfv` otrzymuje tablicę zawierającą *cztery* liczby typu `GLfloat`. Pierwszy parametr każdej z tych procedur określa, którego źródła światła dotyczy wywołanie. Drugi parametr określa, jaką własność zmienia to wywołanie. Kolejno są to:

GL_AMBIENT — kolor światła „rozproszonego” w otoczeniu (niezależnie od położenia źródła światła). Cztery elementy tablicy `amb0` to liczby od 0 do 1, opisujące składowe czerwoną, zieloną i niebieską, oraz współczynnik alfa, który ma znaczenie tylko w pewnych trybach obliczania koloru pikseli, o których tu nie piszę. Domyślnie (czyli jeśli nie wywołamy `glLightfv` z drugim parametrem równym `GL_AMBIENT`), składowe koloru światła rozproszonego mają wartości 0.0, 0.0, 0.0, 1.0.

GL_DIFFUSE — kolor światła, które dochodząc do punktu powierzchni od źródła światła podlega odbiciu rozproszonemu (tzw. lambertowskiemu). Jeśli nie ma innych składowych światła, to obiekty pokolorowane na podstawie takiego oświetlenia są idealnie matowe. Domyślny kolor tego składnika oświetlenia dla źródła `GL_LIGHT0` ma składowe 1.0, 1.0, 1.0, 1.0, czyli jest to światło białe o maksymalnej intensywności, pozostałe źródła światła mają cztery zera.

GL_SPECULAR — kolor światła, które podlega odbiciu zwierciadlanemu (własności tego lustra są opisane dla rysowanych obiektów). W zwykłych sytuacjach składowe tego składnika światła powinny być takie same jak światła podlegającego odbiciu rozproszonemu i takie są domyślne wartości.

`GL_POSITION` — trzeci parametr procedury `glLightfv` określa współrzędne położenia źródła światła. To są współrzędne jednorodnej; jeśli ostatnia z nich jest równa 0, to źródło światła jest położone w odległości nieskończonej, w kierunku określonym przez pierwsze trzy współrzędne. W przeciwnym razie punkt położenia źródła światła znajduje się w skończonej odległości, może być nawet między obiektami w scenie. Domyślnie współrzędne położenia źródła światła są równe 0.0, 0.0, 1.0, 0.0.

`GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, `GL_QUADRATIC_ATTENUATION` — trzy parametry, k_c , k_l i k_q , określane przez wywołania `glLightf` z tymi argumentami określającymi, w jaki sposób intensywność światła maleje z odległością od niego. Współczynnik osłabienia światła jest obliczany ze wzoru

$$a = \frac{1}{k_c + k_l d + k_q d^2},$$

w którym d oznacza odległość źródła światła od oświetlanego punktu. Domyślnie jest $k_c = 1.0$, $k_l = k_q = 0.0$, co jest odpowiednie dla źródeł światła bardzo odległych od sceny. Zmienianie tych parametrów może spowodować nieco wolniejsze rysowanie, ale jak trzeba, to trzeba.

`GL_SPOT_DIRECTION`, `GL_SPOT_CUTOFF`, `GL_SPOT_EXPONENT` — parametry określane za pomocą tych argumentów opisują źródła światła o charakterze reflektora. Podany jest kierunek osi reflektora (domyślnie 0.0, 0.0, -1.0), kąt rozwarcia stożka, w jakim rozchodzi się światło (domyślnie 180°, co oznacza rozchodzenie się światła w całej przestrzeni) i wykładnik (domyślnie 0), którego większa wartość oznacza większe osłabienie światła w pobliżu brzegu stożka.

Wektory współrzędnych opisujących położenie źródeł światła lub kierunek osi reflektora są poddawane przekształceniu opisanemu przez bieżącą macierz na stosie `GL_MODELVIEW`. Rozważmy następujące możliwości:

- Aby położenie źródła światła było **ustalone względem całej sceny**, należy je określić po ustawieniu położenia obserwatora (czyli np. po wywołaniu procedury `gluLookAt`).
- Aby źródło światła było **ustalone względem obserwatora** (który snuje się po scenie ze świeczką i w szlafmycy), parametry położenia źródła światła należy określić po ustawieniu na wierzchołku stosu macierzy jednostkowej, przed wywołaniem `gluLookAt`.
- Aby **związać źródło światła z dowolnym obiektem** w scenie, trzeba położenie źródła światła określić po ustawieniu macierzy przekształcenia, która będzie ustawiona w czasie rysowania tego przedmiotu. Ponieważ źródło to ma oświetlać także wszystkie inne przedmioty, być może rysowane wcześniej niż przedmiot względem którego pozycjonujemy źródło światła (i możemy mieć wtedy inne ustawione przekształcenie), więc powoduje to konieczność obliczenia i umieszczenia na stosie przekształcenia właściwego, co niekoniecznie jest trywialne.

Jeszcze jedno: poszczególne światła włączamy i wyłączamy indywidualnie, wywołując procedury `glEnable ( GL_LIGHT0 )`; lub `glDisable ( GL_LIGHT1 )`; Aby jednak światła były w ogóle brane pod uwagę podczas rysowania, trzeba wywołać `glEnable ( GL_LIGHTING )`;

14.5.2. Własności powierzchni obiektów

Teraz zajmiemy się określaniem własności powierzchni, wpływającymi na jej kolor na obrazie, w oświetleniu określonym w sposób opisany przed chwilą. Własności te określa się za pomocą procedur `glMaterialf` i `glMaterialfv`, które mają trzy parametry.

Pierwszy z nich może przyjmować wartości `GL_FRONT`, `GL_BACK` albo też `GL_FRONT_AND_BACK` i oznacza stronę (albo strony) powierzchni, której dotyczy podana wartość parametru.

Drugi parametr określa własność materiału. Może on być równy

`GL_AMBIENT` — trzeci parametr procedury `glMaterialfv` jest tablicą zawierającą cztery liczby od 0.0 do 1.0. Przez te liczby są mnożone składowe czerwona, zielona, niebieska i alfa światła rozproszonego związanego z każdym źródłem i to jest składnikiem ostatecznego koloru piksela. Domyślnie parametry te mają wartości 0.2, 0.2, 0.2 i 1.0, co oznacza, że obiekt jest ciemnoszary (jak o zmierzchu wszystkie koty ...).

`GL_DIFFUSE` — cztery liczby opisujące zdolność powierzchni do odbijania w sposób rozproszony światła dochodzącego ze źródła światła. W obliczeniu koloru jest uwzględniane jego osłabienie związane z odległością i orientacją powierzchni (kąt między kierunkiem padania światła a wektorem normalnym powierzchni). Aby poprawnie ją uwzględnić, każde wywołanie `glVertex*` należy poprzedzić wywołaniem `glNormal*` z odpowiednim wektorem jednostkowym podanym jako parametr. Domyślnie mamy składowe 0.8, 0.8, 0.8, 1.0.

`GL_AMBIENT_AND_DIFFUSE` — można jednocześnie określić parametry odbicia rozproszonego światła rozproszonego w otoczeniu i światła dochodzącego z konkretnego kierunku.

`GL_SPECULAR` — cztery liczby opisujące sposób odbicia zwierciadlanego, domyślnie 0.0, 0.0, 0.0, 1.0. O ile kolor obiektu jest widoczny w świetle odbitym w sposób rozproszony, to kolor światła z „zajączków” jest bliski koloru światła padającego. Dlatego składowe czerwona, zielona i niebieska powinny mieć takie same wartości w tym przypadku.

`GL_SHININESS` — to jest drugi parametr procedury `glMaterialf`. Oznacza on określanie wykładnika w tzw. modelu Phong’a odbicia zwierciadlanego. Trzeci parametr jest liczbą rzeczywistą, domyślnie 0.0. Im jest większy, tym lepsze lustro, w praktyce można stosować wartości od kilku do kilkuset.

`GL_EMISSION` — cztery składowe światła emitowanego przez powierzchnię (niezależnego od jej oświetlenia), domyślnie 0.0, 0.0, 0.0, 1.0. Światło to nie ma, niestety, wpływu na wygląd innych powierzchni sceny.

Własności materiału na ogół określa się podczas rysowania, tj. bezpośrednio przed narysowaniem obiektu, albo nawet przed wyprowadzeniem każdego wierzchołka (między `glBegin ( ... );` i `glEnd ( );`). Proces ten może więc zabierać dużo czasu. Należy pamiętać, że nie trzeba za każdym razem specyfikować wszystkich własności materiału, wystarczy tylko te, które są inne od domyślnych lub ustawionych ostatnio. Inny sposób przyspieszenia tego procesu polega na użyciu procedury `glColorMaterial`. Procedura ta ma dwa parametry, identyczne jak procedura `glMaterialfv`. Po jej wywołaniu kolejne wywołania `glColor*` mają taki skutek, jak określanie parametrów materiału (czyli kolor nie jest bezpośrednio nadawany pikselom, tylko używany do określenia koloru pikseli z uwzględnieniem oświetlenia). Rysowanie w tym trybie należy poprzedzić wywołaniem `glEnable ( GL_COLOR_MATERIAL );` i zakończyć wywołaniem `glDisable ( GL_COLOR_MATERIAL );`.

14.5.3. Powierzchnie przezroczyste

Pierwsze 3 współrzędne koloru (podawane na przykład jako parametry procedury `glColor*`) opisują składowe R , G , B (tj. czerwoną, zieloną i niebieską). Czwarta współrzędna, A (alfa), opisuje „przezroczystość”. Podczas wyświetlania pikseli obliczany jest kolor (np. na podstawie oświetlenia i własności materiału), który następnie służy do wyznaczenia ostatecznego koloru przypisywanego pikselowi na podstawie *poprzedniego* koloru piksela i koloru nowego. Dzięki temu wyświetlany obiekt może wyglądać jak częściowo przezroczysty. Opisane obliczenie koloru pikseli nazywa się **mieszaniami** (ang. *blending*) i odbywa się po włączeniu go. Do włączania i wyłączania mieszania służą procedury `glEnable` i `glDisable`, wywoływane z parametrem `GL_BLEND`.

Niech R_s , G_s , B_s i A_s oznaczają nowy kolor, zaś R_d , G_d , B_d i A_d poprzedni kolor piksela. Kolor, który zostanie pikselowi przypisany, będzie miał składowe $R = s_r R_s + d_r R_d$, $G = s_g G_s + d_g G_d$, $B = s_b B_s + d_b B_d$, $A = s_a A_s + d_a A_d$, gdzie współczynniki $s_r, \dots, d_a$ są ustalane wcześniej.

Do ustalania współczynników mieszania służy procedura `glBlendFunc`, która ma 2 parametry. Pierwszy określa współczynniki $s_r, \dots, s_a$, a drugi współczynniki $d_r, \dots, d_a$. Dopuszczalne wartości tych parametrów są m.in. takie (poniższa lista nie jest pełna):

<code>GL_ZERO</code>	<code>0, 0, 0, 0</code>	
<code>GL_ONE</code>	<code>1, 1, 1, 1</code>	
<code>GL_DST_COLOR</code>	R_d, G_d, B_d, A_d	(tylko dla nowego koloru)
<code>GL_SRC_COLOR</code>	R_s, G_s, B_s, A_s	(tylko dla poprzedniego koloru)
<code>GL_SRC_ALPHA</code>	A_s, A_s, A_s, A_s	
<code>GL_DST_ALPHA</code>	A_d, A_d, A_d, A_d	

14.5.4. Mgła

Wpływ mgły na barwę rysowanych obiektów zależy od odległości obiektu od obserwatora. Aby określić ten wpływ wywołujemy procedury (przykładowe wartości parametrów mogą być punktem wyjścia do eksperymentów)

```
GLfloat fogcolor = { 0.5, 0.5, 0.5, 1.0 };

glEnable ( GL_FOG );
glFogi ( GL_FOG_MODE, GL_EXP );
glFogfv ( GL_FOG_COLOR, fogcolor );
glFogf ( GL_FOG_DENSITY, 0.35 );
glClearColor ( 0.5, 0.5, 0.5, 1.0 );
```

W tym przykładzie wpływ mgły na barwę zależy w wykładniczy (`GL_EXP`) sposób od odległości punktu od obserwatora. Warto zwrócić uwagę, że tło obrazu powinno być wypełnione kolorem mgły *przed* rysowaniem obiektów na tym tle.

14.6. Ewaluatory

14.6.1. GL — krzywe i powierzchnie Béziera

Ewaluatory w OpenGL-u to są procedury (zawarte w bibliotece GL, a zatem mogą one być zrealizowane w sprzęcie) obliczające punkt na krzywej lub powierzchni Béziera. Jak łatwo się domyślić, służą one do rysowania krzywych i powierzchni, ale nie tylko. Mogą one służyć do obliczania współrzędnych tekstury i koloru. Niestety, nie znalazłem możliwości obliczenia współrzędnych punktu i przypisania ich zmiennym w programie, a szkoda. Ewaluatory są jednowymiarowe (odpowiada to krzywym) lub dwuwymiarowe (to dotyczy powierzchni). Aby użyć ewaluatora należy go najpierw określić i uaktywnić.

Określenie ewaluatora **jednowymiarowego** polega na wywołaniu np. procedury

```
glMap1f ( GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, p );
```

Pierwszy parametr o wartości `GL_MAP1_VERTEX_3` oznacza, że punkty kontrolne krzywej mają trzy współrzędne. Inne możliwe wartości tego parametru to

`GL_MAP1_VERTEX_4` — punkty mają cztery współrzędne (jednorodne). Dzięki temu można rysować tzw. krzywe wymierne, o których na wykładzie nie mówiłem, a które są bardzo przyteczne.

`GL_MAP1_COLOR_4` — punkty mają cztery współrzędne koloru, R, G, B, A. Ten ewaluator służy do obliczania koloru, a nie punktów w przestrzeni.

`GL_MAP1_NORMAL` — ewaluator służy do obliczania wektora normalnego.

GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2,
GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, —

ewaluator służy do obliczania jednej, dwóch, trzech lub czterech współrzędnych tekstury.

Drugi i trzeci parametr określają przedział zmienności parametru — typowe wartości to 0.0 i 1.0, przyjmowane w podstawowym sposobie określenia krzywej Béziera. Kolejny parametr określa liczbę współrzędnych każdego punktu w tablicy **p**, przekazanej jako ostatni parametr. Może być tak, że w tablicy punkty mają więcej współrzędnych niż chcemy uwzględnić (bo na przykład pakujemy obok siebie współrzędne punktu w przestrzeni, a zaraz potem współrzędne koloru i tekstury, które trzeba pomijać). Kolejny parametr, w tym przykładzie 4, to **rząd** krzywej, czyli liczba punktów kontrolnych (o jeden większa niż stopień). Ostatni parametr to tablica punktów kontrolnych.

Uaktywnienie ewaluatora odbywa się przez wywołanie procedury `glEnable`, z parametrem takim, jak pierwszy parametr wywołania procedury `glMap1f`. Ten sam parametr przekazujemy procedurze `glDisable` aby wyłączyć dany ewaluator. Można określić i uaktywnić jednocześnie kilka ewaluatorów, po to, aby jednocześnie określać punkty krzywej i ich kolory. Użycie ewaluatorów polega na wywołaniu `glEvalCoord1f ( t );`, gdzie *t* jest liczbą — parametrem krzywej. Jeśli w chwili wywołania są aktywne ewaluatory `GL_MAP1_VERTEX_3` i `GL_MAP1_COLOR_4`, to takie wywołanie jest prawie równoważne wywołaniu `glColor4f ( ... );` i `glVertex3f ( ... );`, z parametrami o wartościach odpowiednich współrzędnych obliczonych przez te ewaluatory. Różnica polega na tym, że bieżący kolor nie ulega zmianie, tj. kolor obliczony przez ewaluator jest nadawany tylko obliczonemu przez ewaluator punktowi.

Jeśli chcemy narysować ciąg punktów albo łamaną, przy czym punkty te są obliczane przez ewaluator dla argumentów (parametrów krzywej), które tworzą ciąg arytmetyczny, to możemy to zrobić wywołując kolejno:

```
glMapGrid1f ( n, t0, t1 );
glEvalMesh1 ( GL_LINE, i0, i1 );
```

Parametr *n* jest liczbą kroków (odcinków całej łamanej); parametry *t₀* i *t₁* określają końce przedziału zmienności parametru krzywej, który zostanie podzielony na *n* równych części. Parametr `GL_LINE` oznacza, że rysujemy łamaną (aby narysować punkty trzeba podać `GL_POINT`). Parametry *i₀* i *i₁* określają numer pierwszego i ostatniego punktu siatki określonej przez `glMapGrid1f`, które będą obliczone i narysowane.

Ewaluatory **dwuwymiarowe** działają na takiej samej zasadzie. Zamiast znaków `MAP1` w odpowiednich stałych symbolicznych pojawiają się znaki `MAP2`. Do określania ewaluatora np. dla płata Béziera stopnia (*n*, *m*) wywołujemy procedurę

```
glMap2f ( GL_MAP2_VERTEX_3, u0, u1, us, n + 1, v0, v1, vs, m + 1, p );
```

Parametry *u₀*, *u₁*, *v₀*, *v₁* określają przedziały zmienności parametrów odpowiednio *u* i *v*. Parametry *u_s* i *v_s* określają odległości w tablicy (liczb zmiennopozycyjnych, typu `GLfloat` w tym przypadku) między współrzędnymi kolejnych punktów w wierszu i w kolumnie siatki kontrolnej, a zamiast stopnia ze względu na *u* i *v* podaje się rząd. Aby użyć ewaluatora dwuwymiarowego należy go uaktywnić i można wywołać procedurę `glEvalCoord2f ( u, v );`. Są też dostępne procedury `glMapGrid2f` i `glEvalMesh2`, które pomagają w narysowaniu powierzchni w postaci siatki odcinków lub trójkątów, dla siatki regularnej określonej w dziedzinie płata.

14.6.2. GLU — krzywe i powierzchnie B-sklejane

Rysowanie krzywych i powierzchni B-sklejanych w OpenGL-u jest zrealizowane na dwóch poziomach: poziom „niższy” to opisane wcześniej ewaluatory, zdefiniowane w bibliotece GL, na-

tomiast poziom „wyższy” jest określony w procedurach biblioteki GLU. Procedury te obliczają punkty krzywych i powierzchni za pośrednictwem ewaluatorów, po wyznaczeniu reprezentacji Béziera odpowiedniego fragmentu wielomianowego łuku lub powierzchni.

Aby użyć procedur obsługi krzywych i powierzchni sklejanych z biblioteki GLU, trzeba utworzyć obiekt dokonujący podziału krzywej lub powierzchni na kawałki wielomianowe. Robi się to tak:

```
GLUnurbsObj *nurbs_obj;
...
nurbs_obj = gluNewNurbsRenderer ();
```

Następnym krokiem jest określenie własności tego obiektu, czyli szczegółów jego działania. Służy do tego procedura `gluNurbsProperty`, która ma trzy parametry. Pierwszym z nich jest wskaźnik obiektu (w powyższym przykładzie zmienna `nurbs_obj`). Drugi parametr określa własność, którą specyfikujemy za pomocą trzeciego parametru, który jest liczbą rzeczywistą. Drugi parametr może być równy

`GLU_DISPLAY_MODE` — wtedy trzeci parametr równy `GLU_FILL` powoduje wypełnianie wielokątów, które stanowią przybliżenie powierzchni (można wtedy uaktywnić testy widoczności i „włączyć” oświetlenie). Jeśli trzeci parametr ma wartość `GLU_OUTLINE_POLYGON`, to narysowana będzie siatka odcinków przybliżających linie stałego parametru płata.

`GLU_SAMPLING_TOLERANCE` — trzeci parametr określa długość najdłuższego odcinka (na obrazie, w pikselach, domyślnie 50.0, czyli dużo), jaki może być wygenerowany w celu utworzenia obrazu.

`GLU_SAMPLING_METHOD` — wywołanie procedury z trzecim parametrem równym `GLU_PATH_LENGTH`, powoduje takie dobranie gęstości punktów, aby wielokąty przybliżające powierzchnię miały na obrazie boki nie dłuższe niż tolerancja zadana przez wywołanie procedury `gluNurbsProperty` z drugim parametrem równym `GLU_SAMPLING_TOLERANCE`.

Jeśli trzeci parametr jest równy `GLU_DOMAIN_DISTANCE`, to wywołując następnie procedurę `gluNurbsProperty` z drugim parametrem równym kolejno `GLU_U_STEP` i `GLU_V_STEP` należy podać kroki, z jakimi ma być stablicowana powierzchnia, w dziedzinie.

Obiekt przetwarzający krzywe i powierzchnie NURBS można zlikwidować wywołując `gluDeleteNurbsRenderer` (z parametrem — wskaźnikiem podanym wcześniej przez `gluNewNurbsRenderer`).

Aby narysować powierzchnię, należy ustawić oświetlenie i właściwości materiału, a następnie wywołać procedury

```
gluBeginSurface ( nurbs_obj );
gluNurbsSurface ( nurbs_obj, N + 1, u, M + 1, v, dpu, dpv, d, n + 1, m + 1,
                  GL_MAP2_VERTEX_3 );
gluEndSurface ( nurbs_obj );
```

Parametry procedury `gluNurbsSurface` to kolejno wskaźnik obiektu przetwarzającego powierzchnię, liczba i tablica węzłów w ciągu „*u*”, liczba i tablica węzłów w ciągu „*v*” (oznaczenia są takie jak w wykładzie), odległości `dpu` i `dpv` między pierwszą współrzedną punktów kontrolnych odpowiednio w wierszu i kolumnie siatki (porównaj z opisem ewaluatorów), tablica punktów kontrolnych, rząd ze względu na *u* i *v* (o 1 większy niż stopień). Ostatni parametr, określa wymiar przestrzeni (czyli liczbę współrzędnych punktów kontrolnych), w tym przykładzie 3 (rysujemy więc „zwykłą” powierzchnię B-sklejaną). Można też podać ostatni parametr równy `GL_MAP2_VERTEX_4`, który oznacza rysowanie powierzchni wymiernej (punkty kontrolne leżą wtedy w czterowymiarowej przestrzeni jednorodnej), a także `GL_MAP2_TEXTURE_COORD_*` (zamiast `*` musi być 1, 2, 3 lub 4), co oznacza, że ewaluatory wywoływane przez `gluNurbsSurface`

mają generować współrzędne w układzie tekstury, albo `GL_MAP2_NORMAL`, w celu wygenerowania wektorów normalnych powierzchni.

Rysując krzywą NURBS, mamy do dyspozycji procedury `gluBeginCurve`, `gluEndCurve` (mają one jeden parametr, wskaźnik obiektu przetwarzania krzywych, utworzonego przez wywołanie `gluNewNurbsRenderer`) i procedurę `gluNurbsCurve`, której parametrami są: wskaźnik obiektu, liczba i tablica węzłów, odstęp (w tablicy liczb rzeczywistych) między pierwszymi współrzędnymi kolejnych punktów kontrolnych, tablicę punktów kontrolnych i parametr określający typ ewaluatora jednowymiarowego, np. `GL_MAP1_VERTEX_3`.

14.7. Bufor akumulacji i jego zastosowania

Bufor akumulacji jest tablicą pikseli, dzięki której jest możliwy antyaliasing (przestrzenny i czasowy) oraz symulacja głębi ostrości. Sposób jego użycia jest następujący: wykonujemy kolejno kilka obrazów sceny, zaburzając dla każdego z nich położenie obserwatora i rzutni (dzięki czemu możemy osiągnąć antyaliasing przestrzenny i symulację głębi ostrości), oraz umieszczając poruszające się obiekty w położeniach odpowiadających różnym chwilom. Obrazy otrzymane w buforze ekranu (tym, który możemy wyświetlać na ekranie) sumujemy w buforze akumulacji. Dokładniej, wartości R , G , B , A każdego piksela obrazu mnożymy przez $\frac{1}{n}$, gdzie n jest liczbą „akumulowanych” obrazów, i dodajemy do odpowiednich składowych (o początkowej wartości 0) odpowiedniego piksela w buforze akumulacji. W ten sposób po wykonaniu n obrazów mamy w buforze akumulacji ich średnią arytmetyczną.

Jeszcze jedna możliwość zastosowania bufora głębokości wiąże się z symulacją oświetlenia sceny przez nie-punktowe źródła światła. Mając źródła światła „liniowe” (np. świetlówki) lub „powierzchniowe” (takie jak lampy z dużym kloszem) możemy wybrać na każdym takim świecącym przedmiocie kilka punktów i na kolejnych obrazach zbieranych w buforze akumulacji uwidoczniać skutek oświetlenia przez źródła światła w tych punktach. Jeśli wyznaczymy za każdym razem cienie (co nie jest łatwe, ale możliwe przez odpowiednie wykorzystanie tekstur), to otrzymamy również „miękkie cienie”, jakie powinny wystąpić w tak oświetlonej scenie.

14.7.1. Obsługa bufora akumulacji

Aby skorzystać z bufora akumulacji, należy najpierw go zarezerwować. W GLUCie robi się to, wywołując

```
glutInitDisplayMode ( GLUT_RGBA | GLUT_DEPTH | GLUT_ACCUM );
```

Przed rysowaniem pierwszego obrazka (uwaga: pierwszego z serii, która ma dać jeden obraz antyaliasowany) czyścimy bufor akumulacji wywołując

```
glClear ( GL_ACCUM_BUFFER_BIT );
```

Następnie, przed rysowaniem każdego kolejnego obrazka czyścimy ekran i z -bufor:

```
glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

a po narysowaniu go dodajemy wartości pikseli do bufora akumulacji:

```
glAccum ( op, value );
```

Pierwszy parametr powyższej procedury jest kodem operacji, w tym przypadku `GL_ACCUM`, a drugi to mnożnik, który powinien być równy $\frac{1}{n}$, jeśli chcemy zebrać dane z n obrazów. Zamiast

kasować bufor akumulacji, możemy za pierwszym razem wywołać `glAccum` z pierwszym parametrem równym `GL_LOAD`. Są też operacje `GL_ADD` i `GL_MULT`, z których pierwsza dodaje *value* do pikseli bufora akumulacji, a druga mnoży ich wartości przez *value* (mnożnik jest obcinany do przedziału $[-1, 1]$). Aby skopiować zawartość bufora akumulacji do bufora obrazu (którego zawartość możemy oglądać na ekranie), wywołujemy

```
glAccum ( GL_RETURN, value );
```

Parametr *value* powinien mieć wartość 1, ponieważ do bufora obrazu wpisywane są wartości z bufora akumulacji pomnożone przez ten parametr. W zasadzie można by, zbierając informację w buforze akumulacji, podać mnożnik (parametr *value*) $\frac{a}{n}$ dla dowolnego $a \neq 0$, a podczas przepisywania do bufora obrazu podać $value = \frac{1}{a}$, ale dla $a > 1$ może nastąpić nadmiar, a dla $a < 1$ rosną błędy zaokrągleń (pamiętajmy, że w buforze akumulacji wartości *R*, *G*, *B*, *A* są prawdopodobnie reprezentowane przez bajty). Ponieważ jednak parametr *value* za każdym razem podajemy na nowo, więc zamiast średniej arytmetycznej możemy w buforze akumulacji obliczyć średnią ważoną obrazów (suma parametrów *value* musi być równa 1).

14.7.2. Antyaliasing przestrzenny

Przykłady użycia bufora akumulacji, zaczerpnięte z książki, są podane w katalogu `book` w dystrybucji Mesy. Dla wygody zostały określone procedury `accFrustum` i `accPerspective`, które odpowiadają procedurom bibliotecznym `glFrustum` i `gluPerspective`, ale mają dodatkowe parametry, określające zaburzenia położenia obserwatora i klatki na rzutni. Przyjrzymy się tym procedurom.

```
void accFrustum ( left, right, bottom, top, near, far, pixdx, pixdy, eyedx, eyedy, focus );
```

Pierwsze 6 parametrów jest identyczne jak w `glFrustum`. Parametry `pixdx` i `pixdy` określają przesunięcie klatki na rzutni, w pikselach. Parametry `eyedx` i `eyedy` określają przesunięcie obserwatora (środką rzutowania) równoległe do rzutni. Parametr `focus` określa odległość, w której położone punkty mają ostry obraz (o tym mowa dalej, w symulacji głębi ostrości).

Procedura `accFrustum` oblicza liczby

```
dx = -(pixdx*(right-left))/viewport[2] + eyedx*near/focus;
dy = -(pixdy*(top-bottom))/viewport[3] + eyedy*near/focus;
```

(w zmiennych `viewport[2]` i `viewport[3]` są wymiary klatki w pikselach), a następnie wywołuje procedury

```
glMatrixMode ( GL_PROJECTION );
glLoadIdentity ();
glFrustum ( left+dx, right+dx, bottom+dy, top+dy, near, far );
glMatrixMode ( GL_MODELVIEW );
glLoadIdentity ();
glTranslatef ( -eyedx, -eyedy, 0.0 );
```

Procedura `accPerspective` oblicza parametry `left`, `right`, `bottom` i `top` na podstawie swoich pierwszych czterech parametrów (takich jak w procedurze `gluPerspective`) i wywołuje `accFrustum`.

Załóżmy na razie, że `eyedx = eyedy = 0`. Rzuty wszystkich punktów będą przesunięte o `pixdx` pikseli w prawo i o `pixdy` pikseli do góry. Wykonując kolejne obrazki wywołamy procedurę `accPerspective` lub `accFrustum`, podając za każdym razem inne przesunięcia. Nie powinny one wyznaczać regularnej siatki podpikseli (np. dla 4 lub 9 próbek nie powinny one leżeć w środkach kwadracików o boku $\frac{1}{2}$ lub $\frac{1}{3}$). Zamiast tego można je pozaburzać, dodając do każdego przesunięcia na takiej regularnej siatce losowy przyrost o współrzędnych mniejszych niż $\frac{1}{4}$ albo $\frac{1}{6}$.

Przykład jest w programie `book/accpersp.c`.

14.7.3. Symulacja głębi ostrości

Aby otrzymać obraz z głębią ostrości, można wykonać kilka obrazów sceny, zbierając je w buforze akumulacji i podając parametry `eyedx` i `eyedy` procedury `accFrustum` lub `accPerspective`, określające za każdym razem inne przesunięcie obserwatora. Parametr `focus` określa odległość ostrego planu (w jednostkach osi globalnego układu współrzędnych). Parametry `eyedx` i `eyedy` mogą być współrzędnymi punktów zaburzonej regularnej siatki (z dodanym jitterem, tak jak w poprzednim punkcie), trzeba tylko określić mnożnik, który reprezentuje wielkość otworu przysłony (im mniejszy tym mniejsze przesunięcia środka rzutowania, a więc większa głębia ostrości).

Przykład osiągnięcia głębi ostrości jest w programie `book/dof.c`.

14.8. Nakładanie tekstury

Liczba różnych efektów możliwych do osiągnięcia przez nałożenie tekstury na rysowane przedmioty jest trudna do oszacowania. Najprostsze zastosowanie to „pokolorowanie” przedmiotu, którego poszczególne punkty mogą mieć różne własności odbijania światła, przez co na powierzchni tworzy się pewien obraz. Tekstura w OpenGL-u jest jedno-, dwu- albo (nie w każdej implementacji) trójwymiarową tablicą pikseli. Tablica taka może przedstawiać dowolny obraz, np. fotografię, albo obraz wygenerowany przez komputer.

14.8.1. Tekstury dwuwymiarowe

Aby nałożyć na obiekt teksturę, trzeba ją najpierw utworzyć. W tym celu przygotowujemy tablicę tekseli z odpowiednią zawartością. Zaczniemy od tekstury dwuwymiarowej, którą może być obrazek przeczytany z pliku, albo utworzony w dowolny inny sposób.

Wymiary (szerokość i wysokość) tablicy tekseli muszą być równe 2^k , dla $k \geq 6$. Może też być $2^k + 2$, co oznacza, że określamy teksturę na całej płaszczyźnie — pierwszy i ostatni wiersz lub kolumna tekseli może być powielona. W specyfikacji OpenGL 2.0 dopuszczalne są też inne wymiary tekstur, natomiast w razie konieczności, jeśli tablica pikseli, którą dysponujemy, ma inne wymiary, to możemy użyć procedury

```
gluScaleImage ( format, inw, inh, intype, indata, outw, outh, outtype, outdata );
```

Parametr `format` określa zawartość tablicy, np. `GL_RGB`. Parametry `inw` i `inh` to wymiary (szerokość i wysokość) tablicy wejściowej. Parametr `intype` określa typ elementów, na przykład `GL_UNSIGNED_BYTE` (w połączeniu z formatem `GL_RGB` oznacza to, że każdy texsel jest reprezentowany przez kolejne 3 bajty, określające składowe czerwoną, zieloną i niebieską). Parametr `indata` jest wskaźnikiem tablicy z danymi wejściowymi. Parametry `outw` i `outh` określają wymiary tablicy docelowej. Parametr `outtype` może mieć też wartość `GL_UNSIGNED_BYTE`, a `outdata` jest wskaźnikiem tablicy, w której ma się znaleźć wynik. Tablicę taką o właściwej wielkości należy utworzyć przed wywołaniem tej procedury. Jej wartość 0 oznacza sukces, a 1 błąd.

Aby przygotować teksturę do nałożenia na powierzchnię, trzeba kolejno wykonać instrukcje:

```
glPixelStorei ( GL_UNPACK_ALIGNMENT, 1 );
glGenTextures ( 1, &texName );
glBindTexture ( GL_TEXTURE_2D, texName );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
```

```
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGB, w, h,
               0, GL_RGB, GL_UNSIGNED_BYTE, Image );
glTexEnvf ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL );
glEnable ( GL_TEXTURE_2D );
```

Wywołanie procedury `glPixelStorei` powiadamia OpenGL-a, że poszczególne wiersze danych nie są dopełniane nieznaczącymi bajtami, w celu np. wyrównania długości do wielokrotności 2 lub 4 (co czasem jest istotne dla programu generującego teksturę).

Procedura `glGenTextures` tworzy obiekt (lub obiekty) reprezentujący teksturę w OpenGL-u. Pierwszy parametr określa ile takich obiektów ma być utworzonych, drugi jest tablicą (o elementach typu `GLuint`), o odpowiedniej długości — procedura wstawi do niej identyfikatory utworzonych obiektów.

Procedura `glBindTexture` „uaktywnia” odpowiedni obiekt (o podanym identyfikatorze); dalsze wywołania procedur dotyczą tego obiektu. Procedurę tę wywołamy również przed rysowaniem czegoś, w celu związania konkretnej tekstury z tym czymś. **Uwaga:** w starszych wersjach OpenGL-a ponowne wywołanie `glBindTexture` powoduje błąd wykonania programu, dlatego programy przykładowe w Mesie, które nakładają teksturę, sprawdzają numer wersji. Wypadałoby naśladować te przykłady.

Procedura `glTexParameterf` ustawia różne parametry, które mają wpływ na sposób przetwarzania tekstury. Pierwsze dwa wywołania wyżej powodują, że jeśli pewien punkt ma współrzędne poza kwadratem jednostkowym (dziedziną tekstury), to otrzyma kolor taki, jak gdyby tekstura była powielona okresowo w celu pokrycia całej płaszczyzny. Zamiast `GL_REPEAT` można podać `GL_CLAMP`, i wtedy tekstura poza dziedziną będzie taka, jak w pierwszej lub ostatniej kolumnie lub wierszu tablicy tekstei.

Kolejne dwa wywołania `glTexParameterf` określają sposób filtrowania tekstury, jeśli teksele podczas odwzorowania na piksele będą zmniejszane oraz zwiększane. Wartość `GL_NEAREST` trzeciego parametru oznacza wzięcie próbki z tablicy tekstei, a `GL_LINEAR` oznacza liniową interpolację.

Wreszcie `glTexEnvf` powoduje określenie sposobu traktowania tekstury; parametr `GL_DECAL` oznacza kalkomanię; kolor pikseli jest uzyskiwany tylko przez przefiltrowanie tekstury, bez uwzględnienia własności powierzchni określanych za pomocą procedury `glMaterialf` (ale z uwzględnieniem współczynnika α , jeśli go używamy). Inne możliwe tryby to `GL_REPLACE` (przypisanie koloru oraz współczynnika α), `GL_MODULATE` (mnożenie koloru obiektu przez składowe koloru i przypisanie współcz. α tekstury) i `GL_BLEND` (obliczanie kombinacji afinicznej koloru obiektu i tekstury, ze współcz. α).

Bezpośrednio przed wyświetlaniem obiektów, na które ma być nałożona tekstura, powinniśmy wywołać procedurę `glBindTexture` (ale zobacz uwagę wyżej). Następnie *przed* wyprowadzeniem każdego wierzchołka wielokąta powinniśmy podać jego współrzędne w układzie tekstury. W przypadku tekstur dwuwymiarowych stosujemy do tego procedurę `glTexCoord2f`, której dwa parametry, s i t powinny (w zasadzie) mieć wartości z przedziału $[0, 1]$.

14.8.2. Mipmapping

Aby przyspieszyć teksturowanie obiektów, które na obrazie mogą być małe, można określić kilka reprezentacji tekstury o zmniejszonej rozdzielczości. W tym celu możemy kilkakrotnie użyć procedury `gluScaleImage`, za każdym razem zmniejszając dwa razy wymiary tablicy tekstei. Następnie wywołujemy procedury jak wyżej, ale zamiast jednego wywołania `glTexImage2D`, wywołujemy tę procedurę dla każdej reprezentacji tekstury o zmniejszonej rozdzielczości. Drugi parametr procedury określa *poziom* reprezentacji; pierwsza reprezentacja (o maksymalnej rozdzielczości) ma poziom 0, druga (2 razy mniejsza) poziom 1 itd. Należy w takim przypadku

określić *wszystkie* poziomy aż do tekstury o wymiarach 1×1 , w przeciwnym razie będą kłopoty z filtrowaniem na końcowym obrazie.

Aby uprościć konstruowanie reprezentacji tekstury o mniejszych rozdzielczościach, można posłużyć się procedurą

```
gluBuild2DMipmaps ( GL_TEXTURE_2D, GL_RGB, w, h, GL_RGB, GL_UNSIGNED_BYTE, data );
```

Procedura ta dokonuje skalowania reprezentacji i wywołuje `glTexImage2D` dla kolejno otrzymanych tablic tekselei.

14.8.3. Tekstury jednowymiarowe

Tekstury jednowymiarowe nakłada się w podobny sposób. W poprzednich punktach wszędzie, gdzie występuje fragment identyfikatora 2D, należy napisać 1D, a poza tym procedura `glTexImage1D` zamiast dwóch parametrów określających wymiary tablicy tekselei, ma tylko 1. W przypadku tekstur jednowymiarowych współrzędna w układzie tekstury nazywa się *s*, a zatem wywołujemy np. procedurę

```
glTexParameteri ( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT );
```

14.8.4. Tekstury trójwymiarowe

Przykład przygotowania tekstury:

```
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_REPEAT );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL );
glTexImage3D( GL_TEXTURE_3D, 0, GL_RGBA, tex_width, tex_height, tex_depth,
              0, GL_RGBA, GL_UNSIGNED_BYTE, voxels );
```

Jak widać odbywa się to podobnie jak w przypadku dwuwymiarowym. Dziedzina tekstury jest sześcianem, jej punkty są opisane trzema współrzędnymi, *s*, *t*, *r*. Jest jeszcze czwarta współrzędna, *q*, której domyślna wartość to 1. Czwórka współrzędnych jednorodnych (*s*, *t*, *r*, *q*) może być użyta do określenia punktu w dziedzinie tekstury; współrzędnych jednorodnych można też używać dla tekstur jedno- i dwuwymiarowych, podając cztery współrzędne (z których jedna, *r*, albo dwie, *t* i *r* są ignorowane).

14.8.5. Współrzędne tekstury

Współrzędne tekstury podawane przez wywołanie `glTexCoord*` są poddawane przekształceniu, które jest określone za pomocą macierzy przechowywanej na wierzchołku stosu przekształceń tekstury. Przypominam, że każda implementacja OpenGL-a gwarantuje minimum 2 miejsca na tym stosie i aby spowodować, że procedury `glLoadIdentity`, `glTranslate*`, `glRotate*`, `glScale`, `glLoadMatrix*` i `glMultMatrix*` działały na tym stosie, należy wywołać najpierw

```
glMatrixMode ( GL_TEXTURE );
```

Domyślnie (tj. przed wykonaniem pierwszej akcji na tym stosie) macierz przekształcenia tekstury jest jednostkowa.

Ważnym elementem określania tekstury jest możliwość **automatycznego generowania współrzędnych tekstury**. Wywoływanie procedury `glTexCoord*` przed każdym wywołaniem `glVertex*` bywa niewygodne i czasochłonne, a poza tym jest czasem niemożliwe, na przykład wtedy, gdy chcemy nałożyć teksturę na „gotowe” obiekty, takie jak czajnik (tworzony przez `glutTeapot`). W takich przypadkach możemy posłużyć się procedurami

```
glTexGen* ( coord, pname, param );
glTexGen*v ( coord, pname, *param );
```

której kolejne parametry to:

- `coord` — musi mieć wartość `GL_S`, `GL_T`, `GL_U` lub `GL_Q`, która określa jedną z czterech współrzędnych do generowania.
- `pname` — ma wartość
 - `GL_TEXTURE_GEN_MODE` — parametr `param` musi mieć jedną z wartości `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR` albo `GL_SPHERE_MAP`. W pierwszym przypadku odpowiednia współrzędna tekstury jest kombinacją liniową czterech współrzędnych (jednorodnych) wierzchołka, o współczynnikach podanych w tablicy przekazanej jako trzeci parametr procedury `glTexGen*v`, której drugi parametr jest równy `GL_OBJECT_PLANE`. Działa to tak, że wartość współrzędnej jest proporcjonalna do odległości (ze znakiem) od pewnej płaszczyzny. Jeśli `pname=GL_EYE_LINEAR`, to współrzędna tekstury powstaje przez pomnożenie wektora współrzędnych jednorodnych wierzchołka przez wektor, który jest iloczynem wektora $[p_1, p_2, p_3, p_4]$ i macierzy M^{-1} ; wektor $[p_1, p_2, p_3, p_4]$ podajemy wywołując `glTexGen*v` z drugim parametrem równym `GL_EYE_LINEAR`, a macierz M jest przechowywana na szczycie stosu `GL_MODELVIEW`. Zatem, współrzędne tekstury określa się w tym przypadku w układzie obserwatora.
 - `GL_SPHERE_MAP` służy do nakładania tekstury, która opisuje obraz otoczenia danego obiektu, odbijający się w tym obiekcie. Więcej powiem na konkretne zapotrzebowanie.
 - `GL_OBJECT_PLANE`, `GL_EYE_PLANE` — te wartości drugiego parametru procedury `glTexGen*v` określają, w którym układzie podawane są współczynniki kombinacji liniowej branej do automatycznego generowania współrzędnych tekstury.

14.9. Listy obrazowe

14.9.1. Wiadomości ogólne

Lista obrazowa (ang. *display list*) jest strukturą danych, w której są przechowywane ciągi komend OpenGL-a, równoważne skutkom wywołań procedur biblioteki GL (z pewnymi wyjątkami, które nie mogą być umieszczone w liście). Listy obrazowe są przechowywane w pamięci akceleratora graficznego, w związku z czym wykonanie tych komend może być znacznie szybsze niż wykonanie równoważnych procedur. Oszczędność czasu bierze się z wyeliminowania komunikacji między procesorem wykonującym program i akceleratorem oraz innych obliczeń (np. obliczeń wartości parametrów).

Listy obrazowe (jedną lub więcej na raz) tworzymy wywołując procedurę `glGenLists`, której parametr określa liczbę tworzonych list. Wartością procedury jest liczba całkowita, która jest identyfikatorem pierwszej utworzonej listy — pozostałe listy utworzone w tym wywołaniu procedury mają kolejne identyfikatory. Wartość 0 procedury oznacza, że nie było możliwe utworzenie żądanej liczby list.

Utworzone listy są początkowo puste. Aby umieścić w liście zawartość wywołujemy procedurę `glNewList`. Ma ona dwa parametry, z których pierwszy jest identyfikatorem listy obrazowej, a drugi określa tryb jej pracy. Jeśli parametr ten ma wartość `GL_COMPILE`, to komendy odpowiadające wywołaniom procedur OpenGL-a będą tylko umieszczane w liście. Jeśli ma on wartość `GL_COMPILE_AND_EXECUTE`, to procedury OpenGL-a są wykonywane i jednocześnie zapamiętywane w liście.

Po wywołaniu `glNewList` umieszczamy w liście zawartość, wywołując odpowiednie procedury OpenGL-a. Zamknięcie listy sygnalizujemy wywołując procedurę `glEndList` (bez parametrów). Aby wykonać komendy zawarte w liście wywołujemy procedurę `glCallList` z parametrem, który jest jej identyfikatorem.

Aby zlikwidować listy należy wywołać procedurę `glDeleteLists`, której dwa parametry określają pierwszy identyfikator i liczbę list (o kolejnych identyfikatorach), które mają zostać usunięte.

Należy pamiętać, że lista obrazowa jest po zamknięciu „czarną skrzynką”, tj. jej zawartość nie może być zmieniona. Jeśli obiekty wyświetlane przez program uległy zmianie, to listy z komendami wyświetlającymi należy zlikwidować i utworzyć je na nowo. Sposób wykorzystania list może być taki: możemy utworzyć listę obrazową wyświetlającą obiekty, a następnie wyświetlać je wielokrotnie, np. przy różnie określonym rzutowaniu (w sytuacji, gdy oglądamy obiekty z różnych stron; wtedy oczywiście lista zawiera tylko komendy wyświetlania obiektów, ale nie komendy określające rzutowanie). Inny sposób wykorzystania list jest taki: mamy scenę złożoną z kilku obiektów, które mogą zmieniać wzajemne położenie. Wtedy każdy z tych obiektów będzie miał swoją listę obrazową. Możemy też utworzyć listę odpowiadającą całej scenie. Będzie ona zawierając komendy wyświetlenia tych list, przedzielone komendami określającymi przekształcenia mające na celu ustalenie położenia obiektów opisanych w poszczególnych listach. Zmiana położenia obiektów wymaga zlikwidowania i ponownego utworzenia tylko tej jednej listy.

14.9.2. Rysowanie tekstu

Aby umieścić na obrazie tekst przy użyciu OpenGL-a, należy zrobić dwie rzeczy: określić sposób tworzenia obrazów liter i innych znaków, a następnie, mając dany napis (czyli np. ciąg kodów ASCII kolejnych znaków), spowodować wyświetlenie odpowiednich znaków. OpenGL umożliwia zarówno tworzenie obrazów liter poprzez wyświetlanie gotowych obrazków rastrowych, jak i wyświetlanie figur geometrycznych (z odpowiednim rzutowaniem, oświetleniem, a nawet tekstowaniem), które tworzą litery. W obu przypadkach drugi etap (czyli spowodowanie narysowania tekstu) może być taki sam.

Znaki pirsarskie możemy przygotować w ten sposób, że dla każdego znaku tworzymy listę obrazową, która go wyświetla. Lista taka zawiera polecenie umieszczenia na ekranie odpowiedniego obrazu rastrowego, albo narysowania np. bryły, która oglądana z odpowiedniej strony wygląda jak litera. Aby utworzyć zestaw znaków (font) rastrowy, składający się ze spacji i z 26 liter alfabetu angielskiego, na początku działania programu (podczas inicjalizacji) wykonujemy instrukcje

```
GLubyte space[13] = {};
GLubyte letters[][13] = {...};

glPixelStorei ( GL_UNPACK_ALIGNMENT, 1 );
fontofs = glGenLists ( 'Z' );
glNewList ( fontofs+' ', GL_COMPILE )
glBitmap ( 8, 13, 0.0, 2.0, 10.0, 0.0, sp );
glEndList ();
for ( i = 0, j = 'A'; j <= 'Z'; i++, j++ ) {
    glNewList ( fontofs+j, GL_COMPILE );
    glBitmap ( 8, 13, 0.0, 2.0, 10.0, 0.0,
```

```

        letters[i] );
    glEndList ();
}

```

W tym przykładzie litery mają wymiary 13×8 pikseli. Każda z nich jest reprezentowana przez 13 bajtów w tablicy `letters` (jest 1 bit na piksel, co jest określone przez wywołanie procedury `glPixelStorei`). Spacja jest opisana przez 13 bajtów zerowych, w tablicy `space`. Procedura `glBitmap` wyświetla odpowiedni obrazek rastrowy, ale w tym przykładzie jej wywołanie zostaje tylko odnotowane w odpowiedniej liście obrazowej. Dzięki utworzeniu bloku list numerowanych od zera, numer listy zawierającej komendę rysowania każdego znaku jest równy sumie indeksu pierwszej listy i odpowiedniego kodu ASCII.

Parametry procedury `glBitmap` to szerokość, wysokość, dwie współrzędne punktu referencyjnego obrazka (względem dolnego lewego rogu), współrzędne określające przesunięcie następnego obrazka i bajty określające obrazek. Zatem kolejne znaki w przykładzie będą zajmowały szerokość 10 pikseli.

Aby wyświetlić napis składający się z n znaków przechowywanych w tablicy `s` ustawiamy miejsce, od którego ma się zaczynać napis, wywołując procedurę `glRasterPos*` (np. `glRasterPos2i` z dwiema współrzędnymi całkowitymi lub `glRasterPos4fv` z parametrem, który jest tablicą czterech liczb). Punkt podany przez wywołanie tej procedury jest rzutowany zgodnie z ogólnymi zasadami. Następnie wywołujemy procedury

```

glPushAttrib ( GL_LIST_BIT );
glListBase ( fontofs );
glCallLists ( n, GL_UNSIGNED_BYTE, (GLubyte*)s );
glPopAttrib ();

```

Procedury `glPushAttrib` i `glPopAttrib` w tym przykładzie zapamiętują i przywracają zmienne stanu związane z listami obrazowymi. Procedura `glCallLists` wyświetla zawartość kolejnych list obrazowych, których indeksy bierze z napisu. Wywołanie procedury `glListBase` powoduje, że do każdego indeksu będzie dodana liczba `fontofs`, czyli numer pierwszej listy. Warto zwrócić uwagę, że pokazany tu mechanizm umożliwia łatwe korzystanie z wielu różnych fontów, których znaki zajmują rozłączne bloki list obrazowych.

14.10. Implementacja algorytmu wyznaczania cieni

W tym punkcie jest podany zestaw procedur realizujący wyznaczanie cieni³.

```

extern int useSpecular;

int CheckExtensions( void );
void InitSpotlight ( void );
void MoveLight ( float dr );
void RotateLight ( float dhoriz, float dvert );
void ChangeLightColor ( float dr, float dg, float db );
void SetLightAttenuation ( float con, float lin, float sq );
void ChangeSpotCutOff ( float dangle );
void SetLightSpecular ( int val );
void SetupLightToDisplay ( void (*renderScene)(void) );
void DisableLightToDisplay ( void );

void RenderSimpleFloor ( float height, float size );

```

³ Autorem podanych tu procedur jest Tomasz Świerczek

```
void RenderLightPosition( void );
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <GL/glut.h>

#include "lighting.h"

#define FLOOR_TILES 100
#define SHADOW_MAP_SIZE 512

float R; /*jak daleko od środka układu współrzędnych znajduje się nasza latarka*/
GLfloat RGBA[4]; /*składowe koloru światła*/
float Con,Lin,Sq; /*stałe zanikania od odległości w modelu OpenGL; */
/*zanikanie = 1.0 / (Con + odległość * Lin + odległość * odległość * Sq)*/
float SpotCutoff; /*1/2 kąta rozwarcia stożka światła*/
float SpotFrustumRight; /*parametr zależny od SpotCutoff, podawany przy glFrustum()*/
float HorizAngle, VertAngle; /*kąty pod jakimi patrzy nasze światło*/
int useSpecular; /*flaga do włączania oświetlenia odbitego*/

unsigned int ShadowMap = 0;

int CheckExtensions ( void )
{
    const unsigned char *ext;

    ext = glGetString ( GL_EXTENSIONS );
    if ( strstr ( (char*)ext, "ARB_depth_texture" ) != NULL &&
        strstr ( (char*)ext, "ARB_shadow" ) != NULL )
        return 1;
    else
        return 0;
} /*CheckExtensions*/

void InitSpotlight ( void )
{
    R = 10;
    RGBA[0] = RGBA[1] = RGBA[2] = RGBA[3] = 1.0;
    Con = 0.01;
    Lin = 0.02;
    Sq = 0.03;
    SpotCutoff = 45.0;
    SpotFrustumRight = 1.0;
    HorizAngle = 45.0;
    VertAngle = -45.0;
    useSpecular = 1;
} /*InitSpotlight*/

void MoveLight ( float dr )
{
    R += dr;
    R = R > 0.1 ? R : 0.1;
```

```

} /*MoveLight*/

void RotateLight ( float dhoriz, float dvert )
{
    HorizAngle += dhoriz;
    VertAngle += dvert;
} /*RotateLight*/

void ChangeLightColor ( float dr, float dg, float db )
{
    RGBA[0] += dr;
    RGBA[1] += dg;
    RGBA[2] += db;
} /*ChangeLightColor*/

void SetLightAttenuation ( float con, float lin, float sq )
{
    Con = con;
    Lin = lin;
    Sq = sq;
} /*SetLightAttenuation*/

void ChangeSpotCutOff ( float dangle )
{
    SpotCutoff += dangle;
    SpotCutoff = SpotCutoff >= 0.0 ? SpotCutoff : 0.0;
    SpotCutoff = SpotCutoff <= 89.99 ? SpotCutoff : 89.99;
    SpotFrustumRight = tan ( SpotCutoff / 180.0 * M_PI );
} /*ChangeSpotCutOff*/

void SetLightSpecular ( int val )
{
    useSpecular = val;
} /*SetLightSpecular*/

void SetupLightToDisplay ( void (*renderScene)(void) )
{
    GLfloat temp[] = {0.0,0.0,0.0,1.0};
    GLfloat lightViewMatrix[16];
    GLfloat x[] = { 1.0f, 0.0f, 0.0f, 0.0f };
    GLfloat y[] = { 0.0f, 1.0f, 0.0f, 0.0f };
    GLfloat z[] = { 0.0f, 0.0f, 1.0f, 0.0f };
    GLfloat w[] = { 0.0f, 0.0f, 0.0f, 1.0f };

    int i;
    GLint lights;

    /*jeśli trzeba, tworzymy mapę cienia*/
    if ( ShadowMap == 0 ) {
        glGenTextures ( 1, &ShadowMap );
        glBindTexture ( GL_TEXTURE_2D, ShadowMap );
        glTexImage2D ( GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
                      SHADOW_MAP_SIZE, SHADOW_MAP_SIZE, 0,
                      GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE, NULL );
        glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
        glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
    }

```



```

    glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
    glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
}

    /*generujemy zawartość mapy cienia*/
    glPushAttrib ( GL_VIEWPORT_BIT );
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glMatrixMode ( GL_PROJECTION );
    glPushMatrix ();
    glMatrixMode ( GL_MODELVIEW );
    glPushMatrix ();

    glMatrixMode ( GL_PROJECTION );
    glLoadIdentity ();
    glFrustum ( -SpotFrustumRight, SpotFrustumRight,
               -SpotFrustumRight, SpotFrustumRight, 1.0, 60.0 );
    glMatrixMode ( GL_MODELVIEW );
    glLoadIdentity ();

    glTranslatef ( 0.0, 0.0, -R );
    glRotatef ( -VertAngle, 1.0, 0.0, 0.0 );
    glRotatef ( -HorizAngle, 0.0, 1.0, 0.0 );

    /*teraz zapamiętamy macierze przejścia do układu światła*/
    glGetFloatv ( GL_MODELVIEW_MATRIX, lightViewMatrix );
    glViewport ( 0, 0, SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );

    glDisable ( GL_LIGHTING );
    glColorMask ( 0, 0, 0, 0 );
    glPolygonOffset ( 3.0f, 5.0f );
    glEnable ( GL_POLYGON_OFFSET_FILL );

    renderScene ();
    glDisable ( GL_POLYGON_OFFSET_FILL );
    glColorMask ( 1, 1, 1, 1 );

    glMatrixMode ( GL_MODELVIEW );
    glPopMatrix ();
    glMatrixMode ( GL_PROJECTION );
    glPopMatrix ();
    glMatrixMode ( GL_MODELVIEW );

    /*kopiujemy zawartość z-bufora do tekstury*/
    glBindTexture ( GL_TEXTURE_2D, ShadowMap );
    glCopyTexSubImage2D ( GL_TEXTURE_2D, 0, 0, 0, 0, 0,
                        SHADOW_MAP_SIZE, SHADOW_MAP_SIZE );

    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glPopAttrib ();

    /*wracamy do ustawienia naszego światła latarkowego*/
    glEnable ( GL_LIGHTING );

    /* nie chcemy żadnego oświetlenia poza naszą latarką */
    glLightModelfv ( GL_LIGHT_MODEL_AMBIENT, temp );

```

```

/*używanie kolorów z glColor() */
glEnable ( GL_COLOR_MATERIAL );
glColorMaterial ( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE );

glGetIntegerv ( GL_MAX_LIGHTS, &lights );

for( i = 1; i < lights; ++i )
    glDisable ( GL_LIGHT0+i );
glEnable ( GL_LIGHT0 );

temp[0] = temp[1] = temp[2] = 0.0;
temp[3] = 1.0;
glLightfv ( GL_LIGHT0, GL_AMBIENT, temp );
glLightfv ( GL_LIGHT0, GL_DIFFUSE, RGBA );
if ( useSpecular )
    glLightfv ( GL_LIGHT0, GL_SPECULAR, RGBA );
else
    glLightfv ( GL_LIGHT0, GL_SPECULAR, temp );

/*ustawienie macierzy takie, aby uwzględnić położenie kątowne światła...*/
glMatrixMode ( GL_MODELVIEW );
glPushMatrix ();
glRotatef ( HorizAngle, 0.0, 1.0, 0.0 );
glRotatef ( VertAngle, 1.0, 0.0, 0.0 );
glTranslatef ( 0.0, 0.0, R );

/* pozycja światła — w obecnym układzie jest ono w centrum */
temp[0] = 0.0;
temp[1] = 0.0;
temp[2] = 0.0;
temp[3] = 1.0;
glLightfv ( GL_LIGHT0, GL_POSITION, temp );

/* kierunek świecenia latarki */
temp[2] = -R;
glLightfv ( GL_LIGHT0, GL_SPOT_DIRECTION, temp );

/* kąt stożka (dokładniej jego 1/2) */
glLightf ( GL_LIGHT0, GL_SPOT_CUTOFF, SpotCutoff );

/* parametr modyfikujący skupienie światła w środku stożka */
/* (im więcej tym bardziej skupione światło) */
glLightf ( GL_LIGHT0, GL_SPOT_EXPONENT, 1.0 );

/* zanikanie */
glLightf ( GL_LIGHT0, GL_CONSTANT_ATTENUATION, Con );
glLightf ( GL_LIGHT0, GL_LINEAR_ATTENUATION, Lin );
glLightf ( GL_LIGHT0, GL_QUADRATIC_ATTENUATION, Sq );

/*przywrócenie poprzedniej macierzy widoku*/
glPopMatrix ();

glTexGenfv ( GL_S, GL_EYE_PLANE, x );
glTexGenfv ( GL_T, GL_EYE_PLANE, y );
glTexGenfv ( GL_R, GL_EYE_PLANE, z );

```

```

glTexGenfv ( GL_Q, GL_EYE_PLANE, w );

glTexGeni ( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
glTexGeni ( GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
glTexGeni ( GL_R, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
glTexGeni ( GL_Q, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );

glEnable ( GL_TEXTURE_GEN_S );
glEnable ( GL_TEXTURE_GEN_T );
glEnable ( GL_TEXTURE_GEN_R );
glEnable ( GL_TEXTURE_GEN_Q );

glMatrixMode ( GL_TEXTURE );
glLoadIdentity ();
glTranslatef ( 0.5f, 0.5f, 0.5f );
glScalef ( 0.5f, 0.5f, 0.5f );
glFrustum ( -SpotFrustumRight, SpotFrustumRight,
            -SpotFrustumRight, SpotFrustumRight, 1.0, 60.0 );
glMultMatrixf( lightViewMatrix );

glBindTexture ( GL_TEXTURE_2D, ShadowMap );
glEnable ( GL_TEXTURE_2D );

glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB,
                  GL_COMPARE_R_TO_TEXTURE );
glTexParameteri ( GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_LEQUAL );
glTexParameteri ( GL_TEXTURE_2D, GL_DEPTH_TEXTURE_MODE_ARB, GL_INTENSITY );

glEnable ( GL_BLEND );
glBlendFunc ( GL_SRC_ALPHA, GL_ZERO );

glMatrixMode ( GL_MODELVIEW );
} /*SetupLightToDisplay*/

void DisableLightToDisplay ( void )
{
    glDisable ( GL_BLEND );
    glDisable ( GL_TEXTURE_GEN_S );
    glDisable ( GL_TEXTURE_GEN_T );
    glDisable ( GL_TEXTURE_GEN_R );
    glDisable ( GL_TEXTURE_GEN_Q );
    glDisable ( GL_LIGHTING );
    glDisable ( GL_LIGHT0 );
} /*DisableLightToDisplay*/

void RenderSimpleFloor ( float height, float size )
{
    int i,j;
    float tileSize;
    float x;
    GLfloat specular[] = {1.0,1.0,1.0,1.0};

    glMaterialfv ( GL_FRONT_AND_BACK, GL_SPECULAR, specular );
    glMaterialf ( GL_FRONT_AND_BACK, GL_SHININESS, 100.0 );

    tileSize = size / (float)FLOOR_TILES;

```

```

x = -0.5*size - 0.5*TileSize;

glColor4f ( 1.0, 1.0, 1.0, 1.0 );
glNormal3f (0.0, 1.0, 0.0 );

glBegin ( GL_QUADS );
  for ( i = 0; i < FLOOR_TILES; ++i ) {
    float z = -0.5 * size - 0.5 * TileSize;
    for ( j = 0; j < FLOOR_TILES; ++j ) {
      glVertex3f ( x , height, z );
      glVertex3f ( x , height, z + TileSize );
      glVertex3f ( x + TileSize, height, z + TileSize );
      glVertex3f ( x + TileSize, height, z );
      z += TileSize;
    }
    x += TileSize;
  }
glEnd();
} /*RenderSimpleFloor*/

void RenderLightPosition ( void )
{
  glDisable(GL_LIGHTING);
  glColor3f(0.0,1.0,0.0);

  glMatrixMode ( GL_MODELVIEW );
  glPushMatrix ();
  glRotatef ( HorizAngle, 0.0, 1.0, 0.0 );
  glRotatef ( VertAngle, 1.0, 0.0, 0.0 );
  glBegin ( GL_LINES );
    glVertex3f ( -0.1, 0.0, R );
    glVertex3f ( 0.1, 0.0, R );
    glVertex3f ( 0.0, -0.1, R );
    glVertex3f ( 0.0, 0.1, R );
    glVertex3f ( 0.0, 0.0, R+0.1 );
    glVertex3f ( 0.0, 0.0, R-0.1 );
    glVertex3f ( 0.0, 0.0, R );
    glVertex3f ( 0.0, 0.0, R-0.6 );
    glVertex3f ( 0.0, 0.0, R-0.6 );
    glVertex3f ( 0.3, 0.0, R-0.3 );
    glVertex3f ( 0.0, 0.0, R-0.6 );
    glVertex3f ( -0.3, 0.0, R-0.3 );
  glEnd ();
  glPopMatrix ();
} /*RenderLightPosition*/

```

Literatura