Triki najlepszych programistów gier 3D. Vademecum profesjonalisty

Autor: Andre LaMothe
Tłumaczenie: Adam Bochenek (wstęp, rozdz. 1 –3), Jarosław Dobrzański (rozdz. 6 – 9), Sławomir Dzieniszewski (rozdz. 14 – 16, dod. A – F)
ISBN: 83-7361-267-X
Tytuł oryginału: Tricks of the 3D Game Programming Gurus
Format: B5, stron: 1312

Swoje triki zdradzą Ci prawdziwi guru ze świata twórców gier

Tworzenie gier 3D wymaga opanowania nie tylko języka programowania, ale także wielu innych dziedzin wiedzy – analizy matematycznej, rachunku macierzowego i wektorowego oraz geometrii. Oczywiście każda z tych nauk została już opisana w dziesiątkach tomów. Książka, którą trzymasz w ręce, zawiera wszystkie informacje niezbędne do stworzenia gry 3D działającej w środowisku Windows, zebrane w jednym tomie. Korzystając z niej, nie będziesz już musiał przerzucać tysięcy stron w poszukiwaniu potrzebnego Ci wzoru.

Autor książki, wykorzystując ponad dwudziestopięcioletnie doświadczenie w programowaniu, przedstawi Ci:

- algorytmy matematyczne wykorzystywane w grafice 3D,
- zasady programowania w Windows i DirectX,
- algorytmy wyświetlania grafiki 2D i 3D,
- techniki animacji i renderingu 3D,
- mapowanie tekstur,
- techniki symulacji oświetlenia i wiele innych informacji.

Korzystając z tej książki, opracujesz doskonały, oparty wyłącznie na warstwie oprogramowania silnik 3D dla gry, wykorzystasz w swoich grach modele i postaci stworzone w programach 3D, stworzysz realistyczną scenerię gry 3D i zoptymalizujesz kod źródłowy programu pod kątem szybkości jego wykonywania.

O autorze:
Andre LaMothe to autor uznanych przez rynek książek o programowaniu gier i grafiki trójwymiarowej. Ciąg jego sukcesów wydawniczych zapoczątkowało pierwsze wydanie tej książki (ustanowiło ono swego czasu standardy programowania gier dla systemu DOS). Andre LaMothe programuje od ponad 25 lat i oprócz niewątpliwej praktyki posiada stosowne wykształcenie z zakresu matematyki, informatyki i elektrotechniki. Znany jest również jako założyciel firm Xtreme Games LCC, Nurve Networks i XGDC.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 10.

Przycinanie scen 3D

Nie bój się, to nie jest groźba pod Twoim adresem

— Marla, *Fight Club*

Dość długo udawało się nam unikać tematu przycinania scen 3D, przyszedł jednak czas na rozwiązanie i tego problemu. Nie da się już dłużej omijać tematu przycinania. Dlatego w bieżącym rozdziale omówione zostaną podstawy teoretyczne oraz zaprezentowany będzie praktyczny szkielet modułu przycinania scen 3D; wskazane zostaną też powody, dla których konieczne jest realizowanie przycinania scen oraz działający przykład wykorzystania tej techniki. Głównymi wątkami rozdziału będą:

- wprowadzenie do przycinania scen 3D;
- podstawy teoretyczne algorytmów przycinania;
- implementacja przycinania sceny do ostrosłupa widzenia;
- zabawa z przycinaniem.

Przycinanie scen 3D — wprowadzenie

Przycinanie jako jedna z technik przetwarzania sceny w grafice komputerowej wzmiankowane było w niniejszej książce już kilkukrotnie; stosowne omówienia znajdują się również w drugim wydaniu książki *Tricks of the Windows Game Programming Gurus*. Przycinanie to jedna z najważniejszych technik grafiki 3D, gdyż źle przycinane sceny są nie tylko nieprawidłowo wyświetlane na ekranie, ale również ich rzutowanie w ostrosłupie widzenia może doprowadzić do błędów dzielenia przez zero i niepoprawnych odwołań do pamięci. Mając to na uwadze przypomnimy sobie różne techniki przycinania i wskażemy przyczyny ich stosowania.

Przycinanie w przestrzeni obiektu

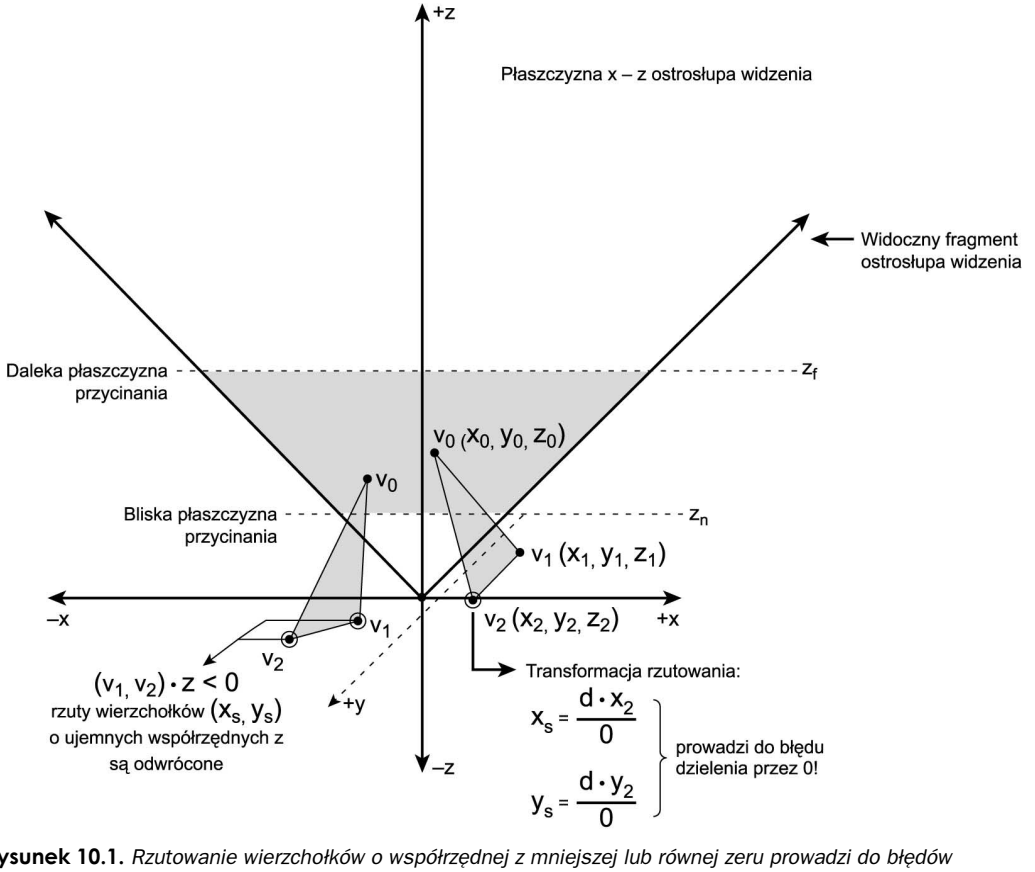
O przycinaniu w przestrzeni obiektu mówimy wtedy, gdy przycinaniu do wyznaczonego obszaru podlegają współrzędne geometryczne elementów sceny. Obszar przycinania może być dwu- lub trójwymiarowy — istotne jest to, że przycinanie zachodzi w matematycznej reprezentacji sceny i dotyczy matematycznych reprezentacji obiektów, trójkątów, elementów podstawowych i wszelkich innych części sceny. Zaletą przycinania w przestrzeni obiektu jest prostota: przy pomocy listy wierzchołków wielokątów tworzących scenę przycina się ich współrzędne do wyznaczonego dwu- lub trójwymiarowego obszaru rzutowania. Listę przyciętych trójkątów przekazuje się do następnego etapu potoku renderowania.

Wady przycinania w przestrzeni obiektu tkwią, jak zwykle, w szczegółach implementacyjnych. W omawianych w tej książce mechanizmach obrazowania jako elementy konstrukcyjne wykorzystywane są zawsze trójkąty. Tymczasem w wyniku przycinania, czy to do prostokąta rzutu dwuwymiarowego, czy do ostrosłupa widzenia, do sceny wprowadzane są wielokąty z więcej niż trzema wierzchołkami (a więc w wyniku przycinania scena przestaje składać się już wyłącznie z trójkątów). Tak więc dla każdej płaszczyzny przycinania

należy przewidzieć potrzebę wprowadzenia dodatkowych wierzchołków przetwarzanych trójkątów, przez co stają się one czworokątami bądź wielokątami o jeszcze większej liczbie kątów. To pierwsza trudność. Gdyby silnik graficzny optymalizowany był pod kątem przetwarzania wielokątów o dowolnej liczbie wierzchołków, nie byłoby problemu, ale dotychczas omówione optymalizacje przetwarzania 3D opierały się zawsze na założeniu, że scena składa się wyłącznie z trójkątów. Dalej, przycinanie w dowolnym systemie przestrzeni obiektu oznacza konieczność przecinania prostych prostymi, prostych płaszczyznami i tak dalej. Z pozoru proste, operacje te okazują się trudne w wydajnej implementacji, wymagając rozwiązania wielu problemów.

Przycinanie 2D w przestrzeni obiektu

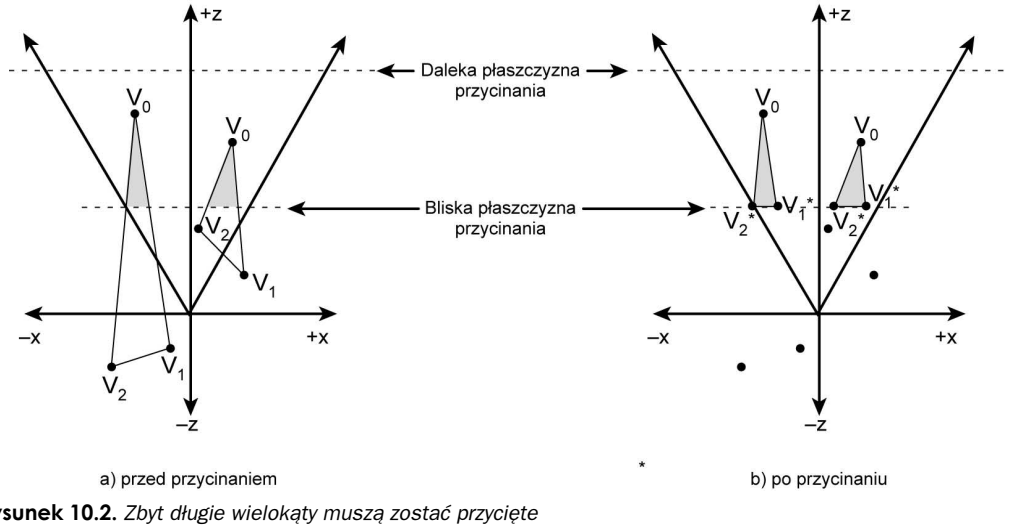
Przycinanie 2D w przestrzeni obiektu nie jest nam całkiem obce, ponieważ wielokrotnie prezentowaliśmy je dla grafiki 2D. Podstawowa koncepcja takiego przycinania zakłada, że dysponujemy gotowymi już *rzutami* prostych i wielokątów, które należy przyciąć do prostokątnego obszaru widzenia. Kod implementujący takie przycinanie prezentowany był już zarówno w tym, jak i poprzednim wydaniu książki. Przycinanie 2D nie może być dłużej pomocne, ponieważ podstawowym problemem jest to, że w przestrzeni trójwymiarowej rzutowaniu podlegają wielokąty, które mogą być potencjalnie tak rozległe, że nie tylko przenikają przez bliską płaszczyznę przycinania, ale również przez płaszczyznę o współrzędnej z równej 0, stąd do rzutowania przekazywane są wierzchołki o ujemnych wartościach tej współrzędnej (jak na rysunku 10.1). Wyłania się więc potrzeba opracowania techniki bardziej agresywnego przycinania 3D w przestrzeni obiektu.



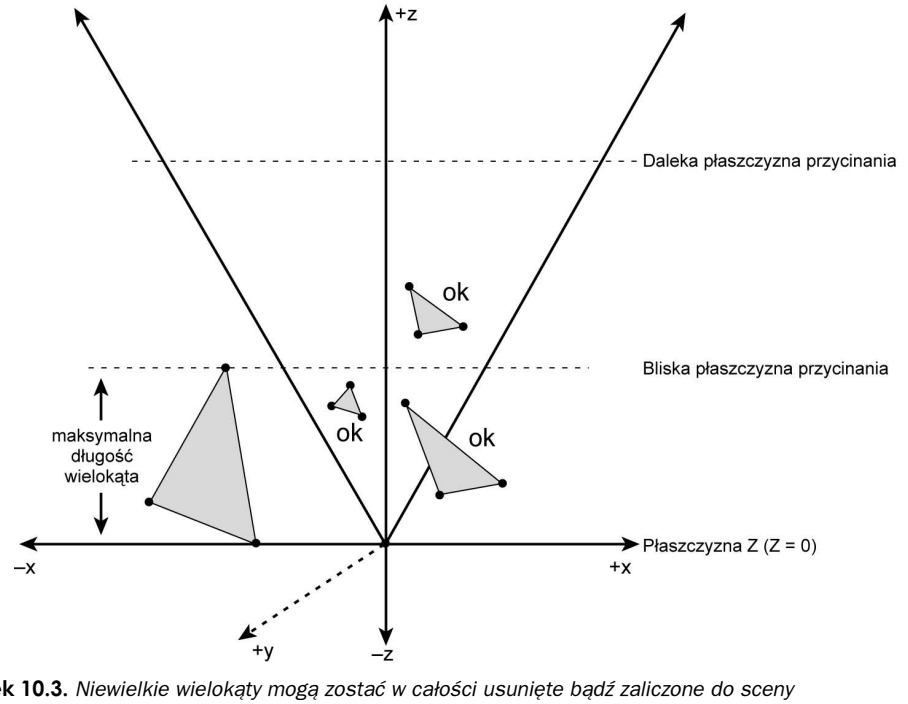
Rysunek 10.1. *Rzutowanie wierzchołków o współrzędnej z mniejszej lub równej zeru prowadzi do błędów*

Przycinanie 3D w przestrzeni obiektu

Jeżeli mechanizm obrazowania sceny ma sobie radzić z wielokątami o dowolnych rozmiarach i rozmieszczeniu, nie da się w prosty sposób uniknąć przycinania sceny do bliskiej płaszczyzny przycinania (patrz rysunek 10.2). Najbardziej prymitywnym sposobem uniknięcia przycinania jest usuwanie w całości ze sceny tych wielokątów, których wierzchołki wystają poza płaszczyznę przycinania, ale wymaga to konstruowania sceny z wielokątów o ograniczonym rozmiarze (patrz rysunek 10.3). Przy tym fragmenty scen przylegające do płaszczyzn przycinania nie będą właściwie obrazowane.



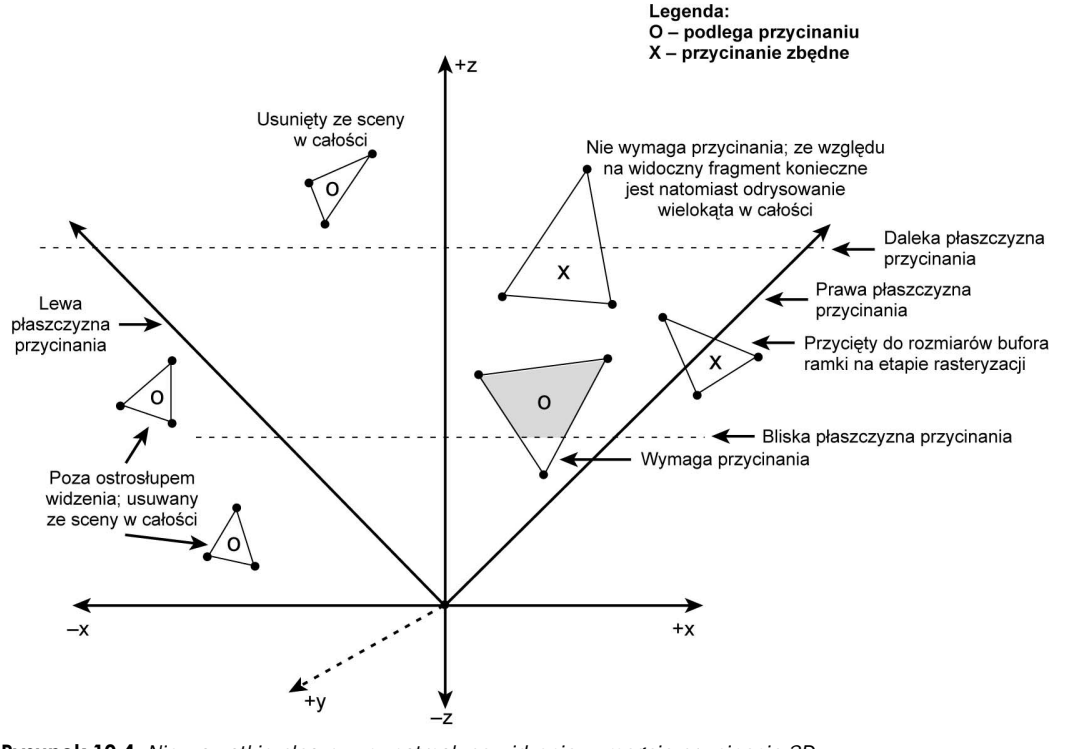
Rysunek 10.2. *Zbyt długie wielokąty muszą zostać przycięte*



Rysunek 10.3. *Niewielkie wielokąty mogą zostać w całości usunięte bądź zaliczone do sceny*

Warunkiem koniecznym prawidłowego obrazowania sceny jest przycinanie wszystkich wielokątów sceny do bliższej płaszczyzny, przy czym uzyskane w wyniku przycięcia wielokąty mogą przestać być trójkątami. Trzeba się więc liczyć z koniecznością ponownego podziału powstałych wielokątów na trójkąty i tym samym przebudowania listy obrazowanych wielokątów. W najgorszym więc przypadku konieczne będzie przycięcie wszystkich wielokątów do bliskiej płaszczyzny przycinania, co sprowadza się do wyliczania punktów przecięcia prostych i płaszczyzn. W szczegółach problem okazuje się poważniejszy. Wiemy już o konieczności ponownego podziału sceny na trójkąty; dodatkowo należy jeszcze przyciąć współrzędne tekstur, przeliczyć dane potrzebne do cieniowania powierzchni i wyliczyć nowe wartości wektorów normalnych wierzchołków — widać, że trudności jest niemało, ale poradzimy sobie i z nimi.

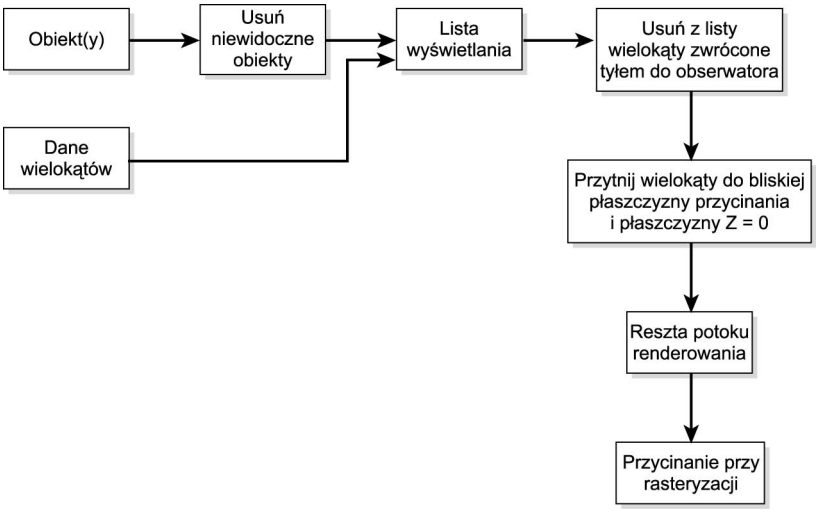
Z drugiej strony, niebawem okaże się, że nie ma potrzeby wykonywania operacji przycinania dla pozostałych płaszczyzn ostrosłupa widzenia. Spójrzmy na rysunek 10.4. Okazuje się, że przycinanie wielokątów do dalekiej płaszczyzny przycinania jest zwykłą stratą czasu. Jaki byłby zysk? Praktycznie żaden. Co gorsza, przycinanie to znacznie zwiększyłoby czas przetwarzania sceny. Wystarczy więc (zamiast przycinać) odrzucać w całości te wielokąty, które znajdują się poza daleką płaszczyzną przycinania — niezależnie od ich odległości od tej płaszczyzny. Wielokąty, które w całości znajdują się poza tą płaszczyzną, zostaną raczej usunięte z wykorzystaniem sfer otaczających niż przycinania.



Rysunek 10.4. *Nie wszystkie płaszczyzny ostrosłupa widzenia wymagają przycinania 3D*

Dalej, również przycinanie na płaszczyznach tworzących górną, dolną, lewą i prawą ścianę ostrosłupa widzenia nie ma sensu. Przycinanie na ścianach ostrosłupa widzenia zajęłoby cenny czas, a służyłoby tylko do tego, żeby rzut sceny nie wykraczał poza obszar dwuwymiarowej płaszczyzny widoku sceny. O to można zaś zadbać na etapie rasteryzacji, w obszarze bufora ramki ekranu, co jest nieporównanie szybsze niż przycinanie w przestrzeni obiektu. Unikając przycinania na płaszczyznach ostrosłupa widzenia eliminujemy również ze sceny i potoku renderowania potencjalne dodatkowe wielokąty, powstałe z podziału wielokątów na granicach obszaru przycinania.

Naturalnie, jeżeli Czytelnikowi zależy na elegancji i kompletności mechanizmu obrazowania, może pokusić się o implementację przycinania dla wszystkich płaszczyzn; zostanie to zresztą pokazane w dalszej części rozdziału. Główny nacisk zostanie jednak położony na następujące etapy przycinania i usuwania wielokątów (patrz rysunek 10.5):



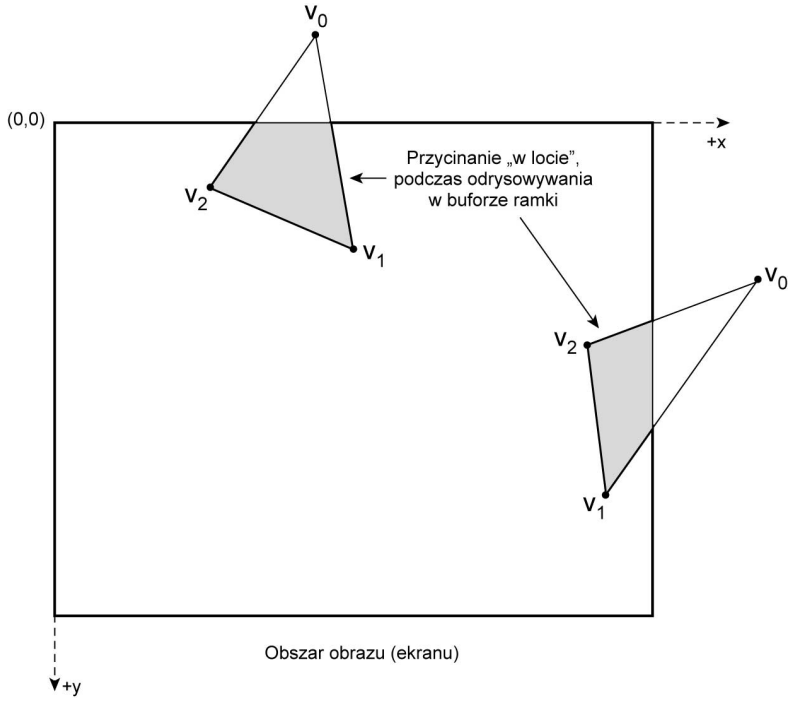
Rysunek 10.5. *Etapy w potoku przycinania 3D*

1. Usunięcie wszystkich zbędnych obiektów.
2. Usunięcie wielokątów zwróconych „tyłem do obserwatora".
3. Przycięcie wszystkich wielokątów do obszaru ostrosłupa widzenia. Pełnego przycinania wymaga jedynie bliska płaszczyzna przycinania w osi Z. Przycinanie na pozostałych pięciu płaszczyznach sprowadza się zaś do prostej kwalifikacji wielokąta do sceny w całości (jeżeli trójkąt znajduje się częściowo lub w całości w obszarze przycinania, zostaje zaliczony do sceny w całości; trójkąty pozostające w całości poza lewą, prawą, górną i dolną płaszczyzną ostrosłupa widzenia można w całości usunąć ze sceny).
4. Przekazanie wszystkich wielokątów do następnych etapów potoku renderowania; wszelkie trójkąty wystające poza obszar płaszczyzny rzutu można przyciąć na etapie rasteryzacji.

Taki system przycinania jest łatwy w implementacji, działa znakomicie, i — co najważniejsze — jest bardzo szybki.

Przycinanie w obszarze obrazu sceny

Należałoby jeszcze sprecyzować, co należy rozumieć pod pojęciem przycinania w obszarze rzutu ekranu. Sytuacja wejściowa prezentowana jest na rysunku 10.6: wielokąt jest gotowy do odrysowania, ale jego fragment wystaje poza jedną z prostych wyznaczających obszar obrazu sceny. Nie jest to problem, o ile przycinanie ma nastąpić w pionie. Wystarczy rozpocząć pętlę odrysowywania wielokąta od wierzchołka uzyskanego przez obliczenie punktu przecięcia trójkąta z obszarem obrazu. Wyliczenie takie to zaledwie jeden dodatkowy wiersz kodu programu. W przypadku przecięcia z lewą bądź prawą krawędzią obrazu procedura odrysowywania jest nieco inna: przycinane są kolejne linie tworzone w pętli odrysowywania. Jest to szybsze niż przycinanie każdego wystającego trójkąta do wszystkich czterech płaszczyzn ostrosłupa widzenia. To ostatnie oznacza bowiem dodanie do sceny potencjalnie wielu nowych wielokątów, a co za tym idzie, oznacza konieczność wyliczania nowych współrzędnych teksturowania, przeliczania wektorów normalnych wielokątów i wierzchołków i tak dalej. Tymczasem aż nadto obciążające obliczeniowo jest już samo przycinanie do bliskiej płaszczyzny przycinania.



Rysunek 10.6. *Przycinanie w obszarze obrazu sceny, podczas rysowywania wielokątów*

Z drugiej strony, nie można dopuścić, aby przycinanie w obszarze bufora ramki doprowadziło do utraty jakichkolwiek informacji 3D; będą one jeszcze potrzebne, między innymi przy okazji konstruowania bufora Z. Operacje w obszarze bufora ramki wymagają ostrożności: gdy dojdzie do interpolacji wartości współrzędnych z dla krawędzi wielokątów, konieczne będzie zachowanie przynajmniej współrzędnych z wierzchołków. Chodzi o to, że podczas ostatecznej rasteryzacji nie można będzie już realizować przycinania wyłącznie na podstawie wartości współrzędnych 2D. Dojdziemy do tego w stosownym czasie.

Omówienie algorytmów przycinania

Przed wkroczeniem w obszar teorii algorytmów przycinania chciałbym uprzedzić, że „poważne” nazwy co niektórych algorytmów niekiedy rozśmieszają, a to dlatego, że każdy z nich można wymyślić samemu — wszystkie są po prostu formalizacją tego, co w domu robi każdy, kto zajmuje się choć trochę algorytmami graficznymi. Nazewnictwo algorytmów jest zresztą nieodłącznym problemem grafiki komputerowej: co i raz ktoś wymyśla „nowy” algorytm i publikuje go; odtąd zaczyna on funkcjonować pod pewną nazwą, niezależnie od innych nazw tego samego algorytmu publikowanego gdzie indziej. Nie zawsze zresztą nowa nazwa starego algorytmu jest dziełem samego autora publikacji — często czytelnicy zaczynają odwoływać się do „nowego” algorytmu pod nową nazwą.

Przycinanie to koncepcja na tyle prosta, że śmieszne byłoby sądzić, że została „odkryta” przez jedną konkretną osobę. Każdy algorytm przycinania jest pewnym wcieleniem zdroworozsądkowego podejścia do przetwarzania grafiki i opiera się na wyliczaniu punktów przecięcia wierzchołków. Algorytmy takie „produkuje” się zwykle pod presją pojawiających się zadań — niejednokrotnie wyważa się wtedy otwarte już drzwi. Do czego zmierzam? Otóż nieco dalej wymienione zostaną z nazwy pewne powszechnie przyjęte algorytmy przycinania, jednak nie powinniście na ich podstawie wyrabiać sobie poglądu, że są one jedynymi możliwymi wcieleniami przycinania. Jak Wam zapewne wiadomo, programista niejednokrotnie spędza całe godziny na rozwiązaniu problemu, po czym dowiaduje się, że jego rozwiązanie jest bardzo podobne do już istniejących algorytmów. W przypadku algorytmów przycinania byłoby prościej, gdyby ich nazwy były bardziej opisowe. Istniejące nazwy brzmią przecież jak nazwy kancelarii prawniczych!

Wydaje się, że aby algorytm otrzymał nazwę, powinien cechować się pewnym wyrafinowaniem, powodującym, że opisywana przezeń technika nie jest bynajmniej oczywista dla osób średnio tylko zaawansowanych w danej dziedzinie. Zgodnie z takim rozumowaniem uzasadnione jest uhonorowanie wynalazcy algorytmu drzewa BSP (ang. *binary space partitioning*) przez nadanie mu jego imienia — podział trójwymiarowej przestrzeni BSP nie jest bowiem algorytmem trywialnym. Kiedy jednak chodzi o przycinanie wielokąta do innego wielokąta, opracowanie algorytmu wymaga godziny (może dwóch) zastanowienia, spędzonego na kombinowaniu metod przycinania odcinków, klasyfikacji wierzchołków, kodowania bitowego czy parametrycznych reprezentacji prostych. I tyle!

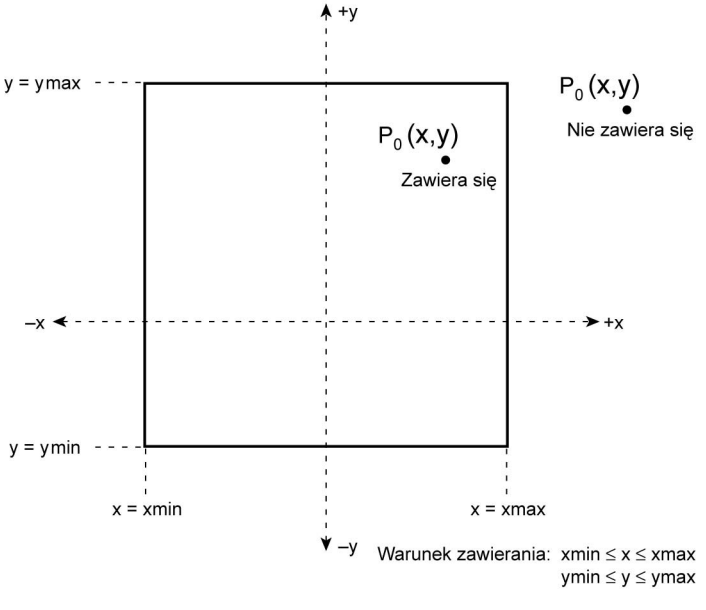
Mam zamiar, rzecz jasna, zachęcić Was do eksperymentów. Co prawda, dziewięćdziesiąt dziewięć procent rozwiązywanych w ten sposób problemów doczekało się już rozwiązania, nie oznacza o jednak, że rozwiązania te zostały gdziekolwiek opublikowane.

Dosyć narzekania. Do dzieła! Przystąpimy teraz do omówienia — z teoretycznego punktu widzenia — najpopularniejszych algorytmów przycinania, po czym zajmiemy się implementacją takiego algorytmu na potrzeby naszego silnika graficznego. Prezentowany algorytm, jako że powstał pod presją konkretnego problemu, będzie hybrydą wielu pomysłów. Na początek nieco informacji podstawowych.

Przycinanie — podstawy

Przycinanie wielokątów sprowadza się do przycinania odcinków, co z kolei sprowadza się do określania, czy pewne punkty zawierają się w danym obszarze dwuwymiarowym bądź w trójwymiarowej bryle. Ilustracją problemu w jego postaci ogólnej jest rysunek 10.7. Otóż dany jest zbiór odcinków, tworzących wchodzące w skład sceny wielokąty, które mają zostać później poddane rzutowaniu. Należy określić, czy elementy te mieszczą się w pewnych zadanych granicach. Da się to sprowadzić do określenia, czy w granicach tych mieszczą się pojedyncze punkty charakterystyczne elementów. Innymi słowy, dane trójkąty determinowane są trzema wierzchołkami, łączonymi trzema odcinkami (krawędziami). Choć to właśnie krawędzie nadają trójkątowi zarys, naprawdę ważne są jedynie wierzchołki. Stąd, na najwyższym poziomie ogólności, problem sprowadza się do sprawdzenia, czy punkt o współrzędnych (x, y) bądź (x, y, z) znajduje się w zadanym obszarze (bryle), a więc do *testu zawierania się punktu*.

Rysunek 10.7. *Test zawierania się na płaszczyźnie dwuwymiarowej*



Test zawierania się punktu w obszarze 2D

Dla danego punktu *p0*(x, y) i prostokątnego obszaru wyznaczanego prostymi xmin, xmax, ymin, ymax (patrz rysunek 10.7), punkt *p0* zawiera się w zadanym obszarze, jeżeli spełnia następujące nierówności:

```
xmin <= x <= xmax
```

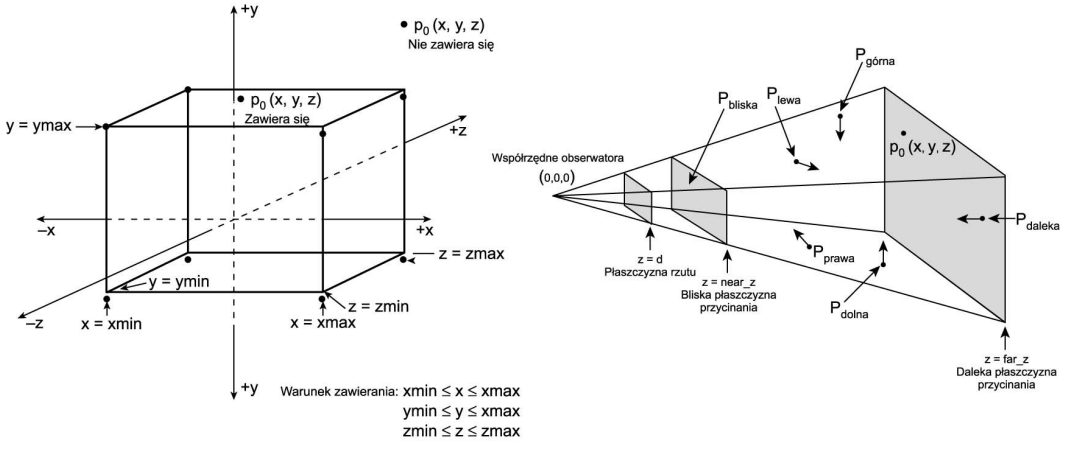
i

```
ymin <= y <= ymax
```

Testy zawierania się punktu w bryle 3D

Przypadek 1. Zawieranie się punktu w bryle prostopadłościennej.

Dla danego punktu *p0*(x, y, z) i prostopadłościennego obszaru wyznaczonego płaszczyznami xmin, ymin, zmin, xmax, ymax, zmax (patrz rysunek 10.8), punkt *p0* zawiera się w zadanym obszarze, jeżeli spełnia następujące nierówności:



a) bryła prostopadłościenna **b) ostrosłup**

Rysunek 10.8. *Testowanie zawierania się punktu w bryle 3D*

```
xmin <= x <= xmax
```

i

```
ymin <= y <= ymax
```

i

```
zmin <= z <= zmax
```

Pojawia się tu wątpliwość, czy warto zawracać sobie głowę przycinaniem do bryły prostopadłościennej, jeżeli z góry wiadomo, że ostrosłup widzenia taką bryłą nie jest? To prawda, ale wystarczy przypomnieć sobie omówienie rzutowania i dyskutowany przy tej okazji efekt „prostowania" ostrosłupa widzenia właśnie do postaci bryły prostopadłościennej. W takim przypadku przycinanie sprowadza się do trywialnych porównań współrzędnych wierzchołków wielokątów sceny ze współrzędnymi wierzchołków ścian bryły zawierania. Nikt też nie mówił, że przycinanie realizowane jest wyłącznie do ostrosłupa widzenia — przecież promienie lasera wystrzeliwane w grze warto przyciąć również do jakiejś skończonej bryły w rozległym świecie gry kosmicznej, aby nie przeliczać ich pozycji w nieskończoność, gdy dawno opuściły penetrowany układ planetarny.

Przypadek 2. Przypadek ogólny zawierania się punktu w bryle ostrosłupa widzenia.

Przypadek ten będzie interesował nas najbardziej. To standardowy przypadek w przycinaniu: danych jest sześć płaszczyzn tworzących ścięty ostrosłup widzenia, do którego należy przyciąć wszelkie elementy geometryczne sceny. Ostrosłup widzenia można zdefiniować na kilka sposobów, na przykład przez określenie współrzędnych z płaszczyzn przycinania (bliższej i dalszej) wraz ze współrzędnymi prostokątnego obszaru na płaszczyźnie bliższej. Jednak niezależnie od sposobu wyznaczenia zadanych będzie sześć płaszczyzn. Nazwijmy je *P*górna, *P*dolna, *P*prawa, *P*lewa, *P*daleka i *P*bliska. Dodatkowo załóżmy, że płaszczyzny zdefiniowano w taki sposób, że ich wektory normalne skierowane są do wewnątrz bryły ściętego ostrosłupa widzenia. Dalej, dla punktu *p0*(x, y, z) można wyznaczyć półprzestrzeń, w której się znajduje. Przynależność do półprzestrzeni dodatniej oznacza, że punkt znajduje się po wewnętrznej stronie płaszczyzny przycinania, przynależność do półprzestrzeni ujemnej — że punkt znajduje się po stronie zewnętrznej, a tym samym poza ostrosłupem widzenia. Przypominacie sobie zapewne, że do określenia, czy dany punkt leży na płaszczyźnie, czy też należy do półprzestrzeni dodatniej bądź ujemnej, służy równanie płaszczyzny, prezentowane w rozdziałach 4. i 5. Załóżmy więc, że operator HS(*p*, pł) zwraca wartość półprzestrzeni punktu *p* względem płaszczyzny pł (dla półprzestrzeni dodatniej wynikiem operatora jest +1, dla ujemnej — –1, a w przypadku, gdy punkt znajduje się dokładnie na płaszczyźnie — 0).

Do określenia zawierania się punktu w bryle niezbędne jest sprawdzenie, czy współrzędne punktu spełniają następujące nierówności:

```
HS(p0, Pgórna) > 0 i HS(p0, Pdolna) > 0
HS(p0, Pprawa) > 0 i HS(p0, Plewa) > 0
HS(p0, Pbliska) > 0 i HS(p0, Pdaleka) > 0
```

Obliczenia te są tylko pozornie skomplikowane.

Przycinanie odcinków

Znamy już sposób określania, czy dany punkt znajduje się wewnątrz, czy na zewnątrz wyznaczonego obszaru przycinania, zarówno na płaszczyźnie, jak i w przestrzeni trójwymiarowej. Kolejny krok to przejście na wyższy poziom abstrakcji i przycinanie odcinków (krawędzi) wyznaczanych punktami i tym samym przycinanie konturów trójkątów. Potrzebujemy tu narzędzia służącego do wykrywania punktów przecięć odcinków z płaszczyznami i innymi odcinkami. Stosowne wzory podane zostały i omówione w rozdziałach 4. i 5., nie będziemy więc od nowa ich wyprowadzać; konieczne będzie jednak odświeżenie pamięci.

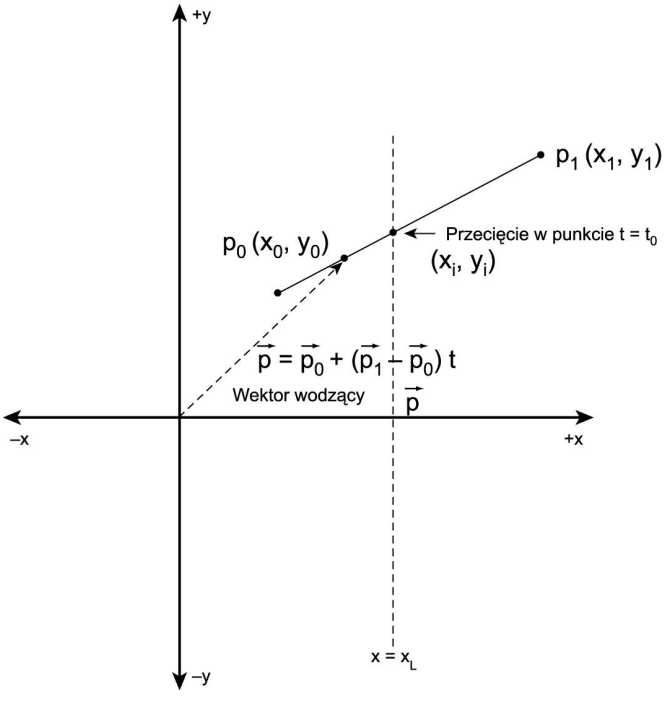
Przycinanie wielokątów to tylko pewne rozszerzenie zagadnienia przycinania odcinków, więc — nie uprzedzając omówienia — rzecz sprowadza się do wymyślenia sposobu przycięcia odcinka względem nieskończonej prostej (na płaszczyźnie) bądź względem płaszczyzny (w przestrzeni 3D). Pełny algorytm przycinania sceny to po prostu iteracyjne lub rekurencyjne stosowanie procedury przycinania dla każdego odcinka wielokąta względem każdej płaszczyzny przycinania ograniczającej ostrosłup widzenia. Naturalnie, algorytmy przycinania są odpowiednio optymalizowane, ale sprowadzają się właśnie do powyższej procedury, i to niezależnie od liczby wymiarów sceny.

Przycinanie odcinków — wersja 2D

Skupmy się na zagadnieniu przycinania odcinków. Na pierwszy ogień pójdzie przypadek przycinania w przestrzeni dwuwymiarowej. Odwołując się do rysunku 10.9 wyobrażamy sobie odcinek łączący punkty *p0*(x0, y0) i *p1*(x1, y1). Odcinek ten ma zostać przycięty względem pionowej prostej wyznaczanej równaniem x = xl.

Przycinanie trójkątów rzadko będzie odbywało względem prostych o dowolnym kierunku — zwykle wystarcza analiza przypadków względem prostych poziomych i pionowych.

Rysunek 10.9.
Proste przycinanie odcinków



Aby przyciąć prostą (odcinek), wystarczy dokonać konwersji jej reprezentacji do postaci parametrycznej i podstawić wartość x w punkcie przecięcia. Oto sposób rozumowania, który prowadzi do takiego wniosku:

W postaci wektorowej pozycja punktu wzdłuż odcinka to:

```
p = p0 + (p1 - p0) * t
```

W postaci rozbitej na składowe:

```
x = x0 + (x1 - x0) * t, dla 0 <= t <= 1
y = y0 + (y1 - y0) * t, dla 0 <= t <= 1
```

Do sprawdzenia, czy dany odcinek przecina zadaną prostą pionową, należy podstawić wartość x definiującą tę prostą (tutaj x = xl) do lewej strony równania dla składowej x, a następnie wyliczyć z tego równania wartość t. Jeżeli t należy do przedziału <0, 1>, odcinek przecina zadaną prostą; dla t spoza tego przedziału wiadomo, że odcinek nie przecina prostej:

```
x = x0 + (x1 - x0) * t
```

Po podstawieniu w miejsce x wartości xl otrzymujemy:

```
xl = x0 + (x1 - x0) * t
```

Stąd t ma wartość:

```
t = (xl - x0)/(x1 - x0)
```

Jeżeli t ma wartość pomiędzy 0 a 1, odcinek przecina prostą. Należy pamiętać, że odcinek zdefiniowany tak, jak w przykładzie, rozciąga się pomiędzy punktami *p0* i *p1*, przy t zmieniającym się od 0 do 1. Wartości t spoza tego zakresu należą co prawda do prostej, nie należą jednak do jej odcinka ograniczanego punktami *p0* i *p1*. Po wyliczeniu t i stwierdzeniu, że mieści się ona w zakresie od 0 do 1,wystarczy podstawić je do równania dla składowej y:

```
y = y0 + (y1 - y0) * t
```

I otrzymujemy współrzędne punktu przecięcia z danego odcinka z prostą pionową x = xl. Wyznaczenie punktu przecięcia z prostą poziomą jest równie proste, tyle, że jako pierwsze rozwiązujemy równanie dla składowej y i uzyskaną stąd wartość t podstawiamy do równania dla składowej x. Oto przykład dla prostej odcięcia wyznaczanej wzorem y = yl:

Po podstawieniu yl do odpowiedniego równania otrzymujemy:

```
yl = y0 + (y1 - y0) * t
```

Stąd wartość t to:

```
t = (yl - y0)/(y1 - y0)
```

Po sprawdzeniu wartości t otrzymujemy informację, czy nastąpiło przecięcie danych odcinków (dla t z zakresu od 0 do 1). Jeżeli tak, podstawiamy otrzymaną wartość do równania składowej x:

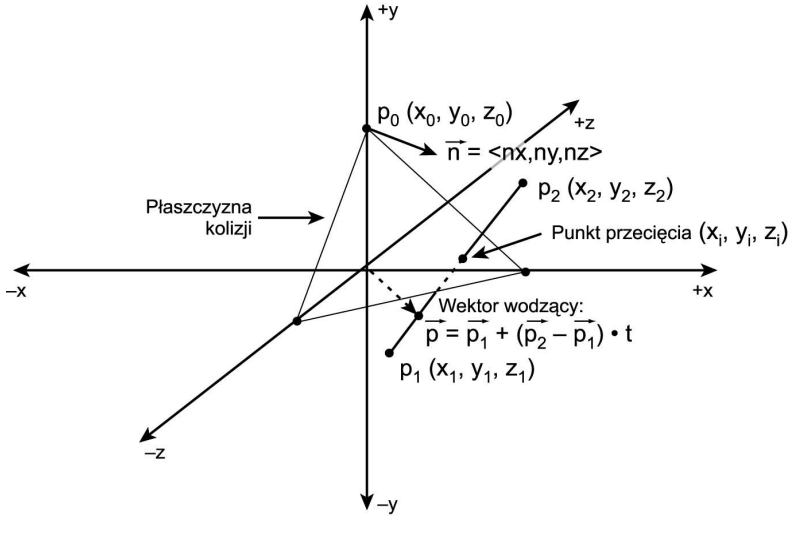
```
x = x0 + (x1 - x0) * t
```

I otrzymujemy drugą współrzędną punktu przecięcia odcinka z poziomą prostą y = yl.

Oczywiście, efektywna implementacja takiego przycinania wymaga podejścia zdroworozsądkowego i wprowadzenia stosownych testów odsiewających. Przykładowo, jeżeli punkty końcowe *p0* i *p1* mają identyczne współrzędne, w ogóle nie trzeba przeprowadzać testu na przecięcie z prostą. Podobnie daremna jest próba obliczenia punktu przecięcia odcinka poziomego z poziomą prostą i tak dalej.

Przycinanie odcinków — wersja 3D

Przypadek przecinania prostych w przestrzeni trójwymiarowej okazuje się równie łatwy jak na płaszczyźnie — cały problem sprowadza się do odpowiedniej reprezentacji płaszczyzn. Na razie nie będziemy jednak wprowadzać żadnych sztuczek i przyjrzymy się ogólnemu przypadkowi przycinania odcinków w przestrzeni 3D przy zadanej parametrycznie prostej i zadanej płaszczyźnie. Problem ilustrowany jest rysunkiem 10.10. Na pierwszy ogień pójdzie równanie płaszczyzny — będzie ona definiowana w postaci punktu zaczepienia i wektora normalnego płaszczyzny.



Rysunek 10.10. *Odcinek przecinający płaszczyznę w przestrzeni trójwymiarowej*

Niech *p0* = (x0, y0, z0) będzie punktem leżącym na płaszczyźnie, a n [nx, ny, nz] — wektorem normalnym tej płaszczyzny. Równanie płaszczyzny przyjmie wtedy postać:

```
nx * (x - x0) + ny * (y - y0) + nz * (z - z0) = 0
```

Odcinek pomiędzy punktami *p1* i *p2* dany będzie parametrycznie, w zapisie wektorowym:

```
p = p1 + (p2 - p1) * t, dla 0<= t <= 1
```

W postaci równań składowych:

```
x = x1 + (x2 - x1) * t
y = y1 + (y2 - y1) * t
z = z1 + (z2 - z1) * t
```

Pora na wyliczenie punktu przecięcia. Ponownie odsyłam Was do rozdziału 4., w którym zaprezentowano wzory niezbędne do realizacji obliczenia; objaśnienie procesu obliczeniowego można jednak sprowadzić do stwierdzenia, że wartości x, y i z prostej parametrycznej podstawiane są do równania płaszczyzny, po czym z równania tego obliczana jest wartość t:

```
nx * (x - x0) + ny * (y - y0) + nz * (z - z0) = 0
```

Po podstawieniu x, y i z do równania płaszczyzny otrzymujemy:

```
nx * ((x1 + (x2 - x1) * t) + ny * ((y1 + (y2 - y1) * t) + nz * ((z1 + (z2 - z1) * t) = 0
```

Po obliczeniu wartości t należy podstawić ją z powrotem do równania prostej — w ten sposób wyliczone zostaną odpowiednie współrzędne punktu przecięcia. Jako, że pokazywaliśmy to już dla przypadku 2D, poszczególne etapy obliczenia zostaną tym razem pominięte. To już wszystko na temat przycinania, sprowadza się on bowiem właśnie do umiejętności obliczania punktów przecięcia prostej z prostą i prostej z płaszczyzną. Teraz wiedzę tę należy wykorzystać w algorytmie, który będzie ją efektywnie implementował. Przejdźmy więc do opisu kilku popularnych algorytmów przycinania.

Algorytm Cohena-Sutherlanda

Algorytm Cohena-Sutherlanda to jedna z najpopularniejszych technik przycinania odcinków do prostokątnych obszarów płaskich bądź do prostopadłościennych brył w przestrzeni trójwymiarowej. Algorytm dzieli się na dwie fazy.

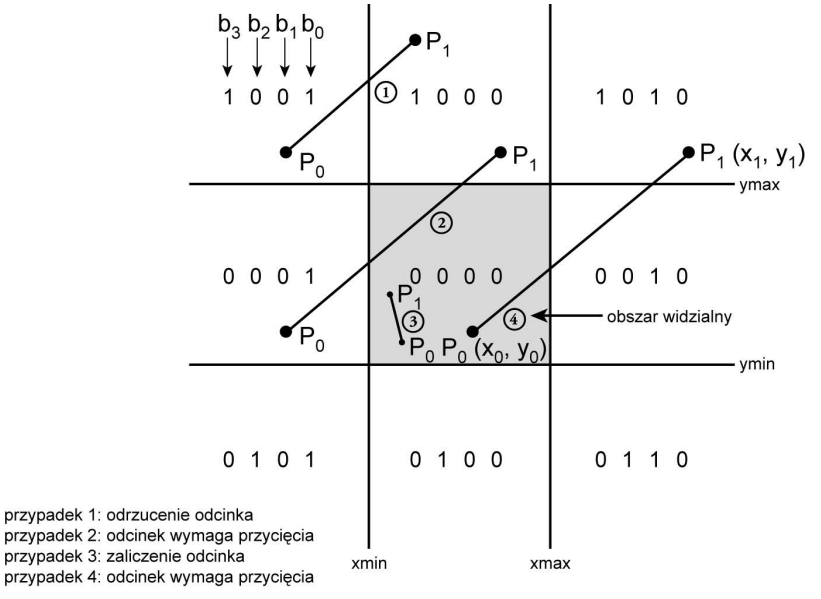
Faza klasyfikacji wierzchołków

W fazie klasyfikacji punktów końcowych każdy odcinek na płaszczyźnie bądź w przestrzeni jest oznaczany jako leżący wewnątrz lub w poza obszarem przycinania; do klasyfikacji wykorzystuje się kodowanie bitowe, reprezentujące pozycję każdego z punktów końcowych odcinka jako wewnątrz (1) lub poza (0) zadanym obszarem. Schemat kodowania klasyfikującego dla przypadku 2D ilustrowany jest rysunkiem 10.11.

Każdy wierzchołek (x, y) odcinka ograniczonego punktami *p0*(x0, y0) i *p1*(x1, y1) wymaga osobnej klasyfikacji. Klasyfikacja punktów końcowych odcinka kodowana jest na czterech (sześciu w przestrzeni trójwymiarowej) bitach. Kod dla punktu *p0* określać będziemy symbolem kb0 (analogicznie, dla *p1* będzie to symbol kb1). Bity reprezentujące poszczególne obszary klasyfikacji mogą być kodowane w dowolny sposób, dla obszaru prostokątnego ograniczanego punktami (xmin, ymin) i (xmax, ymax) przyjęto jednak kody wymienione w tabeli 10.1.

Tabela 10.1. *Kodowanie położenia wierzchołków odcinków w algorytmie Cohena-Sutherlanda*

Nr bitu	Znaczenie dla wartości 1
3	`y > ymax`, powyżej górnej krawędzi obszaru przycinania
2	`y < ymin`, poniżej dolnej krawędzi obszaru przycinania
1	`x > xmax`, na prawo od prawej krawędzi obszaru przycinania
0	`x < xmin`, na lewo od lewej krawędzi obszaru przycinania



Rysunek 10.11. *Schemat kodowania klasyfikującego w algorytmie Cohena-Sutherlanda*

Po dokonaniu klasyfikacji punktów końcowych odcinków przechodzimy do fazy przetwarzania, w której realizowana jest analiza położenia odcinków i dochodzi do faktycznego ich przycinania.

Faza przetwarzania

Faza przetwarzania rozpoczyna się od prostego zaliczania bądź prostego odrzucania odcinka jako leżącego w całości wewnątrz lub na zewnątrz obszaru przycinania. Algorytm ten jest o tyle przyjazny, że sposób kodowania położenia punktów końcowych odcinków eliminuje konieczność konstruowania rozbudowanych instrukcji warunkowych. Dzięki odpowiedniemu kodowaniu bitowemu punktów końcowych odcinka na wiele pytań można odpowiedzieć przy pomocy pojedynczej operacji bitową. Załóżmy, że zachodzi potrzeba określenia, czy dany odcinek nie znajduje się przypadkiem w całości poza obszarem przycinania. Normalnie należałoby sprawdzić, czy oba wierzchołki równocześnie znajdują się nad, pod, na lewo bądź na prawo krawędzi obszaru przycinania — ileż to instrukcji `if`! Dzięki kodom klasyfikującym jedna operacja bitowa rozwiewa szereg wątpliwości. Najprostsze przypadki klasyfikacji odcinków wyglądają następująco:

Test 1.

Jeżeli (kb0 = 0) i (kb1 = 0), odcinek *p0*, *p1* znajduje się w całości wewnątrz obszaru przycinania, więc nie trzeba poddawać go operacji przycinania do krawędzi obszaru (patrz rysunek 10.11).

W innym przypadku należy dokonać logicznej koniunkcji poszczególnych bitów, a jej wynik określi położenie odcinka:

```
kb = (kb0 & kb1)
```

Test 2.

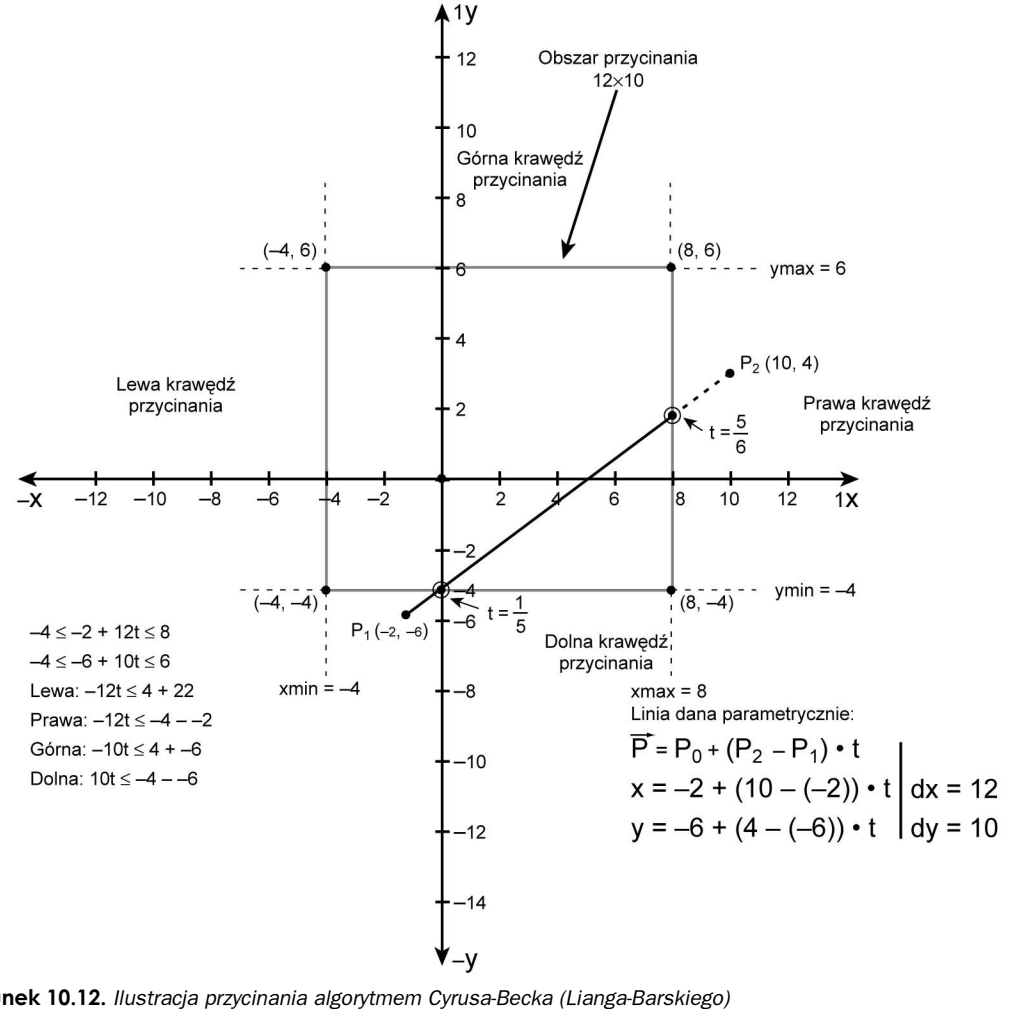
Jeżeli (kb > 0), oba punkty końcowe odcinka znajdują się poza jedną wspólną krawędzią obszaru przycinania i odcinek można odrzucić jako w całości leżący poza tym obszarem (patrz rysunek 10.11).

Jeżeli oba dotychczasowe testy nie pozwoliły na przyjęcie bądź odrzucenie odcinka, należy przyjąć, że potencjalnie odcinek ten może przecinać obszar przycinania w dwóch punktach i wymaga pełnego przycinania. Również ten przypadek ilustrowany jest na rysunku 10.11. W takim układzie można postąpić na

kilka sposobów. Jeden z nich polega na przycięciu odcinka do obu krawędzi przycinania – dzielimy w ten sposób problematyczny odcinek na dwa lub więcej fragmentów i możemy dokonać ponownej klasyfikacji ich punktów końcowych. Do obliczenia punktów przecięcia z krawędziami obszaru przycinania można zastosować algorytm typu „dziel i rządź", albo nawet posiłkować się metodą pełnego przeglądu (ang. *brute force*). Osobiście zwykłem wykorzystywać wyłącznie fazę klasyfikacyjną tego algorytmu; w fazie przetwarzania, po wykonaniu obu dodatkowych testów, przycięcie odcinka można bowiem zrealizować w dowolny sposób.

Algorytm Cyrusa-Becka (Lianga-Barskiego)

Kolejny algorytm jest nieco uogólniony i działa dla prostych parametrycznych i wypukłych (dwu- i trójwymiarowych) obszarów przycinania. Algorytm opiera się na założeniu, że najbardziej naturalną reprezentacją prostej jest reprezentacja parametryczna i do reprezentacji punktów przecięcia wykorzystuje, w miejsce jawnych wartości współrzędnych punktu przecięcia (x, y) czy (x, y, z), wartości t. Dzięki temu wykrywanie przecięć realizowane jest na podstawie stwierdzenia pokrywania się wartości t i wielkości tego pokrycia (rysunek 10.12).



Rysunek 10.12. *Ilustracja przycinania algorytmem Cyrusa-Becka (Lianga-Barskiego)*

Na powyższym rysunku ilustrowany jest wariant przycinania na płaszczyźnie dwuwymiarowej. Krawędzie obszaru przycinania definiowane są wartościami xmin, xmax, ymin i ymax. Odcinek dany jest w postaci parametrycznej:

```
p = p1 + (p2 - p1) * t, dla 0 <= t <= 1
```

albo w postaci równań składowych, przy dx = (x2 – x1) i dy = (y2 – y1):

```
x = x1 + dx * t
y = y1 + dy * t
```

Odcinek znajduje się wewnątrz obszaru przycinania, jeżeli spełnione są następujące nierówności:

```
Dla krawędzi lewej i prawej: xmin <= x1 + dx * t <= xmax
Dla krawędzi dolnej i górnej: ymin <= y1 + dy * t <= ymax
```

Co można uprościć do następującej postaci:

```
Lewa: xmin <= (x1 + dx * t)
Prawa: -(x1 + dx * t) >= -xmax
Góra: -(y1 + dy * t) >= -ymax
Dół: ymin <= (y1 + dy * t)
```

Kolejne uproszczenie spowoduje umieszczenie wyrażeń dx ∗ t i dy ∗ t po tej samej stronie nierówności:

```
Lewa: xmin - x1 <= dx * t
Prawa: -dx * t >= -xmax + x1
Góra: -dy * t >= -ymax + y1
Dół: ymin - y1 <= dy * t
```

Po przemnożeniu nierówności przez –1 otrzymamy wzór 10.1:

Wzór 10.1. *Nierówności dla poszczególnych krawędzi przycinania*

```
Nierówność dla lewej krawędzi obszaru przycinania:    n = 0:  - dx*t <= (-xmin + x1)
Nierówność dla prawej krawędzi obszaru przycinania:   n = 1:    dx*t <= ( xmax - x1)
Nierówność dla górnej krawędzi obszaru przycinania:   n = 2:    dy*t <= ( ymax - y1)
Nierówność dla dolnej krawędzi obszaru przycinania:   n = 3:  - dy*t <= (-ymin + y1)
```

Spędziłem naprawdę sporo czasu na szukaniu błędów w prezentowanych tu przekształceniach. Doprawdy, nietrudno pomylić się podczas zamieniania zwrotów nierówności, dlatego nie należy się przy tym spieszyć. Panuje tu oczywiście znana reguła: po przemnożeniu lub podzieleniu obu stron nierówności przez ujemną liczbę należy zmienić dotychczasowy zwrot nierówności na przeciwny.

Zakładając, że każda z nierówności dana jest w postaci pn ∗ t < qn, gdzie n oznacza n-tą nierówność (według numeracji wprowadzonej powyżej), otrzymujemy następujące przypadki:

- Przypadek 1. Jeżeli (pn = 0), odcinek jest równoległy do krawędzi przycinania i nie może jej przecinać;
- Przypadek 2. Jeżeli (pn >= 0), odcinek znajduje się wewnątrz obszaru przycinania;
- Przypadek 3. Jeżeli (pn < 0), odcinek znajduje się na zewnątrz obszaru przycinania;
- Przypadek 4. Jeżeli (pn < 0), odcinek rozpoczyna się poza obszarem przycinania i kończy w jego wnętrzu;
- Przypadek 5. Jeżeli (pn > 0), odcinek rozpoczyna się wewnątrz obszaru przycinania i kończy poza nim;

Nieco to skomplikowane, jednak po dłuższym zastanowieniu się widać, że jedynymi bardziej zajmującymi przypadkami są przypadki 4. i 5. Tylko w tych przypadkach potrzebne jest faktyczne przycięcie odcinka. Dla każdej nierówności trzeba wyznaczyć t punktu przecięcia, pamiętając przy tym, o który przypadek (4. czy 5.) chodzi:

```
pn * t < qn
```

Stąd:

```
t = qn / pn = rn
```

Obliczenie to realizujemy dla każdej z czterech nierówności dla n = 0, 1, 2, 3; wynik każdego z nich określamy jako rn. Tak więc rn jest wartością t w punkcie przecięcia, obliczaną na podstawie wyrażenia qn / pn, reprezentującego jedną z nierówności z wzoru numer 10.1.

Na koniec należy dokonać klasyfikacji wartości rn, aby określić, które z tych wartości oznaczały odcinek wchodzący do obszaru przycinania (pn < 0), a które odcinek wychodzący z tego obszaru (pn > 0). Dla wartości rn obliczanych dla przypadku (pn < 0) obliczamy wartość t1:

```
t1 = max (0, rn)
```

Dla wartości rn obliczanych dla przypadku (pn > 0) obliczamy wartość t2:

```
t2 = min (1, rn)
```

Cóż to oznacza? Jeszcze raz: próbujemy wykonać proste testy odrzucenia odcinka; jeżeli zawiodą, mamy do czynienia z przypadkiem 4. bądź 5. Aby je rozstrzygnąć, należy obliczyć wartości t przecięcia odcinka z krawędziami obszaru przycinania. Dysponujemy czterema równaniami w ogólnej postaci pn ∗ t < qn, z których należy obliczyć wartość t dla wszystkich czterech nierówności wymienionych we wzorze 10.1. Otrzymane wartości t należy przypisać do symboli rn. Następnie wartości rn poddawane są porządkowaniu według kryterium wchodzenia bądź wychodzenia z obszaru przycinania. Na koniec obliczane są wartości t1 i t2, które określają punkty przecięcia, w których należy uciąć ten uparty odcinek!

Jeżeli (t1 > t2), odcinek prostej znajduje się całkowicie poza obszarem przycinania i można zakończyć test. W przeciwnym przypadku konieczne jest przycięcie odcinka przez podstawienie wartości t1 i t2 do parametrycznego równania prostej, co zaowocuje obliczeniem współrzędnych dwóch wierzchołków, *pp1* i *pp2*:

```
pp1 = p1 + (p2 - p1) * t1
pp2 = p1 + (p2 - p1) * t2
```

Wyliczone w ten sposób punkty ograniczają odcinek wewnątrz obszaru przycinania. Należy się zgodzić, że na pierwszy rzut oka algorytm ten jest szalenie skomplikowany, ale jego śledzenie sprawia problem wyłącznie ludziom — komputery radzą sobie z nim znakomicie. Algorytm działa doskonale i jest dużo lepszy niż algorytm Cohena-Sutherlanda, a to z racji bazowania na reprezentacji parametrycznej i łatwości konwersji do przestrzeni trójwymiarowej.

Przykład realizacji algorytmu

Aby utrwalić znajomość algorytmu i upewnić się, że zostanie on zapamiętany i zrozumiany, spróbujmy wyliczyć prosty przykład. Będzie się on opierał na konfiguracji prezentowanej na rysunku 10.12, na którym — głównie w pierwszym kwadrancie układu współrzędnych — widnieje obszar przycinania, nieco przesunięty tak, aby pokrywał również pozostałe kwadranty. Konfiguracja przykładu definiuje następujące elementy:

```
xmin = - 4, xmax = 8
ymin = - 4, ymax = 6

p1 = (-2, -6), p2 = (10, 4)

p = p1 + (p2 - p1) * t

x = -2 + (10 - (-2)) * t = -2 + 12 * t
y = -6 + (4 - (-6)) * t = -6 + 10 * t
```

Stąd dx wynosi 12, a dy 10. Pora na uruchomienie obliczeń dla każdej nierówności:

```
Lewa: -dx * t <= (-xmin + x1)
Prawa: dx * t <= (xmax - x1)
Góra: dy * t <= ( ymax - y1)
Dół: -dy * t <= (-ymin + y1)
```

Po podstawieniu znanych wartości otrzymujemy:

```
Lewa: -12 * t <= (-(-4) + (-2)) = (2)
Prawa: 12 * t <= (8 - (-2)) = (10)
Góra: 10 * t <= ( 6 - (-6)) = (12)
Dół: -10 * t <= (-(-4) + (-6)) = (-2)
```

Po wyodrębnieniu pn i qn i obliczeniu dla każdego równania wartości rn:

```
Lewa: -12 * t <= (-(-4) + (-2)) = (2), p0 = -12, q0 = 2, r0 = (q0 / p0) = -1 / 6
Prawa: 12 * t <= (10) , p1 = 12, q1 = 10, r1 = (q1 / p1) = 5 / 6
Góra: 10 * t <= (12) , p2 = 10, q2 = 12, r2 = (q2 / p2) = 6 / 5
Dół: -10 * t <= (-2) , p3 = -10, q3 = -2, r3 = (q3 / p3) = 1 / 5
```

Teraz obliczenia należy rozdzielić na dwa przypadki — pierwszy dla (pn < 0); dla tego przypadku obliczamy wartość t1:

```
t1 = max (0, rn) = max (0, -1 / 6, 1 / 5) = 1 / 5
```

Dla wartości t(rn) w przypadku, gdy odcinek skierowany był na zewnątrz obszaru (pn > 0), obliczana jest wartość t2:

```
t2 = min(1, rn) = min(1, 5 / 6, 6 / 5) = 5 / 6
```

Uff! Teraz trzeba sprawdzić, czy przypadkiem t1 jest większe niż t2. Jeżeli tak, to odcinek leży w całości poza obszarem przycinania. Z samego rysunku widać, że taki przypadek nie miał miejsca, a i wyliczenia mówią, że t1 jest mniejsze od t2. Trzeba więc jeszcze podstawić obliczone wartości t1 i t2 do parametrycznych równań prostej — obliczone punkty wyznaczają nową, przyciętą do obszaru, prostą (odcinek)! Po podstawieniu t1 otrzymujemy:

```
x1' = -2 + 12 * (1 / 5) = -2 + 2.4 = 0.4
y1' = -6 + 10 * (1 / 5) = -6 + 10 / 6 = -4.0
```

a po podstawieniu obliczonego t2:

```
x2' = -2 + 12 * (5 / 6) = -2 + 10 = 8
y2' = -6 + 10 * (5 / 6) = -6 + 50 / 6 = 2.3
```

Uzyskany w efekcie przycinania odcinek prostej rozpięty jest pomiędzy punktami *p0'*(0,4; –4,0) i *p1'*(8,0; 2,3). Można to sprawdzić z punktami przycięcia na rysunku 10.12 (powinny się zgadzać!).

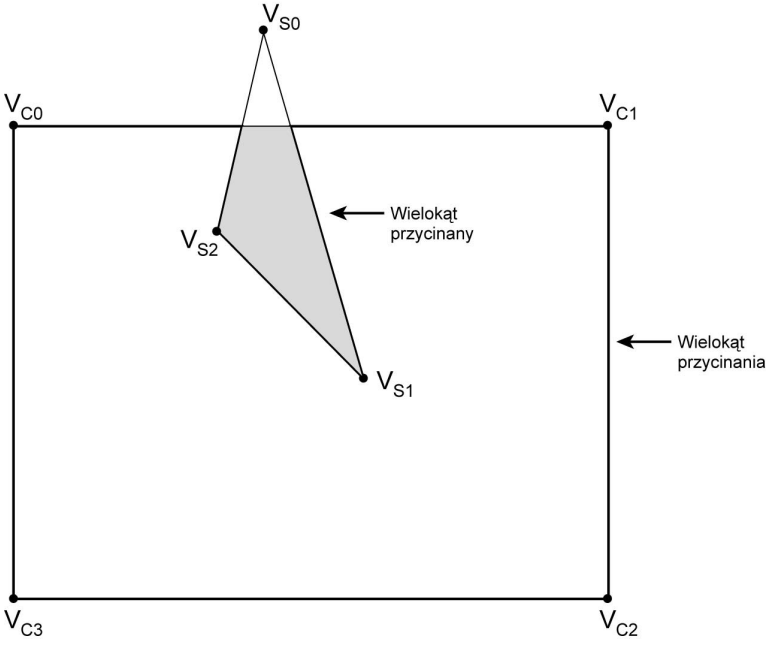
Jeżeli zastanowić się przez chwilę nad tym algorytmem, okaże się, że tak naprawdę pomaga on wyłącznie w bardziej eleganckiej klasyfikacji i organizowaniu fragmentów informacji o scenie. Jak na ironię, w strukturze kodu wielu modułów przycinających pisanych przez osoby nie znające tego algorytmu znaleźć można identyczne przepływy danych, co w sumie oznacza, że algorytm ten jest naprawdę dobrze przemyślany i uniwersalny.

W bieżącym rozdziale nie będziemy implementować tego algorytmu, ale może później, kiedy dojdziemy do optymalizacji przycinania, zastąpimy algorytm Cohena-Sutherlanda dla płaszczyzn prezentowanym tu algorytmem Cyrusa-Becka. Tu nasuwa się wątpliwość, czy ten ostatni nadaje się do przycinania w przestrzeni trójwymiarowej? Otóż tak. Przystosowanie tego algorytmu do trzech wymiarów nie zmienia zasadniczo procedury obliczeniowej, tyle, że nierówności określają wtedy, czy punkt leży w dodatniej, czy w ujemnej półprzestrzeni płaszczyzn przycinania. Algorytm w wersji 3D angażuje więc iloczyny skalarne wektorów i operację sprawdzania półprzestrzeni.

Algorytm Weilera-Athertona

Ostatni z algorytmów omawianych w tej części rozdziału wykorzystuje fakt, że wielokąty sceny złożone są z wierzchołków; każdy wierzchołek może należeć do przycinanego obszaru bądź znajdować się poza nim. Algorytm zaprojektowany jest pod kątem przycinania jednego wielokąta do innego wielokąta. Jeżeli jeden z wierzchołków wielokąta znajduje się poza obszarem przycinania, krawędzie prowadzone od tego

wierzchołka dzielą się na dwie kategorie: na wychodzące z obszaru przycinania i do niego wkraczające. Na rysunku 10.13 wielokąt poddawany operacji przycinania nazywany jest *wielokątem przycinanym*, a obszar przycinania — *wielokątem przycinania*.



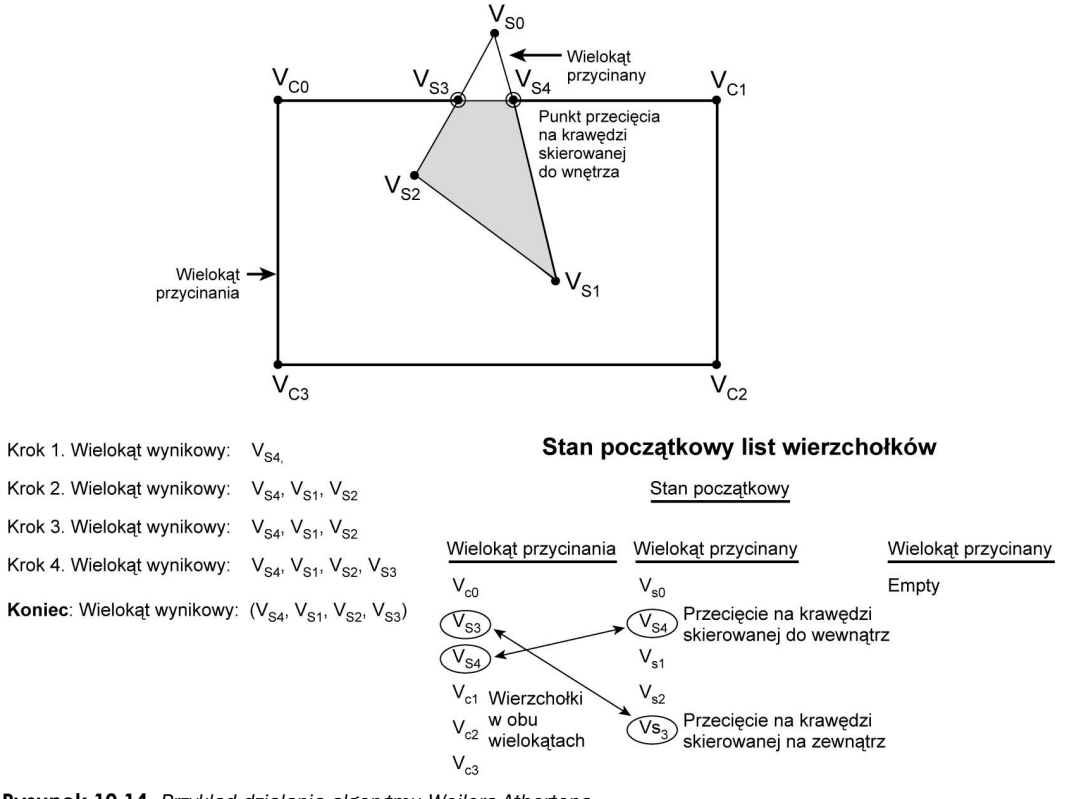
Rysunek 10.13. *Konfiguracja wielokątów ilustrująca działanie algorytmu Weilera-Athertona*

Przygotowanie sceny do przetwarzania algorytmem Weilera-Athertona polega na takim zdefiniowaniu list bądź tablic wierzchołków wielokątów, aby wierzchołki te były uporządkowane w pewien określony sposób. Załóżmy, że wierzchołki wszystkich wielokątów uporządkowane są zgodnie z kierunkiem ruchu wskazówek zegara. Algorytm startuje od wierzchołka zerowego wielokąta przycinanego i wyznacza przecięcia z każdą z krawędzi wielokąta przycinania. Jeżeli uda się znaleźć punkt przecięcia, jest on dodawany zarówno do listy wierzchołków wielokąta przycinanego, jak i do listy wierzchołków wielokąta przycinania (pomiędzy wierzchołkami tworzącymi przecinaną krawędź) i opatrywany znacznikiem krawędzi wychodzącej bądź wkraczającej. Znacznik ten informuje, czy krawędź wielokąta przycinanego, zawierająca punkt przecięcia, była skierowana na zewnątrz, czy też do wewnątrz wielokąta przycinania. Po uzupełnieniu listy wierzchołków generowany jest wielokąt wynikowy przycinania. Tworzy się go w sześciu krokach:

1. Wyszukanie pierwszego punktu przecięcia na liście wierzchołków wielokąta przycinanego — takiego, dla którego ustawiony jest znacznik krawędzi skierowanej do wewnątrz wielokąta przycinania. Punkt ten będzie pierwszym wierzchołkiem wielokąta wynikowego.
2. Przeszukanie listy wierzchołków wielokąta przycinanego aż do znalezienia następnego punktu przecięcia krawędzi wielokąta przycinania. Każdy odwiedzany po drodze punkt dodawany jest do listy wierzchołków wielokąta wynikowego. Drugi odnaleziony punkt przecięcia krawędzi powinien oznaczać krawędź skierowaną na zewnątrz obszaru przycinania.
3. Przeszukanie listy wierzchołków wielokąta przycinania w poszukiwaniu punktu przecięcia identycznego z ostatnio znalezionym punktem na liście wierzchołków wielokąta przycinanego (synchronizacja list wierzchołków).
4. Przeglądanie wielokąta przycinania; każdy odwiedzony kolejno wierzchołek tego wielokąta dodawany jest do listy wierzchołków wielokąta wynikowego. Przeglądanie odbywa się do momentu odnalezienia kolejnego punktu przecięcia. Będzie to punkt przecięcia na krawędzi skierowanej do wewnątrz.

5. Jeżeli znaleziony w kroku 4. punkt przecięcia nie identyczny z jest punktem początkowym wielokąta wynikowego, powrót do kroku 2. (próba zamknięcia wielokąta wynikowego).
6. Jeżeli punkt przecięcia znaleziony w kroku 4. jest identyczny z pierwszym punktem wielokąta wynikowego, wielokąt ten jest już zamknięty, ale niekoniecznie kompletny. Jeżeli wszystkie punkty przecięcia na krawędziach skierowanych do wnętrza wielokąta przycinania wchodzą w skład wielokąta wynikowego, algorytm kończy działanie (do stwierdzenia tego faktu można wykorzystać licznik albo znacznik); w przeciwnym przypadku należy przejść do następnego punktu przecięcia do wewnątrz wielokąta przycinania i przejść do kroku 2.

Zaprezentowany algorytm działa znakomicie i daje sobie radę z przycinaniem dowolnych wielokątów do dowolnych innych wielokątów. Przykładowy przebieg algorytmu zastosowanego do prostego przypadku przycinania ilustruje rysunek 10.14.

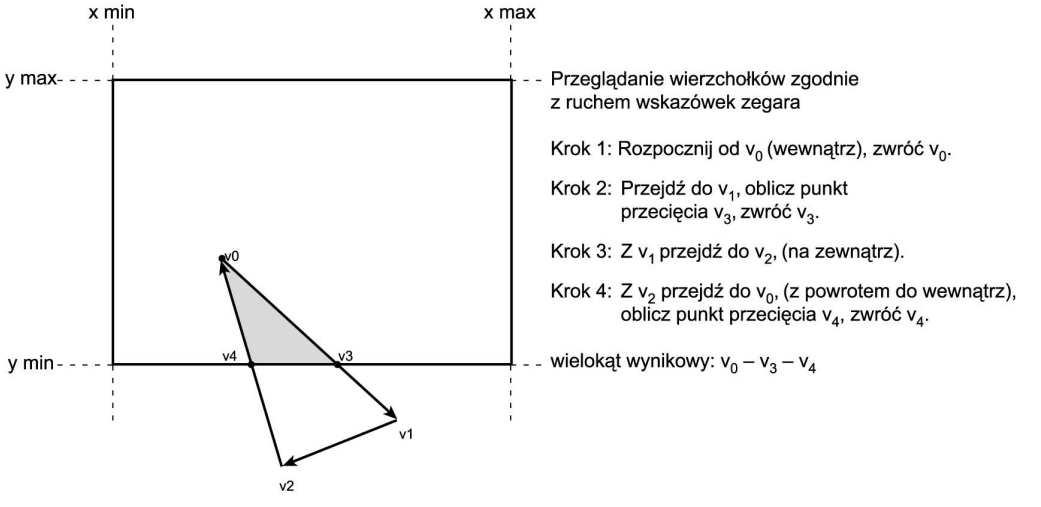


Rysunek 10.14. *Przykład działania algorytmu Weilera-Athertona*

Algorytm wymagający specjalnego przygotowania struktur reprezentujących scenę i zakładający wielokrotne przeglądanie list wierzchołków nie jest, co prawda, najdoskonalszy. Co gorsza, operując na poszczególnych trójkątach nie możemy pozwolić sobie na zbyt długie algorytmy. W większości przypadków potrzebny będzie więc algorytm skonstruowany samodzielnie, wykorzystujący elementy wszystkich prezentowanych algorytmów i — co najważniejsze — zoptymalizowany do konkretnego zadania. Przedtem jednak jeszcze jedna prezentacja algorytmu — wariantu algorytmu Weilera-Athertona.

Zastanówmy się nad sytuacją, w której jeden wielokąt (trójkąt) ma zostać przycięty do pojedynczej prostej lub płaszczyzny (aby wykonać pełne przycinanie, można by powtarzać procedurę dla kolejnych prostych i kolejnych płaszczyzn tworzących obszar przycinania). Rozpoczynamy od klasyfikacji wierzchołka początkowego jako znajdującego się wewnątrz lub na zewnątrz obszaru przycinania. W pierwszym przypadku

wierzchołek zwracany jest jako jeden z wierzchołków wynikowych trójkąta, w drugim następuje przejście do następnego wierzchołka. Jeżeli punkty końcowe krawędzi zostaną różnie sklasyfikowane (jeden jako leżący wewnątrz, drugi jako znajdujący się poza obszarem przycinania), wyliczamy punkt przecięcia i zwracamy go jako wierzchołek wielokąta wynikowego. Zasadniczo więc algorytm polega na przeglądaniu kolejnych krawędzi wielokąta. Jeżeli uda się rozpocząć wewnątrz obszaru przycinania, sprawa jest prosta: każdorazowo po jego opuszczeniu wyliczamy punkt przecięcia, zwracamy go jako wierzchołek wielokąta wynikowego i kontynuujemy przegląd krawędzi. Kiedy dojdziemy do krawędzi wracającej do obszaru przycinania, znów obliczamy punkt przecięcia, zwracamy go jako wierzchołek wielokąta wynikowego i kontynuujemy przegląd. To bardzo uproszczona wersja algorytmu Weilera-Athertona, ale za to bardzo szybka. Osobiście stosuję ją — w bardziej lub mniej modyfikowanych postaciach — w większości projektów, ponieważ uważam, że nie warto wysilać się tworzeniem złożonej listy wierzchołków wielokąta przycinania w przypadku, gdy obszar przycinania to zwykły prostokąt, ani wielokąta przycinanego, gdy wiadomo, że to zawsze trójkąt. Działanie wersji uproszczonej algorytmu ilustruje rysunek 10.15.



Rysunek 10.15. *Uproszczona wersja algorytmu Weilera-Athertona*

Przycinanie — dodatkowe źródła informacji

To byłoby na tyle! Teraz wiecie już, jak realizować przycinanie. Doprawdy, nie jest to zbyt skomplikowana dziedzina — korzysta się z gotowego algorytmu bądź konstruuje algorytm hybrydowy, dostosowany do specyficznych potrzeb, a wszystkie one polegają w zasadzie na wyszukiwaniu punktów przecięć z dwuwymiarowymi obszarami bądź trójwymiarowymi bryłami. Teraz zabierzemy się do implementacji mechanizmu przycinania dla potrzeb naszego silnika graficznego, ale jeżeli ktoś zainteresowany jest pogłębieniem swojej wiedzy o algorytmach przycinania, może zajrzeć do jednej z wymienionych poniżej książek. W pozycjach tych znaleźć można całe piękno podejścia akademickiego — choć nie nadaje się ono do implementowania algorytmów graficznych realizowanych w czasie rzeczywistym, przynajmniej dokładnie zapoznaje z ich wszelkimi matematycznymi niuansami. Oto one:

- Foley, van Dam, Feiner i Hughes, *Computer Graphics: Principles and Practice* (wydawnictwo Addison-Wesley).
- Foley, van Dam, Feiner i Hughes, *Introduction to Computer Graphics* (wydawnictwo Addison-Wesley).
- Alan H. Watt, *3D Computer Graphics* (wydawnictwo Addison-Wesley).
- David F. Rogers, *Procedural Elements for Computer Graphics* (wydawnictwo McGraw-Hill).

Miłej lektury!

Przycinanie do ostrosłupa widzenia — przykład implementacji

Cóż, pora zabrać się za implementację przycinania w naszym silniku 3D! Zadanie polegać będzie na utworzeniu dodatkowego etapu potoku przetwarzania, w którym wielokąty będą przycinane do bryły ostrosłupa widzenia. Jak wiadomo, przycinaniu poddawane będą wielokąty reprezentowane w przestrzeni trójwymiarowej. Moduł przycinania powinien minimalizować liczbę koniecznych wtórnych podziałów wielokątów i przycinać wyłącznie te trójkąty, które tego koniecznie wymagają.

Kiedy trójkąt musi zostać przycięty? Otóż jedynie wtedy, kiedy wykracza poza bliską płaszczyznę przycinania, albo, co gorsza, przenika przez płaszczyznę o współrzędnej z równej 0. Rzutowanie wielokątów o współrzędnych z mniejszych lub równych zeru powoduje bowiem błędy dzielenia i odwrócenie rzutów w płaszczyźnie (x, y), a takie błędy są nie do przyjęcia.

Aż do tego momentu oszukiwaliśmy trochę skrzeczącą rzeczywistość, umieszczając bliską płaszczyznę przycinania ostrosłupa widzenia na tyle daleko od płaszczyzny rzutu i płaszczyzny z = 0, aby żaden z trójkątów sceny nie mógł przez nie przenikać, ponieważ obiekty były odrzucane na długo przedtem, zanim miały szansę przekroczyć płaszczyznę z = 0. Teraz jednak trzeba zapomnieć o usuwaniu całych obiektów i przestawić się na myślenie o scenie wyłącznie w kategorii listy wielokątów. Wielokąty te będą stanowić problem, ponieważ będą mogły znajdować się wszędzie.

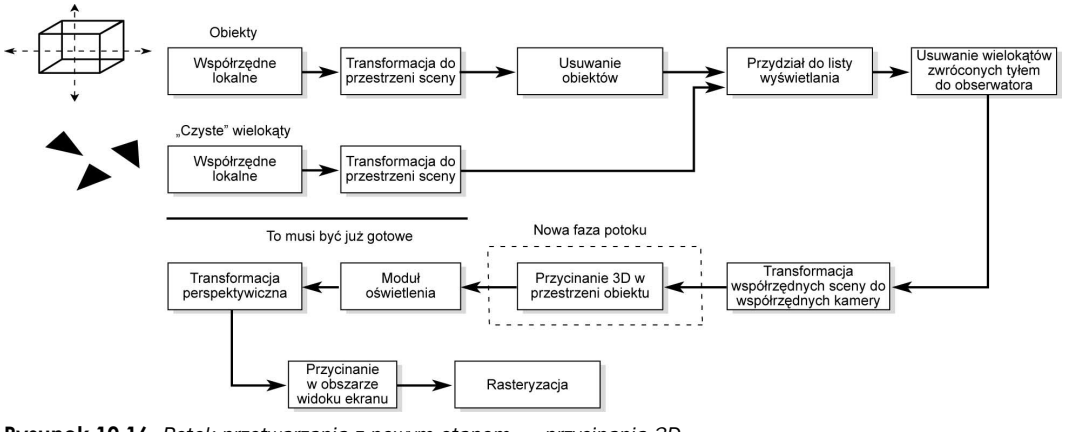
Właściwie chodzić będzie więc o zaprogramowanie modułu przycinającego (usuwającego) takiego, który przycinałby wielokąty z listy obrazowanych wielokątów do bryły ściętego ostrosłupa widzenia — tyle, że rzeczywiste przycinanie wierzchołków wielokątów będzie zachodzić wyłącznie na bliższej płaszczyźnie przycinania. Wielokąty, które po części wystawać będą poza górną, dolną czy jedną z bocznych płaszczyzn ostrosłupa widzenia, będą odrysowywane w całości, zaś te, które znajdą się poza tymi płaszczyznami, będą w całości usuwane z potoku przetwarzania. Naturalnie wielokąty zakwalifikowane do odrysowania mogą przez to wystawać poza obszar rzutu ekranowego, ale zostaną do niego przycięte w rasteryzatorze. Wiemy już, że taka strategia oznacza szybszą i mniej kłopotliwą implementację modułu przycinającego — nie trzeba przycinać wielokątów do każdej z sześciu ścian bryły ostrosłupa widzenia, ponieważ w ostatecznym rozrachunku i tak okazuje się to marnotrawstwem czasu. Wymaga przy tym kłopotliwych operacji wtórnego podziału wielokątów, przeliczania wektorów normalnych wierzchołków i płaszczyzn oraz współrzędnych mapowania tekstur. Jeszcze raz: jedynymi wielokątami koniecznie wymagającymi przycięcia są wielokąty przenikające i wystające poza bliską płaszczyznę przycinania, których dalsze przetwarzanie może potencjalnie doprowadzić do zakłóceń obrazowania sceny lub wyjątków procesora. Zaletą przyjętej strategii jest zaś nie tylko wyeliminowanie wielokątów zwróconych tyłem do obserwatora, ale również usunięcie tych, które znajdują się w całości poza bryłą ostrosłupa widzenia. Z potoku renderowania usunięta zostanie więc znaczna część elementów sceny, co oznacza naturalnie większą wydajność silnika! Zaczynamy!

Przycinanie 3D można umieścić w różnych fazach potoku przetwarzania: w przestrzeni sceny, przestrzeni obserwatora bądź przestrzeni rzutu perspektywicznego. W każdej z tych przestrzeni ujawnia ono odmienne wady i zalety. Przykładowo, przycinanie w przestrzeni sceny pozwala na zaoszczędzenie czasu potrzebnego w innym przypadku na realizację przekształceń potencjalnie niewidocznych elementów sceny do współrzędnych przestrzeni obserwatora, ale wtedy ostrosłup widzenia zakotwiczony jest w pozycji obserwatora i posiada ustaloną orientację. Tak więc przycinanie i usuwanie fragmentów sceny powinno być realizowane na nieco mniej ogólnym poziomie. Z drugiej strony, jeśli dopuścimy wielokąty do przekształcenia z przestrzeni sceny w przestrzeń kamery, to zachowane zostanie ich położenie względne wobec kamery, a płaszczyzny ostrosłupa widzenia przebiegać będą wzdłuż osi widzenia, co bardzo uprości przycinanie. Tak więc w drugim przypadku konieczna będzie każdorazowo pełna transformacja współrzędnych sceny do współrzędnych kamery, ale za to proces przycinania (usuwania) będzie dużo prostszy. Dalej, gdyby poczekać z przycinaniem aż do fazy rzutu perspektywicznego, cała geometria rzutowanej sceny będzie znormalizowana do kanonicznej bryły widzenia, którą stanowi prostopadłościan. Innymi słowy, jeżeli przed przycinaniem zrealizowane zostaną przekształcenia perspektywiczne, nie trzeba będzie przycinać do ostrosłupa, ale do prostopadłościanu — wałkowaliśmy to już kilkukrotnie w rozdziałach o podstawach matematycznych rzutowania 3D.

Cóż więc robić? Na pewno zapomnieć o wariancie przycinania po przekształceniach perspektywicznych — tak długie odwlekanie przycinania to nie jest najlepszy pomysł. W takim wariancie nie tylko będziemy przeliczać oświetlenie dla wielokątów niewidocznych; najgorsze, że potok przetwarzania wypełniony będzie dwukrotnie większą niż potrzebna liczbą wielokątów, z których większość nie zostanie w ogóle wyświetlona. Można by więc przycinać w przestrzeni sceny, ale osobiście nie podoba mi się konieczność ciągłego przekształcania geometrii ostrosłupa widzenia. Ponadto, w takim scenariuszu przekształcane do przestrzeni kamery będą również wszystkie te dodatkowe wielokąty, które powstaną z podziału na granicach przycinania. Przypuśćmy, że scena zawiera 1 000 wielokątów. Po przycięciu może się okazać, że jest ich 1 200 (to, rzecz jasna, wyłącznie szacunki). Do przestrzeni kamery (obserwatora) trzeba więc będzie przekształcić 1 200 wielokątów! Co gorsza, sprawdzanie pod kątem przycinania każdego wielokąta sceny jest nieco bardziej skomplikowane, niż to samo sprawdzanie realizowane względem współrzędnych kamery, kiedy obserwator znajduje się w centrum układu współrzędnych, a ostrosłup widzenia jest wyrównany wzdłuż dodatnich osi układu.

Dla odmiany przypuśćmy jednak, że po przycinaniu w przestrzeni sceny udało się z niej usunąć mniej więcej połowę wielokątów. Wtedy do przekształcenia do przestrzeni kamery zostanie tylko 500 wielokątów — jak widać, decyzja jest trudna. Zdecydowałem jednak o dokonywaniu przycinania po przekształceniu sceny do współrzędnych kamery, ponieważ taki schemat łatwo zrozumieć i prześledzić, a jakoś nie przekonuje mnie wizja zaoszczędzenia większości przekształceń w przestrzeni sceny. Decyzję tę będzie można zresztą poddać rewizji na końcu niniejszej książki, gdy przyjdzie do optymalizowania silnika graficznego.

Podsumowując: będziemy przeprowadzać przycinanie po przekształceniu świata sceny do współrzędnych obserwatora, usuwając ze sceny wszystkie te wielokąty, które znajdą się w całości poza ostrosłupem widzenia i przycinając te które, znajdą się na bliskiej płaszczyźnie przycinania. Częściowe wykraczanie poza dolną, górną, lewą bądź prawą ścianę ostrosłupa widzenia będzie tolerowane, ponieważ wystające trójkąty zostaną przycięte na etapie rasteryzacji. Nowa wersja potoku przetwarzania widnieje na rysunku 10.16.



Rysunek 10.16. *Potok przetwarzania z nowym etapem — przycinania 3D*

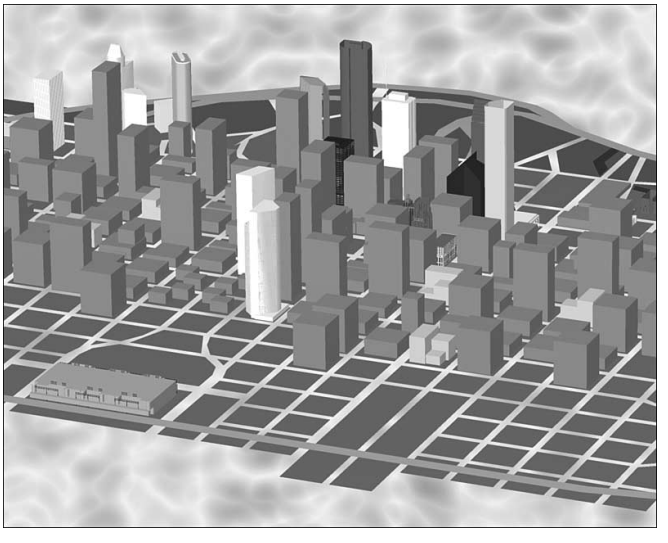
Potok przekształceń geometrycznych i nowe struktury danych

Na rysunku 10.16 widać, że geometria sceny jest wczytywana do systemu za pośrednictwem całych obiektów (ewentualnie generowana za pomocą pojedynczych wielokątów), po czym obiekty są przekształcane, ewentualnie usuwane, dzielone na wielokąty, następnie usuwane są te wielokąty, które są zwrócone tyłem do obserwatora; dalej znowu mamy nieco przekształceń, przycinanie, usuwanie, znowu przekształcenia i wreszcie wyświetlanie. Strasznie dużo tu przekształceń współrzędnych! Rzecz w tym, że potok rozpoczyna się od obiektów — sprawdza się to w grach bez geometrii wnętrz i krajobrazów, jednak w grach rozgrywanych wewnątrz modelowanych pomieszczeń — już nie.

Trzeba się więc przestawić z podejścia obiektowego i zacząć myśleć kategoriami nowych elementów, bardziej odpowiednich do reprezentowania podłoża terenowego, wnętrz budynków i tym podobnych. Właśnie dlatego dotychczas stosowaliśmy jak najmniej wywołań modułów oświetlenia i usuwania obiektów. Od tego momentu należy zamienić pojęcie obiektu na pojęcie zbioru wielokątów, ponieważ już wkrótce do silnika wprowadzone zostaną bardziej zaawansowane struktury danych, obsługujące wielkie obiekty sceny. Nie chcielibyśmy jednak, aby te struktury były po prosu jednym wielkim, bezładnym zbiorowiskiem wielokątów. Wielokąty te mają być uporządkowane, posortowane, wzajemnie połączone i umieszczone w strukturach hierarchicznych.

Weźmy za przykład rysunek 10.17. Obrazuje on w pewnej grze widok zewnętrzny miasta, który chcielibyśmy modelować graficznie. Choć dane poziomu mogą zostać zebrane w kontenerze obiektowym, przydałyby się również struktury pomocnicze, pozwalające na szybkie usunięcie poszczególnych grup elementów składowych sceny. Tu na scenę wkraczają koncepcje drzew BSP, portali, drzew czwórkowych i im podobne.

Rysunek 10.17.
Widok poziomu gry



Aby jednak efektywnie wykorzystać wszystkie te struktury, musimy dysponować sposobem zamienienia ich w pewnym momencie na zbiór „gołych" wielokątów. Zmierzamy więc do tego, że przed implementacją należy podjąć decyzję co do typu gry czy gier, które mają wykorzystywać dany silnik i przyjąć do wiadomości, że w przypadku gier z geometrią wnętrz przycinanie (podobnie, jak usuwanie) wielokątów jest jednym z ważniejszych etapów potoku przetwarzania. Dlatego właśnie zdecydowaliśmy się na wprowadzenie do potoku modułu przycinania i usuwania wielokątów z list wyświetlania zamiast realizować analogiczne operacje na rzecz obiektów gry. Nie można z góry zakładać, że wszystko uda się umieścić w strukturach obiektowych.

Dodawanie przycinania do silnika graficznego

Zacznijmy od początku. Nowy moduł biblioteczny, zawierający kod przycinania, będzie nosił nazwę *T3DLIB8.CPP* (jego plik nagłówkowy to *T3DLIB8.H*). Będzie bardzo krótki, jako że jego zawartość ograniczona zostanie dosłownie do kilku funkcji. Jak zwykle, moduł zostanie poddany krótkiej analizie, bez zagłębiania się w szczegóły i analizowania przyczyn pojawiania się kolejnych funkcji. Przyjrzyjmy się najpierw poszczególnym częściom pliku nagłówkowego.

Plik nagłówkowy

Moduł przycinania wykorzystuje zaledwie kilka stałych, kontrolujących jego zachowanie i położenie płaszczyzn przycinania. Oto definicje tych stałych:

```
// ogólne stałe dla przycinania wielokątów
#define CLIP_POLY_X_PLANE     0x0001 // usuwanie osi x płaszczyzny przycinania
#define CLIP_POLY_Y_PLANE     0x0002 // usuwanie osi y płaszczyzny przycinania
#define CLIP_POLY_Z_PLANE     0x0004 // usuwanie osi z płaszczyzny przycinania
#define CLIP_OBJECT_XYZ_PLANES    (CULL_OBJECT_X_PLANE | CULL_OBJECT_Y_PLANE | \
        CULL_OBJECT_Z_PLANE)
```

W bibliotece znajdują się jedynie dwie funkcje: funkcja przycinania i funkcja generowania terenu (ta ostatnia jest pewną niespodzianką):

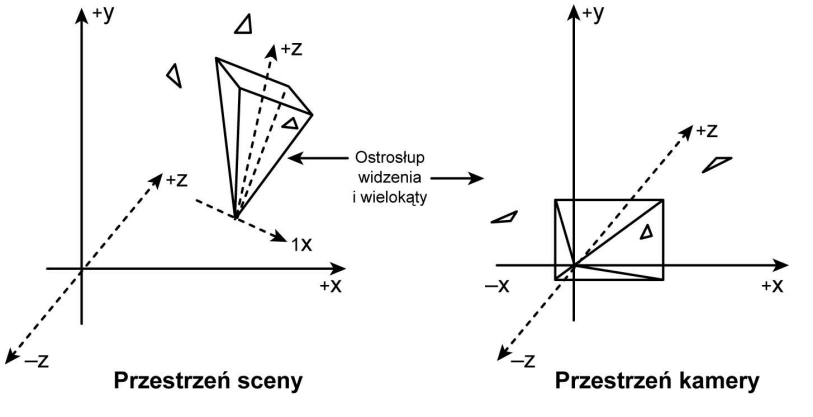
```
void Clip_Polys_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                               // lista wyświetlania do przycięcia
  CAM4DV1_PTR cam,               // kamera
  int clip_flags);               // znaczniki przycinania

int Generate_Terrain_OBJECT4DV2(OBJECT4DV2_PTR obj,   // wskaźnik do obiektu
  float twidth,           // szerokość terenu we współrzędnych x sceny
  float theight,          // wysokość (długość) terenu we współrzędnych y sceny
  float vscale,           // skala pionowa terenu
  char *height_map_file,  // nazwa pliku mapy wysokości kodowanej na 256 kolorach
  char *texture_map_file, // nazwa pliku tekstury terenu
  int rgbcolor,           // kolor terenu (jeżeli brak tekstury)
  VECTOR4D_PTR pos,       // początkowe położenie
  VECTOR4D_PTR rot,       // początkowy kąt obrotu
  int polu_attr);         // atrybuty cieniowania
```

Nic tu skomplikowanego; przejdźmy więc do omówienia działania funkcji przycinania.

Konstruowanie modułu przycinania

Funkcja przycinania musi operować na liście wyświetlania we współrzędnych kamery, co oznacza, że ostrosłup widzenia będzie zakotwiczony w początku układu współrzędnych, a płaszczyzny przycinania będą prostopadłe do dodatniej półosi z (patrz rysunek 10.18).



Rysunek 10.18. *Przycinanie w przestrzeni kamery*

Poniżej znajduje się standardowy kod poszczególnych etapów przetwarzania w bieżącej postaci potoku wyświetlania:

```
Reset_OBJECT4DV2(&obj);

// konstrukcja macierzy jednostkowejMAT_IDENTITY_4X4(&mtrans);

// przekształcenia lokalnego układu współrzędnych
Transform_OBJECT4DV2(&obj, &mtrans, TRANSFORM_LOCAL_TO_TRANS, 1);

// przekształcenie do współrzędnych sceny
Model_To_World_OBJECT4DV2(&obj, TRANSFORM_TRANS_ONLY);
```

```
// wstawienie obiektu do listy wyświetlania
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, &obj, 0);

// usunięcie wielokątów zwróconych tyłem do obserwatora
Remove_Backfaces_RENDERLIST4DV2(&rend_list, &cam);

// przekształcenie do współrzędnych kamery
World_To_Camera_RENDERLIST4DV2(&rend_list, &cam);

//////////////////////////////////////////////////////
// W tym miejscu należałoby przyciąć wielokąty do bryły ostrosłupa widzenia
//////////////////////////////////////////////////////

// oświetlenie sceny
Light_RENDERLIST4DV2_World16(&rend_list, &cam, lights, 4);

// sortowanie listy wielokątów
Sort_RENDERLIST4DV2_World16(&rend_list, SORT_POLYLIST_AVGZ);

// przekształcenie perspektywiczne
Perspective_To_Screen_RENDERLIST4DV2(&rend_list, &cam);

// zablokowanie bufora ekranu
DDraw_Lock_Back_Surface();

// rysowywanie wielokątów
Draw_RENDERLIST4DV2_Solid16(&rend_list, back_buffer, back_lpitch);

// zwolnienie bufora ekranu
DDraw_Unlock_Back_Surface();
```

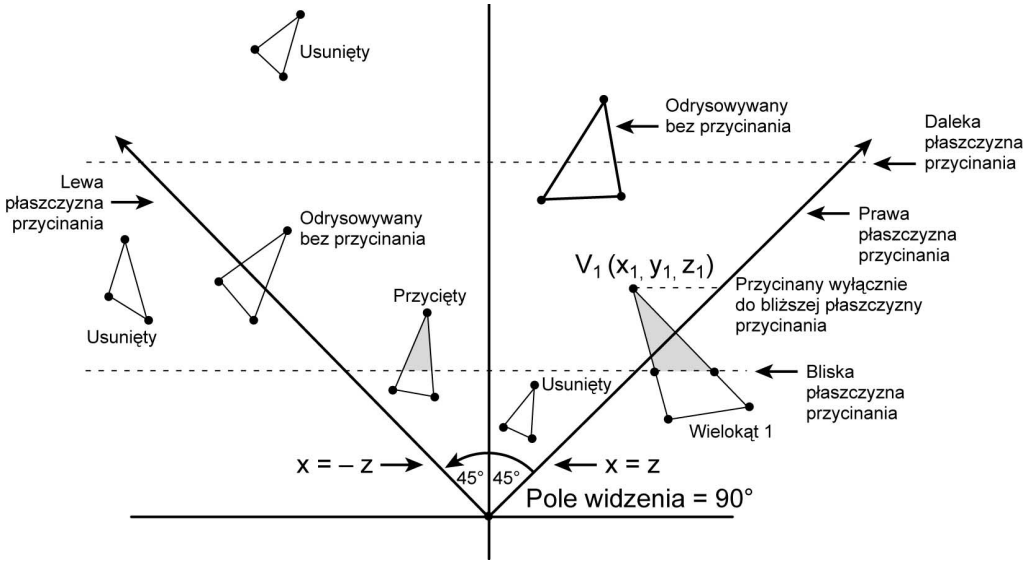
Na wydruku wyróżniono komentarzem miejsce, w którym należałoby przeprowadzić operację przycinania: po przekształcaniu współrzędnych sceny do przestrzeni kamery, ale jeszcze przed obliczeniami oświetlenia. Ten moment jest idealny, ponieważ po przycięciu wielokątów dzięki mniejszej ich liczbie zaoszczędzi się na obliczeniach oświetlenia tych z nich, które i tak nie zostałyby narysowane w buforze ramki obrazu.

Zastanówmy się teraz nad tym, co będzie potrzebne do przycięcia wielokątów z listy wyświetlania. Lista wyświetlania składa się z dwóch struktur danych: z tablicy wielokątów i z tablicy wskaźników do wielokątów. Tablica wskaźników wykorzystywana jest, rzecz jasna, do szybkiego sortowania tablicy z pominięciem czasochłonnych operacji zwalniania i ponownego przydziału pamięci.

Przycinanie do górnej, dolnej i bocznych płaszczyzn przycinania

Zgodziliśmy się już, że nie warto przycinać wielokątów wystających częściowo poza górną, dolną, lewą bądź prawą płaszczyznę ostrosłupa widzenia. Należy jednak sprawdzić, czy badany wielokąt nie znajduje się przypadkiem w całości poza tymi płaszczyznami, albo też czy w całości znajduje się wewnątrz ostrosłupa widzenia. W tym drugim przypadku wielokąt jest kwalifikowany do dalszego przetwarzania w potoku. Wielokąty znajdujące się w całości poza ostrosłupem widzenia są zaś w całości usuwane z listy wyświetlania (bez przycinania), jak to pokazano na rysunku 10.19.

Aby zrealizować powyższe założenia, musimy jedynie znać wierzchołki trójkąta testowanego pod kątem przenikania przez górną, dolną lub jedną z bocznych płaszczyzn przycinania oraz pozycję i orientację owych płaszczyzn. Gdybyśmy zdecydowali się na przycinanie w przestrzeni sceny, najlepiej byłoby wykorzystać do tego celu równania płaszczyzn w postaci wektorowej i wykonywać iloczyn skalarny z każdym wierzchołkiem w celu obliczenia wartości półprzestrzeni wierzchołka względem płaszczyzny. Jeżeli wszystkie trzy wierzchołki miałyby identyczną wartość półprzestrzeni (czyli trójkąt znajdowałby się w całości po jednej ze stron płaszczyzny), trójkąt można by w całości przyjąć bądź w całości odrzucić.



Rysunek 10.19. *Kwalifikacja wielokątów do przycinania*

Możemy jednak rzecz nieco uprościć, wykorzystując wiadomy nam fakt, że płaszczyzny przycinania są równoległe i prostopadłe do osi układu współrzędnych — dysponując taką wiedzą, możemy po prostu do stwierdzenia, czy punkt znajduje się w ostrosłupie widzenia, wykorzystać nachylenie płaszczyzn, czyli kąt pola widzenia. Spójrzmy raz jeszcze na rysunek 10.19. Widniejący na nim *wielokąt 1* jest sprawdzany względem prawej płaszczyzny przycinania, tworzącej ostrosłup widzenia. Możemy na chwilę zignorować oś y układu i rozpatrzyć ten przypadek wyłącznie w płaszczyźnie x-z. Pytanie brzmi: czy wierzchołek V_1(x1, y1, z1) znajduje się przed, czy za prawą płaszczyzną przycinania? Można to stwierdzić na podstawie podobieństwa trójkątów.

Na rysunku 10.19 kąt widzenia został określony jako 90 stopni, czyli lewa i prawa płaszczyzna ostrosłupa widzenia jest nachylona względem płaszczyzny z o 45 stopni. Równanie tych płaszczyzn można więc zdefiniować po prostu jako x = –z dla płaszczyzny lewej i x = z dla prawej (wartość y jest obojętna). Oznacza to, że jeżeli współrzędna x1 badanego wierzchołka jest równa jego współrzędnej z1, wierzchołek ten znajduje się na płaszczyźnie przycinania; jeżeli x1 > z1, wierzchołek znajduje się na zewnątrz płaszczyzny, a jeżeli x1 < z1, wierzchołek położony jest po jej stronie wewnętrznej.

Terminy „wewnątrz" i „strona wewnętrzna" oznaczają tu przestrzeń wewnątrz ostrosłupa widzenia, inaczej dodatnią półprzestrzeń płaszczyzny przycinania. Analogicznie terminy „na zewnątrz" i „po stronie zewnętrznej" oznaczają przestrzeń poza ostrosłupem widzenia, a więc w ujemnej półprzestrzeni płaszczyzny przycinania.

Możemy to uogólnić do postaci następującego twierdzenia: dla danego wierzchołka *v*(x, y, z), przy kącie widzenia równym 90 stopni, zachodzą następujące zależności:

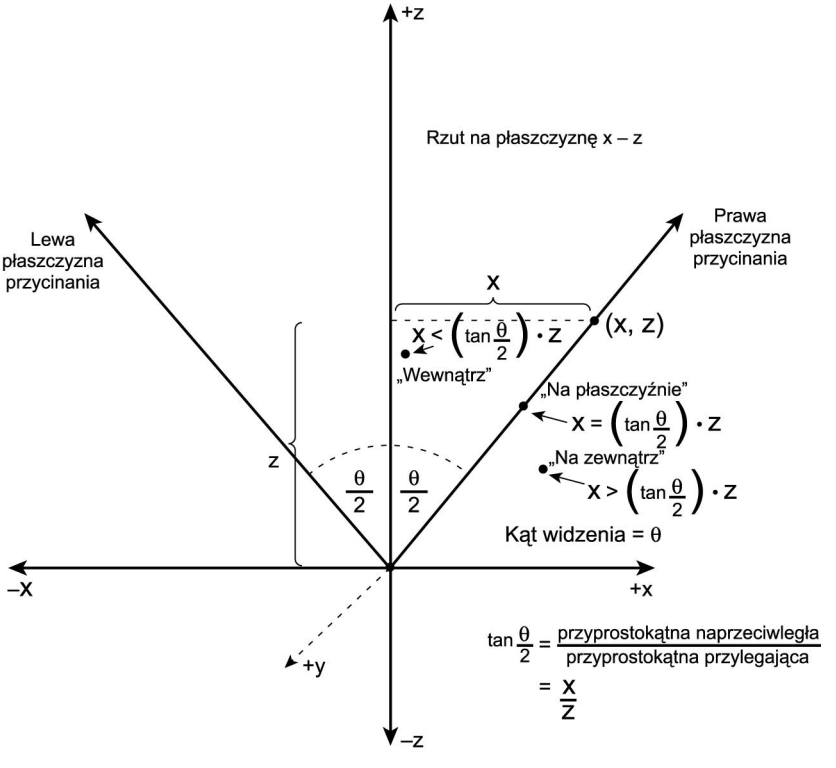
- jeżeli (x > z), v leży na zewnątrz prawej płaszczyzny przycinania;
- jeżeli (x < z), v leży po wewnętrznej stronie prawej płaszczyzny przycinania;
- jeżeli (x = z), v leży na prawej płaszczyźnie przycinania;

Analogiczne testy można przeprowadzić dla lewej płaszczyzny przycinania:

- jeżeli (x > z), v leży po wewnętrznej stronie lewej płaszczyzny przycinania;
- jeżeli (x < z), v leży na zewnątrz lewej płaszczyzny przycinania;
- jeżeli (x = z), v leży na lewej płaszczyźnie przycinania;

Analogiczne wnioski można wysnuć również dla górnej i dolnej płaszczyzny przycinania — wystarczy w miejsce x podstawić y. Jest to oczywiście dość szczególny przykład, bo działa tylko dla pola widzenia równego 90 stopni, jednak zmiana tego kąta powoduje jedynie zmianę nachylenia rzutów płaszczyzn przycinania, a do uwzględnienia takiej zależności wystarczy niewielka zmiana powyższych wzorów. Na rysunku 10.20 prezentowany jest przypadek bardziej ogólnej zależności klasyfikacji położenia wierzchołka od stosunku jego współrzędnych przy danym kącie widzenia.

Rysunek 10.20. *Klasyfikacja wierzchołków na podstawie stosunku współrzędnych i kąta nachylenia płaszczyzn przycinania*



Wykorzystujemy tam wzór:

```
tan(θ) = x / z
```

stąd:

```
x = tan(θ) * z
```

Można na tej podstawie wytypować trzy przypadki położenia wierzchołka:

- jeżeli x = tan(θ) ∗ z, testowany wierzchołek leży na płaszczyźnie przycinania;
- jeżeli x < tan(θ) ∗ z, testowany wierzchołek znajduje się w dodatniej półprzestrzeni płaszczyzny przycinania, a więc wewnątrz bryły ostrosłupa widzenia;
- jeżeli x > tan(θ) ∗ z, testowany wierzchołek znajduje się w ujemnej półprzestrzeni płaszczyzny przycinania, a więc leży poza bryłą ostrosłupa widzenia;

Jak widać, cały problem klasyfikacji wierzchołków jako znajdujących się wewnątrz i poza bryłą ostrosłupa widzenia sprowadza się do porównania współrzędnych (x, y) każdego wierzchołka *v*(x, y, z) z jego współrzędną z, z uwzględnieniem kąta nachylenia płaszczyzn przycinania jako tangensa kąta widzenia bądź wielkości pochodnej (przykładowo, wielkościami takimi są wysokość i szerokość pola widzenia, gdyż jest ono uzależnione od dystansu i kąta widzenia).

Przećwiczyłem ten algorytm co najmniej 500 razy, więc możecie przyjąć na wiarę to, co zostało tu powiedziane. W każdym przypadku, jeżeli wszystkie trzy wierzchołki badanego trójkąta leżą wewnątrz ostrosłupa widzenia, trójkąt jest zaliczany do sceny, a algorytm przechodzi do następnego trójkąta. Jeżeli zaś wszystkie trzy wierzchołki badanego trójkąta leżą poza ostrosłupem widzenia, cały trójkąt jest usuwany z potoku wyświetlania i oznaczany jako „obcięty":

```
// wytnij wielokąt leżący w całości poza ostrosłupem widzenia
SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);
```

Przed zaprezentowaniem kodu warto byłoby — choćby w celu uniknięcia powtórzeń — zaczekać do omówienia przycinania względem płaszczyzny z, niemniej jednak należy się Wam choć krótki fragment kodu, ilustrujący implementację zaprojektowanych dotychczas testów. Oto kod sprawdzający, czy dany wierzchołek trójkąta znajduje się wewnątrz, czy może poza płaszczyznami przycinania:

```
// ponieważ przycinanie odbywa się względem lewej i prawej płaszczyzny przycinania ,
// test polega na porównaniu współrzędnej x wierzchołka z wartością z płaszczyzn;
// do porównania wykorzystywana jest funkcja kąta widzenia
z_factor = (0.5) * cam->viewplane_width / cam->view_dist;

// wierzchołek 0
z_test = z_factor * curr_poly->tvlist[0].z;

if (curr_poly->tvlist[0].x > z_test)
  vertex_ccodes[0] = CLIP_CODE_GX;
else
if (curr_poly->tvlist[0].x < - z_test)
  vertex_ccodes[0] = CLIP_CODE_LX;
else
  vertex_ccodes[0] = CLIP_CODE_IX;
```

Tyle dyskutowania i zaledwie dziesięć wierszy kodu! Hm, jak już wiemy, wystarczy wiedzieć, czy wierzchołek znajduje się po wewnętrznej, czy po zewnętrznej stronie płaszczyzny przycinania. W zależności od tego w kodzie wierzchołek otrzymuje znacznik (jak w algorytmie Cohena-Sutherlanda), który informuje o położeniu względem płaszczyzny ostrosłupa widzenia. Znacznik ten jest później wykorzystywany do prostej eliminacji lub zaliczania wielokąta do dalszego przetwarzania. Tymczasem zajmiemy się analizą algorytmu przycinania względem bliskiej i dalekiej płaszczyzny przycinania — to nieco bardziej złożona procedura.

Przycinanie do bliższej (dalszej) płaszczyzny przycinania

Tutaj sprawy nieco się komplikują. Dyskutowanie o przycinaniu jest przyjemne i proste, ale gdy przychodzi do implementacji w przestrzeni trójwymiarowej, rzecz okazuje się niebanalna! I to nie ze względu na skomplikowanie samego algorytmu — diabeł, jak się okaże, tkwi raczej w pozornie niegroźnych szczegółach… Załóżmy więc, że udało się już wyeliminować z listy wyświetlania wszystkie te wielokąty, które znajdowały się w całości poza ostrosłupem widzenia (na zewnątrz prawej, lewej, górnej i dolnej płaszczyzny przycinania). Pozostałe wielokąty muszą jeszcze zostać sprawdzone względem bliskiej i dalekiej płaszczyzny przycinania. Oba testy są trywialne, ponieważ płaszczyzny przycinania są prostopadłe do dodatniej półosi z, a kamera jest wyrównana właśnie do tej półosi. Stąd dla każdego wierzchołka *vi*(xi, yi, zi) (dla i = 0, 1, 2) badanego trójkąta wystarczy porównać współrzędne zi z wartościami współrzędnych z obu płaszczyzn przycinania; współrzędne te definiowane są w strukturze opisującej kamerę jako składowe `near_clip_z` i `far_clip_z`.

Przykładowo, aby sprawdzić, czy wielokąt znajduje się w całości za daleką płaszczyzną przycinania, należałoby zrealizować następujący pseudokod:

```
jeżeli (v0.z > daleka_pl_przycinania) i
  (v1.z > daleka_pl_z) i
  (v2.z > daleka_pl_z) to usuń wielokąt (v0, v1, v2)
```

Całkiem proste, prawda? Analogicznie wyglądałby test względem bliższej płaszczyzny przycinania:

```
jeżeli (v0.z < bliska_pl_przycinania) i
  (v1.z < bliska_pl_przycinania) i
  (v2.z < bliska_pl_przycinania) to usuń wielokąt (v0, v1, v2)
```

Jeśli wziąć pod uwagę przycinanie względem dalekiej płaszczyzny, nieistotne są przypadki położenia trójkątów częściowo wewnątrz obszaru przycinania, jednak *muszą* one zostać osobno rozpatrzone względem bliższej płaszczyzny. Tutaj nie da się uniknąć szczegółowej analizy położenia wierzchołka. Pseudokod takiej analizy wyglądałby następująco:

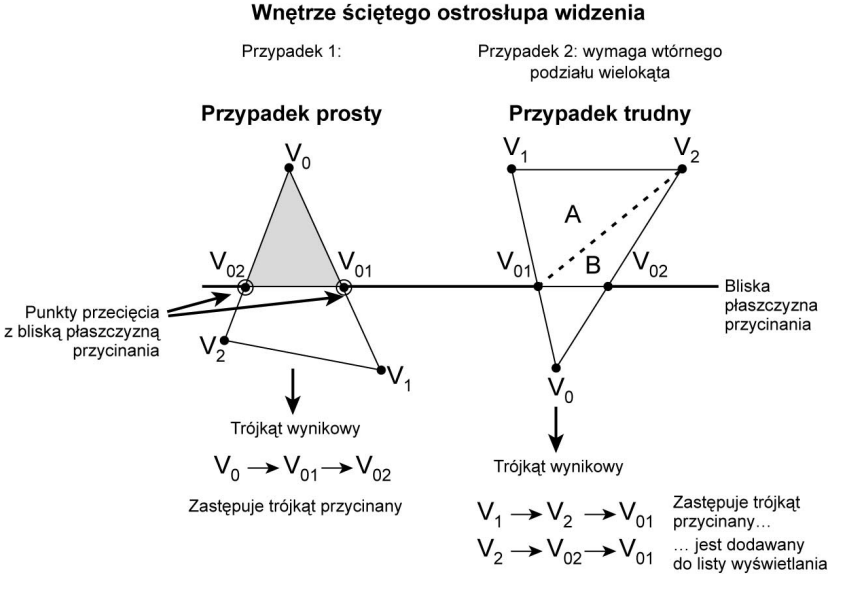
```
dla każdego wielokąta z listy wyświetlania
zacznij
  dla każdego wierzchołka v bieżącego wielokąta P
  zacznij

    Zapamiętaj pozycję wierzchołka v:
    1. Czy v leży wewnątrz ostrosłupa widzenia?
    2. Czy v leży na zewnątrz dalekiej płaszczyzny przycinania?
    3. Czy v leży na zewnątrz bliskiej płaszczyzny przycinania?
  koniec

  jeżeli wszystkie wierzchołki wielokąta znajdują się na zewnątrz
  bliskiej lub dalekiej płaszczyzny przycinania, usuń P z listy wyświetlania w całości

  jeżeli (którykolwiek z wierzchołków znajduje się wewnątrz ostrosłupa widzenia) i
    (którykolwiek z wierzchołków znajduje się na zewnątrz bliskiej płaszczyzny przycinania)
  przytnij P do bliskiej płaszczyzny przycinania
koniec
```

Wytłuszczony fragment pseudokodu oznacza problem do implementacji. Problem ten sprowadza się do tego, że konieczne jest określenie sposobu przycięcia wielokąta do płaszczyzny przycinania, co z grubsza oznacza konieczność przycięcia każdej krawędzi wielokąta wychodzącej poza obszar bryły ostrosłupa widzenia przez bliską płaszczyznę przycinania. Dwa możliwe przypadki takiego przycinania, jeżeli wielokątami są trójkąty, ilustruje rysunek 10.21.



Rysunek 10.21. *Dwa przypadki przycinania względem bliskiej płaszczyzny przycinania*

Przypadek 1. Jeden wierzchołek we wnętrzu obszaru przycinania, dwa poza tym obszarem

To prostszy wariant przycinania. Wystarczy tu przyciąć dwie krawędzie przenikające przez bliską płaszczyznę przycinania i zastąpić trójkąt przycinany nową wersją, z dwoma nowymi wierzchołkami. Jeżeli do trójkąta przypisana była tekstura, należy również przeliczyć współrzędne mapowania tekstury. Oczywiście, należy też ponownie obliczyć wektor normalny trójkąta — zmiana wielkości trójkąta zmienia również jego wektor normalny. Nową długość wektora normalnego należy zachować na potrzeby fazy oświetlenia, która również musi zostać dla danego trójkąta wykonana ponownie.

Przypadek 2. Dwa wierzchołki we wnętrzu obszaru przycinania, jeden poza tym obszarem

Ten przypadek jest dużo bardziej skomplikowany. Odwołajmy się ponownie do rysunku 10.21 — kiedy dwa wierzchołki trójkąta wystają poza obszar przycinania, również mamy dwa punkty przecięcia z bliską płaszczyzną przycinania. Tyle, że w tym przypadku oznacza to, że wynikowy wielokąt jest czworokątem, czego nasz silnik graficzny nie znosi. Należy go podzielić na dwa nowe trójkąty! W tym celu najlepiej utworzyć kopię przycinanego trójkąta i na niej wykonać przycinanie, owocujące powstaniem dwóch nowych trójkątów: A i B. Następnie oryginalny trójkąt przycinany można zastąpić trójkątem A, a trójkąt B po prostu dodać na koniec listy wyświetlania.

Przed przejściem do dalszej lektury warto dokładnie zapoznać się z procedurą postępowania w tym złożonym przypadku. Podsumowując: kiedy przycinajmy trójkąt do płaszczyzny, uzyskujemy w wyniku przycinania jeden lub dwa różne od pierwotnego trójkąty. Jeżeli będzie to jeden trójkąt, wystarczy zastąpić trójkąt przycinany trójkątem przyciętym, co nie powoduje rewolucji w liście wyświetlania. Jeżeli jednak w wyniku przycinania w miejsce trójkąta przycinanego powstaną dwa nowe trójkąty (A i B), nie wystarczy zastąpienie oryginalnego trójkąta nowym — drugi trójkąt należy wstawić do listy wyświetlania.

W tym miejscu można by pomyśleć: „szkoda, że zamiast tablicy wskaźników do wielokątów nie operujemy powiązaną listą", a to dlatego, że podczas sortowania wielokątów względem położenia w osi z wystąpi następująca sytuacja: wielokąt podzielony podczas przycinania będzie miał swojego klona, umieszczonego na końcu listy, o którym z góry wiadomo, że ma współrzędną z sortowania identyczną z taką współrzędną oryginału; sortowanie zostanie spowolnione, ponieważ algorytm sortowania będzie musiał przenieść uzyskany z podziału wielokąt z samego końca listy. Wada ta zostanie wyeliminowania w ramach optymalizacji.

Znamy już scenariusz implementacji, pora na zgłębienie rzeczywistego kodu przycinania. Jak będzie on implementowany? Otóż stworzymy algorytm będący hybrydą wszystkich zaprezentowanych dotychczas algorytmów przycinania. Mamy już gotowy etap wstępnej klasyfikacji wierzchołków. Trzeba tylko znaleźć sposób na wyznaczenie punktów przecięcia trójkąta przycinanego z bliską płaszczyzną przycinania. Wiemy już, że przycinanie takie można podzielić na dwa przypadki (pierwszy — z jednym wierzchołkiem wewnątrz obszaru przycinania, i drugi — z dwoma wierzchołkami wewnątrz tego obszaru) — będą one zakodowane osobno. W każdym z tych przypadków wykorzystane zostaną parametryczne definicje prostej dla dwóch wychodzących poza obszar przycinania krawędzi, w rodzaju:

```
p = v0 + (v1 - v0)*t
```

gdzie *v0* jest wierzchołkiem początkowym krawędzi, a *v1* jej wierzchołkiem końcowym.

Teraz należy podstawić do parametrycznego równania składowej z krawędzi współrzędną z bliskiej płaszczyzny przycinania i wyznaczyć z niego t:

```
p = v0 + (v1 - v0)*t

x = v0.x + (v1.x - v0.x)*t
y = v0.y + (v1.y - v0.y)*t
x = v0.z + (v1.z - v0.z)*t
```

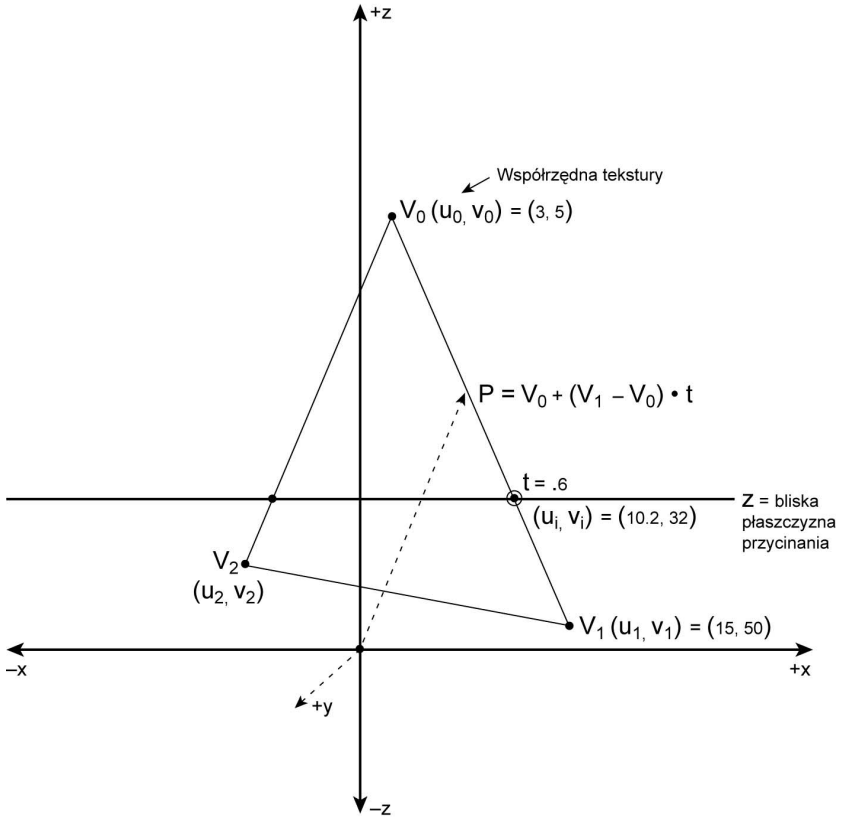
Po podstawieniu wartości `near_clip_z` w miejsce składowej z:

```
near_clip_z = v0.z + (v1.z - v0.z)*t
t = near_clip_z - v0.z/(v1.z - v0.z)
```

Obliczoną wartość t należy podstawić do pozostałych dwóch równań krawędzi dla składowych x i y i proszę — mamy współrzędne punktu przecięcia! Wystarczy wykonać tę procedurę jeszcze raz, dla drugiej krawędzi przecinającej płaszczyznę przycinania. Reszta to już tylko operacje porządkujące, a więc między innymi dopilnowanie, aby oryginalny trójkąt został zastąpiony nowym trójkątem wynikowym przez zastąpienie współrzędnych wierzchołków wyliczonymi współrzędnymi punktów przecięcia oraz aby wprowadzić ewentualnie nowo powstały trójkąt do listy wyświetlania.

Obliczanie współrzędnych mapowania tekstury. Został już tylko szczegół w postaci konieczności obliczenia nowych współrzędnych mapowania tekstury — tu zalecana jest ostrożność. Jeżeli wielokąt podlegał teksturowaniu, teksturę należy przyciąć tak samo, jak przycięty został sam wielokąt; na szczęście można do obliczeń wykorzystać wyliczoną już wartość t. Oto przykład przeliczania współrzędnych mapowania tekstury. Niech dany będzie trójkąt, przycinany w przypadku pierwszym względem bliskiej płaszczyzny przycinania, jak na rysunku 10.22; do każdego wierzchołka trójkąta przypisane zostały współrzędne tekstury, co również zobrazowane jest na rysunku.

Rysunek 10.22. *Przeliczanie nowych współrzędnych mapowania tekstury dla przyciętego trójkąta*



Załóżmy teraz, że obliczono już punkty przecięcia krawędzi trójkąta przycinanego z bliską płaszczyzną przycinania i okazało się, że dla jednego z punktów przecięcia parametr t ma wartość 0,6. Wystarczy teraz skorzystać z tej wartości do liniowej interpolacji współrzędnych tekstury i w ten sposób określić nowe przypisania tekstury do wierzchołków trójkąta. Odwołując się ponownie do rysunku — widać, że wierzchołek 0 posiada współrzędne mapowania tekstury równe u0v0(3, 5), a współrzędne te dla wierzchołka 1 to u_1v_1(15, 50). Nowe wartości współrzędnych mapowania można obliczyć następująco:

```
u' = u0 + (u1 - u0)*t
v' = v0 + (v1 - v0)*t
```

Po podstawieniu wartości parametru t otrzymujemy:

```
u' = 3 + (15 - 3)*(0.6) = 10.2
v' = 5 + (50 - 5)*(0.6) = 32
```

I to byłoby już wszystko w kwestii przycinania względem bliskiej płaszczyzny przycinania, z ewentualnym wtórnym podziałem przycinanego wielokąta i ponownym mapowaniem współrzędnych tekstury włącznie. Na koniec należałoby przeliczyć długości wektorów normalnych wszystkich przyciętych i nowo powstałych wielokątów, ponieważ długości te będą potrzebne na etapie oświetlania sceny. Dla wszystkich przyciętych wielokątów, niezależnie od tego, czy zostały one wtórnie podzielone, czy nie, należy więc wykonać następujący kod:

```
// oblicz u, v
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v1].v, &u);
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v2].v, &v);

// oblicz iloczyn wektorowy
VECTOR4D_Cross(&u, &v, &n);

// oblicz długość wektora normalnego i zapamiętaj ją w polu nlength struktury wielokąta
curr_poly->nlength = VECTOR4D_Length_Fast(&n);
```

To dodatkowy narzut obliczeniowy, ale należy się pocieszać, że w jednej ramce przyciętych do bliskiej płaszczyzny przycinania zostanie najwyżej kilkaset wielokątów, a wykorzystanie szybkiej wersji obliczania wektora normalnego narzut ten dodatkowo zmniejsza, choć wyniki tych obliczeń są nieco gorsze niż w wersji pełnej. Dokładność spada o pięć do dziesięciu procent, ale nie należy zbytnio się tym przejmować, ponieważ fakt, że wielokąt przekroczył częściowo bliską płaszczyznę przycinania wskazuje, że w najbliższym czasie zostanie w ogóle usunięty z pola widzenia.

Cóż, zdaje się, że temat został już dokładnie przewałkowany, czas na prezentację właściwego kodu funkcji przycinającej. Oto kod źródłowy:

```
void Clip_Polys_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                               CAM4DV1_PTR cam, int clip_flags)
{
  // Funkcja ta przycina wielokąty z przekazanej listy wielokątów rend_list
  // względem zadanego obszaru przycinania i ustawia dla wielokąta znaczniki
  // przycinania, dzięki czemu nie przechodzi on do dalszego przetwarzania.
  // Funkcja realizuje rzeczywiste przycinanie wyłącznie względem bliższej
  // i dalszej płaszczyzny przycinania, dla pozostałych płaszczyzn ostrosłupa
  // widzenia realizując proste testy klasyfikujące. Dla tych płaszczyzn,
  // jeżeli wielokąt w całości leży na zewnątrz płaszczyzny, zostaje oznaczony
  // jako usunięty z obszaru widzenia; test ten nie jest równie efektywny dla
  // obiektów, ale w przypadku list wielokątów naprawdę dużych obiektów, których
  // fragmenty są zawsze widoczne, testowanie poszczególnych wielokątów jest
  // dostatecznie efektywne. Funkcja zakłada, że współrzędne wielokątów zostały
  // przekształcone do przestrzeni kamery

  // wewnętrzne kody przycinania
#define CLIP_CODE_GZ 0x0001 // z > z_max
#define CLIP_CODE_LZ 0x0002 // z < z_min
#define CLIP_CODE_IZ 0x0004 // z_min < z < z_max

#define CLIP_CODE_GX 0x0001 // x > x_max
#define CLIP_CODE_LX 0x0002 // x < x_min
#define CLIP_CODE_IX 0x0004 // x_min < x < x_max

#define CLIP_CODE_GY 0x0001 // y > y_max
#define CLIP_CODE_LY 0x0002 // y < y_min
#define CLIP_CODE_IY 0x0004 // y_min < y < y_max

#define CLIP_CODE_NULL 0x0000

  int vertex_ccodes[3]; // przechowuje znaczniki przycinania
  int num_verts_in;     // liczba wierzchołków wewnątrz
  int v0, v1, v2;       // indeksy wierzchołków
```

```
float z_factor,       // do obliczeń przycinania
z_test;               // do obliczeń przycinania
float xi, yi, x01i, y01i, x02i, y02i, // punkty przecięcia
t1, t2,               // wartości parametru t
ui, vi, u01i, v01i, u02i, v02i;   // współrzędne tekstury
int last_poly_index,  // ostatni prawidłowy wielokąt na liście rend_list
insert_poly_index;    // bieżąca pozycja wstawiania nowych wielokątów
VECTOR4D u, v, n;     // do obliczeń wektorowych
POLYF4DV2 temp_poly;  // kopia wielokąta wtórnie dzielonego

// ustawienie indeksów ostatniego i wstawianego wielokąta na liście wielokątów
// nie warto dwukrotnie przycinać wielokątów
insert_poly_index = last_poly_index = rend_list->num_polys;

// przeglądanie listy wielokątów i przycinanie (usuwanie)
for (int poly = 0; poly < last_poly_index; poly++)
{
  // pobierz bieżący wielokąt
  POLYF4DV2_PTR curr_poly = rend_list->poly_ptrs[poly];

  // czy wielokąt nadaje się do przycinania?
  // sprawdzany wielokąt nie powinien być przycięty, usunięty, zwrócony tyłem
  // powinien być aktywny i widoczny

  // Sprawdzenie, czy wielokąt nie jest zwrócony tyłem, został zrealizowane
  // w poprzednim wywołaniu, wystarczy więc sprawdzić odpowiedni znacznik
  if ((curr_poly == NULL) || !(curr_poly->state & POLY4DV2_STATE_ACTIVE) ||
     (curr_poly->state & POLY4DV2_STATE_CLIPPED ) ||
     (curr_poly->state & POLY4DV2_STATE_BACKFACE))
    continue;     // przejdź do następnego wielokąta

  // przycinanie (usuwanie) względem płaszczyzn x
  if (clip_flags & CLIP_POLY_X_PLANE)
  {
    // przytnij (usuń) w oparciu o wartość x płaszczyzny przycinania
    // dla każdego wierzchołka sprawdzaj, czy znajduje się on wewnątrz obszaru
    // przycinania i ustawiaj odpowiednio znacznik przycinania. Nie przycinaj
    // trójkątów, najpierw próbuj je odrzucić. Wielokąty wystające częściowo
    // zostaną przycięte do prostokąta rzutu ekranowego na etapie rasteryzacji,
    // tymczasem usunięte zostaną te wielokąty, które w całości znajdują się
    // poza ostrosłupem widzenia

    // Podczas przycinania do bocznych płaszczyzn przycinania wykorzystywane są
    // równania płaszczyzn lub parametr kąta widzenia; chodzi o określenie
    // stosunku wartości współrzędnych x i z do określenia widoczności wierzchołka
    z_factor = (0.5) * cam->viewplane_width / cam->view_dist;

    // wierzchołek 0
    z_test = z_factor * curr_poly->tvlist[0].z;

    if (curr_poly->tvlist[0].x > z_test)
      vertex_ccodes[0] = CLIP_CODE_GX;
    else
    if (curr_poly->tvlist[0].x < - z_test)
      vertex_ccodes[0] = CLIP_CODE_LX;
    else
      vertex_ccodes[0] = CLIP_CODE_IX;

    // wierzchołek 1
    z_test = z_factor * curr_poly->tvlist[1].z;
    if (curr_poly->tvlist[1].x > z_test)
      vertex_ccodes[1] = CLIP_CODE_GX;
    else
    if (curr_poly->tvlist[1].x < -z_test)
      vertex_ccodes[1] = CLIP_CODE_LX;
```

```
  else
    vertex_ccodes[1] = CLIP_CODE_IX;

  // wierzchołek 2
  z_test = z_factor * curr_poly->tvlist[2].z;
  if (curr_poly->tvlist[2].x > z_test)
    vertex_ccodes[1] = CLIP_CODE_GX;
  else
  if (curr_poly->tvlist[2].x < - z_test)
    vertex_ccodes[1] = CLIP_CODE_LX;
  else
    vertex_ccodes[2] = CLIP_CODE_IX;

  // testy odrzucania;
  // odrzucane są wielokąty znajdujące się w całości poza
  // bocznymi płaszczyznami przycinania
  if ( ((vertex_ccodes[0] == CLIP_CODE_GX) &&
     (vertex_ccodes[1] == CLIP_CODE_GX) &&
     (vertex_ccodes[2] == CLIP_CODE_GX) ) ||

     ((vertex_ccodes[0] == CLIP_CODE_LX) &&
     (vertex_ccodes[1] == CLIP_CODE_LX) &&
     (vertex_ccodes[2] == CLIP_CODE_LX) ) )
  {
    // oznacz jako przycięty - wystający w całości poza ostrosłup widzenia
    SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

    // przejdź do następnego wielokąta
    continue;
  } // if
} // if dla płaszczyzn x

// przycinanie (usuwanie) względem płaszczyzn y
if (clip_flags & CLIP_POLY_Y_PLANE)
{
  // przytnij (usuń) w oparciu o wartość y płaszczyzny przycinania
  // dla każdego wierzchołka sprawdzaj, czy znajduje się on wewnątrz obszaru
  // przycinania i ustawiaj odpowiednio znacznik przycinania. Nie przycinaj
  // trójkątów, najpierw próbuj je odrzucić. Wielokąty wystające częściowo
  // zostaną przycięte do prostokąta rzutu ekranowego na etapie rasteryzacji,
  // tymczasem usunięte zostaną te wielokąty, które w całości znajdują się
  // poza ostrosłupem widzenia
  // Podczas przycinania do górnej i dolnej płaszczyzny przycinania
  // wykorzystywane są równania płaszczyzn lub parametr kąta widzenia;
  // chodzi o określenie stosunku wartości współrzędnych y i z do
  // określenia widoczności wierzchołka;
  z_factor = (0.5) * cam->viewplane_width / cam->view_dist;

  // wierzchołek 0
  z_test = z_factor * curr_poly->tvlist[0].z;
  if (curr_poly->tvlist[0].y > z_test)
    vertex_ccodes[0] = CLIP_CODE_GY;
  else
  if (curr_poly->tvlist[0].y < -z_test)
    vertex_ccodes[0] = CLIP_CODE_LY;
  else
    vertex_ccodes[0] = CLIP_CODE_IY;

  // wierzchołek 1
  z_test = z_factor * curr_poly->tvlist[1].z;
  if (curr_poly->tvlist[1].y > z_test)
    vertex_ccodes[1] = CLIP_CODE_GY;
  else
  if (curr_poly->tvlist[1].y < - z_test)
    vertex_ccodes[1] = CLIP_CODE_LY;
```

```
  else
    vertex_ccodes[1] = CLIP_CODE_IY;

  // wierzchołek 2
  z_test = z_factor * curr_poly->tvlist[2].z;
  if (curr_poly->tvlist[2].y > z_test)
    vertex_ccodes[2] = CLIP_CODE_GY;
  else
  if (curr_poly->tvlist[2].y < - z_test)
    vertex_ccodes[2] = CLIP_CODE_LY;
  else
    vertex_ccodes[2] = CLIP_CODE_IY;

  // testy odrzucania;
  // odrzucane są wielokąty znajdujące się w całości poza
  // bocznymi płaszczyznami przycinania
  if ( ((vertex_ccodes[0] == CLIP_CODE_GY) &&
     (vertex_ccodes[1] == CLIP_CODE_GY) &&
     (vertex_ccodes[2] == CLIP_CODE_GY) ) ||

     ((vertex_ccodes[0] == CLIP_CODE_LX) &&
     (vertex_ccodes[1] == CLIP_CODE_LX) &&
     (vertex_ccodes[2] == CLIP_CODE_LX) ) )
  {
    // oznacz jako przycięty - wystający w całości poza ostrosłup widzenia
    SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

    // przejdź do następnego wielokąta
    continue;
  } // if
} // if dla płaszczyzn y

// przycinanie (usuwanie) względem płaszczyzn z
if (clip_flags & CLIP_POLY_Z_PLANE)
{
  // przytnij (usuń) w oparciu o wartość z płaszczyzny przycinania
  // dla każdego wierzchołka sprawdzaj, czy znajduje się on wewnątrz obszaru
  // przycinania i ustawiaj odpowiednio znacznik przycinania; faktyczne
  // przycinanie do bliższej płaszczyzny przycinania realizowane będzie
  // później grupowo; przycinanie może zaowocować dodaniem do listy
  // rend_list nowego wielokąta

  // wyzeruj liczniki wierzchołków
  // - przydadzą się podczas klasyfikacji trójkąta wynikowego

  num_verts_in = 0;

  // wierzchołek 0
  if (curr_poly->tvlist[0].z > cam->far_clip_z)
  {
    vertex_ccodes[0] = CLIP_CODE_GZ;
  }
  else
  if (curr_poly->tvlist[0].z < cam->near_clip_z)
  {
    vertex_ccodes[0] = CLIP_CODE_LZ;
  }
  else
  {
    vertex_ccodes[0] = CLIP_CODE_IZ;
    num_verts_in++;
  }

  // wierzchołek 1
  if (curr_poly->tvlist[1].z > cam->far_clip_z)
  {
    vertex_ccodes[1] = CLIP_CODE_GZ;
```

```
}
else
if (curr_poly->tvlist[1].z < cam->near_clip_z)
{
  vertex_ccodes[1] = CLIP_CODE_LZ;
}
else
{
  vertex_ccodes[1] = CLIP_CODE_IZ;
  num_verts_in++;
}

// wierzchołek 2
if (curr_poly->tvlist[2].z > cam->far_clip_z)
{
  vertex_ccodes[2] = CLIP_CODE_GZ;
}
else
if (curr_poly->tvlist[2].z < cam->near_clip_z)
{
  vertex_ccodes[2] = CLIP_CODE_LZ;
}
else
{
  vertex_ccodes[2] = CLIP_CODE_IZ;
  num_verts_in++;
}

// testy dla przypadków prostego odrzucania dla wielokątów w całości
// znajdujących się poza bliską płaszczyzną przycinania
if ( ((vertex_ccodes[0] == CLIP_CODE_GZ) &&
   (vertex_ccodes[1] == CLIP_CODE_GZ) &&
   (vertex_ccodes[2] == CLIP_CODE_GZ) ) ||

   ((vertex_ccodes[0] == CLIP_CODE_LZ) &&
   (vertex_ccodes[1] == CLIP_CODE_LZ) &&
   (vertex_ccodes[2] == CLIP_CODE_LZ) ) )
{
  // oznacz jako przycięty - wystający w całości poza ostrosłup widzenia
  SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

  // przejdź do następnego wielokąta
  continue;
} // if

// sprawdź, czy którykolwiek z wierzchołków wystaje przed
// bliższą płaszczyznę przycinania?
if ( ((vertex_ccodes[0] | vertex_ccodes[1] | vertex_ccodes[2]) & CLIP_CODE_LZ) )
{
  // można przystąpić do przycinania wielokąta do bliższej płaszczyzny
  // przycinania;
  // przycinanie do płaszczyzny dalszej można sobie darować, gdyż nieprzycięte
  // względem tej płaszczyzny wielokąty nie sprawiają problemów obliczeniowych;
  // przycinanie uwzględnia dwa przypadki: przypadek 1., gdy jeden wierzchołek
  // trójkąta znajduje się po wewnętrznej stronie płaszczyzny przycinania (dwa
  // wierzchołki po stronie zewnętrznej) albo przypadek 2., gdy po stronie
  // wewnętrznej płaszczyzny przycinania znajdują się dwa wierzchołki trójkąta;

  // krok 1. klasyfikacja typu trójkąta na podstawie liczby wierzchołków
  // po wewnętrznej stronie płaszczyzny przycinania
  // przypadek 1. jest przypadkiem prostym
  if (num_verts_in == 1)
  {
    // należy przyciąć trójkąt względem bliskiej płaszczyzny przycinania;
    // procedura przycinania realizowana jest względem każdej z krawędzi
    // wychodzących z wierzchołka wewnętrznego poza obszar przycinania;
```

```
// konieczne jest obliczenie z parametrycznej postaci równania krawędzi
// punktu przecięcia krawędzi z bliską płaszczyzną przycinania;
// po obliczeniu punktu przecięcia jest on wprowadzany na miejsce
// dotychczasowego punktu końcowego krawędzi, przeliczane są ewentualne
// współrzędne mapowania tekstury; przycinanie w tym przypadku nie wprowadza
// dodatkowych trójkątów - w przypadku 2. konieczne jest dodanie do listy
// rend_list jednego nowego trójkąta

// krok 1. wyszukać wierzchołek znajdujący się po wewnętrznej
// stronie płaszczyzny przycinania
if ( vertex_ccodes[0] == CLIP_CODE_IZ)
 { v0 = 0; v1 = 1; v2 = 2; }
else
if (vertex_ccodes[1] == CLIP_CODE_IZ)
 { v0 = 1; v1 = 2; v2 = 0; }
else
 { v0 = 2; v1 = 0; v2 = 1; }

// krok 2. przycięcie krawędzi
// w skrócie polega to na wygenerowaniu prostej p = v0 +( v1 – v0) * t
// i wyliczenia t dla składowej z równej z płaszczyzny przycinania oraz
// podstawieniu t do równania prostej w celu obliczenia pozostałych
// współrzędnych punktu przecięcia; aby zaoszczędzić nieco czasu, należy
// unikać wywoływania funkcji matematycznych wysokiego poziomu

// przytnij krawędź v0 - v1
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v1].v, &v);

// przecięcie zachodzi dla z = near_clip_z, więc t =
t1 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) / v.z );

// podstaw t do składowych równania krawędzi i wylicz współrzędne
//  x i y punktu przecięcia z płaszczyzną z
xi = curr_poly->tvlist[v0].x + v.x * t1;
yi = curr_poly->tvlist[v0].y + v.y * t1;

// zastąp dotychczasowe współrzędne wierzchołka końcowego krawędzi
curr_poly->tvlist[v1].x = xi;
curr_poly->tvlist[v1].y = yi;
curr_poly->tvlist[v1].z = cam->near_clip_z;

// przytnij krawędź v0 - v2
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v2].v, &v);

// przecięcie zachodzi dla z = near_clip_z, więc t =
t2 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) / v.z );

// podstaw t do składowych równania krawędzi i wylicz współrzędne
//  x i y punktu przecięcia z płaszczyzną z
xi = curr_poly->tvlist[v0].x + v.x * t2;
yi = curr_poly->tvlist[v0].y + v.y * t2;

// zastąp dotychczasowe współrzędne wierzchołka końcowego krawędzi
curr_poly->tvlist[v1].x = xi;
curr_poly->tvlist[v1].y = yi;
curr_poly->tvlist[v1].z = cam->near_clip_z;

// mając t1 i t2 można sprawdzić, czy wielokąt jest teksturowany
// i jeżeli tak, przyciąć współrzędne mapowania tekstury
if (curr_poly->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
{
  ui = curr_poly->tvlist[v0].u0 +
     (curr_poly->tvlist[v1].u0 - curr_poly->tvlist[v0].u0) * t1;
  vi = curr_poly->tvlist[v0].v0 +
     (curr_poly->tvlist[v1].v0 - curr_poly->tvlist[v0].v0) * t1;
  curr_poly->tvlist[v1].u0 = ui;
  curr_poly->tvlist[v1].v0 = vi;
```

```
    ui = curr_poly->tvlist[v0].u0 +
       (curr_poly->tvlist[v2].u0 - curr_poly->tvlist[v0].u0) * t2;
    vi = curr_poly->tvlist[v0].v0 +
       (curr_poly->tvlist[v2].v0 - curr_poly->tvlist[v0].v0) * t2;
    curr_poly->tvlist[v2].u0 = ui;
    curr_poly->tvlist[v2].v0 = vi;
  } // if teksturowania

  // na koniec należy wyznaczyć nową długość wektora normalnego
  // przyciętego trójkąta

  // konstruuj u, v
  VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v1].v, &u);
  VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v2].v, &v);

  // oblicz iloczyn wektorowy
  VECTOR4D_Cross(&u, &v, &n);

  // oblicz szybko długość wektora normalnego i zapamiętaj w polu
  // nlength struktury trójkąta
  // później wartość ta zostanie obliczona dokładniej
  curr_poly->nlength = VECTOR4D_Length_Fast(&n);
} // if liczby wierzchołków wewnątrz obszaru przycinania
else
if (num_verts_in == 2)
{
  // dwa wierzchołki wewnątrz obszaru przycinania
  // w tym przypadku również trzeba przyciąć trójkąt do bliższej
  // płaszczyzny przycinania; dla każdej krawędzi wychodzącej
  // z wierzchołka wewnętrznego obliczany jest punkt przecięcia z bliską
  // płaszczyzną przycinania, tyle, że w przeciwieństwie do przypadku 1.
  // trójkąt zostanie podzielony na dwa nowe trójkąty;
  // podczas pierwszego przycinania do listy wyświetlania dodany zostanie
  // nowy trójkąt, a przy drugim przycinaniu zastąpione zostaną
  // dotychczasowe współrzędne oryginalnego trójkąta

  // krok 0. utworzenie kopii przycinanego trójkąta
  memcpy(&temp_poly, curr_poly, sizeof(POLYF4DV2) );

  // krok 1. wyszukaj wierzchołek zewnętrzny
  if ( vertex_ccodes[0] == CLIP_CODE_LZ)
   { v0 = 0; v1 = 1; v2 = 2; }
  else
  if (vertex_ccodes[1] == CLIP_CODE_LZ)
   { v0 = 1; v1 = 2; v2 = 0; }
  else
   { v0 = 2; v1 = 0; v2 = 1; }

  // krok 2. przycinanie krawędzi
  // w skrócie polega to na wygenerowaniu prostej p = v0 + (v1 - v0) * t
  // i wyliczenia t dla składowej z równej z płaszczyzny przycinania oraz
  // podstawieniu t do równania prostej w celu obliczenia pozostałych
  // współrzędnych punktu przecięcia; aby zaoszczędzić nieco czasu
  // unikamy wywoływania funkcji matematycznych wysokiego poziomu

  // przycięcie krawędzi v0->vl
  VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v1].v, &v);

  // przecięcie zachodzi w punkcie z = near_clip_z, więc t =
  t1 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) / v.z );

  // podstaw t do składowych równania krawędzi i wylicz współrzędne
  //  x i y punktu przecięcia z płaszczyzną z
  x01i = curr_poly->tvlist[v0].x + v.x * t1;
  y01i = curr_poly->tvlist[v0].y + v.y * t1;
```

```
// przycięcie krawędzi v0->v2
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v2].v, &v);

// przecięcie zachodzi w punkcie z = near_clip_z, więc t =
t2 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) / v.z );

// podstaw t do składowych równania krawędzi i wylicz współrzędne
//  x i y punktu przecięcia z płaszczyzną z
x02i = curr_poly->tvlist[v0].x + v.x * t2;
y02i = curr_poly->tvlist[v0].y + v.y * t2;

// po wyznaczeniu obu punktów przecięcia trzeba nadpisać
// wierzchołek 0 wielokąta punktem przecięcia - powstanie
// pierwszy z dwóch trójkątów wynikowych podziału;
curr_poly->tvlist[v0].x = x01i;
curr_poly->tvlist[v0].y = y01i;
curr_poly->tvlist[v0].z = cam->near_clip_z;

// teraz najtrudniejsze - trzeba utworzyć nowy wielokąt, rozpięty
// pomiędzy dwoma punktami przecięcia i wierzchołkiem v2, po czym
// dodać nowy trójkąt na koniec listy rend_list; na razie trójkąt będzie
// konstruowany na kopii tymczasowej; wierzchołek v2 zostanie bez zmian,
// ale v1 trzeba nadpisać wierzchołkiem v01 a v0 wierzchołkiem v02.
temp_poly.tvlist[v1].x = x01i;
temp_poly.tvlist[v1].y = y01i;
temp_poly.tvlist[v1].z = cam->near_clip_z;
temp_poly.tvlist[v0].x = x02i;
temp_poly.tvlist[v0].y = y02i;
temp_poly.tvlist[v0].z = cam->near_clip_z;

// mając t1 i t2 można sprawdzić, czy wielokąt jest teksturowany
// i jeżeli tak, przyciąć współrzędne mapowania tekstury
if (curr_poly->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
{
  // oblicz nowe współrzędne teksturowania pierwszego wielokąta
  u01i = curr_poly->tvlist[v0].u0 +
     (curr_poly->tvlist[v1].u0 - curr_poly->tvlist[v0].u0) *  t1;
  v01i = curr_poly->tvlist[v0].v0 +
     (curr_poly->tvlist[v1].v0 - curr_poly->tvlist[v0].v0) * t1;

  // oblicz nowe współrzędne teksturowania drugiego wielokąta
  u02i = curr_poly->tvlist[v0].u0 +
     (curr_poly->tvlist[v2].u0 - curr_poly->tvlist[v0].u0) * t2;
  v02i = curr_poly->tvlist[v0].v0 +
     (curr_poly->tvlist[v2].v0 - curr_poly->tvlist[v0].v0) *t2;

  // zapamiętaj oba trójkąty
  // trójkąt 1
  curr_poly->tvlist[v0].u0 = u01i;
  curr_poly->tvlist[v0].v0 = v01i;

  // trójkąt 2
  temp_poly.tvlist[v0].u0 = u02i;
  temp_poly.tvlist[v0].v0 = v02i;
  temp_poly.tvlist[v1].u0 = u01i;
  temp_poly.tvlist[v1].v0 = v01i;
} // if obliczania współrzędnych teksturowania

// na koniec należy wyznaczyć nową długość wektora normalnego
// przyciętego trójkąta;

// najpierw trójkąt 1
// konstruuj u, v
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v1].v, &u);
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v2].v, &v);
```

```
            // oblicz iloczyn wektorowy
            VECTOR4D_Cross(&u, &v, &n);

            // oblicz szybko nową długość wektora normalnego i zapamiętaj ją w polu
            // nlength struktury trójkąta
            curr_poly->nlength = VECTOR4D_Length_Fast(&n);

            // trójkąt 2 (temp_poly)
            // konstruuj u, v
            VECTOR4D_Build(&temp_poly.tvlist[v0].v, &temp_poly.tvlist[v1].v, &u);
            VECTOR4D_Build(&temp_poly.tvlist[v0].v, &temp_poly.tvlist[v2].v, &v);

            // oblicz iloczyn wektorowy
            VECTOR4D_Cross(&u, &v, &n);

            // oblicz szybko nową długość wektora normalnego i zapamiętaj ją w polu
            // nlength struktury trójkąta
            temp_poly.nlength = VECTOR4D_Length_Fast(&n) ;

            // prawie gotowe - trzeba jeszcze wstawić trójkąt do listy rend_list
            // jeżeli wielokąt nie zmieści się, funkcja zwróci 0
            Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &temp_poly);
          } // else
        } // if dla przycinania względem bliskiej płaszczyzny przycinania
      } // if dla przycinania w płaszczyznach z
    } // for
  } // end Clip_Polys_RENDERLIST4DV2
```

Czytanie kodów drukowanych w książkach nie jest szczególnie popularnym zajęciem — po pierwsze, kod taki nie jest zwykle najbardziej czytelny. Ale niekiedy warto prześledzić kod wiersz po wierszu i sprawdzić, czy dokładnie rozumie się znaczenie każdego kolejnego wywołania. Taka analiza nie może, co prawda, zastąpić nieco bardziej ogólnych wyjaśnień, wspieranych ewentualnie pseudokodem. Jednak wiele osób uczy się w dokładnie odwrotny sposób, stąd decyzja o zamieszczeniu w książce wydruków kodów. Tak czy inaczej, niezależnie od indywidualnego sposobu uczenia się warto spróbować prześledzić działanie prezentowanej funkcji. Można w niej zaobserwować wiele podstawowych elementów programowania 3D, przy tym jej analiza tylko z pozoru jest czasochłonna — w funkcji powtarzają się przecież obszerne fragmenty kodu.

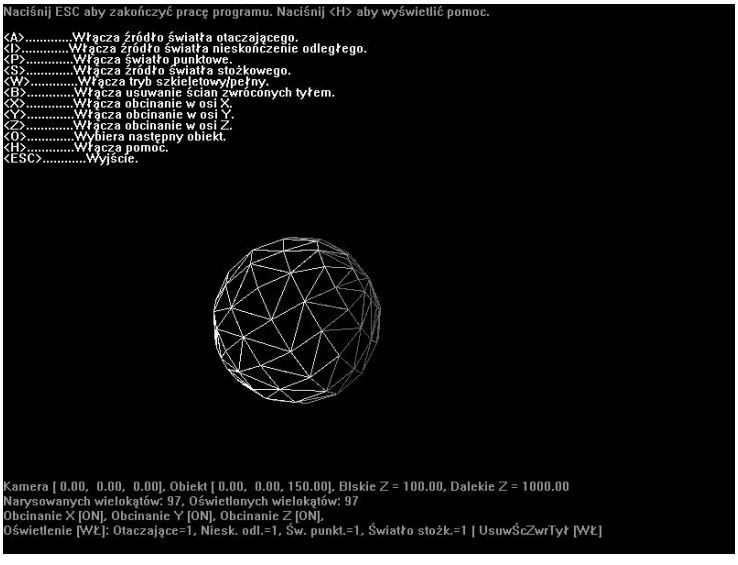
Korzystanie z funkcji jest zaś wprost śmiesznie proste. Jej wywołanie ma następującą postać:

```
Clip_Polys_RENDERLIST4DV2(&rend_list, &cam, CLIP_POLY_Z_PLANE |
                                            CLIP_POLY_X_PLANE |
                                            CLIP_POLY_Y_PLANE);
```

Wystarczy przekazać do funkcji listę wyświetlania (`rend_list`), strukturę opisującą kamerę (`cam`) i znaczniki sterujące przycinaniem, a funkcja zajmie się całą resztą. Zauważmy, że funkcja obsługuje niezależne przycinanie względem płaszczyzn x, y i z. Po włączeniu wszystkich trzech z potoku usuwane są wszystkie przeszkadzające do tej pory trójkąty.

Demonstracja działania funkcji przycinania

Jako przykład wykorzystania jeszcze ciepłej funkcji przycinania wielokątów w przestrzeni trójwymiarowej można wykorzystać dołączony do książki program demonstracyjny, obrazujący wirowanie prostego obiektu w przestrzeni. Kod źródłowy programu znajduje się w pliku *DEMOII10_1.CPP*, a wersja wykonywalna w pliku *DEMOII10_1.EXE*; oba pliki znajdują się na dołączonej do książki płycie CD-ROM. Do skompilowania pliku kodu źródłowego potrzebne są pliki od *T3DLIB1.CPP* do *T3DLIB8.CPP* wraz z odpowiednimi dla nich plikami nagłówkowymi oraz pliki biblioteczne *.LIB* biblioteki DirectX. Zrzut ekranowy działającego programu prezentowany jest na rysunku 10.23. Program demonstracyjny wczytuje obiekt do pamięci i wyświetla go na ekranie, powoli obracając. Obiekt można przysuwać i oddalać od płaszczyzny rzutu oraz przesuwać, wymuszając jego wykroczenie poza obszar widzenia.



Rysunek 10.23. *Zrzut ekranowy programu demonstrującego działanie funkcji przycinania*

Program ma dwojakie zastosowanie. Po pierwsze uwidacznia, że usuwanie całych obiektów ma się nijak do usuwania i przycinania pojedynczych składających się na obiekt wielokątów — dotychczas, jeżeli choć fragment obiektu znajdował się wewnątrz bryły ostrosłupa widzenia, obiekt podlegał dalszym transformacjom (w tym oświetlaniu), co niekiedy prowadziło do błędów wyświetlania i wyjątków procesora. Teraz przycinanie realizowane jest na poziomie poszczególnych wielokątów. Warto zwrócić uwagę na informacje statystyczne, wyświetlane w u dołu ekranu. Wyświetlana na ekranie pomoc zawiera pełną listę klawiszy sterujących demonstracją, ale wystarczy wiedzieć, że klawiszami kursorów zmienia się położenie obiektu, następny obiekt wybiera się klawiszem *O*, a przycinanie względem płaszczyzn x, y i z można włączać i wyłączać klawiszami odpowiednio: *X*, *Y* oraz *Z*.

Zmiany w systemie oświetlenia

Zanim przejdziemy dalej, należałoby jeszcze podjąć kwestię związaną z wpływem przycinania na system oświetlenia. Dotychczas obliczenia związane z oświetleniem obiektów realizowane były w przestrzeni sceny, a więc dla współrzędnych świata sceny. Teraz jednak okazuje się, że takie umiejscowienie modułu oświetlenia w potoku przetwarzania jest ogromnym marnotrawstwem mocy obliczeniowej. Otóż oświetlanie wielokątów należy wstrzymać aż do momentu, w którym zawartość sceny zostanie przycięta — po co oświetlać coś, czego i tak nie będzie widać? Tyle, że aby zaoszczędzić na obliczeniach oświetlenia, trzeba dokonać transformacji współrzędnych świateł z przestrzeni sceny do przestrzeni kamery (dopiero w tej przestrzeni potok jest oczyszczany ze zbędnych wielokątów). Można to zrealizować na kilka sposobów. Jednym z nich jest przepisanie funkcji obliczających oświetlenie i przekształcanie współrzędnych i orientacji źródeł światła do przestrzeni kamery. Problem z oświetlaniem w locie tkwi jednak w tym, że funkcje oświetlające wywoływane są na rzecz poszczególnych obiektów, więc pojawiłby się narzut związany z ciągłym przekształcaniem współrzędnych źródeł światła. Lepiej byłoby na stałe przenieść źródła światła do współrzędnych kamery.

Pojawia się jednak problem zachowania oryginalnych współrzędnych oświetlenia. Cóż, trzeba będzie dokonać aktualizacji struktur opisujących źródła światła — nowa wersja stosownej struktury nosić będzie nazwę `LIGHTV2` i będzie zawierać dodatkowe pola dla współrzędnych oświetlenia przeniesionych do przestrzeni kamery. Trzeba też będzie przepisać każdą funkcję operującą na strukturach opisujących źródła światła tak, aby korzystały z nowego typu struktury. Dobra wiadomość jest zaś taka, że w samych funkcjach oświetlających nie będzie trzeba wprowadzać żadnych zmian poza zmianą napisów `LIGHTV1` na `LIGHTV2`. Spójrzmy na budowę nowej wersji struktury opisującej źródło światła:

```
// druga wersja struktury źródła światła
typedef struct LIGHTV2_TYP
{
  int state;    // stan źródła
  int id;       // identyfikator źródła
  int attr;     // atrybuty źródła

  RGBAV1 c_ambient;   // intensywność źródła otaczającego
  RGBAV1 c_diffuse;   // intensywność źródła rozproszonego
  RGBAV1 c_specular;  // intensywność źródła zwierciadlanego

  POINT4D pos;      // pozycja źródła światła
  POINT4D tpos;
  VECTOR4D dir;     // orientacja źródła światła
  VECTOR4D tdir;

  float kc, kl, kq;   // współczynniki tłumienia
  float spot_inner;   // wewnętrzny kąt światła stożkowego
  float spot_outer;   // zewnętrzny kąt światła stożkowego
  float pf;           // współczynnik wzmocnienia (zaniku) światła stożkowego

  int iaux1, iaux2;   // pola pomocnicze dla kolejnych rozszerzeń
  float faux1, faux2;
  void *ptr;
} LIGHTV2, *LIGHTV2_PTR;
```

Nowe pola, modyfikujące dotychczasową postać struktury i wprowadzające nowe zmienne dla pozycji i orientacji źródła światła, zostały wyróżnione wytłuszczeniem. Jedyna różnica w obsłudze tych struktur polegać będzie na tym, że lokalna pozycja i orientacja źródła światła przechowywane będą w polach `pos` i `dir`, ale dodatkowo w polach `tpos` i `tdir` zapisane zostaną współrzędne źródła względem współrzędnych przestrzeni kamery. Pytanie, w jaki sposób dokonać przekształcenia współrzędnych źródła światła? Cóż, oto funkcja, która się tym zajmie:

```
void Transform_LIGHTSV2(LIGHTV2_PTR lights, // tablica źródeł światła
                        int num_Lights,     // liczba źródeł światła w tablicy
                        MATRIX4X4_PTR mt,   // macierz przekształcenia
                        int coord_select)   // wskazanie kierunku transformacji
{
  // funkcja przekształca współrzędne przekazanych źródeł światła na podstawie
  // macierzy przekształcenia; funkcja służy do umieszczenia źródeł światła
  // w przestrzeni kamery, tak, aby obliczenia oświetlenia dawały poprawne wyniki
  // również wtedy, gdy będą wywoływane już po przeniesieniu wielokątów do
  // przestrzeni kamery; później dokonamy pewnych optymalizacji wykorzystujących
  // informacje o typie źródła światła (i unikniemy niepotrzebnych obrotów),
  // choć przy tysiącach wierzchołków przekształcenie kilku źródeł światła nie
  // wpływa na wydajność
  // UWAGA: funkcja ta musi zostać wywołana nawet wtedy, jeżeli przekształcenie
  // współrzędnych źródeł światła nie jest pożądane, czyli w przypadku
  // realizowania oświetlenia w przestrzeni sceny - w takim przypadku funkcja
  // zajmuje się skopiowaniem do pól roboczych lokalnych wartości współrzędnych
  // i orientacji; funkcję należy wtedy wywołać ze znacznikiem
  // TRANSFORM_COPY_LOCAL_TO_TRANS, a jako macierz przekształceń przekazać NULL

  int curr_light; // indeks bieżącego źródła światła
  MATRIX4X4 mr;  // macierz wykorzystywana do konfigurowania aspektu obrotu

  // wektory orientacji źródła światła muszą zostać obrócone, ale przedtem
  // należy wyzerować w macierzy przekształcenia współczynniki przesunięcia;
  // w przeciwnym przypadku wynik przekształcenia będzie nieprawidłowy
  if (mt != NULL)
  {
    MAT_COPY_4X4(mt, &mr);

    // wyzeruj współczynniki przesunięcia
    mr.M30 = mr.M31 = mr.M32 = 0;
  }
```

```
// jakie współrzędne powinny zostać poddane przekształceniom?
switch(coord_select)
{
  case TRANSFORM_COPY_LOCAL_TO_TRANS:
  {
    // dla wszystkich źródeł światła w tablicy:
    for (curr_light = 0; curr_light < num_lights; curr_light++)
    {
      lights[curr_light].tpos = lights[curr_light].pos;
      lights[curr_light].tdir = lights[curr_light].dir;
    } // pętla for
  } break;
  case TRANSFORM_LOCAL_ONLY:
  {
    // dla wszystkich źródeł światła w tablicy:
    for (curr_light = 0; curr_light < num_lights; curr_light++)
    {
      // przekształć współrzędne lokalne (sceny)
      POINT4D presult;      // przechowuje wynik kolejnych transformacji

      // przekształć pozycję każdego źródła światła
      Mat_Mul_VECTOR4D_4X4(&lights[curr_light].pos, mt, &presult);

      // zapisz wynik
      VECTOR4D_COPY(&lights[curr_light].pos, &presult);

      // przekształć wektory orientacji źródła światła
      Mat_Mul_VECTOR4D_4X4(&lights[curr_light].dir, &mr, &presult);

      // zapisz wynik
      VECTOR4D_COPY(&lights[curr_light].dir, &presult);
    } // pętla for
  } break;
  case TRANSFORM_TRANS_ONLY:
  {
    // dla wszystkich źródeł światła w tablicy:
    for (curr_light = 0; curr_light < num_lights; curr_light++)
    {
      // przekształć każde z „przekształconych" źródeł światła
      POINT4D presult;      // przechowuje wynik kolejnych transformacji

      // przekształć współrzędne pozycji źródła światła
      Mat_Mul_VECTOR4D_4X4(&lights[curr_light].tpos, mt, &presult);

      // zapisz wynik
      VECTOR4D_COPY(&lights[curr_light].tpos, &presult);

      // przekształć wektory orientacji źródła światła
      Mat_Mul_VECTOR4D_4X4(&lights[curr_light].tdir, &mr, &presult);

      // zapisz wynik
      VECTOR4D_COPY(&lights[curr_light].tdir, &presult);
    } // pętla for
  } break;
  case TRANSFORM_LOCAL_TO_TRANS:
  {
    // dla wszystkich źródeł światła w tablicy:
    for (curr_light = 0; curr_light < num_lights; curr_light++)
    {
      // przekształć każde źródło światła i umieść wyniki przekształcenia
      // w polach współrzędnych przekształconych - funkcja będzie najczęściej
      // wywoływana właśnie w tym trybie
      POINT4D presult;      // przechowuje wynik kolejnych transformacji

      // przekształć współrzędne pozycji źródła światła
      Mat_Mul_VECTOR4D_4X4(&lights[curr_light].pos, mt, &lights[curr_light].tpos);
```

```
        // przekształć wektor orientacji źródła światła
        Mat_Mul_VECTOR4D_4X4(&lights[curr_light].dir, &mr, &lights[curr_light].tdir);
      } // pętla for
    } break;
    default: break;
  } // konstrukcja switch
} // Transform_LIQHTSV2
```

Korzystanie z funkcji sprowadza się do jej wywołania i przekazania za pośrednictwem argumentów tablicy źródeł światła, liczby świateł w tablicy, macierzy przekształceń i wyboru rodzaju przekształcania. Funkcja ta jest typową funkcją realizującą przekształcenia na obiektach, a jej działanie nie różni się od działania innych tego typu funkcji. W dziewięćdziesięciu dziewięciu wywołaniach na sto składnia wywołania będzie następująca:

```
Transform_LIGHTSV2(lights2, 4, &cam.mcam, TRANSFORM_LOCAL_TO_TRANS);
```

Wywołanie to powoduje przekształcenie współrzędnych czterech źródeł światła zebranych w tablicy `lights2` względem macierzy przekształceń `cam.mcam`, przy czym transformacja powinna dokonać przekształcenia lokalnych współrzędnych źródeł światła do współrzędnych przestrzeni kamery. Nowe współrzędne umieszczone zostaną w polach współrzędnych transformowanych. Dysonując już nowymi strukturami opisującymi źródła światła i sposobem ich wypełniania należałoby pozmieniać w pliku *T3DLIB.CPP* i *T3DLIB.H* deklaracje odwołujące się do poprzednich wersji struktury:

```
LIGHTV2 lights2[MAX_LIGHTS]; // źródła światła sceny
```

W pliku nagłówkowym trzeba umieścić nowe definicje stałych dla źródeł światła:

```
// definicje dla drugiej wersji struktury źródła światła
#define LIGHTV2_ATTR_AMBIENT     0x0001  // proste światło otaczające
#define LIGHTV2_ATTR_INFINITE    0x0002  // nieskończone źródło światła
#define LIGHTV2_ATTR_DIRECTIONAL 0x0002  // nieskończone źródło światła (alias)

#define LIGHTV2_ATTR_POINT       0x0004 // punktowe źródło światła
#define LIGHTV2_ATTR_SPOTLIGHT1  0x0008 // stożkowe źródło światła typ 1 (prosty)
#define LIGHTV2_ATTR_SPOTLIGHT2  0x0010 // stożkowe źródło światła typ 2 (złożony)
#define LIGHTV2_STATE_ON              1 // światła włączone
#define LIGHTV2_STATE_OFF             0 // światła wyłączone

// znaczniki sterujące kierunkiem transformacji
#define TRANSFORM_COPY_LOCAL_TO_TRANS 3 // kopiowanie bez przekształcania
```

Nowe stałe mają dokładnie te same wartości, co analogiczne stałe definiowane na potrzeby struktury `LIGHTV1`, różnice sprowadzają się do zmiany jednego znaku w przedrostku nazwy stałej. Całkiem nowa jest za to wartość znacznika sterującego transformacją współrzędnych źródła światła: `TRANSFORM_COPY_LOCAL_TO_TRANS`. Jej przekazanie do funkcji transformacji współrzędnych źródeł światła powoduje proste skopiowanie współrzędnych lokalnych do pól współrzędnych transformowanych.

Czeka nas jeszcze żmudna czynność przepisania wszystkich funkcji obliczających oświetlenie. W żadnej z nich nie będzie trzeba wprowadzać znaczących zmian. Poza zmianą nazwy struktury modyfikacje sprowadzą się do wykorzystywania w obliczeniach nie pól `pos` i `dir`, a pól `tpos` i `tdir`. Z racji obszerności kodu wymagającego poprawek nie zostanie tu zamieszczony pełny wydruk poprawianych funkcji; wymienione zostaną za to ich prototypy:

```
int Light_OBJECT4DV2_World2_16(OBJECT4DV2_PTR obj, // przetwarzany obiekt
     CAM4DV1_PTR cam,     // położenie kamery
     LIGHTV2_PTR lights, // lista źródeł światła (może zawierać wiele świateł)
     int max_lights);     // maksymalna liczba świateł na liście

int Light_OBJECT4DV2_World2(OBJECT4DV2_PTR obj, // przetwarzany obiekt
     CAM4DV1_PTR cam,     // położenie kamery
     LIGHTV2_PTR lights, // lista źródeł światła (może zawierać wiele świateł)
     int max_lights);     // maksymalna liczba świateł na liście
```

```
int Light_RENDERLIST4DV2_World2(RENDERLIST4DV2_PTR rend_list, // lista wielokątów
     CAM4Dv1_PTR cam,     // położenie kamery
     LIGHTV2_PTR lights, // lista źródeł światła (może zawierać wiele świateł)
     int max_lights);     // maksymalna liczba świateł na liście

int Light_RENDERLIST4DV2_World2_16(RENDERLIST4DV2_PTR rend_list, // lista wielokątów
     CAM4DV1_PTR cam,     // położenie kamery
     LIGHTV2_PTR lights, // lista źródeł światła (może zawierać wiele świateł)
     int max_lights);     // maksymalna liczba świateł na liście

// system oświetlenia
int Init_Light_LIGHTV2(LIGHTV2_PTR lights, // tablica źródeł światła
     int      index,      // indeks tworzonego źródła światła (0 .. MAX_LIGHTS - 1)
     int     _state,      // stan źródła światła
     int     _attr,       // typ źródła światła, z dodatkowymi kwalifikatorami
     RGBAV1  _c_ambient, // intensywność źródła otaczającego
     RGBAV1  _c_diffuse, // intensywność źródła rozproszonego
     RGBAV1  _c_specular,// intensywność źródła zwierciadlanego
     POINT4D_PTR  _pos,  // pozycja źródła światła
     VECTOR4D_PTR _dir,  // orientacja źródła światła
     float        _kc,   // współczynniki tłumienia
     float        _kl,
     float        _kq,
     float        _spot_inner, // wewnętrzny kąt stożka
     float        _spot_outer, // zewnętrzny kąt stożka
     float        _pf);        // współczynnik wzmocnienia (zaniku) stożka

int Reset_Lights_LIGHTV2(LIGHTV2_PTR lights, // tablica źródeł światła
     int max_lights);                        // liczba świateł w systemie
```

Jedyna różnica pomiędzy nowymi a poprzednimi wersjami tych funkcji polega na wprowadzeniu do nich jednego nowego parametru: wskaźnika do tablicy źródeł światła. Dzięki temu system oświetlenia może operować na więcej niż jednym zestawie świateł systemu.

Nadając źródłu światła pozycję bądź orientację należy zadbać, aby ostatni parametr przekazywanego punktu *POINT4D* (przy nadawaniu pozycji) bądź wektora *VECTOR4D* (przy nadawaniu orientacji) miał wartość 1,0. W przeciwnym przypadku przy transformowaniu współrzędnych źródła światła współczynnik przesunięcia macierzy przekształceń nie będzie miał wpływu na wynik przekształcenia!

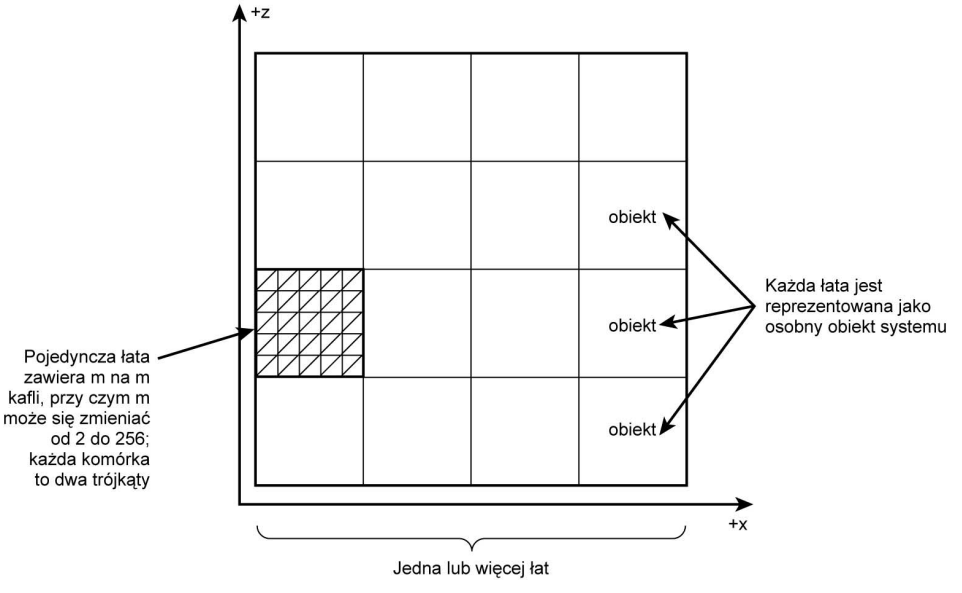
W ramach testu nowych wersji funkcji oświetlenia wystarczy uruchomić znane Wam już programy demonstracyjne. Głównym celem przekonstruowania funkcji oświetlenia było umożliwienie realizacji obliczeń związanych z oświetleniem również w przestrzeni kamery, w celu minimalizacji liczby oświetlanych wielokątów.

Zabawa w terenie

Przyczyn implementowania modułów przycinających jest wiele, ale motywem głównym jest chęć zminimalizowania liczby wielokątów przekazywanych do dalszych etapów potoku przetwarzania i zagwarantowanie, że żaden z wielokątów o współrzędnej z równej 0 nie zostanie przekazany do rzutowania, co mogłoby spowodować błędy w obliczeniach rzutu perspektywicznego. Jak dotychczas, przycinanie testowaliśmy w pustych scenach z niewielką liczbą obiektów, pozbawionych wielkich wielokątów i geometrii wnętrza (np. pomieszczeń), przez co ryzyko, że dojdzie do sytuacji potencjalnie prowokujących błędy rzutowania, było niemal zerowe. Wciąż nie mamy możliwości przetestowania przycinania w scenach z architekturą wnętrz bądź z obszernymi krajobrazami, jak więc możemy oceniać jakość i efektywność przycinania? Ten dotkliwy brak sprowokował mnie do napisania prostego silnika tworzącego wizualizowany teren z wykorzystaniem techniki kafelkowania terenu, tworzonego na podstawie mapy wysokości.

Implementacja rodzi dwa problemy. Po pierwsze, nie dysponujemy żadną strukturą do przechowywania obiektów poza `OBJECT4DV2`, która raczej nie jest przystosowana do reprezentowania wielkich obszarów, ale na razie będzie musiała wystarczyć. Prawdziwym problemem jest jednak fakt, że generowane techniką

kafelkowania tereny składają się w najlepszym razie z tysięcy, a w najgorszym nawet z milionów wielokątów. Jednak przy podejściu zdroworozsądkowym i zastosowaniu algorytmu sektoryzacji terenu jest nadzieja, że uda się wyeliminować nawet 99 procent wielokątów! Dzięki podziałowi na sektory czy też łaty można tworzyć olbrzymie tereny, układając sąsiadująco kolejne łaty terenu i przetwarzając wyłącznie te z nich, które znajdują się w danej chwili w polu widzenia. Każda taka łata terenu będzie pewnym osobnym obiektem, przechowywanym wraz z innymi obiektami łat w tablicy terenu — patrz rysunek 10.24.



Rysunek 10.24. *Przykład reprezentowania rozległego terenu po jego podziale na łaty*

Do uzyskania rozległego terenu można wykorzystać wiele takich obiektów, a następnie za pomocą metody usuwania ze sceny całych obiektów eliminować te łaty terenu, które nie są w danej chwili widoczne i nie muszą być przetwarzane. Nie usuwa to jednak wszystkich trudności — nawet pojedyncza łata może zawierać 32 na 32, 64 na 64, albo nawet 256 na 256 wartości wysokości, a więc składać się z 31 na 31, 63 na 63 bądź nawet 255 na 255 kafelków (liczba kafelków jest zawsze o jeden mniejsza od liczby wartości wysokości w danej łacie, a to dlatego, że jedna z wartości wysokości, leżąca na graniczy dwóch łat, jest dla nich wspólna). Stąd w pojedynczej łacie może znajdować się $31^2 * 2$, $63^2 * 2$ albo nawet $255^2 * 2$ trójkątów!

Tak więc, mimo usunięcia większości obiektów liczba wielokątów do przetworzenia jest porażająca. Porządny silnik wizualizujący trójwymiarowy obraz terenu wymaga realizowanej w czasie rzeczywistym analizy pewnego poziomu szczegółowości obrazowania terenu, pozwalającej na zmniejszenie liczby wielokątów w jednym, a zwiększenie jej w innym miejscu terenu. W efekcie dla obrazowania pojedynczej łaty, składającej się z 64 na 64 kafli, po dwa trójkąty na kafel (razem 8192 trójkątów!), można uzyskać redukcję liczby wielokątów do zaledwie kilkuset! Nie będziemy tu jednak opisywać algorytmów realizujących taką analizę, gdyż nie temu poświęcony jest bieżący rozdział.

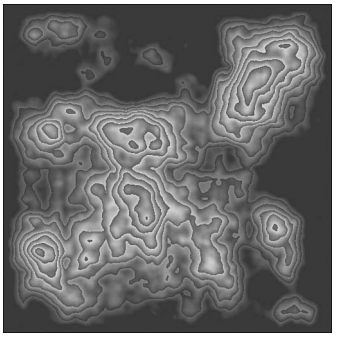
Na razie należałoby powiedzieć, w jaki sposób utworzyć prosty teren, który można by wygenerować za pomocą pojedynczego wywołania funkcji i reprezentować w systemie jako standardowy obiekt, który podda się obróbce w znany już nam sposób.

Funkcja generowania terenu

Trójwymiarowy teren można wygenerować na co najmniej kilka sposobów: wylosować wartości wysokości, wykorzystać dostępne w Internecie dane satelitarne, dokonać konwersji z mapy wysokości kodowanych kolorami i tak dalej. Tutaj pokażemy, jak wygenerować teren na podstawie mapy wysokości kodowanych

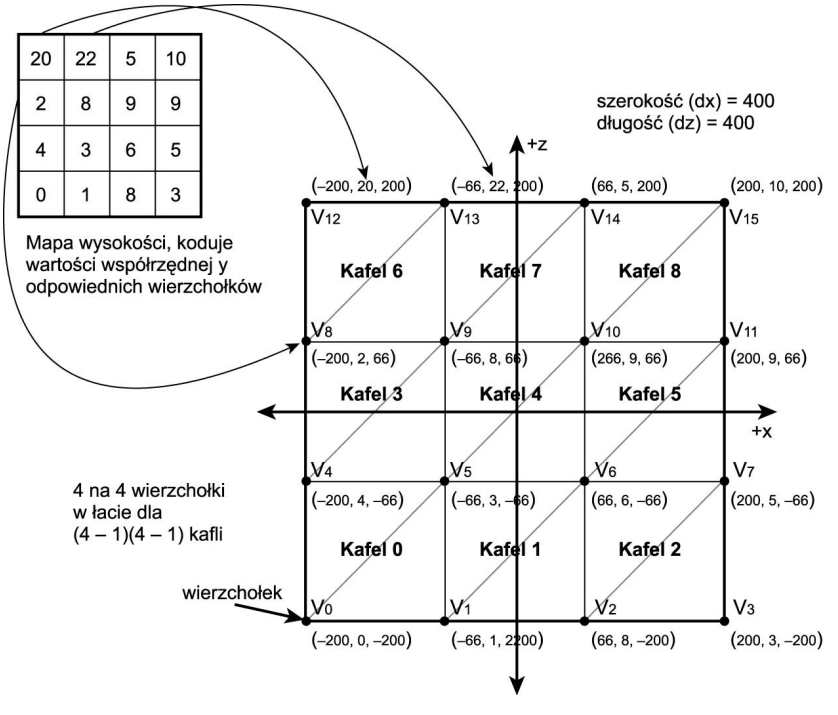
kolorami. Plan działania zakłada wczytanie 256-kolorowego obrazka, w którym poszczególne wartości kolorów kodują wysokości w odpowiednich punktach modelowanego terenu oraz wczytanie obrazka z teksturą, która zostanie nałożona na szkielet terenu. Przykładowy obrazek (rzecz jasna mocno zmniejszony) z mapą wysokości kodowaną kolorami widnieje na rysunku 10.25. Zobaczmy, jak różne kolory pikseli obrazka mogą reprezentować różnice w wysokości sąsiadujących ze sobą kafli terenu.

Rysunek 10.25. *Rysunek rastrowy nadający się do wykorzystania w roli mapy wysokości*



Mapa wysokości powinna być interpretowana w sposób następujący: wygenerowana zostanie regularna siatka, ze środkiem w początku układu współrzędnych sceny, po czym na siatkę tę nałożona zostanie macierz wartości wysokości; każda wartość wysokości będzie proporcjonalna do wartości koloru odpowiedniego piksela mapy wysokości. Dla przykładu, kolor o indeksie 0 oznacza najniższą możliwą wysokość punktu terenu, kolor o indeksie 255 oznacza zaś najwyższy z punktów terenu. Przez odpowiednie przeskalowanie wartości indeksów koloru można dowolnie zwiększać zakres wysokości. Graficzną ilustrację mapy wysokości o rozmiarze 4 na 4 piksele i sposób generowania na jej podstawie macierzy wysokości przedstawia rysunek 10.26.

Rysunek 10.26. *Sposób konstruowania siatki wysokości na podstawie mapy wysokości kodowanych kolorami*



Generowanie siatki wysokości sprowadza się, jak pokazuje rysunek 10.26, do skonstruowania (przy zachowaniu ostrożności w obliczaniu zakresu) kilku pętli `for`. Algorytm generujący tworzy dla każdego kwadratowego kafla wysokości dwa trójkąty, których wysokość (wartości współrzędnych y wierzchołków) definiowana jest indeksem koloru w mapie wysokości i ewentualnie współczynnikiem skalowania. Konieczne jest również określenie ziarnistości siatki, czyli długości i szerokości (albo szerokości i głębokości, jak kto woli) każdego z kafli. Siatka wysokości będzie bowiem generowana w oparciu o płaszczyznę x-z, a wartości kolorów poszczególnych pikseli będą definiowały wartości współrzędnych y (wysokości) w narożnikach odpowiednich kafli. Następnie dla każdego kafla utworzone zostaną dwa trójkąty, których wierzchołki będą później umieszczone na liście wierzchołków obiektu reprezentującego łatę.

Jak już wspomniano, podczas obliczania zakresu pętli `for` należy uważać na pojawiające się błędy przekroczenia zakresu. Błędy te często popełniane są niejako z rozpędu, bo pierwsza odpowiedź na pytanie o liczbę liczb całkowitych w zakresie od 0 do 10 jest zwykle błędna — w zakresie tym mieści się bowiem nie dziesięć, a 11 liczb! A ile potrzeba linii do wyznaczenia na placu pięciu miejsc parkingowych? Cztery, rzecz jasna. Odpowiedzi 10 albo 5 na powyższe pytania to właśnie błędy przekroczenia zakresu. Poprawne określenie granic dla pętli `for` jest niezwykle istotne podczas operacji takich, jak generowanie siatki, ponieważ zbyt duży zakres może spowodować wędrówkę po pamięci i na przykład zakłócić działanie algorytmu albo przynajmniej wprowadzić do siatki dziwne anomalie. W takich przypadkach do kontroli poprawności przebiegów pętli najlepiej wykorzystać metodę ręcznego wykonania pętli określonej dla małego zakresu, jak 1 na 1, ewentualnie 4 na 4. Wyniki testu można spokojnie uogólnić na dowolnie duże zakresy.

Dalej, ręczne konstruowanie struktury obiektu `OBJECT4DV2` nie jest szczególnie skomplikowane. Po prostu strumień informacji, które normalnie są odczytywane z pliku obiektu, należy wygenerować własnoręcznie. Do tego potrzebna będzie lista wierzchołków, lista wielokątów, numery wielokątów i ich atrybuty. Nic z tego nie powinno sprawić większych trudności, z wyjątkiem może atrybutów wielokątów, a więc informacji o teksturowaniu i oświetleniu. Informacje te muszą zostać przekazane do funkcji generującej za pośrednictwem następujących stałych:

```
// definicje stałych wielokątów, wersja 2.

// atrybuty wielokąta i jego ściany
#define POLY4DV2_ATTR_2SIDED        0x0001
#define POLY4DV2_ATTR_TRANSPARENT   0x0002
#define POLY4DV2_ATTR_8BITCOLOR     0x0004
#define POLY4DV2_ATTR_RGB16         0x0008
#define POLY4DV2_ATTR_RGB24         0x0010

#define POLY4DV2_ATTR_SHADE_MODE_PURE      0x0020
#define POLY4DV2_ATTR_SHADE_MODE_CONSTANT  0x0020 // (alias)
#define POLY4DV2_ATTR_SHADE_MODE_EMISSIVE  0x0020 // (alias)

#define POLY4DV2_ATTR_SHADE_MODE_FLAT      0x0040
#define POLY4DV2_ATTR_SHADE_MODE_GOURAUD   0x0080
#define POLY4DV2_ATTR_SHADE_MODE_PHONG     0x0100
#define POLY4DV2_ATTR_SHADE_MODE_FASTPHONG 0x0100 // (alias)
#define POLY4DV2_ATTR_SHADE_MODE_TEXTURE   0x0200

// nowe
#define POLY4DV2_ATTR_ENABLE_MATERIAL  0x0800 // oświetlanie z materiałem
#define POLY4DV2_ATTR_DISABLE_MATERIAL 0x0800 // oświetlanie z kolorami podstawowymi
```

Interesować będą nas te stałe reprezentujące atrybuty, które zostały wyróżnione wytłuszczeniem. Zatrzymajmy się przy nich przez chwilę. Generator terenu może tworzyć płaszczyzny terenowe w trybach koloru ośmiobitowego bądź szesnastobitowego, dlatego na pewno konieczne będzie ustawienie jednego ze znaczników:

```
POLY4DV2_ATTR_8BITCOLOR
POLY4DV2_ATTR_RGB16
```

Dalej, funkcja generowania terenu będzie potrafiła tworzyć tereny kolorowane jednolicie lub cieniowane płasko bądź metodą Gourauda, ponieważ cieniowanie to nie jest zbyt wymagające i sprowadza się do wyliczenia wektorów normalnych wielokątów. Po utworzeniu siatki wyliczenie wektorów normalnych sprowadza się zaś do wywołania następujących funkcji:

```
// oblicz długości wektorów normalnych wielokątów
Compute_OBJECT4DV2_Poly_Normals(obj);

// dla wielokątów cieniowanych metodą Gourauda oblicz wektory normalne wierzchołków
Compute_OBJECT4DV2_Vertex_Normals(obj);
```

Wywołania te wystarczą do obliczenia wszelkich wielkości potrzebnych do cieniowania metodą Gourauda (konkretnie wektorów normalnych wierzchołków), jak również wyposażą obiekt w dane potrzebne do realizacji algorytmów usuwania niewidocznych, bo zwróconych tyłem do obserwatora, ścian, jak i do oświetlenia wielokątów.

W celu ustawienia trybu cieniowania należy wskazać jedną z następujących stałych:

```
POLY4DV2_ATTR_SHADE_MODE_EMISSIVE
POLY4DV2_ATTR_SHADE_MODE_FLAT
POLY4DV2_ATTR_SHADE_MODE_GOURAUD
```

Gdyby zaś teren miał zostać obłożony teksturą, trzeba by włączyć mapowanie tekstury, a więc przekazać do funkcji generującej teren stałą:

```
POLY4DV2_ATTR_SHADE_MODE_TEXTURE
```

Tyle, że przy włączonym teksturowaniu dostępne tryby cieniowania to cieniowanie płaskie i cieniowanie jednolite, nie można więc włączyć równocześnie teksturowania i cieniowania metodą Gourauda. No dobrze, pora na przykład. Aby utworzyć cieniowany metodą Gourauda, wizualizowany w trybie szesnastobitowego koloru, pozbawiony tekstury teren, należy wykorzystać następującą kombinację stałych:

```
POLY4DV2_ATTR_RGB16 | POLY4DV2_ATTR_SHADE_MODE_GOURAUD
```

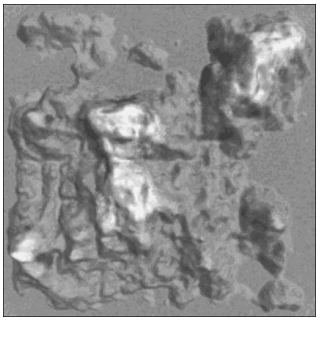
Aby zaś utworzyć teren płasko cieniowany i obłożony teksturą, należy skomponować następującą wartość:

```
POLY4DV2_ATTR_RGB16 | POLY4DV2_ATTR_SHADE_MODE_FLAT | POLY4DV2_ATTR_SHADE_MODE_TEXTURE
```

Jak widać, atrybuty składa się za pośrednictwem logicznej sumy odpowiednich stałych. W porządku, zastanówmy się nad obsługą teksturowania siatki terenu.

Przyjrzyjmy się rysunkowi 10.27. Jest to rysunek, który zostanie wykorzystany do pokrycia terenu teksturą (został wygenerowany przez stosowny program właśnie jako tekstura terenowa, stąd jego odpowiedni wygląd). Rysunek ma rozmiar 256 na 256 pikseli, ale nie jest to wymóg sztywny — rysunki tekstury mogą mieć dowolny rozmiar, byle był on potęgą liczby 2. Tekstura może mieć nawet 8×8 pikseli. Rozmiar 256×256 to największy rozmiar obsługiwanej przez nasz system pojedynczej tekstury.

Rysunek 10.27. *Przykładowa tekstura terenu*



Teksturowanie polega tu na obliczeniu współrzędnych mapowania zadanej tekstury wejściowej przy rozciągnięciu jej na siatkę terenu. Mapowanie to odbywa się przez przypisanie współrzędnych tekstury do poszczególnych wierzchołków siatki. Można by, rzecz jasna, wykorzystać dużo większe tekstury, dopasowując je do wymagań systemu przez podzielenie na 256-pikselowe bloki przypisanie tych bloków do poszczególnych wielokątów, ale na razie wystarczy metoda prostsza, bo przecież nie chodzi nam o pełną

wierność odwzorowania terenu — teren potrzebny jest jedynie do demonstracji działania funkcji przycinania. Tak więc, jeżeli teren ma zostać pokryty teksturą, konieczne będzie przekazanie do funkcji tworzącej teren — obok nazwy pliku mapy wysokości kodowanych kolorami — nazwy pliku rysunku tekstury. Funkcja przeliczy współrzędne mapowania tekstury do obłożenia terenu. Całość nie wydaje się skomplikowana, znów bowiem operować będziemy szerokością, długością i liczbą kafli w siatce terenu.

Podsumowując: jeżeli podczas generowania terenu ma on zostać obłożony teksturą, konieczne będzie przekazanie do funkcji nazwy pliku zawierającego szesnastobitową teksturę. Musi ona być, rzecz jasna, kwadratowa. Funkcja generująca teren dokona odwzorowania wierzchołków kafli w teksturze, obliczając współrzędne mapowania tekstury dla każdego wierzchołka każdego trójkąta każdego kafla.

To tyle na temat funkcji generującej teren. W większości przypadków przyjmuje ona na wejściu dwa pliki. Jeden z nich to kodowana na 256 kolorach mapa wysokości (plik w formacie *.BMP*), która w miarę możliwości powinna być kwadratowa. Zwykle dla mapy wysokości wystarcza rozmiar 40 na 40 pikseli — już i tak oznacza to konstruowanie siatki z co najmniej 39 kafli, a więc z $39^2 * 2$, czyli z 3 042 trójkątów, co w zupełności wystarczy jak na możliwości prostego, programowego silnika graficznego.

Pora na przegląd kodu źródłowego samej funkcji; w dalszej części rozdziału pokazane zostaną również przykłady jej wywołania.

```
int Generate_Terrain_OBJECT4DV2(OBJECT4DV2_PTR obj,       // wskaźnik do obiektu
  float twidth,       // szerokość terenu - wartość osi x we współrzędnych sceny
  float theight,      // długość (głębokość) terenu
  float vscale,       // współczynnik skalowania wartości wysokości
  char *height_map_file,  // nazwa pliku mapy wysokości kodowanych na 256 kolorach
  char *texture_map_file, // nazwa pliku tekstury
  int rgbcolor,       // kolor terenu, jeśli ten nie jest teksturowany
  VECTOR4D_PTR pos,   // początkowa pozycja terenu
  VECTOR4D_PTR dir,   // początkowy kąt obrotu
  int poly_attr)      // atrybuty cieniowania
{
  // funkcja generuje teren o rozmiarze twidth na theight w płaszczyźnie x-z;
  // teren jest definiowany wartościami wysokości kodowanymi jako indeksy
  // kolorów w mapie wysokości, w której kolor o indeksie 0 oznacza poziom
  // minimalny gruntu, a 255 - poziom maksymalny, a więc 1.0. Wysokość może
  // być skalowana współczynnikiem vscale; wygenerowana siatka wysokości
  // złożona będzie z trójkątów, których wierzchołki będą miały współrzędne
  // określone na podstawie mapy wysokości - dla mapy o rozmiarze 256x256
  // siatka wysokości będzie miała rozmiar (255 - 1)x(255 - 1) wielokątów,
  // pokrywających w przestrzeni sceny obszar o rozmiarach twidth na theight;
  // w przypadku określenia pliku tekstury zostanie ona nałożona na teren przez
  // obliczenie jej współrzędnych; funkcja generuje tereny w trybach 8- i
  // 16-bitowego koloru; dla danego trybu koloru należy dobrać odpowiednią
  // głębię koloru tekstury!

  char buffer[256];           // bufor roboczy
  float col_tstep, row_tstep;
  float col_vstep, row_vstep;
  int columns, rows;
  int rgbwhite;
  BITMAP_FILE height_bitmap;  // mapa wysokości

  // krok 1. wyzeruj i zainicjalizuj obiekt
  memset(obj, 0, sizeof(OBJECT4DV2));

  // ustaw stan obiektu na aktywny i widoczny
  obj->state = OBJECT4DV2_STATE_ACTIVE | OBJECT4DV2_STATE_VISIBLE;

  // ustaw pozycję obiektu
  obj->world_pos.x = pos->x;
  obj->world_pos.y = pos->y;
  obj->world_pos.z = pos->z;
  obj->world_pos.w = pos->w;
```

```
// utwórz odpowiednie słowo koloru, uwzględniając wskazaną głębię koloru terenu
// słowo rgbcolor zapisuje się zawsze w formacie rgb5.6.5, więc dla terenu
// 8-bitowego wystarczy je zredukować
if (poly_attr & POLY4DV1_ATTR_8BITCOLOR)
{
  rgbcolor = rgblookup[rgbcolor];
  rgbwhite = rgblookup[RGB16Bit(255,255,255)];
} // if
else
{
  rgbwhite = RGB16Bit(255,255,255) ;
} // else

// ustaw liczbę ramek
obj->num_frames = 1;
obj->curr_frame = 0;
obj->attr = OBJECT4DV2_ATTR_SINGLE_FRAME;

// wyzeruj bitmapę
memset(&height_bitmap, 0, sizeof(BITMAP_FILE));
memset(&bitmap16bit, 0, sizeof(BITMAP_FILE));

// krok 2. wczytaj mapę wysokości
Load_Bitmap_File(&height_bitmap, height_map_file);

// przelicz podstawowe dane o bitmapie
columns = height_bitmap.bitmapinfoheader.biWidth;
rows  = height_bitmap.bitmapinfoheader.biHeight;

col_vstep = twidth / (float)(columns - 1);
row_vstep = theight / (float) (rows - 1);

sprintf(obj->name ,"Teren:%s%s", height_map_file, texture_map_file);
obj->num_vertices = columns * rows;
obj->num_polys  = ((columns - 1) * (rows - 1) ) * 2;

// zapisz niektóre informacje potrzebne przy wizualizacji
// - skorzystaj ze zmiennych pomocniczych obiektu!
obj->ivar1 = columns;
obj->ivar2 = rows;
obj->fvar1 = col_vstep;
obj->fvar2 = row_vstep;

// przydziel pamięć dla wierzchołków i liczby wielokątów
// parametry wywołania są tu nadmiarowe, ale to nic
if (!Init_OBJECT4DV2(obj,  // obiekt przydziału
          obj->num_vertices,
          obj->num_polys,
          obj->num_frames))
{
  Write_Error("\nBłąd generatora terenu (nie można przydzielić pamięci).");
} // if


// krok 3. załaduj teksturę (jeśli przekazano)
if ( (poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE) && texture_map_file)
{
  // wczytaj plik tekstury
  Load_Bitmap_File(&bitmap16bit, texture_map_file);

  // utwórz strukturę mapy bitowej o odpowiedniej głębi koloru i rozmiarze
  obj->texture = (BITMAP_IMAGE_PTR) malloc(sizeof(BITMAP_IMAGE));
  Create_Bitmap(obj->texture, 0, 0,
         Bitmap16bit.bitmapinfoheader.biWidth,
         bitmap16bit.bitmapinfoheader.biHeight,
         bitmap16bit.bitmapinfoheader.biBitCount);
```

```
  // wczytaj rysunek tekstury (później można dodać obsługę różnych głębi koloru)
  if (obj->texture->bpp == 16)
  Load_Image_Bitmap16(obj->texture,
       &bitmap16bit, 0, 0, BITMAP_EXTRACT_MODE_ABS);
  else
  {
    Load_Image_Bitmap(obj->texture,
         &bitmap16bit, 0, 0, BITMAP_EXTRACT_MODE_ABS);
  } // else 8 bit

  // oblicz współczynniki krokowe mapowania współrzędnych tekstury
  col_tstep = (float)(bitmap16bit.bitmapinfoheader.biWidth - 1) /
    (float)(columns - 1);
  row_tstep = (float)(bitmap16bit.bitmapinfoheader.biHeight - 1) /
    (float)(rows - 1);

  // oznacz obiekt jako posiadający teksturę
  SET_BIT(obj->attr, OBJECT4DV2_ATTR_TEXTURES);

  // zrobione, zwolnij bitmapę
  Unload_Bitmap_File(&bitmap16bit);
} // if

Write_Error("\ncolumns = %d, rows = %d", columns, rows);
Write_Error("\ncol_vstep = %f, row_vstep = %f", col_vstep, row_vstep);
Write_Error("\ncol_tstep = %f, row_tstep = %f", col_tstep, row_tstep);
Write_Error("\nnum_vertices = %d, num_polys = %d",
obj->num_vertices, obj->num_polys);

// krok 4. generowanie listy wierzchołków i współrzędnych tekstury
// po kolejnych wierszach
for (int curr_row = 0; curr_row < rows; curr_row++)
{
  for (int curr_col = 0; curr_col < columns; curr_col++)
  {
    int vertex = (curr_row * columns) + curr_col;

    // oblicz współrzędne wierzchołka
    obj->vlist_local[vertex].x = curr_col * col_vstep - (twidth/2);
    obj->vlist_local[vertex].y = vscale *
      ((float)height_bitmap.buffer[curr_col + (curr_row * columns)]) / 255;
    obj->vlist_local[vertex].z = curr_row * row_vstep - (theight/2);

    obj->vlist_local[vertex].w = 1;

    // każdy wierzchołek posiada przynajmniej atrybut punktu
    SET_BIT(obj->vlist_local[vertex].attr, VERTEX4DTV1_ATTR_POINT) ;

    // czy wierzchołek potrzebuje współrzędnych tekstury?
    if ( (poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE) && texture_map_file)
    {

      // współrzędne tekstury
      obj->tlist[vertex].x = curr_col * col_tstep;
      obj->tlist[vertex].y = curr_row * row_tstep;
    } // if

    Write_Error("\nWierzchołek %d: V[%f, %f, %f], T[%f, %f]",
          vertex, obj->vlist_local[vertex].x,
          obj->vlist_local[vertex].y,
          obj->vlist_local[vertex].z,
          obj->tlist[vertex].x,
          obj->tlist[vertex].y);
  } // for dla curr_col
} // for dla curr_row
```

```
// obliczyć transformację obrotu?

// oblicz średni i maksymalny promień obiektu
Compute_OBJECT4DV2_Radius(obj);

Write_Error("\nŚredni promień obiektu = %f, maksymalny = %f",
      obj->avg_radius[0], obj->max_radius[0]);

// krok 5. utwórz listę wielokątów
for (int poly=0; poly < obj->num_polys / 2; poly++)
{
  // po dwa wielokąty na kafel, tworzone wierszami;
  // wyszukiwanie odpowiednich indeksów wierzchołków wielokątów to żmudna
  // procedura: dla danej tablicy wierzchołków m na n należy utworzyć listę
  // wielokątów o rozmiarach (m-1) na (n-1), po dwa wielokąty na kafel i to
  // w odpowiedniej kolejności wierzchołków w wielokącie... można
  // wykorzystać indeksy albo wykonać podwójną pętlę...

  int base_poly_index = (poly % (columns - 1)) +
           (columns * (poly / (columns - 1)) );

  // lewy górny wielokąt kafla
  obj->plist[poly * 2].vert[0] = base_poly_index;
  obj->plist[poly * 2].vert[1] = base_poly_index + columns;
  obj->plist[poly * 2].vert[2] = base_poly_index + columns + 1;

  // dolny prawy wielokąt kafla
  obj->plist[poly * 2 + 1].vert[0] = base_poly_index;
  obj->plist[poly * 2 + 1].vert[1] = base_poly_index + columns + 1;
  obj->plist[poly * 2 + 1].vert[2] = base_poly_index + 1;

  // lista wierzchołków jest nadmiarowa, gdyż taka lista zawarta jest
  // w strukturze obiektu - do użytkownika należy wybór, czy do konstruowania
  // współrzędnych wielokąta wykorzystać wierzchołki lokalne, czy po
  // transformacji; być może dobrze byłoby ustawić listę na NULL
  obj->plist[poly * 2].vlist = obj->vlist_local;
  obj->plist[poly * 2 + 1].vlist = obj->vlist_local;

  // ustaw atrybuty wielokąta zgodnie z przekazanymi atrybutami
  obj->plist[poly * 2].attr = poly_attr;
  obj->plist[poly * 2 + 1].attr = poly_attr;

  // testy upewniające o zgodności drugorzędnych elementów danych

  // ustawienie koloru wielokąta
  obj->plist[poly * 2].color = rgbcolor;
  obj->plist[poly * 2 + 1].color = rgbcolor;

  // sprawdź, czy włączono cieniowanie - jeżeli tak, trzeba obliczyć
  // wektory normalne
  if ( (obj->plist[poly * 2].attr & POLY4DV2_ATTR_SHADE_MODE_GOURAUD) ||
       (obj->plist[poly * 2].attr & POLY4DV2_ATTR_SHADE_MODE_PHONG) )
  {
    // wierzchołki tego wielokąta wymagają obliczenia wektorów normalnych
    // - należy je odpowiednio oznaczyć
    SET_BIT(obj->vlist_local[ obj->plist[poly * 2].vert[0] ].attr,
        VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[ obj->plist[poly * 2].vert[1] ].attr,
       VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[ obj->plist[poly * 2].vert[2] ].attr,
       VERTEX4DTV1_ATTR_NORMAL);

    SET_BIT(obj->vlist_local[ obj ->plist[poly * 2 + 1 ] .vert[0] ].attr,
       VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[ obj->plist[poly * 2 + 1].vert[1] ].attr,
        VERTEX4DTV1_ATTR_NORMAL);
```

```
      SET_BIT(obj->vlist_local[ obj->plist[poly * 2 + 1].vert[2] ].attr,
          VERTEX4DTV1_ATTR_NORMAL);
    } // if

    // jeżeli włączono teksturowanie, włącz znacznik teksturowania
    if (poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
    {
      // obłóż wielokąt teksturą
      obj->plist[poly * 2].texture = obj->texture;
      obj->plist[poly * 2 + 1].texture = obj->texture;

      // przypisz współrzędne tekstury

      // lewy górny wielokąt
      obj->plist[poly * 2].text[0] = base_poly_index;
      obj->plist[poly * 2].text[1] = base_poly_index + columns;
      obj->plist[poly * 2].text[2] = base_poly_index + columns + 1;

      // prawy dolny wielokąt
      obj->plist[poly * 2 + 1].text[0] = base_poly_index;
      obj->plist[poly * 2 + 1].text[1] = base_poly_index + columns + 1;
      obj->plist[poly*2+1].text[2] = base_poly_index+1;

      // zastąp kolor bazowy wielokąta bardziej refleksyjnym
      obj->plist[poly * 2].color = rgbwhite;
      obj->plist[poly * 2 + 1].color = rgbwhite;

      // ustaw atrybuty współrzędnych tekstury
      SET_BIT(obj->vlist_local[ obj->plist[poly * 2].vert[0] ].attr,
         VERTEX4DTV1_ATTR_TEXTURE);
      SET_BIT(obj->vlist_local[ obj->plist[poly * 2].vert[1] ].attr,
          VERTEX4DTV1_ATTR_TEXTURE);
      SET_BIT(obj->vlist_local[ obj->plist[poly * 2].vert[2] ].attr,
         VERTEX4DTV1_ATTR_TEXTURE);

      SET_BIT(obj->vlist_local[ obj->plist[poly * 2 + 1].vert[0] ].attr,
          VERTEX4DTV1_ATTR_TEXTURE);
      SET_BIT(obj->vlist_local[ obj->plist[poly * 2 + 1].vert[1] ].attr,
         VERTEX4DTV1_ATTR_TEXTURE);
      SET_BIT(obj->vlist_local[ obj->plist[poly * 2 + 1].vert[2] ].attr,
          VERTEX4DTV1_ATTR_TEXTURE);
    } // if

    // ustaw rodzaj materiału do emulacji wersji 1.0
    SET_BIT(obj->plist[poly * 2].attr, POLY4DV2_ATTR_DISABLE_MATERIAL);
    SET_BIT(obj->plist[poly * 2 + 1].attr, POLY4DV2_ATTR_DISABLE_MATERIAL);

    // oznacz wielokąt jako aktywny
    obj->plist[poly * 2].state = POLY4DV2_STATE_ACTIVE;
    obj->plist[poly * 2 + 1].state = POLY4DV2_STATE_ACTIVE;

    // lista wierzchołków jest nadmiarowa, gdyż taka lista zawarta jest
    // w strukturze obiektu - do użytkownika należy wybór, czy do konstruowania
    // współrzędnych wielokąta wykorzystać wierzchołki lokalne, czy po
    // transformacji; być może dobrze byłoby ustawić listę na NULL
    obj->plist[poly * 2].vlist = obj->vlist_local;
    obj->plist[poly * 2 + 1].vlist = obj->vlist_local;

    // ustaw listę współrzędnych tekstury
    obj->plist[poly * 2].tlist = obj->tlist;
    obj->plist[poly * 2 + 1].tlist = obj->tlist;
  } // for poly

#if 0
  for (poly; poly < obj->num_polys; poly++)
  {
    Write_Error("\nWielokąt %d: Vi[%d, %d, %d], Ti[%d, %d, %d]", poly,
    obj->plist[poly].vert[0],
```

```
      obj->plist[poly].vert[1],
      obj->plist[poly].vert[2],
      obj->plist[poly].text[0],
      obj->plist[poly].text[1],
      obj->plist[poly].text[2]);
    } // end
#endif

    // oblicz długości wektorów normalnych wielokątów
    Compute_OBJECT4DV2_Poly_Normals(obj);

    // dla wielokątów cieniowanych metodą Gourauda oblicz wektory normalne wierzchołków
    Compute_OBJECT4DV2_Vertex_Normals(obj);

    // zwróć kod prawidłowego wykonania funkcji
    return(1);

} // Generate_Terrain_OBJECT4DV2
```

Funkcja została zaimplementowana w oparciu o moduł wczytujący pliki *.PLG*, a to dlatego, że moduł ten dysponuje możliwością konstruowania obiektu `OBJECT4DV2`; proces konstruowania obiektu został rozszerzony tylko o procedurę generowania terenu. Spójrzmy teraz, w jaki sposób można korzystać z tak zdefiniowanej funkcji. Poniższe wywołanie generuje teren o rozmiarze 32×32, obłożony teksturą, który zostanie rozciągnięty na obszarze o rozmiarze 4000×4000 punktów w przestrzeni sceny. Mapa wysokości zapisana jest w pliku *EARTHHEIGHTMAP01.BMP*, a tekstura — rysunek o rozmiarze 256 na 256 pikseli i 16-bitowej głębi koloru — w pliku *EARTHCOLORMAP01.BMP*:

```
VECTOR4D terrain_pos = {0, 0, 0, 0};
int Generate_Terrain_OBJECT4DV2(&obj_terrain,       // wskaźnik do obiektu
  4000,             // szerokość terenu - wartość osi x we współrzędnych sceny
  4000,             // długość (głębokość) terenu
  700,              // współczynnik skalowania wartości wysokości
  "earthheightmap01.bmp", // nazwa pliku mapy wysokości kodowanych na 256 kolorach
  "earthcolormap01.bmp",  // nazwa pliku tekstury
  RGB16Bit(255,255,255),  // kolor terenu, jeśli ten nie jest teksturowany
  &terrain_pos,           // początkowa pozycja terenu
  NULL,             // początkowy obrót
  POLY4DV2_ATTR_RGB16 | POLY4DV2_ATTR_SHADE_MODE_FLAT |
  POLY4DV2_ATTR_SHADE_MODE_TEXTURE);
```

Znaczenie parametrów przekazywanych do funkcji nie wymaga chyba szerszego komentarza — do funkcji przekazywany jest docelowy rozmiar terenu we współrzędnych przestrzeni sceny, współczynnik skalowania wysokości (wszystkie wartości mapy wysokości zostaną przemnożone przez ten współczynnik, więc najwyższą wysokością będzie 700). Dalej przekazywane są nazwy plików mapy wysokości i tekstury (gdyby teren nie miał być teksturowany, w miejsce nazwy pliku tekstury należałoby przekazać `NULL`). Następny parametr to kolor terenu (przy włączonym teksturowaniu parametr ten jest ignorowany), następnie przekazywana jest początkowa pozycja terenu we współrzędnych sceny (tutaj jest to punkt o współrzędnych (0, 0, 0)), początkowy obrót i zestaw atrybutów wielokątów (tutaj atrybuty dla płaskiego cieniowania wielokątów i trybu 16-bitowej głębi koloru).

Obiektów reprezentujących teren można tworzyć dowolnie dużo, a przez ich odpowiednie łączenie tworzyć rozległe światy, na przykład składające się z 16 na 16 takich obiektów; taka liczba łat wymaga już usuwania części obiektów z potoku przetwarzania. Jednakże — jak już powiedziano — zadaniem tworzonego programu jest jedynie demonstracja obsługi dużych wielokątów wymagających przycinania, a to zadanie spełnia on znakomicie.

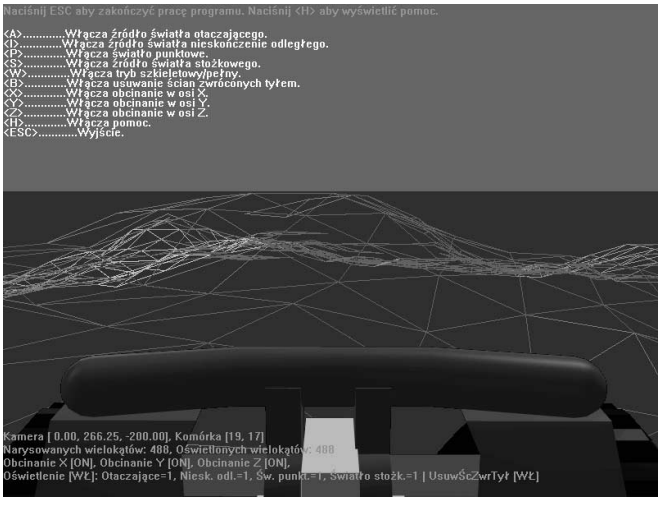
Generowanie mapy wysokości

Dotychczasowy sposób konstruowania terenu jest świetny, ale może istnieje sposób samodzielnego jego wygenerowania, bez podpierania się gotową mapą wysokości? Cóż, naturalnie można spróbować napisać specjalny program. Duże znaczenie ma też odpowiedni dobór tekstury — czapy śnieżne powinny zalegać na szczytach, a nie w depresjach, gdzie powinna królować woda. Większość ludzi korzysta jednak z gotowych

programów generujących teren. Potrafią one wygenerować zarówno plik rysunku z mapą wysokości, jak i odpowiedni plik tekstury. Jednym z takich programów jest *VistaPro*, który doczekał się już bodaj wersji 4.0.1. Licencja tego programu została odsprzedana wielu producentom, więc jeśli ktoś chce go znaleźć, najlepiej zasięgnąć języka w jednej z wyszukiwarek internetowych. Innym programem generującym tereny jest Bryce autorstwa zespołu Metacreations, jednak również ta firma została już sprzedana, a ich produkty przejął Corel. Nie są to, rzecz jasna, jedyne tego typu programy — w sieci są ich dziesiątki, w tym wiele darmowych.

Rajd łazikiem terenowym

W porządku, wypadałoby wreszcie połączyć generator terenu i moduł przycinania i sprawdzić, jak to wszystko działa! W tym celu utworzony został program demonstracyjny (jego kod źródłowy znajduje się w pliku *DEMOII10_2.CPP*, a wersja wykonywalna — w pliku *DEMOII10_2.EXE*; wersja działająca w 8-bitowej głębi koloru to odpowiednio pliki *DEMOII10_2_8b.CPP* i *DEMOII10_2_8b.EXE*). Demonstracja ma postać symulacji jazdy po pustynnej wyspie — patrz rysunki 10.28 i 10.29. Do kompilacji plików niezbędny jest główny plik kodu źródłowego, pliki od *T3DLIB1.CPP* do *T3DLIB8.CPP* (wraz z odpowiednimi plikami nagłówkowymi) oraz pliki *.LIB* biblioteki DirectX.



Rysunek 10.28. *Zrzut ekranowy — program demonstracyjny działający w trybie wizualizacji szkieletowej*

Rysunek 10.29. *Zrzut ekranowy — program demonstracyjny działający w trybie wizualizacji teksturowanej*

Program jest, jak zwykle w przypadku programów demonstracyjnych, bardzo uproszczony. Za pomocą pojedynczego wywołania tworzony jest obiekt terenu (parametry wywołania funkcji generującej są zgodne z tymi prezentowanymi w poprzednich punktach), po czym nisko nad terenem ustawiana jest kamera, której ruchem można sterować tak, jakby obserwator jeździł po wydmach samochodem terenowym. Przed przystąpieniem do zabawy mam jednak jeszcze kilka wyjaśnień.

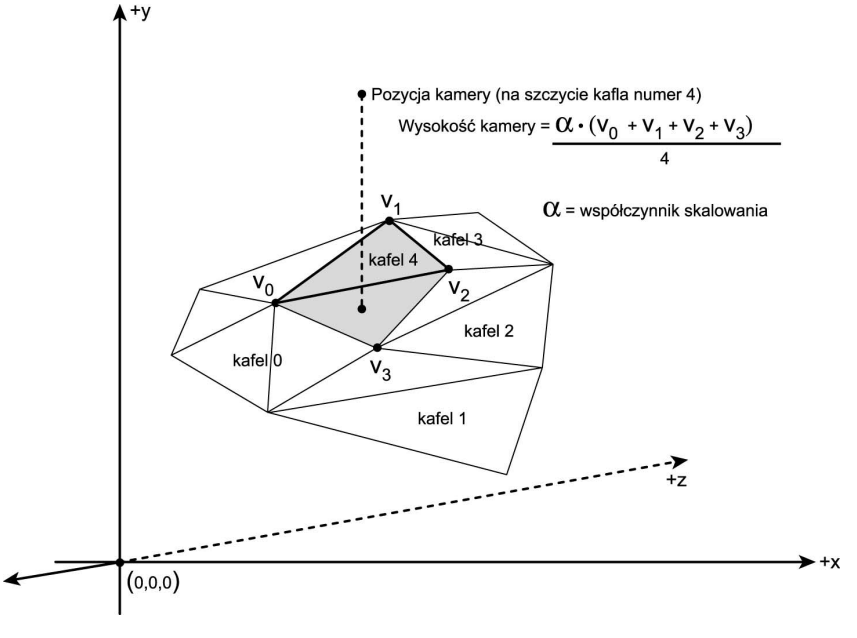
Klawisze sterujące wyświetlane są w ramach ekranu pomocy włączanego klawiszem *H*, ale wystarczy zapamiętać, że przycinanie 3D względem poszczególnych płaszczyzn włącza się i wyłącza klawiszami *X*, *Y* i *Z*. Jeżdżąc wokół wyspy warto zwrócić uwagę na jakość działania przycinania w pobliżu dolnej krawędzi ekranu i przyjrzeć się zmianom w obrazowaniu po włączeniu (i wyłączeniu) przycinania względem poszczególnych płaszczyzn.

Po wyłączeniu przycinania względem płaszczyzny z (a więc między innymi względem bliższej płaszczyzny przycinania) można doprowadzić do zawieszenia programu w wyniku przetwarzania i rzutowania dużych wielokątów, przenikających przez płaszczyznę z = 0. Nie trzeba się naturalnie obawiać, że zawieszenie programu czy jego awaryjne zamknięcie spowoduje trwałe uszkodzenie komputera! W celu lepszej obserwacji efektów przycinania warto włączyć tryb szkieletowy (służy do tego klawisz *W*).

Algorytm poruszania się po terenie

Algorytm wizualizacji terenu i prosty model fizyczny pozwalający obserwatorowi prowadzić wirtualny pojazd poruszający się po wydmach teksturowanej wyspy są powiązane, dlatego zostaną omówione razem. Problem definiowany jest następująco: dana jest kamera, która reprezentuje widok z perspektywy prowadzącego łazik (widok jest oczywiście częściowo przesłonięty „deską rozdzielczą" łazika). Prowadzący-gracz musi mieć możliwość poruszania się nad terenem bez „wjeżdżania" we wzniesienia, stąd konieczność przeliczania wysokości terenu pod kamerą i odpowiednie przesuwanie w pionie pozycji obserwatora. Ilustracją problemu jest rysunek 10.30.

Rysunek 10.30. *Poruszanie się po terenie*



Tak postawione zadanie można jak zwykle wykonać na szereg sposobów. Można by na przykład zaimplementować w programie jak najbardziej zbliżony do rzeczywistego model fizyczny, uwzględniający nawet bieżący pęd łazika, ale byłaby to przesada w przypadku prostego programu demonstracyjnego. Można by również przesadzić w drugą stronę i ustawiać kamerę w najwyższym punkcie danego kafla, ale ruch

pojazdu byłby wtedy bardzo mało rzeczywisty, stąd decyzja o implementacji rozwiązania pośredniego, a więc o uwzględnieniu w modelu fizycznym prędkości, grawitacji i chwilowego przyspieszenia kamery (ponieważ kamera jest dla gracza tym samym, co pojazd, pozycja kamery będzie niekiedy nazywana pozycją łazika i odwrotnie). Oto główne założenia modelu fizycznego:

- Pojazd posiada wektor prędkości — wartość prędkości i jej kierunek, kontrolowane klawiszami kursora (podczas naciskania klawiszy strzałki w przód bądź w tył na pojazd działa stałe przyspieszenie).
- Na pojazd stale działa skierowana w dół „siła przyciągania ziemskiego"; wynikiem jej uwzględnienia jest nadanie pojazdowi przyspieszenia opadania, zwiększającego się w funkcji czasu.
- Na pojazd działa skierowana w górę siła wynosząca go na poziom gruntu; jeżeli pojazd kiedykolwiek przekroczy poziom morza, zostanie wypchnięty w górę.
- W każdym momencie bieżąca pozycja pojazdu w przestrzeni sceny jest wykorzystywana do obliczenia numeru kafla, nad którym porusza się pojazd, a następnie obliczana jest średnia wysokość wierzchołków kafla. Obliczona w ten sposób średnia wysokość kafla jest porównywana z bieżącą wysokością pojazdu, a jeżeli ta ostatnia jest mniejsza niż wysokość terenu, prędkość wznoszenia pojazdu jest zwiększana z chwilowym przyspieszeniem proporcjonalnym do różnicy wysokości. Dodatkowo wysokość pojazdu jest ograniczana tak, aby po wykryciu urwiska o dużym nachyleniu pojazd skokowo zmieniał wysokość tak, aby nie „wbić się" w ścianę urwiska.
- Poza ingerowaniem w wysokość pojazdu poruszającego się po terenie, wraz ze zmianą gradientu wysokości terenu zmieniane jest nachylenie kamery. Daje to iluzję podnoszenia i opuszczania przodu pojazdu zgodnie z nachyleniem zboczy.
- Na koniec model fizyczny jest stabilizowany tak, że w przypadku, kiedy zmiany pozycji lub orientacji są niewielkie i oscylujące, drgania są wygaszane.

Do uwzględnienia tych założeń trzeba będzie zdefiniować kilka stałych fizycznych symulacji:

```
// definicje dla terenu
#define TERRAIN_WIDTH    4000
#define TERRAIN_HEIGHT   4000
#define TERRAIN_SCALE     700
#define MAX_SPEED          20

// definicje dla modelu fizycznego
// - ich zmiana zmienia charakterystykę jezdną pojazdu
float gravity = -.40;      // ciążenie powszechne
float vel_y = 0;           // prędkość pionowa kamery

float cam_speed = 0;     // prędkość kamery
float sea_level = 50;    // poziom morza symulacji
float gclearance =75;    // odstęp kamery od gruntu
float neutral_pitch - 10;  // neutralne nachylenie kamery
```

A oto kod realizujący całość symulacji poruszania się po terenie:

```
// sekcja ruchu pojazdu //
// śledzenie pozycji w terenie polega na określeniu bieżącego kafla i indeksów
// do listy wierzchołków w celu odnalezienia czterech wierzchołków kafla, nad
// którym znajduje się pojazd; po obliczeniu średniej wartości wysokości kafla,
// w zależności od bieżącej wysokości pojazdu i terenu pojazd jest podnoszony


// podczas generowania terenu w strukturze obiektu terenu zapisane zostały
// przydatne informacje dodatkowe:
// ivar1 = columns;
// ivar2 = rows;
// fvar1 - col_vstep;
// fvar2 = row_vstep;

int cell_x = (cam.pos.x + TERRAIN_WIDTH / 2) / obj_terrain.fvar1;
int cell_y = (cam.pos.z + TERRAIN_HEIGHT / 2) / obj_terrain.fvar1;
static float terrain_height, delta;
```

```
// sprawdź, czy pojazd znajduje się w granicach terenu
if ( (cell_x >= 0) && (cell_x < obj_terrain.ivar1) &&
     (cell_y >= 0) && (cell_y < obj_terrain.ivar2) )
{
  // wyznacz indeksy wierzchołków tworzących kafel
  int v0 = cell_x + cell_y * obj_terrain.ivar2;
  int v1 = v0 + 1;
  int v2 = v1 + obj_terrain.ivar2;
  int v3 = v0 + obj_terrain.ivar2;

  // oblicz średnią wysokość kafla
  terrain_height = 0.25 * (obj_terrain.vlist_trans[v0].y +
         obj_terrain.vlist_trans(v1].y +
         obj_terrain.vlist_trans[v2].y +
         obj_terrain.vlist_trans[v3].y) ;

  // oblicz różnicę wysokości kafla i pojazdu
  delta = terrain_height - (cam.pos.y - gclearance);

  // test „wbicia" pojazdu w grunt
  if (delta > 0)
  {
    // natychmiastowe zadziałanie na kamerę
    vel_y += (delta * (0.025));

    // pojazd wbity w grunt, trzeba go podnieść
    cam.pos.y += (delta * 0.3);

    // to już raczej magia niż model fizyczny :) podnieś lub opuść przód pojazdu
    // w zależności od prędkości pojazdu i nachylenia zbocza
    cam.dir.x -= (delta * 0.015);
  } // if
} // if

// hamowanie kamery
if (cam_speed > (0.25) ) cam_speed -= 0.25;
else
if (cam_speed < (-0.25) ) cam_speed += 0.25;
else
  cam_speed = 0;

// dźwięki
DSound_Set_Freq(car_sound_id, 8000 + fabs(cam_speed) * 250);

// stabilizacja orientacji kamery
if (cam.dir.x > (neutral_pitch + 0.3)) cam.dir.x -= (0.3);
else
if (cam.dir.x < (neutral_pitch - 0.3)) cam.dir.x += (0.3);
else
  cam.dir.x = neutral_pitch;

// uwzględnij grawitację
vel_y += gravity;

// test położenia poniżej poziomu morza
if (cam.pos.y < sea_level)
{
  vel_y = 0;
  cam.pos.y = sea_level;
}

// przesuń kamerę
cam.pos.x += cam_speed * Fast_Sin(cam.dir.y);
cam.pos.z += cam_speed * Fast_Sin(cam.dir.y);
cam.pos.y += vel_y;
```

To naprawdę bardzo prosty model — jedynymi danymi wejściowymi są bieżące współrzędne kamery.

To już koniec opisu tego programu demonstracyjnego. Teraz powinniście samodzielnie pobawić się zmiennymi definiującymi fizykę — efektem może być albo wrażenie poruszania się stutonowym czołgiem, albo lekkim skuterem wodnym. Przy odrobinie inwencji własnej można też rozszerzyć program, dodając do niego różne obiekty, by na przykład stworzyć proste wyścigi samochodowe, wyścigi odrzutowców czy konkurs zjazdu na desce snowboardowej — ograniczeniem jest chyba wyłącznie wyobraźnia. Utworzenie programu demonstracyjnego dla deski snowboardowej byłoby naprawdę proste. Wystarczy utworzyć 16 bądź 32 łaty terenu, każda po 32 na 32 kafle, każda zajmująca, dajmy na to, 2000 na 2000 punktów przestrzeni sceny.Następnie należałoby połączyć łaty tak, aby utworzyły jeden duży stok. Moduł usuwający obiekty usuwałby łaty znajdujące się poza zasięgiem wzroku (poza daleką płaszczyzną przycinania), a moduł przycinający zająłby się resztą; przed kamerą wystarczyoby już tylko umieścić wizerunek deski, zmienić mapę wysokości i szusować!

Podsumowanie

Dobrze, że rozdział ten mamy już za sobą — ciągłe ustawianie bliższej płaszczyzny przycinania w odległości roku świetlnego od obserwatora, w celu uniknięcia potencjalnych błędów i niedokładności rzutowania, było już doprawdy męczące. Teraz rzeczy mają się zupełnie inaczej. Poza tym zyskaliście pewne pojęcie o algorytmach przycinania i, co najważniejsze, chyba przekonaliście się, że nie są one tak skomplikowane, jakby się mogło wydawać — wystarczy pamiętać o kilku szczegółach. Dodanie mechanizmu usuwania w całości wielokątów znajdujących się poza daleką płaszczyzną przycinania lub na zewnątrz bryły ściętego ostrosłupa widzenia naprawdę przyspiesza potok, dzięki czemu nasz silnik graficzny jest jeszcze szybszy. Wreszcie jako premię zyskaliśmy prostą grę terenową, którą można przerobić na wyścigi terenowe czy wodne — nie będą to może wyścigi porównywalne z V-Rally, ale jak na początek chyba i tak okazałe. No i — co też ma swoje znaczenie — był to ostatni rozdział, w którym zajmowaliśmy się jeszcze obsługą trybów ośmiobitowych.