POLITECHNIKAWROCŁAWSKA

WYDZIAŁELEKTRONIKI

KIERUNEK: INFORMATYKA (INF)

SPECJALNOSC: INŻYNIERIA SYSTEMÓW INFORMATYCZNYCH (INS)

PRACA DYPLOMOWA

INŻYNIERSKA

Symulator graficzny procesu fizycznego w środowisku DirectX

Graphic simulator of a physical process in DirectX

AUTOR:

Wojciech Sobczak

PROWADZACY PRACĘ:

doc. dr inż. Jacek Jarnicki, ZSKiD

OCENA PRACY:

WROCŁAW, 2016

**Spis treści**

[Spis rysunków 4](#_Toc467452675)

[Spis listingów 5](#_Toc467452676)

[Skróty 6](#_Toc467452677)

[1. Wstęp 7](#_Toc467452678)

[1.1. Wprowadzenie 7](#_Toc467452679)

[1.2. Cel i zakres pracy 7](#_Toc467452680)

[2. Problem 8](#_Toc467452681)

[2.1. Sformułowanie problemu 8](#_Toc467452682)

[2.2. Przegląd dostępnych rozwiązań 8](#_Toc467452683)

[2.2.1. Unity 8](#_Toc467452684)

[2.2.2. Unreal Engine 9](#_Toc467452685)

[2.2.3. Blender 9](#_Toc467452686)

[2.2.4. Autodesk 3DSMax 10](#_Toc467452687)

[3. Aplikacja 12](#_Toc467452688)

[3.1. Założenia 12](#_Toc467452689)

[3.2. Przyjęte technologie 12](#_Toc467452690)

[3.2.1. DirectX 12](#_Toc467452691)

[3.2.2. Bullet Physics 14](#_Toc467452692)

[3.2.3. DirectXTK 15](#_Toc467452693)

[3.3. Struktura aplikacji 16](#_Toc467452694)

[3.3.1. Architektura 16](#_Toc467452695)

[3.3.2. Klasa Simulation 17](#_Toc467452696)

[3.3.3. Klasa Object 17](#_Toc467452697)

[3.3.4. Przestrzeń nazw Scenario 18](#_Toc467452698)

[3.4. Opis działania 19](#_Toc467452699)

[3.4.1. Potok renderowania 19](#_Toc467452700)

[3.4.2. Tworzenie obiektów świata 24](#_Toc467452701)

[3.5. Opis użytkowania 25](#_Toc467452702)

[3.5.1. Sterowanie 25](#_Toc467452703)

[4. Testy 26](#_Toc467452704)

[4.1. Plan testów 26](#_Toc467452705)

[4.2. Wyniki 26](#_Toc467452706)

[4.3. Wnioski 26](#_Toc467452707)

[5. Literatura 27](#_Toc467452708)

# Spis rysunków

[Rysunek 1 Drzewo dziedziczenia klasy Object<T> 18](#_Toc467452590)

[Rysunek 2 Potok renderowanie DirectX 20](#_Toc467452591)

# Spis listingów

[Listing 1 Tworzenie urządzenia karty graficznej 21](#_Toc467452606)

# Skróty

**GPL** (ang. *General Public License*)

**API** (ang. *Application Programming Interface*)

**SIMD** (ang. *Single Instruction, Multiple Data*)

**GPU** (ang. *Graphical Processing Unit*)

1. Wstęp

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris id dapibus enim. Etiam lobortis pulvinar enim in maximus.

* 1. Wprowadzenie

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris id dapibus enim. Etiam lobortis pulvinar enim in maximus.

* 1. Cel i zakres pracy

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris id dapibus enim. Etiam lobortis pulvinar enim in maximus.

1. Problem
   1. Sformułowanie problemu

Właściwie to nie miałem z tym problemu.

* 1. Przegląd dostępnych rozwiązań

W przypadku tego problemu, ciężko jest znaleźć rozwiązanie okalające tak wąski zakres funkcjonalności. Dzieje się tak dlatego, że programy implementujące różnorakie biblioteki symulujące fizykę, są zazwyczaj programami pełniącymi funkcję edytorów scen. Edytory te są rozbudowane, zarówno pod względem implementowanych procesów fizycznych, jak i znacznie bardziej skomplikowanych, niż w tej pracy, efektów graficznych opartych o niezwykle skomplikowany aparat matematyczny. Przykładem takich efektów mogą być systemy cząsteczkowe (particles systems), które np. pozwalają w sposób relatywnie realistyczny symulować zachowanie wody. Przykłady więc będą opisywać programy, których zakres pracy jest znacznie bardziej rozbudowany niż opisywanej tu aplikacji, nie mniej, pozwalający na zrealizowanie dokładnie tych samych procesów.

* + 1. Unity

Produkt „Unity Game Engine”, firmy „Unity Technologies”, jest to środowisko pozwalające na w pełni programowalne budowanie trójwymiarowych oraz dwuwymiarowych scen, z pomocą graficznego edytora. Docelowo produkt reklamowany jest jako narzędzie do tworzenia gier komputerowych. Jest to narzędzie, którego wytworzony zestaw scen, może zostać odtworzony na 22 środowiskach uruchomieniowych do których należą np: Windows, OSX, Linux w dystrybucji firmy Canonical, Ubuntu, PS3, PS4, Xbox360, XONE, oraz wiodące platformy mobilne takie jak Android, IOS oraz BlackBerry.

Rozwiązanie jest niezwykle popularne wśród początkujących twórców gier ze względu na prostotę tworzenia scen, jednak zdarzają się produkcję typu „AAA” (gry wysokobudżetowe), których architektura opiera się o właśnie to rozwiązanie. Przykładem takiego produktu może być „Torment: Tides of Numenera” lub „Pillars of Eternity”.

Większość etapu budowania polega na przeciąganiu myszą z menu edytora, do menu opisywanych obiektów, jako właściwości, oraz nadawaniu parametrów tym właściwościom, takich jak w przypadku obiektu fizycznego masa, środek masy itd.

Możliwość programowania interakcji oraz samych scen została zaimplementowana poprzez umożliwienie użytkownikom używania 3 języków programowania. Najpopularniejszym wyborem, a zarazem najmniej hermetycznym w obrębie środowiska Unity, jest C#, jednak istnieje możliwość wyboru innych języków takich jak „Boo”, który składnią przypomina język „Python” oraz autorski produkt twórców silnika, „Unity Script”, który jest bardzo podobny do języka rozwiązań webowych „JavaScript”.

* + 1. Unreal Engine

„Unreal Engine” jest to produkt firmy „Epic Games”, twórców takich gier jak „Unreal Tournament” czy „Gears of War”, który podobnie jak Unity, jest pełnoprawnym silnikiem do tworzenia gier oraz różnorakich scen trój i dwuwymiarowych. Znaczącą jednak różnicą jest fakt, że silnik ten, dużo częściej pada wyborem twórców wysokobudżetowych gier oraz symulacji. Dzieje się tak dlatego, że silnik ten pozwala na budowanie aplikacji w języku C++ co daje nam znacznie większe możliwości kontroli nad programem niż wysokopoziomowe języki silnika Unity. Drugim powodem częstszego wybierania tego rozwiązania może być fakt, że Epic Games silnik ten tworzy od roku 1998, co daje 8 lat doświadczenia więcej, niż firmie Unity Technologies.

Tworzenie scen oraz definiowanie obiektów w tym silniku to również przeciąganie właściwości z menu to menu obiektu, także pod względem ogólnym te dwa rozwiązania nie różnią się znacząco.

Unreal Engine to również rozwiązanie multiplatformowe, pozwalające wydać aplikację na różne systemy po prostu definiując profile ustawień kompilacji.

Najpopularniejszymi przykładami pokazującymi możliwości tego rozwiązania mogą być „Batman: Arkham Knight”, czy też polska gra „Hatred”.

* + 1. Blender

Blender jest to program do modelowania obiektów 3D oraz tworzenia animacji. Jest to rozwiązanie typu OpenSource na licencji GPL, którego głównym programistą jest, i był od samego początku powstawania programu, Ton Roosendaal. Program ten oprócz możliwości modelowania oferuje wachlarz innych, przez co jest on niezwykle popularny na rynku.

Ze wspomnianych wcześniej możliwości wymienić trzeba, że program posiada własny silnik renderujący, który umożliwia szeroką ingerencję użytkownika w proces rysowania obiektu na ekranie. Do dyspozycji użytkownika oddano np. możliwość tworzenia własnych filtrów oraz shaderów, możliwość użycia rendererów innych niż domyślny Blender Internal, takich jak YafaRay czy LuxRender oraz możliwość definiowania własnych dając do dyspozycji możliwość pisania skryptów w języku Python.

Blender wykorzystuje silnik fizyczny Bullet Physics, użyty również w implementacji pracy inżynierskiej. Pozwala on na budowanie dokładnych symulacji fizycznych z wykorzystaniem modeli tworzonych wewnątrz programu co przekłada się na wygodę użytkowania, gdyż raz wymodelowane przez nas obiekty posiadają zintegrowaną fizykę, której wystarczy tylko nadać odpowiednie właściwości liczbowe.

Rozwiązanie to, ze względu na wymienione wyżej funkcje, pozwala również na tworzenie efektów specjalnych, filmów i animacji, czego przykładem może być produkcja „Big Buck Bunny” oraz „Big Miracle”.

* + 1. Autodesk 3DSMax

3DS Max jest w pełni zamkniętym i komercyjnym produktem firmy Autodesk. Jest on, podobnie jak Blender, programem do tworzenia zaawansowanych modeli 3D oraz tworzenia animacji. Jako że nie jest to rozwiązanie tworzone przez społeczność, posiada znacznie większe i bardziej stałe zaplecze twórców oraz wsparcie, dlatego też jest częściej wybierane w komercyjnych rozwiązaniach.

Jeżeli chodzi o użytkowanie, Blender i 3DSMax są bardzo zbliżone w sensie ogólnym. Definicja obiektów to przeciąganie obiektów z menu, modelowanie siatek odbywa się poprzez przesuwanie poszczególnych linii, itd. Dodano doń również obsługę języka MAXScript, jednak nie umożliwia on ingerencji w proces budowania obrazu, a tylko dodaje możliwość definiowania własnych wtyczek do programu, czy też automatyzowania niektórych procesów twórczych.

Dokładnie tak samo jak w Blenderze, mamy tu możliwość tworzenia symulacji fizycznych, jednak w tym przypadku mamy do czynienia z symulacją wykorzystującą wewnętrzną implementacji fizyki firmy Autodesk. Umożliwia ona nam np. budowanie symulacji cząsteczkowych oraz tworzenie symulacji tkanin.

O przewadze tego rozwiązania nad jego darmowym konkurentem może stanowić portfolio jego filmów, gdyż użyto go do stworzenia efektów specjalnych w produkcjach takich jak „Mad Max: Fury Road”, „The Curious Case of Benjamin Button” czy „Black Hawk Down”.

1. Aplikacja
   1. Założenia

Stworzony symulator ma za zadanie ukazać trzy scenariusze prezentujące różne układy obiektów, które będzie można za pomocą sześcianów „wystrzeliwanych” przez użytkownika niszczyć, w sensie, zaburzać równowagę tych konstrukcji, doprowadzając je do upadku. Ukazane scenariusze to:

* **House of cards** - konstrukcja przypominającą domek z kart, skonstruowana z kostek o kształcie kostek domina
* **Towers** – wieże złożone z ułożonych na sobie sześciennych kostek, które ustawione są na okręgu wokół osi Y przestrzeni
* **Domino** – ułożone w spirale kostki, o kształcie kostek domina

Scenariusze te dostępne będą poprzez rozwijane menu na pasku głównym okna aplikacji. Dodatkowo aplikacja będzie umożliwiała swobodne poruszanie się użytkownika po przestrzeni udostępnionej przez aplikację, umożliwiając tym samym zmianę kierunku oraz kąta wystrzeliwanego obiektu.

Aplikacja ta działać będzie w środowisku Windows, począwszy od wersji 10. W folderze głównym aplikacji wymagać będzie obecności plików tekstur, które zostaną dostarczone wraz z plikiem wykonywalnym.

* 1. Przyjęte technologie
     1. DirectX

Główną technologią użytą w tym projekcie, będącą za razem częścią jego tematu, jest biblioteka DirectX.

DirectX jest stworzonym przez Microsoft, zestawem relatywnie wysokopoziomowych API, skonstruowanych z myślą o obsłudze multimediów takich jak grafika 3D, dźwięk i inne, ukierunkowany został jednak w stronę tworzenia gier wideo oraz symulacji. Został on w całości zaimplementowany w języku C++ budując wielopoziomową strukturę obiektową, tworzącą abstrakcję nad używanymi sprzętowymi komponentami.

Na cały pakiet składa się:

* **Direct3D (D3D)** – biblioteka obsługująca renderowanie obiektów 3D
* **DXGI** – biblioteka udostępniająca interfejsy programistyczne, tworzące logiczną warstwę pomiędzy sprzętęm potrzebnym do wyświetlania grafiki, a programistą. Udostępnia ona interfejsy dla monitorów (ekranów wyświetlania), kart graficznych, oraz umożliwia samodzielne zarządzania buforami klatek.
* **Direct2D** - biblioteka obsługująca renderowanie obiektów 2D
* **DirectWrite** - biblioteka obsługująca renderowanie czcionek
* **DirectCompute** - biblioteka umożliwiające prowadzenie zrównoleglonych obliczeń na karcie graficznej
* **DirectSound3D (DS3D)** – biblioteka umożliwiająca odtwarzanie dźwięku oraz jego manipulację
* **DirectX Media** – jest to zestaw bibliotek skupiający następujące narzędzia:
  + **DirectAnimation –** do tworzenia animacji 2D/3D w środowisku webowym
  + **DirectShow** – do odtwarzania multimediów oraz ich strumieniowania
  + **DirectX Transform –** do obsługi, edycji oraz animacji obrazów
* **DirectX Diagnostics (DxDiag)** – diagnostyczne narzędzie dla systemu Windows, pozwalające na tworzenie raportów na temat obecnych w systemie komponentów DirectX, takich jak urządzenia audio, wideo oraz urządzenia wejścia, takie jak kontrolery gier.
* **DirectX Media Objects** – jest to biblioteka podobna do zdeprecjonowanej już DirectShow. Służy do strumieniowego przesyłania i przetwarzania danych z wejścia do wyjścia. Zwykle są to interfejsy kodeków służące do przetwarzania danych multimedialnych
* **DirectSetup** – narzędzie stworzone do instalacji komponentów DirectX, oraz wykrywania ich wersji
* **XACT3** – stworzona z myślą o platformie XBOX wysokopoziomowa biblioteka do wieloplatformowej, w obrębie firmy Microsoft, obsługi dźwięku.
* **XAudio2** – biblioteka do obsługi dźwięku podobna do XACT3, jednak z tą różnicą, że udostępniony tu interfejs jest znacznie niżej poziomowy.

Z ogromu tych funkcjonalności tego pakietu, w tym projekcie został użyty tylko Direct3D w wersji 11. Biblioteka ta zawiera w sobie zdefiniowane nagłówki do biblioteki DirectXMath, które to są jedną z ważniejszych jej modułów.

DirectXMath jest główną i polecaną przez twórców tutoriali oraz dokumentacji Microsoftu, biblioteką do ułatwienia obliczeń. Jest to o tyle pożyteczna biblioteka, że jej typy zaprojektowane zostały pod kątem obsługi ich przez urządzenia wpierające SIMD, oczywiście jest to tylko dodatek, o którym zdecyduje środowisko kompilacji, całość natomiast kompatybilna jest z najbardziej podstawowym zestawem operacji w obrębie języka C++.

Udostępnia ona API do operacji na wektorach różnej długości (od 2 do 4), macierzy (3x3 oraz 4x4) oraz operacji zachodzących pomiędzy nimi, takich jak mnożenie, odwracanie, transformacja i inne. Dodatkowo jest ona dostosowana do obsługi operacji najczęściej używanych i potrzebnych w tworzeniu aplikacji 3D, np. rotacje wektorów, transformacje wektorów na podstawie kwaternionów czy też różnych macierzy (macierze obrotów, skalowania, translacji), łączenia tych macierzy w macierze transformacji, tworzenie macierzy widoków, perspektyw i wiele, wiele innych przydatnych funkcjonalności, których długo by wymieniać.

Jest to technologia na tyle rozwinięta i popularna, że wykorzystywana jest niemal w każdym rozwiązaniu dla technologii gier wideo. DirectX implementują wcześniej już wspomniane Unity oraz Unreal Engine, oraz każdy silnik chcący posiadać możliwość uruchomienia swojej aplikacji na systemie Windows czy XBOX z najbardziej optymalnym wsparciem sprzętowym.

* + 1. Bullet Physics

Bullet Physics jest to biblioteka przeznaczona do wykrywania kolizji, symulowania dynamiki brył sztywnych oraz ciał deformowalnych (miękkich) takich jak np. tkaniny czy sprężyny. Struktura tej aplikacji została stworzona w języku C++ wykorzystując w pełni model obiektowy, jednak niższe jej warstwy napisane są w języku C, celem zwiększenia wydajności przetwarzania danych.

Główne zalety tego silnika to:

* Wykrywanie kolizji brył sztywnych oraz ciał deformowalnych w trybie ciągłym lub dyskretnym
* Pełne wsparcie dla obiektów deformowalnych
* Udostępnione zestawy określonych „kształtów kolizji”, czyli obiektów geometrycznych na podstawie których wyznaczane są kolizje, oraz nań reakcje. A na nie składają się:
  + - Sfera
    - Prostopadłościan
    - Walec
    - Stożek
    - Otoczka wypukła, wykorzystująca algorytm Gilberta – Johnsona – Keerthiego
    - Otoczka niewypukła
    - Siatka trójkątów
  + Opcjonalny zestaw optymalizacji pod kątem technologii CUDA oraz OpenCL

Dodatkowym atutem dla potencjalnego użytkownika tej biblioteki może być dosyć obszerne forum projektu, na którym dyskutowane są problemy programistów podczas użytkowania tej biblioteki, gdzie często wypowiadają się jej kontrybutorzy. Znacząco ułatwia to użytkowanie podczas realnego tworzenia aplikacji, ponieważ ze względu na złożoność rozwiązania nie można poznać go dogłębnie z każdej strony w stosunkowo krótkim czasie.

Rozwiązanie to jest bardzo docenione w środowisku, o czym świadczy fakt, że jest to zintegrowany w takich programach jak Blender, Houdini, SoftImage czy Cinema 4D. Dodatkowo można wspomnieć, że jako rozwiązanie działające w czasie rzeczywistym również sprawdza się doskonale o czym świadczy fakt że zostało ono użyte w takich produkcjach jak Grand Theft Auto V, DIRT oraz Red Dead Redemption.

* + 1. DirectXTK

DirectXTK, czyli DirectX Tool Kit, jest to biblioteka, a właściwie zestaw klas i struktur pomocniczych, ułatwiających implementację najczęściej budowanych szablonów i konstrukcji aplikacji używających DirectX, a są to np. tworzenie macierzy widoków, perspektyw, zarządzanie pamięcią buforów wierzchołków oraz predefiniowane typy wierzchołków wejściowych dla shaderów.

Projekt ten powstał pod egidą firmy Microsoft i jest on oficjalnie wspierany przez pracowników tej firmy. Jest stale rozwijany co potwierdzają daty udostępnień nowych kawałków kodu na oficjalnym repozytorium. Wersja która została użyta w tym projekcie odnosi się do wersji 11 DirectX, jednak teraz skupiono się na rozwoju wersji dla najnowszej wersji biblioteki Microsoftu i to ona jest głównym zainteresowaniem kontrybutorów.

Jednak o jakich klasach pomocniczych mowa? DirectXTK dostarcza np. klasy predefiniowanych efektów, np. mgły. Proces implementacji takiego efektu sprowadził by się normalnie do żmudnej implementacji skomplikowanego shadera, opartego na nietrywialnym zapewne modelu matematycznym, powodując tym samym, bycie niedostępnym dla przeciętnego twórcy aplikacji (np. na Universal Windows Platform, czyli użytkową aplikację Windows 10, do której również można użyć DirectX). W przypadku tego frameworku sprowadzi się to do inicjalizacji klasy i użycie jednej z jej metod na początku potoku renderowania, czy też w jego trakcie, w zależności od zastosowania. Prowadzi to bezpośrednio to znacznego zwiększenia wydajności pracy i efektowności wyników tej pracy, w nie wymagających dużej wiedzy o grafice komputerowej w aplikacjach użytkowych.

* 1. Struktura aplikacji
     1. Architektura

Aplikacja została napisana w języku C++ z zastosowaniem paradygmatu programowania obiektowego. Było to niejako wymuszone przez użyte narzędzia, ponieważ DirectX został napisany całkowicie obiektowo, a sam Microsoft poleca takie praktyki. Robi to w sposób dosłowny gdy czytamy o tym w pierwszym artykule dotyczącym DirectX’a oraz realizując wszystkie przykłady w tym właśnie stylu. Bullet Engine w używanej w tym projekcie wersji, również posiada całkowicie obiektową strukturę kodu.

Klasy używanych przeze mnie bibliotek, są deklarowanie jako inteligentne wskaźniki. Było to zaleceniem, ze względu na szczególny nacisk na wydajność wszystkich użytych przeze mnie rozwiązań, z jednoczesnym uwzględnieniem łatwości w zarządzaniu kodem oraz pamięcią. W przypadku klas Microsoftu użyta została klasa ComPtr, która ma być inteligentnym wskaźnikiem na typ IUnknown, który jest klasą bazową wszystkich klas biblioteki DirectX. W przypadku Bullet Engine, użycie shared\_ptr lub ComPtr, skutowało tym, że klasy zarządzając samymi sobą kolidowały z klasami wskaźników, co wymusiło użycie ich jako zwykłych wskaźników znanych z C++.

Prawie każda klasa ma swój plik nagłówkowy oraz plik implementacji. Jest to model zalecany, dlatego też został użyty w tym projekcie. Jedynie klasa DXDebugDraw, która jest realizacją interface’u silnika Bullet do rysowania obiektów w celach debugowania działania programu, znajduje się cała w pliku nagłówkowym. Dzieje się tak ze względu na jej prostotę, oraz fakt, że zaimplementowana została tylko 1 metoda, która była wymagana. Dodatkowo bez plików implementacji zrealizowane zostały funkcję przestrzeni nazw Debug, ze względu na deklarację inline, oraz makra z pliku Math.h.

Klasa Simulation, jako największa została rozbita na wiele plików implementacji, ze względu na łatwiejsze zarządzanie kodem programu. Wyodrębnione zostały funkcje obsługi myszy, klawiatury, funkcja renderowania, inicjalizacji oraz rekonfiguracji widoku kamery oraz okna.

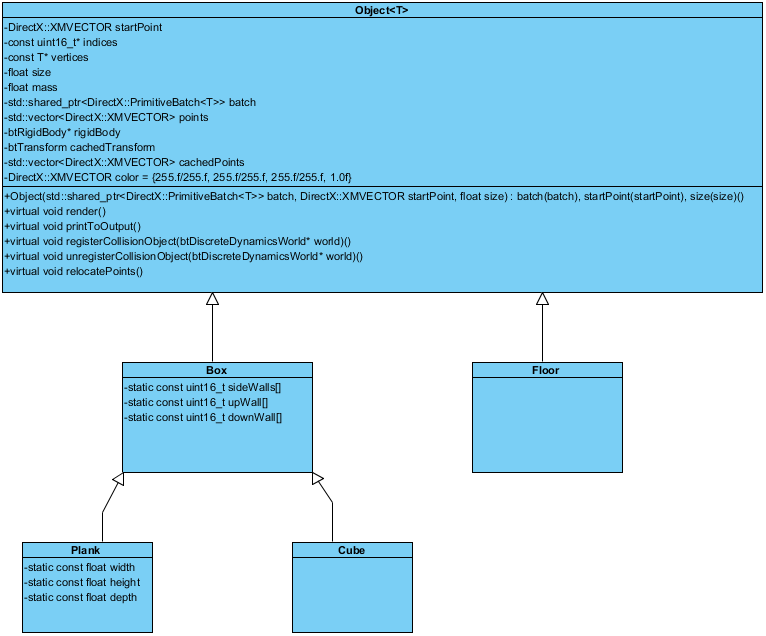
Poszczególne klasy opisane zostaną poniżej,

* + 1. Klasa Simulation

Główną klasą, na którą należałoby spojrzeć analizując działanie tej aplikacji, jest klasa Simulation. Jest to klasa, której metoda initialize inicjuje powstanie całego potoku renderowania DirectX, oraz obiektów silnika fizycznego, przeprowadzając dodatkowo jego konfigurację. Interesującą z punktu widzenia użytkownika metodą tej klasy jest jeszcze metoda initScenario, która poprzez umieszczoną wewnątrz funkcję definiuje jakie obiektu mają się pojawić w środowisku symulacji.

* + 1. Klasa Object

Klasa Object jest to klasa bazowa dla klas mających być wyrenderowanymi oraz posiadających fizykę. Drzewo obiektów prezentuje się następująco:



Rysunek 1 Drzewo dziedziczenia klasy Object<T>

Jak widać na powyższym diagramie, rdzeniem jest klasa Object, która jako klasa szablonowa, definuje jaki typ wierzchołka otrzyma dany obiekt, a może to być np. DirectX::VertexPositionColor reprezentujący informację na temat położenia oraz koloru wierzchołka lub DirectX::VertexPositionColorTexture, reprezentujący informację na temat pozycji, koloru oraz współrzędnej tekstury, która do wierzchołka jest przypisana. Klasie Box, typ definuje makro TEXTURED\_VERTEX\_TYPE.

* + 1. Przestrzeń nazw Scenario

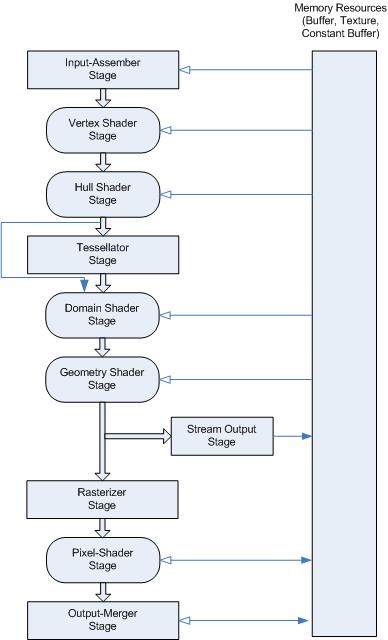
Plik Scenario.h definuje przestrzeń nazw Scenario, która to jest zbiorem 3 funkcji, odpowiadających za wygenerowanie odpowiedniej scenerii dla użytkownika do testów aplikacji. Funkcje te to:

* boxTowers – generuje scenę wież z pudełek
* plankDomino – generuje scenę z domino
* houseOfCards – generuje scenę domku z klocków

Pominę ich opis ponieważ ten został już napisany podczas definiowania założeń.

* 1. Opis działania
     1. Potok renderowania

Potok renderowania w DirectX, to zbiór stanów/procesów, przez które są przepuszczane dane, które wygeneruje użytkownik, a które mają być przekształcone na 2 wymiarową tablicę pikseli, która ma zostać wyświetlona na ekranie. Składa się on z 10 stanów, które pokazuje poniższy diagram:



Rysunek 2 Potok renderowanie DirectX

Poziome strzałki symbolizują dostęp do pamięci urządzenia.

Każdy stan ma swoją określoną funkcję, a mianowicie:

* Input Assembler Stage – stan potoku, w którym definiowane są dane wejściowe do potoku
* Vertex Shader Stage – stan potoku w którym operuje się na wierzchołkach zdefiniowanych w poprzednim stanie potoku. Podczas trwania tego procesu dokonuje się np. transformacji lub morphingu.
* Hull Shader Stage – stan odpowiedzialny za przekształcenie przekazanych doń wierzołków w obiekty wyższego rzędu, tak zwane łaty.
* Tessalation Stage – stan potoku odpowiedzialny za przeprowadzenie tesalacji, czyli podzieleniu posiadanych wielokątów na mniejsze, przez co wyświetlany obiekt może być wyświetlany jako bardziej szczegółowy
* Domain Shader Stage – przekształca dostarczone łaty z poprzednich stanów na obiekty prymitywne takie jak linia czy trójkąt
* Geometry Shader Stage – stan potoku odpowiedzialny za przekształcenia wierzchołków, jednak różniący się od stanu „Vertex Shader Stage” tym, że może on operować na całych prymitywach takich jak linie, trójkąty etc.
* Stream Output Stage – stan potoku pozwalający zapisać dane przetworzone przez poprzednie stany do pamięci urządzenia.
* Rasterizer Stage – stan potoku odpowiedzialny za przekształcenie dotychczas wyliczonych danych na obraz 2D.
* Pixel Shader Stage – stan potoku odpowiedzialny za przekształcenia już danych o pixelach w celu np. wyliczania oświetlenia, lub aplikacji różnych postprocessów.
* Output Merger Stage – stan potoku odpowiedzielny za ostateczną decyzję co ma zostać wyświetlone, które pixele są widoczne (na podstawie testu głębokości), oraz dokonuje mieszania kolorów ostatecznych pixeli.

Warto wspomnieć że nie wszystkie te stany muszą być zaimplementowane, np. proces

tesalacji jest całkowicie opcjonalny.

Całość procesu przebiega w klasie Simulation. Przy wywołaniu metody initialize, zaraz po utworzeniu się obiektu okna aplikacji, zostanie wywołana metoda createDevice. Metoda ta jest odpowiedzialna za stworzenie obiektów definiującego abstrakcję nad kartą graficzną. Przy użyciu funkcji D3D11CreateDevice z pliku nagłówkowego d3d11.h, wypełniamy zawartość obiektów ComPtr wcześniej zadeklarowanych w pliku nagłówkowym. Odbywa się to w następujący sposób:

Listing 1Tworzenie urządzenia karty graficznej

HRESULT hr = D3D11CreateDevice(

nullptr,

D3D\_DRIVER\_TYPE::D3D\_DRIVER\_TYPE\_HARDWARE,

nullptr,

creationFlags,

featureLevels,

\_countof(featureLevels),

D3D11\_SDK\_VERSION,

device.ReleaseAndGetAddressOf(),

&dxFeatureLevel,

deviceContext.ReleaseAndGetAddressOf()

);

if (hr == E\_INVALIDARG) {

hr = D3D11CreateDevice(

nullptr,

D3D\_DRIVER\_TYPE\_HARDWARE,

nullptr,

creationFlags,

&featureLevels[1],

\_countof(featureLevels) - 1,

D3D11\_SDK\_VERSION,

device.ReleaseAndGetAddressOf(),

&dxFeatureLevel,

deviceContext.ReleaseAndGetAddressOf()

);

}

Widzimy jak w pierwszym wywołaniu metody próbujemy uzyskać od systemu obiekt dzięki któremu możemy sterować urządzeniem sprzętowym. Funcje te nie różnią się tlyko jednym parametrem, a mianowicie wskaźnikiem na featureLevels, który to jest tablicą stałych liczbowych reprezentujących wersję obsługiwanej biblioteki DirectX. Blok ten próbuje pobrać od systemu urządzenie kompatybilne z DirectX11, gdy poniesie klęskę, pobiera urządzenie obsługujące poziom niżej. Jeżeli wówczas mu się nie uda aplikacja zostanie zakończona z błędem. Tworzą się dwa bardzo istotne obiekty, a mianowicie obiekt device (ID3D11Device), będący abstrakcją nad urządzeniem graficznym, który np. odpowiedzialny jest za tworzenie zasobów oraz deviceContext (ID3D11DeviceContext) będący obiektem służącym do generowania komend renderowania.

Następnie w tejże funkcji, tworzone są bufory, tekstury, shadery oraz sampler. Tekstury w pojęciu grafiku komputerowej rozumiemy jako obraz, bitmapę, która zostaje zmapowana na obiekt trójwymiarowy. W DirectX jednak teksturę rozumie się jako zasób, nie muszący być przy tym w żaden sposób powiązany z obrazem. Wyróżniamy 3 typy tekstur, 1 wymiarowe, 2 oraz 3 wymiarowe. Można je sobie wyobrazić jako n-wymiarowe tablice znane chociażby z C++. Pojedyncza komórka tekstury nazwana została tekselem, czyli najmniejszym zdefiniowanym typem zdolnym do bycia weń zapisanym i odczytanym. Teksturę można utworzyć również bez typu, lecz wiąże się to z komplikacjami na dalszych etapach tworzenia potoku. W naszym przypadku tworzą się tekstury w rozumieniu grafiki komputerowej, czyli wczytany zostaje plik z obrazem reprezentującym pudełko.

Następnie tworzone są shadery. Shader jest to kod, który interpretowany jest przez kartę graficzną. Do jego tworzenia służy język HLSL (High Level Shader Language). Został on stworzony w celu wykorzystania potencjału współczesnych kart graficznych, które zdolne są do szybkiego przetwarzania wielu obiektów równolegle. Język ten wyposażony jest w typy potrzebne do podstawowych operacji używanych podczas tworzenia aplikacji graficznych, takie jak np. n-wymiarowe wektory, macierze oraz typy liczbowe, definiując im jednocześnie operatory. Sam język zawiera również wiele przydatnych funkcji takich jak odwracanie macierzy, iloczyn wektorowy lub transpozycje macierzy, co czyni go wygodnym podczas wykonywania różnych operacji, dostarczając przy tym wydajność GPU.

W przypadku tej aplikacji shadery tworzone są przez DirectXTK, ponieważ ich samodzielna realizacja wiąże się z samodzielnym zarządzaniem pamięcią oraz implementowaniem funkcjonalności o nietrywialnym modelu matematycznym. Jeden odpowiedzialny jest za renderowania obiektów mających dane tylko o ich kolorach, drugi, obiektów mających dane o kolorach oraz współrzędnych mapowanych nań tekstury. Oba posiadają dane o położeniu obiektu. By teksturowanie z udziałem shaderów mogło zadziałać, potrzebny jest tzw. Sampler. Tworzy się go poprzez zdefiniowanie stanu potoku, który to jest jednym z elementów stanu „Pixel Shader Stage”, a jest odpowiedzialny właśnie enkapsulację informacji na temat próbkowania tekstury.

Następną wywołaną metodą jest metoda createResources, również z klasy Simulation. W niej tworzone są widoki zasobów, czy definicje dostępu do danych zasobów takich jak np. cel renderowania (może to być bufor wyjściowy na kartę graficzną, lub obszar pamięci, z którym dowolnie możemy się obejść). Funkcja ta jest każdorazowo wywoływana przy zmianie wielkości okna, ponieważ od niej bezpośrednio zależy np. rozmiar buforów testu głębokości, czy buforów używanych podczas podwójnego lub potrójnego buforowania klatek.

W pierwszym etapie sprawdzane jest czy wybrany aktualnie poziom multisamplingu jest przez kartę wspierany. Klasa Simulation definiuje tablicę możliwych poziomów multisamplingu, ale możliwym jest że nie każda karta obsłuży je wszystkie. Następnie tworzony jest tzw. SwapChain, czyli obiekt odpowiadający za zarządzanie buforami klatek, konfigurując go poprzez wypełnienie struktury DXGI\_SWAP\_CHAIN\_DESC, z odpowiednim sufiksem w zależności od definiowanego łańcucha. Następnie tworzony jest tzw. bufor tylni (back buffer), czy bufor aktualnie generowanej klatki oraz do niego tworzony jest widok (Render Target View) poprzez interfejs urządzenia.

Jako że będziemy potrzebowali skorzystać z testu głębokości (depth-stencil test), musimy utworzyć określone doń zasoby. W przypadku DirectX zasób ten to również widok (Depth Stenicl View), który również tworzony jest poprzez interfejs urządzenia. Dodatkowo tworzony jest proces testu głębokości definiowany strukturą D3D11\_DEPTH\_STENCIL\_DESC. Następnie definiuje się stan rasteryzacji (Rasterizer Stage), który definiuje się poprzez utworzenie instancji klasy CD3D11\_RASTERIZER\_DESC, która to jest klasą pomocniczą zdefiniowaną w nagłówku d3d11.h, a następnie przekazanie jej do metody interfejsu urządzenia.

Należy wspomnieć iż istnieje możliwość włączenia warstwy debugowania na karcie graficznej. Warstwę tę włącza się ponieważ bardzo trudnym zadaniem jest stworzenie programu pozwalającego na podglądanie w czasie rzeczywistym zasoby karty, oraz ze względu na fakt iż na karcie zachodzą procesy równoległe w znaczącej ilości, niemożliwym wręcz byłoby śledzenie wykonania kodu. Warstwa debugowania pozwala jednak aplikacji na sprawdzanie podanych parametrów w czasie wykonania kodu, zanim ten zostanie wykonany przez kartę graficzną, co znaczenie ułatwię pisanie aplikacji. Jest ona domyślnie wyłączona ze wzgląd na wydajność.

* + 1. Tworzenie obiektów świata

Obiekty świata, tj. obiekty będące obiektami podatnymi na interakcje (obiekty fizyczne), tworzą się poprzez konstruktory poszczególnych z nich. Jednak ich obecność musi zostać odnotowana w silniku Bullet, dlatego każdy z nich posiada metodę registerCollisionObject oraz unregisterCollisionObject. Metoda pierwsza musi zostać wywołana gdy chcemy uczynić nasz obiekt fizycznym. W przeciwnym razie, będzie on statycznym obiektem widocznym na ekranie. Statycznym obiekt można uczynić wyrejestrowując go metodą wymienioną jako drugą.

Jeżeli chcemy by obiekt został wyrenderowany musimy go dodać do odpowiednich wektorów. W klasie Simulation zdefiniowane są 2 wektory obiektów, w zależności od typów ich wierzchołków:

* std::vector<std::shared\_ptr<Object<DirectX::VertexPositionColor>>> coloredShapes
* std::vector<std::shared\_ptr<Object<TEXTURED\_VERTEX\_TYPE>>> texturedShapes

Wektory te są przeiterowywane w każdej iteracji pętli symulacji, w celu ich wyrenderowania na ekranie.

* 1. Opis użytkowania
     1. Sterowanie

Poruszanie się po przestrzeni symulacji zostało zaimplementowane poprzez odpowiednie klawisze klawiatury. Naciskając kolejno:

* **W** – kamera poruszać się będzie w kierunku, w którym spogląda kamera
* **S** – kamera poruszać się będzie w kierunku przeciwnym do kierunku spoglądania kamery
* **A** –kamera poruszać się będzie na lewo, od kierunku spoglądania kamery
* **D** –kamera poruszać się będzie na prawo, od kierunku spoglądania kamery

Poruszanie się w tych kierunkach odbywać się będzie równolegle do płaszczyzny wyznaczającej podłoże dla znajdujących się nań obiektów. Aplikacja umożliwia również poruszanie się w dół i w górę, tym razem prostopadle do podłoża. Naciskając kolejno:

* **Q** – kamera poruszać się będzie do góry
* **E** – kamera poruszać się będzie w dół

Za pomocą myszy możliwe jest rozglądanie się wokół punktu, który wyznacza położenie kamery. Przesuwając myszkę odpowiednio:

* w prawo, przekraczając granicę 80% szerokości okna, kamera obróci się w prawo
* w lewo, przekraczając granicę 20% szerokości okna, kamera obróci się w lewo
* w dół, przekraczając granicę 80% wysokości okna, kamera obróci się w dół
* w górę, przekraczając granicę 20% wysokości okna, kamera obróci się w górę

Analogicznie przesuwając kursor myszy w prawy górny róg ekranu, kamera obróci się jednocześnie w prawo i w górę.

Wystrzeliwanie obiektów odbywa się poprzez kliknięcie lewego klawisza myszy, aby obiekt od razu po kliknięciu zaczął się poruszać należy mieć odblokowany tryb symulacji fizyki, który jest domyślnie wyłączony, a uruchamia się go i blokuje za pomocą klawisza **Z**. Dodatkowo, by umożliwić użytkownikowi minimalne polepszenie jakości obrazu, może on kontrolować poziom multisamplingu wykonywanego przez kartę graficzną. Poziom ten podnosi się za pomocą klawisza **P**, i jest on podnoszony kolejno po obsługiwanych trybach, nie pozwalając na dalsze jego podnoszenie. Analogicznie klawisz **O**, poziom ten zmniejsza.

Wybór stosownego scenariusza odbywa się poprzez wybranie go z rozwijanego menu „Scenarios”, w lewym górnym rogu okna aplikacji.

1. Testy
   1. Plan testów
   2. Wyniki
   3. Wnioski
2. Literatura

[1] F. D. Luna. Introduction to 3D Game Programming with DirectX 11, Wydawnictwo HELION, 2014