Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Inżynieria Systemów Informacyjnych

Rok Akademicki 2007/2008

PRACA MAGISTERSKA

Imię Studenta: *Dariusz Maciejewski*

Tytuł Pracy: *Gra komputerowa "Camera Fighter"*

Promotor:

prof. nzw. dr hab. Przemysław Rokita

Ocena: .....

.....

Podpis Przewodniczącego
Komisji Egzaminacyjnej

Inżynieria Systemów Informacyjnych

Data Urodzenia: 3 grudnia 1983

Data Rozpoczęcia Studiów: 1 października 2006

Curriculum Vitae:

Urodziłem się 3 grudnia 1983 roku w Warszawie.

W latach 1990-1998 byłem uczniem Szkoły Podstawowej im. Wiktora Gomulickiego w Warszawie. W tym czasie zacząłem się interesować informatyką. Od roku 1998 do roku 2002 uczęszczałem do Liceum Ogólnokształcącego im. Adama Mickiewicza w Warszawie. W okresie tym stworzyłem swoje pierwsze gry i programy komputerowe. Po ukończeniu liceum, rozpocząłem studia w języku angielskim na kierunku Electrical and Computer Engineering na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Od końca roku 2005 pracuję jako programista ASP.NET i MS-SQL w firmie programistycznej Eracent. W roku 2006 obroniłem swoją pracę inżynierską pt. “Komponentowy System Tworzenia Grafów – AutoGrapher”. W roku tym rozpocząłem również uzupełniające studia magisterskie na kierunku Inżynierii Systemów Informatycznych na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej.

.....

Podpis Studenta

Egzamin Magisterski

Egzamin odbył się .....

Z wynikiem: .....

Ostateczny wynik studiów: .....

Sugestie i wnioski Magisterskiej Komisji Egzaminacyjnej.....

.....

.....

STRESZCZENIE

Słowa Kluczowe:

ABSTRACT

Title: “Camera Fighter” - a computer game.

Keywords:

Spis Treści:

1. WSTĘP

1.1. CEL PRACY

1.2. ROZWAŻANE METODY IMPLEMENTACJI

1.3. ZAWARTOŚĆ PRACY

2. NOTACJA I OKREŚLENIA UŻYTE W PRACY

podstawowy tekst

`[odnośnik do bibliografii]`

ścieżka lokalna

adres internetowy

```
kod źródłowy bądź nazwa typu/metody
słowo kluczowe
// komentarz w kodzie
deklaracja metody
```

opis metody

```
formuła matematyczna
```

3. ARCHITEKTURA APLIKACJI

3.1. WINDOWS I LINUX

Polać trochę wody

3.2. STRUKTURA KATALOGÓW

/ App Framework	- szkielet aplikacji – okno, bazowa scena
/ Input	- menadżer klawiatury i myszki
/ OGL	- okno aplikacji OpenGL
/ Data	- dane oraz konfiguracja
/ models	- modele, animacje, mapy
/ shaders	- programy GPU
/ textures	- tekstury
/ Graphics	- kod związany z grafiką
/ OGL	- kod specyficzny dla OpenGL, renderowanie map
/ Extensions	- rozszerzenia OpenGL
/ Fonts	- czcionki
/ Render	- renderowanie obiektu xModel
/ Textures	- menadżer tekstur
/ Math	- biblioteka matematyczna
/ Cameras	- kamery, obszar widzenia
/ Figures	- figury 3D
/ Tracking	- śledzenie obiektów
/ Models	- menadżer modeli 3D
/ lib3ds	- biblioteka lib3ds
/ lib3dx	- autorska biblioteka modeli xModel
/ Motion Capture	- rozpoznawanie pozycji gracza z kamer
/ Multiplayer	- pobieranie pozycji gracza przez sieć (projekt)
/ Physics	- silnik fizyczny, bazowe obiekty fizyczne
/ Colliders	- detektory kolizji dla BVH

/ Verlet	- silnik i ograniczenia modelu Verlet'a
/ Source Files	- logika gry, implementacje scen
/ MenuStates	- ekrany dla sceny menu
/ Utils	- narzędzia pomocnicze, szablony klas
/ World	- implementacja obiektów tworzących świat gry

3.3. BIBLIOTEKA MATEMATYCZNA

Na potrzeby projektu przygotowano bibliotekę matematyczną, zwiera ona klasy matematyczne wykorzystywane w programie. Do większości z zaimplementowanych typów dostęp można uzyskać poprzez dołączenie pliku Math/xMath.h.

Do najważniejszych zdefiniowanych typów należą `xBYTE`, `xWORD`, `xDWORD`, `xFLOAT` oraz klasy `xVector3`, `xPoint3`, `xVector4`, `xPoint3`, `xPlane`, `xMatrix` i `xQuaternion`. Klasy te umożliwiają przeprowadzanie wszystkich niezbędnych operacji na reprezentowanych danych.

Dodatkowo biblioteka matematyczna zawiera klasy odpowiedzialne za zarządzanie kamerami, śledzenie obiektów oraz definiuje figury 3D wykorzystywane w hierarchicznej detekcji kolizji.

3.3.1. Figury 3D

Wspomnieć coś, ale szczegóły przy detekcji kolizji

3.3.2. Śledzenie obiektów

Biblioteka zapewnia klasę zarządzającą śledzeniem obiektów. Kandydaci do śledzenia muszą dziedziczyć z klasy `Math::Tracking::TrackedObject`. Dostarcza ona informacje o pozycji oraz rozmiarze sfery otaczającej śledzony obiekt. Dodatkowo daje ona możliwość uzyskania tych samych informacji dla podelementów śledzonego obiektu (np. można śledzić postać, bądź tylko jej dłoń, głowę, itp.).

Klasa `Math::Tracking::ObjectTracker` na podstawie listy dostępnych do śledzenia obiektów może ustalić punkt docelowy do którego powinien dążyć obiekt śledzący. Punkt ten może zostać użyty bezpośrednio, bądź też można zinterpolować

pośrednią pozycję obiektu śledzącego jako przesunięcie oraz opcjonalny obrót wokół wybranego punktu centralnego.

Do ustalenia punktu docelowego można zastosować jeden z wbudowanych algorytmów (śledzenie centrum wszystkich obiektów, śledzenie pojedynczego obiektu lub jego podelementu), bądź zewnętrzny skrypt specyficzny dla danego obiektu śledzącego (np. dla kamery). Docelowo zewnętrzne skrypty powinny być pisane w języku skryptowym LUA. Obecnie możliwe jest zastosowanie jedynie wbudowanych skryptów. Dostępne są skrypty śledzące dla kamer, umożliwiające ustawienie obserwatora w odpowiednim oddaleniu od obiektów, tak by kamera obejmowała je wszystkie. Ustalony punkt docelowy może być automatycznie przesunięty o zadany wektor w przestrzeni śledzonego obiektu, podobiektu lub całego świata (w zależności od specyfiki konkretnego algorytmu).

Przykładowa inicjalizacja i wykorzystanie śledzenia:

```
ObjectTracker tracker;

/* ... */
// Inicjalizacja
tracker.Init();
// Ustawienie współdzielonej listy obiektów.
tracker.Targets = &Targets;
tracker.Mode    = ObjectTracker::TRACK_CUSTOM_SCRIPT;
// Nieużywana w obecnej wersji zmienna skryptu LUA
tracker.ScriptName = "Nazwa skryptu LUA";
// Wskaźnik do wbudowanego skryptu
tracker.Script     = Wskaznik_Do_Funkcji_Typu_TrackingScript;
// Specyficzne dla skryptu dane
tracker.ScriptData = (xBYTE*) Dane_skryptu;

/* ... */
// Aktualizacja obiektu śledzącego
// Awaryjne zainicjowanie punktu docelowego
// (na wypadek błędu skryptu)
tracker.P_destination = P_position;
// Wyliczenie nowej pozycji docelowej
```

```
tracker.UpdateDestination();
// Wyliczenie pozycji pośredniej między bieżącą a docelową
tracker.InterpolatePosition(P_position,
    tracker.P_destination, min(W_weight*T_delta, 1.f));
```

3.3.3. Kamery

Wszystkie kamery dziedziczą z klasy `Math::Cameras::Camera`. Definiuje ona punkty obserwatora i celu obserwacji, a także zawiera w sobie obiekt klasy `Math::Cameras::FieldOfView`, który zapewnia implementację matematyczną rzutów perspektywicznych i prostopadłych na zadany ekran 2D. Zarówno punkt obserwatora jak i cel obserwacji może być dynamicznie modyfikowany z wykorzystaniem klasy śledzącej `Math::Tracking::ObjectTracker`. Poza oczywistym zastosowaniem kamery do ustawiania renderowanego widoku, jest ona również wykorzystywana do optymalizacji procesu renderowanie, poprzez pomijanie niewidocznych obiektów.

Biblioteka zawiera dwie implementacje klasy `Camera` – `CameraHuman` i `CameraFree`. Różnią się one jedynie reakcją na ręczne sterowanie kamerą. Kamera 'ludzka' porusza się równolegle do podłoża, gdy chcemy ją przesunąć naprzód, lub na boki. 'Wolna' kamera, porusza się jak statek kosmiczny – przy przemieszczaniu do przodu, porusza się w stronę w którą patrzy.

Kilka różnych kamer może zostać połączony w zestaw ujęć dzięki zastosowaniu klasy `Math::Cameras::CameraSet`. Klasa ta umożliwia wczytanie podziału ekranu na ujęcia z zadanego pliku konfiguracyjnego. Przykładowe pliki konfigurujące kamery znajdują się w katalogu z danymi /Data.

Przykładowe zastosowanie kamery:

```
CameraHuman Camera;

/* ... */
Camera.Init(eyex, eyey, eyez,
            centx, centy, centz,
            upx, upy, upz);
// Inicjalizacja rzutu (parametry są opcjonalne)
Camera.FOV.InitPerspective(angle, near, far);
// Inicjalizacja ekranu 2D
```

```
Camera.FOV.InitViewportPercent(0.f,0.f,1.f,1.f,
                               Width, Height);
// Opcjonalne ustawienie śledzenia
Camera.CenterTracker.Targets = &Targets;
Camera.CenterTracker.Mode    =
    Math::Tracking::ObjectTracker::TRACK_ALL_CENTER;
Camera.EyeTracker.Targets    = &Targets;
Camera.EyeTracker.Mode       =
    Math::Tracking::ObjectTracker::TRACK_CUSTOM_SCRIPT;
Camera.EyeTracker.ScriptName = "EyeSeeAll_Center";
Camera.EyeTracker.Script     =
    Math::Cameras::Camera::SCRIPT_EyeSeeAll_Center;

/* ... */
// Aktualizacja rzutu na ekran 2D, przy zmianie wymiarów okna
Camera.FOV.ResizeViewport(Width, Height);

/* ... */
// Aktualizacja pozycji (i macierzy) kamery
Camera.Update(T_delta);

/* ... */
// Ustawienie kamery podczas renderowania
ViewportSet_GL(Camera);
```

3.4. BIBLIOTEKA POMOCNICZA

3.4.1. Logowanie zdarzeń

Logowanie zdarzeń do pliku bardzo pomaga w utrwalaniu informacji o błędach oraz kolejności wystąpienia zdarzeń dla późniejszej ich analizy przez programistów. Dlatego w pliku /Utils/Debug.h znalazło się kilka prostych funkcji oraz makr umożliwiających łatwe logowanie do pliku /Data/log.txt. Warte wymienienia są następujące makra:

`LOG(level, fmt, ...)`

loguje informację jeśli jej poziom jest niższy od ustawionego w konfiguracji.

`DEB_LOG(level, fmt, ...)`

loguje informację jeśli jej poziom jest niższy od ustawionego w konfiguracji, dodatkowo wzbogacając ją o czas wystąpienia, nazwę pliku oraz numer linii.

`CheckForGLError(errorFmt, ...)`

jeśli wystąpił błąd OpenGL, zwraca `true`, ponadto loguje zadaną informację, czas, nazwę pliku oraz numer linijki w której odkryto błąd (jeśli poziom logowania wynosi co najmniej 2).

3.4.2. Dostęp do dysku

Ponieważ dostęp do informacji o systemie plików różni się między systemami Windows oraz Linux, w pliku /Utils/Filesystem.h umieszczono klasę `Filesystem`, która udostępnia uniwersalny interfejs dostępu do dysku oraz udostępnia kilka przydatnych funkcji operujących na ścieżkach:

```
class Filesystem {
public:
typedef std::vector<std::string> Vec_string;

static std::string WorkingDirectory;
```

katalog roboczy aplikacji – należy ustawić w funkcji main

```
static Vec_string GetDirectories(const string &path);
```

pobiera listę katalogów znajdujących się pod wskazaną ścieżką

```
static Vec_string GetFiles(const string &path,
                           const char *mask);
```

pobiera listę plików znajdujących się pod wskazaną ścieżką

```
static string GetSystemWorkingDirectory();
```

pobiera bieżący katalog roboczy

```
static void SetSystemWorkingDirectory(const string &path);
```

ustawia bieżący katalog roboczy

```
static string  GetParentDir(const string &path);
```

wyodrębnia z zadanej ścieżki katalog nadrzędny

```
static string  GetFullPath(const string &path);
```

zwraca pełną ścieżkę dostępu do zadanej ścieżki relatywnej

```
static string  GetFileName(const string &file);
```
wyodrębnia nazwę zadanego pliku (z rozszerzeniem)
```
static string  GetFileExt(const string &file);
```
wyodrębnia rozszerzenie zadanego pliku
```
static string  ChangeFileExt(const string &file,
                             const char *ext);
```
zmienia rozszerzenie zadanego pliku
```
};
```

3.4.3. Delegacja metod

Wśród wielu swoich zalet, język programowania C++ ma również wady. Jedną z nich jest brak wsparcia dla programowania sterowanego zdarzeniami. Jako półśrodek można stosować wskaźniki do funkcji – przekazanie do zamkniętej klasy zgodnego z jej specyfikacją wskaźnika, pozwoli wywołać z jej wnętrza nowy, zewnętrznego kodu. Rozwiązanie to rodzi jednak problemy, gdy kod obsługujący zdarzenie znajduje się wewnątrz innej klasy. Nie wystarczy w tej sytuacji, by metoda miała zgodne ze specyfikacją parametry – dla każdej klasy typ wskaźnika będzie inny. Stosując wskaźniki do metod tracimy na uniwersalności i rozszerzalności kodu.

Na potrzeby projektu stworzono prosty szablon delegacji do funkcji, który stara się ominąć te niedogodności. W celu obejścia problemu wskaźników do metod, zastosowano prostą sztuczkę z globalną funkcją pośredniczącą między wywołaniem, a obsługą zdarzenia. Funkcja ta, poza parametrami oraz obiektem w którym zaszło zdarzenie, otrzymuje również nietypowany wskaźnik do klasy docelowej. Jej kod jest bardzo prosty – rzutuje wskaźnik na prawidłowy typ oraz wywołuje odpowiednią metodę docelowej klasy. Kod szablonu można znaleźć w pliku /Utils/Delegate.h, zaś jego przykładowe zastosowanie wyglądać może następująco:

```
// Kod zamkniętej, uniwersalnej klasy
class Window {
public:
  // Deklaracja zdarzenia
  typedef Delegate<Window, unsigned int /*Width*/,
                   unsigned int /*Height*/> WindowResizeEvent;
  WindowResizeEvent OnResize;
```

```
  Window() {
    // Inicjalizacja pustego zdarzenia,
    // parametr to klasa wołająca zdarzenie
    OnResize = WindowResizeEvent(*this);
  }

  Resize(Width, Height) {
    /* ... */
    // Wywołanie zdarzenia
    OnResize(Width, Height);
  }
};

// Kod nowej klasy
void AnyClass_OnResize(Window &window, void* receiver,
                unsigned int &width, unsigned int &height);

class AnyClass {
public:
  Window MainWindow;

  AnyClass() {
    // Dowiązanie funkcji do zdarzenia,
    // parametry to klasa docelowa oraz funkcja pośrednicząca
    MainWindow.OnResize.Set(*this, ::AnyClass_OnResize);
  }

  // Kod obsługi zdarzenia
  void OnResize( Window &window,
                 unsigned int &width, unsigned int &height );
};

void AnyClass_OnResize(Window &window, void* receiver,
                unsigned int &width, unsigned int &height)
{
  // Prosta sztuczka z wywołaniem interesującej nas metody
```

```
    ((AnyClass*)receiver)->OnResize(window, width, height);
}
```

3.4.4. Singletony

Niektóre elementy składowe gry powinny występować zawsze tylko w jednej instancji – może być tylko jeden obiekt aplikacji, wystarczy też jeden menadżer tekstur, wspólny dla wszystkich renderowanych elementów. Idealnie by było, gdyby taki unikalny element gry był łatwo, czytelnie i globalnie dostępny z dowolnego punktu w kodzie. Taką powszechną dostępność zapewnić mogą zmienne globalne, jednakże nie zabezpieczą one programisty przed przypadkowym stworzeniem kilku instancji obiektu, ponadto programista nie ma wpływu na kolejność tworzenia i niszczenia globalnych obiektów.

Wszystkie te postulaty spełnia natomiast szablon modelu Singleton przedstawiony przez Scotta Bilasa w książce [PPG_T1]. Obiekty dziedziczące z tego szablonu są tworzone i niszczone na życzenie programisty, zaś zastosowanie makra `assert` w konstruktorze zabezpiecza nas przed próbą utworzenia wielu instancji jednej klasy:

```
template <typename T> class Singleton {
    static T* ms_Singleton;
protected:
    Singleton()
    {
        assert( !ms_Singleton );
        int offset = (int)(T*)1 - (int)(Singleton <T>*)(T*)1;
        ms_Singleton = (T*)((int)this + offset);
    }
    ~Singleton()
    {  assert( ms_Singleton );  ms_Singleton = 0;  }

public:
    static T& GetSingleton()
    {  assert( ms_Singleton );  return ( *ms_Singleton );  }
    static T* GetSingletonPtr()
    {  return ( ms_Singleton );  }
```

```
    static void CreateS()
       { new T(); }
    static void DestroyS()
       { assert( ms_Singleton ); delete ms_Singleton; }
};
template <typename T> T* Singleton <T>::ms_Singleton = 0;
```

Przykładowa implementacja, definicja prostego makra umożliwia nam łatwy dostęp do obiektu, z dowolnego miejsca w kodzie:

```
class TextureMgr : public Singleton <TextureMgr> {
public:
    HTexture GetTexture( const char* name );
    // ...
};
#define g_TextureMgr TextureMgr::GetSingleton()

void SomeFunction( void )
{
    HTexture stone1 =
       TextureMgr::GetSingleton().GetTexture( "stone1" );
    HTexture wood6 = g_TextureMgr.GetTexture( "wood6" );
    // ...
}
```

3.4.5. Menadżer uchwytów

Gry komputerowe operują na dużych ilościach danych różnego typu – teksturach, czcionkach, dźwiękach, animacjach, itd. Aby zapewnić przejrzysty, bezpieczny oraz szybki dostęp do tych danych, należy zaimplementować wyspecjalizowane bazy danych. W niniejszym rozwiązaniu bazują one na menadżerze uchwytów `HandleMgr` oraz szablonie specyficznego menadżera `Manager` dostępnych po dołączeniu pliku /Utils/Manager.h. Kolekcja `HandleMgr` oraz jej element `Handle` bazują na rozwiązaniu zaprezentowanym przez Scotta Bilasa w książce `[PPG_T1]`.

Rozwiązanie Scotta Bilasa zostało rozszerzone o prosty mechanizm zliczania referencji (dokładniej zliczania ilości pobrań danego uchwytu od menadżera).

Umożliwia on wielokrotnie wykorzystywanie jednego zasobu oraz jego automatyczne zwolnienie dopiero wtedy, gdy wszystkie odwołania zostaną usunięte.

Zaimplementowany menadżer uchwytów umożliwia inwalidację przechowywanych zasobów bez unieważniania uchwytów które na nie wskazują. Podsystemy operujące na zasobach nie muszą być świadome operacji walidacji danych. Dane które tracą ważność (np. identyfikator tekstury Open GL), zostaną automatycznie odtworzone przy kolejnym odwołaniu do zasobu.

```
template <typename RESOURCE, class HANDLE>
class Manager {
public:
void InvalidateItems();
```

oznacza zasoby jako błędne, by mogły zostać automatycznie wczytane przy kolejnym odwołaniu za pomocą istniejącego uchwytu

```
bool IsHandleValid( HANDLE hnd );
```

sprawdza czy podany uchwyt jest poprawny

```
void Release( HANDLE hnd );
```

usuwa jedną referencję do zasobu wskazywanego przez podany uchwyt

```
};
```

Przykładowa implementacja zasobu:

```
struct Texture : public Resource {
// Metody wirtualne które należy zdefiniować
virtual const string &Identifier();
```

zwraca unikalną nazwę zasobu (np. nazwę pliku z którego pochodzi)

```
virtual void Clear();
```

resetuje zmienne zasobu, bez przeprowadzania dodatkowych operacji (zwalniania pamięci, itp.)

```
virtual bool Create();
```

tworzy zasób na podstawie już zdefiniowanych wartości pól (name i fl_mipmap)

```
virtual bool Create( const string& name, bool fl_mipmap )
```

definiuje wartości pól, a następnie tworzy zasób

```
virtual void Dispose();
```
zwalnia i resetuje dane zasobu, z możliwością automatycznego ich odtworzenia przy kolejnym odwołaniu

```
virtual void Invalidate()
```
resetuje dane zasobu, z możliwością automatycznego ich odtworzenia przy kolejnym odwołaniu

```
virtual bool IsDisposed()
```
sprawdza czy dane są zwolnione i wymagają odtworzenia

```
// Domyślna implementacja poniższych metod powinna wystarczyć
virtual void Destroy()
```
zwalnia a następnie resetuje dane zasobu, bez możliwości ich odtworzenia (odwołania stają się nieważne) – domyślna implementacja woła `Dispose()`, a następnie `Clear()`

```
virtual bool Recreate()
```
odtwarza zasób (potencjalnie zwolniony) – domyślna implementacja woła `Dispose()`, a następnie `Create()`

```
// Odziedziczone metody niewirtualne
void Lock();
```
zwiększa liczność referencji

```
void Release();
```
zmniejsza liczność referencji

```
bool IsLocked();
```
sprawdza czy istnieją jakieś referencje do zasobu

```
unsigned int CountReferences()
```
zwraca ilość referencji do zasobu

```
}
```

Przykładowa implementacja menadżera zasobów:

```
class TextureMgr;
typedef Handle <TextureMgr> HTexture;
class TextureMgr : public Manager<Texture, Htexture> {
public:
```

```
HTexture GetTexture( const char* name );
```

zwraca uchwyt do podanej tekstury, jeżeli tekstura jeszcze nie istnieje, tworzony jest nowy uchwyt

```
void BindTexture( HTexture htex );
```

informuje silnik graficzny o chęci wykorzystania danej tekstury podczas kolejnych etapów renderowania

```
};
```

3.5. BIBLIOTEKI MODELI

3.5.1. Biblioteka Lib3ds

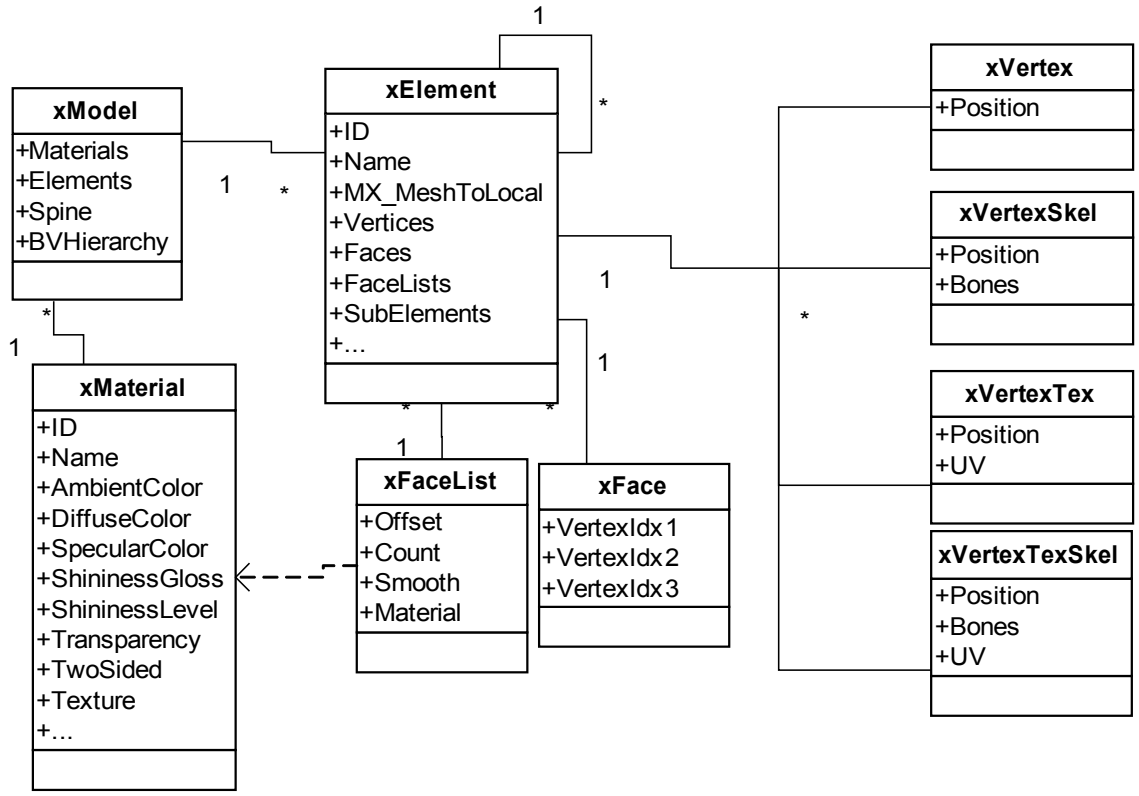
Modele 3D wykorzystywane w grze zostały utworzone w programach SketchUp, 3d Studio Max oraz Blender. Do przenoszenia modeli między programami wybrałem szeroko rozpowszechniony format 3ds. Gotowe modele są importowane do natywnego dla gry formatu 3dx z wykorzystaniem darmowej biblioteki [LIB3DS]. Kod źródłowy biblioteki został załączony do projektu VisualStudio w katalogu /Models/lib3ds/, więc nie trzeba jej instalować w systemie.

3.5.2. Biblioteka Lib3dx

Na potrzeby gry stworzono format plików o rozszerzeniu 3dx. Format ten bazuje na typach zdefiniowanych w bibliotece matematycznej. Do głównych cech formatu należą:

- obsługa modeli statycznych oraz animowanych szkieletowo
- do 255 elementów na model
- do 255 materiałów na model
- do 1 tekstury na materiał (w planach rozszerzenie do 255 tekstur oraz indywidualnego shadera)
- do 65535 wierzchołków wchodzących w skład do 65535 trójkątów na element
- do 32 grup wygładzania na pojedynczy element oraz pojedynczy trójkąt
- do 1 mapowania UV na element (w planach rozszerzenie do 255 mapowań)
- do 4 kości na wierzchołek

- do 255 kości na model (lecz ograniczenie pamięci programu GPU narzuca limit renderowania 32 kości)



Rysunek 1: Drzewo zależności w bibliotece lib3dx

Opis typów wchodzące w skład biblioteki lib3dx

Bibliotekę lib3dx można znaleźć w katalogu /Models/lib3dx/. Jest ona zbyt duża, by szczegółowo opisywać w tej pracy każdy jej element. Śledząc pliki nagłówkowe podczas czytania skróconych charakterystyk znajdujących się poniżej, czytelnik powinien bez problemu domyślić się znaczenia poszczególnych zmiennych i metod.

Animacja szkieletowa:

`xAction` /Models/lib3dx/xAction.h

prosta klasa przechowująca uchwyt do animacji szkieletowej oraz czas jej rozpoczęcia i zakończenia

`xActionSet` /Models/lib3dx/xAction.h

zestaw akcji `xAction`, monitoruje postęp animacji i zwraca układ kości będący mieszanką wszystkich aktywnych animacji

`xAnimation` /Models/lib3dx/xAnimation.h

animacja szkieletowa – zestaw klatek `xKeyFrame`, monitoruje postęp animacji i zwraca układ kości będący interpolacją bieżącej i kolejnej klatki

`xAnimationH` /Models/lib3dx/xAnimationMgr.h

klasa pochodna klasy `xAnimation`, wykorzystywana przez menadżera animacji

`xAnimationInfo` /Models/lib3dx/xAnimation.h

informaja o przebiegu animacji szkieletowej, przechowuje numer klatki oraz postęp i czas trwania całej animacji

`xAnimationMgr` /Models/lib3dx/xAnimationMgr.h

menadżer animacji – obiekt typu `Singleton`

`xKeyFrame` /Models/lib3dx/xAnimation.h

kluczowa klatka animacji szkieletowej, przechowuje układ kości oraz czas wstrzymania i trwania klatki

Model 3D:

`xBone` /Models/lib3dx/xBone.h

kość szkieletu, przechowuje identyfikatory rodzica oraz swoich dzieci, a także punkty początku i końca kości (dla szkieletu w stanie spoczynku). Podczas operacji na szkielecie, przechowuje obrót kości względem rodzica dla obecnie przetwarzanej instancji modelu.

`xBoundingData` /Models/lib3dx/xElementData.h

baza hierarchii przejściowej używanej przez generator hierarchii brył otaczających typu `xBVHierarchy`

`xBoundingHierarchy` /Models/lib3dx/xElementData.h

pojedyńczy węzeł hierarchi `xBoundingData`

`xEdge` /Models/lib3dx/xElementData.h

potencjalna krawędź rzucanego przez model cienia, zawiera indeksy wierzchołków oraz trójkątów

`xElement` /Models/lib3dx/xElement.h

klasa przedstawiająca pojedynczą siatkę 3D modelu, przechowuje wierzchołki, trójkąty, informacje o wygładzaniu, a także pomocnicze informacje, takie jak

potencjalne krawędzie cieni, hierarchiczny podział siatki `xBoundingData` oraz informacje silnika renderującego `xRenderData`

`xFace` /Models/lib3dx/xElementData.h

trójkąt siatki 3D – przechowuje indeksy wierzchołków

`xFaceList` /Models/lib3dx/xElementData.h

przechowuje zakres trójkątów siatki 3D współdzielących materiał – optymalizuje proces renderowania

`xMaterial` /Models/lib3dx/xMaterial.h

zawiera informacje o materiale pokrywającym trójkąty

`xModel` /Models/lib3dx/xModel.h

klasa zawierająca współdzielone informacje o konkretnym modelu 3D – listę elementów `xElement`, materiałów `xMaterial`, szkielet `xSkeleton` oraz ręcznie stworzoną hierarchię brył otaczających `xBVHierarchy`

`xSkeleton` /Models/lib3dx/xSkeleton.h

szkielet modelu – zawiera zestaw kości `xBone` oraz ograniczeń silnika fizycznego Verleta `VConstraint` (więcej informacji w rozdziale poświęconym fizyce szkieletu)

`xTexture` /Models/lib3dx/xMaterial.h

nazwa oraz uchwyt tekstury

`xVertex`
`xVertexSkel`
`xVertexTex`
`xVertexTexSkel` /Models/lib3dx/xVertex.h

struktury opisujące pojedyńczy wierzchołek – zawierają pozycję oraz w zależności od przyrostka: `Skel` – dane o czterech powiązanych kościach, `Tex` – dane o mapowaniu UV

Wsparcie renderingu oraz indywidualne dane konkretnych klonów modelu:

`xElementInstance` /Models/lib3dx/xElementRunTime.h

informacje specyficzne dla konkretnego klonu modelu – przechowuje indywidualne wskaźniki `xGPUPointers` (wykorzystywane tylko przez animowane modele renderowane w trybie list OpenGL), informacje o rzucanych przez element cieniach, bryły otaczające dany element (wykorzystywanych do pomijania renderingu niewidocznych obiektów) oraz wierzchołki elementu poddane animacji (potrzebne

do wyliczenia brył otaczających oraz przez rendering animacji szkieletowej przebiegającej po stronie CPU)

`xGPUPointers` /Models/lib3dx/xElementRunTime.h

przechowuje wskaźniki do danych GPU – identyfikatory list OpenGL, buforów VBO itp.

`xGPUShadowPointers` /Models/lib3dx/xElementRunTime.h

przechowuje wskaźniki do danych GPU związanych z rzucaniem cieni objętościowych

`xModelInstance` /Models/lib3dx/xModel.h

informacje specyficzne dla konkretnego klonu modelu – położenie w przestrzeni, przeliczone transformacje kości oraz zestaw obiektów `xElementInstance`

`xRenderData` /Models/lib3dx/xElementRunTime.h

informacje silnika renderującego wspólne dla wszystkich modeli danego typu – przechowuje wygładzone trójkąty oraz współdzielone wskaźniki `xGPUPointers`

`xShadowData` /Models/lib3dx/xElementRunTime.h

informacje o cieniach objętościowych – zawiera identyfikator światła powiązanego z cieniem, wyliczone wierzchołki i trójkąty cienia oraz wskaźniki danych `xGPUPointers` i `xGPUShadowPointers`

Klasy dodatkowe, bezpośrednio wspierające bibliotekę:

`Model3dx` /Models/Model3dx.h

opakowanie modelu xModel wykorzystywane przez menadżer modeli `ModelMgr`

`ModelMgr` /Models/ModelMgr.h

menadżer modeli xModel

3.6. SZKIELET APLIKACJI

3.6.1. Aplikacja

Większości współczesnych programów wchodzących w interakcję z użytkownikiem tworzonych jest w oparciu o modelu programowania sterowanego zdarzeniami – kolejność wykonywania kodu uzależniona jest od zdarzeń wywoływanych przez system oraz użytkownika. Gry komputerowe wiążą się z tworzeniem systemu czasu rzeczywistego – symulacja świata podzielona jest na kolejne

klatki, w których zdarzenia następują w określonej kolejności. Przebieg gry komputerowej można podzielić na następujące kroki:

(1) Utworzenie aplikacji

(2) Uruchomienie pętli krokowej

(2.1) Początek klatki

(2.2) Aktualizacja stanu świata

(2.3) Przedstawienie obecnego stanu świata na ekranie

(2.4) Zakończenie klatki

(2.5) Rozpoczęcie kolejnej klatki (powrót do 2.1) bądź przejście do kroku 3

(3) Zakończenie aplikacji

Na potrzeby tej pracy utworzono uniwersalną klasę `Application`, która wspiera przedstawiony powyżej model. Posiada ona pojedyncze okno klasy `IWindow`, wspierające określone API graficzne (OpenGL bądź DirectX). Sam obiekt aplikacji nie definiuje zachowania programu – do poprawnego funkcjonowania wymaga obiektu potomnego klasy `IScene`.

```
#define g_Application Application::GetSingleton()
class Application : public Singleton<Application> {
public:
enum AppResult {
    SUCCESS      = 0,
    WINDOW_ERROR = 1,
    EVENT_ERROR  = 2,
    SCENE_ERROR  = 4
};
```

możliwe rezultaty metod `Create(...)` oraz `Invalidate()` - flagi

```
void Clear();
```

zeruje stan klasy

```
int  Create(const char* title, unsigned int width,
            unsigned int height, bool fl_fullscreen,
            IScene &scene);
```

tworzy aplikację, okno OpenGL o podanych parametrach oraz podaną scenę. Scena zostanie automatycznie zniszczona, a jej pamięć zwolniona podczas niszczenia aplikacji!

```
int  Create(IWindow &window, IScene &scene);
```

tworzy aplikację, zainicjalizowane uprzednio okno (`IWindow::PreCreate()`) oraz podaną scenę. Scena i okno zostaną automatycznie zniszczone, a ich pamięć zwolniona podczas niszczenia aplikacji!

```
void Destroy();
```

niszczy aplikację, okno oraz scenę

```
int  Invalidate();
```

inwaliduje scenę (zasoby straciły ważność, np. w wyniku zamknięcia okna OpenGL)

```
int  Run();
```

uruchamia główną pętlę aplikacji – przetwarza kolejkę komunikatów metodą `IWindow::ProcessMessages()`, woła metody `IScene::FrameStart()`, `Update()`, `Render()` oraz `IScene::FrameEnd()`

```
bool Update(float T_delta);
```

jeśli okno aplikacji jest aktywne, woła metodę `IScene::Update()`

```
bool Render();
```

jeśli okno aplikacji jest aktywne, woła metodę `IScene::Render()` oraz `IWindow::SwapBuffers()`

```
IWindow& MainWindow_Get();
```

zwraca główne okno aplikacji

```
IScene& Scene_Get();
```

zwraca bieżącą scenę aplikacji

```
bool Scene_Set(IScene& scene,
               bool fl_destroyPrevious = true);
```

zmienia scenę aplikacji metodą `IScene::Scene_Set()`, parametr wskazuje czy poprzednia scena ma zostać zwolniona, czy też przypisana do zmiennej `IScene::PrevScene`

```
typedef Delegate<Application, bool> ApplicationEvent;
ApplicationEvent OnApplicationCreate;
```
zdarzenie wołane między tworzeniem okna, a tworzeniem sceny
```
ApplicationEvent OnApplicationInvalidate;
```
zdarzenie wołane przed inwalidacją sceny
```
ApplicationEvent OnApplicationDestroy;
```
zdarzenie wołane po zniszczeniu okna oraz sceny
```
};
```

3.6.2. Okno

Współczesne gry i programy w większości porzuciły komunikację poprzez konsolę na rzecz graficznego przedstawiania informacji w oknach. Klasa `IWindow` udostępnia uniwersalny interfejs okna, który następnie klasa potomna implementuje dla konkretnego API systemowego (WinAPI, X11) i graficznego (OpenGL, DirectX). Projekt wykorzystuje klasę `GLWindow` która implementuje API OpenGL poprzez WinAPI bądź X11.

```
class IWindow {
public:
typedef Delegate<IWindow> WindowCreateEvent;
typedef Delegate<IWindow, unsigned int /*Width*/,
                 unsigned int /*Height*/> WindowResizeEvent;

WindowCreateEvent OnCreate;
```
zdarzenie wołane po pomyślnym utworzeniu okna
```
WindowResizeEvent OnResize;
```
zdarzenie wołane przy zmianie rozmiaru okna (również zaraz po utworzeniu okna)

```
virtual void Clear();
```
zeruje stan okna, bez zwalniania pamięci

```
::HDC  HDC();
```
zwraca unikalny identyfikator okna – `HDC` dla WinAPI, `Display *` dla X11

```
virtual void PreCreate(const char *title, unsigned int width,
                       unsigned int height, bool fl_fullscreen);
```

inicjalizuje okno, tak by można je było utworzyć wywołaniem `Create()`

```
virtual bool Create() = 0;
```

tworzy okno zainicjalizowane `PreCreate()`

```
bool Create(const char *title, unsigned int width,
                       unsigned int height, bool fl_fullscreen);
```

inicjuje okno metodą `PreCreate()`, a następnie tworzy je metodą `Create()`

```
virtual void Dispose() = 0;
```

niszczy okno, lecz nie zeruje ustawień, by można je było odtworzyć poprzez `Create()`

```
virtual void Destroy();
```

niszczy i zeruje okno (`Dispose()` + `Clear()`)

```
bool IsDisposed();
```

sprawdza czy okno zostało zwolnione

```
bool IsDestroyed();
```

sprawdza czy okno zostało zwolnione i nie może być odtworzone bez ponownej inicjalizacji `PreCreate()`

```
virtual bool ProcessMessages() = 0;
```

przetwarza kolejkę komunikatów systemowych

```
bool  IsFullScreen();
```

sprawdza czy okno jest pełnoekranowe

```
bool  FullScreen_Set(unsigned int width, unsigned int height,
                     bool fl_fullscreen);
```

ustawia rozmiar i typ okna (zwykłe / pełnoekranowe)

```
bool  IsActive();
```

sprawdza czy okno jest uaktywnione

```
void  Active_Set(bool isActive);
```

ustawia stan aktywności okna

```
int   Height_Get();
```

podaje wysokość okna

```
int   Width_Get();
```
podaje szerokość okna

```
virtual void SwapBuffers() = 0;
```
zamienia bufory okna, gdy okno jest buforowane

```
};
```

3.6.3. Sceny

Aby podzielić aplikację na niezależne logicznie tryby pracy oraz uniezależnić klasę `Application` od logiki, stworzona została klasa `IScene`. Pojedyncza scena przedstawia zamknięty logicznie, samowystarczalny fragment logiki aplikacji. Z innymi scenami powinien ją łączyć jedynie kod odpowiedzialny za przejście do kolejnej sceny. W typowej grze komputerami jako niezależne sceny można by potraktować menu, rozgrywkę, przerywnik filmowy bądź wbudowany edytor. Klasa `IScene` udostępnia następujący interfejs:

```
class IScene {
public:
const char  * Name;
```
unikalna nazwa sceny – wykorzystywana do wybrania odpowiedniego mapowania klawiszy przez klasę `InputMgr`

```
IScene * PrevScene;
```
poprzednia scena, jeśli nie została zniszczona

```
virtual bool Create(int left, int top,
                    unsigned int width, unsigned int height,
                    IScene *prevScene = NULL);
```
tworzy scenę o podanych wymiarach, jeśli nie zdefiniowano parametru `prevScene`, to wartość pola `PrevScene` nie ulegnie zmianie (można ustawić `PrevScene` przed wywołaniem `Create()`)

```
virtual void Destroy();
```
niszczy tę oraz poprzednie sceny poprzednie

```
bool IsDestroyed();
```
sprawdza czy scena została zniszczona

```
virtual void Enter();
```
informuje scenę, że stała się aktywna w wyniku przejścia z innej sceny bądź uruchomienia aplikacji

```
virtual void Exit();
```
informuje scenę, że następuje przejście do innej sceny, bądź zamknięcie aplikacji

```
virtual bool Invalidate();
```
inwaliduje scenę (zasoby straciły ważność, np. w wyniku zamknięcia okna OpenGL)

```
virtual void Resize(int left, int top,
                    unsigned int width, unsigned int height);
```
zmienia wymiary sceny (scena może zajmować tylko podany fragment okna)

```
virtual void FrameStart();
```
początek klatki

```
virtual bool Update(float T_delta) = 0;
```
aktualizacja stanu sceny

```
virtual bool Render() = 0;
```
wyświetlenie sceny na ekranie

```
virtual void FrameEnd();
```
zakończenie klatki

```
virtual bool ShellCommand(string &cmd, string &output);
```
przetwarza polecenie tekstowe `cmd`, zapisuje rezultat do ciągu `output`

```
virtual IScene &Scene_Set(IScene& scene,
                          bool fl_destroyPrevious = true);
```
dokonuje przejścia między scenami, jeśli nie niszczy bieżącej sceny, to zapisuje ją w polu `PrevScene` nowej sceny. Zwraca bieżącą scenę

```
};
```

3.6.4. Obsługa zdarzeń klawiatury i myszki

Gry różnią się od typowych programów interpretacją zdarzeń związanych z klawiaturą. Przeciętny program reaguje na moment wciśnięcia klawisza (np. w edytorze tekstu, pojawi się nowa litera), lub jego zwolnienia. W grach uwaga bardziej skupia się na stanie klawisza (wciśnięty lub nie), niż momencie jego zmiany. Gdy jedziemy

wirtualnym samochodem, to przytrzymany klawisz odpowiedzialny za przyspieszenie, powoduje jednostajny wzrost prędkości auta. Pojawia się w tym miejscu druga różnica między grami, a resztą programów. Na przeciętnej klawiaturze próżno szukać klawisza przyspieszenia. Gra musi więc mapować standardowe klawisze, na ich docelowe znaczenie.

Wsparciu przejścia z modelu zdarzeniowego, wspieranego przez środowiska graficzne, na model stosowany w grach, a także tłumaczeniu znaczenia klawiszy z dosłownego na umowny, służy klasa `InputMgr`. Umożliwia ona zdefiniowanie dla każdej sceny gry indywidualnego mapowania z systemowych kodów klawiszy (Virtual Key Codes) na kody stosowane wewnątrz aplikacji. Zwiększa to czytelność oraz elastyczność kodu. Widząc polecenie GetInputState(IC_Accelerate) dużo łatwiej się domyśleć celu tego zapytania, niż w przypadku polecenia GetKeyState(VK_UP). Ponadto wystarczy zmienić mapowanie, by przedefiniować znaczenie klawiszy – można robić to dynamicznie, w trakcie działania programu, bez wprowadzania zmian w kodzie.

```
typedef unsigned char byte;

class InputMgr {
public:
string Buffer;
```

bufor wykorzystywany do pobierania tekstu z klawiatury

```
int  mouseX;
```

pozioma pozycja myszki (liczona od lewej krawędzi)

```
int  mouseY;
```

pionowa pozycja myszki (liczona od górnej krawędzi)

```
int  mouseWheel;
```

obroty kółka myszki

```
bool FL_enable;
```

flaga aktywności klasy – nieaktywna klasa zwraca false dla wszystkich kodów funkcjonalnych (stan kodów klawiszy zwracany jest normalnie)

```
void Create(int iCodeCount);
```

inicjalizuje menadżera, rezerwując miejsce dla podanego zakresu kodów funkcjonalnych

```
void Destroy();
```
zwalnia pamięć wykorzystywaną przez mapowania i tablicę stanów kodów funkcjonalnych

```
void ClearMappings();
```
usuwa mapowania dla wszystkich scen

```
void Clear();
```
zeruje stan menadżera, bez zwalniania pamięci

```
void LoadMap(const char *fileName);
```
wczytuje mapowania scen z podanego pliku

```
void SaveMap(const char *fileName);
```
zapisuje mapowania scen do podanego pliku

```
void AppendBuffer(byte charCode);
```
dodaje podany znak do bufora `Buffer`, klawisze Enter i Escape są pomijane

```
void KeyDown_Set(byte kCode, bool down);
```
ustawia stan klawisza (i odpowiadającego mu kodu funkcyjnego)

```
bool KeyDown_Get(byte kCode);
```
pobiera stan klawisza

```
void InputDown_Set(int iCode, bool down);
```
ustawia stan kodu funkcyjnego

```
bool InputDown_Get(int iCode);
```
pobiera stan kodu funkcyjnego

```
bool InputDown_GetAndRaise(int iCode);
```
pobiera stan kodu funkcyjnego, a następnie ustawia go na `false` (podniesiony)

```
void SetScene(const char *scene,
              bool FL_dont_process_buttons = true);
```
ustawia mapowanie danej sceny jako bieżące, domyślnie zeruje ten stan i nie inicjuje go na podstawie bieżącego stanu klawiszy

```
void AllKeysUp();
```
ustawia stan wszystkich klawiszy jako `false` (podniesiony)

```
int Key2InputCode(byte kCode);
```
zwraca kod funkcjonalny odpowiadający danemu kodowi klawisza

```
byte Input2KeyCode(int iCode);
```
zwraca kod pierwszego klawisza powiązanego z danym kodem funkcjonalnym

```
void Key2InputCode_Set(byte kCode, int iCode);
```
ustawia mapowanie danego klawisza na daną funkcję (dla bieżącej sceny)

```
void Key2InputCode_SetIfKeyFree(byte kCode, int iCode);
```
ustawia mapowanie danego klawisza na daną funkcję (dla bieżącej sceny), tylko jeśli klawisz nie ma jeszcze zdefiniowanego mapowania

```
string GetKeyName(int kCode);
```
pobiera nazwę klawisza

```
void LoadKeyCodeMap(const char *fileName);
```
wczytuje tablicę nazw klawiszy

```
};
```

3.7. SCENY

Jak wspomniałem przy opisie szkieletu aplikacji, poszczególne etapy działania programu mogą zostać podzielone na niezależne sceny. Na potrzeby gry Camera Fighter stworzono cztery sceny – menu, rozgrywkę, edytor modeli oraz konsolę. W planach mam również stworzenie wizualnego edytora map.

3.7.1. Menu

Klasa `Scenes::SceneMenu` jest pierwszą sceną z jaką ma kontakt gracz. Składa się ono z szeregu ekranów przez które musi przejść użytkownik by rozpocząć rozgrywkę, uruchomić edytor modeli, bądź przeczytać informację o autorach gry. Ponieważ każdy ekran zachowuje się nieco odmiennie, funkcjonalność Menu została rozbita na pomniejsze stany dziedziczące z klasy `Scenes::Menu::BaseState`. Klasa ta reprezentuje pojedynczy ekran menu. Zawiera listę kolejnych, bezpośrednio po nim następujących stanów, oraz odnośnik do stanu rodzica, który nas do niego doprowadził. Statyczne metody klasy `Scenes::Menu::BaseState` umożliwiają przemieszczanie się między stanami. Hierarchia ekranów wygląda następująco:

- `MainState` - stan początkowy, menu główne
  - `SelectMapState` - ekran wyboru mapy – listę map można zmienić w pliku /Data/maps.txt
    - `PlayState` – ekran wyboru graczy oraz stylu walki, umożliwia przejście do sceny rozgrywki – lista graczy (i możliwych dla nich stylów walki) może być zmieniona w pliku /Data/players.txt
  - `EditModelState` – ekran pozwala wybrać pliki z modelami i przejść do sceny edytora modeli. Ponieważ obiekt może się składać jednocześnie z szczegółowej siatki graficznej oraz uproszczonej siatki fizycznej, wybór modelu następuje dopiero po zaznaczeniu dwóch plików (lub dwa razy jednego)
  - `EditMapState` – ekran nieaktywny, w planach ma umożliwiać wybór mapy do edycji i przejście do sceny edytora map.
  - `OptionsState` – ekran nieaktywny, obecnie opcje można zmieniać w pliku /Data/config.txt
  - `CreditsState` – ekran wyświetlający informacje o twórcach gry
  - `ExitState` – przejście do tego stanu skutkuje zakończeniem gry

3.7.2. Rozgrywka

Świat

Akcja gry Camera Fighter rozgrywa się w zamkniętych pomieszczeniach – na ringach, bądź tradycyjnych miejscach treningów wschodnich sztuk walki – dojo. Z racji przebywania w takiej scenerii, przyjęto zasadę, że wszystkie elementy świata to modele 3D. Nie ma możliwości wygenerowania siatki z wykorzystaniem map wysokości i tym podobnych technik.

W grze stosuje się dwa typy modeli – modele statyczne `RigidObj` oraz wywodzące się z nich modele animowane szkieletowo `SkeletizedObj`. Dodatkowo każdy model może być stacjonarny (elementy budynku) bądź ruchomy. Szczegóły implementacji tych klas zostaną omówione w dziale poświęconym fizyce.

Wszystkie modele wchodzące w skład mapy przechowywane są przez instancję klasy `World`. Poza fizycznymi modelami, zawiera ona również model przedstawiający niebo, a także listę świateł. Wszystkie te informacje mogą zostać wczytane z prostych plików tekstowych. Przykładowe mapy można znaleźć w plikach /Data/models/*.map.

Rozgrywka może być obserwowana z dowolnej ilości ujęć, zdefiniowanych w pliku /Data/cameras.txt. Domyślnie akcję widzimy od boku, z góry oraz z oczu pierwszego gracza. W plikach /Data/cameras_*.txt znajdują się przykłady innych ujęć. Pierwsza zdefiniowana kamera może być przemieszczana za pomocą klawiatury, domyślne sterowanie to:

- Numpad8 – obrót do góry wokół obserwatora
- Numpad5 – obrót w dół wokół obserwatora
- Numpad4 – obrót w lewo wokół obserwatora
- Numpad6 – obrót w prawo wokół obserwatora
- Y – przechył w lewo
- I – przechył w prawo
- U – obrót w górę wokół centrum obserwacji
- J – obrót w dół wokół centrum obserwacji
- H – obrót w lewo wokół centrum obserwacji
- K – obrót w prawo wokół centrum obserwacji
- Home – przesunięcie kamery do przodu
- End – przesunięcie kamery w tył
- Delete – przesunięcie kamery w lewo
- Page Down – przesunięcie kamery w prawo
- Page Up – przesunięcie kamery do góry
- Insert – przesunięcie kamery w dół
- Lewy Shift – przyspieszenie ruchu kamery

Logika

Gra umożliwia dwa sposoby sterowania

3.7.3. Konsola

3.7.4. Edytor modeli

Wstęp

Tworzenie szkieletu

Skórowanie modelu

Animacja szkieletowa

4. SILNIK GRAFICZNY

4.1. WSTĘP

4.2. MENADŻER TEKSTUR

4.3. MENADŻER CZCIONEK

4.4. ROZSZERZENIA OPENGL

4.5. RENDEROWANIE MODELI 3DX

4.5.1. Wstęp

4.5.2. Renderowanie jednoprzebiegowe

4.5.3. Renderowanie do bufora głębi

4.5.4. Renderowanie oświetlenia otoczenia

4.5.5. Renderowanie oświetlenia rozproszonego i odblaskowego

4.6. RENDEROWANIE SCENY 3D

4.6.1. Renderowanie jednoprzebiegowe

4.6.2. Renderowanie wieloprzebiegowe

4.6.3. Renderowanie odroczone

4.7. PROGRAMOWANIE POD GPU

4.7.1. GLSlang

4.7.2. HLSL

5. SILNIK FIZYCZNY

5.1. DETEKCJA KOLIZJI

5.1.1. Wstęp

Detekcja kolizji jest niezbędnym elementem każdego symulatora fizyki. Nie mając informacji o zachodzących zderzeniach, symulator nie mógłby na nie zareagować. Obiekty poruszałyby się w 'świecie duchów', przenikając się nawzajem. Ruch byłby jedynie skutkiem zaszytych w silniku sił, takich jak np. siła ciążenia. Obserwowanie takiego świata nie było by ani pouczającym, ani zajmującym widowiskiem.

W wyniku detekcji kolizji silnik fizyczny otrzymuje informacje które umożliwiają symulację oddziaływań między ciałami. Dzięki tej wiedzy możemy przenieść się ze 'świata duchów' do świata materialnego. Detektory najczęściej wykrywają kolizje na jeden z dwóch sposobów: metodą a priori bądź a posteriori `[Wiki_CD]`.

W metodzie a posteriori silnik porusza symulację o pewien mały krok do przodu, a następnie sprawdza, czy istnieją obiekty które przecinają się. W każdym kroku symulacji tworzona jest lista przecinających się obiektów. Na bazie tej listy poprawiane są pozycje i trajektorie ruchu kolidujących obiektów. Nazwa metody bierze się stąd, iż najczęściej symulator przeskoczy moment zaistnienia kolizji i dopiero po jej zaistnieniu podejmuje akcję.

Odmiennym podejściem cechuje się metoda a priori. W metodzie tej algorytm detekcji kolizji musi potrafić przewidzieć dokładne trajektorie obiektów. Wykorzystując tę wiedzę, może precyzyjnie określić moment zderzenia, dzięki czemu ciała nigdy się nie przetną. Momenty kolizji znane są jeszcze przed przemieszczeniem ciał. Niestety kosztem tej wiedzy jest duże skomplikowanie numeryczne algorytmu.

Przewagą metody a posteriori jest możliwość logicznego rozdzielenia algorytmu fizycznego od detektora kolizji. Detektor nie musi mieć wiedzy o zmiennych fizycznych opisujących ruch obiektów oraz o rodzajach materiałów z których są wykonane. Problemem tej metody jest jednak etap poprawiania niezgodnych z fizyką pozycji

obiektów, przez co metoda a priori cechuje dużo większą dokładnością i stabilnością symulacji.

Specjalnym przypadkiem do rozpatrzenia jest stan spoczynku. Jeżeli odległość oraz ruch dwóch obiektów względem siebie jest poniżej pewnego progu, ciała powinny przejść w stan spoczynku, z którego zostaną wyrwane dopiero pod działaniem nowej siły.

5.1.2. Algorytm detekcji kolizji

Oczywistym podejściem do detekcji kolizji jest sprawdzenie kolizji między wszystkim parami symulowanych obiektów. Ponieważ modele 3d składają się ze zbioru trójkątów, detekcja kolizji sprowadza się do sprawdzenia czy nie doszło do przecięcia trójkątów jednego z obiektów z trójkątami drugiego. Powstało wiele różnych algorytmów badających czy doszło do przecięcia dwóch trójkątów.

Jako bazę detektora wybrałem algorytm 'Fast Triangle-Triangle Intersection Test' autorstwa Oliviera Devillers oraz Philippe'a Guigue z INRIA [RR4488]. Algorytm ten w pierwszym kroku sprawdza czy trójkąty przecinają wzajemnie płaszczyzny na których leżą. Jest to niezbędny warunek by mogło dojść do kolizji. Jeżeli istnieje potencjalna kolizja, wystarczy przeprowadzić cztery kolejne testy by uzyskać ostateczne rozstrzygnięcie.

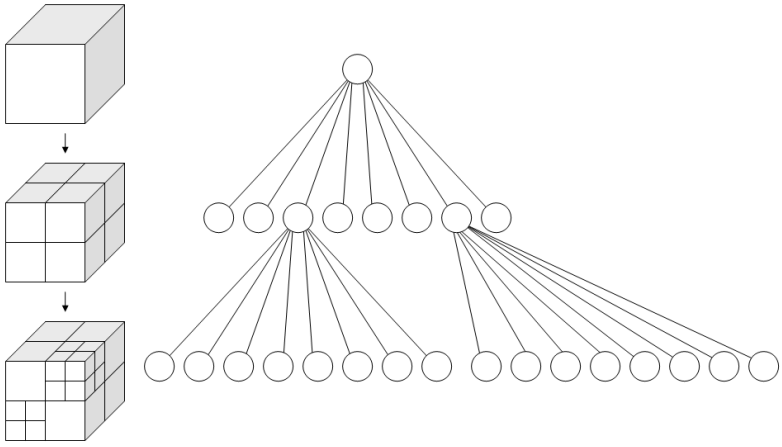
Algorytmy wykrywania przecinających się trójkątów są obecnie uproszczone do granic możliwości. Modele obiektów składają się najczęściej z co najmniej kilkuset trójkątów, a modele postaci mogą być zbudowane nawet z kilku milionów trójkątów. Jak łatwo wyliczyć, sprawdzenie kolizji zachodzącej między dwoma modelami może wymagać przeprowadzenia od dziesiątek tysięcy do milionów testów kolizji trójkątów. Grafika komputerowa umożliwia wykorzystanie sztuczek związanych z mapowaniem modelu, które umożliwiają ograniczenie ilości wierzchołków bez wyraźnej straty na jakości wizualnej. Kolejną techniką obniżającą ilość wierzchołków jest przygotowanie dla detektora kolizji modelu fizycznego, będącego uproszczoną wersją modelu pokazywanego na ekranie. Pozwala to na opisanie modelu postaci za pomocą jedynie

kilkudziesięciu do kilkuset wierzchołków. Nadal daje to jednak kilka tysięcy testów dla tylko jednej pary obiektów.

W celu ograniczenia ilości wymaganych testów kolizji trójkątów, stosuje się kilka technik umożliwiających przycinanie zbioru potencjalnych trójkątów mogących wejść w kolizję z innym obiektem, a także przycinanie zbioru obiektów które mogą się ze sobą przeciąć.

5.1.3. Hierarchie brył otaczających

Technika ta polega na podziale obiektu na hierarchiczne drzewo mniejszych elementów. Każdy węzeł drzewa zawiera informację o otaczającej go bryle. Bryłą tą najczęściej są sfery, sześciany bądź walce.



Rysunek 2: Hierarchiczny podział przestrzeni (na przykładzie Drzewa Ósemkowego).

Jeżeli bryła otaczająca dany węzeł przecina się z bryłą otaczającą węzeł należący do innego obiektu, to oznacza, że występuje potencjalna kolizja obiektów. Aby ją rozstrzygnąć, należy sprawdzić czy zachodzą kolizje między bryłami otaczającymi węzły niższego rzędu. Jeśli algorytm dojdzie do najniższego rzędu hierarchii brył otaczających, będzie musiał dokonać testu kolizji między trójkątami należącymi do tych najmniejszych węzłów w hierarchii. Dzięki zastosowaniu tej techniki już pierwszy test może wykluczyć kolizję między obiektami. Co więcej, wszystkie testy przeprowadzane w węzłach drzewa operują na prostych bryłach geometrycznych, pozwalających na

efektywne sprawdzenie czy zachodzi kolizja. Dopiero test najniższego rzędu wymaga badania kolizji między trójkątami, jednakże jest ich na tym poziomie tylko od kilku do kilkunastu dla każdego z węzłów.

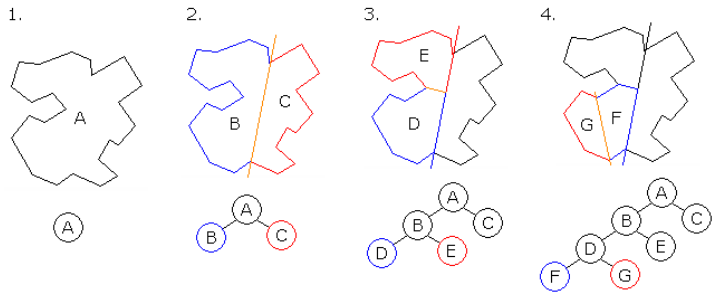
Projektując detektor, warto się również zastanowić, jak wielką precyzję wykrywania kolizji wymaga nasza symulacja. Prawdopodobnie możemy całkowicie zrezygnować z testowania kolizji między trójkątami, gdyż wystarczy nam przybliżenie zapewnione przez hierarchię brył otaczających.

[dopisać doświadczenia z implementacji]

5.1.4. Wybór obiektów mogących kolidować

Złożona scena 3d może składać się z bardzo dużej ilości obiektów. Obiekt poruszający się np. na scenie rozgrywającej się w mieszkaniu może wejść w kolizję z dużą ilością obiektów. Mogą to być zarówno drobne modele przedstawiające np. książki, ale też obiekty znaczniejszych rozmiarów przedstawiające meble i ściany. Sprawdzanie czy poruszający się obiekt wchodzi w kolizję z wszystkim modelami w scenie niepotrzebnie mnoży ilość przeprowadzanych testów, wpływając na znaczne obniżenie szybkości detekcji kolizji. Jeśli postać znajduje się w kuchni, to niema najmniejszego sensu sprawdzanie, czy przypadkiem nie zderzyła się z wanną mieszczącą się na drugim końcu mieszkania. W celu wyeliminowania takich zbędnych testów stosuje się techniki podziału przestrzeni.

Najbardziej uniwersalnym sposobem podziału przestrzeni jest zastosowanie drzewa BSP (Binary Space Partitioning - Binarny Podział Przestrzeni). Każdy węzeł takiego drzewa dzieli opisywaną przez siebie przestrzeń na dwie podprzestrzenie leżące po przeciwnych stronach pewnej hiperpłaszczyzny. Drzewo takie można zastosować do podziału przestrzeni o dowolnej wymiarowości. Pozwala ono na bardzo szybkie odrzucenie obiektów znajdujących się poza rozpatrywaną podprzestrzenią.



Rysunek 3: Drzewo Binarnego Podziału Przestrzeni.

Drzewo BSP dobrze nadaje się do opisu zamkniętych pomieszczeń, jednakże otwarte przestrzenie lepiej opisują drzewa ósemkowe (lub czwórkowe dla przestrzeni dwuwymiarowych). Drzewa takie polegają na otoczeniu sceny sześcianem, który jest następnie dzielony na osiem mniejszych sześcianów. Podział taki zagłębia się rekurencyjnie, aż do osiągnięcia ustalonego kryterium – zadanej głębokości lub minimalnej ilości obiektów w minimalnym sześcianie. Obiekt przypisywany jest do minimalnego sześcianu w którym może się pomieścić oraz wewnątrz którego znajduje się jego środek ciężkości. Wybierając obiekty które mogą wejść w kolizję z danych modelem, należy sprawdzić wszystkie obiekty znajdujące się w jego sześcianie (a więc i w jego sześcianach niższych rzędów) oraz sąsiednie sześciany. W praktyce sprawdzając sześciany sąsiadujące, wystarczy przejrzeć te leżące po stronie rosnących współrzędnych – kolizje z pozostałymi sąsiadami zostały sprawdzone podczas badania kandydatów dla obiektów należących do tamtych podprzestrzeni.

5.2. SYMULACJA FIZYKI

5.2.1. Opis dynamiczny

Posiadając wiedzę o siłach oddziałujących na obiekty oraz kolizjach do których doszło, można przejść do symulacji ruchu. Przykładowym podejściem może być otwarcie podręcznika do fizyki i zastosowanie przedstawionych w nim wzorów opisujących ruch.

Stosując opis dynamiczny na poziomie modelu jako całości, należałoby uwzględnić w algorytmie takie wielkości jak pozycja, prędkość, masa obiektu, a także

działające nań siły. Model taki jest w najprostszym przypadku ciałem sztywnym, nie można więc również zapomnieć o opisujących ruch obrotowy momencie bezwładności oraz momencie siły. Gdy zechcemy symulować ciała odkształcalne, takie jak materiały, powierzchnie elastyczne, bądź rośliny, nasz algorytm rozrośnie się jeszcze bardziej.

[dopisać doświadczenia z implementacji]

5.2.2. Całkowanie Verlet'a

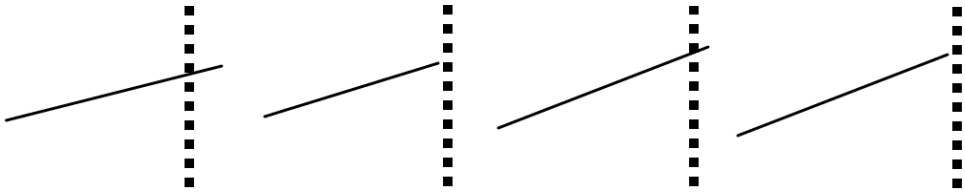
Alternatywne podejście do symulacji zachowania się ciał fizycznych zastosowano po raz pierwszy w grze *Hitman: Codename 47*. Rozwiązanie to przedstawił w swym artykule dyrektor do spraw badań i rozwoju w firmie IO Interactive, Thomas Jakobsen [Jacob]. Polega ono na rozbiciu obiektu na zbiór punktów materialnych i rozpatrywaniu ruchu każdego punktu osobno. Ponadto, w celu poprawy stabilności algorytmu, zrezygnowano z przechowywania prędkości cząsteczki. W efekcie każda cząsteczka opisywana jest przez cztery wielkości fizyczne: pozycję, pozycję w poprzedniej klatce (wymagana do wyliczenia faktycznej prędkości), masę oraz wypadkowe przyśpieszenie cząsteczki. Cały opis ruchu pojedynczego punktu materialnego sprowadza się do następującego wzoru:

$$x_{t+1} = 2 \cdot x_t - x_{t-1} + a \cdot \Delta t^2$$

Podejście to nazywa się całkowaniem Verlet'a i jest szeroko wykorzystywane w symulacji dynamiki molekularnej. Prędkość ciała zaszyta jest w tym wzorze pod postacią różnicy położenia w w stałej jednostce czasu: $x_t - x_{t-1}$.

5.2.3. Nakładanie ograniczeń na punkty materialne

Za pomocą chmury niezależnych punktów materialnych, możemy opisać co najwyżej silnik cząsteczkowy. Jeśli jednak chcielibyśmy by nasze cząsteczki opisywały bardziej złożone ciała, musimy nałożyć na nie ograniczenia.



Rysunek 4: Kolejne kroki procesu spełniania ograniczeń.

W podejściu przedstawionym przez Jakobsena zbiór ograniczeń nałożonych na system spełniany jest poprzez relaksację. Proces ten postaram się opisać na przykładzie dwóch cząsteczek opisujących sztywny pręt. Odległość między tymi cząsteczkami musi być stała. Dodatkowo pręt nie powinien przebić żadnego innego obiektu. Na rysunku 3a widzimy sytuację w której to drugie ograniczenie zostało naruszone. Pręt przechodzi przez ścianę reprezentowaną przez przerywaną linię. W pojedynczym kroku symulacji, przez zadaną ilość iteracji, silnik będzie po kolei spełniał każde z ograniczeń. W ramach spełniania tego ograniczenia, punkt który przebił ścianę został rzutowany na poprawną pozycję. Spełnienie tego ograniczenia naruszyło jednak ograniczenie drugie – cząsteczki znalazły się zbyt blisko siebie (rys. 3b). Aby spełnić ten warunek, punkty muszą zostać rozsunięte – spowoduje to jednak ponowne przebicie ściany (rys. 3c). W kolejnych iteracjach relaksacji ograniczenia będą naruszane w coraz mniejszym stopniu, aż w końcu wszystkie zostaną spełnione.

Co interesujące, w pojedynczej klatce symulacji nie musimy doprowadzać do spełnienia wszystkich ograniczeń. Korygowanie pozycji cząsteczek może być kontynuowane w kolejnych klatkach.

5.2.4. Symulacja ciał odkształcalnych

Aby opisać zachowanie ciał odkształcalnych, takich jak ubrania bądź rośliny, wystarczy każdy wierzchołek siatki modelu symulować jako pojedynczy punkt materialny. Na punkty te powinny zostać nałożone warunki zachowania stałej odległości pokrywające się z krawędziami trójkątów siatki. Dla uzyskania ładnej animacji takich ciał wystarczy tylko pojedyncza iteracja relaksacji na klatkę symulacji.

5.2.5. Symulacja ciał sztywnych

Najprostszym sposobem na przedstawienie ciała sztywnego, jest zastosowanie takiego samego podejścia jak dla ciał odkształcalnych, lecz z nałożeniem ograniczeń stałej odległości na wszystkie pary wierzchołków. Wydajniejszym podejściem jest jednak opisanie całego ciała sztywnego z wykorzystaniem jedynie czterech punktów materialnych połączonych ograniczeniami stałej odległości. Ustawienie tych cząstek w konfiguracji np. czworościanu foremnego zapewni 6 stopni swobody, a więc dokładnie

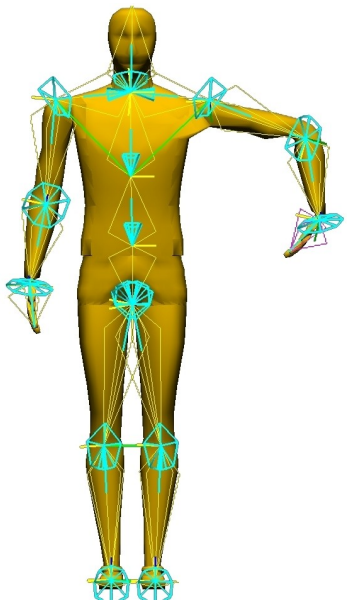
tyle, ile posiada ciało sztywne. W zależności od sposobu ustawienia cząsteczek, ciało będzie miało różny moment bezwładności.

Jedynym problemem jaki pozostaje do rozwiązania jest rozłożenie siły przyłożonej do pewnego punktu (np. punktu kolizji) na siły składowe działające na poszczególne cząsteczki. Dowolny punkt w przestrzeni może być przedstawiony jako liniowa interpolacja punktów materialnych które symulujemy. Korzystając z parametry tej interpolacji możemy dokonać operacji odwrotnej by proporcjonalnie rozłożyć siłę pomiędzy wszystkie cząsteczki.

[dopisać doświadczenia z implementacji]

5.2.6. Fizyka szkieletu

Silnik oparty na całkowaniu Verlet'a bardzo dobrze nadaje się również do symulowania fizyki ciał posiadających szkielet. Wystarczy stawy przedstawić jako punkty materialne, a kości jako warunki zachowania stałej odległości. Dodatkowo należy dodać ograniczenia kątowe, które uniemożliwią 'wykręcanie stawów'. Dobrym rozwiązaniem ograniczającym przecinanie się kości postaci (silnik fizyczny rozpatruje tylko kolizje z innymi modelami), jest dodanie warunków minimalnej odległości pomiędzy takimi stawami jak kolana i kostki stóp.



Rysunek 5: Szkielet z nałożonymi ograniczeniami kątowymi.

W przeciwieństwie do innych typów obiektów, cząsteczki reprezentujące stawy muszą być napędzane nie tylko samym procesem całkowania Verlet'a, ale również przez zewnętrzne źródła ruchu, takie jak animacje przygotowane przez artystów. Rozwiązaniem problemu może być algorytm, który będzie mieszał zewnętrzną animację z wynikiem działania całkowania. Waga każdego ze źródeł powinna być zmienna. Jeśli np. model zostanie uderzony z dużą siłą, to kontrolę powinien przejąć silnik Verlet'a. W pozostałych sytuacjach większą wagę powinno się przykładać do 'woli' postaci, czyli animacji narzuconych przez logikę gry.

[dopisać doświadczenia z implementacji]

6. PODSUMOWANIE

6.1. UWAGI I WNIOSKI

6.2. MOŻLIWOŚCI DALSZEGO ROZWOJU

7. BIBLIOGRAFIA

7.1. ZAGADNIENIA ZWIĄZANE Z PROGRAMOWANIEM GIER

[PPG_T1] 'Perełki programowania gier – tom 1' pod redakcją Marca DeLoura, Wydawnictwo Helion 2002

[LIB3DS] Darmowa biblioteka przetwarzająca pliki 3ds
http://ww.lib3ds.org/

7.2. OPENGL

[GL_TRB] The Red Book
http://fly.srk.fer.hr/~unreal/theredbook/

[GL_TBB] The Blue Book
http://www.rush3d.com/reference/opengl-bluebook-1.0/index.html

[NEHE] NeHe Productions
http://nehe.gamedev.net/

[Tulane] Tulane.edu tutorial
http://www.eecs.tulane.edu/www/Terry/OpenGL/Introduction.html#Introduction

[VBO] Dokumentacja rozszerzenia VBO
http://www.spec.org/gpc/opc.static/vbo_whitepaper.html

[GL_Ext] Dokumentacja rozszerzeń OpenGL
http://www.opengl.org/documentation/specs/man_pages/hardcopy/GL/html/

7.3. ANIMACJA SZKIELETOWA

[Skel] Animacja szkieletowa z wykorzystaniem shaderów wierzchołków
http://grafika.iinf.polsl.gliwice.pl/doc/09-SKE.pdf

[Anim] Podstawy animacji komputerowej
http://sound.eti.pg.gda.pl/student/sdio/12-Animacja.pdf

7.4. DETEKCJA KOLIZJI

[Wiki_CD] Collision Detection – Wikipedia

http://en.wikipedia.org/wiki/Collision_detection

[RR4488] 'Faster-Triangle Intersection Tests' autorstwa Oliviera Devillers i Philippe Guigue z INRIA

http://citeseer.ist.psu.edu/687838.html

ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-4488.pdf

7.5. FIZYKA

[Jacob] Advanced Character Physics – Verlet Integration

http://www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml

[Baraff] Physically Based Modeling: Principles and Practice

http://www.cs.cmu.edu/~baraff/sigcourse/

[Wiki_Doll] Ragdoll Physics – Wikipedia

http://en.wikipedia.org/wiki/Ragdoll_physics

8. DODATEK – ZAWARTOŚĆ PŁYTY KOMPAKTOWEJ