WYŻSZA SZKOŁA INFORMATYKI I ZARZĄDZANIA „COPERNICUS" WE WROCŁAWIU

WYDZIAŁ INFORMATYKI, ADMINISTRACJI I FIZJOTERAPII

Kierunek: **Informatyka (INF)**
Specjalność: **Systemy i sieci komputerowe (SSK)**

PRACA DYPLOMOWA INŻYNIERSKA

Igor Sadza

Tworzenie aplikacji graficznych przy pomocy OpenGL

Creating graphic applications using OpenGL

Ocena pracy: .....
(ocena pracy dyplomowej, data, podpis promotora)

.....
(pieczątka uczelni)

Promotor:

dr Grzegorz Debita

WROCŁAW 2019

Spis treści

1 Wstęp **5**
1.1 Wprowadzenie . . . 5
1.2 Cel pracy i założenia . . . 5
1.3 Motywacja . . . 5
1.4 Zakres . . . 5

Część przeglądowa 7

2 Biblioteka OpenGL **9**
2.1 Przedstawienie biblioteki OpenGL . . . 9
2.1.1 Stan OpenGL . . . 10
2.1.2 Kontekst OpenGL . . . 10
2.1.3 Maszyna stanów . . . 10
2.2 Potok Graficzny . . . 10
2.2.1 Przetwarzanie wierzchołków . . . 11
2.2.2 Łączenie wierzchołków . . . 12
2.2.3 Rasteryzacja . . . 12
2.2.4 Cieniowanie pikseli . . . 12
2.2.5 Przetwarzanie próbek . . . 12
2.3 Obiekty OpenGL . . . 13
2.3.1 Bufory . . . 13
2.3.2 Textury . . . 13
2.4 Shadery OpenGL . . . 16
2.4.1 OpenGL Shading Language (GLSL) . . . 17

Część praktyczna 19

3 Opis wykorzystanych środowisk **21**
3.1 Środowisko wykonawcze . . . 21
3.2 Środowisko programistyczne . . . 21

4 Opis zrealizowanego systemu **23**
4.1 Dodatkowe biblioteki . . . 23
4.2 Obiekty . . . 23
4.3 Zarządzanie zasobami . . . 26

4.4 Renderowanie obiektów 27
4.5 Kolizje obiektów 29
4.6 Generator cząsteczek 29
4.7 Przedstawienie finalnego projektu 31

5 Podsumowanie **33**
5.1 Możliwość kontynuacji 33

Bibliografia **39**

Załączniki **41**
Zawartość płyty CD 41

Dodatki **43**
Dodatek A-Ogólny format plików inicjalizujących 43
Dodatek B-Przedstawienie kompletnego programu cieniującego 43

Oświadczenie o udostępnianiu pracy dyplomowej **45**

Oświadczenie autorskie **46**

1. Wstęp

1.1 Wprowadzenie

Celem pracy było stworzenie w pełni modyfikowalnego systemu, który ma za zadanie zaprezentować technologie, jak i procesy występujące przy wytwarzaniu aplikacji korzystających z interfejsu oprogramowania OpenGL. Projekt reprezentuje dwie funkcje, demonstracyjną która przedstawia stworzoną aplikacje oraz praktyczną która dostarcza narzędzi umożliwiającym edytowanie poszczególnych jej aspektów. W pracy został przedstawiony komplet informacji dotyczących istotnych zagadnień z zakresu grafiki komputerowej i renderingu. Mają one za zadanie zaznajomić czytelnika z terminami i pojęciami wykorzystywanymi przy wytwarzaniu takiego rodzaju oprogramowania.

1.2 Cel pracy i założenia

W ramach pracy zostać ma zrealizowany projekt aplikacji graficznej stworzonej przy pomocy interfejsu OpenGL. Program powinien charakteryzować się wykorzystaniem komponentów i bibliotek „Wolnego i otwartego oprogramowania". Aplikacja ma za zadanie dostarczyć gotowy produkt demonstracyjny wybranej technologi oraz spełnić funkcje edytora wynikającą z inicjalizacji poszczególnych obiektów poprzez pliki tekstowe. Ponadto muszą być wykorzystane wszystkie typy cieniowania wierzchołków oraz powinny być dostarczone narzędzia umożliwiające nakładanie tekstur i obsługę wybranych czcionek.

1.3 Motywacja

Ideą do stworzenia projektu była chęć przedstawienia w prosty, logiczny i poukładany sposób procesu tworzenia aplikacji wykorzystującej interfejs OpenGL. Pomysł narodził się jako odpowiedź na brak polskojęzycznych materiałów dotyczących nowych podejść i technik wykorzystania tej technologi w praktyce.

1.4 Zakres

Zakresem pracy było przedstawienie czytelnikowi podstawowych zagadnień teoretycznych wykorzystywany podczas tworzenia aplikacji za pomocą interfejsu OpenGL. W części przeglądowej zostały omówione takie zagadnienia, jak: bufory, manipulowanie teksturami czy programy cieniujące mają one za zadanie przygotować czytelnika pod część praktyczną, w której zostaną omówione najważniejsze elementy systemu.

Część przeglądowa

2. Biblioteka OpenGL

W ramach wstępu do rozdziału zostanie omówiona pokrótce historia wraz z najważniejszymi faktami interfejsu OpenGL. Następnie będzie przedstawiona terminologia potoku graficznego, po której czytelnik zostanie wprowadzony w pojęcia obiektów biblioteki i teorii tworzenia programów cieniujących.

2.1 Przedstawienie biblioteki OpenGL

W pierwszej połowie lat 90 *Silicon Graphics inc*. był liderem w dziedzinie renderowania grafiki komputerowej. Ich dzieło *IRIS GL* było prekursorem do stworzenia otwartego interfejsu OpenGL, który za pośrednictwem architektury klient-serwer dawał możliwość kontrolowania procesu renderowania. Wieloplatformowość oraz zdolność do pracy z wieloma językami programowania uczyniła OpenGL jednym z najbardziej rozpowszechnionych API, z jakim możemy się dziś spotkać. Implementacja interfejsu spoczywa na barkach dostawców procesorów graficznych (GPU), którzy są odpowiedzialni za opracowywanie sterowników, które tłumaczą wywołania OpenGL bezpośrednio na komendy jednostki obliczeniowej. Dzięki takiemu rozwiązaniu interfejs jest dostarczany razem ze sterownikami kart rozszerzeń. Organizacja *ARB* podejmuje decyzje odnośnie do tego, jaki kurs OpenGL obierze w przyszłości. Organizacja *ARB* zrzesza czołowych producentów elektroniki użytkowej oraz oprogramowania. Od 2008 roku organizacja ARB przedstawia nowe wersje standardu rozszerzonego go o nowe funkcjonalności, uwzględniając kompatybilność wsteczną, co pozwala zachować spójność i porządek specyfikacji.

OpenGL jest zbiorem ponad 500 komend, dzięki którym możemy kontrolować procesy renderowania prymitywów. Niestety biblioteka sama w sobie jest interfejsem niskopoziomowym co oznacza brak wielu narzędzi umożliwiających wczytywanie tekstur, ładowania modeli 3D, tworzeniem kontekstu czy też zarządzaniem oknem programu. Naprzeciw tym problemom przychodzą biblioteki pomocnicze, które rozszerzają możliwości interfejsu OpenGL.

Rozszerzenia

OpenGL dostarcza możliwości rozszerzenia jego funkcjonaliści o nowe narzędzia niedostępne w jego podstawowej wersji. Dostarczane przez producentów elektroniki moduły rozszerzające są zatwierdzane przez zrzeszenie ARB. Przykładem ukazującym użycie rozszerzenia jest listing: 2.1.

```
if(KHR_debug){
DebugMessageControl( ... );
}
```

Listing 2.1. Przykładrozszerzenia OpenGL.

2.1.1 Stan OpenGL

Stan OpenGL (ang.*OpenGL State*) informacja bądź wartość przechowywana w pamięci komputera. Każdy stan jest unikalne zidentyfikowany oraz opisany w specyfikacji interfejsu. Stany dzielimy na 3 kategorie zależne od pełnionych funkcji: renderujące, kolejkujące, inicjalizujące.

2.1.2 Kontekst OpenGL

Kontekst OpenGL (ang.*OpenGL Context*) jest to zbiór wszystkich włączonych stanów wykorzystywanych podczas renderowania. Każda aplikacja może posiadać wiele kontekstów, które są odseparowane i niezależne od siebie, jednakże istnieje możliwość przesyłania danych pomiędzy nimi.

2.1.3 Maszyna stanów

Maszyna stanów (ang. *State Machine*) termin często wykorzystywany w literaturze pochodzący od specyficznego włączania i wyłączania danych funkcji interfejsu poprzez wywoływanie obiektów OpenGL.

2.2 Potok Graficzny

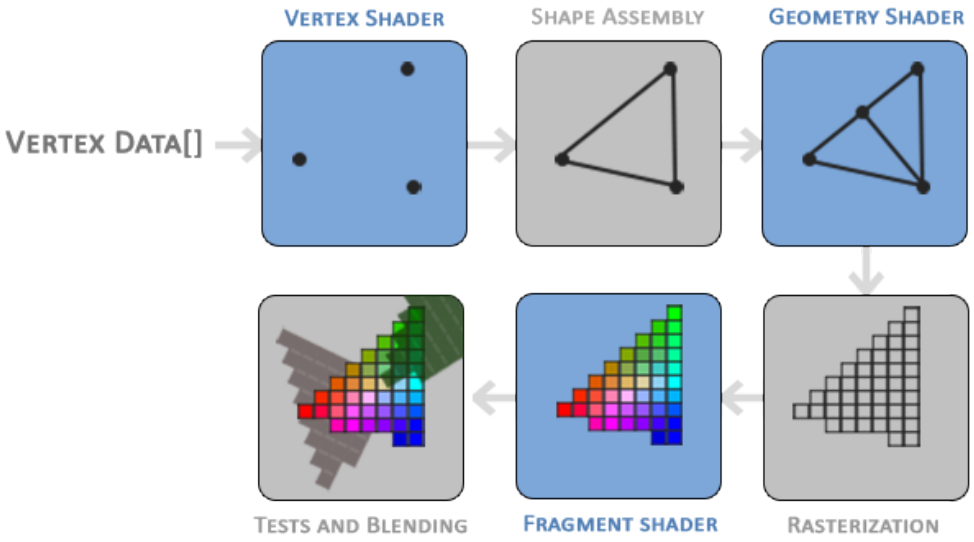
Potok graficzny (ang. *Rendering Pipeline*) jest to określenie faz, jakie OpenGL przechodzi podczas renderowania obiektu. Poniżej został przedstawiony wysokopoziomowy opis poszczególnych etapów, które biblioteka musi przejść. Przedstawia to obraz: 2.1.

Definiowanie wierzchołków

Pierwszym etapem potoku graficznego jest zadeklarowanie listy pozycji wierzchołków obiektu, z których będą w późniejszych etapach tworzone prymitywy. Dane wierzchołków są przetrzymywane w zdefiniowanych przez użytkownika buforach z których będą wysłane do dalszych etapów renderowania. Przykład poprawnie zdefiniowanej listy wierzchołków przedstawia listing: 2.2.

```
float verticies[] = {
// Pierwszy trójkąt
0.0f, 0.0f,
1.0f, 0.0f,
0.0f, 1.0f
};
```

Listing 2.2. Deklaracja współrzędnych prymitywu.



Rysunek 2.1. Przykład potoku graficznego OpenGL.
Źródło: `https://learnopengl.com/img/getting-started/pipeline.png`.

2.2.1 Przetwarzanie wierzchołków

Przetwarzanie wierzchołków (ang. Vertex Proccessing) rozpoczyna się od pobrania danych z utworzonego buforu, w tym etapie to od użytkownika zależy w jaki sposób dane mają być interpretowane ponieważ większa część przetwarzania jest programowalna. Lista poniżej przedstawia wszystkie etapy:

- **Cieniowanie wierzchołkowe** (ang. Vertex Shader) etap pozwalający przeprowadzać podstawowe operacje na każdym wierzchołku dostarczonym z buforu. Etap ten pozwala na transformowanie koordynat wierzchołków w przestrzeni 3D. Cieniowanie wierzchołkowe pozwala użytkownikowi na definiowanie swoich wyjść(ang. Outputs) do innych etapów np: wyjście informacji na temat położenia tekstur względem płaszczyzny,

- **Telestacja** (ang.Tessellation) opcjonalny etap dzielenia wielokątów na mniejsze prymitywy w celu dokładniejszego odwzorowania obiektu na scenie 3D. Interfejs OpenGL przeprowadza telestacje w dwóch etapach:

  - **Tessellation Control Shader** (TCS) - pierwszy etap określa na podstawie informacji przesłanych poprzez bufor ilość możliwych wielokątów do utworzenia z danego obiektu,
  - **Tessellation Evaluation Shader** (TES) - drugi etap definiuje na podstawie interpolacji wartości nowo wygenerowanych wierzchołków .

- **Cieniowanie geometryczne** (ang. Geometry Shader) opcjonalny etap definiowany przez

użytkownika pozwalający wygenerować nowe prymitywy lub przekształcić te, które zostały podane na wejściu. Odpowiada także za usuwanie oraz konwertowanie danych typów prymitywów np: linie staną się punktami.

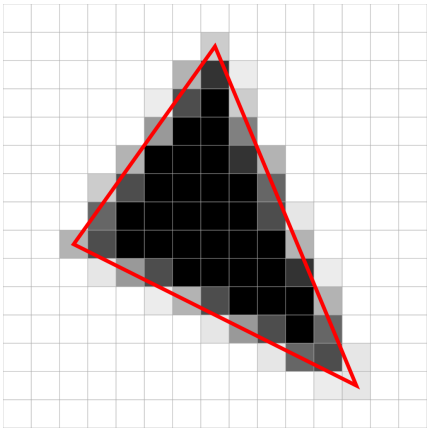
2.2.2 Łączenie wierzchołków

Łączenie wierzchołków (ang. *Primitive Assembly*) etap mający za zadanie połączyć każdy podany wierzchołek, dzięki czemu zostanie utworzony finalny prymityw.

2.2.3 Rasteryzacja

Rasteryzacja (ang. *Rasterization*) działanie mające na celu zmapowanie wierzchołków prymitywu z pikselami urządzenia wyświetlającego.

Przykład rasteryzacji reprezentuje obraz: 2.2.



Rysunek 2.2. Przykład rasteryzacji trójkąta.
Źródło:
`https://astrobites.org/wp-content/uploads/2013/08/Galaxy-Zoo-Flowchart.png`.

2.2.4 Cieniowanie pikseli

Cieniowanie pikseli – jest jednostką odpowiadającą za wyliczanie koloru pikseli. Direct3D używa terminu pixel shader, a OpenGL – fragment shader. Piksele na wejście tego etapu cieniowania są pobierane z rasteryzatora, który wypełnia wielokąty przesyłane z potoku graficznego. Cieniowanie pikseli jest najczęściej używane do oświetlenia sceny i innych powiązanych efektów, np. mip-mapingu lub kolorowania.

2.2.5 Przetwarzanie próbek

Przetwarzanie próbek(ang. *Pre-Sample processing*) końcowy etap przyjmujący dane z fragment shadera. Sprawdza wierzchołek pod kontem przenikania ()

2.3 Obiekty OpenGL

Obiektem OpenGL(ang.*OpenGL Objects*) nazywamy strukturę która po wywołaniu zmienia stan kontekstu interfejsu. W dość dużym uproszczeniu obiekt OpenGL można przedstawić jako funkcję która inicjalizuje pewną zmienną jako referencje. Przykład zadeklarowania obiektu znajduje się na listingu:4.1.

```
GLuint object;
glGenObject(1, &object)
```

Listing 2.3. Sworzenia obiektu.

2.3.1 Bufory

Bufory to obiekty tworzone z myślą o obsłudze pamięci przydzielonej przypisane do odpowiedniego kontekstu OpenGL. Są używane do przechowywania danych poszczególnych obiektów np: wierzchołków oraz pikseli danej tekstur. Pamięć buforu jest liniowa o rozmiarze ustalonym podczas jego tworzenia. Inicjalizacja pozwala ustawić szereg opcji zarządzania pamięcią przez bufor takich jak dynamiczna bądź statyczna alokacja pamięci. Listing: 4.2 prezentuje stworzenie buforu.

```
GLuint vbo;
glGenBuffers(1,&vbo);
glBindBuffer(GL_ARRAY_BUFFER,vbo);
glBufferData(GL_ARRAY_BUFFER,sizeof(verticies),&verticies,GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER,0);
```

Listing 2.4. Stworzenie, powiązanie z kontekstem obiektu buforu.

2.3.2 Textury

Tekstury jako obiekty interfejsu OpenGL pozwalają przechowywać informacje dotyczącą obrazów rastrowych(*Sprite*) pod warunkiem że są w tym samym formacie. Obiekty te są wykorzystywane do nadawania powierzchniom odpowiedniego wyglądu. Dzięki zastosowaniu tekstur możliwe jest iluzoryczne zwiększenie szczegółowości danej powierzchni bez wykorzystania dodatkowych wierzchołków. OpenGL potrafi obsłużyć wszystkie typy tekstur tzn. tekstury 1D, 2D oraz 3D. Przykład stworzenia obiektu prezentuje listing: 2.5.

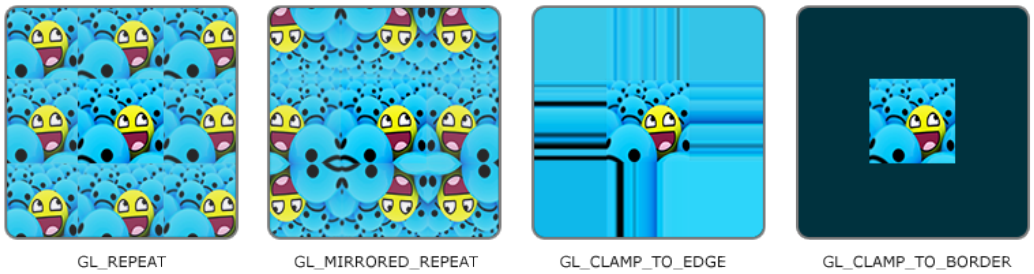
```
GLuint texture;
glGenTextures(1, &texture);
```

Listing 2.5. Tworzenie obiektu tekstury.

Interfejs OpenGL pozwala na zastosowanie szeregu opcji względem obiektu tekstury:

- **Zawijanie tekstur** (ang. *Texture Wraping*) opcja wykorzystywana w wypadku gdy współrzędne definiowanej tekstury wychodzą poza przedział między (0,0) do (1,1). Domyślnie interfejs powtarza nakładanie tekstury aż nie wypełni danej powierzchni, lecz w zależność od preferencji można użyć innych ustawień które OpenGL oferuje:
  - **GL_REPEAT**: Powtarzaj nakładanie tekstury,
  - **GL_MIRRORED_REPEAT**: Nakładaj lustrzane odbicie tekstury,
  - **GL_CLAMP_TO_EDGE**: Rozciągnij daną teksturę do krawędzi powierzchni,
  - **GL_CLAMP_TO_BORDER**: Dopasuj teksturę do obramowania.

  Przykładem użycia odpowiednich funkcji może być obraz: 2.3.



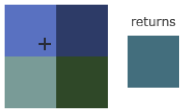
Rysunek 2.3. Przykład zawijania tekstur.
Źródło: `https://learnopengl.com/img/getting-started/texture_wrapping.png`.

- **Filtrowanie tekstur** (ang. *Texture Filtering*) jest odpowiedzialne za ustawienie koloru danego teksla (ang. texel) względem teksli sąsiadujących. Wyróżniamy dwie główne kategorie filtrowanie powiększania, filtr minifikacji (ang. magnification filtering and minification filtering). Ustawienia filtrowania dzielimy na:

  - **GL_NEARSET** wykorzystuje proces interpolacji która wykorzystywana jest do utworzenie nowego teksela na podstawie tekseli sąsiadujących by wypełnieniać luki gdy sytuacja wymaga rozciągnięcia mniejszej tekstury względem płaszczyzny na której ma być wyświetlana. Przykład działania prezentuje obraz: 2.4,

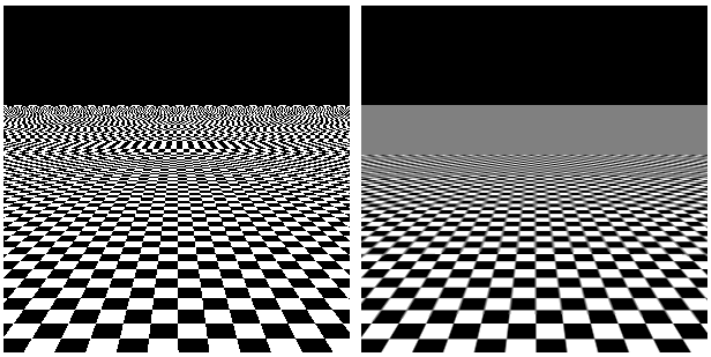
Rysunek 2.4. Komenda GL_NEAREST.
Źródło: `https://learnopengl.com/img/getting-started/filter_nearest.png`.

– **GL_LINEAR** wykorzystuje proces antyaliasingu który ogranicza liczbe błędów występujących przy zmniejszeniu rozdzielczości danej tekstury w wypadku gdy jest ona większa od płaszczyzny. 2.5,



Rysunek 2.5. Komenda GL_LINEAR.
Źródło: `https://learnopengl.com/img/getting-started/filter_linear.png`.

- **Mipmapping** jest formą wykorzystywaną do uniknięcia artefaktów spowodowanych skalowaniem tekstur nałożonych na płaszczyzny względem odległości od punktu widzenia. Działanie mipmapingu prezentuje obraz: 2.6.

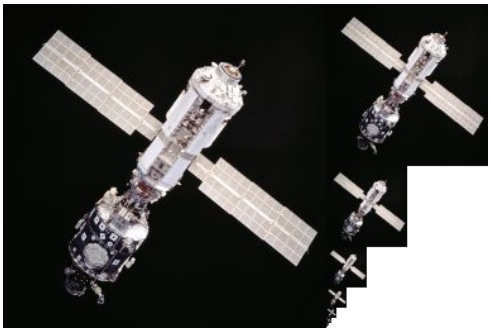


Rysunek 2.6. Przykład sceny z włączonym mipmapingiem i bez niego.
Źródło: `https://upload.wikimedia.org/wikipedia/commons/5/5c/MipMap_Example_STS101.jpg`.

Ustawienia mipmapingu przedstawia poniższa lista:

– **GL_NEAREST_MIPMAP_NEAREST** - Wyznacza najbliższą mipmapę którą można dopasować do rozmiaru piksela na podstawie interpolacji mipmapy sąsiadującej,
– **GL_LINEAR_MIPMAP_NEAREST** - Wyznacza najbliższą mipmapę na podstawie interpolacji liniowej,
– **GL_NEAREST_MIPMAP_LINEAR** - Na podstawie interpolacji dwóch sąsiadujących mipmap wyznacza która najbardziej pasuje do wielkości piksela,
– **GL_LINEAR_MIPMAP_LINEAR** - Ustawia mipmapę piksela na podstawie interpolacji liniowej najbliższych dwóch mipmap.

Przykładem mipmapingu użytego w praktyce jest obraz:2.7.
: Przykład służący za pokazanie stworzenia obiektu tekstury powiązania go z kontekstem



Rysunek 2.7. Przykładowa Mipmapa.
Źródło: https://upload.wikimedia.org/wikipedia/commons/5/5c/MipMap_Example_STS101.jpg.

oraz ustawieniem opcji prezentuje listing:2.6.

```
GLuint texture;
glGenTextures(1, &texture);
// Połączenie obiektu z kontekstem
glBindTexture(GL_TEXTURE_2D, texture);
// Opcje zawijania
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
// Opcje filtrowania
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
// Generowanie mipmap
glGenerateMipmap(GL_TEXTURE_2D);
// Odłączenie obiektu od kontekstu
glBindTexture(GL_TEXTURE_2D, 0);
```

Listing 2.6. Stworzenie, powiązanie z kontekstem obiektu tekstury.

2.4 Shadery OpenGL

Shader jest to krótki program komputerowy który jest kompilowany za pomocą procesora graficznego. Są używane do opisywania właściwość pikseli oraz wierzchołków. Shadery są częścią wielu etapów potoku graficznego. Pozwalają na komunikacje między sobą poprzez wejścia (ang.Inputs) i wyjścia (ang. Outputs).

Spis każdego z shaderów występujących w potoku graficznym reprezentuje lista poniżej:

- **Vertex Shader** - Obsługuje przetwarzanie poszczególnych wierzchołków. Odpowiada za transformacje położeń w przestrzeni 3D,
- **Tessellation Shader** - Etap opcjonalny służący do rozbijania prymitywów na mniejsze, łatwiejsze do odzwierciedlenia obiekty,
- **Geometry Shader** - Cieniowanie geometryczne jest definiowane przez użytkownika pozwala modyfikować siatkę wierzchołków,

- **Compute Shader** - Opcjonalny etap służący do przeprowadzania obliczeń, chociaż może być używany także by wspierać renderowanie.

Przykład kompletnego programu shadera zawirającego cieniowanie wierzchołkowe, geometryczne, jak również manipulowanie poszczególnymi wartościami pikseli prezentuje dodtatek B:5.1.

2.4.1 OpenGL Shading Language (GLSL)

OpenGL Shading Language (GLSL) jest jednym z języków programowania przeznaczonym do tworzenia programów cieniujących. Charakteryzuje się składnią podobną do języka C. Przykład shadera prezentuje listing:2.7.

```
#version 330 core
void main() {
gl_Position = vec4(0.5f, 0.5f, 0.0f, 1.0f);
}
```

Listing 2.7. Shader wierzchołków.

Typy

- **Skalary** (ang. *Scalars*) - Skalarami nazywamy podstawowe typy języka GLSL,
- **Wektory** (ang. *Vectors*) - Każdy z typów podstawowych ma swoje odpowiedniki wektorowe o rozmiarach między 2 a 4 komponenty,
- **Macierze** (ang. *Matrix*) - Macierze w GLSL występują jako typy tylko zmiennoprzecinkowe. Macierze charakteryzują się zmiennymi wierszami i kolumnami,
- **Uniformy** (ang. *Uniforms*) -Typy pozwalające użytkownikowi na przesyłanie odpowiednich wartości do programu,
- **Samplery** (ang. *Samplers*) - Każda tekstura która użytkownik chce wysłać do shadera musi być przekazana przez typ sampler.

Swizzling

Swizzling jest to elastyczna forma zarządzania elementami komponentów. Przykład zastosowania tego elementu jest podany na listingu:2.8.

```
vec2 vector_0;
vec3 vector_1 = vector_0.xyx;
vec4 vector_2 = vector_0.xy + vector_1.xz;
```

Listing 2.8. Przykład swizzlingu.

Część praktyczna

3. Opis wykorzystanych środowisk

Poniżej przedstawione zostały środowiska wykonawcze i programistyczne wykorzystane podczas tworzenia oraz kompilacji projektu.

3.1 Środowisko wykonawcze

Rozwiązanie było tworzone oraz testowane na maszynie wyposażonej w czterordzeniowy procesor Intel Core i5-6600K oferujący taktowanie na poziomie 3,5 GHz, kartę graficzną Nvidia Geforce GTX 970, która wspiera komendy OpenGL w wersji 4.4.

3.2 Środowisko programistyczne

Projekt został stworzony i opracowany na platformę linux-Debian. Do kompilowania zostało użyte narzędzie g++ dostarczane w ramach projektu GNU. Oprogramowanie było tworzone przy wykorzystaniu darmowych narzędzi programistycznych takich jak Visual Studio Code czy CMake. Ponadto zostały wykorzystane takie biblioteki pomocnicze jak glad, glfw, glm, freetype2 oraz stbi_image.

4. Opis zrealizowanego systemu

Rozdział wprowadza czytelnika w kluczowe komponenty systemu takie jak wykorzystane biblioteki, klasy wykorzystanych obiektów, system zarządzania zasobami oraz generator cząsteczek.

4.1 Dodatkowe biblioteki

W projekcie został wykorzystany szereg dodatkowych biblioteki rozszerzających możliwość OpenGL.Spis wykorzystanych bibliotek prezentuje lista:

- **GLFW** - Multiplatformowa biblioteka kliencka przeznaczona do tworzenia powierzchni, okien i kontekstów,
- **GLM** - Narzędzia matematyczne przeznaczona do kooperacji z językiem GLSL poprzez jego emulacje,
- **STB** - Zbiór narzędzi manipulujących danymi obrazami wczytanymi do pamięci,
- **FREETYPE** - Biblioteka interpretująca wiele formatów czcionek.

4.2 Obiekty

Obiektem aplikacji nazywamy strukturę, która przechowuje referencje do poszczególnych drobnych cząstek takich jak obraz, prymityw czy też shader, które są wykorzystywane podczas renderingu. Za inicjalizacje obiektów odpowiadają pliki tekstowe tworzone przy wykorzystaniu szablonu załączonego w dodatku A5.1.

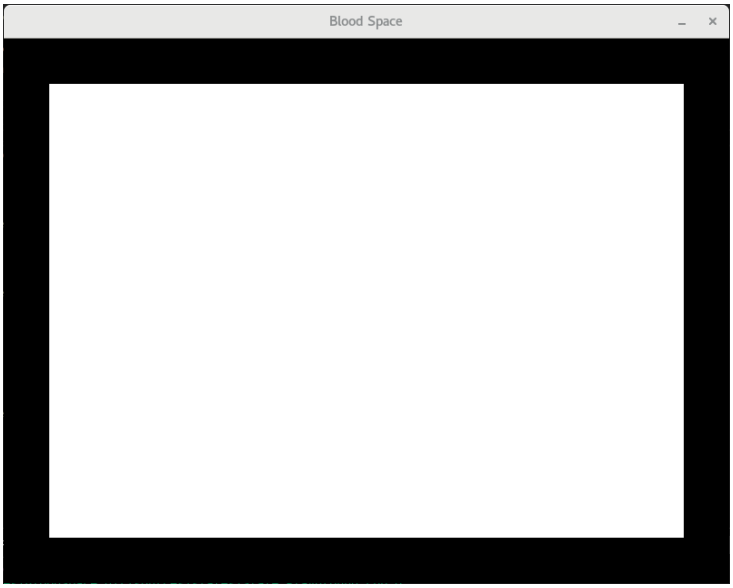
- **Klasa Primitive** odpowiada za inicjalizowanie i przechowywanie informacji o danej powierzchni. Inicjalizacja odbywa się poprzez funkcje szablonów.

```
class Primitive {
private:
// Struktura odpowiadająca za implementacje.
template Type&gt; struct Implementation {
// Manipuluj podanymi wskaźnikami.
template Arg&gt;
static Type parsePointers(const Arg &t_arg, int t_argSize);
// Stwórz wektor z podanych argumentów.
template <class... Args&gt; static Type createVector(Args... t_args);
}; // Implementation
// Wielkość primitywu.
glm::vec2 m_size;
// Pozycja prymitywu.
glm::vec2 m_position;
// Color prymitywu.
glm::vec4 m_color;

public:
```

```
    Primitive();
    // Ustaw wielkość.
    template <class... Args&gt; GLvoid setSize(Args... t_args);
    // Ustaw pozycje.
    template <class... Args&gt; GLvoid setPosition(Args... t_args);
    // Ustaw kolor.
    template <class... Args&gt; GLvoid setColor(Args... t_args);
    // Pobierz wielkość.
    glm::vec2 &getSize();
    // Pobierz pozycje.
    glm::vec2 &getPosition();
    // Pobierz kolor.
    glm::vec4 &getColor();
    }; // Primitive
```

Listing 4.1. Plik nagłówkowy klasy Primitive.

Przedstawienie wyrenderowanego prymitywu reprezentuje obraz:4.1.



Rysunek 4.1. Przykład wyrenderowanego prymitywu.
Źródło własne.

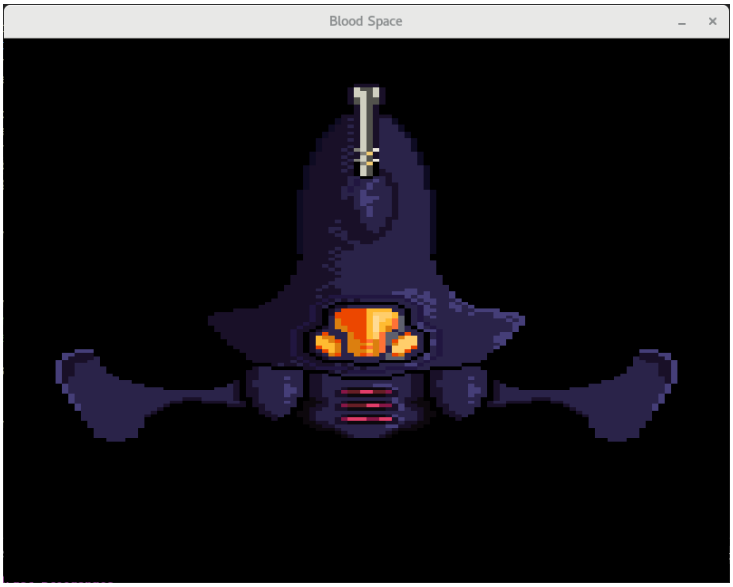
- **Klasa Image** odpowiada za pobieranie obrazu do pamięci oraz przechowywanie informacji takich jak wielkość, szerokość czy też liczba wykorzystywanych pikseli.

```
    class Image {
    private:
    // ID obrazu.
    GLuint m_id;
    // Nazwa obrazu.
    std::string m_name;
    // Piksele obrazu.
    GLubyte* m_pixels;
    // Wielkość obrazu.
    glm::ivec2 m_size;
```

```
    // Stwórz obiekt OpenGL
    GLuint createOpenGLObject();

    public:
    // Konstruktor przyjmuje ścieżkę do pliku.
    Image(const std::string &t_imagePath);
    // Pobierz nazwę.
    std::string getName();
    // Pobierz wielkość obrazu.
    glm::vec2 getSize();
    // Załącz obraz do kontekstu OpenGL
    GLvoid use();
    }; // Image
```

Listing 4.2. Plik nagłówkowy klasy Image

Wyrenderowany obraz przedstawia: 4.2.



Rysunek 4.2. Przykład wyrenderowanego obrazu.
Źródło własne.

- **Klasa Shader** reprezentuje grupe programów cienujących wykorzystywanych podczas renderingu.

```
    class Shader {
    private:
    // Wartość referencj obiektu OpenGL.
    unsigned m_shaderProgram;
    // Nazwa shadera
    std::string m_shaderName;
    // Połącz wybrany shader z programem.
    void link(unsigned t_shaderID, unsigned t_programID);
    // Inicjalizuj wybrany shader ze źródłem danych.
    void init(unsigned t_shaderID, const std::string &t_source);

    public:
```

```
// Default constructor.
Shader();
// Pobierz shader z pliku.
Shader(const std::string &t_shaderFilePath);
// Sprawdź typ podanego argumentu
template Type&gt;
void sendVariable(const std::string &t_uniformName, Type t_value);
// Pobierz nazwe shadera.
std::string getName();
// Uzyj shadera.
void useShader();
}; // Shader
```

Listing 4.3. Plik nagłówkowy klasy Shader.

- **Klasa Font** obsługuje funkcje manipulujące wczytaną czcionką.

```
class Font {
private:
// Rejestr wybranych glifów reprezentujących poszczególne litery alfabetu.
typedef std::map<GLchar, Glyph*> Registry;
Registry m_registry;
// Nazwa czcionki.
std::string m_name;
// Struktura przechowywująca dany styl i wielkość czcionki
FT_Face m_face;
// Tworzymy obiektOpenGL.
GLvoid createOpenGLObject();
public:
// Konstruktor pobierający ścieżke oraz wielkość danej czciąki.
Font(const std::string &t_fontPath, GLuint t_fontSize);
// Pobierz rejestr glifów.
Registry getGlyphs();
// Pobierz nazwę.
std::string getName();
}; // Font
```

Listing 4.4. Plik nagłówkowy klasy Font.

Przykład wykorzystania załadowanej czcionki reprezentuje obraz:4.3.

4.3 Zarządzanie zasobami

Za zarządzanie obiektami aplikacji odpowiada klasa ResourceManager. Jest ona istotna do odpowiedniego działania struktury, ponieważ utrzymuje porządek w przechowywanych zasobach i nie dopuszcza do dublowania takich obiektów jak obrazy, shadery czy też czcionki. Listing 4.5 prezentuje klase ResourceManager.

```
template Type&gt;
class ResourceManager {
private:
```

Rysunek 4.3. Przykład użytej czcionki by wyrenderować napis.
Źródło: własne.

```
// Rejestr przechowywujący obiekt.
static std::vector<Type&gt; m_registry;
// Przeszukuj rejestr poprzez porównywanie obiektów.
static bool search(Type t_object);
public:
// Przeszukuj rejestr w celu znalezienia obiektu wykorzystując jego nazwe.
static bool searchObject(const std::string& t_objectName);
// Rejestruj obiekt.
static void registerObject(const Type &t_object);
// Pobierz wszystkie obiekty.
static std::vector<Type&gt; getObjects();
// Pobierz obiekt
static Type getObject(const std::string& t_objectName);
}; // ResourceManager.hpp
```

Listing 4.5. Plik nagłówkowy klasy ResourceManage.r

4.4 Renderowanie obiektów

- **PrimitiveRenderer** klasa przeznaczona do renderowania prymitywów. Funkcje renderująca przedstawia listing4.6.

```
void PrimitiveRenderer::renderPrimitive() {
// Połącz bufor z kontekstem.
glBindVertexArray(m_vao);
// Renderuj wierzchołki.
glDrawArrays(GL_TRIANGLES, 0, 6);
// Odłącz bufor od kontekstu
glBindVertexArray(0);
}
```

Listing 4.6. Funkcja PrimitiveRenderer.

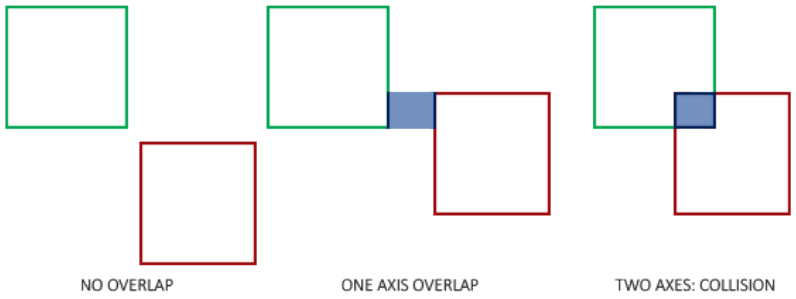
- **TextRenderer** klasa przeznaczona do obsługi, manipulowania oraz renderowania wybranego tekstu. Funkcje renderującą tekst przedstawia listing4.7.

```
// Pobierz obiket Text.
GLvoid TextRenderer::renderText(Text t_text) {
// Użyj shadera.
m_textShader-&gt;use();
// Połącz obiekt z kontekstem.
glBindVertexArray(m_vao);
std::string text = t_text.getText();
std::string::const_iterator ite;
float lineLenght = 0.0f;
for (ite = text.begin(); ite != text.end(); ite++) {
// Pobierz glif
Glyph glyph = *t_text.getFont().getGlyphs()[*ite];
Glyph glyph_H = *t_text.getFont().getGlyphs()['H'];
// Wyznacz pozycje renderowanego glifu.
float xpos = t_text.getPosition().x + lineLenght + glyph.
    getGlyphBearing().x;
float ypos = t_text.getPosition().y + (glyph_H.getGlyphSize().y -
    glyph.getGlyphBearing().y);
// Pobierz wielkość glifu
float w = glyph.getGlyphSize().x;
float h = glyph.getGlyphSize().y;
// Zaktualizuj wierzchołki
float vertices[6][4] = {
{ xpos, ypos + h, 0.0, 1.0 },
{ xpos + w, ypos, 1.0, 0.0 },
{ xpos, ypos, 0.0, 0.0 },

{ xpos, ypos + h, 0.0, 1.0 },
{ xpos + w, ypos + h, 1.0, 1.0 },
{ xpos + w, ypos, 1.0, 0.0 }
};
// Połącz wybrany glif z kontekstem.
glBindTexture(GL_TEXTURE_2D, glyph.getGlyphID());
// Połącz bufor wierzchołków z kontekstem.
glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
// Aktualizuj dane w bugorze.
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
// Odłącz buffor od kontekstu.
glBindBuffer(GL_ARRAY_BUFFER, 0);
// Renderuj wierzchołki.
glDrawArrays(GL_TRIANGLES, 0, 6);
// Odłącz glif od kontekstu.
glBindTexture(GL_TEXTURE_2D, 0);
// Zliczaj szerokość każdego wyrenderowanego glifu.
lineLenght += (glyph.getGlyphAdvance() >> 6);
}
```

Listing 4.7. Funkcja TextRenderer.

4.5 Kolizje obiektów

Kolizje obiektów przeprowadzane są poprzez algorytm AABB który zakłada że obiekty kolizyjne są prostokątami oraz sprawdza czy dana oś obiektu styka się z osią obiektu kolizyjnego. Przedstawia to obraz4.4.



Rysunek 4.4. Kolizja AABB.
Źródło: `https://learnopengl.com/img/in-practice/breakout/collisions_overlap.png`.

Zaimplementowany powyższy algorytm prezentuje listing 4.8.

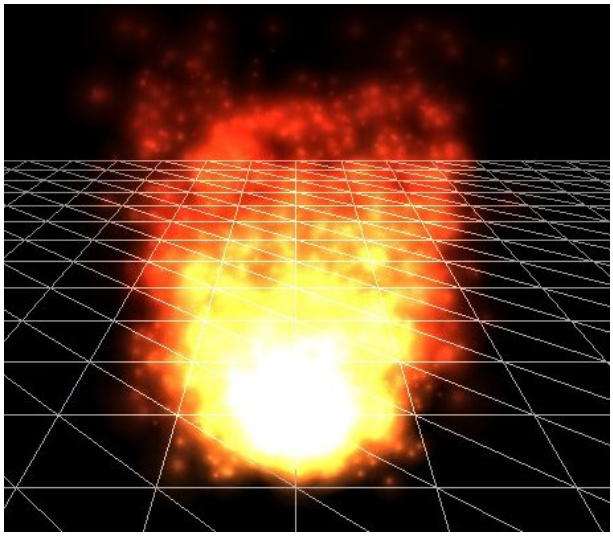
```
static GLboolean checkPrimitiveCollision(Primitive t_one, Primitive t_two) {

// Sprawdź kolizje na osi X.
bool colX= t_one.getPosition().x + t_one.getSize().x >= t_two.getPosition().x
    && t_two.getPosition().x + t_two.getSize().x >= t_one.getPosition().x;

// Sprawdź kolizje na osi Y.
bool colY= t_one.getPosition().y + t_one.getSize().y >= t_two.getPosition().y
    && t_two.getPosition().y + t_two.getSize().y >= t_one.getPosition().y;

// Jeżeli kolizja na obu osiach zwróć prawde.
return colX && colY;
}
```

Listing 4.8. Funkcja CollisionDetection.

4.6 Generator cząsteczek

Generator lub emiter cząsteczek jest to obiekt, który ze swojego miejsca stale tworzy nowe cząsteczki, które rozpadają się w czasie. Za teoretyczny przykład emitera może posłużyć ogień gdzie cząsteczki, które oddalają się od generatora zmieniają swoją barwę na mniej jasną. Obrazem reprezentującym emiter jest4.5.

Przykład pliku nagłówkowego generatora cząsteczek przedstawia listing4.9.



Rysunek 4.5. Przykład generatora cząsteczek.
Źródło: https://learnopengl.com/img/in-practice/breakout/particles_example.jpg.

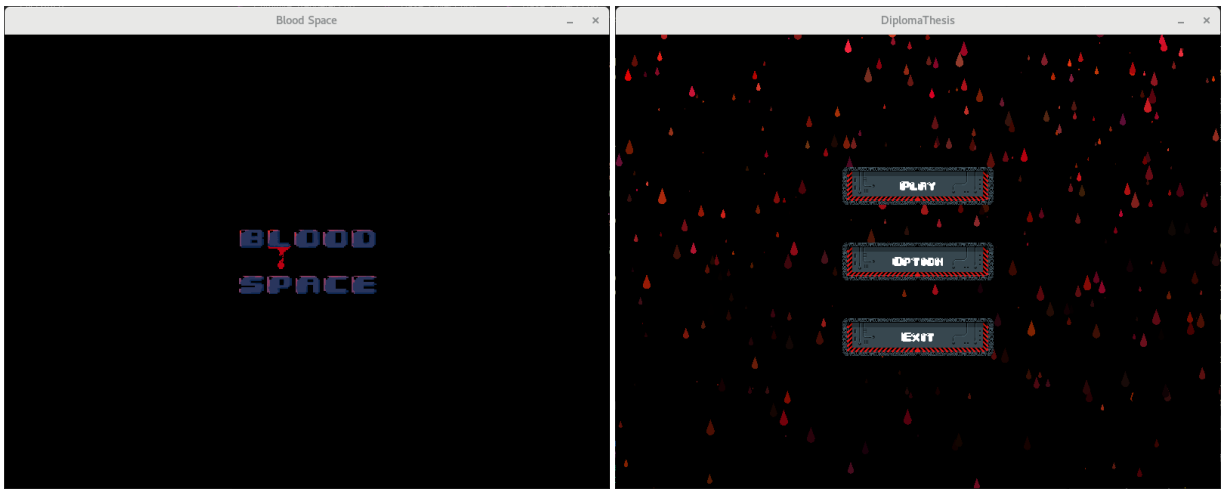
```
class ParticleGenerator {
private:
// Struktura cząsteczki
struct Particle : public Primitive {
private:
// Prędkość cząsteczki
GLfloat m_velocity;
// Życie cząsteczki
GLfloat m_life;
public:
Particle();
// Ustaw prędkość.
GLvoid setVelocity(GLfloat t_velocity);
// Ustaw życie.
GLvoid setLife(GLfloat t_life);
// Pobierz prędkość.
GLfloat getVelocity();
// Pobierz życie.
GLfloat &getLife();
}; // Particle

// Sprite.
Image *m_image;
// Shader.
Shader *m_shader;

// Wielkość cząsteczki.
glm::vec2 m_particleSize;
// Wartości przedziałów wykorzystywane przy funkcjach rand.
glm::vec2 *m_scopePosition;
glm::vec2 m_scopeSize;
glm::vec3 m_scopeColor;

// szybkość rozpadania się poszczególnych wartości cząsteczki.
glm::vec3 m_drains;
// Czy włączyć blending.
```

```
GLboolean m_blending;
// Wartość skalująca.
GLboolean m_scaling;
// Pojemnik cząsteczek.
std::vector<Particle&gt; m_particles;
// Ustaw losowe wartości.
GLvoid randomiseParticle();
// Aktualizuj cząsteczki.
GLvoid updateParticle();
public:
// Pobierz shader cząsteczki, ilość oraz ustaw blending.
ParticleGenerator(const std::string &t_shaderPath, GLuint t_particlesAmount,
    GLboolean t_blending);
// Ustaw nowy obraz.
GLvoid setImage(const std::string& t_imagePath);
// Przekaż obraz jako referencje.
GLvoid setImage(Image &t_image);
// Ustaw przedział koloru.
GLvoid setScopeColor(GLfloat t_x, GLfloat t_y, GLfloat t_z);
// Ustaw przedział wielkości.
GLvoid setParticleSize(GLfloat t_x, GLfloat t_y, GLboolean t_scale);
// Ustaw przedział pozycji emitera.
GLvoid setSpawnRange(GLfloat t_x, GLfloat t_y, GLfloat t_z, GLfloat t_w);
// Ustaw wartości rozpadania.
GLvoid setDrains(GLfloat t_life, GLfloat t_color, GLfloat t_position);
// Renderuj
GLvoid render();
}; //ParticleGenerator
```

Listing 4.9. Plik nagłówkowy klasy ParticleGenerator.

4.7 Przedstawienie finalnego projektu

Poniżej zostaną zaprezentowane obrazy reprezentujące finalny projekt.



Rysunek 4.6. Logo i Menu.
Źródło: własne.

Rysunek 4.7. Finalna rozgrywka.
Źródło: własne.

5. Podsumowanie

Postawione wymagania założeń zostały w pełni zrealizowane. Zostało dostarczone oprogramowanie, które może posłużyć jako przykład wykorzystania interfejsu OpenGL w budowaniu złożonych systemów. Dzięki możliwość inicjalizacji obiektów czy danych stanów poprzez pliki tekstowe teoretyczny użytkownik jest w stanie konfigurować aplikacje według swoich własnych upodobań. Wykorzystane wzorce projektowe dbają o odpowiednie zarządzanie pamięcią w celu uniknięcia jej wycieków, szczególnie podczas renderowania scen z wieloma obiektami.

Problematyczne okazało się odpowiednie zarządzanie projektem, dość często była przykuwana uwaga w kierunku tworzenia narzędzi, których finalny projekt wcale nie potrzebował. Przykładem może być funkcja rozbijająca dany obraz na inne mniejsze, poprzez sprawdzania kolorów teksli. Problemem także okazało się tworzeniem odpowiednich grafik na potrzeby aplikacji wynikającym z braku danych umiejętności plastycznych.

5.1 Możliwość kontynuacji

Pomimo spełnionych założeń możliwa jest kontynuacja projektu, która zakłada wiele scenariuszy. Możliwe jest przeniesienie rozwiązania na platformy mobilne. Także technologie webowe oferują narzędzia do renderowania takie jak interfejs WebGL. W razie kontynuacji projekt powinien, zostać przepisany uwzględniając nowe wzorce czy też techniki programowania. Powinny być dodane nowe tekstury, programy cieniujące czy też wizualne narzędzia edycji.

Spis rysunków

2.1 Przykład potoku graficznego OpenGL.
Źródło: https://learnopengl.com/img/getting-started/pipeline.png . . . 11
2.2 Przykład rasteryzacji trójkąta.
Źródło: https://astrobites.org/wp-content/uploads/2013/08/Galaxy-Zoo-Flowchart.png . . . 12
2.3 Przykład zawijania tekstur.
Źródło: https://learnopengl.com/img/getting-started/texture_wrapping.png . . . 14
2.4 Komenda GL_NEAREST.
Źródło: https://learnopengl.com/img/getting-started/filter_nearest.png . . . 14
2.5 Komenda GL_LINEAR.
Źródło: https://learnopengl.com/img/getting-started/filter_linear.png . . . 15
2.6 Przykład sceny z włączonym mipmapingiem i bez niego.
Źródło: https://upload.wikimedia.org/wikipedia/commons/5/5c/MipMap_Example_STS101.jpg . . . 15
2.7 Przykładowa Mipmapa.
Źródło: https://upload.wikimedia.org/wikipedia/commons/5/5c/MipMap_Example_STS101.jpg . . . 16

4.1 Przykład wyrenderowanego Prymitywu.
Źródło własne . . . 24
4.2 Przykład wyrenderowanego obrazu.
Źródło własne . . . 25
4.3 rzykład użytej czcionki by wyrenderować napisu.
Źródło: własne . . . 27
4.4 Kolizja AABB.
Źródło: https://learnopengl.com/img/in-practice/breakout/collisions_overlap.png . . . 29
4.5 Przykład generatora cząsteczek.
Źródło: https://learnopengl.com/img/in-practice/breakout/particles_example.jpg . . . 30

4.6 Logo i menu.
Źródło własne 31
4.7 Finalna rozgrywka.
Źródło: własne 32

Listings

2.1 Przykładrozszerzenia OpenGL. 9
2.2 Deklaracja współrzędnych prymitywu. 10
2.3 Sworzenia obiektu. 13
2.4 Stworzenie, powiązanie z kontekstem obiektu buforu. 13
2.5 Tworzenie obiektu tekstury. 13
2.6 Stworzenie, powiązanie z kontekstem obiektu tekstury. 16
2.7 Shader wierzchołków. 17
2.8 Przykład swizzlingu. 17
4.1 Plik nagłówkowy klasy Primitive. 23
4.2 Plik nagłówkowy klasy Image 24
4.3 Plik nagłówkowy klasy Shader. 25
4.4 Plik nagłówkowy klasy Font. 26
4.5 Plik nagłówkowy klasy ResourceManage.r 26
4.6 Funkcja PrimitiveRenderer. 27
4.7 Funkcja TextRenderer. 28
4.8 Funkcja CollisionDetection. 29
4.9 Plik nagłówkowy klasy ParticleGenerator. 30
5.1 Przykład pliku inicjalizującego dany obiekt 43
5.2 Vertex Shader 43
5.3 Geometry Shader 43
5.4 Fragment Shader 44

Bibliografia

[1] Joey de Vries, *Your #1 resource for OpenGL* `https://learnopengl.com/`, (odczytano 10 marca 2019)

[2] Khronos group, *OpenGL wiki*, `https://www.khronos.org/opengl/wiki/`, (odczytano 23 stycznia 2019)

[3] Etay Meiri, *OGLdev Modern OpenGL Tutorials main page*, `http://ogldev.atspace.co.uk/`, (odczytano 02 stycznia 2019)

[4] David Wolff, *OpenGL 4 Shading Language Cookbook: Build high-quality, real-time 3D graphics with OpenGL 4.6, GLSL 4.6 and C++17, 3rd Edition* (2018)

[5] Frahaan Hussain , *Learn OpenGL: Beginner's guide to 3D rendering and game development with OpenGL and C++*, Transformations, Projections and Camera (2018) 71-111

Zawartość płyty CD:

- Praca_dyplomowa.pdf.
- Kod_aplikacji.zip

Dodatki

Ogólny format plików inicjalizujących

W celach zachowania spójności pliki inicjalizujące posiadają swoją własna konwencje przedstawiona na listingu 5.1.

Listing 5.1. Przykład pliku inicjalizującego dany obiekt

```
[Obiekt]
RectangleSize = 50.0, 50.0
RectanglePosition = 0.0, 0.0
RectangleColor = 0.0
text = "0,0"
textScale = 1.2
textColor = 1.0
...
```

Przedstawienie kompletnego programu cieniującego

Vertex shader

Listing 5.2. Vertex Shader

```
#version 330 core

void main() {
gl_Position = vec4(0.5f, 0.5f, 0.0f, 1.0f);
}
```

Geometry shader

Listing 5.3. Geometry Shader

```
#version 330 core

layout (triangles) in;
layout (triangle_strip, max_vertices = 64) out;

uniform mat4 t_projection;
uniform mat4 t_model;

void main() {
for (int i = 0; i <= 45; i++) {

float angle = 3.14 * 2.0 / 45 * i;

vec4 offset = vec4(cos(angle), -sin(angle), 0.0, 0.0) / 2.0f;
gl_Position = t_projection * t_model * (gl_in[0].gl_Position + offset);
EmitVertex();
```

```
offset = vec4(cos(angle), sin(angle), 0.0, 0.0) / 2.0f;
gl_Position = t_projection * t_model * (gl_in[0].gl_Position + offset);
EmitVertex();

}
EndPrimitive();
}
```

Fragment shader

Listing 5.4. Fragment Shader

```
#version 330 core

out vec4 m_color;

uniform vec4 t_color = vec4(1.0f);
uniform sampler2D t_sampler;

void main() {

m_color = t_color;
}
```

Wrocław, dnia 2019-03-14

Wydział Informatyki, Administracji i Fizjoterapii

Kierunek studiów: **informatyka (INF)**

Igor Sadza
. . . . . .
(imię i nazwisko studenta)

5887
. . . . . .
(nr albumu)

OŚWIADCZENIE O UDOSTĘPNIANIU PRACY DYPLOMOWEJ

Tytuł pracy dyplomowej: Tworzenie aplikacji graficznych przy pomocy OpenGL

Wyrażam zgodę (nie wyrażam zgody)[1] na udostępnianie mojej pracy dyplomowej.

. . . . . .
(podpis studenta)

[1] Niepotrzebne skreślić.

Wrocław, dnia 2019-03-14

Wydział Informatyki, Administracji i Fizjoterapii

Kierunek studiów: **informatyka (INF)**

Igor Sadza
. . . . . .
(imię i nazwisko studenta)

5887
. . . . . .
(nr albumu)

OŚWIADCZENIE AUTORSKIE

Oświadczam, że niniejszą pracę dyplomową pod tytułem:

Tworzenie aplikacji graficznych przy pomocy OpenGL

napisałem/am samodzielnie. Nie korzystałem/am z pomocy osób trzecich, jak również nie dokonałem/am zapożyczeń z innych prac.

Wszystkie fragmenty pracy takie jak cytaty, ryciny, tabele, programy itp., które nie są mojego autorstwa, zostały odpowiednio zaznaczone i zamieszczono w pracy źródła ich pochodzenia. Treść wydrukowanej pracy dyplomowej jest identyczna z wersją pracy zapisaną na przekazywanym nośniku elektronicznym.

Jednocześnie przyjmuję do wiadomości, że jeżeli w przypadku postępowania wyjaśniającego zebrany materiał potwierdzi popełnienie przeze mnie plagiatu, skutkować to będzie niedopuszczeniem do dalszych czynności w sprawie nadania mi tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz złożenie zawiadomienia o popełnieniu przestępstwa.

. . . . . .
(podpis studenta)