Uniwersytet Wrocławski
Wydział Matematyki i Informatyki

Remigiusz Żukowski
nr albumu: 168611

Automatyczna dekompozycja sceny 3D na portale i sektory

Praca magisterska:
Informatyka
Promotor:
dr Andrzej Łukaszewski

Wrocław, 2008

Spis treści

Wstęp

Grafika komputerowa pojawiła się już w pierwszych latach istnienia komputerów. W początkowym etapie ze względu na wysokie koszty sprzętu do jej przetwarzania i prezentacji dziedzina ta była wąską specjalizacją zarezerwowaną wyłącznie dla instytucji posiadających odpowiednie środki. Z czasem, wraz z postępem technologicznym trafiła w ręce zwykłych ludzi, pokonując drogę od komputerów analogowych sprzężonych z lampami oscyloskopowymi jako urządzeniami do zobrazowania do zastosowań pełniących rolę wyłącznie rozrywkową.

Rysunek 0.1: Realistyczna grafika komputerowa otrzymana metodą *Ray-tracing*.

Nie sposób dziś przejść obojętnie obok obrazów generowanych przez komputery. Ich doskonałość niejednokrotnie sprawia wrażenie wirtualnej rzeczywistości, a dodanie interaktywnej animacji sprawia wrażenie realistycznego dzieła filmowego. Tą dynamikę rozwoju zawdzięcza się rozwojowi sprzętu komputerowego i oprogramowaniu. Dziś wyświetlanie wysokiej jakości trójwymiarowych obrazów to złożony i wieloetapowy proces, gdzie pracują wyrafinowane algorytmy zmagające się z milionami danych opisujących geometrię wirtualnego świata zapisanego w pamięci komputera.

Współcześnie interaktywne wyświetlanie trójwymiarowych scen na komputerach osobistych sprowadza się do rasteryzacji dużej ilości prymitywów graficznych reprezentujących geometrię sceny, będących z reguły trójkątami. Zazwyczaj jest to wykonywane na specjalizowanych procesorach graficznych, charakteryzujących się wysoką wydajnością przetwarzania tego typu danych. Aby osiągnąć wysoką jakość tworzonego obrazu niezbędna jest reprezentacja geometrii sceny w postaci setek tysięcy a nawet milionów trójkątów wzbogaconych o skomplikowane modele materiałów, które modelują interakcję ze światłem na wzór rzeczywistych powierzchni spotykanych w naturze. Pomimo że urządzenia (akceleratory) graficzne cechują się wielką wydajnością, przetwarzanie całości tych danych jest zadaniem nierzadko niewykonalnym z zadowalającą, gwarantującą płynność animacji prędkością. Dlatego dąży się do ograniczenia ilości przetwarzanych danych, przede wszystkim do ich aktualnie widocznego podzbioru. Charakterystyczne jest, że przy wyświetlaniu pojedynczej klatki obrazowej ilość widocznych prymitywów graficznych jest z reguły wielokrotnie mniejsza niż ich całkowita liczba. Jest to konsekwencja prostej obserwacji: w danej chwili widzimy tylko niewielki podzbiór całości modelu (pomieszczenia, budynku, miasta czy też świata). Jednakże zadanie wyznaczania widocznego podzbioru elementów jest wysoce nietrywialne i jak dotąd nie znalazło ogólnego, działającego w każdym przypadku algorytmicznego rozwiązania o akceptowalnej złożoności obliczeniowej i pamięciowej.

Rozdział 1

Widoczność w grafice komputerowej

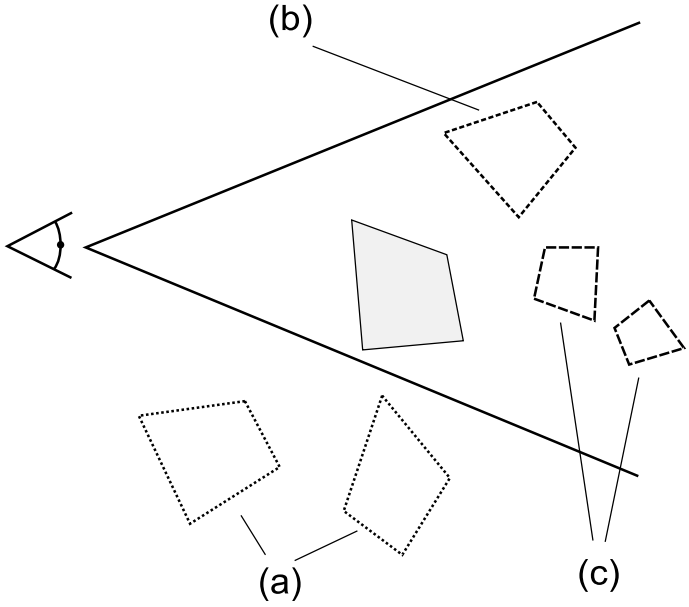
Począwszy od powstania grafiki komputerowej zagadnienie określania widoczności było jednym z jej fundamentalnych problemów. W chwili obecnej problem ten można uznać za rozwiązany w sposób zadowalający poprzez algorytm *Z-bufora*. Jednakże ciągle rosnące rozmiary zbiorów danych opisujących geometrie sceny $3D$ uczyniły niemalże niemożliwym wyświetlanie jej z akceptowalną szybkością przy użyciu klasycznych technik takich jak *Z-bufor*, pomimo wprost niesamowitej wydajności uzyskiwanej dzięki sprzętowemu wspomaganiu. Dlatego zagadnienie widoczności w grafice komputerowej nabiera dziś coraz większego znaczenia.

Selekcja widoczności (*Visibility culling*) polega na szybkim odrzucaniu całkowicie niewidocznych partii geometrii sceny jeszcze przed właściwym procesem usuwania niewidocznych bądź zasłoniętych jej fragmentów, dokonywanym najczęściej przez algorytm *Z-bufora*. Pod omawianym pojęciem kryją się trzy klasyczne schematy:

- **Usuwanie ścian tylnych** (*back-face culling*)
- **Usuwanie ścian będących poza bryłą widzenia** (*view-frustum culling*)
- **Usuwanie ścian przesłoniętych przez inne** (*occlusion culling*)

Rysunek 1.1 symbolicznie przedstawia opisane powyżej schematy. Ostatnia z technik *occlusion culling* uwzględnia wzajemną relację pomiędzy obiektami sceny, dlatego jest realizacja jest o wiele bardziej skomplikowana niż pozostałych dwóch pierwszych, jednakże to ona z wszystkich trzech wymienionych

jest najbardziej istotna, gdyż pozwala na drastyczne zmniejszenie ilości rysowanych obiektów. W scenach opisujących otwarte scenerie w pewnych przypadkach podzbiór widocznych elementów może być równy zbiorowi wszystkich jej elementów, jednak w wielu obszarach mamy do czynienia z gęsto przesłoniętym środowiskiem, gdzie widoczne obiekty są tylko małym podzbiorem całej sceny. Wiele trudności sprawia fakt *złego uwarunkowania zadania*, gdyż nawet małe zmiany położenia punktu widzenia mogą skutkować dużymi zmianami w widoczności.



Rysunek 1.1: Trzy typy selekcji widoczności: **(a)** Usuwanie ścian leżących poza bryłą widzenia. **(b)** Usuwanie ścian tylnych. **(c)** Usuwanie ścian przesłoniętych przez inne.

1.1. Klasyfikacja metod wyznaczania widocznych obiektów

Ze względu na jakość wyznaczonego zbioru widocznych obiektów wyróżniamy następujące grupy metod [Bit02]:

- **Konserwatywne** Metody te konsekwentnie przeszacowują widoczność powodując fałszywe błędy widoczności oznaczające, że pewne niewidoczne fragmenty są uznawane za widoczne. Nie prowadzi to do otrzymania błędnych obrazów, skutkuje jednak koniecznością przetwarzania (rysowania) większej ilości obiektów. W obecnie stosowanych metodach tego typu współczynnik przeszacowania nie przekracza zazwyczaj kilku-

dziesięciu procent, co w większości przypadków jest zupełnie akceptowalnym poziomem.

- **Agresywne** Zawsze niedoszacowują widoczność powodując błędy widoczności i nieprawidłowe obrazy z powodu usuwania pewnej liczby widocznych obiektów. Mogą być użyteczne, jeśli ilość błędów widoczności jest bardzo mała, bądź też zagadnienie widoczności nie może być inaczej rozwiązane w bardziej satysfakcjonujący sposób. Jednakże w praktyce nawet niewielkie błędy widoczności są dostrzegalne i z tego powodu nieakceptowalne.

- **Aproksymacyjne** Jednocześnie przeszacowują i niedoszacowują prowadząc do fałszywych widoczności i fałszywych niewidoczności.

- **Dokładne** Dostarczają optymalny zbiór danych do renderowania, który jest równy sumie wszystkich widocznych na ekranie wielokątów.

W kontekście wyświetlania grafiki problem widoczności jest rozpatrywany w przestrzeni trójwymiarowej, jednak wiele metod wprowadza pewne ograniczenia w tym względzie, z czego wynika następująca klasyfikacja:

- **Metody** 2**-wymiarowe**. Dokonują uproszczenia sceny do dwuwymiarowego planu, w którym występuje płaska podłoga i pionowe ściany.

- **Metody** 2.5**-wymiarowe**. Scena reprezentowana jest jako mapa wysokości, stosuje się wiele wariantów: równomierna bądź nierównomierna kwantyzacja wysokości, reprezentacja jako funkcja dwóch zmiennych, hierarchiczny podział. Z reguły takie uproszczenie jest wystarczające dla potrzeb reprezentacji scen opisujących tereny czy też obszary miejskie.

- **Metody 3-wymiarowe**. Obliczenia widoczności przeprowadzane są prawidłowo dla dowolnych trójwymiarowych scen, bez żadnych ograniczeń dotyczących orientacji kamery.

Ze względu na precyzję działania rozróżnia się, czy algorytm oblicza widzialność z bieżącego punktu widzenia (**from-point visibility**) czy też dokonuje obliczeń, które są ważne w pewnym obszarze (**from-region visibility**). Istotną cechą widoczności *z obszaru* jest jej ważność dla wielu następujących po sobie klatek, co pociąga za sobą amortyzację kosztów obliczeń w czasie. Obok tego często wymienia się cechę predykcji widoczności. Wady to przede wszystkim dłuższe i bardziej skomplikowane obliczenia oraz możliwość znacznego przeszacowania widoczności, głównie ze względu na istotną właściwość metod obszarowych - widoczność *z obszaru* jest sumą widoczności z każdego punktu obszaru, w każdym możliwym kierunku.

Wśród metod wyznaczających widoczność *z punktu* rozróżniamy te działające z **precyzją obrazową** (*image-precision*), co oznacza przetwarzanie „surowych" danych, bądź też z **precyzją obiektową** (*object-precision*), operujące na zrzutowanej do przestrzeni obrazu postaci obiektów.

Dla metod wyznaczających **widoczność z obszaru** istotne jest, czy wykorzystują dostarczoną reprezentację sceny, bądź posługują się reprezentacją alternatywną, uproszczoną bądź zapisaną w innej postaci. Przykładem uproszczonej reprezentacji jest naturalny w scenach architektonicznych podział na sektory (pomieszczenia) i portale (drzwi, okna), zaś *drzewo BSP* jest przykładem reprezentacji alternatywnej.

1.2. Spójność przestrzenna i czasowa

W celu polepszenia efektywności algorytmów ustalania widoczności istotne jest uwzględnienie wszystkich potencjalnych spójności między elementami sceny.

Pod pojęciem *spójności przestrzennej* kryje się *spójność w przestrzeni obiektowej* i *spójność w przestrzeni obrazowej*. Pierwsza z nich oznacza, że obiekty będące elementami sceny 3D leżącymi blisko siebie nierzadko należą do tej samej klasy abstrakcji względem widoczności. Przykładem jest hierarchiczny podział przestrzeni, który bezpośrednio wykorzystuje ten rodzaj spójności. Pojedyncze obiekty są rozważane tylko wtedy, gdy ich widoczność zostanie "potwierdzona„ na wyższym, mniej dokładnym poziomie.

Spójność w przestrzeni obrazowej odnosi się do dwuwymiarowej, zrzutowanej na płaszczyznę obrazu geometrii sceny. Może być wykorzystana przez podział przestrzeni obrazowej używając na przykład *drzew BSP* [FKN80] [NAT90]. Także spójność na poziomie pojedynczych punktów obrazowych może być wykorzystana, na przykład poprzez obserwację, że w algorytmie *Z-bufor* wartości Z sąsiednich punktów zazwyczaj różnią się tylko nieznacznie.

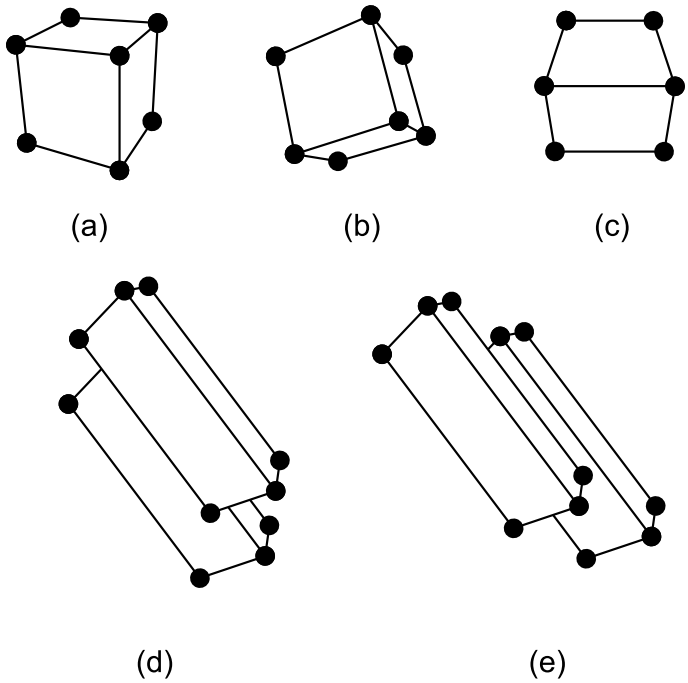
Spójność czasowa polega na wykorzystaniu pewnych informacji obliczonych i zapamiętanych w czasie rysowania wcześniejszych klatek obrazu. Mogą one być używane do ukierunkowanego poszukiwania widocznych partii sceny w bieżącej klatce. Dane te mogą być użyte bezpośrednio, bądź też służyć jako wskazówki, przyczyniając się do zwiększenia efektywności stosowanego algorytmu.

1.3. Graf aspektów

Graf aspektów (*aspect graph*) jest strukturą danych, która w analityczny sposób koduje dokładne informacje o widoczności. Pojęcie to zostało wprowadzone w [PD90] i opiera się na podziale przestrzeni na sektory, w których widoczność nie ulega zmianie. Separatorami pomiędzy sektorami są płaszczyzny nazywane *wizualnymi zdarzeniami (visual events)*.

Do formalnego zdefiniowania *grafu aspektów* niezbędne jest rozpatrzenie grafu *struktury obrazowej*, który jest grafem planarnym stanowiącym projekcję wizualizowanej sceny na płaszczyznę obrazu z danego punktu widzenia. Dwa różne widoki tego samego obiektu mają ten sam *aspekt*, jeśli ich *grafy struktury obrazowej* są izomorficzne. Wierzchołkami *grafu aspektów* są obszary (sektory) takie, że dla dowolnych dwóch punktów w nich leżących ich *grafy struktury obrazowej* są izomorficzne. Krawędziami są połączenia pomiędzy sąsiednimi sektorami, wyrażającymi *zdarzenia wizualne*.

Pesymistyczna złożoność *grafu aspektów* jest bardzo duża i w przestrzeni trójwymiarowej dla rzutu środkowego wynosi $\mathcal{O}(n^9)$, co dla typowych scen zawierających dziesiątki tysięcy obiektów jest niepraktyczne z punktu widzenia złożoności pamięciowej jak i obliczeniowej. Ponadto opisywana struktura charakteryzuje się nadmiarowością polegającą na tym, że różne *aspekty* mogą posiadać te same zbiory widocznych wielokątów. Stanowi to przykład, że dokładne, teoretyczne rozwiązanie problemu okazuje się w praktyce nieprzydatne. Graf aspektów jest ważną strukturą, do której istnienia odwołuje się wiele istniejących algorytmów wyznaczania widoczności.



Rysunek 1.2: **(a)**,**(b)** Przykład dwóch widoków tego samego obiektu posiadających ten sam *aspekt wizualny*, **(c)** Inny widok tego obiektu, posiadający inny aspekt, gdyż jego *graf struktury obrazowej* nie jest izomorficzny z poprzednimi grafami. **(d)**,**(e)** Regiony o różnych *aspektach wizualnych* mogą posiadać te same zbiory widocznych wielokątów.

Rozdział 2

Techniki usuwania niewidocznych obiektów

2.1. Tradycyjne algorytmy wyznaczania widoczności

Będąc obserwatorem trójwymiarowego świata zauważamy, że pewne obiekty są w różnym stopniu przesłaniane przez inne. Zjawisko to zmienia się wraz z punktem widzenia. Odczuwamy także wpływ odległości od widzianych obiektów w postaci zmniejszania się ich rozmiarów. To jak należy odwzorować trójwymiarową przestrzeń na płaszczyznę ekranu, aby otrzymać realistyczny obraz opisywane jest przy pomocy prostych równań matematycznych, rzutujących przestrzeń $3D$ na płaszczyznę. Znacznie trudniejsze okazuje się symulowanie pierwszego z opisywanych zjawisk, czyli wzajemnego przesłanianie się obiektów. Badania na tym polu doprowadziły do powstania szerokiej klasy rozwiązań algorytmów. W tym rozdziale przedstawione zostaną dwa najbardziej znane.

2.1.1. Z-bufor

Algorytm *Z-bufora* [Cat74] jest jednym z najprostszych algorytmów wyznaczania widoczności. Jego prostota pozwala na efektywną implementację sprzętową co sprawia, że dziś występuje nawet w najprostszych układach graficznych. *Z-bufor* jest algorytmem pracującym z precyzją obrazową. Z każdym punktem obrazowym, poza jego kolorem kojarzy się informację o jego głębokości (współrzędnej Z), która jest inicjowana na wartość odpowiadającą nieskończoności. Prymitywy graficzne mogą być przetwarzane w dowolnej kolejności. W procesie ich rasteryzacji, dla każdego punktu obrazowego pokrywającego bieżąco rysowany prymityw oblicza się jego głębokość, którą porównuje się

z głębokością zapisaną w *Z-buforze*. Jeśli jest ona większa, co oznacza, że aktualnie przetwarzany fragment prymitywu jest bliższy punktowi widzenia, to kolor oraz głębokość punktu obrazowego są uaktualniane bieżąco wyznaczonymi wartościami. W przeciwnym wypadku jest on odrzucany, gdyż jest zasłonięty przez poprzednio narysowany prymityw. Algorytm *Z-bufora* może być przyśpieszony przez przetwarzanie obiektów w porządku od najbliższych do najdalszych. Taka kolejność minimalizuje liczbę aktualizacji zarówno koloru każdego z punktów obrazu jak i powiązanej z nim głębokości. Dla dużych i złożonych scen algorytm przetwarza wszystkie obiekty, pomimo że tylko niewielka ich liczba jest aktualnie widoczna, co pociąga za sobą niską wydajność. Pomimo tego faktu, *Z-bufor* jest powszechnie stosowanym algorytmem w końcowym etapie wyznaczania widoczności obiektów w procesie renderowania sceny.

2.1.2. Sortowanie względem głębokości

Idea tego algorytmu [NNS72] polega na rasteryzacji prymitywów w porządku malejącej odległości od punktu widzenia. Algorytm można opisać w postaci sekwencji następujących kroków:

1. Sortowanie wszystkich prymitywów względem ich najdalszej współrzędnej Z.
2. Rozwiązanie możliwych cyklicznych zależności poprzez podział prymitywów na mniejsze.
3. Rasteryzacja w kolejności od najdalszych do najbliższych.

Uproszczona wersja tego algorytmu pomijająca drugi punkt zwana jest **algorytmem malarza** (*Painter's algorithm*) z powodu podobieństwa do sposobu w jaki malarz maluje obraz poprzez zamalowywanie dalszych obiektów tymi położonymi bliżej.

2.2. BSP-rendering

Podstawową słabością opisanych poprzednio metod jest rasteryzacja wszystkich, nawet tych niewidocznych, przesłoniętych przez inne obiektów. Interesujące z tego punktu widzenia okazuje się *drzewo BSP*, gdyż jako struktura danych opisująca scenę $3D$ stoi u podwalin wielu efektywnych algorytmów wyznaczania tylko jej widocznych fragmentów.

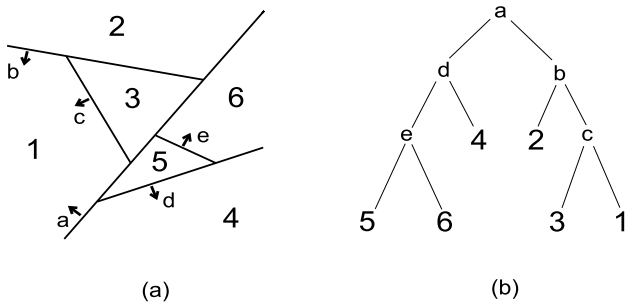
2.2.1. Drzewo BSP

Drzewo BSP to binarne drzewo będące rekurencyjnym, hierarchicznym podziałem $n-$wymiarowej przestrzeni przy pomocy hiperpłaszczyzn. W procesie podziału zadaną podprzestrzeń dzieli się hiperpłaszczyzną na dwie podprzestrzenie, które dalej mogą być rekurencyjnie dzielone w ten sam sposób.

Drzewa BSP dla $n = 3$ są powszechnie stosowanymi w grafice komputerowej strukturami danych wprowadzającymi porządek do reprezentacji sceny $3D$. Określają one podział przestrzeni przy pomocy płaszczyzn. W jednej z pierwszych prac na ten temat [FKN80] zostały opisane: reprezentacja sceny $3D$ złożonej z trójkątów w postaci drzewa BSP i algorytm służący do efektywnego usuwania powierzchni niewidocznych, zaś w pracy [Nay92] zamieszczono ciekawą metodę rzutowania sceny na płaszczyznę, która jest reprezentowana przez $2D$ drzewo BSP, otrzymywane przez algorytm rzutujący scenę $3D$ reprezentowaną przez $3D$ drzewo BSP. Stosowane jest także jako struktura przyśpieszająca znajdowanie przecięć w algorytmach **ray-tracing** [SS92], czy też dla potrzeb kolizji i interakcji pomiędzy obiektami świata $3D$ [Ost03].

W przypadku $n = 2$ mamy do czynienia z podziałem płaszczyzny przy pomocy prostych. Rysunek 2.1 obrazuje relację pomiędzy takim podziałem i odpowiadającym mu drzewem BSP. Pierwszy z podziałów, przy pomocy prostej **a** dzieli całą płaszczyznę na dwie półpłaszczyzny, które w wyniku kolejnych iteracji są dzielone na mniejsze obszary. Orientacja hiperpłaszczyzny będącej tutaj prostą jest oznaczona strzałką i odnosi się do kolejności węzłów w drzewie BSP. Każda z hiperpłaszczyzn odpowiada wewnętrznemu węzłowi drzewa, prawe poddrzewo reprezentuje obszar znajdujący się po stronie wskazywanej przez strzałkę, zaś lewe poddrzewo - obszar przeciwległy. Liście drzewa reprezentują obszary oznaczone cyframi, będące rezultatem pierwotnego podziału.

Dla $n = 1$ struktura BSP przybiera postać binarnego drzewa poszukiwań, czyli generycznej struktury służącej do efektywnej implementacji operacji wyszukiwania na zbiorach danych.



Rysunek 2.1: **(a)** Podział płaszczyzny przy pomocy prostych. **(b)** Drzewo BSP zbudowane na jego podstawie.

2.2.2. Rozszerzone drzewa BSP

Powszechnym zjawiskiem jest rozszerzanie generycznych struktur danych o dodatkowe informacje. W przypadku drzew BSP, liście drzewa mogą zawierać kolekcję obiektów leżących wewnątrz odpowiadającej liściowi podprzestrzeni [RE01], podobnie zaś wewnętrzne węzły mogą zawierać obiekty leżące na hiperpłaszczyźnie podziału [FKN80]. Motywacją dla takiej rozbudowy struktury drzewa BSP jest problem widoczności powierzchni. Dla dowolnej pozycji obserwatora przejście drzewa w określonym porządku dostarcza uporządkowanej względem widoczności permutacji obiektów znajdujących się w drzewie, co wykorzystane jest na przykład w *algorytmie malarza*.

2.2.3. Algorytm malarza z użyciem drzew BSP

W pracy [FKN80] został zaprezentowany *algorytm malarza* używający drzew BSP do wyznaczenia prawidłowego porządku rysowania. Kluczowym zagadnieniem jest budowa drzewa BSP odwzorowującego strukturę sceny. Poniższa rekurencyjna procedura jest rozwiązaniem tego problemu:

1. Niech S będzie zbiorem wszystkich wielokątów sceny.
2. Jeśli $|S|$ jest równe 1 to stwórz liść drzewa zawierający odniesienie do jedynego wielokąta z S.
3. Wybierz płaszczyznę P, która dzieli zbiór S na zbiory S- i S+, zawierające wielokąty leżące odpowiednio po tylnej i przedniej stronie płaszczyzny P. Wielokąty przecinające się z płaszczyzną podziału są dzielone, zaś odpowiednie ich części są dodawane do zbiorów S- i S+.
4. Skonstruuj wewnętrzny węzeł drzewa N posiadający P jako płaszczyznę podziału. Skojarz z nim wszystkie wielokąty leżące na płaszczyźnie podziału.
5. Stwórz dzieci węzła N rekurencyjnie powtarzając kroki 2 do 5, używając S- do stworzenia lewego i S+ do stworzenia prawego poddrzewa.

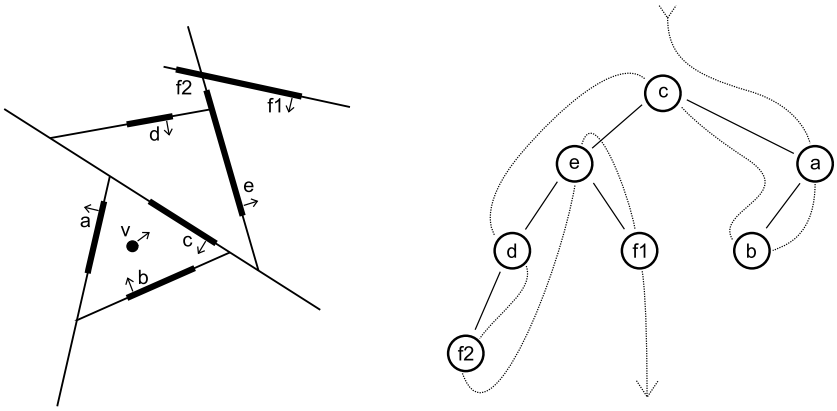
Decydującym krokiem konstrukcji drzewa BSP jest wybór płaszczyzny podziału (krok 3). Najczęściej ogranicza się wybór płaszczyzn podziału do tych, które zawierają się w wielokątach sceny. Przy takim podejściu do konstrukcji drzewa w każdym kroku istnieje skończona liczba wyborów. Porządek w jakim płaszczyzny podziału są wybierane jest bardzo ważny. Istnieje wiele heurystyk które minimalizują liczbę podziałów wielokątów, a więc w konsekwencji minimalizują rozmiar konstruowanego drzewa. Dla potrzeb renderowania, w szczególności dla sortowania ścian względem głębokości drzewo nie musi być zbalansowane. W praktyce nawet drzewo zdegenerowane do listy będzie

skutkować taką samą wydajnością renderowania jak doskonale zrównoważone drzewo. Mimo tego dąży się do zrównoważenia drzewa, gdyż może być ono stosowane do innych zadań takich jak detekcja kolizji, gdzie zrównoważone drzewo ma istotny wpływ na efektywność algorytmów.

Drzewo BSP jest używane w algorytmie malarza do otrzymania posortowanej w kierunku od najdalszych do najbliższych listy wielokątów sceny. Zagadnienie widoczności jest tutaj rozwiązywane w przestrzeni obrazu poprzez zamalowywanie dalszych wielokątów przez bliższe. Mając dany punkt widzenia C procedura renderowania drzewa BSP wygląda następująco:

1. Niech v będzie korzeniem drzewa.
2. Jeśli v jest liściem to narysuj wszystkie wielokąty z nim związane i zakończ.
3. Ustal pozycję punktu C względem płaszczyzny podziału P skojarzonej z węzłem v
4. Rekurencyjnie wykonaj punkty 2-4 dla poddrzewa będącego po przedniej, stronie płaszczyzny P, potem narysuj wielokąty skojarzone z węzłem C, następnie rekurencyjnie wywołaj punkty 2-4 dla poddrzewa będącego po tylnej stronie płaszczyzny podziału P.

Rysunek 2.2 przedstawia przykład sceny $2D$ zawierającej 6 odcinków oraz jej podział przy pomocy prostych. W wyniku podziału jeden z odcinków został podzielony (f). Na równoważnym mu drzewie BSP zaznaczono kolejność odwiedzania węzłów przez procedurę. Obserwator znajduje się w punkcie V.

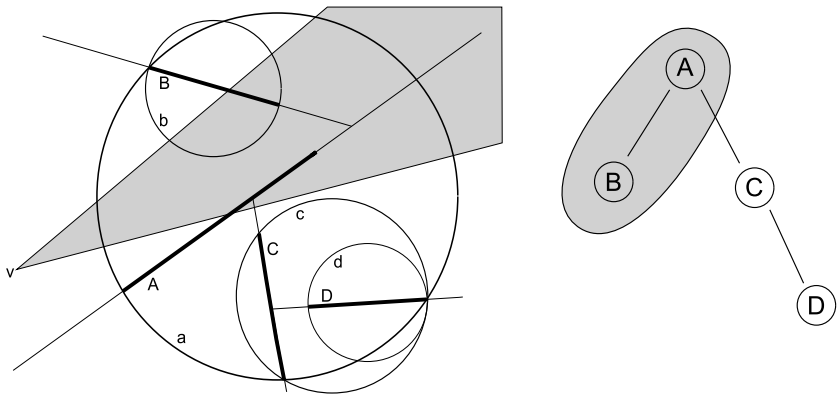


Rysunek 2.2: Podział płaszczyzny, wynikające z niego drzewo BSP oraz kolejność jego przejścia przy rysowaniu z punktu V.

Przedstawiony algorytm w swojej czystej postaci ma niską przydatność i w praktyce nie jest stosowany. Podstawową wadą jest przetwarzane wszystkich

wielokątów składających się na scenę, co z reguły nie jest do zaakceptowania ze względu na wielkość danych. Budowa drzewa BSP jest procesem złożonym i długotrwałym, dlatego w bezpośredni sposób nie można użyć go do reprezentacji dynamicznie zmieniającej się sceny.

W książce [Abr97] opisano rozszerzenie drzew BSP pozwalające na odrzucanie całkowicie niewidocznych poddrzew. Opiera się ono na przypisaniu do każdego węzła drzewa bryły brzegowej, opisującej wszystkie znajdujące się w nim obiekty. W tym konkretnym przypadku użyto sfer, w innych implementacjach używa się także prostopadłościanów ułożonych w dowolnej pozycji (**Oriented bounding box - OBB**) bądź też położonych równo z osiami układu współrzędnych (**Axis-aligned bounding box - AABB**). Przed przetworzeniem każdego z węzłów drzewa sprawdza się, czy część wspólna bieżącej bryły widzenia i bryły brzegowej skojarzonej z węzłem jest pusta. W takim przypadku pomija się przetwarzanie tego węzła, gdyż na pewno jest on niewidoczny.



Rysunek 2.3: Podział płaszczyzny z przypisanymi do węzłów drzewa okręgami opisującymi obiekty znajdujące się w poszczególnych poddrzewach. Testowanie przecinania się bryły widoczności i okręgów opisujących pozwala na odrzucenie poddrzewa **C** w algorytmie *BSP-render*.

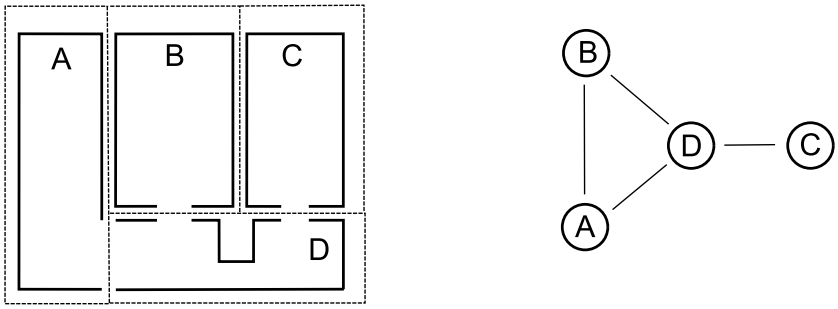
2.3. PVS-rendering

Pojęcie **PVS** (*Possible Visibility Sets* - zbiory możliwych widoczności) wprowadził w pracy [ARFPB90], podając przy tym pierwszy praktyczny algorytm korzystający z tej techniki. Algorytmy rozwiązujące problem widoczności korzystające z **PVS** należą do grupy do grupy algorytmów konserwatywnych, określających *widoczność z obszaru*.

Podstawowym założeniem wszystkich metod bazujących na PVS jest istnienie podziału sceny na zbiór sektorów. Można go przykładowo utożsamiać z naturalnym podziałem wnętrz budynków na poszczególne pomieszczenia,

gdzie sektory z grubsza odpowiadają pokojom i korytarzom. Na podstawie podziału sceny na sektory oraz jej geometrii wyznacza się *graf PVS* zdefiniowany w następujący sposób:

- Węzłami grafu są sektory wynikające z podziału sceny.
- Krawędzie w grafie odpowiadają istnieniu widoczności pomiędzy sektorami. Bardziej formalnie, pomiędzy węzłami i oraz j istnieje krawędź $\iff \exists$ punkty $X_i \in S_i, X_j \in S_j$ że X_i jest widoczny z X_j.



Rysunek 2.4: Podział sceny 2D na cztery sektory (A,B,C,D) oraz zbudowany na ich podstawie graf PVS.

Ogólny szkic algorytmu *PVS rendering* wygląda następująco:

1. Ustal sektor w którym znajduje się kamera, niech S_i będzie tym sektorem.
2. Narysuj obiekty znajdujące się w sektorze S_i oraz w sektorach będących jego sąsiadami

Największą trudnością jest skonstruowanie *grafu PVS*. Jest to złożone zagadnienie, które należy rozpatrywać w wielu aspektach:

- Podziału na sektory.
- Określenia relacji widoczności.
- Reprezentacji grafu.

Ponadto należy wspomnieć, że *PVS rendering* samodzielnie nie rozwiązuje w pełni zagadnienia widoczności [1], w szczególności nie dostarcza informacji o wzajemnym położeniu widocznych obiektów względem współrzędnej Z. Dodatkowe kroki muszą być podjęte w celu prawidłowego renderowania sceny, wystarczającym (i najczęściej stosowanym) rozwiązaniem jest użycie algorytmu *Z-bufora*.

[1] W sensie przytaczanych wcześniej algorytmów jak *Z-bufor* bądź *BSP rendering*

2.3.1. Podział na sektory

Zagadnienie to polega na podziale przestrzeni obejmującej scenę na rozłączne obszary, dające w sumie całą rozpatrywaną przestrzeń. Odnosi się to do wyznaczenia wierzchołków w *grafie PVS*. Nie istnieją żadne warunki czy też ograniczenia na wynikowy zbiór sektorów, należy jednak zdawać sobie sprawę, że wybór takiego anie innego podziału będzie rzutował na złożoność pozostałych kroków oraz decydował o późniejszej sprawności algorytmu *PVS rendering*. Trudno jednoznacznie zdefiniować pojęcie "dobroci„ podziału, można w tym miejscu wymienić ogólne wskazówki [Abr97]:

- **Wielkość sektorów.** Zaletą mniejszych sektorów jest bardziej dokładne (mniej konserwatywne) określanie relacji widoczności, co w konsekwencji będzie przekładać się na mniejszą ilość przetwarzanych prymitywów graficznych. Wadą będzie duży rozmiar grafu CPG, problem w szczególności dotyczy krawędzi, których wzrost jest w pewnym przybliżeniu kwadratowy w stosunku do wzrostu liczby sektorów. Wyważenie pomiędzy większymi i mniejszymi sektorami jest trudnym problemem.

- **Kształt sektorów.** To jaki kształt mają sektory determinuje relację widoczności między nimi. Za optymalny można uznać taki, który minimalizuje widoczność pomiędzy sektorami. W przypadku ręcznego, bądź półautomatycznego ("niejawnego„ dokonywanego przez narzędzia edycyjne) podziału na sektory często korzysta się istniejącego, naturalnego podziału sceny na pomieszczenia, korytarze, czy też przejścia.

Możemy wyróżnić dwa główne modele uzyskiwania podziału:

- **Dostarczenie podziału z zewnątrz.** W tym modelu podział jest integralną częścią danych reprezentujących scenę $3D$. Jednym z pierwszych pomysłów, do dzisiaj najbardziej rozpowszechnionym jest zintegrowanie edycji (tworzenia) sceny 3D wraz z jej podziałem na sektory poprzez posługiwanie się komponentami edycyjnymi takimi jak pomieszczenia lub rejony reprezentujące sektory wraz z dbaniem o takie ułożenie komponentów reprezentujących sektory aby zachowane były warunki podziału.

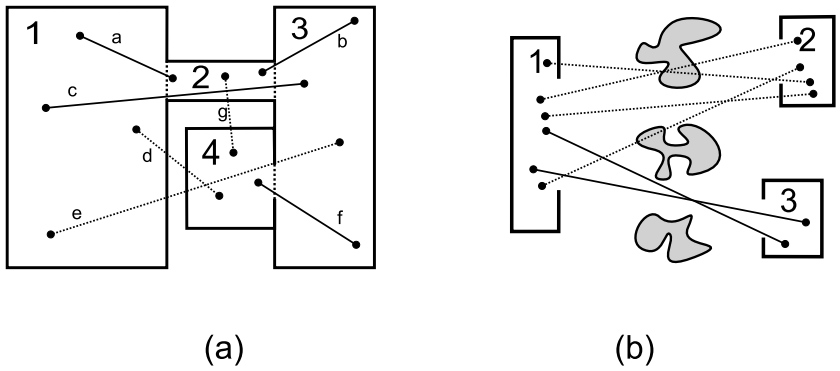
- **Automatyczny podział surowej sceny 3D.** Przestrzeń dzieli się sztywno wedle z góry przyjętych reguł, losowo lub równomiernie dzieląc ją na sześciany [YR95], bądź też stosując drzewo BSP zbudowane na podstawie geometrii sceny [RE01] [Abr97]. Dla terenów otwartych, bądź reprezentujących miejską zabudowę stosuje się też techniki kombinowane, w pracy [WWS00] na bazie triangulacji płaszczyzny terenu buduje się sektory poprzez "wyciągnięcie„ z nich graniastosłupów. Różne metody triangulacji płaszczyzn opisano w pracy [Tel92].

2.3.2. Wyznaczenie relacji widoczności

Zagadnienie *widoczności z obszaru* jest uważane za trudniejsze od zagadnienia *widoczności z punktu*, gdyż właściwie jest ono czterowymiarowe. Dlatego najbardziej popularne są metody uproszczone, kosztowniejsze obliczeniowo. W pionierskiej pracy [ARFPB90] użyto śledzenia promieni pomiędzy losowo wybranymi punktami ze sceny w celu aproksymacji relacji widoczności, analogicznej metody użył J. Carmack w preprocessingu widoczności w grze **Quake** [Abr97]. W obu opisywanych powyżej przypadkach użyto prostego algorytmu rozwiązującego problem "metodą siłową„:

1. Niech graf PVS nie zawiera żadnych krawędzi
2. Powtarzaj punkty 3,4 aż do osiągnięcia zadowalających rezultatów
3. Wylosuj dwa punkty $P1$ i $P2$ a następnie określ sektory $S1$ i $S2$ w których one się znajdują.
4. Jeśli $S1 \neq S2$ i punkty $P1$ i $P2$ są bezpośrednio widoczne to dodaj krawędź $< S1, S2 >$ do grafu.

Algorytm ten pomimo swojej niebywałej prostoty ma dwie poważne wady: niedoszacowywuje zbiór krawędzi grafu PVS, oraz wymaga bardzo dużej liczby iteracji w celu wyznaczenia relacji widoczności z rozsądną jakością. Rysunek 2.5(b) przedstawia sytuację niedoszacowania widoczności, w której sektory 1 i 3 nie zostały wykryte jako *potencjalnie widoczne*.

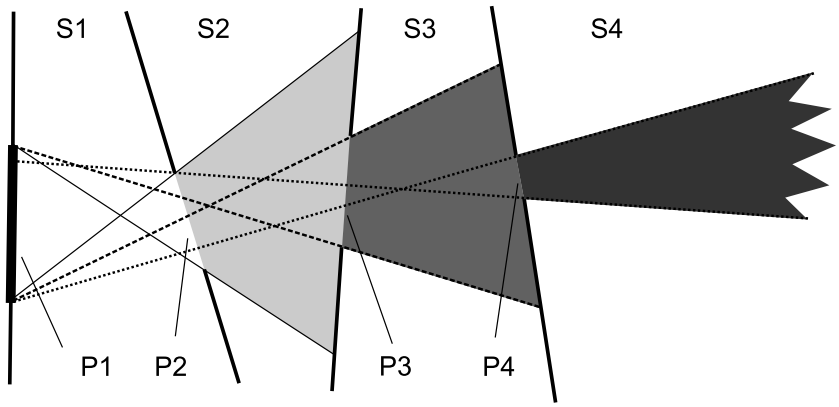


Rysunek 2.5: **(a)** Losowo wybrane pary punktów określające relację widoczności. **(b)** Niefortunny wybór punktów może spowodować niedoszacowania widoczności (sektory 1 i 2). Linią ciągłą połączono pary punktów bezpośrednio widocznych, linią przerywaną - pary punktów niewidocznych.

To "ślepe„ szukanie widoczności pomiędzy sektorami zostało w pracach [TS91] [Tel92] ulepszone przez wprowadzenie **portali**, czyli wielokątów określających relację widoczności pomiędzy sąsiednimi sektorami, a następnie ukierunkowaną propagację promieni przez ich sekwencje. W tejże pracy [Tel92] zaprezentowano algorytm używający techniki *przycinania obszaru antycienia* [2] o następującym schemacie:

1. Dla każdego sektora S wykonaj:
2. Dla każdego portalu $P \in S$ wykonaj:
3. Interpretując P jako obszarowe źródło światła stwórz bryłę antycienia emitowanego przez to światło. Propaguj ją przez sekwencję sektorów przycinając ją za każdym razem przez obszary portali do prawidłowego obszaru antycienia. Wszystkie odwiedzone w ten sposób sektory są *potencjalnie widoczne z* S

Rozszerzenie tej metody na przestrzeń trójwymiarową nie jest trywialne. W pracy [Bit02] opisano algorytm operujący w *przestrzeni Plückera [Dur99]*, operujący na 5-cio wymiarowej reprezentacji danych oraz jego praktyczną implementację. Autorzy podkreślają duże trudności na drodze do uzyskania efektywnej i stabilnej numerycznie implementacji. Rysunek 2.6 ilustruje przycinanie obszaru antycienia przez sekwencję trzech portali.



Rysunek 2.6: Propagacja i przycinanie obszaru antycienia z portalu $P1$ poprzez portale $P2, P3, P4$.

Z innych rozwiązań warto wymienić [ARFPB90], gdzie opisano algorytm o złożoności sześciennej względem liczby wielokątów, w którym wpierw spraw-

[2] Określa się trzy klasy oddziaływań źródeł światła: Obszary bezpośrednio oświetlane (L), obszary będące w półcieniu (P) i obszary nieoświetlone (U). Obszar antycienia jest sumą obszarów L i P

dza się konserwatywnie widoczność pomiędzy wszystkimi parami portali i wielokątów sceny, zaś później na tej postawie tworzy się relację widoczności.

2.3.3. Reprezentacja grafu PVS

W pewnych przypadkach relacja widoczności może być bardzo dużym zbiorem danych. W najbardziej pesymistycznym, gdy widoczność jest pomiędzy wszystkimi sektorami jej rozmiar jest równy kwadratowi liczby sektorów. Ze względu na bardzo dużą złożoność rzeczywistych scen 3D i zazwyczaj niewiele mniej licznym jej podziałem na sektory zagadnienie efektywnej reprezentacji *grafu CPG* może nie być trywialne. Jest to tym bardziej istotne, że liczba krawędzi w grafie może być duża, wielokrotnie większa od liczby wierzchołków. Przypadek gęstego grafu jest jednak mało prawdopodobny, lecz jeśli wystąpi, co oznacza istnienie widoczności pomiędzy prawie każdym sektorem to wnioskuje się, że późniejsze użycie algorytmu **PVS** nie przyniesie istotnego przyśpieszenia renderowania sceny. Taka sytuacja świadczy o niefortunnym wyborze sektorów bądź też poważnym przeszacowaniu widoczności na etapie jej wyznaczania.

Mimo tego nie należy lekceważyć zagadnienia reprezentacji grafu CPG. Jednym z rozwiązań "oszczędzającym„ pamięć jest użycie *list krawędzi* dla każdego wierzchołka, o złożoność pamięciowej $\mathcal{O}(m+n)$m gdzie m to liczba wierzchołków, a n liczba krawędzi grafu. Taką reprezentację opisano w pracach [RE01] [ARFPB90].

Inną formą reprezentacji grafu jest *macierz sąsiedztwa* o wymiarach $n \times n$. Może się wydawać, że jest ona tutaj bezużyteczna ze względu na fakt, że n może wynosić nawet kilkaset tysięcy, co przy użyciu najbardziej zwartej reprezentacji jako wektorów bitowych daje rozmiar danych rzędu megabajtów. Okazuje się, że stosując metody kompresji wektorów bitowych można osiągnąć zadowalające rezultaty w postaci niskiej zajętości pamięci i efektywnych metod dostępu. M. Abrash w [Abr97] podaje, że typowa scena do gry **Quake** składała się z około 10000 trójkątów, zaś skompresowana metodą *Run-length* macierz sąsiedztwa zajmowała około 20 kilobajtów. W pracy [vdPS99] szerzej poruszono to zagadnienie. Analiza typowych macierzy sąsiedztwa grafu CPG pokazuje, że po pierwsze jest to macierz rzadka, po drugie bardzo często w poszczególnych wektorach występują długie sekwencje zer i jedynek. Pozwoliło to autorom opracować dwie metody kompresji:

- **Bezstratne** Zastosowanie kodowanie długich sekwencji zer bądź jedynek krótszymi kodami.
- **Stratne** Użycie kontrolowanego wstawiania do macierzy jedynek w celu złączenia dwóch sąsiednich wierszy bądź kolumn macierzy, bądź też w celu sklejania mniejszych sekwencji jedynek w większe. Prowadzi to do

efektywniejszej kompresji takiej macierzy metodami bezstratnymi w niewielki tylko sposób degradując *graf PVS* do postaci bardziej konserwatywnej.

Najlepsze rezultaty osiągano wpierw stratną, potem bezstratną kompresję, otrzymane rezultaty porównywano z kompresją przy pomocy programu **gzip** otrzymując znacznie lepsze (4-10 krotnie) mniejsze zbiory danych.

2.4. Portal-rendering

Portal rendering jest metodą wyznaczającą zbiór *potencjalnie widocznych sektorów* w czasie renderowania. Zaliczana jest do metod wyznaczających *widoczność z punktu*, gdyż w obliczeniach wykorzystuje pozycję i obszar widzenia kamery. Szczególnie dobrze nadaje się do złożonych, mocno zasłoniętych scen takich jak modele architektoniczne czy też scenerie przedstawiające zamknięte wnętrza. Dodatkowo niewielkim nakładem wymagań w stosunku do użytego mechanizmu rasteryzacji[3] umożliwia implementację dynamicznych, rekurencyjnych odbić lustrzanych a nawet bardziej skomplikowanych efektów zmiany widoczności takich jak jej translacja do innej lokalizacji po napotkaniu "portalu„ przenoszącego widzialność w inne miejsce.

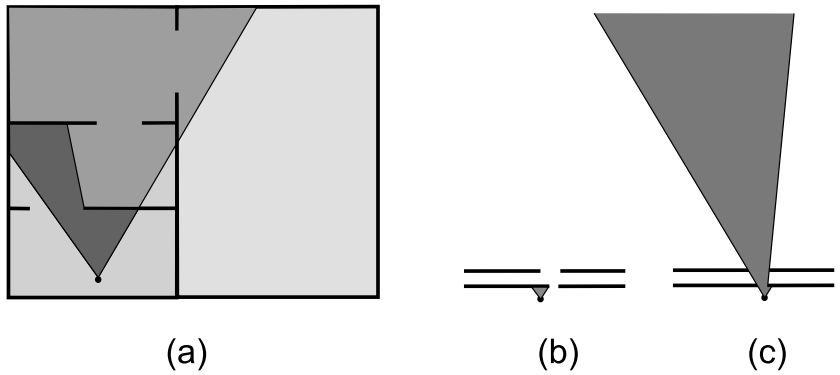
2.4.1. Motywacja

Typowe scenerie reprezentujące wnętrza charakteryzują się tym, że ilość obiektów aktualnie widocznych przez obserwatora (nie przesłoniętych przez ściany) jest z reguły znacznie mniejsza niż obiektów teoretycznie widocznych przez obserwatora (zawierających się w bryle widzenia). Z reguły jest też znacznie mniejsza niż ilość obiektów widocznych z sektora w którym znajduje się obserwator. Widoczność nosi znamiona zadania źle uwarunkowanego: nawet małe zmiany punktu widzenia mogą spowodować duże zmiany widoczności, dlatego metody rozwiązujące widoczność *z punktu* mogą osiągać znacznie lepsze rezultaty w eliminacji niewidocznych powierzchni w porównaniu z metodami rozwiązującymi widoczność *z obszaru*, do jakich należy na przykład *PVS rendering*.

W wielu interaktywnych zastosowaniach wymaga się, aby istniała możliwość dynamicznej modyfikacji sceny. Sprawia to trudności w rozwiązaniach bazujących na "statycznie„ wyliczonej widoczności, która z jednej strony jest niezwykle trudna do uaktualnienia w odpowiedzi na najmniejsze nawet zmiany sceny, z drugiej zaś bardzo czasochłonna do ponownego wyliczenia. *Portal*

[3]Chodzi o możliwość ustawienia w dowolnym momencie obszaru przycinającego rysowanie do zadanego wielokąta wypukłego, wymaganie spełnione przez współcześnie stosowane oprogramowanie udostępniające rysowanie grafiki $3D$

rendering stosunkowo łatwo obsługuje dynamiczną zmianę sceny, wystarczy tylko, aby wszelkie zmiany miały odpowiadające im odwzorowanie w zmianach portali i sektorów. W wielu przypadkach jest to niezwykle łatwe, ze względu na istnienie naturalnego odwzorowanie portali i sektorów na obiekty używane w czasie modelowania: pomieszczenia, otwory drzwiowe i okienne.



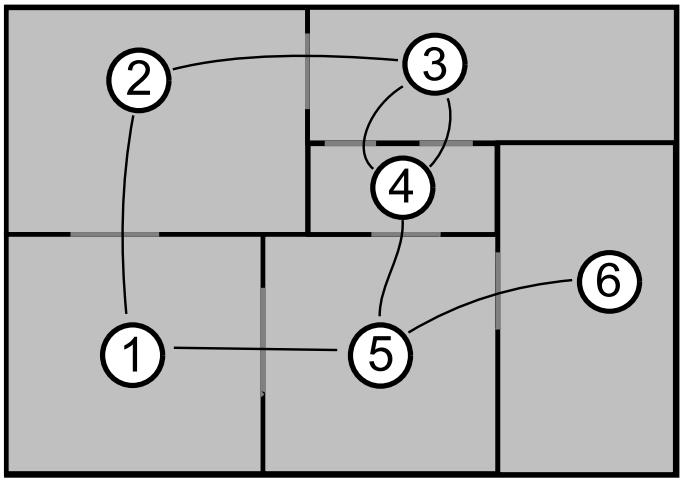
Rysunek 2.7: **(a)** Obszar potencjalnie widoczny z sektora w którym znajduje się kamera (jasnoszary), obszar znajdujący się w bryle widzenia (szary), obszar widoczny przez kamerę (ciemnoszary). **(b)**, **(c)** Nawet niewielka zmiana położenia może spowodować dużą zmianę widoczności.

2.4.2. Wcześniejsze opracowania

W pracy [Jon71] zaproponowano podział sceny na portale i sektory w celu rozwiązania problemu usuwania niewidocznych linii. Autor manualnie dzielił model na wypukłe sektory połączone przeźroczystymi portalami, następnie każdy z wielokątów sceny przypisywał do jednego i tylko jednego sektora. Przeglądanie sceny rozpoczynało się sektorami, w sektorze zawierającym kamerę, rysowane były wszystkie wielokąty w nim zawarte, następnie algorytm rekurencyjnie przetwarzał sektory widoczne przez portale, przycinając nimi bryłę widzenia i używając jej jako maski służącej do obcinania rysowania poszczególnych sektorów. W publikacji [LG95] zaproponowano uproszczoną wersję, w której w miejscu portali użyto przycinania bryły widzenia przez prostokąty, będące konserwatywną aproksymacją rzeczywistego portalu, podejście to umożliwiało znacznie bardziej efektywną (wspomaganą sprzętowo) implementacje, niezależnie zaś [LG95] i [Tel92] opisują rozszerzenie umożliwiające rekurencyjne odbicia lustrzane, gdzie lustra reprezentowane są przez specjalnie oznaczone portale, po napotkaniu których dokonuje się zmiany punktu widzenia zgodnie z fizycznym zjawiskiem odbicia światła oraz wprowadza się pewne dodatkowe zmiany związane ze zmianą skrętności układu odniesienia, następnie algorytm rekurencyjnie kontynuuje swoje działanie.

2.4.3. Opis algorytmu

Ideą algorytmu *Portal rendering* jest identyfikacja niewidocznych (zasłoniętych) obiektów poprzez ustalenie, że są niewidoczne przez sekwencje portali. Algorytm pracuje na reprezentacji sceny 3D w postaci *grafu CPG*, czyli podziału zbioru wielokątów na rozłączne sektory będące wielościanami wypukłymi[4] oraz portalach, czyli przeźroczystych wielokątach wypukłych łączących sąsiednie sektory, określających w ten sposób relację widoczności i sąsiedztwa między nimi. Portal posiada interesującą własność: nic nie jest widoczne za nim jeśli on sam jest niewidoczny. Obserwator znajdujący się w danym sektorze w oczywisty sposób widzi ten sektor oraz te, które są widoczne przez sekwencje portali. Algorytm wykorzystuje tą właściwość propagując widoczność w *grafie CPG* przez sekwencje portali odwiedzając w ten sposób widoczne sektory. Czym jest propagowana widoczność ? Początkowo jest ona *bryłą widzenia*, w kształcie nieskończonej piramidy o wierzchołku zaczepionym w punkcie, w którym znajduje się obserwator. Przy przekraczaniu portali jest przycinana przez ich granice.



Rysunek 2.8: Przykład podziału na sektory i portale.

Chwilowo pomijamy aspekty wyznaczenia *grafu CPG* oraz warunków jakie musi on spełniać, nie jest to potrzebne do wyjaśnienia istoty działania i zostanie szczegółowo rozwinięte w dalszej części. Dodatkowo nie tracąc istoty zagadnienia można uprościć je do przypadku dwuwymiarowego.

Rysunek 2.8 przedstawia przykładowy podział sceny na 6 sektorów prostokątnych sektorów, połączenia (portale) pomiędzy nimi zaznaczono na czerwono. Graf ten można odnieść do rzeczywistego obiektu, będącego rzutem

[4]warunek wypukłości często nie jest stosowany

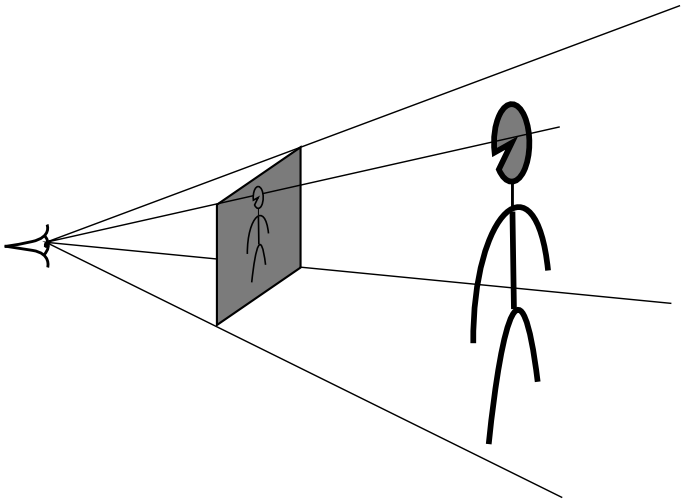
(planem) budynku, zawierającym 6 pomieszczeń, między którymi istnieją połączenia (otwory drzwiowe), reprezentowane przez portale.

Ogólny schemat algorytmu *portal rendering* przedstawia się następująco:

```
S ← sektor w którym znajduje się obserwator
wywołaj Portal-render( S ) i zakończ

procedura Portal-render(Sektor S)
    dla każdego portalu P należącego do S  powtarzaj
        jeśli P jest widoczny to
            SS ← sektor sąsiedni do P
            rekurencyjnie wywołaj Portal-render(SS)
```

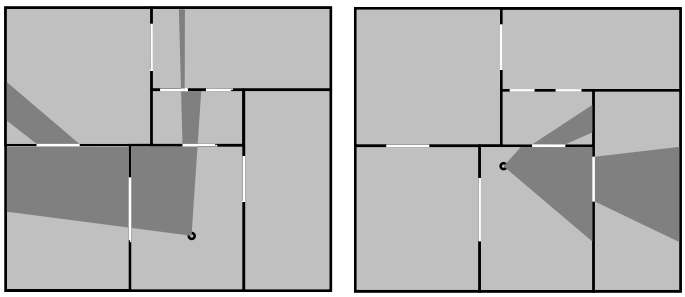
Konkretyzacja wyrażenia: **jeśli portal** P **jest widoczny** wymaga odwołania się do modelu kamery. Nie tracąc na ogólności algorytmu wystarczy ograniczyć się do modelu kamery, w którym rzutuje się przestrzeń 3D na dwuwymiarową płaszczyznę obrazu umieszczoną przed obserwatorem[5]. Konsekwencją tego modelu jest **piramida widzenia**, czyli dla danej instancji kamery wycinek przestrzeni, który jest widoczny i podlega rzutowaniu na płaszczyznę obrazu. Wszystko poza nim jest niewidoczne i nie musi podlegać obrazowaniu. Piramidę widzenia reprezentuje się zazwyczaj jako cztery płaszczyzny zorientowane do wewnątrz, określającą piramidę zaczepioną w punkcie położenia kamery i rozciągającą się w nieskończoność.



Rysunek 2.9: Ilustracja kamery i bryły widzenia.

[5]Można to sobie wyobrazić jako malowanie na szybie umieszczonej przed obserwatorem dokładnie tego, co przez nią widać.

Odpowiedź na pytanie **Czy portal jest widoczny** sprowadza się do sprawdzenia, czy portal zawiera się w piramidzie widzenia. Z założenia jest on wielokątem wypukłym leżącym na płaszczyźnie, zaś przecięcie piramidy widzenia płaszczyzną nierównoległą do żadnej z jej płaszczyzn daje w rezultacie wielokąt wypukły, więc problem ma rozwiązanie w postaci zbadania, czy część wspólna wielokąta powstałego przez przecięcie płaszczyzną portalu piramidy widzenia i wielokąta określającego portal nie jest pusta. W rekurencyjnym wywołaniu **portal-render** dla sąsiedniego sektora przekazywana jest zmodyfikowana piramida widzenia, zbudowana z wielokąta wypukłego będącego częścią wspólną portalu i przecięcia piramidy widzenia z płaszczyzną portalu.



Rysunek 2.10: Piramida widzenia przycinana przez kolejne iteracje algorytmu.

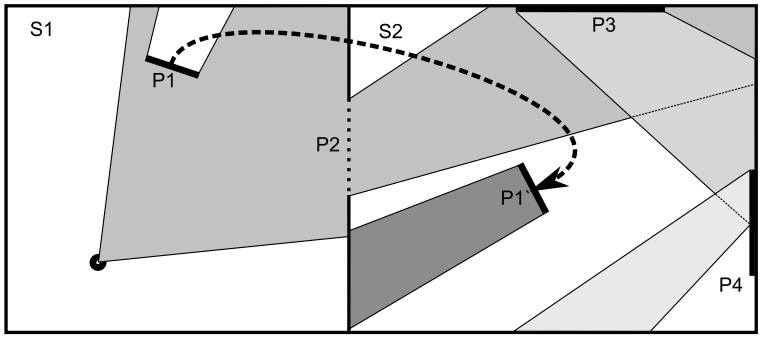
2.4.4. Translacja i odbicie widoczności.

Tradycyjna definicja portalu określa go jako *przeźroczysty wielokąt, wyznaczający relację widoczności między sąsiednimi sektorami*. W publikacjach [LG95] i [Tel92] opisano możliwość modyfikacji parametrów kamery przez portal. Daje to możliwość wpływu na sposób widzenia znajdującego się za portalem otoczenia. Najbardziej popularne transformacje to:

- **Translacja** Po napotkaniu portalu punkt widzenia jest przenoszony do zupełnie innej lokalizacji. W grach 3D (na przykład **Quake III**) w połączeniu z translacją pozycji obserwatora przy przejściu przez płaszczyznę portalu wygląda to jak wirtualny *teleport*, przez który wpierw można zajrzeć do innej lokalizacji.
- **Odbicie lustrzane** Portal reprezentuje lustro dające możliwość rekurencyjnego odbijania zawartości sceny.

Efektem przekraczania portalu jest modyfikacja parametrów kamery: jej położenia i orientacji, zgodnie ze skojarzoną z portalem transformacją. Uaktualniana jest bryła widzenia oraz pewne dodatkowe dane, zależne od konkretnej

implementacji algorytmu. Należy podkreślić wyjątkową łatwość implementacji wyżej wspomnianych efektów w porównaniu z innymi technikami ich uzyskiwania, w szczególności istotna jest możliwość rekurencyjnego nakładania się kolejnych tego typu transformacji.



Rysunek 2.11: Przykład sceny z trzema typami portali. Portal *P2* jest normalnym portalem propagującym widoczność bez żadnych zmian, portale *P1*, *P3*, *P4* pełnią rolę luster, zaś portal *P1* dokonuje translacji widoczności do portalu *P1'*.

2.4.5. Podział na sektory i portale

Na sukces algorytmu *Portal-rendering* pracuje wiele czynników, zaczynając od tych najbardziej kluczowych[6].

- Renderowana scena powinna reprezentować środowisko w miarę zamknięte. Najlepsze są scenerie reprezentujące wnętrza budynków, które wręcz bezpośrednio reprezentują koncepcje sektorów i portali w postaci pomieszczeń oraz łączących je otworów drzwiowych i okiennych. Znane są także inne aplikacje, na przykład w publikacji [HMK+97] zaprezentowano system do wizualizacji wnętrza ludzkiego jelita grubego, którego docelowym zastosowaniem było poszukiwanie objawów raka. Z reguły scenerie reprezentujące otwarty teren nie są przyjazne dla algorytmu *Portal-rendering*, takie przypadki najlepiej przetwarzać metodami należącymi do zagadnienia *outdoor-renderingu*.
- Istotna jest konstrukcja *grafu CPG*, który musi spełniać wiele warunków:
  - Musi być prawidłowy: wszystkie wielokąty sceny muszą należeć do dokładnie jednego sektora, zaś każdy obszar widoczny pomiędzy sektorami musi zawierać się w odpowiadającym mu portalu, definiującym relację widoczności pomiędzy sąsiednimi sektorami.

[6]Oczywiście zlekceważenie nawet najmniej istotnego elementu może zdecydować o porażce całości.

  - Musi być "dobry„ , aby późniejsze stosowanie *Portal-renderingu* przyczyniło się do istotnego przyśpieszenia renderowania, oznacza to tyle, że liczba sektorów i portali powinna być znacząco mniejsza [7] niż liczba wielokątów sceny, sektory powinny być możliwie wypukłe i obejmować to, co jest ich naturalnym wyobrażeniem, czyli pojedyncze pomieszczenia, przejścia czy klatki schodowe. Jednakże definicja dobrego podziału jak i jego miara jest trudna do określenia.

- Prawidłowa i efektywna reprezentacja *grafu CPG* i renderowanej sceny.
- Efektywna implementacja algorytmu *Portal-rendering*.

Z reguły dane wejściowe, czyli scena reprezentowana jako zbiór wielokątów jest materiałem będącym efektem pracy twórczo-artystycznej. Jednakże niezwykle trudno jest automatycznie uzyskać dobry i prawidłowy podział sceny na *graf CPG*. W praktyce bardzo często tą odpowiedzialność zrzuca się na barki człowieka, w postaci ręcznego tworzenia tego podziału podczas tworzenia sceny, bądź też jako część późniejszego jej przetwarzania[8], ewentualnie też niejawnie zaszytego w narzędziach do tworzenia i edycji sceny. Istnieją jednak poważne obawy, czy to rozwiązanie będące efektem ucieczki od próby analizy, zrozumienia i wreszcie automatycznego (algorytmicznego) rozwiązania jest jakościowo dobre.

2.5. Occlusion culling

Istnieje duża grupa technik eliminacji powierzchni niewidocznych bezpośrednio wykorzystujących zjawisko przesłaniania dalszych obiektów sceny przez obiekty bliższe. Jest to odwrócenie zasady działania algorytmu *Portal rendering*, w którym jawnie wyznacza się obszary (*portale*), przez które widoczność jest propagowana. Tym przeciwieństwem jest jawne wyznaczanie obszarów zasłaniających (*occluders*) i ich przetwarzaniu jako elementów uczestniczących w ograniczaniu widoczności. Łatwo jest sformułować ogólny algorytm opierający się na powyższej zasadzie:

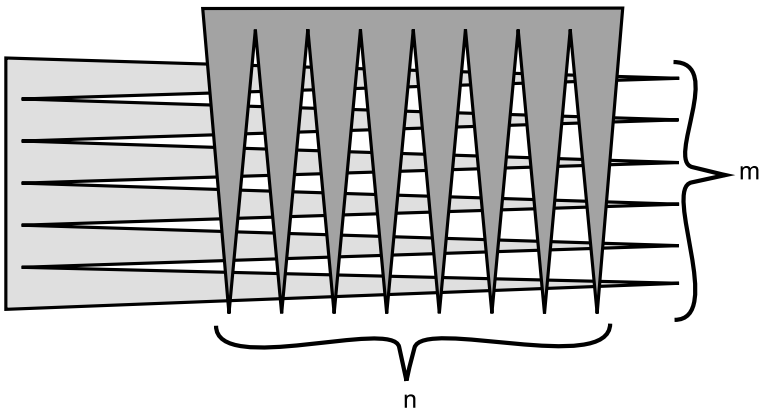
[7] W praktyce kilka rzędów mniejsza.

[8] Praktyka powszechna w segmencie gier komputerowych, na przykład stosowana we wrocławskiej firmie Techland.

Niech V **określa aktualnie niewidoczny obszar**
$V \leftarrow$ **obszar pusty**
Dla każdego obiektu a **ze sceny** S **w kolejności od najbliższych powtarzaj**
    Jeśli obiekt v **nie zawiera się w całości w** V **to**
        narysuj v
        Niech v' **będzie obszarem przesłanianym przez** v
        $V \leftarrow V \bigcup v'$

Przedstawiony algorytm jest pod wieloma względami niepraktyczny. Jako podstawową wadę wskazuje się kosztowną reprezentację obszaru aktualnie niewidocznego. Jest on budowany iteracyjnie, jako suma obszarów przesłanianych przez kolejne obiekty. O ile w algorytmie *Portal rendering* używa się podobnego schematu, w którym oblicza się część wspólną kolejnych *portali*, to w tym przypadku suma kolejnych obszarów zasłaniających w ogólności nie musi bryłą wypukłą. W szczególności, suma dwóch obszarów (niewypukłych) reprezentowanych przez n-kąt może być rzędu $\mathcal{O}(n^2)$, co może pociągać za sobą nieakceptowalny koszt przetwarzania tak dużych struktur danych.

Mimo tego pesymistycznego akcentu konkretne metody opierające się na zjawisku przesłaniania się obiektów okazują się być bardzo silne i są często stosowane w praktyce.



Rysunek 2.12: Przykład niefortunnego przypadku sumowania wielokątów o $m = 6$ i $n = 8$ zębach (składające się z odpowiednio 8 i 10 wierzchołków), dający w wyniku wielokąt składający się z $\mathcal{O}(m * n)$ wierzchołków (dokładnie: z $4 * m * n + 4 = 100$ wierzchołków).

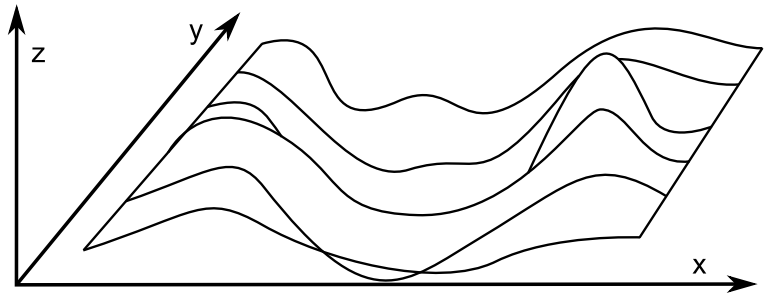
2.5.1. Floating horizon

Jest to popularny algorytm stosowany do wizualizacji matematycznych powierzchni opisanych jako funkcja dwóch zmiennych. W fotorealistycznej grafice 3D praktycznie nie występuje, można go odnaleźć popularnych programach jak arkusze kalkulacyjne lub programy do przygotowywania prezentacji jako narzędzie do wizualizacji zbiorów danych. W ogólności stosuje się go do rozwiązywania problemu widoczności dla przestrzeni 2.5 wymiarowej.

W najprostszej wersji algorytm pracuje z *precyzją obrazową* i wizualizuje funkcje

$$y = f(x, z)$$

jako wiele krzywych kreślonych dla równo oddalonych współrzędnych z. Stosuje się *rzut środkowy* bądź też bardziej tradycyjny *rzut równoległy*.



Rysunek 2.13: Przykład wizualizacji powierzchni funkcyjnej z użyciem algorytmu *floating horizon* w rzucie równoległym.

Ideą algorytmu jest rysowanie kolejnych krzywych jako łamanych, zaczynając od tych najbliższych (z najmniejszą wartością y) ku tym dalszym, każdą z nich rysuje się od lewej do prawej strony. Dodatkowo utrzymuje się dwa wektory punktów y_{min} i y_{max}, które wyznaczają obszar horyzontu. Dla danej łamanej, rysując i-tą jej część rozpiętą na punktach $(x_i, y_i), (x_{i+1}, y_{i+1})$ sprawdza się, jaka jest jej orientacja względem łamanej reprezentującej dolną $(x_i, y_{min}[x_i]), (x_{i+1}, y_{min}[x_{i+1}])$ i górną $(x_i, y_{max}[x_i]), (x_{i+1}, y_{max}[x_{i+1}])$ część horyzontu. Możliwe jest pięć przypadków:

1. **Przetwarzany segment jest w całości ponad górną linią horyzontu** W takim przypadku segment jest rysowany gdyż jest w całości widoczny ponad górną linią horyzontu. Po narysowaniu jej staje się ona nową górną linią horyzontu, uaktualniane są wartości w wektorze $y_{max}[x_i], \ldots, y_{max}[x_{i+1}]$.

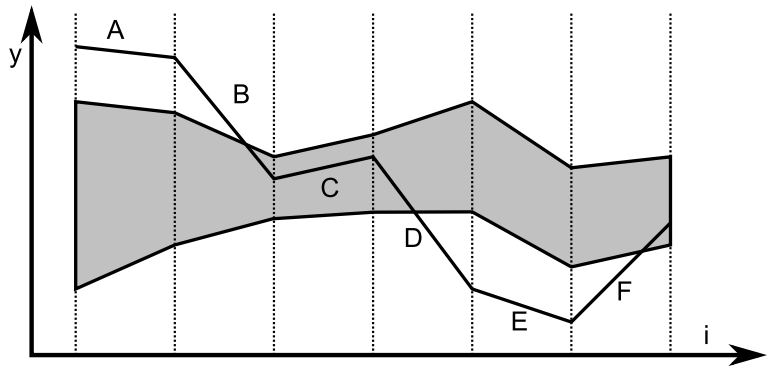
2. **Jeden z punktów segmentu jest ponad górną linią horyzontu, pozostały pomiędzy górną a dolną linią horyzontu** Oznacza to częściową

widoczność segmentu, rysowany jest fragment segmentu od punktu wystającego ponad linię górną horyzontu do punktu przecięcia się górnej linii horyzontu i przetwarzanego segmentu. Jednocześnie uaktualniana jest odpowiednia część tablicy y_{max}. Pozostała część segmentu jest niewidoczna i nie jest rysowana. Ze względu na dyskretny charakter linii horyzontu mogą występować pewne wizualne artefakty.

3. **Oba punkty są pomiędzy górnym a dolnym horyzontem** Segment jest całkowicie niewidoczny, nie są podejmowane żadne akcje.

4. **Jeden z punktów jest poniżej dolnej linii horyzontu, drugi pomiędzy górną a dolną linia horyzontu.** Sytuacja analogiczna do punktu drugiego. Rysowany jest fragment wystający poniżej dolny horyzont i uaktualniany jest odpowiedni fragment tablicy y_{min}

5. **Oba punkty są poniżej dolnej linii horyzontu.** Segment "wystaje„ z dołu i jest rysowany w całości, uaktualniane są odpowiadające mu wpisy w tablicy y_{min}



Rysunek 2.14: Szczegółowe przedstawienie rysowania łamanej, segmenty *A E* są w całości widoczne, gdyż leżą odpowiednio ponad górną i poniżej dolnej linii horyzontu, segmenty *B*,*D*,*F* jako przecinające linię horyzontu są częściowo widoczne, segment *C* jest całkowicie niewidoczny.

W publikacji [Gor02] szczegółowo opisano zastosowanie algorytmu *floating horizon* do wizualizacji powierzchni matematycznych wraz z z ulepszeniami zapobiegającymi wizualnym artefaktom w miejscach "chowania„ się horyzontu, kolorowania oraz interpolacji koloru pomiędzy sąsiednimi warstwami, co sprawia wrażenie obserwowania jednolitej powierzchni.

Istnieje wiele algorytmów wywodzących się z *floating horizon* służących do efektywnego wyświetlania terenów reprezentowanych jako regularna siatka zawierająca wysokości i kolory poszczególnych punktów. Praca [Sze97] szczegółowo porusza zagadnienia związane z tą tematyką. Zagadnienie renderowania

dużych, otwartych scenerii takich jak rozległe tereny jest specyficzne i różni się znacznie od przypadku scen opisujących scenerie zamknięte, takie jak obiekty architektoniczne. Dwie główne różnice to:

- Odsetek obiektów niewidocznych z powodu przesłonięcia przez te bliższe jest niski w porównaniu z wszystkimi obiektami znajdującymi się w bryle widzenia. Przesłanianie występuje, lecz nie należy do czynników mogących istotnie wpłynąć na ograniczenie ilości widocznych obiektów.
- Wraz ze wzrostem odległości od obserwatora gwałtownie rośnie ilość obiektów do wyświetlenia przy jednocześnie malejącej skali odwzorowania.

Sprawia to, że czynnik przesłaniania ma znaczenie drugorzędne. Bardziej istotne jest efektywne rysowanie obiektów dalszych, które widoczne są ze znacznie mniejszą szczegółowością. Popularnym podejściem jest *level of detail* (**LOD**), czyli techniki stopniowej degradacji złożoności rysowanych obiektów wraz z rosnącą odległością od obserwatora.

Modyfikacja algorytmu *floating horizon* dla potrzeb renderowania map wysokości jest niewielka. Pseudokod algorytmu przedstawia rysunek 2.15.

Niech $y[0 \dots x_{max} - 1]$ **reprezentuje bieżący horyzont**
Inicjuj wszystkie elementy wektora y **wartością** y_{max}
Dla kolejnych warstw i **począwszy od** 1 **do** n**-tej wykonaj**
    Dla kolejnych punktów obrazowych x **od** 0 **do** $x_{max} - 1$ **wykonaj**
        $t_x, t_y \leftarrow$ **współrzędne na mapie wysokości odpowiadające rzutowi** i **i** x
        $h \leftarrow$ **wysokość terenu o współrzędnych** $[t_x, t_y]$ **zrzutowana na współrzędne ekranowe**
        jeśli $h < y[x]$ **to**
            narysuj na ekranie prostą $(x, y[x] + 1), (x, h)$ **o kolorze pobranym z mapy kolorów**
            $y[x] \leftarrow h$

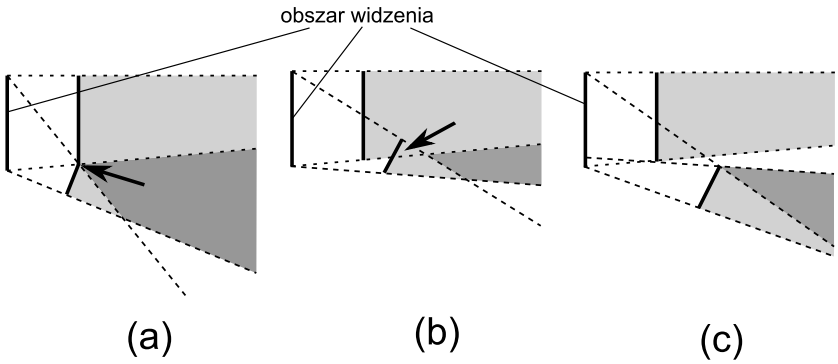
Rysunek 2.15: Pseudokod algorytmu *floating column*

Algorytm ten często nazywany jest *floating column* [Gor02] [Sze97], ze względu na charakterystyczną różnicę w stosunku do pierwowzoru: zamiast rysować linie reprezentujące kolejne ”przekroje„ płaszczyzny rysuje się pionowe słupki pomiędzy nimi. Efekty jego pracy są dobre a implementacja nie nastręcza trudności (możliwe jest nawet zrównoleglenie). Posiada kilka wad wartych wspomnienia: w dalszych odległościach może pojawić się problem *aliasingu*, w bliższych odległościach krawędzie nie są rozmywane dając ostre przejścia, nie toleruje pewnych specyficznych położeń kamery. Implementacja nie pasuje zbytnio do dzisiejszej filozofii systemów graficznych, w których istnieje wyraźny podział na jednostkę sterującą i procesor graficzny.

2.5.2. Occluder fusion

Rozważając widoczność *z obszaru* można zauważyć, że obszar zasłaniany przez pojedyncze obiekty (obszar cienia) jest stosunkowo niewielki. Dlatego istotne jest wzięcie pod uwagę zasłaniania spowodowanego przez grupę blisko siebie położonych obiektów. W przypadku widoczności *z punktu* całkowity obszar cienia jest równy sumie cieni rzucanych przez poszczególne obiekty, co jest łatwe do wyznaczenia. Dla widoczności *z obszaru* nie jest to prawdą, suma cieni poszczególnych obiektów jedynie zawiera się we właściwym obszarze cienia rzucanym przez tą grupę obiektów, który z kolei zależy od wzajemnego wpływu obszarów cienia i półcienia pojedynczych obiektów. Aproksymacja całkowitego obszaru cienia jako suma cieni pojedynczych obiektów jest konserwatywna, jednak nie jest wystarczająco dokładna, gdyż pomija bardzo istotny wkład wzajemnego wpływu przez dwa lub więcej obiektów, sprawiając że takie rozwiązanie jest w praktyce bezużyteczne. Możemy wyróżnić trzy typy interakcji pomiędzy obiektami i ich obszarami cienia i półcienia:

- Połączone obiekty.
- Nakładające się obszary cienia.
- Nakładające się obszary półcienia.



Rysunek 2.16: Trzy typy interakcji obszaru przesłanianego: **a)** Połączone obiekty. **b)** Nakładające się obszary cienia. **c)** Nakładające się obszary półcienia.

Fuzję okluderów stosuje się w wielu algorytmach do wyznaczania widoczności *z obszaru* bądź *z punktu*.

2.5.2.1. Hierarchical Z-Buffer

Algorytm opisany w pracy [GKM93] jako jeden z pierwszych jednocześnie wykorzystuje trzy rodzaje *spójności*: obiektowej, obrazowej i czasowej. Jego koncepcja okazała się trafiona, na jej podstawie powstało wiele podobnych algorytmów, znane są implementacje sprzętowe. Używa się w nim dwóch hierarchicznych struktur danych:

- **Drzewa ósemkowego**, które reprezentuje hierarchiczny podział przestrzeni 3D.
- **Z-piramidy**, której bazą jest "surowy„ *Z-bufor*, zaś kolejne poziomy reprezentują go z coraz mniejszą szczegółowością.

W procesie renderowania przegląda się jej *drzewo ósemkowe* zaczynając od głównego węzła. Ściany tworzące węzeł *drzewa ósemkowego* są rzutowane na przestrzeń ekranową. Wyznacza się ich konserwatywną aproksymację w postaci prostokąta który je obejmuje, o wartości Z równej najmniejszej wartości Z wziętej ze wszystkich ścian tworzących węzeł drzewa. Przy pomocy odpowiedniego poziomu *Z-piramidy* dokonuje się szybkiego sprawdzenia, czy prostokąt jest widoczny. Jeśli tak, rasteryzuje się wszystkie obiekty związane z nim[9] a następnie rekurencyjnie przetwarza się wewnętrzne węzły *drzewa ósemkowego* w kolejności od najbliższych do najdalszych.

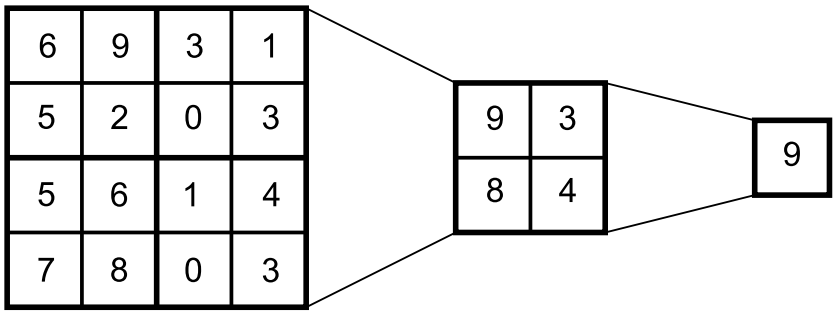
Spójność czasowa jest wykorzystywana poprzez użycie listy poprzednio renderowanych węzłów *drzewa ósemkowego*. Rysując bieżącą ramkę algorytm renderuje wszystkie węzły z tej listy bez żadnych testów zasłaniania. Ze względu na istniejącą spójność między sąsiednimi klatkami obrazu powoduje to narysowanie większości widocznych na ekranie obiektów, przy okazji tworząc dość dobrą zawartość początkową *Z piramidy*. Następnie algorytm przegląda w opisany wcześniej sposób *drzewo ósemkowe* sceny, pomijając te węzły, które zostały już wyrenderowane, gdyż znajdowały się na liście poprzednio rysowanych węzłów, zaś każdy nowo wyrenderowany węzeł dodawany jest do niej. Na koniec dla każdego węzła drzewa z listy dokonuje się test widoczności względem Z piramidy, usuwając te, które zostały uznane za niewidoczne.

Podsumowując, poszczególne typy spójności uzyskuje się na drodze:

- Hierarchicznego podziału sceny na *drzewo ósemkowe*, przez co wykorzystuje się *spójność w przestrzeni obiektowej*.
- Hierarchicznej reprezentacji *Z-bufora* na różnych poziomach szczegółowości, co pozwala wykorzystać *spójność w przestrzeni obrazowej*.

[9] Proces ten uaktualnia zawartość *Z-bufora.* i kolejne poziomy *Z-piramidy*

- Użyciu listy poprzednio widocznych węzłów *drzewa ósemkowego* w celu wykorzystania *spójności czasowej* sąsiednich klatek obrazu.

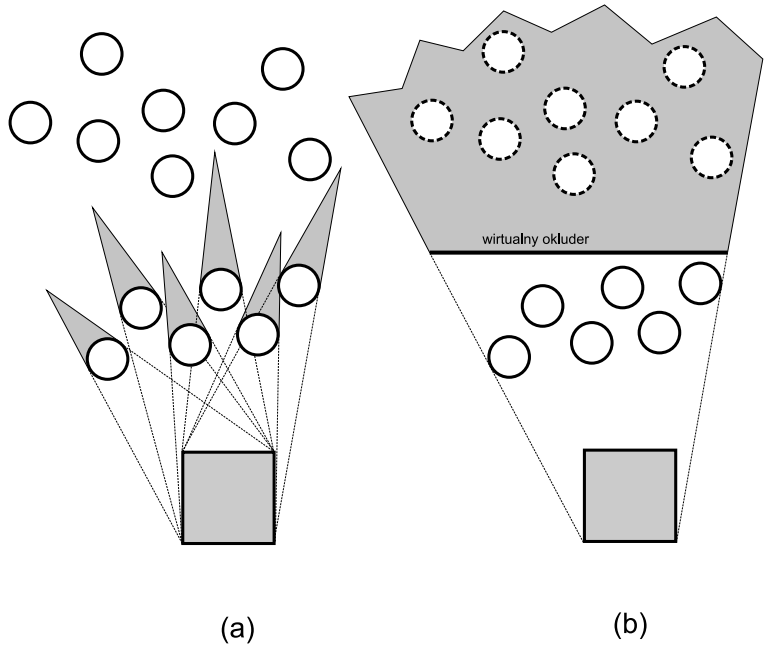


Rysunek 2.17: Hierarchiczny *Z-bufor* lub *Z-piramida*. Po lewej znajduje się *Z-bufor* rozmiaru 4×4, który reprezentuje głębokość na poziome poszczególnych punktów obrazu. Na kolejnych poziomach piramidy brana jest największa wartość z regionu 2×2. Ostatecznie na wierzchołku piramidy znajduje się tylko jedna wartość, reprezentująca maksimum z wszystkich wartości *Z-bufora*

.

2.5.2.2. Virtual occluders

W publikacji [KCCO00] zaprezentowano nowatorskie podejście do reprezentacji i obliczania widoczności *z obszaru*. Idea algorytmu opiera się na użyciu obiektów nazwanych przez autorów *wirtualnymi okluderami*. Dla danej sceny i obszaru widzenia (sektora), *wirtualny okluder* jest wypukłym wielokątem, dla którego gwarantuje się, że jest całkowicie niewidoczny z dowolnego punktu wewnątrz rozpatrywanego obszaru widzenia, oraz jest efektywnym okluderem, to znaczy że wnosi istotny wkład w redukcję obszaru widzenia. *Wirtualne okludery* są kompaktową reprezentacją informacji o wzajemnym przesłanianiu się obiektów z danego sektora. Z reguły każdy z nich reprezentuje połączone zasłanianie od wielu pojedynczych obiektów. Takie podejście umożliwia efektywne wyznaczenie zbioru potencjalnie widocznych obiektów w czasie renderowania i jest znacznie mniej kosztowne pamięciowo niż pamiętanie dla każdego sektora wyliczonych wcześniej *potencjalnie widocznych obiektów* z niego.



Rysunek 2.18: **a)** Suma obszarów cieni poszczególnych obiektów bywa mało znacząca. **b)** Zagregowany obszar cienia reprezentowany przez *wirtualny okluder* jest znacznie większy i istotny z punktu widzenia widoczności.

2.5.2.3. Occluder fusion by point sampling

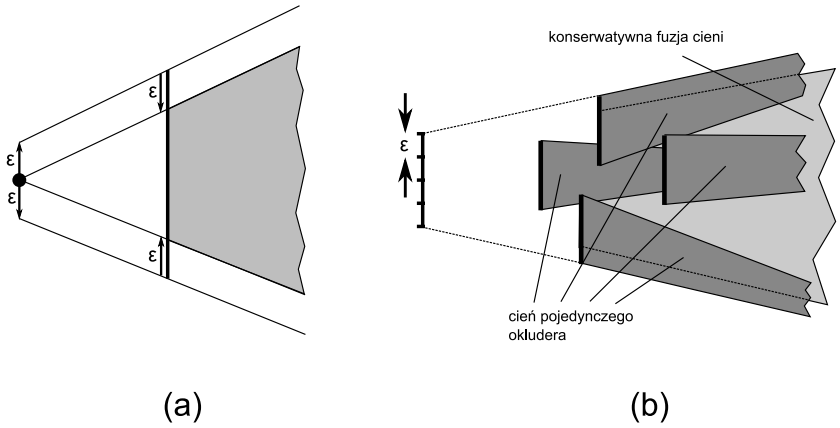
Nowatorskie podejście do problemu widoczności *z obszaru* przedstawiono w [WWS00]. Bazuje ono na obserwacji, że możliwa jest konserwatywna aproksymacja widoczności *z obszaru* z danych o widoczności z punktów rozmieszczonych na krawędziach rozpatrywanego obszaru. Warunkiem koniecznym aby obiekt był całkowicie niewidoczny z każdego punktu należącego do wnętrza obszaru jest to, aby był niewidoczny z każdego z punktu leżącego na obrzeżu obszaru.

Rozpatrując widoczność z punktu i ”kurcząc„ obszar cienia o ϵ otrzymuje się mniejszy obszar o interesującej własności: obiekt sklasyfikowany jako niewidoczny w pomniejszonym obszarze cienia pozostaje niewidoczny względem pierwotnego (nie pomniejszonego) obszaru cienia przy przesunięciu punktu widzenia o nie więcej niż ϵ.

W konsekwencji, użycie pomniejszonych o ϵ obszarów cienia z punktów na obrzeżach sektora, oddalonych od siebie o nie więcej niż ϵ daje prawidłową, konserwatywną aproksymację obszaru niewidocznego z całego sektora. Technika ta pozwala na stosunkowo prostą transformację problemu widoczności *z obszaru* na problem widoczności *z punktu*, dla którego znane są efektywne metody obliczeniowe. Możliwe są dwa scenariusze takiej transformacji:

- Jeśli dla każdego z punktów brzegowych dostępna jest lista *potencjalnie widocznych obiektów* względem pomniejszonego o ϵ obszaru cienia, to zbiór *potencjalnie widocznych obiektów* z sektora może być wyznaczony jako suma wszystkich zbiorów *PVS*. To podejście pozostawia dowolność w wyborze algorytmu wyznaczającego widoczność *z punktu*.
- Jeśli informacja o widoczności z punktów brzegowych jest dostępna w postaci obszarów cienia, to obszar niewidoczny z sektora można wyznaczyć jako część wspólną tych obszarów.

Autorzy opisali implementację systemu do interaktywnej prezentacji scen reprezentujących obszary miejskie, w której zastosowali podział sceny na sektory i wyznaczanie dla każdego z nich zbiorów *PVS* z użyciem opisywanej metody. Poczynione uproszczenie sceny do przestrzeni 2.5 wymiarowej zaowocowało efektywną, wspomaganą sprzętowo implementacją wyznaczania widoczności z punktów brzegowych sektorów wraz ich łączeniem do wynikowego obszaru określającego widoczność z całego sektora.



Rysunek 2.19: **a)** Rozpatrując obszar cienia z punktu pomniejszony o ϵ otrzymuje się dobrą, konserwatywną aproksymację oryginalnego obszaru cienia, ważną w sąsiedztwie o promieniu ϵ. **b)**. Fuzja obszarów cieni wyznaczona opisaną metodą tylko nieznacznie przeszacowuje rzeczywistą widoczność i jest jakościowo znacznie lepsza od obszarów niewidocznych pochodzących od poszczególnych obiektów.

Rozdział 3

Dekompozycja na portale i sektory

3.1. Wstęp

Dekompozycja trójwymiarowej sceny na *portale* i *sektory*, nazywana także *grafem CPG*, jako kompaktowa struktura danych kodująca informację o widoczności znajduje zastosowanie w wielu aspektach grafiki komputerowej i dziedzin z nią powiązanych, takich jak:

- Usuwanie niewidocznych powierzchni (Portal rendering, PVS rendering).
- Interakcje pomiędzy obiektami na scenie (wykrywanie kolizji, planowanie ścieżki ruchu).
- Przyśpieszanie obliczeń globalnego oświetlenia.

Narzędzia do modelowania scen 3D, których złożoność liczy się w dziesiątkach i setkach tysięcy prymitywów graficznych, najczęściej trójkątów, zazwyczaj dostarczają mechanizmy do ich hierarchicznego grupowania i kategoryzowania, aby uprościć, a czasami wręcz aby umożliwić człowiekowi pracę nad tak wielkim zbiorem danych. Typowy scenariusz pracy przy modelowaniu trójwymiarowej sceny polega na użyciu gotowych komponentów reprezentujących podstawowe elementy takie jak stoły, krzesła, drzwi, okna, schody, przejścia, ściany, łączeniu ich w większe grupy reprezentujące pomieszczenia bądź ich fragmenty i składaniu ich w większe byty aż do otrzymania finalnego dzieła.

W praktyce, struktura sceny 3D dostarczonej przez narzędzia edycyjne nie zawiera podziału na *portale i sektory*. W typowym przypadku dostępny jest tylko hierarchiczny podział na logiczne komponenty sceny. Efektywne wyświetlanie takich danych wymaga dodatkowej reprezentacji kodującej informację o

widoczność, którą jest na przykład *graf CPG*. Umiejętność automatycznej dekompozycji na *portale i sektory* jest więc istotnym elementem systemów do obrazowania grafiki komputerowej.

3.2. Drzewo BSP jako dekompozycja na portale i sektory

Drzewo *BSP*, dokładniej opisane w 2.2.1 jest jedną z popularnych struktur wprowadzających porządek w danych będących zbiorem trójwymiarowych obiektów. Występuje w wielu algorytmach dotyczących grafiki 3D, takich jak:

- W algorytmie *BSP-rendering*, służąc do szybkiego sortowania wielokątów w porządku od najdalszych do najbliższych względem obserwatora, służąc w ten sposób do usuwania niewidocznych powierzchni na zasadzie algorytmu malarza.
- W algorytmach wykrywających kolizję z elementami sceny, bądź wyznaczającymi trasę jako reprezentacja pozwalająca na ich efektywną implementację.
- W celu szybkiego znajdowania przecięć promienia z elementami sceny, na przykład na potrzeby *śledzenia promieni*
- W szerokiej klasie algorytmów propagacji widoczności między punktami bądź obszarami.

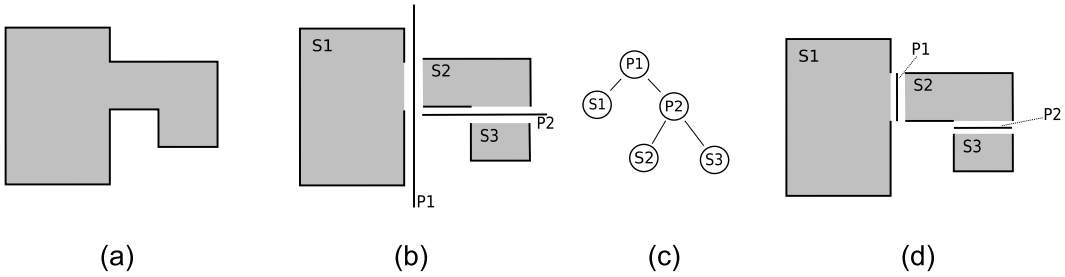
W praktyce bardzo często korzysta się z tej uniwersalności *drzewa BSP*, gdyż w systemach do obrazowania scen 3D równolegle występuje szereg kwestii związanych z wzajemną widocznością i kolizjami.

W dalszej części przyjmuje się, że prymitywy graficzne będące elementami sceny są płaskimi trójkątami o wyróżnionej, widocznej stronie. Taka właśnie reprezentacja jest najczęściej stosowana w procesie wyświetlania sceny.

"Spłaszczając„ hierarchię sceny do listy trójkątów, budując *drzewo BSP* na tym zbiorze danych, przy założeniu, że prymitywy umieszczone są tylko w liściach drzewa, można dostrzec w nim poszukiwaną dekompozycję na portale i sektory, gdzie:

- Sektorami są liście drzewa BSP.
- Portalami są płaszczyzny podziału przycięte do granic sektorów.

Jeśli jest to wymagane, precyzujemy pojęcie *portali i sektorów*, mówiąc o *BSP-portalach* i *BSP-sektorach* ze względu na swoje pochodzenie.

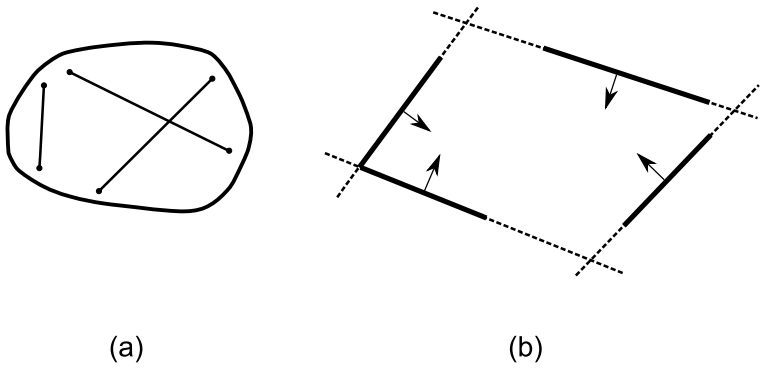


Rysunek 3.1: **(a)** Przykładowa scena 2D. **(b)** Skonstruowane *drzewo BSP* na jej podstawie. **(c)** Jego reprezentacja. **(d)** Przycięte płaszczyzny podziału do granic sektorów tworzące *BSP-portale* łączące *BSP-sektory*.

3.2.1. Wypukły zbiór trójkątów

Ważnym pojęciem związanym z rozpatrywanym zagadnieniem jest **wypukły zbiór trójkątów**. Klasyczna definicja **zbioru wypukłego** mówi o tym, że każdy odcinek, którego końce należą do tego zbioru, w całości się w nim zawiera. Obrazuje to wzajemną widoczność dowolnych dwóch punktów należących do wnętrza *zbioru wypukłego*.

Wypukły zbiór trójkątów posiada właściwość, że każde dwa punkty należące do dowolnych dwóch trójkątów są nawzajem widoczne. We wzajemnej widoczności istotny jest fakt jednostronności powierzchni: punkt jest widoczny wtedy, gdy normalna do powierzchni trójkąta w tym punkcie jest skierowana w kierunku obserwatora.



Rysunek 3.2: **(a)** Wypukły zbiór w klasycznym ujęciu. Odcinek łączący każde dwa punkty należy do zbioru. **(b)** Wypukły zbiór odcinków, każdy punkt odcinka leży po pozytywnej stronie hiperpłaszczyzny każdego innego odcinka.

Opisywana własność jest równoważna temu, że dla każdego trójkąta *wypukłego zbioru* wszystkie jego wierzchołki są położone po pozytywnej stronie

płaszczyzny każdego innego trójkąta z tego zbioru. Jest to łatwe do sprawdzenia poniższym algorytmem:

procedura zbiór-wypukły(lista trójkątów S)
  dla każdego trójkąta $T1$ **należącego do** S **powtarzaj**
    dla każdego trójkąta $T2$ **należącego do** S **powtarzaj**
      jeśli $T1 \neq T2$ **to**
        $P \leftarrow$ płaszczyzna w której zawiera się $T1$
        jeśli którykolwiek z wierzchołków $T1$ leży po negatywnej stronie P **to**
          zwróć zbiór-nie-jest-wypukły
zwróć zbiór-jest-wypukły

3.2.2. Struktura drzewa BSP

Na drzewo BSP, którego ogólną strukturę opisaną w 2.2.1 i 2.2.2 musi zostać nałożonych kilka warunków, w celu poprawnej reprezentacji podziału na *portale i sektory*.

- *Drzewo BSP* budowane jest na podstawie sceny reprezentowanej przez zbiór trójkątów o jednostronnej powierzchni. Trójkąt jest widoczny tylko wtedy, gdy jego normalna jest skierowana w kierunku obserwatora.
- Węzeł wewnętrzny drzewa, reprezentujący pewną podprzestrzeń zawiera płaszczyznę dzielącą ją na dwie niepuste podprzestrzenie oraz odniesienie (wskaźniki) na węzły drzewa reprezentujące te podprzestrzenie.

  W przypadku, gdy jakiś trójkąt zawiera się w płaszczyźnie podziału klasyfikacja do odpowiedniej podprzestrzeni opiera się na porównaniu orientacji płaszczyzny podziału względem normalnej do powierzchni trójkąta. W przypadku zgodności kierunków, trójkąt klasyfikowany jest do podprzestrzeni znajdującej się po pozytywnej stronie płaszczyzny podziału, w przeciwnym przypadku trafia do podprzestrzeni leżącej po jej negatywnej stronie.
- Liście drzewa zawierają listę trójkątów tworzącą zbiór wypukły.
- Każdy węzeł drzewa posiada odniesienie (wskaźnik) na swojego ojca. W przypadku głównego węzła drzewa parametr ten wskazuje na pusty obiekt[1].
- Każdy węzeł drzewa posiada wymiary prostopadłościennej bryły, która opisuje wszystkie zgromadzone w liściach trójkąty.

[1]Wartość **null**, z dokładnością do zapisu dostępnego w językach programowania.

3.2.3. Konstrukcja drzewa BSP

Drzewo BSP rekurencyjnie dzieli przestrzeń na dwa fragmenty przy pomocy płaszczyzn. W innej strukturze hierarchicznego podziału przestrzeni 3D, jaką jest *drzewo ósemkowe*, w każdym kroku dokonuje się deterministycznego podziału trzema ortogonalnymi, zgodnymi z osiami układu współrzędnych płaszczyznami na osiem podprzestrzeni. *Drzewo BSP* charakteryzuje się więc znacznie większą elastycznością, pozwalając na całkowitą dowolność wyboru sposobu podziału.

Paradoksalnie czynnik ten jest dość kłopotliwy ze względu na fakt, że dla każdego węzła istnieje nieskończenie wiele możliwych płaszczyzn podziału. Popularnym podejściem stało się ograniczenie tego zbioru do płaszczyzn zawierających się w trójkątach sceny [Abr97], [NAT90], [Nay92]. W dalszej części, jeśli nie wspomniano inaczej, zakłada się taki właśnie zbiór.

3.2.3.1. Optymalne drzewo BSP

Za optymalne *drzewo BSP* uważa się takie, które ma najmniejszą możliwą sumaryczną liczbę trójkątów [dBCvKO97], [JD97], [FAG83]. Jest to tożsame stwierdzeniu, że w procesie jego konstrukcji dokonano najmniejszej liczby podziałów trójkątów, choć nie oznacza to, że drzewo posiada najmniejszą możliwą liczbę węzłów.

Minimalizacja ilości trójkątów jest wysoce zasadna. Zmniejsza to pamięć użytą do ich przechowywania, co ma niebagatelne znaczenie wobec faktu, że z reguły składowane są w zasobach pamięciowych kart graficznych, które są ograniczone, kłopotliwe w zarządzaniu i z reguły wielokrotnie mniejsze niż pamięć operacyjna komputera.

Poszukiwanie optymalnego drzewa nawet przy restrykcji zbioru płaszczyzn podziału do płaszczyzn zawierających się w wielokątach sceny jest w praktyce niewykonalne. Nie są znane inne metody niż konstrukcja wszystkich możliwych drzew i wybranie tego o najmniejszej liczbie podziałów [Win99]. Pseudokod algorytmu realizującego ten proces wygląda następująco:

```
procedura optymalne-drzewo(lista trójkątów S)
    jeśli |S| = 1
        zwróć liść(S)
    najlepszedrzewo ← null
    najlepszyrozmiar ← ∞
    dla każdego trójkąta t ∈ S wykonaj
        S1 ← wybierz-przednie-trojkaty( t, S)
        S2 ← wybierz-tylne-trojkaty( t, S)
        drzewo ← połacz-poddrzewa(optymalne-drzewo(S1),t,optymalne-drzewo(S2))
        jeśli rozmiar(drzewo) < najlepszyrozmiar
            najlepszedrzewo ← drzewo
            najlepszyrozmiar ← rozmiar(drzewo)
    zwróć najlepszedrzewo
```

zaś jego złożoność równa $\mathcal{O}(n!)$, gdzie n jest liczbą trójkątów czyni go całkowicie niepraktycznym.

3.2.3.2. Zachłanny algorytm konstrukcji drzewa BSP

W publikacji [FKN80] zaproponowany został rekurencyjny, zachłanny algorytm konstrukcji *drzewa BSP*. W każdym kroku rekurencji w którym budowane jest poddrzewo, z zadanej listy trójkątów wybiera się najlepszą płaszczyznę podziału. Podstawowym kryterium jest minimalizacja liczby podziałów trójkątów, pobocznym - zbalansowanie drzewa pod postacią dążenie do równolicznych zbiorów trójkątów leżących po pozytywnej i po negatywnej stronie wybranej płaszczyzny.

Bardzo trudno ocenić jest rezultaty działania tej metody, jako że nieznana jest wielkość optymalnego *drzewa BSP*. W praktyce jest to najczęściej stosowana metoda w komercyjnych produktach [Abr97].

Łatwo zrozumieć ideę stojącą za tym algorytmem. Opiera się ona na przypuszczeniu, że minimalizacja podziałów trójkątów wynikłych z podziału bieżącej podprzestrzeni redukuje całkowitą liczbę podziałów niezbędnych do zbudowania całego drzewa. Istnieje nadzieja, że otrzymane drzewo będzie optymalne. Poniżej przedstawiono pseudokod algorytmu realizującego to zadanie:

procedura drzewo-bsp(lista trójkątów S)
  jeśli $|S| = 1$
    zwróć liść(S)
  $plaszczyznaPodzialu \leftarrow$ **null**
  $najlepszyPodzial \leftarrow \infty$
  dla każdego trójkąta $t \in S$ **wykonaj**
    $kp \leftarrow$ policz-przednie-trojkaty(t, S)
    $kt \leftarrow$ policz-tylne-trojkaty(t, S)
    $ks \leftarrow$ policz-podziały-trojkatow(t, S)
    $koszt \leftarrow$ oblicz-koszt-podzialu(kp, kt, ks)
    jeśli $koszt < najlepszyPodzial$
      $najlepszyPodzial \leftarrow koszt$
      $plaszczyznaPodzialu \leftarrow t$
  $S1 \leftarrow$ wybierz-przednie-trojkaty($plaszczyznaPodzialu$, S)
  $S2 \leftarrow$ wybierz-tylne-trojkaty($plaszczyznaPodzialu$, S)
  zwróć połacz-poddrzewa(drzewo-bsp($S1$),$plaszczyznaPodzialu$,drzewo-bsp($S2$))

W idealnych warunkach (brak podziałów trójkątów, idealnie zbalansowane drzewo) złożoność algorytmu jest wielomianowa. W każdym wywołaniu rekurencyjnym n-razy wykonuje się zewnętrzną pętlę, której wykonanie zajmuje, ze względu na konieczność przejrzenia wszystkich trójkątów $\mathcal{O}(n)$, co sumarycznie daje czas $\mathcal{O}(n^2)$. Zakładając, że zbiory trójkątów przednich i tylnych ($S1$i $S2$) są równoliczne i brak jest podziałów trójkątów całkowity czas wykonania algorytmu dla n trójkątów wyraża się rekurencyjną zależnością:

$$T(n) = 2 * T(n/2) + \mathcal{O}(n^2) \tag{3.1}$$

której rozwiązanie ogólne wyraża się wzorem:

$$T(n) = \mathcal{O}(n^2) \tag{3.2}$$

W rzeczywistych warunkach podane oszacowanie sprawdza się, gdyż całkowita liczba podziałów trójkątów jest mała (w średnim przypadku sumaryczna liczba trójkątów wynosi 1.5 * początkowa liczba trójkątów), a drzewo jest dobrze zbalansowane. Niestety przetwarzanie sceny składającej się z dziesiątek tysięcy trójkątów, co obecnie jest wartością stosunkowo niską, trwa na współczesnym komputerze domowym kilkanaście minut. Czas ten jest do zaakceptowania, szczególnie że rezultaty tego procesu można zapisać, by później mieć do nich błyskawiczny dostęp, oczywiście pod warunkiem, że nie dopuszcza się zmian w scenie. Jednakże zgodnie z formułą 3.2 przetwarzanie zbioru miliona trójkątów, co wciąż nie jest wygórowaną liczbą potrwa 10000 razy dłużej, będąc już czasem całkowicie nieakceptowalnym.

W publikacji [FAG83] autorzy zmierzyli się z opisywanym powyżej problemem. Zastosowana została modyfikacja polegająca na limitowaniu liczby płaszczyzn podziału w procesie wyboru tej najlepszej do pewnego z góry ustalonego poziomu. Ogranicza to liczbę iteracji zewnętrznej pętli algorytmu 3.2.3.2 do stałej wartości, co przyczynia się do liniowej złożoności algorytmu (aczkolwiek z dużą stałą). Stosując losowy wybór trójkątów uczestniczących w wyborze najlepszej płaszczyzny podziału i limitując wielkość tego zbioru do kilkunastu elementów autorzy uzyskali akceptowalne rezultaty. Obecnie metoda ta, wzbogacona o bardziej inteligentne heurystyki tworzenia zbioru płaszczyzn niż zaproponowane przez autorów publikacji jest w powszechnym użyciu i uznawana jest za wystarczającą.

Istnieją takie zastosowania *drzewa BSP*, gdzie zagadnienia jego optymalności i budowy postawione są w innym świetle. Przykładem jest *Ray-tracing*, gdzie struktura *drzewa BSP* używana jest do przyśpieszania znajdowania przecięć promieni z obiektami sceny. W tym zastosowaniu, w pierwszej kolejności kładzie się nacisk na szybkie trawersowanie drzewa, w drugiej zaś zwraca się uwagę na szybkość budowy drzewa. W pewnych przypadkach, na przykład z powodu animacji pewnych jej części, czas budowy drzewa staje się istotny. Ponieważ szybkość trawersowania drzewa zależy przede wszystkim od ułożenia płaszczyzn rozdzielających kolejne poziomy drzewa, zagadnienie minimalizacji ilości podziałów trójkątów w procesie jego budowy spada na dalszy plan. Często też, zamiast podziału trójkątów przechodzących przez płaszczyznę podziału przydziela się je do obydwu poddrzew [Hav00, str. 50].

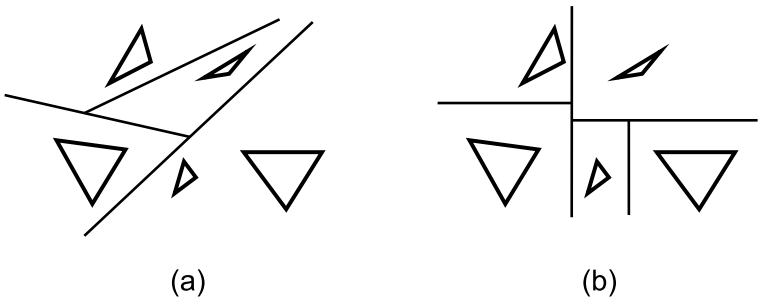
Badania na tym polu doprowadziły do wyewoluowania specyficznej formy *drzewa BSP*, w której płaszczyzny podziału są prostopadłe do osi układu współrzędnych. Taka jego forma nazywana jest **KD drzewem**[2], które pierwotnie zostało zaprojektowana do przechowywania danych punktowych [Ben75]. Jest to specjalizowana struktura danych i jej znaczenie w zakresie podziału przestrzeni dla algorytmów *PVS rendering* i *Portal rendering* jest ograniczone [MBWW07].

Cechę prostopadłości płaszczyzn podziału do osi układu współrzędnych wykorzystuje się w kilku aspektach:

- W celu przyśpieszenia poszukiwania przecięcia promienia z obiektem, podstawowe operacja wykonywana w procesie trawersowania drzewa, czyli obliczenie odległości ze znakiem punktu od płaszczyzny znacznie się upraszczają w przypadku, gdy płaszczyzna jest prostopadła do którejś z osi układu współrzędnych. Ma to znaczący wpływ na szybkość trawersowania *KD drzewa* [Hav00, str. 12, 51]

- W celu szybkiego wyznaczenia płaszczyzny podziału w procesie budowy

[2]Spotyka się też określenia prostoliniowe bądź ortogonalne drzewo BSP



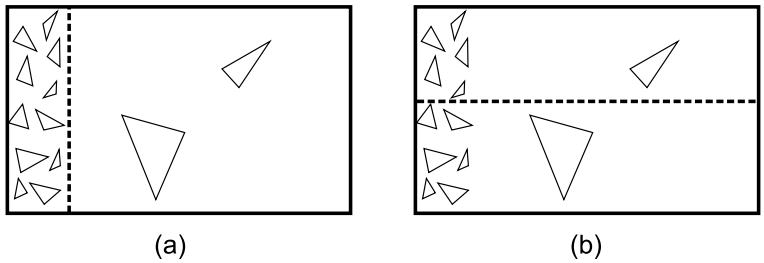
Rysunek 3.3: **(a)** Zbiór danych i klasyczne drzewo BSP. Orientacja hiperpłaszczyzn podziału jest dowolna. **(b)** Ten sam zbiór, podzielony przy pomocy KD drzewa. Hiperpłasczyzny podziału są równoległe do osi X lub Y układu współrzędnych.

KD drzewa. W procesie konstrukcji węzła drzewa możliwe jest wykorzystanie spójności przestrzennej płaszczyzn kandydujących do płaszczyzny podziału w celu wyboru tej najlepszej. [Hav00, str. 52] Rozpatrując dany wymiar, sortuje się wszystkie trójkąty po ich najmniejszej współrzędnej w tym wymiarze. Każdy z nich reprezentuje możliwą płaszczyznę podziału o normalnej zgodnej z rozpatrywanym wymiarem, zawierającej wierzchołek trójkąta o najmniejszej współrzędnej. Przechodząc tą posortowaną listę od początku do końca można w każdym kroku wyznaczać kolejną płaszczyznę podziału, oraz liczbę trójkątów będących przed nią, za nią i przecinających ją. Pozwala to wybrać najlepszą płaszczyznę w czasie $\mathcal{O}(n * log(n))$ (sortowanie) + $\mathcal{O}(n)$ (przejście posortowanej listy). Często zakłada się wybór płaszczyzny, która dzieli zbiór trójkątów na równoliczne zbiory trójkątów leżących przed nią i za nią. W takim przypadku w miejscu sortowania należy wybrać medianę zbioru, co jednocześnie kończy procedurę szukania płaszczyzny podziału.

- Wprowadzając pojęcie *modelu kosztu*, estymującego złożoność późniejszego trawersowania drzewa umożliwia efektywną budowę takiego, które minimalizuje czas jego trawersowania w pewnym przypadku uznanym za średni [GS87] [HHS06] [Hav00, str. 53-61]. W modelu **SAH** (ang. Surface Area Heuristic), dla wskazanej płaszczyzny podziału wyznacza się kosz trawersowania poddrzew przez nią stworzonych, następnie sumuje się te koszty przemnożone przez prawdopodobieństwa, że promień trafi do jednego z nich. Wybiera się taką płaszczyznę podziału, dla której obliczona wartość jest najmniejsza.

  Łatwo wskazać ideę stojącą za tym modelem. Jest nią analogia do *kodów Huffmana*, której celem jest przypisywanie krótszych kodów (krótszych

ścieżek w *drzewie Huffmana*) do częściej dostępowanych danych. Zakładając równomierną dystrybucję promieni, to rozważając wskazany węzeł drzewa prawdopodobieństwo trawersowania jednego z jego poddrzew jest równe stosunkowi objętości podprzestrzeni obydwu poddrzew. Metoda *SAH* wykorzystuje to, faworyzując podziały na podprzestrzenie, z których jedna jest bardzo duża, zawierająca niewiele trójkątów, druga zaś bardzo mała, lecz skupiająca znacznie większą ilość trójkątów. Pozornie stoi to w sprzeczności z heurystyką budowy jak najbardziej zrównoważonego drzewa, w praktyce jest bardzo trafionym pomysłem.



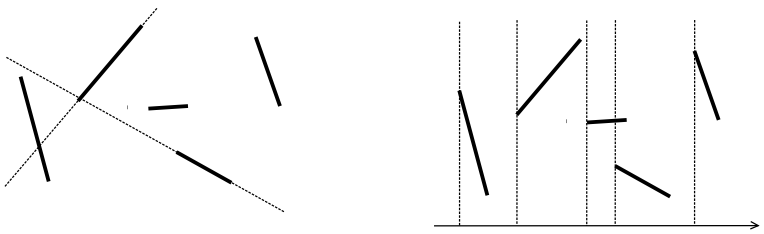
Rysunek 3.4: Wybór płaszczyzny podziału. **(a)** Z punktu widzenia heurystyki *SAH* opłacalny jest podział na małą celę zawierające dużo obiektów i dużą zawierające ich znacznie mniej. **(b)** Wybór płaszczyzny podziału zgodnie z klasycznym schematem, dążącym do podziału na równoliczne zbiory i cele o podobnym rozmiarze.

W celu budowy *drzewa BSP*, na podstawie którego dokonany zostanie podział na *portale i sektory* wybrany został algorytm zachłanny. Poniżej szczegółowo omówione są zagadnienia związane z tym algorytmem.

3.2.3.3. Kryterium zakończenia rekursji

Algorytmy 3.2.3.1 oraz 3.2.3.2 kończą rekurencyjny ciąg wywołań, gdy przetwarzany zbiór stanie się jednoelementowy. Odpowiadają za to dwie pierwsze linie, które sprawdzają liczność zbioru trójkątów, zakańczając w takim przypadku rekursję poprzez utworzenie i zwrócenie węzła drzewa-liścia, zawierającego jeden trójkąt.

W przypadku *drzewa BSP* tworzonego w celu ekstrakcji z niego dekompozycji na *portale i sektory*, kryterium zakończenia rekursji jest otrzymanie *zbioru wypukłego*, który jest dobrym, naturalnym kandydatem na sektor. Jest to pod pewnym względem niezbędne, gdyż stosując domyślny zbiór kandydatów na płaszczyzny podziału w postaci płaszczyzn zawierające się w trójkątach, dla wypukłego zbioru trójkątów którąkolwiek by płaszczyznę nie wybrać do podziału uzyska się zbiór trójkątów leżących na tej płaszczyźnie (najczęściej jed-



Rysunek 3.5: **(a)** Dowolna orientacja płaszczyzn rozdzielających w drzewie BSP wymusza konieczność każdorazowego kategoryzowania obiektów pod kątem położenia względem płaszczyzny podziału w celu zliczenia liczby obiektów leżących przed i za nią. **(b)** Przeglądając płaszczyzny podziału w kierunku rosnących współrzędnych możliwe jest wyznaczenie ilości obiektów leżących przed i za płaszczyzną w czasie stałym, poprzez aktualizację liczników odpowiadających za poprzednią płaszczyznę.

noelementowy) i zbiór gromadzący pozostałe trójkąty. W najlepszym przypadku otrzyma się poddrzewo zdegenerowane do listy, w najgorszym zaś, przy nieumiejętnie zaimplementowanej procedurze budowy drzewa - wejście w nieskończoną rekurencję.

Zbiór trójkątów jest wypukły, jeśli dla każdego trójkąta wszystkie inne trójkąty leżą w całości po pozytywnej stronie płaszczyzny zawierającej się w nim. Aby obalić wypukłość danego zbioru wystarczy wskazać w nim pewien trójkąt oraz drugi trójkąt wraz z punktem do niego należącym takim, że punkt ten leży po negatywnej stronie płaszczyzny zawierającej się w pierwszym trójkącie. Poniżej przedstawiono w postaci pseudokodu algorytm realizujący tą ideę:

procedura jest-zbiorem-wypukłym(lista trójkątów S)
  dla każdego $t_1 \in S$ **wykonaj**
    dla każdego $t_2 \in S$ **wykonaj**
      jeśli $t_1 \neq t_2$
        jeśli istnieje punkt $p \in t_1$ taki że p jest po negatywnej stronie t_2
          zwróć zbiór-nie-jest-wypukły
  zwróć zbiór-jest-wypukły

Szukając punktu należącego do jednego z trójkątów leżącego po negatywnej stronie płaszczyzny zawierającej się w drugim trójkącie wystarczy ograniczyć się do sprawdzenia wierzchołków trójkąta. Ze względu na dwie zagnieżdżone w sobie pętle, przebiegające po wszystkich elementach zbioru algorytm działa w czasie $\mathcal{O}(n^2)$. W przypadku, gdy zbiór nie jest wypukły algorytm może skończyć działanie znacznie szybciej.

Tak zmieniony warunek zakończenia rekursji w procedurze budowy drzewa w oczywisty sposób wpływa na poprzednio opisane rozważania dotyczące budowy drzewa. Wszystkie zmiany sprowadzają się do następujących wniosków:

- Dopuszczenie liści drzew zawierających wiele trójkątów jako zbiorów wypukłych wydatnie wpływa na zmniejszenie sumarycznej ilości węzłów w drzewie.
- Kwadratowy czas sprawdzania wypukłości zbioru trójkątów może negatywnie wpływa na złożoność czasową procedury budowy drzewa. Jednakże, w przypadku rzeczywistych scen rzadko spotykane są wypukłe zbiory trójkątów zawierające więcej niż 100 elementów.

3.2.3.4. Generowanie płaszczyzn rozdzielających

Jak dotąd przedstawiono dwojaki sposób generowania płaszczyzn-kandydatów dzielących podprzestrzenie *drzewa BSP*:

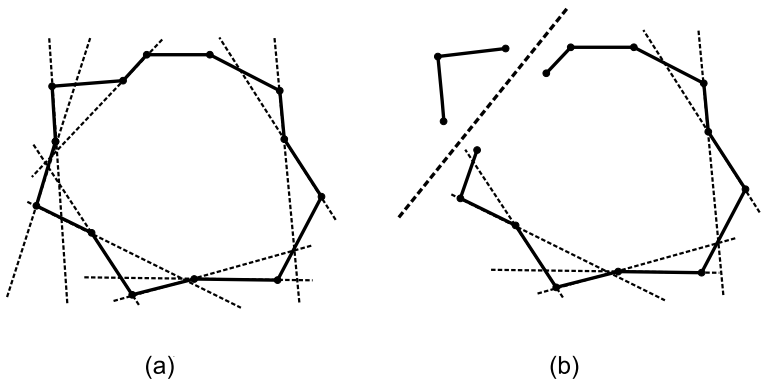
- Płaszczyzny prostopadłe do osi układu współrzędnych. Stosowane głównie w konstrukcji *KD drzew*
- Płaszczyzny zawierające się w trójkątach sceny.

Pierwsze z rozwiązań ma ograniczone znaczenie w zagadnieniach widoczności innych niż *śledzenie promieni* [Hav00]. Drugie z nich, zaproponowane w [FKN80], jest powszechnie uznawane jako wystarczające do generowania płaszczyzn podziału[3]. W praktyce okazuje się nie być wystarczająco dobre.

Podstawową jej wadą jest nieoptymalne działanie przy próbie znalezienia płaszczyzny podziału dla zbioru ”prawie„ wypukłego. W takim przypadku każda z potencjalnych płaszczyzn podziału powoduje albo niezrównoważony podział, albo niepotrzebne podziały trójkątów (najczęściej jedno i drugie). W kolejnych, rekurencyjnych podziałach sytuacja ulega tylko nieznacznej poprawie. Sumarycznie prowadzi to do stworzenia drzewa o nierównym zbalansowaniu, a także zwiększeniu całkowitej liczby trójkątów z powodu koniecznych podziałów trójkątów. Obrazowo przedstawia to rysunek 3.6. Opisywana sytuacja nie jest abstrakcyjna, rzeczywiste fragmenty scen, na przykład w postaci pomieszczeń, są często ”prawie„ wypukłe. Być może bardziej istotnym czynnikiem jest sposób, w jaki tworzone są sceny: graficy bardzo często popełniają pozornie nieistotne błędy takie jak: ściany złożone z trójkątów leżących na ”prawie„ tej samej płaszczyźnie, niedokładne przyleganie trójkątów, łączenie

[3]W tym miejscu można by zacytować wszystkie znane przez autora prace, poruszające problem budowy *drzewa BSP*.

ścian "na zakładkę„ [4], dopuszczanie otworów bądź nieciągłości powierzchni, które są mało widoczne, bądź niewidoczne ze względu na ograniczony obszar poruszania się obserwatora.



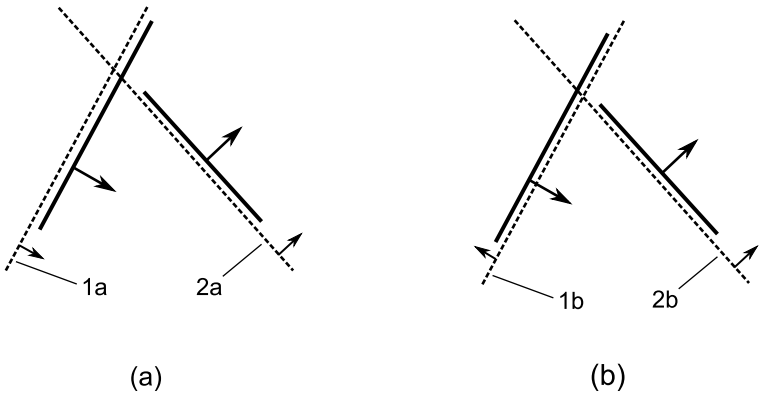
Rysunek 3.6: Zbiór "prawie„ wypukły i próba jego podziału przy pomocy płaszczyzn zawierających się w odcinkach tego zbioru. **(a)** Żaden z zaznaczonych podziałów nie gwarantuje otrzymania zrównoważonych ilościowo zbiorów. **(b)** Wybierając podział dający najlepszy stosunek liczności zbiorów, w rekurencyjnych podziałach problem nie ulega większej poprawie.

Poprzednio założono, że powierzchnia trójkąta widoczna jest tylko z jednej strony. Gdy trójkąt zawiera się w płaszczyźnie podziału kwalifikacja do podprzestrzeni znajdującej się po pozytywnej bądź negatywnej stronie zależy od tego, czy wektory normalne trójkąta i płaszczyzny podziału są zgodne czy przeciwne. Dlatego dla każdej płaszczyzny kandydującej do płaszczyzny podziału należy także sprawdzić jej wersję o przeciwnym wektorze normalnym, gdyż w przeciwnym wypadku, nawet w najprostszych sytuacjach, gdy rozpatrywany zbiór trójkątów jest dwuelementowy może dojść do zbędnych podziałów (rysunek 3.7).

Warto nadmienić o problemach związanych z niedokładnościami obliczeń[5]. Bardzo często spotyka się zbiory trójkątów leżące na jednej płaszczyźnie. Klasyfikacja odległych trójkątów leżących na wspólnej płaszczyźnie może być kłopotliwa, ze względu na to, że odbywa się ona na podstawie pomiaru odległości wierzchołków trójkąta od tej płaszczyzny. W teorii odległość ta jest równa zeru dla wszystkich takich punktów, w praktyce zaś oscyluje wokół zera. Wymaga to stosowania tolerancji we wszelkich porównaniach obliczonych wartości numerycznych i jej starannym doborze, nierzadko dokonywanym indywidualnie dla każdego przetwarzanego modelu.

[4]Prostopadłe do siebie ściany nie mają wspólnej krawędzi, lecz przecinają się nieznacznie po niewidocznej stronie modelu.

[5]Zagadnienie to zostanie szerzej omówione w dalszej części.



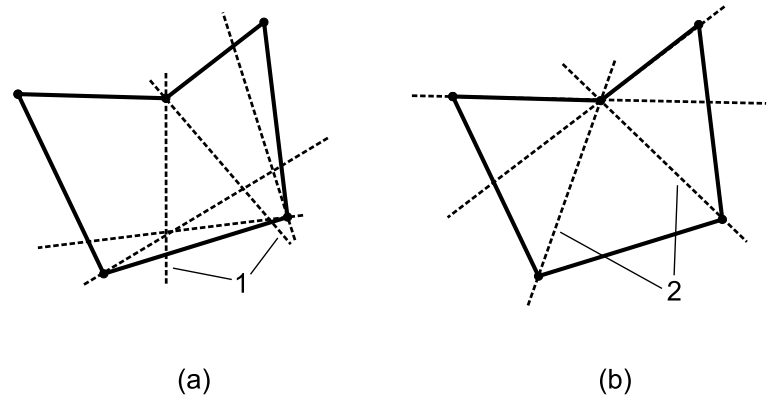
Rysunek 3.7: **(a)** Badając płaszczyzny o orientacji wektora normalnego zgodnej z płaszczyzną trójkąta w przypadku *1a* otrzymuje się podział na zbiór pusty i dwuelementowy zbiór będący zbiorem wejściowym, bądź *1b* podział jednego z trójkątów. **(b)** Wybór płaszczyzny *1b* o orientacji przeciwnej do orientacji płaszczyzny trójkąta tworzy dwa, jednoelementowe, wypukłe zbiory trójkątów. W obydwu przypadkach odsunięto nieco płaszczyzny podziału od trójkątów z których powstały, w celu ilustracji sposobu, w jaki trójkąty zawierające się w płaszczyźnie podziału są kategoryzowane.

Na samym koniec można wspomnieć, że ze względu na specyfikę ekstrakcji *BSP portali* z *drzewa BSP* niewskazane jest, jeśli płaszczyzny podziału zawierają jakieś trójkąty ze sceny. Dlatego poza płaszczyznami zawierającymi się w trójkątach sceny dodatkowo proponuje się sprawdzać następujące zbiory płaszczyzn:

- Dla każdego trójkąta płaszczyzny zawierające się w każdej z jego krawędzi, ortogonalne do płaszczyzny na której leży rozpatrywany trójkąt.
- Gdy rozmiar zbioru trójkątów jest mniejszy od pewnego ustalonego z góry rozmiaru, płaszczyzny rozpięte na wszystkich trójelementowych podzbiorach zbioru stworzonego z wszystkich wierzchołków trójkątów.

Pierwsze z proponowanych rozwiązań jest w większości przypadków wystarczające. Około trzykrotnie zwiększa koszt obliczeń w stosunku do domyślnego sprawdzania płaszczyzn zawierających trójkąty sceny. Generuje bogaty zbiór płaszczyzn podziału, bardzo dobrze sprawdza się dla "prawie„ wypukłych zbiorów. Niemniej jednak nie jest wolne od wad. Zdarza się, że nie może wskazać płaszczyzny, która nie generuje zbędnych podziałów, co jest tym bardziej bolesne im mniej liczny jest zbiór trójkątów. Aby wyeliminować ten problem stosuje się metodę zdefiniowaną w drugim punkcie. Nawiązuje ona do *grafu aspektów*, generując wszystkie płaszczyzny oddzielające aspekty wizualne

zbioru trójkątów. Oznacza to, że jeśli istnieje płaszczyzna podziału nie przecinająca żadnego z trójkątów, gwarantująca akceptowalny podział, to zostanie ona znaleziona. Ze względu na złożoność $\mathcal{O}(n^3)$ jest ona wykorzystywana tylko wtedy, gdy rozmiar zbioru trójkątów jest odpowiednio mały (w praktyce przyjęto próg 30 trójkątów).



Rysunek 3.8: **(a)** Kilka płaszczyzn zaczepionych na krawędziach trójkątów, ortogonalnych do jego płaszczyzny. Jedynką wskazano najbardziej obiecujące podziały, tworzące równoliczne zbiory trójkątów i tylko jeden podział. **(b)** Kilka płaszczyzn podziału rozpiętych na wierzchołkach trójkątów. Dwójka wskazuje na optymalne podziały, każdy z otrzymanych zbiorów jest wypukły, uniknięto dzielenia trójkątów.

3.2.3.5. Wybór płaszczyzny podziału

Poprzednio omówione zostały metody generowania płaszczyzn kandydujących roli płaszczyzny podziału węzła *drzewa BSP*. W tym miejscu poruszony zostanie temat wyboru najlepszej z nich, która zostanie użyta do podziału zbioru trójkątów na te, które znajdują się przed i za nią.

Analizując całościowy proces budowy *drzewa BSP* poprzez rekurencyjny podział zbioru trójkątów przy pomocy płaszczyzn, można go oceniać go w trzech podstawowych kryteriach:

- Minimalizacji całkowitej ilości podziałów trójkątów.
- Uzyskania zrównoważonego drzewa.
- ”Dobroci„ geometrycznej otrzymanych w wyniku podziału podprzestrzeni.

Pierwsze z wymienionych kryteriów zostało uznane za najważniejsze. Minimalizacja całkowitej ilości trójkątów jest ważna, gdyż zmniejsza czas i pamięć

zużywaną w procesie renderowania sceny. Należy pamiętać, że *drzewo BSP* jest strukturą przejściową, służącą do otrzymania dekompozycji na *portale i sektory*, mającą drugorzędne znaczenie w procesie renderowania sceny (najczęściej jest w tym celu całkowicie zbędne). Dlatego warunek otrzymania zrównoważonego, a więc efektywnego w operacjach poszukiwań drzewa spada na dalszy plan.

Zagadnienie analizy i oceny kształtu otrzymanych w wyniku podziału podprzestrzeni ma marginalne znaczenie, szczególnie, że bardzo trudno zdefiniować model je oceniający, na przykład na wzór modelu *SAH* stosowanego przy konstrukcji *KD drzew*. Głównym powodem jest zupełnie inne zastosowanie budowanego *drzewa BSP* w porównaniu z *KD drzewem*.

Wracając do problemu wyboru płaszczyzny podziału w algorytmie budowy *drzewa BSP*, pozornie łatwo jest zapewnić otrzymanie zrównoważonego drzewa: wystarczy w każdym kroku algorytmu wybierać taką płaszczyznę podziału, która powoduje otrzymanie równolicznych zbiorów trójkątów. Jest to rozwiązanie krótkowzroczne, ponieważ z każdego z otrzymanych zbiorów jest rekurencyjnie budowane drzewo, więc w wyniku możliwych (i nieznanych w tym momencie) podziałów trójkątów w wywołaniach rekurencyjnych dany zbiór mogą one być reprezentowany przez znacznie większą liczbę trójkątów, zgrupowanych w węzłach otrzymanego poddrzewa. Sprawia to, że pierwotnie równoliczne zbiory, jako rekurencyjnie zbudowane drzewa nie muszą już reprezentować równolicznych zbiorów trójkątów, zebranych z liści poddrzew.

Podobnie błędne jest wybieranie takich płaszczyzn podziału, które skutkują najmniejszą liczbą podziałów trójkątów. W praktyce doprowadza to do podziału na zbiory o bardzo różniącej się liczbie elementów, w szczególnych przypadkach jeden ze zbiorów jest jednoelementowy. Ponadto okazuje się, że tylko odracza to problem (nieuniknionych) podziałów trójkątów, sprawiając, że algorytm działa na zasadzie "odcinania„ niewielkich kawałków, możliwie najmniejszą liczbą podziałów w danym kroku. Prowadzi to do otrzymania niezrównoważonego drzewa i nie gwarantuje minimalizacji całkowitej liczby podziałów.

W praktyce najlepszym kryterium okazało się jednoczesne stosowanie minimalizacji liczby podziałów i równoliczności zbiorów. Sprowadza się to do poszukiwania płaszczyzny dającej minimalną ilości podziałów trójkątów, przy nie przekraczaniu pewnego z góry ustalonego progu niezrównoważenia ilości elementów w zbiorach trójkątów leżących przed i za płaszczyzną podziału. W przypadku, gdy liczba podziałów przekracza pewną liczbę, równą ustalonemu procentowi całkowitej ilości trójkątów, powtarza się poszukiwanie przy poszerzeniu tolerancji na niezrównoważenie ilości elementów w zbiorach trójkątów leżących przed i za płaszczyzną podziału.

3.2.3.6. Wyznaczenie brył opisujących

Algorytm wyznaczający dekompozycję na *portale i sektory* wymaga, aby każdy węzeł w drzewie posiadał prostopadłościenną bryłę, która opisuje wszystkie trójkąty znajdujące się w jego liściach. Dla uproszczenia obliczeń zakłada się, że krawędzie tej bryły są równoległe do osi układu współrzędnych (jest to więc bryła typu *AABB*). W takim przypadku reprezentuje się ją przez dwa punkty: P_{min} i P_{max}, które wyznaczają współrzędne przekątnej prostopadłościanu.

Wyznaczenie brył dla wszystkich węzłów polega na przejściu drzewa metodą *postorder*, dokonując odpowiednich obliczeń w czasie odwiedzin węzłów:

procedura oblicz-bryłę-opisującą(węzeł drzewa S)
  jeśli S jest liściem
    dla każdego P_{min} = minimum(wszystkie wierzchołki trójkątów)
    P_{max} = maksimum(wszystkie wierzchołki trójkątów)
    zwróć bryła-opisująca(P_{min}, P_{max})
  w przeciwnym wypadku
    $B1$=bryła-opisująca($S.pozytywnepoddrzewo$))
    $B2$=bryła-opisująca($S.negatywnepoddrzewo$)
    P_{min} = minimum($B1.P_{min}, B2.P_{min}$)
    P_{max} = minimum($B1.P_{max}, B2.P_{max}$)
    zwróć bryła-opisująca(P_{min}, P_{max})

3.2.3.7. Zagadnienia dokładności numerycznej

W omawianych algorytmach podstawowym obliczeniem numerycznym jest wyznaczenie *odległości ze znakiem* punktu od płaszczyzny. Płaszczyznę w przestrzeni 3D można określić wskazując punkt p_0 leżący na niej oraz niezerowy wektor normalny $\vec{n}$ prostopadły do jej powierzchni. Wtedy zbiór wszystkich punktów p spełniających równanie:

$$\vec{n} \cdot (p - p_0) = 0 \tag{3.3}$$

jest pożądaną płaszczyzną. Jeśli za $\vec{n}$ podstawić (a, b, c) a za p wartość (x, y, z), wtedy powyższe równanie można zapisać w formie:

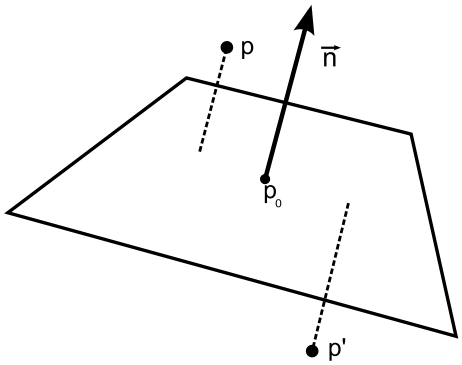
$$ax + by + cz + d = 0 \tag{3.4}$$

nazywanej *równaniem ogólnym płaszczyzny*. Postać ta jest najczęściej stosowana jako reprezentacja płaszczyzny, gdyż znakomicie upraszcza procedurę obliczania *odległości ze znakiem* punktu $p = (p_x, p_y, p_z)$ od niej. Odległość ta dana jest poniższym wzorem:

$$d = \frac{a * p_x + b * p_y + c * p_z + d}{\sqrt{a^2 + b^2 + c^2}} \tag{3.5}$$

który można uprościć, normalizując wektor normalny $\vec{n}$. Upraszcza to mianownik do jedynki. Wyliczoną wartość d, reprezentująca *odległość ze znakiem*, interpretuje się w poniższy sposób:

- Jako wartość bezwzględną: odległość punktu od płaszczyzny.
- Jako znak:
  - **Wartość dodatnia:** Punkt leży po pozytywnej stronie płaszczyzny.
  - **Wartość ujemna:** Punkt leży po negatywnej stronie płaszczyzny.
  - **Wartość równa zero:** Punkt leży na płaszczyźnie.

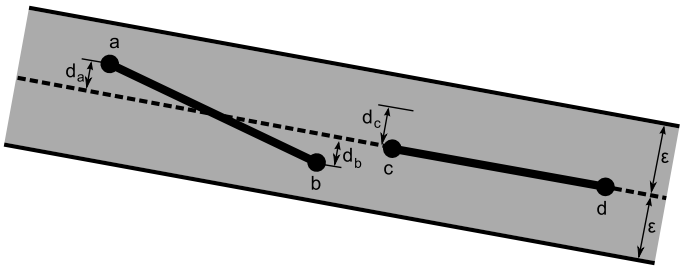


Rysunek 3.9: Płaszczyzna przechodząca przez punkt p_0, prostopadła do wektora normalnego $\vec{n}$. Odległość ze znakiem od punktu p jest większa od zera, gdyż leży po pozytywnej (wyróżnionej przez zwrot wektora normalnego) stronie płaszczyzny. Odległość ze znakiem od punktu p' jest ujemna.

Procedura klasyfikacji trójkąta względem płaszczyzny odwołuje się do omawianej powyżej *odległości ze znakiem*. Wykorzystywana jest do znalezienia najlepszej płaszczyzny podziału w *drzewie BSP* 3.2.3.1, do podziału zbioru trójkątów, do weryfikacji *wypukłości zbioru*, w algorytmach dekompozycji na *portale i sektory*, w algorytmie *portal rendering*, oraz w wielu innych zastosowaniach. Poniżej zaprezentowana jest w postaci pseudokodu:

procedura klasyfikuj-trójkąt(trójkąt T, płaszczyzna P)
  $pozytywne \leftarrow 0$
  $negatywne \leftarrow 0$
  dla każdego punktu $t \in S$ **wykonaj**
    $d \leftarrow$odległość-ze-znakiem(P, t)
    jeśli $d > 0$
      $pozytywne \leftarrow pozytywne + 1$
    w przeciwnym wypadku, jeśli $d < 0$
      $negatywne \leftarrow negatywne + 1$
  jeśli $pozytywne > 0$ **oraz** $negatywne > 0$
    zwróć płaszczyzna-rozcina-trójkąt
  w przeciwnym wypadku, jeśli $pozytywne > 0$
    zwróć trójkąt-jest-po-stronie-pozytywnej
  w przeciwnym wypadku jeśli $negatywne > 0$
    zwróć trójkąt-jest-po-stronie-negatywnej
  w przeciwnym wypadku zwróć trójkąt-zawiera-się-w-płaszczyźnie

W praktyce obliczenia przeprowadzane są ze skończoną precyzją, w arytmetyce zmiennoprzecinkowej. Błędy reprezentacji i niedokładności obliczeń powodują, że przy porównywaniu wartości zmiennoprzecinkowych konieczne jest uwzględnianie pewnej tolerancji. Niech przykładem będzie powyższa procedura **klasyfikuj-trójkąt** i przypadek, gdy jej argumentami są: dowolna płaszczyzna i trójkąt zawierający się w niej. W praktyce, odległości ze znakiem wierzchołków trójkąta od płaszczyzny nie będą zerami, lecz niewielkimi liczbami bliskimi zeru. Oznacza to, że trójkąt leżący na płaszczyźnie może zostać uznany za leżący po jednej z jej stron, a nawet przecinający płaszczyznę. Może to doprowadzić do błędów, w szczególności do nieskończonej rekurencji w procesie budowy *drzewa BSP*.



Rysunek 3.10: Klasyfikacja trójkątów leżących na płaszczyźnie. Ze względu na niedokładność reprezentacji punkty mogą leżeć poza płaszczyzną (punkty *a* i *b*), bądź też obliczona odległość od prostej może być obarczona błędem (punkt *c*). Wyznaczenie odpowiedniej tolerancji ϵ pozwala na poprawną klasyfikację.

Kwalifikacja punktu leżącego na płaszczyźnie musi odbywać się w granicach pewnej tolerancji ϵ. Rzutuje to na linie 5-8 algorytmu 3.2.3.7, które po modyfikacji przyjmują postać:

jeśli $d > \epsilon$
    $pozytywne \leftarrow pozytywne + 1$
w przeciwnym wypadku, jeśli $d < -\epsilon$
    $negatywne \leftarrow negatywne + 1$

Podobnych modyfikacji wymaga opisywany w 3.2.1 algorytm sprawdzający wypukłość zbioru trójkątów.

Należy zastanowić się, na ile realne są opisywane problemy. Teoretycznie, prawdopodobieństwo że w losowym zbiorze trójkątów znajdą się dwa, leżące na tej samej płaszczyźnie, jest równe zeru. Wydaje się, że zawieranie się płaszczyzny i trójkąta może występować tylko w przypadku, gdy rozpatrywana płaszczyzna została stworzona na podstawie danego trójkąta. Sytuację tą można by wykrywać zawczasu i specjalnie obsługiwać. W praktyce ta hipotetyczna sytuacja pojawia się zaskakująco często, jako konsekwencja pewnych faktów:

- W realnych modelach 3D istnieją duże grupy trójkątów leżących na wspólnej płaszczyźnie. Przykładem są trójkąty stanowiące elementy podłogi, sufitu, ścian. Podłogi z pomieszczeń położonych na tej samej kondygnacji leżą na tym samym poziomie, a więc w oczywisty sposób także na tej samej płaszczyźnie. Do tej sytuacji przyczyniają się też programy edycyjne grafiki 3D, w których powszechnie stosuje się narzędzia "przyciągania„ edytowanych elementów grafiki do istniejących już punktów, krawędzi czy tez płaszczyzn.

- Z reguły informacja o tym, na jakiej płaszczyźnie leży dany trójkąt nie jest dostępna w modelu 3D. Wymusza to odtworzenie jej na podstawie współrzędnych wierzchołków trójkątów. Jeśli przyjmiemy, że $p1 = (x_1, y_1, z_1)$, $p2 = (x_2, y_2, z_2)$ oraz $p3 = (x_3, y_3, z_3)$ są trzema wierzchołkami trójkąta, to płaszczyzna w której się zawiera wyraża się wzorem:

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = 0 \tag{3.6}$$

  który rozwija się do kilkunastu mnożeń i dodawań, oraz wyciągania pierwiastka kwadratowego w celu otrzymania równania ogólnego płaszczyzny ze znormalizowanym wektorem normalnym. Z powodu błędów nu-

merycznych i jej skończonej precyzji tak odtworzone płaszczyzny trójkątów, pierwotnie leżące na wspólnej płaszczyźnie, mogą się nieznacznie różnić. Problem staje się istotny, jeśli wspólna płaszczyzna jest bardzo duża (na przykład podłoga), zaś trójkąty z których odtwarza się płaszczyznę bardzo małe i oddalone od siebie. Zagadnienie to jest znane w literaturze, w [Kir92, str. 225-230] opisywany jest algorytm zbierający "prawie„ współpłaszczyznowe trójkąty w grupy, położone na wspólnej płaszczyźnie. W zastosowaniach wymienia się konstrukcję *drzewa BSP* o mniejszej liczbie podziałów.

- Analogicznie do powyższej sytuacji, w scenie istnieją zbiory krawędzi należących do różnych trójkątów, leżące na wspólnej płaszczyźnie. Mogą naturalnie występować w modelu a także pojawić się też w wyniku podziału trójkątów płaszczyznami[6]. Heurystyki generowania płaszczyzn podziału *drzewa BSP*, opisane w 3.2.3.4, mogą generować wspomniane płaszczyzny w oparciu o współrzędne wierzchołków, co analogicznie jak w poprzedniej sytuacji może generować kłopoty przy klasyfikacji trójkątów, których krawędzie zawierają się w generowanych płaszczyznach.
- W sytuacji, gdy niezbędny jest podział trójkąta na dwa bądź trzy, istotne jest, aby płaszczyznę dla każdego z nich skopiować z pierwotnego trójkąta zamiast odtwarzać ją na podstawie współrzędnych wierzchołków. Jest to szczegół implementacyjny, warty wspomnienia ze względu na istotny wkład w proces poprawy stabilności budowy *drzewa BSP*.

Omawiane zagadnienia uwiarygodniają podstawową trudność implementacji algorytmu budowy i przetwarzania *drzewa BSP*, którą jest odporność na niedokładności obliczeń. Przykładowo, w przypadku procedury **klasyfikuj-trójkąt**, przy całkowitych rozmiarach sceny 3D rzędu tysięcy i minimalnych długościach krawędzi trójkątów rzędu dziesiątych, konieczne okazało się użycie ϵ równego 10^{-5}, przy obliczeniach w typie **double**, gwarantującym precyzję 15 cyfr dziesiętnych.

3.3. Ekstrakcja portali i sektorów z drzewa BSP

Cechą interpretacji *drzewa BSP* jako podziału na *portale i sektory* jest bardzo duża liczba sektorów, z reguły nie mniejsza niż 1/2 liczby wszystkich trójkątów sceny 3D. Ma to istotne znaczenie w zastosowaniach:

- W *PVS renderingu*, gdzie dokonuje się statycznego wyliczenia widoczności sektorów z każdego sektora, jest to podział jak najbardziej akceptowalny.

[6]Są to skrajne płaszczyzny zbioru trójkątów, całkowicie nieprzydatne jako potencjalne płaszczyzny podziału.

Sektory są wystarczająco małe, aby zbiór *potencjalnie widocznych sektorów* nie przeszacowywał widoczności, jednocześnie na tyle mały, aby reprezentacja *zbioru PVS* była rozsądnych rozmiarów, a jego wyznaczenie było możliwe w sensownym czasie.

- W *Portal renderingu*, w którym zbiór widocznych sektorów jest wyznaczany dynamicznie dla każdej pozycji i orientacji kamery, dla rzeczywistych scen zawierających dziesiątki tysięcy trójkątów jest on stanowczo za duży. Narzut obliczeń związanych z przetwarzaniem tak dużej liczby portali i sektorów przekracza zyski w postaci renderowania bardzo dokładnie wyznaczonego zbioru widocznych fragmentów sceny. Dlatego dla potrzeb *Portal renderingu* konieczne jest dodatkowe przetwarzanie, zmniejszające ilość, zarówno portali jak i sektorów. Najbardziej rozpowszechnione jest selektywne łączenie sąsiednich sektorów, oraz uproszczanie i grupowanie portali.

Poniżej przedstawiony jest w detalach algorytm służący do budowy *portali i sektorów* na podstawie *drzewa BSP*, oraz jego selektywnego upraszczania na potrzeby *Portal renderingu*.

3.3.1. Wyznaczanie sektorów

Wyznaczanie sektorów (dokładniej: *BSP sektorów*) jest najprostszą częścią opisywanego algorytmu. Sektorami są liście *drzewa BSP*, więc do ich stworzenia wystarczy przejść drzewo, tworząc dla każdego liścia nową strukturę reprezentującą sektor. Zawiera ona następujące dane:

- Lista trójkątów (skopiowana z liścia drzewa).
- Lista portali z którymi się styka (początkowo pusta).

Dodatkowo na potrzeby algorytmu wyznaczającego portale między sektorami, a także w celu szybszego dostępu do danych, wprowadza się dodatkowe pola i powiązania między obiektami:

- Dla sektora, powiązanie z liściem drzewa, z którego został stworzony, tworzone w momencie konstrukcji sektora.
- Dla liścia drzewa, powiązanie z sektorem go reprezentującym, tworzone w momencie konstrukcji sektora.
- Dla węzła drzewa, bryła prostopadłościenna, opisująca trójkąty zgromadzone w poddrzewie reprezentowanym przez niego, wyznaczana po zbudowaniu *drzewa BSP*.

3.3.2. Wyznaczanie potencjalnych portali

Potencjalne portale są wielokątami wypukłymi zawartymi w płaszczyznach podziału *drzewa BSP*, bez przypisanych połączeń sektor-sektor. Połączenia te będą odtworzone w dalszym stadium, powodując uzyskanie gotowej struktury *BSP portali* i *BSP sektorów* powiązanych w *graf CPG*.

Wielokąt jest reprezentowany jako lista jego wierzchołków wraz z płaszczyzną na której się znajdują. Jej orientacja jest istotna, musi być skierowana ku wnętrzu sektora.

Potencjalnych portali szuka się w każdym *BSP sektorze*. Przetwarza się go w trzech, następujących po sobie etapach. Znalezione wielokąty gromadzi się w jednej kolekcji, usuwając ewentualne duplikaty[7].

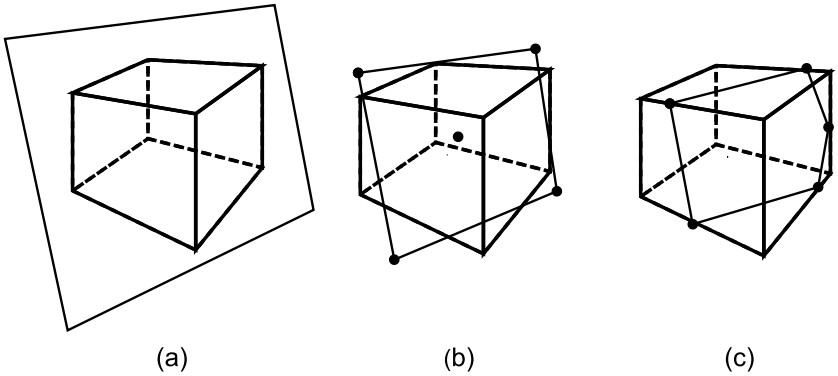
3.3.2.1. Wyznaczenie listy wielokątów

Celem tego kroku jest stworzenie listy wielokątów zawierających się w płaszczyznach ograniczających rozpatrywany sektor. Najpierw znajduje się odpowiadający mu węzeł *drzewa BSP*, następnie przechodząc ścieżkę do korzenia zbiera się wielokąty zbudowane na płaszczyznach podziału węzłów, przycięte do bryły opisującej główny węzeł drzewa. Budowa każdego z takich wielokątów polega wpierw na stworzeniu dużego wielokąta-kwadratu o punktach leżących na płaszczyźnie podziału, o długości boku równej podwojonej długości przekątnej bryły opisującej główny węzeł drzewa i środku znajdującym się gdzieś w jej wnętrzu. Łatwo wykazać, że taki prostokąt przecina całą swoją powierzchnią tą bryłę. Następnie przycina się go sześcioma płaszczyznami, w których zawierają się ściany bryły opisującej główny węzeł. Przycięcie każdą z płaszczyzn polega na odcięciu od wielokąta części znajdującej się po jej negatywnej stronie. Przetwarzanie pojedynczej płaszczyzny podziału przedstawiono na rysunku 3.11.

3.3.2.2. Przycięcie do granic sektora

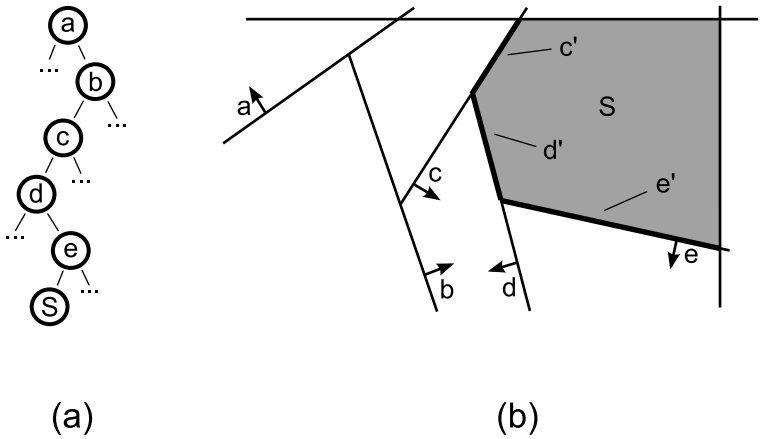
Stworzone wielokąty mogą "wystawać„ poza granicę sektora. Celem kolejnego kroku jest pozbycie się tych "wystających„ części, poprzez przycięcie każdego z wielokątów wszystkimi płaszczyznami na ścieżce od liścia *drzewa BSP* odpowiadającemu aktualnie przetwarzanemu sektorowi do korzenia. Jeśli płaszczyzna przycinająca posiada wektor normalny skierowany przeciwnie do sektora (oznacza to, że podprzestrzeń, w której znajduje się sektor leży po negatywnej stronie płaszczyzny podziału), to przycięcia dokonuje się płaszczyzną o przeciwnym zwrocie normalnej. W tym miejscu można wyjaśnić wcześniejsze przycinanie "dużego„ wielokąta do granic bryły otaczającej główny węzeł, ma to na celu eliminację wystających poza bryłę opisującą całą scenę *potencjalnych*

[7]Dwa wielokąty uznaje się za takie same, jeśli są rozpięte na tym samym zbiorze punktów.



Rysunek 3.11: Kolejne etapy przetwarzania pojedynczej płaszczyzny podziału. **(a)** Płaszczyzna podziału względem bryły opisującej głównego węzła. **(b)** Zbudowany, leżący na niej duży kwadrat. **(c)** Duży kwadrat po przycięciu wszystkimi sześcioma płaszczyznami bryły otaczającej.

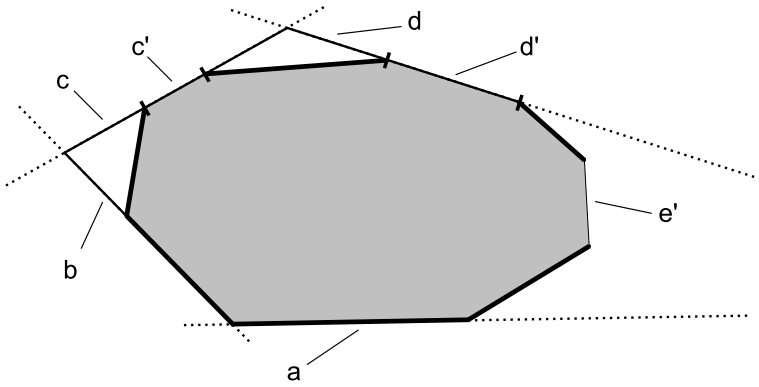
portali w sektorach, które w *drzewie BSP* reprezentowane są przez częściowo otwarte podprzestrzenie. W czasie tego procesu może się okazać, że pewne wielokąty zostały przycięte "do zera„ (wszystkie wierzchołki wielokąta znajdowały się po negatywnej stronie płaszczyzny przycięcia). Sytuacja ta oznacza, że przetwarzany wielokąt nie sąsiaduje bezpośrednio z przetwarzanym sektorem. Tak otrzymane "puste„ wielokąty są usuwane z wyjściowego zbioru. Rysunek 3.12, dla jasności sytuacji przedstawiający przypadek dwuwymiarowy, obrazuje omawiany proces.



Rysunek 3.12: **(a)** Fragment ścieżki w *drzewie BSP*, od węzła głównego *a*, do liścia (sektora) *S*. **(b)** Jej reprezentacja graficzna wraz z płaszczyznami bryły ograniczającej. Po przycięciu pozostają wielokąty *c′*,*d′*,*e′*.

3.3.2.3. Przycięcie do otworów w modelu

Poprzednia operacja zapewnia otrzymanie listy wielokątów leżących na płaszczyznach podziału, sąsiadujących z rozpatrywanym sektorem. Konieczne jest jeszcze jedno przetwarzanie, odwołujące się do geometrii w nim zgromadzonej, przycinające wielokąty do granic otworów w części sceny 3D, którą sobą reprezentuje. W ten sposób uzyskuje się wielokąty, które reprezentują otwory w strukturze sektora. Ponieważ geometria sektora jest bryłą wypukłą, zadanie to sprowadza się do przycięcia każdego wielokąta płaszczyznami zawierającymi wszystkie trójkąty, o normalnych skierowanych w kierunku ich widocznej strony.



Rysunek 3.13: W poprzednim kroku uzyskano wielokąty a,b,c,d,e. Po przycięciu do granic sektora pozostają c' oraz d'. Otwór e' nie jest reprezentowany przez żaden z wielokątów, gdyż nie leży na żadnej z płaszczyzn podziału.

Należy wyraźnie zaznaczyć, że powyższa operacja uzasadniona jest założeniem, że o ile trójkąty widoczne są z jednej strony, o tyle widoczność blokują z obydwu. Założenie to jest jak najbardziej uzasadnione, gdyż jednostronność powierzchni trójkąta to nic innego jak implementacja *eliminacji ścian tylnych* a nie własność sceny 3D. Narzędzia służące do modelowania scen respektują to założenie, aczkolwiek można się spotkać ze scenami nie stworzonymi wedle tej zasady.

Odrębnego rozpatrzenia wymaga sytuacja, gdy przycinany wielokąt leży na jednej z płaszczyzn przycinających. Występują tutaj trzy przypadki:

- Wielokąt *potencjalnego portalu* całkowicie zawiera się w tym trójkącie. Jest to trywialny przypadek oznaczający, że blokuje on w całości widoczność przez ten wielokąt pomiędzy sąsiadującymi sektorami i może zostać odrzucony.

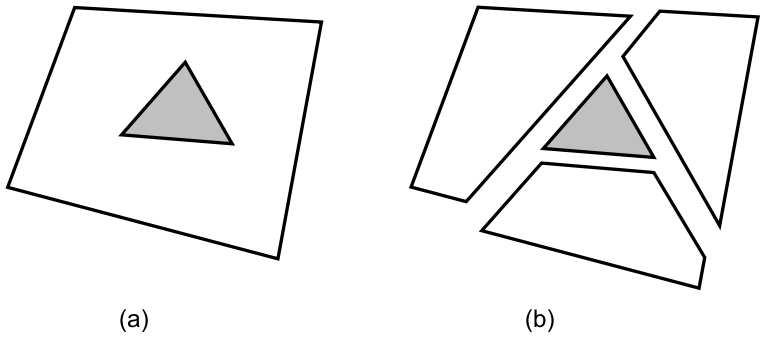
- Wielokąt *potencjalnego portalu* nie przecina się z danym trójkątem. Płasz-

czyzna trójkąta nie ma w tym przypadku wpływu na rozpatrywany wielokąt, który pozostawia się bez zmian.

- Trójkąt częściowo przecina się z wielokątem *potencjalnego portalu*. Część wspólna przecięcia nie uczestniczy w propagowaniu widoczności przez portal i powinna być odcięta z *potencjalnego portalu*. Nie należy tego wykonywać wprost, gdyż może to doprowadzić do powstania wielokąta niewypukłego, bądź posiadającego otwory. Jest to nieakceptowalne, ponieważ wymaga się aby wielokąt portalu był wypukły. W takim przypadku stosuje się dwa rozwiązania:

  – Pozostawienie wielokąta *potencjalnego portalu* bez zmian, godząc się z pewnym przeszacowaniem obszaru portalu.

  – Rozcięcie wielokąta krawędziami trójkąta na wiele wielokątów, co umożliwia odrzucenie części wspólnej, gdyż pojawia się ono jako jeden z rozciętych wielokątów. Skutkiem takiego postępowania może być duża liczba małych, współpłaszczyznowych, sąsiadujących ze sobą portali, co jest nieporęczne w późniejszym ich przetwarzaniu. Próbuje się temu zaradzić poprzez ich późniejsze łączenie, bądź aproksymację wielokątem reprezentującym ich otoczkę wypukłą.



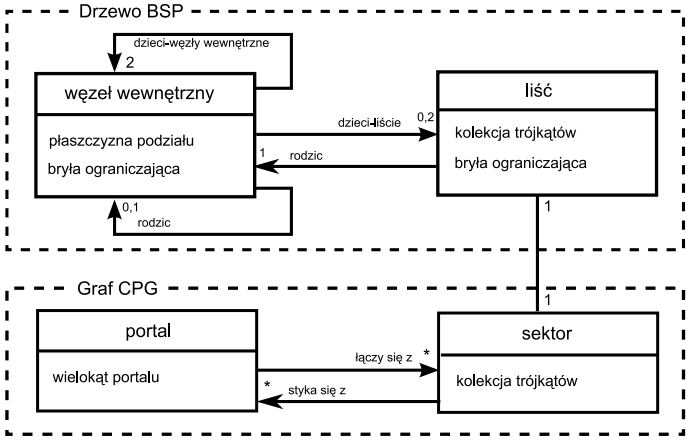
Rysunek 3.14: **(a)** Przykład sytuacji, w której z czworokąta należy wykluczyć trójkątny obszar. **(b)** Rozcięcie czworokąta krawędziami trójkąta umożliwia wykluczenie tego obszaru. W rezultacie otrzymuje się trzy wypukłe wielokąty.

3.3.3. Łączenie portali z sektorami

Operacja łączenia portali z sektorami przyporządkowuje *potencjalnym portalom* listę sektorów z którymi się łączą. Powoduje to stworzenie grafu widoczności, w którym węzłami są *BSP sektory*, zaś krawędziami łączące je *BSP portale*. Aby wyrazić to powiązanie, strukturę *BSP portalu* definiuje się jako:

- Wielokąt reprezentujący *potencjalny portal* wraz z płaszczyzną na której leży.
- Listę *BSP sektorów*, z którymi portal się styka.

Dla ułatwienia, a wręcz dla umożliwienia korzystania z tego grafu, każdy *BSP sektor*, opisany w punkcie 3.3.1 rozszerza się o listę *BSP portali*, z którymi ten się łączy. Ostateczny diagram połączeń pomiędzy obiektami *grafu CPG* i *drzewa BSP* prezentuje rysunek 3.15.



Rysunek 3.15: Powiązania pomiędzy obiektami *drzewa BSP* i *grafu CPG*.

Algorytm łączenia portali i sektorów wyraża się w poniższych czynnościach:

3.3.3.1. Kojarzenie portali i sektorów

Celem tego kroku jest wyznaczenie dla każdego *potencjalnego portalu* wszystkich sektorów, z którymi się styka. W tym celu każdy wielokąt *potencjalnego portalu* "opuszcza„ się w dół drzewa, zaczynając od jego korzenia. Przetwarzając wewnętrzny węzeł drzewa, w zależności od wzajemnej orientacji płaszczyzny podziału i wielokąta postępuje się wedle poniższych kryteriów:

- Gdy wielokąt znajduje się w całości po pozytywnej bądź po negatywnej stronie płaszczyzny podziału, to przetwarza się rekurencyjnie poddrzewo w którym się zawiera.
- Płaszczyzna podziału przecina wielokąt, bądź wielokąt leży w płaszczyźnie podziału. Wtedy rekurencyjnie przetwarza się obydwa poddrzewa.

Ostatecznie przetwarzanie dojdzie do liścia, w którym *potencjalny portal* został utworzony a także do tych, które się z nim stykają. Pozostaje tylko sprawdzić, czy *potencjalny portal* umożliwia propagację widoczności do geometrii reprezentowanej przez sektory, do których został przypisany. Stosuje się do tego uproszczoną[8] metodę opisaną w 3.3.2.3, polegającą na sprawdzeniu orientacji wielokąta *potencjalnego portalu* względem każdej płaszczyzny trójkąta geometrii, oraz:

- Odrzuceniu sektora, gdy istnieje taki trójkąt należący do geometrii reprezentowanej przez sektor, że wielokąt znajduje się w całości po jego negatywnej stronie.
- Odrzuceniu sektora, gdy istnieje trójkąt leżący na tej samej płaszczyźnie co wielokąt, zawierający w całości rozpatrywany wielokąt[9].

W przeciwnym wypadku stwierdza się, że wielokąt *potencjalnego portalu* uczestniczy w propagacji widoczności do przetwarzanego sektora i ustanawia między nimi powiązanie.

3.3.3.2. Usuwanie nieprawidłowych portali

Portale, które łączą się tylko z jednym sektorem, nie mają wpływu na propagację widoczności i mogą być bezpiecznie usunięte. Należy zastanowić się, czy taka sytuacja rzeczywiście może się zdarzyć. W przypadku rzeczywistych modeli 3D nierzadko jest to prawdą. Bardzo często w wyniku błędów modelowania pojawiają się (niewielkie) fragmenty sceny, w których pojedynczy trójkąt blokuje widoczność "w jedną stronę,,. Czasami też scena projektowana jest pod kątem obserwacji tylko z pewnych, ściśle określonych obszarów, co przyczynia się do uproszczenia geometrii. Rysunek 3.16 jest przykładem, w którym specyficzna konstrukcja sceny determinuje utworzenie portalu połączonego tylko z jednym sektorem.

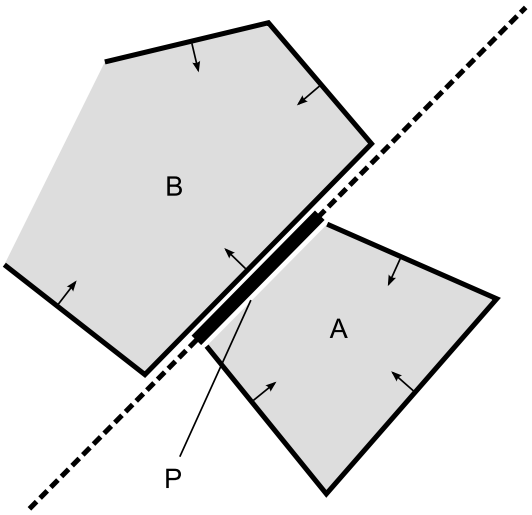
3.3.3.3. Usuwanie nadmiarowych portali

Można zauważyć, że algorytm konstrukcji dekompozycji na *portale i sektory* dopuszcza łączenie się portalu z wieloma sektorami. Stoi to w sprzeczności z definicją *portalu*, jako obiektu propagującego widoczność pomiędzy dwoma *sektorami* (2.3.1).

Analiza działania przedstawionego algorytmu pozwala odnaleźć przyczynę takiego zachowania. Jest nią przyjęta metoda generowania *potencjalnych portali* oraz ich dystrybucji w *drzewie BSP*, co w wyniku możliwości dzielenia tej samej

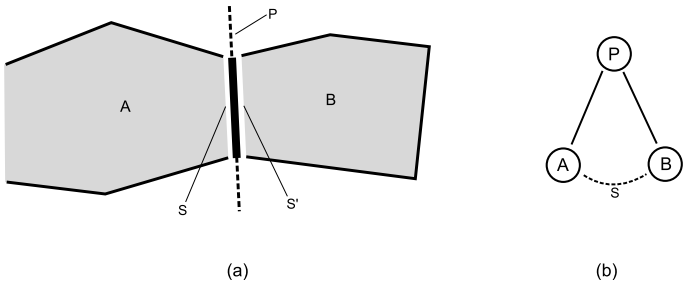
[8] Nie stosuje się przycinania wielokąta przez płaszczyzny.

[9] W rzeczywistości warto zastosować pewną tolerancję, aby uniknąć błędów numerycznych, a także aby odrzucić portale widoczne dopiero w skali mikro.



Rysunek 3.16: Przykład sytuacji, gdy portal łączy się tylko z jednym sektorem. Portal *P* zostanie połączony z sektorem *A*, z sektorem *B* nie, gdyż nie jest widoczny z wnętrza geometrii reprezentowanej przez sektor.

płaszczyzny przez więcej niż dwa liście drzewa może prowadzić do wspomnianej sytuacji. Rozważmy najczęściej spotykany przypadek sąsiadowania dwóch sektorów:



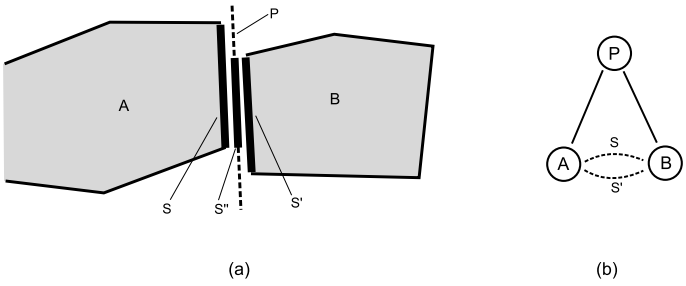
Rysunek 3.17: **(a)** Typowy przykład sąsiadowania dwóch sektorów. Otwór w sektorze *A*, leżący na płaszczyźnie podziału jest oparty na tych samych punktach co otwór w sektorze *B*. **(b)** Drzewo BSP przedstawionego przykładu. Liniami przerywanymi oznaczono relację widoczności reprezentowaną przez portale.

Otwór w sektorze **A** leżący na płaszczyźnie podziału, w procesie opisanym w rozdziale 3.3.2 zostanie sklasyfikowany jako *potencjalny portal* **S**i zapamiętany. Podobnie otwór z sektora **B** zostanie sklasyfikowany jako *potencjalny portal* **S'**, jednak ze względu na to, że jest rozpięty na tych samych punktach co **S**, zostanie odrzucony. W procesie kojarzenia sektorów i portali (3.3.3.1) portal **S** trafi

tylko i wyłącznie do sektorów **A** i **B**, prawidłowo opisując relację widoczności pomiędzy nimi.

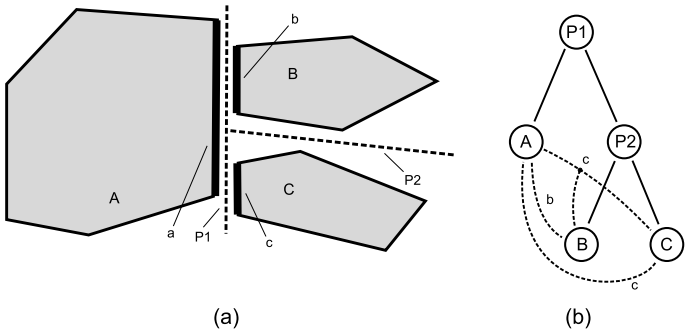
W rzeczywistości zdarzają się bardziej złożone przypadki. Jednym z nich jest wariant poprzednio omawianego przykładu, w którym dwa sektory sąsiadują ze sobą posiadając otwory leżące na wspólnej płaszczyźnie, rozpięte na innych punktach. Pokazuje to rysunek 3.18. W takim przypadku *potencjalny portal* **S** trafi do sektorów **A** i **B**, tworząc między nimi relację widoczności. Podobnie *potencjalny portal S'*, który także trafi do sektorów **A** i **B**, reprezentując widoczność między nimi. Powoduje to zauważalną nadmiarowość, gdyż widoczność pomiędzy obydwoma sektorami jest reprezentowana przez dwa portale, podczas gdy wystarczyłby w zupełności jeden, będący ich częścią wspólną. Istnieje dwojakie rozwiązanie tej sytuacji:

- Zastąpienie obydwu portali jednym, będącym ich częścią wspólną. Jest to prawidłowe, gdyż część wspólna brył wypukłych także pozostaje bryłą wypukłą.
- Usunięcie jednego z portali. Ideologią takiego postępowania jest aproksymacja części wspólnej portali przez jeden z nich. Jest ona konserwatywna, a więc nie powoduje błędów widoczności. Nadmiarowość propagacji widoczności można zmniejszyć usuwając portal o większej powierzchni.



Rysunek 3.18: Inny wariant sąsiadowania sektorów. **(a)** Otwory sektorów *A* i *B* leżą na tej samej płaszczyźnie, lecz rozpięte są na innych punktach. *S"* przedstawia "idealny„ portal, będący częścią wspólną portali *S* i *S'*. **(b)** Drzewo BSP przedstawionego przykładu. Liniami przerywanymi oznaczono relację widoczności reprezentowaną przez portale.

Przedstawiony przykład jest szczególnym przypadkiem pewnej ogólnej sytuacji, w której więcej niż dwa sektory dzielą wspólną płaszczyznę. Może to prowadzić do portali, które łączą więcej niż dwa sektory. Pokazane jest to na rysunku 3.19.



Rysunek 3.19: Ogólna sytuacja sąsiadowania sektorów. **(a)** Trzy sektory stykają się z jedną płaszczyzną. Portal *a* łączy wszystkie trzy sektory. **(b)** Drzewo BSP przedstawionego przykładu. Liniami przerywanymi oznaczono relację widoczności reprezentowaną przez portale.

Potencjalne portale **b** i **c** utworzą połączenia z sektorami, odpowiednio **A** i **B** oraz **A** i **C**. Przypadek *potencjalnego portalu* **a** jest zaskakujący, gdyż połączy on sektory **A**, **B** i **C**. Widać, że połączenie to jest nonsensowne: promienie świetlne propagują się po prostej, więc nie jest możliwe, aby sektor **B** był widoczny z sektora **C**. Portal **a** jako redundantny może być więc usunięty. Powyższe rozważania prowadzą do algorytmu usuwającego nadmiarowe portale:

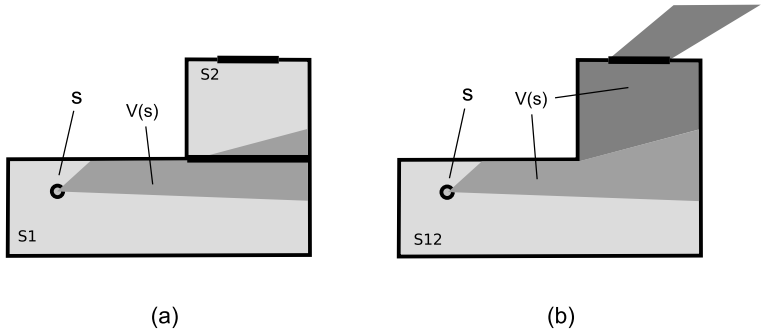
1. Dla każdego sektora wykonaj poniższe kroki:
2. Zgrupuj portale leżące na tej samej płaszczyźnie. Dla każdej grupy wykonaj poniższe czynności:
3. W przypadku znalezienia dwóch portali łączących dwa sektory które propagują widoczność do tego samego sektora, odepnij od sektorów i usuń ten o większej powierzchni.
4. W przypadku znalezienia portalu łączącego dwa sektory oraz portalu łączącego więcej niż dwa sektory propagujących widoczność do wspólnego sektora, odepnij od sektora ten drugi portal.

Algorytm ten, jaki i rozważania z których on wynika tracą sens, gdy z jednego sektora, z jednej płaszczyzny generuje się wiele *potencjalnych portali*. Taka sytuacja może miejsce, gdy *potencjalny portal* i jeden z trójkątów sektora są współpłaszczyznowe. Dokonany wtedy podział, wykluczający część zawierającą tego trójkąta może doprowadzić do powstania w miejscu dzielonego *potencjalnego portalu* wiele mniejszych. Najlepszym rozwiązaniem jest odroczenie usuwania nadmiarowych portali do czasu zakończenia procedury łączenia sektorów, gdyż zagadnienie to pojawia się tam jeszcze raz, w postaci łączenia portali od połączonych sektorów i eliminacja tych, które są zbędne (propagują widoczność pomiędzy połączonymi sektorami).

3.4. Łączenie sektorów

Bezpośrednia konstrukcja *grafu CPG* z *drzewa BSP* w przypadku realnych scen nie bardzo nadaje się do wykorzystania w algorytmie *Portal-rendering*. Uzyskany podział jest zbyt złożony, gdyż sektorów jest tylko niewiele mniej niż wynosi całkowita liczba ścian w scenie. Powoduje to, że koszt wyznaczania widocznych sektorów w algorytmie *Portal-rendering* jest dominujący w całym procesie renderowania sceny, duże większy niż zysk w postaci nie rysowania niewidocznych fragmentów sceny.

Można zauważyć, że słabość zaprezentowanego algorytmu leży w generowaniu wypukłych sektorów. Okazuje się, że własność ta nie jest warunkiem niezbędnym dla poprawnego działania algorytmu *Portal-rendering*. Jej brak sprawia, że algorytm przestaje być dokładny i może wskazywać pewne sektory jako widoczne, podczas gdy w rzeczywistości takimi nie są.



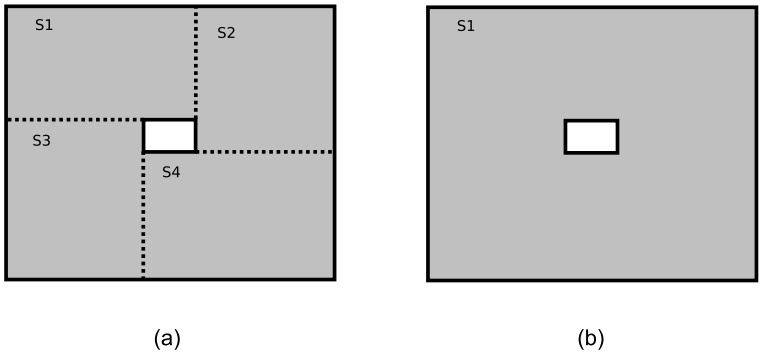
Rysunek 3.20: **(a)**: Podział sceny na dwa wypukłe sektory, oraz obszar widzenia *V(s)* z ustalonego punktu *s*. **(b)** Rezygnacja z warunku wypukłości może generować fałszywy obszar widoczności, przedstawiony jako ciemniejszy fragment obszaru *V(s)*.

W praktyce niewielkie zaburzanie wypukłości może istotnie przyczynić się do zmniejszenia ilości sektorów, tylko nieznacznie zwiększając konserwatywność wyznaczania widoczności.

Aby zmniejszyć liczbę sektorów stosuje się zmniejszenie liczby sektorów poprzez ich kontrolowane łączenie. Polega to na heurystycznej ocenie połączenia dwóch, bądź większej liczby sąsiadujących ze sobą sektorów i zrealizowanie go jeśli uznane zostanie jako wartościowe. W ten sposób udaje się osiągnąć wystarczająco niską liczbę sektorów.

3.4.1. Istniejące rozwiązania

W pracy [LH03] przedstawiono algorytm łączenia *BSP sektorów*. Dla każdego sektora a także dla połączonych sektorów definiuje się *koszt renderowania*,



Rysunek 3.21: **(a)** Podział sceny ze słupem w środku na cztery wypukłe sektory. **(b)** Rezygnacja z warunku wypukłości umożliwia objęcie całego obszaru jednym sektorem.

będący funkcją ilości i rozmiaru trójkątów w nich zawartych, oraz rozmiaru zajmowanego w przestrzeni. W przedstawionym algorytmie dąży się do uzyskania sektorów o koszcie renderowania nie większym i nie mniejszym niż zadane granice. Dolna granica mówi o tym, że koszt przetwarzania sektora przez algorytm *Portal-rendering* będzie większy niż koszt jego renderowania, górna dba o to, aby pewne sektory nie skupiały w sobie zbyt wiele geometrii, prawdopodobnie poważnie "psujących„ wypukłość. Całość zagadnień zbiera poniżej opisany algorytm:

1. Policz funkcję kosztu renderowania dla wszystkich sektorów i stwórz listę sektorów posortowaną według niego.

2. Dopóki istnieje sektor o koszcie mniejszym niż dolna granica wykonuj następny punkt.

3. Weź sektor o najmniejszym koszcie renderowania i przejrzyj wszystkie sektory z którymi łączy się przez portale. Wybierz taki, z którym połączenie będzie najlepsze wedle pewnej heurystycznej oceny, zaś koszt renderowania po połączeniu nie będzie większy niż górna granica. Połącz tą parę w jeden sektor i uaktualnij listę usuwając z niej obydwa pierwotne sektory i wstawiając w odpowiednie miejsce połączony.

W podsumowaniu autorzy przedstawiają dane świadczące o skuteczności przedstawionej metody.

Zgoła inne spojrzenie na problem dekompozycji na *sektory* występuje w pracy [MBWW07]. Główną ideą jest uzyskanie w procesie wstępnego przetwarzania informacji o scenie w postaci zbioru odcinków nieprzecinających żadnego z trójkątów sceny z wyjątkiem ich końców, które leżą na trójkątach.

Następnie dokonuje się rekurencyjnego podziału sceny przy pomocy płaszczyzn prostopadłych do osi układu współrzędnych, używając zgromadzonej w zbiorze odcinków informacji o widoczności do wyboru optymalnego podziału. W rezultacie otrzymuje się zbiór sektorów wraz z informacją o tym, jakie sektory widoczne są z każdego innego sektora.

3.4.2. Algorytm łączenia sektorów.

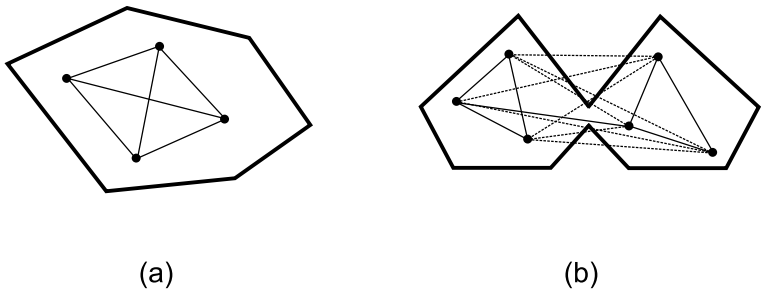
Algorytm łączenia zaprezentowany w [LH03] nie uwzględnia tego, jak zmienia się wypukłość łączonych sektorów i czy mieści się w akceptowalnym zakresie. Ponadto objętość sektorów szacuje jako objętość bryły je otaczającej, co jak wspominają autorzy może prowadzić do jej znacznych przeszacowań. Stało się to motywacją do ulepszenia, w którym "wypukłość„ łączonych sektorów jest mierzona i kontrolowana, zaś objętość sektorów wyznaczana jest w sposób znacznie bardziej dokładny.

3.4.2.1. Współczynnik wypukłości.

Wypukłość jest własnością, którą obiekt może posiadać bądź nie. Aby mierzyć stany pośrednie, definiujemy **współczynnik wypukłości** nad zadanym, niepustym zbiorem punktów P, z których każdy należy do wnętrza obiektu:

$$wspolczynnik\ wypuklosci = \frac{liczba\ par\ punktow,\ ktore\ sa\ nawzajem\ widoczne}{liczba\ wszystkich\ par\ punktow} \tag{3.7}$$

Dla zbioru wypukłego, niezależnie od wybranego zbioru punktów *współczynnik wypukłości* jest równy 1, ze względu na to, że każde dwa punkty należące do niego są nawzajem widoczne. W przypadku zbiorów niewypukłych jego wartość jest mniejsza od jedności.



Rysunek 3.22: Ilustracja *współczynnika wypukłości*. **(a)** Wypukły obiekt, wszystkie pary punktów są widoczne, wartość współczynnika wynosi 1. **(b)** Tylko 7 z 15 par punktów jest wzajemnie widocznych, co daje współczynnik równy 7/15.

3.4.2.2. Objętość sektorów.

Nie jest łatwo obliczyć dokładną objętość wnętrza sektora. Jest ona równa objętości bryły wyznaczonej przez płaszczyzny podziału *drzewa BSP*, przyciętej płaszczyznami bryły otaczającej całą scenę oraz płaszczyznami zawierającymi się w ścianach trójkątów. W praktyce, na potrzeby algorytmu łączenia sektorów wystarczająca jest przybliżona wartość, którą można szybko obliczyć korzystając z poniższej metody *Monte Carlo*:

1. Rozszerz strukturę sektora o licznik trafień i nadaj mu początkową wartość równą zeru. Wprowadź globalny licznik trafień inicjowany na zero.
2. Wykonuj poniższe kroki aż do czasu otrzymania satysfakcjonującej dokładności:
3. Wylosuj punkt $P = (p_x, p_y, p_z)$ znajdujący się w bryle otaczającej całą scenę.
4. Znajdź liść *drzewa BSP*, który zawiera punkt P. W tym celu przejdź drzewo od korzenia, na każdym kroku kierując się do podwęzła, który zawiera P, aż do napotkania liścia.
5. Przejdź do sektora skojarzonego ze znalezionym węzłem *drzewa BSP*. Upewnij się, że punkt leży wewnątrz bryły wypukłej reprezentowanej przez ten sektor poprzez sprawdzenie, czy P leży po pozytywnej stronie płaszczyzn zawartych we wszystkich trójkątach zgromadzonych w sektorze. W przypadku pozytywnej odpowiedzi zwiększ skojarzony z sektorem licznik trafień o 1. Niezależnie od tego zwiększ globalny licznik trafień o 1.

Ostatecznie objętość każdego z sektorów szacuje poniższe wyrażenie:

$$V(Sektor\ S) = \frac{objetosc\ bryly\ otaczajacej * licznik\ trafien(S)}{globalny\ licznik\ trafien} \tag{3.8}$$

Przedstawiona metoda jest bardzo szybka, gdyż znalezienie sektora który zawiera wylosowany punkt wymaga przejścia ścieżki w drzewie od korzenia do liścia. Ponieważ drzewo jest bardzo dobrze zrównoważone, można założyć że koszt tej operacji jest logarytmiczny względem liczby liści. Dodatkowo w każdym sektorze znajduje się niewielka liczba trójkątów, zazwyczaj 1 lub 2. Dla scen składających się z wielu tysięcy elementów możliwe jest sprawdzenie kilkunastu milionów punktów w czasie kilku sekund.

3.4.2.3. Pseudokod algorytmu.

```
połącz-sektory( kolekcja sektorów S )
    posortuj S po objętości sektorów, rosnąco

    /* Łączenie sektorów o zerowej objętości */
    dopóki objętość pierwszego sektora z S ≠ 0 powtarzaj
        s ← pierwszy sektor z S
        s' ← sąsiad s o największej objętości
        usuń s z kolekcji S
        jeśli objetość(s')> 0 to
            połącz s' z s
        posortuj S po objętości sektorów, rosnąco

    /* Łączenie pozostałych sektorów */
    połączono ← prawda
    dopóki polaczono = prawda powtarzaj
        połączono ← fałsz
        i ← 0
        dopóki połączono = fałsz i i < liczba-elementów(S) powtarzaj
            s ← i-ty element z kolekcji S
            s' ← pusty
            najepsza wypukłość ← 0

            /* Szukanie sektorów do połączenia: s i jeden z jego sąsiadów */
            dla każdego sektora s'' sąsiadującego z s wykonaj
                bieżąca wypukłość ← współczynnik-wypukłości(s połączony-z s'')
                jeśli objętość(s)+objętość(s'') < maksymalna objętość i
                    bieżąca wypukłość > minimalna wypukłość i
                    (s' jest pusty lub bieżąca wypukłość > najlepsza wypukłość) to
                        s' ← s''
                            najlepsza wypukłość ← bieżąca wypukłość
            /* Połączenie znalezionej pary sektorów */
            jeśli s' jest niepusty to
                połączono ← prawda
                usuń s z kolekcji S
                połącz s z s'
                posortuj S po objętości sektorów, rosnąco
            w przeciwnym wypadku
                i ← i + 1
```

Algorytm w czasie działania pracuje na kolekcji sektorów, posortowanej rosnąco po objętości. W pierwszej kolejności wybiera się sektory o zerowej objętości i próbuje się je połączyć z ich sąsiadami o niezerowej objętości. Jeśli jakiś z przetwarzanych sektorów nie posiada sąsiadów, bądź też mają one zerową objętość, taki sektor usuwa się z przetwarzanej kolekcji.

Należy zastanowić się nad sytuacją, w której sektor posiada zerową objętość. Może ona wynikać z następujących powodów:

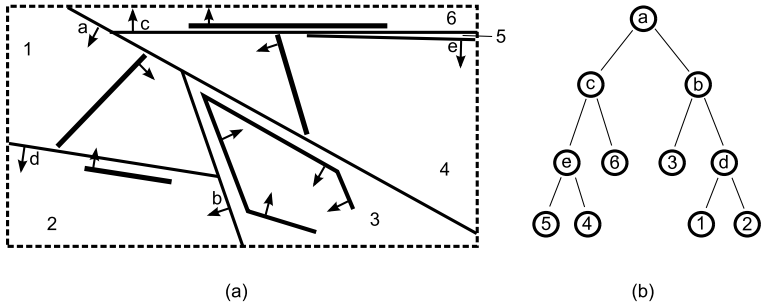
- Skojarzony z sektorem węzeł *drzewa BSP* ma zerową objętość. Taka sytuacja może być spowodowana nieprawidłową konstrukcją *drzewa BSP*, w której pojawiają się sektory o zerowej objętości, istnieniem tak małych sektorów, że w procedurze szacowania objętości żaden z dystrybuowanych punktów nie trafił do ich wnętrza, bądź też nieprawidłowej dystrybucji punktów losowych.

- Przycięcie wielościanu reprezentowanego przez węzeł *drzewa BSP* ścianami bryły otaczającej sprawia, że objętość jest równa zeru. Najczęstszą przyczyną takiego stanu jest sytuacja, w której sektor zawiera fragment zewnętrznej geometrii modelu 3D, zawierającej się w jednej ze ścian bryły otaczającej.

- Geometria sektora w przecięciu z węzłem *drzewa BSP* powoduje otrzymanie bryły o (prawie) zerowej objętości. Najczęściej spotykaną tego sytuacją jest jeden trójkąt leżący równolegle do jednej z płaszczyzn ograniczających sektor widoczną stroną skierowaną w jej kierunku, w bardzo niewielkiej odległości. Ze względu na sposób dystrybucji trójkątów leżących w płaszczyźnie podziału nie jest możliwe, aby taki trójkąt leżał w płaszczyźnie podziału.

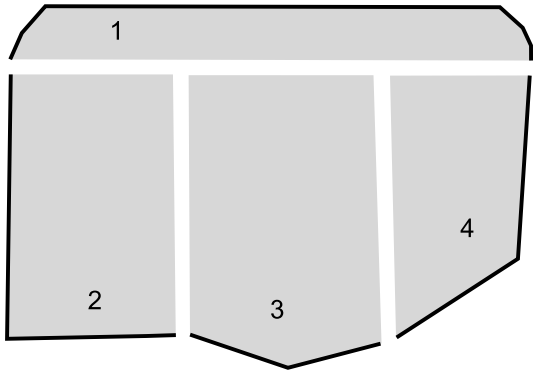
W praktyce pierwszy z wymienionych scenariuszy nie występuje. Większość sektorów o zerowej objętości posiada sąsiadów z którymi można je połączyć. Pozostaje niewielka liczba sektorów, która reprezentuje ściany skierowane na zewnątrz sceny, a więc niewidocznych z żadnego innego miejsca dany sektor.

W drugiej kolejności algorytm powtarza procedurę szukania pary sektorów do połączenia tak długo, aż nie będzie możliwe znalezienie takiej pary. Oznacza to zakończenie algorytmu. Procedura szukania pary sektorów przegląda wszystkie sektory z kolekcji S zaczynając od jej początku, czyli od sektorów o najmniejszej objętości. Dla każdego z nich sprawdza się wszystkich jego sąsiadów jako kandydatów do połączenia z nim samym. Jeśli połączenie okaże się to się możliwe, dokonuje się połączenia obydwu sektorów w jeden, uaktualnia kolekcję S i zakańcza procedurę.



Rysunek 3.23: **(a)** Przykład drzewa BSP i sytuacji, w której pojawiają się sektory o zerowej objętości. Strzałkami zaznaczono orientację płaszczyzn podziału i widoczną stronę trójkątów. Sektor *2* zawiera trójkąt, który jest zwrócony w kierunku płaszczyzny *d* o odległość zbyt dużą aby został uznany za leżący na niej i umieszczony w sektorze *1*, jednocześnie zbyt małą aby którykolwiek z wylosowanych punktów trafił w ten rejon. Sektor *5*, rozpięty na płaszczyznach *c* oraz *e* jest zbyt mały by możliwe stało się trafienie w niego punktem. Sektor *6* przycięty z góry bryłą otaczającą scenę staje się sektorem o zerowej objętości. Objętość pozostałych sektorów jest niezerowa. **(b)** Drzewo BSP z zaznaczonymi węzłami i liściami.

Wyjaśnienia wymaga dlaczego w każdej iteracji przegląda się kolekcję S od początku, pomimo że poprzednio stwierdzono już, że nie jest możliwe ich połączenie. Dokonanie jakiegokolwiek połączenia sektorów może sprawić, że wcześniejsze nieudane próby złączenia zakończą się wreszcie sukcesem, co ilustruje rysunek 3.24.



Rysunek 3.24: Sektor 1 nie może być bezpośrednio połączony z żadnym ze swoich sąsiadów, gdyż za każdym razem otrzymuje się za niski współczynnik wypukłości. Może być połączony tylko do sektora będącego połączeniem sektorów 2, 3 i 4.

Osobnym zagadnieniem jest stwierdzenie, że daną parę sektorów warto ze sobą połączyć. Ideą algorytmu *Portal rendering* jest wykorzystanie podziału sceny na obiekty (sektory), w których zasłanianie nie występuje (bądź występuje w bardzo ograniczonym zakresie), oraz relacji widoczności pomiędzy nimi w postaci portali. W tym kontekście sektory są atomowymi obiektami, które w zależności od pozycji i orientacji kamery są klasyfikowane jako widoczne lub nie. Dlatego także ważne jest limitowanie wielkości pojedynczego sektora (w sensie fizycznych rozmiarów), aby poza redukcją fragmentów zasłoniętych uzyskiwać jednocześnie redukcję fragmentów niewidocznych, leżących poza bryłą widzenia. Jednakże duża ilość małych sektorów niechybnie doprowadzi do tego, że algorytm *Portal rendering* stanie się wąskim gardłem całego potoku renderowania grafiki $3D$.

Prezentowany algorytm łączy sektory parami. W praktyce okazało się to całkowicie wystarczające, mimo że można sobie wyobrazić taką dekompozycję początkową, którą niemożliwe jest zredukować w ten sposób. W razie gdyby okazało się to konieczne, możliwe jest rozszerzenie w którym rozpatruje się łączenie sektorów trójkami, czwórkami a nawet większymi grupami. Przyjęto, że sektory można połączyć po spełnieniu poniższych warunków:

- Łączone sektory muszą ze sobą sąsiadować. To znaczy, że istnieje portal, który styka się z jednym i drugim.
- Wielkość łączonych sektorów, wyrażona w objętości nie może być większa niż ustalony próg. W praktyce najlepiej posługiwać się progiem zdefiniowanym jako procent objętości zajmowanej przez całą scenę. Rozsądną wartością, sprawdzającą się w większości przypadków jest kilka procent.
- Wzajemne przesłanianie w połączonym sektorze występuje w bardzo ograniczonym zakresie. Warunek ten wyraża się w tym, że *współczynnik wypukłości* połączonej pary sektorów jest nie mniejszy niż ustalona wartość. Aby precyzyjniej zbadać to połączenie można posłużyć się takim pomiarem, w którym początki badanych odcinków leżą w jednym sektorze zaś końce w drugim.

Należy także zdefiniować czym jest połączenie dwóch lub większej ilości sektorów w jedną grupę. Na tym etapie wystarczające jest, aby sektor miał wiedzę o tym, jakie jeszcze sektory są z nim połączone. Nie jest wymagane, a wręcz jest niewskazane aby usuwać portale portale pomiędzy sektorami z tej samej grupy.

3.4.2.4. Wyznaczanie współczynnika wypukłości.

Korzystając z metody *Monte Carlo* łatwo można wyznaczyć *współczynnik wypukłości* dla grupy sektorów, stosując się wprost do definicji: losując pary punktów

i badając widoczność pomiędzy nimi. Stosunek liczby par punktów wzajemnie widocznych do liczby wszystkich przebadanych par jest wtedy dobrą aproksymacją szukanego współczynnika.

Pierwszym problemem jest umiejętność wylosowania punktu leżącego wewnątrz grupy sektorów. Najłatwiej pokonać go losując jeden z sektorów z prawdopodobieństwem zależnym od jego objętości (objętości sektorów zostały uprzednio wyznaczone), następnie wylosować punkt leżący wewnątrz wskazanego sektora. Ponieważ jest to trudne do uzyskania, spamiętuje w sektorze pewną liczbę punktów, które trafiły do niego w trakcie procedury wyznaczania objętości.

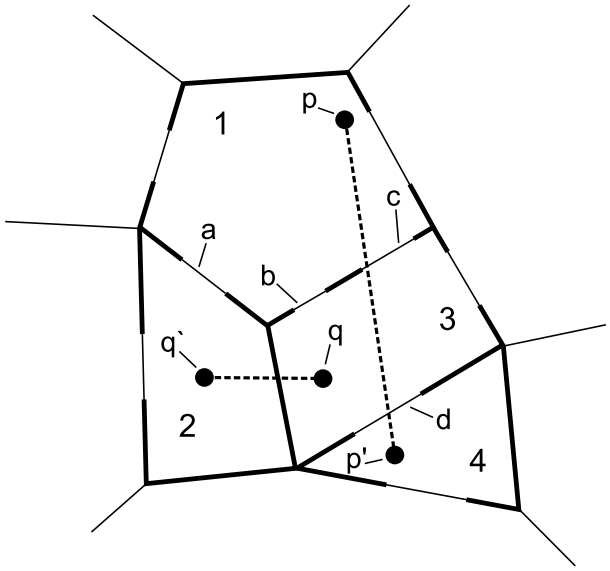
Drugim problemem jest jest odpowiedź na pytanie, czy wylosowana para punktów jest nawzajem widoczna. Możliwe są dwa podejścia do tego problemu:

- Próba potwierdzenia widoczności. Z powodzeniem można zastosować algorytm, który poszukuje sekwencji portali propagujących widoczność z jednego punktu do drugiego. Poniżej zamieszczono jego schemat:
  - Znajdź sektor w którym znajduje się pierwszy punkt z pary.
  - Przejdź *podgraf CPG* ograniczony do sektorów z rozpatrywanej podgrupy. Przechodź tylko przez te krawędzie, czyli w tym przypadku portale, które zawierają punkt będący rzutem na swoją płaszczyznę odcinka łączącego badaną parę punktów.
  - Jeśli uda osiągnąć się sektor w którym leży drugi punkt z pary, to oba punkty uznaje się za widoczne.
- Próba obalenie widoczności. Polega na poszukiwaniu trójkąta, który zasłania widoczność pomiędzy oboma punktami. Jeśli oba punkty traktować jako początek i koniec promienia światła, to zagadnienie to ma efektywne rozwiązanie w postaci znajdowania przecięć promieni z obiektami z użyciem *drzewa BSP*.

Teoretycznie obydwie metody są sobie równoważne. W praktyce rozwiązanie polegające na szukaniu przecięcia promienia jest lepsze, z dwóch podstawowych powodów:

- Jest znacznie szybsze. Przewaga uwidacznia się wraz ze wzrostem liczby połączeń sektorów.
- Może być dokładniejsze. O ile potencjalna konserwatywność w wyznaczaniu portali jest akceptowalna gdyż nie ma negatywnego wpływu na renderowanie sceny, o tyle w tym kontekście może powodować zawyżanie *współczynnika wypukłości*, przez uznawanie pewnych niewidocznych

w rzeczywistości par punktów za widoczne. Sytuacja ta nie ma miejsca, kiedy skrupulatnie zastosowana została eliminacja nadmiarowych portali bądź ich części, co jak wcześniej zostało powiedziane będzie mieć miejsce dopiero po zakończeniu etapu łączenia sektorów.



Rysunek 3.25: Przykład zbioru połączonych sektorów 1, 2, 3, 4. Punkty p i p' są widoczne przez sekwencję portali c, d. Punkty q i q' są niewidoczne, gdyż istnieje ściana blokująca widoczność, z drugiej zaś strony nie istnieje sekwencja portali propagująca między nimi widoczność.

3.5. Przebudowa portali.

Operacja przebudowy portali jest ostatnim krokiem automatycznej dekompozycji na *portale i sektory* i ma na celu uproszczenie struktury połączeń między sektorami. Rozpatruje się ją w kilku aspektach:

- Eliminacja portali pomiędzy sektorami z tej samej grupy. Połączenie sektorów w grupę oznacza bezwarunkową widoczność między nimi. Portale wyrażające widoczność między sektorami z grupy są redundantne i mogą zostać usunięte.
- Eliminacja nakładających się portali. Mechanizm klasyfikacji portali do sektorów może przypinać do jednego sektora wiele współpłaszczyznowych portali, których części wspólne nie są puste. Problem omówiony został w rozdziale 3.3.3.3, gdzie przedstawiono jego rozwiązanie.

- Uproszczenie portali. Jest to analogiczny do łączenia sektorów problem, rozpatrywany z punktu widzenia portali. Może okazać się, że widoczność reprezentowana jest przez bardzo dużą liczbę portali, co rzutuje na efektywność algorytmu *Portal rendering*. W podstawowej wersji dotyczy on zbioru portali leżących na tej samej płaszczyźnie. Celem jest dokonanie jego konserwatywnej aproksymacji zbiorem o mniejszej złożoności, liczonej w liczbie portali jak i liczby wierzchołków poszczególnych wielokątów.

Rozdział 4

Wyniki działania algorytmu

Wszystkie algorytmy omówione w rozdziale trzecim, a więc budowa *drzewa BSP*, ekstrakcja *portali i sektorów* oraz upraszczanie ich struktury zostały zaimplementowane w języku C++ w środowisku Microsoft Visual C++ 2003[1]. Jako danych opisujących scenę 3D użyto formatu X-File z pakietu DirectX[2], do którego wczytywania użyto parsera z pakietu Microsoft DirectX SDK[3]. Jest to popularny format opisu scen 3D, do którego istnieją konwertery z wielu innych formatów opisujących grafikę 3D, bądź też eksportery dostępne w komercyjnych edytorach grafiki trójwymiarowej. Do wizualizacji sceny 3D został użyty moduł Direct3D[4] z pakietu DirectX, ponadto skorzystano z wielu algorytmów i struktur danych z bibliotek STL[5] oraz Boost[6].

4.1. Dane testowe.

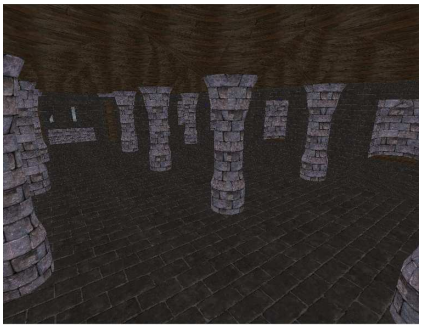
Jako danych testowych użyto scen dołączonych do edytora DeleD 3D Editor[7], prezentujące lokacje zamknięte o różnym stopniu złożoności, które wyeksportowano do formatu X-Object korzystając ze wspomnianego edytora. Poniżej przedstawiono ich krótką charakterystykę:

[1] http://en.wikipedia.org/wiki/Microsoft_Visual_Studio
[2] http://msdn.microsoft.com/en-us/library/bb173011(VS.85).aspx
[3] http://msdn.microsoft.com/en-us/directx/aa937788.aspx
[4] http://en.wikipedia.org/wiki/Direct3D
[5] http://www.sgi.com/tech/stl/
[6] http://www.boost.org/
[7] http://www.delgine.com

Nazwa pluku	barricade.x
Liczba wierzchołków	2520
Liczba trójkątów	1308
Liczba materiałów	18
Rozmiary sceny	2432 x 904 x 2208

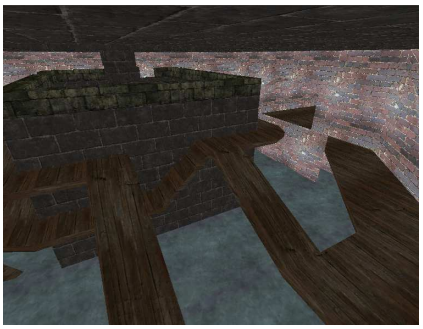
Nazwa pliku	castle.x
Liczba wierzchołków	1522
Liczba trójkątów	752
Liczba materiałów	24
Rozmiary sceny	1792 x 1288 x 1920

Nazwa pliku	chapel.x
Liczba wierzchołków	4564
Liczba trójkątów	2436
Liczba materiałów	19
Rozmiary sceny	2560 x 1680 x 3006



Nazwa pliku	sacredplace.x
Liczba wierzchołków	19604
Liczba trójkątów	9552
Liczba materiałów	100
Rozmiary sceny	2154 x 1230 x 5115

Nazwa pliku	scene.x
Liczba wierzchołków	24018
Liczba trójkątów	12714
Liczba materiałów	103
Rozmiary sceny	2051 x 800 x 2561



Nazwa pliku	waterworld.x
Liczba wierzchołków	1840
Liczba trójkątów	934
Liczba materiałów	67
Rozmiary sceny	2496 x 1041 x 3104

4.2. Budowa drzewa BSP.

Algorytm budowy *drzewa BSP* opiera się na schemacie opisanym w punkcie 3.2.3.2 wraz ze wszystkimi rozszerzeniami polegającymi na:

- Zakańczaniu rekursji w momencie otrzymania zbioru wypukłego.
- Posługiwaniu się rozszerzonym wachlarzem płaszczyzn rozdzielających.
- Minimalizacji podziałów trójkątów.

Poniżej omówiono każdy z powyższych aspektów:

4.2.1. Testowanie wypukłości zbioru.

W punkcie 3.2.3.3 omówiono kryterium zakańczania rekurencyjnych wywołań algorytmu budowy *drzewa BSP*, którym jest otrzymanie wypukłego zbioru trójkątów. Wspomniano o pesymistycznym, kwadratowym czasie wykonania tego testu względem liczby trójkątów i o tym, że w średnim przypadku jest on znacznie niższy. Dla każdej ze scen zmierzono średni rozmiar testowanego zbioru, średnią liczbę wywołań dominującej czasowo operacji (obliczenie odległości punktu od prostej) oraz sumaryczny czas spędzony w procedurze badania wypukłości zbioru. Wyniki przedstawia poniższa tabelka:

Scena	Wywołań	rozmiar zbioru	liczba testów	czas	czas całkowity
Barricade	1912	9.65	61.79	0.0066	266.8
Castle	1077	9.40	56.58	0.0036	177.2
Waterworld	1243	9.96	84.34	0.0047	204.5
Chapel	2588	11.8	56.27	0.012	347.8

Tabela 4.1: Rezultaty budowy drzewa BSP dla sceny Barricade. Limit testowanych trójkątów w jednym kroku rekursji: 1000, próg użycia heurystyki z punktu czwartego: w pierwszej serii 20, w drugiej 0 trójkątów.

Jak widać sumaryczny czas wszystkich wywołań procedury sprawdzania wypukłości jest znikomo mały w stosunku do całkowitego czasu spędzonego w procedurze budowy *drzewa BSP*. Jest to spowodowane faktem, że procedura sprawdzania wypukłości jest wołana tyle razy ile jest węzłów w drzewie, zaś w każdym wywołaniu wykonuje się średnio tylko kilkadziesiąt obliczeń.

4.2.2. Generowanie płaszczyzn podziału.

W procesie budowy poddrzewa BSP z danego zbioru trójkątów sprawdza się następujące płaszczyzny podziału:

1. Zawierające się w trójkątach sceny.
2. Prostopadłe do płaszczyzn trójkątów, zawierające się w ich krawędziach.
3. Zawierające się w wierzchołkach trójkątów, ortogonalne do osi układu współrzędnych.
4. Rozpięte na wszystkich podzbiorach trójelementowych zbioru wierzchołków wszystkich trójkątów (warunkowo, przy odpowiednio małym zbiorze trójkątów).

Dodatkowo wprowadza się limit ilości trójkątów użytych do generowania płaszczyzn w przypadkach 1-3, oraz granica wielkości zbioru trójkątów, powyżej której test 4 nie będzie wykonywany, ze względu na swój wysoki, sześcienny koszt względem liczby trójkątów.

W pierwszym etapie przeanalizowano wpływ metod 1-3 na proces tworzenia *drzewa BSP*, jako że należą one do jednej grupy heurystyk tworzenia płaszczyzn podziału na podstawie wskazanego trójkąta. Dodatkowo przebadano przypadek stosowania bądź nie stosowania heurystyki opisanej w punkcie 4. Wyniki przedstawiono w tabelach 4.2, 4.3, 4.4.

Heurystyki	Czas	Liści w drzewie	Podziałów trójkątów
1+2+3	123	973	537
1+2	54	1027	610
1+3	119	1066	573
2+3	120	1056	573
1	36	1048	610
2	51	1077	566
3	61	1090	534
1+2+3	124	1151	880
1+2	34	1195	1053
1+3	100	1311	1560
2+3	119	1212	960
1	46	1154	2016
2	35	1368	1195
3	110	2247	3333

Tabela 4.2: Rezultaty budowy drzewa BSP dla sceny Barricade. Limit testowanych trójkątów w jednym kroku rekursji: 1000, próg użycia heurystyki z punktu czwartego: w pierwszej serii 20, w drugiej 0 trójkątów.

Heurystyki	Czas	Liści w drzewie	Podziałów trójkątów
1+2+3	84	617	490
1+2	29	620	508
1+3	74	623	512
2+3	79	642	475
1	32	624	522
2	37	647	442
3	77	768	679
1+2+3	67	660	629
1+2	25	728	736
1+3	60	648	757
2+3	67	675	575
1	27	723	1701
2	28	841	878
3	63	998	1432

Tabela 4.3: Rezultaty budowy drzewa BSP dla sceny Waterworld. Limit testowanych trójkątów w jednym kroku rekursji: 1000, próg użycia heurystyki z punktu czwartego: w pierwszej serii 20, w drugiej 0 trójkątów.

W przypadku stosowania heurystyki opisanej w punkcie 4 okazuje się, że równoczesne użycie trzech pierwszych heurystyk prowadzi do uzyskania najlepszych rezultatów, zarówno pod kątem minimalizacji liczby liści w drzewie, sumarycznej liczby podziałów trójkątów i wysokości drzewa, ale jest najbardziej kosztowne czasowo. Użycie dowolnych dwóch heurystyk prowadzi do nieznacznie gorszych rezultatów. Zaskakujące jest, że użycie tylko jednej heurystyki nie powoduje drastycznego pogorszenia wyników. Sytuacja zmienia się diametralnie, gdy czwarta heurystyka nie jest stosowana. W takiej sytuacji stosowanie tylko pierwszej bądź trzeciej prowadzi do bardzo złych rezultatów. W samodzielnej pracy najlepiej sprawdza się druga heurystyka, charakteryzująca się przy tym niskim kosztem czasowym. Najlepiej wypadają wszystkie trzy heurystyki stosowane jednocześnie, zaś pod względem minimalizacji czasu wykonania najlepiej wypada jednoczesne stosowanie pierwszej i drugiej.

W następnej kolejności przetestowano wpływ limitu testowanych trójkątów w jednym kroku rekursji na budowę drzewa. Dla każdego trójkąta użyto wszystkich trzech heurystyk generowania płaszczyzn, wyniki zawierają tabele 4.5, 4.7, 4.7.

Oczywistą obserwacją jest wzrost czasu obliczeń przy zwiększaniu liczby testowanych trójkątów. Obserwuje się poprawę jakości drzewa, co jest najlepiej widoczne w przypadku najbardziej złożonej sceny SacredPlace. Podobnie wypadają testy na scenie Waterworld. Zaskakująco nierówne rezultaty otrzymane w przypadku pierwszej sceny, która przedstawia scenerię zewnętrzną, stosunkowo mało zasłoniętą.

Ostatnia z analiz dotyczy parametru maksymalnej liczby trójkątów, przy której wyznacza się zbiór wszystkich ich wierzchołków a następnie testuje płaszczyzny rozpięte na każdych trzech różnych punktach z tego zbioru.

4.3. Podział na portale i sektory.

4.4. Łączenie sektorów i portali.

4.5. Weryfikacja poprawności.

4.6. Podsumowanie.

Heurystyki	Czas	Liści w drzewie	Podziałów trójkątów	Wysokość drzewa
1+2+3	641	6866	3843	25
1+2	502	7431	4061	25
1+3	442	6808	4646	27
2+3	464	7271	4332	67
1	157	8002	6733	66
2	173	8032	4621	41
3	433	8028	5607	27
1+2+3	512	7609	5605	33
1+2	192	8428	7006	35
1+3	530	9004	11418	38
2+3	550	8709	6781	33
1	2135	12089	27379	45
2	242	10592	9716	34
3	650	17844	26747	38

Tabela 4.4: Rezultaty budowy drzewa BSP dla sceny Barricade. Limit testowanych trójkątów w jednym kroku rekursji: 200, próg użycia heurystyki z punktu czwartego: w pierwszej serii 12, w drugiej 0 trójkątów.

Limit	Czas	Liści w drzewie	Podziałów trójkątów	Wysokość drzewa
5	38	1008	635	21
15	39	1009	626	21
40	44	972	511	20
150	68	1026	570	23
500	100	960	536	21
1500	128	1051	610	20

Tabela 4.5: Rezultaty budowy drzewa BSP dla sceny Barricade dla różnych limitów testowanych trójkątów w jednym kroku rekursji. Próg testowania płaszczyzn rozpiętych pomiędzy wszystkimi wierzchołkami: 20 trójkątów.

Limit	Czas	Liści w drzewie	Podziałów trójkątów	Wysokość drzewa
5	37	724	1669	21
15	30	601	556	20
40	37	646	549	20
150	46	624	509	22
500	69	616	489	21
1000	78	608	474	22

Tabela 4.6: Rezultaty budowy drzewa BSP dla sceny Waterworld dla różnych limitów testowanych trójkątów w jednym kroku rekursji. Próg testowania płaszczyzn rozpiętych pomiędzy wszystkimi wierzchołkami: 20 trójkątów.

Limit	Czas	Liści w drzewie	Podziałów trójkątów	Wysokość drzewa
5	130	7572	5323	27
15	151	6957	4102	27
40	209	7038	4303	31
150	408	6865	3962	26
500	914	6781	3784	27
3000	3748	6815	3894	27

Tabela 4.7: Rezultaty budowy drzewa BSP dla sceny Barricade dla różnych limitów testowanych trójkątów w jednym kroku rekursji. Próg testowania płaszczyzn rozpiętych pomiędzy wszystkimi wierzchołkami: 12 trójkątów.

Limit	Czas	Liści w drzewie	Podziałów trójkątów	Wysokość drzewa
0	57	1150	870	24
5	72	1101	792	23
12	67	999	605	24
24	83	1031	542	20
60	254	987	444	20

Tabela 4.8: Rezultaty budowy drzewa BSP dla sceny Barricade dla różnych limitów testowanych trójkątów w jednym kroku rekursji. Próg : 200 trójkątów.

Limit	Czas	Liści w drzewie	Podziałów trójkątów	Wysokość drzewa
0	39	680	675	22
5	46	645	583	22
12	51	629	545	21
24	61	622	521	23
60	191	570	381	23

Tabela 4.9: Rezultaty budowy drzewa BSP dla sceny WaterWorld dla różnych limitów testowanych trójkątów w jednym kroku rekursji. Próg : 200 trójkątów.

Limit	Czas	Liści w drzewie	Podziałów trójkątów	Wysokość drzewa
0	404	7870	6132	27
5	430	7454	5191	38
12	436	6860	3805	28
24	598	6696	3153	24
60	2676	6506	2809	27

Tabela 4.10: Rezultaty budowy drzewa BSP dla sceny Sacredplace dla różnych limitów testowanych trójkątów w jednym kroku rekursji. Próg : 200 trójkątów.

Bibliografia

[Abr97] Michael Abrash. *Graphics Programming Black Book*. Keitb Weiskamp, 1997. [cytowanie na str. 13, 15, 16, 18, 39, 40]

[ARFPB90] John M. Airey, John H. Rohlf, and Jr. Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 41–50, New York, NY, USA, 1990. ACM. [cytowanie na str. 13, 16, 17, 18]

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. [cytowanie na str. 42]

[Bit02] Jiří Bittner. Hierarchical techniques for visibility computations, 2002. [cytowanie na str. 4, 17]

[Cat74] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974. [cytowanie na str. 8]

[dBCvKO97] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, first edition edition, 1997. [cytowanie na str. 39]

[Dur99] Frédo Durand. *3D Visibility: analytical study and applications*. PhD thesis, Université Joseph Fourier, Grenoble I, July 1999. [cytowanie na str. 17]

[FAG83] Henry Fuchs, Gregory D. Abram, and Eric D. Grant. Near real-time shaded display of rigid objects. *SIGGRAPH Comput. Graph.*, 17(3):65–72, 1983. [cytowanie na str. 39, 42]

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM. [cytowanie na str. 6, 10, 11, 40, 46]

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238, New York, NY, USA, 1993. ACM. [cytowanie na str. 31]

[Gor02] Dan Gordon. The floating column algorithm for shaded, parallel display of function surfaces without patches. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):76–91, 2002. [cytowanie na str. 28, 29]

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987. [cytowanie na str. 43]

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. [cytowanie na str. 42, 43, 46]

[HHS06] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. On the fast construction of spatial data structures for ray tracing. pages 71–80, Sep 2006. [cytowanie na str. 43]

[HMK+97] Lichan Hong, Shigeru Muraki, Arie Kaufman, Dirk Bartz, and Taosong He. Virtual voyage: Interactive navigation in the human colon. *Computer Graphics*, 31(Annual Conference Series):27–34, 1997. [cytowanie na str. 24]

[JD97] A. James and A. M. Day. The priority face determination tree. In *Eurographics UK Proceedings*, 1997. [cytowanie na str. 39]

[Jon71] Cliff B. Jones. A new approach to the 'hidden line' problem. *Comput. J.*, 14(3):232–237, 1971. [cytowanie na str. 20]

[KCCO00] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Virtual occluders: An efficient intermediate pvs representation, 2000. [cytowanie na str. 32]

[Kir92] David Kirk, editor. *Graphics Gems III*. Academic Press Professional, Inc., San Diego, CA, USA, 1992. [cytowanie na str. 55]

[LG95] David Luebke and Chris Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 105–ff., New York, NY, USA, 1995. ACM. [cytowanie na str. 20, 23]

[LH03] Sylvain Lefebvre and Samuel Hornus. Automatic cell-and-portal decomposition. Technical Report 4898, INRIA, July 2003. [cytowanie na str. 66, 68]

[MBWW07] Oliver Mattausch, Jiří Bittner, Peter Wonka, and Michael Wimmer. Optimized subdivisions for preprocessed visibility. In *GI '07: Proceedings of Graphics Interface 2007*, pages 335–342, New York, NY, USA, 2007. ACM. [cytowanie na str. 42, 67]

[NAT90] Bruce Naylor, John Amanatides, and William Thibault. Merging bsp trees yields polyhedral set operations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 115–124, New York, NY, USA, 1990. ACM. [cytowanie na str. 6, 39]

[Nay92] Bruce F. Naylor. Partitioning tree image representation and generation from 3D geometric models. pages 201–212, 1992. [cytowanie na str. 10, 39]

[NNS72] M. E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. In *ACM'72: Proceedings of the ACM annual conference*, pages 443–450, New York, NY, USA, 1972. ACM. [cytowanie na str. 9]

[Ost03] Nathan Ostgard. `http://www.devmaster.net/articles/quake3collision/`, 2003. [cytowanie na str. 10]

[PD90] Harry Plantinga and Charles R. Dyer. Visibility, occlusion, and the aspect graph. *Int. J. Comput. Vision*, 5(2):137–160, 1990. [cytowanie na str. 6]

[RE01] Samuel Ranta-Eskola. Binary space partitioning trees and polygon removal in real time 3d rendering. Uppsala Master Theses in Computing Science, 2001. [cytowanie na str. 11, 15, 18]

[SS92] Kelvin Sung and Peter Shirley. Ray tracing with the bsp tree. pages 271–274, 1992. [cytowanie na str. 10]

[Sze97] F. Szenberg. An algorithm for visualization of a terrain with objects, 1997. [cytowanie na str. 28, 29]

[Tel92] Seth J Teller. Visibility computations in densely occluded polyhedral environments. Technical report, Berkeley, CA, USA, 1992. [cytowanie na str. 15, 17, 20, 23]

[TS91] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *SIGGRAPH Comput. Graph.*, 25(4):61–70, 1991. [cytowanie na str. 17]

[vdPS99] Michiel van de Panne and A. James Stewart. Effective compression techniques for precomputed visibility. In *Eurographics Workshop on Rendering*, pages 305–316, June 1999. [cytowanie na str. 18]

[Win99] Andrew Steven Winter. An investigation into real-time 3d polygon rendering using bsp trees. Project Dissertation submitted to the University of Wales Swansea in Partial Fulfilment for the Degree of Bachelor of Science, April 1999. [cytowanie na str. 39]

[WWS00] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs, 2000. [cytowanie na str. 15, 33]

[YR95] Roni Yagel and Wiliam Ray. Visibility computation for efficient walkthrough of complex environments. Teleoperators and Virtual Environments, Vol. 5, No. 1, 1995. [cytowanie na str. 15]