



POLSKO-JAPOŃSKA WYZSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Wydział Informatyki
Katedra multimedialna
Programowanie Gier Komputerowych

Paweł Mieczkowski

Nr albumu 4458

**Aplikacja czasu rzeczywistego z wykorzystaniem
technologii DirectX, PhysX oraz języka skryptowego
Lua na podstawie gry AtBall**

Praca inżynierska
Promotor:
Mgr inż. Filip Starzyński

Warszawa, 2010

Spis treści

Spis treści	2
1 Wstęp	6
1.1 Temat pracy	6
1.2 Cel pracy	7
1.3 Zakres pracy	7
2 Teoria	8
2.1 Wprowadzenie	8
2.2 Gry komputerowe	8
2.2.1 Czym jest gra komputerowa?	8
2.2.2 Historia gier komputerowych	12
2.2.3 Proces tworzenia gry komputerowej	13
2.3 Podstawowe pojęcia dotyczące wyświetlania grafiki w 3D	14
2.3.1 Wektor	15
2.3.2 Macierz	15
2.3.3 Kwaternion	16
2.3.4 Wierzchołek	16
2.4 Technologia DirectX	17
2.4.1 Historia	17
2.4.2 DirectX Graphics	18
2.4.3 DirectX Input	18
2.4.4 DirectX Audio	19
2.4.5 DirectX Play	19
2.5 Proces renderingu	19
2.5.1 Wstęp	19
2.5.2 Przekształcanie wierzchołków	20
2.5.2.1 Macierz świata	20
2.5.2.2 Macierz widoku	22
2.5.2.3 Macierz projekcji	23
2.5.3 Fixed-function/Programmable Pipeline	24

2.5.4	Rasteryzacja	25
2.5.5	Pixel shader	26
2.5.6	Post-Processing	26
2.5.7	Optymalizacja	27
2.5.7.1	Frustrum Culling	27
2.5.7.2	Occlusion Culling	28
2.6	Techniki wyświetlania obrazu	29
2.6.1	Oświetlenie	30
2.6.1.1	Wstęp	30
2.6.1.2	Źródła światła	30
2.6.1.3	Cieniowanie płaskie	31
2.6.1.4	Cieniowanie Gourauda	31
2.6.1.5	Cieniowanie Phonga	31
2.6.2	Teksturowanie	32
2.6.2.1	Nakładanie tekstur	32
2.6.2.2	Filtrowanie tekstur	32
2.6.2.3	Mapowanie normalnych	34
2.7	Fizyka w grach komputerowych - biblioteka PhysX	35
2.7.1	Wstęp	35
2.7.2	Scena	36
2.7.3	Aktorzy	36
2.7.4	Wykrywanie kolizji	38
2.7.5	Sterowanie obiektyami	38
2.7.6	Debugowanie symulacji	39
2.7.7	Podsumowanie	39
2.8	Logika gry - skrypty	40
2.8.1	Wstęp	40
2.8.2	Język Lua	41
2.8.3	Składnia języka	41
2.8.4	Integracja z językiem C++	42
2.8.4.1	ToLua++	43
2.8.4.2	LuaBind	43
3	Praktyka	44
3.1	Architektura silnika	44
3.1.1	Wstęp	44
3.1.1.1	Narzędzia	44
3.1.2	Inicjalizacja	45
3.1.2.1	Game - główny obiekt gry	45
3.1.2.2	Stany	46
3.1.2.3	Pętla gry	47
3.1.2.4	Uaktualnianie logiki	48

3.1.2.5	Wyświetlanie obrazu	49
3.1.3	Grafika 2D - Interfejs użytkownika	50
3.1.3.1	Główne menu	50
3.1.3.2	Okno wiadomości	51
3.1.4	Zasoby	52
3.1.4.1	Menedżer zasobów	53
3.1.5	Obiekty w grze	53
3.1.5.1	Podstawowy obiekt - CEntity	53
3.1.5.2	Aktor w symulacji sceny PhysX - CEntityEx	54
3.1.5.3	Sterowanie	54
3.1.5.4	Obsługa kolizji	55
3.1.5.5	Łączenie obiektów	56
3.1.5.6	Zarządzanie obiektami	56
3.1.6	Scena	56
3.1.6.1	Symulacja fizyczna	57
3.1.6.2	Wyświetlanie obiektów	57
3.1.7	Kamera	57
3.1.7.1	Statyczna	58
3.1.7.2	Dynamiczna	58
3.1.8	Obsługa dźwięku	58
3.1.8.1	Dźwięk 3D	59
3.1.9	Zarządzanie czasem - Timery	59
3.1.9.1	Tworzenie	60
3.1.9.2	Wykonywanie akcji	60
3.1.9.3	Timery systemowe	60
3.1.10	Obsługa skryptów	61
3.1.10.1	Rejestrowanie metod	62
3.1.10.2	Wywoływanie metod skryptowych	62
3.2	Tworzenie gry AtBall	63
3.2.1	Opis gry	63
3.2.2	Wstępna konfiguracja	64
3.2.2.1	Zmienne konfiguracyjne	64
3.2.2.2	Obsługa zmiany stanu gry	64
3.2.3	Tworzenie sceny	65
3.2.4	Poziomy gry	66
3.2.5	Elementy rozgrywki	66
3.2.5.1	Platforma zmiany typu kuli	66
3.2.5.2	Wiatrak	68
3.2.5.3	Zbieranie obiektów	69
4	Podsumowanie	71
4.1	Ocena projektu	71

5 BIBLIOGRAFIA	72
Spis rysunków	73
Spis wydruków	75

Rozdział 1

Wstęp

Niniejszy rozdział ma formę wstępu do pozostazej części pracy. Został podzielony na trzy części. Pierwsza część zawiera wyjaśnienie tematu. Druga przedstawia cel oraz założenia jakie przyjął autor przygotowując projekt. Trzecia zawiera konspekt i ogólny podział pracy.

1.1 Temat pracy

Aplikacja komputerowa to program udostępniający użytkownikowi określone funkcje i wykonujący zadane zadania. Można je podzielić na dwie grupy: **sterowane zdarzeniami i czasu rzeczywistego**.

Aplikacje sterowane zdarzeniami działają w oparciu o reakcje lub dane wprowadzane przez użytkownika. Dobrym przykładem są tutaj dynamiczne serwisy WWW. Generują one swoją treść wynikową na podstawie informacji wprowadzonych przez osobę. Kolejnym przykładem są aplikacje do obsługi baz danych. Użytkownik wprowadza kryteria wyszukiwania, następnie aplikacja po otrzymaniu informacji wykonuje operacje niezbędne do pobrania konkretnych danych, zwraca je użytkownikowi i czeka na dalsze polecenia.

Aplikacje czasu rzeczywistego są rozbudowaniem aplikacji sterowanych zdarzeniami. Mimo że mogą reagować na sygnały zewnętrzne, wykonują w tle inne czynności. Idealnym przykładem takiej aplikacji jest gra komputerowa, której głównym elementem jest pętla. Wykonuje się ona nieprzerwanie przez cały okres działania aplikacji. Najczęściej w jej skład wchodzi:

1. Pobranie informacji dotyczących interakcji użytkownika
2. Uaktualnienie stanu logicznego w zależności od reakcji użytkownika oraz minionego czasu od poprzedniego cyklu (np. fizyki, dźwięku, kamery, pozycji obiektów w scenie)

3. Generowanie wynikowego obrazu (wykonanie jednej klatki, na które składa się przekształcenie wierzchołków modeli do pozycji docelowej, nałożenie tekstur oraz świateł, aż do wyliczenia docelowego koloru piksela na ekranie)

Taka ilość operacji jest bardzo kosztowna obliczeniowo. Aby zachować płynność gry i animacji aplikacja powinna wyświetlać obraz z prędkością od 15 do 30 FPS¹, ponieważ tyle wystarczy aby oszukać ludzki wzrok i przekonać o płynności. Należy w tym celu położyć duży nacisk na metody optymalizacyjne takie jak wyłączenie obiektów niezawierających się w polu widzenia kamery z procesu renderingu, który jest najbardziej złożony obliczeniowo.

1.2 Cel pracy

Stworzenie rozbudowanej gry to skomplikowany i długoplanowy proces, w który zaangażowanych jest co najmniej kilka osób. Autor z powodu ograniczeń czasowych oraz małego doświadczenia w tworzeniu tego typu aplikacji obrał sobie za cel zaimplementowanie prostego silnika graficznego, oraz na jego podstawie stworzenie małej gry zręcznościowej.

Nacisk został położony na dwa zagadnienia występujące w każdej większej produkcji:

- Realistyczna fizyka. Do tego celu wykorzystany został silnik fizyczny **PhysX** firmy **NVIDIA**.
- Użycie zewnętrznych skryptów do oddzielenia logiki aplikacji od kodu źródłowego silnika. Są to pliki tekstowe, które są interpretowane² podczas działania aplikacji. Wybór padł na język skryptowy **LUA**.

1.3 Zakres pracy

Praca została podzielona na 4 główne części.

1. Wstęp, który ma na celu wprowadzenie czytelnika w temat oraz przedstawia cel pracy
2. Część teoretyczna zawierająca opis technologii oraz bibliotek, stanowiący swoisty wstęp do części implementacyjnej projektu
3. Część praktyczna opisująca proces powstawania oraz implementacji silnika graficznego oraz właściwej gry
4. Podsumowanie, w którym autor wyciąga wnioski odnośnie ukończonej pracy, a także problemy na jakie natrafił podczas procesu implementacji

¹ang: Frames Per Second - Klatek na sekundę

²każda instrukcja jest czytana z pliku i wykonywana

Rozdział 2

Teoria

2.1 Wprowadzenie

Część teoretyczna stanowi wprowadzenie teoretyczne do tematu tworzenia gier komputerowych. Zawarte są w niej następujące informacje:

1. Definicja, rodzaje, historia oraz proces tworzenia gry komputerowej
2. Podstawowe pojęcia oraz definicje związane z wyświetlaniem grafiki w 3D
3. Zapoznanie z biblioteką Microsoft DirectX oraz jej elementami
4. Omówienie procesu tworzenia jednej klatki obrazu czyli proces renderingu
5. Przedstawienie silnika fizycznego NVIDIA PhysX
6. Krótkie wprowadzenie w tematykę języków skryptowych na podstawie języka Lua

2.2 Gry komputerowe

2.2.1 Czym jest gra komputerowa?

Gra komputerowa jest aplikacją komputerową, która dostarcza użytkownikowi dawki rozrywki poprzez wykonywanie określonych zadań wpływając na obiekty oraz ich zachowanie w sztucznie stworzonym środowisku. W zależności od typu gry, zawiera ona cel oraz określone zasady, którymi trzeba się kierować, aby ukończyć poziom lub całą grę. Bardziej otwarte produkcje umożliwiają dowolną (nieskończoną) grę w obrębie wirtualnego świata. Przykładem takich są symulatory pojazdów (np. pociągów) lub internetowe gry wieloosobowe, w których rozrywka toczy się na bieżąco, umożliwiając graczom praktycznie nieskończoną zabawę.

Podając definicję gry komputerowej warto wspomnieć o pojęciu **gry konsolowej**. Do jej uruchomienia wymagana jest zewnętrzna konsola przeznaczona wyłącznie do gier, dzięki czemu (w przeciwieństwie do gry uruchamianej na komputerze) cała moc obliczeniowa może zostać wykorzystana na potrzeby gry. Przykładem najpopularniejszych konsol do gier są Xbox firmy Microsoft oraz PlayStation 3 firmy Sony.

Prezentacja głównych gatunków gier:

- **Platformowe i zręcznościowe** – Gra polegająca na kierowaniu postacią lub innym obiektem oraz pokonywanie przeszkód. W przypadku gier zręcznościowych sterowanie najczęściej wymaga zręczności i dokładności w celu ukończenia poziomu lub planszy.



Rysunek 2.1: Spyro The Dragon

- **Strategiczne** – Strategie skupiają się na odpowiednim planowaniu militarnym w celu efektywnego wykorzystania jednostek do pokonania przeciwnika oraz ekonomicznym gospodarowaniu surowcami. Gatunek ten dzieli się na **Strategiczne gry turowe** (TBS¹) oraz **Strategie Czasu Rzeczywistego** (RTS²)



Rysunek 2.2: Age of empires 2

- **RPG**³ – Połączenie gry platformowej i przygodowej. Główny aspekt gier RPG to dbanie o rozwój swojej postaci. Przypisanych jest do niej wiele cech, które są rozwijane w trakcie trwania gry. Oprócz tego w większości gier każda postać musi dokonać wyboru jednej z profesji np. wojownik, czarodziej, ork, lub innych przewidzianych

¹ang: TBS – Turn-based strategy

²ang: RTS – Real-time strategy

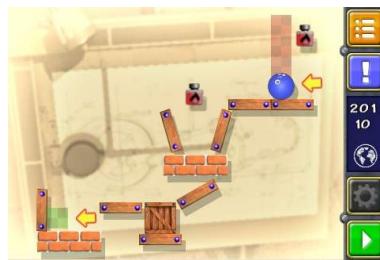
³ang: Role Playing Games

dla danej fabuły, które wstępnie zwiększą wybrane właściwości (np mocniejsza magia ale słabsze posługiwanie mieczem). Popularną w ostatnich latach odmianą są gry MMORPG⁴, dzięki którym tysiące graczy z całego świata mogą się zmierzyć w wirtualnym świecie.



Rysunek 2.3: Final Fantasy 9 – RPG w świecie fantasy

- **Logiczne** – Głównym celem tych gier są zagadki i łamigłówki, które należy rozwiązać aby ukończyć grę. Również gry wieloosobowe takie jak szachy i warcaby zaliczane są do tego gatunku.



Rysunek 2.4: Crazy Machines – rozwiązywanie fizycznych łamigłówek

- **Przygodowe (fabularne)** – Gry, w których zręcznościowe sterowanie jest ograniczone do minimum a główny nacisk położony jest na fabułę oraz rozwiązywanie zagadek.



Rysunek 2.5: Fahrenheit – gra fabularna

- **Sportowe** – Emulują tradycyjne gry takie jak piłka nożna, baseball, golf itp.

⁴ang: Massively Multiplayer Online Role Playing Games



Rysunek 2.6: Fifa 2009 – piłka nożna

- **Wyścigowe** – Popularny gatunek, w którym gracz zasiada za sterami samochodów lub innych pojazdów i bierze udział w wyścigach, w celu pokonania komputerowych przeciwników, lub w przypadku gry przez internet – innych graczy.



Rysunek 2.7: Need For Speed Carbon – wyścigi uliczne

- **Symulacyjne** – Symulatory symulują środowisko, w którym gracz może cieszyć się rzeczywistymi warunkami np. kierowania pociągiem lub latania samolotem. Niektóre gry symulacyjne dodają pewne założenia lub cel wymagany do ukończenia gry np. zestrzelenie odpowiedniej ilości samolotów, podczas gdy inne (np. The Sims) umożliwiają grę bez żadnych ograniczeń.



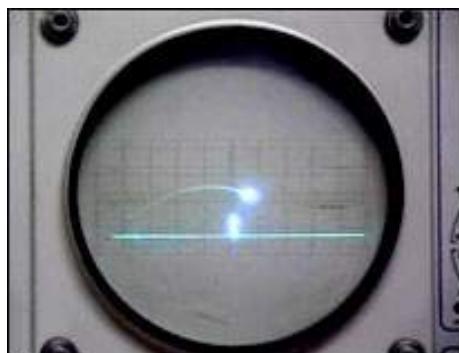
Rysunek 2.8: Microsoft Train Simulator – Symulator jazdy pociągiem

2.2.2 Historia gier komputerowych

Historia gier komputerowych, mimo dynamicznego rozwoju w latach siedemdziesiątych, została zapoczątkowana w 1947 roku, kiedy to Thomas T. Goldsmith Jr. i Estle Ray Mann

zaprojektowali symulację pocisku rakietowego uderzanego w wyznaczony cel na oscyloskopie CRT⁵. Kąt wystrzelenia pocisku oraz jego prędkość była sterowana odpowiednimi gałkami.

Kolejnym krokiem było przeniesienie na ekran monitora gry Kółko i Krzyżyk, a w 1958 roku stworzenie przez Wiliama Higinbothana gry Tennis for Two.



Rysunek 2.9: Tennis for Two - jedna z pierwszych gier komputerowych

W 1961 powstała pierwsza ogólnodostępna i wpływowa gra Spacewar! zaprojektowana przez grupę studentów z MIT⁶, która została zaprezentowana rok później podczas MIT Science Open House.

W 1969 roku Ralph Baer (dwa lata po stworzeniu pistoletu świetlnego) stworzył prototyp pierwszej konsoli do gier wideo. Po sprzedaniu go firmie Magnavox, wydana została w 1972 roku pod nazwą Magnavox Odyssey.

Początek lat siedemdziesiątych to również start automatów do gry, które za pieniądze umożliwiały zabawę w różne gry. Mimo braku sukcesu po wydaniu pierwszego modelu umożliwiającego grę w Spacewar! z powodu zbyt dużej trudności gry, firma Atari osiągnęła sukces wydając automat do gry PONG.

Prawdziwy wzrost zainteresowania wirtualną rozrywką nastąpił po stworzeniu i rozpoznanieniu pierwszych komputerów osobistych, dzięki czemu co raz więcej osób mogło cieszyć się grami w domu. Przez kolejne lata razem z rozwojem technologii komputerowej, nowych technologii wyświetlania obrazu oraz bibliotek programistycznych, a także możliwości sprzętowych takich jak karty graficzne, powstawały co raz to nowocześniejsze produkcje, zapewniające jeszcze bardziej realistyczne efekty wizualne. Koniec XX wieku przyniósł szybki rozwój konsol wideo. Dominującymi modelami były konsole GameBoy, Playstation, Sega Saturn oraz Nintendo 64. Z początkiem XXI wieku nastąpiły czasy nowych wersji konsol takich jak Playstation 2, Xbox, Dreamcast, aż do dzisiejszych czasów, w których dominują Playstation 3, Wii oraz Xbox 360.

⁵ang: Cathode-Ray Tube - kineskop z działem elektronowym

⁶Massachusetts Institute of Technology



Rysunek 2.10: Konsola Sony PlayStation 3

2.2.3 Proces tworzenia gry komputerowej

Tworzenie gry komputerowej to bardzo złożony proces, dlatego podczas produkcji zaangażowanych jest od kilku do nawet kilkuset osób. Najczęściej przyjmowaną metodą projektową jest poniższy schemat:

1. **Pomysł** – Pierwszym krokiem przy tworzeniu gry komputerowej jest decyzja odnośnie typu gry, analiza zainteresowania wśród docelowych odbiorców oraz ustalenie wstępnych informacji dotyczących możliwości budżetowych oraz ludzkich niezbędnych do pomyślnego zrealizowania projektu.
2. **Projektowanie** – W tej fazie tworzy się dokument zawierający informacje o grze. Musi w nim być zawarty pomysł gry, jej cele i fabuła, gatunek oraz docelowa grupa odbiorców. Kolejnym etapem jest tworzenie dokumentacji projektu⁷, w którym zawarte są wszystkie informacje o grze. Dokument ten jest podstawą do dalszych prac nad projektem.
3. **Tworzenie prototypu** – Prototyp jest produktem, który ma pokazać ogólne założenia projektu oraz przykładowy wygląd. Najczęściej jest to przykładowy poziom gry pokazujący podstawowe aspekty sterowania i fabuły. Dzięki takiemu rozwiązaniu projektanci mogą uzupełnić braki w dokumentacji projektowej, oraz zwrócić uwagę na problemy, które mogą wystąpić przy tworzeniu właściwej gry.
4. **Produkcja gry** – Kiedy Design Doc jest gotowy następuje projektowanie i implementacja właściwego silnika gry. Na tym etapie kierownicy rozdzielają zadania pomiędzy zespoły programistów, grafików i designerów. Po zaimplementowaniu całej funkcjonalności następuje przejście do fazy testowania.
5. **Testowanie i usuwanie błędów** – Testowanie jest jednym z najważniejszych etapów tworzenia gry. Od zespołu testerów oraz wykonanej przez nich pracy zależy sukces całego przedsięwzięcia. Zbyt duża liczba krytycznych błędów lub niedociągnięć po wydaniu gry może negatywnie wpłynąć na recenzje produktu oraz zrazić odbiorców do kupna.

⁷tzw. Design Doc

6. **Wydanie i marketing** – Każdy projekt ma wyznaczane kamienie milowe ⁸, które określają planowane ukończenie poszczególnych części projektu. W przypadku pozytywnego wyniku testów firmy wypuszczają wersje końcowe. Czasami jednak termin końcowy zostaje przekroczony z powodu dużej ilości błędów i nienadążania zespołu naprawiającego je, dlatego udostępnia się wersję beta wąskiej grupie docelowych użytkowników, którzy zgłaszą poprawki i uwagi, tak aby końcowy produkt miał jak najmniej wad. Częstym przypadkiem po oficjalnym wydaniu gry jest znalezienie krytycznych błędów przez końcowych użytkowników, dlatego szybką reakcją firmy musi być wydanie patch'a ⁹ poprawiającego go, ponieważ każda usterka może spowodować niekorzystny dla firmy spadek wizerunku oraz zainteresowania produktem.

2.3 Podstawowe pojęcia dotyczące wyświetlania grafiki w 3D

Przed przejściem do omawiania bibliotek programistycznych oraz procesu renderingu niezbędne jest wyjaśnienie najważniejszych pojęć stosowanych w przekształcaniu i wyświetlaniu grafiki 3D.

2.3.1 Wektor

Wektory w grafice 3D mają dwa zastosowania. Pierwszym jest reprezentacja punktu w przestrzeni. Przykładowo na dwuwymiarowej osi liczbowej punkt ma współrzędne $x = 2.5$ i $x = -5$. Wektor reprezentujący pozycję wygląda następująco: $\vec{pos} = (2.5, -5)$. W grafice trójwymiarowej dodatkową osią jest głębia (osi Z). Przyjmując $z = 12$, przykładowy wektor otrzymuje postać: $\vec{pos} = (2.5, -5, 12)$.

Drugim zastosowaniem wektorów jest identyfikacja kierunku. Wektor taki ma postać linii w układzie współrzędnych o zdefiniowanym punkcie początkowym (startu) i końcowym (końca) wizualnie zakończonym strzałką. Mimo, że początek wektora nie musi znajdować się w początku układu współrzędnych, znacząco ułatwia to wszelkie operacje i obliczenia, ponieważ w tym wypadku wystarczą jedynie współrzędne punktu końcowego, który określa kierunek wektora.

Oprócz standardowych operacji dodawania, odejmowania i mnożenia wektorów, wyróżniamy w grafice komputerowej trzy szczególne działania:

1. **Normalizacja** – Znormalizowany wektor otrzymuje się przez podzielenie wszystkich elementów wektora przez jego długość. Otrzymany wektor ma ten sam kierunek, zwrot i długość równą 1. Ułatwia to obliczenia i uniezależnia wynik od długości wektorów.

⁸ang: Milestone

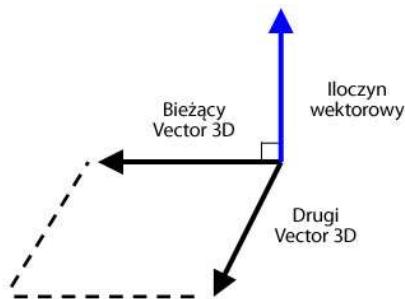
⁹dodatek do wydanej gry poprawiający błędy lub dodający nowe funkcje

2. **Iloczyn skalarny** – Operacja najczęściej stosowana w obliczeniach grafiki komputerowej.

$$\vec{a} \circ \vec{b} = \vec{a} \cdot \vec{b} \cdot \cos\alpha$$

Dzięki powyższemu równaniu można otrzymać kąt α między dwoma znormalizowanymi wektorami. Przydaje się on najczęściej do obliczania natężenia światła padającego na element.

3. **Iloczyn wektorowy** – Wynikiem iloczynu wektorowego jest wektor prostopadły do dwóch podanych wektorów. Przykładowy wynik tej operacji jest pokazany na rysunku 2.11



Rysunek 2.11: Iloczyn wektorowy

2.3.2 Macierz

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \quad (2.1)$$

Macierz, w przeciwieństwie do wektora (tablicy jednowymiarowej), jest tablicą wielowymiarową. Do jej elementów można się odwoływać poprzez podanie numeru wiersza oraz kolumny.

W grafice komputerowej najważniejszą rolą macierzy są przekształcenia. Operacje translacji, rotacji i skalowania za pomocą macierzy zostaną dokładnie omówione w rozdziale 2.5.2 poświęconemu procesowi renderingu.

2.3.3 Kwaterion

Kwaterion¹⁰ to wektor czteroelementowy, którego pierwsze trzy elementy oznaczają os, a czwarty wartość obrotu wyrażonego w radianach według danej osi.

¹⁰ang: Quaternion

W grafice komputerowej kwaterniony znaczco ułatwiają obracanie elementów, głównie dzięki operacji kompozycji:

$$Q_w = Q_1 \cdot Q_2$$

Dla $Q_1 = (x_1, y_1, z_1, \alpha)$ i $Q_2 = (x_2, y_2, z_2, \beta)$, Q_w oznacza obrót wokół osi x_1, y_1, z_1 o kąt α , a następnie wokół osi x_2, y_2, z_2 o kąt β .

Drugą ważną operacją jest interpolacja, dzięki której mamy możliwość obliczania pośrednich wartości obrotów w danym czasie. Umożliwia ona tworzenie płynnych animacji np. kamery.

2.3.4 Wierzchołek

Wierzchołek¹¹ jest to punkt w przestrzeni 3D zawierający pozycję jednego punktu. Dwa wierzchołki łączą się w krawędź, a trzy krawędzie w trójkąt - podstawowy budulec modeli trójwymiarowych. Wierzchołki mogą zawierać dodatkowe informacje wymagane przy procesie renderingu takie jak:

- **Koordynaty UV** – Określają współrzędne punktu na płaszczyźnie tekstury
- **Wektor normalny** – Wektor prostopadły do powierzchni trójkąta
- **Kolor** – Kolor wierzchołka
- **Tangent i binormal** – Wartości przydatne przy efektach takich jak mapowanie wypukłości

2.4 Technologia DirectX

Rozdział przedstawia technologię Microsoft DirectX, jej historię oraz podział na główne elementy.

2.4.1 Historia

DirectX jest zestawem bibliotek wspomagających generowanie grafiki dwu i trójwymiarowej. Pierwotnie dostępna była biblioteka WinG, która rozszerzała możliwości WinAPI¹², jednak brak bezpośredniego dostępu do warstwy sprzętowej bardzo ograniczał ówczesne programowanie gier.

Pierwsze wydanie pojawiło się w roku 1995 wraz z wydaniem systemu operacyjnego Windows 95. Główną różnicą między WinG a biblioteką DirectX był bezpośredni dostęp do warstwy sprzętowej, dzięki czemu programiści nie musieli już tworzyć własnych systemów

¹¹ang: Vertex

¹²Application Programming Interface

generowania i wyświetlania grafiki 3D tylko mogli skupić się na logice aplikacji.

Wraz z nowymi wersjami bibliotek rozszerzane były one o nowe możliwości i funkcje, jednak dopiero wersje 6.1 i 7 wydane w 1999 roku udostępniły obsługę potoków renderujących. Wersja 8.0 wprowadziła programowalny model przetwarzania - Vertex i Pixel Shader. Dzięki temu programista mógł zaprogramować w języku podobnym do asemblera sposób wykonywania obliczeń na wierzchołkach i pikselach bezpośrednio na karcie graficznej.

Ostatnią wersją przeznaczoną dla systemu Windows XP jest DirectX 9.0c, w której do pisania shaderów został przygotowany język HLSL¹³ ze składnią zbliżoną do języka C. Za pomocą tego narzędzia możliwe stało się tworzenie efektów niemożliwych do wykonania za pomocą standardowych procedur.

Wraz z premierą systemu Windows Vista została udostępniona nowa wersja biblioteki DirectX oznaczona numerem 10.0. Główną cechą, która odróżnia ją od poprzednich wersji jest brak kompatybilności wstecz. Powodem jest przepisanie całej biblioteki na nowo, dzięki czemu zoptymalizowano kod i przystosowano tylko do najnowszych kart graficznych. Zmniejszono też narzut interfejsu API na proces tworzenia grafiki, czyli moc obliczeniową CPU¹⁴, który wykorzystywał około 40% cykli w wersji 9.0c, do 20% w wersji 10.0. Dzięki temu główna jednostka obliczeniowa zyskała więcej czasu na zaawansowane obliczenia AI¹⁵ lub odwzorowywanie fizyki.

Z powodów ograniczeń sprzętowych autor w dalszej części pracy oraz projekcie będzie opierać się na wersji 9.0c.

DirectX jest komponentem, który stanowi wspólne środowisko dla czterech głównych usług: graphics, input, audio i play.

2.4.2 DirectX Graphics

DirectDraw jest częścią DirectX odpowiedzialną za wyświetlanie grafiki 2D. Z powodu dynamicznego rozwoju grafiki 3D, został on połączony z Direct3D tworząc wspólny pakiet DirectX Graphics.

Direct3D odpowiada za wyświetlanie grafiki trójwymiarowej. Umożliwia wykorzystywanie akceleracji sprzętowej oraz zaawansowanych możliwości kart graficznych:

- **Obsługa i filtrowanie tekstur** – Przykładowa technika o nazwie MipMapping (dokładniejszy opis w rozdziale 2.6.2.2) umożliwia poprawę jakości wyświetlonej tekstuury w zależności od odległości wyświetlonego obiektu od pozycji kamery

¹³High Level Language

¹⁴Central Processing Unit - Procesor

¹⁵Artificial intelligence - Sztuczna inteligencja

- **Z-Bufor** – Przechowywanie informacji odnośnie odległości (głębii) wyświetlanych pikseli
- **Oświetlenie** – Wbudowana obsługa cieniowania płaskiego oraz Gouraud'a (więcej o oświetleniu w rozdziale 2.6.1)
- **Alfa blending** – Technika pozwalająca na tworzenie przezroczystych obiektów
- **Anti-aliasing** – Poprawianie jakości obrazu
- **Vertex/Pixel shaders** – Obsługa shaderów
- ... i wiele innych

DirectX Graphics wspomaga pisanie aplikacji graficznych nie tylko na platformy Windows, ale również konsole Xbox i Xbox 360 firmy Microsoft.

2.4.3 DirectX Input

DirectX Input jest komponentem odpowiedzialnym za obsługę urządzeń wejściowych takich jak mysz, klawiatura czy urządzenia przystosowane do gier typu joystick, gamepad itp. Może on pracować w dwóch trybach:

1. **Bezpośredni** – W tym trybie odczytywany jest aktualny stan wejściowy. Jest on najszybszy, jednak niesie za sobą poważną wadę: między dwoma odczytami stan urządzenia może się zmienić wielokrotnie (np. wcisnięcie i zwolnienie klawisza), przez co taka operacja zostanie pominięta.
2. **Buforowany** – Uzupełnia tryb bezpośredni o buforowanie zmian stanu urządzenia, dzięki czemu każda zmiana zostanie zarejestrowana i może zostać obsłużona. Jest to najdokładniejsza metoda, dlatego jest najczęściej wykorzystywana w grach komputerowych.

Ogromną zaletą biblioteki DirectX Input jest tzw. Action Mapping umożliwiający przyznanie akcji do konkretnych przycisków. Dzięki takiemu rozwiązaniu po przypisaniu konkretnej akcji, np. ruch do przodu strzałce na klawiaturze i przyciskowi "w górę" na gamepadzie, w aplikacji otrzymujemy komunikat wystąpienia danej akcji - niezależnie od tego jakie urządzenie ją wygenerowało.

2.4.4 DirectX Audio

Głównym elementem DirectX Audio jest biblioteka DirectSound, umożliwiająca szybki dostęp do karty dźwiękowej i odtwarzanie/nagrywanie dźwięków. Dodatek DirectSound3D zapewnia obsługę dźwięku trójwymiarowego.

DirectMusic, mimo że wykorzystuje wewnętrznie DirectSound, jest dużo bardziej rozbudowanym komponentem umożliwiającym odtwarzanie wielu formatów takich jak OGG i MP3.

2.4.5 DirectX Play

DirectPlay pozwala na tworzenie aplikacji i gier sieciowych. Umożliwia tworzenie dwóch typów sieci:

1. **Klient-Serwer** – Sieć w której zarządzaniem połączeniami i obsługą klientów zajmuje się serwer. Informacje wysyłane z urządzeń klienckich trafiają najpierw do serwera, który następnie rozsyła daną informację do wybranych lub wszystkich uczestników.
2. **Peer to Peer** – Komunikacja odbywa się na zasadzie "każdy z każdym" czyli wysyłając informację trafia ona bezpośrednio do wszystkich uczestników sieci.

2.5 Proces renderingu

2.5.1 Wstęp

Proces renderingu jest najbardziej skomplikowanym i czasochłonnym procesem podczas działania gry, ponieważ musi się on wykonywać w czasie rzeczywistym i w ciągu jednej sekundy wywołać co najmniej 15-30 razy, aby zachować płynność wyświetlanego obrazu. W dawnych czasach duża część operacji była wykonywana przez CPU przez co tworzenie zaawansowanych efektów było bardzo czasochłonne i niemożliwe w przypadku aplikacji czasu rzeczywistego. Dzięki dynamicznemu rozwojowi układów graficznych większość obliczeń została przeniesiona na procesory GPU karty graficznej.

Niniejszy rozdział ma na celu przedstawienie procesu renderowania jednej klatki obrazu, od przekształcania wierzchołków, przez teksturowanie i oświetlenie, do post-processing'u gotowej klatki.

2.5.2 Przekształcanie wierzchołków

2.5.2.1 Macierz świata

Każdy model reprezentowany jest przez wierzchołki. Ich liczba mieści się w przedziale od kilku do nawet kilkuset tysięcy. Każdy z nich ma ustaloną pozycję w lokalnym układzie współrzędnych, czyli względem środka modelu.

Pierwszą operacją jaką należy wykonać w celu wyświetlenia modelu na ekranie jest transformacja każdego wierzchołka do wspólnego układu współrzędnych. Tworzy się w tym celu macierz świata, przez którą należy pomnożyć pozycję wierzchołka. Ponieważ macierze

o wymiarach 4x4 są standardem w obliczeniach, mnożąc wektor należy dodać do niego czwartą pozycję o wartości 1):

$$\vec{pos} = (x, y, z, 1)$$

Wyróżniamy trzy podstawowe operacje przekształcające: translacja, rotacja, skalowanie.

1. **Translacja** – Ogólna postać macierzy translacji na pozycję x_2, y_2, z_2 jest następująca:

$$M_{translation} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_2 & y_2 & z_2 & 1 \end{bmatrix} \quad (2.2)$$

Aby otrzymać wektor $\vec{pos} = (5, 2, -3, 1)$ przesunięty względem pozycji $x = 4, y = 3, z = 10$ należy wykonać działanie:

$$\vec{posT} = [5, 2, -3, 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 4 & 3 & 10 & 1 \end{bmatrix} \quad (2.3)$$

Wynikową pozycję są pierwsze trzy elementy wynikowego wektora.

2. **Skalowanie** – Macierz skalowania przez wartości x_2, y_2, z_2 według odpowiednich osi X, Y i Z prezentuje się następująco:

$$M_{scale} = \begin{bmatrix} x_2 & 0 & 0 & 0 \\ 0 & y_2 & 0 & 0 \\ 0 & 0 & z_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.4)$$

Działanie w wyniku którego otrzymujemy przeskalowany wektor wygląda identycznie jak w przypadku operacji translacji - należy przemnożyć wektor przez macierz skalowania.

3. **Rotacja** – Macierz obrotu dla każdej osi X, Y i Z jest inna:

- Względem osi X:

$$M_{rotationX} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha & 0 \\ 0 & -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.5)$$

- Względem osi Y:

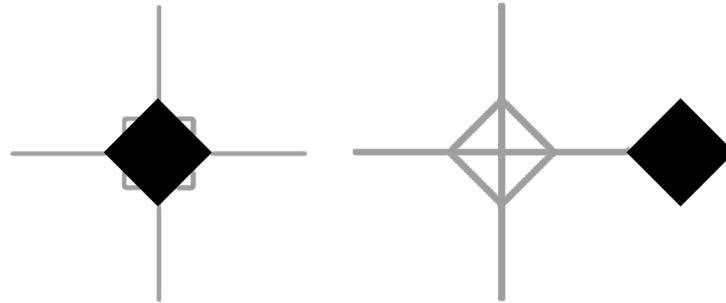
$$M_{rotationY} = \begin{bmatrix} \cos \alpha & 0 & -\sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.6)$$

- Względem osi Z:

$$M_{rotationZ} = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 & 0 \\ -\sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.7)$$

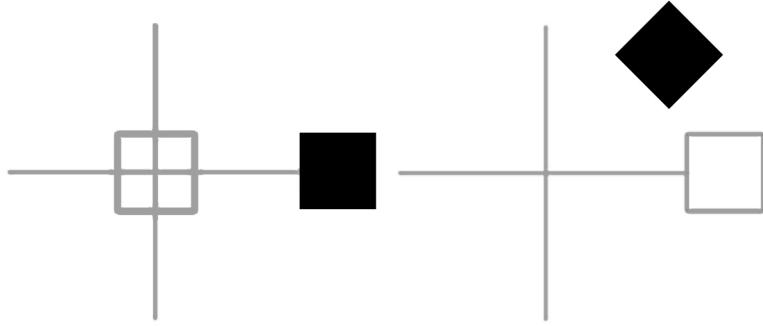
Mnożąc odpowiednie macierze możemy otrzymać macierz, która jest połączeniem kilku transformacji. Przykładowo chcąc powiększyć model dwukrotnie, następnie obrócić go względem osi X o 25° i Y o 125° oraz przemieścić względem punktu $(5, 1, -2)$, należy stworzyć następującą macierz świata:

$$M_{World} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(25^\circ) & \sin(25^\circ) & 0 \\ 0 & -\sin(25^\circ) & \cos(25^\circ) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(125^\circ) & 0 & -\sin(125^\circ) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(125^\circ) & 0 & \cos(125^\circ) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 1 & -2 & 1 \end{bmatrix} \quad (2.8)$$



Rysunek 2.12: Kolejność mnożenia: rotacja * translacja

Ważnym elementem jest kolejność mnożenia macierzy. Na rysunku 2.12 obiekt został najpierw obrócony względem początku układu, a następnie przesunięty. Rysunek 2.13 pokazuje odwrotne mnożenie: obiekt najpierw został przesunięty, a następnie obrócony względem początku układu. Mimo, że taka sytuacja najczęściej jest niepożądana, ma swoje zastosowanie w rysowaniu obiektów, których pozycja zależy od innego obiektu. Dobrym



Rysunek 2.13: Kolejność mnożenia: translacja * rotacja

przykładem jest układ planet. Rysując księżyc ziemi najpierw obracamy go o pewien kąt, następnie przesuwamy w odpowiednią stronę od planety. Jako że ziemia jest również obiektem, który krąży wokół słońca, ma swoją macierz świata, przez którą musimy pomnożyć macierz księżyca. Dzięki temu pozycja księżyca zostaje przekształcona względem nowej pozycji ziemi.

2.5.2.2 Macierz widoku

Kolejnym etapem jest przekształcenie wierzchołków z globalnej przestrzeni do przestrzeni obserwatora (kamery). W tym celu należy zbudować macierz widoku i pomnożyć przez nią macierz świata obiektu. Pozycję kamery określają następujące wektory:

- \vec{Pos} – Pozycja kamery w globalnej przestrzeni
- \vec{Up} – Kierunek w góre
- \vec{Right} – Kierunek w prawo
- \vec{LookAt} – Kierunek patrzenia kamery

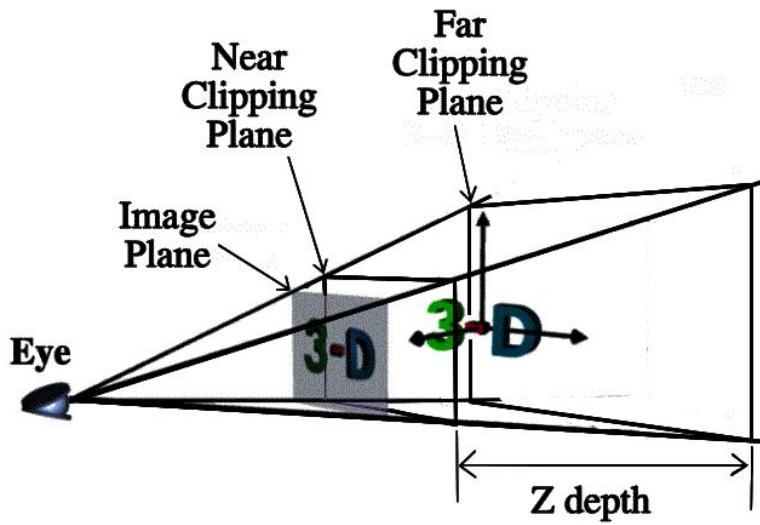
Wektory \vec{Up} , \vec{Right} i \vec{LookAt} są znormalizowane i prostopadłe do siebie. Mając te dane można zbudować macierz (funkcja *Dot* oblicza iloczyn skalarny między dwoma wektorami):

$$M_{View} = \begin{bmatrix} Right.x & Up.x & LookAt.x & 0 \\ Right.y & Up.y & LookAt.y & 0 \\ Right.z & Up.z & LookAt.z & 0 \\ -Dot(Pos, Right) & -Dot(Pos, Up) & -Dot(Pos, LookAt) & 1 \end{bmatrix} \quad (2.9)$$

2.5.2.3 Macierz projekcji

Zadaniem macierzy projekcji jest przeniesienie obrazu trójwymiarowego na ekran dwuwymiarowy monitora zapewniając wrażenie trójwymiarowości (głębii) obiektów. Biblioteka DirectX umożliwia stworzenie gotowej macierzy projekcji za pomocą funkcji [D3DXMatrixPerspectiveFovLH\(\)](#). Parametry jakie przyjmuje to:

1. Kąt widzenia, najczęściej 45°



Rysunek 2.14: Budowa bryły obcinania

2. Stosunek długości oraz szerokości ekranu
3. Odległość od obserwatora do Near Clipping Plane i Far Clipping Plane (patrz rys. 2.14)

Powstała na obrazku 2.14 bryła pokazuje nie tylko proces przekształcania obrazu z 3D na 2D, ale również jest bryłą obcinania tzn. wierzchołki które wychodzą poza nią nie są uwzględniane.

2.5.3 Fixed-function/Programmable Pipeline

W dawnych czasach (przed DirectX 8.0) dostępna była jedynie ustalona kolejność wykonywania operacji w procesie przekształcania wierzchołków na piksele przez kartę graficzną (tzw. fixed-function pipeline):

1. Przekształcenie świata
2. Przekształcenie widoku
3. Przekształcenie projekcji
4. Obcinanie ¹⁶

Macierze podawało się poprzez odpowiednie wywołania funkcji. Wszelkie modyfikacje i efekty musiały być umieszczane w kodzie aplikacji, przez co modyfikacja pojedynczych wierzchołków była niemożliwa. Światła oraz teksturowanie również były ograniczone do możliwości używanej biblioteki.

Wraz z wprowadzeniem z wersją 8.0 programowalnego vertex shadera to ograniczenie zniknęło. Programista zyskał pełną kontrolę nad procesem przekształcania wierzchołków

¹⁶ang: Culling

przy pomocy małych programów zwanych shaderami uruchamianych na procesorach vertex shader na karcie graficznej. Takich procesorów jest przeważnie od kilku do kilkunastu, dzięki czemu karta graficzna może równolegle wykonywać obliczenia zmniejszając tym samym czas potrzebny na wygenerowanie klatki.

Pierwsza wersja udostępniała dość skomplikowany i niewygodny język przypominający asemblera. Dopiero w wersji 9.0 wprowadzono język HLSL, który składnią przypomina język C. Program wykonuje się dla każdego wierzchołka przyjmując na wejście informacje takie jak pozycja, koordynaty tekstuury czy wektor normalny, a następnie zwraca przekształconą pozycję oraz inne informacje, które służą w kolejnych krokach renderingu. Przykładowy kod wykonujący przekształcenie jednego wierzchołka został zamieszczony poniżej.

```
float4x4 WVP;
float4x4 World;
float4x4 View;
float4x4 Projection;
struct VS_Input{
    float4 Pos      : POSITION;
    float3 Normal   : NORMAL0;
    float2 Tex0     : TEXCOORD0;
};

struct VS_Output{
    float4 Pos      : POSITION;
    float2 Tex0     : TEXCOORD0;
    float3 Normal   : TEXCOORD1;
};

VS_Output VertexShader(VS_Input In){
    VS_Output Out = (VS_Output) 0;
    Out.Pos    = mul(In.Pos, WVP);
    Out.Tex0   = In.Tex0;
    Out.Normal = mul(float4(In.Normal, 1), World);
    return Out;
}

technique Sample{
    pass P0{
        VertexShader = compile vs_2_0 VertexShader();
    }
}
```

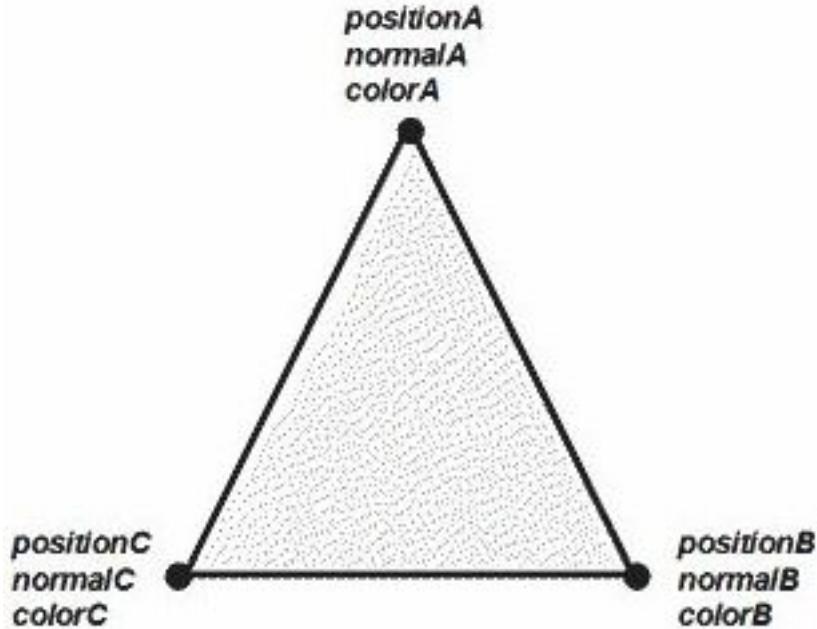
Wydruk 2.1: Przykładowy Vertex Shader

Powyższy kod wskazuje wyraźnie na pełną kontrolę i swobodę w wykonywaniu operacji, dzięki czemu możliwe jest implementowanie interesujących efektów takich jak deformacja obiektu w czasie rzeczywistym.

2.5.4 Rasteryzacja

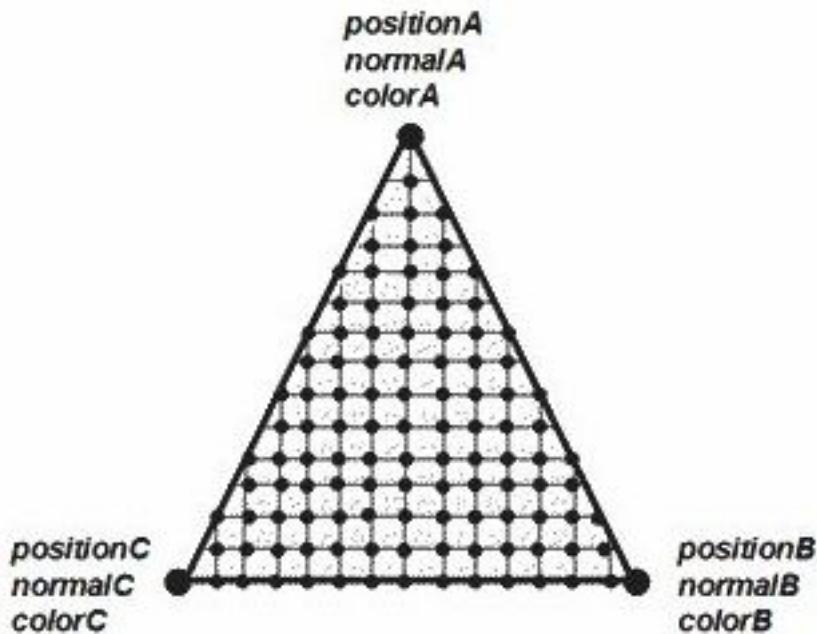
Głównym celem procesu rasteryzacji jest jak najwierniejsze przedstawienie płaskiej figury na urządzeniu rastrowym dysponującym ograniczoną rozdzielczością (ilością pikseli

na ekranie). Rasteryzator przygotowuje dane dla kolejnego etapu renderingu: pixel shadera. Na zasadzie interpolacji liniowej opartej na informacjach o wierzchołkach trójkąta generuje pozycję, kolor, koordynaty tekstuury, i inne wartości, które następnie są przekazywane do programu obliczającego wynikowy kolor każdego pixela. Rysunek 2.15 pokazuje trójkąt



Rysunek 2.15: Trójkąt przed rasteryzacją

złożony z trzech wierzchołków, w których zapisana jest informacja o pozycji, wektorze normalnym i kolorze. Po procesie rasteryzacji na rysunku 2.16 pokazane są wyznaczone



Rysunek 2.16: Trójkąt i wyznaczone piksele po procesie rasteryzacji

piksele wewnętrznych trójkąta. Każdemu z nich przypisana jest interpolowana wartość każdej

składowej wierzchołka.

2.5.5 Pixel shader

Pixel shader, podobnie jak vertex shader, jest procesorem na karcie graficznej. O ile vertex shader operuje na pojedynczych wierzchołkach, pixel shader zajmuje się wyliczeniem końcowego koloru dla każdego piksela na ekranie. Program pixel shader przyjmuje na wejście wartości wyliczone podczas etapu rasteryzacji i na ich podstawie wylicza końcowy kolor używając wartości koloru przekazanego na wejście lub mapując kolor tekstury za pomocą przekazanych koordynatów UV.

```
(...)
struct VS_Output{
    float4 Pos      : POSITION;
    float2 Color    : COLOR;
    float3 Normal   : TEXCOORD0;
};

struct PS_Output{
    float4 Color : COLOR;
};

VS_Output VertexShader(VS_Input In){
    (...) /* Przekształcenie wierzchołków */
}

PS_Output PixelShader(VS_Output In){
    PS_Output Out = (PS_Output) 0;
    Out.Color = In.Color;
}

technique Sample{
    pass P0{
        VertexShader = compile vs_2_0 VertexShader();
        PixelShader  = compile ps_2_0 PixelShader();
    }
}
```

Wydruk 2.2: Przykładowy Pixel Shader

Pixel shader dzięki pełnej kontroli nad wyliczaniem wynikowego koloru daje możliwość implementacji oświetlenia, teksturowania, i wielu innych zaawansowanych technik.

2.5.6 Post-Processing

Ostatnim elementem, aczkolwiek nieobowiązkowym, występującym w procesie renderingu są efekty nakładane na obraz po wyrenderowaniu klatki wideo. Zapisując wygenerowany obraz do tekstury, możemy nałożyć na niego różne filtry zaimplementowane za pomocą kolejnego wywołania programu pixel shader. Taki shader przyjmuje na wejście koordynaty tekstury dla każdego piksela i zwraca nowy obliczony kolor. Rysunek 2.17 pokazuje przykładowe działanie filtrów. Na pierwsze dwa zdjęcia został nałożony filtr



Rysunek 2.17: Przykładowe filtry post-processing'owe

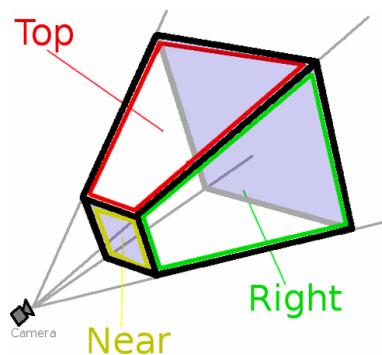
rozmazujący/rozmywający. Dwa pozostałe to efekt starego uszkodzonego filmu oraz efekt sepii.

2.5.7 Optymalizacja

Współczesne karty graficzne pozwalają na wykonanie wszystkich opisanych powyżej czynności w odpowiednim czasie. Czasem jednak ta moc nie wystarcza, dlatego istotnym elementem procesu renderingu obrazu są techniki optymalizacyjne. Autor prezentuje poniżej dwie z nich: Frustum Culling oraz Occlusion Culling.

2.5.7.1 Frustrum Culling

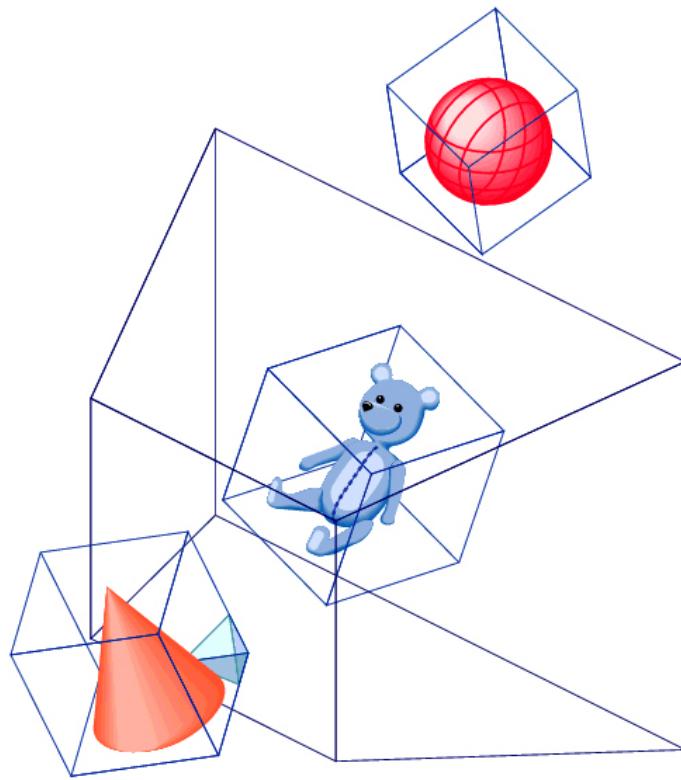
Frustum Culling to technika obcinania bryłą widzenia. Jest wykonywana w kodzie aplikacji, jeszcze przed rozpoczęciem procesu renderingu. Pierwszym etapem jest określenie płaszczyzn budujących bryłę widzenia powstałą podczas tworzenia macierzy projekcji.



Rysunek 2.18: Budowa bryły obcinania

Przed renderowaniem obiektu należy sprawdzić czy zawiera się on wewnętrz powstałej figury (lub przecina ją). Ponieważ sprawdzanie prawdziwych modeli byłoby bardzo nieefektywne, dla każdego tworzy się bryłę otaczającą: sześcian lub kulę.

Na rysunku 2.19 pokazano 3 obiekty podlegające sprawdzeniu. Model maskotki w całości zawiera się w bryle. Sześciyan otaczający stożek przecina przednią płaszczyznę bryły widzenia, więc również zostanie wyświetlony. Piłka natomiast jest poza zasięgiem widzenia i zostanie odrzucona.

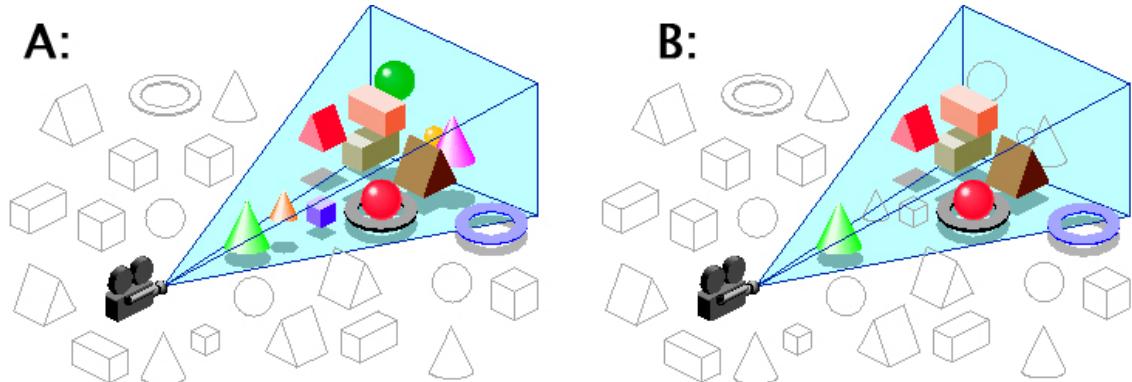


Rysunek 2.19: Wykrywanie widocznych obiektów

2.5.7.2 Occlusion Culling

Frustrum culling daje zadowalający efekt w postaci odrzucenia większości otaczających efektów. Istnieją jednak sytuacje, w których to nie wystarcza. Prostym przykładem jest scena, na której przed kamerą stoi duży dom, a za nim ciągnie się pasmo drzew, budynków i innych szczegółowych obiektów. Renderowanie ich zajęło by dużo czasu, podczas gdy na wynikowym obrazie nie byłyby widoczne. Rozwiązaniem tego problemu jest Occlusion Culling, czyli obcinanie niewidocznych obiektów. Aby tego dokonać należy wyrenderować bryły otaczające do tymczasowego obrazu, a następnie sprawdzić ilość widocznych pikseli każdego obiektu. Jeśli ich suma wynosi zero, obiekt można wyłączyć z procesu renderingu.

Rysunek 2.20 przedstawia scenę przed i po obcinaniu.



Rysunek 2.20: A: Frustrum Culling, B: Occlusion Culling

2.6 Techniki wyświetlania obrazu

Niniejszy rozdział przedstawia techniki uatrakcyjniające wyświetlanie obrazu. Pierwszą z nich jest obsługa świateł w przestrzeni 3D i ich nakładanie na powierzchnię modelu. Zostaną opisane trzy podstawowe metody cienowania, jak i różne typy źródeł światła. Drugim etapem jest nakładanie tekstuury oraz przedstawienie informacji dotyczących filtrowania tekstuur i efektu mapowania wypukłości.

2.6.1 Oświetlenie

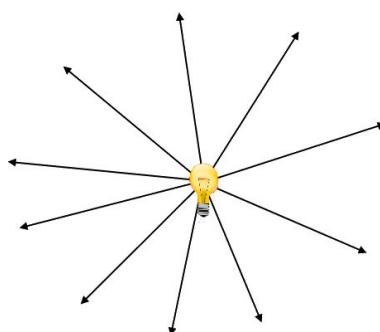
2.6.1.1 Wstęp

Najważniejszym zadaniem programistów grafiki jest zapewnienie jak największego realizmu prezentowanego obrazu. Aby to zapewnić implementuje się modele oświetlenia. Zaawansowane techniki świetlne takie jak raytracing¹⁷ są zbyt złożone obliczeniowo, aby można je było stosować w aplikacjach czasu rzeczywistego. W grach komputerowych najczęściej stosuje się metody cienowania: płaskie, Gouranda i Phonga. Różnią się one ilością obliczeń, odpowiednio dla trójkątów, wierzchołków i pojedynczych pikseli.

2.6.1.2 Źródła światła

Wyróżniamy dwa typy źródeł światła:

1. **Otaczające** – Obiekt jest oświetlony ze wszystkich stron jednocześnie. Światło to nie posiada kierunku ani pozycji, jedynie kolor oraz natężenie z jakim pada na obiekt.
2. **Kierunkowe** – Światło posiadające określony kierunek i pozycję, choć czasami ma znaczenie tylko jedno z nich. Przykładowe warianty:
 - Żarówka jest przykładem światła punktowego¹⁸ (rysunek 2.21). Posiada pozycję oraz świeci we wszystkie strony, dlatego kierunek może być wyliczany dynamicznie jako wektor o początku: pozycji światła i końcu: pozycji obiektu.

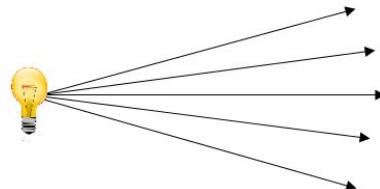


Rysunek 2.21: Point light

¹⁷Metoda śledzenia promieni

¹⁸ang: point light

- Latarnia uliczna lub latarka w praktyce jest żarówką (światło punktowe), które odbija się od zwierciadła, kierując promień światła w określoną stronę. Obliczenia towarzyszące temu zjawisku są zbyt czasochłonne w przypadku aplikacji czasu rzeczywistego, dlatego stosuje się światło reflektorowe (rysunek 2.22), które zawiera pozycję i kierunek, a czasami też kąt, który określa zakres światła (jego granicę, po której przekroczeniu na obiekt nie padają już promienie).



Rysunek 2.22: Reflect light

- Słońce, mimo że jest przykładem światła punktowego, w praktyce jest tak duże, że przyjmuje się stały kierunek padania promieni słonecznych, ignorując jego pozycję. Dzięki temu każdy obiekt oświetlany jest pod takim samym kątem.

Mając określone źródło światła należy nim odpowiednio potraktować oświetlany obiekt. Proces ten nazywamy cieniowaniem.

2.6.1.3 Cieniowanie płaskie

Cieniowanie płaskie to najprostsza metoda cieniowania. Polega na obliczeniu jasności powierzchni jednego trójkąta w zależności od kąta zawartego między wektorem normalnym danej powierzchni oraz wektorem światła. Mimo że obliczenia są szybkie, ponieważ są wykonywane raz dla każdego trójkąta, nie dają satysfakcjonujących efektów. Poważną wadą jest jeden odcień każdej ścianki i widoczne przejścia (krawędzie) między kolorami.

2.6.1.4 Cieniowanie Gourauda

W przeciwieństwie do cieniowania płaskiego, cieniowanie Gourauda wymaga policzenia natężenia światła dla trzech wierzchołków każdego trójkąta. Następnie na zasadzie interpolacji liniowej wyliczane są wynikowe kolory pikseli, które mieszają się z właściwym kolorem fragmentu obiektu. Dzięki temu zyskuje się płynny obraz przejścia między kolorami.

2.6.1.5 Cieniowanie Phonga

Cieniowanie Phonga jest najlepszą z trzech metod cieniowania, ale też najbardziej złożoną obliczeniowo. O ile dwie poprzednie operowały na trójkątach/wierzchołkach (obliczenia wykonywane przez vertex shader), ta oblicza natężenie światła wykorzystując interpolowany wektor normalny dla każdego piksela (pixel shader). Dzięki temu otrzymujemy dokładne odwzorowanie efektów takich jak odbicie światła, które w przypadku cieniowania Gouraud

bywa zniekształcone lub niewyraźne.

Rysunek 2.23 przedstawia wszystkie 3 metody cieniowania.

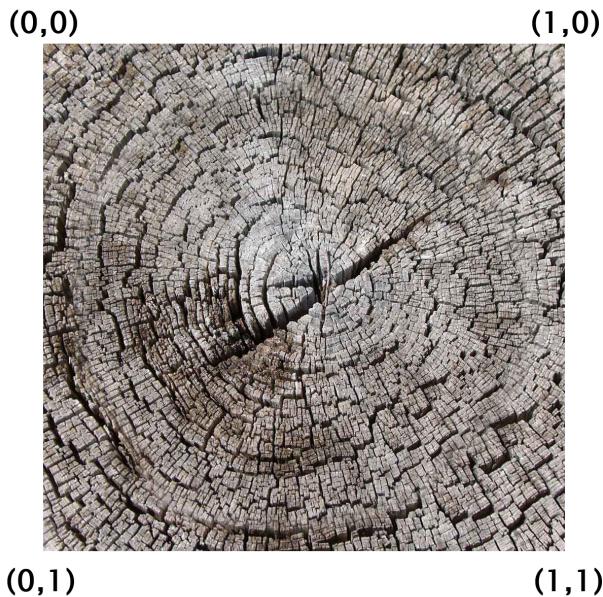


Rysunek 2.23: Od lewej: cieniowanie płaskie, Gourauda, Phonga

2.6.2 Teksturowanie

2.6.2.1 Nakładanie tekstur

Mimo, że oświetlenie daje pewną dawkę realizmu, dopiero teksturowanie zapewnia rzeczywisty wygląd. Tekstura jest plikiem graficznym, który jest nakładany na model według ustalonych na każdym wierzchołku koordynatów UV. Są to współrzędne X i Y tekstyury, które mieszczą się w przedziale od 0 do 1, począwszy od punktu (0, 0) w lewym górnym rogu i (1, 1) w prawym dolnym (rysunek 2.24). W procesie rasteryzacji koordynaty są interpolowane, określając które fragmenty tekstyury przypadają na poszczególne piksele.



Rysunek 2.24: Układ koordynatów UV tekstyury

Dzięki współrzędnym tekstyury dla bardzo złożonych modeli można przygotować jedną teksturę i za pomocą odpowiednio ustawionych koordynatów "wyciągać" poszczególne fragmenty z jednego pliku graficznego.

2.6.2.2 Filtrowanie tekstur

Rzadko się zdarza, że wielkość tekstury odpowiada dokładnej wielkości obiektu na który jest ona nakładana. W takim przypadku obraz trzeba powiększyć lub pomniejszyć. Proces ten jest nazywany filtrowaniem.

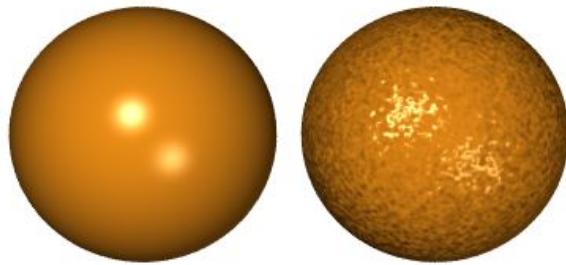
Poniżej zostały opisane 3 sposoby filtrowania:

1. **Point (próbkowanie punktowe)** – Jest to najprostsza metoda zmiany rozmiaru tekstury, jednak najmniej dokładna i najrzadziej używana. Obraz jest zwyczajnie rozciągany lub zmniejszany. Przy zwiększeniu np. czterokrotnym każdy piksel tekstury zajmuje cztery teksele¹⁹ tekstury wyjściowej. Przy zmniejszaniu czterokrotnym zostaje brany pod uwagę co czwarty piksel. Próbkowanie punktowe często powoduje efekt pikseloży, czyli widoczne pojedyncze teksele tekstury i brak łagodnych przejść między nimi.
2. **Linear (filtrowanie liniowe)** – W tej metodzie podczas rozciągania lub zmniejszania obrazu dla każdego piksela obliczana jest średnia z czterech przylegających pikseli. Dzięki temu zyskuje się łagodne przejście między kolorami i nie występuje efekt pikseloży. Mniejsze obrazy stają się rozmażane, jednak jest to lepszy efekt od ostrych w przypadku próbkowania punktowego.
3. **MipMapping** – Mipmapping to technika polegająca na tworzeniu mniejszych lub większych wersji tekstury w zależności od odległości obiektu od obserwatora. Przykładowo obiekt mający wymiary 13x15 wymaga tekstury o wymiarach 16x16, a większy, którego wymiary to 34x54 potrzebuje tekstury 64x64. Dla obu przypadków odwoływanie się do najwyższej tekstury nie ma sensu, ponieważ spora część pikseli i tak nie zostanie wyświetlona. Zaletą mipmappingu jest wstępne usuwanie zakłóceń skalowanych tekstur (np metodą filtrowania liniowego) oraz zwiększenie prędkości teksturowania. Wadą jest większa ilość pamięci potrzebnej do przechowywania skalowanych wersji tekstury. Mipmapping jest podstawą dla dwóch kolejnych metod filtrowania:
 - **Trójliniowe** – Jest to rozszerzenie filtrowania liniowego, polegające na uwzględnieniu mipmap oraz zamazywaniu granic między nimi przez interpolację między kolejnymi poziomami mniejszych lub większych wersji tekstury w zależności od odległości.
 - **Anizotropowe** – O ile filtrowanie trójliniowe jest z powodzeniem wykorzystywane w większości przypadków - nie sprawdza się np. przy wyświetlaniu oteksturowanego płaskiego terenu ciągnącego się wgłąb ekranu. Filtrowanie anizotropowe rozwiązuje ten problem uwzględniając (oprócz pozycji) kąt patrzenia kamery. Dzięki temu filtrowanie jest intensywniejsze w głąb ekranu.

¹⁹Najmniejszy punkt tekstury wyświetlany na ekranie

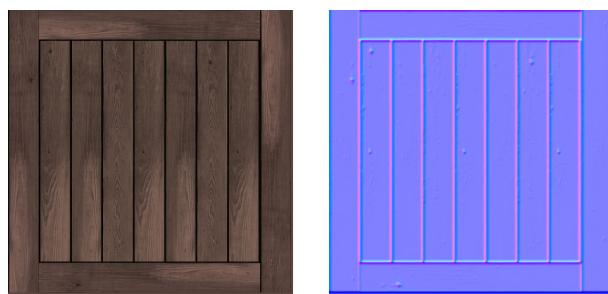
2.6.2.3 Mapowanie normalnych

Tworzenie bardzo szczegółowych modeli, mimo że nadają one większego realizmu grze, w większości przypadków jest kłopotliwe, ponieważ znaczco zwiększa czas renderingu i utrudniają np. wykrywanie kolizji (lub nie są do tego wymagane). Mapowanie normalnych²⁰ jest techniką, która umożliwia sztuczną wizualizację wypukłości lub wklęsłości na powierzchni przy użyciu odpowiedniej gry światel. Rysunek 2.27 pokazuje działanie tego efektu. Obie kule są reprezentowane przez ten sam model. Dzięki technice mapowania normalnych na prawej zostały sztucznie wymodelowane wypukłości.



Rysunek 2.25: Model przed i po mapowaniu

Technika ta polega na zastępowaniu wektorów normalnych prostopadłych do powierzchni wektorami zapisanymi na tzw. normal mapie, czyli pliku graficznym, którego każda składowa koloru RGB odpowiada współrzędnym nowego wektora normalnego. Należy zaznaczyć, że cieniowanie obiektu musi odbywać się w pixel shaderze, w którym oprócz pobranego koloru tekstuury pobiera się kolor z normal mapy, a następnie wykorzystując nowy wektor wykonywane są pozostałe obliczenia.



Rysunek 2.26: Przykład tekstuury i normal mapy

²⁰Normal Mapping

2.7 Fizyka w grach komputerowych - biblioteka PhysX

Niniejszy rozdział ma na celu przedstawienie silnika fizycznego PhysX firmy NVIDIA, służącego do tworzenia środowiska fizycznego, zapewniającego grom komputerowym zbliżonej do rzeczywistości dynamiki obiektów.

2.7.1 Wstęp

Gry komputerowe z roku na rok stają się ładniejsze, wykorzystując najnowsze możliwości kart graficznych, zapewniając gracjom jak najwierniejsze odwzorowanie rzeczywistego wyglądu. Ważnym elementem, obok grafiki, jest dynamika gry. Składają się na nią głównie:

- **Sterowanie** – Sposób poruszania postacią
- **Dynamika obiektów** – Działanie na obiekty sił fizycznych takich jak grawitacja, wiatr, tarcie itp.
- **Wykrywanie kolizji** – Wykrywanie kolizji obiektów oraz odpowiednie reagowanie w przypadku wystąpienia zderzenia

Wszystkie te trzy elementy sprowadzają się do implementacji praw fizyki w grach komputerowych. Dokładne odwzorowanie zjawisk fizycznych jest trudne i bardzo złożone obliczeniowo. Dla każdego obiektu posiadającego własności fizyczne takie jak masa i wielkość, w zależności od działających na niego sił, należy obliczyć nową pozycję, prędkość, kierunek, tarcie, oraz inne wymagane dla danego środowiska właściwości. Wymaga to wielu zaawansowanych obliczeń wykonywanych w czasie rzeczywistym.

W większości przypadków odpowiedzialnym za obliczenia fizyczne jest procesor CPU. Nie jest on wystarczająco szybki aby dokładnie odwzorować środowisko fizyczne, dlatego stosuje się uproszczenia wzorów, które znacznie przyśpieszają obliczenia i jednocześnie zmniejszają realizm działania gry.

Technologia PhysX firmy NVIDIA przenosi obliczenia związane z fizyką na jednostki obliczeniowe karty graficznej. Jest to wydajny silnik fizyczny pozwalający na tworzenie wirtualnego środowiska i symulację zjawisk fizycznych dzięki wielordzeniowym procesorom umieszczonym na karcie graficznej, zwiększający wykładniczo moc obliczeniową potrzebną do przetwarzania fizyki.

Silnik fizyczny PhysX wykorzystując sprzętową akcelerację karty graficznej umożliwia modelowanie realistycznych zjawisk fizycznych takich jak:

- Wybuchy i eksplozje razem z generowaniem dymu i szczątek rozbitych elementów
- Ciecze, które posiadają gęstość i objętość

- Efekty atmosferyczne takie jak realistyczna mgła, dym, deszcz, które wchodzą w interakcje z otoczeniem
- Materiały, które reagują na wiatr i czynniki zewnętrzne (możliwe np. rozdarcie materiału, deformacja)
- Dodatkowe wsparcie dla kół pojazdów, pozwalające na tworzenie bardziej realistycznych symulacji samochodowych
- Zaawansowana obsługa kolizji, wraz z generowaniem zdarzeń

2.7.2 Scena

Scena jest środowiskiem, w którym przeprowadzana jest symulacja fizyczna środowiska. Ustawia się w niej parametry, między innymi:

- Grawitacja, najczęściej 9.8 m/s dla symulacji warunków ziemskich.
- Grubość skóry obiektów, która jest ustawiana dla każdego nowego obiektu sceny. Określa ona w jakim stopniu obiektu się przenikają.
- Typ symulacji, czyli sposób w jaki PhysX będzie dokonywać obliczeń: sprzętowo (wykorzystany będzie procesor karty graficznej) i programowo (użycie procesora CPU).

2.7.3 Aktorzy

Aktorem określa się obiekt fizyczny posiadający właściwości fizyczne takie jak masa, kształt, położenie, materiał z jakiego jest zbudowany itd. Aktorzy dzielą się na dwie grupy:

- **Dynamiczni** – Na obiekty dynamiczne działają siły oraz są wykrywane między nimi (a także między obiektem dynamicznym i statycznym) kolizje. Przykłady: piłka, deska, samochód.
- **Statyczni** – Są to obiekty ze stałą pozycją, kolizje między nimi nie są wykrywane. Przykłady: budynek, teren.

Pierwszym krokiem w celu stworzenia aktora jest określenie materiału. Ustawia się w nim między innymi następujące własności:

- **Restitution (elastyczność)** – Sklonność obiektu do odbijania się podczas kolizji.
- **Static Friction (tarcie statyczne)** – Sposób, w jaki obiekt zachowuje się po działaniu na niego siły. Ustawienie wysokich wartości sprawia, że obiekt będzie bardziej skłonny do toczenia się aniżeli do przesuwania.
- **Dynamic Friction (tarcie dynamiczne)** – Określa opór, który stawia obiekt w momencie przesuwania po powierzchni. Wyzsza wartość powoduje większe zapotrzebowanie na siłę w trakcie przesuwania.

Kolejnym krokiem jest utworzenie kształtu aktora. Służy on do fizycznej reprezentacji obiektu w symulacji. Na podstawie kształtów wykrywane są kolizje.

- Prostopadłościan, Kula i Kapsuła to podstawowe kształty. Z nich najczęściej buduje się bardziej złożone. Dla przykładu z kilku kapsułek można wymodelować ręce, nogi i ciało człowieka, natomiast do reprezentacji głowy użyć boksa lub kuli.
- Płaszczyzna reprezentuje płaską przestrzeń. Najczęściej jest to "podłoga" symulacji, która oddziela część pustą (dół) od pozostałą, w której znajdują się obiekty.
- Convex Mesh służy do modelowania bardziej złożonych obiektów, których nie da się w prosty sposób zbudować z wymienionych powyżej kształtów. Mimo dowolnego wypukłego kształtu ma on ograniczenie do 256 poligonów.

Powysze kształty stosuje się zarówno do dynamicznych jak i statycznych obiektów, ponieważ zachodzi między nimi możliwość wykrycia kolizji. Poniższe stosuje się do modelowania obiektów statycznych, ponieważ wykrycie między nimi kolizji nie jest możliwe:

- Triangle Mesh pozwala na dokładną reprezentację modelu np. budynku.
- Heightfield jest zoptymalizowany pod kątem reprezentacji terenu.

Po utworzeniu kształtu obiektu i przypisaniu mu materiału należy ustawić pozostałe właściwości obiektu takie jak: masa (lub gęstość), pozycja oraz orientacja startowa.

2.7.4 Wykrywanie kolizji

Sprawdzenie kolizji między obiektami na zasadzie każdy z każdym przy dużych scenach było by zbyt wymagające obliczeniowo, a także w większości przypadków niepotrzebne. PhysX optymalizuje to dzieląc świat na mniejsze przestrzenie, dzięki temu każdy obiekt jest sprawdzany pod kątem wystąpienia kolizji jedynie z najbliższymi obiektami w jego otoczeniu.

Po ustaleniu pary obiektów będących dostatecznie blisko siebie następuje sprawdzenie typów obiektów (co najmniej jeden z nich musi być dynamiczny), a następnie grup kolizyjnych, którymi użytkownik może kontrolować wykrywanie kolizji między określonymi obiektami.

Po wykryciu kolizji generowane jest zdarzenie (raport), dzięki któremu programista może obsłużyć wystąpienie kolizji. Można również zdefiniować zdarzenia, które mają być raportowane. Przykładowo jeśli chcemy aby PhysX poinformował nas tylko o uderzeniu obiektu w podłogę, ustawiamy mu flagę NX_NOTIFY_ON_START_TOUCH. Pozostałe zdarzenia, takie jak przesuwanie lub oderwanie od podłogi nie będą zgłasiane.

Pewną odmianą raportów kolizji są tak zwane **triggery**. Działają one na zasadzie generowania informacji o wejściu w przestrzeń określonego obiektu, przebywaniu w niej oraz opuszczeniu. Obiektem takim może być dowolny kształt opisany w poprzednim rozdziale, zaznaczając, że obsługa kolizji convex mesh i triangle mesh, oraz triangle mesh i triangle mesh, nie są obsługiwane. Przykładowym zastosowaniem triggerów jest tworzenie systemu automatycznego otwierania drzwi po wejściu na schodek.

2.7.5 Sterowanie obiektami

Sterowanie obiektami różni się dla obiektów dynamicznych i statycznych.

Obiekty dynamiczne to ciała fizyczne, dla których w trakcie działania symulacji nie można bezpośrednio zmienić pozycji ani orientacji. Działają na nie siły fizyczne, dlatego sterowanie polega głównie na nadawaniu im sił z dowolnego punktu i w dowolnym kierunku. Przykładem takiego rozwiązania jest toczenie kuli. Początek siły zaczepiony jest w środku obiektu, natomiast koniec skierowany jest w stronę, w którą kula ma się przetoczyć. Wartym dodania jest fakt, że siłę można zaczepić nie tylko w lokalnym układzie obiektu, ale też w globalnych współrzędnych świata.

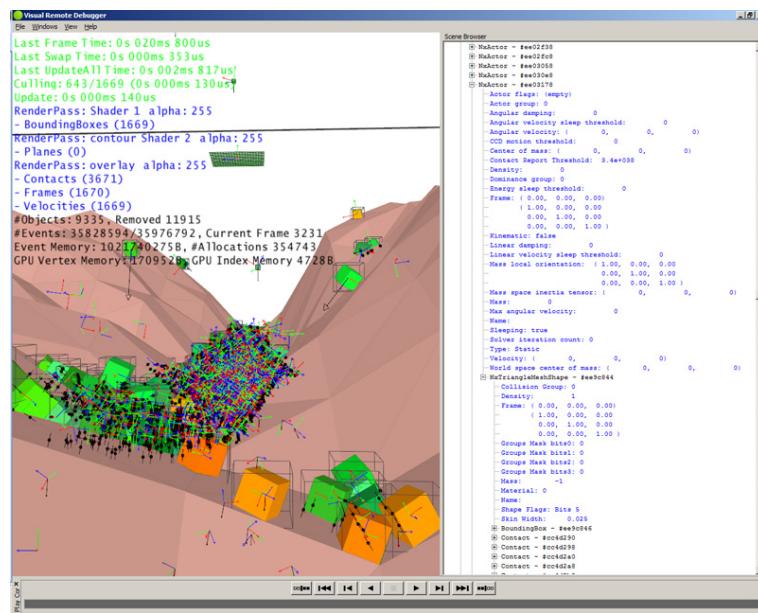
Obiekty statyczne to obiekty na które nie działają żadne siły. Daje to możliwość dynamicznej zmiany pozycji i orientacji aktora poprzez odpowiednie wywołanie funkcji. To rozwiązanie stosuje się do kierowania postaciami, które przez cały czas zachowują pionową postawę, lub w przypadku gier FPP First Person Perspective, w których poruszamy się po świecie oglądając go oczami bohatera (sterując kamerą). Ręczna implementacja takiego zachowania jest możliwa, jednak wiąże się z wieloma trudnościami. PhysX wychodząc na przeciw zapotrzebowaniu udostępnia narzędzie, które to ułatwia, czyli Character Controller. Pozwala on na stworzenie obiektu, który można w prosty sposób poruszać. Udostępnia również dwie właściwości, które określają możliwości przemieszczania się obiektu na danym terenie:

- **Auto Stepping** – Umożliwia ustawienie różnicy w wysokości wybojów, które obiekt pokona bez problemu, a na których się zatrzyma. Przykładem są schody, które postać powinna pokonać automatycznie, oraz wysoki podium, na który trzeba wskoczyć.
- **Walkable Parts** – Domyslnie obiekt można przemieścić gdziekolwiek. Nie jest to zgodne z rzeczywistością. Ta opcja pozwala na ustalenie limitu pochyłości w kątach, do którego obiekt może swobodnie poruszać się, a po przekroczeniu dalsze sterowanie w danym kierunku staje się niemożliwe. Przykładem jest górzysty i wyboisty teren. Postać porusza się swobodnie po pochyłych wzgórzach, których kąt jest mniejszy od 45 stopni.

2.7.6 Debugowanie symulacji

PhysX Visual Debugger jest narzędziem, które nie tylko pozwala na kontrolowanie symulacji fizycznej, ale także pomaga w przypadku rozwiązywania problemów. Główne zalety PVD:

- Wizualizacja wszystkich obiektów na scenie, sił, kolizji, pól siłowych i innych
- Kontrolowanie obiektów oraz sił działających na nie za pomocą lewego przycisku myszki
- Dostęp do wszystkich informacji i właściwości każdego obiektu
- Swobodna kamera umożliwiająca podgląd sceny pod różnym kątem



Rysunek 2.27: Podgląd sceny w programie PhysX Visual Debugger

2.7.7 Podsumowanie

Fizyka jest niemalże obowiązkowym elementem każdej gry. Jak najwierniejsze odwzorowanie zjawisk fizycznych jest nie tylko przyjemne dla oka, ale również odczuwalne podczas sterowania głównym obiektem gry. Technologia NVIDIA PhysX pozwala twórcom gier na stworzenie wydajnego środowiska fizycznego, dzięki czemu mogą się oni skupić głównie na logice aplikacji.

2.8 Logika gry - skrypty

2.8.1 Wstęp

Małe gry ze zdefiniowanym celem i fabułą wykorzystują niewielką ilość modeli i grafik. W takich produkcjach programiści programują poziomy bezpośrednio w kodzie aplikacji. W większych przedsięwzięciach zachodzi konieczność oddzielenia części logicznej od kodu aplikacji z dwóch powodów:

- Tworzeniem poziomów zajmuje się oddzielny zespół, złożony z członków, którzy najczęściej nie zajmują się programowaniem
- Każda, nawet najmniejsza zmiana fragmentu logiki, wiąże się z ponowną komplikacją całej aplikacji

Wykorzystanie języka skryptowego rozwiązuje ten problem. Twórcy gry nie tylko mogą umieścić w nich budowanie poziomów, ale także implementować nowe algorytmy AI, oprogramowywać GUI²¹, i wiele innych aspektów gry, które wymagają częstych zmian i szybkiego sprawdzenia w praktyce.

Skrypty są pisane w języku interpretowanym. Oznacza to, że instrukcje są czytane po kolej i wykonywane, bez procesu komplikacji. W dawnych czasach jedynym wyjściem była własna implementacja języka skryptowego i interpretera. Dzisiaj twórcy gier mają do dyspozycji wiele gotowych. W projekcie praktycznym został wykorzystany język Lua, dlatego też właśnie on zostanie przedstawiony w niniejszym rozdziale.

2.8.2 Język Lua

Lua to język skryptowy przeznaczony głównie do rozszerzania funkcjonalności aplikacji, jednak często jest stosowany jako samodzielny język. Stworzony został w 1993 roku przez trzech studentów katolickiego uniwersytetu w Rio de Janeiro.

Język ten został zaimplementowany w ANSI C jako mała, darmowa i przenośna biblioteka. Głównymi atutami są prostota i wydajność. Właśnie te cechy sprawiają, że język Lua jest najczęściej wybieranym językiem dla potrzeb gier komputerowych.

2.8.3 Składnia języka

Opis całej składni języka wykracza poza ramy tej pracy. Autor przedstawia jedynie podstawowe elementy języka.

²¹ang: Graphical User Interface - Interfejs użytkownika

Kod skryptów Lua przypomina język Pascal. Zawiera jednak automatyczne mechanizmy zarządzania pamięcią, dzięki którym znikają problemy związane z wyciekami pamięci czy przepełnieniami buforów. Jest to język dynamicznych typów, co oznacza, że typ zmiennych oraz ich instancje są tworzone przy pierwszym przypisaniu do nich wartości:

```
--Prosty komentarz
--[[[
    Wieloliniowy komentarz
]]--
zmienna = 5
zmienna = false
zmienna = "Hello!"
```

Wydruk 2.3: Dynamiczna zmiana typu zmiennej

Podstawowe typy danych to: nil, boolean, number, string, funkcja, userdata i tablica. Nil jest typem z jedną wartością nil, która reprezentuje brak wartości. Typ logiczny jaki reprezentuje Nil to false. Typ userdata jest przydatny podczas komunikacji Lua z innymi językami i może przechowywać dowolną wartość. Tablice są dynamicznym asocjacyjnym zbiorem danych, który może przechowywać dane dowolnego typu, z wyjątkiem wartości nil. Tablice są podstawą budowania bardziej złożonych typów takich jak mapy, listy, grafy, drzewa itp. Dostępne są podstawowe instrukcje warunkowe i sterujące: if, while, repeat, repeat-until i for.

Wyróżniamy dwa rodzaje zmiennych: globalne i lokalne. Zmienne globalne są domyślne i dostępne w całym kodzie. Zmienne lokalne wymagają poprzedzenia deklaracji słowem local i ograniczają się jedynie do zakresu bloku, w którym się znajdują:

```
zmiennaGlobalna = 5
if zmiennaGlobalna == 5 then
    local zmiennaLokalna = 25
end
print zmiennaGlobalna — Wyświetli wartość 5
print zmiennaLokalna — Błąd, zmienna nie istnieje
```

Wydruk 2.4: Zasięg zmiennych

Funkcje w języku Lua mogą przyjmować i zwracać wiele wartości różnego typu. Parametry funkcji, którym nie zostanie podana wartość, przyjmują wartość nil:

```
function s(a, b)
    if a > 2 then
        return a+b, a*b
    else
        return a-b, a/b
    end
end
e, f, g = s(3, 1)
— e = 4, f = 3, g = nil
```

```
h = s(1, 3)
-- h = -2
```

Wydruk 2.5: Parametry funkcji

Język Lua nie ma wbudowanej obsługi obiektów i klas. Jedyną strukturą danych jest Tabela, jednak używa się jej do implementowania innych typów za pomocą meta-tablic²². Umożliwiają one zmianę zachowania tablic. Prostym przykładem może być tablica przechowująca wektor 3d, mający trzy pola reprezentujące współrzędne. Podczas operacji dodawania dwóch takich obiektów Lua sprawdza, czy istnieje meta-metoda o nazwie `_add`, którą programista może zaimplementować w celu obsłużenia dodawania dwóch wektorów.

2.8.4 Integracja z językiem C++

Biblioteka Lua umożliwia wymianę informacji oraz wywoływanie funkcji Lua w kodzie C++ i odwrotnie. Podczas wymiany informacji tworzony jest wirtualny stos, na którym umieszczane są parametry funkcji i wartości zwarcane. Dla każdego wywołania tworzony jest oddzielny stos. Dzięki temu wywołując z poziomu C++ funkcję Lua, która w swoim ciele wywołuje inną funkcję C++, mamy za każdym razem dostęp do osobnego stosu każdego z wywołań i możemy je obsłużyć bez żadnych kolizji. Do wartości przechowywanych na stosie możemy odwołać się używając dwóch sposobów: indeksy dodatnie pobierają wartości od dołu (od najstarszego elementu), indeksy ujemne pobierają wartości od góry. Proces wywołania funkcji ze skryptu składa się z:

1. Umieszczenia parametrów funkcji na stosie
2. Wywołania funkcji ze skryptu
3. Pobrania zwracanych przez funkcję wartości ze stosu

Z poziomu C++ możemy również uzyskać dostęp do wszystkich zmiennych globalnych w skrypcie.

Wywoływanie funkcji języka C++ w skrypcie jest możliwe poprzez rejestrację statycznej metody pośredniczącej. Jej parametrem jest wskaźnik do obiektu, za pomocą którego mamy dostęp do metod umieszczających i pobierających informacje ze stosu.

Łączenie skryptówLua z językiem C++ jest niekiedy trudnym zadaniem, dlatego stworzone zostały zewnętrzne biblioteki wspomagające wymianę informacji, a także wprowadzające ulepszenia takie jak obsługa programowania obiektowego oraz dostępu do klas i obiektów kodu C++ z poziomu skryptu.

²²ang: metatable

2.8.4.1 ToLua++

ToLua++ jest biblioteką, która umożliwia eksportowanie klas z C++ za pomocą definicji klasy umieszczonej w osobnym pliku. W skrypcie można korzystać z tych klas tak jakby były fizycznie klasami Lua. Obsługiwany jest polimorfizm obiektów i klas, a także dziedziczenie jednokrotne i wielodziedziczenie. Możliwe jest także przeciążanie funkcji i operatorów.

2.8.4.2 LuaBind

O ile ToLua++ jest jedynie narzędziem wspomagającym integrację Lua z C++, LuaBind jest oddzielną wersją języka, która rozszerza jego możliwości, jednocześnie zapewniając wygodną formę rejestrowania klas i metod. Główną zaletą LuaBind jest uniezależnienie od stosu języka Lua, dzięki czemu możliwe jest wywoływanie bezpośrednich metod języka C++ bez potrzeby implementowania funkcji pośredniczących.

Rozdział 3

Praktyka

Część praktyczna pracy opisuje proces tworzenia projektu. Została podzielona na dwie części:

1. **Architektura silnika** – Opis architektury i działania silnika graficznego, na podstawie którego tworzona będzie gra, oraz zewnętrznych narzędzi wykorzystanych przy procesie tworzenia.
2. **Tworzenie gry AtBall** – Proces tworzenia właściwej gry.

3.1 Architektura silnika

3.1.1 Wstęp

Pierwszym krokiem podczas tworzenia projektu było stworzenie silnika graficznego, który umożliwiałby przygotowywanie prostych gier platformowych. Autor zdecydował się na jak najdokładniejsze oddzielenie procesu kreowania gry od kodu źródłowego aplikacji. W tym celu został wykorzystany zewnętrzny edytor świata, w którym twórca gry przygotowuje scenę ustawiając pozycję, orientację i skalę obiektów, a także ustawia tekstury. Mając gotowy układ modeli następuje przygotowanie odpowiednich skryptów, które określają zachowanie obiektów podczas gry. Główną funkcją jest możliwość obsługi kolizji obiektów, dzięki czemu budowanie fabuły gry staje się łatwe i przyjemne.

3.1.1.1 Narzędzia

Proces tworzenia projektu wymagał wykorzystania następujących narzędzi oraz bibliotek:

- **Microsoft Visual Studio** – Zintegrowane środowisko programistyczne umożliwiające tworzenie, testowanie i debugowanie zaawansowanych aplikacji (także aplikacji sieciowych oraz serwisów internetowych).

- **Visual Assist X** – Dodatek do Visual Studio poprawiający i rozszerzający wybrane funkcje środowiska. Główną cechą jest lepsza obsługa IntelliSense¹ i kolorowania składni. Dodatkowo ułatwia utrzymywanie porządku w kodzie źródłowym za pomocą refactoringu².
- **Microsoft DirectX** – Biblioteka wspomagająca wyświetlanie grafiki na ekranie.
- **NVIDIA PhysX** – Silnik fizyczny firmy NVIDIA umożliwiający stworzenie wirtualnej sceny, zapewniającej obiektom fizyczne oddziaływanie między sobą.
- **IrrEdit** – Darmowy edytor świata, umożliwiający tworzenie scen zawierających nie tylko modele 3D, ale również kamery, światła i dynamicznie generowane tereny. Wygenerowaną scenę można zapisać w wygodnym formacie XML.
- **IrrXML** – Darmowa biblioteka umożliwiająca parsowanie plików XML.
- **Lua** – Biblioteka zapewniająca obsługę skryptów napisanych w języku Lua.

3.1.2 Inicjalizacja

Niniejszy rozdział zawiera ogólnikową prezentację procesu inicjalizacji oraz działania silnika. Wybrane klasy i mechanizmy zostaną omówione dokładniej w dalszej części pracy.

3.1.2.1 Game - główny obiekt gry

Klasa `CGame` jest jądrem całego silnika. Odpowiada za inicjalizację poszczególnych modułów, obsługę głównej pętli oraz zarządzanie stanami gry. Wykorzystuje ona wzorzec projektowy **Singleton** (wydruk 3.1):

```
class CGame{
private:
    CGame();
public:
    ~CGame();
    static CGame& GetInstance(){
        static CGame inst;
        return inst;
    }
    (...) //pozostałe metody
}
#define Game CGame::GetInstance()
```

Wydruk 3.1: Singleton

Zapewnia on dostęp tylko do jednej instancji obiektu za pomocą metody `GetInstance()`, która zwraca referencję na obiekt. Konstruktor jest prywatny, tak więc ręczne stworzenie

¹Podpowiadanie składni

²Dynamiczna zmiana wybranych nazw zmiennych, klas, metod itd.

obiektu klasy `CGame` jest niemożliwe. Instrukcja preprocesora `#define` tworzy alias, dzięki któremu za pomocą nazwy `Game` możemy wywoływać poszczególne metody klasy `CGame`.

Uruchomienie aplikacji polega na wywołaniu metody `CGame::Run()` (wydruk 3.2).

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd) {
    int result = Game.Run(hInstance, nShowCmd);
    return result;
}
```

Wydruk 3.2: Uruchomienie gry

Metoda ta wykonuje dwie operacje. Pierwszą z nich jest zainicjowanie poszczególnych komponentów wymaganych do dalszego funkcjonowania silnika. W praktyce sprowadza się to do wykonania metody `X::Init()`, gdzie `X` jest aliasem do klasy, która jest singletonem, zbudowanym na wzór klasy `CGame`.

Proces inicjalizacji przebiega następująco:

1. Załadowanie skryptu konfiguracyjnego, który zawiera wybraną rozdzielcość oraz nazwę pierwszego poziomu gry.
2. Inicjalizacja obiektu `Window`, który tworzy okno programu.
3. Inicjalizacja obiektu `DirectX` tworzącego obiekty urządzeń wykorzystywanych w procesie renderingu. Ładowane są również trzy pliki efektów (shadery): normalny, z efektem mapowania wypukłości (normal mapping) i dla SkyBox'a.
4. Inicjalizacja systemu wejścia `Input` odpowiedzialnego za wykrywanie operacji wejściowych z klawiatury i myszy.
5. Inicjalizacja obiektów `PhysX` oraz `Scene` odpowiedzialnych za obsługę silnika NVIDIA PhysX, oraz zarządzanie obiektami w scenie.
6. Inicjalizacja pozostałych składników: `SoundMgr` odpowiedzialny za obsługę dźwięku, `Gui` zajmujący się wyświetlaniem grafiki 2D (interfejs użytkownika, komunikaty itp.) oraz `SkyBox` ładujący teksturę sześcienną i wyświetlający ją imitując sztuczne tło świata.

3.1.2.2 Stany

Stan określa sposób działania aplikacji, reakcji na operacje wejściowe użytkownika, oraz wyświetlanie obrazu na ekranie.

- **start1, start2** – Odpowiedzialne za wyświetlenie dwóch początkowych obrazów bezpośrednio po uruchomieniu aplikacji.

- **mainMenu** – Wyświetlenie menu gry oraz reakcja na zmianę i wybór konkretnej pozycji.
- **instructions** – Wyświetlenie obrazu zawierającego informacje na temat gry oraz sterowania.
- **loading, loading2** – Stan **loading** powoduje wyświetlenie ekranu informującego o procesie ładowania poziomu oraz zmianę na stan **loading2**, który wywołuje metodę `CGame::NextLevel()` ładującą nowy poziom.
- **game** – Symulacja obiektów na scenie oraz wyświetlenie ich na ekranie.
- **pause, softPause** – Pauza gry, w której następuje zatrzymanie symulacji i sterowania obiektemi. W stanie **pause** dodatkowo obraz sceny zastępowany jest przez obraz informujący o zatrzymaniu gry.
- **endGame, closing** – Stany odpowiedzialne za wyświetlenie informacji o zamykaniu aplikacji oraz usuwanie obiektów i kończenie działania programu.
- **error** – Wyświetlenie komunikatu o błędzie oraz zamykanie programu.

Przejścia między stanami zachodzą w trzech przypadkach:

1. Po upływie określonego czasu. Przykładem jest droga: start1 → start2 → mainMenu, w której stan zmieniany jest na kolejny po upływie określonej liczby sekund.
2. Po akcji użytkownika np. wciśnięcie przycisku ENTER przy wybranej opcji menu.
3. Ręczna zmiana stanu poprzez wywołanie metody `CGame::ChangeStatus(nowyStan)`.

3.1.2.3 Pętla gry

Po utworzeniu wymaganych urządzeń i przygotowaniu obiektów następuje uruchomienie pętli czasu rzeczywistego. Wykonuje się ona przez cały okres działania aplikacji. Przerwanie jej powoduje wywołanie metody `CGame::Clear()`, która jest odpowiedzialna za bezpieczne usuwanie urządzeń i obiektów.

Pętlę można podzielić ją na cztery części:

1. Sprawdzenie, czy nie został ustawiony stan **closing**. Jeśli tak – pętla jest przerywana.
2. Domyśle obsłużenie zdarzeń przez system operacyjny (np. maksymalizacja lub zmiana rozmiaru okna).
3. Uaktualnienie logiki aplikacji.
4. Wyświetlenie grafiki.

Dwie ostatnie operacje mogą spowodować błąd, z powodu którego dalsze poprawne działanie aplikacji może stać się niemożliwe. W takim przypadku również pętla jest przerywana (wydruk 3.3):

```
int CGame::Run( ... ) {
    (...) //inicjalizacja silnika
    MSG msg;
    while ( true ){
        if( status == closing ) break; //zamykanie aplikacji
        if ( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) ){
            if( msg.message == WM_QUIT ) break;
            TranslateMessage(&msg); DispatchMessage(&msg);
        }
        if( Update() == false ) break; //błąd logiki - przerywanie
        if( Render() == false ) break; //błąd wyświetlania - przerywanie
    }
    Clear(); return 0;
}
```

Wydruk 3.3: Pętla gry

3.1.2.4 Uaktualnianie logiki

Proces uaktualniania logiki składa się z dwóch głównych operacji:

1. Wywołanie metody `Update()` dla poszczególnych obiektów głównych.
2. Reakcja na zdarzenia wygenerowane przez użytkownika.

Pierwszą czynnością jest uaktualnienie obiektów niezależnych od aktualnego stanu gry:

- **FPSCounter** odpowiada za obliczanie różnicy czasu między dwiema iteracjami pętli gry oraz wyliczanie wartości FPS.
- **SoundMgr** uaktualnia system dźwięku (więcej o obsłudze dźwięku w rozdziale 3.1.8).
- **TimerMgr** zajmuje się uaktualnianiem czasu zegarów, oraz wykonywanie określonych akcji (więcej o zarządzaniu czasem w rozdziale 3.1.9).
- **Input** pobiera aktualny stan urządzeń wejściowych.

Następnie w zależności od ustawionego stanu gry następuje sprawdzenie wcisniętych klawiszy.

Dla stanu **game** sprawdzane jest wybranie pauzy gry oraz restartu poziomu. W zależności od tego czy użytkownik wybrał żądane działanie, stan gry jest zmieniany na **pause** lub **loading**. Dodatkowo sprawdzana jest reakcja na przycisk ESC, który powoduje przejście do stanu **mainMenu** oraz spacji, której wcisnięcie jest sygnalizowane poprzez wywołanie

metody w skrypcie danego poziomu.

W przypadku stanu **instructions** po wciśnięciu klawisza ESC następuje powrót do stanu **mainMenu**.

Kolejnym krokiem jest uaktualnienie graficznego interfejsu użytkownika (który zostanie opisany w rozdziale 3.1.3) uwzględniając ustalony stan gry.

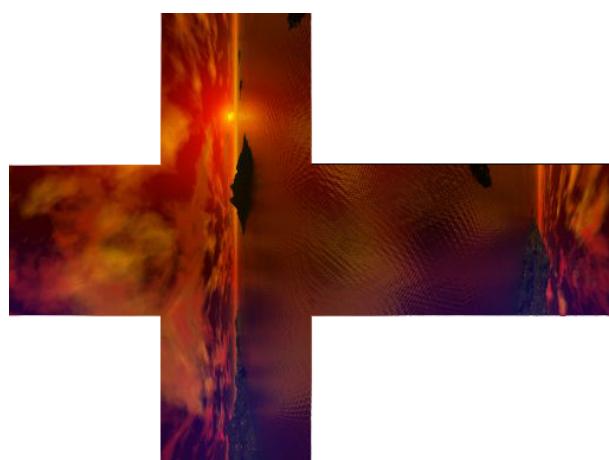
Jeśli aktualnym stanem gry jest **loading**, następuje przejście do stanu **loading2**, który przy kolejnej iteracji pętli gry załaduje nowy poziom.

Po wykonaniu powyższych operacji, jeśli stan gry dalej jest ustalony na **game**, następuje symulacja obiektów na scenie oraz uaktualnienie pozycji kamery.

3.1.2.5 Wyświetlanie obrazu

Wyświetlenie obrazu sprowadza się do wywołania metody `CDirectx::Render()`. Na początku następuje sprawdzenie czy urządzenie służące do wyświetlania obrazu nie zostało stracone. Taka sytuacja może wystąpić np. przy minimalizacji pełnoekranowej aplikacji. Jeśli tak, dokonywana jest próba odzyskania urządzenia. W przypadku powodzenia proces renderingu jest kontynuowany. Jeśli urządzenie dalej jest stracone, dalsze wykonywanie metody zostaje wstrzymane. Może zaistnieć również sytuacja, w której urządzenia nie da się odzyskać, wtedy działanie aplikacji zostaje zakończone.

Status **game** oznacza renderowanie ekranu gry. Pierwszym krokiem jest wyświetlenie skybox'a otaczającego całą scenę. Tym zadaniem zajmuje się klasa `CSkyBox`, która przy inicjalizacji silnika łada tekstuury z pliku DDS (rysunek 3.1) oraz tworzy model sześcianu. Następnie podczas renderowania nakładana jest na niego załadowana tekstura tworząc sześcian otaczający całą scenę.



Rysunek 3.1: Tekstura mapująca sztuczne tło świata

Następnie wyświetlane są obiekty ze sceny. Dla każdego typu shadera (normalny i mapowanie wypukłości) wywoływana jest metoda `CScene::Render(typ)` podając jako parametr typ shadera. Szczegółowy proces renderowania sceny oraz poszczególnych obiektów zostanie opisany w kolejnych rozdziałach.

Cała scena renderowana jest do tekstury w celu nałożenia efektów takich jak sepia, rozmycie lub zamiana kolorów na czarno-białe. Zostaje ona nałożona na kwadratowy model, który jest wyświetlany na powierzchni całego ekranu. Nazwa efektu, którą można ustawić za pomocą metody `CDirectx::SetPostProcessingEffect()`, jest nazwą techniki zaimplementowanej w pliku `post.fx`.

Ostatnim etapem jest wywołanie metody `CGui::Render()`, która w zależności od aktualnego stanu gry wyświetla odpowiednie elementy interfejsu użytkownika.

3.1.3 Grafika 2D - Interfejs użytkownika

Ważnym aspektem każdej gry, oprócz wyświetlonej grafiki 3D, są elementy dwuwymiarowe. Najczęściej są to okna (np. z wiadomością), przyciski (np. menu główne gry) i grafiki pełnoekranowe (zawierające informacje o ładowaniu poziomu lub pauzie gry). Za wyświetlanie obrazów odpowiada klasa `CGame`. Wykorzystuje do tego obiekt typu `ID3DXSprite`, który dodatkowo umożliwia rysowanie z uwzględnieniem głębi oraz przezroczystości.

Podczas inicjalizacji silnika ładowane są wszystkie wymagane tekstury. Rozmiar obrazów pełnoekranowych jest ustawiany na wysokość i szerokość okna (aktualna rozdzielcość). Następnie wybrane z nich są wyświetlane w zależności od ustawionego stanu gry.

3.1.3.1 Główne menu

Menu główne gry (rysunek 3.2) sprowadza się do narysowania następujących elementów:

1. Obraz pełnoekranowy stanowiący tło.
2. Ramka menu.
3. Przyciski.

Wymiary ramki są skalowane według proporcji, uwzględniając rzeczywisty wymiar tekstury, aktualną i maksymalną rozdzielcość ekranu aplikacji. Pozycja oraz środek obiektu (wymagane przy pozycjonowaniu obrazka na ekranie) są wyliczane uwzględniając obliczone wcześniej nowe wymiary obrazka (wydruk 3.4).

```
menuBackgroundX = (900 * Window.GetWidth()) / 1920;  
menuBackgroundY = (774 * Window.GetHeight()) / 1200;  
menuBackgroundPos = D3DXVECTOR3(Window.GetWidth() / 2, Window.GetHeight()  
/ 2, 0.0f);
```

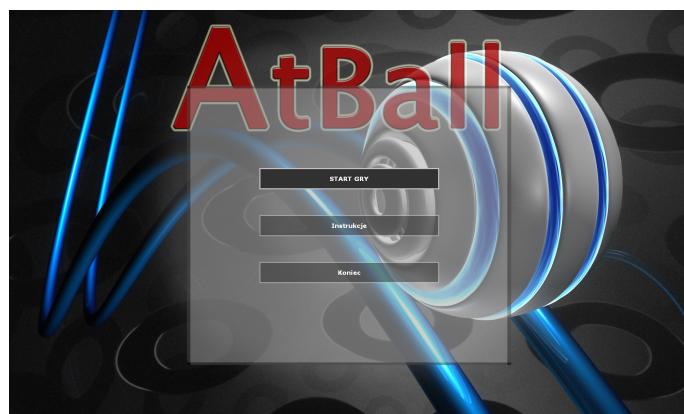
```
menuBackgroundCen = D3DXVECTOR3( menuBackgroundX / 2 , menuBackgroundY / 2 , 0.0 f  
);
```

Wydruk 3.4: Proporcjonalne skalowanie obrazków

Pozycja i wymiar przycisków jest wyliczany analogicznie, jednak brana jest pod uwagę również wartość przesunięcia obiektów w górę i w dół.

Działanie każdego przycisku określa klasa `CGuiButton`. Zawiera informacje takie jak:

1. Wyświetlany tekst na przycisku.
2. Informacja o zaznaczeniu. Wyróżniamy dwa stany:
 - Wciśnięty - przycisk jest rysowany nieprzezroczysty.
 - Zwolniony - dodawana jest przezroczystość.
3. Akcja, która zostanie wykonana w przypadku wciśnięcia ENTER na wybranym przycisku. W praktyce jest to po prostu zmiana stanu gry. Dostępne są cztery możliwości:
 - **START** – Stan **loading**.
 - **RESUME** – Stan **game**.
 - **INSTRUCTIONS** – Stan **instructions**.
 - **EXIT** – Stan **endGame**.



Rysunek 3.2: Menu główne gry

Metoda `CGame::Update()` sprawdza czy aktualnym stanem gry jest **mainMenu**. Jeśli tak, następuje sprawdzenie wciśnięcia klawiszy "strzałka w górę" i "strzałka w dół", które umożliwiają wybór opcji w menu. Po wykryciu wciśniętego klawisz Enter następuje wykonanie akcji przypisanej do wybranego przycisku.

3.1.3.2 Okno wiadomości

Okno wiadomości jest wyświetlane w stanie **game** i **softPause**. Jego treść jest ustawiana za pomocą metody `CGui::SetMessage()`. Proces rysowania polega na wyświetleniu przezroczystego ekranu okienka, oraz nałożenia tekstu wiadomości (rysunek 3.3). W przypadku ustawienia pustej wiadomości rysowanie okna zostaje pominięte.



Rysunek 3.3: Okno wiadomości w trakcie gry

3.1.4 Zasoby

Model siatki wierzchołków (mesh) jest przechowywany w klasie `CMesh`. Zawiera ona wskaźnik na właściwy model `ID3DXMesh`. Oprócz ładowania modelu z pliku wykonuje także dodatkowe operacje, takie jak wyliczanie wektorów normalnych dla modelu i optymalizacja wierzchołków. Dostępne są również metody przygotowujące kształt aktora triangle mesh i convex mesh, oraz funkcja skalująca model.

Klasa `CTexture` odpowiada za przechowywanie obiektu `IDirect3DTexture9` zawierającego załadowaną teksturę.

3.1.4.1 Menedżer zasobów

Tworzeniem oraz zarządzaniem modelami i teksturami zajmuje się klasa `CMedia`. Wykorzystuje ona wzorzec singleton, dzięki czemu jej instancja jest dostępna w dowolnym miejscu programu. Zawiera dwie mapy przechowujące wskaźniki do obiektów `CMesh` i `CTexture`. Indeksowane są za pomocą ciągu znaków, dzięki czemu istnieje możliwość nadawania unikalnych nazw dla poszczególnych elementów.

Głównymi zadaniami menedżera zasobów są:

- Dodawanie obiektów (lub ich automatyczne tworzenie po podaniu nazwy pliku).
- Usuwanie obiektów.

- Pobieranie obiektów (np. w celu wyświetlenia ich na ekranie).
- Czyszczenie pamięci zajmowanej przez obiekty (wykonywane przy kończeniu działania aplikacji).

3.1.5 Obiekty w grze

Każdy widzialny obiekt występujący na scenie musi implementować interfejs `IRenderable`, który wymusza przedefiniowanie metod `Render()`, `Update()` oraz `GetPosition()`. Zawiera również podstawowe dane dotyczące obiektu: nazwę, widoczność (która określa czy dany obiekt ma być wyświetlany) oraz typ (zwyczajny lub wykorzystujący efekt mapowania wypukłości).

3.1.5.1 Podstawowy obiekt - `CEntity`

Podstawową klasą reprezentującą obiekt w przestrzeni 3D jest `CEntity`. Przechowuje ona niezbędne wartości wymagane do aktualniania stanu oraz wyświetlania modelu takie jak:

- Wskaźnik do obiektu `CMesh`.
- Wskaźnik do tekstury obiektu (klasa `CTexture`).
- Macierze pozycji, rotacji i skalowania.
- Wektor rotacji (animacji obiektu) dzięki któremu możemy obiekt wprawić w rotację.
- Tekstura zawierająca wektory normalne (normal mapa) w przypadku obiektów wykorzystujących efekt mapowania normalnych.
- Bryła otaczająca wykorzystywana w metodzie optymalizacyjnej Frustrum Culling. Jest ona reprezentowana przez obiekt klasy `CFrustumShape` która na podstawie ustawionego mesha generuje otaczający ją boks lub kulę.

Uaktualnianie stanu obiektu za pomocą metody `Update()` polega na wyliczeniu nowych wartości rotacji obiektu na podstawie ustawionych wcześniej wartości animacji. Jeśli są one ustawione na 0 obiekt nie zmienia orientacji.

Proces wyświetlania obiektu został rozdzielony na dwie metody:

1. `Render()` – Sprawdza, czy obiekt znajduje się w bryle widzenia obserwatora (wykorzystując bryłę otaczającą).
2. `RenderEx()` – Ustawia odpowiednie parametry pliku efektu i rysuje model. Ta metoda wywoływana jest wewnątrz metody `Render()`.

3.1.5.2 Aktor w symulacji sceny PhysX - CEntityEx

W celu dodania obiektu do symulacji fizycznej należy stworzyć aktora oraz ustawić jego fizyczne właściwości. Wykorzystywana jest do tego rozszerzona klasa `CEntityEx` odpowiedzialna za przygotowanie kształtu aktora i dodanie go do symulacji. Zawiera (oprócz podstawowych z klasy bazowej `CEntity`) następujące dodatkowe wartości:

- Obiekt typu `NxActor` przechowujący wskaźnik do aktora na scenie.
- Kształt obiektu w symulacji.

Opcjonalne wartości:

- Obiekt klasy `CForceField` przypisujący pole siłowe do danego aktora o określonej sile i rozmiarze.
- Obiekt sterowania aktorem.
- Obiekt klasy `CTrigger` (więcej w rozdziale 3.1.5.4).

3.1.5.3 Sterowanie

Sterowanie obiektem jest możliwe po utworzeniu obiektu klasy implementującej interfejs `IUpdateable`, zawierającego metodę `Update()` wywoływaną przy każdym uaktualnieniu obiektu, a następnie przypisaniu go do docelowego obiektu.

Przykładową klasą obsługującą sterowanie obiektem klasy `CEntityEx` jest `CStoneControl`. Proces uaktualniania pozycji jest następujący:

1. Sprawdzenie stanu gry: tylko stan `game` umożliwia sterowanie obiektem.
2. Pobranie pozycji aktora i kamery.
3. Wyliczenie kierunku obiektu:

$$\vec{Direction} = \vec{cameraPosition} - \vec{entityPosition}$$

Odwrocenie wektora:

$$\vec{Direction} = \vec{Direction} * (-1)$$

Wyzerowanie wartości Y (nadawanie sił odbywa się jedynie w kierunku osi X i Z)

$$\vec{Direction.y} = 0$$

Normalizacja wynikowego wektora:

$$normalize(\vec{Direction})$$

4. Ewentualne obsłużenie żądania wyzerowania sił. Odbywa się to w dwóch krokach:

- Uśpienie aktora za pomocą metody `NxActor::putToSleep()`.
 - Obudzenie aktora za pomocą jednorazowego nadania minimalnej wartości siły w dowolnym kierunku.
5. Reakcja na wcisnięte klawisze strzałek na klawiaturze i ewentualnie nadanie sił w określonym kierunku:
- Góra: kierunek do przodu wyliczony z powyższego wzoru
 - Dół: kierunek odwrócony
 - Lewo, Prawo: wektor kierunkowy jest obracany o +/- 90°

3.1.5.4 Obsługa kolizji

Klasa `CEntityEx` zawiera mapy procedur skryptowych wykonywanych w wypadku wystąpienia kolizji sprawdzanego obiektu z innym.

```
map<NxContactPairFlag, string> callbackProcedures;
map<NxShapeFlag, string> triggerProcedures;
```

Wydruk 3.5: Kontenery zawierające nazwy metod do obsługi kolizji

Wyróżniamy dwa typy kolizji:

1. Proste kolizje zachodzące między obiekty. Za ich obsługę odpowiada klasa `CCollisionResponse`, która w przypadku wystąpienia kolizji sprawdza ustawienie procedury odpowiedzialnej za dany typ kolizji (np. `NX_NOTIFY_ON_START_TOUCH`). Jeśli taka jest ustawiona, następuje jej wywołanie.
2. Wejście w określony obszar (a także opuszczenie go lub pozostawanie w jego wnętrzu). Taki obszar może być przypisany do konkretnego aktora za pomocą obiektu klasy `CTrigger`, która generuje prostopadłościę o określonych wymiarach i ustawia go na pozycji obiektu. Obsługą zdarzeń generowanych poprzez kontakt obiektów z danym obszarem zajmuje się klasa `CTriggerCallback`, która, podobnie jak w przypadku prostych kolizji, sprawdza istnienie przypisanej metody i wywołuje ją.

3.1.5.5 Łączenie obiektów

Często występuje sytuacja, w której istnieje potrzeba stworzenia obiektu dynamicznego ze skomplikowanego modelu. Ponieważ takie modele najczęściej zawierają wkleśle fragmenty i są złożone z większej liczby trójkątów - wygenerowanie convex mesha nie jest możliwe. Problem ten został rozwiązyany przy pomocy łączenia obiektów.

Klasa `CEntityEx` zawiera opcjonalne pole zawierające wskaźnik do obiektu typu prosteego (`CEntity`). Proces łączenia wygląda następująco (przyjmując, że obiekt **A** jest typu `CEntity`, a obiekt **B** typu `CEntityEx`):

1. Stworzenie obiektu widzialnego **A** (właściwy wyświetlany model).

2. Stworzenie obiektu **B**, w którym modelem jest przygotowany wcześniej model reprezentujący wypukły kształt kolizyjny w symulacji (niewyświetlany na ekranie).
3. Przypisanie obiekowi **B** obiektu łączącego **A**. Po tej operacji obiekt **A** jest usuwany ze sceny.

Wyświetlanie modelu polega na pobraniu nowej pozycji i orientacji z obiektu **B**, który istnieje w scenie oraz podlega symulacji fizycznej, a następnie przekazaniu jej do metody `CEntity::RenderEx()` obiektu **A**, która rysuje widzialny obiekt na wskazanej pozycji i ze wskazaną orientacją.

3.1.5.6 Zarządzanie obiektyami

Przechowywaniem obiektów zajmuje się menedżer obiektów. Podobnie jak menedżer zasobów zawiera on mapę obiektów typu `CEntity` (oraz klas pochodnych), umożliwiając dodawanie, usuwanie i pobieranie jednostek. Umożliwia również wyczyszczenie mapy jednocześnie usuwając wszystkie obiekty.

3.1.6 Scena

Przechowywaniem widocznych na scenie obiektów oraz wykonywaniem symulacji fizycznej zajmuje się klasa `CScene`. Umożliwia ona dodawanie oraz usuwanie obiektów. Czyszczenie sceny usuwa jedynie informacje o przechowywanych obiektach. Właściwym usuwaniem obiektów zajmują się opisane w poprzednich rozdziałach menedżery obiektów i zasobów. Wyróżniamy dwa rodzaje scen:

- Symulacja fizyczna
- Wyświetlanie obiektów

3.1.6.1 Symulacja fizyczna

Podczas inicjalizacji silnika tworzona jest wirtualna scena. Ustawiana jest grawitacja, rejestrowane są klasy reakcji na zdarzenia kolizji oraz tworzona jest dolna płaszczyzna, która odgranicza przestrzeń pustą od zawierającej obiekty.

Wydruk 3.6 przedstawia instrukcje, które są wykonywane w celu dokonania pojedynczej symulacji. Pierwszym krokiem jest wywołanie metody `NxScene::simulate()`, która dokonuje symulacji każdego obiektu, sumując wszystkie siły działające na obiekty dynamiczne i odpowiednio je przemieszczając, a także wykrywa kolizje między obiektemi. Następną operacją jest pobranie wyników symulacji. Uaktualnia ona wszystkich aktorów na scenie, dzięki czemu posiadają zaktualizowaną pozycję oraz orientację.

```
scene->simulate(1.0 f /60.0 f );
scene->flushStream();
```

```
scene->fetchResults(NX_RIGID_BODY_FINISHED, true);
```

Wydruk 3.6: Wykonanie symulacji fizycznej

3.1.6.2 Wyświetlanie obiektów

Klasa `CScene` zawiera dwie mapy obiektów typu `IRenderable` dla każdego typu obiektu (zwykły i normal mapping). Zmiana pliku efektu podczas renderowania jest kosztowna obliczeniowo. Wprowadzony podział zapewnia tylko jedną zmianę w czasie przygotowywania klatki obrazu.

Metoda `CScene::Update()`, poza symulacją sceny fizycznej, wywołuje dla każdego obiektu metodę `IRenderable::Update()`, która w przypadku obiektów typu `CEntityEx` uaktualnia nową pozycję oraz orientację. Wyświetlanie obiektów dokonuje się poprzez wywołanie metody `IRenderable::Render()` iterując po mapie aktualnie używanego shadera.

3.1.7 Kamera

Kamera jest obiektem w przestrzeni 3D, który określa pozycję obserwatora. Klasa `CCamera` zawiera następujące parametry:

- Współrzędne punktu reprezentującego pozycję kamery.
- Punkt patrzenia. Możliwe jest również pobranie kierunku kamery używając wzoru:

$$\vec{direction} = \text{normalize}(\vec{lookAt} - \vec{position})$$

- Wartości wymagane do tworzenia macierzy widoku oraz projekcji.

Dostępne są dwa tryby kamery, od których zależy sposób uaktualniania pozycji: statyczna i dynamiczna.

3.1.7.1 Statyczna

Kamera statyczna zapewnia swobodne sterowanie kamerą w określonych kierunkach. Uaktualnianie pozycji polega na dodawaniu lub odejmowaniu określonej wartości liczbowej do poszczególnych składowych wektora pozycji i kierunku patrzenia. Przykładowa obsługa ruchu do przodu została pokazana na wydruku 3.7.

```
float camSpeed = 20.0f;  
if (Input.CameraForward()) {  
    position.z += camSpeed * FPSCounter.GetElapsedTime();  
    lookAt.z += camSpeed * FPSCounter.GetElapsedTime();  
}
```

Wydruk 3.7: Przesunięcie kamery statycznej do przodu

3.1.7.2 Dynamiczna

Kamera dynamiczna umożliwia śledzenie pozycji określonego obiektu. Wydruk 3.8 pokazuje przemieszczenie kamery względem pozycji obiektu. Dodatkowo uwzględniana jest odległość od obiektu w poziomie (`distance`), odległość w pionie (`distanceUp`) i rotacja (`rotation`).

```
lookAt = followedObject->GetPosition();  
float x = lookAt.x - sin(rotation)*distance;  
float y = lookAt.y + distanceUp;  
float z = lookAt.z - cos(rotation)*distance;
```

Wydruk 3.8: Uaktualnienie pozycji kamery dynamicznej

Sterowanie kamerą ogranicza się do obracania pozycji wokół obiektu. Wykonywane jest to za pomocą ruchu myszy w lewo i prawo. Różnica w odległości między kolejnymi uaktualnieniami pozycji kurSORA jest mnożona przez stałą prędkość kamery i dodawana lub odejmowana do wartości zmiennej `rotation`. Możliwa jest również zmiana wysokości kamery za pomocą ruchów do przodu i do tyłu przy wcisniętym lewym przycisku myszy. W tym przypadku określona wartość jest dodawana do obu zmiennych `direction` i `directionUp`.

3.1.8 Obsługa dźwięku

Do obsługi dźwięku została wykorzystana darmowa wieloplatformowa biblioteka irrKlang obsługująca wiele formatów takich jak: WAV, MP3, OGG, FLAC, MOD, XM, IT, S3M. Zarządzaniem i odgrywaniem dźwięków zajmuje się klasa `CSoundMgr`. Główne zadania menedżera dźwięku:

- Tworzenie unikalnych aliasów do plików.
- Ładowanie i odtwarzanie dźwięków 2D i 3D.

Klasa `CSoundMgr` umożliwia odtworzenie dźwięku jeden raz lub w nieskończonej pętli. W celu zachowania referencji do dźwięku, przydatnej do wstrzymania lub wcześniejszego zakończenia odtwarzania, tworzony jest obiekt klasy `CSound`, któremu przypisana zostaje unikalna nazwa. Dodawany jest on do mapy obiektów. Krótkie dźwięki, które są wywoływane często (np. odgłos uderzenia obiektu lub zmiana opcji w menu) nie wymagają zachowywania referencji. Dłuższe, takie jak podkład muzyczny, odgrywane najczęściej w nieskończonej pętli, powinny mieć konkretną nazwę i pozycję w mapie. Niepożądaną sytuacją jest ustawienie zapętlonego dźwięku muzycznego bez stworzenia obiektu, ponieważ odgrywałby się on aż do zamknięcia aplikacji.

Operacje dostępne w klasie `CSound` to:

- Wstrzymanie/wznowienie odgrywania
- Zakończenie odtwarzania

- Zatrzymanie pętli odtwarzania
- Sprawdzenie stanu odtwarzania (gra lub został zakończony)

Metoda `CSoundMgr::Update()` iteruje po wszystkich dostępnych obiektach. Najpierw usuwa obiekty, które zakończyły odtwarzanie. Na pozostałych obiektach wykonywana jest metoda `CSound::Update()`.

3.1.8.1 Dźwięk 3D

Klasa `CSound3D` jest rozszerzeniem klasy `CSound`. Odpowiada za obsługę dźwięków 3D. Istnieją dwie możliwości odgrywania:

- Stała pozycja w przestrzeni 3D. Możliwe jest również odtworzenie takiego dźwięku bez konieczności tworzenia obiektu `CSound3D`.
- Pozycja dynamiczna pobierana z przypisanego obiektu. W tym przypadku jedyną możliwością jest stworzenie obiektu `CSound3D`, który podczas uaktualniania pobiera pozycję obiektu i uaktualnia ustawienie dźwięku w przestrzeni.

3.1.9 Zarządzanie czasem - Timery

Ważnym elementem każdej gry jest wykonywanie pewnych operacji po upływie określonego czasu. Przykładem jest zmiana stanu gry `start1 → start2` po upływie 4 sekund. Także w samej grze występują sytuacje, w których np. po wyświetleniu okna z informacją zostaje ono automatycznie zamknięte po upływie kilku sekund.

3.1.9.1 Tworzenie

Pojedynczy timer reprezentowany jest za pomocą obiektu klasy `CTimer`. Zawiera ona niezbędne metody odpowiedzialne za wstrzymanie, wznowienie i sprawdzenie stanu timera. Najważniejszymi polami są czas startu oraz końca.

Wyróżniamy dwa typy timerów:

- **Prosty** (`CTimerSimple`) wykonuje wskazaną akcję po upływie ustawionego czasu.
- **Odliczający** (`CTimerCounting`) jest rozszerzeniem timera prostego. Różnicą w stosunku do poprzedniego, w którym akcja wykonywana jest tylko raz, jest ciągłe wykonywanie akcji, aż do momentu przekroczenia ustawionego czasu.

Przechowywaniem i uaktualnianiem timerów zajmuje się menedżer `CTimerMgr`. Poza metodami pozwalającymi na dodawanie, usuwanie i pobieranie obiektów, umożliwia też wstrzymanie i wznowienie wszystkich timerów.

Metoda `CTimer::Update()` wywoływana podczas uaktualniania logiki wykonuje dwie operacje:

1. Wykrycie timerów zakończonych i usuwanie ich z mapy obiektów.
2. Sprawdzenie pozostałych w celu ewentualnego wykonania ustalonej akcji.

3.1.9.2 Wykonywanie akcji

Ustawienie akcji wykonywanej podczas działania lub zakończenia timera polega na stworzeniu obiektu klasy `CCallback` i przypisaniu go do obiektu timera. Istnieją dwa warianty do wyboru:

- Wykonanie akcji systemowej (`CCallbackEngine`). Umożliwia ona wykonanie metody `CGame::NextStatus()`, który zmienia stan gry w zależności od poprzedniego, lub uaktualnienie czasu licznika gry (a także wywołanie odpowiednich metod w skrypcie poziomu, więcej w rozdziale 3.2).
- Wywołanie funkcji w skrypcie Lua (`CCallbackLua`).

3.1.9.3 Timery systemowe

Timery wewnętrzne, dodawane przez silnik, zaczynają się od przedrostka "SYSTEM". Wykorzystywane są do mechanizmów poprawiających działanie aplikacji:

- Reakcja na wcisnięcie przycisku strzałki w góre lub w dół w stanie **mainMenu** powoduje zmianę pozycji zaznaczonego elementu. Stan przycisku pobierany jest przy każdej iteracji pętli głównej programu, co powoduje wielokrotne wykonanie zmiany stanu wybranej pozycji w menu w ciągu sekundy, ponieważ czas wcisnięcia i zwolnienia przycisku na klawiaturze pokrywa się najczęściej z kilkoma wykonaniami procesu uaktualniania. Algorytm rozwiązujący omawiany problem jest następujący:
 1. Sprawdzenie istnienia timera "SYSTEM GUITime". Jeśli istnieje - sprawdzanie zostaje przerwane.
 2. Jeśli powyższy timer nie istnieje, następuje sprawdzenie stanu odpowiednich klawiszy i zmiana stanu elementów menu.
 3. Ostatnim krokiem jest ustawienie timera "SYSTEM GUITime" (bez przypisywania żadnej akcji) na $\frac{1}{10}$ sekundy, co jest wystarczającym czasem, w którym użytkownik może wcisnąć i zwolnić odpowiedni przycisk.
- Ustawienie stanów **start1** i **start2** powoduje dodanie timera, który po upływie 4 sekund wywoła akcję systemową, która z kolei wywoła metodę `CGame::NextStatus()` przechodząc do kolejnego stanu.

```
CTimer *timer = new CTimerSimple(4);
CCallback *callback = new CCallbackEngine(CFT_NEXTSTATUS);
timer->SetCallback(callback);
TimerMgr.AddTimer("SYSTEM start", timer);
```

Wydruk 3.9: Ustawianie zmiany stanu aplikacji po upływie 4 sekund

- W grze dostępny jest licznik czasu, który odlicza czas do końca poziomu. Na życzenie użytkownika ustawiany jest timer o nazwie "SYSTEM GameTime", który wywołuje dwie ustalone metody w skrypcie poziomu:

TimeoutCounting – Wykonywana przez cały okres działania zegara

Timeout – Wykonywana po upłynięciu czasu

3.1.10 Obsługa skryptów

Zewnętrzne skrypty umożliwiają przeniesienie wykonywania określonych operacji na zewnątrz kodu źródłowego. Dzięki temu zmiany w konfiguracji aplikacji lub logice poziomów gry mogą zostać uaktualniane bez konieczności komplikacji całego projektu, a nawet w trakcie działania programu.

Ładowaniem oraz obsługą skryptów Lua zajmuje się klasa **CScript**. Przechowywane są, tak jak w przypadku pozostałych menedżerów, w mapie obiektów. Indeksami są nazwy plików (bez rozszerzenia .lua). Dodanie nowego skryptu sprowadza się do wykonania trzech czynności:

1. Stworzenie struktury przechowującej skrypt oraz załadowanie go do pamięci
2. Dodanie obiektu do kontenera (mapy)
3. Rejestracja metod

Poza standardowymi operacjami dodawania, usuwania i pobierania obiektów, klasa udostępnia metodę **CScript::Reload()**. Umożliwia ona wyczyszczenie sceny oraz załadowanych zasobów, a następnie ponowne uruchomienie skryptu, dzięki czemu konfiguracja aktualnego poziomu zostaje odświeżona.

3.1.10.1 Rejestrowanie metod

Biblioteka Lua umożliwia rejestrowanie statycznych metod, tym samym umożliwiając ich wywołanie w kodzie skryptu. Odbywa się to poprzez wywołanie funkcji **lua_register()** (przykładowe wywołanie pokazuje wydruk 3.10) przyjmującej jako parametry: wskaźnik do struktury skryptu, nazwę metody (dostępną w skrypcie) oraz adres rejestrowanej funkcji.

```
lua_register(Lua, "ShowMessage", &LuaGame::ShowMessage);
```

Wydruk 3.10: Rejestracja metody

Aplikacja udostępnia wiele metod pogrupowanych w następujące kategorie:

- **Resource** – Ładowanie modeli i tekstur.
- **Sound** – Odtwarzanie dźwięków.
- **Entity** – Tworzenie i obsługa obiektów.

- **Camera** – Zmiana typu kamery i śledzonego obiektu.
- **Timer** – Dodawanie timerów.
- **Game** – Pozostałe funkcje np. Pauza gry.

Każda metoda przyjmuje jako parametr wskaźnik do skryptu, w którym zarejestrowana funkcja została wywołana. Następnie sprawdzana jest ilość parametrów przekazanych do funkcji. Metoda `CScript::CheckInput()` po podaniu wskaźnika do skryptu oraz ilości oczekiwanych parametrów zwraca `true` lub `false`. Następnym krokiem jest sprawdzenie typów parametrów za pomocą funkcji `lua_isstring`, `lua_isnumber` itd. Jeśli wszystko się zgadza następuje pobranie wartości parametrów ze stosu i wykonanie pozostałych czynności.

3.1.10.2 Wywoływanie metod skryptowych

Klasa `CScript` umożliwia wywoływanie funkcji skryptu z poziomu kodu C++. Odbywa się to w wielu przypadkach takich jak:

- Zgłoszenie kolizji między dwoma obiektami
- Wykonanie akcji timera po upływie wyznaczonego czasu
- Zmiana stanu gry

Metoda `CScript::CallFunction` posiada kilka wariantów, które umożliwiają wywoływanie wybranej funkcji określonego skryptu. Parametry jakie mogą być przekazane to napis, dwa napisy, liczba, lub wywołanie bez parametru. Możliwe jest również pobranie wartości zmiennej globalnej za pomocą metody `CScript::GetXXXTYPE()`, gdzie `XXX` jest jednym z trzech typów:

- **String** - Wartość tekstowa
- **Integer** - Liczba całkowita
- **Float** - Wartość zmiennoprzecinkowa

W przypadku niepowodzenia lub braku ustawionej wartości zmiennej, zwracana jest wartość domyślna ustawiana.

3.2 Tworzenie gry AtBall

W celu prezentacji możliwości przygotowanego silnika, stworzona została prosta gra platformowa. Niniejszy rozdział opisuje proces jej powstawania, od opisu pomysłu, do implementacji poszczególnych elementów

3.2.1 Opis gry

AtBall to zręcznościowa gra platformowa. Gracz steruje jednym z trzech typów kul:

- **Kamień** – Sterowanie jest utrudnione, a w niektórych sytuacjach, takich jak stromy podjazd pod góre, nawet niemożliwe. Zaletą tego typu kuli jest możliwość łatwego przesuwania na raz kilku obiektów (np. skrzyń).
- **Papier** – Kula papierowa imituje zgniecioną kartkę papieru. Jej lekka waga znacznie ułatwia sterowanie, jednak w niektórych przypadkach wymaga dużej zręczności. Przesuwanie cięższych obiektów jest utrudnione lub niemożliwe. Główną zaletą tego typu jest interakcja z wiatrakiem, który umożliwia włatywanie na niedostępne dla innych typów poziomy.
- **Drewno** – Najwygodniejszy w sterowaniu typ kuli. Średnia waga umożliwia proste przesuwanie jednego obiektu, a także podjazd pod góre.

Typ kuli zmienia się po najechaniu na odpowiednią platformę. Położenie tych obiektów jest zależne od aktualnych przeszkód do pokonania lub zadań do wykonania. Każda platforma ma przypisany jeden docelowy rodzaj kuli.

Celem gry jest dotarcie do końca poziomu, zbierając przy tym obiekty podobne do trójwymiarowej gwiazdy zwane Aurorami. Należy zebrać odpowiednią ich ilość, aby odblokować możliwość ukończenia planszy. Przykładowym zadaniem jest zebranie 10 Auror w celu aktywowania wiatraka, dzięki któremu papierowa kula podleci na platformę kończącą poziom.

3.2.2 Wstępna konfiguracja

3.2.2.1 Zmienne konfiguracyjne

Skrypt **config.lua** zawiera informacje, które muszą zostać ustawione przed uruchomieniem aplikacji:

- **screenSize** – Numer wybranej rozdzielczości. Dostępne wartości to:

- 1 800x600
- 2 1024x768
- 3 1280x800
- 4 1280x1024
- 5 1440x900
- 6 1680x1050
- 7 1920x1200

Preferowanym ustawieniem jest rozdzielcość panoramiczna.

- **skyboxTexture** – Ścieżka do pliku DDS zawierającego teksturę tła sceny.

```
skyboxTexture = "media/textures/skybox/Islands.dds"
```

Wydruk 3.11: Ustawienie tekstury SkyBoxa

- **level** – Nazwa pliku skryptu pierwszego poziomu (bez rozszerzenia .lua).

3.2.2.2 Obsługa zmiany stanu gry

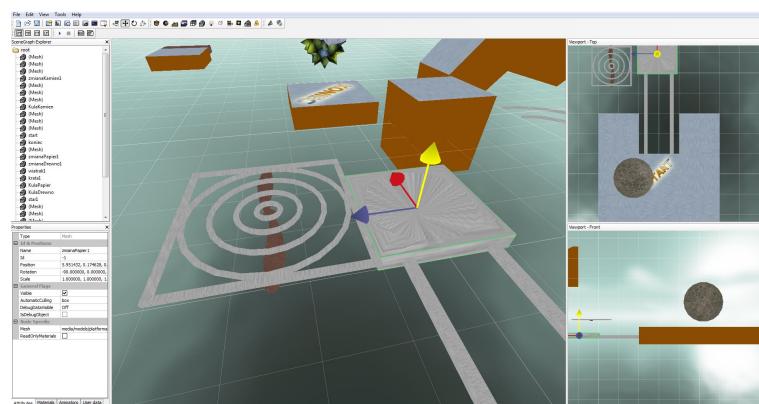
W pliku konfiguracyjnym istnieje możliwość wykonania dowolnych czynności podczas zmiany stanu gry. Służą do tego metody `onEngineStart()`, `onScreen1()`, `onScreen2()`, `onMainMenu()` itd., które przyjmują jako parametr poprzedni stan gry. Przykładowo w metodzie wykonywanej po starcie silnika można przygotować dźwięki takie jak tło menu, niezależnych od konkretnego poziomu, w metodzie `onScreen1` rozpoczęć odtwarzanie muzyki, a w `onLoading` wyłączyć ją i pozwolić skryptowi poziomu na ustawienie odpowiedniego tła muzycznego.

Przygotowane zostały również metody wykonywane przy zmianie pozycji w menu głównym (`onMenuChange()`) oraz wybraniu konkretnej pozycji (`onMenuEnter()`), które również umożliwiają odtworzenie odpowiednich dźwięków.

3.2.3 Tworzenie sceny

Aplikacja udostępnia odpowiednie metody, służące do dodawania zasobów oraz tworzenia obiektów na scenie, a także ustawiania ich właściwości.

Ręczne tworzenie sceny, zwłaszcza rozbudowanej, jest mało wygodne i nieefektywne. Autor zdecydował się na wykorzystanie darmowego edytora świata IrrEdit firmy AMBIERA. Umożliwia on dodawanie obiektów, ustawianie ich pozycji, orientacji, skali oraz tekstury.



Rysunek 3.4: Edytor świata IrrEdit

Gotowa scena zapisywana jest w formacie XML³. Aby dodać modele do sceny gry należy wywołać funkcję `ResourceParseIrrXML()`, która przyjmuje dwa parametry: ścieżkę

³Extensible Markup Language

do pliku XML wygenerowanego w programie IrrEdit oraz nazwę metody wywoływanej dla każdego obiektu. Metoda ta zawiera 12 parametrów:

1. Nazwa obiektu
2. Nazwa pliku modelu i tekstury
3. Po trzy wartości dla pozycji, rotacji i skali, odpowiednio dla osi X, Y i Z

Każde wywołanie tej metody powinno sprawdzać, czy niezbędne zasoby (model i tekstura) zostały załadowane, lub załadować je. Następnie następuje dodanie obiektu oraz (na podstawie modelu i tekstury) ustawienie odpowiednich właściwości. Wydruk 3.12 pokazuje przykładowe dodanie modelu śmigła wiatraka.

```
function IrrXmIParserCallback(name, mesh, texture, posX, posY, posZ,
    rotX, rotY, rotZ, scaleX, scaleY, scaleZ)
    (...) — pominiecie sprawdzania załadowania modelu i tekstury
    if mesh == "wiatrak.x" then
        return CreateWiatrak(name, mesh, texture, posX, posY, posZ, rotX,
            rotY, rotZ, scaleX, scaleY, scaleZ)
    end
    (...) — sprawdzanie i ewentualne dodanie innych obiektów
end
function CreateWiatrak(name, mesh, texture, posX, posY, posZ, rotX, rotY,
    rotZ, scaleX, scaleY, scaleZ)
    local name2 = CreateEntity(name, mesh, texture, posX, posY, posZ, rotX
        , rotY, rotZ, scaleX, scaleY, scaleZ, 0, EntityType.noPhysx, "")
    EntitySetAnimation(name2, 0, 600.0, 0)
end
```

Wydruk 3.12: Dodawanie obiektu wiatraka

3.2.4 Poziomy gry

Poziomy gry są przygotowywane w osobnych skryptach Lua. Nazwa pierwszego jest ustawiana w pliku konfiguracyjnym. Każdy z kolejnych zawiera informacje o następnym, przechowując nazwę skryptu w zmiennej `NextLevel`. Jeśli taka nie występuje, po zakończeniu aktualnego poziomu następuje koniec gry.

Twórca poziomu może ustawić czas w sekundach za pomocą funkcji `SetTimeout()`, która uruchamia licznik odliczający do zera. Jego działanie może zostać obsłużone następującymi metodami wywoływanymi przez aplikację:

- **TimeoutCounting** wywoływana ciągle, przekazująca w parametrze czas w milisekundach pozostały do końca odliczania. Może być np. wykorzystana do zmiany tła muzycznego, jeśli graczowi pozostało jedna minuta do końca.

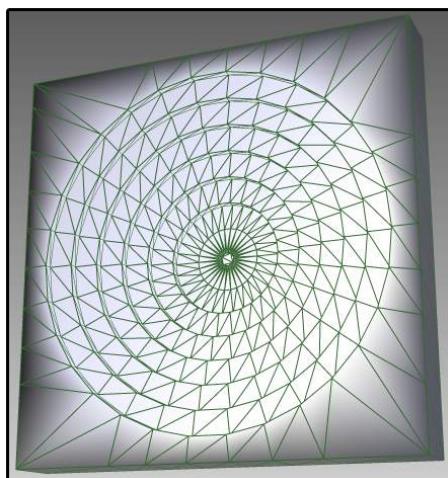
- **Timeout** wykonywana po dojściu licznika do zera. Pożądaną akcją jest wyświetlenie informacji o zakończeniu poziomu, a następnie wywołanie funkcji `Reset()` która ładuje poziom od nowa.

3.2.5 Elementy rozgrywki

Rozdział ten zawiera opis implementacji kilku elementów wykorzystywanych w grze AtBall.

3.2.5.1 Platforma zmiany typu kuli

Platforma zmiany typu kuli jest reprezentowana przez płaski kwadratowy model z kulistym wgłębieniem. Ustawiane są trzy różne tekstury, które wskazują na typ, na jaki zostanie zmieniona kula po wjechaniu we wgłębienie.



Rysunek 3.5: Model reprezentujący platformę zmiany kuli

Na scenie istnieją trzy modele reprezentujące konkretny rodzaj typu kuli. Jeden jest wyświetlany i obsługiwany przez sterowanie, natomiast dwa pozostałe są ukryte. Zmiana typu kuli powoduje ukrycie aktualnego obiektu i przeniesienie nowego na pozycję starego.

Proces zmiany wygląda następująco:

1. Wykrycie kolizji kuli z obiektem. Kula docelowo powinna znajdować się wewnątrz wgłębienia, aby mogła bezwładnie stoczyć się w stronę środka. Wykrycie kolizji z całym modelem nie daje satysfakcjonującego efektu, ponieważ znajdują się tam płaskie elementy. Rozwiązaniem tego problemu jest dynamiczne stworzenie obiektu odrobinę mniejszego od właściwego modelu, oraz obsługa zdarzenia wejścia w kontakt za pomocą triggerów. Dwie funkcje pokazane na wydruku 3.13 tworzą obiekt służący wykrywaniu kolizji o wymiarach $1.0 \times 0.1 \times 1.0$ (podane wymiary zostaną odpowiednio przeskalowane w zależności od ustawionej skali obiektu), a następnie przypisują obsługę zdarzenia kontaktowego w funkcji `zmianaTypuKuliStart()`.

```
EntitySetTrigger(platforma, 1.0, 0.1, 1.0)
EntitySetTriggerCallback(name2, "zmianaTypuKuliStart", Trigger.
    onEnter)
```

Wydruk 3.13: Tworzenie triggera

2. Wyzerowanie sił działających na obiekt oraz dezaktywowanie sterowania. Umożliwia to kuli bezwładne opadnięcie w stronę środka wgłębenia.

```
EntityClearForces(aktualnaKula)
EntityDisableUpdateObject(aktualnaKula)
```

Wydruk 3.14: Zerowanie sił

3. Opadnięcie kuli, w zależności od jej masy, wymaga czasu. W związku z tym, po wyłączeniu sterowania i dezaktywowaniu sił, dodawany jest timer, który po 4 sekundach wykonuje kolejną funkcję.

```
TimerAdd(nazwaTimera, 4.0, TimerType.simple, TimerCallback.Lua, "
    zmianaTypuKuliEnd")
```

Wydruk 3.15: Dodanie timera zmiany kuli

4. Po upływie czasu następuje zmiana typu. Wykonywane są następujące operacje:

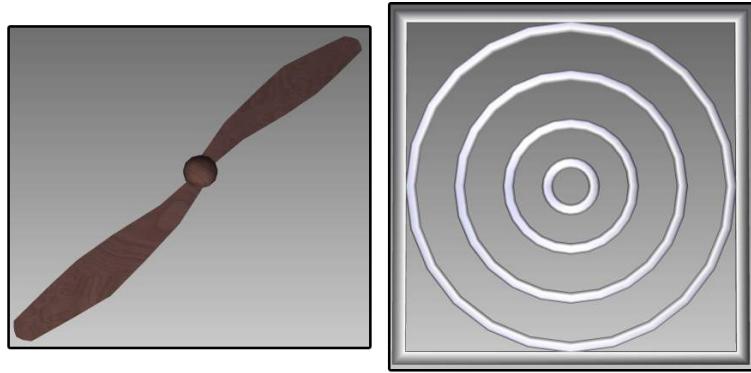
- Pobranie pozycji aktualnej kuli i ukrycie obiektu.
- Ustawienie pbranej pozycji obiekowi reprezentującemu nowy typ kuli, a następnie pokazanie go.
- Uaktualnienie ustawienia kamery (przypisanie nowego obiektu do śledzenia) oraz ustawienie sterowania.

3.2.5.2 Wiatrak

Wiatrak symulujący działanie wiatru wiejącego w określonym kierunku jest zbudowany z dwóch niezależnych modeli.

Pierwszym obiektem jest śmigło wiatraka, które zostaje umieszczone na scenie jako prosty model (bez tworzenia aktora symulacji fizycznej). W celu nadania rotacji wywołana zostaje funkcja `EntitySetAnimation()`, która uruchamia prostą animację obiektu obracając go według wskazanych osi i prędkości przypisanej do każdej z nich.

Kratka, która stanowi podstawę wiatraka, reprezentowana jest przez obiekt, umieszczony w odpowiedniej odległości od śmigła. W celu dodania pola siłowego, które wprawia obiekt znajdujący się w jej zasięgu w ruch, należy wywołać funkcję `EntitySetForceField()`. Oprócz nazwy obiektu, do którego zostanie przypisany efekt, zostają podane wartości wektora, które wskazują na kierunek działania siły. Ostatni parametr określa wartość siły, z jaką pole działa na obiekty. Wymiar pola siłowego dostosowuje się do skali obiektu. Rozszerzany jest nie tylko wymiar podstawy, ale również wysokość pola, dzięki czemu im większy wiatrak



Rysunek 3.6: Śmigło i podstawa wiatraka

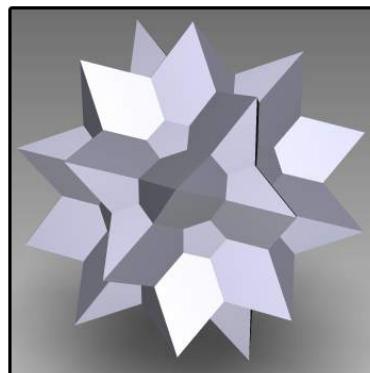
tym wyżej obiekty uniosą się wchodząc w jego zakres.

Docelowym działaniem wiatraka jest oddziaływanie wyłącznie na kulę papierową. Ograniczeniem domyślnej budowy pola siłowego w bibliotece PhysX jest brak uwzględniania masy obiektu, toteż na wszystkie obiekty nakładana jest jednakowa siła. Aby rozwiązać ten problem autor zdecydował się na włączanie wszystkich wiatraków, jeśli typ kuli zostanie zmieniony na papierowy. W przeciwnym wypadku pole siłowe zostaje usunięte. Pozostawienie animacji obracającego się śmigła daje wrażenie poprawnego działania wiatraka, jednak najechanie pozostałymi typami kul nie powoduje żadnej reakcji.

3.2.5.3 Zbieranie obiektów

Zadaniem gracza jest zebranie odpowiedniej ilości gwiazdek zwanych Aurorami. Obiekty te wiszą w powietrzu jednocześnie obracając się wokół własnej osi.

Dodanie nowego obiektu wymaga użycia łączenia obiektów (rozdział 3.1.5.5) z dwóch



Rysunek 3.7: Model Aurory

powodów:

- Dodanie stałej animacji rotacji jest dostępne tylko dla prostych obiektów
- Wykrycie kolizji z obiektem w celu inkrementacji licznika zebranych Auror wymaga stworzenia aktora w symulacji fizycznej

Pierwszym krokiem jest dodanie widzialnego obiektu, oraz ustawienie animacji. Następnie następuje stworzenie obiektu symulacji, tworząc kształt kulisty aktora. Użyty zostaje do tego wcześniej załadowany model obiektu widzialnego. Ostatnią operacją jest połączenie dwóch obiektów w jeden złożony.

```
local nameRender = CreateEntity(name, mesh, texture, posX, posY, posZ,
    rotX, rotY, rotZ, scaleX, scaleY, scaleZ, 0, EntityType.noPhysx, "")
EntitySetAnimation(nameRender, 50.0, 200.0, 25.0)
local namePhysx = CreateEntity(nameRender .. "PhysX", mesh, texture,
    posX, posY, posZ, rotX, rotY, rotZ, scaleX, scaleY, scaleZ,
    PhysxShape.sphere, EntityType.physx, "")
EntityRaiseBodyFlag(namePhysx, BodyFlag.kinematic)
EntitySetBindObject(namePhysx, nameRender)
```

Wydruk 3.16: Tworzenie połączonych obiektów

Algorytm zebrania jednej Aurory prezentuje się następująco:

1. Przypisanie zdarzenia wykrywającego wejście w obiekt Aurory poprzez stworzenie dodatkowego obiektu o wymiarach odrobinę większych od właściwego aktora:

```
EntitySetTrigger(namePhysx, 1.1, 1.1, 1.1)
EntitySetTriggerCallback(namePhysx, "zebranieAurory", Trigger.
    onEnter)
```

Wydruk 3.17: Przypisanie reakcji na kolizję z obiektem Aurory

2. Obsłuszenie wywołanej funkcji `zebranieAurory()`, która wykonuje następujące operacje:

- Sprawdzenie czy kolizja nastąpiła z kulą.
- Ukrycie obiektu Aurory razem z dezaktywacją obiektu do wykrywania kolizji.
- Zwiększenie licznika zebranych Auror o 1 i uaktualnienie nowej wartości na ekranie.

```
ZebraneAurory = ZebraneAurory + 1
SetStarsCount(ZebraneAurory)
```

Wydruk 3.18: Uaktualnienie licznika zebranych Auror

Rozdział 4

Podsumowanie

4.1 Ocena projektu

Główym założeniem projektu było stworzenie silnika graficznego, umożliwiającego odzielenie warstwy logicznej gry od kodu aplikacji. Cel ten został osiągnięty, czego dowodem jest przygotowana gra komputerowa. Skrypty opisujące poszczególne poziomy umożliwiają programistom budowanie sceny, ustalenie celu ukończenia planszy, a także kreację dowolnych zadań, które muszą zostać wykonane.

Przygotowanie tak rozbudowanej aplikacji jaką jest gra komputerowa wymagało kilku miesięcy intensywnej pracy. Zimplementowanie wszystkich funkcjonalności przewidzianych w procesie planowania okazało się niemożliwe. Głównymi powodami były:

- Ograniczenia czasowe. Zaawansowane gry komputerowe wymagają nawet kilku lat pracy i dużego zespołu programistycznego. Autor przygotowywał pracę w jednoosobowym zespole, tak więc koniecznością była implementacja jedynie najważniejszych elementów aplikacji, skupiających się na głównym założeniu projektu. Z tego powodu pominięte zostało rozbudowywanie części graficznej aplikacji, odpowiedzialnej za efekty takie jak system cząsteczek.
- Niedostateczna wiedza i doświadczenie w projektowaniu i implementowaniu dużych projektów. Skutkowało to częstym przepisywaniem wielu elementów (a nawet całego projektu) na nowo.

Mimo wielu napotkanych trudności, autorowi udało się stworzyć od podstaw funkcjonalny silnik graficzny, przygotowany do dalszego rozwoju poszczególnych elementów, między innymi:

- Zarządzanie GUI z poziomu skryptów.
- Dodanie obsługi światła, cieni i efektów cząsteczkowych.
- Łączenie obiektów za pomocą joint'ów (funkcja dostępna w bibliotece PhysX).

Rozdział 5

BIBLIOGRAFIA

1. Mike Dickheiser, "Perełki programowania gier Vademecum Profesjonalisty TOM 6", Helion 2008
2. Frank D. Luna, "Introduction to 3D Game Programming with DirectX 9.0c - A Shader Approach", Wordware Publishing 2006
3. <http://www.gamedev.pl>
4. <http://charibo.wordpress.com>
5. <http://pclab.pl/art17706.html>
6. <http://www.wikipedia.pl>
7. Dokumentacje bibliotek: DirectX, nvidia PhysX, Lua, irrKlang

Spis rysunków

2.1	Spyro The Dragon	9
2.2	Age of empires 2	9
2.3	Final Fantasy 9 – RPG w świecie fantasy	10
2.4	Crazy Machines – rozwiązywanie fizycznych łamigłówek	10
2.5	Fahrenheit – gra fabularna	11
2.6	Fifa 2009 – piłka nożna	11
2.7	Need For Speed Carbon – wyścigi uliczne	11
2.8	Microsoft Train Simulator – Symulator jazdy pociągiem	12
2.9	Tennis for Two - jedna z pierwszych gier komputerowych	12
2.10	Konsola Sony PlayStation 3	13
2.11	Iloczyn wektorowy	16
2.12	Kolejność mnożenia: rotacja * translacja	22
2.13	Kolejność mnożenia: translacja * rotacja	22
2.14	Budowa bryły obcinania	23
2.15	Trójkąt przed rasteryzacją	25
2.16	Trójkąt i wyznaczone piksele po procesie rasteryzacji	25
2.17	Przykładowe filtry post-processing'owe	27
2.18	Budowa bryły obcinania	28
2.19	Wykrywanie widocznych obiektów	28
2.20	A: Frustrum Culling, B: Occlusion Culling	29
2.21	Point light	30
2.22	Reflect light	31
2.23	Od lewej: cieniowanie płaskie, Gourauda, Phonga	32
2.24	Układ koordynatów UV tekstuury	33
2.25	Model przez i po mapowaniu	34
2.26	Przykład tekstuury i normal mapy	35
2.27	Podgląd sceny w programie PhysX Visual Debugger	40
3.1	Tekstura mapująca sztuczne tło świata	50
3.2	Menu główne gry	52
3.3	Okno wiadomości w trakcie gry	52

3.4	Edytor świata IrrEdit	65
3.5	Model reprezentujący platformę zmiany kuli	67
3.6	Śmigło i podstawa wiatraka	69
3.7	Model Aurory	69

Spis wydruków

2.1	Przykładowy Vertex Shader	24
2.2	Przykładowy Pixel Shader	26
2.3	Dynamiczna zmiana typu zmiennej	41
2.4	Zasięg zmiennych	42
2.5	Parametry funkcji	42
3.1	Singleton	45
3.2	Uruchomienie gry	46
3.3	Pętla gry	48
3.4	Proporcjonalne skalowanie obrazków	51
3.5	Kontenery zawierające nazwy metod do obsługi kolizji	55
3.6	Wykonanie symulacji fizycznej	57
3.7	Przesunięcie kamery statycznej do przodu	58
3.8	Uaktualnienie pozycji kamery dynamicznej	58
3.9	Ustawianie zmiany stanu aplikacji po upływie 4 sekund	61
3.10	Rejestracja metody	62
3.11	Ustawienie tekstury SkyBoxa	64
3.12	Dodawanie obiektu wiatraka	66
3.13	Tworzenie triggera	67
3.14	Zerowanie sił	67
3.15	Dodanie timera zmiany kuli	68
3.16	Tworzenie połączonych obiektów	70
3.17	Przypisanie reakcji na kolizję z obiektem Aurory	70
3.18	Uaktualnienie licznika zebranych Auror	70