POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRONIKI I TECHNIK INFORMACYJNYCH
INSTYTUT INFORMATYKI

Rok Akademicki 2007/2008

PRACA DYPLOMOWA MAGISTERSKA

Dariusz Maciejewski

Gra komputerowa “Camera Fighter”

Opiekun pracy
prof. nzw. dr hab. Przemysław Rokita

Ocena: .....

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Specjalność: Inżynieria Systemów Informacyjnych

Data Urodzenia: 3 grudnia 1983

Data Rozpoczęcia Studiów: 1 października 2006

Życiorys:

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 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 DYPLOMOWY

Złożył egzamin dyplomowy w dniu ..... 200.....r.

z wynikiem .....

Ogólny wynik studiów .....

Dodatkowe wnioski i uwagi Komisji .....

.....

.....

STRESZCZENIE

[napisać]

Słowa kluczowe:

gra, C++, OpenGL, 3D, szkielet gry, silnik graficzny, silnik fizyczny

---

ABSTRACT

"Camera Fighter" - a computer game.

[napisać]

Key words:

game, C++, OpenGL, 3D, game framework, graphical engine, physical engine

Spis Treści:

1. WSTĘP

1.1. CEL PRACY

[napisać]

1.2. ROZWAŻANE METODY IMPLEMENTACJI

[napisać]

1.3. ZAWARTOŚĆ PRACY

[napisać]

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
/ Textures	- wspólny kod związany z teksturami
/ OGL	- kod specyficzny dla OpenGL, renderowanie map
/ Extensions	- rozszerzenia OpenGL
/ Render	- renderowanie obiektu xModel
/ 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, ich opis znajduje się w rozdziale poświęconym konfiguracji gry.

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

```
};
```

3.4.6. Menadżer zbierania statystyk i monitorowanie wydajności

[napisać coś o StatMgr i Profiler]

3.5. BIBLIOTEKI MODELI

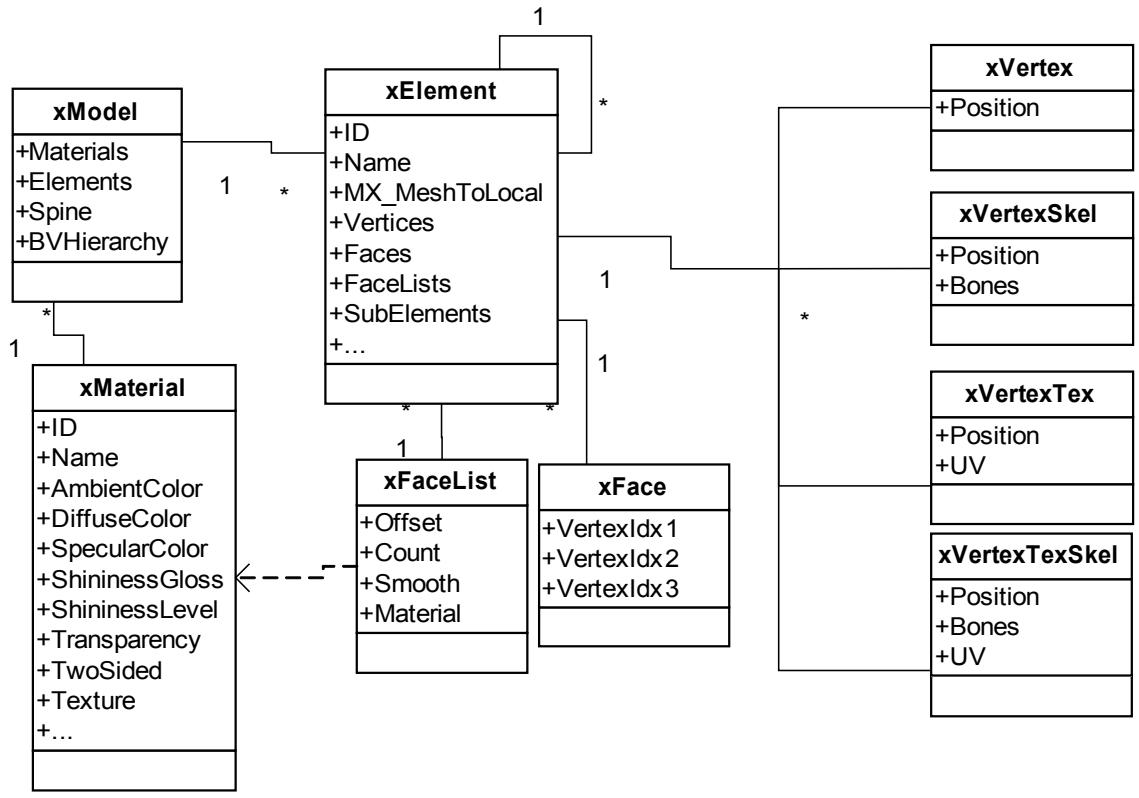
3.5.1. Biblioteka Lib3ds

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

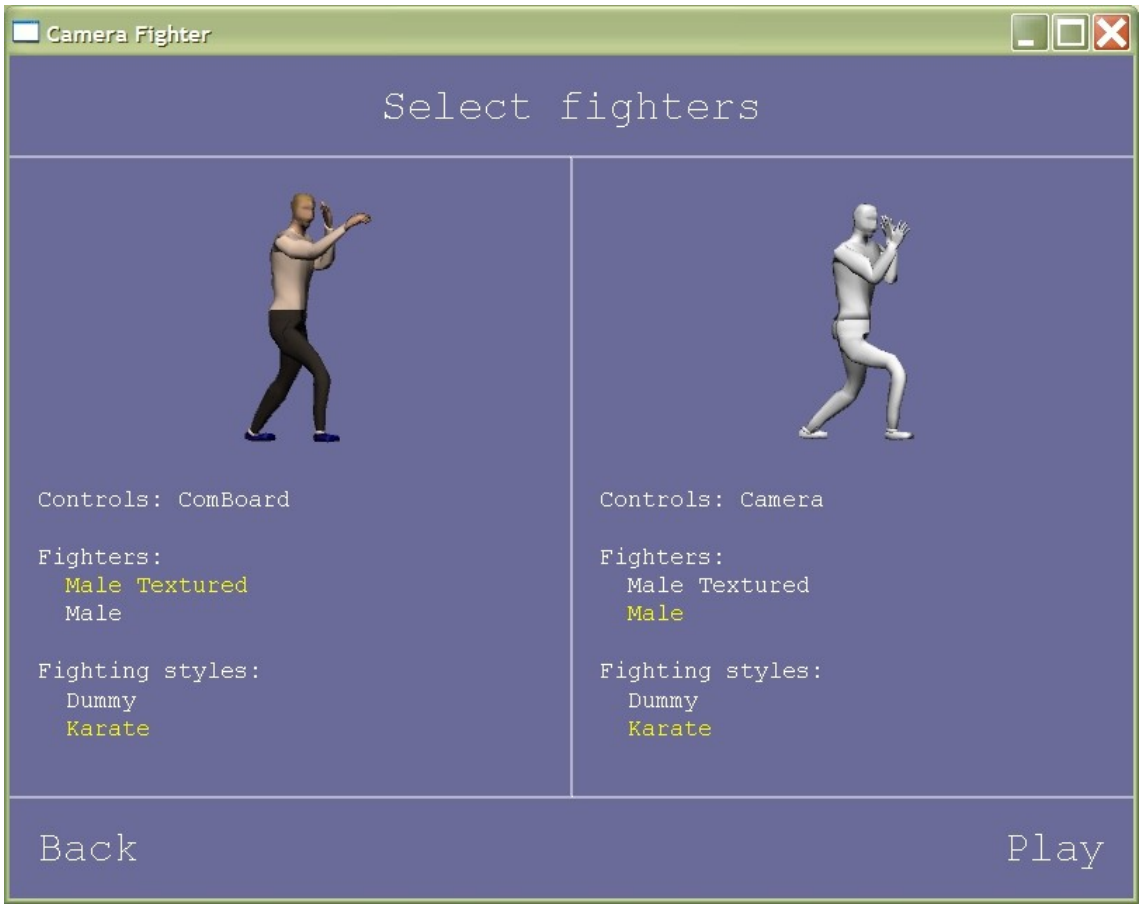
```
};
```

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:



Rysunek 2: Ekran wyboru graczy

- `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

Rysunek 3: 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

ComBoard

Gra udostępnia dwa sposoby sterowania – pierwszy gracz sterowany jest z klawiatury, drugi z kamer. Za interpretację sterowania klawiaturą odpowiedzialna jest klasa `ComBoard`. Klasa ta zawiera logikę wykonywania kombinacji ruchów i ciosów składających się na dany styl walki. Style walki można wczytywać z plików tekstowych, ich przykłady znajdują się w plikach /Data/models/anim/*.txt.

Na styl walki składa się lista akcji jakie może wykonać postać. Pojedyncza akcja to zestaw animacji szkieletowych, czas jej trwania oraz akcja do której postać powinna przejść po jej wykonaniu. Ponadto każda akcja może posiadać listę 'kombosów' – natychmiastowych przejść do innych akcji w wyniku wciśnięcia we właściwym momencie odpowiedniego klawisza.

Poza akcjami, styl walki zawiera również informację o odpowiadających sobie kościach modelu. Wiedza o lustrzanych kościach umożliwia wykorzystanie tej samej animacji do poruszania dowolną ręką bądź nogą.

Ponieważ gracz sterowany kamerą nie ma możliwości przemieszania postaci, za jego automatyczne podążanie za przeciwnikiem również odpowiada system ComBoard. Każdy styl walki powinien zawierać listę odnośników do akcji, które wpływają na pozycję gracza oraz ich przybliżony wpływ na końcowe położenie. Na podstawie tych informacji ComBoard może wybrać najbardziej optymalny ruch, jaki doprowadzi go do punktu docelowego.

Domyślne skojarzenia klawiszy dla systemu ComBoard są następujące:

- A – blok lewą ręką (kombo LeftHandGuard)
- S – cios lewą ręką (kombo LeftPunch)
- D – cios prawą ręką (kombo RightPunch)

- F – blok prawą ręką (kombo RightHandGuard)
- Z – blok lewą nogą (kombo LeftLegGuard)
- X – kopnięcie lewą nogą (kombo)
- C – kopnięcie prawą nogą (kombo RightKick)
- V – blok prawą nogą (kombo RightLegGuard)
- strzałka do góry – ruch do przodu (kombo Forward)
- strzałka do dołu – ruch w tył (kombo Backward)
- strzałka w lewo – obrót w lewo (kombo Left)
- strzałka w prawo – obrót w prawo (kombo Right)

Znaczenie klawiszy jest oczywiście mocno przybliżone, gdyż faktyczna akcja zależy od kombinacji klawiszy – np. gdy LeftKick spowoduje przejście do akcji “kopnięcie”, to klawisz Forward może w tym kontekście oznaczać przejście do akcji “kopnięcie do góry”, a nie zaniechanie kopnięcia i wykonanie kroku do przodu. Wszystko zależy od projektanta danego stylu walki.

3.7.3. Konsola

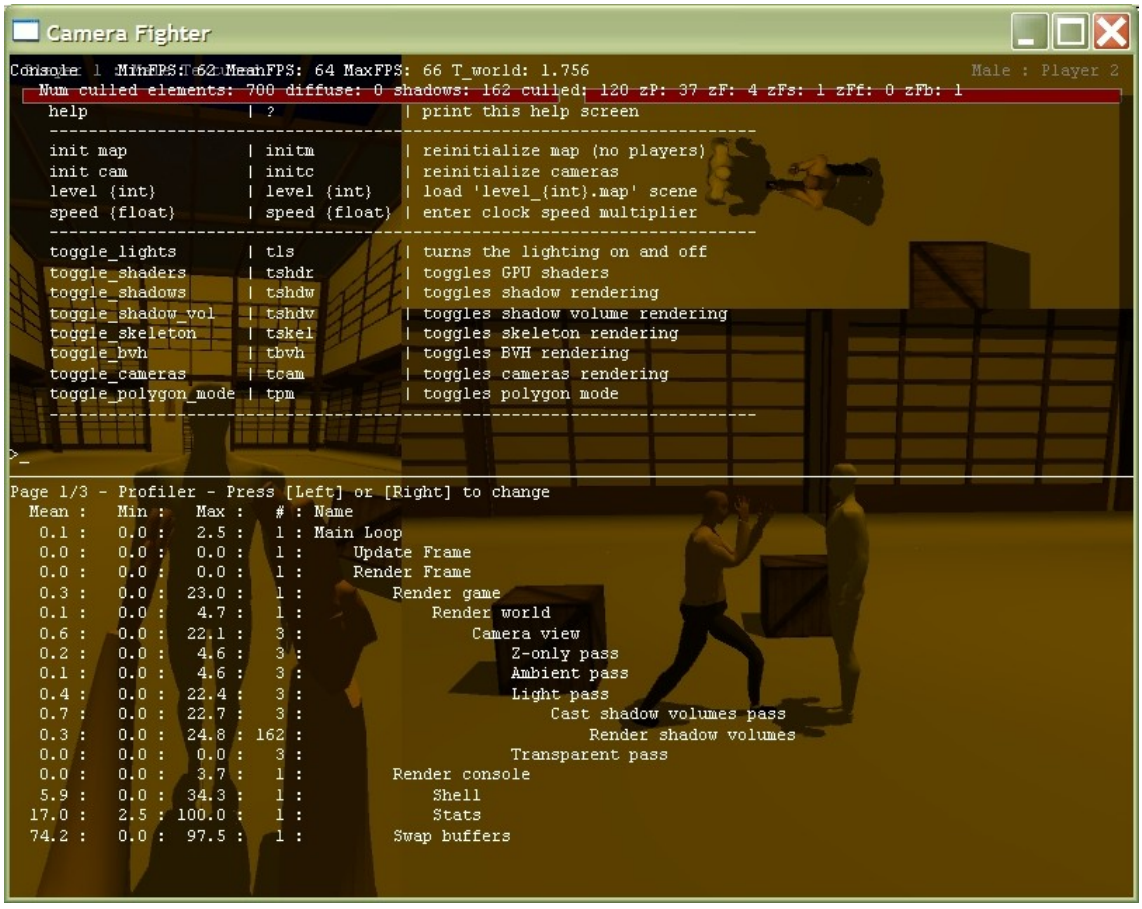
Scena konsoli jest odmienna od pozostałych scen. Gdy zostaje pierwszy raz uruchomiona, przejmuje kontrolę nad poprzednią sceną i wyświetla się w półprzezroczystym okienku ponad nią. W zależności od stanu w jakim znajduje się konsola, czas oraz komunikaty klawiatury docierające do poprzedniej sceny mogą być wstrzymywane lub nie. Co więcej, próba przejścia do innej sceny zaowocuje zmianą poprzedniej sceny, a nie zastąpieniem konsoli.

Zadaniem konsoli jest wyświetlanie informacji diagnostycznych oraz umożliwienie użytkownikowi wydawania scenom poleceń tekstowych. Lista dostępnych poleceń dostępna jest po wysłaniu polecenia '?'.

Domyślne skojarzenia klawiszy sterujące konsolą:

- ` – pokazanie konsoli
- Escape – ukrycie konsoli
- Page Up – przewinięcie tekstu o stronę do góry

- Page Down – przewinięcie tekstu o stronę do dołu
- strzałka do góry – przewinięcie tekstu o linijkę do góry
- strzałka do góry – przewinięcie tekstu o linijkę do dołu
- strzałka w lewo – zmiana strony diagnostycznej na poprzednią
- strzałka w prawo – zmiana strony diagnostycznej na następną
- kółko myszy – przewinięcie tekstu



Rysunek 4: Konsola podczas rozgrywki

3.7.4. Edytor modeli

Wstęp

Gra posiada wbudowany edytor modeli, który pełni trzy role. Po pierwsze, pozwala dodać do modelu szkielet oraz powiązać wierzchołki modelu z kośćmi. Po

drugie, umożliwia tworzenie animacji bazujących na ruchu szkieletu. Na koniec, można w nim ręcznie stworzyć hierarchię brył otaczających (Bounding Volumes Hierarchy).

Edytor wspiera jednoczesną edycję dwóch modeli, gdy obiekt ma być reprezentowany przez siatkę graficzną oraz uproszczoną siatkę fizyczną. Jest to przydatne podczas tworzenia szkieletu i animacji, gdyż można kontrolować, czy oba modele pokrywają się gdy poruszymy kośćmi.

Do sceny edytora można się dostać ze sceny menu oraz ze sceny rozgrywki – wystarczy kliknąć interesujący nas model na widoku z głównej kamery.

Podczas edycji mamy do wyboru siedem typów widoku, dostępnych skrótami klawiaturowymi (domyślnie klawisze '1'-'7'):

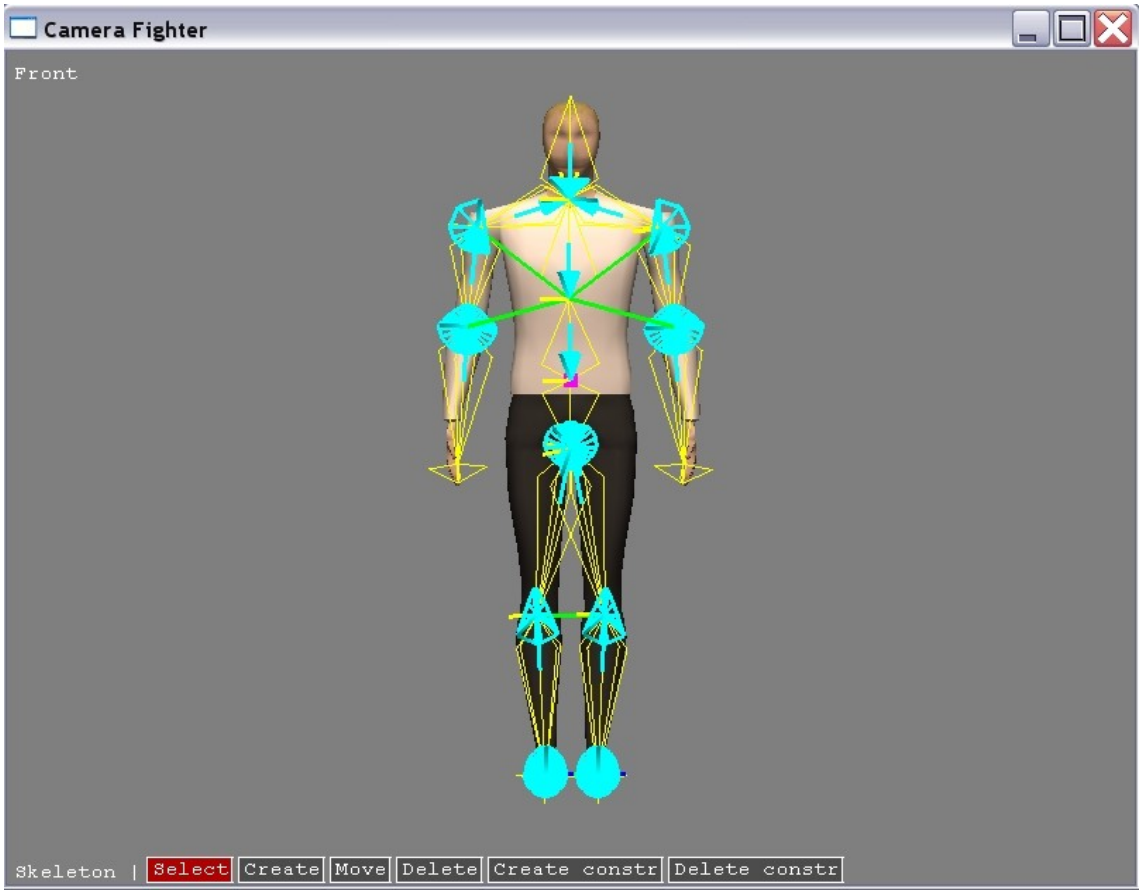
- rzut prostopadły od przodu
- rzut prostopadły od tyłu
- rzut prostopadły od lewej
- rzut prostopadły od prawej
- rzut prostopadły od góry
- rzut prostopadły od dołu
- dowolny rzut perspektywiczny

Do sterowania kamerami służą domyślnie następujące skróty:

- kursory – przesuwanie widoku w pionie i poziomie
- Page Up / Page Down – przemieszczanie obserwatora oraz punktu obserwacji do przodu / do tyłu. Odsunięcie punktu obserwacji może być wymagane dla dużych modeli, bądź dużych przybliżeń, gdyż widok może wtedy przecinać model w jego środku
- kółko myszy – zmiana powiększenia (dystansu między obserwatorem a punktem obserwacji)
- 'U', 'J', 'H', 'K' – dla kamery perspektywicznej, orbitowanie wokół punktu obserwacji

Tworzenie szkieletu

Utworzenie pierwszej – podstawowej kości skutkuje dodaniem struktur związanych ze szkieletem do modelu. Wiąże się to głównie ze zmianą typu wierzchołków z `xVertex` (bądź `xVertexTex`) na `xVertexSkel` (bądź `xVertexSkelTex`). Pierwsza kość ma specjalne zastosowanie – nie posiada długości, jest jedynie punktem. Jej zmiana definiuje przesunięcie, a nie obrót całego modelu. Najlepiej umieścić ją w okolicach centrum masy modelu.



Rysunek 5: Edytor szkieletu

Podczas edycji szkieletu mamy dostępne następujące akcje (wybierane przyciskiem na dole ekranu, bądź skrótem klawiaturowym):

- [Select] (skrót 'N') – kliknięcie na kości spowoduje jej zaznaczenie, kliknięcie na wolną przestrzeń zaznaczy kość podstawową

- [Create] – kliknięcie spowoduje dodanie we wskazanym miejscu kości potomnej do obecnie zaznaczonej
- [Move] (skrót 'M') – przeciągnięcie myszki spowoduje przesunięcie końca zaznaczonej kości do wskazanego punktu
- [Delete] (skrót 'Delete') – powoduje usunięcie zaznaczonej kości
- [Create constraint] – przechodzi w tryb dodawania ograniczenia ruchu szkieletu
- [Delete constraint] – kliknięcie spowoduje usunięcie wskazanego ograniczenia

Edycja ograniczeń ruchu

Po przejściu w tryb dodawania ograniczeń należy wybrać typ ograniczenia, jakie chcemy utworzyć. Do wyboru mamy:

- [max] – określa maksymalną odległość w jakiej mogą znaleźć się kości
- [min] – określa minimalną odległość w jakiej mogą znaleźć się kości
- [const] – określa stałą odległość w jakiej powinny znajdować się kości – ograniczenie to jest tworzone automatycznie dla sąsiadujących kości
- [ang] – ograniczenie kąta zgięcia kości względem stanu spoczynku
- [weight] – pozwala zdefiniować masę wybranej kości

Po wybraniu typu ograniczenia należy wybrać kości, których ono dotyczy, a następnie podać parametry właściwe dla danego ograniczenia.

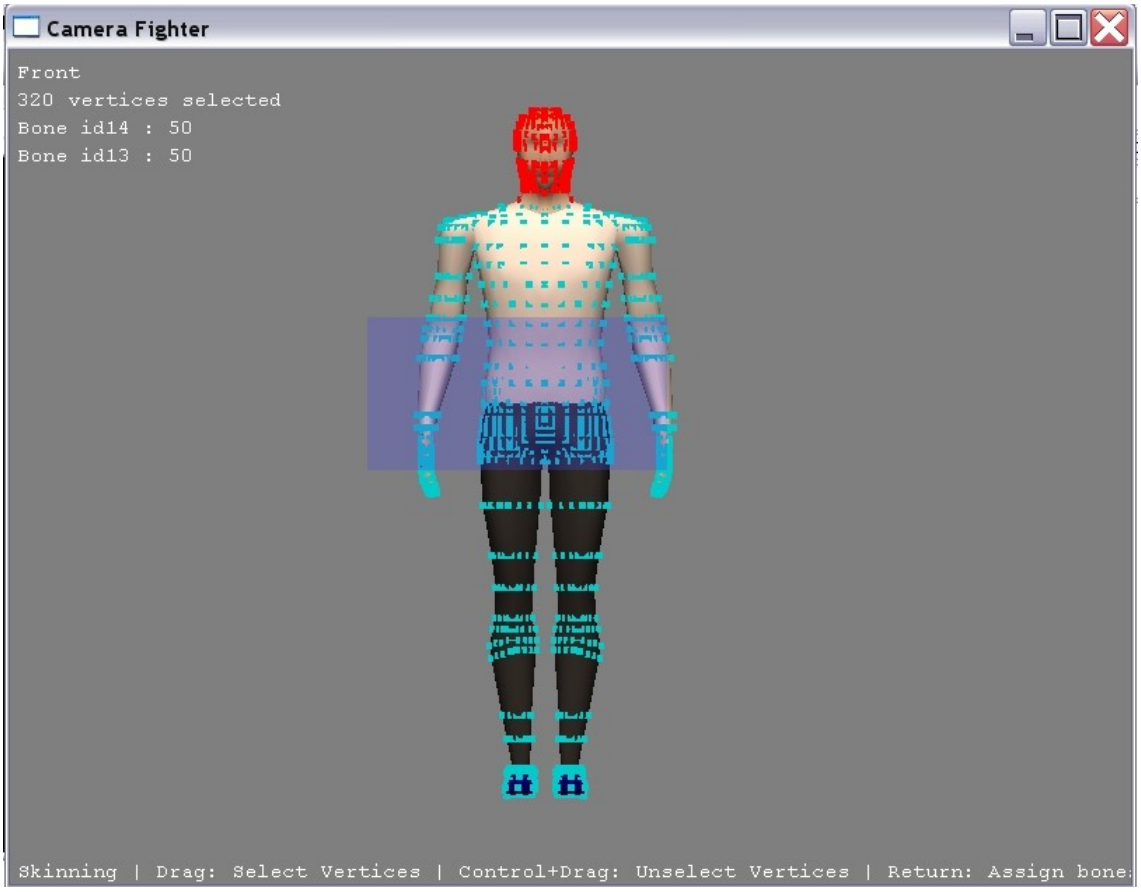
Skórowanie modelu

Mając gotowy szkielet, możemy przejść do przypisania wierzchołków do kości. Można zdefiniować do czterech różnych kości na jeden wierzchołek, najczęściej będą to dwie sąsiednie kości. Proces skórowania modelu można dzieli się na następujące kroki:

- kliknięcie myszką – wybór elementu który chcemy skórować
- przeciągnięcie myszki – zaznaczenie wierzchołków
- klawisz Control + przeciągnięcie myszki – odznaczenie wierzchołków
- klawisz Enter – przejście do wyboru kości
- kliknięcie – wybór kości

- podanie procentowego wpływu kości na wybrane wierzchołki
- jeśli suma wpływu wszystkich kości osiągnęła 100%, powrót do wyboru wierzchołków
- powrót do wyboru kości

Wierzchołki z przypisanymi kośćmi mają kolor turkusowy, zaznaczone wierzchołki są czerwone, a pozostałe są żółte.



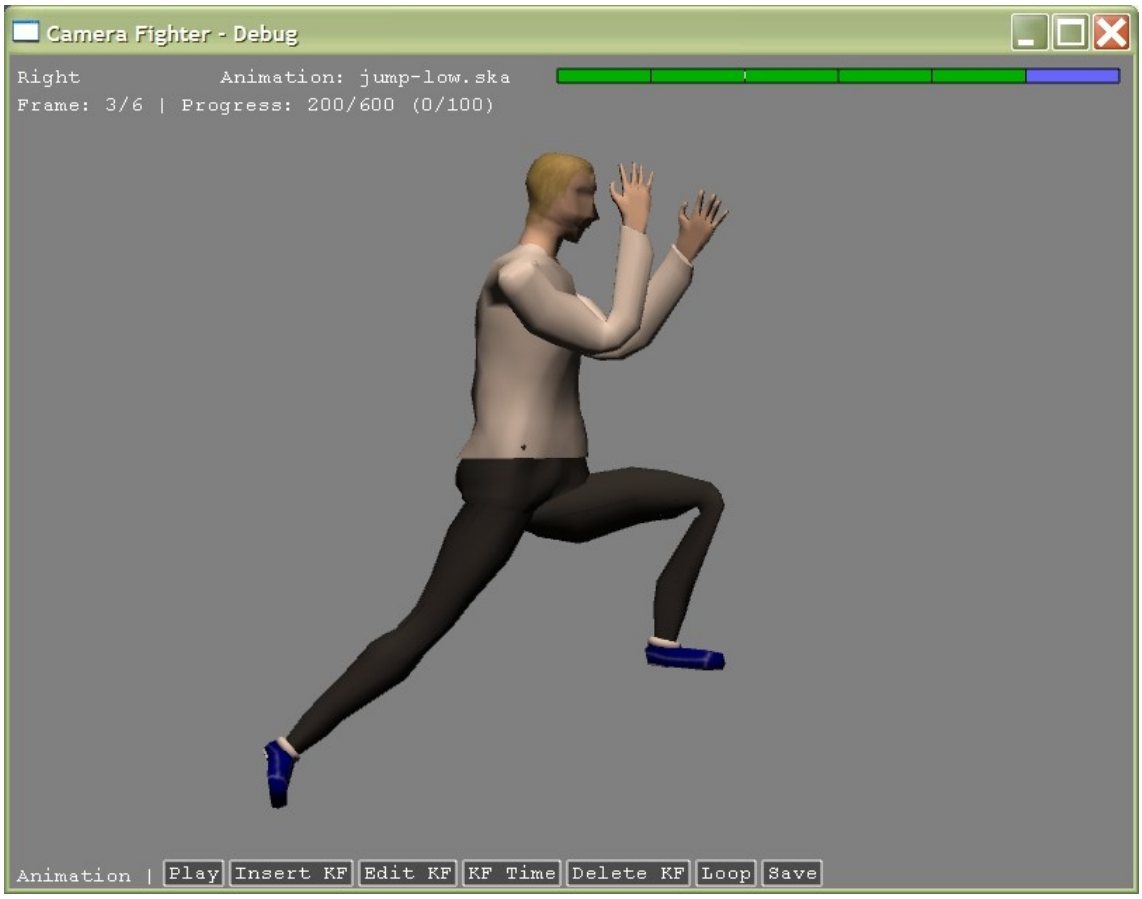
Rysunek 6: Proces skórowania modelu

Animacja szkieletowa

Gdy wszystkie wierzchołki zostaną już przypisane do kości, można przejść do tworzenia animacji opierających się na ruchu szkieletu.

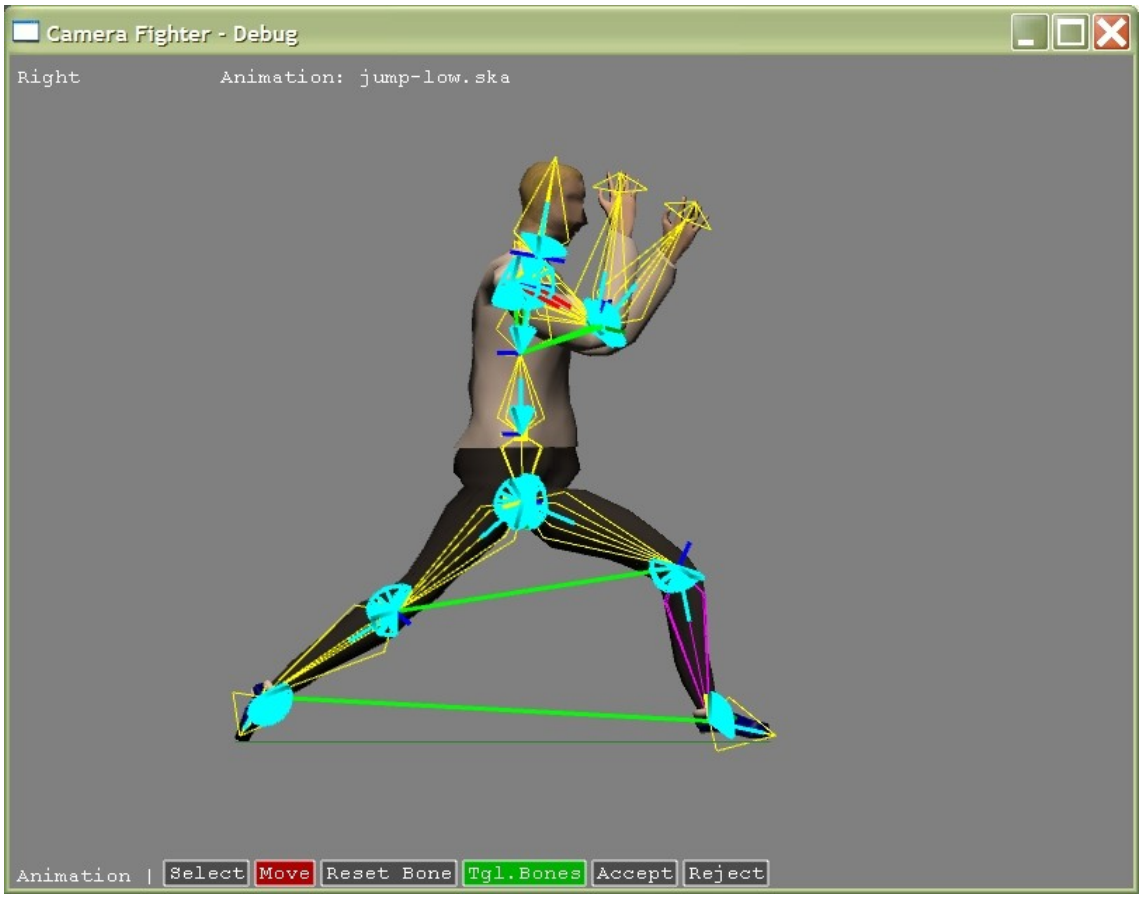
W lewym górnym rogu edytora wyświetlany jest numer edytowanej klatki oraz ilość wszystkich klatek animacji. Obok widnieje licznik postępu – wyświetla moment

animacji który obecnie widać oraz całkowitą długość animacji. W nawiasach podane są dodatkowo te same informacje dla pojedynczej klatki kluczowej. Postęp animacji jest również przedstawiony graficznie w prawym górnym rogu ekranu. Czarne kreski oznaczają początek klatek kluczowych, natomiast biała kreska to obecna pozycja animacji.



Rysunek 7: Edytor animacji szkieletowych

Klatka kluczowa może składać się z dwóch etapów – zamrożenia i interpolacji. Etapy te oznaczone są na pasku postępu odpowiednio niebieskim i zielonym kolorem. Gdy animacja znajduje się w stanie zamrożenia, pozycja szkieletu jest wierną kopią pozycji zdefiniowanej dla danej klatki. Gdy przejdziemy do etapu interpolacji, pozycja będzie płynnie przechodziła do pozycji zdefiniowanej w kolejnej klatce (bądź do stanu spoczynku dla ostatniej klatki). Czas trwania etapów obecnie oglądanej klatki można zdefiniować klikając przycisk [KF Time] na dole ekranu.



Rysunek 8: Edycja klatki kluczowej

Przewinięcia obecnie oglądanego momentu animacji można dokonać na kilka sposobów:

- Przycisk [Play] (domyślnie skrót 'P') – uruchomienie animacji – przytrzymywanie klawisza modyfikacji (domyślnie 'Control') spowolni bieg czasu
- Klawisz modyfikacji + klawisz lewo / prawo (domyślnie 'Control' + strzałka w lewo / prawo) – płynne przewijanie animacji we wskazanym kierunku
- Klawisz modyfikacji + klawisz lewo / prawo (domyślnie 'Control' + strzałka w dół / górę) – przeskoczenie do początku / końca klatki

Pozostałe opcje dostępne dostępne podczas edycji animacji:

- [Play] (skrót 'P') – omówione powyżej

- [Insert KF] – wstawia nową klatkę kluczową na bieżącej pozycji, stan szkieletu będzie taki jak widać obecnie
- [Edit KF] – edytuje bieżącą klatkę kluczową
- [KF Time] – umożliwia podanie czasów zamrożenia i interpolacji bieżącej klatki
- [Delete KF] (skrót 'Delete') – usuwa bieżącą klatkę
- [Loop] – wskazuje iż animacja będzie odtwarzana w kółko – ostatnia klatka będzie interpolowana do pierwszej klatki, a nie do pozycji spoczynkowej
- [Save] – umożliwia zapisanie animacji do pliku

Tworząc nową animację, posiadamy pojedynczą klatkę o długości 1 sekundy. Klikając przycisk [Edit KF] przejdziemy w tryb edycji bieżącej klatki. Edytując klatkę mamy dostępne następujące polecenia:

- [Select] (domyślnie skrót 'N') – kliknięcie na wskazanej kości spowoduje jej zaznaczenie i przejście do trybu [Move], kliknięcie na wolnej przestrzeni zaznaczy kość podstawową
- [Move] (skrót 'M') – przeciąganie myszką powoduje obrót zaznaczonej kości
- [Reset Bone] (skrót 'Delete') – anuluje obroty zaznaczonej kości
- [Tgl. Bones] (skrót 'B' – dostępny również z pozostałych ekranów edytora) – pokazuje bądź ukrywa szkielet
- [Accept] (skrót 'Enter') – akceptuje zmiany szkieletu i wraca do edycji animacji
- [Reject] (skrót 'Escape') – anuluje zmiany szkieletu i wraca do edycji animacji

Gdy mamy utworzoną pierwszą klatkę, możemy przejść do etapu dodawania kolejnych klatek. Bardzo często chcemy aby animacja kończyła się w pozycji początkowej.

Za pomocą prostej sztuczki możemy spowodować wykonanie wiernej kopii pierwszej klatki na pozycji końcowej:

- włączyć zapętlanie animacji ([Loop])
- przejść na koniec animacji ('Control' + 'strzałka do góry')

- wstawić nową klatkę ([Insert KF] -> [Accept])
- jeśli nie potrzebne, wyłączyć zapętlenie animacji ([Loop])

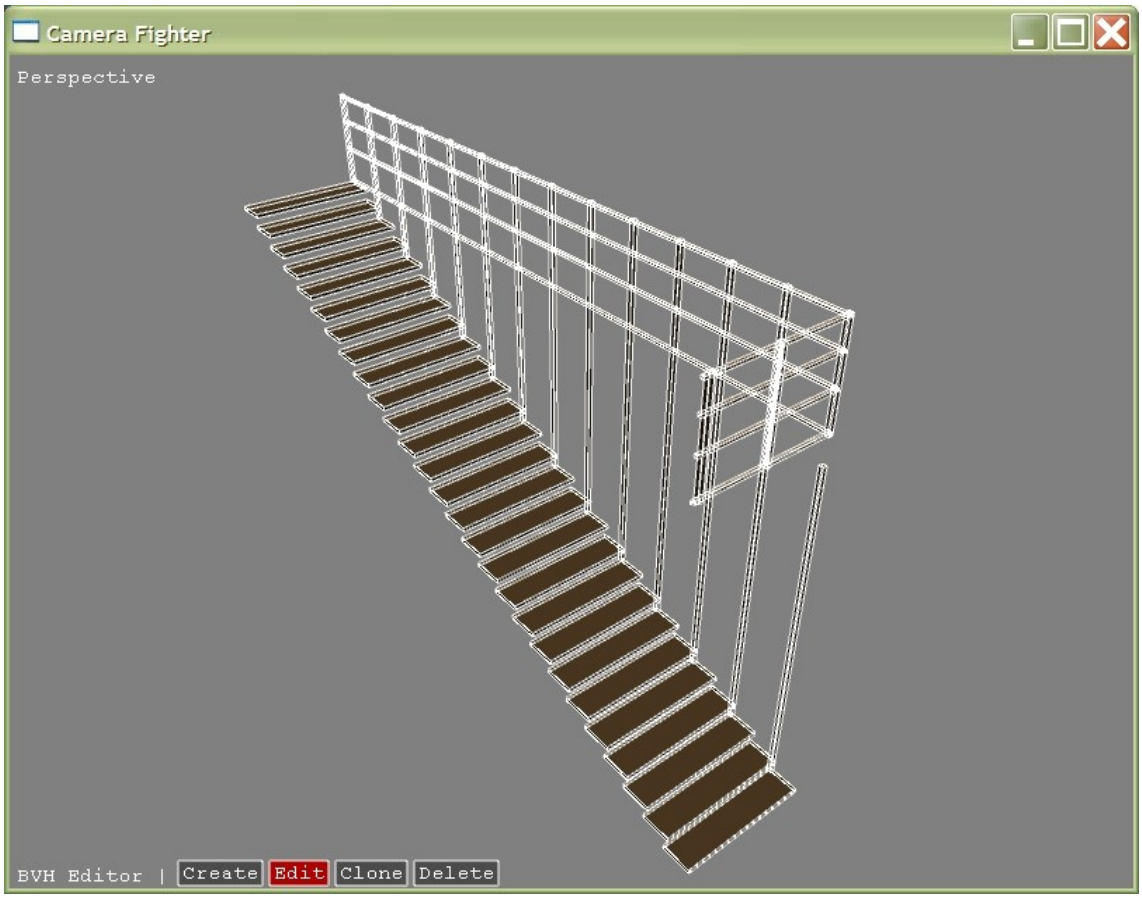
Do kopiowania klatki sąsiadującej z wstawianą klatką, można zastosować podobną sztuczkę:

- przesunąć się do początku kopiowanej klatki
- wstawić nową klatkę ([Insert KF] -> [Accept])
- ustawić czas trwania obu klatek ([KF Time])

Definiowanie BVH

Zastosowanie Hierarchii Brył Otaczających podczas detekcji kolizji ma trzy zalety – po pierwsze, umożliwia ona wczesne wykrycie braku kolizji, bez szczegółowej analizy wszystkich elementów modelu. Po drugie, detekcja kolizji dla kilku prostych figur geometrycznych jest znacznie szybsza od testowania złożonych siatek 3D. Na koniec, informacja o zderzeniu z bryłą opisującą np. całą ścianę niesie ze sobą dużo szerszy kontekst niż wiedza o zderzeniu z niewielkim trójkątem.

Gra potrafi automatycznie wygenerować Hierarchię Brył Otaczających bazującą na sześcianach zorientowanych oraz na najniższym poziomie, na detekcji kolizji między trójkątami siatki 3D. Edytor modeli umożliwia stworzenie własnej hierarchii, która będzie bazować tylko na bryłach otaczających – daje to dużo prostszą hierarchię, która jednocześnie lepiej wpasowuje się w prawidłowy kształt elementu.



Rysunek 9: Edycja Hierarchii Brył Otaczających

Edytor posiada cztery polecenia:

- [Create] – wchodzi w tryb tworzenia nowej figury – po wyborze typu nowej figury (sfera, kapsułka, pudełko), przechodzi w tryb edycji
- [Edit] – po kliknięciu na figurze, wchodzi w tryb jej edycji
- [Clone] – po kliknięciu na figurze, tworzy jej kopię i wchodzi w tryb edycji
- [Delete] – po kliknięciu na figurze, usuwa ją

Edycja każdego typu figury wygląda nieco odmiennie.

- Sfera
  - przeciąganie myszką zmienia promień
  - przytrzymanie klawisza modyfikacji (domyślnie 'Control'), a następnie kliknięcie i przeciąganie myszką przemieszcza figurę

- Kapsułka
  - przeciąganie w okolicach centrum figury zmienia promień
  - przeciąganie w okolicach końców figury zmienia długość
  - zmiana długości przy wciśniętym drugorzędnym klawiszu modyfikacji (domyślnie 'Shift') powoduje dodatkowo obrót figury
  - przemieszczanie jak dla sfery
- Pudełko
  - przeciąganie w okolicach końców wskazanych osi powoduje zmianę szerokości w danym wymiarze
  - zmiana szerokości z wciśniętym drugorzędnym klawiszem modyfikacji (domyślnie 'Shift') powoduje dodatkowo obrót figury
  - przemieszczanie jak dla sfery i kapsułki

3.8. KONFIGURACJA

3.8.1. Wstęp

Obecnie brakuje wizualnej konfiguracji gry. Jej zachowanie można zmienić w plikach konfiguracyjnych – część z nich była wymieniona przy okazji prezentacji podsystemu z którym są powiązane. W rozdziale tym przedstawiony zostanie krótki przewodnik po plikach konfiguracyjnych gry. Wiele opcji można zmienić dynamicznie, w trakcie działania programu, szczegółowa lista poleceń dla bieżącej sceny dostępna jest w pomocy konsolowej (polecenie '?' w konsoli).

Wszystkie pliki konfiguracyjne przestrzegają następujących zasad:

- puste linie oraz linie zaczynające się od znaku '#' są ignorowane
- nierozpoznana treść linijki jest ignorowana
- główne sekcje rozpoczynają się linijką `[nazwa_sekcji]`
- gdy nazwa sekcji nie jest rozpoznawana, to wszystkie linijki, aż do kolejnej sekcji są ignorowane. Właściwość ta jest często wykorzystywana w plikach definiujących mapy, do pomijania modeli używam sekcji `[skip model]`

- treść linijki następująca po wymaganych parametrach jest pomijana, ale dla przejrzystości, poprzedzam takie komentarze znakiem '#'
  ```
  Lighting 2 # 0 = none, 1 = basic, 2 = full
  ```
- parametry tekstowe wczytywane są jako pojedyncze słowa, chyba że cały ciąg objęty zostanie cudzysłowami, aby zawrzeć cudzysłów w ciągu, należy wstawić dwa cudzysłowy obok siebie:
  ```
  name "Dluga nazwa z ""cudzyslowami"" w tresci"
  ```
- gdy jakaś opcja jest nieobecna, przyjmuje się dla niej wartość domyślną

3.8.2. Gra

Plik \Data\config.txt zwiera główne opcje wpływające na zachowanie gry. W domyślnej wersji tego pliku znajduje sie lista wszystkich dostępnych opcji. Konfiguracja podstawowa dzieli się na dwie sekcje – `[graphics]` i `[general]`.

Sekcja graficzna

- `lighting` [0/1/2] – konfiguracja renderowania świateł
  - `0` – brak świateł
  - `1` – jedno światło w nieskończoności
  - `2` – światła zdefiniowane w pliku definiującym mapę
- `shadows` [0/1/2] – czy renderować cienie
  - `0` – brak cieni
  - `1` – cienie są rzucane przez ciała sztywne
  - `2` – cienie są rzucane przez ciała sztywne oraz graczy
- `shaders` [0/1] – czy korzystać z programów po stronie GPU
- `multisampling` [0/1/2/...] – poziom antyaliasingu (będzie pomnożony *2), niektóre stare karty mają problem z renderowaniem tekstu, gdy włączymy multisampling.
- `useVBO` [0/1] – czy renderować z wykorzystaniem rozszerzenia VBO (jeśli jest dostępne) – pomocne gdy VBO generuje błędy (zdarzyło mi się to na jednym laptopie)

- `shadowMap` [2^n] – wielkość tekstury dla teksturowych cieni (shadow maps), w obecnej wersji wszystkie cienie są rzucane objętościowo (shadow volumes) i parametr ten nie ma znaczenia
- `vSync` [0/1] – czy karta ma oczekiwać na synchronizację pionową monitora – warto wyłączyć by przekonać się o faktycznej wydajności programu
- `windowX` [liczba] – początkowa szerokość okna aplikacji
- `windowY` [liczba] – początkowa wysokość okna aplikacji
- `fullScreen` [0/1] – czy gra ma rozpocząć się w trybie pełnoekranowym (można przełączyć dynamicznie klawiszem F11)
- `fullScreenX` [liczba] – szerokość aplikacji pełnoekranowej (musi być obsługiwana przez kartę graficzną)
- `fullScreenY` [liczba] – wysokość aplikacji pełnoekranowej (musi być obsługiwana przez kartę graficzną)
- `show_ShadowVolumes` [0/1] – czy pokazywać obrys cieni objętościowych
- `show_Skeleton` [0/1] – czy pokazywać szkielet postaci
- `show_BVH` [0/1] – czy pokazywać najniższy poziom BVH
- `show_Cameras` [0/1] – czy pokazywać symboliczne obrysy kamer

Sekcja ogólna

- `console` [0/1] – czy konsola ma być dostępna
- `scene` [menu/game/test] – wybór sceny początkowej
  - `menu` – po uruchomieniu gry zobaczymy menu
  - `game` – gra od razu uruchomi się w trybie rozgrywki, pokazując mapę z pliku \Dane\models\level_*.map, gdzie * zdefiniowana jest przez opcję `level`
  - `test` – uruchomiona zostanie scena testująca BVH (zmiana figur: 0-9)
- `level` [liczba] – początkowy poziom scen `game` i `test`
- `speed` [liczba] – mnożnik zmiennoprzecinkowy prędkość rozgrywki

- `logging` [0/1/2/3] – poziom logowania błędów do pliku /Data/log.txt
  - 0 – prawie nic nie jest raportowane
  - 1 – logowane są ważne błędy
  - 2 – logowane są wszystkie błędy
  - 3 – logowane są wszystkie błędy oraz ostrzeżenia
- `3dsTo3dx` [0/1] – czy napotkane w mapach modele 3ds mają być automatycznie zapisywane do plików 3dx – włączenie grozi nadpisaniem istniejących modeli

3.8.3. Klawiatura

Plik \Data\keyboard.txt zwiera mapowanie klawiatury na kody funkcyjne. W pierwszej kolumnie znajduje się nazwa klawisza zdefiniowana w pliku \Data\keys.txt (nazwa klawisza nie może zawierać spacji). W drugiej kolumnie podaje się numeryczny kod funkcyjny – powiązanie tych numerów z nazwami można znaleźć w pliku \Source Files\InputCodes.h.

3.8.4. Kamery

W pliku \Data\cameras.txt znajduje się konfiguracja widoku z kamer podczas rozgrywki. Domyślnie jest to podział na trzy ujęcie – od boku, od góry oraz z oczu pierwszego gracza. W plikach \Data\cameras_*.txt znajdują się alternatywne ustawienia kamer.

Plik konfiguracyjny może zawierać dowolną ilość sekcji `[camera human]` lub `[camera free]`. Napotkanie jednej z tych sekcji rozpoczyna definicję nowej kamery podanego typu:

- `name` [tekst] – nazwa ujęcia
- `eye` [x y z] – pozycja obserwatora
- `center` [x y z] – pozycja punktu obserwacji
- `up` [x y z] – wektor wskazujący górę ekranu
- `speed` [liczba] – zmiennoprzecinkowa szybkość śledzenia kamery

- `(viewport)` – rozpoczyna podsekcję definiującą okno docelowe
  - `left` [0-100] – procent szerokości okna gdzie zaczyna się widok
  - `top` [0-100] – procent wysokości okna gdzie zaczyna się widok
  - `width` [0-100] – szerokości widoku
  - `height` [0-100] – wysokość widoku
- `(fov)` – rozpoczyna podsekcję definiującą rzut, ustawia typ rzutu na prostopadły
  - `angle` [stopnie] – kąt widzenia kamery, zmienia rzut na perspektywiczny
  - `front` [liczba] – odległość bliskiej granicy widzenia
  - `back` [liczba] – odległość dalekiej granicy widzenia
- `(track eye)`, `(track center)` – rozpoczyna podsekcję definiującą automatyczne śledzenie przez dany punkt
  - `mode` [tekst] – ustawia tryb śledzenia
    - `nothing` – wyłącza śledzenie
    - `object` – śledzenie wybranego obiektu
    - `subobject` – śledzenie wybranego podobiektu
    - `all_center` – śledzenie środka wszystkich dostępnych elementów
    - `nazwa skryptu` – śledzenie z wykorzystaniem określonego skryptu – obecnie dostępne są tylko wbudowane skrypty:
      - `EyeSeeAll_Center` – skrypt obserwatora – stara się objąć polem widzenia wszystkie dostępne obiekty (od boku). Za centrum widoku przyjmuje średnią ze środków wszystkich obiektów
      - `EyeSeeAll_CenterTop` – modyfikacja poprzedniego skryptu pozwalająca na widok z góry
      - `EyeSeeAll_Radius` – skrypt obserwatora – stara się objąć polem widzenia wszystkie dostępne obiekty, za centrum przyjmuje środek bryły otaczającej wszystkie obiekty

- `object` – numer śledzonego obiektu
- `subobj` – numer śledzonego podobiektu (dla śledzonego obiektu)
- `shift` – przesunięcie punktu docelowego w układzie współrzędnych zależnym od skryptu

3.8.5. Lista map

Plik \Data\maps.txt przechowuje listę map dostępnych do wyboru w menu. Składa się on z szeregu sekcji `[map]`. Każda sekcja może zawierać właściwości:

- `name` [tekst] – wyświetlana nazwa mapy
- `file` [ścieżka] – ścieżka dostępu do mapy
- `img` [ścieżka] – ścieżka dostępu do zdjęcia mapy

3.8.6. Lista graczy

Postacie dostępne do wyboru w menu są wymienione w pliku \Data\players.txt. Plik ten dzieli się na sekcje `[player]`. Każda z nich powinna definiować następujące właściwości:

- `name` [tekst] – wyświetlana nazwa modelu
- `model` [ścieżka] – plik z modelem graficznym
- `fastm` [ścieżka] – plik z uproszczonym modelem fizycznym
- `customBVH` [0/1] – czy ma być stosowana własna, czy automatyczna hierarchia brył otaczających – dla graczy lepiej sprawdza się automatyczna
- `mass` [liczba] – masa modelu
- `style` [tekst ścieżka]– nazwa oraz plik definiujący dostępny dla tego modelu styl walki (powtórzenie tej właściwości doda kolejne style)

Poza wymienionymi powyżej opcjami, dostępne są również inne właściwości dostępne dla modeli definiowanych w plikach z mapami. Większość z nich nie ma jednak sensu w tym kontekście (np. wyłączenie fizyki), bądź zostanie nadpisana przez opcje wybrane w menu.

3.8.7. Styl walki

Przykładowy styl walki dostępny jest w pliku \Data\models\anim\karate.txt. Plik taki powinien zawierać trzy typy sekcji – `[auto]`, `[mirror]` oraz `[action]`.

Sekcja "auto"

Sekcja ta zawiera listę akcji, które model może podjąć bez udziału gracza, by przemieścić się do punktu docelowego:

- `stop` [akcja] – wskazanie akcji spoczynkowej
- `step` [akcja dystans 0/1] – podana akcja przemieści gracza do przodu o podany dystans, flaga na końcu mówi, czy akcja może zostać przerwana, jeśli cel zostanie osiągnięty przed jej zakończeniem
- `back` [akcja dystans 0/1] – podana akcja przemieści gracza do tyłu o podany dystans, flaga na końcu mówi, czy akcja może zostać przerwana, jeśli cel zostanie osiągnięty przed jej zakończeniem
- `left` [akcja kąt 0/1] – podana akcja obróci gracza w lewo o podany kąt (w stopniach), flaga na końcu mówi, czy akcja może zostać przerwana, jeśli cel zostanie osiągnięty przed jej zakończeniem
- `right` [akcja kąt 0/1] – podana akcja obróci gracza w prawo o podany kąt (w stopniach), flaga na końcu mówi, czy akcja może zostać przerwana, jeśli cel zostanie osiągnięty przed jej zakończeniem

Sekcja “mirror”

Sekcja ta zawiera listę kości, które są lustrzanym odbiciem, np. lewa dłoń z prawą dłonią. Dane te umożliwiają stosowanie tej samej animacji dla obu stron ciała:

- `bones` [liczba liczba]– dodaje parę podanych kości do listy odbić

Sekcja “action”

Sekcja ta definiuje pojedynczą akcję, jaką może podjąć model. Zastosowanie wskazanych kombinacji klawiszy pozwala na przechodzenie między akcjami. Wszystkie czasy trwania są podawane w milisekundach.

- `name` [tekst] – nazwa akcji
- `anim` [ścieżka czas_rozpoczęcia czas_zakończenia] – plik z animacją jaka ma zostać odtworzona w podanej akcji. Zdefiniowanie kilku animacji połączy je w jedną
- `time` [liczba] – czas trwania animacji, 0 dla zapętlenia
- `post` [akcja] – następna akcja do wykonania (domyślnie jest to pierwsza zdefiniowana akcja)
- `rotate` [0/1] – akcja powoduje obrót – po jej zakończeniu macierz modelu zostanie obrócona o różnicę początkowego i końcowego skrętu pierwszej kości (domyślnie 0)
- `mirror` [0/1] – akcja powoduje odwrócenie pozycji – po jej zakończeniu wszystkie animacje będą odbijane w lustrze (domyślnie 0)
- `(combo)` – rozpoczyna podsekcję definiującą możliwe przejście do innej akcji
  - `action` [akcja] – akcja docelowa
  - `key` [nazwa] – przycisk powodujący przejście do kolejnej akcji, dostępne przyciski to: `Forward`, `Backward`, `*`, `*Punch`, `*Kick`, `*HandGuard`, `*LegGuard`. `*` = `Left` / `Right`

- `first` [liczba] – czas otwarcia okna w którym można wykonać kombinację (domyślnie 0)
- `last` [liczba] – czas zamknięcia okna w którym można wykonać kombinację (domyślnie 0 = nieskończoność)
- `time` [liczba] – moment rozpoczęcia docelowej akcji, jeśli czas zamknięcia okna kombinacji jest różny od zera, to do wartości tej zostanie dodany czas jaki upłynął od otwarcia okna kombinacji (domyślnie 0)
- `prec` [0/1] – jeśli 1, to do `time` nie będzie dodawany czas jaki upłynął od otwarcia okna kombinacji (domyślnie 0)
- `shift` [0/1] – jeśli 1, to macierz pozycji postaci zostanie przemieszczona w momencie wykonania kombinacji o przesunięcie kości podstawowej

3.8.8. Mapa

Przykładowa mapa dostępna jest w pliku \Data\models\level3.map. Definicja mapy składa się z czterech rodzajów sekcji – `[general]`, `[light]`, `[model]`, `[person]`.

Sekcja główna

Sekcja `[general]` definiuje globalne ustawienia mapy. Można w niej również wskazać inne mapy, które powinny zostać dołączone do bieżącej mapy.

- `import` [ścieżka] – ścieżka do dodatkowej mapy, która powinna zostać wczytana
- `skybox` [ścieżka] – ścieżka do modelu 3D, który będzie służył jako niebo
- `skycolor` [r g b] – kolor nieba, wartości kolorów w przedziale [0.0, 1.0]
- `spawn1rot` [x y z] – obrót pierwszego gracza
- `spawn1pos` [x y z] – pozycja pierwszego gracza
- `spawn2rot` [x y z] – obrót drugiego gracza
- `spawn2pos` [x y z] – pozycja drugiego gracza

Oświetlenie

Sekcja `[light]` definiuje pojedyncze światło.

- `type` [nazwa] – typ światła
  - `infinite` – światło nieskończone
  - `point` – światło punktowe
- `state` [1/0] – czy światło jest włączone
- `position` [x y z] – pozycja światła punktowego
- `direction` [x y z] – kierunek światła nieskończonego
- `color` [r g b] – kolor światła, wartości kolorów w przedziale [0.0, 1.0]
- `softness` [0.0-1.0] – miękkość rzucanych cieni
- `spot_dir` [x y z] – kierunek snopa światła
- `spot_cut` [kąt] – szerokość snopa światła
- `spot_att` [liczba] – tłumienie snopa światła
- `att_const` [liczba] – współczynnik stałego tłumienia światła
- `att_linear` [liczba] – współczynnik liniowego tłumienia światła
- `att_square` [liczba] – współczynnik kwadratowego tłumienia światła

Obiekty

Sekcja `[model]` definiuje pojedynczy model znajdujący się na scenie. Jedyną wymaganą właściwością jest ścieżka do pliku z modelem.

- `name` [tekst] – nazwa modelu
- `model` [ścieżka] – plik z modelem graficznym
- `fastm` [ścieżka] – plik z uproszczonym modelem fizycznym
- `customBVH` [0/1] – czy ma być stosowana własna, czy automatyczna hierarchia brył otaczających – dla postaci lepiej sprawdza się automatyczna, dla martwych obiektów własna
- `position` [x y z] – pozycja modelu – kolejne wpisy będą ze sobą łączone

- `rotation` [x y z] – obrót modelu – kolejne wpisy będą ze sobą łączone
- `velocity` [x y z] – prędkość we wskazanym kierunku, `position` i `rotation` zerują prędkość
- `physical` [0/1] – czy obiekt podlega ciążeniu
- `locked` [0/1] – czy obiekt jest stacjonarny (czy nie może się przemieszczać)
- `mass` [liczba] – masa obiektu
- `restitution` [liczba] – współczynnik odbicia – jaki procent energii zostanie przekazane drugiemu obiektowi
- `restitution_self` [liczba] – współczynnik odbicia – ile energii zostanie zachowane
- `shadows` [0/1] – czy obiekt rzuca cienie

W sekcji `[person]` można użyć dowolne właściwości sekcji `[model]`, ponadto dochodzą następujące parametry:

- `animation` [ścieżka czas_rozpoczęcia czas_zakończenia] – plik z animacją jaka ma być odtwarzana dla modelu (czas podany w ms)
- `style` [tekst ścieżka]– nazwa oraz plik definiujący dostępny dla tego modelu styl walki
- `control` [nazwa] – typ kontroli nad obiektem
  - `camera` – obiekt kontrolowany kamerą
  - `network` – obiekt kontrolowany przez sieć (projekt)
  - `comboard` – obiekt kontrolowany klawiaturą
  - `ai` – obiekt kontrolowany AI
- `enemy` [liczba] – przeciwnik na którym postać będzie skupiała uwagę

4. SILNIK GRAFICZNY

4.1. WSTĘP

Silnik graficzny jest jednym z głównych podsystemów gry. Silnik taki powinien być bardzo elastyczny i wydajny. Stworzenie uniwersalnego silnika pozwala na jego ponowne wykorzystanie w kolejnym produkcie, co znacznie skraca czas i zmniejsza koszty wydawania kolejnych tytułów. Jednakże nawet najbardziej elastyczny silnik na nic się zda, jeśli nie będzie wydajny. Gracze oczekują coraz większej szczegółowości oglądanych scen, poza tym gra to nie tylko obrazy, ale i równie wymagające obliczeniowo fizyka oraz sztuczna inteligencja. Poza dostarczeniem pięknych widoków trzeba jeszcze dać programistom pole do popisu w zakresie logiki gry. Silnik graficzny musi zostawić pozostałym podsystemom jak najwięcej czasu na obliczenia. Często większym powodzeniem cieszą się gry prostsze graficznie, lecz skupiające się na zapewnieniu maksymalnej grywalności.

Obecnie na rynku istnieje wielu producentów skupiających się na tworzeniu tylko jednego podsystemu gry. Można znaleźć niezależne silniki graficzne oraz fizyczne, a także elementy wyspecjalizowane w wąskiej dziedzinie – np. skupiające się tylko na renderowaniu drzew. Takie rozproszone tworzenie kodu ma wiele zalet. Twórcy gry mogą się skupić na jej logice – powtarzalne elementy, takie jak proces renderowania mają już gotowy. Z jednej strony gra może zostać ukończona dużo szybciej, po drugie, taki zewnętrzny silnik tworzony jest przez ekspertów w danej dziedzinie. Możemy być pewni, że silnik taki jest dopracowany w najmniejszych szczegółach, a autorzy stale rozszerzają jego możliwości. Po wpisaniu w wyszukiwarce internetowej hasła 'graphics engine', na pierwszych pozycjach pojawiają się otwarte silniki graficzne `[OGRE3D]` oraz `[Horde3D]`. Łatwo zgadnąć czemu tak jest – wykorzystanie komercyjnego silnika jest niesamowicie drogie (wg. informacji serwisu www.fudzilla.com – około 400 tysięcy dolarów!), więc znajdują się poza obszarem zainteresowań większości programistów.

Porównując cechy silników OGRE 3D oraz Horde3D widać nieco odmienne podejście do tematu. Obie biblioteki, dla maksymalnej przenośności opierają się na OpenGL, choć OGRE dodatkowo wspiera Direct3D.

Biblioteka OGRE skupia się na zapewnieniu szerokiej kompatybilności – sposób renderowania dostosowuje się do sprzętu, na jakim uruchomiono program. W przypadku starego sprzętu, silnik wspiera się przetwarzaniem po stronie CPU – przykładem może być np. animacja szkieletowa, która na nowszym sprzęcie może zostać przetworzona przez GPU. OGRE umożliwia również definiowanie wielu wersji tego samego materiału – biblioteka automatycznie dobierze wersję optymalną dla sprzętu na którym będzie pracować.

Dla odmiany, silnik Horde3D stawia na szybkość kosztem kompatybilności. Przeznaczony jest on dla najnowszego sprzętu, który obsługuje minimum Shader Model 2.0. Może to być bardzo trafiony wybór, gdyż obecne gry są tak wymagające, iż posiadacze starszego sprzętu i tak mogą zapomnieć o próbach ich uruchomienia. Ponadto Shader Model jest standardem kart graficznych od wprowadzenia Dierect3D w wersji 9.0, więc większość potencjalnych graczy zdążyła zaktualizować swój sprzęt.

Oba wymienione silniki zapewniają oczywiście szereg standardowych funkcjonalności, jak obsługa tekstur w najpopularniejszych formatach, pomijanie niewidocznych obiektów czy wsparcie zmiennej szczegółowości renderowanych modeli.

Zastosowanie gotowego silnika jest dobrym rozwiązaniem dla profesjonalistów, którzy mają pojęcie o zasadach ich działania i mogą w pełni wykorzystać ich potencjał. Dla mnie posunięcie takie było kuszące, ale nie niosło jednak wielkiej wartości dydaktycznej – a przecież moim głównym celem podczas tworzenia gry Camera Fighter było dogłębne poznanie procesu tworzenia oraz funkcjonowania gier. Z tego powodu postanowiłem stworzyć od podstaw własny silnik graficzny. Zrozumiałe jest, że odbiega on jakością od rozwiązań tworzonych przez całe zespoły doświadczonych w tej dziedzinie programistów, lecz umożliwił on przeanalizowanie funkcjonowania każdego elementu nowoczesnego silnika graficznego. Silnik ten opiera się na bibliotece

OpenGL, dzięki czemu uzyskałem przenośność między systemami operacyjnymi. Szkielet silnika starałem się jednak tworzyć tak, by stosunkowo łatwo można go było rozszerzyć o obsługę Direct3D, bądź innych systemów renderujących.

4.2. MENADŻER TEKSTUR

Tekstury są jednym z kosztowniejszych pamięciowo elementów grafiki. Dobrej jakości tekstura może zajmować nawet kilka megabajtów. Dlatego ważne jest, aby tekstury współdzielone przez różne występowały w pamięci tylko w jednej instancji. Temu celowi służy menadżer zasobów `TextureMgr`. Klasa ta bazuje na typach `Singleton` oraz `Manager` przedstawionych w rozdziale poświęconym architekturze aplikacji. Jej kod jest niezwykle prosty, gdyż niemal całą logikę implementują klasy bazowe. Menadżer ten przechowuje zasoby typu `Graphics::OGL::Texture`, które odpowiedzialne są za wczytanie plików z dysku oraz interakcję z biblioteką OpenGL. Jeśli system graficzny unieważni teksturę, menadżer automatycznie odtworzy ją przy kolejnym odwołaniu, dzięki czemu kod renderujący zawsze otrzymuje poprawne dane.

```
class TextureMgr;
#define g_TextureMgr TextureMgr::GetSingleton()
typedef Handle <TextureMgr> HTexture;

class TextureMgr : public Singleton<TextureMgr>,
           public Manager<Graphics::OGL::Texture, Htexture> {
public:
HTexture GetTexture   ( const char* name );
```

zwraca uchwyt do wskazanej tekstury (niepotrzebny uchwyt musi zostać zwolniony odziedziczoną metodą `Release()`)

```
const string& GetName( HTexture htex ) const;
```

zwraca nazwę (ścieżkę) wskazanej tekstury

```
int GetWidth( HTexture htex ) const;
```

zwraca szerokość tekstury

```
int GetHeight( HTexture htex ) const;
```

zwraca wysokość tekstury

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

ustawia teksturę jako bieżącą

```
};
```

4.3. MENADŻER CZCIONEK

Biblioteka OpenGL nie udostępnia metod wyświetlających tekst na ekranie. Istnieje jednak kilka metod umożliwiających renderowanie informacji tekstowych ([GL_Fonts]).

Pierwszą możliwością jest stworzenie tekstury zawierającej wszystkie wymagane znaki, a następnie dla każdej litery renderowanie fragmentu tekstury. Zaletą tego rozwiązania jest niezależność od systemu operacyjnego.

Innym rozwiązaniem jest użycie dynamicznych bitmap (`glBitmap()`). Zarówno środowisko Windows jak i X11 udostępniają funkcje, które dynamicznie wygenerują mapy bitowe dla wskazanego typu i rozmiaru czcionki. W przypadku systemu Windows mamy dodatkowo możliwość generowania czcionek trójwymiarowych. Mapy takie kompilowane są do sąsiadujących listy poleceń OpenGL, dzięki czemu całą linijkę tekstu można wyświetlić pojedynczym wywołaniem funkcji `glCallLists()`. Rozwiązanie to wymusza stworzenie oddzielnego kodu dla Windowsa i Linuxa, jednakże daje możliwości dynamicznego wyboru pożądanych czcionek.

Podejście jakie przyjąłem opiera się na czcionkach generowanych dynamicznie według opisu 13 lekcji na stronie [NeHe]. Aby różne podsystemy gry mogły współdzielić czcionki, a także by scentralizować zarządzanie cyklem życia czcionek, stworzyłem klasę zarządzającą `FontMgr`. Podobnie jak inne menadżery zasobów, jest to klasa typu `Singleton` opierająca się na typie `Manager`. Zasób jakim zarządza menadżer to `Graphics::OGL::Font`.

```
class FontMgr;
typedef Handle <FontMgr> HFont;
#define g_FontMgr FontMgr::GetSingleton()

class FontMgr : public Singleton<FontMgr>,
                public Manager<Graphics::OGL::Font, HFont> {
public:
```

```
HFont GetFont ( const char* name, int size );
```
zwraca uchwyt do wskazanej czcionki (niepotrzebny uchwyt musi zostać zwolniony odziedziczoną metodą `Release()`)

```
const string& GetName( HFont hfnt ) const;
```
zwraca nazwę wskazanej czcionki

```
int GetSize( HFont hfnt ) const;
```
zwraca rozmiar wskazanej czcionki

```
const Graphics::OGL::Font* GetFont( HFont hfnt ) const;
```
zwraca wskaźnik bezpośrednio do zasobu czcionki

```
};
```

Podsystemy korzystające z czcionek najczęściej mają do wyświetlenia wiele linijek tekstu – konsola może zapełniać treścią nawet cały ekran. Aby zwiększyć wydajność renderowania, postanowiłem umożliwić wyświetlanie tekstu poprzez bezpośrednie wołanie metod zasobu. Jeśli aktualny wskaźnik na zasób będzie pobierany na początku każdej serii poleceń wyświetlających tekst, to nie ma groźby, że będzie nieważny. Pozwala to na uniknięcie wielu zbędnych procesów wyszukiwania i walidacji czcionki przez menadżera.

```
struct Font : public Resource {
public:
string Name;
```
nazwa czcionki

```
int Size;
```
rozmiar czcionki

```
float LineH() const;
```
wysokość linijki tekstu (rozmiar + interlinia)

```
virtual bool Create( const std::string& name, int size = 14);
```
tworzy czcionkę o podanej nazwie i rozmiarze

```
void Print (float x, float y, float z, float maxHeight,
            int skipLines, const char *text) const;
```
wypisuje we wskazanym miejscu podany tekst 2D. Tekst może zajmować wiele linijek – metoda pominie `skipLines` pierwszych linii, a następnie będzie

renderować tekst do momentu osiągnięcia końca tekstu bądź zajęcia `maxHeight` jednostek (każda linijka zajmuje `LineH()` jednostek)

```
void Print (float x, float y, float z,
            const char *text) const;
```

wypisuje we wskazanym miejscu podany tekst 2D

```
void PrintF  (float x, float y, float z,
              const char *fmt, ...) const;
```

wypisuje we wskazanym miejscu podany tekst 2D, formatowanie tekstu jest zgodne ze specyfikacją metody `printf()`

```
void Print   (const char *text) const;
```

wypisuje podany tekst 2D na bieżącej pozycji rastra (`glRasterPos*()`)

```
void PrintF  (const char *fmt, ...) const;
```

wypisuje podany tekst 2D na bieżącej pozycji rastra (`glRasterPos*()`), formatowanie tekstu jest zgodne ze specyfikacją metody `printf()`

```
float Length (const char *text) const;
```

zwraca długość podanego tekstu

```
virtual void Clear();
virtual bool Create();
virtual void Dispose();
virtual void Invalidate();
virtual bool IsDisposed();
virtual const string &Identifier();
```

opis tych metod znajduje się przy opisie klasy `Resource` w rozdziale 3.4.5.

```
};
```

4.4. ROZSZERZENIA OPENGL

Od czasu zaprezentowania pierwszej wersji OpenGL minęło już 16 lat. W między czasie ukazało się 7 rewizji specyfikacji. Każda kolejna wersja udostępnia nowe narzędzia. W dużej mierze ich inicjatorami są producenci kart graficznych – gdy stworzą funkcjonalność specyficzną dla ich produktu, często jest ona po jakimś czasie wprowadzana do standardu. Nim jednak to się stanie, jest ona dostępna dla programistów w formie rozszerzeń. Dzięki temu programiści mogą korzystać z pełnych

możliwości sprzętu, a pozostali producenci mają czas na zaimplementowanie ich w swoich produktach – by dosłownie – stały się ogólnym standardem.

Rozszerzenia OpenGL mają jeszcze jedną zaletę – gdy tworzymy program w obecnym standardzie, będzie on działał tylko na najnowszym sprzęcie. Aby program działał na uboższym sprzęcie, musimy pisać w starszym standardzie. Nowe możliwości kart możemy wykorzystywać poprzez rozszerzenia i tylko wtedy gdy są dostępne.

Aby uzyskać dostęp do rozszerzenia, musimy podjąć dwa kroki. Po pierwsze, na etapie programowania, musimy dostarczyć pliki nagłówkowe, które definiują nowe stałe oraz wskaźniki do nowych funkcji. Kolejny krok wykonywany jest podczas działania programu – jeśli karta graficzna udostępnia pożądane rozszerzenie, to możemy pobrać od OpenGL wskaźniki do interesujących nas funkcji i wykorzystać je w dalszym kodzie.

Całą inicjalizację rozszerzeń zebrałem w jednej klasie o nazwie `GLExtensions`:

```
struct GLExtensions {
static bool Exists_ARB_ShaderObjects;
static bool Exists_ARB_VertexShader;
static bool Exists_ARB_FragmentShader;
static bool Exists_ARB_VertexBufferObject;
static bool Exists_EXT_StencilWrap;
static bool Exists_EXT_StencilTwoSide;
static bool Exists_ARB_Multisample;
```

flagi sygnalizujące dostępność danego rozszerzenia

```
static bool Initialize();
```

inicjalizuje rozszerzenia (tylko te, które wykorzystuję w grze)

```
static bool Exists(const char *extensionName);
```

sprawdza czy istnieje podane rozszerzenie

```
static void SetVSync(bool enable);
```

ustawia oczekiwanie na synchronizację pionową

```
};
```

4.5. RENDEROWANIE MODELI 3DX

4.5.1. Wstęp

Podstawowym elementem gry Camera Fighter jest model typu xModel. Na jego potrzeby stworzono wyspecjalizowaną klasę `RendererGL`, która odpowiedzialna jest za renderowanie tego typu obiektów. Klasa ta udostępnia szereg poleceń renderujących oraz metody zarządzające danymi modelu, które są przechowywane w pamięci karty graficznej. Wszystkie sposoby renderowania, za wyjątkiem metod specjalnego zastosowania, renderujących szkielet, BVH oraz wierzchołki, pomijają renderowanie elementów, które znajdują się poza obszarem widzenia.

```
class RendererGL : public Renderer {
public:
bool UseVBO;
```

flaga mówiąca czy korzystać z VBO

```
bool UseList;
```

flaga mówiąca czy korzystać z list OpenGL (gdy nie korzysta się z VBO)

```
virtual void RenderModel( xModel &model,
                  xModelInstance &instance, bool transparent,
                  const FieldOfView &FOV );
```

jednoprzebiegowe renderowanie modelu – renderowane będą tylko przeźroczyste bądź nieprzeźroczyste trójkąty znajdujące się w obszarze widzenia

```
virtual void RenderDepth( xModel &model,
                  xModelInstance &instance, bool transparent,
                  const FieldOfView &FOV );
```

renderowanie do bufora głębokości dla techniki “depth first rendering” – renderowane będą tylko przeźroczyste bądź nieprzeźroczyste trójkąty znajdujące się w obszarze widzenia

```
virtual void RenderAmbient ( xModel &model,
                  xModelInstance &instance,
                  const Vec_xLight &lights, bool transparent,
                  const FieldOfView &FOV );
```

renderowanie modelu oświetlonego światłami otoczenia (“ambient pass”) – renderowane będą tylko przeźroczyste bądź nieprzeźroczyste trójkąty znajdujące się w obszarze widzenia

```
virtual void RenderAmbient ( xModel &model,
                  xModelInstance &instance,
                  const xLight &light, bool transparent,
                  const FieldOfView &FOV );
```

renderowanie modelu oświetlonego pojedynczym światłem otoczenia (“ambient pass”) – renderowane będą tylko przeźroczyste bądź nieprzeźroczyste trójkąty znajdujące się w obszarze widzenia

```
virtual void RenderDiffuse ( xModel &model,
                  xModelInstance &instance,
                  const xLight &light, bool transparent,
                  const FieldOfView &FOV );
```

renderowanie modelu oświetlonego światłem bezpośrednim (“diffuse & specular pass”) – renderowane będą tylko przeźroczyste bądź nieprzeźroczyste trójkąty znajdujące się w obszarze widzenia

```
virtual void RenderShadowVolume ( xModel &model,
                  xModelInstance &instance,
                  xLight &light,
                  const FieldOfView &FOV );
```

renderowanie cieni objętościowych rzucanych przez wskazane światło

```
virtual void RenderShadowMap ( xModel &model,
                  xModelInstance &instance,
                  const xShadowMap &shadowMap,
                  const FieldOfView &FOV );
```

renderowanie modelu pokrytego wskazaną mapą cieni

```
virtual void CreateShadowMapTexture ( xModel &model,
                  xModelInstance &instance,
                  xDWORD &shadowMapTexId,
                  xWORD width, xMatrix &mtxBlockerToLight );
```

utworzenie mapy cieni

```
virtual void RenderSkeleton ( xModel &model,
                  xModelInstance &instance,
                  xWORD selBoneId = xWORD_MAX );
```

wyświetlenie szkieletu, wskazana kość będzie podświetlona innym kolorem

```
virtual void RenderVertices ( xModel &model,
                  xModelInstance &instance,
                  SelectionMode selectionMode = smNone,
                  xWORD selElementId = xWORD_MAX,
                  vector<xDWORD> * selectedVertices = NULL );
```

wyświetlenie wierzchołków modelu, może być również wykorzystana do wyrenderowania nie pokrytego materiałami modelu (optymalizacja renderingu w trybie zaznaczania)

```
virtual void RenderFaces ( xModel &model,
                  xModelInstance &instance,
                  xWORD selectedElement,
                  vector<xDWORD>  * facesToRender );
```

testowa metoda renderująca kolejne trójkąty, pokryte losowym kolorem

```
virtual void RenderSkeletonSelection ( xModel &model,
                  xModelInstance &instance,
                  bool selectConstraint = false );
```

rendering szkieletu dla trybu zaznaczania, należy wskazać, czy zaznaczyć chcemy kość, czy ograniczenie kości

```
virtual void InvalidateGraphics(xModel &model,
                  xModelInstance &instance);
```

unieważnia informacje o zasobach przechowywanych po stronie karty graficznej

```
virtual void FreeGraphics(xModel &model,
                  xModelInstance &instance,
                  bool freeShared = true);
```

zwalnia zasoby przechowywane po stronie karty graficznej, w zależności od flagi `freeShared`, zwalnia tylko indywidualne, bądź również współdzielone zasoby

```
virtual void InvalidateBonePositions(
                    xModelInstance &instance);
```

unieważnia pozycje kości w pamięci karty graficznej – wymagane do odświeżenia modelu gdy renderowany jest z wykorzystaniem list OpenGL

```
static void InitVBO (xElement *elem);
```

inicjalizuje i wypełnia danymi bufory VBO

```
static void SetMaterial(xColor color, xMaterial *mat,
                    bool toggleShader = true);
```

ustawia podany materiał w silniku OpenGL, flaga `toggleShader` mówi, czy w razie potrzeby, ma być włączony odpowiedni dla materiału program GPU

```
};
```

4.5.2. Renderowanie jedno i wieloprzebiegowe

Najbardziej oczywistym sposobem renderowania modelu, jest renderowanie jednoprzebiegowe – raz na klatkę wysyłamy do karty graficznej informacje o modelu, każdy fragment modelu, za jednym przejściem jest oświetlany wszystkimi światłami w jego pobliżu. Rozwiązanie takie jest wydajne, ale nie pozwala na przedstawianie takich efektów jak cienie czy odbicia. Aby je uzyskać musimy zastosować rendering wieloprzebiegowy.

4.5.3. Bufor głębi

Jednym z etapów renderowania wieloprzebiegowego może być renderowanie do bufora głębi ('depth first rendering'). Etap ten jest wykonywany na początku całego procesu renderowania. W jego wyniku posiadamy czysty bufor koloru, jednakże w buforze głębi znajdują się już kompletne informacje o scenie. Etap ten jest wspierany przez nowoczesne karty graficzne – proces renderowania tylko do bufora głębi wykonują kilkakrotnie szybciej, niż renderowanie z włączonym zapisem do bufora koloru.

Korzyścią z wypełnienia w pierwszej kolejności bufora głębi jest zmniejszenie ilości fragmentów które będzie musiała wyrenderować karta w kolejnych krokach. Normalnie część wyrenderowanych trójkątów zostanie zasłonięta trójkątami

renderowanymi w następnej kolejności. Na podstawie zawartości bufora głębi, mogą one być we wczesnym etapie odrzucone przez kartę graficzną.

4.5.4. Światło otoczenia

Po wypełnieniu bufora głębi możemy przejść do oświetlenia modeli światłem odbitym. Przyjąłem uproszczony model, w którym cały model jest równomiernie oświetlany wszystkimi światłami w zasięgu których się znajduje. Rodzi to pewną niedokładność, gdyż np. bardzo długa ściana będzie na całej odległości oświetlona nikłym światłem odbitym zapałki. Jednakże intensywność takiego światła odbitego będzie na tyle mała, iż uznałem, że nie będzie ona miała wielkiego znaczenia na wynikowy obraz. Zaletą przyjętego uproszczenia jest możliwość jednoprzejściowego, szybkiego wyrenderowania tego kroku dla wszystkich świateł.

4.5.5. Światło rozproszone i odblaskowe

Kolejnym krokiem jest wyrenderowanie fragmentów modelu, które są oświetlone światłem bezpośrednio na nie padającym. Każde światło renderujemy w oddzielnym przejściu, gdyż musi być one poprzedzone renderingiem cieni objętościowych. Cienie objętościowe renderowane są do bufora brudnopisu (stencil buffer), na podstawie zawartości tego bufora, karta graficzna pomija renderowanie fragmentów, na które pada cień.

4.5.6. Przeźroczyste obiekty

Gdy scena zawiera już wszystkie nieprzeźroczyste obiekty, można przejść do renderowania obiektów przeźroczystych. Przy założeniu, że obiekty przeźroczyste nie tłumią światła ani nie mogą przyjmować cieni, wystarczy dla nich renderowanie jednoprzebiegowe.

4.6. RENDEROWANIE SCENY 3D

4.6.1. Wstęp

Za rendering całej sceny odpowiedzialna jest klasa WorldRenderGL. Może ona renderować jedną z trzech technik – jednoprzebiegowo bez oświetlenia,

wieloprzebiegowo oraz uproszczenie - wieloprzebiegowo. Istnieje jeszcze jedna technika renderowania, której niestety nie zdążyłem zaimplementować i przetestować – renderowanie odroczone.

4.6.2. Renderowanie wieloprzebiegowe

Pełne renderowanie wieloprzebiegowe przebiega zgodnie z modelem przedstawionym w poprzednim podrozdziale. Wykonywane są kolejno następujące przebiegi renderowania:

- renderowanie do bufora głębi
- przebieg oświetlenia odbitego
- przebieg rzucania cieni oraz oświetlenia bezpośredniego
- renderowanie obiektów przeźroczystych

Dodatkowo można uruchomić przebiegi renderowania szkieletu, obrysów cieni oraz hierarchii BVH.

Uproszczona wersja tego algorytmu renderuje tylko jedno światło oraz pomija krok renderowania do bufora głębi.

4.6.3. Renderowanie odroczone

'Deferred shading' jest ciekawą techniką umożliwiającą jednoprzejściowe renderowanie modeli. Polega ona na zastosowaniu pewnej sztuczki – zamiast renderować do bufora docelowego, należy wykorzystać funkcję nowoczesnych kart graficznych, umożliwiającą jednoczesne renderowanie do kilku buforów pomocnicznych (MRT). Zamiast renderowania wynikowego obrazu, renderujemy do nich jednak informacje o materiałach danego fragmentu. W efekcie uzyskamy dwu wymiarowe tekstury zawierające informacje o tym, w jaki sposób powinien być oświetlony dany piksel. Renderowanie pojedynczego światła sprowadza się do wyrenderowania prostokąta pokrytego tymi teksturami i zajmującego cały ekran. Odpowiedni program GPU renderując fragmenty tego prostokąta, będzie oświetlał je na podstawie informacji zawartych w pokrywających go teksturach.

Technika ta pozwala znacznie odciążyć kartę graficzną od wielokrotnego przetwarzania skomplikowanych geometrii. Każde kolejne światło łączy się jedynie z renderowaniem jednego dodatkowego prostokąta.

4.7. PROGRAMOWANIE POD GPU

Nowoczesne karty graficzne umożliwiają przetwarzanie przez GPU prostych programów przetwarzających wierzchołki, bądź pojedyncze piksele renderowanych figur. GPU jest procesorem wyspecjalizowanym w przetwarzaniu geometrii 3D, więc zrobi to dużo efektywniej od CPU. Programy GPU pozwalają również zaimplementowanie takich efektów jak zaawansowane oświetlenie czy mapowanie wypukłości.

Współcześnie wykorzystuje się trzy języki cieniowania – GLSlang dla OpenGL, HLSL dla DirectX oraz propagowany przez firmę nVidia język Cg. Cg oraz HLSL to w rzeczywistości jeden i ten sam język.

4.7.1. GLSlang

OpenGL Shading Language jest dostępny jako standardowe rozszerzenie od wersji OpenGL 1.5. Wersja OpenGL 2.0 udostępnia go w podstawowym API. W grze Camera Fighter wykorzystywany jest poprzez rozszerzenia.

Animując postacie po stronie CPU należy w każdej klatce wysłać do karty graficznej nowe położenie wierzchołków modelu. Program animacji szkieletowej po stronie GPU operuje na bazowych pozycjach wierzchołków – jedyne co musimy robić, to aktualizować pozycję kości.

Podstawowy model oświetlenia OpenGL jest wydajny, ale bardzo uproszczony. Oświetla on wnętrze trójkątów interpolując kolor jego wierzchołków. Dla siatek składających się z małych trójkątów działa to bardzo dobrze. Problem pojawia się, gdy mamy duże trójkąty, np. fragmenty podłogi, bądź ściany. Jeśli światło nie dociera do wierzchołków, to w uproszczonym modelu, cała ściana będzie ciemna, nawet gdy źródło światła będzie bardzo blisko jej powierzchni. Jednym z rozwiązań jest podział dużych trójkątów na wiele małych – jest to jednak niepożądane, gdyż niepotrzebnie

komplikujemy geometrię modelu. Lepszym rozwiązaniem jest niezależne obliczanie oświetlenia dla każdego fragmentu figury – zdefiniowanie własnego modelu oświetlenia umożliwiają nam programy GPU.

W grze Camera Fighter wykorzystuję możliwość programowania GPU w dwóch wyżej wymienionych celach. Stworzyłem kilkanaście prostych programów, każdy odpowiedzialny jest za inny typ źródła światła (globalne, nieskończone, punktowe, snopowe). Każdy z tych programów występuje w odmianie umożliwiającej teksturowanie bądź animowanie szkieletowe modelu.

Programy znajdują się w katalogu /Data/shaders/. Ich nazewnictwo podlega następującym zasadom:

Przedrostki:

- NoLights_ – brak światła
- Global_ – globalne światło otoczenia
- Infinite_ – światło nieskończone
- Point_ – światło punktowe
- Spot_ – światło snopowe

Treść:

- _A_ – światło otoczenia
- _D_ – światło rozproszone
- _S_ – światło odblaskowe
- _DS_, _ADS_ – kombinacje powyższych

Przyrostek:

- _Tex – program pokrywa model teksturą
- _noTex – program nie pokrywa modelu teksturą
- _Tex_Skel – model jest teksturowany oraz animowany szkieletowo
- _noTex_Skel – model jest animowany szkieletowo bez teksturowania

Za animację szkieletową odpowiedzialny jest program /Data/shaders/skeletal.vert. Programy z przyrostkiem '_Skel' nie implementują same tej funkcjonalności, a jedynie odwołują się do tego pliku.

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 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 metodę a priori cechuje dużo większa dokładność i stabilność 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 testujących przecinanie się 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.

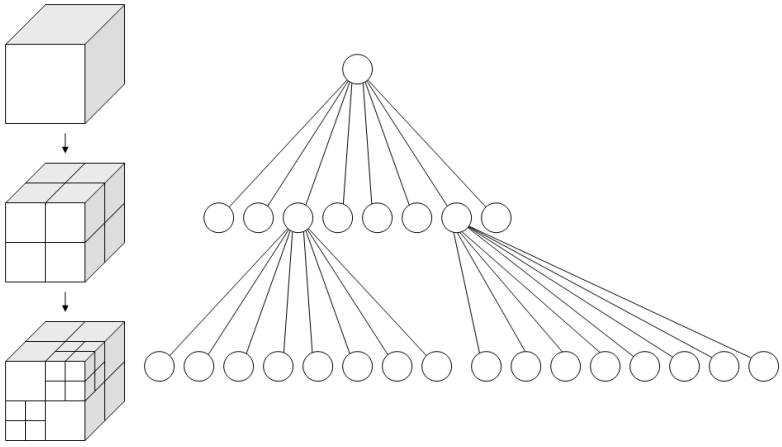
Algorytmy wykrywania przecinających się trójkątów są obecnie uproszczone do granic możliwości. Modele obiektów składają się najczęściej z co najmniej kilkuset trójkątów, a modele postaci mogą być zbudowane nawet z kilku milionów trójkątów. Jak łatwo wyliczyć, sprawdzenie kolizji zachodzącej między dwoma 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 10: Hierarchiczny podział przestrzeni (na przykładzie Drzewa Ósemkowego) – źródło Wikipedia

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

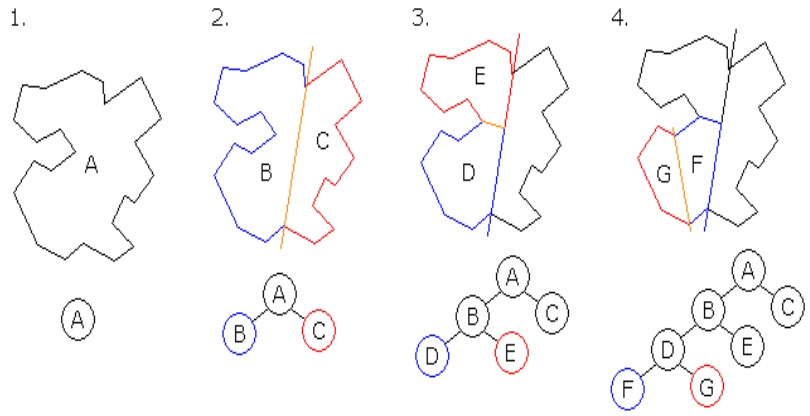
efektywne sprawdzenie czy zachodzi kolizja. Dopiero test najniższego rzędu wymaga badania kolizji między trójkątami, jednakże jest ich na tym poziomie tylko od kilku do kilkunastu dla każdego z węzłów.

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.

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 11: Drzewo Binarnego Podziału Przestrzeni – źródło Wikipedia

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.

5.1.5. Zastosowana implementacja detekcji kolizji

W grze Camera Fighter wszystkie modele posiadają hierarchię brył otaczających bazującą na klasie `xBVHierarchy`. Hierarchia ta może występować w dwóch odmianach – może być automatycznie wygenerowana bądź stworzona ręcznie w edytorze.

Automatyczny generator BVH rekurencyjnie dzieli obszar zajmowany przez model na sześciany zgodne z lokalnymi osiami modelu. Każdy taki sześcian zostaje

następnie pojedynczym węzłem hierarchii reprezentowanym przez sześcian zorientowany (Oriented Bounding Box). Musi on być sześcianem zorientowanym, gdyż osie modelu nie muszą się pokrywać z osiami świata – szczególnie gdy obiekt będzie się mógł obracać. Na najniższym poziomie automatycznie generowanej hierarchii znajduje się fragment siatki który został wykorzystany do obliczenia sześcianów otaczających daną gałąź.

Tworząc hierarchię ręcznie, do wyboru posiadamy trzy typy figur – sferę, kapsułkę (cylinder zakończony półsferami) oraz sześcian zorientowany. Cała stworzona hierarchia zostanie automatycznie otoczona sferą. Ręczne tworzona hierarchia będzie prawdopodobnie prostsza, a jednocześnie lepiej dopasowana do faktycznego kształtu modelu.

Minusem automatycznej generacji, jest to, iż na najniższym poziomie kolizje wykrywane są w oparciu o siatki modelu. Daje to teoretycznie większą dokładność detekcji, lecz przed detekcją wierzchołki muszą zostać przetransformowane do przestrzeni świata. W celu optymalizacji klasa `xMeshData` przechowująca informacje BVH siatki transformuje tylko te wierzchołki, które potrzebne są do detekcji w danej gałęzi. Co więcej, zmiana wierzchołków (np. podczas animacji szkieletowej) zmusza nas do ponownego przeliczenia hierarchii.

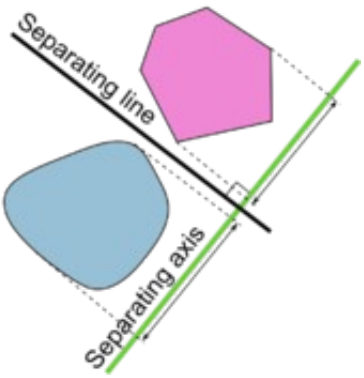
Z powodu dokładności oraz wydajności, lepszym wyborem jest ręczne tworzenie BVH, jednakże niektóre, skomplikowane obiekty trudno jest przybliżyć tymi prostymi kształtami. Próba przybliżenia kończyn człowieka kapsułkami rodziła problemy, gdy do kolizji dochodziło na łączeniach sąsiednich brył – jedna bryła sugerowała jako poprawkę przesunięcie w stronę drugiej bryły i korekcje znosiły się wzajemnie. Potencjalnym rozwiązaniem tego problemu mogło by być stworzenie nowego typu bryły – np. cylindra otwartego na końcach, który będzie wypychał kolidujące obiekty tylko w stronę swoich bocznych ścian. Jako tymczasowe rozwiązanie, musiałem w przypadku postaci pozostać przy automatycznie generowanej hierarchii bazującej na trójkątach.

Za wykrywanie kolizji dwóch obiektów odpowiedzialna jest klasa `BVHCollider`. Podczas przechodzenia po drzewie wybierane są potencjalnie kolidujące węzły drzewa. Za sprawdzenie czy doszło do kolizji między dwoma bryłami odpowiada klasa `FigureCollider`. W zależności od typu figur woła ona przystosowaną dla nich funkcję testującą.

Detekcja kolizji między dwoma sferami jest stosunkowo prosta. Wystarczy sprawdzić czy ich środki są w większej odległości od siebie niż wynosi suma ich promieni. Podobnie można przetestować kolizję między dwoma kapsułkami, bądź sferą i kapsułką, tyle że zamiast odległości od środków, należy uwzględnić odległość od odcinków leżących na osiach kapsułek.

By sprawdzić czy sfera przecina się z pudełkiem, trzeba sprawdzić odległość jej środka od ścian, krawędzi bądź wierzchołków pudełka. Najprościej dokonać tego poprzez przeniesienie centrum sfery do układu współrzędnych zdefiniowanego przez osie pudełka, a następnie uwzględniać w teście odległość w danej osi tylko wtedy, gdy wierzchołek nie znajduje między przeciwległymi ścianami pudełka. Jeśli suma wszystkich uwzględnianych odległości od ścian jest mniejsza od promienia sfery, to doszło do kolizji.

Sprawdzanie przecinania się dwóch pudełek, pudełka z kapsułką, bądź pudełka z trójkątem wydaje się bardziej złożone. Pomocna okazuje się jednak teoria osi separującej (Separating Axis Theorem – `[Wiki_SAT]`). Mówi ona, że jeśli dwa wypukłe obiekty nie przecinają się, to ich rzuty na pewną oś (zwaną osią separującą) nie będą na siebie nachodziły. Potencjalnymi osiami separującymi są wektory normalne oraz krawędzie ścian, a także ich iloczyny wektorowe. Dla dwóch pudełek musimy przetestować ich rzuty na 15 osi. Dla pudełka i trójkąta wystarczy przetestować wektory normalne pudełka, trójkąta oraz iloczyny wektorowe krawędzi trójkąta i normalnych pudełka – daje to w sumie 13 rzutowań. Trudniejszym przypadkiem jest wybór osi dla kolizji z kapsułką. Ponieważ jest ona zaokrąglona, posiada nieskończenie wiele normalnych. Jako potencjalne cechy kapsułki wybieram oś kapsułki oraz wektor łączący najbliższe punkty pudełka i kapsułki.



Rysunek 12: Oś separująca bryły – źródło Wikipedia

Z racji prostoty świata gry Camera Fighter (sceny są niewielkie i zawierają mało ruchomych obiektów), implementowanie technik podziału sceny na podprzestrzenie nie było potrzebne, więc postanowiłem skupić się na innych elementach gry.

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, należy 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.

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

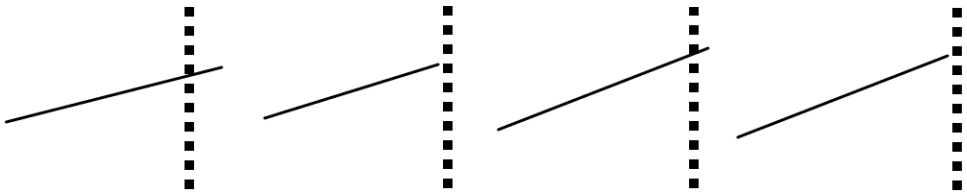
[Jakob]. Polega ono na rozbiciu obiektu na zbiór punktów materialnych i rozpatrywaniu ruchu każdego punktu osobno. Ponadto, w celu poprawy stabilności algorytmu, zrezygnowano z przechowywania prędkości cząsteczki. W efekcie każda cząsteczka opisywana jest przez cztery wielkości fizyczne: pozycję, pozycję 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 13: 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.

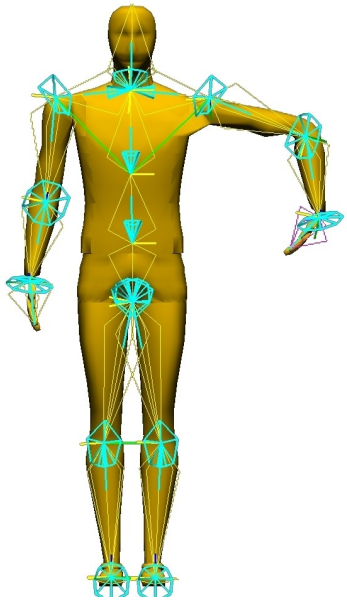
5.2.5. Symulacja ciał sztywnych

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.

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 14: 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.

5.2.7. Zastosowana implementacja fizyki

Centrum silnika fizycznego jest klasa `PhysicalWorld`. Jej głównym zadaniem jest przeprowadzanie testów zderzeń między obiektami sceny. Jeśli dojdzie do kolizji, to następuje przekazanie pędu. Ponieważ dwa obiekty mogą kolidować nawet w kilkunastu różnych punktach, do każdego punktu kolizji przykładany jest tylko odpowiedni ułamek całkowitej siły zderzenia. Kolidujące obiekty mogą posiadać różne natury fizyczne (ciała sztywne, rośliny, postacie), więc za właściwą reakcję na kolizję odpowiedzialne są już same obiekty. Pojedyncza klatka gry ma następujący przebieg:

(1) Wyczyszczenie informacji o kolizjach każdego obiektu

(2) Przeprowadzenie testów kolizji

(2.1) Jeśli zaistniała kolizja, rozdzielenie pędów obiektów i przekazanie szczegółów kolizji do zainteresowanych obiektów

(3) Reakcja obiektów na kolizje

Bazowym obiektem fizycznym jest klasa `IPhysicalObject`. Dziedziczy ona z klasy `TrackedObject`, wiec każdy obiekt gry może być potencjalnie śledzony przez klasę `ObjectTracker`. Interfejs jaki zapewnia `IPhysicalObject` wygląda następująco:

```
class IPhysicalBody : public TrackedObject
{
protected:
bool IsDefaultsApplied() const;
```

sprawdza czy zastosowano już domyślne właściwości obiektu

```
bool IsModified() const;
```

sprawdza czy pozycja obiektu została zmodyfikowana

```
public:
bool IsCreated() const;
```

sprawdza czy obiekt został utworzony

```
bool FL_stationary;
```

czy obiekt jest stacjonarny (nieruchomy)

```
bool FL_phantom;
```

czy obiekt nie podlega kolizjom

```
bool FL_physical;
```

czy obiekt podlega sile grawitacji

```
float W_restitution;
```

współczynnik przekazania pędu drugiemu obiektowi

```
float W_restitution_self;
```

współczynnik zachowania pędu przy kolizji – najczęściej jest taki sam jak `W_restitution`, dla ludzi ustawiam go na 0, aby obiekt nie drgał przy ruchach wynikających z animacji szkieletowej

```
xBVHierarchy BVHierarchy;
```

hierarchia brył otaczających obiekt

```
vector<CollisionPoint> Collisions;
```

informacje o ostatnich kolizjach obiektu

```
IPhysicalBody &Modify();
```

oznacza obiekt jako zmodyfikowany – dotyczy to głównie jego pozycji

```
virtual xMatrix &MX_LocalToWorld_Set();
```

pozwala bezpośrednio zmienić macierz położenia obiektu, oznacza obiekt jako zmodyfikowany

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

zatrzymuje obiekt w miejscu

```
virtual xFLOAT GetMass() const = 0;
```

zwraca masę obiektu

```
virtual xVector3 GetVelocity() const = 0;
```

zwraca średnią prędkość całego obiektu

```
virtual xVector3 GetVelocity(const CollisionPoint &CP_point)
                    const = 0;
```

zwraca prędkość wskazanego punktu – klasa `PhysicalWorld` wykorzystuje tą informację podczas przekazywania pędu

```
virtual void ApplyFix(const CollisionPoint &CP_point) = 0;
```

poprawia błędną (kolidującą) pozycję obiektu

```
virtual void ApplyAcceleration(const xVector3 &NW_accel,
                    xFLOAT T_time) = 0;
```

nadaje całemu obiektowi przez zadany czas wskazane przyspieszenie

```
virtual void ApplyAcceleration(const xVector3 &NW_accel,
                    xFLOAT T_time,
                    const CollisionPoint &CP_point) = 0;
```

nadaje przyspieszenie wskazanemu punktowi obiektu

```
virtual void ApplyDefaults();
```

nadaje domyślne wartości właściwościom obiektu

```
virtual void Create();
```

tworzy obiekt, jeśli nie nadano mu jeszcze wartości domyślnych, zostanie to uczynione teraz

```
virtual void FrameStart();
```

przygotowuje obiekt do kolejnej klatki

```
virtual void Update(xFLOAT T_time);
```

aktualizuje stan obiektu – aplikuje reakcje na kolizje i inne zdarzenia fizyczne

```
virtual void Render();
```

przygotowuje obiekt do renderowania

```
virtual void FrameEnd();
```

kończy klatkę

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

niszczy obiekt

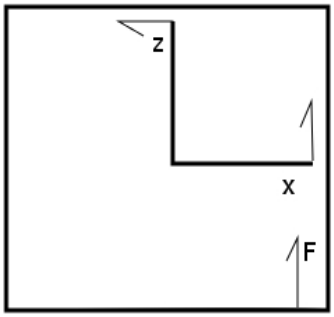
```
};
```

Model fizyczny jakiemu podlega obiekt musi być zdefiniowany w klasie potomnej. Fizyka ciał sztywnych jest zaimplementowana w klasie `PhysicalBody`. Opiera się ona na klasycznych równaniach dynamiki. Jej ruch opisuje liniowa prędkość średnia całego obiektu, oraz prędkość obrotowa wokół centrum masy. Dla uproszczenia przyjęto równomierny rozkład masy w ciele. Prędkość pojedynczego punktu ciała jest sumą prędkości liniowej oraz prędkości wynikającej z obrotu wokół centrum masy.

Obiekty animowane szkieletowo posiadają własną implementację fizyki zdefiniowaną w klasie `SkeletizedObj`. Początkowo bazowała ona na całkowaniu Verleta – dawało to ładne efekty dla postaci podlegających jedynie siłom fizyki. Problemy pojawiały się niestety przy próbach mieszania pozycji cząsteczek z gotowymi

animacjami szkieletowymi. Mechanizm Verleta kontynuował ruch zainicjowany przez animację powodując niestabilność całego obiektu. W ostatecznej wersji każdy staw nadal jest traktowany jako cząsteczka, jej prędkość jest jednak przechowywana w bezpośredniej postaci, tak jak dla ciał sztywnych. Z silnika Verleta zachowałem jedynie mechanizm relaksacji naruszonych ograniczeń ruchu kości. Procesowi relaksacji podlegają jednak jedynie kości które zostały 'skrzywdzone' uderzeniem, bądź sąsiadują z innymi 'skrzywdzonymi' kośćmi. Pozostałe kości poruszają się zgodnie z definicjami zawartymi w animacjach. Zastosowanie silnika Verleta byłoby możliwe, gdyby animacje były definiowane przez siłę, jaką należy nadać stawowi, by znalazł się w pożądanej pozycji, a nie przez bezpośrednie podanie docelowego miejsca – jest to ciekawe rozwiązanie do zbadania w przyszłości.

Zastosowanie silnika Verleta dla ciał sztywnych dawało całkiem dobre efekty, jednakże poprawne wprowadzenie ciała w ruch obrotowy wymagało złożonego wyliczania sił jakie działają na cząsteczki składowe. Aby ruch wirowy nie był nadmiernie tłumiony przez proces relaksacji, trzeba było przykładać do pozostałych cząsteczek siły prostopadłe do siły źródłowej. W ostatecznej wersji pozostałem przy szkolnym modelu fizyki ciał sztywnych.



Rysunek 15: Aby uniknąć tłumienia podczas procesu relaksacji, należy przyłożyć siłę do cząsteczki 'x' oraz prostopadłą siłę do cząsteczki 'z'

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. GRAFIKA

[OGRE3D] Object-Oriented Graphics Rendering Engine
http://www.ogre3d.org/

[Horde3D] Otwarty silnik renderujący Horde3D
http://www.horde3d.org/

[GL_Fonts] Wskazówki dotyczące renderowania tekstu
http://www.opengl.org/resources/faq/technical/fonts.htm

[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

[Jakob] 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