AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Michał Krawczyk

kierunek studiów: **informatyka stosowana**

Gra RPG oparta na wokselach

Opiekun: **dr. inż. Janusz Malinowski**

Kraków, styczeń 2017

Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

.....
(czytelny podpis)

Ocena opiekuna

Ocena recenzenta

Spis treści

1 Wstęp **6**

2 Użyte biblioteki i narzędzia **6**
2.1 OpenGL . . . 6
2.1.1 Rozwój OpenGL . . . 6
2.1.2 Potok graficzny . . . 6
2.2 GLFW . . . 8
2.3 SOIL2 . . . 8
2.4 GLSL . . . 8

3 Woksele **8**
3.1 Wstęp . . . 8
3.2 Organizacja danych . . . 9
3.3 Metody wyświetlania wokseli . . . 10
3.3.1 Podejście naiwne . . . 10
3.3.2 Sprawdzanie otoczenia . . . 10
3.3.3 Algorytm greedy mesh . . . 10
3.4 Teksturowanie modelu . . . 13
3.5 Proceduralne generowanie terenu . . . 14

4 Organizacja świata gry **17**
4.1 Drzewo list cyklicznych . . . 17
4.2 Drzewo ósemkowe . . . 18
4.2.1 Organizacja chunk'ów . . . 20
4.2.2 Wykrywanie kolizji . . . 22
4.2.3 Obcinanie niewidocznych powierzchni . . . 23

5 Kolizje **24**
5.1 Zderzenie AABB - AABB . . . 24
5.2 Zderzenie AABB - teren . . . 24
5.3 Zderzenie promień - teren . . . 25

6 Obcinanie obiektów poza bryłą widoku **26**
6.1 Otrzymywanie płaszczyzn bryły widoku . . . 27
6.2 Sprawdzanie widoczności AABB . . . 28

7 Budowa silnika gry **29**
7.1 Menadżery zasobów . . . 29
7.2 Obsługa wejścia . . . 30
7.3 Obsługa rysowania . . . 31
7.4 VEngine . . . 32

8 Podsumowanie **32**

9 Bibliografia **34**

1 Wstęp

Już niedługo po tym, jak zaczęto produkować pierwsze elektroniczne komputery, próbowano wykorzystywać ich moc obliczeniową do nieco mniej ambitniejszych celów niż ich pierwotne przeznaczenie. Dobrym przykładem tego zjawiska jest gra *Tennis for Two*, stworzona w 1958 roku przez Williama Higinbothama lub kółko i krzyżyk z 1952 roku stworzona przez A.S.Douglasa. Pierwszą grą wideo, która odniosła wielki sukces komercyjny był Pong z 1972 roku. Od tego czasu możliwości obliczeniowe wzrosły o rzędy wielkości oraz powstały specjalistyczne procesory graficzne (GPU - Graphics Processing Unit), które obecnie są w stanie wygenerować obraz w rozdzielczości w rozdzielczości 4K zachowując dużą realistykę obrazu.
Mimo potężnej mocy obliczeniowej kart graficznych, wielu producentów gier bawi się konwencją i wymyśla nietypowe, niekoniecznie zadziwiające graficznie gry, które odnoszą wielki sukces z powodu ciekawego i unikalnego pomysłu. Jedną z takich gier była gra Minecraft wydana w 2011 roku, której pomysł może nie był unikalny, bo czerpała ona garściami z produkcji Infiminer bazującej na tych samych założeniach, czyli na świecie zbudowanym z wokseli, który może być modyfikowany przez gracza. Mimo to, grze Minecraft udało się trafić na niszę na rynku gier komputerowych.
Sukces gry Minecraft spowodował pojawienie się bardzo wielu gier opartych na podobnych założeniach. Popularność wokseli może się wiązać z kilkoma rzeczami: podejście do świata jako do cegiełek daje projektantom możliwość generowania bardzo ciekawego świata, a graczom pozwala ten świat bez trudu modyfikować, co znacznie cięższej osiągnąć w standardowej grze.
Ponadto, z punktu widzenia programisty uważam, że próba stworzenia gry wokselowej może być bardzo ciekawym i kształcącym procesem, dającym nieco inne spojrzenie na grafikę 3D poprzez wymuszenie na programiście innego podejścia do optymalizacji i projektowania świata gry. Dlatego też postanowiłem spróbować zmierzyć się z tym problemem. Użyłem do tego języka C++ oraz bibliotek OpenGL 4.5, GLFW oraz SOIL2.

2 Użyte biblioteki i narzędzia

2.1 OpenGL

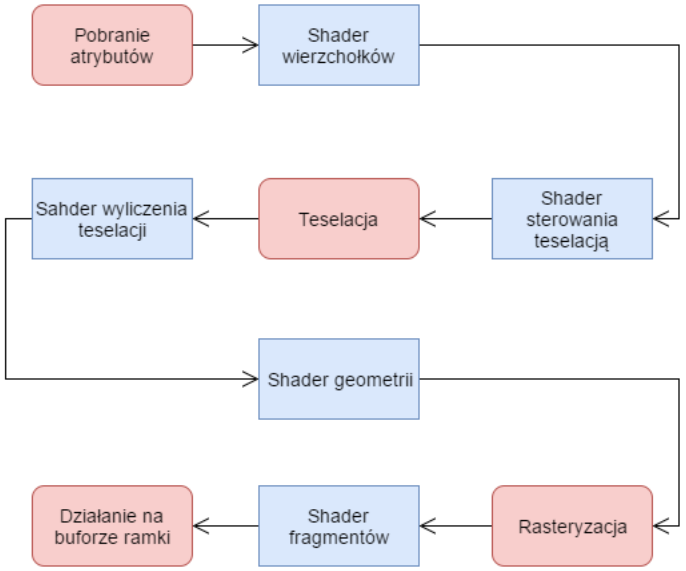
2.1.1 Rozwój OpenGL

OpenGL wywodzi się z firmy SGI (Silicon Graphics Inc.) z systemu IRIS GL. SGI w styczniu 1992 roku udostępniło zmodyfikowane API IRIS GL jako otwarty standard OpenGL - czyli Open Graphics Library. Jest to API do programowania grafiki zachowujące średni poziom abstrakcji - na tyle wysoki, by szczegóły implementacji sterownika i architektura sprzętu nie były zauważalne, ale z drugiej strony na tyle niski, by można było zachować dostęp do sprzętu i wykorzystać jego zaawansowane funkcje.
Wraz z wydaniem OpenGL założono organizację ARB (ang. Architectural Review Board), do której należą największe firmy IT. Tworzą one i rozwijają specyfikację OpenGL. Najnowszą obecnie specyfikacją jest OpenGL 4.5, wydaną w sierpniu 2014.
W 2008 roku podzielono specyfikację OpenGL na dwa profile: profil główny, nie wspierający już przestarzałych funkcji wynikających z innej architektury nowoczesnych kart graficznych, oraz profil zgodności, który jest zgodny z wersją 1.0.

2.1.2 Potok graficzny

Nowoczesne karty graficzne posiadają wiele małych programowalnych procesorów nazywanych rdzeniami cieniowania. Ich przepustowość nie jest bardzo wysoka i są znacznie prostsze od standardowych procesorów, ale ich ilość umożliwia wykonywać wiele operacji równolegle. Rysunek 2.1 przedstawia uproszczony

schemat potoku graficznego. Programując w OpenGL mamy dostęp do wszystkich shaderów - kwadraty o ostrych rogach, pozostałe etapy potoku wykonują kod o stałej funkcji. Potok można podzielić na dwie



Rysunek 2.1: Uproszczony schemat potoku graficznego. Rysunek na podstawie [src]

części: przednią, gdzie operujemy na wartościach wektorowych, oraz tylną, gdzie operujemy na pikselach. Ostatnim etapem strony przedniej jest shader geometrii. W procesie rasteryzacji trójkąty zostają zamienione na piksele.

- Kolejne etapy potoku muszą się ze sobą komunikować i przekazywać sobie dane. Proces pobierania atrybutów jest odpowiedzialny za dostarczanie danych do shadera wierzchołków.
- Shader wierzchołków dokonuje transformacji współrzędnych i przekazuje dane dalej. Może dokonywać wstępnych obliczeń dla danych podawanych dalej. Shader wierzchołków jest jedynym wymaganym shaderem by potok działał poprawnie, ale dopóki nie zdefiniujemy shadera fragmentów, nie zobaczymy nic na ekranie.
- Kolejnym etapem jest teselacja, która składa się z dwóch programowalnych shaderów i jednej stałej funkcji. Teselacja jest procesem polegającym na dzieleniu prymitywów wysokiego rzędu (płaty) na trójkąty. Możliwe jest także dzielenie prostych prymitywów na ich większą ilość.
- Shader geometrii ma dostęp do wszystkich wierzchołków danego prymitywu i może bezpośrednio zmniejszać lub zwiększać ilość danych. Można w nim jawnie tworzyć wierzchołki oraz zmienić tryb prymitywu, np zamienić trójkąt na punkty.
- Przed przystąpieniem do rasteryzacji należy pozbyć się zbędnych prymitywów. Przed przycięciem wierzchołków dzielimy każdą współrzędną wierzchołka przez czwartą współrzędną w we współrzędnych jednorodnych. Proces ten nazywa się dzieleniem rzutowym. Po tej operacji znajdujemy się w znormalizowanej przestrzeni urządzenia. W OpenGL obszar ten to od -1 do 1 dla x oraz y, a dla z od 0 do 1. Wszystkie punkty leżące poza tym obszarem będą niewidoczne i można dla nich pominąć proces rasteryzacji. Ponadto trzeba przetransponować obszar normalny do współrzędnych okna (ang. viewport). Dla większej optymalizacji OpenGL może usunąć przednie lub tylne ściany

trójkątów - albo nawet zarówno przednie jak i tylne. Stronę trójkąta determinuje kolejność rysowania wierzchołków. Domyślnie stroną przednią są wierzchołki podane z ułożeniem przeciwnym do wskazówek zegara, ale można to zmienić w trakcie działania programu.

- Rasteryzacja określa które fragmenty prymitywu zostaną przesłonięte przez inne.
- Shader fragmentów jest odpowiedzialny za wyliczenie koloru piksela. Można wziąć pod uwagę współrzędne UV tekstury, oświetlenie i kolor ściany.
- Na ostatnim etapie dane są przesyłane do bufora ramki i możemy zobaczyć wynik na ekranie.

2.2 GLFW

Biblioteka GLFW jest lekką i prostą biblioteką zapewniającą dostęp do okienek oraz do sygnałów wejścia z myszki oraz z klawiatury. Obecnie najnowsza wersja to GLFW 3.2.1 z 18 sierpnia 2016r. Operacje wejścia mogą być odczytywane albo za pomocą callbacków albo za pomocą pollingu. Ponadto GLFW zapewnia tryby wyświetlania pełnoekranowego, trybu okienkowego i trybu pełnoekranowego okienkowego (windowed fullcreen). Prostota i lekkość biblioteki idealnie współgrala z założeniem, że bardzo dużo rzeczy chciałem zaimplementować sam.

2.3 SOIL2

SOIL2 jest biblioteką do szybkiego wczytywania i generowania tekstur dla OpenGL. Pozwala jedną komendą załadować z pliku teksturę, odwrócić współrzędne y, by tekstura nie była wyświetlana do górny nogami, wygenerować mipmapy i automatycznie wykryć format pliku. SOIL2 to kontynuacja biblioteki SOIL (Simple OpenGL Image Library), która na chwilę obecną przestała być wspierana i wiele opcji już przestało działać. W bibliotece SOIL2 te niedogodności poprawiono.

2.4 GLSL

GLSL (ang. OpenGL Shading Language) jest językiem opartym na języku C i służy do programowania shaderów. Posiada wiele wbudowanych struktur i funkcji takich jak macierze, wektory, modulo, oraz wiele funkcji unikalnych dla konkretnych shaderów. Znajomość języka C pozwala bez najmniejszych problemów programować shadery. Po napisaniu shadera, należy go skompilować za pomocą odpowiedniej funkcji OpenGL.
Zastosowanie shaderów nie ogranicza sie wyłacznie do wyswietlania grafiki. Istnieją shadery obliczeniowe, które można wykorzystać do symulacji, np. do implementacji algorytmu stada, co znacznie poszerza zastosowanie biblioteki OpenGL. Niejednokrotnie, przy dużej ilości operacji SIMD, obliczenia są wykonywane na kartach graficznych zamiast na zwykłym procesorze.

3 Woksele

3.1 Wstęp

Woksele (ang. voxel - volumetric picture element) można rozumieć jako piksele w przestrzeni trójwymiarowej. Znalazły one szerokie zastosowanie w medycynie np. przy analizie wyników RMN (rezonansu magnetycznego jądrowego), ale także w branży gier komputerowej. Tak jak piksel przechowuje informacje o jego kolorze, podobnie woksele mogą przechowywać różne atrybuty: od podstawowych informacji o kolorze oraz pozycji danego woksela, do bardziej abstrakcyjnych informacji, które mogą tworzyć bazę szczegółowych informacji na temat obiektu, który przedstawiają.

Na przykładzie silnika gry wokselowej: oprócz typu danego woksela, można przechowywać informacje o oświetleniu, wilgotności, temperaturze oraz całym szeregu informacji, które mogą być używane do symulacji pewnych zjawisk. Należy jednak mieć na uwadze, że gdy chcemy przedstawić cały świat za pomocą wokseli z dużą ilością abstrakcyjnych informacji, bardzo łatwo może zabraknąć programowi pamięci operacyjnej. Załóżmy, że chcemy przedstawić świat gry o rozmiarze 512x512x512 metrów. Listing 3.1 przedstawia prostą definicję woksela.

```
struct Voxel {
  unsigned char type;
};
```

Listing 3.1: Definicja prostego woksela

Zakładając, że jeden woksel będzie odpowiadał sześcianowi o wymiarach 1x1x1 metr, ilość potrzebnej pamięci będzie wynosić 512^3B=128MB. Zwiększając rozdzielczość, wielkość obiektu który chcemy przedstawić lub ilość przechowywanych informacji, łatwo można wyczerpać cały zasób dostępnej pamięci operacyjnej. W medycynie większe zużycie pamięci jest bardziej akceptowalne niż w przypadku silnika gry, ponieważ silnik musi przechowywać dodatkowo informacje o innych obiektach występujących w świecie oraz wszelkie zasoby, takie jak tekstury i animacje.
By jak najbardziej skompresować informacje w wokselu, można nie przechowywać w nim dokładnej informacji o używanej teksturze, jego pozycji itd. Informacje te można umieścić na stałe w kodzie funkcji generującej model z tablicy wokseli lub wczytywać je z pliku do statycznych tablic. Dzięki takiemu podejściu, informacje o wyglądzie woksela możemy zapisać w zaledwie jednym bajcie, zamiast 4 bajtów: 3 dla koloru i 1 dla tekstury. Jako że woksele tworzą siatkę podobnie jak piksele i są równo odległe, pozycję woksela w świecie możemy obliczać na podstawie jego lokalizacji w strukturze danych, która będzie reprezentowała wszystkie woksele.
Kolejnym problemem jest wyświetlanie bardzo dużej ilości wokseli na raz (w powyższym przykładzie tych elementów jest ponad 134 miliony). W dalszych rozdziałach przedstawię bliżej problemy z wokselami oraz przykładowe metody ich rozwiązania.

3.2 Organizacja danych

Powszechną praktyką przy tworzeniu wokselowych silników gier jest agregowanie wokseli w większe zbiory, nazywanymi z języka angielskiego *chunks*. Takie podejście ułatwia organizację danych, umożliwia optymalizację wyświetlania całych chunk'ów, zamiast pojedynczych wokseli.
Najprostszym podejściem do przedstawienia chunk'a będzie zwykła trójwymiarowa tablica:

```
struct Chunk {
  const int size = 16;
  Voxel voxels[size][size][size];
};
```

Listing 3.2: Prosta agregacja wokseli w chunk

Można też zastosować tablicę jednowymiarową by nie posiadać postrzępionej pamięci, co powinno być bardziej wydajne jeśli chodzi o pamięc cache. Zakładając, że świat ma być modyfikowalny i interaktywny, rozmiar chunk'a należy dobrać tak, aby czas generowania modelu 3D z tablicy wokseli był na tyle niski, by można to było wykonać w czasie rzeczywistym - czas na wygenerowanie jednej klatki, przy założeniu, że chcemy otrzymać ilość klatek na sekundę równą częstotliwości odświeżania obsługiwaną przez współczesne monitory (60Hz), to 0,017ms.
Ponadto, rozmiar chunk'a nie może być zbyt niski, gdyż wtedy korzyść z optymalizacji ilości wierzchołków w modelu będzie znikoma.
Gdy już zostanie dobrany odpowiedni rozmiar chunk'a, następnie grupuje się go w większą strukturę, która

będzie reprezentować cały obiekt. Może to być zwykła tablica trójwymiarowa albo drzewo ósemkowe. W swoim silniku zastosowałem drzewo ósemkowe, co zostało opisane w rozdziale 4.2.

3.3 Metody wyświetlania wokseli

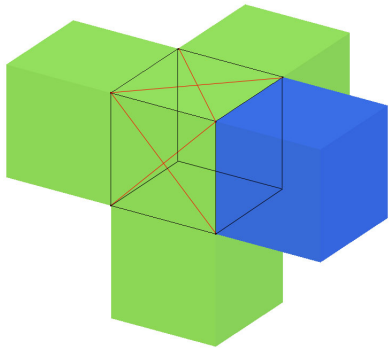
Tablica, którą przechowujemy jako surowe dane w postaci voxeli i chunk'ów, jeszcze nie nadaje się do wyświetlania. Są to surowe dane, które trzeba przerobić na model 3D. Należy dobrać odpowiednią metodę stworzenia siatki modelu (ang. *meshing*).
Można zastosować kilka podejść, cechujących się różnym stopniem zużycia czasu procesora do obliczenia siatki wierzchołków, ale też dających różną ilość zbędnych wierzchołków.

3.3.1 Podejście naiwne

Najprostszym podejściem jest brak jakiejkolwiek optymalizacji i wyświetlanie każdego woksela osobno. Przy bardzo niewielkiej ilości wokseli takie podejście może być użyte, jednak nie daje żadnych korzyści z grupowania wokseli w chunki. Przy założeniu, że generujemy dla każdego woksela osobne ściany i użyjemy definicji struktury, którą przedstawiłem na *Listingu 3.2*, to w najgorszym wypadku, gdy cała kostka będzie wypełniona, będziemy musieli wygenerować aż $16^3 * 6 = 24576$ ścian. Dla całego wypełnionego świata o rozmiarach 512x512x512 wokseli, będzie to aż 800 milionów ścian, co jednoznacznie wyklucza tą metodę z użycia przy tworzeniu wokselowego silnika gry.

3.3.2 Sprawdzanie otoczenia

Warto mieć na uwadze, ze duża część świata będzie otoczona przez woksele nieprzezroczyste. Można sprawdzić z każdej strony, czy dany woksel jest otoczony przez nieprzezroczysty woksel. Jeśli tak będzie, to można pominąć generowanie przesłoniętej ściany. W przypadku całego wypełnionego sześcianu, ilość ścian potrzebnych do wygenerowania będzie równa 1536. Im wewnątrz obiektu będzie większa ilość dziur, tym wyniki będą gorsze.



Rysunek 3.1: Przekątnymi zaznaczono ściany wygenerowane podczas algorytmu sprawdzania otoczenia wokseli

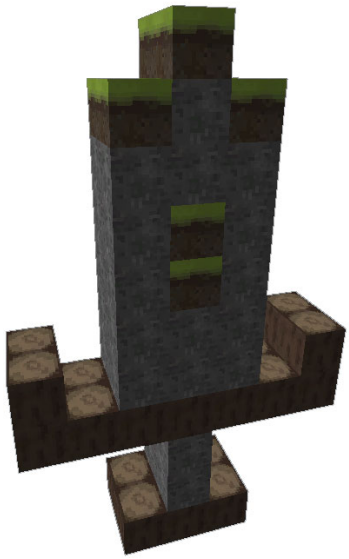
.

3.3.3 Algorytm greedy mesh

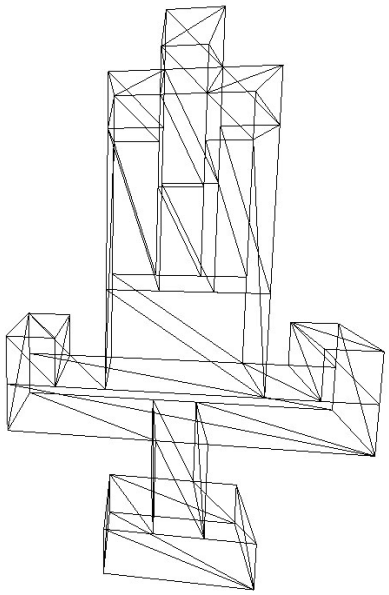
Algorytm greedy mesh sprowadza problem z wyświetlaniem wokseli, do problemu wyświetlania przekrojów kolejnych warstw wokseli dla każdego kierunku. Sprowadza to problem trójwymiarowy do dwuwymiarowej

przestrzeni. Problemem będzie znalezienie najmniejszej ilości wielokątów do pokrycia jednej płaszczyzny. Zdefiniujmy czworokąt jako zbiór parametrów: (x, y, w, h), gdzie:

- (x, y) to lewy dolny róg czworokąta
- (w, h) to szerokość i wysokość czworokąta



Rysunek 3.2: Model stworzony z różnych typów wokseli



Rysunek 3.3: Siatka stworzonego modelu

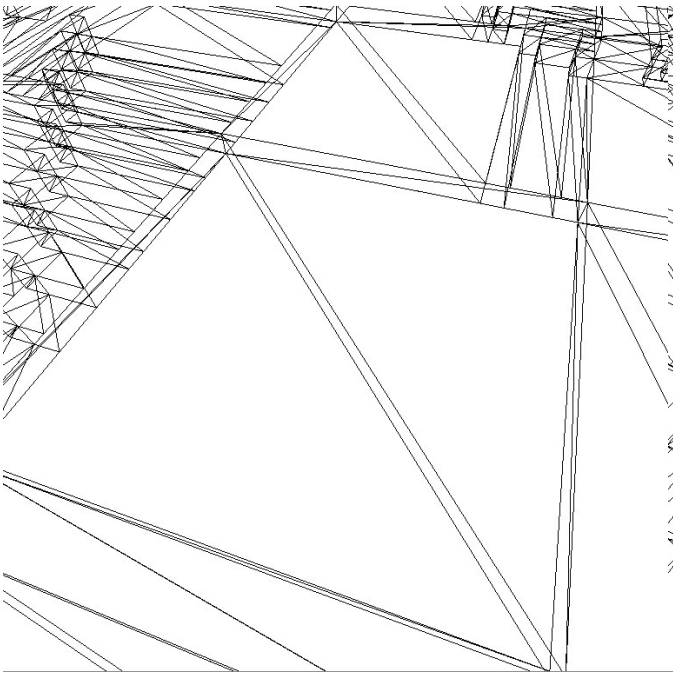
Zdefiniujmy pewien podzbiór czworokątów:

$$Q_{x,y,w,h} = \{(s,t) \in \mathbb{R}^2 : x \leq s \leq x + w \wedge y \leq t \leq y + t\}$$

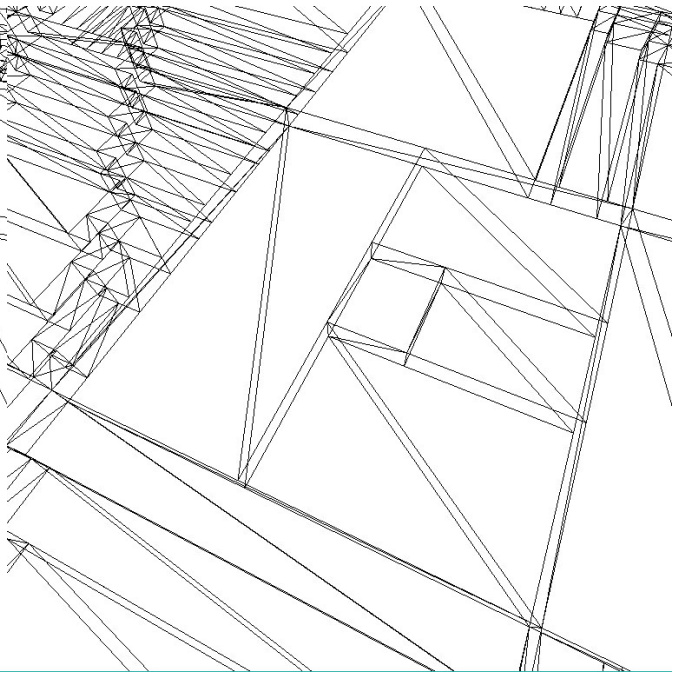
I ustalmy porządek liniowy czworokątów znajdujących się w podzbiorze Q_i. Chcemy posegregować wszystkie czworokąty od dołu do góry (po współrzędnej y), a następnie od lewej do prawej (po współrzędnej x). $(x_0, y_0, w_0, h_0) \leq (x1, y_2, w_2, h_2) \Leftrightarrow$ gdy zostanie spełniony jeden z warunków w następującej kolejności:

- jeśli $y_0 \neq y_1$, to $y_0 < y_1$,
- w przeciwnym wypadku, jeśli $x_0 \neq x_1$, to $x_0 < x_1$
- w przeciwnym wypadku, jeśli $w_0 \neq w_1$, to $w_0 > w_1$
- w przeciwnym wypadku, $h_0 >= h_1$

W pierwszej kolejności będziemy brali czworokąty najbliżej lewego dolnego rogu czworokąta, a następnie najszersze i najdłuższe. Posiadając posortowane czworokąty, najmniejszy element będzie czworokątem, który chcemy zaakceptować.
Schemat algorytmu, uwzględniający także przysłonięcie obiektu przez kolejne warstwy wokseli, jest następujący:



Rysunek 3.4: Wycinek terenu

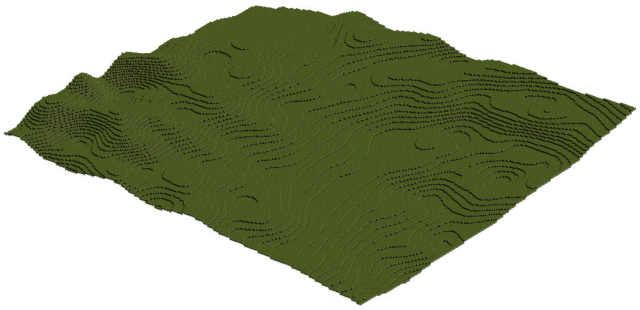


Rysunek 3.5: Ten sam wycinek z wydrążoną dziurą

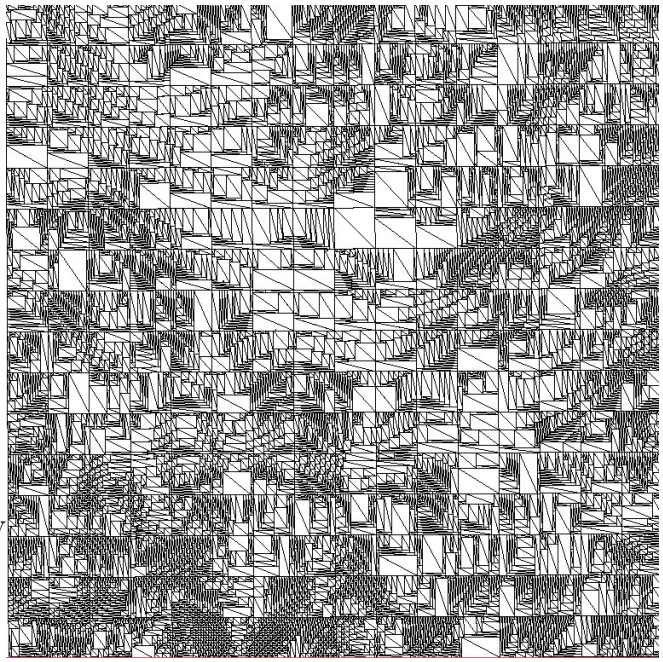
Dla każdego kierunku *front* (z wartości ujemnych do dodatnich gdy $front = 0$ oraz z wartości dodatnich do ujemnych gdy $front = 1$), wzdłuż każdej osi *dir* (X, Y, Z):
dla każdego przekroju chunk'a $P_{dir}[d]$, gdzie $d \in (0, size_{dir})$, utwórz maskę $mask[size_u][size_v]$ gdzie $size_u$, $size_v$ to wymiary chunk'a prostopadłe do kierunku *dir* wykonuj następujące kroki:

1. Porównaj każdy element na obecnej płaszczyźnie z elementem na wcześniejszej płaszczyźnie (jeśli płaszczyzna nie istnieje, to potraktuj to tak, jakby woksel był pusty).
   Jeśli woksel w obecnej płaszczyźnie $P_{dir}[d]$ nie jest pusty i jest przysłonięty wokselem nieprzezroczystym, to zapisz w masce wartość 0, w przeciwnym wypadku:
   Dodaj do maski:
   - jeśli $front = 0$, to weź element z płaszczyzny obecnej $P_{dir}[d]$
   - jeśli $front = 1$, to weź element z płaszczyzny poprzedniej $P_{dir}[d-1]$
2. Niech $u = 0$ i $v = 0$
   a) Jeśli element $mask[u][v]$ istnieje to:
      i. Przyjmujemy, że jest to punkt x_0, y_0 naszego *greedy mesh*. Obliczamy szerokość w czworoboku, scalając ze sobą takie same ściany. Następnie obliczamy wysokość h czworoboku, zachowując te same warunki co w przypadku szerokości.
      ii. Na podstawie znajomości osi oraz kierunku, wydobądź odpowiednie ściany woksela i dodaj odpowiednie wierzchołki do siatki.
      iii. Wymaż z maski wygenerowany czworobok.
      iv. Dokonaj przesunięcia w masce: $u+ = w$,
   b) W przeciwnym wypadku inkrementuj u
   c) Gdy $u >= size_u$, przypisz $u = 0$, $v+ = 1$

d) Jeśli $v < size_v$, przejdź do kroku a)



Rysunek 3.6: Rzut izometryczny pierwszej warstwy terenu (512x512 wokseli)



Rysunek 3.7: Siatka tego terenu widziana od góry

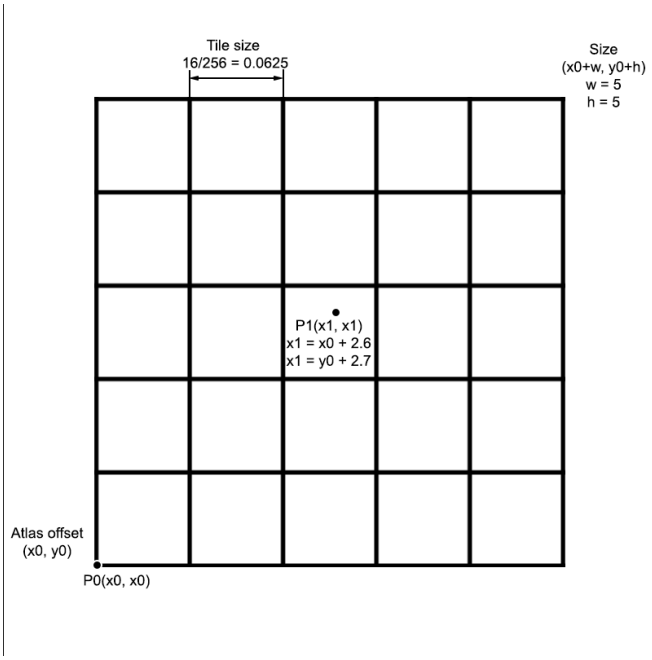
3.4 Teksturowanie modelu

Wynikiem algorytmu greedy mesh są scalone ściany wielu wokseli tego samego typu. Wykorzystanie pojedynczych tekstur dla każdego rodzaju ściany nie sprawiłoby problemu z nakładaniem tekstury na wynikowy czworokąt, jednak wymusiłoby to przechowywania ogromnej ilości różnych tekstur oraz bardzo częste dowiązywanie tekstur do potoku graficznego, co jest kosztowną operacją.
Rozwiązaniem jednego z tych problemów może być zastosowanie atlasu tekstur. Jest to jeden plik graficzny na którym znajdują się wszystkie używane w grze tekstury. Użyłem rozmiaru 256x256 pikseli, przy czym szerokość jednej tekstury to 16 pikseli. Pozwala to pomieścić łącznie 256 różnych tekstur. Komplikuje się natomiast sposób teksturowania czworokątów uzyskanych z algorytmu greedy mesh. Aby poprawnie mapować teksturę, należy dostarczyć do shadera fragmentów informację o pozycji tekstury w atlasie, rozmiarze jednej tekstury w atlasie po przeskaloawniu do współrzędnych (u, v) oraz ile razy w kierunku u oraz kierunku v ma być powtórzona tekstura. Problem ten przedtawia rysunek 3.8.
Aby obliczyć punkt początkowy tekstury P_0 we współrzędnych (u, v), należy pomnożyć indeks x i y, odpowiadający pozycji tekstury w mapie, przez szerokość jednego kafelka tekstury - w moim przypadku $16/256 = 0.0625$.
Kolejnym krokiem będzie zawinięcie współrzędnych tekstury, by co każdy odstęp kafelka powtarzała schemat. Zdefiniujmy punkt P_1 jak na rysunku 3.8. Można zauważyć, że parametry w oraz h służą jedynie poprawnemu skalowaniu tekstury i jej równomiernemu rozłożeniu. Aby uzyskać pożądany efekt, wystarczy obliczyć $(u_1 \mod 1.0) * tileSize$ i analogicznie dla współrzędnej v. Shader fragmentów na listingu 3.3 realizuje to zdanie.

```
1 #version 450 core
2
3 //Wejscie z shadera wierzcholkow
```

```
in VsOut {
  //wspolrzedne x_n oraz y_n
  vec2 texCoord;
  //wspolrzedne x_0 oraz y_0 - lokalizacja tekstury w atlasie
  vec2 texOffset;
};

//szerokosc jednego kafelka
uniform float tileSize;
uniform sampler2D tex1;

void main()
{
  vec2 resultUV;
  resultUV.x = (vsOut.texOffset.x * tileSize) + mod(vsOut.texCoord.x, 1.0) * (tileSize);
  resultUV.y = (vsOut.texOffset.y * tileSize) + mod(vsOut.texCoord.y, 1.0) * (tileSize);
  color = texture(tex1, resultUV);
}
```

Listing 3.3: Obliczanie współrzędnych tekstury w shaderze fragmentów dla algorytmu siatek z algorytmu greedy mesh



Rysunek 3.8: Mapowanie tekstur dla czworoboków o różnej wielkości

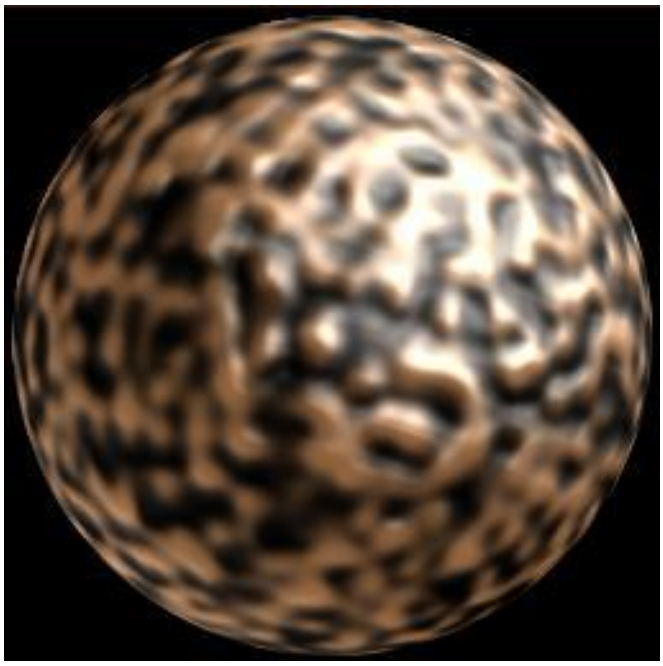
Rysunek 3.9: Atlas 256/256 pikseli, używany w silniku VEngine

Wynikiem będzie teksturowanie, jakie możemy zaobserować na rysunku 3.2.

3.5 Proceduralne generowanie terenu

Wygenerowanie zróżnicowanego terenu w grze znacznie zwiększa pozytywne doznania gracza i urozmaica czas spędzony na eksploracji. Niweluje to powtarzalność rozgrywki wynikającą z odwiedzania tych samych lokalizacji. Na potrzeby aplikacji zastosowałem do generowania terenu szum Perlina, który jest powszechnie używany w branży gier komputerowych do generowania organicznego szumu, mogącego od-

powiadać nierównościom przedmiotów przy mapowaniu wypukłości, generowaniu tekstury ognia, a także przy generowaniu terenu. Szum ten cechuje się gładkim przejściem między kolejnymi wartościami liczb pseudolosowych.



Rysunek 3.10: Mapowanie wypukłości przy pomocy filtru Perlina - źródło http://mrl.nyu.edu/ perlin/noise/

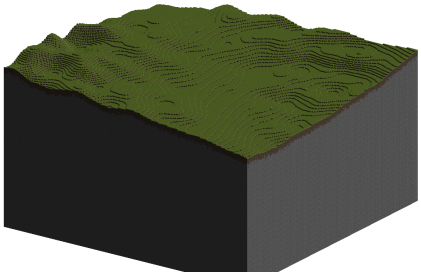
Do wygenerowania terenu użyłem szumu dwuwymiarowego. Do wygenerowania bardziej złożonych światów, z naturalnymi jaskiniami, szum trójwymiarowy wydaje się być bardziej odpowiedni, jednak zwiększa to znacząco czas obliczeń potrzebny do wyliczenia szumu oraz wymaga bardziej złożonego algorytmu. Do sterowania wyglądem terenu, prócz ziarna, korzystam z kilku zmiennych sterujących:

- **smootness** - steruje gładkością terenu w skali globalnej. Im większy ten współczynnik, tym teren będzie mniej różnorodny i bardziej rozpięty w skali całego świata.
- **details** - ilość lokalnych detali. Zwiększenie tego parametru spowoduje powstanie większej ilości małych górek i dolin. By nazwa zmiennej miała sens, wartość rzeczywistą, używaną przy generowaniu terenu, obliczam z następującego wzoru: $detail = \frac{32.0f}{givenDetail}$
- **spread** - parametr sterujący rozpiętością terenu w górę i dół, zwiększenie tego parametru spowoduje zwiększenie różnicy pomiędzy wysokościami kolejnych wokseli.
- **seaOffset** - poziom morza, można sterować wysokością całego terenu za pomocą tego parametru
- **rockOffset** - Głębokość od klocka ziemi po której zamiast ziemi mają być generowane skały. Jako że ta odległość będzie generowana przy pomocy rozkładu normalnego, jest to parametr μ w rozkładzie normalnym.
- **rockSharpness** - wariancja rozkładu normalnego dla generowania skał.

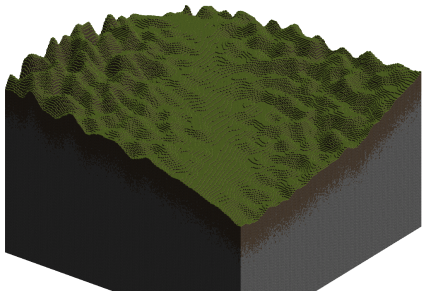
Funkcję determinującą wartość wokseli w danym miejscu w przestrzeni przedstawiłem na listingu 3.4.

```
Voxel
TerrainGenerator::GetVoxel(int x, int y, int z)
{
  //Zmienna odpowiadajaca za ogolna wysokosc, niska czestotliwosc
  float height = _perlinGenerator.GetNoise(x / _smoothness, z / _smoothness, 0.2f);
  //Parametr nierownosci, niska czestotliwosc
  float rough = _perlinGenerator.GetNoise(x / _smoothness, z / _smoothness, 0.7f);
  //Parametr odpowiadajacy za ilosc detali, wysoka czestotliwosc
  float detail = _perlinGenerator.GetNoise(x / _details, z / _details, 0.3f);
  //Formula opisujaca wysokosc
  height = (height + (rough * detail)) * _spread + _seaOffset;

  //Zeby odleglosc skal od ziemi byla nieco bardziej roznorodna,
  //zastosowalem rozklad normalny do odleglosci skal od powierzchni
  int rocks = (int)_distribution(_generator);

  //Determinujemy czy woksel w tym miejscu bedzie istnial
  if (y < height) {
    //Jesli znajdujemy sie ponizej pewnej zadanej wysokosci, to wstawiamy skale
    if (y < height - 1 - rocks)
      return Voxel(Voxel::STONE);
    //Jesli jest to pierwszy voxel od gory, nalezy wstawic trawe
    else if (y == height - 1)
      return Voxel(Voxel::GRASS);
    //W kazdym innym przypadku wstawiamy ziemie
    else
      return Voxel(Voxel::DIRT);
  }
  else {
    return Voxel(Voxel::NONE);
  }
}
```

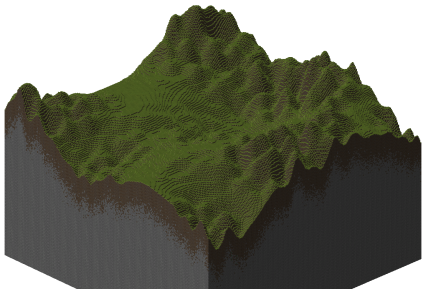
Listing 3.4: Funkcja determinująca rodzaj woksela w przestrzeni



Rysunek 3.11: Losowy teren o parametrach: rockOffset = 7, rockSharpness = 1, smoothness = 256, detail = 1, spread = 32

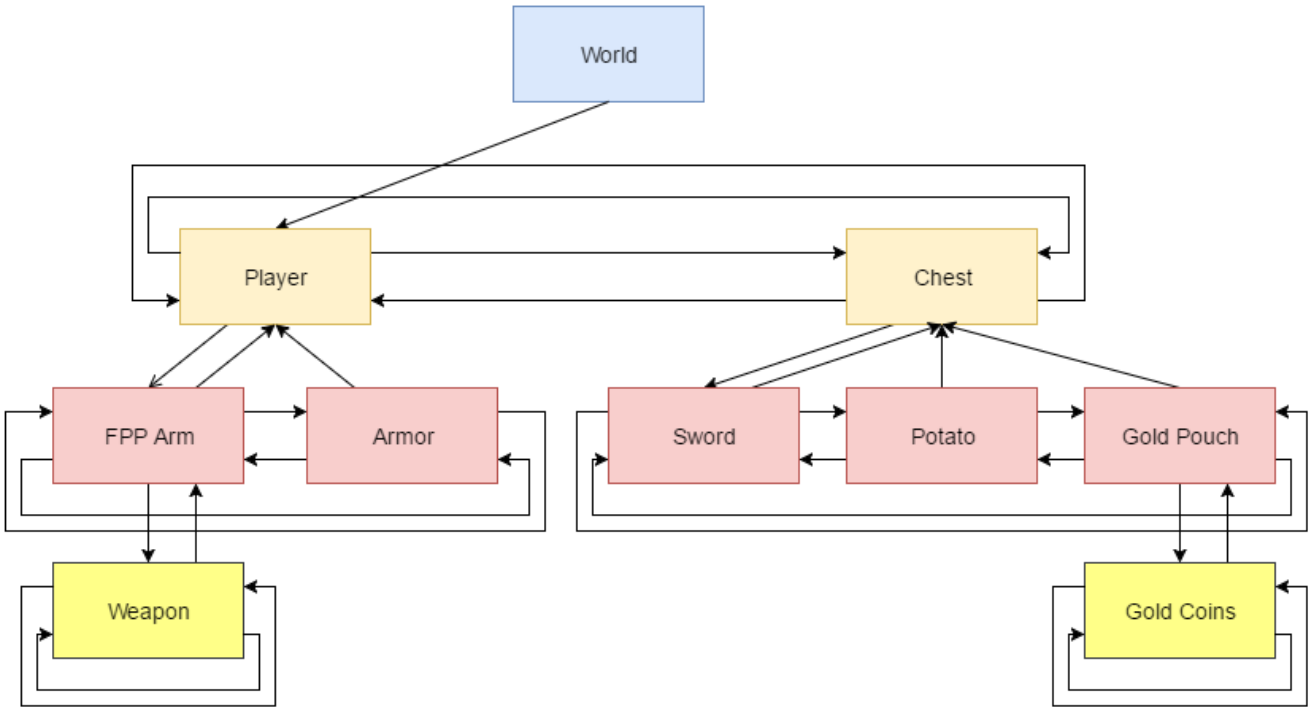


Rysunek 3.12: Losowy teren o zmienionych parametrach: detail = 2, smoothness = 128, rockOffset = 25, rockSharpness = 6



Rysunek 3.13: Losowy teren o dodatkowo zmienionym parametrze sperad = 64

Do generowania terenu w programie używam funkcji generującej wartości dla całych chunk'ów. Pozwala to zmniejszyć ilość próbkowania szumu, ponieważ dla każdego chunk'a o rozmiarach 16x16x16 wokseli, wystarczy wygenerować szum o rozdzielczości 16x16.



Rysunek 4.1: Przykładowa struktura węzłów połączonych

4 Organizacja świata gry

4.1 Drzewo list cyklicznych

Drzewo list cyklicznych jest bardzo prostą, ale też potężną strukturą do przedstawienia obiektów świata gry oraz ich zarządzaniem. Korzysta ono z cechy programowania obiektowego, jaką jest polimorfizm. Węzeł drzewa cyklicznego posiada powiązania z rodzicem, pierwszym dzieckiem, następnym i poprzednim elementem, przy czym pierwszy i ostatni dodatkowo wskazują na siebie, stąd nazwa list cyklicznych. Takie hierarchiczne podejście do problemu przedstawienia zależności między obiektami, znacznie zwiększa rozszerzalność programu, ponieważ by dodać nowy obiekt, wystarczy dziedziczyć po już istniejącym obiekcie, dziedzicząc wszystkie jego właściwości. Jedyne co musimy zrobić, to zaimplementować jedną z metod wywoływanych na konkretnym etapie pętli świata gry.
Metody obsługiwane przez drzewo, to:

- **OnInit()** - uruchomi się tylko raz, na samym początku inicjalizacji obiektu
- **OnUpdate()** - będzie uruchamiać się co generowaną klatkę
- **OnLateUpdate()** - będzie uruchamiać się po uruchomieniu wszystkich Update, jeśli chcemy by coś wykonało się na końcu - np aktualizacja pozycji kamery
- **OnDraw(Renderer* renderer)** - uruchamiana co każdą klatkę w momencie rysowania - patrz rysunek 7.4
- **OnLateDraw(Renderer* renderer)** - używane, by narysować coś na końcu - np by ominąć bufor głębokości lub wyrenderować obiekty przezroczyste

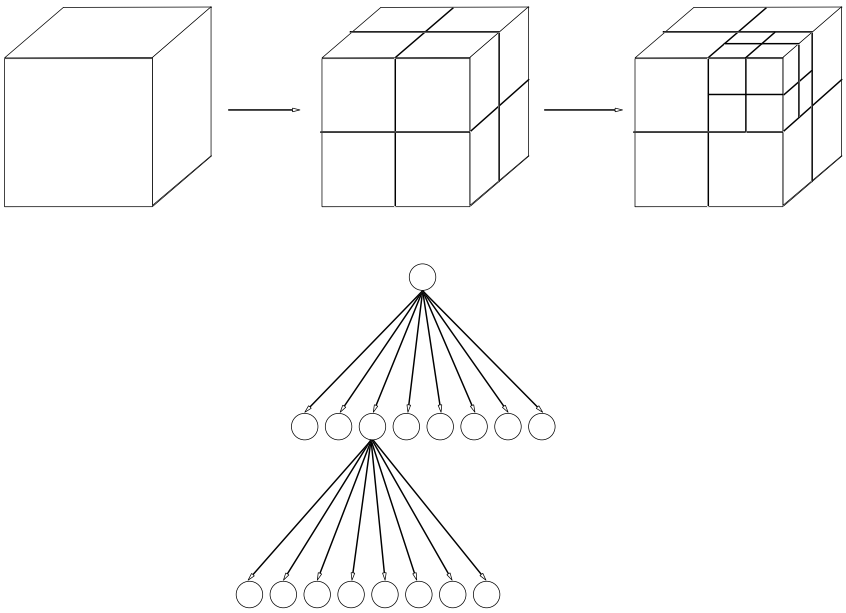
- **OnCollision(CollisionInfo* info)** - obsługa kolizji. By zdefiniować klasę obiektu, można użyć funkcji *CompareTag(const std::string& tag)*, po wcześniejszym przypisaniu znacznika do obiektu - dostępne tylko dla fizycznych obiektów
- **OnDestroy()** - uruchamiane w momencie gdy obiekt jest niszczony

Zaimplementowałem też funkcje, która potrafi klonować całe gałęzie drzewa. Mógłbym dla przykładu sklonować obiekt Chest wraz ze wszystkimi wezłami potomnymi i wstawić ją w inne miejsce. Aby to było możliwe, każda klasa dziedzicząca musi zaimplementować konstruktor kopiujący i metodę *GameObject* Clone()*. Pozwala to na zapisanie w pamięci gotowych szablonów obiektów lub całych gałęzi obiektów i wstawiania ich w odpowiednie miejsce. Kolejną zaletą tego jest możliwość wykorzystania macierzy przekształceń rodzica. Mamy obiekt Player, który ma jako dziecko obiekt FPP Arm. Niech to będzie model ręki widoczny z pierwszej osoby. Powinien on zawsze podążać za pozycją swojego rodzica. Gdybyśmy nie stosowali hierarchicznej struktury danych, kolejność przekształceń mogłaby być różna i pozycja obiektu FPP Arm mogłaby być obliczona zanim zostanie obliczona pozycja jego rodzica. Wtedy pozycja dziecka byłaby opóźniona o jeden cykl pętli gry w stosunku do pozycji rodzica. Podejście hierarchiczne pozwala wywołać metodę na korzeniu drzewa, które zrealizuje odpowiednie akcje u siebie, a następnie wywoła ją dla wszystkich swoich dzieci o obiektów siostrzanych. W sposób rekurencyjny zostanie to wywołane dla wszystkich obiektów drzewa. Podobnie sytuacja ma się z obiektem Weapon, który powinien podążać za FPP Arm.
...TODO: obrazek hierarchii obiektów...
Każdy GameObject posiada obiekt klasy Transform opisującą pozycję, skalę oraz rotację obiektu. Klasa Transform posiada wskaźnik na rodzica, którego macierz przekształcenia dołożymy do swojej. Jednak nie chcemy zawsze obliczać macierzy, a jedynie gdy pozycja danego obiektu lub jego rodzica się zmieni. Dlatego wewnątrz klasy transform umieściłem dwie macierze: lokalną macierz modelowania i macierz modelowania będącą złożeniem macierzy rodzica i swojej lokalnej. Pozwala to na zaoszczędzenie paru operacji mnożenia macierzy, ponieważ spora część obiektów może być statyczna.

4.2 Drzewo ósemkowe

Drzewo ósemkowe, w przeciwieństwie do drzewa list cyklicznych, jest posortowaną strukturą danych. Każdy węzeł może mieć 8 dzieci, a elementy zostają przydzielone do każdego dziecka według pewnych wariantów.
Uproszczona definicja drzewa ósemkowego, którą będę poniżej sukcesywnie uzupełniał o dodatkowe pola i metody, została przedstawiona na listingu 4.1.

```
1  class Octree
2  {
3  public:
4    typedef std::list<PhysicalObject *> PhysicalObjects;
5    typedef std::queue<PhysicalObject *> PhysicalObjectsQueue;
6    typedef std::queue<Chunk *> ChunksQueue;
7    typedef std::vector<Chunk *> Chunks;
8
9    void SetBoundingArea(const BoundingBox& area);
10
11   bool IsSmallestLeaf();
12
13   //Umieszcza w drzewie wszystkie oczekujace elementy
14   void UpdateTree();
15   //Aktualizuje elementy drzewa i usuwa puste galezie
16   void Update();
17
```



Rysunek 4.2: Hierarchia drzewa ósemkowego w trzech i dwóch wymiarach

```
18   //Dodaje elementy do kolejki obiektow oczekujacych na umieszczenie w drzewie
19   void Add(PhysicalObject* object);
20   void Add(const PhysicalObjects& objects);
21   void Add(Chunk* chunk);
22   void Add(const Chunks& chunks);
23
24   //Modyfikuje jeden woksel w drzewie
25   void Insert(const Voxel& voxel, Vector3 coordinates);
26
27   //Renderuje wszystkie chunk'i w drzewie
28   void Draw(Renderer* renderer);
29
30   //Sprawdza kolizje promienia z terenem
31   void CheckRayCollision(Ray *ray, RayIntersection* intersectionInfo);
32 private:
33   //Obszar ktory jest obejmowany przez wezel
34   BoundingBox _area;
35
36   //Wskaznik na rodzica wezla
37   Octree *_parent;
38   //Dzieci wezla
39   Octree *_children[8];
40
41   //Kolejki oczekujacych obiektow
42   PhysicalObjectsQueue _pendingObjects;
43   ChunksQueue _pendingChunks;
44
45   //Minimalny rozmiar drzewa
46   const float _minimumSize = (float)Chunk::dimension;
```

```
47    const int _initialMaxLifetime = 8;
48    const int _maximumLifetime = 64;
49
50    void Insert(PhysicalObject *obj);
51    void Insert(Chunk* chunk, VoxelMesh* mesh);
52    //Budowanie drzewa za pierwszym razem
53    void BuildTree();
54  };
```

Listing 4.1: Definicja drzewa ósemkowego

Aby dodać elementy do drzewa, należy użyć metody *Add*. Spowoduje to dodanie elementu do kolejki, odpowiednio kolejki oczekujących chunk'ów oraz oczekujących obiektów. Potem, przy wywołaniu metody UpdateTree(), dodawanie elementów do drzewa występuje na dwa sposoby:

1. Dodawanie elementów za pierwszym razem - budowanie drzewa
2. Dodawanie elementów do istniejącego drzewa

W pierwszym przypadku wykonujemy następujące kroki:

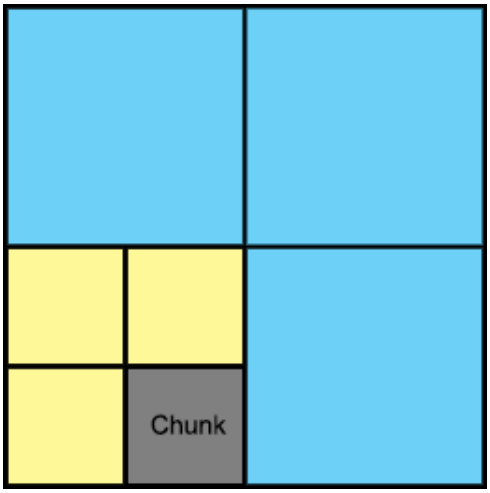
1. Wepchnij wszystkie elementy do korzenia
2. Porównaj każdy obiekt znajdujący się w węźle z każdym dzieckiem by dopasować go do najbardziej odpowiedniego potomka
3. Jeśli rozprowadziłeś wszystkie możliwe obiekty, to dla każdego nowo utworzonego węzła potomnego powtarzaj krok 2

Jest za to odpowiedzialna metoda BuildTree().
Gdy drzewo już jest zbudowane, dokonujemy dla każdego wkładanego do drzewa obiektu, by nie tworzyć drzewa od samego początku.
Aktualizacją już istniejących elementów zajmuje się metoda Update().
Wykorzystałem drzewo ósemkowe do trzech celów:

- Organizacja chunk'ów
- Optymalizacja wykrywania kolizji
- Usuwanie niewidocznych obiektów

4.2.1 Organizacja chunk'ów

Zastosowanie drzewa ósemkowego do wyświetlania chunk'ów pozwala na organizacje danych oraz ograniczenie zużycia pamięci. Załóżmy, że najmniejszy węzeł będzie posiadał rozmiar chunk'a oraz każdy węzeł będzie posiadał informacje o tym, czy posiada w swojej hierarchii dziecko z niepustym chunk'iem (lub posiada taki chunk, jeśli jest najmniejszym węzłem). Jeśli mamy pewien świat gry, w którym bardzo duża część powierzchni jest powietrzem - czyli woksele są puste, ale chcemy, by gracz mógł tą przestrzeń wykorzystać bo należy ona do świata gry, to przechowując chunk'i w postaci tablicy musielibyśmy przechowywać każdy element siatki który by był pusty. Przy użyciu drzewa ósemkowego możemy przechowywać jedynie największy możliwy węzeł i zawrzeć informacje, że żaden węzeł potomny (jeśli istnieje), nie posiada żadnego niepustego chunk'a.
Przykład takiego podziału jest przedstawiony na rysunku 4.3. Kolorem żółtym zaznaczono najmniejsze, puste węzły. Natomiast kolorem niebieskim węzły, dla których nie dokonano podziału do najmniejszego



Rysunek 4.3: Sposób podziału drzewa w dwóch wymiarach, gdy wykorzystywany jest tylko jeden niepusty chunk

węzła.

Rozszerzmy definicję drzewa ósemkowego o pola :

```
class Octree
{
  /*...*/

private:
  /*...*/
  //Lista chunkow, ktore zostana seryjnie dodane do drzewa
  Chunks _chunks;
  //Wskaznik do chunka, uzywane tylko jesli wezel jest lisciem
  Chunk* _chunk;
  //Model chunka
  VoxelMesh* _chunkMesh;
  //Pole bitowe zawierajace informacje, ktore wezly potomne posiadaja niepustego chunka
  uint8_t _chunkChildren;
};
```

Listing 4.2: Rozszerzenie drzewa o

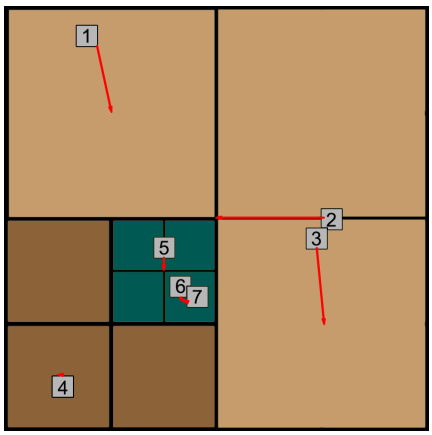
Gdy używamy metody Add, chunk jest wkładany do kolejki. Następnie, jeśli budujemy drzewo za pierwszym razem to najprawdopodobniej do drzewa będzie włożony cały wygenerowany teren. By nie tworzyć podziału drzewa dla każdego obiektu, możemy elementy sukcesywnie przekładać do wektora **_chunk** w kolejnych, pasujących węzłach potomnych, dopóki nie zostaną one rozprowadzone do najmniejszych węzłów drzewa. Pozwala to w bardziej efektywny sposób włożyć dużą liczbę obiektów do drzewa, ponieważ dla każdego węzła wyznaczamy zakres współrzędnych obejmowanych przez jego węzły pochodne tylko raz.
Gdy dodajemy pojedynczy chunk do drzewa, już po jego zbudowaniu, metoda Insert w każdym węźle musi obliczyć obszary obejmowane przez węzły potomne, by dopasować chunk do odpowiedniego dziecka. Pole bitowe **_chunkChildren** jest informacja o węzłach potomnych posiadających niepuste chunk'i. Jako że w jednym drzewie będą przechowywane informacje zarówno o obiektach fizycznych jak i chunk'ach, pole bitowe pozwala pominąć sprawdzenie zawartości węzłów, które istnieją bo przechowują obiekty fizyczne. Ponadto, sprawdzenie *if(_chunkChildren == 0)* jest mało kosztowną operacją, pozwalająca sprawdzić

czy węzły potomne przechowują niepuste chunk'i.

4.2.2 Wykrywanie kolizji

Scena w grze komputerowej może zawierać setki, a nawet tysiące różnych obiektów fizycznych. Naiwny algorytm sprawdzenia kolizji to sprawdzanie ich każdy obiekt z każdym. Jeśli kolizje sprawdzamy tylko przy pomocy kuli - tzn. jeśli odległość od środków kul jest mniejsza od sumy ich promieni - to koszt tych operacji nie będzie aż tak wielki. Jednak w przypadku systemu, który używa bardziej dokładnych algorytmów obliczania kolizji, złożoność obliczeniowa $O(n^2)$ może skutecznie uniemożliwić rozgrywkę przy większej liczbie obiektów. Jeśli mamy ich 1000, to należy dokonać 1 000 000 sprawdzeń kolizji co każdą pętlę.

Można spróbować podzielić świat gry na mniejsze segmenty, a następnie posegregować obiekty według

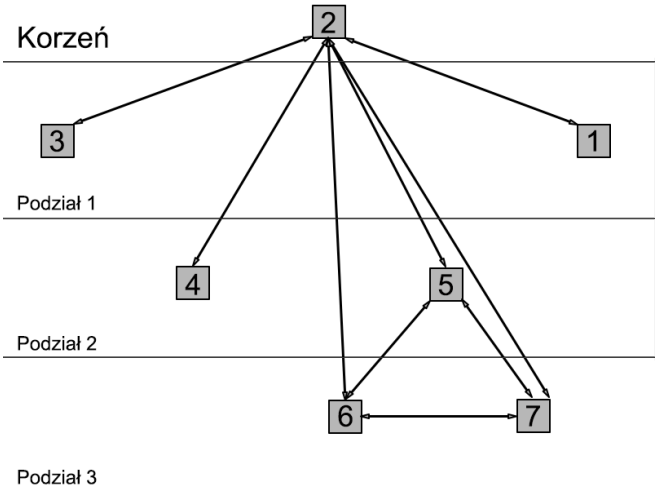


Rysunek 4.4: Przykładowy rozkład obiektów w 2D - strzałki symbolizują przynależność do węzła

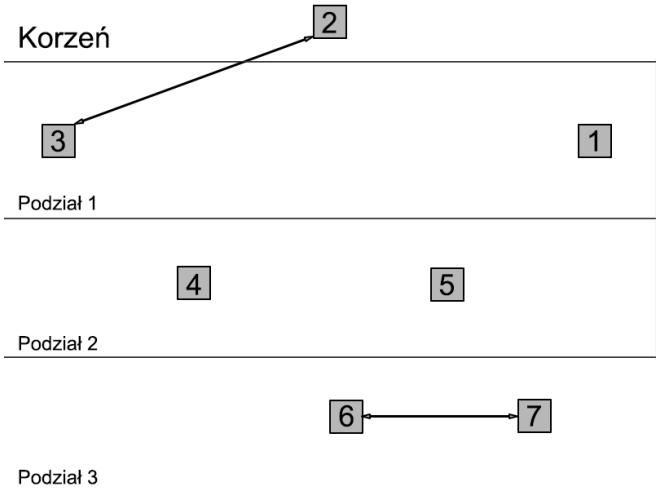
ich pozycji i rozmiaru, do odpowiednich sektorów. Świat gry został podzielony przez drzewo ósemkowe, wystarczy teraz jedynie posegregować odpowiednio obiekty.
Możemy rozróżnić dwa przypadki:

1. Obiekt w całości znajduje się w węźle potomnym
2. Obiekt jedynie przecina węzeł potomny

W pierwszym przypadku wystarczy dodać obiekt do węzła. Jeśli nie istnieją **żadne** węzły potomne wkładany obiekt będzie pierwszym obiektem w danym węźle, to nie musimy robić kolejnego podziału. Węzeł zostaje przypisany do obecnego węzła. Gdyby jednak w tym węźle był już jeden element, to należy dokonać sprawdzenia ze wszystkimi dziećmi dla obydwóch obiektów.
W drugim przypadku, jeśli obiekt przecina kilka węzłów potomnych, to przechowujemy obiekt w najmniejszym węźle, który może w całości pomieścić dany obiekt. Sytuację tą przedstawia rysunek 4.4. Obiekt 2 nie mieści się w żadnym węźle potomnym, dlatego zostanie przydzielony do korzenia, ponieważ potencjalnie może kolidować z każdym obiektem w każdym węźle potomnym. Podobnie wygląda sytuacja z obiektem 5. Na rysunku 4.5 widzimy jakie kolizje musimy potencjalnie sprawdzić. Można zastosować pewną optymalizację: w pierwszej kolejności sprawdzać kolizję z każdym węzłem potomnym. Jeśli taka kolizja nie nastąpi, możemy pominąć sprawdzanie kolizji ze wszystkimi obiektami znajdującymi się w węźle. Pozostałe kolizje do sprawdzenia po takiej operacji prezentuje rysunek 4.6.



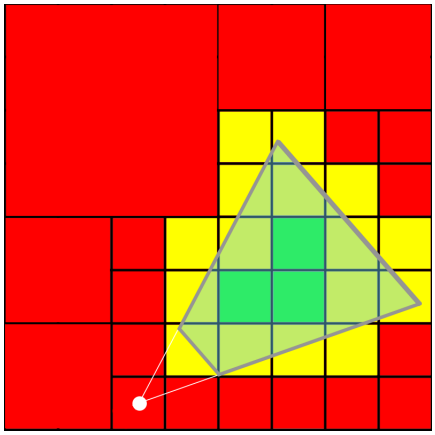
Rysunek 4.5: Kolizje, które teoretycznie należałoby sprawdzić



Rysunek 4.6: Kolizje do sprawdzenia, po sprawdzeniu kolizji z węzłami potomnymi

4.2.3 Obcinanie niewidocznych powierzchni

W tym podrozdziale omówię jedynie optymalizację obcinania obiektów poza bryłą widoku, wynikającą z korzystania z drzewa ósemkowego. Dokładny opis samego algorytmu znajduje się w rodziale 6.
Drzewo ósemkowe pozwala pominąć sprawdzanie widoczności dużej ilości obiektów w świecie. Tym większą, im większy jest świat.
Mając drzewo ósemkowe. w pierwszej kolejności należy sprawdzić widoczność węzła. Jeśli węzeł jest niewidoczny, to możemy pominąć dalsze sprawdzanie. Przykładowe obszary sprawdzania w rzucie 2D przedstawia rysunek 4.7.



Rysunek 4.7: Obszary, które należy sprawdzić przy takiej bryle widoku. Czerwone obszary znajdują się w całości poza bryłą widoku.

5 Kolizje

Aby umożliwić interakcję ze światem gry, należy zadbać o to, by obiekty mogły ze sobą oddziaływać. Każdy fizyczny obiekt można przybliżyć pewną figurą geometryczną lub zbiorem prostszych figur. Korzystając z silnika wokselowego większość obiektów można dobrze przybliżyć AABB (axis aligned bounding box), pod warunkiem, że nie będą one zbyt nieproporcjonalne jeśli chodzi o długość i szerokość obiektu - wtedy przy obrotach mogłoby wystąpić przenikanie obiektów albo zbyt duża ilość pustej przestrzeni i kolizje byłyby przez to niedokładne.
W silniku opracowałem trzy rodzaje zderzeń:

1. AABB - AABB
2. AABB - teren
3. promień - AABB

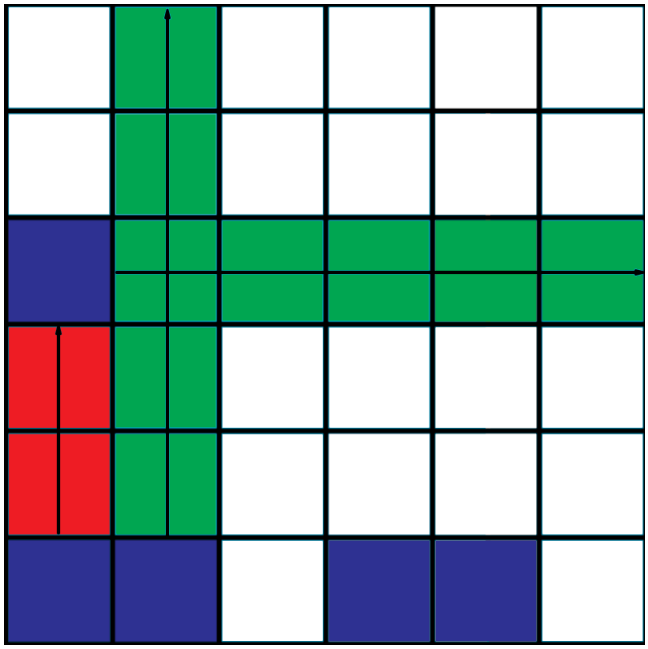
5.1 Zderzenie AABB - AABB

Wykrywanie zderzeń dwóch AABB jest bardzo proste. Załóżmy, że mamy dwa sześciany: A oraz B. Ich środki wyznaczają punkty $a = (a_x, a_y, a_z)$ oraz $b = (b_x, b_y, b_z)$, a ich wymiary to kolejno: (a_{xs}, a_{ys}, a_{zs}) oraz (b_{xs}, b_{ys}, b_{zs}). Po rzutowaniu AABB na każdą z osi, zderzeni nastąpi jeśli zostaną spełnione wszystkie warunki:

$$2|a_x - b_x| <= a_{xs} + b_{xs}$$
$$2|a_y - b_y| <= a_{ys} + b_{ys}$$
$$2|a_z - b_z| <= a_{zs} + b_{zs}$$

5.2 Zderzenie AABB - teren

Wykrywanie kolizji z terenem w silniku wokselowym nie jest najprostszym zadaniem, dla obiektów o rozmiarze rozciągającym się na więcej wokseli. Jeśli jakiś obiekt koliduje z terenem, to chcielibyśmy wiedzieć z jakiej strony nastąpiła kolizja, by później móc ją odpowiednio zinterpretować w procedurze obsługi kolizji.
Na potrzeby wykrywania tych kolizji opracowałem własny algorytm. Nie jest bardzo wydajny i dziwnie zachowuje się na krawędziach wokseli, ale na potrzeby projektowanego silnika jest wystarczający.
Najprostszym sposobem by określić, czy kolizja w ogóle nastąpiła, to rzutowanie AABB na współrzędne wokseli (zaokrąglenie współrzędnych w dół), i sprawdzenie czy na tym całym obszarze nie występuje jakiś woksel.
Założeniem algorytmu jest sprawdzanie kolizji podczas ruchu, a nie teleportacji obiektu.
By zdeterminować kierunek kolizji sprawdzamy wszystkie płaszczyzny. Załóżmy, że sprawdzamy kolizję od dołu. Weźmy $floor(y\_min)$ AABB dla którego będziemy sprawdzać kolizję.
Jeśli kolizja nastąpiła, to dla $u \in (floor(z\_min), floor(z\_max))$ oraz $v \in (floor(x\_min), floor(x\_max))$ musi istnieć przynajmniej jeden taki niepusty woksel $V(v, y\_min, u)$, który od góry nie jest zakryty przez żaden inny niepusty woksel. Takie samo sprawdzenie wykonujemy dla innych płaszczyzn. Jeśli nie będzie takiego woksela, to kolizja będzie z innej strony albo nie będzie jej wcale. Problem następuje przy wyskakiwanie na woksele o wysokości 1. Wtedy zarówno zostanie wykryta kolizja z boku jak i od góry.
Przykładowy schemat działania algorytmu pokazuje rysunek 5.1. Niebieskie obszary to woksele, zielone obszary to ścieżki determinujące kierunek kolizji. Czerwone ścieżki to kolizje, które nie determinują kierunku.
Przy analizie wyników kolizji należy wziąć pod uwagę kierunek poruszania się i jeśli przykładowo kolizja



Rysunek 5.1: Prezentacja algorytmu. Kolizja zostanie wykryta z lewej strony i od dołu

następuje ze strony północnej i poruszamy się w stronę ujemnej osi z, to taka kolizja miała prawo nastąpić i należy ją obsłużyć. W przeciwnym wypadku ignorujemy ją.
Zastosowanie drzewa ósemkowego do przechowywania wokseli wprowadziło bardzo dużo kompilkacji przy testowaniu kolizji AABB z wieloma chunkami. Przede wszystkim musimy wydobyć ciągły obszar chunk'ów, z którymi będziemy testować obiekt. By uprościć sprawę, ograniczyłem rozmiar AABB do rozmiaru chunk'a. Gwarantuje to, że będzie on kolidował maksymalnie z 8 chunk'ami.
By uzyskać potrzebne chunki, testuję 8 wierzchołków AABB z chunkami w węzłach potomnych i wybieram te unikalne. Następnie układam je w regularną strukturę. Porównujemy środki kolejnych chunk'ów ze środkiem AABB. Ograniczenie rozmiaru zapewniło, ze środek AABB będzie zawsze między którymiś ośmioma chunkami. Według tej relacji chunk'i zostają indeksowane jak na rysunku 5.2.
Mając regularną strukturę danych można zauważyć, że jeśli nasze koordynaty wykroczą poza obszar danego chunk'a, zaczynając z indeksem zero, możemy automatycznie zwiększać indeks chunk'a w strukturze:

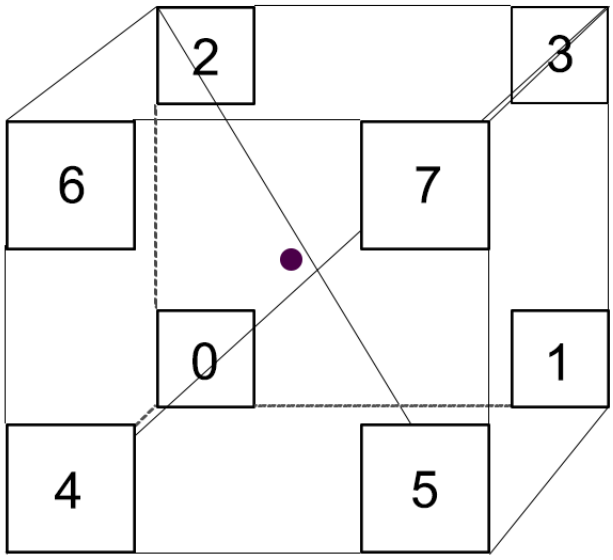
- jeśli $x > chunk\_size$ to $chunkIndex = chunkIndex + 1$
- jeśli $y > chunk\_size$ to $chunkIndex = chunkIndex + 2$
- jeśli $z > chunk\_size$ to $chunkIndex = chunkIndex + 4$

Tym sposobem jesteśmy w stanie ujednolicić strukturę chunków znajdujących się w różnych węzłach w drzewie ósemkowym i skrócić znacząco kod sprawdzania kolizji dla 6 płaszczyzn. Udało mi się go ograniczyć dzięki temu podejściu do dwóch pętli for - jedna kierunku ruchu ujemnego, druga dla kierunku ruchu dodatniego.

5.3 Zderzenie promień - teren

Kolizja promień-teren została wykorzystana do modyfikacji terenu. Promień to półprosta o początku w punkcie $P_0 = (x_0, y_0)$ oraz znormalizowanym kierunku $\hat{d}$ i prametrze t określającym punkt promienia.

$$ray = P_0 + \hat{d}t$$



Rysunek 5.2: Położenie kolejnych chunków w stosunku do środka sześcianu z którym badana jest kolizja

Do wykrywania kolizji promienia z terenem wykorzystałem bardzo naiwne podejście: interpolacje promienia z pewną dokładnością, dopóki długość promienia nie zostanie przekroczona.
Dokładność dobrałem w taki sposób, żeby zminimalizować możliwość kolizji z wokselem leżącym na przykład po przekątnej. Promień sprawdzany jest w drzewie ósemkowym, testując jego obecną pozycję. Jeśli interpolacja pozycji promienia nie przekroczyła jego długości, ale wyszliśmy poza zakres drzewa ósemkowego, to zostaje on wypchnięty rodzica danego drzewa i testowany na nowo.
Do określenia strony dodawania kolizji użyłem bardzo prostego podejścia. By określić kierunek przeciwny do tego, w którym się patrzę, obliczam $\min(\hat{d}\cdot\pm\hat{X}, \hat{d}\cdot\pm\hat{Y}, \hat{d}\cdot\pm\hat{Z})$, gdzie $\hat{X}, \hat{Y}, \hat{Z}$ określają kierunki wzdłuż osi X, Y, Z, a $\hat{d}$ to kierunek na który patrz się gracz. Znormalizowany wektor skierowany wzdłuż którejś z osi, dla której wystąpiło minimum, zostanie dodany do współrzędnych dla których wykryto kolizję promienia z wokselem. Bardziej precyzyjny system wykrywania kolizji wymagałby określenia płaszczyzny przecięcia woksela przez promień i dodania woksela nad tą płaszczyzną.

6 Obcinanie obiektów poza bryłą widoku

Wyświetlanie wielkiej ilości wokseli jest bardzo kosztowne. Dla każdego chunk'a musimy dostarczyć shaderowi macierz modelowania, a także odpowiednią liczbę wierzchołków, które nie są przechowywane przez jeden VAO (Vertex Array Object), by nie trzeba było od nowa przeliczać całego świata w momencie modyfikacji jednego woksela.
Rozwiązaniem polepszającym optymalizację i redukcję rysowanych obiektów jest pominięcie rysowania

obiektów znajdujących się poza bryłą widzenia (ang. View frustum culling). Obiekty te i tak nie zostałyby rysowane, jednak pomijając je na etapie dowiązywania zasobów do karty, pomijamy koszty związane z przesyłem danych do karty graficznej.
Dla widoku perspektywicznego bryła widzenia ma kształt ostrosłupa ściętego. Każdy obiekt fizyczny można otoczyć AABB, czyli sześcianem, którego płaszczyzny są równoległe bądź prostopadłe do wszystkich osi. To uproszczenie pozwala na szybkie testowanie widoczności sześcianu przez płaszczyzny tworzące bryłę widoku.

6.1 Otrzymywanie płaszczyzn bryły widoku

Każdy punkt, który jest analizowany przez shadery, powinien się znajdować w znormalizowanej przestrzeni przycięcia. Bryła widoku w tej przestrzeni jest przedstawiona jako sześcian wyrównany do wszystkich osi, ze środkiem w punkcie $P_c = (0, 0, 0)$ i o wymiarach $a = 2$.
Aby punkt $p = (x, y, z, 1)$ w przestrzeni 3D przenieść do przestrzeni przycięcia, należy go przekształcić za pomocą macierzy modelowania i projekcji:

$$p_c = P * M * c$$

Otrzymany w ten sposób punkt $p_c = (x_c, y_c, z_c, w_c)$ jest we współrzędnych jednorodnych. Aby go znormalizować, należy podzielić go przez współczynnik w_c

$$p_{norm} = (x_c/w_c, y_c/w_c, z_c/w_c) = (x_n, y_n, z_n)$$

. Punkt p_{norm} będzie znajdywał się wewnątrz bryły widoku, jeśli: $-1 < x_n < 1$, $-1 < y_n < 1$, $-1 < z_n < 1$. Zatem punkt p_c będzie znajdował się wewnątrz bryły widoku jeśli: $-w_c < x_c < w_c$, $-w_c < y_c < w_c$, $-w_c < z_c < w_c$. Wynik mnożenia macierzy $P * M$ zapiszmy jako

$$PM = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Załóżmy, że chcemy określić pozycję punktu względem płaszczyzny w znormalizowanej płaszczyźnie przycięcia określonej wektorem normalnym $\hat{n} = (1.0f, 0.0f, 0.0f)$ i parametr $D = -1$. Punkt p_c będzie po dodatniej stronie tej płaszczyzny gdy spełniony będzie warunek: $-w_c < x_c$.
Doknując mnożenia przez macierze PM otrzymujemy:

$$x_c = xa_{11} + ya_{12} + za_{13} + a_{14}$$

$$w_c = xa_{41} + ya_{42} + za_{43} + a_{44}$$

Zatem po podstawieniu, spełnione musi być poniższe równanie

$$-(xa_{41} + ya_{42} + za_{43} + a_{44}) < xa_{11} + ya_{12} + za_{13} + a_{14}$$

Po przekształceniu, otrzymujemy równanie płaszczyzny $Ax + By + Cz + D > 0$ we współrzędnych świata:

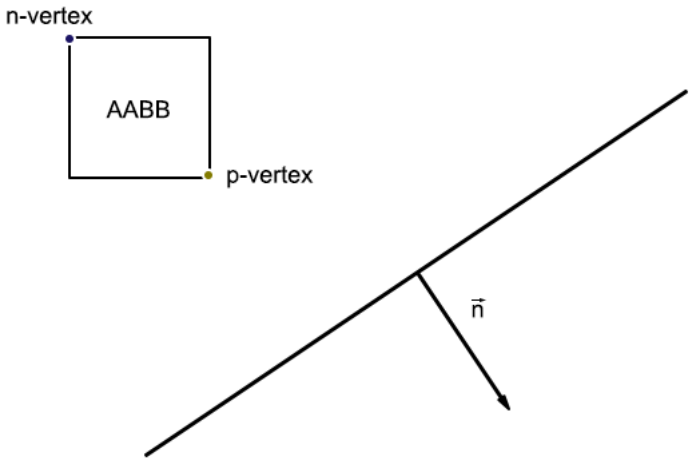
$$(a_{11} + a_{41})x + (a_{12} + a42)y + (a_{13} + a43)z + a_{14} + a44 > 0$$

Analogiczne operacje wykonujemy dla pozostałych płaszczyzn w znormalizowanej przestrzeni przycięcia. Otrzymujemy w ten sposób 6 płaszczyzn definiujących bryłę widoku we współrzędnych świata.

6.2 Sprawdzanie widoczności AABB

Do szybkiego sprawdzenia kolizji AABB z bryłą widoku można skorzystać z dwóch punktów p-vertex (ang. positive vertex) oraz n-vertex (ang. negative vertex), czyli wierzchołki maksymalne i minimalne. Pozycję tych wierzchołków, w relacji do płaszczyzny, przedstawia rysunek 6.1.

Jeśli wierzchołki maksymalne znajdują się po dodatniej stronie dla każdej z płaszczyzn, to znaczy,



Rysunek 6.1: Lokalizacja p-vertex oraz n-vertex w stosunku do przykładowej płaszczyzny

że AABB przecina płaszczyznę. Jeśli chcielibyśmy dodatkowo sprawdzić, czy obiekt w całości znajduje się wewnątrz bryły widoku, należałoby dodatkowo sprawdzić pozycję minimalnych wierzchołków - jeśli wszystkie znajdą się po dodatniej stronie każdej z płaszczyzn, to obiekt jest w całości wewnątrz bryły widoku.

Pozycję tych wierzchołków można znaleźć na podstawie normalnej płaszczyzny, co przedstawia listing 6.1.

```
void
BoundingBox::GetPVertex(Vector3* pVertex, const Vector3& normal) const
{
  pVertex->x = normal.x >= 0 ? _max.x : _min.x;
  pVertex->y = normal.y >= 0 ? _max.y : _min.y;
  pVertex->z = normal.z >= 0 ? _max.z : _min.z;
}

void
BoundingBox::GetNVertex(Vector3* nVertex, const Vector3& normal) const
{
  nVertex->x = normal.x >= 0 ? _min.x : _max.x;
  nVertex->y = normal.y >= 0 ? _min.y : _max.y;
  nVertex->z = normal.z >= 0 ? _min.z : _max.z;
}
```

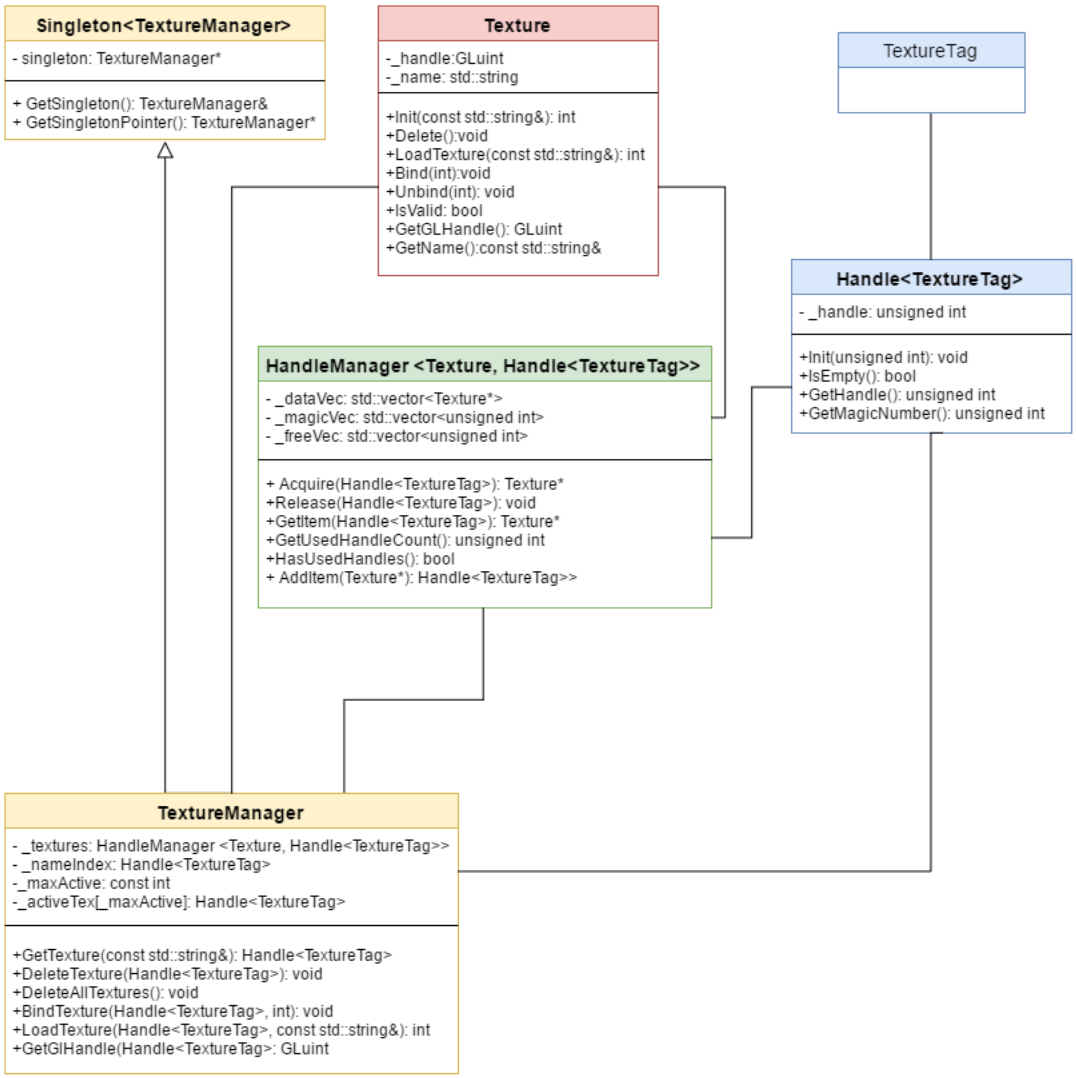
Listing 6.1: Wyznaczanie p-vertex i n-vertex

Posiadając wierzchołek, wystarczy jedynie sprawdzić jego odległość od płaszczyzny, korzystając z równania płaszczyzny $p - vert \cdot \hat{n} + D = distance$. Wektor $\hat{n}$ nie musi być znormalizowany dopóki interesuje po której stronie płaszczyzny znajduje się punkt. Jeśli chcielibyśmy znać dokładną odległość, należy ją znormalizować.

7 Budowa silnika gry

Silnik gry jest bardzo złożoną strukturą składającą się z wielu ważnych komponentów, które ściśle ze sobą współpracują. Silnik powinien posiadać menadżery zasobów, by nie duplikować wiele razy tych samych danych oraz by zarządzać dostępem do nich, system zarządzania wejściem/wyjściem by zapewnić komunikację aplikacji z użytkownikiem oraz struktury organizujące świat gry. Nie przedstawię tu wszystkich klas i struktur, jedynie skupię się na najciekawszych i najważniejszych. Pominę drzewo węzłów cyklicznych oraz drzewo ósemkowe, ponieważ zostały one częściowo opisane w rozdziale 4.

7.1 Menadżery zasobów



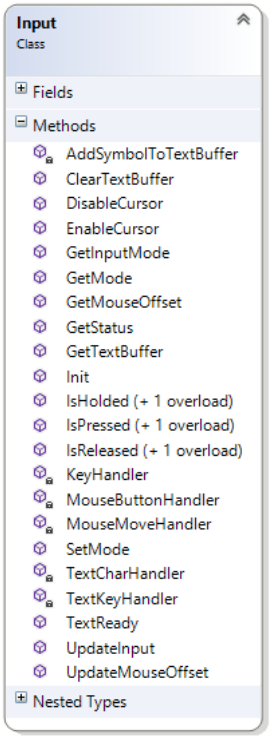
Rysunek 7.1: Menadżer tekstur i jego relacje

Silnik korzysta z kilku menadżerów zasobów:

1. Menadżer shaderów
2. Menadżer programów OpenGL

3. Menadżer pipeline'ów OpenGL
4. Menadżer tekstur
5. Menadżer modeli
6. Menadżer tablic wokseli
7. Menadżer plików tekstowych

Wszystkie klasy menadżerów są dostępne dla obiektów używanych w silniku za pomocą klasy Singleton. Pozwala nam to utworzyć menadżer zasobów w dowolnym miejscu w programie, w przeciwieństwie do zmiennych globalnych, które są inicjalizowane wraz ze startem programu. Po utworzeniu w dowolnym miejscu menadżera przy pomocy operatora new, możemy się odwoływać do dowolnego menadżera wpisując, np *textureManager*. Makrodefinicja wywoła funkcję statyczną klasy *Singleton::GetSingleton()*, przez co odwołanie z punktu widzenia programisty nie różni się niczym od odwołania do obiektu globalnego. Menadżer nie zapewnia bezpośredniego dostępu do zasobów, dzięki nie trzeba śledzić ilości referencji dla wskaźników, a gdy ktoś zwolni zasób, odpowiednia asercja da programiście znać, że dany element nie istnieje. Taki komunikat może być znacznie czytelniejszy niż błąd spowodowanym odwołaniem się do nieistniejącej pamięci. Wystarczy przechować odpowiedni uchwyt do zasobu po jego uzyskaniu.



Rysunek 7.2: Klasa input

Ponadto, klasy menadżerów pozwalają na dostęp do obiektów za pomocą nazwy, co znacznie poprawia czytelność kodu. Jeśli chcemy uzyskać jakiś istniejący obiekt lub utworzyć nowy, na przykład teksturę, wystarczy wywołać *textureManager.GetTexture(„Atlas")*. Rysunek 7.1 przedstawia klasę menadżera na przykładzie menadżera tekstur. Inne klasy menadżerów zostały stworzone w analogiczny sposób.

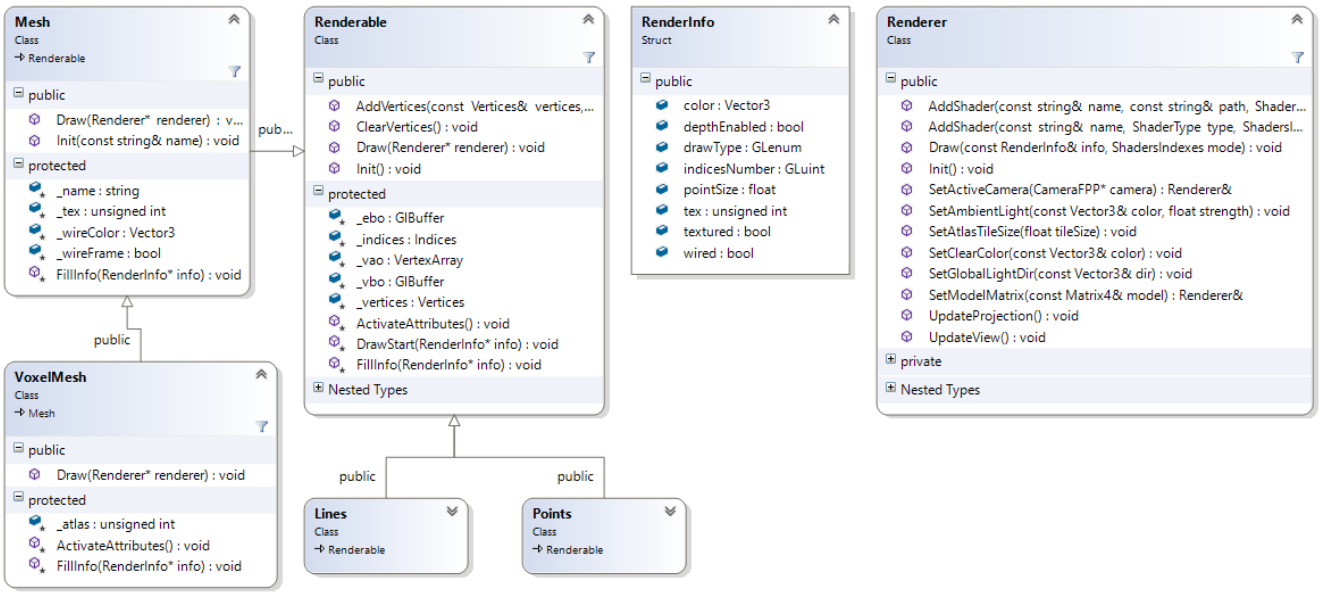
7.2 Obsługa wejścia

Klasa Input obsługuje zdarzenia wejścia. Jest odpowiedzialna za wybieranie odpowiedniego trybu wprowadzania tekstu (obsługuje tryb buforowany do wpisywania ciągów znaków oraz wpisywanie pojedynczych klawiszów). Jest także odpowiedzialna za pobieranie ruchu kursora jaki został wykonany od ostatniego wywołania funkcji UpdateInput(), dlatego ta funkcja powinna być wywoływana co wygenerowaną klatkę. Dodatkowo może ukrywać i pokazywać kursor myszy (gdy jest schowany nigdy nie opuści obszaru okna podczas ruchu) oraz przechowuje stany konkretnych klawiszy. Do sprawdzania ich stanu, posiadamy trzy funkcje do których jako argument można podać wcześniej zdefiniowane nazwy klawiszy w postaci łańcucha znakowego, lub kod klawisza wykorzystywany przez GLFW.

- Funkcja *IsHolded* pozwala sprawdzić, czy klawisz został przytrzymany dłużej niż jeden cykl pętli gry
- Funkcja *IsPressed* pozwala sprawdzić, czy klawisz został właśnie wciśnięty. Ten stan utrzymuje się jedynie jeden cykl pętli gry.
- Funkcja *IsReleased* pozwala sprawdzić, czy klawisz jest puszczony.

W obecnej fazie aplikacja wykorzystuje jedynie część funkcjonalności tej klasy. Klasa ta ma wszystkie metody statyczne aby miały do niej dostęp wszystkie obiekty w grze.

7.3 Obsługa rysowania



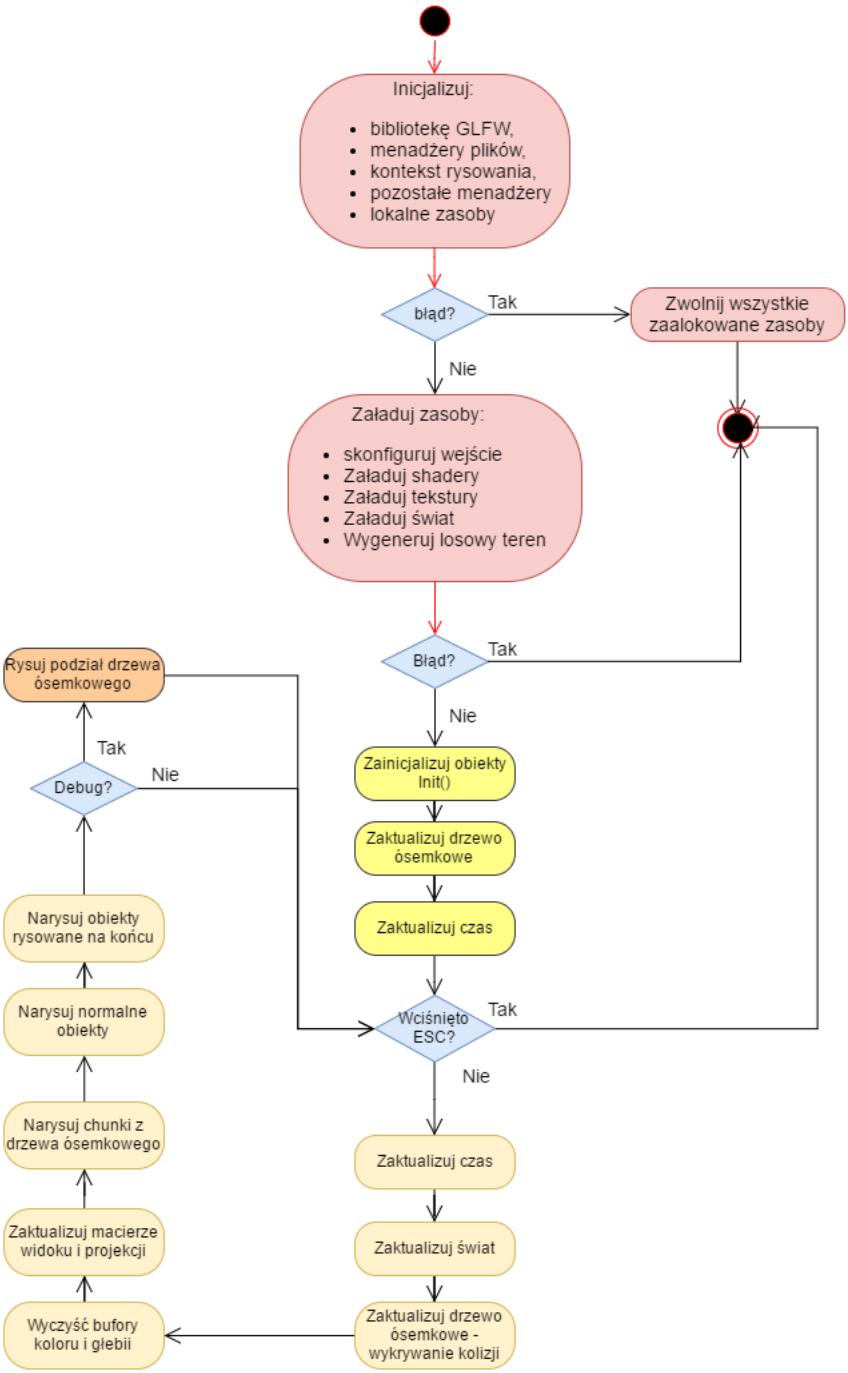
Rysunek 7.3: Klasy odpowiedzialne za renderowanie

W silniku gry pewien abstrakcjonizm dotyczący biblioteki graficznej jest wymagany, by ułatwić projektowanie samej rozgrywki. Odbywa się to kosztem elastyczności jaką daje korzystanie bezpośrednio z OpenGL'a.
Podzieliłem koncepcję renderowania na dwa etapy. Pierwszym etapem jest organizacja danych o wierzchołkach oraz dodatkowych atrybutów przekazywanych do shadera. Czynnościami tymi zajmuja się klasy dziedziczące po abstrakcyjnej klasie Renderable. Klasa Renderable posiada opakowane w klasy struktury OpenGL'a, takie jak bufory i VAO. Dodatkowo dostarcza 3 istotne funkcje wirtualne: FillInfo, która uzupełnia specjalną strukturę służącą do przesyłu danych między etapem pierwszym, a drugim, ActivateAttributes, która pozwala na dowiązywanie do shadera dodatkowych atrybutów dla każdego wierzchołka oraz najważniejsza funkcja Draw. Draw przygotowuje dane do przesyłu do OpenGL, a następniepowinno wywołać funkcję Draw klasy Renderer, do której wskaźnik jest przekazywany jako argument, wybierając shader który ma być użyty do renderingu.
Etap drugi przedstawia klasa Renderer. Otrzymuje ona wypełnioną strukturę RenderInfo i na jej podstawie ustawia odpowiednie zmienne uniform shadera. By zaoszczędzić kosztowny czas przesyłu danych do karty graficznej, klasa ta pamięta wszystkie obecnie zapisane wartości dla danego etapu shadera (lub dla wszystkich, jeśli są one wspólne) i podmienia je tylko wtedy, gdy argument, który ma być wysłany jest różny od obecnie przechowywanego. Klasa Renderer jest pomostem między shaderami a użytkownikiem i głównym celem przy jej tworzeniu było ukrycie szczegółów implementacji shaderów. Posiada ona też wskaźnik na kamerę z której ma być renderowany obraz, więc można bez problemu w trakcie gry zmieniać źródło obrazu i wprowadzać np. przerywniki filmowe.
Takie abstrakcyjne podejście do problemu wyświetlania prymitywów za pomocą biblioteki OpenGL pozwoliło mi znacznie ułatwić pracę oraz stworzyć bardziej przejrzysty i czytelniejszy kod, a dodawanie

nowych typów shaderów czy nowych typów meshów nie wymaga zbyt wielu modyfikacji.

7.4 VEngine

Wszystkie elementy składające się na świat gry, obsługę wejścia/wyjścia zostały scalone i zsynchronizowane za pomocą klasy VEngine. Czynności wykonywane przez klase zostały zaprezentowane na rysunku 7.4. Silnik posiada obiekt klasy Renderer oraz World, który prezentuje scenę. Jeśli projektant gry



Rysunek 7.4: Główna pętla silnika VEngine

chciałby stworzyć wiele poziomów i ładować je dynamicznie, to można w pamięci przechowywać wiele poziomów i wczytywać je podmieniając jedynie wskaźnik na świat oraz wywołując na nowym świecie Init().

8 Podsumowanie

W ramach pracy inżynierskiej udało mi się opracować:

1. Efektywny system optymalizacji wyświetlania dużej ilości wokseli poprzez scalanie ścian przy pomocy algorytmu greedy mesh. Algorytm ten znacznie zmniejsza ilość wyświetlanych prymitywów oraz pozwala na *meshing* w czasie rzeczywistym

2. Generator różnorodnego terenu z innymi warstwami na powierzchni i u spodu

3. Drzewo ósemkowe, które pozwala na organizację chunk'ów oraz obiektów fizycznych, których kolizje są wykrywane, a także szereg optymalizacji wynikających z braku konieczności przechowywania pustych chunków oraz szybszy test widoczności obiektów

4. Test widoczności obiektów oraz szybkie otrzymywanie płaszczyzn bryły widoku

5. Prosty, jednak niedoskonały system kolizji z terenem

6. Prosty sposób modyfikacji terenu

7. Bibliotekę matematyczną obejmującą wektory, macierze oraz kwaterniony.

Silnik wokselowy wymaga bardzo wielu zabiegów optymalizacyjnych by działał płynnie. Część rzezcy można znacznie uprościć przy użyciu wokseli, ale inne bardzo się komplikują i wymagają innego podejścia od standardowego w grafice 3D.
Aplikacja wymaga jeszcze bardzo dużo pracy, by mogła być uważana za pełnoprawną grę. Przede wszystkim należałoby poprawić system kolizji oraz budowania, przyjrzeć się algorytmowi generowania siatki z wokseli, by zaglądał do sąsiadujących chunk'ów i sprawdzał widoczność skrajnych ścian, dodać graficzny interfejs użytkownika, statystyki gracza, przeciwników ze sztuczną inteligencją oraz zarządzanie pamięcią by można było zapisywać stan gry, a także generować losowy teren w tracie rozgrywki. Obecnie aplikacja działa na jednym wątku, co też potencjalnie można by usprawnić - drzewo ósemkowe nadaje się bardzo dobrze do zrównoleglenia, ponieważ w wielu przypadkach węzły są od siebie niezależne.
Mimo niezrealizowania bardzo dużej ilości potencjalnie potrzebnych rzeczy, udało mi się bardzo dużo nauczyć na temat tworzenia interaktywnych aplikacji 3D oraz ich optymalizacji. Stworzenie od podstaw całego silnika było też niemałym wyzwaniem projektowym i nie raz zdarzało mi się przebudowywać całe klasy i struktury programów, co jest uczy by w przyszłości więcej czasu poświęcać na badanie tematu i projektowanie, by zaoszczędzić mnóstwo czasu na przerabianiu kodu.

9 Bibliografia