Uniwersytet Mikołaja Kopernika
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Mateusz Sławkowski

Nr albumu: 273054

Praca inżynierska
na kierunku informatyka stosowana

Tutorial: symulacje wykorzystujące Nvidia PhysX z wizualizacją w OpenGL.

Opiekun pracy dyplomowej
dr hab. Jacek Matulewski
Katedra Informatyki Stosowanej

Toruń 2020

Pracę przyjmuję i akceptuję
……..
data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej
.....
data i podpis pracownika dziekanatu

Chciałbym podziękować
mojemu promotorowi
dr Jackowi Matulewskiemu
za poświęcony czas oraz cenną
pomoc przy pisaniu tej pracy.

Spis treści

Streszczenie

Praca przedstawia możliwości biblioteki PhysX w symulacjach fizycznych, co jest zilustrowane aplikacją symulującą stół bilardowy z kulami, które polegają prawom fizyki, m.in. można je rozbić i zderzają się o siebie oraz odbijają się od stołu. W aplikacji tej grafika jest renderowana z użyciem biblioteki OpenGL. Główny model stołu bilardowego wczytany jest z pliku w formacie *wavefront*. Z punktu widzenia biblioteki PhysX obiekt ten jest aktorem statycznym, czyli oddziałuje z innymi obiektami, ale sam nie podlega dynamice. Przygotowana została także funkcja generująca sfery imitujące kule bilardowe. Przedstawiono także sposób rysowania wielu takich samych obiektów na scenie. Dzięki bibliotece PhysX sfery te stały się dynamicznymi, wzajemnie na siebie odzdziaływującymi kulami. Aby móc lepiej przyglądać się scenie, zdefiniowane zostały także funkcje imitujące zachowanie kamery, które pozwalają na poruszanie jej po scenie.

Słowa kluczowe: PhysX, OpenGL, symulacja, aktor

Summary

This work presents the capabilities of PhysX library in physical simulations, which is illustrated by the application simulating pool table with balls which are subject to physical laws, i.a.they can be shattered and they collide with each other and with the table. In this application the graphics is rendered with the use of OpenGL library. The main model of pool table is loaded from *wavefront* type file. In terms used by PhysX library creators this object is called a static actor, which means it interacts with other object but is not subject to the dynamics itself. A code for generating spheres which are meant to imitate billard balls as well as code responsible for drawing multiple identical objects on the scene were also prepared and discussed. Thanks to the PhysX library, these spheres became dynamic, mutually affecting each other balls. To have a better look at the scene, the function to control the camera is also implemented.

Key words: PhysX, OpenGL, simulation, actor

Wstęp

Wiele współczesnych gatunków gier, zarówno nisko jak i wysokobudżetowych na komputery osobiste, konsole oraz sprzęt mobilny, opartych jest w znacznym stopniu na fizyce. Oczywistymi przykładami są gry wyścigowe z fizyką pojazdów, gry akcji z fizyką otoczenia, ale również takie gry jak mobilne: *Angry Birds* lub *Cut the rope*. O sukcesie tych gier nie zdecydowała tylko atrakcyjna grafika, ale przede wszystkim mechanika gry oparta na fizyce.

Tak się składa, że dominującym graczem w obu polach, a więc grafiki i fizyki, jest obecnie Nvidia. Oczywiście firma ta najbardziej znana jest z produkcji kart graficznych, ale to, że te karty mogą również obsługiwać silnik PhysX kupiony i rozwijany przez tę firmę stawia ją na uprzywilejowanej pozycji w wyścigu z innymi silnikami dostępnymi na rynku. Co więcej, Nvidia udostępnia także darmowe narzędzia dla programistów korzystających z PhsyX.

Rozpoczynając naukę nowego framwork'u naturalnym źródłem wiedzy jest oficjalna dokumentacja [1] [2]. W przypadku Nvidia PhysX jest ona bardzo obszerna, a przy tym nie ma charakteru tutorialu, który ułatwiłby naukę. Co więcej sporo w niej luk i niedopowiedzeń, co może łatwo zniechęcić do PhysX. Także fakt, że jest dostępna jedynie w języku angielskim może stanowić dodatkową barierę. Stąd pomysł na powstanie tej pracy. Jej celem jest przygotowanie tutorialu w języku polskim, który pomoże rozpocząć naukę silnika PhysX, pokonać pierwsze trudności związane z instalacją i wprowadzić najważniejsze koncepcje, jakie należy poznać, aby wydajnie z niego korzystać.

1. Wprowadzenie teoretyczne

1.1 Nvidia PhysX SDK

PhysX SDK (ang. *software developer kit*) to skalowalne open-source (na licencji BSD 3) narzędzia do oprogramowywania silnika fizycznego rozwijane przez Nvidie jako część usługi Nvidia GameWorks. PhysX posiada wsparcie na wielu platformach, takich jak: Windows, Linux, Ios, Android i inne. Dodatkowo narzędzia te są wbudowane w najbardziej popularne silniki gier komputerowych takich jak Unreal Engine czy Unity3D [3]. Podczas pisania niniejszej pracy wykorzystywana była najnowsza wersja PhysX SDK 4.1.1 z 13 sierpnia 2019 roku.

W silniku fizycznym PhysX najważniejszym obiektem jest scena reprezentująca przestrzeń, w której znajduje się reszta obiektów. Do sceny można dodawać aktorów. Są to obiekty posiadające zdefiniowaną masę oraz kształt. Powszechnie nazywa się je „bryłami sztywnymi" (ang. rigid-body), ponieważ ich zachowanie wyznaczane jest przez prawa dla bryły sztywnej. Ich najważniejszą własnością jest fakt, że zachowują swoją oryginalną formę niezależnie od użytej na nich siły [4]. Dzielą się na dwa typy:

- bryły sztywne dynamiczne – są to obiekty na które wpływają prawa fizyki,
- bryły sztywne statyczne – są to obiekty na które prawa fizyki nie mają żadnego wpływu, ale oddziałują na inne obiekty na scenie.

Podczas symulacji, aktorzy dynamiczni mogą wpływać na siebie wzajemnie chociażby odbijając się od siebie. Bryły statyczne świetnie nadają się do tworzenia otoczenia, które nie powinno mieć możliwości zmiany żadnych ze swoich parametrów fizycznych np. wielkie skały. Umieszczenie na scenie obu rodzajów aktorów bardzo urozmaica symulację. Podstawowymi parametrami aktorów są: kształt, rozmiar, masa oraz rodzaj materiału, który decyduje m.in. o współczynniku tarcia.

1.2 OpenGL

OpenGL jest środowiskiem służącym do tworzenia aplikacji z grafiką zarówno 2D, jak i 3D. Jest jednym z najpowszechniej używanych API (ang. *Application*

Programming Interface) czyli interfejsem programistycznym, umożliwiającym kontrolę nad kartą graficzną przeznaczoną do tworzenia aplikacji na wielu platformach systemowych. Dzięki wbudowanym funkcjom służącym wizualizacji, mapowaniu tekstur czy tworzenia specjalnych efektów, umożliwia szybkie tworzenie i rozbudowę korzystających z niego programów, bez ograniczeń narzucanych przez gotowe silniki gier [5].

Rozwojem biblioteki zajmuje się grupa Khronos złożona z ponad 150 firm tworzących zarówno sprzęt fizyczny jak i oprogramowanie [6]. W jej skład wchodzą min. członkowie takich spółek jak Nvidia, Samsung, Intel, Huawei, Sony, Valve, AMD, Apple czy Google [7].

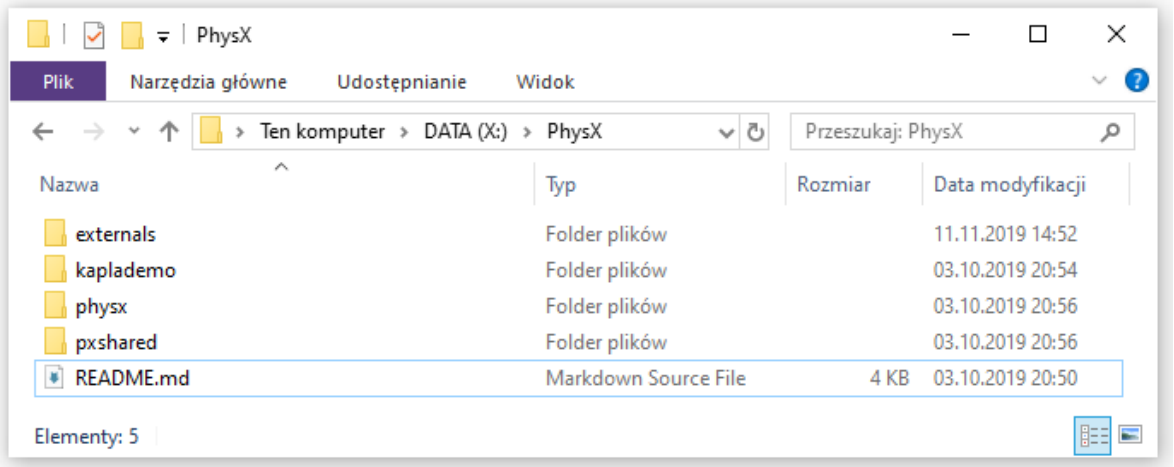
OpenGL wbrew powszechnemu przekonaniu nie jest *open-source* [8]. Jego kod źródłowy nie został bowiem opublikowany. Natomiast OpenGL opisuje interfejs programistyczny oraz przewidziane zachowanie, jest on więc *open-specification*. Jednak w przeciwieństwie do standardów ISO, za których dostęp trzeba zapłacić, OpenGL może zostać pobrany całkowicie za darmo.

2. Przygotowanie środowiska do pracy

2.1 Generowanie bibliotek PhysX

Aplikacja przygotowana na potrzeby tego poradnika będzie wykorzystywała środowisko programistyczne Microsoft Visual Studio Community 2017. W tym rozdziale przedstawiony zostanie proces generowania odpowiednich bibliotek PhysX niezbędnych do dalszej pracy.

Najpierw należy pobrać kod źródłowy aktualnej wersji PhysX SDK z oficjalnego GitHuba Nvidii [9]. W przeszłości, aby mieć możliwość korzystania z SDK, potrzebna była rejestracja w programie developerskim Nvidia GameWorks i uzyskanie odpowiednich dostępów. Jednakże wraz z wersją 4.0 wszystkie kody źródłowe, również do poprzedniej wersji SDK zostały powszechnie udostępnione. Po pobraniu i rozpakowaniu na wybranym dysku głównego folderu, należy zadbać, aby pozostawić jego oryginalną nazwę, ponieważ w razie jej zmiany, skrypty generujące biblioteki nie będą mogły znaleźć odpowiednich ścieżek dla pobranych presetów. Aktualną strukturę folderów PhysX SDK (wersja 4.1.1) można zobaczyć na rysunku 2.1.1.



Rys. 2.1.1 Struktura folderów w PhysX SDK 4.1.1

Następnym krokiem będzie wygenerowanie bibliotek odpowiednich dla platformy systemowej. Aby to zrobić, trzeba uruchomić plik wykonywalny *generate_projects.bat*, który do poprawnego działa wymaga, aby został zainstalowany Python w wersji co najmniej 2.7.6 oraz CMake w wersji minimalnie 3.12 lub 3.14 w przypadku korzystania z

Visual Studio Community 2019 [10]. Dodatkowo CMake musi zostać zainstalowany pod konkretną ścieżką *„PhysX\externals\cmake\x64”*, ponieważ w innym wypadku plik wykonywalny *.bat* go nie odnajdzie (rys. 2.1.2). Po uruchomieniu pozostaje wybrać pożądany *preset*. Na potrzeby tej pracy będzie to *vc15win64*.

```
if exist "%PHYSX_ROOT_DIR%/../externals/cmake/x64/bin/cmake.exe" (
    SET "PM_CMAKE_PATH=%PHYSX_ROOT_DIR%/../externals/cmake/x64"
    GOTO CMAKE_EXTERNAL
)

where /q cmake
IF ERRORLEVEL 1 (
    ECHO Cmake is missing, please install cmake version 3.12 and up.
    set /p DUMMY=Hit ENTER to continue...
    exit /b 1
)
```

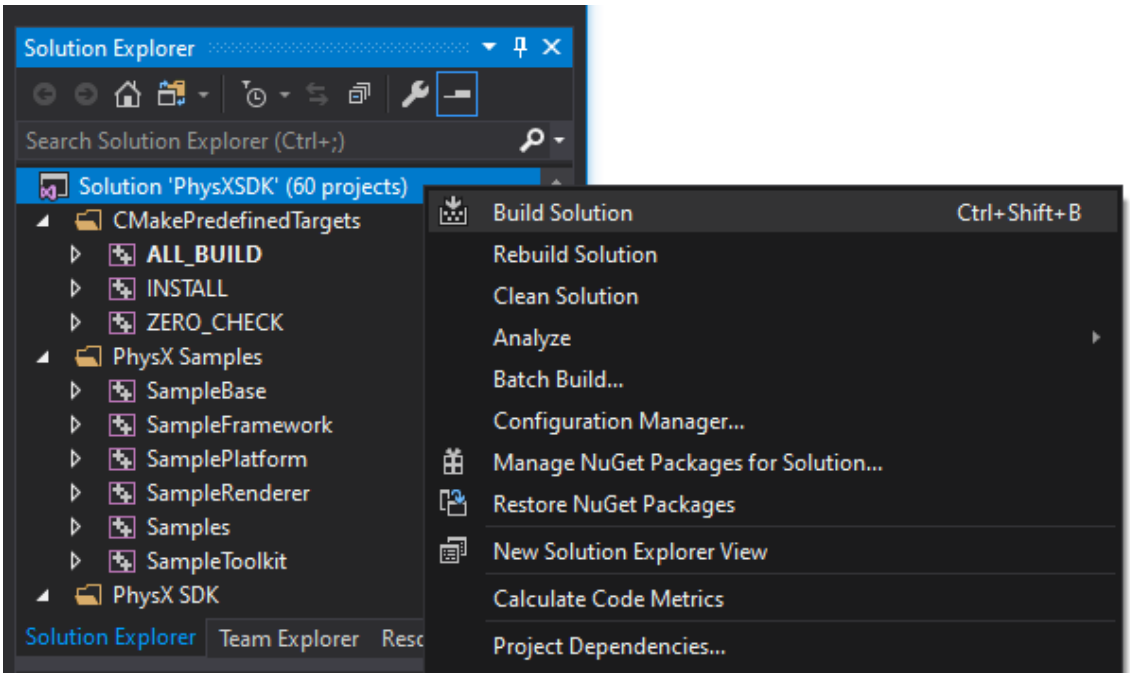
Rys. 2.1.2 Kod źródłowy odpowiedzialny za znalezienie CMake’a.

Kiedy program zakończy działanie, wygenerowane biblioteki powinny znajdować się w podfolderze *compiler/vc15win64/*. Następnym krokiem będzie uruchomienie znajdującego się tam pliku rozwiązania *PhysXSDK.sln*. Zanim biblioteki zostaną zbudowane, należy wybrać pożądaną konfigurację. Można wybrać spośród następujących wersji [11]:

- *Debug* – przydatna do analizowania błędów dzięki dodatkowym asercjom (ang. *assert)*, czyli predykatom pomagającym w identyfikacji błędow. Mogą one być jednak zbyt uciążliwe przy codziennej pracy z SDK. Optymalizacja jest wyłączona dla tej konfiguracji,
- *Checked* – zawiera kod wykrywający błędne parametry, a także niewłaściwe użycia API, które mogą powodować trudne do wyjaśnienia błędy podczas działania programów,
- *Profile* – pomija pewne kontrole, ale wciąż zawiera instrumentację pamięci a także niezbędne klasy do obsługi *Physics Visual Debuggera* (PVD) (zob. rozdział 4),
- *Release* – nastawiony na największą prędkość, pomija większość kontroli.

Programiści Nvidii do codziennej pracy zalecają konfigurację C*hecked*, która będzie także używana w tej pracy. Na przyszłość należy też pamiętać, że nie można

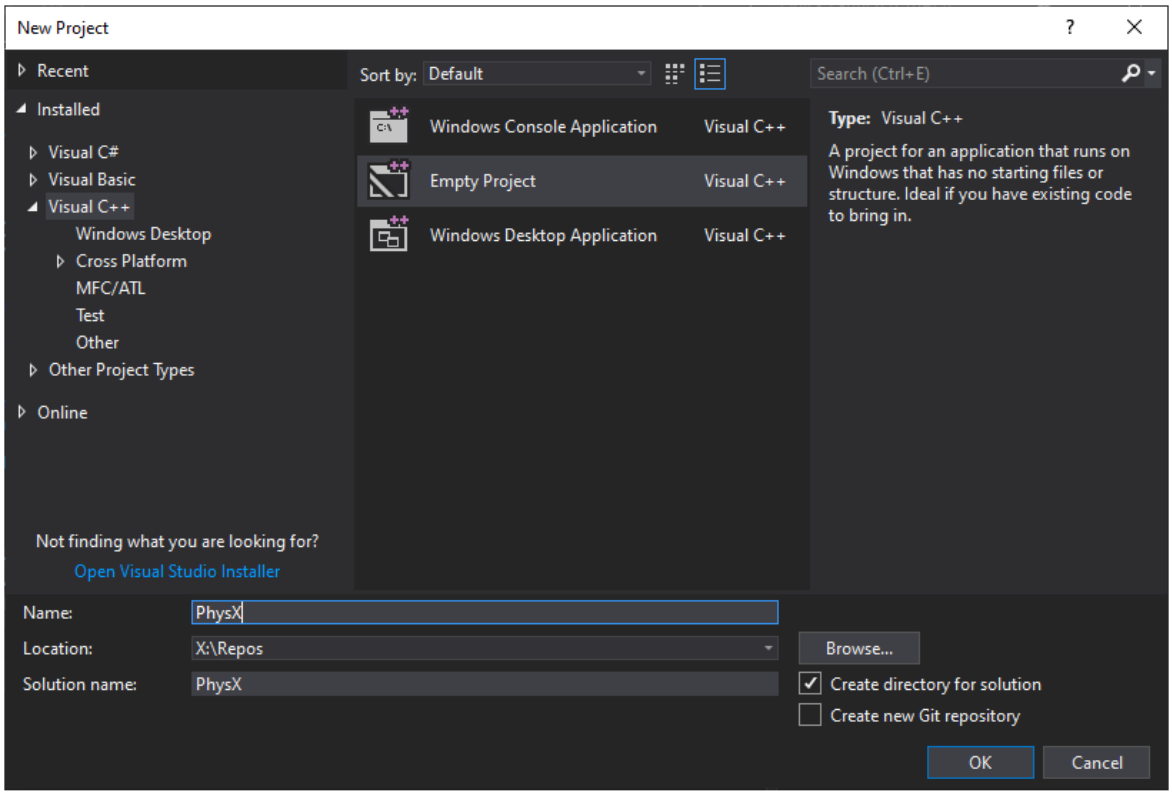
mieszać ze sobą pików z różnych konfiguracji, ponieważ w takim przypadku symulacja nie zadziała. Teraz pozostaje już tylko zbudować wszystkie biblioteki (rys. 2.1.3). Jeżeli kompilacja zakończy się powodzeniem, w podoknie *Outputu* pojawi się napis *Build Succeded*.



Rys. 2.1.3 Budowanie bibliotek PhysX

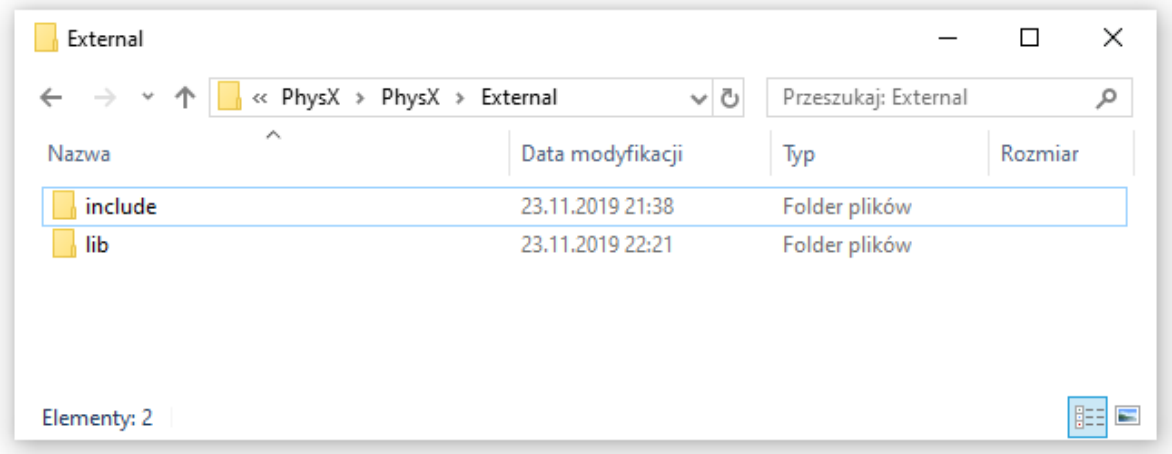
2.2 Konfiguracja projektu w Visual Studio Community 2017

Aby program wykorzystujący skompilowane przed chwilą biblioteki PhysX działał poprawnie, jego projekt musi zostać odpowiednio skonfigurowany. Najpierw jednak trzeba go stworzyć. Po otwarciu Visual Studio należy z menu głównego wybrać *File*, a następnie *New* i *Project*. Z okna, które się wówczas pojawi należy wybrać *Visual C++* a następnie *Empty Project* (rys. 2.2.1). Nazwa i lokacja projektu pozostają dowolne.



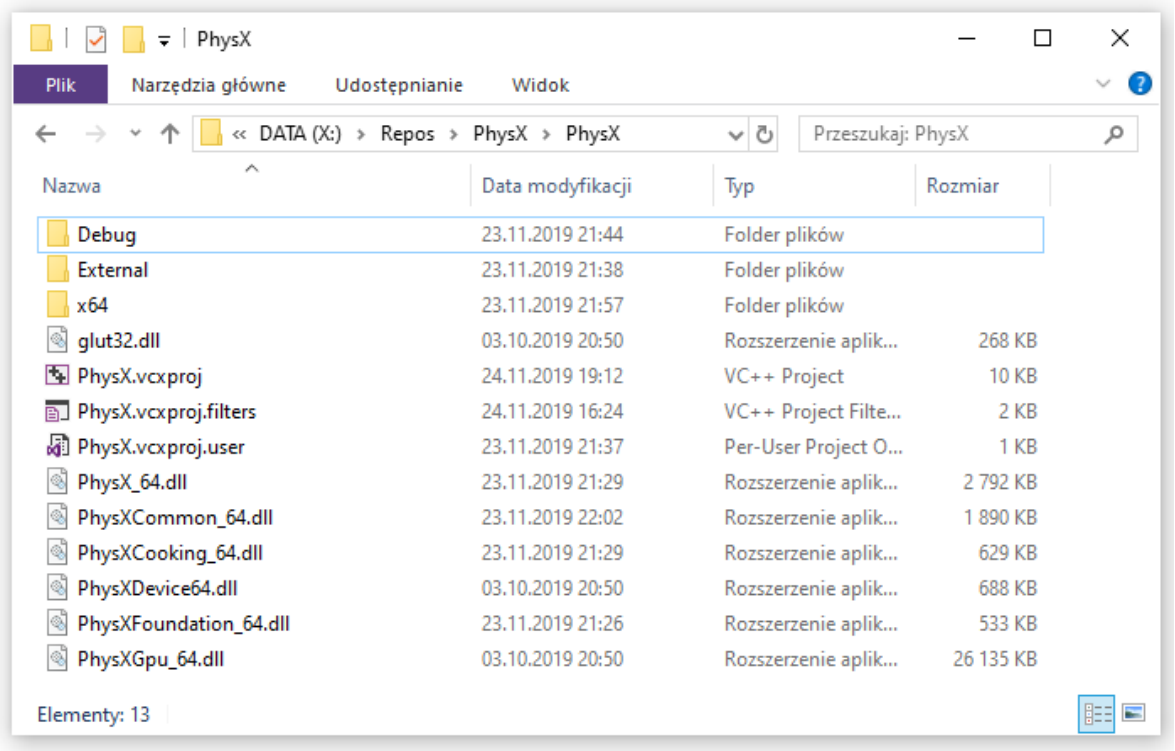
Rys. 2.2.1 Okno wyboru projektów

Aby tworzony program był przenaszalny i jego rozbudowa mogła być kontynuowana na innym komputerze, musi mieć dostęp do skompilowanych wcześniej bibliotek. Najlepiej jest utworzyć folder *External* w ścieżce projektu, a w nim umieścić folder *lib*, który będzie zawierał pliki bibliotek funkcji PhysX oraz *include*, w którym umieszczone będą ich odpowiednie pliki nagłówkowe. (rys. 2.2.2).



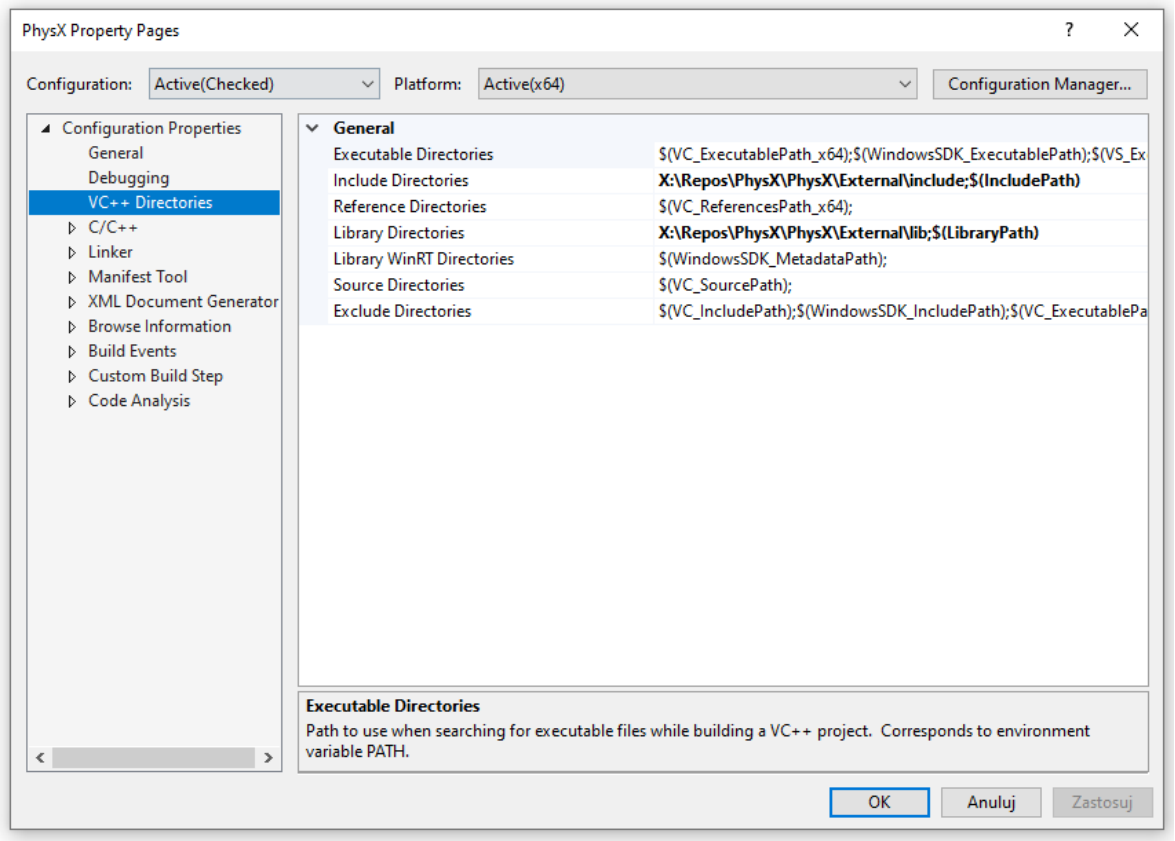
Rys. 2.2.2 Przykładowa struktura folderu projektu

Do folderu *include* należy skopiować całą zawartość folderów ze ścieżek *PhysX/physx/include*, a także z *PhysX/pxshared/include*. W pierwszym z nich znajdują się wszystkie pliki nagłówkowe oprócz tych z klasy `PxFoundation`, która będzie często używana w programie. Następnie do folderu *lib* należy przekopiować pliki z rozszerzeniem *.lib* oraz *.pdb* z katalogu znajdującego się pod ścieżką *PhysX/physx/bin/win.x86_64.vc141.mt/<wybrana_wersja_konfiguracji>/*. Pozostaje jeszcze przekopiować wszystkie pliki z rozszerzeniem *.dll*, znajdujące się w tym samym folderze co pliki *.lib* - do głównego folderu projektu (rys. 2.2.3).



Rys. 2.2.3 Pliki .dll umieszczone w głównym folderze aplikacji

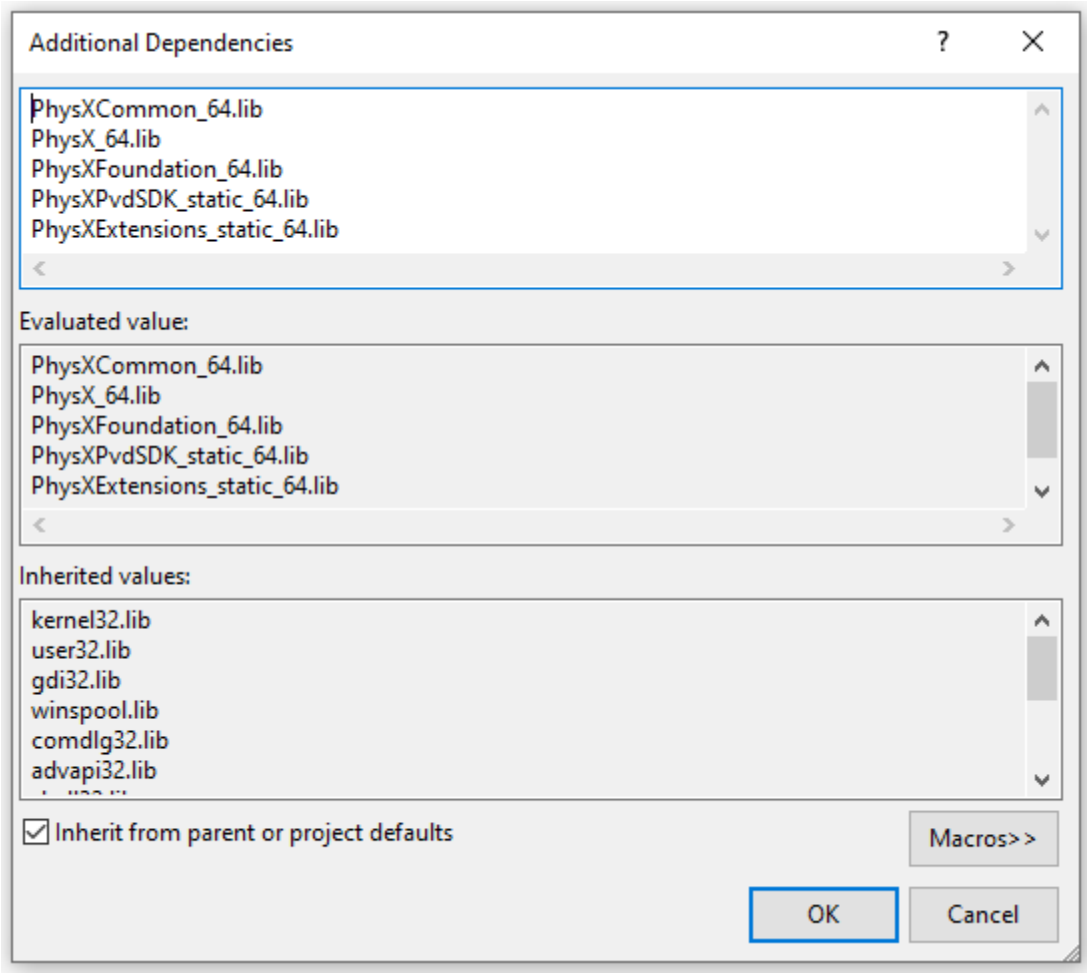
Kolejnym krokiem będzie konsolidacja (linkowanie) skopiowanych właśnie plików w Visual Studio. Aby to zrobić, należy otworzyć okno ustawień projektu, klikając prawym przyciskiem myszy na pozycję projektu w podoknie *Solution Explorer* i wybierając z menu pozycję *Properties*. Następnie w zakładce *VC++ Directories* należy zmienić ścieżki w *Include Directories* oraz *Library Directories* na te w folderze aplikacji (rys. 2.2.4).



Rys. 2.2.4 Okno opcji VC++ Directories w Visual Studio ustawionym folderami include oraz lib.

Kolejnym krokiem powinno być dodanie bibliotek, z których będzie trzeba skorzystać w projekcie do linii komend *Linkera*. Aby to zrobić, w otwartym już oknie opcji projektu, trzeba przejść do zakładki *Linker*, a potem w *Input*. Teraz należy rozwinąć pełne okno kategorii *Additional Dependencies* klikając w pole tekstowe a następnie w strzałkę i opcję *Edit*. W tym miejscu trzeba dopisać wszystkie biblioteki z których zamierza się korzystać w projekcie (rys. 2.2.5). Najbardziej potrzebne są:

- *PhysXCommon_64.lib* - wymagane zawsze,
- *PhysX_64.lib* – wymagane zawsze,
- *PhysXFoundation_64.lib* – wymagane zawsze,
- *PhysXPvdSDK_static_64.lib* – odpowiedzialne za funkcje umożliwiające łączenie się z PVD,
- *PhysXExtensions_static_64.lib* – zawierające domyślne implementacje części klas.



Rys. 2.2.5 Dodatkowe zależności Linkera

Pozostały ostatnie opcje do zmiany. Należy wejść do zakładki *C/C++*, a następnie w *Preprocesor* i dodać do definicji preprocesora makra PX_PHYSX_STATIC_LIB [12] oraz NDEBUG [13] w ten sam sposób w jaki dodawane były biblioteki w ustawieniach *Linker*. Są one potrzebne. W przypadku ich braku nie będzie można używać asercji wraz ze statycznymi bibliotekami, które będą wykorzystywane w projekcie. Teraz zostało już tylko w zakładce *Code Generation* ustawić *Runtime Library* na *MT* [14], ponieważ w taki sposób kompilowane są biblioteki *PhysX*. W przypadku korzystania z konfiguracji Debug, należy ustawić wartość *MT/d*.

Po wykonaniu wszystkich wymienionych wyżej kroków, aplikacja powinna działać bez zarzutu. Należy jednak wziąć pod uwagę, że czynności te mogą się różnić w zależności od wersji PhysX SDK. W razie problemów niezwiązanych z błędami w kodzie, warto przejrzeć ustawienia prostych dołączonych projektów ilustrujących narzędzia PhysX. Znajdujący się w nich kod często jest obszerniejszy i bardziej chaotyczny niż jego

fragmenty wykorzystane w tej pracy, dlatego przed bezkrytycznym skopiowaniem jego ustawień, warto najpierw dowiedzieć się do czego służą. W innym wypadku można pogorszyć stan tworzonej aplikacji. Te przykładowe projekty znajdują się w zbudowanej już wcześniej solucji *PhysXSDK.sln*.

3. Prosta aplikacja na silniku Nvidia PhysX

3.1 Tworzenie plików oraz podstawowa inicjalizacja

Pierwszym krokiem w rozwijaniu projektu korzystającego z biblioteki PhysX jest utworzenie pliku z kodem źródłowym C++ i funkcją `main`. Aby to zrobić, trzeba prawym przyciskiem myszy kliknąć na *Source Files* w oknie podglądu solucji a następnie wybrać *Add* oraz *New Item*, a potem *C++ File*. Plik można nazwać dowolnie, jednakże zwykle zaleca się, aby główny plik źródłowy nazywał się *Source.cpp* lub *Main.cpp*. Następnie należy dodać klasę, w której będą znajdować się wszystkie funkcje opisane w tej pracy. Aby to zrobić koniecznym jest dodanie kolejnego pliku do folderu *Source Files*, ale tym razem zamiast *New File* trzeba wybrać pozycję *Class* i w oknie, które się wówczas pojawi wpisać nazwę tworzonej klasy. Także ona może ona być dowolna. Dla ustalenia uwagi można nazwać ją jednak `PhysX`. Aby aplikacja budowała się poprawnie, w pliku *Source.cpp* trzeba zdefiniować funkcję `main` oraz dołączyć świeżo stworzoną klasę odpowiednią dyrektywą `#include`. Dobrą praktyką jest dodanie do pliku źródłowego dyrektywy `#ifndef` gwarantującej, że plik ten zostanie przetworzony przez kompilator jedynie raz [15]. W pliku źródłowym powinno się także dodać dyrektywę `#include` dołączającą nagłówek *PxPhysicsAPI.h*, który zawiera w sobie wszystkie pliki nagłówkowe zawarte w SDK. Jako że używana jest przestrzeń nazw *physx* można dopisać `using namespace phyx`, aby nie musieć uwzględniać tej przestrzeni nazw przy każdej zmiennej. Dla wygody można także dopisać `using namespace std`. W pliku *PhysX.cpp* należy jeszcze dołączyć plik nagłówkowy klasy oraz bibliotekę *iostream.h* (listing 3.1.1).

Listing 3.1.1 Początkowa zawartość pliku PhysX.h

```
#ifndef PHYSX_H
#define PHYSX_H

#include <PxPhysicsAPI.h>

using namespace physx;
using namespace std;

class PhysX {
  public:
```

```
    PhysX();
   ~PhysX();
  private:
};
#endif
```

Teraz czas przejść do stworzenia prywatnych obiektów wymaganych do inicjalizacji PhysX (listing 3.1.2). Pola, które będą przechowywały ich wskaźniki należy dla bezpieczeństwa zainicjować w liście inicjacyjnej konstruktorze wartością NULL (listing 3.1.3). W przeciwnym przypadku, będą zawierać tzw. „śmieciowe adresy" [16]. Trzeba także pamiętać o zwolnieniu ich pamięci, w destruktorze klasy (listing 3.1.4) w kolejności odwrotnej do ich inicjalizacji.

Listing 3.1.2 Prywatne obiekty w pliku PhysX.cpp

```
private:
  PxDefaultErrorCallback mDefaultErrorCallback;
  PxDefaultAllocator mDefaultAllocatorCallback;

  PxFoundation* mFoundation;
  PxPvd* mPvd;
  PxPhysics* mPhysics;
  PxDefaultCpuDispatcher* mDispatcher;
  PxScene* mScene;
  PxMaterial* mMaterial;
```

Listing 3.1.3 Lista inicjalizacyjna klasy PhysX

```
PhysX::PhysX() :
mFoundation(NULL),
mPvd(NULL),
mPhysics(NULL),
mDispatcher(NULL),
mScene(NULL),
mMaterial(NULL){}
```

Listing 3.1.4 Destruktor klasy PhysX

```
PhysX::~PhysX() {
    mMaterial->release();
    mScene->release();
    mDispatcher->release();
    mPhysics->release();
    mPvd->release();
```

```
    mFoundation->release();
    cout << ”All variables are released.” << endl;
    system(”Pause”);
}
```

Kolejnym krokiem będzie przypisanie wyżej stworzonym zmiennym właściwych obiektów. Po ich inicjalizacji można dodać instrukcje warunkową `if`, aby upewnić się, że obiekty zostały utworzone. Nic nie stoi na przeszkodzie, aby wszystkie inicjalizacje wykonać w jednej funkcji, jednakże stosując zasady czystego kodu, a konkretniej zasadę *single-responsibility* [17] w niniejszej pracy inicjalizacja każdego konkretnego obiektu znajduje się w osobnej funkcji. Następnie, aby nie wywoływać ich wszystkich po kolei w funkcji `main`, zostały one spięte w funkcję `initEssentialVariables` (listing 3.1.10), która uruchamia wszystkie funkcje inicjalizujące poszczególne obiekty. [18]

Klasa `PxFoundation` (listing 3.1.4) jest podstawą do tworzenia dalszych obiektów z PhysX SDK. Dozwolona jest tylko jedna instancja tego obiektu w programie. Zawiera on w sobie interfejs potrzebny do alokowania pamięci oraz interfejs odpowiedzialny za zgłaszanie błędów. W tworzonej aktualnie aplikacji wykorzystane będą ich domyślne implementacje, zdefiniowane w bibliotece *Extensions*, ale nic nie stoi na przeszkodzie, aby napisać je samemu.

Listing 3.1.4 Funkcja inicjalizująca obiekt PxFoundation

```
void PhysX::initFoundation() {
    mFoundation = PxCreateFoundation(PX_PHYSICS_VERSION,
        mDefaultAllocatorCallback,
        mDefaultErrorCallback);
    if (!mFoundation) cout << “PxCreateFoundation failed!” << endl;
}
```

Następnym krokiem będzie napisanie funkcji, która zajmie się nawiązywaniem łączności z aplikacją PVD (listing 3.1.5). Najpierw trzeba jednak zdefiniować trzy stałe (makra) używane jako argumenty funkcji `PxDefaultPvdSocketTransportCreate`, które przechowują parametry takiego połączenia (listing 3.1.6). Są nimi `PVD_HOST` przechowujący adres hosta aplikacji PVD, która łączy się z programem przy pomocy gniazda sieciowego TCP/IP i nasłuchuje na domyślnym porcie `PVD_PORT`. Trzeba także dodać stałą, której wartością jest interwał `PVD_TIMEOUT` po którym program ma rozłączyć się z PVD [19].

Listing 3.1.5 Funkcja inicjalizująca obiekt PxPvd

```
void PhysX::initPvd() {
    mPvd = PxCreatePvd(*mFoundation);
    if (!mPvd) cout << "PxCreatePvs failed!";
    PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(
        PVD_HOST, PVD_PORT, PVD_TIMEOUT);
    if (!transport) cout <<
        "PxCreateDefaultPvdSocketTransportCreate failed!" << endl;
    mPvd->connect(*transport, PxPvdInstrumentationFlag::eALL);
}
```

Listing 3.1.6 Wymagane dyrektywy define

```
#define PVD_HOST "127.0.0.1"
#define PVD_PORT 5425
#define PVD_TIMEOUT 1000
```

Funkcja `PxCreatePhysics` (listing 3.1.7) tworzy instancję klasy `PxPhysics`. Jej parametr `recordMemoryAllocations` decyduje o tym czy debugger ma mapować wszystko co zostało za-alokowane. Jeżeli tworzony program ma łączyć się z PVD trzeba podać w argumencie utworzony obiekt klasy `PxPvd`. W przeciwnym razie można podać `NULL`.

Listing 3.1.7 Funkcja inicjalizująca obiekt PxPhysics

```
void PhysX::initPhysics() {
    bool recordMemoryAllocations = true;
    mPhysics = PxCreatePhysics(PX_PHYSICS_VERSION,
        *mFoundation,
        PxToleranceScale(),
        recordMemoryAllocations,
        mPvd);
    if (!mPhysics) cout << "PxCreatePhysics failed!" << endl;
}
```

Funkcja `PxDefaultCpuDispatcherCreate` (listing 3.1.8) tworzy *dispatcher* [20] odpowiedzialny za organizowanie pracy CPU. Posiada dwa opcjonalne argumenty, z których pierwszy odpowiada za liczbę wątków których ma używać, a drugi definiuje tablicę zawierającą maski pomagające w rozprowadzaniu procesów CPU.

Listing 3.1.8 Funkcja inicjalizująca obiekt PxDispatcher

```
void PhysX::initDispatcher() {
```

```
    int numberOfThreads = 2;
    mDispatcher = PxDefaultCpuDispatcherCreate(numberOfThreads);
    if (!mDispatcher) cout <<
        „PxDefaultCpuDispatcherCreate failed!” << endl;
}
```

Ostatni z obiektów definiuje materiał aktorów, którzy będą tworzeni w następnym podrozdziale. W tym przypadku, pierwsze dwa parametry metody `createMaterial` odpowiadają za wartości tarcia statycznego oraz dynamicznego. Trzeci określa natomiast wartość współczynnika restytucji określającego stopień elastyczności zderzenia, czyli ilość energii kinetycznej utraconej podczas kolizji dwóch poruszających się obiektów.

Listing 3.1.9 Funkcja inicjalizująca PxMaterial

```
void PhysX::initMaterial() {
    PxReal staticFriction = 0.5f;
    PxReal dynamicFriction = 0.5f;
    PxReal restitution = 0.6f;
    mMaterial = mPhysics->createMaterial(staticFriction,
        dynamicFriction,
        restitution);
    if (!mMaterial) cout << “CreateMaterial failed!” << endl;
}
```

W pliku *Source.cpp* wystarczy teraz jedynie utworzyć obiekt klasy `PhysX`, a następnie wywołać na jego rzecz metodę `InitEssentialVariables` (listing 3.1.10). Kiedy wszystkie wymagane obiekty są już utworzone, czas stworzyć scenę.

Listing 3.1.10 Funkcja uruchamiająca wszystkie potrzebne inicjalizacje

```
void PhysX::initEssentialVariables() {
    initFoundation();
    initPvd();
    initPhysics();
    initDispatcher();
    initMaterial();
}
```

3.2 Przygotowanie sceny z aktorami

Scena jest abstrakcyjnym fragmentem przestrzeni na którym wyświetlane będą wszystkie symulacje. Klasa sceny może mieć tylko jedną instancję. Do jej utworzenia potrzebny będzie obiekt typu `PxSceneDesc`, który zawiera deskryptory opisujące scenę.

Aby poprawnie zainicjalizować obiekt `mScene` potrzebne będzie ustalenie wektora grawitacji, dodanie *dispatcher'a* CPU oraz filtru cieniowania, który w przypadku tej pracy będzie domyślnym filtrem z biblioteki *Extensions*. Pozostaje utworzyć scenę poprzez wywołanie metody `createScene` z klasy `PxPhysics` (listing 3.2.1).

Listing 3.2.1 Przykładowa inicjalizacja sceny

```
void PhysX::intiScene() {
    PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());
    sceneDesc.gravity = PxVec3(0.0f, -9.81f, 0.0f);
    sceneDesc.cpuDispatcher = mDispatcher;
    sceneDesc.filterShader = PxDefaultSimulationFilterShader;
    mScene = mPhysics->createScene(sceneDesc);
    if (!mScene) cout << "CreateScene failed" << endl;
}
```

Następnym krokiem będzie dodanie klienta debuggera PVD dla sceny, który udostępni użytkownikowi usługi związane ze zmianą pozycji na scenie, takie jak poruszanie kamerą czy natychmiastowe renderowanie. Obiekt klienta tworzony jest poprzez wywołanie metody `getScenePvdClient` na rzecz obiektu sceny. Po jego utworzeniu trzeba ustawić jego flagi odpowiedzialne za przesyłanie transmisji do aplikacji PVD (listing 3.2.2).

Listing 3.2.2 Przykładowa inicjalizacja Clienta Pvd Sceny

```
void PhysX::setSceneClient() {
    PxPvdSceneClient* pvdClient = mScene->getScenePvdClient();
    if (pvdClient) {
        pvdClient->setScenePvdFlag(
            PxPvdSceneFlag::eTRANSMI_CONSTRAINTS, true);
        pvdClient->setScenePvdFlag(
            PxPvdSceneFlag::eTRANSMI_CONTACTS, true);
        pvdClient->setScenePvdFlag(
            PxPvdSceneFlag::eTRANSMI_SCENEQUERIES, true);
    }
}
```

Kiedy połączenie z PVD jest już ustanowione, można zabrać się za tworzenie aktorów. Pierwszym krokiem będzie utworzenie statycznego aktora, który będzie służył za płaską powierzchnię podłoża, na której będzie miała miejsce cała symulacja (listing 3.2.3).

Listing 3.2.3 Przykładowa inicjalizacja powierzchni pod symulację

```
void PhysX::createPlane() {
```

```
    PxRigidStatic* groundPlane = PxCreatePlane(*mPhysics,
        PxPlane(0, 1, 0, 0), *mMaterial);
    if (!groundPlane) cout << „PxCreatePlane failed!” << endl;
    mScene->addActor(*groundPlane);
}
```

Następnie tworzeni są pozostali aktorzy i ustawiani są na scenie. W przypadku tego przykładowego programu, będą nimi po prostu sfery. Aby podczas symulacji zobaczyć działającą na nie grawitację zostaną ustawieni w stos. W tym celu zdefiniować należy funkcję, do której przekazywane będą wartości transformacji (tak naprawdę translacji) aktorów, wielkość stosu oraz promień sfer. Wewnątrz funkcji należy utworzyć wskaźnik, który będzie wskazywał na kształt aktora. Następnie w pętli `for` w każdej jej iteracji tworzeni będą aktorzy, którzy będą tak naprawdę dynamicznymi bryłami sztywnymi. Na koniec każdej iteracji sfery będą dodawane do sceny. Po zakończeniu pętli tworzącej aktorów wypada zwolnić pamięć przetrzymywaną przez wskaźniki (listing 3.2.4).

Listing 3.2.4 Przykładowa funkcja dodająca aktorów do sceny

```
void PhysX::createStack(const PxTransform& transformationVector,
        PxU32 height,
        PxReal radius) {
    PxShape* actorShape = mPhysics->createShape(
        PxSphereGeometry(radius), *mMaterial);
    for (PxU32 i = 0; i < height; i++) {
        for (PxU32 j = 0; j < height – I; j++) {
            PxTransform localTm(PxVec3(
                PxReal(j * 2) – PxReal(radius – i),
                PxReal(i * 2 + 1), 0) * radius);

            PxRigidDynamic* actorBody = mPhysics->createRigidDynamic(
                transformationVector.transform(localTm));

            actorBody->attachShape(*actorShape);
            PxRigidBodyExt::updateMassAndInertia(*actorBody, 10.0f);
            mScene->addActor(*actorBody);
        }
    }
    actorShape->release();
}
```

Następnie zdefiniowana jest dodatkowa zmienna prywatna typu `PxReal` o nazwie `stackZ. W` liście inicjalizacyjnej konstruktora jest zainicjowana wartośią równą `0.0f`. Zmienna ta pomaga określić wielkość przesunięcia względem siebie kolejnych stosów sfer

na osi *z*. By urozmaicić symulację, zdefiniowana jest dodatkowa funkcja o nazwie `createMultipleStacks` (listing 3.2.5) wywołująca metodę `createStack` (listing 3.2.4) pożądaną ilość razy. Funkcja ta jest wywoływana w funkcji `main`.

Listing 3.2.5 Funkcja tworząca wiele stosów

```
void PhysX::createMultipleStacks(PxU32 numberOfStacks, PxU32 stackHeight, PxReal
sphereRadius) {
    for (PxU32 i = 0; i < numberOfStacks; i++) {
       createStack(PxTransform(PxVec3(0,0, stackZ -= 10.0f)), stackHeight,
             sphereRadius);
    }
}
```

Pozostaje już tylko zdefiniować funkcję tworzącą pętle renderowania (listing 3.2.6) i uruchomić w niej wszystkie zdefiniowane metody klasy `PhysX` (z pliku *Source.cpp*). Metoda ta może pobierać parametr typu `PxReal` z domyślną wartością równą `1.0f/60.0f`. Parametr ten decyduje o częstości postępowania symulacji. W razie potrzebny zmniejszenia prędkości symulacji można użyć mniejszej wartości np. `1.0f/240.0f`.

Listing 3.2.6 Funkcja tworząca pętle symulacji

```
void PhysX::renderLoop(PxReal renderSpeed) {
    while (mScene) {
        mScene->simulate(renderSpeed);
        mScene->fetchResults(true);
    }
}
```

Pozostało uzupełnić plik *Source.cpp* o obiekt klasy `PhysX`, który powinien być globalny, aby mógł być wykorzystywany przez funkcje zwrotne (ang. callbacks), które zostaną opisane niżej. Trzeba także pamiętać o uruchomieniu omówionych wcześniej funkcji (listing 3.2.7).

Listing 3.2.7 Zaktualizowana funkcja main.

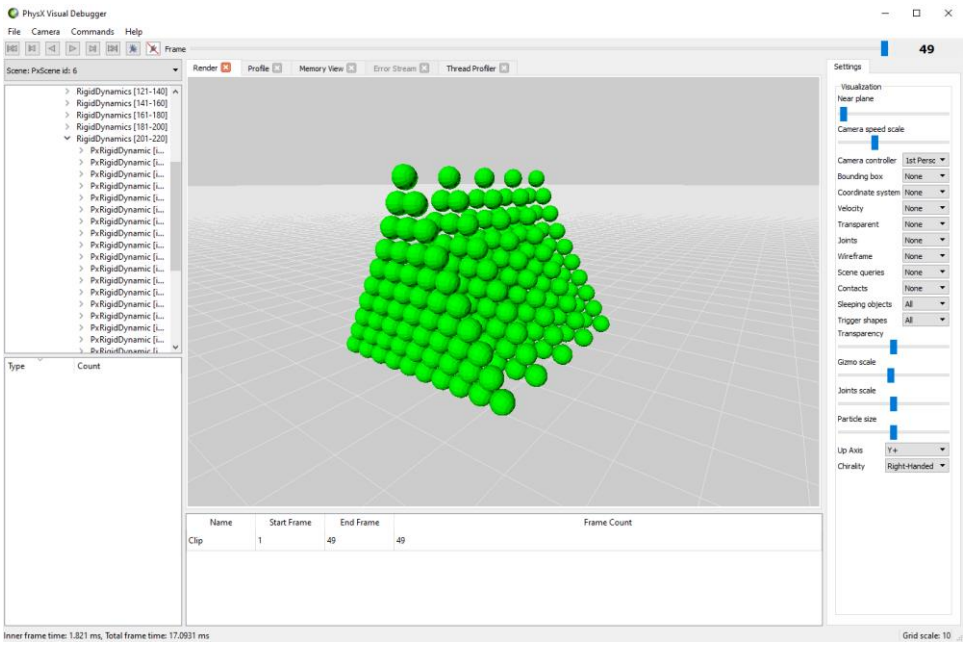
```
PhysX physicsObject;

int main() {
       PxReal slowMotion = 1.0f / 240.0f;
       PxReal normalMotion = 1.0f / 60.0f;

       physicsObject.initEssentialVariables();
```

```
        physicsObject.initScene();
        physicsObject.setSceneClient();
        physicsObject.createPlane();

        PxU32 numberOfStacks = 5;
        physicsObject.createMultipleStacks(numberOfStacks);
        physcisObject.renderLoop(normalMotion);
        return 0;
}
```

3.3 Uruchomienie aplikacji w PhysX Visual Debugger

Aktualnie stworzona aplikacja nie jest jeszcze w stanie sama uruchomić wizualizacji stworzonych wcześniej obiektów, których zachowanie i interakcja kontrolowane są poprzez bibliotekę PhysX. Aby sprawdzić zachowanie tych obiektów można użyć stworzonego przez Nvidie programu o nazwie „*PhysX Visual Debugger*" (PVD). Aplikacja ta pozwala wizualizować, debugować oraz oddziaływać ze stworzoną w PhysX reprezentacją fizyczną sceny.

Aby w PVD zobaczyć stworzoną wcześniej scenę, wystarczy, aby był uruchomiony zanim zostanie wystartowana wcześniej przygotowana aplikacja. Na rysunku 3.3.1 można zobaczyć niedawno utworzone stosy sfer.



Rys 3.3.1 Stosy sfer w aplikacji PVD

Przy standardowej, zdefiniowanej wcześniej w funkcji `main` (listing 3.2.7) prędkości `normalMotion` o wartości 1/60 symulacja postępuje dosyć szybko. Warto, wobec tego skorzystać ze zwolnionej wersji, aby móc dokładniej przyjrzeć się ruchom sfer.

W niniejszej pracy program PVD nie będzie szczegółowo omawiany, ponieważ większy nacisk położony zostanie na samodzielne wizualizowanie fizyki w środowisku OpenGL, jednak warto wspomnieć, że aplikacja ta pozwala na podgląd wielu parametrów aktorów takich jak ich klatki, punkty kontaktu, granice, prędkość, czy chociażby środek masy [21].

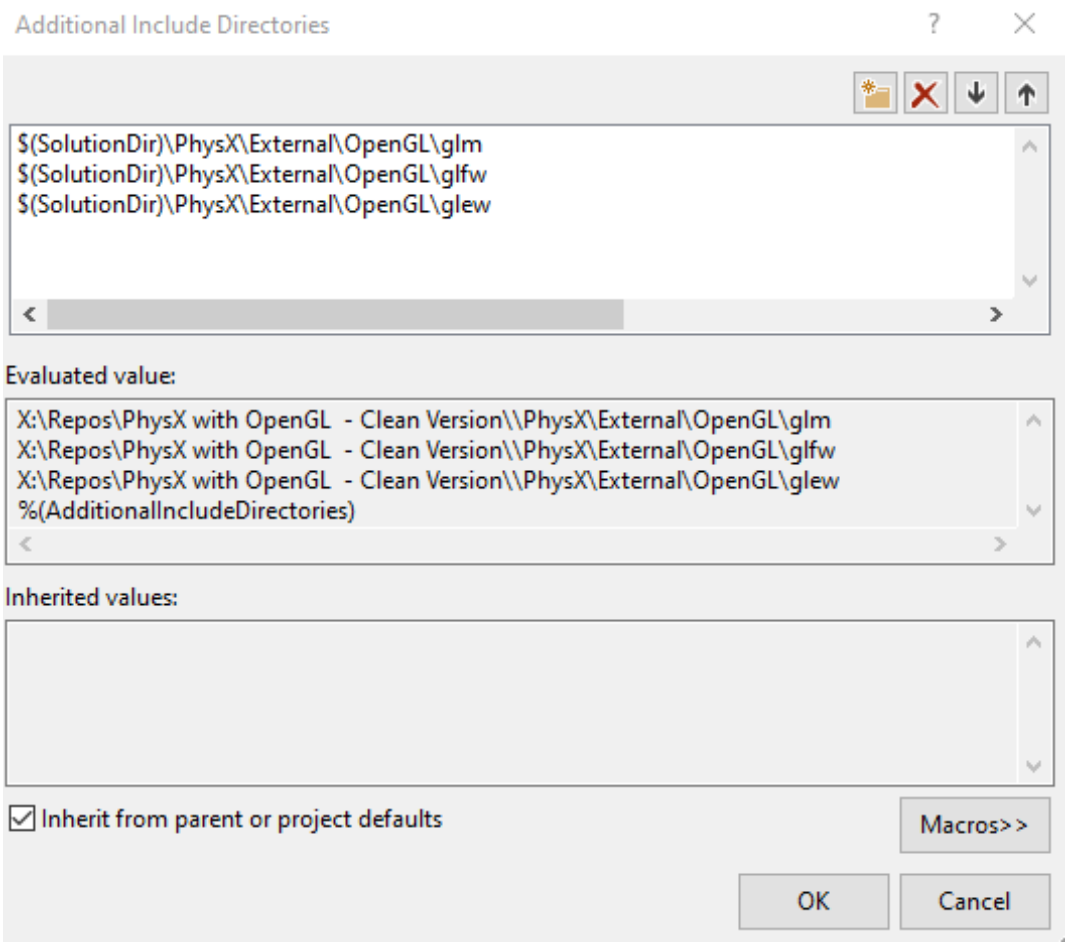
4. Uruchomienie symulacji w środowisku graficznym OpenGL

4.1 Wczytanie graficznych bibliotek

Aby korzystać z funkcjonalności biblioteki OpenGL nie trzeba pobierać żadnych dodatkowych bibliotek. Za jego pomocą można rysować scenę wraz ze znajdującymi się na niej aktorami, ale do stworzenia okna, w którym będzie wykonywała się cała symulacja, trzeba użyć zewnętrznej biblioteki. Takich bibliotek jest wiele, a każda posiada swoje mocne i słabe strony. Najczęściej używanymi są GLUT, SDL, WGL oraz GLFW. W niniejszej pracy zastosowana zostanie ta ostatnia, gdyż cechuje ją większa funkcjonalność, jednak kosztem złożoności. Żeby ułatwić trochę pracę z OpenGL, dodana zostanie także matematyczna biblioteka GLM posiadająca m.in. definicje macierzy oraz wektorów oraz biblioteka GLEW ułatwiającą używanie funkcjonalności OpenGL.

Najlepszym sposobem na dodanie wyżej wymienionych bibliotek do projektu jest stworzenie kolejnego folderu o nazwie *OpenGL* wewnątrz istniejącego już *External*. Następnie dodanie do niego folderów pobranych i jeżeli to potrzebne, skompilowanych już bibliotek. Foldery te zostały nazwane odpowiednio *glew*, *glfw* oraz *glm*.

Kolejnym krokiem jest dodanie linków do tych folderów we właściwościach projektu w *Additional Include Directories* (*C/C++/General*) (rys. 4.1.1). Te same ścieżki trzeba także dodać do *Additional Library Directories* w linkerze (zakładka *General*). Do linkera trzeba jeszcze dodać pliki: *opengl32.lib*, *glew32sd.lib*, *glfw3dll.lib* oraz *glu32.lib*, a do głównego folderu projektu ich wersje *.dll*. Można to zrobić w taki sam sposób jak wcześniej w przypadku bibliotek PhysX.



Rys. 4.1.1 Additional Include Directories

Ostatnim krokiem przy dodawaniu bibliotek jest utworzenie pliku nagłówkowego o nazwie *OpenGL.h*, który zawierać powinien wszystkie dodawane biblioteki potrzebne do poprawnego działania programu (listing 4.1.1). Warto dodać także dyrektywę `#define _USE_MATH_DEFINES`, która później pozwoli na dodanie do kodu wartości liczby π (makro `PI`).

Listing 4.1.1 Zawartość pliku nagłówkowego OpenGL.h

```
#ifndef OPENGL_H
#define OPENGL_H
#define _USE_MATH_DEFINES

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cstdlib>
#include <ctime>
#include <iostream>
```

```
#include <sstream>
#include <fstream>

#include <cmath>
#include <math.h>
#include <vector>
#include <array>

#include <glew.h>
#include <glfw3.h>

#include <glm.hpp>
#include <gtc/matrix_transform.hpp>
#include <gtc/type_ptr.hpp>
#endif // !OPENGL_H
```

Dobrym nawykiem jest dodawanie plików nagłówkowych poszczególnych bibliotek tylko tam, gdzie są używane. Jednak podejście takie jak przedstawione wyżej zdecydowanie ułatwia pracę w średniej wielkości projektach.

4.2 Utworzenie okna

Pierwszym krokiem do utworzenia okna aplikacji będzie przygotowanie nowej klasy `Window`. W jej pliku nagłówkowym należy zdefiniować dyrektywy `GLEW_STATIC` oraz `GLFW_STATIC`, które pozwolą na wykorzystywanie wersji statycznych tych bibliotek oraz publiczny obiekt typu `GLFWwindow*`, którego adres będzie przekazywany do innych funkcji jako wskaźnik na okno. Następnie trzeba utworzyć konstruktor z parametrami `width` i `height`, które określą wymiary okna. Na potrzeby tej pracy, okno będzie miało je ustawione na `width = 1024` oraz `height = 768`. Pierwszym działaniem konstruktora powinno być zainicjalizowanie biblioteki GLFW. Można to zrobić jednocześnie walidując czy operacja się udała (listing 4.2.1). W przypadku błędu, wypisanie w konsoli wyniku funkcji `stderr` ułatwi debugowanie zaistniałego problemu.

Listing 4.2.1 Inicjalizacja biblioteki GLFW

```
void Window::initGlfw() {
	if (!glfwInit()) {
		std::cout << stderr << "Failed to initialize GLFW!" << std::endl;
	}
}
```

Kolejnym krokiem będzie przekazanie do biblioteki GLFW sugestii dotyczących konfiguracji, co można zrobić korzystając z funkcji `glfwWindowHint` (listing 4.2.2 pokazuje zbiór takich poleceń). Dla stabilności aplikacji i łatwiejszego zapoznania się z OpenGL wykorzystywana będzie jego wersja 3.3 z trybem rdzennym `CORE_PROFILE` i zachowaną kompatybilnością w przód. Dodatkowo ustawiony zostanie parametr niepozwalający na dynamiczną zmianę rozmiaru okna. Jest to potrzebne, ponieważ w przeciwnym wypadku, podczas dynamicznej zmiany wymiarów okna, rzeczywisty obszar rysowania nie zmieniałby się i znajdujące się na nim obiekty mogłby zostać zasłonięte przez tworzące się w takiej sytuacji pasy tła.

Listing 4.2.2 Konfiguracja GLFW

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
```

Następnie trzeba utworzyć i zainicjować okno (listing 4.2.3). Można to zrobić przy pomocy funkcji `glfwCreateWindow`, do której przekazywane są wymiary i tytuł okna, wskaźnik na docelowy monitor oraz wskaźnik na okno, z którym mają być dzielone zasoby. Jeżeli aplikacja ma działać w trybie pełnoekranowym, koniecznie trzeba podać wskaźnik na monitor, na którym aplikacja ma działać (czwarty argument), w przeciwnym wypadku należy podać wartość `nullptr`, co spowoduje, że aplikacja będzie działała w trybie okna. Jeżeli aplikacja nie ma współdzielić zasobów, w ostatnim parametrze także trzeba podać `nullptr`. Funkcja `glfwCreateWindow` zwraca wartość, która pozwala sprawdzić, czy udało się stworzyć okno i przypisać mu kontekst. Kontekst ten pozwala na wskazanie okna, którego dotyczą kolejne komendy (może ich być wiele).

Listing 4.2.3 Inicjalizacja okna

```
void Window::initWindow(int width, int height) {
	mWindow = glfwCreateWindow(width, height, "OpenGL", nullptr, nullptr);
	if (mWindow == nullptr) {
		std::cout << stderr << "Failed to create GLFW window!" << std::endl;
		glfwTerminate();
	}
	glfwMakeContextCurrent(mWindow);
}
```

Następnym krokiem będzie zainicjalizowanie biblioteki GLEW. Można to zrobić w taki sam sposób jak w przypadku biblioteki GLFW (listing 4.2.4), ale należy pamiętać o wcześniejszym przypisaniu wartości `GL_TRUE` do globalnej zmiennej `glewExperimental`, która jest potrzebna podczas korzystania z funkcjonalności `CORE_PROFILE`.

Listing 4.2.4 Inicjalizacja biblioteki GLEW

```
void Window::initGlew() {
	glewExperimental = GL_TRUE;
	if (glewInit() != GLEW_OK) {
		std::cout << stderr << "Failed to initialize GLEW!" << std::endl;
	}
}
```

Po tym trzeba uruchomić kilka dodatkowych metod, aby okno zachowywało się w pożądany sposób (listing 4.2.5). Pierwszą z nich jest `glfwSetInputMode`, do której przekazana zostanie wartość `GL_TRUE` dla `GLFW_STICKY_KEYS`, aby ułatwić wychwytywanie wciśniętych klawiszy oraz `GLFW_CURSOR_DISABLED` dla `GLFW_CURSOR`, aby wyłączyć kursor i tym samym uniemożliwić wyjechanie nim poza okno. Wyjście z aplikacji będzie możliwe tylko klawiszem *Esc* albo kombinacją *Alt* + *Tab*. Na koniec za pomocą `glEnable` należy włączyć `GL_DEPTH_TEST` oraz `GL_CULL_FACE`. Pierwszy z nich odpowiada za test głębokości. Pomaga on rozróżnić, które z rysowanych elementów są bliżej kamery, a które dalej [22] (opis całego potoku renderowania można znaleźć w oficjalnej dokumentacji OpenGL [23]). Taki test polega na porównywaniu wartości głębokości rysowanego frgamentu z wartością *z-bufora*. Jeżeli z takiego działania wyniknie, że porównywany fragment jest przesłonięty innym, jest on odrzucany. W przeciwnym wypadku zastępuje fragment, który poprzednio znajdował się na bliżej. Drugi z parametrów pozwala uniknąć rysowania wielokątów, które nie pojawią się na ekranie, czyli tzw. usuwanie „tylnych stron" trójkątów. Ustalenie, która strona jest tylnia jest kontrolowane przez funkcję `void glFrontFace(GLenum mode)`, której parametr może przyjmować wartość `GL_CW` lub `GL_CCW`. Pierwsza wartość oznacza, że wierzchołki trójkątów są „nawijane" względem jego środka zgodnie z ruchem wskazówek zegara a drugi, że dzieje się to w sposób przeciwny. Przy większych projektach takie działania pomagają programowi na wykonywanie mniejszej ilości operacji co klatkę. Poprawia to jego optymalizację, a przy mniejszych jest po prostu dobrą praktyką [24].

Listing 4.2.5 Konfiguracja GLFW

```
glfwSetInputMode(mWindow, GLFW_STICKY_KEYS, GL_TRUE);
glfwSetInputMode(mWindow, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
```

Kolejnym krokiem będzie utworzenie instancji klasy `Window` i zapisanie jej wskaźnika do globalnej zmiennej `window` (kod z pliku *Source.cpp*). Podobnie jak obiekt `PhysX`, będzie on w późniejszych etapach potrzebny przy funkcjach zwrotnych i jest to jedyny powód, dla którego nie jest on obiektem lokalnym. W funkcji `main` należy utworzyć główną pętle programu (listing 4.2.6). Warunkiem jej kontynuacji będzie to, że wartość zwracana przez funkcje `glfwWindowShouldClose` równa jest `false`. Jest ona ustawiana na wartość `true` przy przyciśnięciu klawisza *Esc*. Trzeba jeszcze tylko pamiętać o wyczyszczeniu bufora koloru poprzez zapełnieniem go wskazanym kolorem. Na koniec głównej pętli programu uruchamiana jest metoda `glfwPollEvents`, która sprawdza, czy zostały wywołane jakieś *eventy* takie jak ruch myszą czy chociażby wciśnięcie klawiszy, a następnie uaktualnia ona stan okna i wywołuje odpowiednie funkcje zwrotne. Wraz z nią uruchamiana jest jeszcze funkcja `glfwSwapBuffers`, która odświeży okno poprzez wymianę buforów. Jest to konieczne, ponieważ OpenGL wykorzystuje podwójne buforowanie. Jest to konieczne, ponieważ podczas pojedynczego przejścia pętli programu, obraz zapisywany jest do bufora tylnego piksel po pikselu. Wyświetlenie takiego bufora mogłoby spowodować wystąpienie artekfaktów. Dlatego użytkownikowi wyświetlana jest zawartość bufora przedniego, do którego przenoszona jest zawartość bufora tylnego, dopiero po zakończeniu rysowania. Po zakończeniu pętli należy użyć funkcji `glfwTerminate`, aby wyłączyć okno i wyczyścić zasoby zaalokowane na potrzeby GLFW .

Listing 4.2.6 Główna pętla programu

```
while (!glfwWindowShouldClose(window.mWindow)) {
            glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

            if (glfwGetKey(window.mWindow, GLFW_KEY_ESCAPE) == GLFW_PRESS){
                  glfwSetWindowShouldClose(window.mWindow, true);
            }
```

```
            glfwPollEvents();
            glfwSwapBuffers(window.mWindow);
}
glfwTerminate();
```

Program można już uruchomić, a w rezultacie powinno otworzyć się okno z szarym tłem (rys 4.2.1).

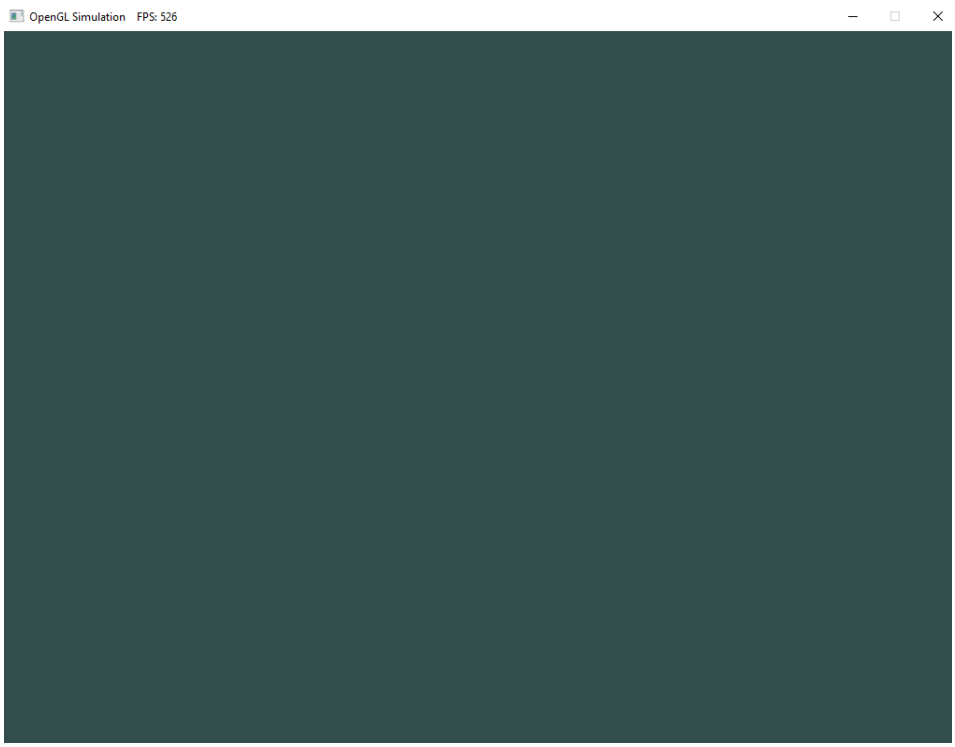
Rys. 4.2.1 Puste okno klasy `window`

Dodatkowym akcentem kończącym konfigurację okna będzie pokazanie aktualnej liczby wyświetlanych klatek na sekundę (*FPS'ów)* na pasku tytułu okna. Zostanie to zrobione w funkcji `showFps`, która będzie uruchamiana w każdym przejściu głównej pętli programu (listing 4.2.7). Wewnątrz niej trzeba utworzyć statyczną zmienna typu `int`, która będzie przechowywać ilość klatek zliczonych podczas jednej sekundy działania programu. Następnie za pomocą `glfwGetTime` trzeba zebrać dwa czasy i zapisać je do dwóch zmiennych typu `double`, z czego jedna z nich musi być statyczna. Kolejnym krokiem jest utworzenie instrukcji warunkowej `if`, której zawartość będzie uruchamiana, gdy różnica zebranych czasów będzie wynosić minimum jedną sekundę. Wewnątrz funkcji uaktualniany jest czas przetrzymywany w statycznej zmiennej oraz wypisywana na pasku okna jest ilość klatek zebranych podczas jednej sekundy działania programu. Należy

pamiętać, aby na koniec wyzerować także `frameCounter`. Rezultat można zobaczyć na rysunku 4.2.2.

Listing 4.2.7 Funkcja zliczająca *FPSy*

```
void Window::showFps() {
	static int frameCounter = 0;
	static double previousTime = glfwGetTime();

	double currentTime = glfwGetTime();
	float deltaTime = static_cast<float>(currentTime) -
	    static_cast<float>(previousTime);
	frameCounter++;

	if (deltaTime >= 1.0f) {
		previousTime = currentTime;

		std::stringstream windowTitle;
		windowTitle << "OpenGL Simulation    FPS: " << frameCounter;
		glfwSetWindowTitle(mWindow, windowTitle.str().c_str());

		frameCounter = 0;
	}
}
```



Rys. 4.2.2 Okno klasy `window` z dodanym licznikiem FPS w tytule

4.3 Rysowanie obiektów

Okno aplikacji jest już stworzone, więc nadszedł czas, aby wyświetlić na nim jakiś obiekt. Niektóre *frameworki* jak np. Unity pozwalają na tworzenie prostych obiektów w prosty sposób. Wybierając OpenGL trzeba liczyć się, że wszystko trzeba zrobić samemu od podstaw. Daje to jednak o wiele większą kontrolę nad tym co właściwie jest rysowane oraz w jaki sposób.

Na potrzeby tej pracy, rysowany będzie stół bilardowy z kulami. Jednakże, aby móc zweryfikować, czy tak skomplikowany obiekt jest renderowany poprawnie, najpierw trzeba przejść przez podstawy. Trzeba wobec tego zacząc od narysowania białego sześcianu.

Pierwszym krokiem będzie stworzenie klasy `Actor`, która tymczasowo będzie zajmować się rysowaniem sześcianu. Warto zaznaczyć, że obiekty w OpenGL rysowane są na podstawie podanych przez użytkownika zestawów trzech punktów składających się ze współrzędnych *x*, *y* oraz *z*. W dalszej części pracy takie zbiory wierzchołków będą nazywane werteksami, a w tym przypadku są to werteksy pozycji. Środek okna jest domyślnie w położeniu równym (0, 0, 0). Łatwo więc wyliczyć położenie poszczególnych wierzchołków sześcianu. Jako że do rysowania użyta zostanie metoda `glDrawArrays` z parametrem `GL_TRIANGLES`, werteksy trzeba podawać tak jakby sześcian był złożony z trójkątów, więc obiekt ten będzie się składać z 36 werteksów (6 na każdą ścianę) (listing 4.3.3). Alternatywną i zdecydowanie bardziej optymalną metodą rysowania byłoby użycie `GL_TRIANGLE_STRIP` zamiast zwykłych trójkątów. Korzystając z takiego rozwiązania trzeba by dodatkowo przygotować bufor przechowujący indeksy i dopiero na ich podstawie rysować obiekt. Jednakże dla tak małego projektu, możliwa do zaoszczędzenia pamięć nie jest wystarczająco istotna.

Zanim jednak zadeklarowana zostanie taka tablica, należy utworzyć 3 prywatne obiekty typu `GLuint`, które będą reprezentowały bufory. Pierwszym z nich jest VAO (ang. *Vertex Array Object*) czyli bufor przechowujący wszystkie następujące po nim wywołania innych buforów, aż do momentu jego wyłączenia [25]. Kolejnym będzie VBO (ang. *Vertex Buffer Object*) czyli bufor przechowujący położenie werteksów, a ostatnim

bufor przechowujący kolor konkretnych trójkątów, który jest tak naprawdę kolejnym buforem VBO stworzonym do przechowywania danych innego typu.

Warto stworzyć dwie prywatne metody `prepareVertices` oraz `prepareColor`, które można chwilowo zostawić puste. Następnie w konstruktorze trzeba wygenerować VAO, uruchomić obie funkcje, które wypełnią odpowiednie bufory danymi, a następnie wyłączyć VAO, który aż do momentu rysowania obiektu nie będzie potrzebny (listing 4.3.1). Należy pamiętać, aby w destruktorze klasy zwolnić wszystkie niepotrzebne już bufory (listing 4.3.2).

Listing 4.3.1 Konstruktor klasy `Actor`

```
Actor::Actor() {
	glGenVertexArrays(1, &mVao);
	glBindVertexArray(mVao);

	prepareVertices();
	prepareColor();

	glBindVertexArray(0);
}
```

Listing 4.3.2 Destruktor klasy `Actor`

```
Actor::~Actor() {
	glDeleteVertexArrays(1, &mVao);
	glDeleteBuffers(1, &mVbo);
	glDeleteBuffers(1, &mCbo);
}
```

Kolejnym krokiem będzie uzupełnienie pustych funkcji wypełniających bufory. W przypadku używania tak małej rozdzielczości jak w tym programie, należy pamiętać, że werteksy trzeba dostosować tak aby mieściły się na ekranie. W późniejszym czasie będzie można regulować wielkość obiektów, ale będzie to omawiane dopiero w rozdziale 4.5. Na razie warto posługiwać się wartościami `0.5f` zamiast `1.0f`. Na początek, trzeba wygenerować bufor położenia, zdefiniować wszystkie werteksy sześcianu, a potem za pomocą metody `glBindBuffer` zaznaczyć, że w tym momencie wypełniany będzie wskazany bufor. Następnie przy pomocy `glBufferData` należy wskazać rozmiar tablicy werteksów oraz sama tablicę. Teraz pozostało jedynie utworzyć wskaźnik atrybutu

o numerze 0 wskazujący na pierwszy element tablicy werteksów (listing 4.3.3). Dokładnie takie same działania należy wykonać przy wypełnianiu funkcji `prepareColor`, a jedyną różnicą będzie sama zawartość tablicy kolorów oraz numer wskaźnika atrybutu (listing 4.3.4).

Listing 4.3.3 Funkcja przygotowująca bufory werteksów

```
void Actor::prepareVertices() {
	glGenBuffers(1, &mVbo);

	GLfloat vertices[] = {
		-0.5f,-0.5f,-0.5f,
		-0.5f,-0.5f, 0.5f,
		-0.5f, 0.5f, 0.5f,
		 0.5f, 0.5f,-0.5f,
		-0.5f,-0.5f,-0.5f,
		-0.5f, 0.5f,-0.5f,

		 0.5f,-0.5f, 0.5f,
		-0.5f,-0.5f,-0.5f,
		 0.5f,-0.5f,-0.5f,
		 0.5f, 0.5f,-0.5f,
		 0.5f,-0.5f,-0.5f,
		-0.5f,-0.5f,-0.5f,

		-0.5f,-0.5f,-0.5f,
		-0.5f, 0.5f, 0.5f,
		-0.5f, 0.5f,-0.5f,
		 0.5f,-0.5f, 0.5f,
		-0.5f,-0.5f, 0.5f,
		-0.5f,-0.5f,-0.5f,

		-0.5f, 0.5f, 0.5f,
		-0.5f,-0.5f, 0.5f,
		 0.5f,-0.5f, 0.5f,
		 0.5f, 0.5f, 0.5f,
		 0.5f,-0.5f,-0.5f,
		 0.5f, 0.5f,-0.5f,

		 0.5f,-0.5f,-0.5f,
		 0.5f, 0.5f, 0.5f,
		 0.5f,-0.5f, 0.5f,
		 0.5f, 0.5f, 0.5f,
		 0.5f, 0.5f,-0.5f,
```

```
            -0.5f, 0.5f,-0.5f,

             0.5f, 0.5f, 0.5f,
            -0.5f, 0.5f,-0.5f,
            -0.5f, 0.5f, 0.5f,
             0.5f, 0.5f, 0.5f,
            -0.5f, 0.5f, 0.5f,
             0.5f,-0.5f, 0.5f
      };

      glBindBuffer(GL_ARRAY_BUFFER, mVbo);
      glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);
      glEnableVertexAttribArray(0);
}
```

Listing 4.3.4 Funkcja przygotowująca bufor koloru

```
void Actor::prepareColor() {
      glGenBuffers(1, &mCbo);

      GLfloat color[] = {
            0.583f,  0.771f,  0.014f,
            0.609f,  0.115f,  0.436f,
            0.327f,  0.483f,  0.844f,
            0.822f,  0.569f,  0.201f,
            0.435f,  0.602f,  0.223f,
            0.310f,  0.747f,  0.185f,
            0.597f,  0.770f,  0.761f,
            0.559f,  0.436f,  0.730f,
            0.359f,  0.583f,  0.152f,
            0.483f,  0.596f,  0.789f,
            0.559f,  0.861f,  0.639f,
            0.195f,  0.548f,  0.859f,
            0.014f,  0.184f,  0.576f,
            0.771f,  0.328f,  0.970f,
            0.406f,  0.615f,  0.116f,
            0.676f,  0.977f,  0.133f,
            0.971f,  0.572f,  0.833f,
            0.140f,  0.616f,  0.489f,
            0.997f,  0.513f,  0.064f,
            0.945f,  0.719f,  0.592f,
            0.543f,  0.021f,  0.978f,
            0.279f,  0.317f,  0.505f,
            0.167f,  0.620f,  0.077f,
            0.347f,  0.857f,  0.137f,
```

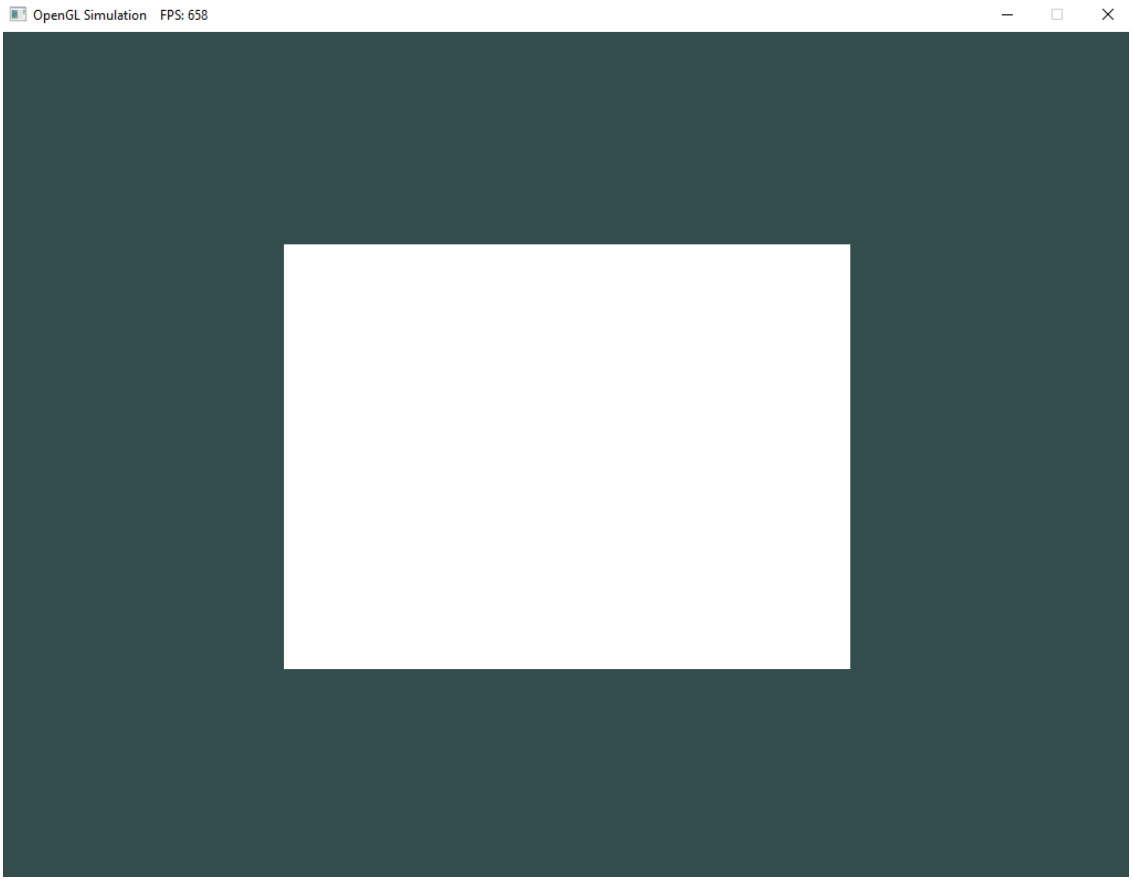
```
            0.055f,  0.953f,  0.042f,
            0.714f,  0.505f,  0.345f,
            0.783f,  0.290f,  0.734f,
            0.722f,  0.645f,  0.174f,
            0.302f,  0.455f,  0.848f,
            0.225f,  0.587f,  0.040f,
            0.517f,  0.713f,  0.338f,
            0.053f,  0.959f,  0.120f,
            0.393f,  0.621f,  0.362f,
            0.673f,  0.211f,  0.457f,
            0.820f,  0.883f,  0.371f,
            0.982f,  0.099f,  0.879f
        };

        glBindBuffer(GL_ARRAY_BUFFER, mCbo);
        glBufferData(GL_ARRAY_BUFFER, sizeof(color), color, GL_STATIC_DRAW);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);
        glEnableVertexAttribArray(1);
}
```

Ostatnim krokiem na tym etapie jest zdefiniowanie metody `draw`, która włącza wygenerowany w konstruktorze obiekt VAO i na jego podstawie rysuje sześcian korzystając z metody `glDrawArrays`, do której przekazywany jest sposób rysowania (w tym przypadku GL_TRIANGLES) oraz wielkość tablicy werteksów (listing 4.3.5).

Listing 4.3.5 Funkcja `draw` klasy `Actor`

```
void Actor::draw() {
        glBindVertexArray(mVao);
        glDrawArrays(GL_TRIANGLES, 0, 36);
        glBindVertexArray(0);
}
```

Pozostało w funkcji `main` utworzyć obiekt klasy `Actor` i we wcześniej przygotowanej pętli wywołać jego metodę `draw`. Rezultatem powinien być biały prostokąt (rys. 4.3.1), który w rzeczywistości jest kolorowym sześcianem, ale jest rysowany w ten właśnie sposób, ponieważ w projekcie nie ma jeszcze dodanych programów cieniowania, czyli tzw. shaderów. Shadery odpowiadają m.in. za nadawanie koloru lub tekstury obiektom. Są one napisane w podobnym do C, języku GLSL (*OpenGL Shading Language*) zawierającym funkcje ułatwiające operacje na wektorach i macierzach [26].



Rys. 4.3.1 Biały sześcian narysowany przy pomocy OpenGL

4.4 Podstawowy shading

Jak zostało wspomniane wyżej, aby nadać jakiś kolor lub teksturę obiektom trzeba utworzyć shadery. Shadery zawsze zaczynają się od deklaracji wersji i następującej po niej listy zmiennych wchodzących i wychodzących. Wszystkie działania na zmiennych odbywają się w funkcji `main` programu shadera [27]. Do poprawnego działania potrzebne są minimum dwa shadery: vertex shdaer i fragment shader. Shaderów może być więcej, ale na potrzeby tej pracy nie są potrzebne więc zostaną pominięte.

Pierwszą czynnością potrzebną do tego, aby sześcian narysować z kolorem zadeklarowanym w jego werteksach, będzie stworzenie shadera werteksów (listing 4.4.1) oraz shadera fragmentów (4.4.2). Dane z głównego programu będą wysyłane najpierw do tego pierwszego, w nim przetwarzane i przesyłane do drugiego. Shadery z listingów 4.4.1 i 4.4.2 są bardzo proste i jedyne co robią to ustalają kolor, jednak w późniejszej części pracy zostaną rozbudowane.

Listing 4.4.1 Vertex shader

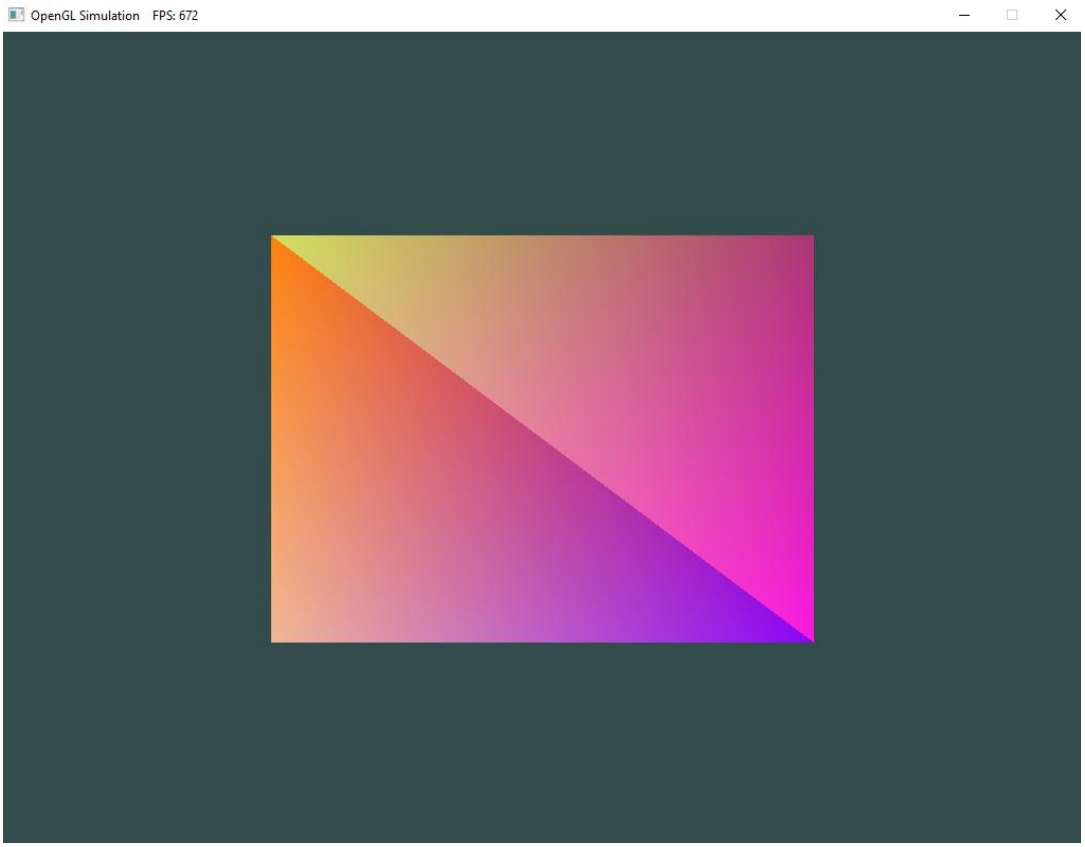
```
#version 330 core
layout(location = 0) in vec3 vertices;
layout(location = 1) in vec3 color;

out vec4 fragmentColor;

void main(){
	gl_Position = vec4(vertices, 1.0);
	fragmentColor = vec4(color, 1.0);
}
```

Listing 4.4.2 Fragment shader

```
#version 330 core
in vec4 fragmentColor;
out vec4 color;

void main() {
    color = fragmentColor;
}
```

Kiedy shadery są już utworzone, trzeba je jeszcze skompilować, bowiem w OpenGL kompilacja jest wykonywana w trakcie działania programu. W tym celu przygotowana została klasa Shader, która posiada publiczny obiekt `mShadersId` typu `GLuint` reprezentujący program Shaderów oraz funkcje zaczytujące i kompilujące poprzednio utworzone shadery, a ostatecznie łączące je w ramach `mShadersId`. Taka klasa jest bardzo ogólna, więc na potrzeby pracy została napisana na podstawie wielu jej przykładów, które można znaleźć w internecie [28] [29] [30]. Z tego powodu w pracy nie będzie ona omawiana bardziej szczegółowo. Należy tylko pamiętać, aby w pliku `Source.cpp` dodać inicjalizację tej klasy z podanymi nazwami shadera werteksów oraz fragmentów, a na koniec, w głównej pętli programu uruchomić funkcję `glUseProgram` z parametrem `mShadersId`.

Po uruchomieniu programu powinien być teraz widoczny wielokolorowy prostokąt składający się z dwóch trójkątów (rys. 4.4.1).



Rys. 4.4.1 Sześcian po wczytaniu *Shaderów*

4.5 System współrzędnych

Nadszedł czas, żeby zweryfikować to, czy stworzony obiekt, rzeczywiście jest sześcianem. Dotychczas widoczny był przecież tylko z jednej strony. Będzie do tego potrzebna zmiana położenia kamery względem obrotu sześcianu.

Na początek trzeba utworzyć nową klasę `Camera`, z 3 publicznymi polami typu `glm::mat4` (macierze 4x4) o nazwach: `mModel`, `mView` i `mProjection`. Przy inicjalizacji każdej z nich należy pamiętać o ustawieniu w pierwszej linii kodu macierzy jednostkowych za pomocą `glm::mat4(1.0f)`, gdyż wszystkie wykonywane na nich operacje składają się głównie z mnożeń. Nieustawienie początkowej wartości mogłoby spowodować wyzerowanie wszystkich obliczeń.

Macierz `mModel` będzie odpowiadać za translacje, skalowania oraz rotacje poszczególnych obiektów. W jego funkcji inicjalizującej (listing 4.5.1) warto przy deklaracji w pliku `Camera.h` ustawić domyślne wartości równe `0.0f`. Dzięki temu

możliwe będzie wykonywanie jedynie części operacji zamiast wszystkich. Ważna jest także kolejność transformacji macierzy. Najpierw trzeba zacząć od zmiany pozycji, następnie obrócić, a dopiero na koniec skalować. Poniżej można zobaczyć, w jakich elemtnach macierzy jest przechowywana oraz jak przebiega transformacja macierzy w przypadku translacji (1), skalowaniu (2) i rotacji względem osi *x* (3), *y* (4) oraz *z* (5) [31]. Macierz `mModel` będzie zmieniać się dla każdego rysowanego obiektu, dlatego należy pamiętać, aby dostosować ją przed każdą funkcją rysującą.

$$\begin{pmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + X * 1 \\ y + Y * 1 \\ z + Z * 1 \\ 1 \end{pmatrix} \quad (1)$$

$$\begin{pmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} SX * x \\ SY * y \\ SZ * z \\ 1 \end{pmatrix} \quad (2)$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ cos\theta * y - sin\theta * z \\ sin\theta * y + cos\theta * z \\ 1 \end{pmatrix} \quad (3)$$

$$\begin{pmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} cos\theta * x + sin\theta * z \\ y \\ -sin\theta * x + cos\theta * z \\ 1 \end{pmatrix} \quad (4)$$

$$\begin{pmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} cos\theta * x - sin\theta * y \\ sin\theta * x - cos\theta * y \\ z \\ 1 \end{pmatrix} \quad (5)$$

Listing 4.5.1 Inicjalizacja macierzy modelu

```
void Camera::initModel(glm::vec3 position, glm::vec3 rotation, GLfloat angle,
	glm::vec3 scale) {
	mModel = glm::mat4(1.0f);

	if (position != glm::vec3(0.0f, 0.0f, 0.0f)) {
		mModel = glm::translate(mModel, position);
	}
```

```
	if (rotation != glm::vec3(0.0f, 0.0f, 0.0f)) {
		mModel = glm::rotate(mModel, glm::radians(angle), rotation);
	}
	if(scale != glm::vec3(0.0f, 0.0f, 0.0f)) {
		mModel = glm::scale(mModel, scale);
	}
}
```

Macierz `mView` wskazuje na jakie miejsce sceny skierowana jest kamera, a tym samym co jest możliwe do zobaczenia w oknie aplikacji. OpenGL nie posiada żadnej formalnej funkcjonalności kamery, a poruszanie się jest imitowane poprzez przesuwanie całej sceny w pożądanym kierunku za pomocą macierzy `mView`. Trzeba pamiętać, że OpenGL korzysta z prawoskrętnego układu współrzędnych, więc dodatnia część osi *x* znajduje się po stronie prawej, osi *y* u góry, natomiast oś *z* skierowana jest w kierunku użytkownika [32]. Żeby inicjalizacja `mView` (listing 4.5.2) zadziałała poprawnie, trzeba zdefiniować 3 dodatkowe prywatne zmienne: `mPosition`, `mDirection` i `mUp`. Pierwsza zawiera wektor położenia kamery na scenie, druga kierunek patrzenia, a ostatnia wskazuje, gdzie jest „góra" kamery. Macierz widoku tak naprawdę odpowiada wynikowi funkcji `glm::lookAt`, która tak jak jej nazwa wskazuje: ustawia macierz tak, aby „patrzyła" na wskazane miejsce. Z każdym ruchem kamery na scenie, macierz `mView` będzie się zmieniać, dlatego trzeba aktualizować ją w każdym przejściu głównej pętli programu.

Listing 4.5.2 Inicjalizacja macierzy widoku

```
void Camera::initView() {
	mView = glm::mat4(1.0f);
	glm::vec3 target = mPosition + mDirection;
	mView = glm::lookAt(mPosition, target, mUp);
}
```

Macierz `mProjection` tworzona jest metodą `glm::perspective`, która jako argumenty pobiera: FoV (ang. *Field of View*), *near plane* oraz *far plane*. Przed wywołaniem funkcji przygotowującej macierz projekcji, w liście inicjalizacyjnej konstruktora trzeba zdefiniować zmienne przechowujące przesyłane do metody `glm::perspective` wartości. Na potrzeby tej pracy `fov = 45.0f`, `nearPlane =`

`0.1f`, a `farPlane = 100.0f`. Inicjalizację macierzy projekcji tak jak macierzy widoku wystarczy przeprowadzić tylko raz na przejście głównej pętli programu.

Listing 4.5.3 Inicjalizacja macierzy projekcji

```
void Camera::initProjection() {
	mProjection = glm::mat4(1.0f);
	mProjection = glm::perspective(glm::radians(mFoV), mAspectRatio,
	mNearPlane, mFarPlane);
}
```

Kiedy wyżej opisane macierze są przygotowane trzeba jeszcze uaktualnić shadery tak aby mogły z nich korzystać (listing 4.5.4). Można to zrobić poprzez zastosowanie modyfikatorów zmiennych w shaderach, czyli tzw. uniformów. Mogą one zostać przekazane przez użytkownika z głównego programu do Shaderów. Aby zastosować uniformy, należy znaleźć ich lokację w shaderach za pomocą funkcji `glGetUniformLocation` do której przekazać trzeba identyfikator shadera `shadersId` oraz nazwę danego uniformu z shadera werteksów. Te funkcje wywoływane są w programie raz, najlepiej zaraz po stworzeniu obiektu klasy `Shader` (listing 4.5.5). Obliczając `gl_Position` wewnątrz shadera werteksów należy pamiętać, aby obliczyć iloczyn macierzy modelu, widoku i rzutowania przez pozycję przekazaną z buforów w dokładnie takiej kolejności (C++ mnoży od prawej do lewej).

Listing 4.5.4 Vertex shader z wykorzystaniem macierzy transformacji

```
#version 330 core
layout(location = 0) in vec3 vertices;
layout(location = 1) in vec3 color;

out vec4 fragmentColor;

uniform mat4 projection;
uniform mat4 model;
uniform mat4 view;

void main() {
    gl_Position = projection * view * model * vec4(vertices, 1.0);
    fragmentColor = vec4(color, 1.0);
}
```

Listing 4.5.5 Ustawienie lokacji Uniformów

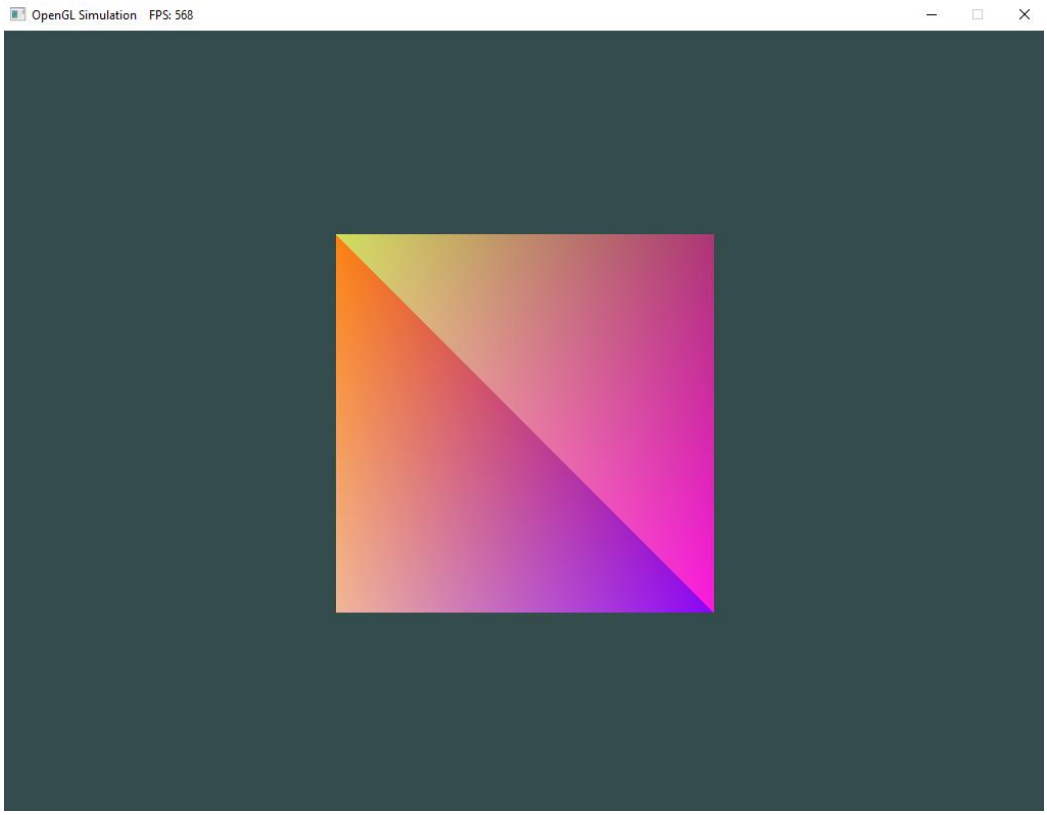
```
GLint modelLocation = glGetUniformLocation(shader.mShadersId, "model");
```

```
GLint viewLocation = glGetUniformLocation(shader.mShadersId, "view");
GLint projectionLocation = glGetUniformLocation(shader.mShadersId, "projection");
```

Teraz pozostało dodać inicjalizację macierzy modelu, widoku i projekcji do głównej pętli programu, a zaraz po nich przy pomocy funkcji `glUniformMatrix4fv` przekazać ich zawartość do Shaderów (listing 4.5.6). Po uruchomieniu programu zamiast prostokąta pojawi się kwadrat, można więc wywnioskować, że wyświetlanie jest już poprawnie zmodyfikowane (rys 4.5.1).

Listing 4.5.6 Inicjalizacja macierzy i przekazanie ich do shadera

```
camera.initModel();
camera.initView();
camera.initProjection();

glUniformMatrix4fv(modelLocation, 1, GL_FALSE, &camera.mModel[0][0]);
glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &camera.mView[0][0]);
glUniformMatrix4fv(projetionLocation, 1, GL_FALSE, &camera.mProjection[0][0]);

actor.draw();
```

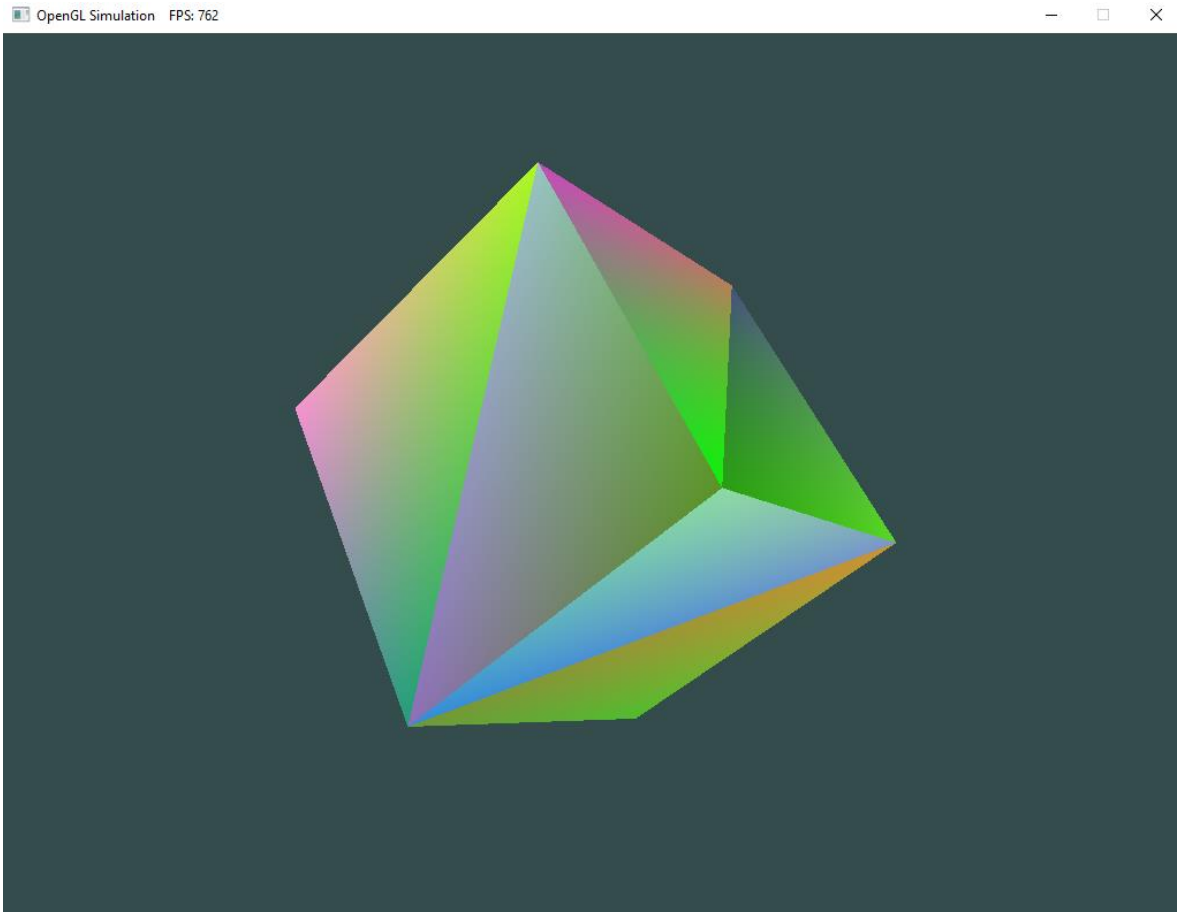


Rys. 4.5.1 Wielokolorowy kwadrat po przekazaniu wszystkich macierzy do shadera

Jednak, aby mieć stuprocentową pewność, że stworzony obiekt jest bryłą 3D, należy w macierzy modelu uwzględnić obrót (listing 4.5.7). Wektor rotacji informuje o kierunku osi obrotu aktora. Należy wskazać również kąt obrotu. Po dodaniu paru linijek kodu widocznych na listingu 4.5.7, po uruchomieniu programu można zaobserwować powoli obracający się sześcian (rys. 4.5.2). Aby przyspieszyć jego obracanie się wystarczy zwiększyć wartość przez jaką przemnażane jest `glfwGetTime`, czyli wielkość kąta obrotu w każdej iteracji.

Listing 4.5.7 Inicjalizacja macierzy z dodaną rotacją

```
glm::vec3 position = glm::vec3(0.0f);
glm::vec3 rotation = glm::vec3(1.0f, 0.5f, 0.0f);
GLfloat angle = glfwGetTime() * 20.0f);

camera.initModel(position, rotation, angle);
```



Rys. 4.5.2 Obracający się sześcian

4.6 Kontrola położenia kamery

Położenie kamery może być kontrolowane przy pomocy klawiatury. Do implementacji tego sposobu kontroli wystarczy zaledwie jedna funkcja, która będzie korzystać z dodatkowej prywatnej zmiennej `mMoveSpeed`. Będzie ona reprezentować prędkość poruszania się kamery na scenie. Jej domyślna wartość będzie wynosić `0.02f`. W klasie `Camera` będzie zdefiniowany tzw. „*setter"* (listing 4.6.1), aby w razie potrzeby móc zmodyfikować tę wartość. Za obsługę klawiszy odpowiedzialna będzie funkcja `processKeys` (listing 4.6.2) do której przekazywany będzie przez parametr wskaźnik do okna typu `GLFWwindow*`. Wewnątrz funkcji, instrukcjami warunkowymi `if` należy obsłużyć klawisze: *W, S, A, D, Space oraz Shift*. Można tutaj przenieść także instrukcję obsługującą klawisz *esc* z głównej pętli programu. Dla każdego klawisza należy odpowiednio zmodyfikować zmienną `mPosition`. Jeżeli kamera ma poruszać się wzdłuż osi *x* w kierunku dodatnim, należy odjąć znormalizowany wynik mnożenia zmiennej odpowiadającej za kierunek ze zmienną odpowiadającą za „górę" sceny, a następnie przemnożyć otrzymaną wartość przez wartość pola `mMoveSpeed`. Jeżeli kamera ma się poruszać w kierunku ujemnym, wynik należy odjąć. Dla osi *y* oraz osi *z* wystarczy dodać lub odjąć odpowiednio zmienne `mUp` i `mDirection` przemnożone przez `mMoveSpeed`. Pozostaje wywołać funkcję `processKeys` pod koniec pętli głównej programu i po jego uruchomieniu można przetestować poruszanie kamerą.

Listing 4.6.1 Funkcja ustawiająca prędkość poruszania się

```
void Camera::setMoveSpeed(GLfloat moveSpeed) {
	mMoveSpeed = moveSpeed;
}
```

Listing 4.6.2 Funkcja processKeys

```
void Camera::processKeys(GLFWwindow* window) {
	if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
		glfwSetWindowShouldClose(window, true);
	}
	if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
		mPosition += mDirection * mMoveSpeed;
	}
	if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
		mPosition -= mDirection * mMoveSpeed;
	}
```

```
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS) {
                mPosition -= glm::normalize(glm::cross(mDirection, mUp)) *
                        mMoveSpeed;
        }
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS) {
                mPosition += glm::normalize(glm::cross(mDirection, mUp)) *
                        mMoveSpeed;
        }
        if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS) {
                mPosition += mUp * mMoveSpeed;
        }
        if (glfwGetKey(window, GLFW_KEY_LEFT_SHIFT) == GLFW_PRESS) {
                mPosition -= mUp * mMoveSpeed;
        }
}
```

Kolejną częścią klasy `Camera` jest rozglądanie się po scenie kursorem myszki z miejsca, w którym znajduje się użytkownik. Potrzebna jest do tego funkcja `processMouse` przyjmująca jako argumenty pozycję kursora na oknie w osi *x* i *y* oraz czułość z jaką kamera ma się poruszać (listing 4.6.3). Tej ostatniej warto przypisać domyślną czułość równą `0.1f`. W pliku nagłówkowym klasy `Camera` trzeba także zdefiniować kolejne prywatne pola: `mFirstMouseMove`, `mLastXPos`, `mLastYpos`, `mYaw` oraz `mPitch`. Pierwsze z nich jest typu `bool` i jest potrzebne do tego, aby ustawić początkowe wartości położenia kursora w oknie. Kolejne dwie są typu `GLfloat` i informują o tym, w jakim miejscu w oknie kursor znajdował się poprzednio. Dzięki temu możliwe będzie wyznaczenie przesunięcia kursora myszy. Ostatnie dwie także są typu `GLfloat` i reprezentują kąty Eulera. Opisują one obrót punktu wokół początku układu współrzędnych. Należy pamiętać, aby każdej z utworzonych zmiennych przypisać wartość w liście inicjacyjnej konstruktora. Dla `mFirstMouseMove` będzie to wartość `true`, dla `mPitch` będzie to `-90.0f`, aby mieć pewność, że kamera będzie na początku zwrócona w stronę ujemnej półosi *z*. Reszta zmiennych przyjmie wartość `0.0f`.

Funkcja `processMouse` (listing 4.6.3) najpierw sprawdza czy wcześniej został już wykonany jakiś ruch myszą. Jeżeli tak się nie stało, ustala wartości zmiennych `mLastXPos` i `mLastYPos` przechowujących współrzędne pozycji kursora i zmienia `mFirstMouseMove` na `false`. Jeżeli nie jest to pierwszy ruch myszą, funkcja oblicza różnicę pomiędzy obecnym, a poprzednim położeniem kursora oraz przemnaża ją przez

czułość, z jaką ruch kamery ma się odbyć. Następnie uaktualnia zmienne `mYaw` i `mPitch` odpowiedzialne za przechowywanie kątów Eulera oraz poprzedniego położenia kursora. Na koniec oblicza wektor kierunku i przypisuje go zmiennej `mDirection` [32].

Listing 4.6.3 Funkcja processMouse

```
void Camera::processMouse(GLdouble xPos, GLdouble yPos, GLfloat sensitivity) {
	if (mFirstMouseMove) {
		mLastXPos = static_cast<GLfloat>(xPos);
		mLastYPos = static_cast<GLfloat>(yPos);
		mFirstMouseMove = false;
	}
	else {
		GLfloat xOffset = static_cast<GLfloat>(xPos) - mLastXPos;
		GLfloat yOffset = mLastYPos - static_cast<GLfloat>(yPos);

		xOffset *= sensitivity;
		yOffset *= sensitivity;

		mYaw += xOffset;
		mPitch += yOffset;

		mLastXPos = static_cast<GLfloat>(xPos);
		mLastYPos = static_cast<GLfloat>(yPos);
	}

	mDirection = glm::normalize(glm::vec3(
		cos(glm::radians(mYaw)) * cos(glm::radians(mPitch)),
		sin(glm::radians(mPitch)),
		sin(glm::radians(mYaw)) * cos(glm::radians(mPitch))
	));
}
```

Wywołując tę funkcję, trzeba przekazać do niej położenie kursora myszy. Aby je zdobyć, należy zastosować funkcję zwrotną (ang. *callback*) obsługującą zdarzenia związane z poruszaniem myszką (listing 4.6.4). Funkcja taka posiada zawsze taki sam „prototyp" zmienić można jedynie nazwę. Dobrym nawykiem jest trzymanie wszystkich definicji w jednym miejscu, a implementacji w innym. W tej pracy definicje funkcji używanych w pliku *Source.cpp* znajdują się nad globalnymi zmiennymi, natomiast ich implementacje dopiero pod funkcją `main`. Żeby stworzone metody zadziałały, trzeba jeszcze uaktywnić przygotowany przed chwilą *callback*. Robi się to przy użyciu funkcji

`glfwSetCursorPosCallback`, do której przekazywany jest wskaźnik do okna z pola `mWindow` oraz funkcja zwrotna `mouseMovementCallback` widoczna na listingu 4.6.5. Po uruchomieniu programu kamera powinna reagować na ruchy myszką.

Listing 4.6.4 Definicja i implementacja funkcji zwrotnej odpowiedzialnej za poruszanie kursorem

```
static void mouseMovementCallback(GLFWwindow* window, GLdouble xpos, GLdouble
                                  ypos);

static void mouseMovementCallback(GLFWwindow* window, GLdouble xPos, GLdouble
                                  yPos) {
	camera.processMouse(xPos, yPos);
}
```

Listing 4.6.5 Uaktywnienie funkcji zwrotnej obsługującej ruchy myszką

```
glfwSetCursorPosCallback(window.mWindow, mouseMovementCallback);
```

Główne funkcje kamery są już zaimplementowane, ale nic nie stoi na przeszkodzie, aby dodać jeszcze możliwość oddalania i przybliżania sceny za pomocą rolki myszy. Można to zrobić manipulując zmienną `mFov`, która odpowiada za kąt widzenia kamery przy pomocy funkcji `processScroll` (listing 4.6.6). Funkcja ta korzystając z argumentu `offset` zwiększa lub zmniejsza wartość zmiennej `mFov`, co przypomina przybliżanie lub oddalanie się. Funkcja ta wykorzystuje prywatne pola reprezentujące maksymalną i minimalną wartość *FOV*. Dodatkowe instrukcje warunkowe są potrzebne na wypadek, gdyby zwiększenie lub zmniejszenie wartości pola `mFov` miało przekroczyć jego minimalną lub maksymalną wartość. Na potrzeby tej pracy `mMinFov` wynosi `1.0f`, a `mMaxFov` wynosi `-90.0f`.

Listing 4.6.6 Funkcja processScroll

```
void Camera::processScroll(GLdouble offset) {
	if (mFoV >= mMinFov && mFoV <= mMaxFov) {
		mFoV -= static_cast<GLfloat>(offset);
	}
	else if (mFoV <= mMinFov) {
		mFoV = mMinFov;
	}
```

```
        else if (mFoV >= mMaxFov) {
                mFoV = mMaxFov;
        }
}
```

Aby uzyskać arugment `offset` mówiący oo zmianie pozycji rolki myszy, przysyłany do funkcji `processScroll` – należy użyć funkcji zwrotnej, obsługującej ruchy rolki myszy (listing 4.6.7). Można to zrobić podobnie jak w przypadku zmiany pozycji kursora myszy. Do uaktywnienia nowej funkcji zwrotnej należy użyć funkcji `glfwSetScrollCallback` (listing 4.6.8).

Listing 4.6.7 Definicja i implementacja scroll callback'u

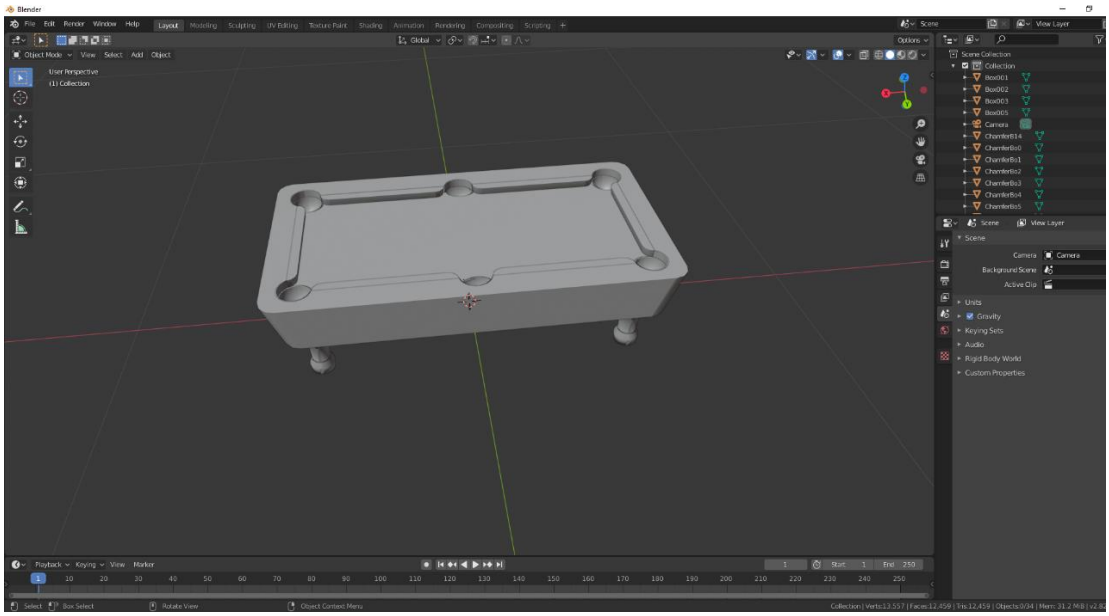
```
static void scrollCallback(GLFWwindow* window,
                           GLdouble xOffset, GLdouble yOffset);

static void scrollCallback(GLFWwindow* window,
                           GLdouble xOffset, GLdouble yOffset) {
        camera.processScroll(yOffset);
}
```

Listing 4.6.8 Użycie scroll callback'u w funkcji main

```
glfwSetScrollCallback(window.mWindow, scrollCallback);
```

4.7 Odczyt obiektu 3D z pliku w formacie wavefront

Kolejnym krokiem będzie umieszczenie na scenie trójwymiarowego stołu bilardowego. Pierwszym krokiem w tym kierunku jest utworzenie projektu stołu w programie do kreacji obiektów 3D. Takim programem jest np. *Blender* (rys. 4.7.1). Jest on dobrą opcją, ponieważ posiada wiele ciekawych i dosyć prostych narzędzi, jest darmowy i stale aktualizowany. Kiedy projekt obiektu jest już utworzony, trzeba go wyeksportować z wybranego programu do formatu *wavefront* (pliki o rozszerzeniu *.obj*). W przypadku korzystania z programu *Blender*, podczas eksportowania obiektu należy pamiętać o zaznaczeiu opcji *include UVs, write Normals, writeMaterials* oraz *Triangulate Faces*.



Rys. 4.7.1 Przykładowy stół bilardowy w programie *Blender*

Teraz pojawiają się dwie możliwości. Można wykorzystać istniejącą zewnętrzną bibliotekę do wczytania takiego pliku lub własnoręcznie napisać klasę, która będzie się tym zajmować. Aby mieć pewność, że obiekt został poprawnie wczytany, na potrzeby tej pracy, przygotowana została klasa `ObjectLoader`.

Żeby móc napisać taką klasę, trzeba zrozumieć co oznaczają konkretne linie w pliku *wavefront* (listing 4.7.1). Mogą się one zaczynać od różnych znaków, które oznaczają:

- # – komentarze,
- v – współrzędne położenia werteksów
- vt – współrzędne tekstur werteksu
- vn – normalną werteksu
- f – listy werteksów tworzących powierzchnie wielokątów (ang. *faces*)

Plik *.obj* posiada jeszcze kilka innych rodzajów linii, ale nie będą one potrzebne w tej pracy.

Listing 4.7.1 Przykładowe linie z pliku typu *.obj*

```
# Blender v2.74 (sub 0) OBJ File: ''
v  1.000000 1.000000 -1.000000
vt 0.748573 0.750412
vn 0.000000 0.000000 -1.000000
f  5/1/1 1/2/1 4/3/1
```

Teraz można przejść do stworzenia trzech prywatnych pól klasy `ObjectLoader` służących do przechowywania współrzędnych. Są one typu `vector<glm::vec3>` dla werteksów pozycji i normalnych, oraz `vector<glm::vec2>` dla werteksów tekstur. Dodatkowo trzeba zdefiniować prywatną zmienną typu `GLuint` zliczającą ilość wielokątów, z których zbudowany jest obiekt. W konstruktorze klasy `ObjectLoader` należy także uwzględnić jako argument ścieżkę do wczytywanego pliku. Do utworzonych wektorów wypada przygotować „*gettery*”, żeby klasa rysująca mogła pobrać przechowywane w nich wartości (listing 4.7.2).

Listing 4.7.2 Funkcje zwracające współrzędne

```
std::vector<glm::vec3> ObjectLoader::getVertices() {
	return mVertices;
}

std::vector<glm::vec2> ObjectLoader::getUvs() {
	return mUvs;
}

std::vector<glm::vec3> ObjectLoader::getNormals() {
	return mNormals;
}
```

Następnie trzeba dodać instrukcję warunkową wywołującą jeszcze niestworzoną funkcję `loadObj` i sprawdzającą czy wczytanie obiektu się udało (listing 4.7.3). Jeżeli obiekt zostanie poprawnie wczytany, ilość wszystkich wektorów położeń werteksów powinna być dokładnie taka sama, jak normalnych i współrzędnych teksturowania. Można to prosto sprawdzić wypisując w konsoli rozmiary wszystkich wektorów.

Listing 4.7.3 Konstruktor klasy `ObjectLoader`

```
ObjectLoader::ObjectLoader(std::string filePath) : mIndicesCount(0) {
	if (!loadObj(filePath)) {
		std::cout << "Couldn't load the Object from file." << std::endl;
	}
	else {
		std::cout << "Object loaded successfully." << std::endl;
		std::cout << "Vertices = " << mVertices.size() << std::endl;
		std::cout << "Texture Coordinates = " << mUvs.size() << std::endl;
		std::cout << "Normals = " << mNormals.size() << std::endl;
		std::cout << "Faces = " << mIndicesCount << std::endl << std::endl;
	}
}
```

Trzeba również stworzyć funkcję `loadObj` (listing 4.7.4), która zwierać będzie pętlę zajmującą się wczytywaniem i parsowaniem kolejnych wierszy pliku. Można też stworzyć cztery funkcje parsujące poszczególne atrybuty werteksów i chwilo zostawić je puste. Funkcja `loadObj` najpierw otwiera plik z podanej wcześniej ścieżki i sprawdza czy ta operacja się udała. Jeżeli tak to uruchamiana jest pętla, która będzie wczytywała kolejne linie pliku, aż plik się skończy. Z każdej linii oddzielane są pierwsze dwa znaki. Na ich podstawie można wywnioskować zawartość linii i na tej podstawie przekazać ją dalej do odpowiedniej funkcji. Na koniec uruchamiana jest nieistniejąca jeszcze funkcja zajmująca się tzw. operacją indeksowania, czyli poukładaniem werteksów w odpowiedniej kolejności, tak aby na ich podstawie OpenGL mógł poprawnie narysować obiekt.

Listing 4.7.4 Funkcja `loadObj`

```
bool ObjectLoader::loadObj(std::string filePath) {
	std::ifstream file(filePath);
	if (!file.is_open()) {
		std::cout << "Could't open the file!" << std::endl;
		return false;
	}
	else {
		std::cout << "File " << filePath << " opened." << std::endl;
	}

	std::string line;
	while (!file.eof()) {
		std::getline(file, line);
		std::string type = line.substr(0, 2);

		if (type.compare("v ") == 0) {
			loadVertices(line);
		}
		else if (type.compare("vt") == 0) {
			loadUvs(line);
		}
		else if (type.compare("vn") == 0) {
			loadNormals(line);
		}
		else if (type.compare("f ") == 0) {
			loadFaces(line);
		}
	}
```

```
	removeIndexes();

	return true;
}
```

Pierwsze trzy typy danych są dosyć proste do przetworzenia. Najpierw jednak trzeba jednak utworzyć krótką funkcję `splitLine`, której zadaniem będzie utworzenie pojemnika (wektora łańcuchów), przechowującego werteksy pobrane z przekazanej do funkcji linii pliku (listing 4.7.5). Do funkcji oprócz linii trzeba przekazać także znak separacji (ang. *delimiter)* na podstawie którego mają być rozoddzielane fragmenty linii. Można więc od razu zdefiniować stałą (`const`) będącą prywatnym polem klasy `ObjectLoader` typu `char` o nazwie `mDelimiter`, która będzie zawierać znak separacji (spację). Funkcja splitLine dzieli linie na mniejsze fragmenty wyznaczone przez znak separacji. Poszczególne fragmenty są dodawane do pojemnika, który jest zwracany przez tę funkcję. Trzeba pamiętać o dodaniu instrukcji warunkowej, która będzie zapobiegać przekazania pierwszych znaków identyfikujących typ danych do zwracanego pojemnika.

Listing 4.7.5 Funkcja `splitLine`

```
std::vector<std::string> ObjectLoader::splitLine(const std::string &line,
	char delimiter) {
	std::vector<std::string> container;
	std::string token;
	std::stringstream ss(line);

	while (std::getline(ss, token, delimiter)) {
		if (token != "v" && token != "vn" && token != "vt" && token != "f"){
			container.push_back(token);
		}
	}

	return container;
}
```

Pozostaje przygotować funkcje odpowiedzialne za parsowanie linii z poszczególnymi typami wektorów, które będą wykorzystywać dane przygotowane przez funkcję `splitLine` i zapisywać je do trzech prywatnych pojemników o nazwach `mRawVertices`, `mRawUvs`, `mRawNormals` (listing 4.7.6, 4.7.7 oraz 4.7.8). Dla

werteksów należy dodatkowo przygotować tzw. „*getter*”, ponieważ będą później używane przez klasę PhysX.

Listing 4.7.6 Funkcja `loadVertices`

```
void ObjectLoader::loadVertices(std::string line) {
	glm::vec3 vertex;
	std::vector<std::string> container = splitLine(line, mDelimiter);

	vertex.x = std::stof(container[0]);
	vertex.y = std::stof(container[1]);
	vertex.z = std::stof(container[2]);

	mRawVertices.push_back(vertex);
}
```

Listing 4.7.7 Funkcja `loadUvs`

```
void ObjectLoader::loadUvs(std::string line) {
	glm::vec2 uv;
	std::vector<std::string> container = splitLine(line, mDelimiter);

	uv.x = std::stof(container[0]);
	uv.y = std::stof(container[1]);

	mRawUvs.push_back(uv);
}
```

Listing 4.7.8 Funkcja `loadNormals`

```
void ObjectLoader::loadNormals(std::string line) {
	glm::vec3 normal;
	std::vector<std::string> container = splitLine(line, mDelimiter);

	normal.x = std::stof(container[0]);
	normal.y = std::stof(container[1]);
	normal.z = std::stof(container[2]);

	mRawNormals.push_back(normal);
}
```

Funkcja zajmująca się wczytywaniem wielokątów jest nieco bardziej skomplikowana, ponieważ dane, które będzie wczytywać są potrzebne do ułożenia pozostałych współrzędnych położenia i innych atrybutów werteksów w odpowiedniej kolejności (4.7.9). Wewnątrz niej trzeba zdefiniować kolejny znak separacji, ponieważ

każda linia zawiera trzy werteksy trójkąta, a z każdym z nich związana jest dodatkowo tekstura i normalna oddzielona znakiem „/”. Trzeba, wobec tego stworzyć kolejne pojemniki, które będą przechowywać indeksy poszczególnych współrzędnych. Zmienną odpowiadającą za indeksy werteksów należy pozostawić jako publiczną, bo będzie wykorzystywana przez klasę PhysX. Pozostałe dwie można ustawić jako prywatne. Następnie za pomocą funkcji `splitLine` trzeba podzielić linię na trzy fragmenty, z których wewnątrz pętli trzeba wyłuskiwać indeksy poszczególnych werteksów, tekstur i normalnych. Trzeba pamiętać o dodaniu instrukcji warunkowych przed uzupełnieniem `mUvIndices` oraz `mNormalIndices`, ponieważ odpowiadające im współrzędne nie zawsze muszą istnieć.

Listing 4.7.9 Funkcja `loadFaces`

```
void ObjectLoader::loadFaces(std::string line) {
	char faceDelimiter = '/';

	std::vector<std::string> containerOne = splitLine(line, mDelimiter);
	std::vector<std::string> containerTmp;

	for (int i = 0; i < 3; i++) {
		containerTmp = splitLine(containerOne[i], faceDelimiter);

		mVertexIndices.push_back(std::stof(containerTmp[0]));

		if (!mRawUvs.empty()) {
			mUvIndices.push_back(std::stof(containerTmp[1]));
		}
		if(!mRawNormals.empty())
			mNormalIndices.push_back(std::stof(containerTmp[2]));
		}

	mIndicesCount += 3;
}
```

Kolejna funkcja to `removeIndexes` (listing 4.7.10), której zadaniem jest posegregowanie współrzędnych w dobrej kolejności. Wewnątrz pętli, której warunkiem jest wielkośc wektora przechowującego indeksy, dla każdej współrzędnej, trzeba znaleźć jej kolejne wierzchołki. Ponieważ indeksy współrzędnych są w pliku *.obj* numerowane od 1, a nie od 0 tak jak w języku C++, trzeba pamiętać, aby od szukanego indeksu odjąć tę liczbę. Należy też pamiętać o dodaniu instrukcji warunkowych przed współrzędnymi

tekstur i normalnych, ponieważ tak jak w przypadku poprzedniej funkcji mogą one nie istnieć.

Listing 4.7.10 Funkcja `removeIndexes`

```
void ObjectLoader::removeIndexes() {
	for (unsigned int i = 0; i < mIndicesCount; i++) {
		int vertexIndex = mVertexIndices[i];
		glm::vec3 vertex = mRawVertices[vertexIndex - 1];
		mVertices.push_back(vertex);

		if (mTextCoordsCount) {
			int uvIndex = mUvIndices[i];
			glm::vec2 uv = mRawUvs[uvIndex - 1];
			mUvs.push_back(uv);
		}

		if (mNormalsCount) {
			int normalIndex = mNormalIndices[i];
			glm::vec3 normal = mRawNormals[normalIndex - 1];
			mNormals.push_back(normal);
		}
	}
}
```

Do przechowywania pomocniczych plików z obiektami lub teksturami warto utworzyć nowy folder o nazwie *resources*, aby zachować przejrzystość. Przy podawaniu ścieżki do pliku wystarczy podać względną ścieżkę względem folderu projektu (listing 4.7.11).

Listing 4.7.11 Utworzenie obiektu klasy `ObjectLoader`

```
std::string tableFilePath = "External/resource/tableDone.obj";
ObjectLoader tableObj(tableFilePath);
```

Ponieważ utworzone przed chwilą zmienne nie zostały jeszcze przekazane do klasy rysującej, po uruchomieniu programu stół nie zostanie wyświetlony. Zanim to nastąpi klasę `Actor` trzeba zmodyfikować w taki sposób, żeby umiała rysować obiekty na podstawie informacji z pliku *wavefront*. Jeżeli w konsoli nie pojawiły się żadne błędy, można założyć, że dane zostały wczytane poprawnie. Można to aktualnie zweryfikować tylko poprzez debugowanie kodu.

4.8 Rysowanie skomplikowanych obiektów przy pomocy OpenGL

Stworzona wcześniej klasa `Actor`, nie została napisana w elastyczny sposób, więc nie da się zmienić ani kształtu, ani koloru obiektów poprzez przekazanie jej danych. Sposób działania jej buforów jest też dosyć ograniczony. Aby mogła współdziałać z klasą `ObjectLoader` trzeba ją zmodyfikować.

Pierwszym krokiem będzie utworzenie prywatnych zmiennych typu `vector` o nazwach `mVertices`, `mUvs` i `mNormals`, które mają docelowo przechowywać dane o obiekcie 3D odczytane z pliku. W argumencie konstruktora trzeba podać obiekt klasy `ObjectLoader`, a jego listę inicjacyjną należy uzupełnić nowe pola (listing 4.8.1). Kolejną zmianą jest funkcja `prepareBuffers`, która generuje obiekt VAO i wywołuje funkcje pomocnicze przygotowujące odpowiednie bufory (listing 4.8.2).

Listing 4.8.1 Nowy konstruktor klasy `Actor`

```
Actor::Actor(ObjectLoader objLoad) : mVertices(objLoad.mVertices),
	mUvs(objLoad.mUvs), mNormals(objLoad.mNormals) {
	prepareBuffers();
}
```

Listing 4.8.2 Funkcja `prepareBuffers`

```
void Actor::prepareBuffers() {
	glGenVertexArrays(1, &mVao);
	glBindVertexArray(mVao);

	preparePosition();
	prepareUv();
	prepareNormal();

	glBindVertexArray(0);
}
```

Trzeba także przygotować pola przechowujące bufory. Istniejące już pole `mVbo` będzie dalej obsługiwać wektory odpowiadające za pozycję. Do współrzędnych teksturowania utworzone zostanie osobne pole `mUvBuffer`, a do normalnych `mNormalBuffer`. Funkcje implementujące generowanie buforów będą działać podobnie jak ich poprzednie wersje, jednakże teraz zamiast podawać stałe wartości

w `glBufferData`, należy podać wskaźnik na pierwszą komórkę odpowiednio `mVertices` (listing 4.8.3), `mUvs` (listing 4.8.4) lub `nNormals` (4.8.5).

Listing 4.8.3 Funkcja `preparePosition`

```
void Actor::preparePosition() {
	glGenBuffers(1, &mVbo);
	glBindBuffer(GL_ARRAY_BUFFER, mVbo);
	glBufferData(GL_ARRAY_BUFFER, mVertices.size() * sizeof(glm::vec3),
		&mVertices[0], GL_STATIC_DRAW);

	glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
	glEnableVertexAttribArray(0);
}
```

Listing 4.8.4 Funkcja `prepareUv`

```
void Actor::prepareUv() {
	glGenBuffers(1, &mUvBuffer);
	glBindBuffer(GL_ARRAY_BUFFER, mUvBuffer);
	glBufferData(GL_ARRAY_BUFFER, mUvs.size() * sizeof(glm::vec2), &mUvs[0],
		GL_STATIC_DRAW);

	glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
	glEnableVertexAttribArray(1);
}
```

Listing 4.8.5 Funkcja `prepareNormal`

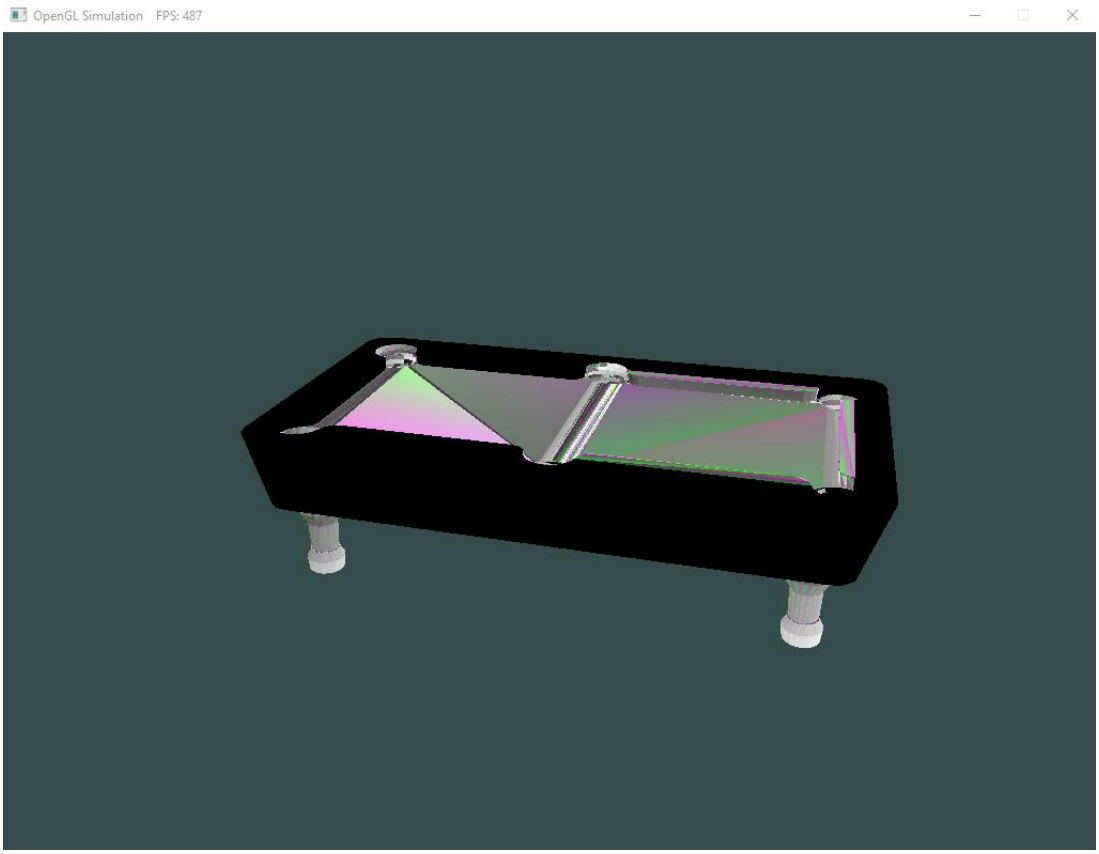
```
void Actor::prepareNormal() {
	glGenBuffers(1, &mNormalBuffer);
	glBindBuffer(GL_ARRAY_BUFFER, mNormalBuffer);
	glBufferData(GL_ARRAY_BUFFER, mNormals.size() * sizeof(glm::vec3),
		&mNormals[0], GL_STATIC_DRAW);

	glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)0);
	glEnableVertexAttribArray(2);
}
```

Funkcja `draw` pozostanie niemal niezmieniona, jednakże teraz zamiast na sztywno wpisanej liczby trójkątów, dynamicznie pobiera ich ilość bazując na wielkości wektora `mVertices`.

Listing 4.8.6 Uaktualniona funkcja `draw`

```
void Actor::draw() {
	glBindVertexArray(mVao);
	glDrawArrays(GL_TRIANGLES, 0, static_cast<GLsizei>(mVertices.size()));
	glBindVertexArray(0);
}
```

Zmienia się także polecenie tworzące instancję klasy `Actor` w funkcji `main`. Należy w głównej pętli programu zmienić funkcję `initModel` tak aby nie pobierała żadnych argumentów. Dzięki temu po uruchomieniu programu powinien być widoczny stół bildardowy wyglądający dokładnie tak samo jak we wcześniej użytym do jego stworzenia programie *Blender*. Jak widać na rysunku 4.8.1, obiekt ma dziwne kolory w niektórych miejscach, ponieważ shadery nie są jeszcze dostosowane do zaktualizowanej klasy `Actor`.

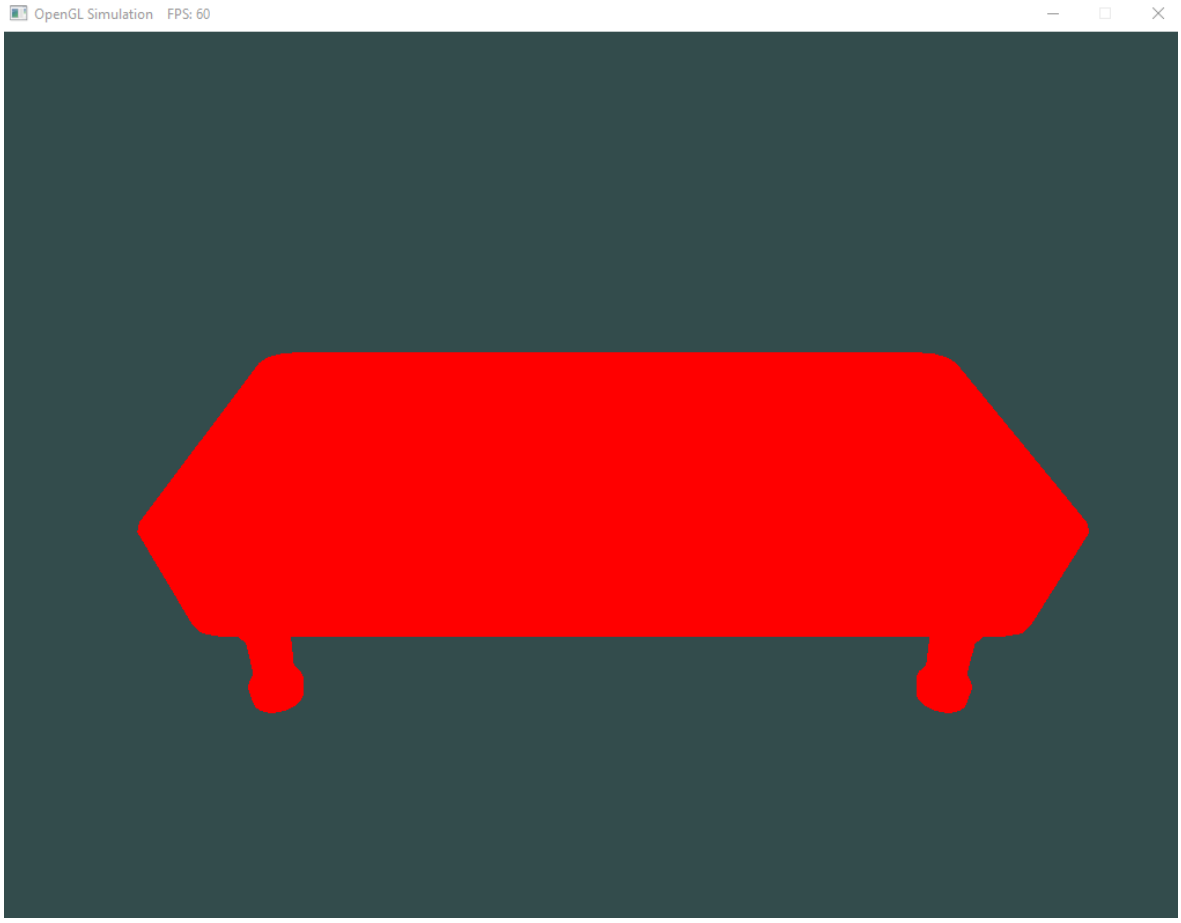


Rys. 4.8.1 Stół bilardowy po uaktualnieniu klasy `Actor`

4.9 Zaawansowany shading

Do shadera dostarczana jest niewystarczająca liczba werteksów koloru. Efektem jest to, że część trójkątów, z których obiekt jest zbudowany ma kolor, a część pozostaje czarna. Celem tego podrozdziału będzie, wobec tego nadanie obiektowigładkiej czarnej faktury i dodanie do sceny oświetlenia.

Aby stół posiadał jeden kolor, wystarczy chwilowo w pliku *fragmentShader.frag* ustawić zmienną `color` na pożądaną wartość koloru (składowe RGB). W tej pracy tymczasowo będzie to `1.0f, 0.0f, 0.0f`, czyli kolor czerwony (rys. 4.9.1).



Rys. 4.9.1 Stół bilardowy z ustawionym czerwonym kolorem

Jak widać na powyższym rysunku, nie da się teraz odróżnić wewnętrznych krawędzi stołu, ponieważ każdy jego trójkąt ma dokładnie taki sam kolor. Aby to zmienić trzeba dodać oświetlenie. W tymcelu konieczna będzie zmiana kodu z pliku zawierającego vertex shader i uaktualnienie zmiennych, które do niego wchodzą oraz tych, które są przekazywane dalej do shadera fragmentów (listing 4.9.1). Zmienne uniform

odpowiadające za modyfikacje macierzy MVP i jej składowych pozostają niezmienione, jednakże dodana zostanie nowa zmienna uniform, która przechowywać będzie pozycję oświetlenia na scenie (listing 4.9.1).

Listing 4.9.1 Zmienne w pliku *vertexShader.vs*

```
layout(location = 0) in vec3 position;
layout(location = 1) in vec2 uv;
layout(location = 2) in vec3 normal;

out vec3 POSITION;
out vec2 UV;
out vec3 NORMAL;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform vec3 lightPosition;
out vec3 camDirection;
out vec3 lightDirection;
```

Następnie w funkcji `main` tego shadera trzeba ustalić położenie i normalne werteksów korzystając z przekazanych do shadera danych. Trzeba także obliczyć wartość zmiennych `camDirection` oraz `lightDirection`, które zostaną przekazane do shadera fragmentów (listing 4.9.2).

Listing 4.9.2 Obliczenia w pliku *vertexShader.vs*

```
void main() {
	gl_Position = projection * view * model * vec4(position, 1.0f);

	POSITION = (model * vec4(position, 1.0f)).xyz;
	NORMAL = (view * model * vec4(normal, 0.0f)).xyz;
	UV = uv;

	vec3 camVertex = (view * model * vec4(position, 1.0f)).xyz;
	camDirection = vec3(0.0f, 0.0f, 0.0f) - camVertex;

	vec3 camLightPos = (view * vec4(lightPosition, 1.0f)).xyz;
	lightDirection = camLightPos + camDirection;
}
```

Zmian wymaga także kod shadera fragmentów z pliku *fragmentShader.frag*, dodanie nowych zmiennych wejściowych, przesłanych z shadera werteksów oraz dodanie nowej zmiennej uniform odpowiadajacych za pozycję światła na scenie oraz próbnik tekstur, który pomoże przy ustaleniu materiału powierzchni stołu (listing 4.9.3).

Listing 4.9.3 Zmienne w pliku *fragmentShader.frag*

```
in vec3 POSITION;
in vec2 UV;
in vec3 NORMAL;

out vec3 color;

in vec3 camDirection;
in vec3 lightDirection;
uniform vec3 lightPosition;
```

Następnie na początek funkcji `main` shadera fragmentów trzeba ustalić kolor padającego światła, który w tej pracy będzie biały, moc z jaką światło ma padać oraz dystans jaki przebywa od swojego źródła do obiektu (listing 4.9.4).

Listing 4.9.4 Ustawienia światła

```
vec3 lightColor = vec3(1.0f, 1.0f, 1.0f);
float lightPower = 50.0f;
float lightDistance = length(lightPosition - POSITION);
```

Zaraz po ustawieniach światła, trzeba ustalić jaki kolor rozproszony, otaczający oraz odbicia materiału obiektu (listing 4.9.5). W tej pracy do stołu bilardowego nie będzie przypisana żadna tekstura, a kolor będzie ustawiony na czerwony.

Listing 4.9.5 Ustawienia materiałów

```
vec3 materialDiffuseColor = vec3(1.0f, 0.0f, 0.0f);
vec3 materialAmbientColor = vec3(0.2f, 0.2f, 0.2f);
vec3 materialSpecularColor = vec3(0.3f, 0.3f, 0.3f);
```

Trzeba także obliczyć kąt pomiędzy padającym światłem, a normalną obiektu (listing 4.9.6), co jest potrzebne do obliczania kolorów pojedynczych pikseli w komponencie oświetlenia rozproszonego. Oprócz tego potrzebne jest też obliczenie kąta pomiędzy wektorem od kamery do obliczanego piksela, a wektorem odbicia (listing 4.9.7),

ponieważ będzie to miało kluczowe znaczenie przy ustalaniu komponentu rozbłysku oświetlenia.

Listing 4.9.6 Obliczenie wartości koloru rozproszenia

```
vec3 normal = normalize(NORMAL);
float cosTheta = clamp(dot(normal, normalize(lightDirection)), 0, 1);
```

Listing 4.9.7 Obliczenie wartości koloru odbicia

```
vec3 reflection = reflect(-normalize(lightDirection), normal);
float cosAlpha = clamp(dot(normalize(camDirection), reflection), 0, 1);
```

W pliku *fragmentShader.frag*, należy także dodać kod ustalający kolor, jaki ma posiadać oświetlony stół bilardowy, uwzględniający światło otoczenia, rozproszone i rozbłysku (listing 4.9.8), wykorzystując wszystkie obliczone wcześniej wartości. Parametr rozbłysku decydujący o wielkości plamy „zajączka" można kontrolować zmniejszając lub zwiększając drugi argument funkcji `pow`. Im mniejsza wartość tym szersze światło. Utworzony kolor reprezentuje tzw. model Phonga [33].

Listing 4.9.8 Obliczenie wartości koloru

```
vec3 diffuseColor = materialDiffuseColor * lightColor * lightPower * cosTheta /
     pow(lightDistance, 2);
vec3 ambientColor = materialDiffuseColor * ambientPercentage;
vec3 specularColor = materialSpecularColor * lightColor * lightPower *
     pow(cosAlpha, 5) / pow(lightDistance, 2);
color = diffuseColor + ambientColor + specularColor;
```

Należy również pamiętać o ustaleniu wartości zmiennych uniformów shadera w funkcji `main` pliku *Source.cpp* (listing 4.9.9) i przekazaniu do niego w ten sposób pozycji światła na scenie (listing 4.9.10).

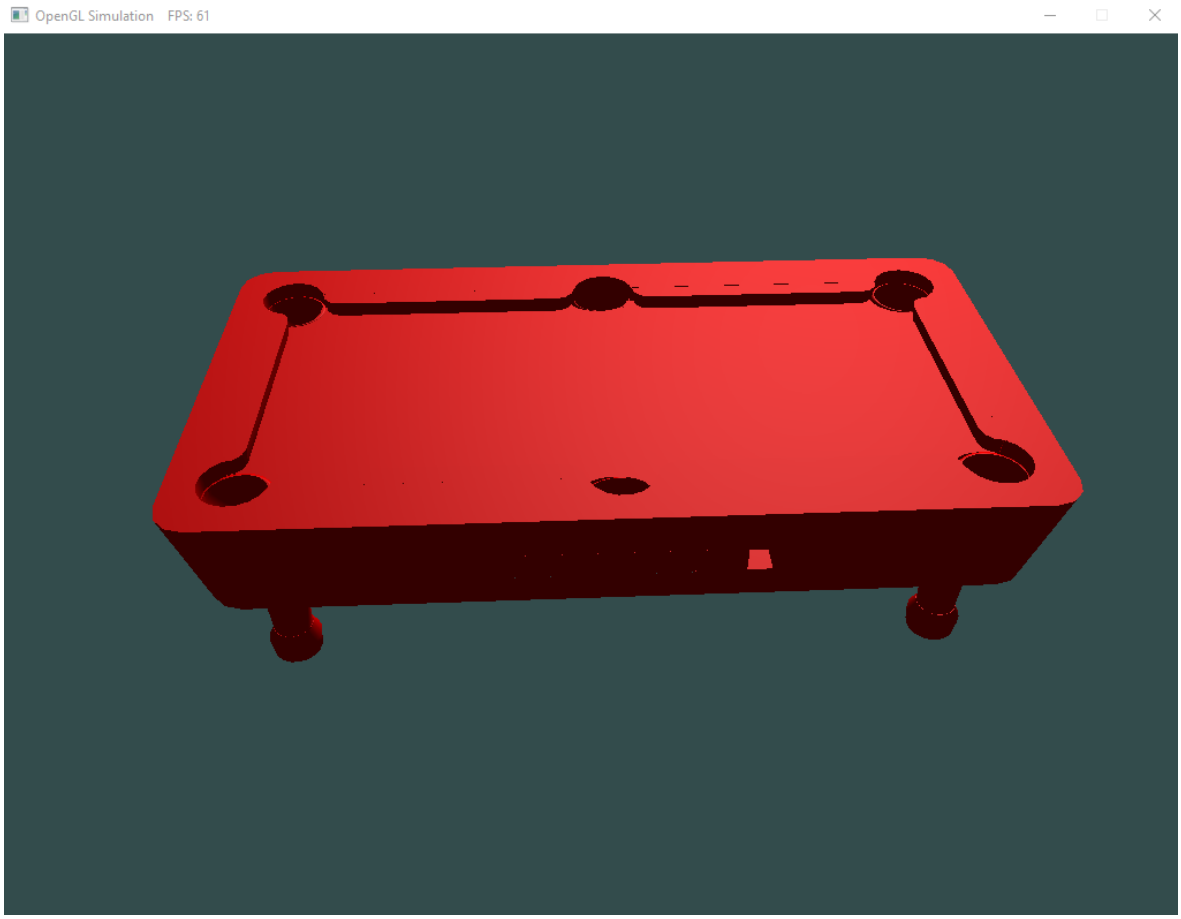
Listing 4.9.9 Dodanie uniformów potrzebnych do stworzenia oświetlenia sceny

```
GLuint lightID = glGetUniformLocation(shader.mShadersId, "lightPosition");
```

Listing 4.9.10 Przekazanie wartości światła z głównej pętli do Shaderów

```
glm::vec3 lightPos = glm::vec3(2.0f, 5.0f, -6.0f);
glUniform3f(lightID, lightPos.x, lightPos.y, lightPos.z);
```

Po wykonaniu wszystkich powyższych zmian i uruchomieniu programu, powinien ukazać się oświetlony stół (rys. 4.9.2).



Rys. 4.9.2 Stół bilardowy z dodanym oświetleniem

4.10 Statyczny TriangleMesh w PhysX

Stół jest już rysowany za pomocą OpenGL, więc nadszedł czas, żeby dodać mu także właściwości fizyczne, co uzyskamy przekazując dane o nim także do klasy `PhysX`. Pierwszym krokiem będzie zainicjowanie biblioteki *Physx Cooking*. Jest ona odpowiedzialna za serializację dużych zbiorów danych, a takim właśnie zbiorem są wczytane z pliku werteksy i ich indeksy. Zainicjalizowanie tej biblioteki następuje poprzez wywołanie metody `PxCreateCooking` z podanymą w argumentach wersją biblioteki Physx, obiektem `*mFoundation` oraz skalą tolerancji (listing 4.10.1).

Listing 4.10.1 Funkcja `initCooking`

```
void PhysX::initCooking() {
```

```
	mCooking = PxCreateCooking(PX_PHYSICS_VERSION, *mFoundation,
		PxCookingParams(PxTolerancesScale()));
	if (!mCooking) cout << ("PxCreateCooking failed!") << endl;
}
```

Kiedy biblioteka PhysX Cooking jest już zainicjalizowana, można przejść do utworzenia deskryptora siatki (4.10.2). Aby móc przekazać do niego niepoukładane werteksy (a w zasadzie tylko ich położenia) oraz ich indeksy, trzeba utworzyć dwie pomocnicze funkcje konwertujące obiekty typu `vector<glm::vec3>` na `PxVec3*` (listing 4.10.3) oraz `vector<unsigned int>` na `PxU32*` (listing 4.10.4). Zamiast tworzyć pomocnicze funkcje w klasie `PhysX`, można też podczas pobierania danych z pliku w klasie `ObjectLoader`, zapisywać je w dwóch różnych typach. W tej pracy zostało wykorzystane pierwsze podejście. Dzięki temu wszystko co jest związane z fizyką dzieje się w jednej klasie `PhysX`. Przy większym projekcie możnaby rozważyć zmianę architektury i rozłożyć klasę `PhysX` na kilka klas.

Listing 4.10.2 Funkcja `createMeshDesc`

```
PxTriangleMeshDesc PhysX::createMeshDesc(ObjectLoader &object) {
	PxTriangleMeshDesc meshDesc;

	std::vector<glm::vec3> pxVertices = object.getPxVertices();
	meshDesc.points.count = static_cast<PxU32>(pxVertices.size());
	meshDesc.points.stride = sizeof(PxVec3);
	meshDesc.points.data = convertVerticesToPx(pxVertices);

	std::vector<unsigned int> pxIndices = object.getPxIndices();
	meshDesc.triangles.count = static_cast<PxU32>(pxIndices.size()) / 3;
	meshDesc.triangles.stride = 3 * sizeof(PxU32);
	meshDesc.triangles.data = convertIndicesToPx(pxIndices);

	if (!meshDesc.isValid()) {
		std::cout << "Triangle mesh descriptor is not valid" << std::endl;
	}

	return meshDesc;
}
```

Listing 4.10.3 Funkcja `convertVerticesToPx`

```
PxVec3* PhysX::convertVerticesToPx(vector<glm::vec3> glVertices) {
	PxVec3* pxVertices = new PxVec3[glVertices.size()];
```

```
	for (int i = 0; i < glVertices.size(); i++) {
		pxVertices[i].x = glVertices[i].x;
		pxVertices[i].y = glVertices[i].y;
		pxVertices[i].z = glVertices[i].z;
	}

	return pxVertices;
}
```

Listing 4.10.4 Funkcja `convertIndicesToPx`

```
PxU32* PhysX::convertIndicesToPx(vector<unsigned int> glIndices) {
	PxU32* pxIndices = new PxU32[glIndices.size()];

	for (int i = 0; i < glIndices.size(); i++) {
		pxIndices[i] = glIndices[i] - 1;
	}

	return pxIndices;
}
```

Kiedy deskryptor siatki jest już przygotowany można przejść do utworzenia samej siatki (listing 4.10.5). Tworzy się ją przy pomocy obiektu biblioteki Cooking używając funkcji `cookTriangleMesh` z podanym w argumentach deskryptorem oraz pustym buforem typu `PxDefaultMemoryOutputStream`. Następnie trzeba utworzyć zmienną typu `PxDefaultMemoryInputData`, do której należy przekazać dane zebrane w buforze. Następnie należy wywołać na rzecz obiektu `mPhysics` metodę `createTriangleMesh` z podanymi w argumencie zebranymi danymi. Tworzenie siatki składającej się z trójkątów w taki sposób nazywane jest „Offline Cooking'iem". Innym sposobem jej utworzenia byłoby skorzystanie z „Dynamic Cooking'u". Kod zajmowałby zdecydowanie mniej miejsca i byłby właściwie wywołaniem pojedynczej metody. Nie jest to jednak zalecane przez twórców SDK, którzy zachęcają do używania tej metody jedynie w przypadku, gdy siatka obiektu musiałaby się dynamicznie zmieniać [34].

Listing 4.10.5 Funkcja `createTriangleMesh`

```
PxTriangleMesh* PhysX::createTriangleMesh(PxTriangleMeshDesc meshDesc) {
	PxDefaultMemoryOutputStream buffer;

	if (!mCooking->cookTriangleMesh(meshDesc, buffer{
```

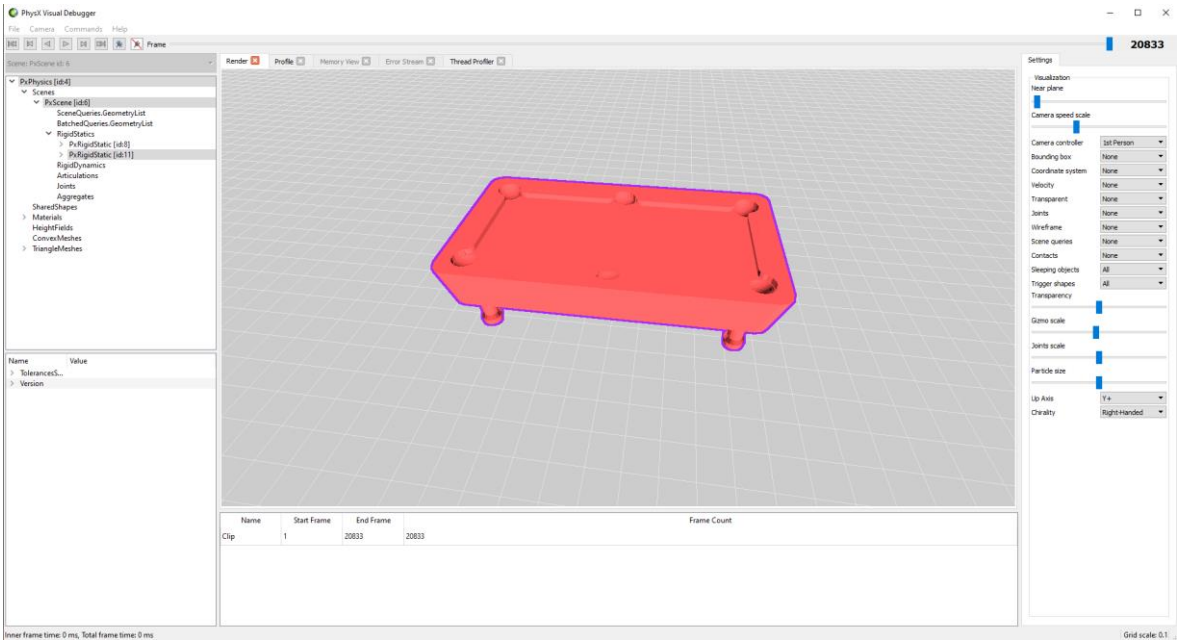
```
            std::cout << "Triangle mesh is nullptr" << std::endl;
            return nullptr;
        }
        PxDefaultMemoryInputData input(buffer.getData(), buffer.getSize());

        return mPhysics->createTriangleMesh(input);
}
```

Pozostało już tylko utworzyć fizycznego aktora z przygotowanej siatki (listing 4.10.6). Zostanie to zrealizowane w nowej funkcji `createStaticActorFromMesh` umieszczonej w klasie `PhysX`. Najpierw trzeba utworzyć kształt aktora metodą `createShape` wywoływaną na rzecz obiektu `mPhyscis`. W jej argumentach należy podać siatkę stołu bilardowego, materiał z jakiego ma zostać wykonany i wartość `true` oznaczającą, że kształt aktora będzie ekskluzywny tzn. nie może być dzielony z innymi obiektami. Kolejnym krokiem jest utworzenie ciała aktora używając metody `createRigidStatic` z podaną w argumencie pozycją obiektu na scenie. Aktor będzie statyczny, ponieważ stół bilardowy ma być jedynie fragmentem otoczenia dla odbijających się na nim kul. Teraz trzeba już tylko przyłączyć wcześniej utworzony kształt do ciała aktora i dodać go do sceny. Na koniec należy pamiętać o zwolniniu obiektu `staticShape`, ponieważ nie będzie ona już potrzebne.

Listing 4.10.6 Funkcja `createStaticActorFromMesh`

```
void PhysX::createStaticActorFromMesh(ObjectLoader &object, PxVec3 position) {
        PxTriangleMeshDesc meshDesc = createMeshDesc(object);

        PxTriangleMesh* triangleMesh = createTriangleMesh(meshDesc);

        PxShape* staticShape = mPhysics->
            createShape(PxTriangleMeshGeometry(triangleMesh), *mMaterial, true);

        PxRigidStatic* staticBody = mPhysics->
            createRigidStatic(PxTransform(position));

        staticBody->attachShape(*staticShape);

        mScene->addActor(*staticBody);

        staticShape->release();
}
```

Po uruchomieniu programu nie da się jeszcze zauważyć żadnych zmian, ponieważ do rysowania w OpenGL nie zostały wprowadzone żadne zmiany. Jednak, gdy przed jego startem włączony zostanie PVD, będzie można zauważyć, iż pojawił się tam stół bilardowy (rys. 4.10.1). Potwierdza to, że wersja fizyczna stołu jest poprawnie utworzona i można przejść do tworzenia kul.



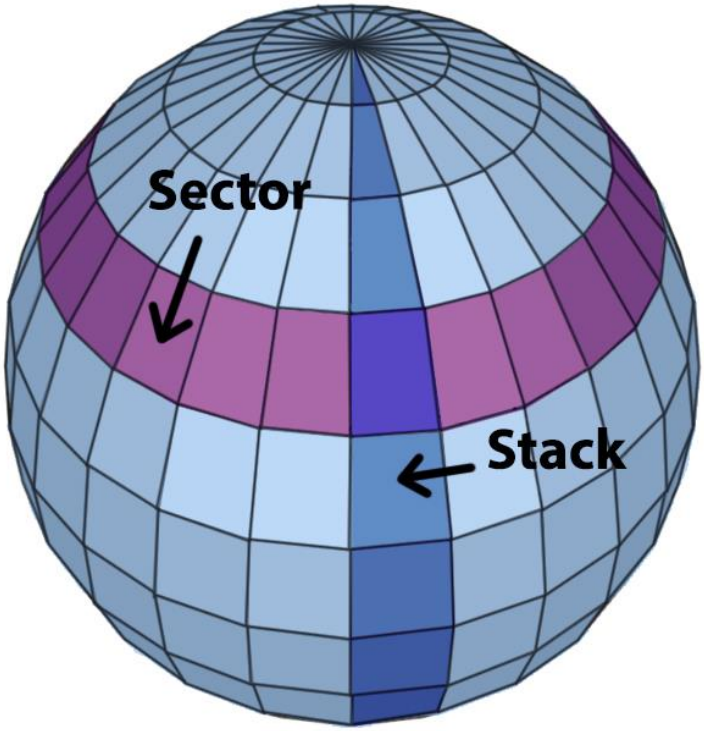
Rys 4.10.1 Stół widoczny w programie PVD

4.11 Generacja obiektów sferycznych w OpenGL

W przeciwieństwie do stołu, sfer nie wystarczy przygotować w zewnętrznym programie i po prostu wczytać przy pomocy gotowych funkcji. Obiekty stworzone w taki sposób mimo swojej prostoty, przynajmniej na pierwszy rzut oka, dla biblioteki Physx będą bardzo skomplikowane. Ich siatki składałby się z wielu tysięcy, a nawet kilukdziesięciu tysięcy werteksów, więc dynamiczne zmieniane ich pozycji na scenie wymagałoby ogromu obliczeń. Physx nie wspiera takiego typu rozwiązania i jeżeli aktorzy mają przemieszczać się na scenie podczas symulacji i wzajemnie na siebie oddziaływać, muszą mieć znacznie mniejszą siatkę. Sfery trzeba więc wygenerować własnoręcznie.

Pierwszym działaniem w tym kierunku powinno być przygotowanie klasy `Sphere`. Żeby nie powielać kodu, można ustawić klasę `Actor` jako jej klasę bazową. Dzięki temu nie będzie trzeba dodawać funkcji przygotowującej bufory pozycji, tekstur

i normalnych. Jako argumenty konstruktora należy podać zmienne `stacks` oraz `sectors` typu `GLint` (listing 4.11.1). Pierwsza przekazuje ilość pionowych pasów wielokątów, z których ma się składać powierzchnia kuli, natomiast druga odpowiada za liczbę wielokątów w paśmie (rys. 4.11.1). Teraz trzeba utworzyć trzy nowe metody `generateVertices`, `generateIndices` oraz `prepareIndices` i kolejno je wywołać (listing 4.11.1). Przed tą ostatnią należy jeszcze dodać `prepareBuffers` z klasy bazowej.



Rys. 4.11.1 Sfera z oznaczeniami

Listing 4.11.1 Konstruktor klasy `Sphere`

```
Sphere::Sphere(GLint stacks, GLint sectors) {
	generateVertices(stacks, sectors);
	generateIndices(stacks, sectors);

	prepareBuffers();
	prepareIndices();
}
```

Funkcja `generateVertices` generuje werteksy wykorzystując do obliczeń proste wzory (6 - 8), a następnie przekazuje je do prywatnych pojemników przechowujących pozycje, tekstury i normalne (listing 4.11.2).

$$x = \cos\frac{stackStep}{stackCount} * 2\pi * \sin\frac{sectorStep}{sectorCount} * \pi \quad (6)$$

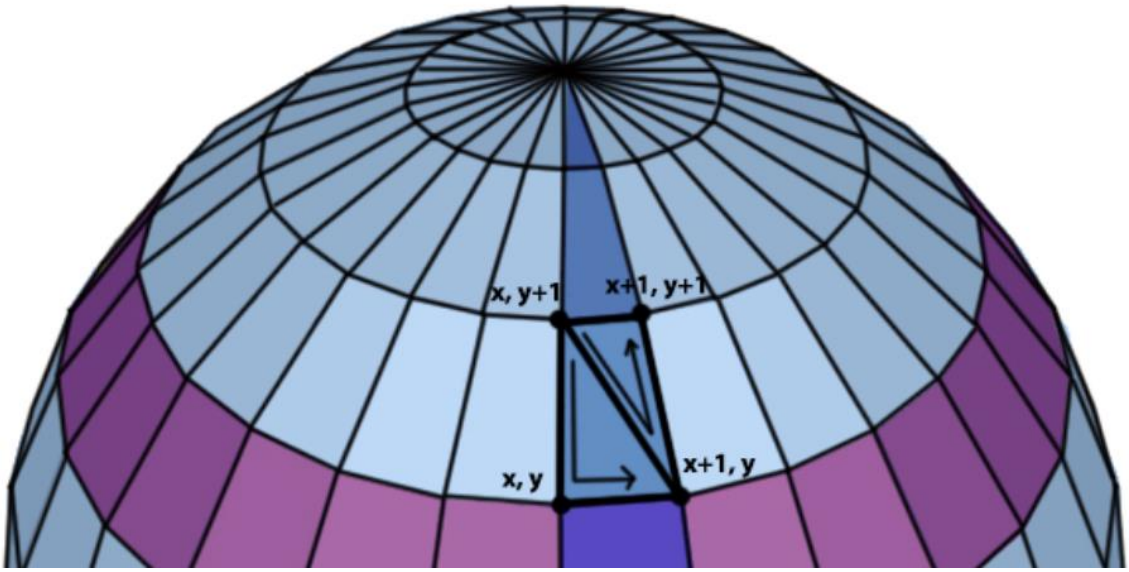
$$y = \cos\frac{sectorstep}{sectorCount} * \pi \quad (7)$$

$$z = \sin\frac{stackStep}{stackCount} * 2\pi * \sin\frac{sectorStep}{sectorCount} * \pi \quad (8)$$

Listing 4.11.2 Funkcja `generateVertices`

```
void Sphere::generateVertices(GLint stacks, GLint sectors) {
	for (int y = 0; y <= sectors; ++y) {
		for (int x = 0; x <= stacks; ++x) {
			float xSegment = static_cast<float>(x) /
				static_cast<float>(stacks);
			float ySegment = static_cast<float>(y) /
				static_cast<float>(sectors);

			float xPos = static_cast<float>(std::cos(xSegment * 2.0 *
				M_PI) * std::sin(ySegment * M_PI));
			float yPos = static_cast<float>(std::cos(ySegment * M_PI));
			float zPos = static_cast<float>(std::sin(xSegment * 2.0*M_PI)
				* std::sin(ySegment * M_PI));

			mVertices.push_back(glm::vec3(xPos, yPos, zPos));
			mUvs.push_back(glm::vec2(xSegment, ySegment));
			mNormals.push_back(glm::vec3(xPos, yPos, zPos));
		}
	}
}
```

Kiedy werteksy są już wygenerowane, trzeba jeszcze przygotować odpowiadające im indeksy, żeby OpenGL wiedział w jakiej kolejności rysować przygotowane dla niego trójkąty (listing 4.11.3). Na rysunku 4.11.2 można zobaczyć schemat kolejności indeksów w przypadku sfery.



Rys. 4.11.2 Sposób znajdowania indeksów dla generowanych trójkątów

Listing 4.11.3 Funkcja `generateIndices`

```
void Sphere::generateIndices(GLint stacks, GLint sectors) {
	for (int y = 0; y <= sectors; ++y) {
		for (int x = 0; x <= stacks; ++x) {
			mIndices.push_back((y + 1) * (stacks + 1) + x);
			mIndices.push_back(y * (stacks + 1) + x);
			mIndices.push_back(y * (stacks + 1) + x + 1);

			mIndices.push_back((y + 1) * (stacks + 1) + x);
			mIndices.push_back(y * (stacks + 1) + x + 1);
			mIndices.push_back((y + 1) * (stacks + 1) + x + 1);
		}
	}
}
```

Funkcja `prepareBuffers` jest już gotowa, pozostało więc przygotowanie metody `prepareIndices`, która porządkuje indeksy werteksów. Podobnie jak w przypadku poprzednich buforów, dla niej także trzeba stworzyć dedykowaną zmienną buforową mIbo typu GLuint. Jedyną różnicą jest brak `glVertexAttribPointer`, co wynika z faktu, iż przekazywane dane są jedynie indeksami potrzebnymi przy rysowaniu werteksów w odpowiedniej kolejności.

Listing 4.11.4 Funkcja `prepareIndices`

```
void Sphere::prepareIndices() {
	glBindVertexArray(mVao);

	glGenBuffers(1, &mIbo);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mIbo);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLushort) * mIndices.size(),
		&mIndices[0], GL_STATIC_DRAW);

	glBindVertexArray(0);
}
```

Jako że rysowanie będzie teraz przebiegało na podstawie indeksów zamiast poukładanych werteksów, trzeba użyć metody `glDrawElements` zamiast wcześniej używanej `glDrawArrays` (listing 4.11.5).

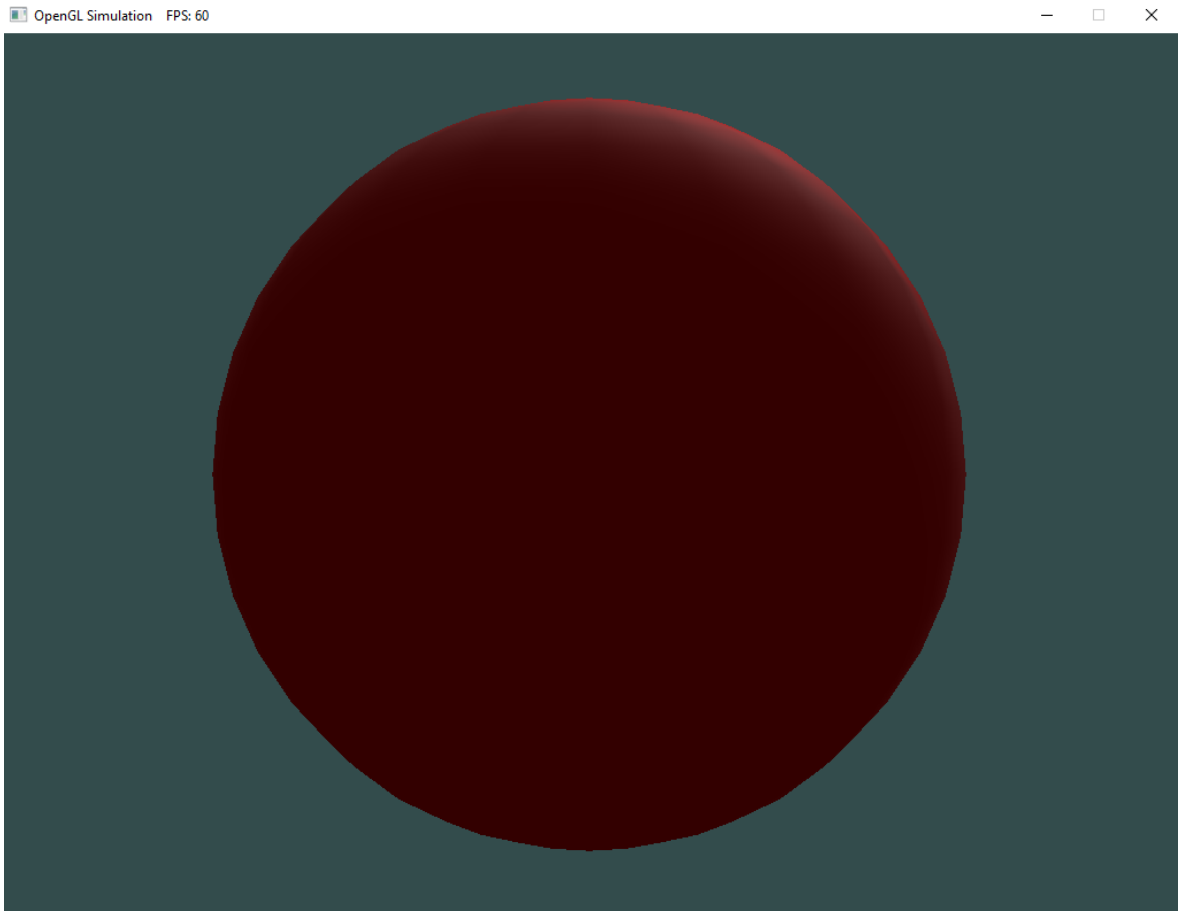
Listing 4.11.5 Funkcja draw klasy `Sphere`

```
void Sphere::draw() {
	glBindVertexArray(mVao);
	glDrawElements(GL_TRIANGLES, static_cast<GLsizei>(mIndices.size()),
		GL_UNSIGNED_SHORT, 0);
	glBindVertexArray(0);
}
```

Klasa `Sphere` jest już przygotowana, pozostało więc utworzyć jej instancję w funkcji main. Aby rysowana sfera była wystarczająco „kulista” należy przekazać w konstruktorze argumenty `sectors` i `stacks` równe przynajmniej 20 (listing 4.11.6). Po uruchomieniu programu, w oknie powinna pojawić się duża czerwona kula (rys. 4.11.3).

Listing 4.11.6 Utworzenie sfery w funkcji `main`

```
GLint sphereStacks = 20, sphereSectors = 20;
GLfloat sphereRadius = 0.02f;
Sphere sphere(sphereStacks, sphereSectors);
```

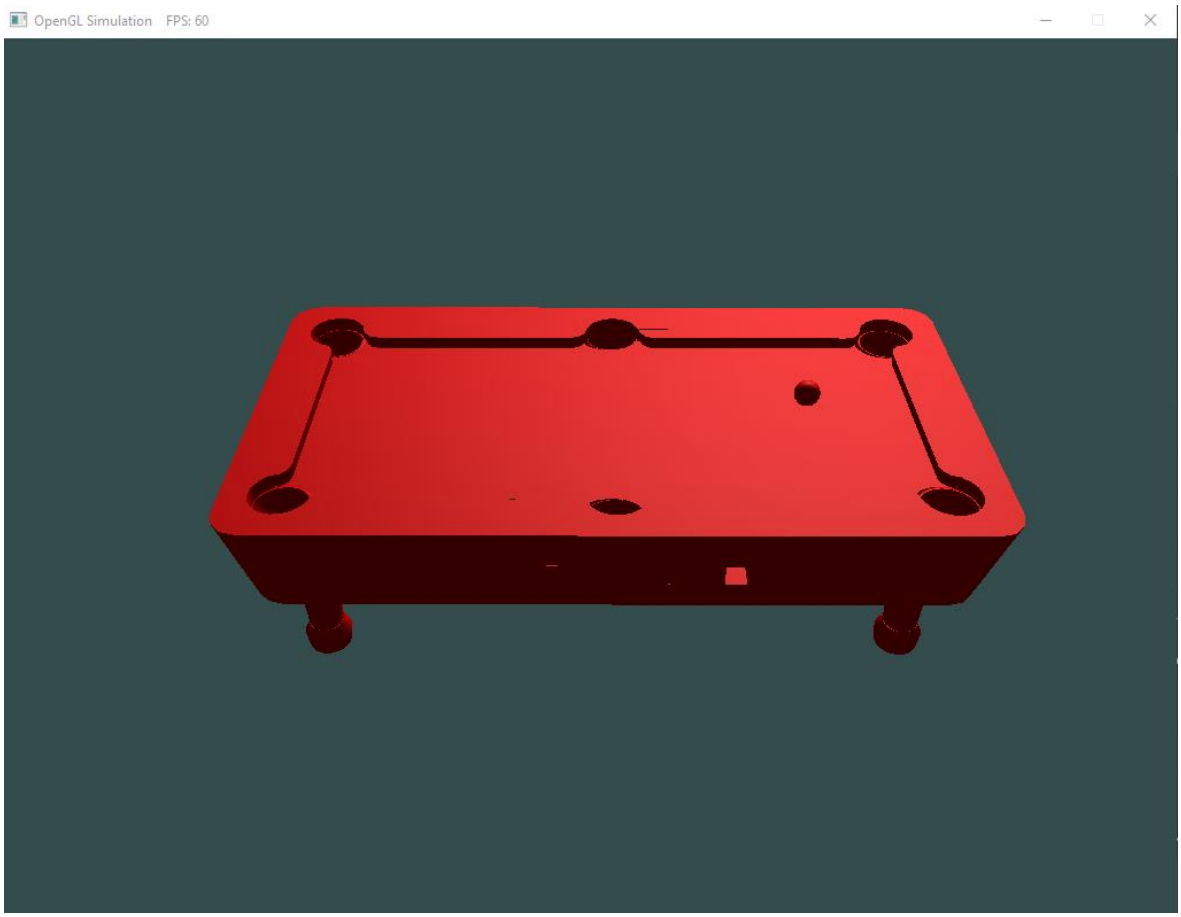


Rys 4.11.3 Sfera przed aktualizacją modelu

Sfery mają reprezentować kule bilardowe, trzeba je więc odpowiednio przeskalować. W głównej pętli programu przed narysowaniem kuli należy uaktualnić dla niej macierz modelu i ponownie przekazać wartości zmiennych uniform do shaderów (listing 4.11.7). Po tych zmianach, gdy program zostanie uruchomiony, w oknie powinien ukazać się stół, a na nim mała kula bilardowa (rys. 4.11.4).

Listing 4.11.7 Rysowanie sfery w głównej pętli programu

```
camera.initModel(glm::vec3(0.3f, 0.3f, 0.0f),
       glm::vec3(0.0f, 0.0f, 0.0f), 0.0f,
       glm::vec3(sphereRadius));

glUniformMatrix4fv(modelLocation, 1, GL_FALSE, &camera.mModel[0][0]);
glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &camera.mView[0][0]);
glUniformMatrix4fv(projetionLocation, 1, GL_FALSE, &camera.mProjection[0][0]);

sphere.draw();
```



Rys. 4.11.4 Stół bilardowy z pojedynczą kulą

4.12 Stworzenie dynamicznych obiektów sferycznych

Teraz, gdy kule bilardowe mogą już być rysowane, nadszedł czas, żeby dodać do nich „fizykę". W tym celu przygotowana zostanie funkcja `addDynamicSphereToScene` w klasie `PhysX`, która zajmie się tworzeniem dynamicznych aktorów reprezentujących kule (listing 4.12.1). Argumentami tej funkcji są promień i pozycja sfery na scenie. W tej funkcji, za pomocą funkcji `createShape` wywoływanej na obiekcie `mPhysics` należy stworzyć kształt sfery. Następnie w taki sam sposób przy pomocy funkcji `createRigidDynamic`, przygotować jej ciało i podpiąć do niego kształt. Aby nadać aktorowi wartości fizyczne, trzeba wywołać metodę `updateMassAndInertia`, która sprawi, że kule zdobędą pewną masę i pod wpływem dodanej wcześniej grawitacji, będą mogły wpaść do przygotowanych w stole bilardowym łuz. Pozostało dodać aktora do sceny. Część funkcji odpowiedzialnych za czynności wykonywane na konkretnym obiekcie, jest możliwa do użycia jedynie na obiekcie typu

PxRigidDynamic. Z tego powodu funkcja addDynamicSphereToScene zwraca obiekt sphereBody, co umożliwia przechowywanie go w np. zewnętrznym pojemniku.

Listing 4.12.1 Funkcja addDynamicSphereToScene

```
PxRigidDynamic* PhysX::addDynamicSphereToScene(PxReal radius, PxVec3 position) {
	PxShape* sphereShape = mPhysics->createShape(PxSphereGeometry(radius),
		*mMaterial);

	PxRigidDynamic* sphereBody = mPhysics-
		>createRigidDynamic(PxTransform(position));

	sphereBody->attachShape(*sphereShape);

	PxRigidBodyExt::updateMassAndInertia(*sphereBody, 10.0f);

	mScene->addActor(*sphereBody);
	sphereShape->release();

	return sphereBody;
}
```

W bilardzie, wszystkie uderzenia wykonywane są jedynie na białej kuli. Aby w trakcie symulacji przyłożyć do niej jakąś siłe, wprawiającą w ruch, trzeba posiadać dostęp do jej ciała. Dlatego właśnie trzeba stworzyć pomocniczą funkcję addSpecialSphereToTable, która uruchomi funkcję tworzącą sfery po czym jej wynik przypisze do publicznej zmiennej mWhiteBall (listing 4.12.2).

Listing 4.12.2 Funkcja addSpecialSphereToTable

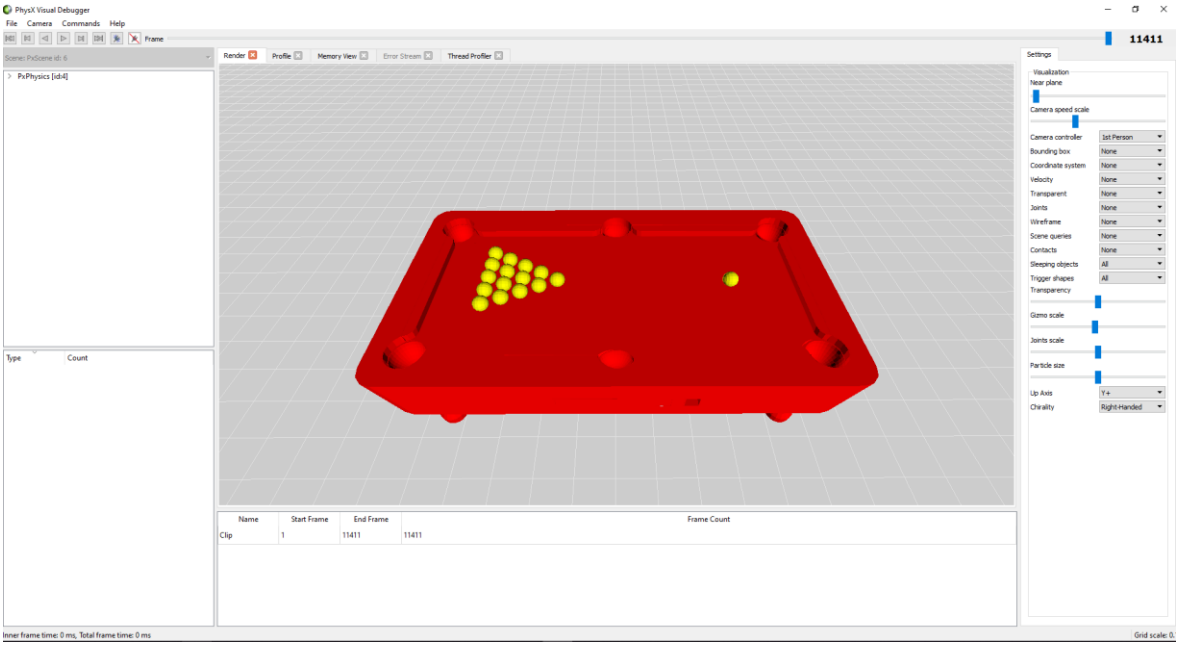
```
void PhysX::addSpecialSphereToTable(PxReal radius, PxVec3 position) {
	PxRigidDynamic* specialSphere = addDynamicSphereToScene(radius, position);
	mWhiteBall = specialSphere;
}
```

Teraz w pliku *Source.cpp*, należy przygotować funkcję addPhysicsBallsToTable, która obliczy odpowiednie położenie dla każdej z kul i stworzy ich fizyczne wersje (listing 4.12.3). Funkcja ta jest napisana tak, aby kule na stole zostały ułożone w trójkąt. Na koniec dodawana jest także biała kula, za pomocą której będzie można rozbić pozostałe.

Listing 4.12.3 Funkcja `addPhysicsBallsToTable`

```
static void addPhysicsBallsToTable(PhysX &pxObject, GLint rows, GLfloat radius) {
	float xBallPos = 0.0f;
	float yBallPos = 0.3f;
	float zBallPos = 0.0f;

	float xFirstBallInRowPos = -0.15f;
	float zFirstBallInRowPos = 0.0f;

	for (int i = 0; i < rows; i++, xFirstBallInRowPos -= 0.045f,
		zFirstBallInRowPos -= 0.025f) {
		xBallPos = xFirstBallInRowPos;
		zBallPos = zFirstBallInRowPos;

		for (int j = rows; j >= rows - i; j--, zBallPos += 0.045f) {
			pxObject.addDynamicSphereToScene(radius, PxVec3(xBallPos,
				yBallPos, zBallPos));
		}
	}

	PxVec3 whiteBallPos = PxVec3(0.3f, yBallPos, 0.0f);
	pxObject.addSpecialSphereToTable(radius, whiteBallPos);
}
```

Po wywołaniu powyższej funkcji w pętli głównej programu, w oknie GLFW nic się jeszcze nie zmieni, natomiast w programie PVD można podejrzeć, że kule rzeczywiście ułożone są tak jak powinny (rys. 4.12.1).



Rys 4.12.1 Kule bilardowe widoczne w PVD

Teraz trzeba przygotować funkcję `updateObjectsTransforms`, która w każdym kolejnym kroku głównej pętli programu będzie uwzględniać przesunięcia kul bilardowych (listing 4.12.4). Na początek funkcja ta sprawdza ilu jest dynamicznych aktorów na scenie i na tej podstawie tworzy przechowujący ich pojemnik. Następnie, zbiera do niego wszystkich aktorów z flagą `eRIGID_DYNAMIC` jeżeli tacy istnieją. Na koniec, uruchamiana jest pętla, która przy pomocy pomocniczych funkcji `setGlobalPose` (listing 4.12.5) i `setGlobalRotation` (listing 4.12.6) pobiera dla każdego aktora jego aktualną pozycję, rotację i kąt rotacji na scenie i przekazuje je do odpowiednich kontenerów.

Listing 4.12.4 Funkcja `updateObjectsTransforms`

```
void PhysX::updateObjectsTransforms(std::vector<glm::vec3>& position,
	std::vector<glm::vec3>& rotation, std::vector<GLfloat>& angle) {
	PxU32 actorsCount = mScene->getNbActors(PxActorTypeFlag::eRIGID_DYNAMIC);
	vector<PxRigidActor*> actorsList(actorsCount);

	if (actorsCount) {
		mScene->getActors(PxActorTypeFlag::eRIGID_DYNAMIC,
			(PxActor**)&actorsList[0], actorsCount);

		for (PxRigidActor* actor : actorsList) {
			PxTransform transform = actor->getGlobalPose();
			setGlobalPosition(transform, position);
			setGlobalRotation(tranfsorm, rotation, angle);
		}
	}
}
```

Listing 4.12.5 Funkcja pomocnicza `setGlobalPosition`

```
void PhysX::getGlobalPosition(PxTransform transform,
	std::vector<glm::vec3>& position) {
	glm::vec3 pos;

	pos.x = transform.p.x;
	pos.y = transform.p.y;
	pos.z = transform.p.z;

	position.push_back(pos);
}
```

Listing 4.12.6 Funkcja pomocnicza `setGlobalRotation`

```
void PhysX::setGlobalRotation(PxTransform transform, std::vector<glm::vec3>&
	rotation, std::vector<GLfloat>& angle) {
	glm::vec3 rot;
	GLfloat ang;

	rot.x = transform.q.x;
	rot.y = transform.q.y;
	rot.z = transform.q.z;
	ang = transform.q.w;

	angle.push_back(ang);
	rotation.push_back(rot);
}
```

Żeby przygotowana przed chwilą funkcja zadziałała poprawnie trzeba jeszcze stworzyć odpowiednie kontenery w pliku *Source.cpp* (listing 4.12.7). Należy także narysować wszystkie sfery. OpenGL do rysowania wielu takich samych aktorów nie potrzebuje ich jako osobno zdefiniowanych obiektów klasy `Sphere`. Wystarczy w pętli nayrysować wiele razy jeden obiekt tej klasy, zmieniając za każdym razem położenie i na nowo przekazując wartości do Shaderów (4.12.8). Rezulat uruchomienia programu po wprowadzeniu powyższych zmian można zobaczyć na rysunku 4.12.2.

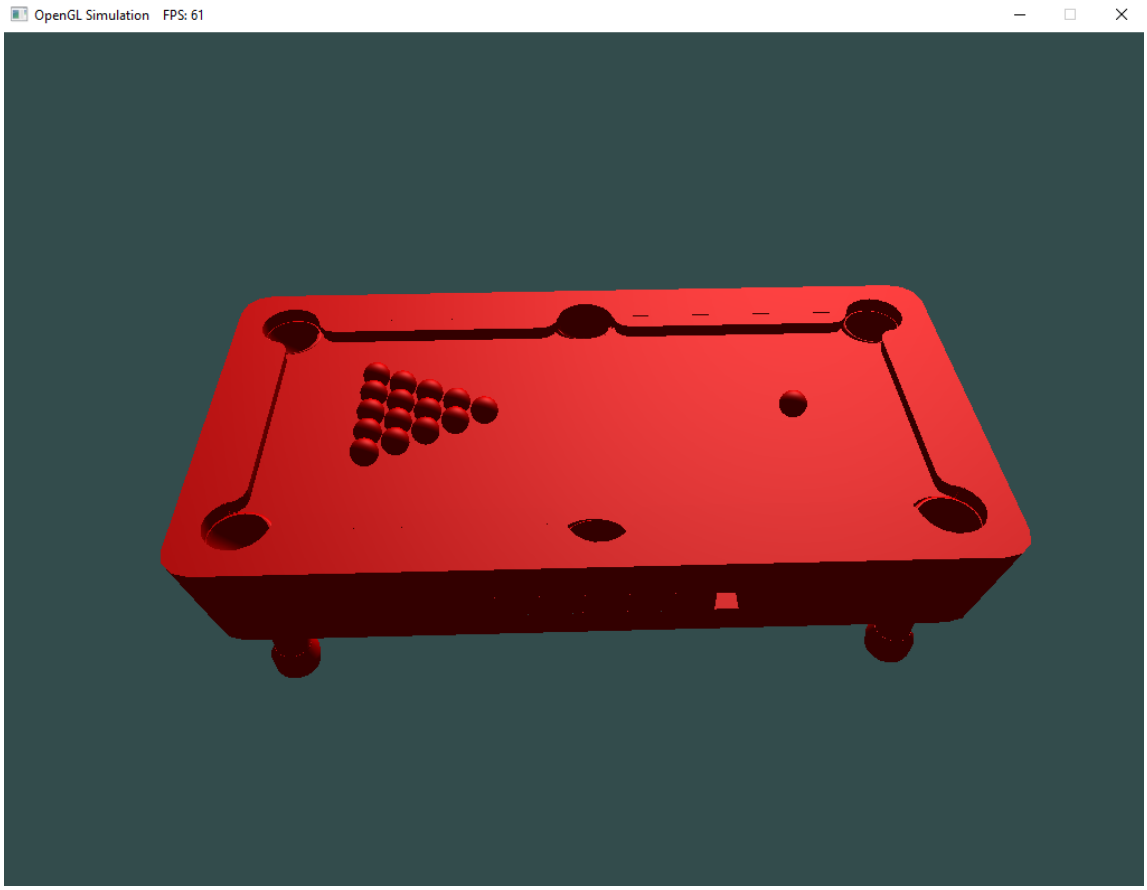
Listing 4.12.7 Przygotowanie wektorów translacji

```
std::vector<glm::vec3> position;
std::vector<glm::vec3> rotation;
std::vector <GLfloat> angle;

physicsObject.updateObjectsTransforms(position, rotation, angle);
```

Listing 4.12.8 Rysowanie wielu kul w głównej pętli programu

```
for (int i = 0; i < position.size(); i++) {
	camera.initModel(position[i], rotation[i], angle[i],
		glm::vec3(sphereRadius));
	camera.initMvp();

	glUniformMatrix4fv(modelLocation, 1, GL_FALSE, &camera.mModel[0][0]);
	glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &camera.mView[0][0]);
	glUniformMatrix4fv(projetionLocation, 1, GL_FALSE,
		&camera.mProjection[0][0]);
```

```
        sphere.draw();
}
```



Rys. 4.12.2 Stół bilardowy z kulami

Żeby zademonstrować, że fizyka jest w pełni podpięta do biblioteki OpenGL, trzeba stworzyć funkcję `pushWhiteBall`, która do przygotowanej wcześniej specjalnej kuli za pomocą funkcji `addForce` przyłoży siłęwskazaną przez podany w argumencie wektor siły (listing 4.12.9).

Listing 4.12.9 Funkcja `pushWhiteBall`

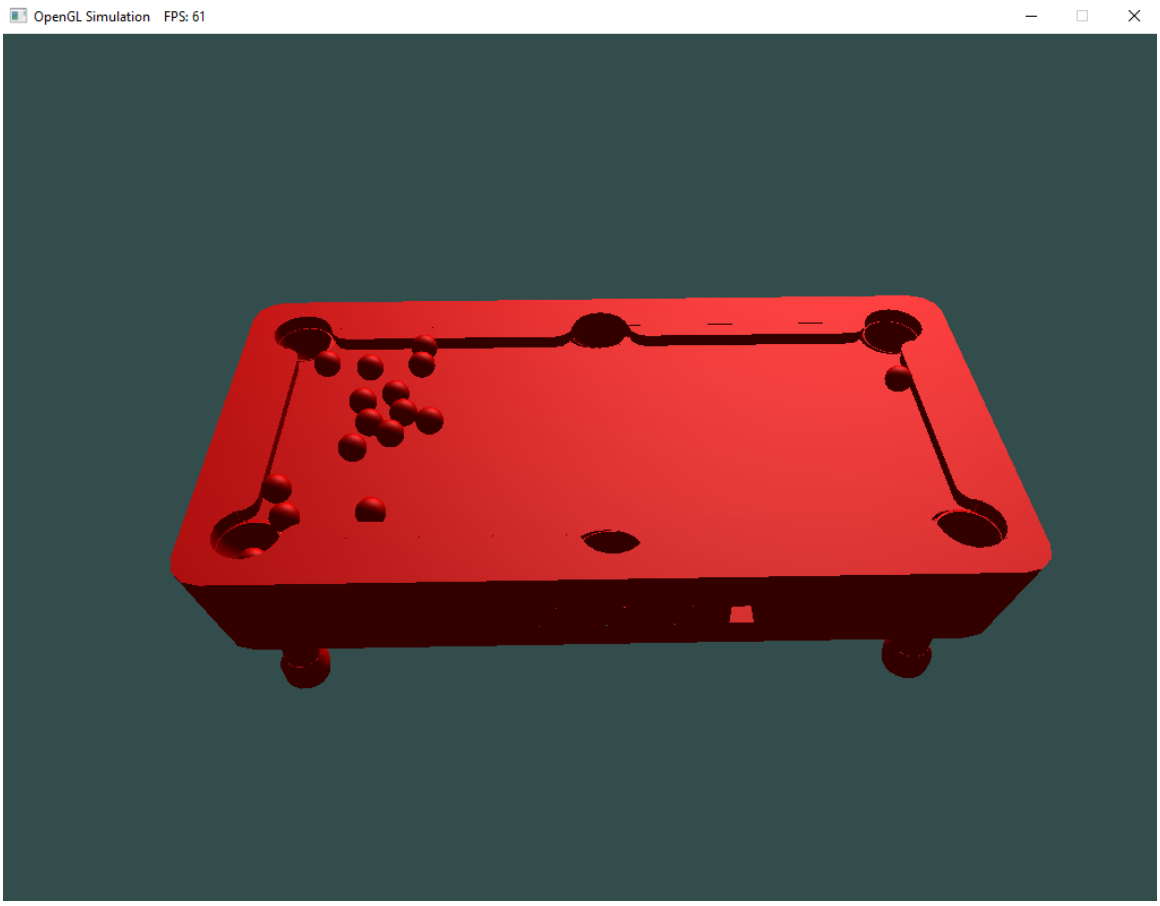
```
void PhysX::pushWhiteBall(PxVec3 forceVector) {
        mWhiteBall->addForce(forceVector);
}
```

Ostatnim krokiem jest wywołanie funkcji `pushWhiteBall`. Zostanie to zrobione przy pomocy lewego przycisku myszy. Żeby umożliwić takie działanie, trzeba dodać kolejną funkcję zwrotną odpowiedzialną za wyłapywanie używania przycisków myszy (4.12.10). W jej ciele wystarczy dodać instrukcję warunkową uruchamiającą funkcję

popychającą białą kulę przy kliknięciu lewym przyciskiem myszy. Gdy uruchomiony zostanie program, po kliknięciu lewego przycisku myszy, biała kula zostanie posłana w kierunku pozostałych kul. Do białej kuli można przykładać siłę wielokrotnie (zawsze w tym samym kierunku) do momentu, aż nie wpadnie ona do wgłębienia łuzy lub wypadnie poza stół (rys. 4.12.3).

Listing 4.12.10 Definicja i implementacja funkcji zwrotnej związanej z klikaniem myszą

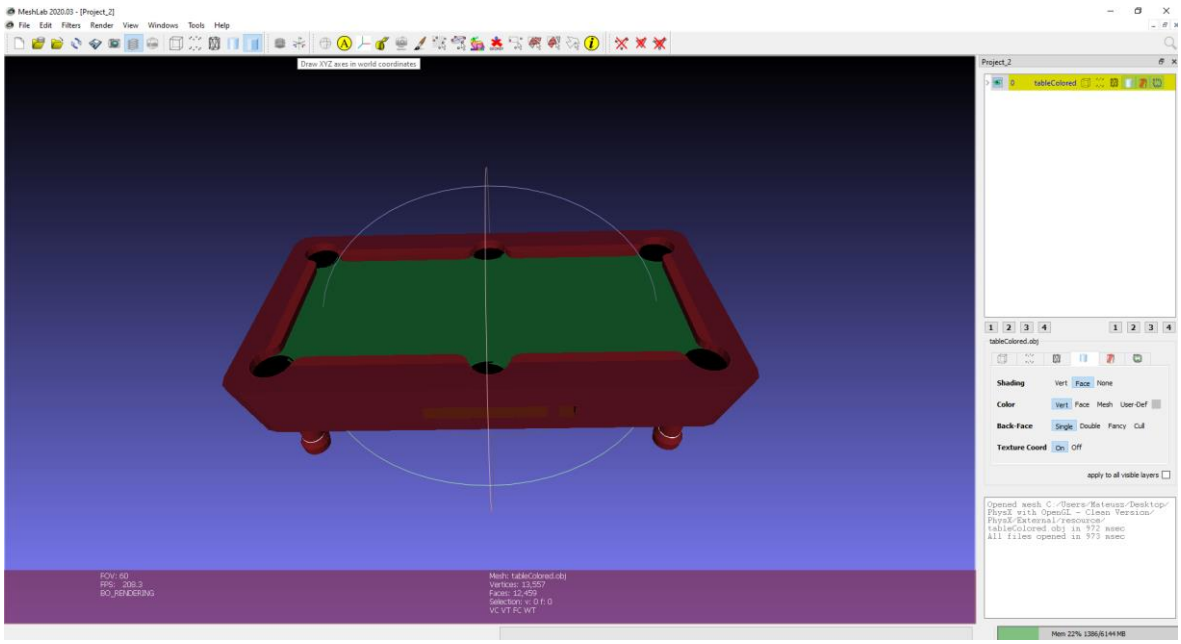
```
static void mouseButtonCallback(GLFWwindow* window, int button, int action,
	int mods);

static void mouseButtonCallback(GLFWwindow* window, int button, int action,
	int mods) {
	if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
		PxVec3 forceVector = PxVec3(-0.065f, 0.0f, 0.0f);
		physicsObject.pushWhiteBall(forceVector);
	}
}
```



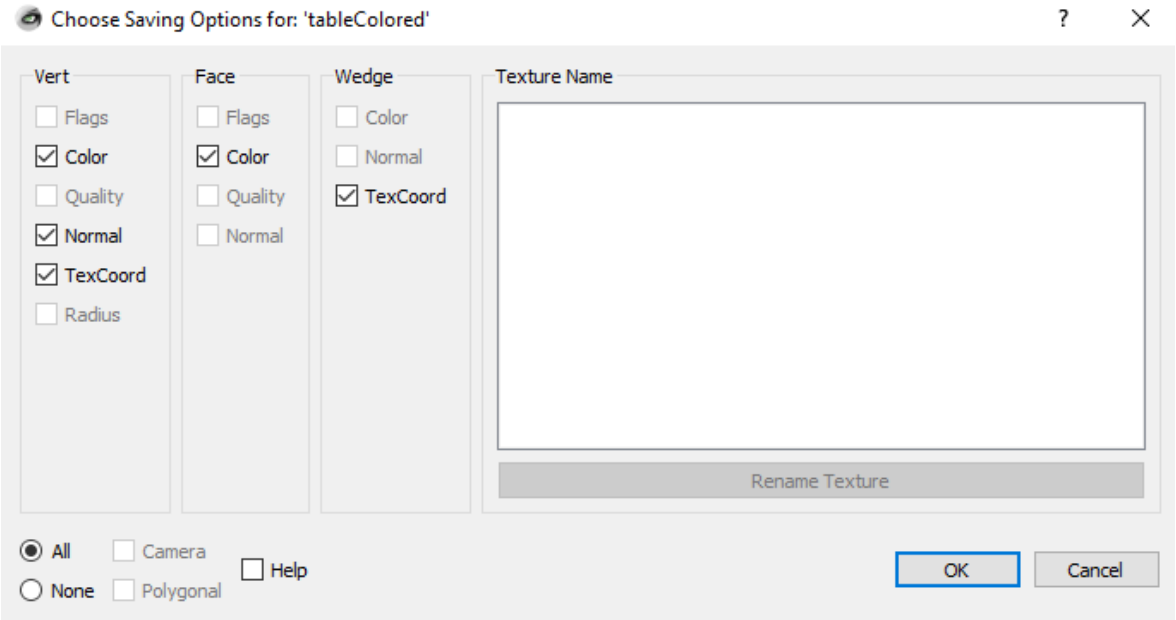
Rys. 4.12.3 Stół bilardowy z rozbitymi kulami

4.13 Dodanie koloru do obiektów

Ostatnim szlifem będzie dodanie koloru do poszczególnych elementów stołu oraz do kul. Na początek należy zaznaczyć, że formatu plików typu *wavefront* (pliki z rozszerzeniem *.obj*) nie uwzględnia kolorów w danych werteksów. Jego głównym przeznaczeniem jest przechowywanie położeń werteksów potrzebnych do poprawnego narysowania obiektu. Dodanie kolorów dla poszczególnych werteksów nie jest niemożliwe bez naruszenia specyfikacji. Można to jednak zrobić wykorzystując możliwości programu *MeshLab*. Najpierw trzeba ustalić kolory werteksów w zewnętrznym programie do edycji obiektów 3D. W programie *Blender* bardzo prosto można dodać kolor używając trybu *vertex paint*. Program ten nie wspiera jednak eksportowania plików typu *.obj* z werteksami kolorów. Aby obejść to ograniczenie, należy wyeksportować plik w formacie *Polygon File Format* (*.ply*), a następnie zaimportować go do pustego projektu w programie *MeshLab* (rys. 4.13.1). Przy wczytywaniu pliku, program ten obróci obiekt o pewien kąt. Na szczęście prostym filtrem o nazwie *Transform: Rotate* w zakładce *Normals, Curvatures & Orientation* można cofnąć taką operację. Pozostaje wyeksportować werteksy obiektu do pliku z rozszerzeniem *.obj*. Należy pamiętać, aby zaznaczone były wszystkie opcje widoczne na rysunku 4.13.2.



Rys. 4.13.1 Stół w porgramie *MeshLab*



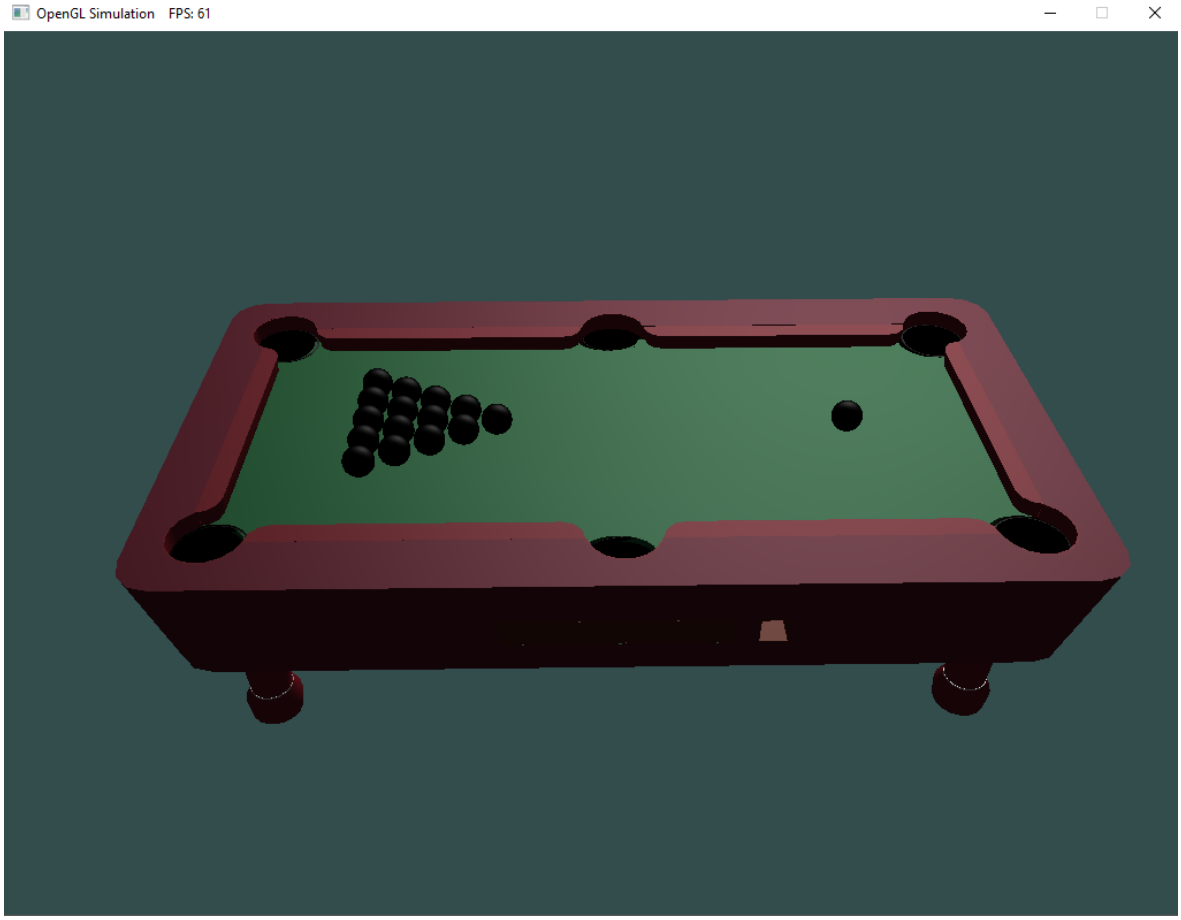
Rys. 4.13.2 Opcje potrzebne przy eksporcie pliku z programu *MeshLab*

Gdy obiekt jest już zapisany w pliku o odpowiednim formacie, można starać się go wczytać z kodu programu. Niestety przygotowana wcześniej klasa `ObjectLoader` nie wspiera werteksów z dodanymi kolorami. Trzeba więc zmienić typ pola `mVertices` na `std::vector<std::array<float, 6>>`. Następnie w funkcji `preparePosition` (listing 4.13.1) należy pierwsze trzy werteksy przekazać do bufora jako atrybut pozycji, a pozostałe trzy jako atrybut koloru.

Listing 4.13.1 Funkcja `preparePosition` ustawiająca pozycje i kolor

```
void Actor::preparePosition() {
	glGenBuffers(1, &mVbo);
	glBindBuffer(GL_ARRAY_BUFFER, mVbo);
	glBufferData(GL_ARRAY_BUFFER, mVertices.size() * 6 * sizeof(float),
		&mVertices[0], GL_STATIC_DRAW);

	glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
		(GLvoid*)0);
	glEnableVertexAttribArray(0);
	glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float),
		(GLvoid*)(3*sizeof(float)));
	glEnableVertexAttribArray(3);
}
```

Ostatnim krokiem potrzebnym do narysowania kolorowego stołu bilardowego jest uwzględnienie atrybutu koloru w shaderach. W shaderze werteksów wystarczy dodać nowe pola - `layout(location = 3) in vec3 color` oraz `out vec3 COLOR`, a następnie w funkcji `main` ustawić wartość pola `COLOR` na `color`. W shaderze fragmentów trzeba natomiast umożliwić wczytanie koloru z shadera werteksów za pomocą pola `in vec3 COLOR`, a następnie przypisać `COLOR` do zmiennej `materialDiffuseColor`. Po uruchomieniu tak zmodyfikowanego programu powinien pojawić się stół bilardowy z elementami o kolorach wczytanych z pliku(rys. 4.13.3).



Rys. 4.13.3 Kolorowy stół bilardowy

Na powyższym rysunku widać, że sfery mają jeszcze czarny kolor, ponieważ nie została im nadana żadna barwa. Kolor najlepiej dodać już podczas generacji werteksów (listing 4.13.2). Najłatwiejszym sposobem będzie przekazanie kolorów poprzez nowy argument `color` typu `std::array<float, 3>`.

Listing 4.13.2 Funkcja `generateVertices` z możliwością dodania koloru

```
void Sphere::generateVertices(GLint stacks, GLint sectors, std::array<float, 3>
color) {
	for (int y = 0; y <= sectors; ++y) {
		for (int x = 0; x <= stacks; ++x) {
			float xSegment = static_cast<float>(x) /
				static_cast<float>(stacks);
			float ySegment = static_cast<float>(y) /
				static_cast<float>(sectors);

			float xPos = static_cast<float>(std::cos(xSegment * 2.0*M_PI)
				* std::sin(ySegment * M_PI));
			float yPos = static_cast<float>(std::cos(ySegment * M_PI));
			float zPos = static_cast<float>(std::sin(xSegment * 2.0*M_PI)
				* std::sin(ySegment * M_PI));

			std::array<float, 6> triangle = { xPos, yPos, zPos, color[0],
				color[1], color[2] };

			mVertices.push_back(triangle);
			mUvs.push_back(glm::vec2(xSegment, ySegment));
			mNormals.push_back(glm::vec3(xPos, yPos, zPos));
		}
	}
}
```

Kiedy funkcja generująca werteksy jest już przygotowana, w pliku *Source.cpp* trzeba utworzyć metodę `createGlSpheres` (listing 4.13.3), która wypełni przekazany do niej pojemnik sferami. Tworzone sfery muszą być wskaźnikami typu `shared_pointer`, ponieważ metoda `push_back` dodaje do danego kontenera jedynie kopię przekazanego do niej obiektu. Oznacza to, że bufory oryginalnego obiektu po wyjściu z pętli zostają w destuktorze usuniętę, a w pojemniku znajdują się jedynie wskaźniki przechowujące nieaktualne już adresy obiektów [35].

Listing 4.13.3 Funkcja `createGlSpheres`

```
void createGlSpheres(vector<std::shared_ptr<Sphere>>& sphereContainer,
	GLint count, GLfloat stacks, GLfloat sectors,
	std::vector<std::array<float, 3>> colors) {
	for (GLint i = 0; i < count + 1; i++) {
		std::shared_ptr<Sphere> sphere = std::make_shared<Sphere>(stacks,
			sectors, colors[i]);
		sphereContainer.push_back(sphere);
	}
```

```
}
```

Kolory sfer zostaną ustalone w funkcji `createRandomSphereColors` (listing 4.13.4), która wygeneruje je wykorzystując funkcję `rand`. Specjalna kula została dodana jako ostatnia, więc na koniec wektora przechowującego kolory trzeba dodać kolor biały.

Listing 4.13.4Funkcja `createRandomSphereColors`

```
std::vector<std::array<float, 3>> createRandomSphereColors(GLfloat count) {
	std::vector<std::array<float, 3>> colors;
	std::array<float, 3> color;
	srand(time(NULL));

	for (int i = 0; i < count; i++) {
		for (int j = 0; j < 3; j++) {
			color[j] = static_cast<float>(rand() % 100) / 100.0f;
		}

	colors.push_back(color);
	}

	color = { 1.0f, 1.0f, 1.0f };
	colors.push_back(color);

	return colors;
}
```

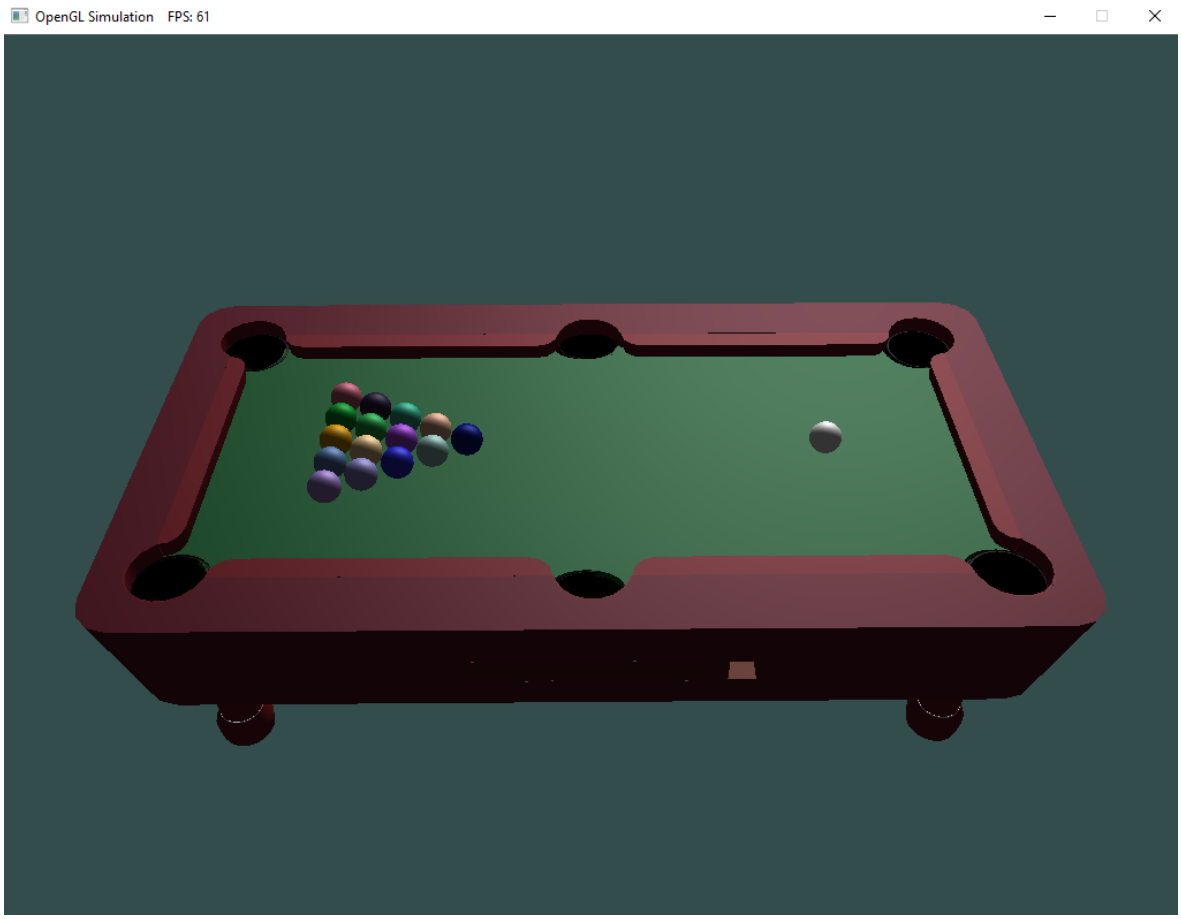
Pozostaje wywołać powyższe metody przed rozpoczęciem pętli głównej programu (listing 4.13.5) oraz zmienić sposób uruchamiania funkcji `draw` dla sfery (listing 4.13.6). Po uruchomieniu programu, pojawi się stół bilardowy z kulami o losowych kolorach (rys. 4.13.4).

Listing 4.13.5 Przygotowanie sfer w funkcji `main`

```
std::vector<std::array<float, 3>> colors = createRandomSphereColors(sphereCount);
vector<std::shared_ptr<Sphere>> sphereContainer;
createGlSpheres(sphereContainer, sphereCount,
	sphereStacks, sphereSectors, colors);
```

Listing 4.13.6 Wywołanie funkcji `draw` na pojedynczej sferze

```
sphereContainer[i]->draw();
```



Rys. 4.13.4 Kule bilardowe z losowymi kolorami

Podsumowanie

Moim celem było uruchomienie i zmuszenie do współpracy bibliotek PhysX i OpenGL, co było dość trudnym zadaniem inżynierskim, ale na szczęście wykonalnym. Praca dokumentuje moje wysiłki. Nadałem jej formę tutorialu, żeby ułatwić wykorzystanie materiałów przez kolejnych studentów, którzy chcieliby rozwijać projekty korzystające z tych dwóch bibliotek. Powstał dzięki temu sprawdzony poradnik, który wedle mojej wiedzy jest obecnie jedynym w języku polskim.

Bibliografia

[1] *Nvidia PhysX SDK Documentation*, Nvidia 2019, dokumentacją dotycząca PhysX dołączona do SDK

[2] *Nvidia PhysX API Documentation*, Nvidia 2019, dokumentacją dotyczącą API PhysX dołączona do SDK

[3] https://developer.nvidia.com/physx-sdk - [dostęp: 01.12.2019r.]

[4] Krishna Kumar, *Learning Physics Modeling with PhysX*, 2013

[5] https://www.opengl.org/about/ - [dostęp: 29.11.2019r.]

[6] https://www.khronos.org/about/ - [dostęp: 29.11.2019r.]

[7] https://www.khronos.org/about/directors-officers/ - [dostęp: 29.11.2019r.]

[8] https://opensource.com/resources/what-open-source - [dostęp: 29.11.2019r.]

[9] https://github.com/NVIDIAGameWorks/PhysX.- [dostęp: 25.11.2019r.]

[10] PhysX/physx/platform_readme.html

[11] https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/BuildingWithPhysX.html - [dostęp: 26.11.2019r.]

[12] https://stackoverflow.com/questions/58189023/physx-linking-problem-with-come-functions-imp-pxcreatebasephysics-reference - [dostęp: 27.11.2019r.]

[13] https://stackoverflow.com/questions/16717549/how-to-properly-link-and-include-libraries - [dostęp: 27.11.2019r.]

[14] https://docs.microsoft.com/en-us/cpp/build/reference/md-mt-ld-use-run-time-library?view=vs-2019 - [dostęp: 01.01.2020r.]

[15] https://www.techonthenet.com/c_language/directives/ifndef.php - [dostęp: 28.11.2019r.]

[16] https://www.learncpp.com/cpp-tutorial/6-7a-null-pointers/ - [dostęp: 28.11.2019r.]

[17] https://clean-swift.com/single-responsibility-principle-for-methods/ - [dostęp: 28.11.2019r.]

[18] Robert C. Martin, *Czysty Kod*, Helion, Gliwice 2014

[19] https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/VisualDebugger.html - [dostęp: 29.11.2019r.]

[20] https://dev.to/l04db4l4nc3r/the-dispatcher-4l4k - [dostęp: 02.01.2020r.]

[21] https://developer.nvidia.com/physx-visual-debugger - [dostęp: 26.04.2020r.]

[22] https://www.khronos.org/opengl/wiki/Depth_Test - [dostęp: 27.04.2020r.]

[23] https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview - [dostęp: 07.06.2020r.]
[24] https://stackoverflow.com/questions/2393429/should-i-always-use-gl-cull-face - [dostęp 27.04.2020r]
[25] https://learnopengl.com/Getting-started/Hello-Triangle - [dostęp: 28.04.2020r]
[26] J. Matulewski, *Grafika 3D czasu rzeczywistego*, PWN, Warszawa 2014
[27] https://learnopengl.com/Getting-started/Shaders - [dostęp: 30.04.2020r.]
[28] https://www.khronos.org/opengl/wiki/Shader_Compilation - [dostęp: 29.05.2020r.]
[29] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/ - [dostęp: 29.05.2020r.]
[30] https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/loading.php - [dostęp: 29.05.2020r.]
[31] https://open.gl/transformations - [dostep: 29.05.2020r.]
[32] https://learnopengl.com/Getting-started/Camera - [dostęp: 31.05.2020r.]
[33] https://learnopengl.com/Lighting/Basic-Lighting
[34] https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/Geometry.html - [dostęp: 10.05.2020r.]
[35] https://stackoverflow.com/questions/28929452/mesh-class-called-with-default-constructor-not-working-opengl-c - [dostęp: 02.06.2020r.]