POLITECHNIKA WARSZAWSKA – WYDZIAŁ MATEMATYKI I TECHNIK INFORMACYJNYCH

Wyświetlanie sceny złożonej z dużej ilości wielokątów

Michał Rostocki

Promotor: Dr Jerzy Wojciechowski

Warszawa, Grudzień 2002

Spis treści:

1. Wprowadzenie

Obecnie grafika komputerowa znajduje najbogatsze zastosowanie w grach komputerowych. Ich twórcy od zawsze pragnęli stworzyć doskonałą iluzję drugiego świata. Aby tego dokonać potrzebowali narzędzi mogących przedstawić inną rzeczywistość w jak najbardziej wiarygodny sposób. Często realizm przedstawianego świata w dużej mierze decyduje o atrakcyjności gry. Praca ta ma za zadanie przybliżyć jedną z obecnie stosowanych metod obrazowania złożonego wirtualnego środowiska graficznego.

1.1. Definicje

Płaszczyzna – płaszczyzna jest zdefiniowana przez normalną oraz odległość od początku układu współrzędnych, normalna pozwala na określenie czy dowolny punkt jest za, przed lub na płaszczyźnie. W pracy płaszczyzna będzie często równoważna normalnej lub wielokątowi, który na niej leży.

Wielokąt – zbiór punktów leżących na jednej płaszczyźnie, tworzących wielokąt wypukły. Dwa wielokąty leżące na tej samej płaszczyźnie, posiadające tą samą teksturę, wspólną krawędź i tworzące wielokąt wypukły można zoptymalizować zamieniając na jeden wielokąt. Operacja optymalizacji będzie bardzo często używana gdyż zmniejsza ostateczną liczbę wielokątów w scenie. Najmniejszym wielokątem jest trójkąt.

Normalna – wersor prostopadły do płaszczyzny zawierającej wielokąt, jednoznacznie określający widoczną stronę wielokąta.

Tekstura – mapa bitowa używana do wypełniania rysowanego wielokąta wzorem.

Solid - wielościan foremny zdefiniowany przez listę płaszczyzn oraz ich tekstur.

Drzewo – rekurencyjna struktura danych, w pracy będą występować drzewa binarne czyli takie, które w każdym węźle zawierają informacje tylko o dwóch potomkach. Liście drzewa są równoważne sektorom, na które jest podzielona scena.

Portal – wielokąt łączący przylegające do siebie dwa sektory sceny.

Łata (patch) – niepodzielny kubiczny obszar przestrzeni określający kolor i stopień jasności w danym miejscu.

Mapa światła (lightmap) – tekstura przypisana do wielokąta wygenerowana na podstawie łat zawartych w tym wielokącie. Wielokąt może posiadać teksturę i mapę światła jednocześnie, wynik takiego złożenia polega na pomnożeniu składowych kolorów przez siebie, funkcja ta jest realizowana przez niektóre akceleratory graficzne.

Scena – zbiór wielokątów i solidów definiujący pewne trójwymiarowe środowisko. Obserwator może przemieszczać się po scenie. Jest ona na tyle duża, że niemożliwe jest jej całkowite przedstawienie z powodu zbyt dużej ilości wielokątów potrzebnych do narysowania.

1.2. Sformułowanie zadania

Moce obliczeniowe procesorów nieustannie rosną a karty graficzne są w stanie wyświetlać coraz więcej wielokątów w czasie rzeczywistym. Jednak wciąż, realizm nawet najnowszych gier pozostawia wiele do życzenia. Często przedstawiane sceny składają się z dużej ilości trójkątów a narysowanie każdego z nich pochłania cenny czas. Wyświetlenie wszystkich wielokątów bez jakiegokolwiek preprocessingu nie pozwoliłoby na zachowanie płynności animacji. Stąd bierze się potrzeba redukcji ilości rysowanych elementów bez straty jakości przedstawianego obrazu. W tej pracy zostanie przedstawiony algorytm umożliwiający znaczne przyspieszenie czasu rysowania sceny złożonej z dużej ilości wielokątów przez wcześniejsze zbadanie ich wzajemnego zasłaniania. Rodzaj sceny i jej ograniczenia zostaną przedstawione przy opisie algorytmu.

1.3. Proste rozwiązania

Przed opisem głównego algorytmu należy powiedzieć o elementarnych metodach przyspieszania czasu rysowania dużej ilości wielokątów.

1.3.1. Normalne

Najbardziej podstawowa z metod polega na porównywaniu wektora normalnego rysowanego wielokąta z kierunkiem patrzenia obserwatora. Jeśli kąt miedzy wektorami nie zawiera się w przedziale od –90 do 90 stopni oznacza to że obserwator widzi tył wielokąta. Przy rysowaniu obiektów wypukłych nie należy rysować wielokątów odwróconych tyłem gdyż i tak zostaną one zasłonięte (np. obserwując kulę zawsze widzimy tylko jej połowę). W pracy tej wszystkie wielokąty są widoczne tylko z jednej strony.

1.3.2. Z-Bufor

Większość kart graficznych posiada wbudowaną obsługę Z-bufora czyli bufora głębokości. W owym buforze zostaje zapamiętana odległość każdego narysowanego piksela od obserwatora. Dzięki temu na ekranie można narysować elementy znajdujące się tylko przed tymi, które zostały wcześniej narysowane. Metoda ta posiada jednak pewne ograniczenia i np. nie pozwala na rysowanie obiektów półprzezroczystych przed (w sensie czasu) nieprzezroczystymi.

1.3.3. Stożek widzenia (Frustrum culling)

Ta metoda pozwala na proste odrzucenie całych obiektów lub ich części przez sprawdzenie czy znajdują się one w stożku widzenia. Stożek ten jest najczęściej definiowany przez pozycje obserwatora, jego kierunek patrzenia i kąt pola widzenia. Stożek widzenia można reprezentować jako nieskończony stożek lub graniastosłup opisany pięcioma płaszczyznami. Nakład obliczeń potrzebnych do sprawdzenia czy dany obiekt znajduje się poza stożkiem jest minimalny gdyż wystarczy sprawdzać czy w stożku zawiera się bryła w której w całkowicie znajduje się rysowany obiekt. Bryłami takimi są najczęściej kule i prostopadłościany.

1.4. Zaawansowane rozwiązania

Opisane wyżej metody są automatycznie realizowane przez większość akceleratorów graficznych. Rozwiązania opisane niżej są bardziej skomplikowane i wymagają oddzielnego, bardziej szczegółowego traktowania.

1.4.1. Poziom detalu (LOD – level of detail)

Ponieważ obiekty znajdujące się daleko od obserwatora w rzucie perspektywicznym są mniejsze mogą być przybliżone mniej dokładnymi odpowiednikami, które zarówno zajmują mniej pamięci jak i pochłaniają mniej czasu przy rasteryzacji. Technika ta zarówno dotyczy obiektów trójwymiarowych jak i tekstur. Niestety wymaga ona wcześniejszego przygotowania odpowiednich modeli szkieletowych gdyż wszelkie automatyczne algorytmy często dają niepożądane skutki. Skrajną odmianą tej metody jest tzw. billboarding –

zastępowanie odległych lub płaskich przedmiotów jedynie prostokątem z teksturą. Metoda ta idealnie nadaje się do reprezentacji obiektów, które obserwowane pod różnymi kątami wyglądają podobnie (np. drzew pod warunkiem, że obserwator nie znajduje się nad nimi).

1.4.2. Portale

Jeśli scena zostanie podzielona na sektory do których zostaną przypisane portale to jednoznacznie zdefiniują one to co widzi obserwator znajdujący się w danym sektorze. Prosty algorytm wizualizacji będzie polegał na znalezieniu sektora z obserwatorem. Ograniczeniu stożka widzenia do każdego z portali i dalszym rysowaniu wszystkich sektorów. Ponieważ stożek widzenia będzie się zmieniał możliwe będzie odrzucanie zasłoniętych elementów sceny. Metoda ta niestety nie uwzględnia wielokrotnego „wchodzenia” do sąsiadujących sektorów co przy dużej liczbie portali między dwoma sektorami może znacznie wpłynąć na czas rysowania sceny. Aby móc skorzystać z tej metody potrzeba podzielić scenę na sektory i znaleźć portale. Opisany niżej algorytm właśnie tym się zajmuje.

2. Opis Algorytmu

Opisany w pracy algorytm polega na odpowiednim przygotowaniu sceny z dużej ilości wielokątów. Realizowane jest to podobnie jak w przypadku opisanego wcześniej sposobu z wykorzystaniem portali. Scena najpierw jest podzielona na sektory. Dla każdego sektora znalezione zostają jego portale oraz lista reszty widocznych sektorów (lista widoczności). Późniejsza wizualizacja nie wymaga już korzystania z portali ponieważ każdy sektor posiada listę wszystkich sektorów które są widoczne.

2.1. Ogólny opis

2.2. Zastosowania

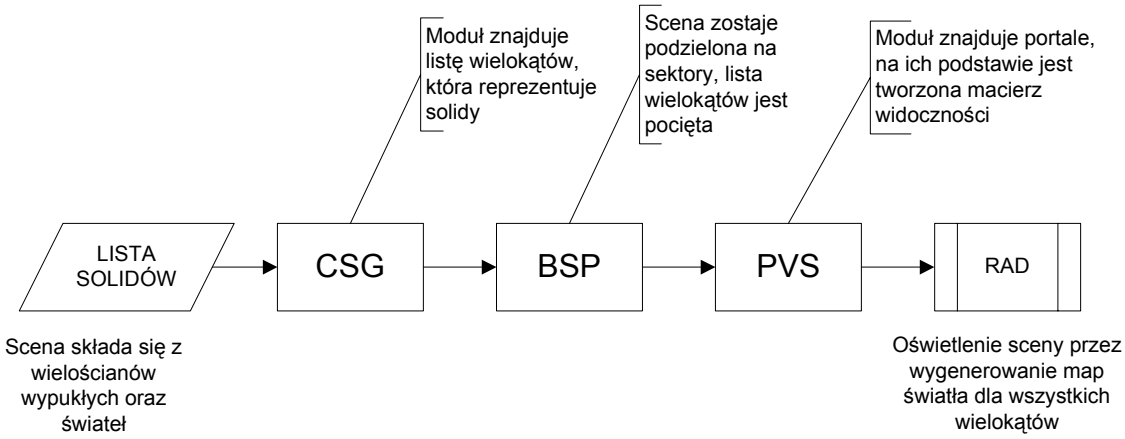
Opisywana metoda może być zastosowana tylko dla specyficznych scen. Dwie najbardziej istotne cechy to statyczny charakter sceny oraz ograniczona widoczność. Scena nie może ona się zmieniać w trakcie wizualizacji, choć możliwe jest oczywiście dodanie innych dynamicznych elementów, jednak nie zostają one uwzględnione w algorytmie. Druga cecha to wysoki stopień samo zasłaniania się sceny. Idealnym przykładem może być labirynt, gdzie z jednego pomieszczenia widać tylko kilka sąsiadujących. Sceną dla której algorytm będzie całkowicie nieprzydatny będzie np. duży teren, las lub łąka. Wizualizowana scena powinna być duża w sensie porównania do skali obserwatora, w innym wypadku zaistnieje potrzeba narysowania jednocześnie całej sceny gdzie algorytm będzie zupełnie nieprzydatny. Rodzaj opisywanych scen ma zastosowanie w aplikacjach typu interactive walk-through czyli scen po których przemieszcza się obserwator. Techniki te są używane przy wizualizacjach architektonicznych oraz bardzo często w grach komputerowych.

Sama scena powinna być przygotowana w odpowiedni sposób. Podstawowym elementem sceny jest ‘solid’ wielościan wypukły (np. sześcian czy czworościan). Każda ściana solida posiada informacje o teksturze. Tworzenie sceny za pomocą solidów jest bardzo łatwe, daje informację o gęstości sceny która jest konieczna w celu wyznaczania kolizji.

Poza możliwością szybkiego wizualizowania sceny. Algorytm definiuje widoczność na przestrzeni sceny, która może znacznie przyspieszyć m.in. proces znajdowania oświetlenia.

2.3. Moduły

Cały proces znajdowania listy widoczności został podzielony na 3 moduły.



Rys.1

2.3.1. CSG – Constructive Solid Geometry

Constructive Solid Geometry jest zbiorem metod służących do tworzenia obiektów za pomocą tzw. prymitywów – brył prostych oraz elementarnych operacji na tych bryłach, takich jak dodawanie, odejmowanie, scalanie i części wspólne.

2.3.1.1. Ogólny opis modułu

Pierwszym etapem obliczeń jest wczytanie informacji o scenie. Jest ona opisana za pomocą listy wielościanów wypukłych. Wielościany jednoznacznie definiują „gęste" miejsca sceny, które są niedostępne dla obserwatora, nie byłoby to możliwe w przypadku opisania sceny za pomocą listy trójkątów, gdyż te posiadają tylko informacje o powierzchniach a nie o przestrzeni. Ten moduł ma za zadanie stworzenie listy wielokątów opisującej scenę. Z listy tej zostaną wyeliminowanie wielokąty niewidoczne a wielokąty zawierające się w innych zostaną zoptymalizowane.

2.3.1.2. Solidy – bryły – wielościany wypukłe

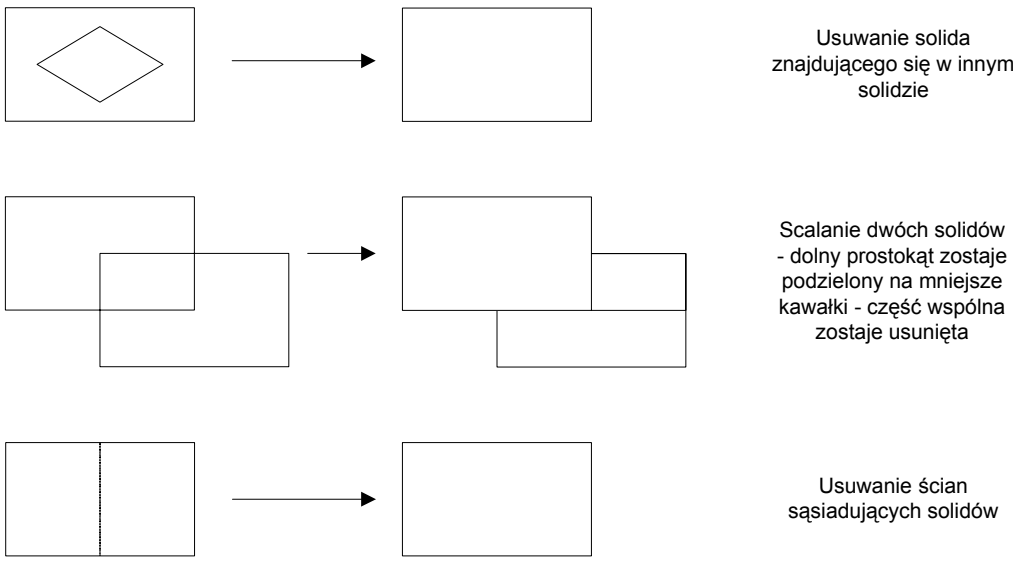
Solid czyli wielościan wypukły jest opisany za pomocą listy płaszczyzn. Lista ta jest prawidłowa jeśli jej płaszczyzny nie powtarzają się i rzeczywiście tworzą wypukły twór. Stworzenie opisywanego solida można przedstawić jak obcinanie z gigantycznego sześcianu części leżących przed każdą z płaszczyzn z listy. Moduł CSG zamienia wielościany wypukłe na odpowiadające im listy wielokątów. Jest to realizowane w dwóch krokach. Najpierw dla każdej płaszczyzny jest tworzony bardzo duży prostokąt, który na niej leży. Średnica prostokąta powinna być dwukrotnie większa od średnicy sceny. Następnie każda płaszczyzna obcina od wszystkich prostokątów fragmenty leżące przed nią. Prostokąt powstały z danej płaszczyzny jest pomijany ponieważ ich normalne są współliniowe. Powstałe wielokąty są wielokątami wypukłymi i ostatecznie opisują kształt solida.

Solid opisuje przestrzeń w formie *implicite* – wystarczy sprawdzany punkt porównać z wszystkimi wielokątami tworzącymi dany solid. Jeśli punkt znajduje się choć przed jedną płaszczyzną leży on na zewnątrz solida w przeciwnym przypadku (leży za każdą z płaszczyzn) punkt należy do „gęstego" obszaru solida. Wielokąt wypukły należy do solida jeśli wszystkie jego punkty leżą w środku.

Do każdego wielokąta tworzącego solid można przypisać teksturę.

2.3.1.3. Dodawanie solidów

Dodawanie solidów jest istotną operacją, dzięki której można znacznie zredukować ilość wielokątów sceny. Polega na usunięciu części wspólnych, lub w szczególnych przypadkach scalenie wielokątów stykających się dwóch solidów. Aby scalić dwa solidy należy najpierw pociąć tworzące je wielokąty płaszczyznami wielokątów drugiego solida. Przecinanie nie następuje gdy normalne są współliniowe. Każde przecięcie dzieli wielokąt na dwie wypukłe części. Teraz solidy są opisane przez nowe listy wielokątów. Jeśli jakiś z wielokątów należy do drugiego solida należy go usunąć z listy. Ostatecznym krokiem jest optymalizacja pozostałych list wielokątów. W przypadku gdy po tej operacji lista jednego z wielościanów jest pusta oznacza to, że całkowicie zawiera się on w drugim solidzie i należy go usunąć.



Rys.2

2.3.1.4. Cechy sceny po CSG

Lista powstała przez dodanie wielokątów z wszystkich ocalałych po scalaniu solidów opisuje scenę tylko za pomocą wielokątów. Dla dowolnych dwóch wielokątów wiadomo, że nie posiadają one wspólnej powierzchni oraz nie przecinają się. Dodatkowo lista ta nie ma wielokątów należących do gęstych obszarów sceny.

2.3.1.5. Pseudokod modułu

```
funkcja stworzenie-listy-wielokątów(lista solidów)
{
      zapamiętaj tekstury;
            zapamiętaj powierzchnie;
      dla (każdego solida)
      stwórz listę wielokątów, które go tworzą;
            dla (każdej z list)
      odejmij części należące do innych solidów;
             zapamiętaj widoczne części wspólne solidów;
      optymalizuj wielokąty
      {
                  łącz wielokąty o wspólnych krawędziach;
                  kasuj wielokąty zawierające się w innych;
      }
      usuń solidy o pustych listach;
}
```

2.3.2. BSP – Binary Space Partitioning

Drzewa BSP są uniwersalną strukturą organizującą dane. Nadają się one idealnie do magazynowania danych, które będą często przeszukiwane. Dzięki temu złożoność wszelkich procesów można z liniowej zamienić na logarytmiczną.

2.3.2.1. Ogólny opis modułu

Moduł BSP na podstawie listy wielokątów dzieli scenę na sektory a wynik końcowy zostaje umieszczony w drzewie binarnym. Lista ta zostanie poważnie zmodyfikowana podczas tworzenia drzewa BSP (binary space partitioning), ponieważ wiele z wielokątów zostanie pociętych na mniejsze kawałki. Mimo zwielokrotnienia całkowitej liczby wielokątów krok ten jest opłacalny.

2.3.2.2. Struktura drzewa

Drzewo opisujące scenę jest drzewem binarnym i składa się z liści i węzłów.

Liście są węzłami bez potomków i są one utożsamiane z sektorami, na które została podzielona scena. Każdy liść zawiera listę wielokątów i portali danego sektora, oraz rodzaj wypukłości (patrz: rodzaje liści). Wielokąty w liściu nie przecinają się. Jest to wyjątkową zaletą gdyż wielokąty każdego sektora można narysować w dowolnej kolejności bez wykorzystania Z-bufora (konieczne natomiast jest użycie normalnych). Przestrzeń sektora jest ograniczona przez wszystkie płaszczyzny z rodziców liścia i jest również wielościanem wypukłym.

Węzły nie zawierają wielokątów ani portali. Posiadają natomiast informację o płaszczyźnie podziału, rozmiarze całego poddrzewa i potomkach. Dwaj potomkowie węzła leżą odpowiednio całkowicie przed lub za płaszczyzną podziału w sensie ich wielokątów.

Cała przestrzeń sceny jest podzielona na sektory według płaszczyzn węzłów drzewa. Aby sprawdzić w jakim sektorze mieści się dany punkt należy rekurencyjnie sprawdzić po której stronie płaszczyzny danego węzła się on znajduje. Jeśli znajduje się on na płaszczyźnie danego węzła należy wybrać dowolnego z potomków. Jeśli z przodu lub z tyłu należy

kontynuować szukanie odpowiednio w przednim bądź tylnim poddrzewie. Zakończenie szukania następuje po dotarciu do liścia. Pierwszym sprawdzanym węzłem jest korzeń drzewa.

Pseudokod znalezienia liścia:

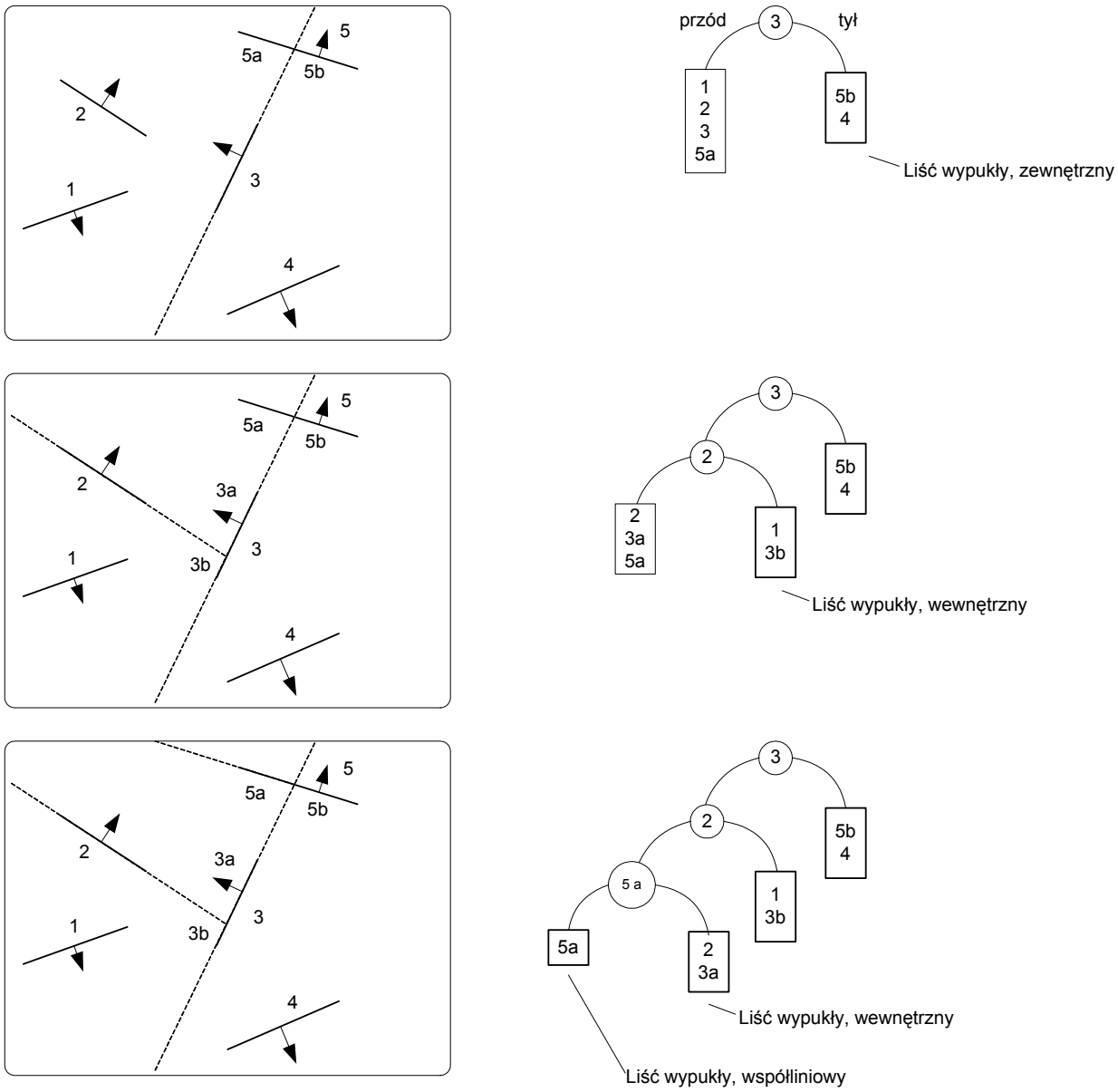
```
klasa Węzeł
{
      Węzeł * przód,
            * tył;
            rozmiar poddrzewa;
      Płaszczyzna;
      Portale;
      Wielokąty;
      ...
};

Węzeł * Węzeł :: znajdź-liść(p)
{
      Węzeł *node = ten węzeł;
      dopóki (node nie jest PUSTY)
      {
            jeśli (node posiada potomka przód lub tył)
                jeśli (p z przodu płaszczyzny node) node=node->przód;
                w przeciwnym wypadku              node=node->tył;
            w przeciwnym wypadku
                  wynik = node;
      }
      wynik = PUSTY;
}
```

Warto zauważyć, że przeszukiwanie drzewa odbywa się ze złożonością logarytmiczną, gdyż w każdym węźle odrzucamy połowę przestrzeni.

Przestrzeń każdego liścia jest ograniczona płaszczyznami węzłów jego przodków. Dlatego można ją utożsamiać z solidem. Ściany takiego solida posłużą później do znalezienia portali.

2.3.2.3. Tworzenie drzewa

Budowa drzewa przebiega rekurencyjnie według metody „dziel i zwyciężaj”. Danymi wejściowymi jest lista wielokątów. Jeśli jest ona wypukła dany węzeł zostaje liściem w przeciwnym wypadku należy znaleźć płaszczyznę podziału i pociąć wszystkie wielokąty z listy. Te które znajdują się przed płaszczyzną umieszczamy w przednim poddrzewie i dalej budujemy drzewo, podobnie czynimy z wielokątami, które leżą za płaszczyzną. Wielokąty leżące na płaszczyźnie dzielącej można teoretycznie umieścić w dowolnym poddrzewie.



Rys.3

Powstają tu dwie nieścisłości: kiedy lista wielokątów jest wypukła oraz jak znaleźć płaszczyznę dzielenia.

Wybór płaszczyzny dzielącej jest bardzo skomplikowanym zadaniem. Istnieje wiele kryteriów mogących definiować ostateczny kształt drzewa. Kryteriami najczęściej spotykanymi będą: minimalizacja przecinanych wielokątów i zrównoważenie drzewa. Jednak najbardziej istotnym kryterium jest ostateczna liczba liści w drzewie. Niestety wymaga ona wielokrotnego budowania i niszczenia całego poddrzewa aby sprawdzić jak korzystna była sprawdzana płaszczyzna. Sprawdzenie wszystkich możliwości zajmowałoby zbyt wiele czasu. Dlatego trzeba ograniczyć się jedynie do dwóch pierwszych kryteriów.

W pierwszej wersji programu, budującego drzewo BSP jedynie z listy wielokątów (tzw. polygon-soup) bez wcześniejszego modułu CSG, zaimplementowałem wybieranie wielokąta, którego płaszczyzna tworzy najlepsze drzewo pod względem małej liczby liści. Metoda ta była jedynie wykorzystywana przy małych liczbach wielokątów, gdzie spodziewana rekurencja poddrzewa była niewielka. Zbyt długi czas obliczeń nie był proporcjonalny do zaledwie niewielkiego wzrostu jakości.

Kandydatów na płaszczyznę dzielącą należy szukać w płaszczyznach wielokątów listy którą dzielimy. Dla każdego wielokąta, którego płaszczyzna nie została wcześniej użyta do podziału drzewa należy znaleźć trzy wartości: liczbę przecinanych wielokątów, różnicę pomiędzy wielokątami leżącymi przed i za danym wielokątem oraz odległość od środka prostopadłościanu zawierającego dzieloną listę.

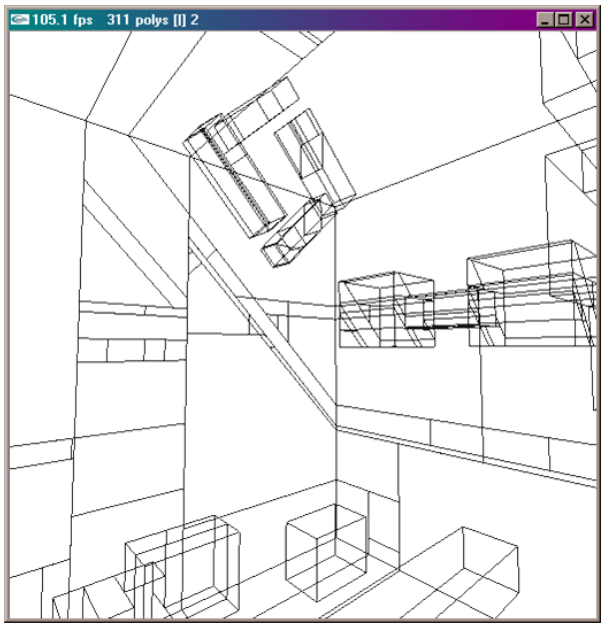
Oznaczmy te wartości jako **P** – liczba przecięć, **R** – różnica oraz **O** – odległość. Wybierając wielokąt o najmniejszej wartości **P** otrzymamy drzewo z najmniejszą liczbą przecięć lecz nie będzie ono zrównoważone. Liniowa struktura drzewa może ograniczyć miejscami czas przeszukiwania. Gdy zminimalizujemy |**R**| drzewo będzie zrównoważone lecz liczba wykonanych przecięć znacznie wydłuży całkowitą liczbę wielokątów w scenie. Płaszczyzna wielokąta o najmniejszej wartości **O** leży najbliżej środka całej listy i prawdopodobnie równomiernie (choć nie wiadomo z jakim skutkiem) dzieli całą listę. Jedynym przypadkiem, gdzie należy oprzeć się na minimalizacji **O** jest zbyt duży rozmiar listy dla której szukamy płaszczyzny dzielącej. Złożoność minimalizacji |**R**| i **P** wynosi aż n^2 a dla **O** jedynie n operacji na wielokątach.

Opieranie się jedynie na minimalizacji jednej wartości nie daje szczególnie zadowalających rezultatów. Dlatego należy zminimalizować formułę:

$$\mathbf{F} = \mathbf{P} * wspP + \mathbf{R} * wspR + \mathbf{O} * wspO$$

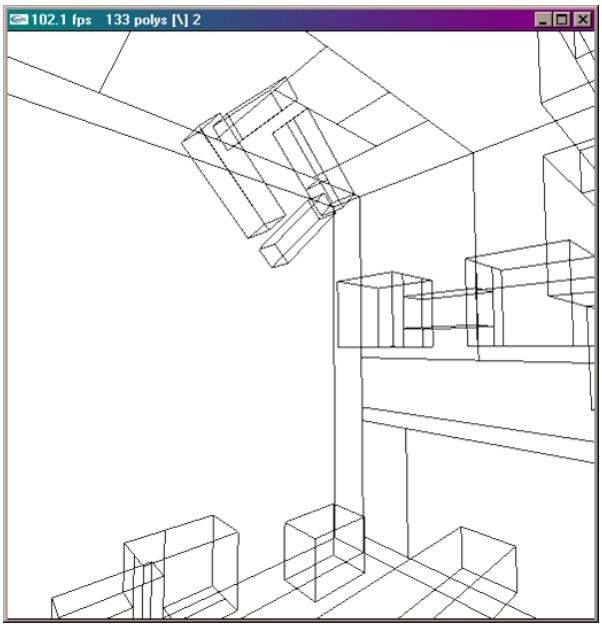
Gdzie *wspP, wspR* i *wspO* określają wagi dla wartości **P**, **R** i **O**. Przeprowadzone testy wykazały, że najmniejszą liczbę liści przy minimalizacji **F** można otrzymać dla wartości *wspP*=(30..50), *wspR*=1 i *wspO*=0.

Niżej znajduje się przykładowa scena , dla której zastosowano różne kryteria wyboru płaszczyzny dzielącej. Scena składa się z 23 solidów, które są reprezentowane przez 138 wielokątów.



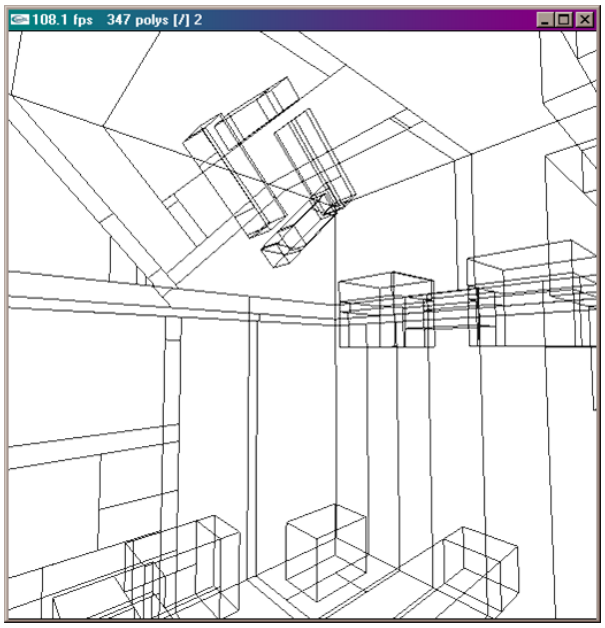
wspR = 0
wspP = 0
wspO = 1

349 wielokątów
146 liści



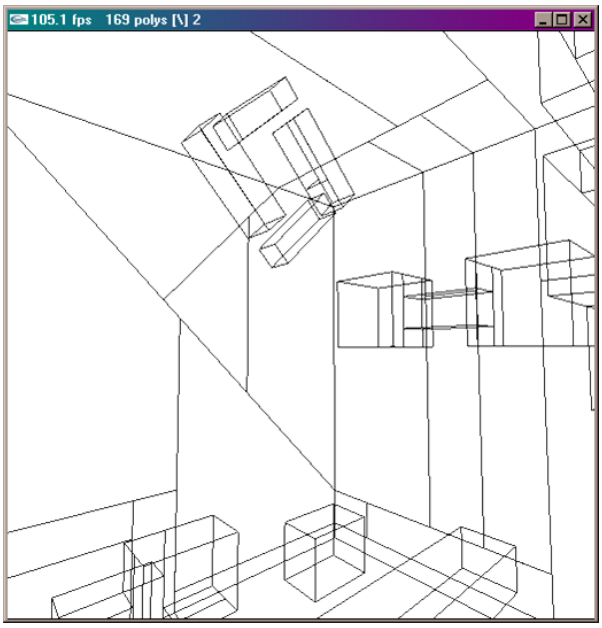
wspR = 0
wspP = 1
wspO = 0

163 wielokąty
75 liści



wspR = 1
wspP = 0
wspO = 0

409 wielokątów
146 liści



wspR = 1
wspP = 50
wspO = 0

196 wielokątów
81 liści

Rys. 4

Wynikiem przecinania listy wielokątów wybraną płaszczyzną są trzy listy wielokątów: wielokąty leżące przed, za i na płaszczyźnie. Nie bez znaczenia jest to co się stanie z wielokątami leżącymi „na". Na początku każdy z wielokątów z listy „na" dodawałem do pozostałych list tak aby zachować ich wypukłość (wypukła lista jest równoważna liściowi – więc dąży do jak najmniejszej liczby liści). Jednak ostatecznie ta metoda okazała się gorsza od odkrytej zupełnie przypadkiem. Każdy z wielokątów z listy „na" umieszcza się w liście „przed" jeśli jego normalna jest zbliżona do płaszczyzny cięcia. Pozostałe wielokąty umieszczamy w liście „za". Metoda ta jest nie tylko szybsza ale również daje mniejszą liczbę liści.

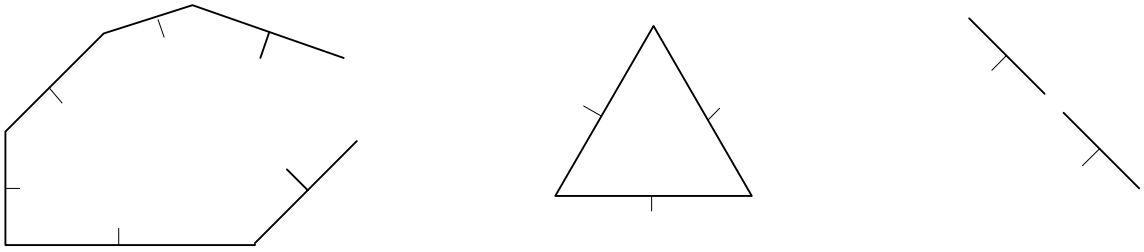
Z uwagi na rekurencyjny charakter metody budowania drzewa istnieje możliwość implementacji tego algorytmu w sposób wielowątkowy.

Zakończenie budowy poddrzewa następuje gdy liść jest wypukły. Oznacza to, że został spełniony jeden z 3 warunków:

a) wszystkie wielokąty leżą na tej samej płaszczyźnie,
b) każdy z wielokątów leży przed pozostałymi lub na tej samej płaszczyźnie co pozostałe i nie jest spełniony warunek a),
c) każdy z wielokątów leży za pozostałymi lub na tej samej płaszczyźnie co pozostałe i nie jest spełniony warunek a).

Dla odpowiednich warunków powstają typy liści:

a) współliniowy,
b) wewnętrzny,
c) zewnętrzny.



Rys.5

Wielokąty leżą na tej samej płaszczyźnie wtedy i tylko wtedy gdy odległość pomiędzy ich normalnymi wynosi mniej niż pewna mała wartość progowa. Istotne jest zorientowanie wielokąta w przestrzeni (normalne muszą mieć nie tylko ten sam kierunek ale i zwrot).

W drzewie znaczącą przewagę stanowią liście wewnętrzne. Dlatego opisana metoda postępowania z wielokątami leżącymi na płaszczyźnie cięcia wydaje się mieć sens. Wielokąty z tej płaszczyzny należy podzielić na dwie grupy: te których normalna jest identyczna z normalną płaszczyzny i te o normalnej z odwrotnym zwrotem. Umieszczanie wielokątów z tym samym zwrotem w poddrzewie leżącym z przodu sprawia, że wielokąty z tej listy są nie tylko rozdzielone ale również ich normalne są zazwyczaj skierowane do środka liścia. Dlatego są one lepiej rozkładane w liściach zewnętrznych.

Po stworzeniu drzewa należy dla każdego poddrzewa znaleźć rozmiary poddrzew. Odbywa się to rekurencyjnie a informacja jest zapisywana za pomocą prostopadłościanu zorientowanego zgodnie z osiami (tzw. Axis-aligned bounding boxes – AABB). Prostopadłościan zawiera w sobie wszystkie wielokąty z liści poddrzewa.

2.3.2.4. Optymalizacja drzewa

Otrzymane drzewo należy zoptymalizować, podobnie jak poprzednie operacja i ta odbywa się rekurencyjnie. Polega ona na usunięciu węzłów tylko z jednym potomkiem. Takie węzły wydłużają drzewo, przez co przeszukiwanie będzie trwało dłużej. Dla każdego liścia również warto zoptymalizować listę jego wielokątów oraz jeszcze raz upewnić się, czy nie jest ona wypukła po połączeniu z listą jego brata. Dwa liście są braćmi jeśli mają wspólnego potomka.

Optymalizacja jest procesem koniecznym. Choć wydaje się to dziwne, lepsze drzewa otrzymuje się przez zastosowanie wyżej opisanych kryteriów i późniejsze poprawienie struktury. Postępując przeciwnie należałoby wybierać płaszczyznę dzielącą z wielokątów o minimalnej wartości |**R**|. Jednak powstałoby wtedy zbyt dużo cięć.

2.3.2.5. Cechy drzewa

Gotowe drzewo w sposób automatyczny podzieliło całą scenę na sektory. Każdy sektor jest ograniczony przez płaszczyzny węzłów jego przodków. Wielokąty w każdym sektorze nie przecinają się i tworzą listę wypukłą. Już samo drzewo znacznie może przyspieszyć proces rysowania całej sceny. Po pierwsze można całkowicie zrezygnować z Z-bufora gdyż odpowiednie przejście przez drzewo zawsze zwróci wielokąty w kolejności od najdalszego do najbliższego lub odwrotnie (druga metoda jest przydatna gdy używamy techniki stencil-buffer polegającej na pojedynczym zapalaniu każdego piksela ekranu). Kolejnym przyspieszeniem jest informacja o rozmiarach poddrzewa. Podczas przechodzenia drzewa można odrzucić te węzły, których prostopadłościany ograniczające nie znajdują się w stożku widzenia. Również węzeł, którego płaszczyzna nie przecina stożka widzenia posiada jednego potomka, który jest niewidoczny. Ważne jest to, że odrzucamy nie każdy liść z osobna lecz całe ich grupy.

Pseudokod wizualizacji drzewa BSP:

```
Węzeł :: rysuj( Obserwator )
{
	jeśli (ten węzeł jest PUSTY) koniec;
	jeśli (Obserwator NIE widzi poddrzewa tego węzła) koniec;

	jeśli (węzeł jest liściem)
	{
		wizualizuj wielokąty liścia;
		koniec;
	}
	w przeciwnym wypadku
	{
	jeśli (Obserwator przed płaszczyzną węzła)
	{
		tył->rysuj( Obserwator );
		przód->rysuj( Obserwator );
	}
	w przeciwnym wypadku
	{
		przód->rysuj( Obserwator );
		tył->rysuj( Obserwator );
	}
}
```

Podział na sektory przyspiesza też znacznie czas znalezienia odpowiedzi na pytanie w jakim ośrodku znajduje się dowolny punkt. Wystarczy znaleźć sektor, który zawiera sprawdzany punkt i w zależności od rodzaju liścia sprawdzić czy jest on za lub przed wielokątami z tego sektora. Dla liścia wewnętrznego jeśli punkt jest choćby za jednym z wielokątów to wtedy jest w ośrodku gęstym, w przeciwnym przypadku – jest przed wszystkimi wielokątami jest w ośrodku rzadkim. Dla liści zewnętrznych proces ten jest analogiczny lecz warunki są odwrócone.

Można również zbudować drugie drzewo służące jedynie do weryfikowania kolizji. Tworzy je się podobnie do pierwszego. Należy jednak zastosować inne kryteria dla płaszczyzny przecinającej. Tu drzewo powinno być bardziej zrównoważone co znacznie przyspieszy późniejsze operacje przeszukiwania. Liście drzewa nie zawierają informacji o wielokątach lecz o gęstości przestrzeni. Dlatego mimo wypukłości listy każdy z wielokątów, którego płaszczyzna jeszcze nie dzieliła drzewa będzie protoplastą węzła.

Korzyści płynące z drzewa kolizji są dwie. Przede wszystkim znacznie przyspiesza się czas odpowiedzi na pytanie w jakiej przestrzeni znajduje się punkt. Poza tym można sprawdzać kolizję dla promieni. Znając początek i koniec wektora możemy precyzyjnie określić w którym miejscu przecina on gęsty ośrodek. W tym celu należy rekurencyjnie porównywać wektor z płaszczyzną węzła. Jeśli leży on po jednej stronie postępujemy podobnie jak z szukaniem liścia zawierającego punk – przechodzimy do tego liścia. W przeciwnym wypadku – gdy wektor przecina płaszczyznę dzielimy go na dwie części i sprawdzamy najpierw tą stronę poddrzewa do której należy początek wektora, a potem koniec. W przypadku dojścia do liścia należy sprawdzić jego typ. Jeśli jest to liść o gęstym ośrodku koniec wektora oznacza miejsce kolizji.

2.3.3. PVS – Possible Visible Set

Possible Visible Set jest zwykłym zbiorem indeksów sektorów widzianych z danego sektora. Jest w nim zakodowana informacja o zasłoniętych miejscach sceny. Zbiór ten, jest nazywany również macierzą widoczności gdzie element o współrzędnych [*i*, *j*] określa czy z sektora *i* widać sektor *j*. Relacja ta jest symetryczna, dlatego i macierz widoczności jest macierzą symetryczną.

2.3.3.1. Ogólny opis modułu

Zadaniem tego modułu jest znalezienie portali, przypisanie ich do sektorów oraz odtworzenie relacji wzajemnej widoczności.

2.3.3.2. Znajdowanie i optymalizacja portali

Na początku należy znaleźć portale. Mogą one znajdować się tylko w rzadkim ośrodku sceny – ośrodek gęsty, czyli ściany zasłaniają inne elementy sceny. Portale łączą przylegające do siebie sektory, dlatego każdy portal będzie leżał na jednej z płaszczyzn węzłów drzewa.

Dla każdego liścia zostanie stworzony solid całkowicie go zawierający. Listą jego ścian będą wszystkie płaszczyzny przodków danego liścia. Samo znalezienie płaszczyzn nie wystarczy do stworzenia poprawnego solida. Płaszczyzny muszą być odpowiednio zorientowane – tak aby środek liścia leżał zawsze za każdą z nich. Dlatego podczas rekurencyjnego przeglądania drzewa należy odwrócić zwroty normalnych płaszczyzn dla węzłów leżących przed swoim rodzicem. Po wygenerowaniu wielokątów tworzących dany solid odejmujemy od każdej ściany solida gęste miejsca sceny. Polega to na odjęciu od wielokątów listy solida wielokąty z listy liścia. Odejmowanie następuje jedynie dla liści wewnętrznych. Dla liści zewnętrznych oraz współliniowych portalami są wszystkie wielokąty z listy zawierającego je solida. Proces odejmowania wielokątów polega na usunięciu części wspólnych dwóch wielokątów leżących na jednej płaszczyźnie. Ponieważ wszelkie operacje dotyczą wielokątów wypukłych odejmowanie często wiąże się z podziałem na mniejsze wielokąty. Ostateczne listy przyszłych portali należy zoptymalizować.

2.3.3.3. Przypisanie portali do liści

Wszystkie portale należy zgrupować w jedną listę i znaleźć odpowiadające im liście. Znalezienie liści polega na „wrzuceniu" portalu do drzewa i zapamiętaniu gdzie „wypadł". Wrzucanie przebiega podobnie do szukania sektora. Porównujemy płaszczyznę węzła z portalem. Jeśli leży on po jednej ze stron zmieniamy węzeł na odpowiedniego potomka i kontynuujemy. Jeśli portal leży na płaszczyźnie należy go wrzucić do obydwu potomków. Dla każdego portalu powinno to nastąpić raz – jest to płaszczyzna rozdzielająca dwa sektory. Jeśli portal jest przecinany płaszczyzną węzła oznacza to błąd. Każdy portal jest wielokątem ograniczonym przestrzenią liścia dlatego nie może wystawać poza jego granice. Część portali wpadnie tylko do jednego liścia – te portale należy usunąć gdyż nic przez nie nie widać.

Po zakończeniu tego etapu do każdego z liści są przypisane jego portale. Dzięki informacji do jakich sektorów należy każdy portal dla każdego sektora można znaleźć jego bezpośrednich sąsiadów. W tym miejscu można usunąć wszystkie liście do których nie dotrze obserwator jeśli będzie poruszał się tylko w powietrzu (przejście z sektora do sektora odbywa się zawsze przez pewien je łączący portal) oraz z pewnego punktu początkowego dla całej sceny. Wystarczy zastosować metodę flood-fill rozpoczynając od sektora zawierającego punkt startowy i rekurencyjnie odwiedzić sąsiadujące sektory. Operacja ta nie jest konieczna lecz dla niektórych scen redukuje liczbę niepotrzebnych liści (np. scena która jest pudełkiem zostanie opisana za pomocą 6 solidów. Jeśli punk początkowy będzie znajdował się w środku obserwator nigdy nie wyjdzie na zewnątrz – dlatego można usunąć tamte liście).

2.3.3.4. Znajdowanie macierzy widoczności

Znajdowanie macierzy widoczności jest kluczową częścią całego algorytmu. Proces ten polega na znalezieniu wszystkich widocznych sektorów z każdego sektora. Jest to najbardziej czasochłonna część całego algorytmu z uwagi na bardzo duży poziom rekurencji. Przebiega on analogicznie dla każdego sektora. I polega na zawężaniu stożka widzenia do kolejnych portali widzianych sektorów.

Opisanie całego procesu jest dość skomplikowane, dlatego wprowadzę kilka pomocniczych definicji. Sektor źródłowy to sektor, dla którego aktualnie jest znajdowana lista widocznych sektorów. Portal źródłowy to kolejny portal tego sektora. Sektor docelowy to bezpośredni sąsiad sektora źródłowego, są one połączone portalem źródłowym. W pierwszym

kroku automatycznie dodajemy do listy widoczności sektora źródłowego jego sąsiadów (sektory docelowe). Następnie dla każdego portalu sektora docelowego rozpoczynamy rekurencyjne poszukiwanie listy widoczności, pod warunkiem, że oba portale nie są współliniowe. Gdyby były współliniowe nic nie można by przez nie zobaczyć.

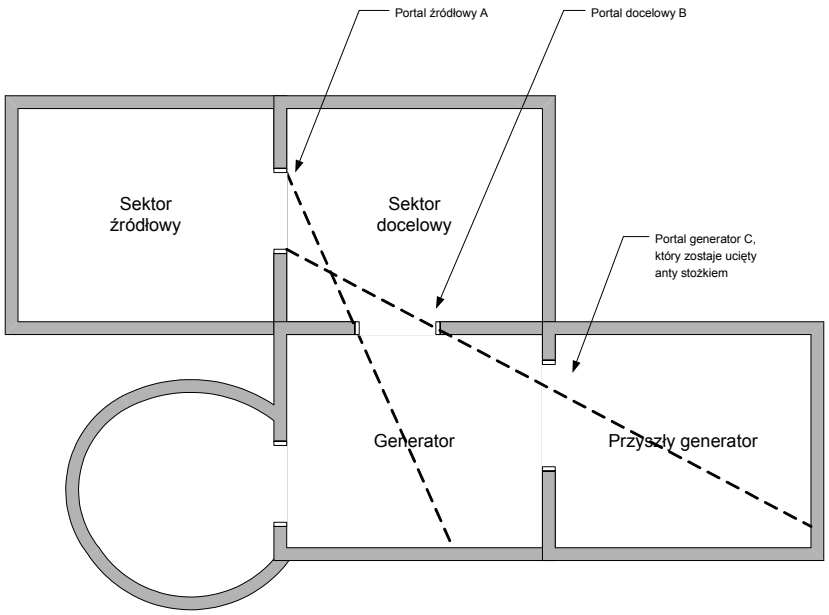
Przeszukiwanie rekurencyjne operuje na trzech sektorach: źródłowym, docelowym i generatorze, oraz na trzech portalach o tych samych nazwach, które łączą kolejno sektory źródłowy z docelowym, docelowy z generatorem, generator z kolejnym sektorem (rysunek 6). Sektor generator jest liściem, do którego prowadzi portal docelowy. Portal generator jest kolejnym portalem tego sektora. Mając portale źródłowy i docelowy należy sprawdzić jakie portale z generatora widać z sektora źródłowego. Najdoskonalszą metodą jest zastosowanie odwrotnego stożka widzenia. Dla tych dwóch portali tworzymy listę wszystkich możliwych płaszczyzn które zawierają krawędź z jednego portalu oraz punkt z drugiego i dodatkowo leżą między danymi portalami. Będąc w sektorze źródłowym nie zobaczymy nic co jest za sektorem docelowym i co znajduje się za wygenerowanymi płaszczyznami. Proces generowania płaszczyzn należy powtórzyć dwa razy dla par źródło-cel oraz cel-źródło aby jak najbardziej zawęzić pole widzenia. Każdy z portali generatora, którego choć część jest widoczna w antystożku prowadzi do kolejnych sektorów. Obcięte portale generatora należy zapamiętać, gdyż im są one mniejsze tym mniejsza będzie widoczność. Nowy obcięty portal generator staje się portalem docelowym, analogicznie z sektorem. Wtedy przechodzimy do następnego kroku rekurencji.

Dwa warunki znacznie przyspieszają obliczenia bez konieczności generowania antystożka. Są to sytuacje w których portal generator jest po tej samej stronie źródłowego portalu co źródłowy liść. Analogicznie dla sektora i portalu docelowego. Pozwala to na nie powracanie do sektora docelowego i źródłowego.

Przeszukiwanie odbywa się dla każdego sektora osobno, dlatego proces ten można z powodzeniem zrównoleglić na maszynie wieloprocesorowej uzyskując idealną skalowalność.

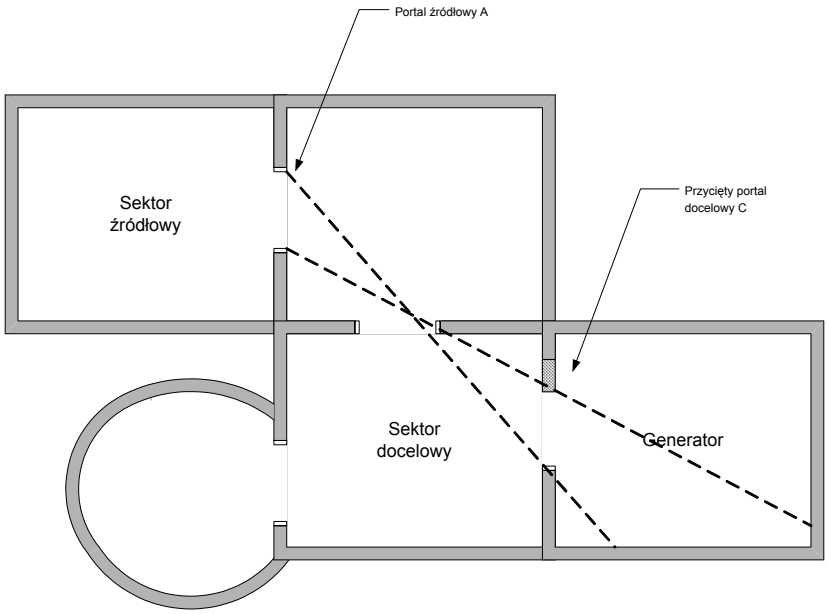
Maksymalny poziom rekurencji jest ograniczony liczbą liści. Przykładem może być długi korytarz podzielony na mniejsze pomieszczenia ścianami działowymi z przejściami. Z pierwszego do ostatniego pomieszczenia algorytm będzie musiał przejść przez wszystkie pozostałe pomieszczenia.

Niestety podobnie jak podczas wizualizacji za pomocą portali i tu jeśli między dwoma sektorami występuje więcej portali, algorytm będzie musiał wielokrotnie wkroczyć do sektora docelowego za każdym razem obcinając trochę inaczej generowane portale.



Antystożek zostaje stworzony na podstawie dwóch portali.

Z sektora źródłowego nie widać pokoju owalnego ponieważ portal do niego prowadzący jest poza antystożkiem



W kolejnym kroku szukania PVS dla sektora źródłowego tylko widoczna część portalu C stanie się portalem docelowym. Stożek zawęża się - informacja o zasłanianiu jest zapamiętywana

Rys.6

Pseudokod modułu:

```
funkcja znajdź_pvs()
{
      dla (każdego liścia sceny źródłowy_liść)
      {
      dla (każdego portalu źródłowy_portal liścia źródłowy_liść)
      {
          docelowy_liść – liść, który łączy źródłowy_portal z źródłowy_liść;
          dodaj do PVS źródłowy_liść liść docelowy_liść;

          dla (każdego portalu docelowy_portal liścia docelowy_liść)
          {
            jeśli (docelowy_portal nie jest współliniowy z źródłowy_portal)
            {
                  rekurencyjny_pvs(
                                    docelowy_liść,
                                    źródłowy_liść,
                                    docelowy_portal,
                                    źródłowy_portal
                                    );
            }
          }
      }
    }
}
```

```
funkcja rekurencyjny_pvs( docelowy_liść, źródłowy_liść,
                          docelowy_portal, źródłowy_portal )
{
      generator_liść – liść, który łączy docelowy_portal z docelowy_liść

      dodaj do PVS źródłowy_liść liść generator_liść
      dodaj do PVS generator_liść liść źródłowy_liść

      dla (każdego portalu generator_portal liścia generator_liść)
      {
            jeśli (generator_portal po tej samej stronie
                  źródłowy_portal co źródłowy_liść) kontynuuj;
            jeśli (generator_portal po tej samej stronie
                  docelowy_portal co docelowy_liść) kontynuuj;

            obcinaj_do_portali( nowy_gen_portal, generator_portal,
                                docelowy_portal, źródłowy_portal )
            obcinaj_do_portali( nowy_źród_portal, źródłowy_portal,
                                docelowy_portal, generator_portal )

            jeśli (new_gen_portal i new_src_portal są w porządku)
            rekurencyjny_pvs( generator_liść, źródłowy_liść,
                              new_gen_portal, new_src_portal );
      }
}

funkcja obcinaj_do_portali(&nowygen, gen, cel, źródło)
{
      nowygen = gen;

      obetnij nowygen płaszczyznami które zawierają krawędzie cel i źródło
      i rozdzielają cel i źródło;
}
```

2.3.3.5. Inne metody generowania portali

Opisany algorytm znajdowania list widoczności potrzebuje jedynie definicji sektorów z odpowiadającymi im portalami. Wyniki dwóch poprzednich modułów są jedynie pewną formą pomocniczą. W scenie gdzie sektory i portale zostaną stworzone np. przez projektującego ją grafika algorytm również będzie działał poprawnie.
Pewną alternatywą dla modułów CSG i BSP może być generowanie płaszczyzn podziału dla np. największych wielokątów sceny, lub wielokątów równomiernie dzielących całą scenę. Listy wielokątów w liściach nie muszą być wypukłe.

Algorytm można również dostosować do scen opisanych za pomocą mniejszych elementów, które definiują fragmenty przestrzeni i będą pełniły funkcję gotowych sektorów. Przykładem może być budynek opisany za pomocą pokoi i korytarzy. Jeśli dla tych elementarnych składowych wcześniej stworzymy portale to algorytm również będzie funkcjonował poprawnie.

Możliwe też jest podzielenie sceny wg drzewa ósemkowego, i stworzenie portali w rzadkim ośrodku sceny.

2.3.3.6. Cechy macierzy widoczności

Znaleziony PVS można zapisać na wiele sposobów, albo jako listę indeksów sektorów albo jako trójkątną macierz z wartościami 0 i 1. Macierz ta jest trójkątna z powodu swej symetrii i dlatego jedną połowę można zignorować. Metoda macierzowa ma tę zaletę, że pozwala na proste upakowanie danych metodą RLE (Run Length Encoding) z uwagi na częste występowanie długich ciągów zer i sporadyczne jedynki. Ewentualnie przy dużej widoczności można postąpić odwrotnie choć dla takich scen ten algorytm nie powinien być stosowany.

Dla każdej sceny warto też określić współczynnik widoczności. Jest to uśredniony stosunek widzianych sektorów do liczby wszystkich sektorów dla każdego liścia. Widoczność rzędu 90% lub więcej oznacza, że scena nie powinna być wizualizowana za pomocą tego algorytmu gdyż prawdopodobnie liczba wielokątów powstałych podczas tworzenia drzewa BSP jest znacznie większa od pierwotnego rozmiaru listy wygenerowanej przez moduł CSG.

2.3.4. RAD

Ten moduł nie stanowi integralnej części algorytmu. Służy on do znalezienia map światła (też map cieni), na których zostanie opisany poziom jasności wszystkich wielokątów sceny. Niestety cienie obliczone w ten sposób są całkowicie statyczne niemniej samo obliczenie map światła znacznie podnosi realizm wizualizowanej sceny.

2.3.4.1. Ogólny opis modułu

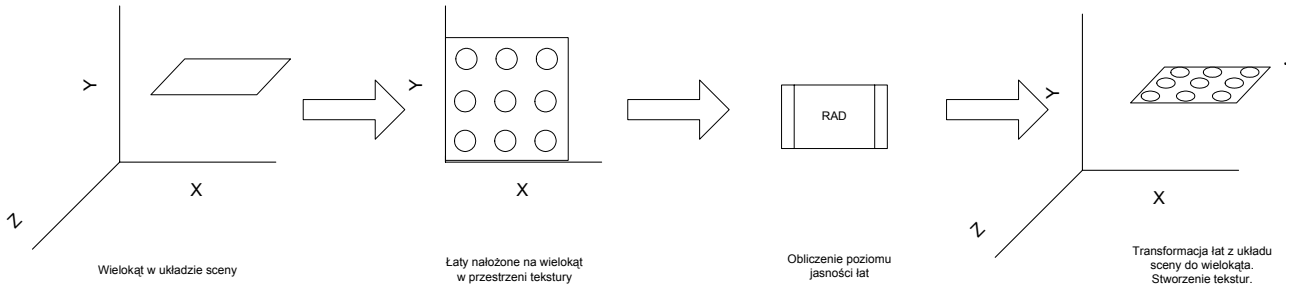
Moduł ten działa w kilku etapach. Wczytuje definicje źródeł światła, dokonuje dyskretyzacji powierzchni sceny na niepodzielne już łaty. Dla każdej z łat oblicza stopień jej jasności oraz dla każdego wielokąta buduje jego mapę światła.

2.3.4.2. Łaty (patches) i mapy światła (lightmaps)

Dla wszystkich wielokątów należy stworzyć listy łat, które się w nim zawierają. Ponieważ mapy światła zostaną zapisane w postaci map bitowych należy określić ich stopień dokładności. Współczynnikiem to definiującym jest rozmiar najmniejszej łaty.

Listę łat najprościej znaleźć przekształcając wielokąt tak aby leżał na osi XY. Po znalezieniu prostokąta go zawierającego i podzieleniu go na odpowiednią względem rozmiaru liczbę łat, należy sprawdzić czy zawierają się w nim i jeśli tak to z powrotem przekształcić do przestrzeni wielokąta.

Podobnie postępuje się z gdy łaty są gotowe. Tworzona jest z nich mapa bitowa, która zostaje przypisana do wielokąta. Aby wygładzić charakterystyczne „zęby” na mapach cieni warto je rozmyć przez uśrednienie sąsiednich pikseli.



Rys.7

2.3.4.3. Wykorzystanie PVS i portali przy śledzeniu promieni światła

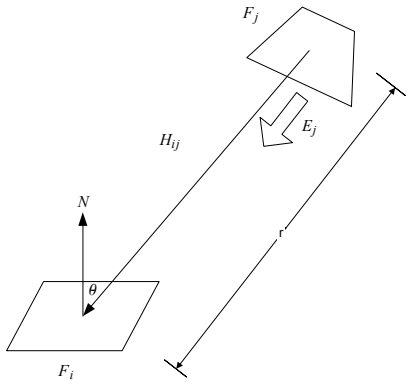
Obliczenie stopnia natężenia światła dla danej łaty sprowadza się do zsumowania wszystkich promieni światła padających na daną łatę. Istotną operacją jest tu sprawdzenie czy promień od światła do łaty przecina na swojej drodze jakieś wielokąty. Jeśli tak się dzieje łata nie zostanie zapalona i w danym miejscu będzie cień.

Złożoność numeryczna takiego procesu jest dość duża. Jest ona iloczynem liczby łat, źródeł światła i wielokątów z których składa się scena. Znacznym przyspieszeniem jest wykorzystanie informacji zawartych w macierzy widoczności. Szybko można założyć, że promień nie pada na łatę leżącą w sektorze z którego nie widać sektora ze światłem. Gdy jest takie połączenie warto też wykorzystać informację o portalach. Promień ze światła do łaty może przechodzić przez wiele sektorów ale zawsze musi przecinać portale. Jeśli w każdym sektorze znajdziemy portal, przecinany przez promień, to również znajdziemy kolejny sektor. Przez to zredukujemy liczbę sprawdzanych sektorów.

2.3.4.4. Równanie światła

W zastosowanym module wykorzystałem popularne równanie światła:

$$F_i = \sum_j \frac{H_{ij} F_j E_j \cos\theta}{r^2}$$



Rys. 8

F_i – kolor docelowej łaty

F_j – kolor punktu dającego światło

E_j – energia punktu dającego światło

H_{ij} – funkcja określająca widzialność pomiędzy łatami i i j

r – odległość pomiędzy łatami i i j

θ – kąt pomiędzy normalną płaszczyzny łaty i a kierunkiem padania światła

j – indeksy wszystkich źródeł światła

2.4. Wizualizacja wspomagana macierzą widoczności

Rysowanie sceny z wyznaczoną macierzą widoczności jest bardzo proste. Najpierw należy znaleźć sektor w którym znajduje się obserwator. Następnie rysuje się wszystkie sektory, które są z niego widoczne. Jeśli nie jest używany Z-bufor należy wyświetlić scenę przechodząc przez drzewo BSP wg metody opisanej wcześniej lecz rysując tylko te sektory, które są widoczne. W innym wypadku wystarczy narysować widoczne sektory w dowolnej kolejności. Jak widać algorytm jest przydatny tylko dla scen o małym współczynniku widzialności.

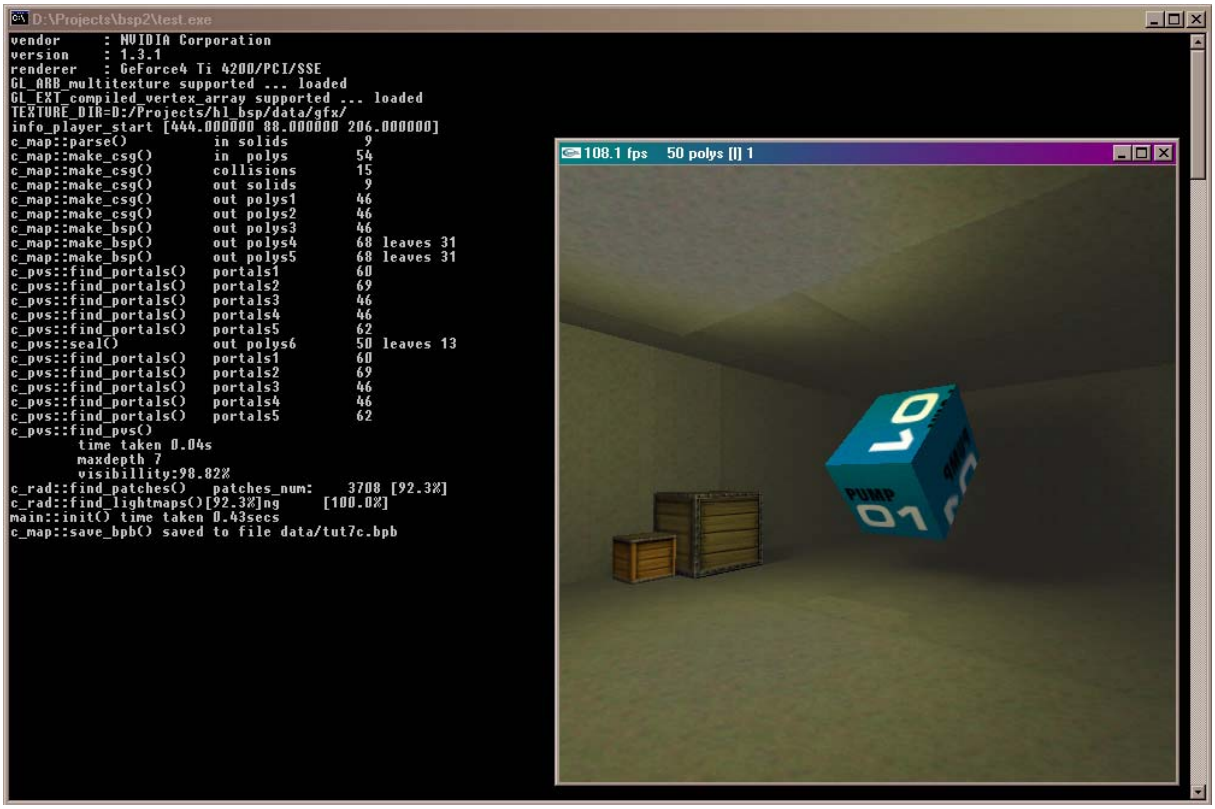
Ważną zaletą jest możliwość przypisania do sektora dodatkowych elementów, które również mogą być odrzucane w procesie rysowania sceny. W przypadku elementów dynamicznych takich jak drzwi lub obiektów należących do kilku liści jednocześnie należy przypisać te obiekty do większej liczby sektorów.

3. Implementacja

Stworzyłem program, który wykorzystuje opisany algorytm w celu znajdowania macierzy widoczności dla scen opisanych solidami. Program został napisany w języku C++, działa na platformie Windows 98/NT/2000/XP, może być również skompilowany pod systemem Linux, jednak wymaga to drobnych modyfikacji. Do zarządzania grafiką wykorzystałem biblioteki OpenGL, GLU i GLUT ze względu na ich łatwość obsługi, możliwość współpracy z akceleratorami graficznymi oraz uniwersalną przenośność.

3.1. Opis działania programu

Program posiada dwie funkcje: budowę drzewa, znajdowanie portali, macierzy widoczności oraz map światła oraz wizualizowanie scen. Ponieważ program działa w trybie konsoli parametry przekazywane do niego należy zapisywać w pliku setup2.txt.



Rys. 9

W trakcie obliczeń program pokazuje aktualny postęp pracy oraz wyświetla statystyki dotyczące m.in. całkowitej liczby wielokątów, portali i liści.

3.2. Tworzenie map

Przykładowe mapy zostały stworzone za pomocą edytora Valve Hammer Editor 3.4. Edytor ten jest przeznaczony do edycji map dla gier Half-Life i wszystkich jej klonów. Pozwala on na łatwe i intuicyjne edytowanie mapy za pomocą solidów.

3.2.1. Ogólne zasady

Tworzone sceny powinny w miarę możliwości pasować do specyfikacji. Otwarte przestrzenie lub pomieszczenia z dużą ilością okien lub przejść spowodują znaczny wzrost współczynnika widoczności co znacznie wydłuży czas pracy modułu PVS jak i podważy zasadność stosowania algorytmu.

3.2.2. Unikanie dodatkowych cięć

Projektowanie za pomocą solidów pozwala na swobodne operowanie bryłami. Jednak należy pamiętać, że algorytm tworzy drzewo BSP i wtedy wiele wielokątów zostanie pociętych na mniejsze części. Sposobem na redukcję niepotrzebnych cięć jest oznaczenie solidów mogących te cięcia spowodować. Nie będą one uwzględniane podczas przygotowywania listy wielokątów i tworzenia drzewa. Można w ostatnim kroku przypisać je do zawierających je sektorów. Podczas wizualizacji należy wyświetlić wielokąty liścia oraz przypisane do niego solidy. Elementy sceny, które tworzą dodatkowe cięcia są najczęściej schody lub drobne detale (stworzony program nie obsługuje tego rozwiązania).

3.3. Testy / Wyniki

Przeprowadzone testy wykazały, że, istnieje wprost proporcjonalna zależność między liczbą liści, portali, wielokątów, czasem budowy drzewa i poszukiwaniem macierzy widoczności. Większa ilość portali wydłuża czas generowania antystożków. Większa ilość liści sprawia, że w drzewie jest więcej płaszczyzn dzielących więc jest też więcej portali. Z tego samego powodu powstaje więcej cięć i dlatego wzrasta liczba wielokątów. Budowa drzewa wydłuża się gdy przybywa wielokątów, wskutek przecinania płaszczyznami dzielącymi (potrzeba sprawdzenia kryteriów).

Najmniejsze ilości sektorów otrzymuje się dla dużych wartości wspP i małych wspR.

Przykładowe czasy kompilacji map na komputerze z procesorem Intel Celeron 1200 MHz, pamięcią 256MB i pracującym pod systemem Microsoft Windows 2000.

Mapa	solidy	*wspP*	*wspR*	Portale	Liście	Widzialność / osiągnięty poziom rekurencji	Czas PVS	Czas razem
maciek05_bsp5.map	272	50	1	1512	288	27.4% / 24	58s	79s
b08.map	142	50	1	422	200	12.1% / 15	0.21s	31s
b08.map	142	0	1	1330	373	11.7% / 25	2.92	55s
b08.map	142	1	0	396	187	12.6% / 15	0.20s	38s

Plik maciek05_bsp5.map przedstawia 1 piętrową willę z 14 pomieszczeniami, bez mebli, b08.map to labirynt 100 pomieszczeni w układzie 10x10 połączonych losowo przejściami.

4. Podsumowanie

Przedstawiony wyżej algorytm jest skuteczny przy wizualizacji scen o dużym stopniu samozasłonięcia. Można go stosować zarówno do generowania portali jak i do znajdowania macierzy widoczności na podstawie portali i sektorów.

Udoskonalenia, które należałoby wprowadzić w programie dotyczyłyby zrównoleglenia pracy na maszynach wieloprocesorowych (generowanie drzewa i szukanie PVS) oraz uwzględniania elementów dynamicznych.

5. Bibliografia

Foley, van Dam, Feiner, Hughes – Computer Graphics principles and practice.
Dan Royer – Creating a cutting-edge engine

exaflop.org:
Nathan Whitaker - Binary Space Partitioning Trees
Nathan Whitaker - Extracting Connectivity Information From A BSP Tree

flipcode.com:
Edward Kmett - Fine Occlusion Culling Algorithms
Edward Kmett - Scene Traversal Algorithms
Edward Kmett - A Hybrid Approach to Visibility
Jacco "The Phantom" Bikker - Building a 3d portal engine
Paul "Midnight" Nettle - Ask Midnight: Game development questions & answers
Paul "Midnight" Nettle - Fast BSP tree generation using binary searches [An informal paper]

GraphicsPapers.com – internetowa baza danych dokumentów poświęconych grafice.

Silicon Graphics, Incorporated - BSP Tree Frequently Asked Questions (FAQ)

6. Formaty danych

6.1. Ustawienia parametrów programu

Plik setup2.txt zawierający globalne parametry programu jest zwykłym plikiem tekstowym, którego każda linia jest poleceniem ustawiającym wartość zmiennej. Zmienne mogą być liczbami i łańcuchami znaków. Zmiennych numerycznych można budować wyrażenia arytmetyczne wykorzystując wcześniej zdefiniowane zmienne. Łańcuchy można konkatenować.

$TEXTURE_DIR - określa ścieżkę z plikami tekstur, są to pliki w formacie GIF

$INPUT_FILE - określa plik wejściowy z definicją sceny

$OUTPUT_FILE - plik wyjściowy o rozszerzeniu .tpb lub .bpb

POLYGON_EPS - wartość progowa określająca dokładność operacji na wielokątach

POLYGON_SPLIT_EPS - grubość wielokątów

POLYGON_SHARE_EDGE_EPS - minimalna odległość między wierzchołkami, które są scalane w procesie optymalizacji

BSP_DIFF_MULTIPLIER - wartość *wspR*

BSP_SPLIT_MULTIPLIER - wartość *wspP*

BSP_MIDDIST_MULTIPLIER - wartość *wspO*

BSP_EPS - wartość progowa określająca dokładność operacji przy budowie drzewa

MY_SPLIT_METHOD - metoda znajdowania płaszczyzny dzielącej listę wielokątów 0 dla pierwszego wielokąta z listy 1 dla zastosowania kryterium z *wspP*, *wspR* i *wspO*.

PVS_RECURSE_MAX_DEPTH - maksymalny stopień zagłębienia przy szukaniu PVS

PATCH_DENSITY - gęstość łat w scenie

AMBIENT_R, AMBIENT_G, AMBIENT_B - wartości składowych kolorów minimalnej wartości w mapach światła

LMP_MAIN_PATCH - waga wartości głównego piksela mapy cieni przy uśrednianiu tekstury

LMP_CLOSE_PATCH - waga każdego z ośmiu sąsiadów głównego pixela. Dla wartości LMP_MAIN_PATCH = 1 i LMP_CLOSE_PATCH = 0 tekstura nie będzie uśredniana.

SAVE_LIGHTMAPS - sposób zapisywania map światła : 0 nie zapisuje, 1 zapisuje jako zbiór plików w formacie BMP, 2 umieszcza mapy światła w pliku .BPB

PROGRESS_MODE – sposób wyświetlania postępu pracy podmodułów 0 i 3-brak, 1 – dynamiczny, 2 – postęp co 10%. Opcja ta jest przydatna gdy następuje przekierowanie danych programu strumieniem do pliku.

NOGRAPHICS – 0 – nie włącza obliczonej mapy, 1 – włącza i czeka na wyjście

6.2. Opis sceny

Pliki .map opisujące scenę są również plikami tekstowymi. Ich struktura wygląda następująco:

Struktura całego pliku:	OPIS SCENY LISTA OBIEKTÓW

OPIS SCENY:	{
	DEFINICJE
	LISTA SOLIDÓW
	}

DEFINICJE:	"nazwa zmiennej" "wartość" ....
LISTA SOLIDÓW:	SOLID ...

SOLID:	{
	ŚCIANA
	...
	}

ŚCIANA	(W1) (W2) (W3) TEKSTURA [S] [T] ROT SX SY
W1 W2 W3	Współrzędne punktów, przez które przechodzi płaszczyzna (nie mogą być współliniowe) w formacie X Y Z
TEKSTURA	Nazwa pliku z tekstura bez rozszerzenia i ścieżki dostępu.
S T	Wektory mapujące wielokąt na przestrzeń tekstury wg wzoru: $\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} ((S \circ P)/SX + S.u)/W \\ ((T \circ P)/SY + T.u)/H \end{bmatrix}$ U,V – współrzędne w przestrzeni tekstury; P – wierzchołek wielokąta któremu odpowiada U,V; W,H – szerokość i wysokość tekstury, S.u i T.u – przesunięcie w przestrzeni tekstury. Format : X Y Z u
SX SY	Skala w osiach U i V tekstury
ROT	Kąt obrotu tekstury (nie używany)
LISTA OBIEKTÓW	OBIEKT ....

OBIEKT	{
	”classname” ”nazwa klasy”
	DEFINICJA
	...
	}

Tylko dwa rodzaje obiektów są obsługiwane przez program : **light** dla informacji o światlach i **info_player_start** o położeniu początkowym obserwatora. Światło wymaga zdefiniowania zmiennych **_light** w formacie R G B F (składowe koloru od 0 do 255 i natężenie światła) oraz **origin** w formacie X Y Z (określa położenie punktu światła). Punkt początkowy musi mieć określone położenie – **origin**. Ten obiekt służy jedynie do kasowania niedostępnych liści.

6.3. Wynik w formacie tekstowym

Pliki .tbp (Text PVS BSP) opisują scenę w postaci drzewa, do każdego z liści dołączony jest zbiór indeksów widzialnych sektorów.

Struktura całego pliku –korzeń drzewa:	WĘZEŁ

WĘZEŁ:	Node
	{
	NORMALNA
	MIN
	MAX
	MID
	ID
	COVEX_TYPE
	CONTENTS
	PVS
	LISTA WIELOKĄTÓW
	LISTA PORTALI
	PRZÓD
	TYŁ
	}

PRZÓD, TYŁ:	Węzły drzewa opisane wg definicji, jeśli węzeł nie ma potomka to znajduje się tam ciąg NULL
NORMALNA, MIN, MAX, MID:	Wektory opisane w formacie X Y Z U, MIN i MAX określają rozmiar prostopadłościanu ograniczającego. MID środek liścia.
ID:	Unikalny indeks liścia lub –2 dla węzłów
CONVEX_TYPE:	Rodzaj wypukłości -1 dla liści wewnętrznych -2 dla liści zewnętrznych -4 dla współliniowych
CONTENTS:	Nieużywana wartość określająca gęstość danego liścia – zawsze –1.

LISTA WIELOKĄTÓW, LISTA PORTALI:	Polylist
	{
	liczba wielokątów
	WIELOKĄT
	...
	}

PVS	Liczba indeksów indeks1 indeks2 indeks3 PVS liścia zawiera indeksy innych liści

WIELOKĄT	Polygon
	{
	NORMALNA
	SPH
	TEXTURA
	MAPA ŚWIATŁA
	ID POWIERZCHNI
	ID LIŚĆ 1
	ID LIŚĆ 2
	LISTA WIERZCHOŁKÓW WIELOKĄTA
	LISTA WSPÓŁRZĘDNYCH TEKSTURY i MAPY ŚWIATŁA
	}

SPH	Sfera ograniczająca wielokąt w formacie X Y Z R, gdzie R promień.
ID POWIERZCHNI	Unikalny indeks płaszczyzny, na której leży wielokąt (nieistotne)
ID LIŚĆ 1 i 2	Indeksy liści do których prowadzi portal, wartości nieistotne dla wielokątów
LISTA WIERZCHOŁKÓW i LISTA WSPÓŁRZĘDNYCH TEKSTURY i MAPY ŚWIATŁA	Liczba wierzchołków X Y Z U X Y Z U ... Współrzędne wielokąta mają zapisany kolor wierzchołka w systemie heksadecymalnym w miejscu U. Współrzędne tekstury i mapy światła

6.4. Wynik w formacie binarnym

Pliki .bpb (Binary PVS BSP) jest binarną wersją poprzedniego formatu. Pliki tekstowe były zbyt wolno wczytywane z powodu dużej ilości operacji na ciągu znaków takich jak szukanie końca definicji elementu lub zamiana liczby z formatu ASCII do postaci binarnej.

Współrzędne wektorów są zapisane jako 4 bajtowe liczby zmienno przecinkowe a wszystkie liczby elementów (wierzchołków, indeksów, wielokątów) i indeksy jako 4 bajtowe liczby stałoprzecinkowe. Pominięte zostały nawiasy klamrowe i słowa kluczowe **node**, **polygon** i **polylist**. Nazwy tekstur i map cieni poprzedza liczba określająca długość łańcucha lub 0 jeśli dany wielokąt nie ma tekstury. W przypadku zapisywania map światła do pliku .bpb w zamiast liczby z długością łańcucha występuje wartość 0xFFFFFFFF po której następują szerokość i wysokość mapy światła oraz sama mapa światła (która zajmuje szerokość*wysokość*3 bajtów). Przed węzłami PRZÓD i TYŁ są umieszczone wartości 0 i 1 odpowiadające kolejno przypadkom gdy nie ma lub jest potomek.

7. Ilustracje

1. Przepływ danych przez moduły
2. CSG – operacja scalania solidów
3. BSP – przykład tworzenia drzewa 2D
4. BSP – różne kryteria wyboru płaszczyzny dzielącej
5. BSP – typy liści drzewa
6. PVS – ilustracja działania rekurencyjnego znajdowania PVS
7. RAD – generowanie łat do mapy cieni
8. RAD – równanie światła
9. Przykład działania programu