Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Radosław Głogowski

Agresywne wyznaczanie widocznych obiektów

Praca magisterska

Promotor
dr Andrzej Łukaszewski

Wrocław 2009

Spis treści

1 Wstęp

Celem pracy była implementacja i testowanie algorytmu wyznaczania widocznych obiektów. Spośród wielu istniejących rozwiązań wybrano algorytm GVS Wonki [3] z 2006 roku. Został on poddany licznym modyfikacjom i rozszerzeniom mających wpływ na zmniejszenie czasu działania. Jako przykładowe zastosowanie wybrano przyspieszenie generowania obrazów metodą śledzenia promieni. Sprawdzono czas renderowania w scenach mających ponad milion elementów, a poprzez redukowanie ich rozmiarów poruszono problem przechodzenia scen w czasie rzeczywistym.

Dla potrzeb wyznaczania kolejnych zbiorów widocznych obiektów opracowano algorytm aktualizujący, który bazując na istniejącym rozwiązaniu wyznacza widoczne elementy z sąsiedniego obszaru. W odróżnieniu od metod badających widoczność dla całej sceny np. Chhugani [10] i Bittner [11] jest to metoda przyspieszająca obliczenia bez używania dodatkowej pamięci. Idea algorytmu jest bliska rozwiązaniu wyświetlania dużych miast Koltuna [4], jednak nie wymaga ona założeń co do typu sceny.

W drugim rozdziale zostało opisane wyznaczanie widocznych obiektów wraz z wybranymi algorytmami. W rozdziale trzecim opisano algorytm GVS będący agresywnym wyznaczaniem widocznych obiektów. Rozdział czwarty i piąty zawiera własne pomysły związane z problematyką wyboru widocznych obiektów. Przedstawiono w nich rozszerzenia i modyfikacje algorytmu GVS. W piątym rozdziale został opisany przykład zastosowania algorytmu widoczności oraz metoda aktualizacji zbioru widocznych obiektów. W rozdziałach szóstym i siódmym zamieszczono wyniki i podsumowanie. Zaimplementowane algorytmy zostały opisane w dodatku A.

2 Problematyka i przegląd rozwiązań

Jednym z podstawowych problemów grafiki komputerowej jest określenie widocznych obiektów sceny. Rozwiązaniem jest wygenerowany przez kartę graficzną obraz zawierający widok z określonego punktu. Wyobrażając sobie w jaki sposób narysować scenę rozważamy metodę polegającą na sprawdzaniu widoczności kolejnych obiektów. Innym podejściem spotykanym również w malarstwie jest uwzględnienie najpierw dalekiego planu, następnie bliższych drzew, budynków itd. Przyczynia się to do częściowego zamalowania obszarów nakładając kolejne warstwy. Problem widoczności, czyli klasyfikacji obiektów na widoczne i zasłonięte, sprowadza się do wyznaczenia elementów, które *de facto* znajdą się na obrazie.

Problem widoczności rozszerza się o widok we wszystkich kierunkach. Poprzez odrzucenie niewidocznej geometrii sceny zmniejsza się zbiór obiektów, które należy uwzględniać przy rysowaniu. Obecne wiele rozwiązań skupia się na odrzuceniu części sceny schowanej pozostawiając wszystkie obiekty widoczne. Najpopularniejszy i powszechnie stosowany jest *Z-Buffer*, który ma dodatkowo wsparcie sprzętowe. Wyznacza on tylko widoczne punkty odrzucając każdy ukryty (zasłonięty).

Innymi technikami podczas określania widoczności są:

- uwzględnienie piramidy widzenia i odrzucanie geometrii sceny poza nią (ang. *view-frustum culling*)
- pomijanie trójkątów (płaszczyzn) odwróconych tyłem (ang. *back-face culling*)

- wyznaczanie zasłaniania się obiektów i obszarów sceny (ang. *occlusion culling*)

Z bardziej złożonym zadaniem znajdowania widocznych obiektów mamy do czynienia w przypadku niepunktowego obszaru obserwatora np. prostopadłościanu lub wycinka płaszczyzny. Obecnie jest to popularny problem, mający szerokie zastosowanie nie tylko w grafice komputerowej. O ile wyznaczanie dokładnie wszystkich widocznych elementów EVS (ang. exact visible set) jest zadaniem niezwykle złożonym, to istnieje wiele rozwiązań aproksymacyjnych wyznaczających zbiór potencjalnie widocznych obiektów PVS (ang. potentialy visible set) w krótszym czasie.

Algorytmy określania widocznej geometrii z obszaru zwykle klasyfikowane są jako:

- dokładne (ang. *exact*), których wynikiem jest EVS,
- agresywne (ang. *aggressive*) , wyznczające PVS, t.że, $PVS \subseteq EVS$,
- przybliżone (ang. *approximate*), $PVS \approx EVS$,
- konserwatywne (ang. *conservative*), $PVS \supseteq EVS$

Ograniczenie rozmiaru sceny do widocznych obiektów jest jedną z metod przyspieszania generowania obrazu. Inną równie popularną techniką jest uwzględnienie poziomu szczegółowości **LOD** ang. *level of detail*. Metody upraszczania geometrii wyznaczają aproksymacje obiektów sceny, następnie podczas generowania obrazu obiekty zastępowane są ich przybliżoną wersją.

2.1 Klasyfikacja scen

W literaturze traktującej o wyznaczaniu widocznej geometrii spotyka się klasyfikację względem wymiaru sceny:

- sceny 2D reprezentujące np. piętra wieżowców, pomieszczenia. Sceny takie można utożsamiać z planami budynków na płaszczyźnie. Wysokość ścian określona jest przez płaszczyzny sufitu i podłogi.
- mapa wysokości 2.5D - sceny będące rozszerzeniem sceny 2D o wysokość ścian. Często spotykanymi modelami 2.5D są miasta lub rzeźby terenu takie jak kaniony.
- scena 3D - model zawierający rozmieszczone dowolnie obiekty w przestrzeni trójwymiarowej (najczęściej są to trójkąty).

2.2 Geometria sceny widoczna z punktu

Techniki używane podczas szukania obiektów sceny widocznych z punktu są podstawowymi metodami grafiki komputerowej. Wśród nich znajdują się zarówno efektywne algorytmy wyznaczania zasłoniętych obszarów jak również odrzucanie elementów poza piramidą widzenia. Algorytmy te często znajdują zastosowanie w uogólnionym problemie wyznaczania widoczności z obszaru, a także w metodach śledzenia promieni. W algorytmie MLRTA [13] zastosowano odrzucanie wierzchołków kd-drzewa, które nie zawierają się w piramidzie widzenia.

Powszechnie wykorzystuje się hierarchiczne struktury danych typu kd-drzewa czy drzewa ósemkowe, dzięki którym widoczność, a właściwie zasłonięcie, określa się dla całych grup obiektów.

2.2.1 Selekcja obiektów w polu widzenia

Piramida widzenia (ang. *view frustum*) VF wyznacza przestrzeń widzianą przez obserwatora. Cztery płaszczyzny definiują ściany piramidy, które „przechodzą" przez krawędzie ekranu. Często stosuję się dodatkowe dwie płaszczyzny określające głębokość widzenia. Przestrzeń piramidy widzenia jest więc częścią wspólną odpowiednich półprzestrzeni, które zadane są przez płaszczyzny π_i:

$$\pi_i : n_i \cdot x + d_i = 0 \qquad i = 0, 1, ...5 \tag{1}$$

gdzie, n_i jest wektorem normalnym , d_i odległością , a x dowolnym punktem. Przyjmując półprzestrzeń wskazywaną przez wektor normalny za zewnętrzną (tj. obszar, z którego widoczny jest „przód") mówimy, że punkt x jest na zewnątrz płaszczyzny π_i, gdy $n_i \cdot x + d_i > 0$. Zatem x jest wewnątrz piramidy widzenia VF gdy jest po wewnętrznej stronie wszystkich płaszczyzn.

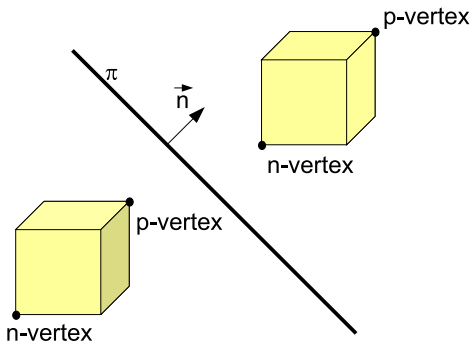
Odrzucanie obiektów z użyciem piramidy widzenia (ang. *view frustum culling*) polega na określeniu położenie obiektu względem każdej ze ścian. Elementy sceny poza piramidą są traktowane jako niewidoczne przez co są pomijane podczas renderowania. W celu przyspieszania powyższego testu wraz z obiektami pamiętane są punkty ekstremalne, z którymi utożsamiany jest prostopadłościan zamykający ów element (and. *bounding box*). Równie popularne jest wykorzystywanie otaczających sfer (ang. *bounding spheres*). Naiwny algorytm bazuje na sprawdzeniu wszystkich skrajnych wierzchołków „otoczki" względem piramidy. Relacja między prostopadłościanem BB a piramidą widzenia VF może być następująca:

- BB leży w środku VF, gdy $BB \cap VF = BB$
- BB na zewnątrz VF, gdy $BB \cap VF = \oslash$
- BB przecina VF, w przeciwnym przypadku,

W połączeniu z hierarchiczną strukturą np. kd-drzewa możemy sklasyfikować całe poddrzewo za widoczne lub nie.

Innym podejściem jest klasyfikacja wierzchołków względem kolejnych płaszczyzn VF. Jeśli są na zewnątrz płaszczyzny to prostopadłościan jest od razu określony jako niewidoczny. Taki algorytm prowadzi jednak do błędnych odpowiedzi, gdyż może klasyfikować niewidoczny prostopadłościan jako przecinający się z piramidą widzenia. Mimo tego podejście to zostało efektywnie zoptymalizowane przez Ulfa Assarssona i Tomasa Möllera [12]. Zastosowano prostopadłościany, których ściany są odpowiednio równoległe do płaszczyzn $x = 0$, $y = 0$, $z = 0$ (AABB, ang. *axis-aligned bounding box*). Ich reprezentacja sprowadza się do dwóch punktów ekstremalnych **bmin** i **bmax**. Idea algorytmu jest następująca:

- dla każdej płaszczyzny π_i $(i = 1, 2, 3, 4)$ określa **p-vertex** i **n-vertex** będące punktami ekstremalnymi (rysunek 1),
- wstawiając punkt do równania π_i klasyfikujemy jego położenie.



Rysunek 1: Określanie położenia *bounding boxa* względem płaszczyzny π z użyciem punktów ekstremalnych.

- współrzędne punktów $\mathbf{p\text{-}vertex}_j$ i $\mathbf{n\text{-}vertex}_j$, gdzie $j = x, y, z$ wyznaczane są następująco:

$$\begin{aligned} \mathbf{p\text{-}vertex}_j &= \begin{cases} \mathbf{bmax}_j & dla\ n_j > 0 \\ \mathbf{bmin}_j & wpp. \end{cases} \\ \mathbf{n\text{-}vertex}_j &= \begin{cases} \mathbf{bmin}_j & dla\ n_j > 0 \\ \mathbf{bmax}_j & wpp. \end{cases} \end{aligned} \tag{2}$$

- jeśli istnieje płaszczyzna, dla której $n_i \cdot \mathbf{n\text{-}vertex} + d_i > 0$, wtedy prostopadłościan jest na zewnątrz,
- jeśli dla każdej płaszczyzny $n_i \cdot \mathbf{p\text{-}vertex} + d_i < 0$ prostopadłościan jest wewnątrz piramidy widzenia,
- w pozostałych przypadkach algorytm klasyfikuje prostopadłościan jako przecinający piramidę widzenia.

Problem określania przecięcia prostopadłościanu z piramidą widzenia można rozwiązać poprzez uwzględnienie płaszczyzn wyznaczających *bounding box* i przecinania ich przez piramidę widzenia. Algorytm *„inverse view frustum culling"* został opisany w publikacji Reshetova [13], jako jedna z metod przyspieszenia śledzenia promieni. Idea algorytmu jest następująca:

- wszystkie promienie zawarte w piramidzie widzenia mają **zgodne znaki** dla każdej współrzędnej. W przeciwnym przypadku piramida jest dzielona.
- w celu określenia jak piramida widzenia VF przecina prostopadłościan BB wierzchołka, tj. czy tylko lewe poddrzewo, tylko prawe lub obydwa, wyznacza się **położenie przecięcia** VF z π. Jeśli jest poza prostopadłościanem BB, to przecinane jest tylko jedno poddrzewo (rysunek 2),
- powyższy test przecięcia VF z π zredukowany jest tylko do **jednej współrzędnej** (np. w sytuacji na rysunku 2 porównywane są wartości **y**),

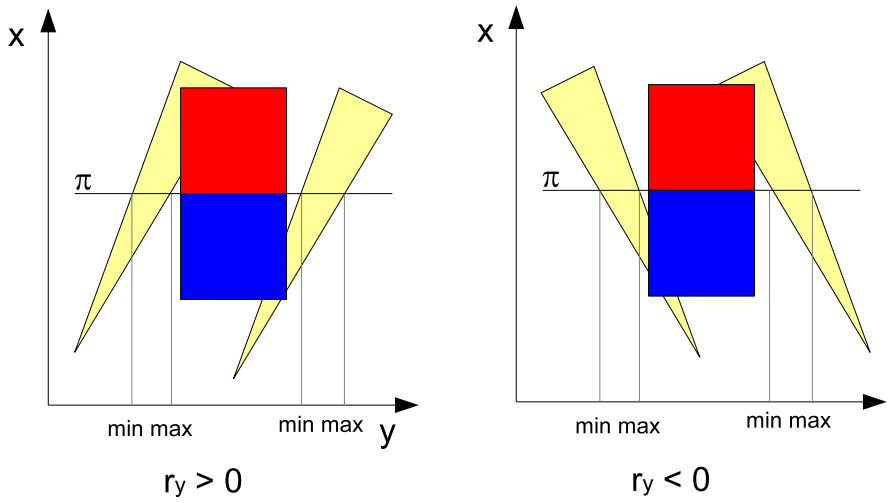
- liście drzewa poddawane są dodatkowemu testowi: dla wszystkich trzech możliwych par współrzędnych (**xy, yz, zx**) VF i BB rzutowane są na trzecią współrzędną. Przypadki w których piramida widzenia omija liść są rozpoznawalne z użyciem dwóch porównań np. dla pary **xy** prostopadłościan zadany jest jako

$$\beta = \{(x, y) : bmin_x < x < bmax_x \; i \; bmin_y < y < bmax_y\} \tag{3}$$

zatem piramida widzenia omija BB gdy jeden z poniższych warunków:

$$\begin{array}{l} min\,\{y_{in}\} > max\,\{x_{out}\} \\ min\,\{x_{in}\} > max\,\{y_{out}\} \end{array} \tag{4}$$

gdzie x_{in}, x_{out} wyznaczają przecięcia unormowanych promieni zawartych w VF z płaszczyznami BB. Punkty $r(x_{in})$ i $r(x_{out})$ leżą odpowiednio na płaszczyznach $x = bmin_x$ i $x = bmax_x$, gdzie $r(t)$ jest równaniem parametrycznym promienia. Analogicznie dla y_{in} i y_{out}.



Rysunek 2: Określanie przecięcia piramidy widzenia VF i prostopadłościanu BB (π jest płaszczyzną podziału wierzchołka kd-drzewa). Zakładamy, że VF przecina BB oraz promienie zgrupowane w VF mają zgodny znak współrzędnej **y**. Piramida przecina tylko górną (czerwoną) lub tylko dolną (niebieską) część gdy przecięcie z płaszczyzną π (wartości **min** i **max**) jest poza prostopadłościanem.

2.2.2 Odrzucanie tylnych ścian

Założenie, że tylko jedna strona powierzchni płaskich jest widoczna, prowadzi do efektywnych metod zmniejszania liczby elementów sceny, które wykorzystywane są podczas wyświetlania obrazu (czy też uwzględniane są w kolejnych obliczeniach).

W związku z tym, że płaszczyzna dzieli przestrzeń na dwie części, tj. część z przodu i z tyłu względem jej wektora normalnego, należy prostym testem sklasyfikować położenie obserwatora. Mając dany trójkąt T reprezentowany przez wierzchołki v_0, v_1, v_2 określane jest położenie obserwatora O następująco:

równanie płaszczyzny $\pi : n \cdot x + d = 0$ (gdzie x jest dowolnym punktem na płaszczyźnie) zawierającej T wyznaczamy obliczając:

$$\begin{aligned} n &= (v_1 - v_0) \times (v_2 - v_0) \\ d &= -n \cdot v_0 \end{aligned} \tag{5}$$

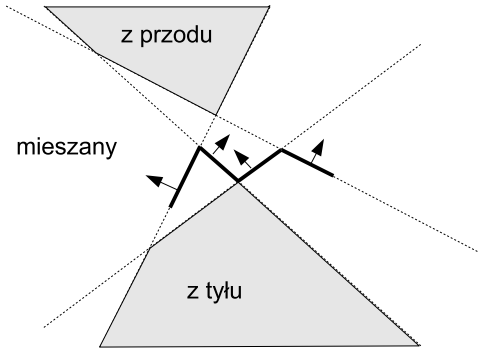
następnie po podstawieniu O do równania π określamy trójkąt jako widoczny z tyłu gdy otrzymamy wynik ujemny. Otrzymana wartość (bez znaku) jest odległością punktu O od T, przemnożoną przez $n \cdot n$.

Również sklasyfikowanie wielokąta T jako widocznego z tyłu jest możliwe poprzez zbadanie czy $n \cdot (O - p) < 0$, gdzie p jest dowolnym punktem z T.

2.2.3 Hierarchiczne odrzucanie tylnych ścian

Rozpatrując sytuację, w której położenie obserwatora nie jest stałe, tj. generując obraz z różnych punktów obszaru sceny, obliczenia dla wszystkich trójkątów za każdym razem staje się mało wydajne. Rozwiązaniem tego problemu jest grupowanie obiektów, tak by klasyfikacja czy grupa jest odwrócona przodem czy tyłem odbywało się za pomocą jednego testu. Subodh Kumar i Dinesh Manocha w [2] zaprezentowali hierarchiczne odrzucanie odwróconych obiektów (ang. *Hierarchical back-face culling*). Idea tego algorytmu jest następująca:

płaszczyzna każdego trójkąta dzieli przestrzeń 3D na obszary, z których jest on widoczny z przodu lub z tyłu. Korzystając z reprezentacji płaszczyzn w przestrzeni dualnej następuje ich grupowanie w klastry. W przypadku dużej liczby elementów następują kolejne podziały na 4 lub 8 podgrup aż do osiągnięcia minimalnej mocy podzbioru. Dla każdej grupy budowane są podziały na[1] (rysunek 3):



Rysunek 3: Określanie jak zorientowane są płaszczyzny względem dowolnych punktów obszarów. Punkty obszarów **z tyłu** i **z przodu** są rozwiązaniami problemu programowania liniowego zadanego nierównościami dla płaszczyzn.

1. region, z którego wszystkie trójkąty są widoczne z tylu (ang. *back region*)

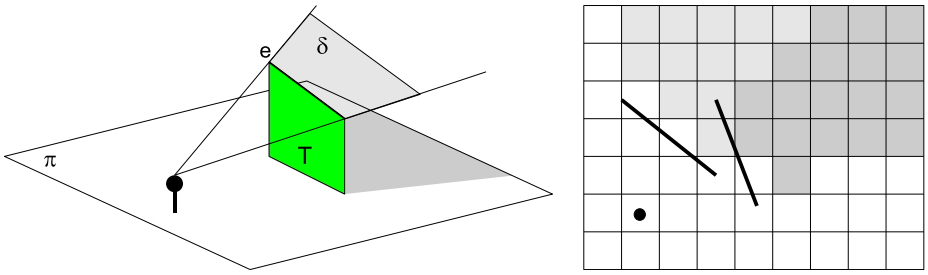
[1] Autorzy wykorzystują algorytmy rozwiązujące problem programowania liniowego oraz znajdowania otoczki wypukłej 3D

2. region, z którego wszystkie trójkąty są widoczne z przodu (ang. *front region*)
3. obszar mieszany zawierający obiekty widoczne z tylu lub z przodu (ang. *mixed region*). Nie jest on pamiętany.

Wykrywanie obiektów zwróconych przodem do obserwatora odbywa się, poprzez sklasyfikowanie do którego regionu należy punkt obserwatora dla każdego klastra.

2.2.4 Łączenie zasłoniętych obszarów

Peter Wonka i Dieter Schmalstieg zaproponowali metodę szybkiego odrzucania niewidocznych obiektów dla scen 2.5D [5]. Sprowadza się ona do wykorzystania kart graficznych do generowania obrazu zawierającego sumę zasłoniętych obszarów (rysunek 4).



Rysunek 4: Po lewej: Zasłonięty obszar δ wyznaczany jest na podstawie punktu obserwatora i górnej krawędzi e obiektu zasłaniającego. Dzięki projekcji δ na podłoże π powstaje mapa zasłonięcia która wraz z informacją o wysokości pozwala na określenie widoczności z punktu. Po prawej: mapa zasłonięcia *cull map*. Wykorzystując z-bufor i wsparcie sprzętowe rozwiązany jest problem nakładających się obszarów.

Idea jest następująca:

- scena zostaje umieszczona w strukturze - regularnej siatce 2D. Każdy obiekt jest umieszczony w jednej lub wielu komórkach.
- podczas wyświetlania sceny następuje dynamiczny wybór zbioru obiektów zasłaniających, dla których rysowane są cienie w odpowiednim buforze *cull map* (rysunek 4). Jako kryterium wyboru obiektów rzucających cień brana jest jedynie odległość od obserwatora.
- z każdym pikselem *cull mapy* utożsamiona jest odpowiada komórka w siatce 2D. Razem z informacją o wysokości padającego cienia rysowane są jedynie obiekty wyższe.
- w celu uzyskania konserwatywnego algorytmu należy skorygować rozmiar cienia. Przy rasteryzacji piksel p należący do wielokąta W jest rysowany gdy jego środek leży wewnątrz W. W algorytmie piksel reprezentuje kwadratowy obszar powierzchni zatem poprawna mapa zasłonięcia zawierać powinna te obszary, które

są całkowicie zawarte w cieniu, a nie tylko ich środek. Dotyczy to obszarów leżących na brzegu wielokąta. Determinuje to zatem korekcje rozmiaru cieni polegającą na ich zwężeniu.

Wyżej opisaną metodę można zastosować również dla dowolnych scen, wymaga on jednak odpowiedniego jej przetworzenia, tj. wyodrębnienia np. budynków, drzew, odcinków dróg oraz wyboru obiektów zasłaniających.

Pseudokod algorytmu rysującego potencjalnie widoczne obiekty:

```
for each C(i,j) in data grid
   if C(i,j).z > cullmap(i,j)
      for each object O(k) in C(i,j)
         if O(k).z > cullmap(i,j).z and O(k) not already rendered
            render O(k)
```

2.3 Geometria sceny widoczna z obszaru

W niniejszym rozdziale zostaną omówione wybrane algorytmy rozwiązujące problem widoczności z obszaru wraz z metodami np. aproksymacji zasłoniętego obszaru, wykorzystania algorytmów widoczności z punktu czy też przestrzeni dualnych.

Interesującymi rozwiązaniami są metody analizujące sceny w **przestrzeniach dualnych**. Spośród spotykanych rozwiązań na szczególną uwagę zasługują prace Koltuna [4] i Nirensteina [7] opisane w rozdziałach 2.3.2 oraz 2.3.5. W pierwszej zaproponowano wyznaczanie widocznych obiektów sceny 2.5D w dualnej przestrzeni promieni 2D. Druga publikacja opisuje dokładny algorytm wyznaczania widoczności wykorzystując reprezentacje obiektów i promieni w przestrzeni Plückera.

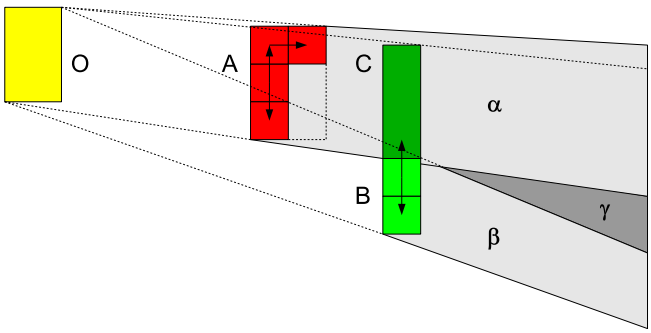
2.3.1 Algorytm Schauflera i przestrzeń zasłaniająca

Gernot Schaufler w [1] przedstawił wyznaczanie widocznych obiektów dzieląc przestrzeń sceny i łącząc obszary ograniczające widoczność . Algorytm operuje na scenach zarówno 2.5D jak i 3D, opiera się on na dyskretyzacji sceny na komórki (np. sześciany) a następnie określeniu, które z nich są widoczne. Ogólny zarys algorytmu jest następujący:

- w wolumetrycznej reprezentacji sceny, takiej jak **drzewa ósemkowe** (ang. *octtrees*) i **drzewa czwórkowe** (and. *quadtrees*) oznacza się liście jako **puste, wypełnione i brzegowe**. **Puste** odpowiadają obszarom niezawierającym elementów sceny. Jako **wypełnione** oznaczone są woksle (ang. *voksel*) w całości zawarte w obiektach. Pozostałymi są **brzegowe**, dla których dodatkowo wprowadza się dokładniejszy podział co powoduje większą głębokość drzewa,
- algorytm dla wybranych obiektów oznacza zasłonięty przez nie obszar,
- w celu optymalizacji obliczeń jako obiekt zasłaniający traktuje się fragment przestrzeni w kształcie prostopadłościanu,
- wybór obiektów, czyli **wypełnionych liści**, blokujących promienie z obszaru obserwatora następuje według kryteriów: odległości od obserwatora i rozmiaru. Bliższe obiekty ograniczają większy obszar, natomiast duże są na mniejszej głębokości w drzewie,

- wybrany zasłaniający obszar jest **odpowiednio powiększany** o przyległe do niego **wypełnione** wierzchołki tworząc w ten sposób duży obiekt ograniczający widoczność (*occluder fusion*). Zasłonięte wierzchołki traktowane są również jako **wypełnione** i służą powiększaniu prostopadłościanu (rysunek 5),

- definiując odpowiednio płaszczyzny styczne do zasłaniającego obszaru i obserwatora oraz trawersując drzewo zaznaczane są niewidoczne wierzchołki. Hierarchiczna struktura danych pozwala na skrócenie ścieżki w drzewie.



Rysunek 5: Rozszerzanie obiektu zasłaniającego o wypełnione woksle. Dla obszaru O rozszerzone obiekty A i B wyznaczają cienie α i β. Obszar γ jest zasłonięty dopiero po powiększeniu B o C, który będąc w cieniu A traktowany jest jako wypełniony obszar. Sklasyfikowanie C jako obszar pusty, wypełniony lub brzegowy nie jest uwzględniane.

W publikacji zostały opisane również specyficzne metody rozszerzania obiektu zasłaniającego oraz reprezentacji zasłoniętego obszaru dla scen typu 2.5D oraz 3D.

2.3.2 Selekcja widoczności w dualnej przestrzeni promieni

Vladen Koltun, Yiorgos Chrysanthou i Daniel Cohen-Or zaprezentowali w [4] algorytm wyznaczania widocznych elementów sceny z obszaru w przestrzeni dualnej promieni. Jest on związany z problemem przechodzenia złożonej sceny i wyświetlania jej w czasie rzeczywistym (ang. *online walkthrough*). Założenia oraz idea rozwiązania są następujące:

- rozpatrywanie scen typu miasto (**2.5D**), dla których określamy podział na prostopadłościany (komórki).

- zastosowanie **kd-drzew** do reprezentacji sceny.

- rozmiar obszaru, w którym porusza się obserwator jest tak dobrany, by czas potrzebny na jego przejście był wystarczający do wyznaczenia **PVS dla sąsiednich regionów**. Autorzy zaprezentowali system, w którym wyznaczanie **PVS** odbywa się na serwerze, a wyniki przesyłane są do aplikacji klienta.

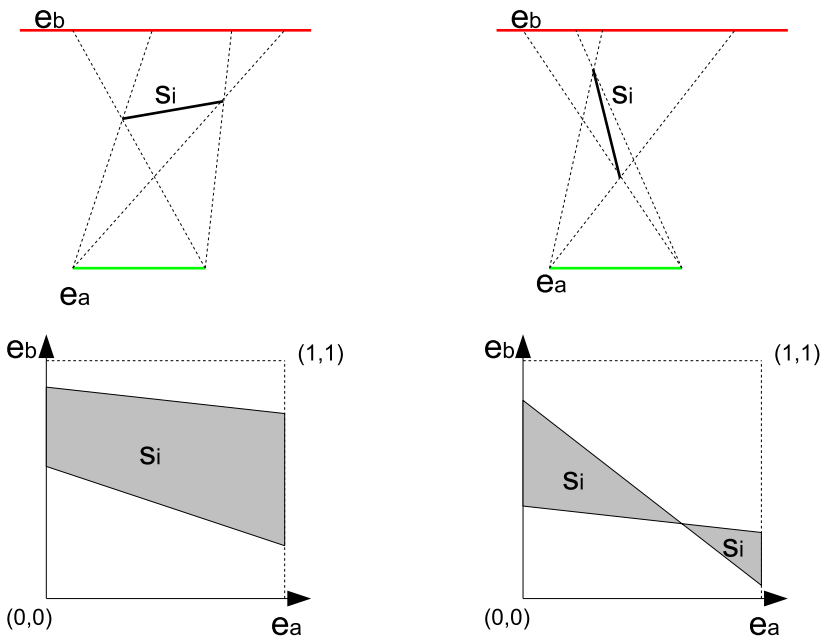
- zrezygnowano z fazy *preprocesingu*, którym wyznacza się widoczne obiekty dla wszystkich komórek.

- dla danego obszaru obserwatora algorytm trawersuje kd-drzewo określając dla każdego wierzchołka, czy *bounding box* z nim związany jest niewidoczny - co jest zarazem warunkiem stopu przechodzenia ścieżki w drzewie.

- widoczność między dwoma komórkami zredukowana jest do widoczności między dwoma krawędziami. Analizowana jest ona w dualnej dwuwymiarowej przestrzeni, w której każdemu promieniowi mającemu początek na jednej krawędzi i przecinającej drugą odpowiada punkt w przestrzeni dualnej.

W ostatnim punkcie wykorzystane jest założenie o wymiarze sceny 2.5D, *de facto* wymiar ten dotyczy wybranych obiektów zasłaniających niewidoczne obszary np. fasady budynków. Problem widoczności między dwoma komórkami *cell-to-cell* został więc zmodyfikowany do następującego zadania:

> dla danych komórek A i B i zbioru obiektów S (2.5D) wystarczy określić czy **górne krawędzie** e_a i e_b komórek A oraz B są całkowicie zasłonięte przez elementy z S (rysunek 6).



Rysunek 6: Krawędzie e_a i e_b oraz obiekt zasłaniający s_i. U góry w uproszczonej scenie, na dole w dualnej przestrzeni promieni

Zauważając, że obiekty (powierzchnie) z S przecinając płaszczyznę przechodzącą przez e_a i e_b determinują zbiór odcinków S', redukujemy problem widoczności następująco:

- definiujemy zbiór punktów $RS = \{(x, y) : 0 \leq x, y \leq 1\}$, w którym punktowi (x, y) odpowiada odcinek łączący $e_a(x)$ z $e_b(y)$.

- ograniczając się tylko do promieni między e_a i e_b elementowi $s \in S'$ odpowiada zamknięty obszar RS w postaci trapezu lub dwóch trójkątów (rysunek 6)

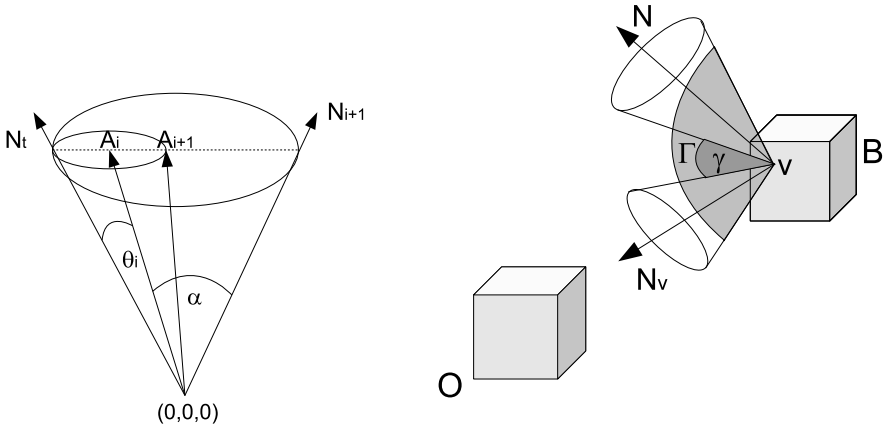
- generując obraz (bitmapę) elementów S' w RS stwierdzamy, czy komórki A i B są widoczne:
  - **tak** - jeśli istnieje niezamalowany punkt w RS
  - **nie** - jeśli RS został w całości pokryty elementami S'

Wybór obiektów 2.5D zasłaniających obszary sceny został opisany m.in. w pracach Isabla Navazo [15] i Vladena Koltuna [14].

2.3.3 Grupowanie i odrzucanie tylnych ścian

Jedną z metod redukcji liczby widocznych elementów jest odrzucanie obiektów odwróconych tyłem. Poprzez dyskretyzację obszaru obserwatora i reprezentację wybranymi punktami możemy odrzuć te elementy, które widoczne są z tyłu ze wszystkich punktów. Dzięki czemu możemy wykorzystać algorytmy widoczności dla pojedynczego punktu. Również przeprowadzenie testu widoczności z tyłu dla jednej płaszczyzny okazuje się być trywialnym zadaniem. Zakładając, że komórka obserwatora jest zadana punktami ekstremalnymi *bmin* i *bmax* analogicznie jak w opisanym algorytmie *frustum culling* wystarczy zbadać wartość równania płaszczyzny π dla odpowiedniego punktu ekstremalnego **p-vertex**.

Jatin Chhugani w pracy doktorskiej[10] przedstawił m.in. interesujący algorytm odrzucania obiektów *„Cell-based hierarchical back-face culling"* . Wykrywanie tylnych ścian opiera się o konstrukcje stożków (and. *bounding cone*) zawierających wektory normalne oraz wektory do obserwatora, a następnie badanie kątów między nimi (rysunek 7).



Rysunek 7: Po lewej: tworzenie otaczającego stożka *bounding cone*. Po prawej: użycie stożków i kątów Γ, γ do wyznaczenia widocznych obiektów z przodu i z tyłu.

Algorytm jest następujący:

- danymi są komórka obserwatora O oraz zbiór figur płaskich zamkniętych w prostopadłościanie B.
- wektory normalne powierzchni elementów zamykane są w stożku (podobna konstrukcja opisana jest w pracy Sederberga [16]):

– stożek reprezentowany jest przez parę wektor A_i oraz kąt θ_i, stożek ten zawiera wszystkie wektory, które tworzą z A_i kąt co najwyżej θ_i.

– zaczynając od pierwszego wektora N_1 budowany jest stożek reprezentowany przez wektor $A_1 = N_1$ i $\theta_1 = 0$.

– wyznaczając kolejne A_{i+1} i θ_{i+1} sprawdzamy kąt międzyA_i a N_{i+1}. Jeśli N_{i+1} jest wewnątrz stożka to $A_{i+1} = A_i$ oraz $\theta_{i+1} = \theta_i$. W przeciwnym przypadku należy stożek rozszerzyć:

$$\begin{aligned} A_{i+1} &= (N_t + N_{i+1}) / \|N_t + N_{i+1}\| \\ cos(\theta_{i+1}) &= A_{i+1} \cdot N_{i+1} \end{aligned} \tag{6}$$

gdzie N_t leży na płaszczyźnie zawierającej A_i i N_{i+1}, tworzy kąt θ_i z A_i i leży po drugiej stronie wektora A_i co N_{i+1} (por. rysunek 7, α jest kątem między wektorami A_i i N_{i+1}).

- (N, θ) zawiera wektory normalne i zaczyna się w dowolnym punkcie komórki B, oznaczamy go przez v

- z v budowany jest (V_v, α_v) - stożek widoczności (ang. *visibility cone*) zawierający promienie z v w dowolny obszar O.

- wyznaczane są: najmniejszy i największy kąt między dowolnymi wektorami normalnymi a wektorami widoczności:

$$\begin{aligned} \Gamma_v &= cos^{-1}(N \cdot V_v) + \theta + \alpha_v \\ \gamma_v &= cos^{-1}(N \cdot V_v) - \theta - \alpha_v \end{aligned} \tag{7}$$

- obliczając (V_v, α_v) i kąty Γ_v, γ_v w narożnikach B znajdujemy wartości ekstremalne: maksymalny Γ i minimalny γ.

- jeśli γ jest większy od $\pi/2$, to wszystkie elementy są zwrócone tyłem. Jeśli Γ jest mniejszy od $\pi/2$ wtedy wszystkie elementy zwrócone są przodem. W przeciwnym przypadku zbiór elementów jest dzielony na pół i rekurencyjnie analizowany.

2.3.4 Wykorzystanie widoczności z punktu w algorytmie Wonki

Bazując na rozwiązaniu widoczności z punktu [5] Wonka, Schmalstieg wraz z Michaelem Wimmerem rozszerzyli algorytm na wyznaczanie widoczności z obszaru w [6]. Algorytm jest konserwatywny, a idea sprowadza się do dyskretyzacji sceny na komórki, dla których wyznaczany jest zbiór charakterystycznych punktów obserwatora. Następnie jest obliczana widoczność sceny jako suma wyników widoczności z punktów.

Rozwiązanie opiera się o następujące obserwacje:

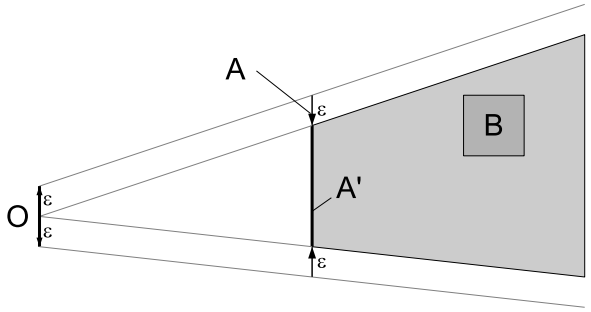
- warunkiem wystarczającym widoczności z prostopadłościanu jest widoczność z jego ścian,

- obiekt może zostać błędnie sklasyfikowany jako zasłonięty jeśli jest widoczny jedynie z obszaru między punktami z P,

- obliczenie konserwatywnej aproksymacji widoczności jest możliwe poprzez zbadanie jej dla dyskretnego zbioru punktów P, gdy dokona się korekcji zasłanianego obszaru,

- zwężona powierzchnia zasłaniająca o ϵ determinuje mniejszy cień. Spełniona jest wtedy własność:

  > obiekt B sklasyfikowany jako zasłonięty przez A' (będący skurczonym A o ϵ) z punktu obserwatora O pozostaje zasłonięty przez A (oryginalny obiekt) z dowolnego punktu O' przesuniętego o co najwyżej ϵ punktu O ($\|O' - O\| \leq \epsilon$) (rysunek 8).



Rysunek 8: Obiekty B zasłonięty przez zmniejszony A' jest również niewidoczny z otoczenia O', ponieważ przysłania go A

Należy zwrócić uwagę, że zwężanie obiektów zasłaniających różni się dla brył i powierzchni. Zmniejszenie o ϵ jest wystarczające dla obiektów wolumetrycznych (ang. *volumetric occluder*), co zostało udowodnione w publikacji Wonki [5]. Natomiast w przypadku płaskich obiektów zasłaniających (ang. *planar occluder*) zwężanie w każdym kierunku determinowane jest przez pozycję obserwatora i obiektu. Zatem analogiczne zmniejszanie jak dla brył nie jest wystarczające. Przykład takiej sytuacji jest na rysunku 9. Cień α figury płaskiej A' (powstałego przez „skurczenie" A o ϵ) zawiera elementy B i C. Przesunięcie obserwatora o ϵ sprawia, ze B staje się widoczny. Cień β jest właściwie wyznaczony, tj. każdy obiekt w nim zawarty nie jest widoczny z otoczenia O.

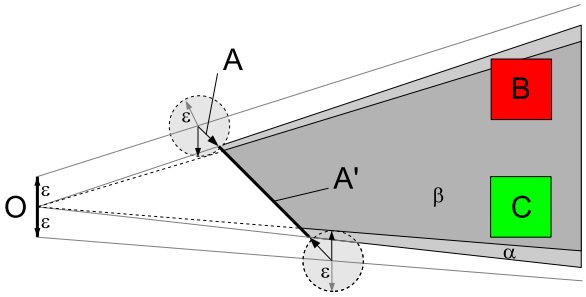
Algorytm selekcji widocznych obiektów sceny można opisać następująco:

1. podział sceny na komórki i wybór ϵ. Dla scen 2.5D zastosowano triangulację Delaunaya.

2. zwężanie obiektów zasłaniających w zależności od ϵ

3. dla każdej komórki sceny:

   - określenie wystarczającej liczby próbek reprezentujących komórkę
   - wyznaczenie widoczności z każdego punktu
   - wyznaczenie widoczności z komórki poprzez scalenie **PVS** uzyskanych dla punktów lub wygenerowanych cieni (rozdział 2.2.4)



Rysunek 9: Obiekt $B \in \alpha$ zasłonięty przez zmniejszony A' o ϵ jest widoczny z otoczenia O'. Po uwzględnieniu położenia obiektu względem obserwatora obiekt $C \in \beta$ pozostaje zasłonięty przez A

2.3.5 Dokładny algorytm widoczności Nirentseina

Nirenstein, Blake i Gain opublikowali interesujący algorytm odrzucania niewidocznych obiektów [7]. W przeciwieństwie do dużej liczby rozwiązań aproksymacyjnych jest to algorytm dokładny, tj. wyznaczający tylko widoczne obiekty z zadanego obszaru. W praktyce, z powodu błędów numerycznych powstających przy reprezentacji i obliczeniach, wynik nie gwarantuje jednak selekcji wszystkich widocznych elementów. Algorytm jest w stanie operować na bardzo złożonych scenach, w których liczba trójkątów jest rzędu miliona. Idea algorytmu polega na reprezentacji wielokątów (z przestrzeni R^3) i linii przecinających je L (ang. *stabbing lines*) w pięciowymiarowej przestrzeni euklidesowej otrzymanej z przestrzeni *Plückera* a następnie odejmowaniu zasłoniętych linii ze zbioru L.

Współrzędne i hiperppowierzchnia Plückera

Współrzędne Plückera są niezwykle użyteczne w operacjach na liniach i promieniach 3D jako wektorach w szesciowymiarowej przestrzeni. Przekształcenie $\Pi : \mathbb{R}^6 \rightarrow \mathbb{P}^5$ jest zdefiniowane następująco:

> Niech l będzie promieniem (lub linią) przechodzącą przez punkty $P, Q \in \mathbb{R}^3$ w postaci odpowiednio (p_x, p_y, p_z) i (q_x, q_y, q_z).
> Wtedy $\Pi(l) = (\pi_0, \pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$, gdzie:

$$\begin{array}{lcllcl} \pi_0 & = & q_x - p_x & \pi_3 & = & q_z p_y - q_y p_z \\ \pi_1 & = & q_y - p_y & \pi_4 & = & q_x p_z - q_z p_x \\ \pi_2 & = & q_z - p_z & \pi_5 & = & q_y p_x - q_x p_y \end{array} \tag{8}$$

Własności:

- współrzędne $(\pi_0, \pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$ są jednorodne, tj. przemnożenie wszystkich współrzędnych przez dowolny dodatni skalar daje w wyniku inną reprezentację tej samej linii,
- przekształcenie Π jest różnowartościowe,

- przestrzeń rzutową $\mathbb{P}^5$ można interpretować jako $\mathbb{R}^5$ wraz z połączonymi hiperpłaszczyznami w nieskończoności w dodatnim i ujemnym kierunku. Zgodnie z pierwszym punktem szóstkę można podzielić przez wybraną współrzędną np. π_3, następnie operować na elementach $\mathbb{R}^5$. Należy dodatkowo zadbać o odpowiednią przesunięcie linii w scenie by nie dzielić przez zero.
- nie każdy punkt (szóstka współrzędnych Plückera) odpowiada linii w 3D,
- definiując następująco iloczyn skalarny π i x ($\pi, x \in \mathbb{P}^5$):

$$\begin{gathered} D_\pi(x) : \mathbb{P}^5 \to \mathbb{R} \\ D_\pi(x) = \pi_0 x_3 + \pi_1 x_4 + \pi_2 x_5 + \pi_3 x_0 + \pi_4 x_1 + \pi_5 x_2 \end{gathered} \tag{9}$$

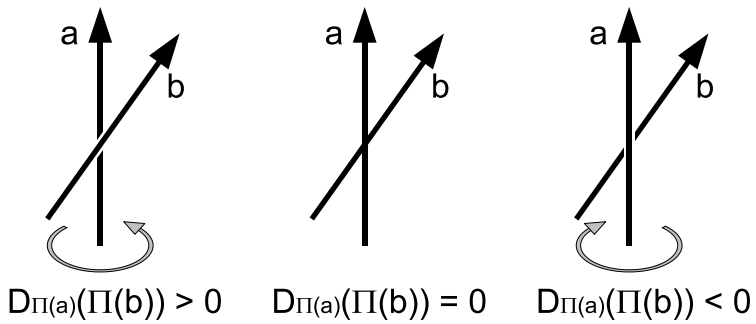
  zbiór rozwiązań $D_\pi(x) = 0$ określa dualną płaszczyznę π w $\mathbb{P}^5$ - hiperpłaszczyznę Plückera.

W celu ograniczenia zbioru do obrazu linii 3D względem przekształcenia Π definiuje się hiperpowierznię Plückera. Idea polega na ograniczeniu się do elementów dla których iloczyn skalarny z samym sobą daje w wyniku 0, mianowicie:

$$G = \{x \in \mathbb{P}^5 : D_x(x) = 0\} \setminus \{0\} \tag{10}$$

Praktyczne zastosowanie współrzędnych Plückera

Badanie wartości iloczynu skalarnego $D_\pi(x)$ umożliwia określenie położenia względem siebie dwóch promieni. Jeśli wartość wynosi zero, wtedy dwa promienie przecinają się. W przeciwnym przypadku drugi promień mija pierwszy z prawej lub lewej strony (patrząc na promień „z góry" przypomina to obieganie zgodnie z ruchem wskazówek zegara lub w stronę przeciwną). Interpretacja graficzna przedstawiona jest na rysunku 10.



Rysunek 10: Interpretacja graficzna wartości iloczynu skalarnego $D_{\Pi(a)}(\Pi(b))$: a i b - promienie w $\mathbb{R}^3$. Po lewej: promień b omija a w przeciwną stronę do ruchu wskazówek zegara. Na środku: a i b incydentne. Po prawej: promień b omija a zgodnie z ruchem wskazówek zegara.

Dzięki powyższym własnościom dość łatwo zaimplementować jest algorytm wykrywania przecięcia promienia r z wielokątem T . Traktując krawędzie figury jako odpowiednio skierowane promienie e_i promień r przecina T gdy:

$$D_\pi(\Pi(r)) > 0, \forall \pi \in \{\Pi(e_1), \Pi(e_2), \cdots, \Pi(e_n)\} \tag{11}$$

Innym ciekawym algorytmem wykorzystującym współrzędne i przestrzeń Plückera jest wyznaczanie przecięcia promienia z zastosowaniem SIMD [21].

Algorytm wyznaczania widocznych obiektów

Idea selekcja widocznych obiektów jest zbliżona do rozwiązania Vladena Koltuna [4] (rozdział 2.3.2):

- widoczność wyznaczana jest poprzez testowanie czy para powierzchni jest wzajemnie widoczna,
- powierzchnie są wzajemnie widoczne gdy istnieje promień je przecinający, który nie trafia w żaden obiekt zasłaniający. Analogicznie do algorytmu Koltuna, w którym w scenach 2.5 wymiarowych problem redukuje się do wzajemnej widoczności górnych krawędzi,
- zbiór rozpatrywanych linii L między parą płaszczyzn (wielokątów) reprezentowany jest w hiperpowierzchni Plückera. Jest to zbiór spełniający nierówność 11 dla dwóch wielokątów.
- każdy zasłaniający wielokąt - *de facto* trójkąt - blokuje część linii O_i redukując zatem zbiór możliwych promieni L.
- para płaszczyzn jest wzajemnie widoczna gdy L pozostanie niepuste, tj. $L \setminus \bigcup_{i \in I} O_i \neq \oslash$.
- do odejmowania zbiorów zastosowano algorytm oparty o **CSG** (ang. Constructive Solid Geomtery) w 5D.

2.4 Poziom szczegółowości

Wprowadzenie poziomu szczegółowości LOD (ang. *Level of detail*) jest naturalną metodą upraszczania geometrii sceny polegającą na wyświetlaniu uproszczonych modeli znajdujących się daleko od obserwatora. W przypadku scen, których złożoność graniczy z pojemnością pamięci operacyjnych komputerów klasy PC, stosowanie LOD jest konieczne. Poniżej przedstawiona została klasyfikacja spotykanych rozwiązań.

Systemy stosujące poziomy szczegółowości LOD oraz aproksymacje modeli sceny, można podzielić na:

- dyskretne (statyczne) - wyznaczają zbiór aproksymacji każdego modelu, wraz z współczynnikiem błędu. Podczas działania zamieniają odpowiednią wersję obiektu w zależności od obserwatora tj. orientacji obiektu i odległości od niego, np. przez zbadanie ilorazu odległości i współczynników błędu. O stosowanych technikach można przeczytać m.in. w pracach C.Eriksona, D. Manocha [23] opisującej kontrukcję LOD dla dużych statycznych (jak i dynamicznych) scen. Natomiast w pracy M.Garlanda i P. Heckberta [24] przedstawiono metody aproksymacji powierzchni.
- ciągłe (progresywne) - wyznaczają hierarchię aproksymacji oryginalnego modelu. Opis metody można znaleźć w pracy H. Hoppe [25].

- zależne od widoku - są rozszerzeniem systemu ciągłego o uwzględnienie orientacji modelu względem obserwatora. Dzięki czemu bliższa część modelu jest wyświetlana z większą dokładnością. Metoda jest szczególnie użyteczna w przypadku dużych a zarazem szczegółowych modeli. Została on opublikowana w pracy Xia i Varshneya [26].

3 Ukierunkowane próbkowanie widoczności

Powstanie efektywnych algorytmów śledzenia promieni jak np. [13] otworzyło drogę metodom selekcji widocznych obiektów opartych na śledzeniu promieni. Takim algorytmem jest ukierunkowane próbkowanie widoczności **GVS** (ang. *Guided visiblility sampling*) opublikowane w pracy [3] autorstwa: Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina i Alexander Reshetov. Opracowana metoda służy do rozwiązywania następującego problemu:

- *mając dany zbiór trójkątów T określić, które są w całości lub częściowo widoczne z wyznaczonego obszaru (powierzchni)*

Bazując na modelu promieni i przecinanych obiektów rozwiązanie można przybliżyć do obserwatora, który ogląda scenę w dowolnych kierunkach i porusza się po wyznaczonej powierzchni. Utożsamiamy w ten sposób widoczny w danym kierunku obiekt z promieniem i trójkątem w 3D.
Mówiąc bardziej formalnie specyfikacje zadania można przedstawić następująco:
Niech:

- Ω - zbiór wszystkich promieni wychodzących z zadanego obszaru,
- T - zbiór trójkątów sceny,
- $v : \Omega \rightarrow T$ funkcja widoczności zdefiniowana następująco:

$$v(r) \textit{ jest pierwszym trójkątem przeciętym przez promień } r$$

Trójkąt $t \in T$ jest widoczny, wtedy i tylko wtedy gdy istnieje trafiający w niego promień $r \in \Omega$.

Zadanie wyznaczania widocznych obiektów sprowadza się wiec do znalezienia zbioru promieni Ψ takiego, że $v(\Psi)$ zawiera wszystkie widoczne trójkąty. Oczywistym faktem jest, że spośród możliwych rozwiązań lepszymi są takie, które zawierają mniejszą ilość promieni (Ω jest także rozwiązaniem).

Algorytm **GVS** jest **agresywną** techniką, która skupia się na wyznaczeniu w sposób przyrostowy zbioru Ψ_i, który określa widoczny podzbiór sceny T_i. Bazując na znalezionych już obiektach wyznacza kolejne promienie eksplorujące otoczenie T_i jak również luki. W opublikowanej pracy rozwiązanie jest złożeniem trzech następujących algorytmów:

- *random sampling* - losowe generowanie promieni, używane w celu zainicjowania kolejnych algorytmów.

- *adaptive border sampling* - (adaptacyjne próbkowanie krawędzi) - odpowiada za szybkie analizowanie sąsiednich trójkątów poprzez próbkowanie krawędzi znalezionego obszaru (tj. otoczenia znalezionych widocznych trójkątów).
- *reverse sampling* (próbkowanie wstecz) - w celu zbadania obszarów sceny będących na granicy widoczności (np. luki między trójkątami).

Kolejne podrozdziały zawierają szczegółowy opis wyżej wymienionych algorytmów.

3.1 Próbkowanie pseudo-losowe

Podstawowym i za razem najprostszym algorytmem wyznaczania promieni znajdujących widoczne obiekty sceny jest losowanie z rozkładem jednostajnym elementu ze zbioru Ω. Algorytm składa się z dwóch faz:

1. losowanie punktu na płaszczyźnie, tj położenia początku promienia
2. losowanie kierunku promienia.

Należy jednak zwrócić uwagę na drugi punkt, gdyż wylosowany kierunek nie może być dowolny. Mianowicie wyznaczany wektor jest akceptowalny, tylko wtedy, gdy tworzy wektorem normalnym płaszczyzny kąt co najwyżej 90^o. Warunek taki jest spełniony gdy $\vec{v} \circ \vec{n}$, gdzie $\vec{n}$ jest wektorem normalnym powierzchni, po której porusza się obserwator. Przykładów rozwiązań jest wiele, a wybór nie ma dużego znaczenia, gdyż algorytm **GVS** korzysta głównie z metod opisanych w następnych rozdziałach.

Zastosowany algorytm wyznaczania losowego promienia (położenia i kierunku):

$$u = \xi_1, \quad v = \xi_2, \quad \phi = 2\pi\xi_3, \quad \theta = arcsin\xi_4$$

gdzie $\xi_i \in [0, 1]$ są wartościami pseudolosowymi (sekwencje Haltona [20]).

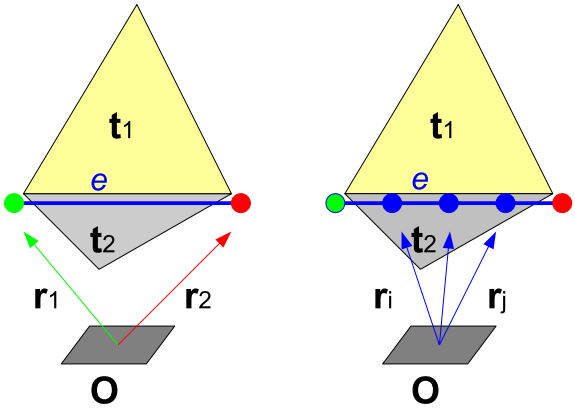
3.2 Adaptacyjne próbkowanie otoczenia

Próbkowanie adaptacyjne krawędzi (**ABS**, ang. *adaptive border sampling*) jest algorytmem opartym na mutacji promieni, który w oparciu o trafiony trójkąt wyznacza deterministycznie kolejne promienie. ABS analizuje przyległe trójkąty w scenie nie zmieniając położenia początku promienia.

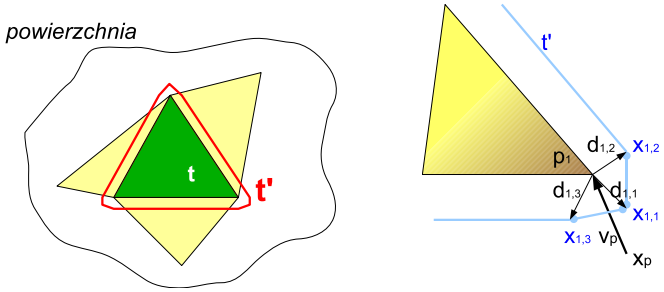
Ideą algorytmu jest dopasowanie próbkowania do geometrii sceny, tak aby dalekim złożonym obszarom, skupiającym większą liczbę trójkątów, wyznaczyć więcej promieni skierowanych w ich kierunku. W ten sposób nie zostaną pominięte obiekty małe lub położone daleko od obserwatora. Prawdopodobieństwo znalezienia ich jest większe niż w przypadku algorytmów próbkujących regularnie. Warto zauważyć, że badanie otoczenia znalezionego obiektu jest idealną metodą do wyznaczania widocznych fragmentów powierzchni, która to w praktyce jest po prostu zbiorem przylegających trójkątów.

Algorytm jest następujący:

Trójkąt t trafiony pierwszy raz przez promień r jest minimalnie powiększany. Bazując na otrzymanym wielokącie t', próbkowanie odbywa się względem jego krawędzi. W przypadku gdy w końcach badanego odcinka zostały znalezione różne obiekty następuje jego podział. Rekurencyjnie analizowane zostają kolejne części krawędzi (rysunek 11).



Rysunek 11: Próbkowanie krawędzi e. Promienie r_1 i r_2 skierowane w kierunku końców odcinka e trafiają w dwa różne obiekty. Determinuje to rekurencyjne dzielenie odcinka i kolejne próbkowania r_i, r_j itd.



Rysunek 12: Powiększanie trójkąta. Po lewej: próbkowanie otoczenia skutkuje znalezieniem przyległych trójkątów powierzchni. Po prawej: wyznaczanie otoczenia t', wektory $d_{i,i}$ są prostopadłe do wektora v_p.

Kształt t' jest wyznaczany nie tylko na podstawie bazowego trójkąta t. Ze względu na niedokładności obliczeń numerycznych i błędów zaokrągleń pod uwagę brane są również:

1. ułożenie trójkąta względem obserwatora, tj. jak trójkąt jest obrócony
2. odległość trójkąta od obserwatora - mające kluczowe znaczenie dla dalekich obiektów

Uwzględniając dodatkowo małe odległości między krawędziami z t' a t istnieje ryzyko, że promienie trafią znowu w bazowy trójkąt t. Zaproponowana metoda wyznaczania t' jest następująca: Dla każdego wierzchołka trójkąta (rysunek 12) p_i wyznaczane są trzy wektory $d_{i,i-1}$, $d_{i,i}$, $d_{i,i+1}$. Wektory te są prostopadłe do promienia padającego na wierzchołek trójkąta oraz odpowiednio do krawędzi trójkąta (z wyjątkiem $d_{i,i}$, który jest kombinacją dwóch sąsiednich wektorów). Poprzez przesunięcie wierzchołków odpowiednio o wektory $d_{i,j}$ zostaje wyznaczonych 9 wierzchołków wielokąta t' (de facto łamanej, gdyż wszystkie wierzchołki $x_{i,j}$ nie leżą na jednej płaszczyźnie):

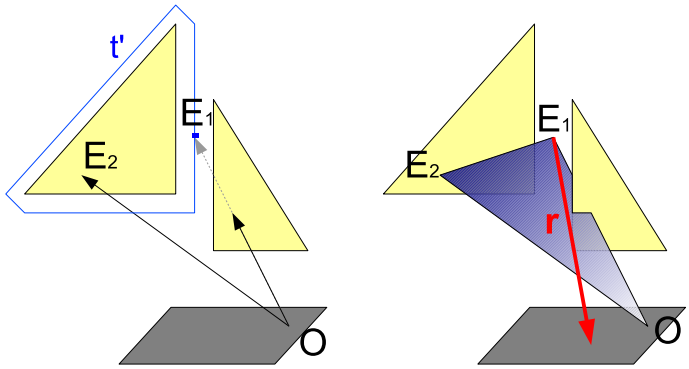
$$
\begin{aligned}
d_{i,i-1} &= Norm((p_i - x_p) \times (p_{i+1} - p_i)) \\
d_{i,i+1} &= Norm((p_i - x_p) \times (p_i - p_{i-1})) \\
d_{i,i} &= \begin{cases} Norm(d_{i,i-1} + d_{i,i+1}) & if\ d_{i,i-1} \cdot d_{i,i+1} > 0 \\ Norm((p_i - x_p) \times d_{i,i-1} + d_{i,i+1} \times (p_i - x_p)) & else \end{cases} \\
x_{i,j} &= p_i + \epsilon \cdot |p_i - x_p| \cdot d_{i,j}
\end{aligned} \tag{12}
$$

gdzie $Norm$ jest operatorem normalizacji. Należy zwrócić uwagę, że w przypadku gdy trójkąt jest widoczny z tyłu należy zamienić wektory $d_{i,j}$ na przeciwne.

Wadą algorytmu jest brak możliwości analizy luk między trójkątami, które widoczne są jedynie z innego punktu sceny. Również obszary sceny z wieloma małymi nie połączonymi siatkami trójkątów są niemal niemożliwe do eksplorowania. Sceny zawierające losowo ułożone trójkąty, np. las, w którym każdy liść jest reprezentowany przez np. 2 trójkąty, pozostają ciągle dużym wyzwaniem.

3.3 Próbkowanie wsteczne

Próbkowanie wstecz służy generowaniu promieni w obszary będące na granicy widoczności. Główną wadą omówionego w poprzednim podrozdziale rozwiązania był brak możliwości penetrowania luk między trójkątami. Luki takie widoczne były jedynie z innych punktów sceny. Choć pytanie, czy z danego obszaru widzenia znaleziona luka jest widoczna jest *de facto* problem widoczności, to okazało się możliwe w wielu przypadkach wyznaczenie wektora omijającego bliższy trójkąt, który skierowany jest w niezbadany obszar (rys 13).



Rysunek 13: Próbkowanie luki między dwoma trójkątami. Po lewej: znalezione przecięcie promienia z trójkątem (punkt E_2), przewidywane trafienie (punkt E_1) wygenerowane podczas *adaptive border sampling*, widoczne trafienie w bliższy trójkąt. Po prawej: wygenerowanie nowego promienia, który wyznacza nowy *zarodek* dla *ABS*, wektor $\vec{r}$ leży na płaszczyźnie wyznaczonej przez punkty E_1, E_2 i O.

Działanie algorytmu jest następujące. Niech (patrz rysunek 13):

- E_2 - punkt przecięcia promienia z trójkątem t,
- E_1 - jeden z punktów należących do powiększenia trójkąta t',

- O - punkt początkowy - położenie obserwatora,

jeśli podczas próbkowania krawędzi t' zostanie znaleziony obiekt bliższy, wyznacz promień r który:

- leży na płaszczyźnie wyznaczonej przez punkty E_1, E_2 i O
- omija bliższy obiekt przechodząc jak najbliżej jego krawędzi

Przecięcie promienia r z obszarem widzenia wyznacza nowy punkt dla obserwatora, zaś promień o przeciwnym zwrocie penetruje lukę. Staję się on kolejnym punktem dyskretyzacji obszaru obserwatora i zarazem zarodkiem dla próbkowania adaptacyjnego.

3.4 Algorytm GVS

Z połączenia powyższy trzech metod powstał następujący algorytm GVS wyznaczania widocznych obiektów z obszaru:

```
main()
   while (not finished)
      (xp,xd) = generate_random_ray()
      handle_ray((xp,xd))
      while (not queue.empty())
         adaptive_border_sampling(queue.dequeue()))

handle_ray(x)
   if v(x) not in PVS
      PVS+=v(x)
      queue+=x

adaptive_border_sampling(x)
   t' = enlarge(v(x), eps)
   for each point(p) in t'
      handle_ray((xp, p-xp))
   for each edge(pl, pr) in t'
      subdiv_edge(pl, pr)

subdiv_edge(pl, pr)
   x = (xp, pl - xp)
   y = (xp, pr - xp)
   check_discontinuity(x)
   check_discontinuity(y)
   if v(x) = v(y) or |hit(x) - hit(y)| < eps
      return
   else
      p = (pl + pr) /2
      handle_ray((xp, p - xp))
      subdiv_edge(pl, p)
```

```
      subdiv_edge(p, pr)

check_discontinuity(x)
   if |predicted_hit(x) - xp| - |hit(x) - xp| > tresh
      xn = reverse_sampling(x)
      if start(xn) in view cell
         handle_ray(xn)
```

gdzie:

- $v(x)$ - pierwszy trójkąt przecięty przez promień x,
- $hit(x)$ - punkt trafienia promieniem x trójkąta $v(x)$
- $predicted\_hit(x)$ - przewidywany punkt trafienia. W przypadku rozszerzania trójkąta t punkt ten określany jest jako przecięcie promienia x z płaszczyzną zawierającą trójkąt t. Równie dobrym przybliżeniem jest punkt leżący na t'.

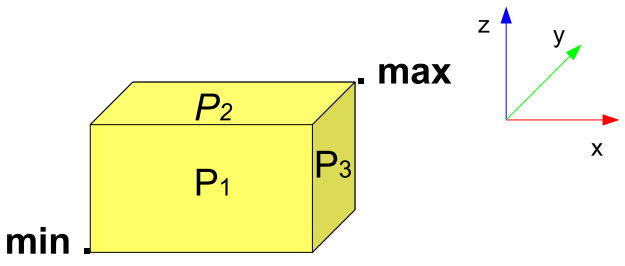
Koniec algorytmu następuje, gdy zostanie spełniony jeden z następujących warunków:

- wygenerowano 10 milionów promieni,
- nie więcej niż 50 nowych trójkątów zostało znalezionych po sprawdzeniu 1 miliona promieni.

4 Rozszerzenia algorytmu GVS

4.1 Obszar obserwatora

Stosunkowo łatwo jest uogólnić pole widzenia *view cell* na dowolny obszar. Rozszerzenie sprowadza się jedynie do znalezienia algorytmu generowania próbek losowych z nowego otoczenia obserwatora. Dodatkowo dla potrzeb algorytmu próbkowania wstecznego (ang. *reverse sampling*) należy zwrócić uwagę, by znajdowanie przecięć promienia z komórką obserwatora było łatwo obliczalne. Efektywnym rozwiązaniem jest wybranie prostopadłościanu AABB (rysunek 14).



Rysunek 14: Prostopadłościan jako obszar widzenia

Losowanie z rozkładem jednostajnym następuje według poniższego algorytmu:

- wylosuj scianę P_i z prawdopodobieństwem

$$ppb(P_i) = area(P_i) / \sum_{k=1}^{6} area(P_k)$$

  gdzie $area(P)$ to pole powierzchni P

- wylosuj promień wychodzący ze ściany wyznaczając[19]:
  - (u, v) - początek promienia, leżący na P_i
  - r_1, r_2 - (pseudo) losowe liczby z przedziału $[0, 1]$
  - oblicz losowy wektor jednostkowy (z uwzględnieniem wektora normalnego powierzchni P_i) np.:

$$\begin{aligned} r_x &= \cos(2\pi r_1)\sqrt{1 - r_2^2} \\ r_y &= \sin(2\pi r_1)\sqrt{1 - r_2^2} \\ r_z &= r_2 \end{aligned}$$

    Powyższe wartości odpowiadają powierzchni z wektorem normalnym $\vec{n} = [0, 0, 1]$. W pozostałych przypadkach obliczenia są analogiczne (otrzymane wartości można też odpowiednio przestawić lub też przemnożyć przez -1 by zmienić zwrot wektora).

Przecięcie promienia z *bounding boxem* jest jednym z podstawowym zadań w grafice komputerowej. Jego rozwiązanie pozwala na weryfikację, czy dany promień omija scenę, czy też skierowany jest w jej wnętrze. Zatem do sprawdzenia czy promień wsteczny przecina obszar obserwatora można zastosować efektywny algorytm *Liang-Barsky Line Clipping* [2].

4.2 Klasyfikacja promieni

Generowane przez GVS promienie, dzięki którym znajdowane są kolejne widoczne elementy sceny, można sklasyfikować następująco:

- wsteczne - penetrujące nieciągłości (luki) w próbkowaniu wstecznym,
- punktowe i krawędziowe - skierowane w punkty i krawędzie wyznaczone przez próbkowanie adaptacyjne otoczenia
- losowe - promienie inicjujące, wyznaczane przez próbkowanie losowe.

Warto zauważyć, że czas działania algorytmu ściśle związany jest z liczbą promieni. Zakładając niewielki rozmiar PVS względem sceny[3] i liczby rozpatrywanych promieni (zdeterminowanej przez sam algorytm oraz warunek stopu) czas działania algorytmu uzależniony jest od:

- liczby losowych promieni,

[2] źródło: http://www.siggraph.org/education/materials/HyperGraph/scanline/clipping/cliplb.htm
[3] W zaprezentowanych wynikach w [3] wyznaczany PVS stanowił 0,1% całej sceny.

- promieni skierowanych w krawędzie, *de facto* liczby różnych obiektów wykrytych przy jej analizie.

Chcąc otrzymać w krótszym czasie PVS, nawet za cenę większego błędu, należy zatem ograniczyć występowanie powyższych dwóch przypadków. Osiągnąć to można poprzez ograniczenie maksymalnej liczby promieni oraz modyfikację algorytmu próbkowania adaptacyjnego poprzez:

- dobór wartości **tresh** określającej minimalną długość dzielonej krawędzi
- zmianę sposobu podziału krawędzi zamiast dzielenia na pół.

4.3 Strategie próbkowania krawędzi

Czas analizy otoczenia obiektu związanej z próbkowaniem punktów i odcinków w głównej mierze zależy od krawędzi, tj. liczby widocznych obiektów, które przecinane są przez promienie skierowane w krawędź. Bardziej formalnie mówiąc, czas poświęcany przez algorytm na uwzględnienie krawędzi E_1E_2 z punktu obserwatora O zależy od liczby obiektów przecinanych przez wycinek $\angle E_1OE_2$ płaszczyzny wyznaczonej przez punkty E_1, O, E_2. Dzielenie krawędzi i rekurencyjne analizowane jej części zdeterminowane jest przez dwa cele:

- znalezienie jak największej liczby widocznych obiektów z O na krawędzi E_1E_2,
- wyznaczenie zasłaniającego obszaru, bardziej formalnie: ograniczenie długości promieni przecinających odcinek, które wyznaczają pustą przestrzeń w płaszczyźnie E_1OE_2.

Głównym czynnikiem decydującym o kolejnych podziałach krawędzi jest określenie czy promienie skierowane w jej końce trafiają w ten sam obiekt T. Z tego faktu oraz z założenia, że trójkąty są obiektami wypukłymi, wynika, iż każdy promień w odcinek trafi w T lub obiekt bliższy. Algorytm zatem nie znajduje dokładnie wszystkich widocznych obiektów (zgodnie z jego klasyfikacją do agresywnych algorytmów wyznaczających widoczność). Stąd też wniosek, iż określanie zasłanianego obszaru służy zredukowaniu liczby generowanych promieni przy analizie krawędzi. Drugim, mniej znaczącym warunkiem stopu jest minimalny rozmiar krawędzi. Jej wielkość ma oczywisty wpływ na czas algorytmu.

Należy zwrócić tu również uwagę na fakt, że dla wszystkich promieni krawędziowych badane są nieciągłości sceny i inicjowany jest algorytm *reverse sampling*. Poniższa tabela 1 zawiera przykładową statystykę wyznaczanego PVS dla sceny *sibenik*.

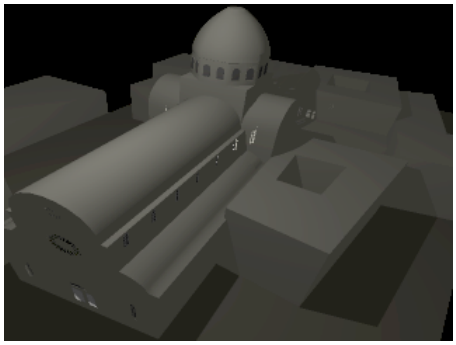
Zbiór widocznych obiektów został wyznaczony dla otoczenia obserwatora, z którego widok na scenę jest na rysunku 15.

W kolejnych częściach rozdziału przedstawiona została analiza oraz **mutacje** algorytmu **podziału odcinka** uwzględniające:

- maksymalną liczbę podziałów danego odcinka (tj. głębokość rekurencji),
- odległości odcinka od punktu obserwatora,
- wykrywanie przyległych obiektów.

Tablica 1: Statystyka promieni algorytmu GVS dla sceny *Sibenik* zawierającej 75 tys. trójkątów.

rozmiar PVS	7,494
liczba promieni w krawędzie	549,626
liczba promieni w punkty	22,482
liczba wykrytych nieciągłości	319,415
liczba promieni w nieciągłości	116,146



Rysunek 15: Widok z badanego obszaru obserwatora na zbiór widocznych obiektów.

Powyższe modyfikacje przyczyniają się do przyspieszenia algorytmu analizy krawędzi (poprzez redukcję liczby generowanych promieni) kosztem zwiększenia prawdopodobieństwa pominięcia widocznego elementu w trakcie obliczeń.

4.3.1 Głębokość rekursji

Modyfikacja warunku stopu: **v(x) = v(y) or |hit(x) - hit(y)| < tresh** poprzez zwiększenie wartości **tresh** wpływa na głębokość rekurencji. Mniejsza wartość powoduje dokładniejszą analizę, gdyż generowanie większej liczby promieni wpływa na prawdopodobieństwo znalezienia obiektu. Z drugiej strony jednak zwiększa liczbę przypadków brzegowych, tj. próbkowanie brzegów obiektów przy których wystąpić mogą błędy numeryczne. Związane są one z reprezentacją promienia i odcinka, jak również z błędami obliczeń. Konsekwencją może być ominięcie przysłaniającego obiektu i dodanie do zbioru elementów, które są niewidoczne. Przykładem takiej sytuacji jest próbkowanie otoczenia wierzchołków trójkątów należących do triangulacji powierzchni.

Zwiększenie wartości $tresh$ zmniejsza prawdopodobieństwo trafienia obiektu, jednak jednocześnie znacznie przyśpiesza działanie całego algorytmu.

Warto zauważyć, że zmniejszenie maksymalnej głębokości rekurencji o k wiąże się ze zwiększeniem $tresh$ 2^k razy, a zastosowana wartość $tresh$ w **GVS** wynosi w przybliżeniu $2^{-14} \approx 6 \cdot 10^{-5}$.

W celu porównania algorytmów dla różnych wartości $tresh$ można określić prawdopodobieństwo trafienia obiektu „leżącego na krawędzi". Rozważając płaszczyznę wyznaczoną przez próbkowany odcinek i punkt obserwatora szansa trafienia obiektu jest proporcjonalna do rozmiaru jego rzutu (rysunek 16). Poniżej zdefiniowano *Rzut obiektu na widoczny odcinek*. Jest to podobna konstrukcja widoczności między dwoma od-

cinkami w dualnej przestrzeni promieni jaka została opisana w rozdziale 2.3.2.

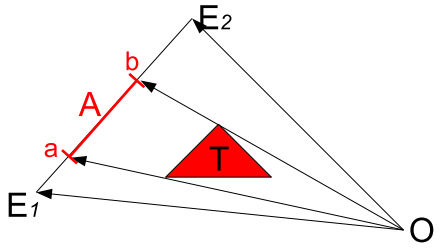
Niech:

- E_1E_2 *odcinek o długości 1*
- O *- punkt początkowy promieni.*
- $r(t)$ *- promień o początku* O *przecinający krawędź* E_1E_2 *w punkcie*

$$E(t) = E_1 + t \cdot (E_2 - E_1)$$

Odcinek wyznaczony przez punkty $E(a)$ *i* $E(b)$ *jest rzutem obiektu* T *na odcinek* E_1E_2 *wtedy i tylko wtedy gdy* a *i* b *są odpowiednio minimalnym i maksymalnym elementem zbioru:*

$$A_T = \{\ t\ :\ r(t)\ przecina\ T\ \}$$



Rysunek 16: Rzut obiektu T na odcinek E_1E_2 z punktu O

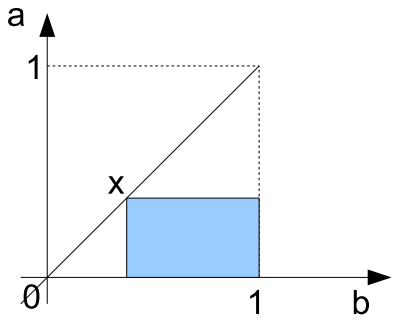
Z powyższej definicji wynika, że promień $r(x)$ trafia we wszystkie obiekty T_i, których rzut na odcinek spełnia warunek: $0 < a_i < x < b_i < 1$. Stąd z szansą znalezienia obiektu związemy wartość P_x - prawdopodobieństwo trafienia rzutu T_i:

$$P_x = 2 \cdot P_{a_i<x} \cdot P_{x<b_i} = 2x \cdot (1 - x)$$

Interpretacja graficzna jest następująca. Na układzie współrzędnych (b,a) każdy punkt (b_i, a_i) reprezentuje odcinek między $E(a_i)$ a $E(b_i)$. Wprowadzając ograniczenia: $a_i, b_i > 0$ i $a_i, b_i < 1$ oraz $a_i < b_i$, wszystkie możliwe odcinki zawarte są w trójkącie wyznaczonym przez równania: $a < b$, $a < 1$, $b > 0$. Zbiór trafianych odcinków przez promień $r(x)$ jest prostokątem o bokach x i $1 - x$, zaczepionym górnym lewym narożnikiem w punkcie (x, x) (rysunek 17)

Rozwiązaniem problemu znajdowania najlepszego promienia (tj. o największej szansie trafienia w obiekt) jest promień $r(1/2)$. Badając funkcję $p(x) = 2x \cdot (1 - x)$ można wykazać, że osiąga ona maksimum w punkcie $x = 1/2$ i wynosi $P_x = 0.5$. Co ostatecznie również uzasadnia wyznaczanie podziału na pół.

Zwiększenie liczby promieni zwiększa szanse trafienia obiektu. Równocześnie zmniejsza się prawdopodobieństwo pominięcia obiektu (przeciwne zdarzenie). Można rozważyć jedynie promienie wychodzące z danego punktu i przecinające badaną krawędź.



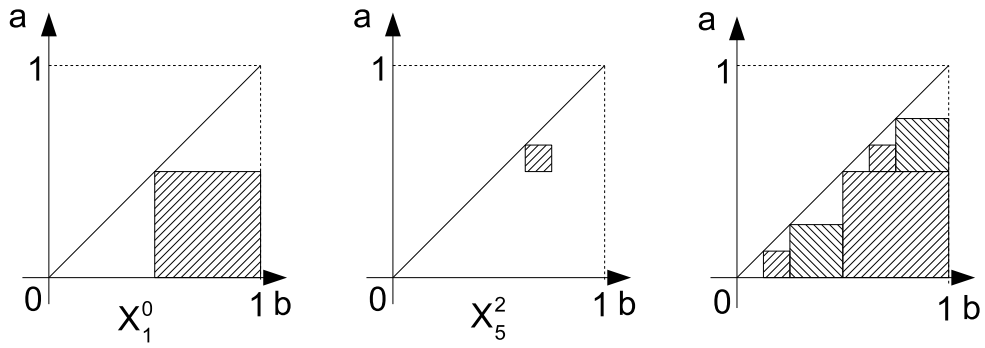
Rysunek 17: Interpretacja graficzna prawdopodobieństwa trafienia przez $r(x)$

Na ich podstawie możemy określić prawdopodobieństwo trafienia obiektu podczas próbkowania adaptacyjnego krawędzi. Poniżej pokazujemy, że zmniejszenie głębokości rekurencji d o 1 zwiększa szanse pominięcia o co najwyżej 2^{1-d}.

Niech A będzie rzutem obiektu T na odcinek E_1E_2 zakładając, że

- $|E_1E_2| = 1$
- promienie w końce odcinka trafiają w różne obiekty. Jest to warunek konieczny dla zainicjowania kolejnych podziałów

Niech X_i^k będzie zbiorem odcinków o długości co najwyżej 2^{-k} przecinanych przez promień r_i^k) (rysunek 18).



Rysunek 18: Interpretacja graficzna zbiorów X_i^k. Po prawej zbiór wszystkich odcinków trafianych przez promienie przedstawiony jako suma zbiorów X_i^k.

Zauważmy, że podczas działania algorytmu z każdym wywołaniem (podziałem) możemy określić zbiór X_i^k. Wraz z pierwszym wywołaniem analizy krawędzi wyznaczony zostaje promień $r_1^0 = r(0.5)$. Zbiór rzutów (tj. odcinków), które przecinane są przez r_1^0 zawiera wszystkie elementy (a, b), takie, że $a < 0.5$ i $b > 0.5$ bez ograniczenia długości odcinków. Interpretacja graficzna takiego zbioru przedstawiona jest na rysunku 18. Wywołanie procedury dla lewej i prawej części inicjuje budowanie zbiorów

X_1^1 i X_2^1. Ograniczenie długości odcinków w nich zawartych determinuje uwzględnienie tylko tych, których nie przecina r_1^0. W kolejnych krokach sytuacja się powtarza. Mianowicie wywołania rekurencyjne na poziomie k budują rozłączne zbiory $X_1^k, X_2^k, ..., X_{2_k}^k$. Tym sposobem zbiór wszystkich trafianych odcinków można podzielić na zbiory rozłączne. Jeśli: X - zbiór wszystkich odcinków trafianych przez promienie r_i^k, to

$$X = \bigcup_{i \in K, j \in D} X_i^j \tag{13}$$

Niech Z_i^k określa zdarzenie przecięcia promieniem r_i^k odcinka o długości co najwyżej 2^{-k}, czyli odcinków zawartych w X_i^k. Wtedy:

$$P(Z_i^k) = S_i^k / S \tag{14}$$

gdzie S_i^k jest polem powierzchni zawierającej odcinki X_i^k i wynosi: $S_i^k = (2^{-k})^2$, a S jest polem powierzchni zawierającej wszystkie odcinki. (patrz interpretacja graficzna na rysunku). Rozłączność zbiorów X_i^k implikuje niezależność zdarzeń Z_i^k. Więc

$$P(Z_i^k \cap Z_j^l) = 0 \quad i \neq j \ \vee \ k \neq l \tag{15}$$

Zatem prawdopodobieństwo trafienia w odcinek wynosi przy maksymalnym poziomie rekurencji d:

$$P_d = P(\bigcup_{i \in K, j \in \{1..d\}} Z_i^j) = \sum_{k=1}^{d} \sum_i P(Z_i^k) \tag{16}$$

Zmiana maksymalnej głębokości wywołań rekurencyjnych o 1 powoduje zmianę powyższej wartości:

$$P_{d-1} = \sum_{k=1}^{d} \sum_i P(Z_i^k) = P_d - \sum_i P(Z_i^d) \tag{17}$$

Szacując sumę po prawej stronie mającą maksymalnie 2^d składników:

$$\sum_i P(Z_i^d) \leq \sum_{i=1}^{2^d} S_i^d / S = 2 \cdot \sum_i (2^{-d})^2 = 2^{1-d} \tag{18}$$

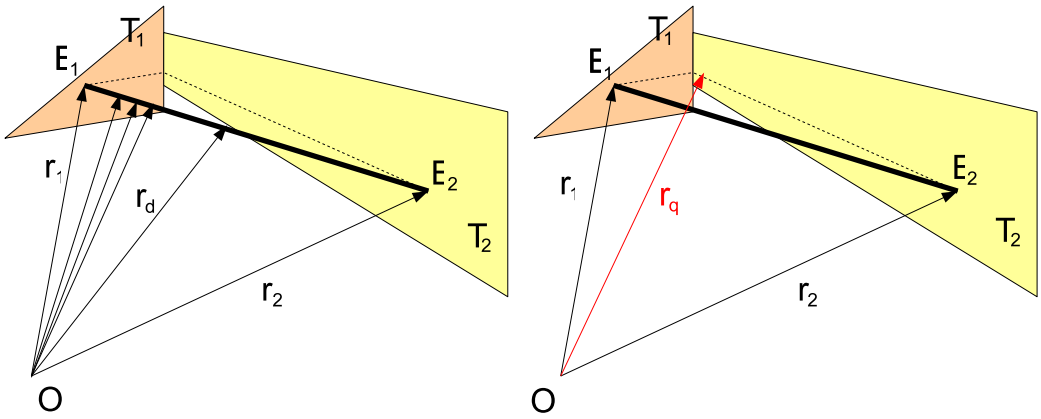
Biorąc pod uwagę fakt, że podczas działania algorytmu poziom, na którym kolejne podziały nie są już uwzględniane, nie zawsze jest maksymalny, P_d maleje o $s \cdot 2^{-d}$, gdzie s to liczba wystąpień maksymalnych głębokości rekurencji tj. *de facto* liczby spełnień warunku $\mathbf{|hit(x) - hit(y)| < tresh}$.

4.3.2 Odległość odcinka

Prostą modyfikacją algorytmu jest heurystyka polegająca na uwzględnieniu odległości odcinka od obserwatora. Idea polega na generowaniu większej liczby próbek w obiekty bliższe. Jeśli l jest odległością środka odcinka E_1E_2 od punktu O to wartość $tresh$ zostaje zwiększona przez l, tj. $tresh = tresh \cdot l$. W przeciwieństwie do jednakowego ograniczenia maksymalnego poziomu wywołań, uwzględnianie odległości ma wpływ na nią dynamiczny. Mianowicie maksymalna głębokość rekurencji zmniejszy się o $k = \lceil log_2 l \rceil$

4.3.3 Przyległe obiekty

Z analizy algorytmu podziału odcinka i rekurencyjnego próbkowania jego części można wywnioskować, iż maksymalny poziom rekurencji zostanie osiągnięty przynajmniej raz, wtedy gdy końce odcinka E_1E_2 wskazują różne obiekty. Rozważając przypadek tylko dwóch obiektów przyległych widocznych dla obserwatora z punktu O (przedstawiony na rysunku 19) wygenerowanych zostanie d promieni, gdzie d to maksymalny poziom wywołań algorytmu. Test czy dwa obiekty są przyległe (tj. mogą być traktowane jako jeden obiekt zasłaniający scenę) sprowadza się sprawdzenia jednego z promieni skierowanych w okolice ich brzegów. Należy zwrócić uwagę, że dwa obiekty są przyległe względem punktu obserwatora analogicznie do nieciągłości między dwoma obiektami, która jest widoczna w zależności od obszaru patrzenia.



Rysunek 19: Próbkowanie krawędzi E_1E_2 dla obiektów przyległych wyznaczających obszar niewidoczny za nimi. Po lewej rekurencyjny podział odcinka. Po prawej promień r_q skierowany w otoczenie lewego obiektu trafia w prawy, mu przyległy.

Warto zauważyć, że ostatnie dwa wygenerowane promienie przez algorytm rekurencyjnego podziału wyznaczają właśnie dwie próbki brzegowe. Do sprawdzenia czy obiekty sąsiadują ze sobą na linii badanego odcinka E_1E_2 wystarczy więc **jeden promień**: Niech:

- $E(t) = E_1 + t(E_2 - E_1)$ będzie równaniem parametrycznym odcinka E_1E_2,
- $r(t)$ będzie promieniem trafiającym w punkt $E(t)$ z punktu O,
- $v(r)$ będzie obiektem trafionym przez promień r.

Oznaczając obiekty trafiane przez promienie $r(0)$ i $r(1)$ jako T_1 i T_2 (rysunek 19) oraz definiując dwa zbiory:

$$\begin{array}{rcl} L & = & \{t \ : \ v(r(t)) \ = \ T_1)\} \\ R & = & \{t \ : \ v(r(t)) \ = \ T_2)\} \end{array} \tag{19}$$

możemy wywnioskować, że dwa obiekty widziane z O są przyległe gdy $L_{max} = max(L) \geq min(R) = R_{min}$. Zatem w celu sprawdzenia czy dwa elementy tworzą spójny obiekt zasłaniający scenę dla obserwatora w punkcie O wystarczy sprawdzić czy:

$$v(r(L_{max} + \epsilon)) = T_2 \ \ \textbf{albo} \ \ v(r(R_{min} - \epsilon)) = T_1$$

Podczas testu wystarczy sprawdzenie jednego z powyższych warunków. Graficzna interpretacja tak wygenerowanego promienia jest widoczny na rysunku 19 (promień r_q).

Na podstawie powyższej analizy można stworzyć algorytm, którego idea opiera się na generowaniu promieni w punkty brzegowe znajdowanych obiektów. Agresywny algorytm próbkowania w kierunku krawędzi można opisać następująco:

```
process_edge(pl,pr)
   check_discountinuity((xp, pr-xp));
   m = pl;
   while ( |m - pl| < |pr - pl| - tresh) {
      check_discountinuity((xp, m-xp));
      if v(m) = v(pr)
         return;
      m = edgepoint(v(m), m, pr, o)
   }
```

gdzie $edgepoint(v(M), M, P_r, O)$ jest algorytmem wyznaczania punktu z brzegu obiektu $v(M)$, który leży w wycinku płaszczyzny $\angle MOP_r$. Powyższy algorytm wyznacza promienie brzegowe z prawej strony dla kolejno znajdowanych (od lewej) obiektów. Dochodząc ostatecznie do obiektu wskazywanego przez promień przecinający prawy koniec odcinka. Algorytm zostanie opisany w rozdziale 4.3.5.W celu zwiększenia prawdopodobieństwa znalezienia obiektu przed krawędzią prostą modyfikacją jest wymuszenie uwzględnienia promienia w środek odcinka, np. poprzez zmianę metody *enlarge* tak, by generowała dla każdej krawędzi dodatkowy punkt w jej środku.

Wyznaczenie promienia w *reverse samping* jest analogiczne do obliczenia punktu brzegowego. Na rysunku 20 przedstawiono znajdowanie promienia i punktu, który leży w zadanej płaszczyźnie. Obydwa problemy sprowadzają się do wyznaczenia przecięcia krawędzi trójkąta z dowolną płaszczyzną.

4.3.4 Wyznaczanie przecięcia

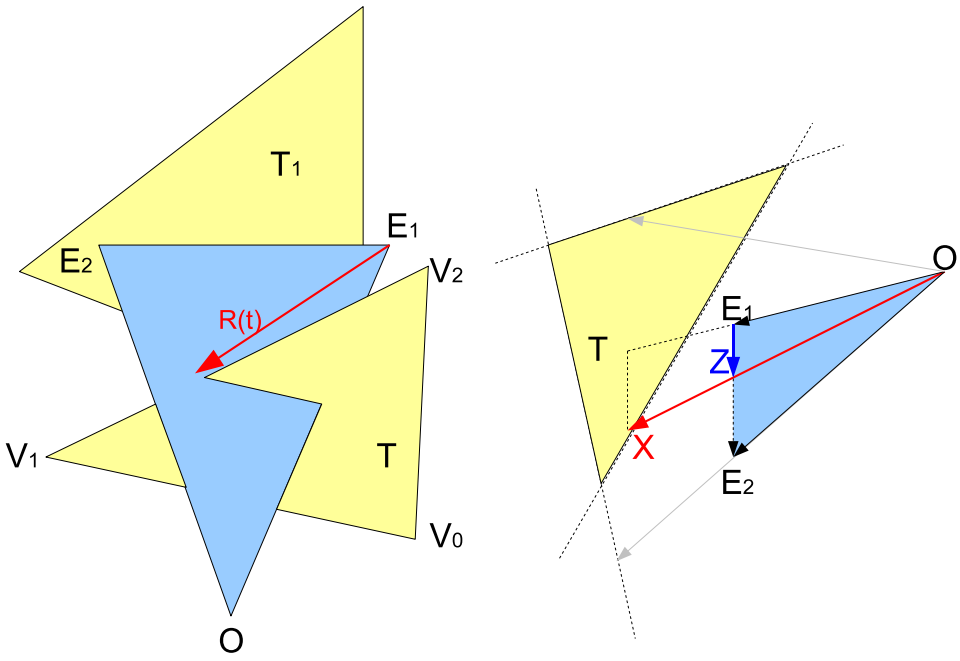
Przecięcie krawędzi trójkąta T z zadaną płaszczyzną zostało zilustrowane na rysunku 20. Intuicyjnym rozwiązaniem jest znalezienie przecięć trzech promieni $V_0\vec{V}_1$, $V_1\vec{V}_2$ $V_2\vec{V}_0$ z trójkątem ΔOE_1E_2 poprzez użycie np. algorytmu Möller'a [8] lub przecięć dwóch trójkątów ΔOE_1E_2 i $\Delta V_0V_1V_2$ [9]. Jednak wiążą się z tym niepotrzebne obliczenia (np. współrzędnych parametrycznych trójkąta znalezionego przecięcia). Należy zatem rozwiązać zadanie algebraicznie, podobnie jak zrobili to Möller i Trumbore [8].

Równanie trójkąta T w postaci parametrycznej jest następujące:

$$T(u,v) \;=\; (1-u-v)V_0 + uV_1 + vV_2 \tag{20}$$

Zauważmy, że interesują nas punkty na krawędziach, więc rozpatrując trzy przypadki $u=0$, $v=0$, $u+v=1$ otrzymujemy trzy równania z jednym parametrem. Każde sprowadza się do równania odcinka (zakładając $V_3 = V_0$) :

$$T(u) \;=\; A + u\cdot(B-A) \;\; gdzie: \;\; A = V_i,\; B = V_{i+1},\; i = \; 0,1,2 \tag{21}$$



Rysunek 20: Znajdowanie promienia między obiektami T_1 i T, leżącego na płaszczyźnie wyznaczonej przez punkty O, E_1, E_2. Po lewej: *reverse sampling* w którym $O\vec{E}_1$ przecina T. Po prawej: uogólnienie problemu dla dowolnych O, E_1, E_2, punkt Z na prostej zawierającej E_1E_2 określa promień trafiający w **punkt brzegowy** X

Zgodnie z przedstawioną sytuacją na rysunku 20, wektor przechodzący przez X przecina odcinek E_1E_2 w punkcie Z. Można go przedstawić jako przesunięcie E_1 o wektor $E_2\vec{E}1$ przemnożony przez skalar s:

$$\begin{aligned} Z(s) &= E_1 + sE \\ E &= E_2 - E_1 \end{aligned} \tag{22}$$

Oczywistym jest, że X leży na prostej przechodzącej przez O i Z, więc:

$$\begin{aligned} X(t) &= O + t \cdot (Z - O) \\ X(t,s) &= O + t \cdot (E_1 + sE - O) \end{aligned} \tag{23}$$

Otrzymujemy równanie $T(u) = X(t,s)$, któremu odpowiada:

$$\begin{aligned} A + u(B - A) &= O + t \cdot (E_1 + sE - O) \\ ts(-E) + u(B - A) + t(O - E_1) &= O - A \end{aligned} \tag{24}$$

podstawiając $r = ts$, $X_{BA} = B - A$, $Y = O - E_1$, $T_A = O - A$ równanie 24 można przedstawić jako:

$$[-E, X_{BA}, Y] \begin{bmatrix} r \\ u \\ t \end{bmatrix} = T_A \tag{25}$$

Rozwiązaniem powyższego równania (korzystając z zasady Cramera dla układów $n \times n$) jest:

$$\begin{bmatrix} r \\ u \\ t \end{bmatrix} = \frac{1}{|-E, X_{BA}, Y|} \begin{bmatrix} | & T_A, & X_{BA}, & Y & | \\ | & -E, & T_A, & Y & | \\ | & -E, & X_{BA}, & T_A & | \end{bmatrix} \tag{26}$$

Obliczanie wyznacznika $|A, B, C|$ (macierzy 3×3) można zrobić na wiele sposobów używając iloczynu wektorowego i skalarnego (dowód jest trywialny, chociażby poprzez rozpisanie lewych stron poniższych równań metodą Sarrusa)

$$\begin{aligned} |A, B, C| &= (A \times B) \cdot C \\ |A, B, C| &= -(A \times C) \cdot B \\ |A, B, C| &= -(C \times B) \cdot A \end{aligned} \tag{27}$$

Korzystając z zależności (27) otrzymujemy ostatecznie równanie:

$$\begin{bmatrix} r \\ u \\ t \end{bmatrix} = \frac{1}{(E \times Y) \cdot X_{BA}} \begin{bmatrix} (T_A \times X_{BA}) \cdot Y \\ (E \times Y) \cdot T_A \\ (T_A \times X_{BA}) \cdot E \end{bmatrix} = \frac{1}{P \cdot X_{BA}} \begin{bmatrix} Q_{AB} \cdot Y \\ P \cdot T_A \\ Q_{AB} \cdot E \end{bmatrix} \tag{28}$$

gdzie $Q_{AB} = T_A \times X_{BA}$ i $P = E \times Y$.

Bazując na powyższym równaniu oraz założeniu $r = ts$, możemy wyznaczyć wartości parametru s:

$$s \;=\; r \cdot \frac{1}{t} \;=\; \frac{Q_{AB} \cdot Y}{P \cdot X_{BA}} \cdot \frac{P \cdot X_{BA}}{Q_{AB} \cdot E} \;=\; \frac{Q_{AB} \cdot Y}{Q_{AB} \cdot E} \tag{29}$$

W powyższym wywodzie dolne indeksy AB i A zostały tak dobrane by podkreślić zależność od wyboru wierzchołków trójkąta T (patrz równanie 21). Mając do wyboru dwa wierzchołki z trzech otrzymujemy trzy wartości s_i $(i = 0, 1, 2)$. Aby nie wyznaczać wszystkich możliwych należałoby określić która wartość s_i jest tą właściwą, inaczej mówiąc, które wierzchołki trójkąta należy wziąć pod uwagę. Wystarczającym testem jest zbadanie parametru u występującego w zmodyfikowanym równaniu trójkąta (21). Wybór wierzchołków prowadzi do poprawnego rozwiązania gdy $0 < u < 1$, wtedy punkt leży na krawędzi opisanej wzorem (21). Wartość u otrzymujemy z równania (27):

$$u \;=\; \frac{P \cdot T_A}{P \cdot X_{AB}} \tag{30}$$

Warto zauważyć, że P nie zależy od wierzchołków trójkąta T i wystarczy wyznaczyć go tylko raz. Zatem punkty A i B są dobrze określone gdy:

$$0 < P \cdot T_A \quad \wedge \quad P \cdot T_A < P \cdot X_{AB}$$

lub

$$P \cdot T_A < 0 \quad \wedge \quad P \cdot X_{AB} < P \cdot T_A$$

Należy zwrócić uwagę, że istnieją co najwyżej dwa punkty leżące na przecięciu krawędzi trójkąta i płaszczyzny (oczywiście poza zdegenerowanymi przypadkami gdy krawędź leży w całości na płaszczyźnie). W celu zastosowania algorytmu do *reverse sampling* należy brać pod uwagę wynik z $s > 0$.

4.3.5 Algorytm

Dane:

- $V[0], V[1], V[2]$ - wierzchołki trójkąta (dla uproszczenia $V[3] = V[0]$)
- $E2$, $E1$, O - punkty określające powierzchnie.

Wynik:

- wartość parametru s na podstawie którego obliczamy promień $R(t)$ trafiający w krawędź trójkąta:

$$\begin{aligned} R(t) &= O + t \cdot (Z - O) \\ Z &= E1 + s \cdot (E2 - E1) \end{aligned} \tag{31}$$

Czas działania Opisany algorytm jest szybszy od metody opartej o znajdowanie przecięcia krawędzi z trójkątem. Średnie czasy działania algorytmów są o 30% lepsze mimo optymalizacji w znajdowaniu przecięcia Mölera polegającej na nie wyznaczaniu parametrów u i v. Wynika to z następującej własności:

> wyznaczanie zamiast punktu na brzegu trójkąta promienia przez niego przechodzącego jest efektywniejsze.

Algorytm:

```
edgepoint(V, E1, E2, O)
   edge  = E2 - E1
   edge1 =  O - E1
   edge2 =  O - E2
   pvec  = CROSS(edge, edge1)

   for (i = 0; i < 3; i++) {
      X = V[i+1] - V[i];
      T = O - V[i];
      l = DOT(pvec, T)
      m = DOT(pvec, X)

      if ( 0 < l && l < m )  ||  (l < 0 && m < l ) ) {
         qvec = CROSS(T,X)
         m = DOT(qvec, edge)
         if (m != 0) {
            s = DOT(Q, edge1) / m
            if (s > 0)
              return s;
         } else
            return 0
      }
   }
   return 0;
```

4.4 Obiekty wypukłe

Opublikowany algorytm GVS operuje na scenie zawierającej jedynie trójkąty, jednak już po krótkiej jego analizie można dojść do wniosku, że można go uogólnić na dowolne typy obiektów. Pierwszym krokiem jest **zdefiniowanie otoczenia** danego elementu sceny, które jest widoczne z badanego punktu obserwatora. Kolejnym i za razem ostatnim jest ustalenie sposobu wyznaczania promienia dla **próbkowania wstecznego**. Mianowicie takiego, który penetruje obszar między dwoma obiektami, omijając je jak najbliżej zgodnie z ideą algorytmu. O ile zbiór punktów z otoczenia t' nie wymaga założeń co do elementów sceny, to w przypadku analizy krawędzi wymagane jest, by trafiane przez ABS obiekty były wypukłe. Zgodnie z poniższą definicją takiego obiektu:

Obiekt wypukły. *Obiekt zawierający w całości odcinki między dowolnie wybranymi punktami obiektu*

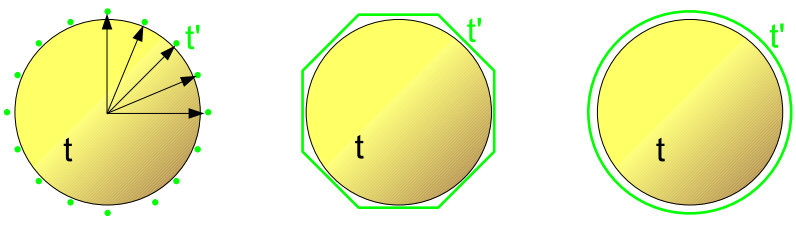
Algorytm próbkowania otoczenia, w szczególności warunek stopu kiedy oba końce wskazują ten sam obiekt, zawęża elementy sceny do **obiektów wypukłych**. Przykładem takiego obiektu jest kula.

4.4.1 Kula

Jednym z praktycznych zastosowań innych obiektów niż trójkąt jest użycie kuli ze względu na:

- szybkie znajdowanie przecięcia promienia ze sferą
- zmniejszenie złożoności sceny poprzez wyeliminowanie triangulacji sfery i zastąpienie kilkunastu (a nawet kilkudziesięciu) trójkątów jednym obiektem.
- wygląd wyrenderowanej kuli.

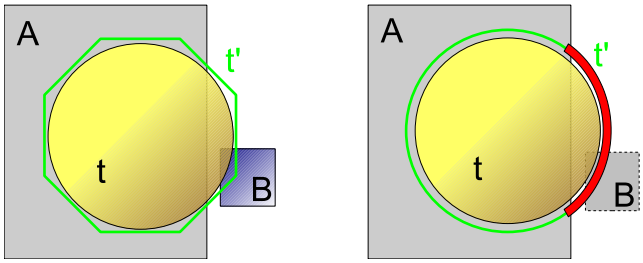
Algorytm ABS polega na próbkowaniu otoczenia obiektu, które jest wyznaczone w niewielkiej odległości od obiektu. W celu analizy otoczenia kuli należy wyznaczyć zbiór punktów E_{points} i krawędzi E_{edges}, takich, że promienie z obszaru obserwatora (a właściwie aktualnie rozpatrywanego punktu położenia) w ich kierunku omijały możliwie jak najbliżej obiekt. O ile wyznaczenie elementów E_{points} jest dość trywialne, to w przypadku krawędzi tj. stycznych wydaje się być bezcelowe, a rozsądniejszym wydaje się rozpatrzenie łuków (rysunek 21).



Rysunek 21: Przykładowe otoczenie kuli widziane z punktu obserwatora. Od lewej: zbiór punktów, odcinki, łuki.

Jednak analiza łuków, czy nawet całego okręgu będącego otoczeniem kuli, mija się z ideą rekurencyjnego podziału „krawędzi". W przypadku odcinka można założyć,

że trafienie tego samego obiektu w końcach odcinka (tj. promieni skierowanych w jego końce) jest warunkiem stopu. Z definicji obiektu wypukłego, a z założenia tylko takie występują w scenie, odcinek między dowolnymi jego dwoma punktami jest w nim zawarty. W przypadku wycinka łuku zdanie takie jest fałszywe (choć odcinek miedzy końcami łuku jest zawarty) i prowadzić może do zbyt wczesnego zakończenia próbkowania (rysunek 22). Należy zatem pozostać przy otoczeniu złożonego z odcinków stycznych do obiektu.



Rysunek 22: Warunek stopu w rekurencyjnym podziale odcinka lub łuku. Promienie w końce łuku trafiają w obiekt A powodując zakończenie analizowania otoczenia obiektu t' i pominięcia B.

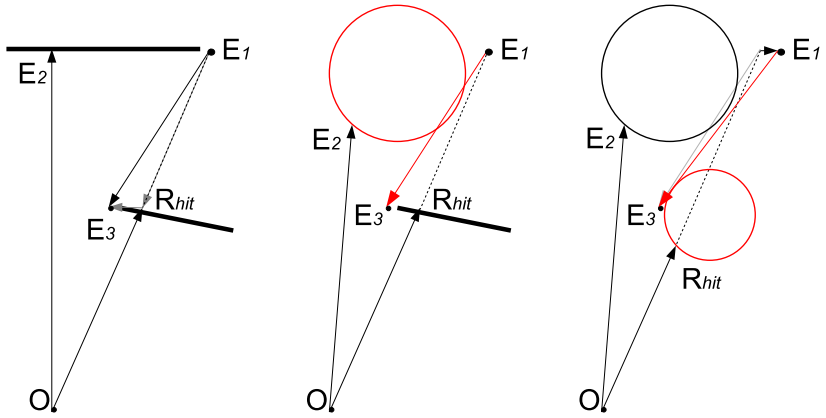
Końce odcinków otoczenia kuli leżą na okręgu. Jest on określony przez obszar styczny między kulą a stożkiem mającym początek w punkcie obserwatora. W związku z czym obliczenia sprowadzają się do wyznaczenia odpowiednich punktów na okręgu.

Z występowaniem kul w scenie, jak również dowolnych brył, wiąże się niezbędna modyfikacja przy wyznaczaniu promienia podczas próbkowania wstecznego. Rozpatrując jedynie trójkąty algorytm polega na znalezieniu promienia w wyznaczonej płaszczyźnie, tak by omijał minimalnie bliższy trójkąt. Z promieniem tym można utożsamić prostą przechodzącą przez dwa punkty:

- należący do otoczenia dalszego obiektu,
- należący do otoczenia obiektu bliższego.

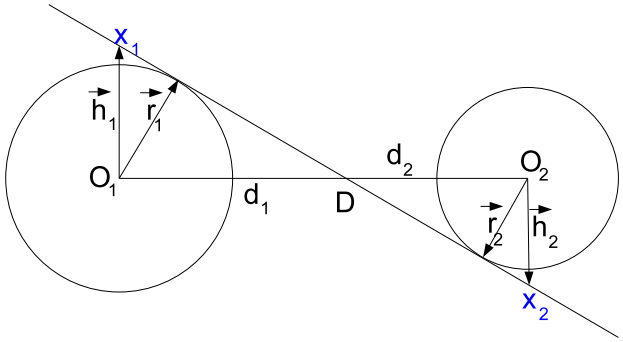
Działanie algorytmu można również przedstawić jako obrót promienia wychodzącego z punktu E_1 aż do momentu trafienia w otoczenie bliższego trójkąta (rysunek 23 po lewej). Pojawia się jednak problem w przypadku kul, ponieważ obrót promienia przyczyni się do przecięcia dalszego obiektu (rysunek 23 w środku). Należy wziąć pod uwagę, iż próba przesunięcia początku promienia (punkt E_1) prowadzi do analogicznego problemu (rysunek 23 po prawej).

Zakładając, że dwa rozpatrywane elementy sceny przez *reverse sampling* są kulami, znalezienie promienia sprowadza się do wyznaczenia prostej stycznej do tychże dwóch obiektów. Dodatkowo, zgodnie z algorytmem, styczna ta leży na płaszczyźnie wyznaczonej przez punkty: O, E_1, E_2 (punkt R_{hit} także leży na płaszczyźnie i może być użyty). Korzystając z faktu, że przecięcie sfery płaszczyzną jest zawsze okręgiem (w szczególnym przypadku punktem, jednak taki przypadek nie zachodzi, ponieważ promień $O\vec{E}_2$ nie wyznacza stycznej w E_2) algorytm sprowadza się do prostego zadania na płaszczyźnie:



Rysunek 23: Wyznaczanie promienia penetrującego przestrzeń między obiektami poprzez przesunięcie. Po lewej: poprawnie wyznaczony promień w przypadku trójkątów, w środku: przecięcie z dalszym obiektem, po prawej: przecięcie z obiektem bliższym po przesunięciu E_1.

Mając dane dwa dowolne okręgi na płaszczyźnie, wyznaczyć prostą styczną do nich, taką, że okręgi leżą po przeciwnych jej stronach



Rysunek 24: Styczna przechodząca między kulami określona przez punkty x_i.

Próbkowanie wsteczne dla kul obliczane jest poprzez wyznaczenie (rysunek 24):

- przecięć kul z płaszczyzną: O_i - rzuty środków kul, r_i - promieni okręgów powstałych w wyniku przecięcia.
- wektorów h_i leżących na płaszczyźnie, prostopadłych do O_1O_2
- punktów x_i będących przesunięciem środków okręgów o wektor $\vec{h_i}$.

Długość h_i oblicza się wzorem:

$$h_i = r_i \sqrt{\frac{d^2}{d^2 - (r_1 + r_2)^2}} \tag{32}$$

gdzie d jest długością odcinka O_1O_2.

Zmiany w algorytmie nie mają wpływu na wyznaczanie promienia wstecznego między dwoma trójkątami. Używając wzoru 32 do obliczeń dla konstrukcji stycznej dla kuli i trójkąta należy dodatkowo określić środek O_i i promień r_i okręgu:

- gdy obiekt dalszy jest trójkątem: $O_i = E_1, r_i = 0$,
- gdy obiekt bliższy jest trójkątem: $O_i = E_3, r_i = 0$,

Przykładowo dla dwóch trójkątów wartości h_i wyniosą 0, a promień wsteczny zdeterminowany jest przez środki okręgów, czyli punkty z otoczenia trójkątów.

4.4.2 Inne obiekty

W celu zastosowania algorytmu GVS dla dowolnych obiektów wypukłych, analogicznie jak w przypadku kul należy zdefiniować otoczenie widziane z zadanego punktu obserwatora oraz określić wyznaczanie promienia wstecznego. O ile otoczenie jest problem prostym, to próbkowanie wsteczne między różnymi obiektami może okazać się zadaniem nietrywialnym.

4.4.3 Badanie nieciągłości

Algorytm *reverse sampling* zastosowany w GVS występuje w szczególnych przypadkach działania algorytmu próbkującego otoczenie obiektu. Poprzez określenie przewidywanego punktu przecięcia promienia (ang. *predicted hit*) a następnie wyznaczenie rzeczywistego przecięcia ze sceną, możliwe stało się badanie luk - nieciągłości sceny. Mianowicie przerwa między dwoma obiektami może być niewidoczna w zależności od miejsca z którego widzi ją obserwator. Sprawdzając odległość miedzy przewidywanym a rzeczywistym punktem przecięcia zostaje zainicjowane szukanie promienia penetrującego obszar między dwoma elementami sceny.

Zastosowanie różnych obiektów determinuje różne „rodzaje" wyznaczania otoczenia. W związku ze znanym przewidywanym punktem przecięcia (punkt w otoczeniu lub w płaszczyźnie zwierającej obiekt) możliwe jest badanie nieciągłości sceny. W przedstawionym algorytmie analiza taka odbywa się jedynie w przypadku krawędzi, choć mogłaby być również przeprowadzona dla punktów:

- otoczenie składające się jedynie z punktów uniemożliwia inicjowanie *reverse sampling*,
- w punktach może występować nieciągłość, która nie zostaje wykryta przy badaniu krawędzi,
- nieciągłość zostanie pominięta gdy wyznaczony penetrujący ją promień nie przecina obszaru obserwatora. Badanie nieciągłości w punkcie zwiększa prawdopodobieństwo analizy luk niewielkim kosztem.

Algorytm ABS można zmodyfikować następująco:

```
adaptive_border_sampling(x)
   t' = enlarge(v(x), e)
   for each p in t' {
      handle_ray((xp, p-xp)
      check_discountinuity((xp, p-xp));
   }
   ...
```

Dodatkowo należy uwzględnić fakt, że przewidywanego punktu przecięcia nie da się poprawnie określić dla promieni losowych. Również może okazać się to niemożliwe dla promieni punktowych z ABS. Również definicja *predicted hit* z pracy [3] jako przecięcie promienia przechodzącego przez otoczenie t' z płaszczyzną zawierającą trójkąt t może prowadzić do wartości w „nieskończoności". Przykładem takiej sytuacji jest obserwator leżący na płaszczyźnie trójkąta. Problem ten można jednak obejść poprzez użycie punktów z otoczenia jako przewidywanego trafienia.

4.5 Promienie losowe i warunek stopu

Promienie losowe wyznaczane są w celu zainicjowania algorytmu próbkowania adaptacyjnego. W zaproponowanym rozwiązaniu GVS jedyną kontrolą nad ich liczbą są dwa warunki stopu. Pierwszy określa maksymalną liczbę promieni. Drugi jest spełniony, gdy liczba nowo znajdowanych obiektów przypadających na ostatnie n promieni będzie zbyt mała. W [3] jest to 50 obiektów na ostatnie n = 1mln promieni. Próbkowanie losowe używane jest jedynie, gdy pozostałe algorytmy nie wyznaczyły nowych obiektów i kolejnych promieni. Rozważając wielowątkowość można rozpatrzeć następującą sytuację: każdy z k wątków losuje promień, znajduje jego przecięcie następnie używa ABS i *reverse sampling* do wyznaczania PVS. Uzyskanie analogicznego efektu możemy otrzymać poprzez modyfikację GVS, polegającą na początkowym wygenerowaniu k promieni. Powstają więc dwie heurystyki:

1. k promieni na końcu (strategia leniwa **last**) - oryginalne działanie algorytmu GVS z dodatkowym ograniczeniem na liczbę losowych promieni.

2. k promieni na początku (strategia gorliwa **first**) - polegająca na wylosowaniu k promieni inicjujących GVS.

Zakładając, że losowane promienie z 1 i 2 są takie same, np. pseudolosowe z takim samym zarodkiem, występuje istotna różnica podczas działania algorytmu. Mianowicie:

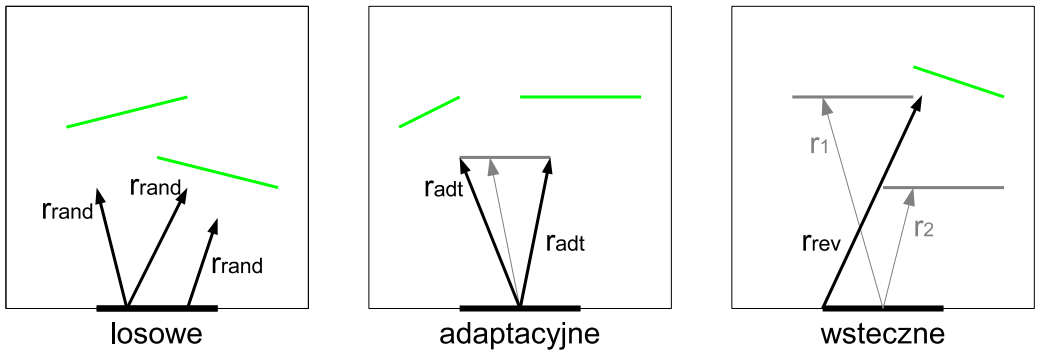
- algorytm w 1 przypadku kończy działanie wyznaczając mniej niż k promieni losowych. Spowodowane jest to spełnieniem jednego z warunków stopu. Warto zauważyć, że przyczynić się to może do pominięcia próbkowania widocznej części sceny. Jest to przypadek, w którym GVS analizuje złożony obszar sceny w danym kierunku (np. wzdłuż ulicy miasta) i osiąga maksymalną liczbę promieni. Pozostawia on niezbadaną część sceny w pozostałych kierunkach, np. przeciwnym.

- wymuszając wygenerowanie dokładnie k losowych promieni (np. określając go jako główny warunek stopu) analizę opisane heurystyki można porównać pod

względem liczby zarodków. Zarodkami są punkty w obszarze obserwatora, w których mają początek promienie GVS. Ich liczba i różnorodność wpływa na budowane otoczenie widocznych obiektów, czyli na przestrzeń analizowaną przez ABS. Należy zwrócić uwagę, że zarodki powstają zarówno przy próbkowaniu losowym i wstecznym. Jednak w strategii *last* jest ich znacznie mniej, gdyż promień losowy trafić może w obiekt już widoczny.

5 Zmiana obszaru obserwatora

Algorytm Petera Wonki *Guided visibility sampling* można przedstawić na przykładzie dwu wymiarowym. Scena będąca płaszczyzną zawiera elementy - np. odcinki. Algorytm **GVS** generując losowe promienie z obszaru obserwatora znajduje odcinki inicjujące poszukiwania widocznych elementów w ich otoczeniu. Adaptacyjne próbkowanie polega na rozszerzeniu odcinka i sprawdzeniu promieni przechodzących przez jego końce, natomiast próbkowanie wsteczne wyznacza promienie omijające elementy, tj. penetrujące nieciągłości między nimi (rysunek 25).



Rysunek 25: Interpretacja algorytmu **GVS** na przykładzie sceny 2-wymiarowej. Przedstawione zastały trzy metody próbkowania.

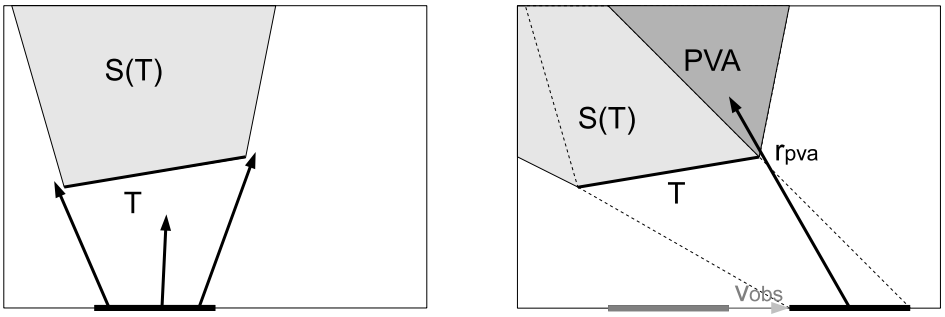
Jedną z zaproponowanych w [3] metod modyfikacji algorytmu GVS jest znalezienie odpowiednich promieni inicjujących zamiast początkowych promieni losowych. Jest to efektywna optymalizacja algorytmu, która powoduje szybsze i dokładniejsze wyznaczanie PVS. W przypadku algorytmów i scen, dla których określamy zbiór obiektów zasłaniających (np. fasady budynków, powierzchnie dróg itd.) odrzucane są elementy przez nie zasłonięte. Takie podejście zostało opisane w 2.3.2 i 2.3.4. Możliwe jest więc zatem zdefiniowanie promieni inicjujących jako tych, które trafiają w wybrane obiekty ograniczające widoczność. Rozważając dowolną scenę i zbiór idealnych inicjujących promieni GVS wyznaczyłby dokładnie wszystkie widoczne obiekty. Obserwacja ta prowadzi do modyfikacji polegającej na rozszerzeniu wyniku PVS o punkty pierwszego trafienia każdego obiektu lub odpowiadające im promienie. Warunek stopu „liczby nowych znalezionych obiektów na ostatnie k promieni" zostałby spełniony po $|PVS| + k$ próbkowaniach. Tak obrany zbiór inicjujących promieni posłużyć może do wyznaczenia PVS dla zmodyfikowanego obszaru obserwatora poprzez:

- przesunięcie - oczekując, że widoczność odległej części sceny zostanie niezmieniona. Ewentualnie zostanie ona przysłonięta. Zatem wynik GVS zawiera podzbiór

poprzedniego PVS oraz zbiór nowych obiektów znalezionych przez algorytmy próbkowania adaptacyjnego i wstecznego.

- powiększenie - dla którego istotna różnica leży w próbkowaniu wstecznym, tj. promieni penetrujących nieciągłości sceny widocznych po rozszerzeniu obszaru obserwatora. Łatwo zauważyć, że jest to szczególny przypadek przesunięcia obszaru, w którym nie zmniejszono poprzedniego zbioru PVS.

Przesunięcie obserwatora (obszaru) o wektor v_{obs} powoduje zmianę zasłanianego regionu (rysunek 26). Z jednej strony część widocznej do tej pory sceny zostaje się zasłonięta, z drugiej powstaje potencjalnie widoczny obszar (**PVA**). W związku z ideą próbkowania do znajdowania PVS generowane są promienie skierowane w PVA, zamiast sprawdzania które z obiektów przestały być widoczne. Szukanie przecięć promieni inicjujących mających trafić w PVS automatycznie spowoduje odrzucenie niewidocznych już obiektów.



Rysunek 26: Obszar $S(T)$ zasłaniany przez obiekt T. Przesunięcie pola widzenia powoduje odsłonięcie niewidocznej do tej pory części sceny PVA (*potentialy visible area*).

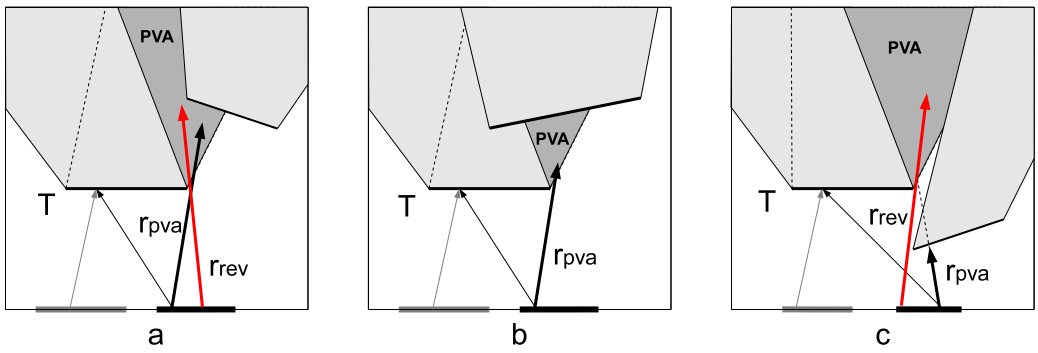
Badanie widoczności obszaru PVA jest analogiczne do adaptacyjnego próbkowania w GVS. Poniżej zostały przedstawione możliwe trafienia promieniem r_{pva}, który omija obiekt T (rysunek 26) i przechodzi przez jego otoczenie. Jeśli v jest funkcją widoczności wówczas:

- $v(r_{pva}) \notin PVS$ - znaleziony został nowy obiekt, dla którego (jak i dla jego otoczenia) widoczność jest badana przez GVS,
- $v(r_{pva}) \in PVS$ - obszar widoczny również z poprzedniego pola widzenia. Określając przewidywane trafienie (ang. *predicted hit*) przez punkt należący do otoczenia rozszerzenia T i sprawdzając odległość rzeczywistego trafienia (*hit*) możliwe jest zbadanie luki.

Przypadek $v(r_{pva}) \in PVS$ jest analogiczny do **próbkowania wstecznego**. Mianowicie w przypadku:

- braku nieciągłości $|predicted\ hit - hit| < tresh$. Znalezione elementy tworzą powierzchnię. PVA dla obiektu T jest zasłonięty całkowicie lub częściowo przez przyległy obiekt $v(r_{pva})$. Zatem analogicznie do metody rozszerzania obiektu zasłaniającego o mu przyległe w algorytmie Schauflera [1] wystarczy zbadać PVA dla obiektu $v(r_{pva})$.

- wystąpienia nieciągłości $|predicted\ hit - hit| \geq tresh$. Obiekt $v(r)$ ogranicza obszar PVA dla T. Zgodnie z analogią do próbkowania wstecznego wystarczy wyznaczyć promień r_{rev} omijający T i $v(r_{pva})$ (rysunek 27).



Rysunek 27: Przypadki nieciągłości w próbkowaniu wstecznym: (a) obiekt $v(r_{pva})$ jest za obiektem T, r_{rev} penetruje PVA; (b) obszar PVA ograniczony przez $v(r_{pva})$, promień r_{rev} nie istnieje; (c) obiekt $v(r_{pva})$ jest przed T.

W algorytmie GVS *reverse sampling* następuje gdy spełniony zostanie warunek $predicted\ hit - hit > tresh$, tj. gdy promień w otoczenie trafia w punkt bliższy niż przewidywany. Jednak w przypadku analizy obszarów PVA promień r_{rev} należy wyznaczyć również dla $predicted\ hit - hit < -tresh$. Powyższe obserwacje prowadzą do algorytmu wyznaczania widocznych obiektów z obszaru obserwatora przesuniętego o wektor v_{obs}:

1. dla każdego obiektu PVS i punktu jego trafienia wyznacz promień r znajdź przecięcie $v(r)$,
2. dla każdego $v(r)$ będącego już w PVS zbadaj obszar PVA oraz ewentualne nieciągłości,
3. wszystkie znalezione obiekty umieść w tymczasowym zbiorze PVS'
4. zainicjuj algorytm GVS promieniami $r : v(r) \notin PVS$ oraz znalezionym już zbiorem PVS'

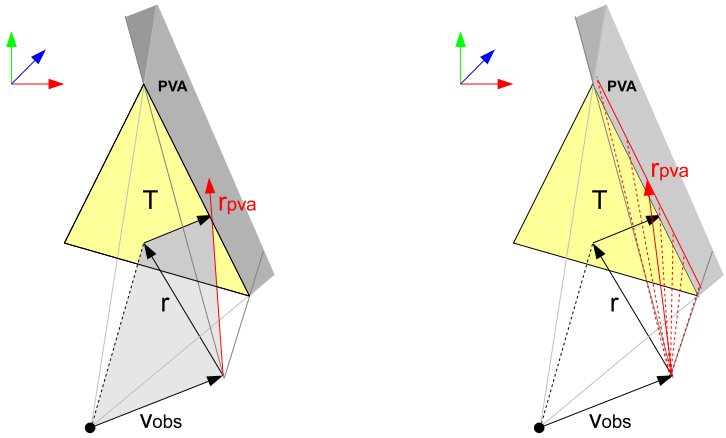
5.1 Próbkowanie odsłoniętego obszaru PVA

Przypadek scen 3D i analiza odsłoniętego w niej obszaru po przesunięciu obserwatora jest bardziej złożony niż opisany wcześniej problem 2D. Choć badanie PVA_T sprowadza się do problemu widoczności z obszaru, to wystarczającymi próbkami są promienie przechodzące otoczenie obiektu T (analogicznie jak pojedyncze promienie w próbkowaniu wstecznym wystarczające są do penetrowania nieciągłości między obiektami). Prowadzi to do zmodyfikowanych algorytmów użytych w GVS

- poszerzania trójkąta (**enlarge**) z uwzględnieniem wektora przesunięcia obserwatora v_{obs},

- próbkowania krawędzi (**subdiv_edge**), w którym zredukowana jest liczba podziałów przez zastosowanie np. promieni w punkty brzegowe.

Na rysunku 28 przedstawiono dwa podejścia do próbkowania odsłoniętego obszaru PVA. Prostym lecz agresywnym podejściem jest generowanie jednego promienia pen-



Rysunek 28: Generowanie promieni potencjalnie trafiających w obszar PVA. Po lewej wyznaczanie pojedynczego promienia na podstawie punktu trafienia w obiekt T i wektora przesunięcia V_{obs}. Po prawej dokładniejsze próbkowanie PVA wzdłuż krawędzi obiektu T.

etrującego PVA (rysunek 28 po lewej). Bazując na prostej obserwacji, w której rozmiar PVA maleje wraz ze wzrostem odległości od obserwatora, rozwiązanie takie jest wystarczające dla odległych obiektów. Ów promień można łatwo wyznaczyć bazując na algorytmie wyznaczania punktu brzegowego (rozdział 4.3.5). Dokładniejszym rozwiązaniem badania odsłoniętego obszaru jest wyznaczenie krawędzi i analogiczne postępowanie jak w adaptacyjnym próbkowaniu otoczenia w GVS. Jednak zamiast stosowania metody poszerzania trójkąta T wystarczy przesunąć krawędzie o ϵ wzdłuż wektora V'_{obs} (V'_{obs} jest rzutem wektora V_{obs} na płaszczyznę zawierającą trójkąt T). Uwzględnić należy te krawędzie e'_i (przesunięte $e_i \in T$ o wektor V_{obs}), które leżą poza T - *de facto* takie krawędzie są co najwyżej dwie, a test jest trywialny (na rysunku 28 po prawej przedstawiono przypadek z jedną krawędzią).

5.2 Algorytm

Idea algorytmu polega na sprawdzeniu czy zmienił się zbiór widocznych obiektów z poprzedniego obszaru, próbkowaniu obszarów odsłoniętych oraz zastosowaniu **GVS** do badania nowych obszarów sceny. Pseudokod algorytmu jest następujący:

```
main(pvs, vobs)
   EPVS = empty;
   for each (t, hit) in pvs
      x = (xp, hit - xp);
      handle_ray(x);
```

```
   GVS.PVS+= EPVS;
   for each (t, hit) in EPVS
      pva_sampling(t, hit, vobs);
   GVS.main()

handle_ray(x)
   if v(x) in pvs
      if v(x) not in EPVS
         EPVS+= v(x);
   else
      GVS.handle_ray(x);

pva_sampling(t, hit, vobs)
   m = edgepoint(t, hit, hit + vobs, o)
   xm = (xp, m - xp)
   GVS.handle_ray(x);
   t' = enlarge(t, vobs);
   for each edge(pl,pr) in t'
      process_edge(pl,pr)

process_edge(pl,pr)
   check_discountinuity((xp, pr-xp));
      m = pl;
      while ( |m - pl| < |pr - pl| - tresh)
         check_discountinuity((xp, m-xp));
         if v(m) = v(pr)
            return;
         m= edgepoint(v(m), m, pr, o)

check_discontinuity(x)
   if |predicted_hit(x) - hit(x)| > tresh
     xn = reverse_sampling(x)
     if start(xn) in view cell
        GVS.handle_ray(xn)
```

gdzie:

- **pvs** i **vobs** są parametrami wywołania algorytmu: poprzedni PVS oraz wektor przesunięcia obszaru obserwatora V_{obs}
- **v(x)** jest pierwszym obiektem przeciętym przez promień x,
- **edgepoint** jest algorytmem wyznaczania punktu brzegowego opisanym w rozdziale 4.3.5,

W wyniku działania algorytmu wyznaczony zostaje zbiór potencjalnie widocznych obiektów wraz z punktami trafienia. Otrzymane dane wystarczające są do dalszych przesunięć obszaru obserwatora.

5.3 Przechodzenie sceny

W problemie przechodzenia sceny i wyświetlania jej z różnych obszarów następuje odrzucanie niewidocznych obiektów. Zakładając dużą liczbę elementów występujących w scenie oraz stosunkowo mały rozmiar widocznych obiektów operowanie na PVS umożliwia przemieszczanie się w scenie w czasie rzeczywistym. Mając dany podział sceny wraz z odpowiednimi PVS , które zostały obliczone w fazie *preprocessingu* obserwator przechodząc z komórki do komórki uwzględnia zredukowaną scenę. W niektórych opublikowanych rozwiązaniach (np. w 2.3.2) rezygnuje się z wcześniejszego przygotowania podziału sceny a zbiór widocznych obiektów wyznaczany jest *ad hoc*. Podejście takie opisane zostało właśnie w pracy Kumara [4]. Dla odpowiednio wyznaczonego obszaru obserwatora czas potrzebny na wyznaczenie PVS dla sąsiednich komórek oraz przesłanie wyniku z serwera do klienta jest wystarczający na selekcje właściwych obiektów. Przy generowaniu obrazu metodami śledzenia promieni należy dodatkowo uwzględnić czas potrzebny na zmodyfikowanie struktury danych (kd-drzewo) lub zbudowanie jej od nowa.

Łącząc opisane wyżej techniki można stworzyć systemy, w których:

- wszystkie obliczenia dokonywane są przed rozpoczęciem poruszania się po scenie,
- podział sceny na komórki i odpowiadające nim zbiory wyznaczane są przed przechodzeniem, a kd-drzewa budowane są *ad-hoc*
- PVS jak i kd-drzewo budowane są na żądanie.

5.3.1 Szybkie budowanie kd-drzewa

Schemat znajdowania PVS i budowy kd-drzewo *ad hoc* jest o tyle ciekawy, że obsługuje on również sceny dynamiczne, w których zbiór PVS zmienia się nie tylko pod wpływem przesunięcia obserwatora, lecz również z powodu zmian w scenie np. poruszających się obiektów. Jednym z opublikowanych efektywnych algorytmów jest równoległe budowanie kd-drzewa opisane w pracy Maxima Shevtova, Alexeia Soupikova i Alexander Kapustina *Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes* [17], którego idea jest następująca:

- w oparciu o algorytm budowania kd-drzewa Havrana [18] dla wierzchołków zawierających co najmniej **32 elementy** stosuje się aproksymacje SAH. Sprowadza się to do regularnego podziału przestrzeni względem **31 płaszczyzn**. Wyznaczają one **32 kubełki**, z którymi związane są liczby zawieranych elementów. Pozwala to na proste wyznaczenie liczby obiektów po każdej stronie płaszczyzny. W przypadku co najwyżej 32 elementów obliczany jest dokładny SAH.
- w trakcie „rozrzucania" obiektów do kubełków stosowane jest adaptacyjne pomijanie elementów. Mianowicie uwzględniony jest jedynie co l-ty element, gdzie $l = log_{10}N$, a N jest liczbą wszystkich obiektów w danym wierzchołku.
- w wyniku podziału wierzchołka płaszczyzną obiekty często trafiają do lewego i prawego poddrzewa. Powoduje to wzrost wymaganej pamięci i uniemożliwia wykorzystanie tego samego obszaru pamięci komputera. Shevtov zaproponował modyfikację zarządzania pamięcią:

- obszary pamięci (kontenery) połączone są w listę, tak, że ostatnio używany kontener jest na końcu listy.
- dla każdego obszaru pamiętany jest wskaźnik początkowy, rozmiar i wskaźnik na koniec. Obszar miedzy wskaźnikami odpowiada używanej pamięci.
- przydzielanie oraz zwalnianie pamięci odbywa się poprzez przesunięcie końcowego wskaźnika o odpowiednią ilość.
- jeśli w kontenerze nie ma wystarczająco miejsca, tworzony jest nowy. Zwykle operacja tworzenia nowego obszaru odbywa się 2-4 razy podczas całej konstrukcji drzewa.

Podczas konstrukcji kd-drzewa „z góry na dół" i budowania najpierw lewego, a później prawego poddrzewa wymagane jest użycie dwóch puli. Odpowiadają one lewym i prawym poddrzewom. Tym sposobem odwoływanie się do pamięci każdej puli odbywa się w ostatnim na liście kontenerze.

Według autorów algorytmu [17] konstrukcja kd-drzewa możliwa jest w czasie 1.7 sekundy dla sceny z 7 milionami trójkątów, 2.4 sekuny dla 10 milionów trójkątów.

6 Implementacja i wyniki

Większość czasu poświęcona pracy została pochłonięta przez implementację oraz testowanie algorytmów. Do reprezentacji scen użyto formatu **mgf**. Zaimplementowane w języku C++ z wykorzystaniem biblioteki STL i OpenGL zostały algorytmy opisane w rozdziale 3 wraz z rozszerzeniami (roz. 4) oraz przeliczanie PVS wraz z szybką budową kd-drzewa (opisane w rozdziale 5). Ważniejszymi z zaimplementowanych algorytmów, które nie zostały opisane, są: wybrane metody budowania i trawersowania kd-drzewa opisane w pracy Havrana [18], podstawowy algorytm śledzenia promieni z modelem oświetlenia Blinna-Phonga. Dla potrzeb testów na rozmaitej gamie modeli zaimplementowano generatory scen. Wszystkie zaimplementowane algorytmy zamieszczone są na dołączonej do pracy płycie CD. Ich opis znajduje się w dodatku A.

Przedstawione w niniejszym rozdziale wyniki algorytmów wyznaczania widocznych obiektów zawierają następujące wartości:

- **rozmiar PVS** - liczba potencjalnie widocznych obiektów, które zostały wyznaczone przez algorytmy,
- **błędne piskele** - średnia liczba błędnych pikseli określająca dokładność obliczonego PVS. Błąd wyznaczony jest na podstawie $n = 1,000,000$ próbek generujących obrazy 1000×1000 (piramida widzenia o kącie $60°$):
  - sceny zawierającej wszystkie obiekty
  - sceny ograniczonej do zbioru PVS.

  Błąd określono przez liczbę promieni, dla których otrzymano różne wyniki przecięcia ze sceną. Na podstawie dziesięciu różnych obrazów, dla których wyznaczono błędną liczbę pikseli został określony średni błąd.

- **liczba promieni** - GVS polega na próbkowaniu sceny promieniami, tj. znajdowaniu ich przecięć. Liczba takich promieni jest ściśle związana z czasem działania algorytmu. Ze względu na różne możliwe (istniejące) algorytmy i implementacje rozwiązujące problem przecięcia promienia ze sceną (np. *OpenRT*, *MLRTA*) czas działania może się znacząco różnić. Stąd niezależnie nawet od mocy obliczeniowej podawana jest liczba generowanych promieni.

- **czas działania** - czas działania zaimplementowanych algorytmów. Testy przeprowadzono na notebooku $ASUSA6RP$ o następującej specyfikacji:

  - procesor: Intel Celeron M 1.6 GHz
  - pamięć RAM: 1 GB
  - karta graficzna: Ati Radeon XPRESS 200M 128MB
  - system operacyjny: linux UBUNTU.

  Należy zwrócić uwagę, że czas śledzenia miliona promieni z wykorzystaniem algorytmów opisanych w pracy Havrana[18] wyniósł nawet 40 s. Zastosowanie MLRTA, wykorzystanie SSE oraz większa moc obliczeniowa pozwala na śledzenie do 800 tys. do 1,2 mln promieni na sekundę, w porywach nawet do kilku milionów. Wyniki takie zostały zaprezentowane w pracy Wonki [3], gdzie użyto komputera: Intel Pentium4 3.2GHz, 4GB RAM.

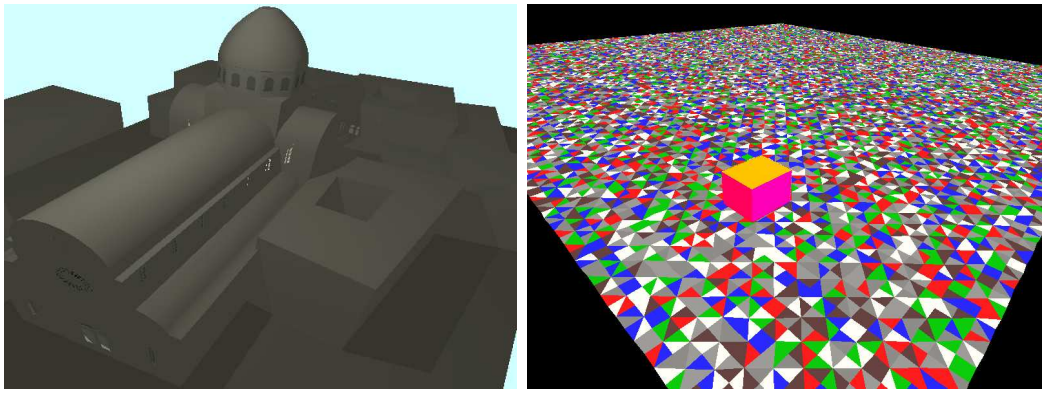
6.1 Strategie próbkowania krawędzi

Rysunek 29: Katedra widziana od wewnątrz.

Opisane algorytmy próbkowania krawędzi wpływają na czas działania jak i na dokładność otrzymanego rozwiązania, tj. zbioru PVS. Czas działania jest ściśle związany

z liczbą generowanych promieni i szukaniu przecięcia w scenie. Zatem redukcja promieni w krawędzie jest najprostszą metodą przyspieszenia GVS. Testy zostały przeprowadzone na scenach:

- **sibenik** - scena katedry. Zawiera około 75000 trójkątów. Uwzględniono dwa obszary obserwatora, z których przykładowe widoki na katedrę widoczne są na rysunkach 29 (ze środka katedry) oraz 30 (z „lotu ptaka").
- **trójkąty** - scena zawierająca płaską powierzchnię złożoną z połączonych ze sobą 20,000 trójkątów. Widok sceny przedstawiony jest na rysunku 30.



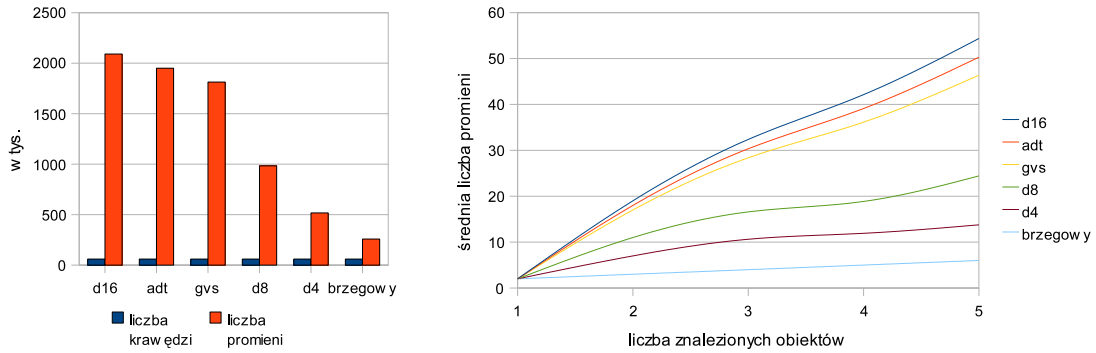
Rysunek 30: Po lewej: katedra z „lotu ptaka". Po prawej: prostopadłościan obserwatora oraz podzielona płaszczyzna na trójkąty

W testach algorytmu GVS zastosowano:

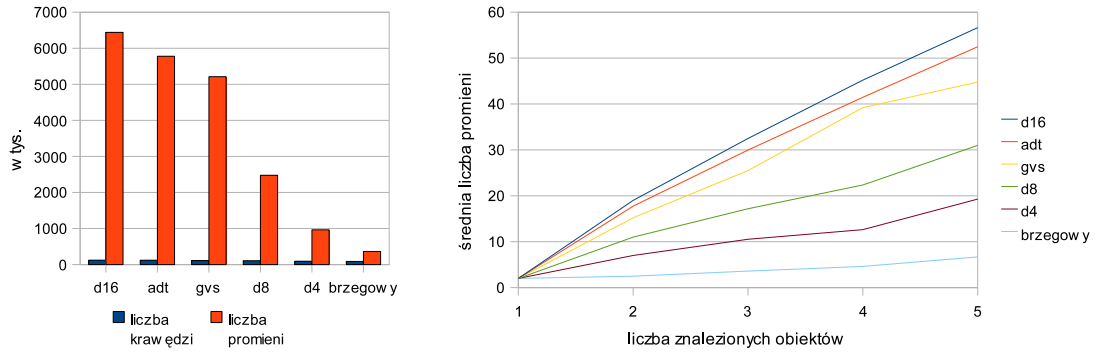
- warunek stopu rozszerzony o maksymalną liczbę losowych promieni - 1 mln próbek.
- następujące modyfikacje algorytmy analizy krawędzi:
  - ograniczenie głębokości wywołań rekurencyjnych (oznaczone: *d4, d8, gvs, d16*), rozdział 4.3.1,
  - uwzględnianie odległości do krawędzi od obserwatora (*adt*), rozdział 4.3.2,
  - wykrywanie przyległych obiektów (*brzegowy*), rozdział 4.3.3,

Na wykresach (rysunki 31, 32, 33) przedstawiono porównanie rozmiaru wyznaczonego PVS z liczbą generowanych promieni w krawędzie (po lewej), oraz ich średnią ilość względem liczby znalezionych obiektów podczas próbkowania krawędzi (po prawej). Tabele 2, 3, 4 zawierają czas wyznaczania poszczególnych PVS wraz z uzyskanym średnim błędem.

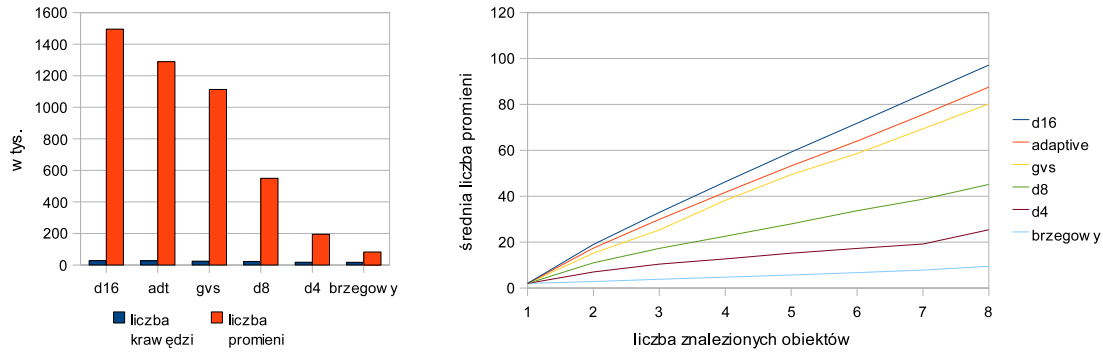
Dokładnym rozwiązaniem widoczności dla sceny siatki trójkątów z umieszczonym nad nią obserwatorem są wszystkie obiekty. Powstałe błędy podczas działania GVS wynikają zarówno z niedokładności reprezentacji promieni i obliczeń. Wraz z większą odległością trójkątów od obserwatora rośnie kąt między wektorem do obserwatora (leżącym na promieniu trafiającym trójkąt) a wektorem normalnym powierzchni co powoduje dodatkowo próbkowanie otoczenia w odległości dużo większej niż obrany ϵ. Na rysunku 34 pokazane zostały nieznalezione trójkąty.



Rysunek 31: Statystyki krawędzi dla sceny siatki trójkątów



Rysunek 32: Statystyki krawędzi dla sceny katedry z obserwatorem wewnątrz niej.



Rysunek 33: Statystyki krawędzi dla sceny katedry z obserwatorem na zewnątrz.

6.2 Obiekty wypukłe

Zastosowanie obiektów wypukłych zamiast powierzchni trójkątów będących triangulacją sfer wpływa na zmniejszenie liczby rozpatrywanych elementów sceny. W związku z czym zmniejsza się znacznie czas potrzebny na wczytanie sceny, budowę kd-drzewa, znajdowanie przecięcia promienia oraz wyznaczanie widocznych obiektów. Poniżej zostały przedstawione wyniki *GVS* z obsługą kul i trójkątów.

Tablica 2: Wynik algorytmu GVS (z ograniczeniem losowych promieni do 10 tys.) dla obserwatora wewnątrz katedry. Rozmiar sceny: 76,650.

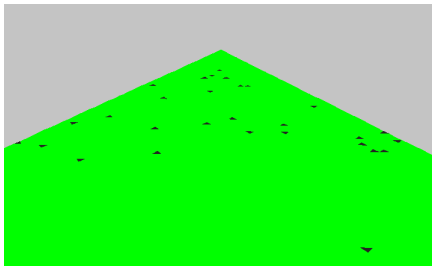
Algorytm	PVS (%)	Liczba iteracji	Błędne piksele	Czas (s)
d4	29,627 (38%)	939,727	44	64.57
gvs	38,206 (49%)	4,213,010	1	286.13
brzegowy	29,642 (38%)	585,874	4	33.09

Tablica 3: Wynik algorytmu GVS (z ograniczeniem losowych promieni do 10 tys.) dla obserwatora na zewnątrz katedry. Rozmiar sceny: 76,650.

Algorytm	PVS (%)	Liczba iteracji	Błędne piksele	Czas (s)
d4	5,530 (7%)	178,086	122	8.26
gvs	8,221 (10%)	938,041	2	40.51
brzegowy	6,016 (7%)	138,136	1	5.95

Tablica 4: Wynik algorytmu GVS (z ograniczeniem losowych promieni do 10 tys.) dla obserwatora nad siatką trójkątów. Rozmiar sceny: 20,000. Wszystkie trójkąty są widoczne.

Algorytm	PVS	Liczba iteracji	Błędne piksele	Czas (s)
d4	19,971	500,597	2	7.02
gvs	19,983	1,210,404	1	16.06
brzegowy	19973	352,121	2	4.36



Rysunek 34: Nieznalezione obiekty (ciemnoszare) z powodu dużej odległości od obserwatora i błędów numerycznych.

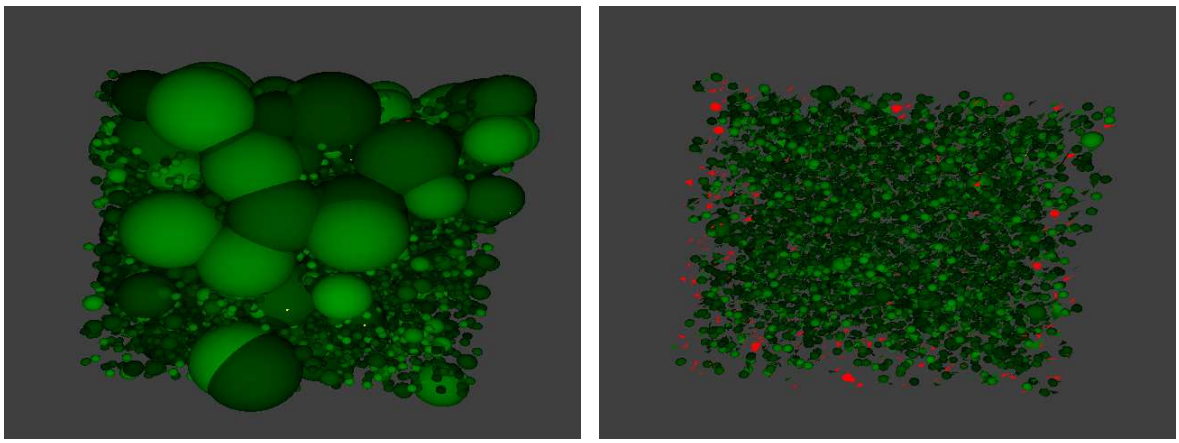
Testy przeprowadzono na dwóch scenach:

1. kule - około 13 tys. kul, które zostały zawarte w prostopadłościanie o wymiarach $40 \times 40 \times 40$.

2. kule i trójkąty - jest to poprzednia scena, w której zastąpiono połowę kul trójkątami,

3. duże i małe kule - 13 tys. kul o losowych rozmiarach zawartych w scenie o wymiarach $40 \times 40 \times 40$.

Widok obiektów sceny zaprezentowany jest na rysunku 35. PVS pierwszej sceny został wyznaczony przez GVS z oryginalnym warunkiem stopu - co spowodowało wygenerowanie około 2 milionów promieni. Na rysunku 35 po prawej przedstawiono widok PVS kul i trójkątów wyznaczonym z ograniczeniem do co najwyżej 100 tys. losowych promieni.



Rysunek 35: Obrazy PVS wygenerowane metodą śledzenia promieni. Po lewej scena kul, po prawej losowych kul i trójkątów. Na czerwono zaznaczono obiekty nie znalezione przez GVS.

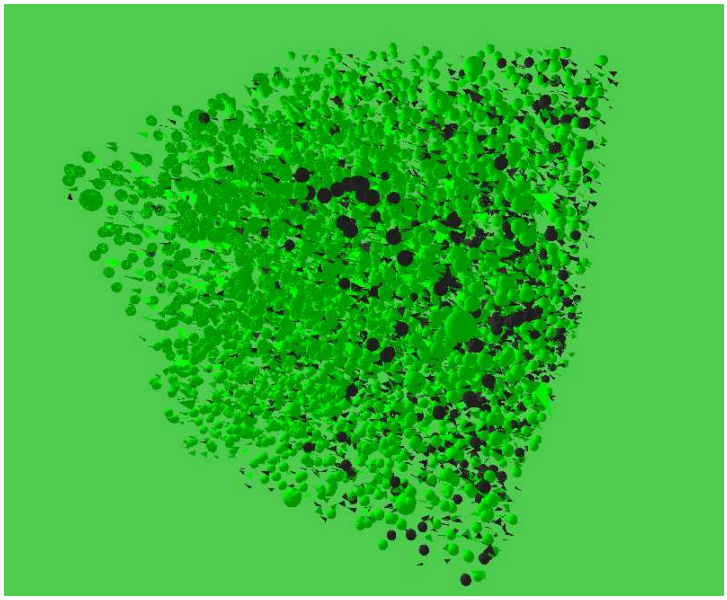
Na rysunku 36 przedstawiono zasłonięte elementy z innego widoku niż obszar dla którego wyznaczono PVS.

W algorytmie **GVS** dla tak losowych scen jak 1 i 2 około 50% promieni jest generowanych losowych, co powoduje zwiększony czas działania algorytmu. Uzyskany błąd (tabela 5) spowodowany jest głównie poprzez **błędy numeryczne**.

Tablica 5: Wyniki działania algorytmu GVS dla scen z kulami.

Scena	Rozmiar	PVS	Liczba iteracji	Błędne piksele	Czas (s)
kule	10,116	6,096	2,035,198	25	29.96
kule i trójkąty	10,116	7,422	2,075,007	38	30.05

W przypadku triangulacji kul rozmiar sceny zwiększył się do około 1 mln, co wpłynęło na wzrost liczby potrzebnych iteracji do 10 mln (spowodowane bardzo dużym rozmiarem PVS, tj. około 60-70 % sceny). W wyniku błędów numerycznych w PVS znajdowane były również niewidoczne części kul, tj. zasłonięte trójkąty należące do triangulacji danej kuli.



Rysunek 36: Niewidoczne obiekty (ciemnoszare) z tyłu sceny.

Redukcja liczby losowych promieni do 100 tys. przyczynia się do szybszego zakończenia działania algorytmu kosztem większego błędu, który spowodowany jest małym prawdopodobieństwem trafienia losowym promieniem w obiekt. Na rysunku 35 po prawej zaznaczone na czerwono nieznalezione elementy znajdują się na obrzeżach sceny. Elementy te nie stanowią spójnego obszaru (modelu) ani też nie ograniczają widoczności, przez co ani **ABS**, ani **próbkowanie wsteczne** nie klasyfikuje ich jako widoczne. Tabela 6 zawiera wyniki algorytmu GVS po uwzględnieniu 100 tys. losowych promieni.

Tablica 6: Wyniki działania algorytmu GVS dla scen z kulami z ograniczeniem losowych promieni.

Scena	Rozmiar	PVS (%)	Liczba iteracji	Błędne piksele	Czas (s)
kule i trójkąty	10,116	7,495 (74%)	1,118,050	1482	16.79
duże i małe kule	11,301	1,101 (9%)	278,660	17	1.51

6.3 Promienie losowe

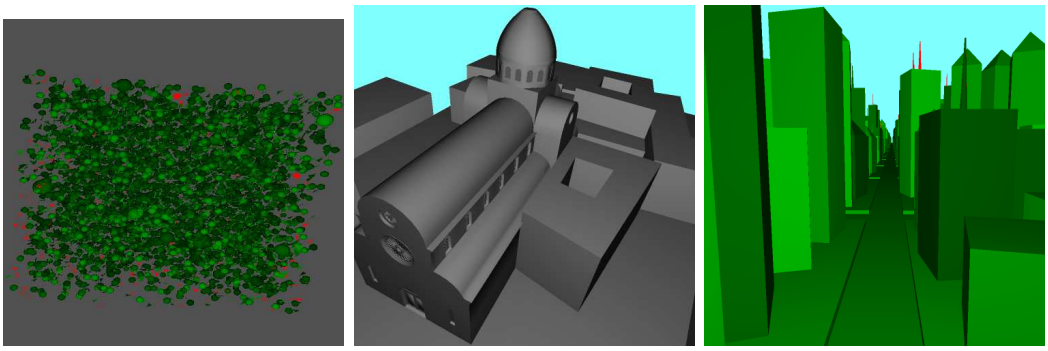
Strategie próbkowania losowych promieni:

- **first** - gorliwa, w której promienie losowe rozpatrywane są **na początku GVS**,
- **last** - leniwa, w której promienie losowe rozpatrywane są gdy **kolejka GVS jest pusta**,

Zostały one poddane następującym testom:

- scena losowych 10 tys. trójkątów i kul (rysunek 37 po lewej):,

- **sibenik** - katedra zawierająca około 80 tys. trójkątów, obszar obserwatora nad katedrą (rysunek 37 na środku),
- miasto złożone z 770 tys. elementów, obszar obserwatora na jednym ze skrzyżowań (rysunek 37 po prawej).



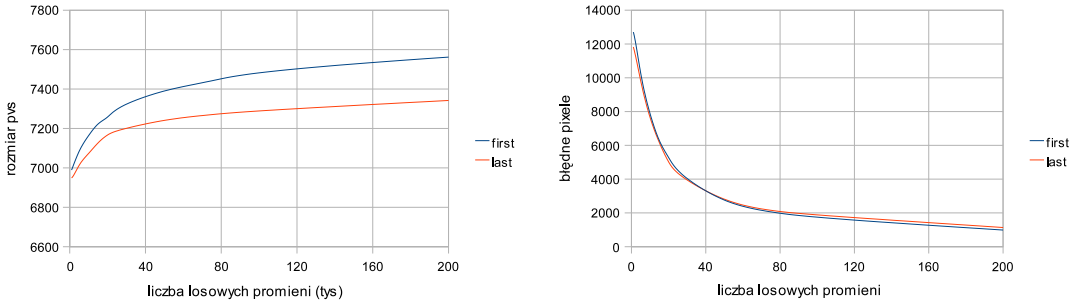
Rysunek 37: Widoczne obiekty znajdowane przez algorytm GVS

Scena zawierająca losowe obiekty jest przykładem będącym dużym wyzwaniem dla wszelkich algorytmów wyznaczających zbiór widocznych obiektów. Stosunkowo duży rozmiar PVS stanowiący 70% sceny i brak przyległych powierzchni, powoduje duże błędy przy stosowaniu małej liczby losowych promieni. Dla sceny *sibenik* już samo próbkowanie adaptacyjne wyznacza satysfakcjonujące rozwiązanie. Scena miasta jest standardowym przykładem dużej sceny, w której znaczna liczba trójkątów jest zasłaniana przez obiekty widoczne - fasady budynków. Poprzez ustawienie obszaru obserwatora na skrzyżowaniu dróg algorytm GVS znajduje obiekty wzdłuż ulic ukierunkowując próbkowanie w obszary jeszcze nie zbadane.

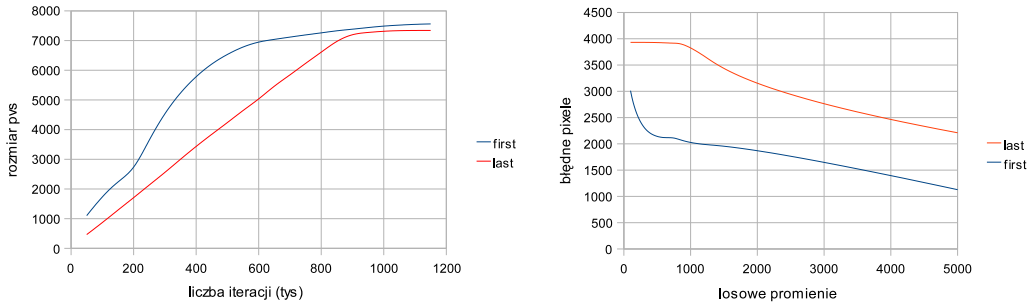
Przybliżenie zbioru widocznych obiektów o stosunkowo niedużej liczbie błędnych pikseli (rysunek 38) jest tym lepsze im więcej zostało wygenerowanych losowych promieni. Również różnica w liczbie zarodków uzyskanych przy 200 tys. losowych próbek powoduje, że strategia *first* obarczona jest mniejszym błędem. Przykładowo dyskretyzacja obszaru obserwatora punktami, z których szukano widocznych obiektów wyniosła 3,201 punktów dla strategii *first* oraz 742 punkty dla *last*). Należy jednak zwrócić uwagę, że nie jest znana liczba promieni losowych, które ostatecznie wygeneruje oryginalny GVS. Liczbę losowych próbek trzeba więc określić np. na podstawie liczby obiektów w scenie. W otrzymywanych wynikach już przy 100 tys. promieni otrzymano mniejszą liczbę błędnych piskeli.

Wykres na rysunku 39 pokazuje wzrost zbioru widocznych elementów względem kolejnych iteracji algorytmu, tj. wygenerowania kolejnych próbek. Strategia losowania na początku promieni efektywniej wyznacza PVS. Również średnia liczba błędnych pikseli, pokazana na rysunku 39 jest mniejsza

W przypadku sceny zawierającej katedrę algorytmy znajdowały podobny PVS, który zawierał około 6300 trójkątów. Maksymalny błąd 15 pikseli zaobserwowano dla dwóch strategii, zaś w przypadku leniwej strategii promieni losowych już po pierwszym



Rysunek 38: Rozmiar PVS i liczba błędnych pikseli dla sceny losowych trójkątów i kul.



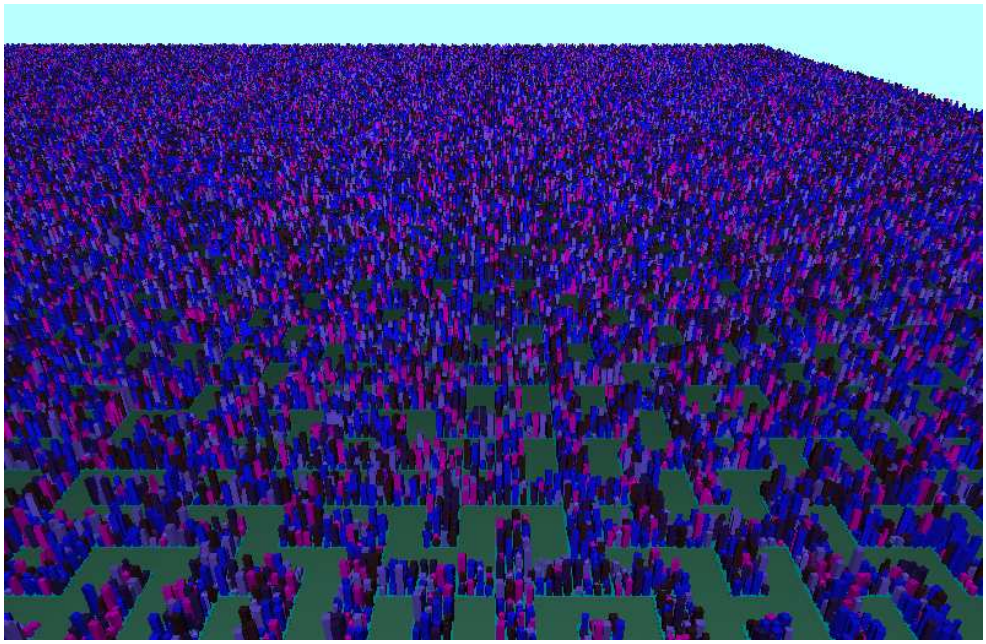
Rysunek 39: Po lewej: rozmiar PVS w trakcie kolejnych iteracji algorytmu dla sceny losowych trójkątów i kul. Po prawej: Liczba błędnych pikseli względem liczby losowych promieni dla sceny miasta.

promieniu trafiającym w scenę GVS wyznacza niemal wszystkie widoczne trójkąty. Stąd wniosek, że kolejność losowych promieni nie ma istotnego znaczenia dla tego typu scen. Zgodnie z oczekiwaniami próbkowanie adaptacyjne skutecznie wyznacza widoczne powierzchnie złożone z przyległych trójkątów, natomiast próbkowanie wsteczne efektywnie analizuje nieciągłości. Promienie znalazły około 1500 obiektów.

6.4 Wyświetlanie dużych scen

Generowanie obrazu metodą śledzenia promieni pozwala na szybkie wyświetlenie sceny, gdy pomija się odbite promienie i wyznaczanie cieni. Oświetlenie wraz z padającymi na scenie cieniami można aproksymować poprzez wyznaczenie zbioru widocznych obiektów ze światła - co jednak pozwala jedynie na zweryfikowanie czy dany obiekt jest cały w cieniu lub czy jest częściowo oświetlony. Dokładniejsza metoda przybliżania oświetlenia została zaprezentowana w pracy [1].

Zakładając, że wszystkie obiekty są w całości oświetlone przechodzenie złożonej sceny można zaprogramować jako wyznaczanie zbioru widocznych obiektów, następnie budowanie kd-drzewa i wyświetlenie obiektów. Taka metoda okazuje się przydatna przy scenach zawierających kilka milionów obiektów. Na rysunku 40 przedstawiono widok na scenę wygenerowanego miasta, które zawiera około **2.8 miliona trójkątów**. Statystyka sceny miasta:



Rysunek 40: Przykładowa scena miasta zawierająca 2,8 mln trójkątów.

- liczba obiektów: 2,759,991,
- rozmiar pliku mgf: 19.5 MB,
- czas wczytania sceny: 39.11 s,
- czas dokładnej budowy kd-drzewa: 209,99 s.

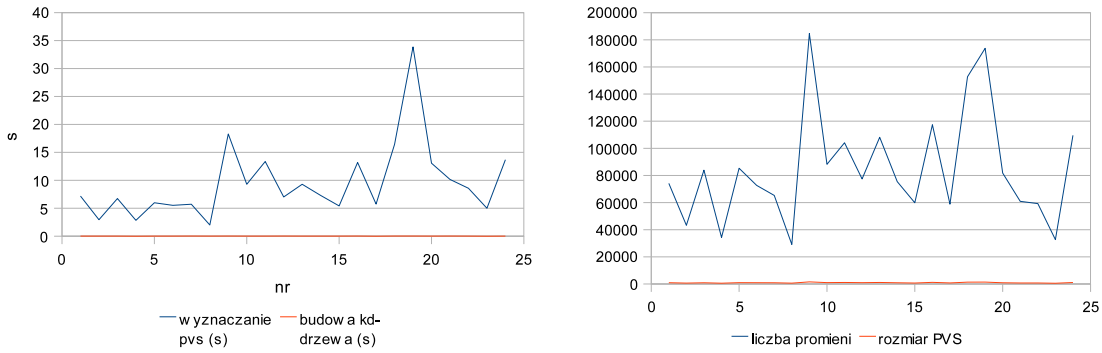
Umieszczenie obserwatora wewnątrz sceny determinuje rozmiar zbioru widocznych obiektów stanowiący nieraz 0.1% całego modelu. Średni czas generowania obrazu uproszczoną metodą śledzenia promieni (obraz o rozdzielczości 320×240:

- **6.5 - 10.4** s. - z użyciem kd-drzewa dla **wszystkich** trójkątów,
- **0.06 - 0.9** s. - z zastosowaniem kd-drzewa zawierającego jedynie obiekty widoczne.

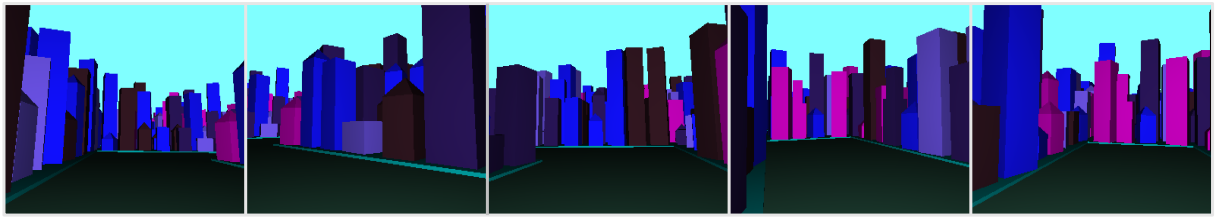
Powyższe czasy nie uwzględniają wyznaczania PVS ani przebudowy struktur danych sceny. Średni czas GVS wyniósł 83 sekundy, zaś zredukowanie liczby generowanych promieni w krawędzie zmniejszyło czas o prawie połowę do 46 sekund. Podczas przechodzenia sceny czasy algorytmów aktualizowania PVS i przebudowę kd-drzewa (opisanych w rozdziale 5) wynosiły od 4 do 33 sekund. Wykresy zawierające statystyki wyznaczania PVS i budowy kd-drzew przedstawione są na rysunku 41

Opisana selekcja widocznych obiektów z rozdziału 5 jest metodą agresywną, co powoduje pojawienie się artefaktów np. w scenie miasta brakujących ścian. Są one jednak nieraz niezauważalne, szczególnie gdy obserwator nie wie jak wygląda dokładna scena. Przykładowe obrazy PVS podczas przechodzenia sceny miasta pokazane są na rysunku 42.

Warto zauważyć, że zastosowanie algorytmu MLRTA pozwoliłoby na dokładniejsze wyznaczanie widocznych obiektów w dużo krótszym czasie, co ostatecznie umożliwiłoby wyświetlanie złożonych scen w czasie rzeczywistym.



Rysunek 41: Wyniki kolejnych aktualizacji widocznego zbioru. Po lewej: czas wyznaczania PVS oraz czas przebudowy kd-drzewa (wykres prawie pokrywający się z osią OX). Po prawej: liczba generowanych promieni oraz rozmiar PVS.



Rysunek 42: Wybrane obrazy generowane metodą śledzenia promieni podczas przechodzenia sceny miasta.

7 Podsumowanie

Problem wyznaczania widocznych obiektów stosunkowo niedawno doczekał się efektywnych algorytmów. Wśród wielu rozwiązaniach w większości aproksymacyjnych istnieje zaledwie kilka algorytmów dokładnych np. Nirenstein [7] w 3D i Bittner [22] w 2.5D. W publikacji [3] przedstawiono porównanie algorytmu GVS z dokładnym algorytmem Nirensteina. Różnica między otrzymywanymi wynikami była niewielka, lecz to właśnie algorytm GVS lepiej dokonywał selekcji widocznych obiektów. Powstałe różnice nie wynikają jednak z idei owych rozwiązań lecz z błędów obliczeń numerycznych. Przykładowo testując algorytm Möllera można zauważyć, że promienie skierowane w krawędzie odległego trójkąta omijają go. Konsekwencją tego jest dłuższy czas działania GVS oraz błędna klasyfikacja niewidocznych elementów sceny. W wyniku zatem paradoksalnie można otrzymać konserwatywny zbiór widocznych obiektów.

Dzięki prostej idei algorytmu GVS w dość łatwy sposób można dokonywać jego modyfikacji. Zarówno dowolny kształt obszaru obserwatora jak i uogólnienie na obiekty wypukle pozwoliły na obliczanie PVS z użyciem OORT (ang. *Object-Oriented Ray Tracing*). Jedynym problem stanowiło wyznaczanie promienia w *reverse sampling*, które dla zróżnicowanych figur geometrycznych nie jest już trywialne. Opracowane i opisane w rozdziale 4 metody redukcji liczby generowanych promieni doprowadziły do znacznego przyspieszenia algorytmu. Ograniczenia te wpływają jednak na aproksy-

mację zbioru PVS, choć w wielu przypadkach, tj. w scenach przedstawiających rzeczywiste modele a nie losowe zbiory trójkątów, liczba błędnych pikseli była nieznacznie większa. Również idea aktualizacji PVS z wykorzystaniem opisanych w rozdziale 5 promieni inicjujących przyczyniła się do szybszej selekcji widocznych obiektów. Poprzez zredukowanie liczby generowanych promieni w wielu przypadkach wyznaczanie PVS można przyspieszyć nawet siedmiokrotnie.

W pracy przetestowano wykorzystanie PVS podczas śledzenia promieni. Zgodnie z oczekiwaniami znacznie przyspieszyło to generowanie obrazów, jednak wpływa to jedynie na promienie początkowe. Użycie promieni odbitych wymaga uwzględnienia już całej sceny. Analogiczny problem pojawia się w oświetleniu pośrednim i bezpośrednim. Również i tu możliwa jest aproksymacja z zastosowaniem metod wyznaczania widoczności z obszaru (obliczenie oświetlenia opisał np. Schaufler w [1]).

Przechodzenie sceny miasta okazało się być nietrywialnym problemem, a wyznaczanie widocznych obiektów z kolejnych obszarów obserwatora nie zawsze satysfakcjonujące. Zarówno wyniki metody aktualizowania zbioru PVS jak i algorytmu GVS dla obszarów w scenie miasta zależą od wielu czynników. Próbkowanie widoczności z powierzchni obszaru powoduje pominięcie widocznych obiektów w przypadku dużego obszaru obserwatora lub obszaru, którego powierzchnia jest bliska obiektom sceny lub je przecina.

Ciekawym i niedawno opublikowanym rozwiązaniem jest algorytm Bittnera [11]. Idea jest zbliżona do GVS, jednak tu próbkowanie odbywa się w dwóch kierunkach: dla każdego promienia dodatkowo badany jest promień przeciwny. Tym sposobem znajdowane są dwa obiekty, które widoczne są ze wszystkich obszarów, przez które przeszły promienie. W wyniku otrzymujemy zbiory PVS dla wszystkich obszarów obserwatora.

Poza powstawaniem kolejnych algorytmów widoczności i efektywnych metod śledzenia promieni następuje rozwój architektury komputerowej. Znane już od kilku lat instrukcje SSE wprowadzone w procesorach *Pentium III* umożliwiły wykonywanie działań na 4-elementowych wektorach. Zastosowanie ich w algorytmie MLRTA [13] miało znaczny wpływ na efektywność algorytmu śledzenia promieni. Obecnie szczególną uwagę należy zwrócić na rozwój GPU (ang. *graphics processing unit*) i wykorzystanie wielu koprocesorów graficznych. Zarówno konstrukcja GPU w *Larrabbe Intela* jak i rozwiązanie śledzenia promieni w *NVIRT* przyczynią się do znacznego przyspieszenia *ray tracingu*. Warto zauważyć, że dzięki temu wyznaczanie widocznych obiektów będzie jeszcze efektywniejsze.

A Zaimplementowane programy

A.1 Lista programów

- fpvs - wyznaczanie PVS i zapis do pliku,
- gcmppvs - porównanie dwóch wyników PVS, wyświetlenie ich i oznaczenie kolorami (OpenGL),
- pvserror - wyznaczanie błędu PVS, generuje obraz PPM,
- rvs - wyznaczanie kolejnych zbiorów PVS i zapis ich do plików (program pomocniczy),
- rtrt - wyświetlanie scen (OpenGL) z ograniczonym widokiem do zbioru PVS, przechodzenie sceny, aktualizacja zbioru PVS,
- cpvs.hpp - („calculate pvs") moduł opakowujący algorytmy wyznaczania widocznych obiektów.

A.2 Kompilacja

W katalogu src/makefiles znajdują się wszystkie pliki makefile służące kompilacji programów. Plik MAKE_xxx odpowiada programowi xxx. Plik wykonywalny zostaje umieszczony w katalogu bin.
Kompilacja wszystkich programów:

```
sh scripts/compile_all.sh
```

Kompilacja wybranego programu:

```
sh scripts/compile_<prog>.sh
```

gdzie <prog> = fpvs | gpvs | gcmppvs | pvserror | rvs | rtrt

A.3 Uruchamianie

A.3.1 Programy

fpvs, którego wywołanie jest następujące:

```
./fpvs <properties> [-edgediv <int>] [-maxdepth <int>] [-edgetype <0|1> ]
[-maxrand <int>] [-randstep <0|1>] [-eeps <double>] [-pvsout <file>]
```

gdzie:

<properties> - plik z ustawieniami programu, zawiera m.in. ścieżkę do pliku sceny, współrzędne obserwatora, definiuje obszar widzenia

-edgediv K - wymuszenie podziału krawędzi otoczenia na K części

-maxdepth D - ograniczenie głębokości rekursji algorytmu podziału krawędzi

-edgetype 0|1 - typ przetwarzania krawędzi, 0 - gvs, 1 - punkt brzegowy

-maxrand N - maksymalna liczba losowych promieni

-randstep 0|1 - jeśli 1, to zostaną generowane jedynie losowe promienie - algorytm RAND

-eeps E - wartość epsilon - odległość otoczenia od obiektu (metoda „enlarge")

-pvsout P - zapis PVS do pliku P

gcmppvs, którego wywołanie jest następujące:

./gcmppvs <properties>

gdzie <properties> zawiera informacje o scenie, obserwatorze oraz ścieżkach do dwóch plików z PVS:

PVS_FILENAME - plik z pierwszym PVS

PVS2_FILENAME - opcjonalny parametr, jeśli niezdefiniowany, tylko pierwszy PVS jest wyświetlany

pvserror, którego wywołanie jest następujące:

./pvserror <properties> [-rand <long> | [-pvs <file>] [-limitpvs <long>] [-ppm <filename>]]

gdzie:

properties - plik z ustawieniami programu

rand R - tryb losowych promieni, bez generowania obrazu, w wyniku liczba błędnych 'pikseli'

pvs F - pliku wejściowy z PVS

limitpvs K - ograniczenie PVS do pierwszych K obiektów

ppm F - obraz wynikowy w formacie PPM

rvs, którego wywołanie jest następujące:

./rvs <properties> -vec vec[0] vec[1] vec[2] -steps <steps> -inpvs <file> -outpvs <file>

gdzie:

properties - plik z ustawieniami programu

vec v0 v1 v2 - wektor przemieszczania po scenie [v0,v1,v2]

steps S - liczba kroków (max 999)

inpvs FIN - wejściowy plik z PVS

outpvs FOUT - wyjściowy "PREFIX"pliku z PVS

uwaga: w wyniku działania powstaje wiele wynikowych PVS zatem są one zapisywane w plikach ze zmienianym sufiksem: jeśli FOUT = pvs_out_xxx.txt wówczas zostaną wygenerowane: pvs_out_001.txt, pvs_out_002.txt .... pvs_out_<S>.txt.

rtrt

./rtrt <properties>

gdzie <properties> jest plikiem z ustawieniami który zawiera informacje m.in. o obszarze i położeniu obserwatora, scenie itd.

A.3.2 Moduły, kod źródłowy, klasy

Główny moduł cpvs opakowuje algorytmy wyznaczanie widocznych obiektów. Zawiera on metody:

- cpvs_RAND - algorytm RAND - generowanie losowych próbek, znajdowanie przecięcia, które klasyfikuje obiekt jako widoczny,
- cpvs_GVS - algorytm GVS z pracy Wonki "Guided Visibility Sampling"[3],
- cpvs_RVS - algorytm aktualizujący PVS.

Poniżej został opisany kod źródłowy według katalogów:

- gvs/ - klasy i metody związane z algorytmem GVS wraz z rozszerzeniami, m.in. metody rozszerzania obiektów („enlarge”),
- pvs/ - klasa reprezentacji zbioru widocznych obiektów,
- rt/ - klasy i metody związane ze śledzeniem promieni wg prac Havrana [18] i Shevtova [17], tj.:
  - implementacja kd-drzewa wraz z algorytmem szybkiego budowaniem struktury,
  - moduł zarządzania pamięcią przy budowanie kd-drzewa
  - trawersowanie drzewa
  - metody i klasy uproszczonej wersji RayTracera
- scene/ - klasy i metody wczytywania (w formacie MGF) oraz reprezentacji w pamięci świateł, materiałów i obiektów sceny. Obiekty - primitives - zawierają funkcje opakowujące metody wykorzystywane w GVS (np. enlarge)
- settings/ - moduł służący do przechowywania parametrów dla zaimplementowanych aplikacji. Przykładowe pliki definujace ustawienia znajdują się w katalogu tests
- utils/ - wydzielone makra i przydatne funkcje m.in.: obsługa kamery, mapowanie tonów.

A.3.3 Plik z ustawieniami

Opis parametrów znajduje się w pliku src/settings/settings.hpp

A.4 Sceny

A.4.1 Generatory scen

- citymaze - generator mapy miasta w postaci 01 (0 - pusty blok, 1 wypełniony)
- citygen - generator sceny miasta do pliku mgf, opcjonalnie wykorzystuje wynik citymaze
- mesh - generator płaszczyzny podzielonej na trójkąty
- sphtrigen - generator sceny złożonej z losowych trójkątów i kul

A.4.2 Kompilacja i uruchamianie

Kompilacja wszystkich generatorów możliwa jest za pomocą skryptu scripts/compile_all.sh. W celu kompilacji wybranego programu ze względu na ich prostotę wystarczy np. polecenie: gcc src/program.cpp -o bin/program Na płycie dołączono skrypt generate_scenes.sh generujący pliki mgf na których przeprowadzano testy algorytmów.

Więcej szczegółów w pliku readme oraz plikach nagłówkowych.

Literatura

[1] Gernot Schaufler, Julie Dorsey, Xavier Decoret, Francois X. Sillion. Conservative Volumentric Visibility with Occluder Fusion. *Proceedings of SIGGRAPH 2000.*

[2] Subodh Kumar, Dinesh Manocha, Bill Garrett, Ming Lin. Hierarchical Back-Face Culling. *7th Eurographics Workshop on Rendering*, 1996.

[3] Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, Alexander Reshetov. Guided Visibility Sampling. *Proceedings of SIGGRAPH 2006.*

[4] Vladen Koltun, Yiorgos Chrysanthou, Daniel Cohen-Or. Hardware-accelerated from-region visibility using a dual ray space. *12th Eurographics Workshop on Rendering*, 205-216, 2001.

[5] Peter Wonka, Dieter Schmalstieg. Occluder Shadows for Fast Walkthroughs of Urban Environments. *Proceedings of EUROGRAPHICS'99*

[6] Peter Wonka, Michael Wimmer, Dieter Schmalstieg. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. *Rendering Techniques 2000*, 71-82.

[7] Shaun Nirenstein, Edwin Blake, James Gain. Exact From-Region Visibility Culling. *Rendering Techniques 2002 (Proceedings of the Thirteenth Eurographics workshop on Rendering).*

[8] Tomas Möller, Ben Trumbore. Fast, minimum storage ray/triangle intersection. *Journal of Graphics Tools*, 2, 1: 21-28, 1997.

[9] Tomas Möller. A Fast Triangle-Triangle Intersection Test. *Journal of Graphics Tools*, 2, 2: 25-30, 1997.

[10] Jatin Chhugani, Budirijanto Purnomo, Shankar Krishnan, Jonathan D. Cohen, Suresh Venkatasubramanian, David Johnson, D.S.2005. vLOD:High-fidelity walkthrough of large virtual environments. *IEEE Trans. on Visualization and Computer Graphics 11*, 1, 35–47.

[11] Jiri Bittner, Oliver Mattausch, Peter Wonka, Vlastimil Havran, Michael Wimmer. Adaptive Global Visibility Sampling. *Proceedings of SIGGRAPH 2009.*

[12] Ulf Assarsson, Tomas Möller. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*, 5(1), 9-22, 2000.

[13] Alexandr Reshetov, Alexei Soupikov, Jim Hurley. Multi-Level Ray Tracing Algorithm. *Proceedings of SIGGRAPH 2005.*

[14] Vladlen Koltun, Daniel Cohen-Or. Selecting Effective Occluders for Visibility Culling. *EUROGRAPHICS 2000 / A. de Sousa, J.C. Torres.*

[15] Carlos Andujar, Carlos Saona-Vazquez, Isable Navazo. Lod visibility culling and occluder synthesis. *Computer Aided Design*, 1999.

[16] Thomas W. Sederberg , Ray J. Meyers. Loop detection in surface patch intersections. *Computer Aided Geometric Design*, 5, strony 161-171, 1988.

[17] Maxim Shevtov, Alexei Soupikov, Alexander Kapustin. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. *EUROGRAPHICS 2007 / D.Cohen0Or and P. Slavik.*

[18] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* Praca doktorska, Faculty of Electrical Engineering, Czech Technical University, Prague, Maj 2001.

[19] Philip Dutré. Global illumination compendium, Wrzesień 2003.

[20] Harald Niederreiter. Random Number Generation and Quasi-Monte Carlo Methods. *SIAM Philadelphia*, 1992.

[21] Maxim Shevtsov, Alexei Soupikov, Alexander Kapustin. Ray-Triangle Intersection Algorithm for Modern CPU Architectures. *Proceedings of GraphiCon 2007*, 33-39, 2007.

[22] Jin Bittner, Peter Wonka, Michael Wimmer. Fast Exact From-Region Visibility in Urban Scenes. *Eurographics Symposium on Rendering 2005*, 223 - 230, Czerwiec 2005.

[23] Carl Erikson, Dinesh Manocha, William V. Baxter III. Holds for faster display of large static and dynamic environments. *Proceedings Symposium on Interactive 3D graphics*,111-120, 2001.

[24] Michael Garland, Paul S. Heckbert. Surface simplification using quadric error metrics. *Proceedings of ACM SIGGRAPH 1997*, 209-216.

[25] Hugues Hoppe. Progressive meshes. *Proceedings of SIGGRAPH 1996.*

[26] Julie C. Xia, Amitabh Varshney. A dynamic view-dependent simplification for polygonal models. *Proceedings of IEEE Visualization*, 327-334, San Fransisco, CA, 1996.