



**Politechnika
Śląska**

PROJEKT INŻYNIERSKI

Edytor i symulator układów planetarnych

Paweł KUPCZAK
Nr albumu: 295679

Kierunek: Informatyka

Specjalność: Grafika Komputerowa i Oprogramowanie

PROWADZĄCY PRACĘ
dr inż. Michał Staniszewski

KATEDRA Grafiki, Wizji Komputerowej i Systemów Cyfrowych
Wydział Automatyki, Elektroniki i Informatyki

Gliwice 2024

Tytuł pracy

Edytor i symulator układów planetarnych

Streszczenie

Interaktywne symulacje pozwalają na eksplorację scenariuszy w zdefiniowanych układach planetarnych, często pozwalając na definicję takich układów przez użytkownika. Symulacje takie pozwalają analizować i oceniać rozwój układów z czasem oraz na interwencję w ich stan, dodając nowe planety lub modyfikując istniejące. Niniejsza praca opisuje aplikację, której zadaniem jest stworzenie środowiska do tworzenia i obserwowania układów stworzonych przez użytkownika, skupiając się na technikach używanych w programowaniu grafiki 3D w celu osiągnięcia konkretnych celów. Opisuje ona sposób symulacji układów oraz opisuje wszelkie techniki programowania 3D użyte w celu implementacji różnych rozwiązań wspomagających proces jej rozwijania, jak i samego użytkownika. Praca zawiera również elementy analizy wydajności tejże symulacji oraz samej aplikacji oraz proponuje dalsze, możliwe drogi jej rozwoju w celu polepszenia prezentacji graficznej i zwiększenia możliwości modyfikacji prezentacji graficznej planet.

Słowa kluczowe

OpenGL, Symulacja, Edytor

Thesis title

Planetary systems editor and simulator

Abstract

Interactive simulations allow for the exploration of scenarios in defined planetary systems, often enabling users to define such systems themselves. Such simulations enable the analysis and evaluation of the development of these systems over time and to interfere in their state by adding new planets or modifying existing ones. This paper describes an application whose purpose is to create an environment for creating and observing user-created systems, focusing on techniques used in 3D graphics programming to achieve specific goals. It describes the simulation process and describes various 3D programming techniques used to implement different solutions supporting the development process and user interaction. The paper also includes elements of performance analysis of the simulation and the application itself, proposing further possible paths for its development to enhance graphical presentation and increase the possibilities of modifying planets visual aspects.

Key words

OpenGL, Simulation, Editor

Spis treści

1 Wstęp	1
1.1 Cel pracy	1
1.2 Zakres pracy	2
1.3 Zakres tworzenia aplikacji	3
1.3.1 Projektowanie architektury	3
1.3.2 Projektowanie interfejsu graficznego	3
1.3.3 Implementacja	3
1.3.4 Testowanie	3
2 Analiza tematu	5
2.1 Przegląd głównej technologii	5
2.2 Istniejące rozwiązania	6
3 Wymagania i narzędzia	9
3.1 Wymagania funkcjonalne	9
3.2 Wymagania niefunkcjonalne	10
3.2.1 Wydajność	10
3.2.2 Stabilność	11
3.2.3 Dostępność	11
3.3 Opis narzędzi	11
3.3.1 Visual Studio Community 2022	11
3.3.2 GitHub	12
3.3.3 CMake	12
4 Specyfikacja zewnętrzna	13
4.1 Wymagania sprzętowe	13
4.2 Instalacja	14
4.2.1 Pobranie repozytorium	14
4.2.2 Budowana aplikacji	14
4.3 Przykład działania aplikacji	15

5 Specyfikacja wewnętrzna	21
5.1 Architektura systemu	21
5.1.1 Obsługa zdarzeń	21
5.1.2 Obsługa wejść	22
5.1.3 Aktualizacja stanu	22
5.1.4 Renderowanie	22
5.1.5 Obsługa jednostkowego kroku czasowego	23
5.2 Główne moduły	23
5.2.1 Główny moduł aplikacji	24
5.2.2 Warstwy i sceny	24
5.2.3 Zarządzanie teksturami	24
5.2.4 Fizyka/symulacja	25
5.2.5 Renderer	27
5.3 Wykorzystywane techniki	31
5.3.1 Oświetlenie	31
5.3.2 Wybieranie obiektów na scenie	35
5.3.3 Mapowanie wektorów normalnych	36
5.3.4 Struktura obiektów	38
5.3.5 Użyte biblioteki	39
6 Weryfikacja i walidacja	41
7 Podsumowanie i wnioski	45
Bibliografia	47
Spis skrótów i symboli	51
Spis rysunków	54
Spis tabel	55

Rozdział 1

Wstęp

W dzisiejszym świecie, zaawansowane narzędzia do tworzenia i symulacji układów planetarnych oferują niezrównane możliwości badania kosmosu. Jednak przedstawienie mojego projektu jako edytora i symulatora układów planetarnych to dla mnie nie tylko szansa na dostarczenie przydatnego narzędzia, ale także okazja do nauki i eksploracji różnych aspektów związanych z tworzeniem oprogramowania z dziedziny grafiki 3D.

Projekt inżynierski, który przedstawiam, to nie tylko próba stworzenia interaktywnej symulacji układów planetarnych, lecz także eksperyment w dziedzinie tworzenia edytora. Moim celem jest zgłębienie tajników pracy z grafiką 3D, interakcji z użytkownikiem oraz efektywności symulacji w czasie rzeczywistym. W kontekście tego wyzwania, podjąłem się nauki różnych technik, które często stosowane są w profesjonalnych silnikach do tworzenia aplikacji czasu rzeczywistego. Chciałbym zrozumieć, w jaki sposób takie silniki radzą sobie z różnymi problemami oraz jak dostosować te techniki do własnych potrzeb.

Niniejszy projekt ma nie tylko dostarczyć narzędzie do eksploracji kosmicznych układów planetarnych, lecz również stworzyć przestrzeń do eksperymentów i nauki, pozwalając mi zrozumieć, jakie wyzwania niesie ze sobą tworzenie zaawansowanych edytorów i symulacji.

1.1 Cel pracy

Przedmiotem niniejszej pracy jest opracowanie aplikacji desktopowej, dedykowanej do kreowania i symulacji układów planetarnych. Edytor ma za zadanie umożliwić manipulację różnymi właściwościami planet, takimi jak ich rozmieszczenie, parametry fizyczne czy wygląd, za pośrednictwem intuicyjnego interfejsu graficznego. W ramach edytora użytkownik będzie miał dostęp do stałego podglądu aktualnego stanu sceny oraz możliwość zapisywania i wczytywania stworzonych scen.

Symulacja powinna być realizowana w dedykowanej warstwie aplikacji, co umożliwi monitorowanie ewolucji układu w czasie. Użytkownik będzie miał możliwość dostosowywania listy parametrów, takich jak skala czasu, stała grawitacyjna G czy niektóre właściwości

planet. Dodatkowo, użytkownik będzie mógł zatrzymywać i wznowiać proces symulacji wedle własnych potrzeb.

1.2 Zakres pracy

Praca składa się z siedmiu rozdziałów, z których każdy skupia się na konkretnej części pracy inżynierskiej, lub samej aplikacji, która jest jej przedmiotem. Obecny rozdział „Wstęp” skupia się na:

- Wprowadzeniu czytelnika w motywacje, które kierowały przy wyborze tematu pracy.
- Postawieniu celów aplikacji oraz zwięzłym opisaniu, czym ona powinna być.
- Opisaniu poszczególnych rozdziałów pracy.
- Przybliżeniu zakresu tworzenia aplikacji.

Następny rozdział „Analiza tematu” opisuje dziedzinę, w której osadzony jest temat pracy oraz stawia go w niej. Opisana jest ogólna idea symulatorów, przedstawione są pewne wybory oraz istniejące rozwiązania, osadzając omawianą aplikację pomiędzy nimi.

Rozdział trzeci „Wymagania i narzędzia” stawia konkretne wymagania funkcjonalne oraz niefunkcjonalne, które powinny być spełnione w celu uznania aplikacji za kompletną. Oprócz tego opisane będą narzędzia wykorzystane do stworzenia aplikacji, które są też polecane w celu modyfikacji kodu źródłowego i budowy projektu we własnym zakresie.

Tematem rozdziału czwartego „Specyfikacja zewnętrzna” są kroki potrzebne w celu uruchomienia aplikacji, zaczynając od opisania wymagań sprzętowych, idąc w opis instalacji aplikacji w celu umożliwienia jej uruchomienia, kończąc na jej przykładowym działaniu.

Następnym rozdziałem jest „Specyfikacja wewnętrzna”, który jest też najbardziej obfitym. Jego celem jest skupienie się na technicznych aspektach aplikacji, omawiając jej wewnętrzną architekturę, jej najważniejsze części oraz opisując techniki wykorzystane w celu osiągnięcia postawionych celów.

Rozdziałem szóstym jest „Weryfikacja i validacja”, która opisuje kroki wykonane w celu upewnienia się, że cele aplikacji oraz jej funkcjonalności działają poprawnie.

Ostatni rozdział „Podsumowanie i wnioski” podsumowuje całą aplikację i stawia ją naprzeciwko postawionych celów i wymagań. Opisane są również dalsze plany dotyczące rozwoju aplikacji.

1.3 Zakres tworzenia aplikacji

1.3.1 Projektowanie architektury

W trakcie wcześniejszego planowania architektury, szczególnie istotnym aspektem jest identyfikacja konkretnych problemów, które mogą pojawić się w trakcie implementacji. Na przykład, wcześniejsze spojrzenie na architekturę z góry pozwala zidentyfikować potencjalne konflikty między modułami aplikacji, co umożliwia ich rozwiązanie jeszcze przed implementacją. To z kolei skraca czas, jaki byłby potrzebny na późniejsze naprawy, przyczyniając się do efektywniejszego przebiegu projektu. Jest to szczególnie ważne przy projektach większych, lub posiadających dużą ilość modułów.

1.3.2 Projektowanie interfejsu graficznego

Interfejs graficzny to kluczowy element każdej aplikacji, bezpośrednio kształtujący użytkownictwo i doświadczenie użytkownika. Użytkownik powinien być w stanie swobodnie poruszać się po interfejsie, łatwo zrozumieć funkcje, a całość powinna być spójna. Badania potwierdzają, że intuicyjny interfejs ma bezpośredni wpływ na retencję użytkowników i ich satysfakcję. Zbyt skomplikowany lub niespójny interfejs może prowadzić do frustracji, błędów użytkownika i obniżonej efektywności korzystania z aplikacji. Szczególnie wymagające są aplikacje o złożonym interfejsie, gdzie kluczowym wyzwaniem jest znalezienie odpowiedniego balansu między dostępnością a skomplikowaniem.

1.3.3 Implementacja

Faza realizacji architektury oprogramowania stanowi istotny moment w cyklu tworzenia, gdyż obejmuje konwersję projektu architektury w kod źródłowy, który ostatecznie będzie wykonywany. W tym etapie wprowadza się funkcjonalności zgodnie z projektem architektury oraz innymi założeniami. Proces implementacji obejmuje testowanie i optymalizację kodu w celu zapewnienia jego poprawności i efektywności. Podczas implementacji rozwiązywane są również bardziej szczegółowe problemy, które nie były widoczne na etapie planowania.

1.3.4 Testowanie

Główym celem testowania oprogramowania jako całości jest jednoznaczne określenie, czy każdy jego element działa zgodnie z oczekiwaniemi i spełnia stawiane mu wymagania. Przeprowadzane testy obejmują konkretne aspekty systemu, takie jak sprawdzanie poprawności wyników generowanych przez symulację fizyki, ocenę funkcjonalności oraz doświadczenia użytkownika w interfejsie użytkownika, czy też sprawdzenie spełnienia każdego wymaganego elementu. Przykładowo, możemy testować, czy system utrzymuje od-

powiednią efektywność. W trakcie tego etapu identyfikowane są potencjalne problemy, co umożliwia ich rozwiązanie.

Rozdział 2

Analiza tematu

Symulatory układów planetarnych dają możliwości, które nie byłyby możliwe do zrealizowania bez ich pomocy. Nasz wszechświat, z perspektywy kogoś na Ziemi, działa ze stałą jednostką czasową, a nasz wpływ na właściwości innych planet jest znikomy. Nikt nie jest w stanie sprawdzić w naszym świecie co by się stało, gdyby masa Słońca nagle spadła do masy Neptuna, albo jak drobna zmiana w prędkości innej planety wpłynie na cały układ. Niektóre rzeczy są możliwe do obliczenia, ale wynikiem będą same liczby, a czasami są niemożliwe do zwykłego przewidzenia.

Z pomocą interaktywnych symulatorów istnieje możliwość wizualizacji tego wszystkiego, wraz z pełną kontrolą całego układu. Użytkownik może wtedy zadawać sobie te wszystkie pytania, ale tym razem może szybko znaleźć na nie odpowiedź. Taki symulator może również dać użytkownikowi możliwość budowania własnych układów, z własnymi planetami, pozwalając na zwykłą zabawę siłami grawitacji w swoim wymyślonym układzie z nieistniejącymi planetami.

2.1 Przegląd głównej technologii

Symulatory fizyczne zazwyczaj nie są lekkie obliczeniowo, co nadaje pewne ograniczenie w wyborze technologii do zbudowania narzędzia.

Z tego powodu językiem wykorzystanym do napisania aplikacji jest C++, który jest znacznie szybszy od języków wyższego poziomu oraz kompilowany jest bezpośrednio do kodu maszynowego, zamiast być interpretowanym przez osobny program, co pozwala kompilatorowi ten kod dużo bardziej zoptymalizować.

Prezentacja danych graficznie potrafi być wymagająca, głównie przy dużej liczbie obiektów, przez co jest to ważny obszar aplikacji, nad którym należy się zastanowić. Istnieją biblioteki oferujące uproszczenie komunikacji z kartą graficzną, jednak często nie są one wyspecjalizowane, oraz posiadają systemy nie dające najlepszych rezultatów. Z tego powodu te wszystkie systemy postanowiono napisać od zera z wykorzystaniem API OpenGL, co daje absolutną kontrolę nad procesem generowania grafiki, a więc łatwiej jest

go wyspecjalizować pod nasze potrzeby.

2.2 Istniejące rozwiązania

Temat symulacji układów planetarnych był poruszany od dawna, a więc pojawiło się wiele aplikacji oferujących takie rozwiązania:

- Universe Sandbox 1/2 - jest to ogromny projekt tworzony od 2008 roku, którego celem jest jak najdokładniejsza i interaktywna symulacja. Skala symulowanej przestrzeni jest ogromna, z uwzględnieniem najdrobniejszych szczegółów, pozwalając na obserwację najmniejszych rzeczy (na przykład zmiany klimatyczne) oraz pewnych zdarzeń i ich skutków, na przykład zderzenie się dwóch planet lub supernova. Użytkownik może tworzyć własne układy planetarne, galaktyki a nawet wszechświaty. W celu symulowania tak ogromnej ilości ciał, program ten używa techniki symulacji N-ciały, która traktuje każde ciało osobno i determinuje siły, którymi działają na nie każde inne ciała - zdecydowana większość symulatorów używa tej techniki. [9] [13]
- SpaceEngine - celem tej aplikacji jest symulacja całego widzialnego wszechświata, używając do tego aktualnych danych astronomicznych oraz algorytmów proceduralnie generujących obszary poza obserwowalnym wszechświatem, wraz z terenem planet. [2] Program ten również pozwala na obserwację terenu planet, oferuje wiele bardzo ciekawych wizualnych efektów i sposobów na ich prezentację w bardzo kinematograficzny sposób. [12]
- AstroGrav - zaawansowany symulator grawitacyjny, który pozwala na modelowanie układów planetarnych, gwiazdnych i galaktycznych. Program oferuje precyzyjne narzędzia do symulacji ruchu wielu ciał grawitacyjnych jednocześnie. Skupia się on bardziej na zaawansowanej analizie i modelowaniu układów grawitacyjnych, przez co nie jest on tak przystępny jak Universe Sandbox. Także wykorzystuje on symulację N-ciały, również pomagając w obliczeniach trajektorii ruchu obiektów za pomocą algorytmów integracji numerycznej, takich jak metoda Verleta [5] czy metoda Runge-Kutta. [1] [10]
- Celestia - darmowy i otwartoźródłowy symulator kosmiczny, który umożliwia interaktywną eksplorację symulowanego wszechświata. Skupia się bardziej na proceduralnej generacji kosmosu i eksploracji, ale również jest używany w celach edukacyjnych i badawczych, jak i miłośników astronomii, oferując też podgląd obecnego układu z Ziemi, z której użytkownicy mogą obserwować utworzone przez siebie scenariusze w kosmosie. Użytkownicy mogą również pisać własne wtyczki do tej aplikacji, otwierając drogę do bardzo spersonalizowanego doświadczenia. [11]

Aplikacja omawiana w tej pracy stawia na większą prostotę, nie symulując całego wszechświata, ale skupiając się na idei prostego modelowania układów i podstawowej fizyki. Również z czasem (niekoniecznie w wersji obecnie omawianej) aplikacja ta będzie celowała w stanie się niejako silnikiem graficznym wyspecjalizowanym do sfer, co w efekcie da dużo więcej możliwości tworzenia indywidualnego doświadczenia pod kątem wizualnym, czego powyższe narzędzia nie oferują w pełni.

Rozdział 3

Wymagania i narzędzia

3.1 Wymagania funkcjonalne

Wymagania funkcjonalne opisują zachowania systemu, które powinny być zaimplementowane w celu stworzenia oczekiwanej wersji aplikacji. Wymogiem funkcjonalnym jest jedna (lub kilka połączonych) właściwość systemu, które są potrzebne użytkownikowi do poprawnego użytkowania aplikacji. Poniższa lista przedstawia pożądane funkcjonalności aplikacji:

1. Edytor

- Użytkownik może dodać planetę lub gwiazdę za pomocą przycisków.
- System współrzędnych można zmienić za pomocą listy wyboru.
- Opcja zorientowania kamery na jednej z trzech głównych osi jest dostępna pod trzema przyciskami.
- Dostępne są kontrolki typu „toggle” do przełączania siatki na scenie oraz skybox'a.
- Nazwa sceny jest możliwa do edycji poprzez interfejs graficzny.
- Widoczna jest lista wszystkich obiektów istniejących na scenie.
- Kliknięcie na obiekt w takiej liście ustawia go jako „wybrany”.
- Kliknięcie na obiekt w oknie sceny również ustawia go jako „wybrany”.
- Na wybranym obiekcie można pokazać tzw. „gizmos”, służące do manipulacji translacją, rotacją oraz skalą obiektu.
- Gdy obiekt jest wybrany, zostają pokazane jego właściwości, które można modyfikować.
- Właściwości numeryczne można modyfikować poprzez przesuwanie pola edycji, lub jej wpisanie.

- Właściwości tekstowe można edytować poprzez interfejs graficzny.
- Właściwości będące związanymi z plikami (tekstury) można wgrywać, poprzez wciśnięcie przycisku.
- Kolory można edytować poprzez „color picker”, lub wpisanie wartości.
- Zaznaczoną planetę można usunąć wciskając klawisz Delete.
- Planetę można odznaczyć poprzez zaznaczenie innej planety, kliknięcie na tło sceny lub wciśnięcie klawisza Escape.
- Wystawione są właściwości kamery do edycji: pozycja, pole widzenia, bliska / daleka płaszczyzna odcinania.
- Dostępny jest przycisk przechodzący w tryb symulacji.
- Scenę można zapisać, lub wgrać scenę z dysku, za pomocą dostępnych przycisków.

2. Symulacja

- Symulację można wstrzymać lub wznowić za pomocą przycisków.
- Z symulacji można wrócić do edytora za pomocą przycisku.
- Ustawienia symulacji są dostępne po wciśnięciu przycisku.
- W symulacji możliwe do modyfikacji są: stała grawitacyjna G (jej skalar), skalar czasu (prędkość symulacji), oraz możliwe do podglądu są: czas trwania symulacji (ile czasu symulacja działa) i upływ czasu „w symulacji” (tzn. ile według symulacji minęło czasu).
- Obiekty, tak jak w edytorze, można wybrać poprzez kliknięcie na nie.
- Po wybraniu obiektu pojawia się okno z informacjami na jego temat.
- Do wybranego obiektu można się „przyczepić”, wtedy kamera będzie się na ten obiekt zawsze spoglądać.

3.2 Wymagania niefunkcjonalne

Wymagania niefunkcjonalne definiują ogólne oczekiwania dotyczące systemu, koncentrując się na ogólnych cechach, zamiast konkretnych aspektów, które są uwzględnione w wymaganiach funkcjonalnych.

3.2.1 Wydajność

Z uwagi na charakter aplikacji, jest ona podatna na spowolnienia związane z symulacją, ponieważ każdy obiekt oddziałuje z każdym innym, co skutkuje wzrostem kwadratowym

w miarę dodawania kolejnych obiektów. Dlatego też program korzysta z wielowątkowości do obliczeń sił grawitacyjnych między planetami, umożliwiając efektywne wykorzystanie zasobów użytkownika w celu maksymalnego przyspieszenia tych obliczeń, realizując je równocześnie.

Dodatkowo, moduł Renderera został zoptymalizowany pod kątem efektywnego przedstawiania dużej ilości obiektów, minimalizując czas potrzebny na komunikację i interakcję z kartą graficzną.

3.2.2 Stabilność

Aby zachować stabilność symulacji, konieczne jest, aby każdy krok symulacji był wykonywany w stałym interwale lub, innymi słowy, określonej liczbie razy na sekundę. Proste uzależnienie ruchu planet od czasu między kolejnymi klatkami aplikacji może okazać się niewłaściwe, gdyż nawet najmniejsze różnice mogą generować znaczne skutki w odpowiednio długim okresie czasu. Różnice wynikające z tego, czy aplikacja działa z częstotliwością 60 razy na sekundę, czy 600 razy na sekundę, mogą być istotne. W związku z tym w tej aplikacji zastosowano system TPS (Ticks Per Second), który zapewnia, że niezależnie od prędkości działania aplikacji, fizyka jest aktualizowana dokładnie N razy na sekundę. To gwarantuje spójność wyników zarówno dla aplikacji pracujących szybciej, jak i wolniej.

3.2.3 Dostępność

Aplikacja została zaprojektowana z myślą o łatwości obsługi i przejrzystości, umożliwiając użytkownikowi intuicyjne poruszanie się po interfejsie graficznym. Z tego względu interfejs jest prosty, pozbawiony ukrytych opcji, dający do osiągnięcia minimalistycznego i funkcjonalnego charakteru.

3.3 Opis narzędzi

3.3.1 Visual Studio Community 2022

Visual Studio, stworzone przez Microsoft, to wszechstronne środowisko programistyczne, z głównym naciskiem na języki C++ oraz C#. Zapewnia ono kompleksowe narzędzia, obejmujące edytor tekstu, środowisko do budowania aplikacji, system projektów oraz narzędzia do debugowania. Dodatkowo, użytkownicy korzystają z wielu wygodnych funkcji, takich jak podświetlanie składni, „IntelliSense” pomagające w trakcie pisania kodu, a także integracja z systemem Git. Ogromnym atutem jest także bogaty ekosystem rozszerzeń, opracowywanych zarówno przez Microsoft, inne firmy, jak i niezależnych deweloperów, co umożliwia dostosowanie środowiska do indywidualnych potrzeb. Dodatkowo, warto zauważyć, że wersja Community jest dostępna bezpłatnie.

3.3.2 GitHub

Jest to jedna z najpopularniejszych usług hostingowych dla repozytoriów Git, które pozwalają na łatwą wspólną pracę nad projektem i śledzeniem zmian, co pozwala na łatwe przełączanie się między różnymi wersjami repozytorium. Kontrola wersji jest bardzo ważnym elementem pracy nad projektem, zwłaszcza gdy pracuje nad nim kilka lub więcej osób.

3.3.3 CMake

Nawet pomimo tego, że środowisko Visual Studio dostarcza swoje własne narzędzia do zarządzania projektami i budowania, CMake wyróżnia się jako standardowe, wieloplatformowe rozwiązanie do generowania plików projektu. To narzędzie daje użytkownikom ogromną elastyczność, umożliwiając nie tylko tworzenie projektów w środowisku Visual Studio, ale także, na przykład, umożliwia użytkownikom systemu Linux skomplikowane procesy budowania poprzez system Unix Makefile. Dzięki temu CMake staje się uniwersalnym narzędziem, znakomicie dostosowującym się do różnorodnych potrzeb programistycznych i preferencji systemowych.

Rozdział 4

Specyfikacja zewnętrzna

4.1 Wymagania sprzętowe

W zależności od złożoności algorytmów i rozmiaru danych występujących w programie, wymagania sprzętowe mogą rozciągać się od praktycznie zerowych, do takich, które wymagają bardzo dobrego i drogiego sprzętu. Mówiąc o minimalnych wymaganiach, najczęściej punktuje się:

- Procesor.
- Pamięć RAM.
- Ilość wolnego miejsca na dysku.
- Karta graficzna.
- System operacyjny.

W przypadku tej aplikacji - nie posiada ona szczególnych wymagań minimalnych, jednak należy wspomnieć o kilku punktach:

- Aplikacja powinna działać dobrze z każdym w miarę standardowym procesorem, jednak przy scenie z dziesiątkami, albo i setkami, obiektów, starsze procesory mogą nie nadążyć z ilością obliczeń.
- W zależności od rozmiaru i ilości używanych tekstur, zużycie pamięci RAM oraz VRAM może nagle wzrastać - pamięci RAM oraz VRAM rozmiaru 4GB będą wystarczające dla większości przypadków.
- Karta graficzna, lub zintegrowany moduł graficzny, musi obsługiwać standard OpenGL 4.3 lub wyżej. Jest to jedyny „twardy” wymóg.

4.2 Instalacja

4.2.1 Pobranie repozytorium

W pierwszym kroku, należy pobrać kod źródłowy z repozytorium za pomocą narzędzia Git. Wykonać to można poprzez graficzne klienty Git, lub za pomocą polecenia:

```
git clone --recursive https://github.com/komorxd/SolarSystemSim.git
```

Pobierze ono repozytorium do katalogu SolarSystemSim.

4.2.2 Budowana aplikacji

Budowa aplikacji różni się w zależności od systemu operacyjnego:

1. Windows

Do zbudowania aplikacji wystarczy środowisko programistyczne Visual Studio, z narzędziami do pracy z systemem CMake. Repozytorium można wtedy otworzyć z środowisku, zbudować konfigurację CMake i zbudować sam program, jak każdy inny projekt.

2. Linux

W tym przypadku, należy się upewnić, czy użytkownik posiada kompilator GCC / G++, oraz aplikację CMake. Jeżeli tak, to wystarczy uruchomić skrypt *build.sh* z odpowiednim argumentem, którym jest jeden z ciągów znaku:

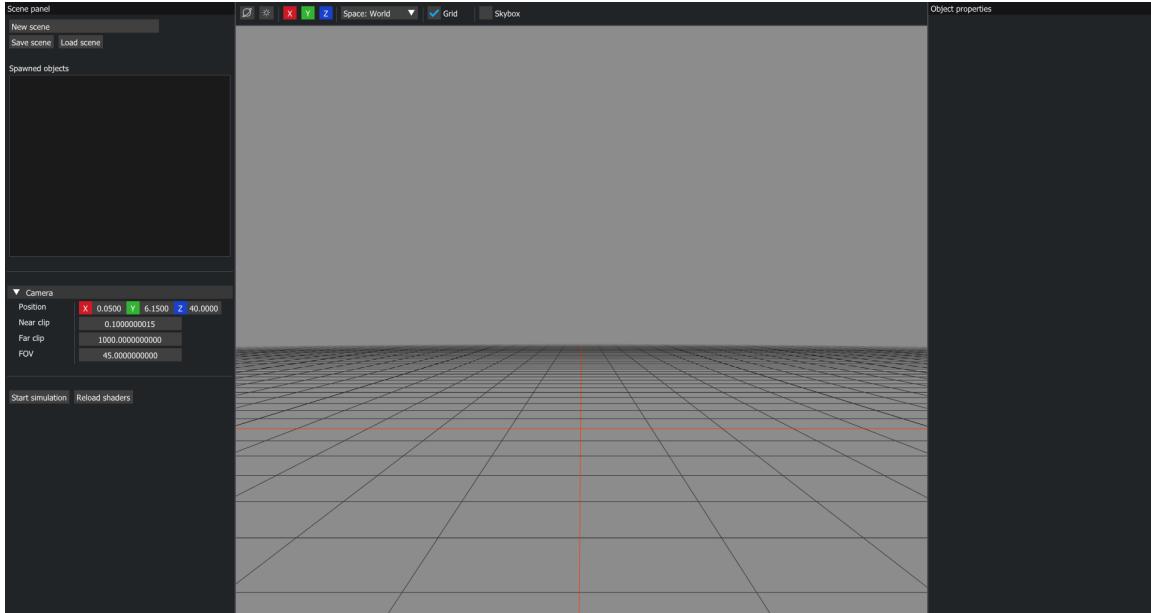
- „debug” / „d”.
- „release” / „r”.
- „prod” / „p”.

Argument ten odpowiada za ustawienie konfiguracji aplikacji - wersja do użytkowania to wersja „prod”.

Po zbudowaniu projektu, plik wykonywalny aplikacji umieszczany jest w jednym z trzech katalogów, zależnie od konfiguracji komplikacji. Struktura katalogów przyjmuje format „out/bin/x86-64-Konfiguracja-System operacyjny”. Na przykład, dla konfiguracji „release” na systemie Windows, ścieżka do pliku wykonywalnego to „out/bin/x86-64-Release-Windows”.

4.3 Przykład działania aplikacji

Po uruchomieniu pliku wykonywalnego, pojawi się edytor z pustą sceną, widoczny na rysunku poniżej:

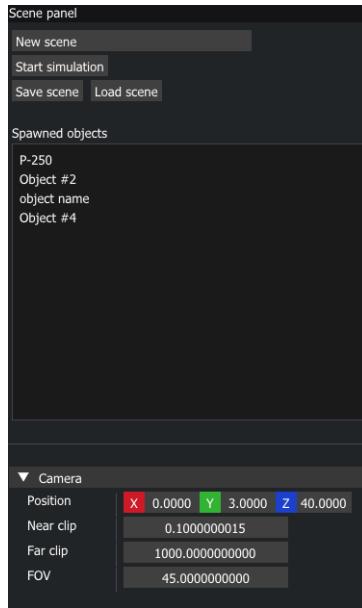


Rysunek 4.1: Edytor z pustą sceną

Widok edytora składa się z 4 komponentów:

- panel sceny (lewy).
- górny pasek.
- panel obiektu (prawy).
- widok sceny.

Panel sceny znajduje się na lewej części okna. Znajdują się na nim kontrolki odpowiedzialne za zarządzanie sceną, ale nie jej konkretnymi obiekty. Posiada on przyciski do zapisania obecnej sceny oraz załadowania sceny zapisanej na dysku. Pod nimi znajduje się lista obiektów istniejących na scenie, które można poprzez tę listę zaznaczyć. Dalej modyfikować można własności kamery, głównie pole widzenia, w celu modyfikacji prezentacji.



Rysunek 4.2: Panel sceny

Górny pasek jest na górze okna i zawiera on kilka narzędzi. Głównymi są dwa pierwsze - dodatnie nowej planety oraz dodanie nowej gwiazdy. Po wciśnięciu któregoś z nich, na scenie pojawi się nowy obiekt w środku układu. Następne trzy przyciski pozwalają na zorientowanie kamery równolegle do wybranej osi układu współrzędnych względem wybranego obiektu (w przypadku gdy żaden obiekt nie jest wybrany, wybrane osie są osiami układu współrzędnych). Pozwala to na przykład na szybkie ustawienie kamery bezpośrednio nad obiektem. Następnie do wyboru są dwa tryby gizmos - lokalne i globalne. Na przykładzie operacji translacji, oznacza to, że przy ustawieniu globalnym „w górę” zawsze będzie oznaczać prosto w górę, a przy ustawieniu lokalnym - będzie to kierunek „w górę” po rotacji.

Ostatnie dwie kontrolki pozwalają nam na przełączanie skybox'a oraz siatki - w celu wizualizacji wyglądu sceny bez konieczności włączania symulacji.



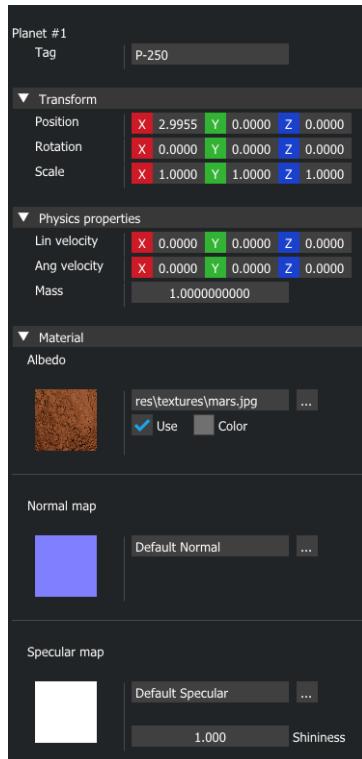
Rysunek 4.3: Górnny pasek

Panel obiektu znajduje się po prawej i są na nim widoczne poszczególne komponenty wybranego obiektu, które użytkownik może modyfikować:

- Tag - jest to nazwa obiektu, ta samą którą można zaobserwować na liście obiektów na panelu sceny.
- Transform - reprezentacja macierzy definiującej przekształcenie obiektu. Jest to jego pozycja, rotacja oraz skala.

- Własności fizyczne - wartości istotne dla samej symulacji, a więc głównie masa obiektu i jego prędkość.
- Materiał - informacje o wyglądzie obiektu. Można poprzez przyciski wgrać specjalne tekstury dla obiektu, wybrać jego kolor oraz siłę odbitego światła.

Za pomocą tego panelu, użytkownik może z łatwością modyfikować obiekty zgodnie z jego upodobaniami, oraz nadawać im docelowe wartości masy i prędkości.



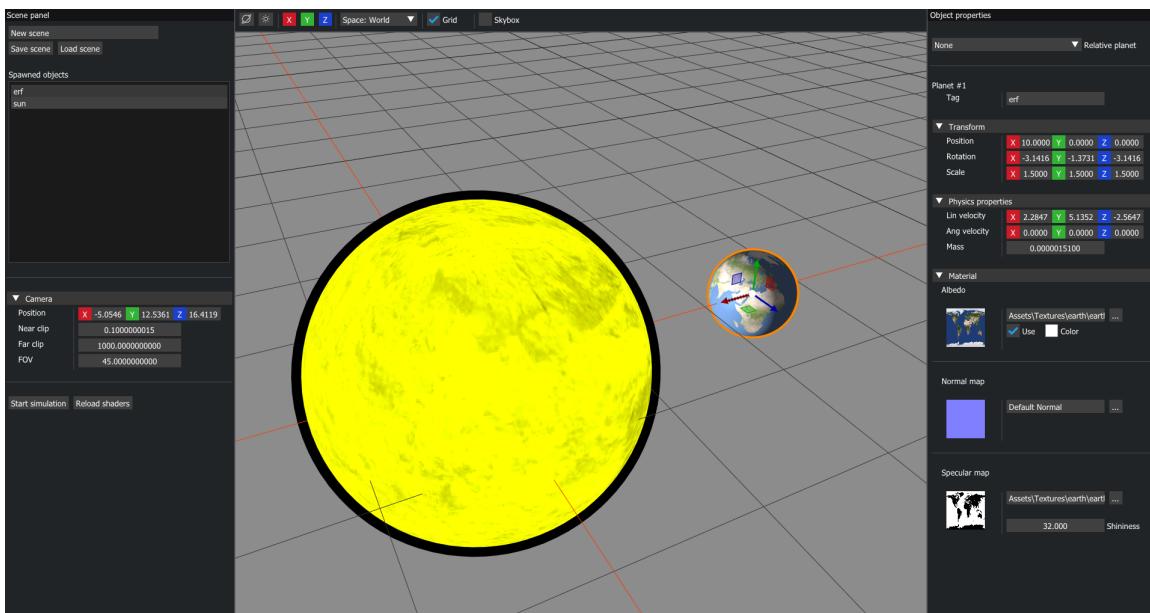
Rysunek 4.4: Panel obiektu

Widok sceny reprezentuje obecny stan obiektów na scenie. Obiekt można wybrać poprzez kliknięcie na niego lewym przyciskiem myszy - wówczas zmieni się kolor obramowania wybranego obiektu. Aplikacja obsługuje tak zwane „gizmos”, które pozwalają na bezpośrednią manipulację komponentem Transform. Istnieją jego cztery tryby:

- Brak - dostępny pod klawiszem Q, jest to brak gizmos.
- Translacja - dostępna pod klawiszem W, manipuluje pozycją obiektu.
- Rotacja - dostępna pod klawiszem E, manipuluje rotacją obiektu.
- Skala - dostępna pod klawiszem R, manipuluje skalą obiektu.

Trzymając prawy przycisk myszy, możliwe jest obracanie kamery, a trzymając środkowy przycisk - przesuwanie kamery względem płaszczyzny zdefiniowanej przed jej kąty

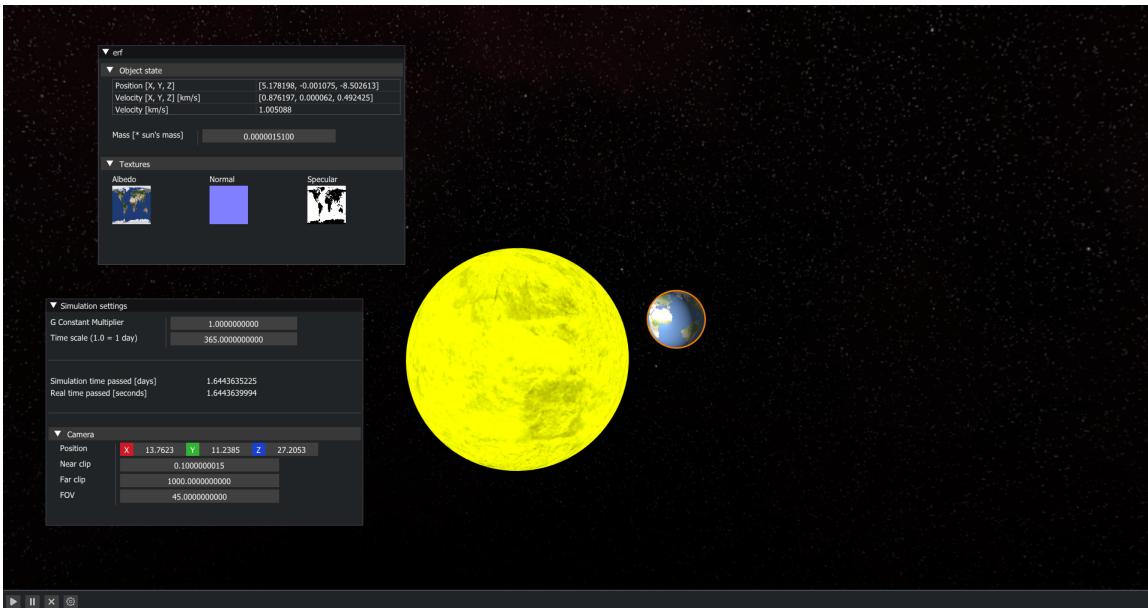
widzenia. Mając wybrany obiekt, za pomocą klawisza F można przesunąć się blisko, na wprost tej planety. Te opcje dają użytkownikowi więcej opcji poruszania się po scenie i interakcji z nią, co przyczynia się do wygodniejszego i wydajniejszego użytku. Na poniższym rysunku przedstawiona jest scena z dodanymi obiektami:



Rysunek 4.5: Edytor z wypełnioną sceną

W tym etapie został utworzony układ słoneczny z centralną gwiazdą, nadając każdej z nich odpowiednie masy i prędkości. Dla dodatkowej atrakcyjności, na planetę nałożone zostały tekstury, nadając jej bardziej realistyczny wygląd.

Kolejnym krokiem jest uruchomienie symulacji poprzez naciśnięcie przycisku dostępnego na panelu sceny. Wówczas użytkownik może śledzić dynamiczny rozwój układu wraz z upływem czasu, oraz zmieniać konfigurację samej symulacji.



Rysunek 4.6: Widok symulacji

Na końcu z symulacji można wrócić do edytora za pomocą przycisku na dolnym pasku, zapisać scenę i tworzyć dalszy układ.

Rozdział 5

Specyfikacja wewnętrzna

5.1 Architektura systemu

Aplikacja opiera się na architekturze zorientowanej wokół koncepcji głównej pętli, co oznacza, że kluczowym elementem jest ciągły cykl operacji, który obejmuje kolejno:

- obsługa zdarzeń.
- obsługa wejść.
- aktualizacja stanu.
- renderowanie.

Jeden taki cykl nazywany jest „klatką”. Poza kolejnością, występuje również obsługa jednostkowego kroku czasowego. Główna pętla stanowi rdzeń aplikacji oraz umożliwia płynne działanie aplikacji w czasie rzeczywistym, co jest istotnym aspektem dla interaktywnego projektu.

5.1.1 Obsługa zdarzeń

Obsługa zdarzeń koncentruje się na zdarzeniach związanych z oknem aplikacji, które trafiają do kolejki zdarzeń. Do najczęściej występujących zdarzeń zaliczają się:

- Wciśnięcie, puszczenie lub trzymanie klawisza na klawiaturze.
- Poruszanie kursem myszy w obszarze okna aplikacji.
- Wciśnięcie lub puszczenie przycisku myszy.

W każdej klatce działania aplikacji, kolejka zdarzeń jest czyszczona, a zdarzenia przekazywane są kolejno do aktualnej warstwy aplikacji. Warstwa ma możliwość obsługi zdarzenia lub przekazania go dalej w strukturze, na przykład do warstwy sceny. Ten proces umożliwia elastyczną reakcję na zdarzenia, dostosowaną do bieżącego stanu aplikacji.

5.1.2 Obsługa wejść

Obsługa wejść skupia się na monitorowaniu aktualnego stanu konkretnych klawiszy klawiatury oraz przycisków myszy, w zależności od bieżącego kontekstu aplikacji. Warto wyraźnie odróżnić tę funkcję od obsługi zdarzeń. Na przykład, w przypadku zdarzeń informacja o wcisnięciu klawisza zostanie przekazana raz - w klatce, w której zdarzenie to wystąpiło. Natomiast w obsłudze wejść sprawdzamy jedynie, czy dany klawisz w danym momencie jest wcisnięty, bez informacji o konkretnym momencie zdarzenia i jego powtarzalności.

Aby to zobrazować, przyjrzyjmy się przykładowi: podczas pisania na klawiaturze, gdy przytrzymamy klawisz, zostanie wypisana litera. Po chwili przerwy, jeśli dalej trzymamy klawisz, litera będzie ciągle się powtarzała. To są właśnie zdarzenia - najpierw zdarzenie wcisnięcia przycisku, potem zdarzenie trzymania przycisku. W przypadku sprawdzania stanu wejścia, nie wystąpiłaby ta chwila przerwy, co utrudniłoby wpisywanie tekstu.

W kontekście aplikacji czasu rzeczywistego, ta subtelna różnica jest kluczowa. Na przykład, gdy chcemy poruszyć postacią w grze komputerowej w lewo, nie pragniemy tej krótkiej przerwy, którą obserwujemy przy pisaniu tekstu. Wprowadzenie tekstu wymaga właśnie tej przerwy, co jest istotne, szczególnie w przypadku interaktywnych scenariuszy.

5.1.3 Aktualizacja stanu

Proces aktualizacji stanu gry obejmuje wszystkie komponenty wymagające uwagi w każdej klatce, a często musi być dostosowany do prędkości działania aplikacji, wybranej w ilości klatek na sekundę. Przykładem może być obsługa kamery, gdzie wszystkie operacje związane z ruchem są wykonywane w każdej klatce. Ruch ten jest skalowany z uwzględnieniem czasu między kolejnymi klatkami aplikacji, co efektywnie definiuje zmianę na sekundę. Taki zabieg ma na celu stworzenie niezależności komponentów od prędkości aplikacji, zapewniając spójne doświadczenie na różnych maszynach.

5.1.4 Renderowanie

Po obsłudze wszystkich poprzednich kroków istnieje nowa, zaktualizowana wersja sceny, którą należy pokazać użytkownikowi w oknie aplikacji. Dzieje się to w kroku renderowania, gdzie czyszczona jest obecna zawartość okna, rysowany jest cały interfejs graficzny, cała scena oraz znajdującej się na niej obiekty.

Renderowanie to złożony proces, obejmujący różne elementy, takie jak przygotowanie widoku kamery, oświetlenie, czy też renderowanie obiektów z uwzględnieniem ich właściwości graficznych. W tym etapie każdy piksel na ekranie jest precyzyjnie kalkulowany, a ostateczny rezultat to dynamiczna, zaktualizowana wizualizacja całej sceny.

Warto również zauważyc, że renderowanie jest kluczowym elementem gier i aplikacji graficznych, wpływając bezpośrednio na wrażenia użytkownika. Dbałość o efektywność tego procesu jest istotna, szczególnie w kontekście płynności działania aplikacji.

5.1.5 Obsługa jednostkowego kroku czasowego

Jak wcześniej wspomniano, poza kolejnością wszystkich poprzednich kroków, występuje również aktualizacja związana ze zdefiniowanym krokiem czasowym aplikacji. Jest ona wykonywana w osobnym wątku, konkretną ilość razy na sekundę. Ten krok zajmuje się tymi samymi rzeczami, które mogłyby być obsłużone w zwykłej aktualizacji, ale wymagają bardzo wysokiej spójności. W przypadku tej aplikacji, jest to ruch planet i aktualizacja ich stanu, dla których spójność jest bardzo ważna.

Standardowe aktualizacje co klatkę, choć niezależne od prędkości aplikacji, nie zapewniają spójnych wyników podczas symulacji. Na przykład, maszyna działająca z częstotliwością 1000 klatek na sekundę generuje znacznie więcej stanów niż maszyna pracująca z częstotliwością 60 klatek na sekundę. Wrażliwe symulacje, zwłaszcza przy dłuższym czasie trwania, mogą być znacznie bardziej podatne na niestabilności w wyniku tej różnicy w ilości generowanych stanów. Dlatego właśnie zastosowano osobną aktualizację zależną od tego kroku czasowego (TPS - Ticks Per Second), aby zagwarantować deterministyczność oraz spójność w rezultatach symulacji niezależnie od prędkości działania aplikacji.

5.2 Główne moduły

Główne moduły reprezentują części odpowiedzialne za konkretne aspekty aplikacji. Są to więc wydzielone sekcje kodu wykonujące bardzo konkretne zadanie, bądź większe encje używające jednego lub wielu komponentów w celu realizacji określonego celu.

Do głównych modułów należą:

- Główny moduł aplikacji.
- Warstwy i sceny.
- Zarządzanie teksturami.
- Fizyka/symulacja.
- Renderer.

Wszystkie moduły współpracują ze sobą, używając mniejszych komponentów, w celu stworzenia kompletnego doświadczenia użytkowego aplikacji.

5.2.1 Główny moduł aplikacji

Główny moduł definiuje cykl życia aplikacji i jest jej rdzeniem - zajmuje się on oknem, wysyłaniem zdarzeń okna oraz przebiegiem obecnej warstwy aplikacji. Inicjalizuje też on wszystkie inne moduły oraz biblioteki wymagające takiej inicjalizacji. Po tym, moduł ten jest miejscem, w którym znajduje się i wykonywana jest główna pętla aplikacji, w której warstwa aplikacji decyduje o dalszym przebiegu. W przypadku zakończenia tej pętli, kończy się zadanie tego modułu, a wraz z nim - cały program.

5.2.2 Warstwy i sceny

Warstwy i sceny są sercem aplikacji - one definiują co aktualnie jest widoczne na ekranie, oraz co użytkownik może w danym momencie zrobić. Należy jednak rozdzielić ich role.

Scena zawiera zbiór obiektów, na których wykonuje własną logikę, zawiera ona między innymi planety, gwiazdy, oraz kamery. Odpowiedzialna jest ona również za bezpośrednią interakcję z użytkownikiem, na przykład za wykrycie zaznaczenia jakiejś planety. Wysyła ona każdy obiekt do renderera, wraz z otoczeniem, takim jak siatka czy skybox.

Warstwa natomiast definiuje co obecnie użytkownik może zrobić ze sceną, oraz interfejs graficzny. Sam widok sceny też jest elementem interfejsu, na który jest nakładana tekstura, do której renderowana jest scena.

5.2.3 Zarządzanie teksturami

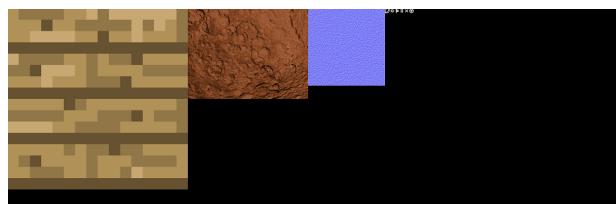
Jednym z problemów związanych z programowaniem grafiki jest minimalizacja wywołań funkcji rysujących, ponieważ są one kosztowne. Celem jest więc wyrenderowanie jak największej ilości obiektów przy każdym takim wywołaniu. Można do tego dążyć wieloma sposobami, ale w tej sekcji mowa jest o teksturach. Tekstury są wgrywane na kartę graficzną i przetrzymywane w jej wewnętrznej pamięci, jednak aby ktoś mógł je wykorzystać, należy ją przypisać do jednego ze slotów - ich ilość jest jednak ograniczona, głównie do 32-64. Sprawia to problem przy ich większej ilości, ponieważ może to wywołać dużo większą komunikację z GPU niż jest to pożądane, oraz dodatkowe wywołania funkcji rysujących. Istnieje wiele sposobów na rozwiązanie tego problemu, przy tworzeniu tej aplikacji rozważane były trzy:

- Dynamiczne zmienianie ID tekstur i slotów - rozwiązanie wiążące się z dużą ilością komunikacji z GPU, które polega na wyrenderowaniu maksymalnej liczby tekstur, po czym załadowania następnych oczekujących. Zaletą na pewno jest mało kosztowna refaktoryzacja kodu do takiego systemu, jednak to rozwiązanie nie skaluje się najlepiej, ponieważ komunikacja z GPU nie jest tania.

- Tablice tekstur 2D - karty graficzne posiadają bufore na tablice tekstur 2D, zawane też teksturami 3D, które pozwalają załadować wiele tekstur pod jeden slot, co zdecydowanie jest zaletą. Niestety ogromnym problemem jest to, że jest to tablica, a więc na każdy element alokowana jest ta sama ilość pamięci. W efekcie oznacza to, że rozmiar każdej jednej tekstury będzie rozmiarem największej tekstury w całej tablicy, co jest marnotrawstwem pamięci.
- Atlas tekstur - zamiast traktować każdą teksturę oddzielnie, alokowana jest jedna, bardzo duża textura, na którą nanosimy wszystkie inne tekstury. W celu rozróżnienia tekstur, należy przetrzymywać informacje o koordynatach UV oraz rozmiarach konkretnych tekstur na tym atlasie. Rozwiążanie jest to bardzo skalowalne, ponieważ nadmiar alokowanej pamięci jest minimalny, a wszystkie tekstury są widziane pod tylko jednym slotem. Kosztem jest jednak cały system zarządzania atlasem i dynamiczne obliczanie koordynatów UV w programach cieniących. [3]

Ze względu na skalowalność, wybrana została opcja trzecia, czyli atlas tekstur. Tym właśnie zajmuje się obecnie przedstawiany moduł.

Przy każdym dodaniu nowej tekstury, sprawdzane jest, czy da się ją zmieścić na obecnym atlasie. Jeśli tak nie będzie, alokowany jest jeszcze większy atlas, a tekstury są na nowo ładowane, wraz z tą nową. Aktualizowane są koordynaty UV, ze względu na nowy rozmiar atlasu, ale obiekty używające tych tekstur nie widzą żadnej zmiany - po prostu dostają inne koordynaty. To rozwiązanie gwarantuje skalowalność wraz ze wzrostem liczby tekstur, oraz minimalne marnotrawstwo pamięci.



Rysunek 5.1: Fragment przykładowego atlasu z teksturami

5.2.4 Fizyka/symulacja

Jest to zdecydowanie najmniejszy moduł, ze względu na to, że aplikacja zajmuje się tylko siłami grawitacyjnymi. Wystawia on dwie funkcjonalności:

- Aktualizacja sceny o krok fizyczny.
- Obliczenie punktów, po których poruszy się obiekt w następnych N krokach.

Aktualizacja sceny opiera się o proste równanie siły między dwoma ciałami, oraz tym jak siła wpływa na przyspieszenie ciała:

$$\vec{F} = G \frac{m_1 m_2}{r^2} \quad (5.1)$$

, gdzie G jest stałą grawitacyjną, m_1 i m_2 to masy ciał, a r to odległość między nimi, oraz:

$$\vec{a} = \frac{\vec{F}}{m} \quad (5.2)$$

, gdzie \vec{F} jest wcześniej wyliczoną siłą, a m jest masą tego ciała. Uproszczony kod implementujący taką funkcjonalność znajduje się poniżej:

```
1 vec3 dir = normalize(other.Position - planet.Position);
2 double distance2 = distance2(planet.Position, other.Position);
3 double m1 = planet.Mass;
4 double m2 = other.Mass;
5
6 vec3 F = dir * m1 * m2 / distance2;
7 F *= G_CONSTANT_MULTIPLIER * SCALE_FACTOR;
8 vec3 addAccel = F / (m1 * SUN_MASS);
9 planet.Velocity += Application::TPS_STEP * addAccel;
```

Rysunek 5.2: Uproszczony kod aktualizujący prędkość na podstawie sił.

Takie obliczenie występuje dla każdej z par obiektów i jest wykonywane asynchronicznie, jako że każdy stan jest od siebie niezależny w tym momencie, co pozwala na wydajniejszą symulację większej ilości obiektów.

Należy jednak zwrócić uwagę na szczegół w tych obliczeniach, głównie w stałych denotowanych wielkimi literami: `G_CONSTANT_MULTIPLIER`, `SCALE_FACTOR`, `SUN_MASS`.

Jednym z problemów takich symulacji są wartości, takie jak masy ciał. W przypadku słońca, jest to wartość rzędu 10^{30} . Mimo ogromnej masy, nie jest to też najbardziej masywne ciało we wszechświecie, a takie wartości są też wymnażane. Czy w przypadku masy odległości, stawia to problem przetłumaczenia tego na system jednostek w naszej symulacji. Stawia to też problem przy reprezentacji liczb - istnieją takie wartości, przy których wymnożeniu stracimy kompletnie precyzję, lub wynik po prostu wyjdzie poza obsługiwany zakres danego formatu. Jednym ze sposobów na obejście tego jest skalowanie wartości. Przykładowo, w linii 8 wcześniejszego listingu: `glm::dvec3 addAccel = F / (m1 *`

SUN_MASS); - masa jest jedną z przeskalowanych rzeczy. W tym przypadku, w symulacji wartość masy '1.0' jest równa masie słońca w prawdziwym świecie. Do tego samego służą pozostałe stałe - skalary pozwalające nam przetłumaczyć wartości na takie, z którymi poradzi sobie symulator, oraz pilnujące, aby żadna wartość nie wyszła poza zakres.

5.2.5 Renderer

Renderer jest centralną częścią odpowiadającą za całe przygotowanie oraz przebieg procesu renderowania obiektów. Do jego zbudowania użyto API OpenGL - jest to standard, który istnieje od 1992 roku i mimo że w środowiskach profesjonalnych używa się nowszych rozwiązań, to do dzisiaj pozostaje jednym z najpopularniejszych wyborów wśród ludzi zaczynających z tą dziedziną programowania, ze względu na swoją uniwersalność i ilość materiałów edukacyjnych w Internecie. [14]

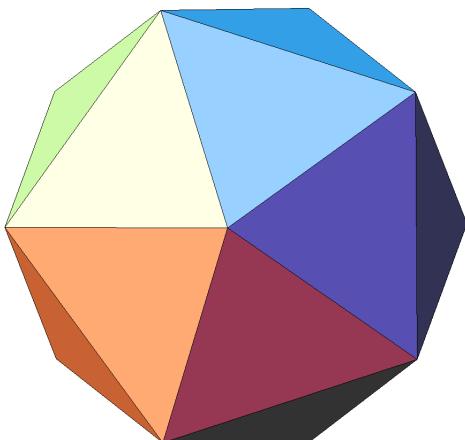


Rysunek 5.3: Logo OpenGL

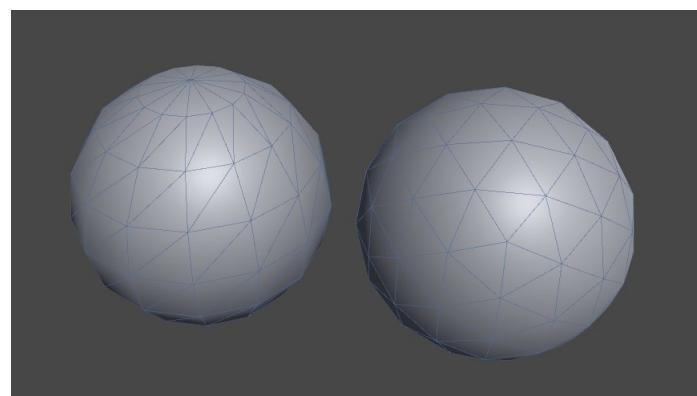
Siatki sfer

W przypadku tego projektu, najważniejszym jest renderowanie sfer, a więc potrzebne są dane o wierzchołkach sfery. Istnieje wiele rodzajów sfer, gdzie dwoma najpopularniejszymi są:

- Sfery UV (UV Spheres) - sfery generowane na podstawie punktów geograficznych, z nierównomiernie rozmieszczonymi wierzchołkami. Są one proste do stworzenia, a ich koordynaty wierzchołków pokrywają się z koordynatami UV tekstur. Niemniej jednak ich wykorzystanie wierzchołków jest mało wydajne, a ich dokładność nie jest najlepsza.
- Ikosfery (Icospheres) - oparte są na dwudziestościanie, którego ściany są następnie dzielone na jeszcze mniejsze a nowe wierzchołki wysuwane na odpowiednią odległość, aż do pożąданiej dokładności. Wierzchołki są równomiernie rozłożone, oraz można uzyskać dokładniejsze i bardziej gładkie wyniki niż przy sferach UV. Proces stworzenia takiej sfery jest jednak bardziej złożony.



Rysunek 5.4: Dwudziestościan - podstawa ikosfery (math.wikia.com)



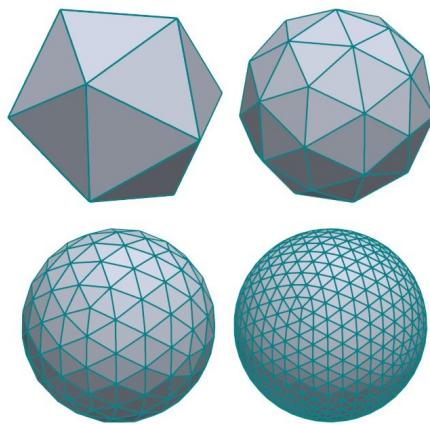
Rysunek 5.5: Porównanie siatek sfer: UV (lewo), Ikosfera (prawo)

W tej aplikacji wykorzystywane są ikosfery. Dane o wierzchołkach są generowane raz i przesyłane na kartę graficzną. Zgodnie z opisem, na początku należy wygenerować wierzchołki dwudziestościanu, a następnie podzielić jego każdą ścianę na kolejne wierzchołki, czym zajmuje się funkcja, której uproszczony kod widoczny jest poniżej:

```

1 void Subdivide(
2     vec3 a,
3     vec3 b,
4     vec3 c,
5     int32_t depth
6 )
7 {
8     if (depth == 0)
9     {
10         vec3 normA = normalize(a);
11         vec3 normB = normalize(b);
12         vec3 normC = normalize(c);
13
14         Vertices.push_back({ normA, normB, normC });
15
16         return;
17     }
18
19     vec3 ab = normalize(a + b);
20     vec3 bc = normalize(b + c);
21     vec3 ca = normalize(c + a);
22
23     Subdivide(a, ab, ca, depth - 1);
24     Subdivide(ab, b, bc, depth - 1);
25     Subdivide(ca, bc, c, depth - 1);
26     Subdivide(ab, bc, ca, depth - 1);
27 }
```

Rysunek 5.6: Uproszczony kod generujący wierzchołki ikosfery



Rysunek 5.7: Proces generowania ikosfery

Ten rekurencyjny algorytm dzieli coraz to mniejsze ściany, aż dojdzie do maksymalnej głębokości, na którą pozwolił kontekst wywołujący tę funkcję (parametr `depth`) - im większa głębokość, tym dokładniejsza sfera.

Optymalizacja renderowania sfer

Jak wcześniej wspomniano w tej pracy, istotna jest optymalizacja ilości wywołań funkcji rysujących oraz wymiany danych pomiędzy CPU oraz GPU. Bardzo ważną decyzją jest sposób renderowania sfer, ponieważ to najbardziej się liczy przy wydajności całego procesu. Do renderowania podobnych obiektów można podejść na wiele sposobów, ale najważniejszymi będą:

- Renderowanie każdej sfery osobno - jest to najprostsze, ale i najgorsze rozwiązanie, ponieważ na każdą sferę przypada jedno wywołanie funkcji rysujących, oraz za każdym razem trzeba wgrywać nowe dane, co jest wolne.
- Batch rendering - jest to technika polegająca na alokacji dużego bufora na GPU, a następnie wgranie do niego jak największej ilości danych wielu wierzchołków. Rozwiązanie to pozwala nam ograniczyć funkcje rysowania do minimum i jest świetnym rozwiązaniem dla prostych siatek, składających się z małej ilości wierzchołków. Jednak sfera posiada ich minimum kilkaset, przez co to rozwiązanie jest bezużyteczne, ponieważ wgranie wielu takich danych wierzchołków na raz będzie wolne.
- Instancjonowanie - jest to technika, która idealnie pasuje do zaistniałej sytuacji, ponieważ każda sfera ma taką samą siatkę. Pozwala ona na wgranie danych wierzchołków do jednego bufora raz, a do innego bufora można wtedy wgrać to, co faktycznie będzie różne dla innych sfer, na przykład informacje o materiale. W efekcie otrzymujemy bardzo wydajne pamięciowo rozwiązanie, które również jest bardzo szybkie, ponieważ renderowanie wielu instancji odbywa się przez wywołanie jednej funkcji.

Ze względu na idealne dopasowanie, w aplikacji użyto metody instancjonowania, co pozwala nam renderować tysiące obiektów z tą samą siatką bez utraty wydajności.

Programy cieniujące

Program cieniujący, nazywany również shaderem, stanowi kluczowy element procesu renderowania na karcie graficznej. Dwa główne rodzaje tych programów, programy wierzchołków i fragmentów, pełnią kluczowe role w manipulacji wizualnej każdego elementu sceny.

Wykonywany dla każdego wierzchołka w buforze, program wierzchołków przyjmuje dane dotyczące konkretnego wierzchołka jako wejście i generuje nowe współrzędne w celu

przekształcenia geometrii obiektów. W ramach swojego działania może dokonywać różnorodnych przekształceń geometrycznych, takich jak translacje, rotacje czy skalowanie. Ponadto może obliczać wszelkie dodatkowe informacje związane z wierzchołkami, takie jak wektory normalne, które są istotne w procesie oświetlenia.

Wykonywany dla każdego fragmentu (piksela) w buforze GPU, program fragmentów skupia się na obliczeniach związanych z oświetleniem oraz decyduje o końcowym kolorze danego fragmentu, który staje się jego wyjściem. W ramach tego programu może następować korzystanie z tekstur oraz różne operacje matematyczne mające na celu określenie finalnego koloru piksela na ekranie.

Te dwa typy programów cieniących są jedynymi wymaganymi, ponieważ bez nich tak na prawdę nie wiadomo jak dane w buforach GPU powinny być przetworzone - ich współdziałanie jest niezbędne do stworzenia obrazu sceny.

Renderer odpowiedzialny jest za stworzenie, komplikację oraz wgranie tych programów na kartę graficzną. Odpowiadające sobie programy wierzchołków i fragmentów łączone są w jeden, właściwy program. Renderer przełącza się między programami, jako że jest ich kilka, a w trakcie rysowania używany jest tylko jeden. Całość systemu zapewnia spójność wykonywanych operacji oraz oferuje metody dla kontekstów zewnętrznych, umożliwiające wykorzystanie tego systemu.

5.3 Wykorzystywane techniki

W tej sekcji zaprezentowanych zostanie kilka wybranych technik użytych w aplikacji. Będą to techniki ściśle związane z programowaniem grafiki 3D, które służą zwiększeniu wygody użytkowania, zwiększeniu stopnia modyfikacji wyglądu planet oraz samego cieniowania.

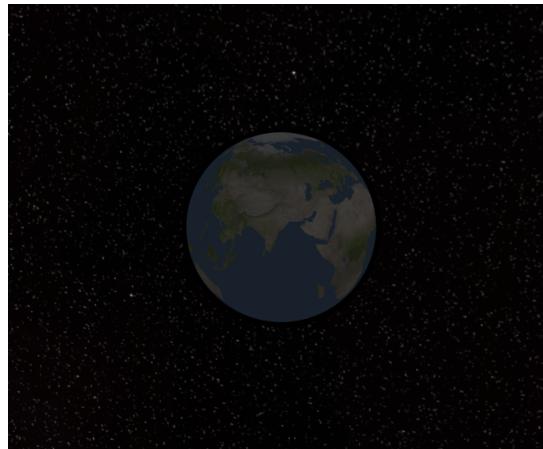
5.3.1 Oświetlenie

Kluczowym aspektem, determinującym odczucie głębokości sceny, jest oświetlenie. Ostateczny obraz, który obserwujemy na monitorze, jest dwuwymiarowy, jednak właściwe zastosowanie oświetlenia nadaje trójwymiarowy charakter przedstawianej przestrzeni. W tej aplikacji wykorzystano model oświetlenia Blinn-Phonga, będącego rozwinięciem klasycznego modelu Phonga. Model ten, choć nieco modyfikuje pewne aspekty obliczeń, oferuje zadowalające rezultaty przy umiarkowanym obciążeniu obliczeniowym.

Na początku przyjrzyjmy się modelowi Phonga, jako że jest on podstawą użytego modelu. Zakłada on istnienie trzech rodzajów światła, które zostaną omówione poniżej.

Światło otaczające (ang. ambient)

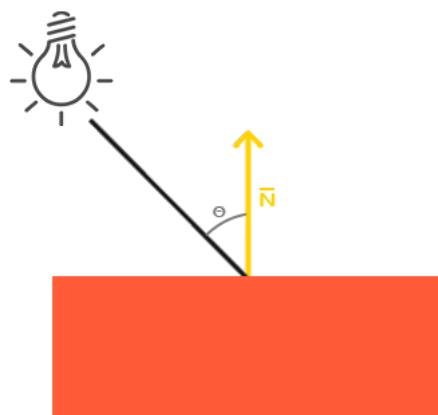
Jest to światło o stałej intensywności, które jest odpowiedzialne za ogólną jasność obiektu niezależnie od źródła światła. Jest to rodzaj światła tła, które równomiernie oświetla wszystkie obiekty w scenie, przez co przy kompletnym braku innego źródła światła - są one widoczne.



Rysunek 5.8: Światło otaczające - brak głębi

Światło rozproszone (ang. diffuse)

Jest to efekt istnienia faktycznego źródła światła, które oświetla obiekt z konkretnej strony. Intensywność tego światła zależy od kąta padania na powierzchnię - im bardziej powierzchnia jest prostopadła do kierunku światła, tym jest mocniej oświetlana.



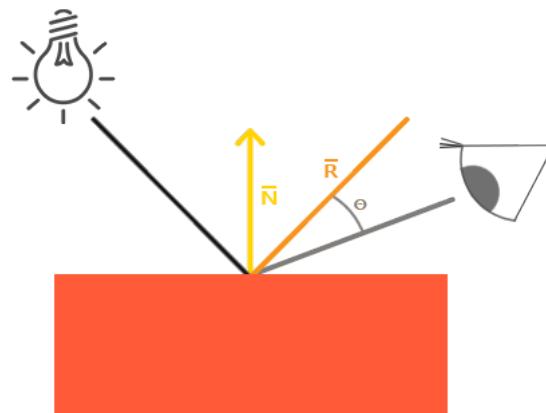
Rysunek 5.9: Działanie światła rozproszonego - intensywność zależy od kąta między normalną powierzchni \vec{N} , a źródłem światła (Źródło: learnopengl.com)



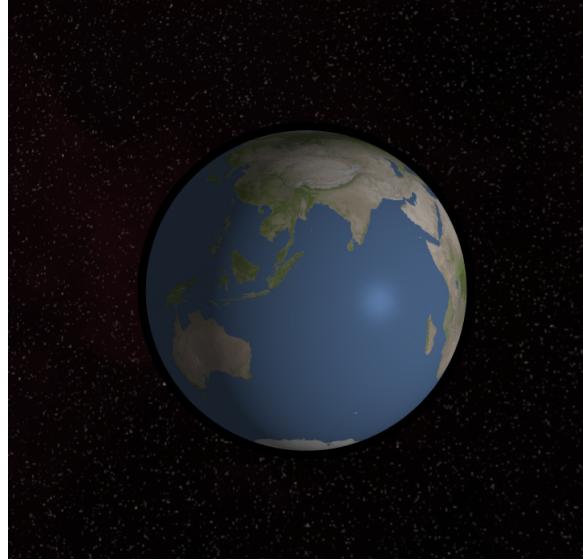
Rysunek 5.10: Światło rozproszone - widoczny kierunek światła

Światło odbite (ang. specular)

Jest ono odpowiedzialne za efekt odbicia się światła od powierzchni obiektu. Jego intensywność zależy od kąta odbicia światła i kąta widzenia obserwatora. Im bliższe są one do siebie, tym to odbicie jest bardziej intensywne.

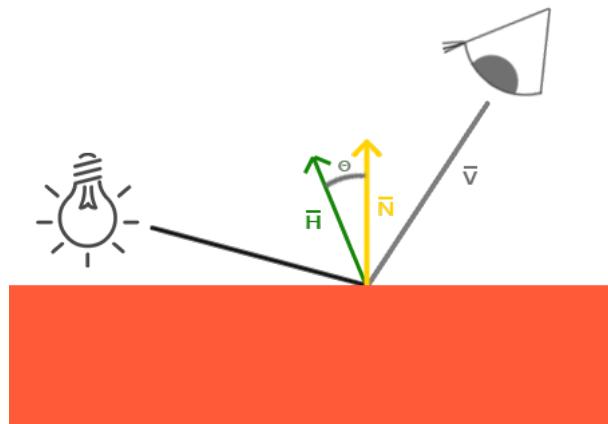


Rysunek 5.11: Działanie światła odbitego - intensywność zależy od kąta między kierunkiem odbitego światła \vec{R} , a obserwatorem (Źródło: learnopengl.com)

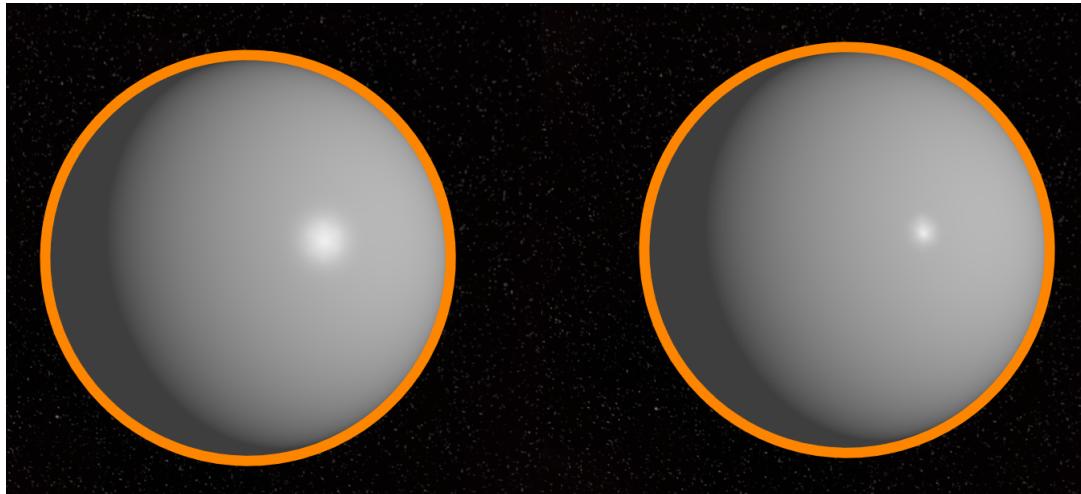


Rysunek 5.12: Światło odbite - widoczny punkt odbicia światła

W modelu Blinn-Phonga istnieje różnica w kontekście światła odbitego. W przeciwieństwie do koncentracji na wektorze światła odbitego, model ten wykorzystuje wektor pośredni pomiędzy wektorem obserwatora a źródłem światła. Intensywność jest wtedy większa, gdy wektor pośredni jest bardziej pokryty z normalną powierzchni. Takie podejście przyczynia się do uzyskania bardziej realistycznych rezultatów, zwłaszcza przy większych kątach odbicia.



Rysunek 5.13: Korekta modelu Blinn-Phong - intensywność rośnie, im wektor środkowy \vec{H} jest bardziej pokryty z normalną powierzchni \vec{N} (Źródło: learnopengl.com)



Rysunek 5.14: Porównanie modelu Blinn-Phong (po lewej) z modelem Phong (po prawej)

Połączenie wszystkich tych zabiegów daje rezultat w postaci przyzwoitego oświetlenia, które nadaje całej scenie wrażenie głębi. Pozwala to również na większą kreatywność autora sceny, ze względu na parametryzację takich właściwości jak kolor światła, intensywność czy siła odbitego światła danego obiektu.

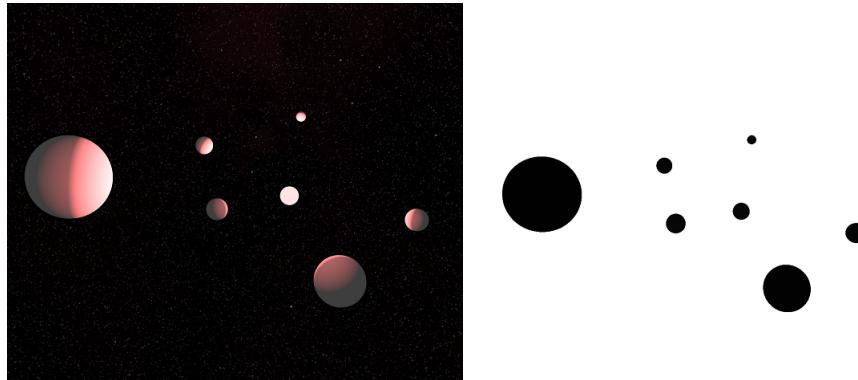
5.3.2 Wybieranie obiektów na scenie

Podczas edycji sceny użytkownik może wybrać obiekt poprzez kliknięcie na niego myszką. O ile w scenie dwuwymiarowej nie jest to trudne zadanie, ponieważ pozycję kurSORA myszki można w bardzo prosty sposób przetłumaczyć na płaską przestrzeń, problem ten staje się nietrywialny w przestrzeni 3D ze względu na głębię oraz nieliniowość bryły widzenia kamery. Do rozwiązania tego problemu istnieją dwie popularne metody:

- Rzutowanie promienia - jest to technika polegająca na wypuszczeniu promienia (prostej) w przestrzeń, zaczynając od pozycji obserwatora, w kierunku zaznaczonym kursorem myszy z uwzględnieniem głębi. Następnie sprawdzane są punkty przecięcia tego promienia z obiektami i na tej podstawie identyfikowany jest odpowiedni obiekt. Metoda ta jest precyzyjna oraz prosta w implementacji, jednak złożoność obliczeniowa i wydajność nie skalują się najlepiej wraz ze wzrostem ilości obiektów na scenie.
- Color picking - technika identyfikacji obiektów przy użyciu renderera. Scena jest renderowana osobno do tekstury za pomocą specjalnego programu cieniąjącego, który nie uwzględnia oświetlenia. Każdy obiekt jest kolorowany na podstawie jego unikalnego ID, co generuje teksturę o różnych odcieniach szarości. Następnie, poprzez analizę koloru fragmentu, na którym znajduje się kurSOR myszy, możemy zidentyfikować obiekt o konkretnym ID. Metoda ta również jest precyzyjna i uniwersalna

dla każdej geometrii obiektów oraz zawsze poprawnie rozstrzyga przysłaniające się obiekty. Jest to jednak koszt dodatkowego rysowania sceny.

W tej aplikacji wykorzystano Color picking ze względu na jego precyzję i niezależność od głębi czy geometrii - jako że operujemy tylko na wyjściowej teksturze. Obiekty są generowane na białym tle, a obiekty kolorowane są zaczynając od koloru czarnego. W tej metodzie w niektórych przypadkach można doświadczyć konfliktów nakładających się kolorów, jeśli ich różnica jest zbyt mała, jednak w przypadku tej aplikacji nie napotkano tego problemu.

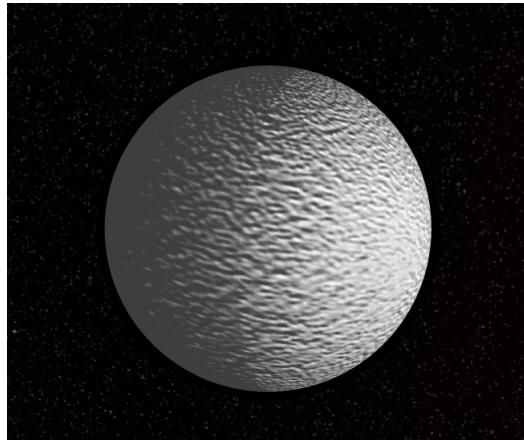


Rysunek 5.15: Przykładowa tekstura wykorzystywana do Color picking'u - wszystkie obiekty wyglądają na idealnie czarne, jednak nie są to identyczne kolory.

5.3.3 Mapowanie wektorów normalnych

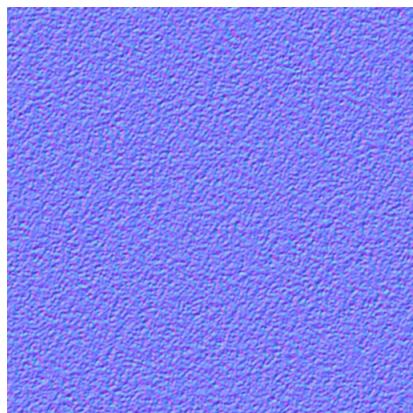
Wektory normalne powierzchni, jak wspomniano przy oświetleniu, wpływają na to, w jaki sposób światło odbija się od danego fragmentu powierzchni. W prostym przypadku, gdy te wektory są obliczane na podstawie wierzchołków, cały obiekt prezentuje się bardzo gładko, co można zaobserwować na dotychczasowych zrzutach ekranu. Niemniej jednak użytkownik może zechcieć nadać obiekowi bardziej złożoną geometrię, gdyż planety nie są idealnie gładkie. Jednym ze sposobów osiągnięcia tego celu jest ingerencja w geometrię planety, jednak takie podejście jest skomplikowane i kosztowne. Istnieje bardziej efektywne rozwiązanie - mapowanie wektorów normalnych.

Iluzję nierównej powierzchni, można uzyskać poprzez manipulację wektorami normalnymi. Gdy te wektory nie pokrywają się z idealnymi, światło odbija się w różny sposób, co prowadzi do efektu nierównej powierzchni.



Rysunek 5.16: Planeta z nowymi wektorami normalnymi - widoczna nierówność

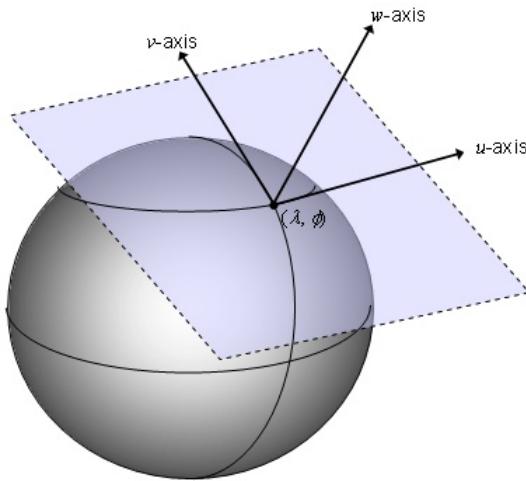
W omawianej aplikacji, użytkownik może wgrać teksturę, której piksele reprezentują wartości wektorów normalnych. Dane tej tekstury są używane w programie fragmentów, które są traktowane jak wektory normalne danego fragmentu, dając użytkownikowi pełną kontrolę nad zachowaniem światła przy każdej części obiektu.



Rysunek 5.17: Tekstura normalna wykorzystana do poprzedniego przykładu

W takim rozwiążaniu istnieje jednak pewien problem - jak można zauważyc, ogólny „kierunek” wektorów tej tekstury jest mniej więcej taki sam. Oznacza to, że w przypadku rotacji obiektu, albo krzywizny planety, wektory te nie będą pokrywały się z oczekiwanyimi, co całkowicie zepsuje oświetlenie. Rozwiązaniem tego problemu jest przekształcenie tych wektorów do **przestrzeni stycznej**. Jest to przestrzeń, która jest definiowana dla każdego wierzchołka obiektu, za pomocą trzech wektorów - wektora normalnego, stycznego i bitangenckiego. Wektor normalny jest taki sam jak dla wierzchołka, wektor styczny jest wektorem stycznym do powierzchni, a bitangencki jest efektem iloczynu wektorowego dwóch poprzednich wektorów. Te trzy wektory opisują lokalny system współrzędnych dla poszczególnych wierzchołków, który możemy opisać macierzą. Z tym narzędziem, po odczytaniu wartości wektora normalnego z tekstuury, możemy wymnożyć go przez tę macierz, czego skutkiem będzie jego transformacja do lokalnego systemu, co poprawnie zorientuje

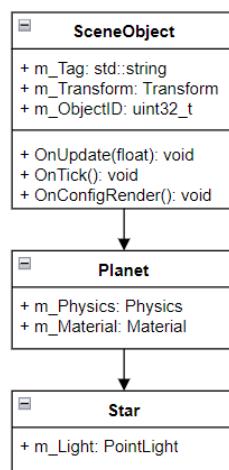
odczytane wektory normalne. [15]



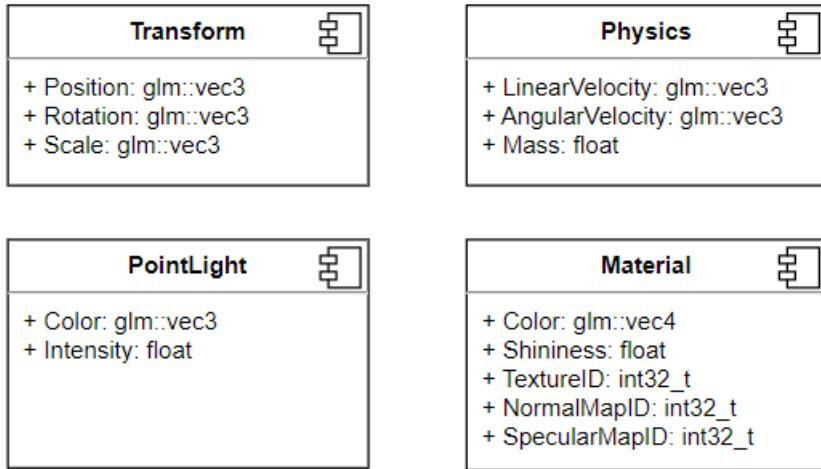
Rysunek 5.18: Przykład płaszczyzny przestrzeni stycznej na wierzchołku sfery

5.3.4 Struktura obiektów

Ze względu na niewielką ilość typów obiektów w tej aplikacji, ich struktura nie jest skomplikowana.



Rysunek 5.19: Diagram przedstawiający relacje klas



Rysunek 5.20: Struktura komponentów

Podstawową strukturą jest klasa *SceneObject*, która zawiera komponent *Tag* oraz ID obiektu. Komponent *Tag* jest po prostu nazwą obiektu. Klasa ta również dba o to, aby każdy obiekt dostał unikatowe ID przy tworzeniu.

Dziedziczy po nim klasa *Planet*, która reprezentuje zwykłą planetę. Posiada ona dodatkowo komponenty: *Transform*, *Physics*, *Material*. Na instancjach tej klasy wykonywane są symulacje i to ich reprezentacje są wyświetlane na widoku sceny.

Jako że gwiazda, w kontekście tej aplikacji, jest planetą która dodatkowo jest źródłem światła, to reprezentującą ją klasa *Sun* dziedziczy po klasie *Planet*, implementując dodatkowo komponent *PointLight*.

Taka struktura i hierarchia obiektów ułatwia proces wprowadzania ich nowych typów, co otwiera drzwi dla wielu możliwości, w których kierunku aplikacja mogłaby być dalej rozwijana.

5.3.5 Użyte biblioteki

W tej sekcji opisane zostaną najważniejsze użyte biblioteki w tym projekcie. Jako że sama aplikacja była pisana „od zera”, tzn. nie wykorzystano gotowego silnika (na przykład *Unity*), wiele systemów wspomaga lub implementują biblioteki. Były one ograniczone do minimum, jednak te użyte były kluczowe w przyspieszeniu procesu budowania aplikacji ze względu na udogodnienia, lub implementację bardzo skomplikowanych zagadnień, takich jak wieloplatformowy interfejs użytkownika.

GLAD

Jak wspomniano wcześniej, aplikacja budowana jest z użyciem API OpenGL. OpenGL nie jest biblioteką, a standardem, który jest udostępniany przez producentów układów graficznych, ponieważ to oni implementują ten standard. Należy jednak załadować udostę-

nione funkcje do aplikacji, a jest ich bardzo dużo. Z tego względu wykorzystano bibliotekę *GLAD*, która posiada definicje tych funkcji oraz ładuje je do aplikacji. [6]

GLFW

Podstawową rzeczą do działania omawianej aplikacji jest okno, w którym wszystko się dzieje. Okna są jednak koncepcją związaną bezpośrednio z systemem operacyjnym, na którym aplikacja jest uruchamiana - z poziomu kodu, inaczej wygląda zarządzanie oknem na systemie Windows, a inaczej na systemie zbudowanym na jądrze Linux. Ze względu na założenie, że aplikacja będzie wieloplatformowa, aplikacja wykorzystuje bibliotekę, która zajmuje się rozstrzyganiem, których wersji używać. Wystawia ona również wygodne funkcjonalności, takie jak przechwytywanie zdarzeń okna, oraz bezpośrednio współpracę ze standardem OpenGL. [7] [4]

Dear ImGui

Dear ImGui jest najpopularniejszą biblioteką do budowania GUI wśród ludzi lubiących się z programowaniem grafiki. Jest to interfejs typu „immediate”, co oznacza, że każdy element jest renderowany i aktualizowany co klatkę, w przeciwieństwie do klasycznych aplikacji desktopowych, gdzie aktualizacja GUI występuje wtedy, gdy wystąpi jakieś zdarzenie (na przykład wcisnięcie przycisku). Rozwiążanie ImGui jest bardziej wymagające, bo to renderowanie zawsze występuje, ale ze względu na prostotę elementów jest to praktycznie zerowy koszt, a sam interfejs jest przez to bardziej responsywny. Biblioteka jest ta stworzona z myślą deklaratywności oraz prostoty - wystarczy bardzo mało kodu, aby stworzyć responsywny interfejs graficzny, a wywołania kolejnych funkcji po sobie bezpośrednio odpowiadają temu, co będzie widoczne na ekranie. [8]

Rozdział 6

Weryfikacja i walidacja

Ten rozdział zostanie poświęcony testowaniu i weryfikacji funkcjonalności aplikacji. W czasie tworzenia oprogramowania zdarzają się błędy, które należy wykryć i naprawić. Mogą to być bardzo proste błędy, takie jak używanie złej wartości skalarnej przy obliczeniach, które są proste i szybkie do naprawienia, ale zdarzają się też błędy w logice systemów, który nie muszą być łatwo wykrywalne.

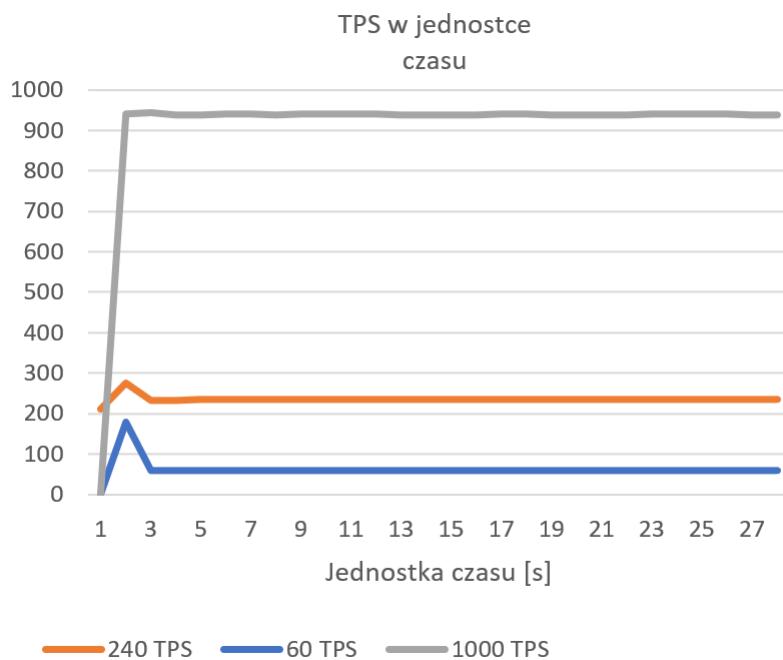
Ciągłe testowanie aplikacji w trakcie jej tworzenia pomaga zapobiec takim zdarzeniom i głównie taki sposób został przyjęty przy tworzeniu tego projektu, opierając się na cyklu *planowanie – > implementacja – > testowanie*. Przedstawione zostaną przykładowe testy użyte w czasie implementowania aplikacji.

Sama fizyka była testowana na podstawie testów jednostkowych oraz empirycznych. Testy jednostkowe obejmowały między innymi:

- Symulacja sceny z jednym obiektem - w takiej scenie obiekt nie powinien się w ogóle poruszać.
- Symulacja sceny z więcej niż jednym obiektem - w takiej scenie, zakładając poprawne wartości fizyczne obiektów, powinny się one przyciągać do siebie.
- Symulacja sceny, gdzie któryś z obiektów posiadał ujemną lub zerową masę - każde ciało posiada masę większą od zera, więc obiekt z niepoprawną masą nie powinien wpływać na siły w całej scenie oraz sam nie powinien podlegać żadnym siłom.
- Symulacja sceny, gdzie dystans między obiektami wynosi dokładnie 0 - ze względu na brak kolizji w aplikacji, taka sytuacja może się zdarzyć. Jednak ze względu na wzór na siłę grawitacji między dwoma ciałami, przy zerowej odległości od siebie byłoby to dzielenie przez 0, do czego nie powinno dojść.
- Symulacja sceny z dwoma identycznymi obiektami - jest to jedna z sytuacji, gdzie wynik jest deterministyczny i zależność między siłami jest zawsze taka sama, tzn. siły te są identycznymi wektorami, ale zwróconymi naprzeciw siebie. Taki test jest dodatkowym testem potwierdzającym poprawność obliczeń.

Ze względu na chaos i nieprzewidywalność rozwoju układu z więcej niż dwoma obiektami, testy większych układów sprowadzały się do utworzenia sceny imitującej znaną konfigurację, na przykład Słońce - Ziemia - Mars, po czym konfrontowaniu rozwoju sceny z tym, czego oczekujemy w prawdziwym świecie - przykładowo czy okres Marsa w symulacji jest zbliżony do 1.88 okresu Ziemi.

Testowana była również stabilność systemu TPS, jako że jest ona ważną częścią stabilności symulacji. Taktowanie TPS powinno być jak najbardziej stałe, aby osiągnąć spójność symulacji. Takty były monitorowane a ich wartości naniesione na wykres w celu wizualizacji:



Rysunek 6.1: Wykres stabilności TPS

Z wykresu można odczytać, że z wyjątkiem startu aplikacji, ilość TPS pozostaje spójna. Skok na początku pojawia się zawsze i wynika z uruchamiania aplikacji, jednak nie wpływa na symulację, która w tym czasie nie może się odbywać. Mimo stabilności TPS można zauważać, że im większa jest jego ustawiona wartość, tym bardziej oddala się od tej wartości - przy 60 TPS, co każdą sekundę faktyczną ilość wynosiła 60 lub 59. Przy 1000 TPS - ta wartość już spadła do 940 z błędem +/- 1. Wynika z tego, że system ten nie jest idealnie stabilny, ponieważ zdarza mu się stracić takt, nawet jeśli czasami wykonywany jest jeden nadmiarowy. Co takt jednak aktualizacja jest wykonywana z tym samym krokiem czasowym, więc wyniki symulacji dalej będą takie same. Różnica pojawi się z czasem pojawienia się tych wyników - jeśli w ciągu sekundy zakładamy 59 TPS zamiast 60, to w ciągu minuty stracimy jedną sekundę. Czas obliczany w symulacji będzie jednak dalej poprawny.

Proces generacji ikosfery również został potraktowany testami jednostkowymi, jako że

jest to deterministyczny algorytm. Dla głębokości generacji od 1 do 4, sprawdzone zostały ilości wierzchołków oraz długości wektorów składających się na każdy wierzchołek, jako że powinny to być wektory normalne.

Testem wydajnościowym został potraktowany proces symulacji, jako iż jest to najważniejsza część aplikacji pod tym względem.

Tabela 6.1: Porównanie szybkości symulacji synchronicznej i asynchronicznej

Ilość obiektów	Synchronicznie	Asynchronicznie
10	0.03ms	0.06ms
20	0.1ms	0.1ms
50	0.6ms	0.23ms
100	2.2ms	0.6ms
200	8ms	1.8ms
500	58ms	10ms
1000	220ms	36ms

Samo zastosowanie równoległości obliczeń, jak oczekiwano, nadało ogromne przyspieszenie. Dzięki temu testowi również można założyć, że każda scena z typową ilością obiektów nie będzie degenerowała wydajności do nieakceptowalnych poziomów - przy aż 200 obiektach aplikacja utrzymywała około 500 klatek na sekundę, gdzie 200 jest ogromną liczbą, która nie jest oczekiwana przy rzeczywistych scenach.

Rozdział 7

Podsumowanie i wnioski

Cel pracy został w większości osiągnięty - sama symulacja nie jest tak idealna jak zakładano, a w zasadzie skalowanie wartości fizycznych, ale za to jest w pełni funkcjonalna. Pisanie tej aplikacji pozwoliło mi zapoznać się z wieloma technikami, które zostały w tej pracy opisane, jak i mniejszymi aspektami z tej domeny. Planowane jest dalsze rozwijanie tej aplikacji głównie w kierunku Renderera, aby wykorzystać techniki o których czytałem, jednak ze względu na wymagania oraz czas nie zostały zaimplementowane - głównie byłyby to:

- PBR (Physically Based Rendering) - technika do obliczania interakcji obiektu ze światłem na podstawie jego własności fizycznych, celująca w jak największy realizm. Technika ta też pozwala na większą kontrolę nad wyglądem planety od strony użytkownika. Ten sposób można zaobserwować w aplikacjach zbudowanych na silniku Unreal Engine.
- Cienie - w tej aplikacji obiekty nie rzucają cieni, które nie są trywialnym i tanim efektem. Wymaga to wielu dodatkowych renderów i tekstur oraz technik optymalizacyjnych - ze względu na koszt.
- HDR (High Dynamic Range) oraz Bloom - HDR jest techniką, która używa tekstur, w których piksele mogą mieć wartości wyższe niż całkowita biel. Jest to użyteczne do wyznaczania wyjątkowo jasnych fragmentów ekranu, co można wykorzystać do stworzenia efektu Bloom, który jest efektem symulującym wysoką intensywność źródła światła i rozlewania się tego światła wokół. Efekt jest bardzo popularny w aplikacjach czasu rzeczywistego, ponieważ pozwala on dużo bardziej zwiększyć immersję.

Bibliografia

- [1] J. C. Butcher. „A history of Runge-Kutta methods”. W: *Applied Numerical Mathematics* 20 (1996), s. 247–260.
- [2] Kate Compton. „Creating spherical worlds”. W: *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches* (2007), 82–es.
- [3] NVIDIA Corporation. *Improve Batching Using Texture Atlases*. 2004. URL: https://download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf.
- [4] Dokumentacja biblioteki *GLFW*. URL: <https://www.glfw.org/docs/latest/>.
- [5] Helmut Grubmüller. „Generalized Verlet algorithm for efficient molecular dynamics simulations with long-range interactions”. W: *Molecular Simulation* 6 (1991), s. 121–142.
- [6] Repozytorium *GLAD* używanej gałęzi. URL: <https://github.com/Dav1dde/glad/tree/c>.
- [7] Repozytorium *GLFW*. URL: <https://github.com/glfw glfw>.
- [8] Repozytorium *ImGui*. URL: <https://github.com/ocornut/imgui>.
- [9] Rainer Spurzem. „Direct N-body simulations”. W: *Journal of Computational and Applied Mathematics* 109 (1999), s. 407–323.
- [10] Strona główna *AstroGrav*. URL: <http://www.astrograv.co.uk/>.
- [11] Strona główna *Celestia*. URL: <https://celestiaproject.space/>.
- [12] Strona główna *SpaceEngine*. URL: <https://spaceengine.org/>.
- [13] Strona główna *Universe Sandbox*. URL: <https://universesandbox.com/>.
- [14] Użyta dokumentacja *OpenGL*. URL: <https://docs.gl>.
- [15] Joey de Vries. *Normal mapping*. URL: <https://learnopengl.com/Advanced-Lighting/Normal-Mapping>.

Dodatki

Spis skrótów i symboli

GPU Graphics Processing Unit (karta graficzna)

CPU Central Processing Unit (procesor)

TPS Ticks Per Second

Spis rysunków

4.1	Edytor z pustą sceną	15
4.2	Panel sceny	16
4.3	Górny pasek	16
4.4	Panel obiektu	17
4.5	Edytor z wypełnioną sceną	18
4.6	Widok symulacji	19
5.1	Fragment przykładowego atlasu z teksturami	25
5.2	Uproszczony kod aktualizujący prędkość na podstawie sił	26
5.3	Logo OpenGL	27
5.4	Dwudziestościan - podstawa ikosfery (math.wikia.com)	28
5.5	Porównanie siatek sfer: UV (lewo), Ikosfera (prawo)	28
5.6	Uproszczony kod generujący wierzchołki ikosfery	29
5.7	Proces generowania ikosfery	29
5.8	Światło otaczające - brak głębi	32
5.9	Działanie światła rozproszonego - intensywność zależy od kąta między normalną powierzchni \vec{N} , a źródłem światła (Źródło: learnopengl.com)	32
5.10	Światło rozproszone - widoczny kierunek światła	33
5.11	Działanie światła odbitego - intensywność zależy od kąta między kierunkiem odbitego światła \vec{R} , a obserwatorem (Źródło: learnopengl.com)	33
5.12	Światło odbite - widoczny punkt odbicia światła	34
5.13	Korekta modelu Blinn-Phong - intensywność rośnie, im wektor środkowy \vec{H} jest bardziej pokryty z normalną powierzchni \vec{N} (Źródło: learnopengl.com)	34
5.14	Porównanie modelu Blinn-Phong (po lewej) z modelem Phong (po prawej)	35
5.15	Przykładowa tekstura wykorzystywana do Color picking'u - wszystkie obiekty wyglądają na idealnie czarne, jednak nie są to identyczne kolory.	36
5.16	Planeta z nowymi wektorami normalnymi - widoczna nierówność	37
5.17	Tekstura normalna wykorzystana do poprzedniego przykładu	37
5.18	Przykład płaszczyzny przestrzeni stycznej na wierzchołku sfery	38
5.19	Diagram przedstawiający relacje klas	38
5.20	Struktura komponentów	39

6.1 Wykres stabilności TPS	42
--------------------------------------	----

Spis tabel

6.1 Porównanie szybkości symulacji synchronicznej i asynchronicznej 43