UNIWERSYTET JANA DŁUGOSZA w CZĘSTOCHOWIE

Wydział Nauk Ścisłych, Przyrodniczych i Technicznych

Kierunek: Informatyka
Specjalność: Inżynieria oprogramowania

Filip Świszcz
nr albumu: 80236

Interaktywna wizualizacja systemu słonecznego
Interactive visualization of the solar system

Praca inżynierska przygotowana
pod kierunkiem
prof. Andrzeja Zbrzeznego
mgr. inż. Jolanty Podolszańskiej

Częstochowa, 2026

Streszczenie

Aplikacja desktopowa będąca symulatorem systemu słonecznego ma na celu dostarczenie interaktywnego narzędzia edukacyjnego, pozwalającego użytkownikowi na eksplorację kosmosu w czasie rzeczywistym. Program umożliwia swobodne poruszanie się w przestrzeni trójwymiarowej, manipulację upływem czasu oraz obserwację ruchu planet po orbitach obliczonych na podstawie praw fizyki. W pracy omówiono proces projektowania silnika graficznego od podstaw w języku C, implementację własnej biblioteki matematycznej, obsługę shaderów oraz stworzenie autorskiego formatu plików modeli (.orb) w celu optymalizacji ładowania danych.

Słowa kluczowe

C, GLSL, OpenGL, GLFW, GLEW, Git, Make, Symulacja fizyczna

Spis treści

Wstęp 5

1. Analiza tematu i specyfikacja wymagań 6
1.1. Podstawy teoretyczne 6
1.2. Przegląd istniejących rozwiązań 7
1.3. Słownik pojęć 7
1.4. Wymagania funkcjonalne 9
1.5. Wymagania niefunkcjonalne 9
1.6. Założenia techniczne projektu 10

2. Środowisko i narzędzia wytwórcze 11
2.1. Język programowania C 11
2.2. Biblioteka graficzna OpenGL 4.1 Core Profile 12
2.3. Biblioteki pomocnicze 13
2.3.1. GLFW 13
2.3.2. GLEW / GLCoreARB 13
2.3.3. stb_image 13
2.4. Narzędzia budowania i systemy kontroli wersji 14
2.4.1. Make 14
2.4.2. Git 14

3. Architektura systemu i przepływ danych 15
3.1. Struktura kontekstu programu (`static struct context`) 15
3.1.1. Centralny punkt dostępu do danych 15
3.1.2. Separacja domen: podział na niezależny podsystem graficzny i logiczny . . . 16
3.2. Cykl życia aplikacji 17
3.2.1. Faza inicjalizacji 18
3.2.2. Pętla główna 18
3.2.3. Zarządzanie pamięcią 19

4. Specyfikacja warstwy technologicznej 20
4.1. Dedykowana biblioteka matematyczna (`r_math`) 20
4.1.1. Implementacja typów wektorowych i macierzowych 20
4.1.2. Kluczowe algorytmy transformacji 21
4.2. Podsystem renderowania (`r_renderer`) 22

4.2.1. Abstrakcja obiektu graficznego (`object_t`) . . . 22
4.2.2. Zarządzanie buforami OpenGL . . . 22
4.2.3. Potok cieniujący (Shading Pipeline) . . . 23
4.3. System zarządzania zasobami 3D . . . 24
4.3.1. Dedykowany format pliku `.orb` . . . 24
4.3.2. Narzędzie `otob` . . . 26

5. Implementacja logiki symulacji . . . 27
5.1. Fizyka i model danych ciał niebieskich (`r_physics`) . . . 27
5.1.1. Struktura planety (`planet_t`) . . . 27
5.1.2. Integracja z grafiką . . . 28
5.1.3. Algorytm pozycyjny . . . 29
5.2. System czasu i kalendarza . . . 29
5.2.1. Obsługa czasu rzeczywistego i symulowanego . . . 29
5.2.2. Mechanizm sterowania upływem czasu . . . 30
5.3. Interfejs i interakcja w przestrzeni 3D . . . 31
5.3.1. Wizualizacja orbit . . . 31
5.3.2. System markerów i nawigacja . . . 32

Zakończenie . . . 33

Spis tabel . . . 37

Spis rysunków . . . 38

Wstęp

Grafika komputerowa i symulacje fizyczne stanowią jeden z najbardziej dynamicznie rozwijających się obszarów informatyki. Choć współczesny rynek oprogramowania zdominowany jest przez zaawansowane silniki gier (takie jak Unity czy Unreal Engine), które automatyzują większość procesów renderowania, dogłębne zrozumienie zasad działania potoku graficznego wymaga zejścia do poziomu niskopoziomowego programowania.

Celem niniejszej pracy inżynierskiej jest zaprojektowanie i implementacja interaktywnej wizualizacji Układu Słonecznego („SOLAR”), wykonanej w języku C z wykorzystaniem biblioteki graficznej OpenGL. Projekt ten nie jest jedynie wizualizacją astronomiczną, lecz przede wszystkim próba stworzenia autorskiego, minimalistycznego silnika graficznego. Kluczowym założeniem pracy jest rezygnacja z gotowych bibliotek wspomagających obliczenia matematyczne (takich jak GLM) na rzecz własnej implementacji algebry liniowej oraz stworzenie dedykowanego formatu plików modeli 3D (.orb), zoptymalizowanego pod kątem czasu ładowania i zużycia pamięci.

Wybór języka C podyktowany był chęcią uzyskania pełnej kontroli nad zarządzaniem pamięcią oraz wydajnością aplikacji, co jest kluczowe w systemach czasu rzeczywistego. Aplikacja umożliwia użytkownikowi swobodną eksplorację przestrzeni kosmicznej, manipulację upływem czasu oraz obserwację ruchu planet po orbitach wyznaczonych zgodnie z prawami mechaniki nieba.

Praca została podzielona na sześć rozdziałów. W rozdziale pierwszym przedstawiono podstawy teoretyczne, analizę istniejących rozwiązań oraz specyfikację wymagań projektowych. Rozdział drugi charakteryzuje środowisko programistyczne i uzasadnia wybór technologii (język C, OpenGL). Rozdział trzeci omawia ogólną architekturę systemu, cykl życia aplikacji oraz organizację struktur danych. Rozdział czwarty stanowi dokumentację techniczną warstwy silnika, obejmującą autorską bibliotekę matematyczną, podsystem renderowania oraz zarządzanie zasobami. Całość zamyka rozdział piąty poświęcony implementacji właściwej symulacji, w tym modelowi fizycznemu ciał niebieskich, systemowi czasu oraz interfejsowi użytkownika.

[1, 2, 3, 4]

ROZDZIAŁ 1

Analiza tematu i specyfikacja wymagań

W niniejszym rozdziale przedstawiono teoretyczne podstawy zagadnienia, analizę dostępnych na rynku rozwiązań oraz zdefiniowano wymagania stawiane tworzonej aplikacji.

1.1. Podstawy teoretyczne

Projekt łączy w sobie zagadnienia z dwóch dziedzin: mechaniki nieba oraz grafiki komputerowej. W kontekście astronomicznym, ruch ciał niebieskich w symulacji opiera się na uproszczonym modelu keplerowskim. Pozycja planety w przestrzeni trójwymiarowej wyznaczana jest na podstawie parametrów orbitalnych, takich jak: promień orbity, okres obiegu, inklinacja (nachylenie orbity względem płaszczyzny odniesienia) oraz długość węzła wstępującego.

[RYSUNEK 1.1]

W aspekcie graficznym, aplikacja wykorzystuje programowalny potok renderujący (Programmable Pipeline) biblioteki OpenGL. Kluczową rolę odgrywają tu shadery, czyli małe programy wykonywane na procesorze graficznym (GPU), odpowiedzialne za przekształcanie wierzchołków (Vertex Shader) oraz ustalanie koloru pikseli (Fragment Shader).

[RYSUNEK 1.2] [Rysunek 1.3]

1.2. Przegląd istniejących rozwiązań

Na rynku istnieje wiele zaawansowanych symulatorów kosmosu, takich jak:

1.3. Słownik pojęć

1. **Epoka J2000:** standardowa epoka astronomiczna używana jako punkt odniesienia dla współrzędnych niebieskich. W projekcie odpowiada dacie 1 stycznia 2000, godzina 12:00 czasu ziemskiego (TT). W kodzie symulacji (`r_physics.c`) czas liczony jest jako liczba sekund, które upłynęły od tego momentu.

2. **Pętla główna:** fundamentalny wzorzec architektoniczny w silnikach czasu rzeczywistego, polegający na cyklicznym wykonywaniu operacji obsługi wejścia, aktualizacji stanu logicznego (Update) oraz renderowania obrazu (Render). W projekcie realizowana przez funkcję `g_game_update()` w pętli while (`!glfwWindowShouldClose()`).

3. **Inklinacja (Nachylenie orbity):** jeden z elementów orbitalnych Keplera, określający kąt pomiędzy płaszczyzną orbity ciała niebieskiego a płaszczyzną odniesienia (w tym przypadku ekliptyką).

4. **Kwaterion:** czterowymiarowa liczba zespolona używana w grafice komputerowej do reprezentacji obrotów w przestrzeni 3D.

5. **Mapa sześcienna (Cube Map):** rodzaj tekstury składającej się z sześciu kwadratowych obrazów, reprezentujących ściany sześcianu otaczającego obserwatora. W projekcie wykorzystywana do renderowania tła (Skybox), obsługiwana w shaderze przez typ zmiennej `samplerCube`.

6. **Macierz widoku (View Matrix):** macierz 4x4 transformująca współrzędne ze świata globalnego (World Space) do przestrzeni kamery (View Space). Generowana przez funkcję `r_look_at()` na podstawie pozycji obserwatora i punktu skupienia wzroku.

7. **Macierz projekcji (Projection Matrix):** macierz 4x4 odpowiedzialna za rzutowanie perspektywiczne przestrzeni trójwymiarowej na dwuwymiarową płaszczyznę ekranu, z uwzględnieniem kąta widzenia (FOV) i proporcji obrazu. Generowana przez funkcję `r_perspective()`.

8. **Siatka (Mesh):** zbiór wierzchołków, krawędzi i ścian definiujący kształt obiektu 3D. W strukturze silnika (`mesh_t`) składa się z buforów pozycji, normalnych oraz współrzędnych tekstury.

9. **ORB:** binarny format plików modeli 3D zaprojektowany na potrzeby projektu. Przechowuje zrzut pamięci struktur wierzchołków i materiałów, co eliminuje konieczność parsowania tekstu w czasie rzeczywistym. Pliki te tworzone są przez narzędzie `otob`.

10. **otob (Object to Binary):** narzędzie pomocnicze zaimplementowane w języku C, służące do konwersji modeli z formatu Wavefront OBJ do formatu binarnego ORB. Realizuje deduplikację wierzchołków przy użyciu tablicy haszującej.

11. **Program cieniujący (Shader):** krótki program wykonywany na procesorze graficznym (GPU). W projekcie wykorzystywane są shadery wierzchołków (`.vs`) do transformacji geometrii oraz shadery fragmentów (`.fs`) do obliczania koloru pikseli.

12. **Zmienna uniform:** rodzaj zmiennej w języku GLSL, która jest stała dla wszystkich wierzchołków/fragmentów w ramach jednego wywołania rysowania (np. macierze transformacji, pozycja światła).

13. **VAO (Vertex Array Object):** obiekt OpenGL przechowujący konfigurację atrybutów wierzchołków (takich jak format danych, przesunięcia w pamięci). Pozwala na szybkie przełączanie między różnymi ustawieniami buforów bez konieczności ich ponownego definiowania w każdej klatce.

14. **VBO (Vertex Buffer Object):** bufor pamięci na karcie graficznej, przechowujący surowe dane wierzchołków (pozycje, normalne, UV).

15. **Węzeł wstępujący (Ascending node):** punkt na orbicie, w którym ciało niebieskie przecina płaszczyznę odniesienia, przechodząc z półkuli południowej na północną. Jeden z parametrów pozycjonujących planetę w przestrzeni 3D (pole node w strukturze `planet_t`).

1.4. Wymagania funkcjonalne

1. **Symulacja układu planetarnego:** System musi wizualizować Słońce oraz 9 planet (wliczając Plutona) wraz z ich ruchem obrotowym i orbitalnym.
2. **Interaktywna kamera:** Użytkownik musi mieć możliwość swobodnego przemieszczania się w przestrzeni za pomocą myszy i klawiatury (sterowanie typu *first-person*).
3. **Manipulacja czasem:** Użytkownik może przyspieszać czas (do skali lat na sekundę), zatrzymywać go oraz cofać, obserwując zmiany położenia planet.
4. **System etykiet i orbit:** Ze względu na skalę kosmiczną, aplikacja musi renderować linie orbit oraz znaczniki (*markery*) wskazujące pozycje planet, nawet gdy ich rozmiar geometryczny jest mniejszy niż jeden piksel na ekranie.
5. **Obsługa układów satelitarnych:** System musi obsługiwać hierarchiczne relacje *parent-child* (np. Księżyc krążący wokół Ziemi, która krąży wokół ciała centralnego).
6. **Oświetlenie:** Obiekty muszą być cieniowane dynamicznie (np. z wykorzystaniem modelu oświetlenia Phonga) w zależności od ich położenia względem punktowego źródła światła (Słońca).

1.5. Wymagania niefunkcjonalne

1. **Wydajność:** Aplikacja musi utrzymywać płynność animacji na poziomie co najmniej 60 klatek na sekundę (FPS) przy renderowaniu pełnego układu słonecznego. Aktualny pomiar FPS ma być prezentowany użytkownikowi w pasku tytułu okna aplikacji w celu monitorowania obciążenia sprzętowego.
2. **Zgodność ze standardami:** Kod źródłowy projektu musi być zgodny ze standardem języka C99 oraz normą POSIX, co ma zapewnić przenośność (*portability*) między systemami operacyjnymi z rodziny Unix.
3. **Efektywność ładowania danych:** Dzięki zastosowaniu binarnego formatu danych (`.orb`), czas wczytywania geometrii i inicjalizacji sceny powinien być zminimalizowany poprzez wyeliminowanie kosztownego parsowania tekstowego na rzecz bezpośredniego odczytu struktur do pamięci.
4. **Zarządzanie pamięcią:** Aplikacja musi realizować ścisłą kontrolę alokacji pamięci operacyjnej i wideo, nie dopuszczając do wycieków pamięci (*memory leaks*). Wszystkie zaalokowane zasoby (tekstury, bufory VBO/VAO, programy cieniujące) muszą być poprawnie zwalniane w fazie zamykania programu.
5. **Modularność architektury:** System musi realizować separację warstwy renderowania od warstwy logicznej symulacji (zgodnie z wzorcem architektonicznym separacji odpowiedzialności), umożliwiając niezależny rozwój i testowanie obu modułów.

1.6. Założenia techniczne projektu

Decyzje technologiczne podjęte w projekcie wynikają z chęci poznania niskopoziomowych mechanizmów działania komputerów.

Język programowania C: Wybór języka C podyktowany jest koniecznością ręcznego zarządzania pamięcią (malloc/free) oraz operowania na wskaźnikach, co jest kluczowe dla wydajnej współpracy z sterownikami kart graficznych. Zrezygnowano z obiektowości C++ na rzecz struktur danych i funkcji operujących na stanie.

OpenGL 4.1 Core Profile: Użycie nowoczesnego profilu OpenGL wymusza stosowanie shaderów i buforów VBO/VAO, eliminując przestarzałe funkcje (tzw. immediate mode). Zapewnia to zgodność z nowoczesnymi kartami graficznymi oraz systemem macOS.

Dedykowany format .orb: Zamiast parsować tekstowe pliki .obj przy każdym uruchomieniu (co jest wolne i wymaga wielu alokacji pamięci), zaprojektowano format binarny .orb. Zawiera on gotowe zrzuty pamięci struktur wierzchołków, co pozwala na ich bezpośrednie wczytanie do RAM jednym wywołaniem fread.

Brak bibliotek matematycznych: Zamiast popularnej biblioteki GLM, wszystkie operacje na macierzach 4x4 i wektorach (mnożenie, projekcja perspektywiczna) zaimplementowano własnoręcznie w module `r_math`.

ROZDZIAŁ 2

Środowisko i narzędzia wytwórcze

Wybór odpowiedniego zestawu technologii i narzędzi programistycznych jest kluczowym etapem realizacji każdego projektu inżynierskiego. W przypadku silnika graficznego i symulacji fizycznej, priorytetem była wydajność, przenośność oraz możliwość bezpośredniej interakcji ze sprzętem graficznym. W niniejszym rozdziale omówiono język programowania, biblioteki oraz narzędzia wykorzystane w procesie tworzenia aplikacji „SOLAR".

2.1. Język programowania C

Jako podstawowe narzędzie implementacji wybrano język C w standardzie C99. Decyzja ta, choć może wydawać się nietypowa w dobie wysokopoziomowych języków obiektowych, była podyktowana specyficznymi wymaganiami inżynierii gier i symulacji fizycznych. Język C oferuje bezkompromisową wydajność oraz niskopoziomowy dostęp do pamięci operacyjnej, co jest krytyczne przy przetwarzaniu tysięcy wierzchołków i obliczaniu fizyki orbitalnej w każdej klatce obrazu. W przeciwieństwie do języków wyposażonych w automatyczne odśmiecanie pamięci (Garbage Collection), C wymusza na programiście ręczne zarządzanie alokacją i dealokacją zasobów. W projekcie wykorzystano to do stworzenia dedykowanych struktur danych, takich jak dynamiczne tablice zaimplementowane w module `d_array`, które minimalizują narzut pamięciowy i fragmentację sterty. Ponadto, API biblioteki OpenGL zostało pierwotnie zaprojektowane właśnie dla języka C, co eliminuje konieczność stosowania dodatkowych warstw abstrakcji (wrapperów) i pozwala na bezpośrednią interakcję ze sterownikami karty graficznej.

[RYSUNEK 2.1]

2.2. Biblioteka graficzna OpenGL 4.1 Core Profile

Warstwa wizualna aplikacji została oparta na bibliotece OpenGL w wersji 4.1 Core Profile. Jest to otwarty standard interfejsu programistycznego, służący do generowania grafiki dwu- i trójwymiarowej. Wybór wersji 4.1 stanowił świadomy kompromis pomiędzy dostępem do nowoczesnych funkcji a kompatybilnością sprzętową. Wersja ta jest ostatnią natywnie wspieraną przez systemy operacyjne macOS, co zapewnia przenośność aplikacji między platformami Windows, Linux i Apple bez konieczności modyfikacji kodu źródłowego. Zastosowanie profilu rdzennego (Core Profile) oznacza całkowitą rezygnację z przestarzałego modelu ”Fixed Function Pipeline”na rzecz w pełni programowalnego potoku renderującego. Wymusiło to implementację własnych shaderów w języku GLSL (OpenGL Shading Language), odpowiedzialnych za transformację geometrii oraz obliczanie oświetlenia pikseli bezpośrednio na procesorze graficznym (GPU). Takie podejście pozwala na uzyskanie znacznie wyższej wydajności i elastyczności w kreowaniu efektów wizualnych niż w przypadku gotowych rozwiązań.

[RYSUNEK 2.2]

2.3. Biblioteki pomocnicze

OpenGL jest jedynie specyfikacją API graficznego i nie zawiera funkcji do obsługi systemu operacyjnego (tworzenie okna, obsługa klawiatury) ani ładowania plików. W związku z tym wykorzystano zestaw lekkich, specjalizowanych bibliotek.

2.3.1. GLFW

Biblioteka GLFW posłużyła do utworzenia kontekstu OpenGL, zarządzania oknem aplikacji oraz obsługi urządzeń wejścia (mysz, klawiatura). W pliku `g_game.c` wykorzystano funkcje takie jak glfwGetCursorPos do sterowania kamerą oraz glfwGetKey do obsługi poruszania się w przestrzeni. GLFW zapewnia abstrakcję warstwy systemowej, dzięki czemu aplikacja może być kompilowana zarówno na systemach Windows, Linux, jak i macOS.

2.3.2. GLEW / GLCoreARB

Ze względu na specyfikę ładowania funkcji OpenGL, które są dostarczane przez sterowniki karty graficznej, konieczne jest użycie mechanizmu ładującego wskaźniki do tych funkcji.

- Na systemach Windows/Linux wykorzystano bibliotekę GLEW
- Na systemie macOS (zdefiniowanym makrem __APPLE__) wykorzystano nagłówki GLCoreARB

2.3.3. stb_image

Do wczytywania tekstur (formaty .jpg, .png) użyto biblioteki `stb_image`. Jest to biblioteka typu ”header-only”(zawarta w jednym pliku nagłówkowym), co znacząco upraszcza proces kompilacji. W module `d_util.c` funkcja stbi_load dekoduje pliki graficzne do surowej tablicy bajtów, która następnie jest przesyłana do pamięci karty graficznej funkcją glTexImage2D.

2.4. Narzędzia budowania i systemy kontroli wersji

Proces wytwórczy oprogramowania wspierany był przez standardowe narzędzia inżynierskie.

2.4.1. Make

Do automatyzacji procesu kompilacji wykorzystano narzędzie Make. Plik konfiguracyjny Makefile definiuje reguły kompilacji poszczególnych modułów (.c) do plików obiektowych (.o), a następnie ich linkowania w plik wykonywalny. Pozwala to na szybką rekompilację tylko tych części kodu, które uległy zmianie, oraz łatwe zarządzanie flagami kompilatora i linkera (np. -lGL -lglfw).

2.4.2. Git

Zarządzanie kodem źródłowym odbywało się przy użyciu systemu kontroli wersji Git. Umożliwiło to śledzenie historii zmian, eksperymentowanie z nowymi funkcjami (np. implementacja formatu .orb) na osobnych gałęziach (branch) oraz zabezpieczenie kodu przed utratą. Struktura repozytorium została podzielona na katalogi logiczne: src (kod źródłowy), assets (zasoby), shader (kody shaderów) oraz tools (narzędzia pomocnicze).

ROZDZIAŁ 3

Architektura systemu i przepływ danych

Projekt aplikacji został oparty na modularnej architekturze, charakterystycznej dla silników czasu rzeczywistego napisanych w języku C. Kluczowym założeniem projektowym było rozdzielenie warstwy danych symulacyjnych od warstwy prezentacji graficznej oraz centralizacja zarządzania stanem aplikacji. W niniejszym rozdziale omówiono strukturę danych, przepływ sterowania oraz metody zarządzania pamięcią.

3.1. Struktura kontekstu programu (`static struct context`)

Fundamentem architektury oprogramowania jest statyczna struktura danych context, zdefiniowana w pliku nagłówkowym `g_game.h`. Pełni ona rolę głównego kontenera stanu aplikacji, agregując w sobie podsystemy odpowiedzialne za okno, wejście, renderowanie oraz logikę sceny.

3.1.1. Centralny punkt dostępu do danych

Zdecydowano się na wykorzystanie wzorca zbliżonego do Singletona, gdzie instancja context jest zmienną statyczną dostępną w obrębie modułu gry, a dostęp do niej odbywa się bezpośrednio, bez konieczności przekazywania wskaźników do każdej funkcji pomocniczej. Takie podejście upraszcza sygnatury funkcji i przyspiesza dostęp do kluczowych zmiennych w pętli głównej.

Struktura dzieli się na logiczne sekcje:

— **fps:** Przechowuje dane diagnostyczne o wydajności, takie jak czas między klatkami (delta time) i liczbę klatek na sekundę.

— **camera:** Zawiera wektory pozycji i orientacji obserwatora oraz parametry sterowania (czułość myszy, prędkość ruchu).

— **renderer:** Grupuje zasoby niskopoziomowe, takie jak shadery, tekstury i bufory obiektów graficznych.

— **scene:** Przechowuje wysokopoziomowy stan symulacji, w tym obiekty ciał niebieskich (planets), zegar symulacji (clock) oraz elementy interfejsu (ui).

[RYSUNEK 3.1]

3.1.2. Separacja domen: podział na niezależny podsystem graficzny i logiczny

Jednym z najważniejszych założeń architektonicznych projektu jest ścisła separacja danych graficznych od danych logicznych. Zrealizowano to poprzez podział na struktury renderer oraz scene.

Podsystem graficzny (`struct renderer`) operuje na abstrakcji obiektu renderowalnego object_t. Obiekt ten jest kontenerem przechowującym surowe dane geometryczne (siatkę `mesh_t`), właściwości materiału (`material_t`) oraz wskaźniki do zasobów OpenGL (shadery, tekstury). Renderer iteruje po tablicy obiektów i rysuje te, które posiadają flagę visible ustawioną na wartość pozytywną, nie posiadając wiedzy na temat tego, co dany obiekt reprezentuje w świecie symulacji.

Podsystem logiczny (`struct scene`) operuje na obiektach typu `planet_t` (zdefiniowanych w `r_physics.h`). Struktura ta zawiera parametry fizyczne i orbitalne, takie jak masa, promień, inklinacja czy węzeł wstępujący. Połączenie między tymi dwoma światami realizowane jest wyłącznie poprzez wskaźnik `object_t` *object zawarty w strukturze planety. Dzięki temu logika symulacji oblicza pozycję ciała niebieskiego, a następnie aktualizuje macierz modelu w powiązanym obiekcie graficznym, nie ingerując w proces samego rysowania.

3.2. Cykl życia aplikacji

Aplikacja działa w oparciu o klasyczny schemat pętli gry (Game Loop), sterowany z poziomu funkcji main, która sekwencyjnie wywołuje trzy główne procedury: inicjalizację (`g_game_init()`), pętlę aktualizacji (`g_game_update()`) oraz czyszczenie zasobów (`g_game_stop()`).

[RYSUNEK 3.2]

3.2.1. Faza inicjalizacji

Funkcja `g_game_init()` odpowiada za przygotowanie środowiska pracy przed wejściem w pętlę główną. Proces ten przebiega etapowo:

1. **Inicjalizacja kontekstu OpenGL:** Konfiguracja biblioteki GLFW, utworzenie okna i załadowanie wskaźników na funkcje OpenGL za pomocą GLEW.
2. **Kompilacja shaderów:** Wczytanie kodu źródłowego shaderów wierzchołków (_vs) i fragmentów (_fs), ich kompilacja oraz linkowanie do programów GPU.
3. **Ładowanie zasobów:** Import tekstur za pomocą biblioteki `stb_image` oraz wczytanie modeli 3D z dedykowanego formatu binarnego .orb do pamięci operacyjnej.
4. **Konfiguracja sceny:** Inicjalizacja struktur planetarnych, powiązanie ich z obiektami graficznymi oraz wygenerowanie geometrii orbit i markerów.

3.2.2. Pętla główna

Centralnym elementem sterującym jest funkcja `g_game_update()`. W każdej iteracji pętli while (!glfwWindowShouldClose) wykonywane są następujące kroki:

1. **Obsługa wejścia:** Odczyt pozycji myszy i stanu klawiatury w celu aktualizacji kamery oraz sterowania czasem (funkcje `g_game_handle_mouse()` i `g_game_handle_keyboard()`).
2. **Aktualizacja logiczna:** Przeliczenie czasu symulacji (`g_game_clock_update()`) oraz aktualizacja stanu fizycznego planet (`r_physics_planet_state_update()`). W tym kroku obliczane są nowe współrzędne kartezjańskie na podstawie elementów orbitalnych.
3. **Renderowanie:** Wyczyszczenie buforów ekranu, ustawienie zmiennych uniform (macierze projekcji i widoku), a następnie rysowanie obiektów sceny: planet, orbit, markerów oraz tła (Skybox).

3.2.3. Zarządzanie pamięcią

Ze względu na wymóg ręcznego zarządzania pamięcią w języku C, w projekcie zaimplementowano własny mechanizm dynamicznych tablic. Biblioteka `d_array.h` definiuje generyczne kontenery (np. `uint32_array_t`), które automatycznie zwiększają swoją pojemność w miarę dodawania nowych elementów.

Funkcja wstawiająca element sprawdza, czy aktualny rozmiar (size) osiągnął pojemność (capacity). W przypadku braku miejsca, pojemność jest podwajana, a blok pamięci realokowany za pomocą funkcji `realloc()`:

[RYSUNEK 3.3]

Taki mechanizm zapewnia elastyczność przy ładowaniu nieznanej z góry liczby wierzchołków z plików modeli. Zwalnianie zasobów odbywa się w funkcji `g_game_stop()`, która usuwa bufory OpenGL i zwalnia zaalokowaną pamięć RAM, zapobiegając wyciekom pamięci.

ROZDZIAŁ 4

Specyfikacja warstwy technologicznej

Silnik graficzny aplikacji został zaimplementowany od podstaw, co wymagało opracowania własnych rozwiązań w zakresie algebry liniowej, obsługi potoku OpenGL oraz zarządzania zasobami. Poniższy rozdział stanowi dokumentację techniczną kluczowych modułów technologicznych, stanowiących fundament dla warstwy symulacyjnej.

4.1. Dedykowana biblioteka matematyczna (`r_math`)

W celu uniknięcia zewnętrznych zależności (takich jak biblioteka GLM) oraz pełnego zrozumienia transformacji geometrycznych zachodzących w potoku graficznym, zaimplementowano autorski moduł matematyczny `r_math`. Biblioteka ta operuje bezpośrednio na surowych strukturach danych, co zapewnia kompatybilność binarną z układem pamięci oczekiwanym przez shadery GLSL.

4.1.1. Implementacja typów wektorowych i macierzowych

Podstawą biblioteki są struktury wektorowe `vec2_t`, `vec3_t` oraz `vec4_t`, reprezentujące odpowiednio punkty w przestrzeni 2D (współrzędne tekstur), 3D (pozycje, normalne) oraz 4D (kwaterniony, wektory jednorodne). Operacje arytmetyczne na wektorach (dodawanie, odejmowanie, mnożenie przez skalar) zaimplementowano w formie makr preprocesora (np. `#define vec3_add()`), co pozwala na agresywną optymalizację kodu przez kompilator i unikanie narzutu wywoływania funkcji dla prostych operacji matematycznych.

Macierze 4x4 zdefiniowano jako strukturę `mat4_t` zawierającą dwuwymiarową tablicę `float m`. Układ danych jest zgodny z formatem column-major stosowanym przez OpenGL, co umożliwia bezpośrednie przesyłanie macierzy do shaderów funkcją `glUniformMatrix4fv()` bez konieczności ich transpozycji w locie.

4.1.2. Kluczowe algorytmy transformacji

Moduł `r_math.c` implementuje funkcje niezbędne do przekształcania współrzędnych z przestrzeni modelu do przestrzeni ekranu:

— **Macierz widoku (`r_look_at()`):** Konstruuje macierz transformacji kamery poprzez wyznaczenie ortonormalnej bazy wektorów: forward (kierunek patrzenia), right (iloczyn wektorowy forward i wektora up) oraz up (iloczyn wektorowy right i forward).

— **Macierz projekcji (`r_perspective()`):** Generuje macierz rzutowania perspektywicznego, definiującą frustum widzenia (bryłę widokową). Funkcja przyjmuje kąt widzenia (FOV), proporcje ekranu (aspect ratio) oraz płaszczyzny obcinania near i far.

— **Operacje na kwaternionach:** Do reprezentacji orientacji obiektów wykorzystano kwaterniony (`quat_t`), co eliminuje problem blokady Gimbala (Gimbal Lock), charakterystyczny dla kątów Eulera. Funkcja `r_rotate_quat()` konwertuje kwaternion rotacji na macierz 4x4, gotową do mnożenia z macierzą modelu.

[RYSUNEK 4.1]

4.2. Podsystem renderowania (`r_renderer`)

Moduł renderera stanowi warstwę abstrakcji nad surowym API OpenGL 4.1. Jego zadaniem jest zarządzanie cyklem życia obiektów graficznych oraz optymalizacja wywołań rysowania.

4.2.1. Abstrakcja obiektu graficznego (`object_t`)

Centralną strukturą renderera jest `object_t`, zdefiniowana w pliku `r_renderer.h`. Stanowi ona niezależny kontener danych, całkowicie odseparowany od logiki gry. Struktura ta agreguje:

— **Siatkę (`mesh_t`):** Zawiera bufory wierzchołków i indeksów oraz identyfikatory obiektów OpenGL (VAO, VBO, IBO).

— **Materiał (`material_t`):** Przechowuje właściwości optyczne powierzchni, takie jak współczynniki odbicia światła (ambient, diffuse, specular) oraz emisyjność.

— **Transformację (`transform_t`):** Lokalna pozycja, rotacja i skala obiektu.

— **Zasoby:** Wskaźniki do używanego programu cieniującego (`shader_t`) oraz tekstury.

4.2.2. Zarządzanie buforami OpenGL

Proces przesyłania geometrii do karty graficznej realizuje funkcja `r_renderer_object_upload()`. Wykorzystuje ona mechanizm Vertex Array Object (VAO) do zapamiętania konfiguracji atrybutów wierzchołków. Dane wierzchołków są przesyłane do bufora VBO w układzie interleaved (przeplatanym), gdzie w jednej strukturze `vertex_t` sąsiadują ze sobą:

1. **Pozycja (3 x `float`):** `layout (location = 0)`
2. **Normalna (3 x `float`):** `layout (location = 1)`
3. **Współrzędne UV (2 x `float`):** `layout (location = 2)`

Takie ułożenie danych w pamięci zwiększa lokalność odwołań cache GPU. Do rysowania wykorzystywany jest bufor indeksów (IBO) oraz funkcja `glDrawElements()`, co pozwala na redukcję liczby przetwarzanych wierzchołków poprzez ich współdzielenie między trójkątami.

[RYSUNEK 4.2]

4.2.3. Potok cieniujący (Shading Pipeline)

System wykorzystuje programowalne shadery napisane w języku GLSL 4.10. Zaimplementowano dwa główne potoki renderowania:

1. **Rendering obiektów (Default Shader):** Shader fragmentów (`default.fs`) implementuje model oświetlenia Phonga/Blinna. Kolor finalny piksela obliczany jest na podstawie tekstury albedo (`u_Texture`) oraz kąta padania światła (iloczyn skalarny wektora normalnego i wektora światła). Dodatkowo obsłużono parametr `u_Emissive` (uniform typu int), który pozwala na renderowanie obiektów świecących własnym światłem (Słońce), ignorując obliczenia cieniowania.

2. **Rendering tła (Skybox):** Do renderowania przestrzeni kosmicznej wykorzystano technikę Cube Mapping. Shader `skybox.fs` pobiera kolor z tekstury sześciennej (samplerCube) na podstawie wektora pozycji wierzchołka. W fazie rysowania (`g_game.c`), macierz widoku jest modyfikowana poprzez wyzerowanie składowych translacji, co sprawia, że tło wydaje się być nieskończenie odległe względem obserwatora.

4.3. System zarządzania zasobami 3D

W celu optymalizacji procesu ładowania aplikacji oraz zmniejszenia zużycia pamięci, zrezygnowano z bezpośredniego wczytywania formatów tekstowych (takich jak OBJ) w czasie rzeczywistym na rzecz własnego formatu binarnego oraz dedykowanego konwertera.

4.3.1. Dedykowany format pliku `.orb`

Format `.orb` został zaprojektowany jako zrzut pamięci struktur silnika, co eliminuje konieczność kosztownego parsowania tekstu (`atoi()`, `atof()`) podczas uruchamiania gry. Plik składa się z trzech sekcji:

1. **Nagłówek:** Dwie 4-bajtowe liczby całkowite określające liczbę wierzchołków oraz liczbę indeksów.
2. **Dane siatki:** Bezpośredni zrzut tablicy struktur `vertex_t` oraz tablicy indeksów `uint32_t`.
3. **Definicja materiału:**: Blok danych odpowiadający strukturze `material_t` (współczynniki oświetlenia, przezroczystość, gęstość).

Dzięki takiej strukturze, funkcja `r_renderer_object_read()` może wczytać cały model za pomocą zaledwie kilku wywołań `fread()`, co drastycznie skraca czas inicjalizacji sceny.

[RYSUNEK 4.3]

4.3.2. Narzędzie `otob`

Narzędzie `otob` jest niezależnym programem konsolowym służącym do konwersji modeli Wavefront OBJ na format .orb. Kluczowym wyzwaniem implementacyjnym była różnica w sposobie indeksowania: format OBJ stosuje oddzielne indeksy dla pozycji, normalnych i UV, podczas gdy OpenGL wymaga jednego indeksu per wierzchołek (unikalna kombinacja wszystkich atrybutów).

Algorytm konwersji wykorzystuje autorską implementację tablicy haszującej (`map_t`) z algorytmem haszującym FNV-1a.

1. Program parsuje ściany (`f`) pliku OBJ, odczytując tryplety indeksów (`v/vt/vn`).
2. Dla każdego trypletu sprawdzane jest, czy taka kombinacja występuje już w mapie. Kluczem mapy jest wektor `vec3_t` zawierający indeksy z pliku OBJ.
3. Jeśli kombinacja jest nowa, tworzony jest nowy wierzchołek `vertex_t` (poprzez scalenie danych z osobnych tablic źródłowych), dodawany jest do tablicy wierzchołków, a jego nowy indeks zapisywany jest w mapie.
4. Jeśli kombinacja istnieje, do bufora indeksów (IBO) dodawany jest odzyskany z mapy indeks istniejącego wierzchołka.

Taki proces zapewnia deduplikację wierzchołków, co jest kluczowe dla wydajności renderowania przy użyciu buforów indeksowych (`glDrawElements()`).

ROZDZIAŁ 5

Implementacja logiki symulacji

Niniejszy rozdział skupia się na implementacji zasad fizyki orbitalnej, systemie zarządzania czasem oraz mechanizmach interakcji użytkownika z wirtualnym Układem Słonecznym.

5.1. Fizyka i model danych ciał niebieskich (`r_physics`)

Model fizyczny aplikacji został zaprojektowany w oparciu o uproszczoną mechanikę nieba, wykorzystującą parametry orbitalne do deterministycznego wyznaczania pozycji ciał niebieskich w dowolnym momencie czasu.

5.1.1. Struktura planety (`planet_t`)

- **Dane identyfikacyjne i wizualne:** Nazwa planety (`name`) oraz kolor bazowy (`color`) wykorzystywany przez elementy interfejsu.
- **Elementy orbitalne Keplera:**
  - `orbit.radius`: Średnia odległość od ciała centralnego
  - `orbit.period`: Czas pełnego obiegu wokół orbity (rok gwiazdowy)
  - `orbit.phase`: Przesunięcie początkowe, pozwalające ustalić pozycję planety w epoce startowej (np. J2000)
  - `inclination`: Kąt nachylenia płaszczyzny orbity do płaszczyzny odniesienia (ekliptyki)
  - `node`: Kąt określający orientację orbity w płaszczyźnie odniesienia
- **Parametry rotacyjne:** `spin` (okres obrotu wokół własnej osi) oraz `tilt` (nachylenie osi obrotu).
- **Hierarchia:** Wskaźnik `struct planet *parent`, umożliwiający tworzenie układów zależnych (np. Księżyc krążący wokół Ziemi, która krąży wokół Słońca).

[RYSUNEK 5.1]

5.1.2. Integracja z grafiką

Kluczowym aspektem architektury jest sposób połączenia warstwy fizycznej z warstwą prezentacji. Zamiast dziedziczenia, zastosowano kompozycję: struktura `planet_t` zawiera wskaźnik `object_t *object`.

Obiekt graficzny (`object_t`) przechowuje siatkę geometryczną, teksturę i shader, ale nie posiada wiedzy o swojej pozycji w kontekście astronomicznym. W każdej klatce symulacji, funkcja aktualizująca stan planety oblicza nową pozycję, a następnie modyfikuje macierz modelu (`u_Model`) powiązanego obiektu graficznego.

Dzięki temu rozwiązaniu, renderer pozostaje niezależnym modułem, który jedynie ”wyświetla”to, co obliczyła fizyka. Pozwala to również na łatwą zmianę reprezentacji wizualnej planety (np. podmianę modelu na dokładniejszy) bez ingerencji w kod fizyki.

5.1.3. Algorytm pozycyjny

Wyznaczanie pozycji ciał niebieskich realizowane jest przez funkcję `r_physics_planet_state_update()` oraz pomocniczą funkcję `r_physics_orbit_to_local()`. Proces ten przebiega w następujących krokach:

1. **Obliczenie anomalii średniej:** Na podstawie aktualnego czasu symulacji (`time`) wyznaczany jest kąt na orbicie (`orbit_angle`). Uwzględnia się przy tym okres orbitalny oraz fazę początkową.
2. **Wyznaczenie pozycji na płaszczyźnie orbity:** Współrzędne obliczane są przy założeniu orbity kołowej (dla uproszczenia mimośrodów bliskich zeru):

$$x = r \cdot \cos(\theta), \quad z = r \cdot \sin(\theta) \tag{5.1}$$

3. **Transformacja do układu lokalnego:** Pozycja jest poddawana obrotom zgodnie z parametrami orbitalnymi. Najpierw następuje obrót o kąt inklinacji wokół osi X, a następnie obrót o kąt węzła wstępującego wokół osi Y.
4. **Uwzględnienie hierarchii:** Jeśli planeta posiada rodzica (pole parent nie jest NULL), do obliczonej pozycji dodawany jest wektor pozycji rodzica. Pozwala to na poprawne symulowanie układów satelitarnych.

5.2. System czasu i kalendarza

Zarządzanie czasem jest krytycznym elementem symulacji astronomicznej, pozwalającym na obserwację zjawisk zachodzących w skali od minut do stuleci.

5.2.1. Obsługa czasu rzeczywistego i symulowanego

Wewnętrzny zegar symulacji (`context.scene.clock.time`) reprezentowany jest przez zmienną typu double, przechowującą liczbę sekund, jakie upłynęły od epoki J2000 (1 stycznia 2000, 12:00 TT). Taka reprezentacja zapewnia wysoką precyzję i ułatwia obliczenia astronomiczne.

W celu prezentacji daty użytkownikowi, zaimplementowano funkcję `r_physics_clock_to_tm()`, która konwertuje czas liniowy na strukturę kalendarzową `struct tm` (rok, miesiąc, dzień). Wykorzystano w tym celu standardowe funkcje biblioteki C (`mktime, localtime_r`), traktując rok 2000 jako punkt odniesienia.

5.2.2. Mechanizm sterowania upływem czasu

Aplikacja umożliwia dynamiczną zmianę prędkości upływu czasu. Zaimplementowano to przy użyciu tablicy prędkości `speeds` oraz kursora wskazującego na aktualnie wybrany mnożnik. Tablica zawiera wartości od ujemnych (cofanie czasu), przez zero (pauza), aż po wartości dodatnie symulujące upływ lat w ciągu sekund.

W każdej klatce pętli głównej, czas symulacji jest inkrementowany o wartość: `delta_time * speeds[cursor]`.

Sterowanie odbywa się za pomocą klawiszy klawiatury (H - zwolnienie/cofanie, K - przyspieszenie, J - reset do czasu rzeczywistego), co pozwala użytkownikowi na płynną nawigację w chronologii Układu Słonecznego.

5.3. Interfejs i interakcja w przestrzeni 3D

Ze względu na specyficzną skalę kosmosu (ogromne odległości i stosunkowo małe obiekty), konieczne było zaprojektowanie elementów wizualnych ułatwiających orientację w przestrzeni.

5.3.1. Wizualizacja orbit

Linie orbit nie są częścią modeli 3D, lecz są generowane proceduralnie podczas inicjalizacji sceny (`g_game_ui_orbits_init()`). Algorytm iteruje przez kąt od 0 do 2π z zadanym krokiem, wykorzystując tę samą funkcję fizyczną `r_physics_orbit_to_local()`, która służy do pozycjonowania planet.

Wygenerowane wierzchołki są przesyłane do osobnego bufora VBO i rysowane jako prymityw `GL_LINE_LOOP`. Dzięki temu orbity idealnie pokrywają się z trajektorią ruchu planet, uwzględniając ich inklinację i ekscentryczność. Do ich renderowania użyto dedykowanego shadera (`orbit.vs`, `orbit.fs`), który umożliwia nadanie im stałego koloru i jasności, niezależnie od oświetlenia słonecznego.

[RYSUNEK 5.2]

5.3.2. System markerów i nawigacja

Aby planety były widoczne nawet z dużej odległości, zastosowano system markerów. Są to proste okręgi rysowane wokół pozycji planety. Marker jest skalowany w taki sposób, aby był zawsze zauważalny, ale skalowanie to odbywa się w shaderze lub poprzez macierz modelu markera (`model_marker`), która jest niezależna od skali samej planety.

Nawigacja w przestrzeni trójwymiarowej została zrealizowana w modelu ”Free Look”(swobodna kamera).

1. **Mysz:** Ruch myszy modyfikuje kąty Eulera kamery (`yaw, pitch`), na podstawie których wyliczany jest wektor kierunku patrzenia (`target_position`).
2. **Klawiatura:** Klawisze W/S/A/D przesuwają pozycję kamery wzdłuż wektorów lokalnych (przód/tył oraz lewo/prawo), wyznaczanych jako iloczyn wektorowy kierunku patrzenia i wektora ”góry”(`head_position`). Klawisz SHIFT modyfikuje wartość `speed` kamery, pozwalając na zmianę prędkości poruszania.

Implementacja ta, zawarta w `g_game_handle_mouse()` i `g_game_handle_keyboard()`, pozwala na intuicyjne przemieszczanie się po Układzie Słonecznym i obserwację zjawisk z dowolnej perspektywy.

Zakończenie

Celem niniejszej pracy inżynierskiej było zaprojektowanie i implementacja interaktywnej wizualizacji Układu Słonecznego, działającej w czasie rzeczywistym w oparciu o język C oraz bibliotekę graficzną OpenGL 4.1. Zrealizowany projekt, obejmujący zarówno silnik graficzny, jak i warstwę symulacyjną, pozwolił na spełnienie wszystkich założonych wymagań funkcjonalnych i niefunkcjonalnych.

Możliwe kierunki rozwoju

- Implementacja cieni dynamicznych (Shadow Mapping)
- Rozszerzenie modelu fizycznego o grawitację n-ciał
- Proceduralne generowanie atmosfery
- System dźwięku przestrzennego
- Rozbudowa zawartości astronomicznej
- Implementacja wielowątkowości do ładowania zasobów

Bibliografia - książki

[1] Brian Schwab i Brian Schwab. *AI game engine programming*. Charles River Media Hingham, 2004.

Bibliografia - artykuły

[2] Marc LeBlanc i in. „Mechanics, dynamics, aesthetics: a formal approach to game design”. W: *lecture at Northwestern University* (2004).

Bibliografia - strony internetowe

[3] GameDev Glossary. *GameDev Glossary: Library Vs Framework Vs Engine*. Dostęp 12 grudnia 2022. 2015. URL: https://gamefromscratch.com/gamedev-glossary-library-vs-framework-%5C%5Cvs-engine/.

[4] Joey de Vries. *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL*. 2025. URL: https://learnopengl.com/.

Spis tabel

Spis rysunków

Spis listingów