

# Laboratorium Programowania Komputerów

---

Temat:

Gra typu strzelanka wieloosobowa wykorzystująca bibliotekę SDL  
i OpenGL

Autor: Jakub Klimek, Michał Jankowski  
Informatyka, semestr 4, gr. 1

Prowadzący: Dr hab. inż. Jolanta Kawulok

Repozytorium:

<https://github.com/jakubklimek97/simple-shooter/>

<b>Temat</b>	<b>7</b>
<b>Wykorzystane zagadnienia</b>	<b>7</b>
Jakub Klimek	7
Michał Jankowski	7
<b>Wykonane klasy</b>	<b>8</b>
<b>Analiza i Projektowanie</b>	<b>8</b>
Analiza problemu	8
Algorytm	9
<b>Specyfikacja zewnętrzna</b>	<b>11</b>
Uruchamianie programu	11
Wymagania sprzętowe	11
Obsługa programu	11
Serwer	13
Klient	14
Gra	14
<b>Specyfikacja wewnętrzna</b>	<b>15</b>
Użyte biblioteki	16
Klasy	16
Bullet	16
Konstruktory	17
Funkcje	17
Camera	17
Konstruktory	18
Typy wyliczeniowe	18
Funkcje	18
Entity	18
Konstruktory	19
Funkcje	19

Image2D	20
Konstruktory	21
Funkcje	21
LightObject	21
Konstruktory	21
Funkcje	22
Loader	22
Typy wyliczeniowe	23
Funkcje	23
Mesh	24
Konstruktory	25
Struktury	25
Funkcje	25
Model	25
Konstruktory	26
Funkcje	26
Networking	27
Typy wyliczeniowe	27
Funkcje	28
Scene	29
Konstruktory	29
Funkcje	29
SceneMultiplayerGame	31
Konstruktory	32
Funkcje	32
Shader	32
Konstruktory	33
Funkcje	33
Terrain	34
Funkcje	35

Collision	35
Funkcje	35
DirectionalLight	36
Funkcje	36
Konstruktory	36
SpotLight	36
Funkcje	37
Konstruktory	37
Fog	37
Konstruktory	38
Funkcje	38
Geometry	38
HeightMap	38
Funkcje	39
HUD	41
Konstruktory	42
Funkcje	42
Skybox	43
Konstruktory	44
Funkcje	44
Sound	45
Konstruktory	45
Funkcje	45
SoundManager	46
Konstruktory	46
Funkcje	46
SpriteRenderer	47
Konstruktory	48
Funkcje	48
SpriteTexture	48

Konstruktory	48
Funkcje	48
TextManager	49
Konstruktory	49
Funkcje	49
TextureClass	50
Konstruktory	51
Funkcje	51
HeightMapBuffer	52
Konstruktory	53
Funkcje	53
ResourceManager	53
Konstruktory	54
Funkcje	54
CShader	55
Konstruktory	55
Funkcje	55
<b>Wydruk najważniejszych funkcji</b>	<b>56</b>
Główna pętla programu	56
Scena rozgrywki	56
Inicjalizacja sceny	56
Wyświetlanie sceny	57
Obsługa połączenia sieciowego (serwer)	58
Pobieranie wysokości terenu	60
Ładowanie modelu	60
Tworzenie i komplikacja shadera	61
Renderowanie HUDu gry	61
Wyświetlanie Skyboxa	62
Pobieranie utworów muzycznych (SFX i Muzyka)	62
Ładowanie tekstur do HeightMapy	63

<b>Testowanie</b>	<b>64</b>
Wycieki pamięci	64
<b>Wnioski</b>	<b>65</b>
Jakub Klimek	65
Michał Jankowski	65

## Temat

Napisać grę dla wielu graczy w 3D wykorzystującą OpenGL oraz bibliotekę SDL. Gra ma wykorzystywać protokół TCP/IP do komunikacji oraz obsługiwać efekty dźwiękowe.

## Wykorzystane zagadnienia

### Jakub Klimek

Zagadnienie	Klasa::funkcja
Wyjątki	Shader::Shader( <code>const char*</code> vertexPath, <code>const char*</code> fragmentPath)
RTTI	Entity* Scene::addObject(Entity* object)
Algorytmy	<code>void</code> Scene::removeObject(Entity * object)
Wątki	<code>void</code> SceneMultiplayerGame::InitScene() <code>void</code> SceneMultiplayerGame::serverConnectionHandlerThread() <code>void</code> SceneMultiplayerGame::connectionHandlerThread()
Kontenery	<code>bool</code> Loader::loadShaders() <code>bool</code> Loader::loadTextures2D() <code>bool</code> Loader::loadModels()
Iteratory	<code>void</code> Scene::DrawObjects()

### Michał Jankowski

Zagadnienie	Klasa::funkcja
Wyjątki	Shader ResourceManager::loadShaderFromFile( <code>const char</code> * vShaderFile, <code>const char</code> * fShaderFile)
RTTI	<code>bool</code> HeightMap::LoadMapFromImage( <code>const char</code> *path, <code>const std::string</code> &directory)
Algorytmy	<code>void</code> CSkybox::LoadSkyBoxVector( <code>vector&lt;string&gt;</code> vec) <code>void</code> HUD::InitHUD( <code>vector&lt;reference_wrapper&lt;const Shader&gt;&gt;</code> _LifeShader)
Wątki	<code>void</code> SceneMultiplayerGame::handleEvents(SDL_Event & e)
Kontenery	<code>bool</code> HeightMap::LoadMapFromImage( <code>const char</code> *path, <code>const std::string</code> &directory) <code>void</code> HUD::SetVertices() CSkybox::CSkybox(Shader&Program, string Front, string Back, string Left, string Right, string Top, string Bottom) <code>void</code> HUD::SetIndices()
Iteratory	<code>void</code> HUD::RenderHUD( <code>vector&lt;reference_wrapper&lt;const Shader&gt;&gt;</code> _LifeShader) CSkybox::CSkybox()

## Wykonane klasy

Jakub Klimek	Michał Jankowski
Bullet Camera Entity Image2D LightObject Loader Mesh Model Networking Scene SceneMultiplayerGame(struktura, sieć, ciało funkcji) Shader Terrain	Collision CDirectionalLight CSpotLight Fog HeightMap HUD ResourceManager CShader ShaderProgram CSkybox Sound SoundManager SpriteRenderer SpriteTexture TextManager CTexture HeightMapBuffer SceneMultiPlayerGame(Wykrywanie kolizji i wystrzały)

## Analiza i Projektowanie

### Analiza problemu

Tworzona aplikacja ma działać w trybie graficznym, a rozgrywka ma przebiegać w przestrzeni 3D. Pierwszym zagadnieniem do rozważenia był wybór odpowiedniej biblioteki. Zdecydowaliśmy się na połączenie bibliotek SDL i Glew ze względu na fakt, że są one bardzo dobrze udokumentowane.

Aby ułatwić import obiektów, zdecydowaliśmy się użyć biblioteki Assimp. Zamienia ona zawartość plików modeli 3D na zdefiniowaną strukturę, z której w łatwy sposób możemy skopiować dane do struktur OpenGL.

Kolejnym problemem był przesył danych. Zdecydowaliśmy się na protokół TCP, ze względu na prostszą implementację. Nie przewidywaliśmy dużej ilości przesyłanych danych, dlatego nie martwiliśmy się o zwiększyony narzut.

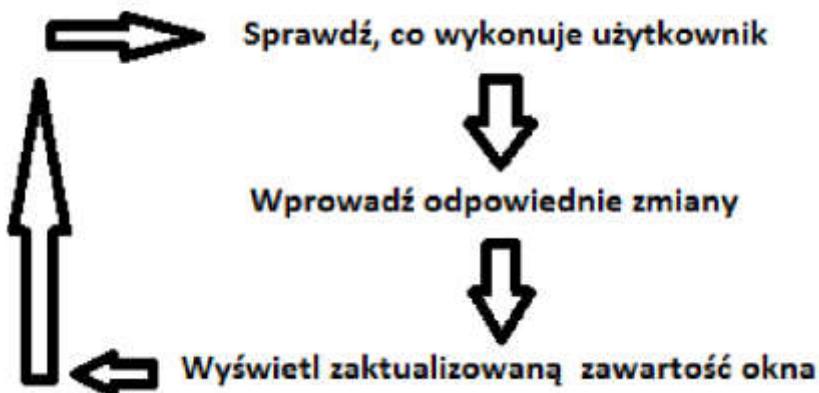
Tworzona gra będzie wykorzystywała kilka algorytmów niezbędnych niezbędnych z punktu logiki gry. Algorytmy mają być podzielone na:

- Kolizyjny gracza z terenem – umożliwia utrzymanie gracza na powierzchni mapy uniemożliwiając mu przenikanie przez nią;
- Kolizyjny gracza z pociskiem broni – seria kilku warunków w zależności od aktualnej pozycji gracza pozwala nam na sprawdzenie czy dany gracz został trafiony przez wystrzał z pistoletu.

Aby zapewnić płynność działania, aplikacja będzie musiała działać na kilku wątkach - moduł sieciowy musi działać stale, bez względu na pętlę samej gry.

## Algorytm

Główny algorytm działania będzie oparty o nieskończoną pętlę:

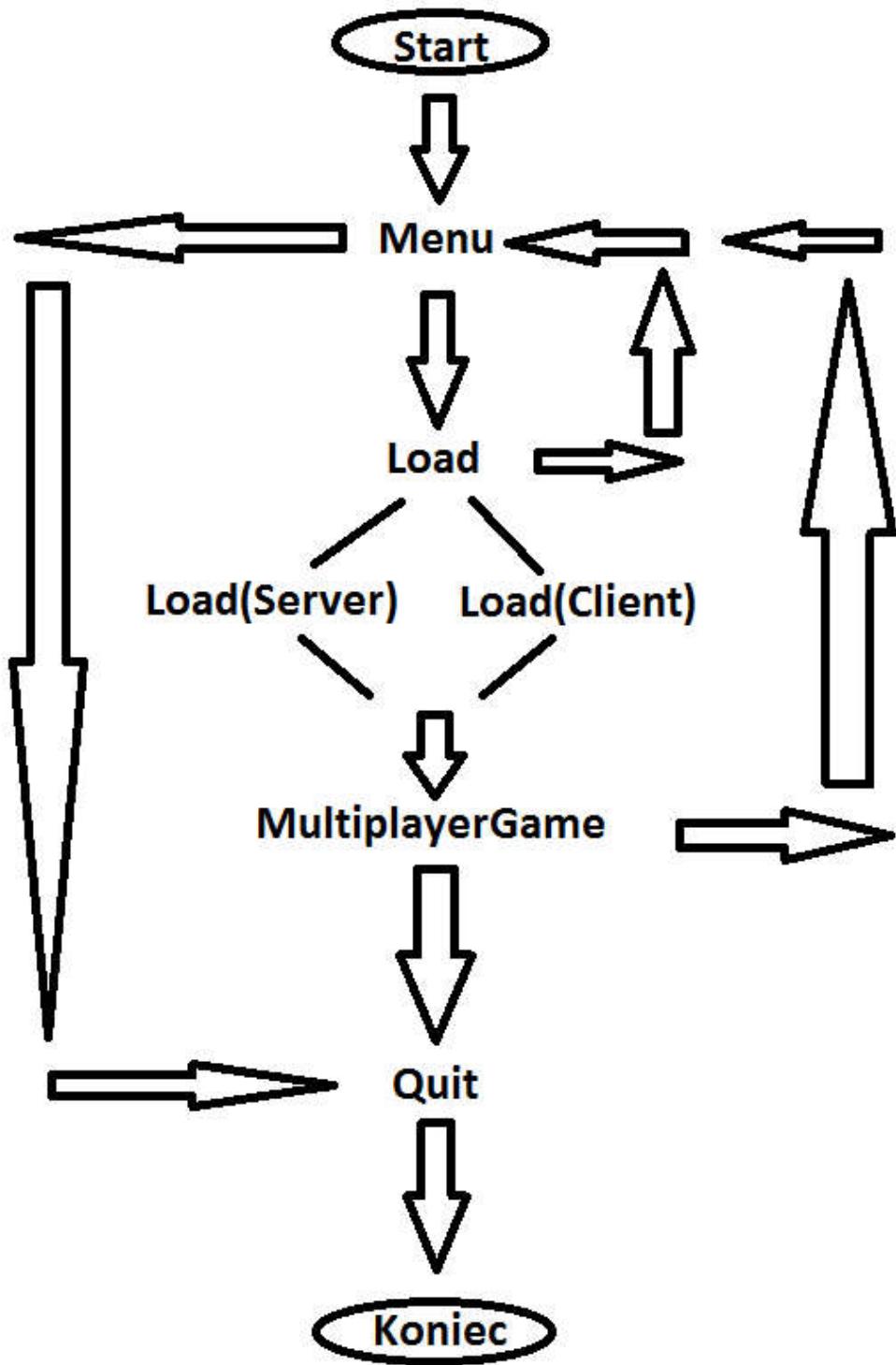


Wartownikiem będzie system przełączania sceny, tzn. pętla będzie się wykonywać tak długo, aż użytkownik nie znajdzie się w scenie wyjściowej (SCENE\_QUIT).

Zmiany scen będą wywoływanego na wprowadzania zmian. System został zaprojektowany w taki sposób, aby zmiany scen mogły się odbywać bez względu na obecny stan systemu.

Jako scenę definiujemy wydzieloną zawartość programu, realizującą odpowiednią funkcję, która jest samowystarczalna. Dynamika działania aplikacji zostanie zachowana dzięki przełączaniu się między scenami.

Planowane sceny wraz z przejściami między nimi:



Innymi słowy, program zaczyna się w menu głównym i każda próba wyjścia z gry następuje po wybraniu odpowiedniej pozycji w menu.

Na koniec pozostał jeden istotny problem - przechowywanie shaderów, tekstur, modeli. Ze względu na prostotę programu, zdecydowaliśmy się na ładowanie powyższych obiektów podczas startu aplikacji i zwalnianie zasobów przy wyjściu. Aby zachować ciągłość pamięci, obiekty te będziemy przechowywać w wektorach.

## Specyfikacja zewnętrzna

### Uruchamianie programu

Program nie wymaga parametrów podawanych przy uruchamianiu.

### Wymagania sprzętowe

Aby program mógł się uruchomić, karta graficzna komputera musi obsługiwać rysowanie przy użyciu Shaderów w wersji przynajmniej 3.3.

### Obsługa programu

Program odpala się w oknie, po którym poruszamy się używając klawiszy strzałek.

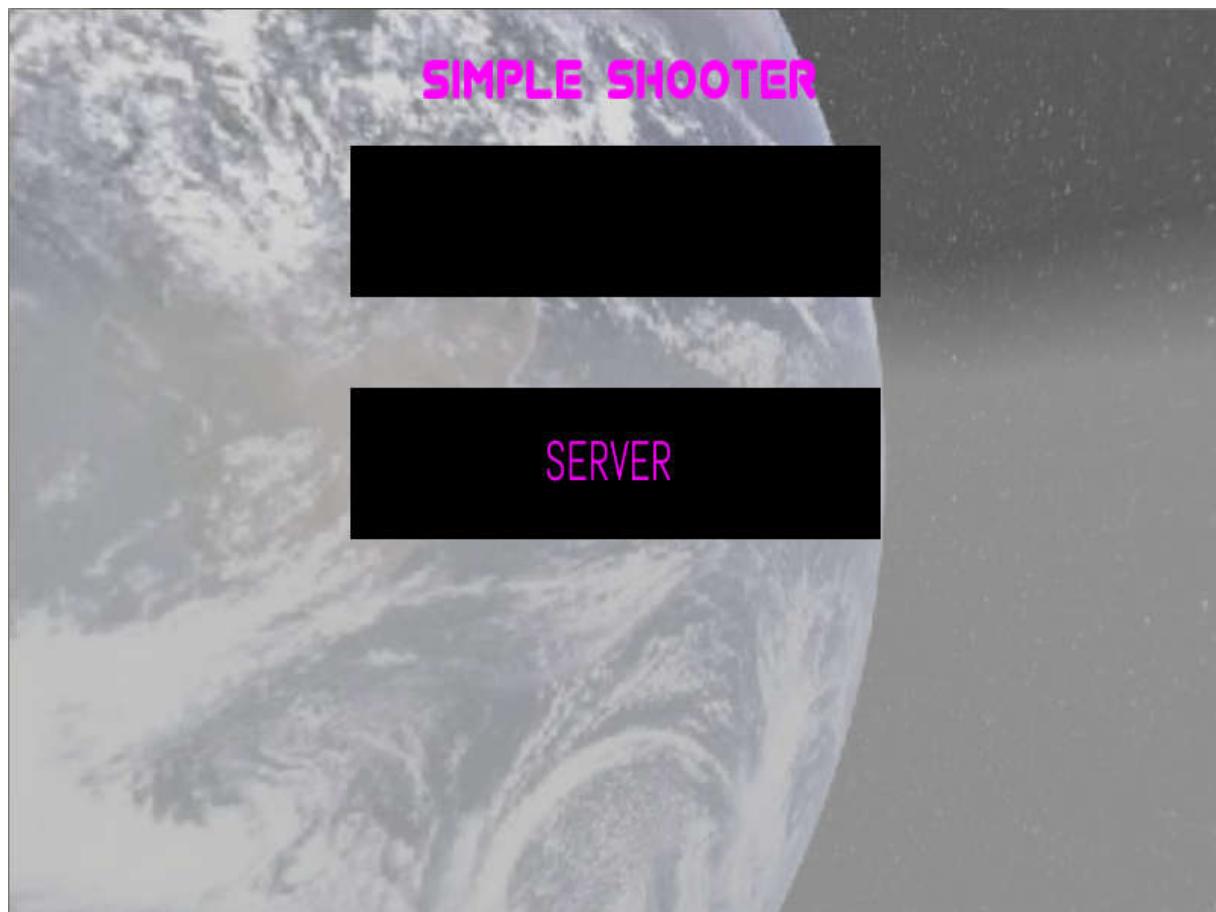
Pierwszym oknem, na które trafi użytkownik będzie menu główne:



Użytkownik ma do dyspozycji 2 przyciski

- New Game - rozpoczyna nową grę
- Exit - wychodzi z gry

Po wybraniu **New Game**, użytkownik przechodzi do kolejnego okna, gdzie może wybrać, czy chce być serwerem, czy klientem:



## Serwer

Jako serwer, użytkownik czeka na nawiązanie połączenia. Przy użyciu klawisza <Backspace> może wycofać się do ekranu wyboru roli:



Jeżeli zostanie nawiązane połączenie z klientem, rozpocznie się gra.

## Klient

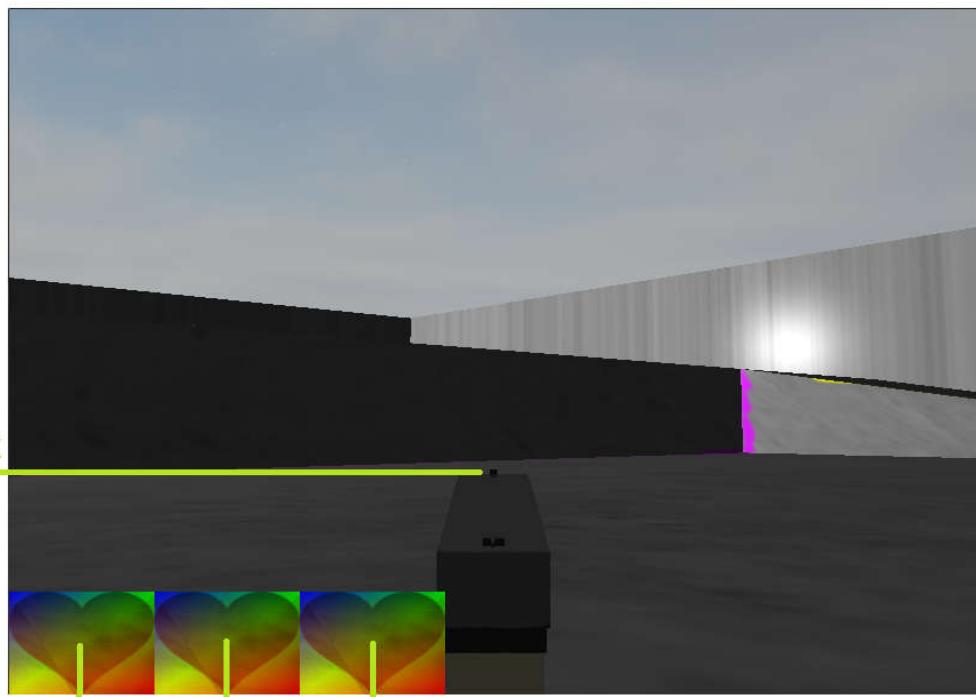
Klientowi zostaje przedstawiony ekran, na którym wpisuje adres IP serwera:



Po wpisaniu adresu IP i potwierdzeniu klawiszem <Enter>, zostanie podjęta próba nawiązania połączenia. Jeżeli się powiedzie, rozpocznie się gra.

## Gra

Na ekranie widoczne są następujące informacje:



## Przeciwnik



Celem gry jest trzykrotne trafienie drugiego gracza. Pierwszy, który to zrobi - wygrywa.

Gracz porusza się przy użyciu klawiszy WASD, skacze Spacją i strzela LPM.

## Specyfikacja wewnętrzna

Gra zdefiniowana jest w następujących plikach:

Pliki nagłówkowe .h	Pliki źródłowe .cpp
Bullet.h	Bullet.cpp
Camera.h	Camera.cpp
Collision.h	DirLight.cpp
DirLight.h	Entity.cpp
Entity.h	Flashlight.cpp
Flashlight.h	Fog.cpp
Fog.h	Geometry.cpp
Geometry.h	HeightMap.cpp
HeightMap.h	HUD.cpp
HUD.h	Image2D.cpp
Image2D.h	LightObject.cpp
LightObject.h	Loader.cpp
Loader.h	Mesh.cpp
Mesh.h	Model.cpp
Model.h	Networking.cpp
Networking.h	ResourceManager.cpp
ResourceManager.h	Scene.cpp
Scene.h	SceneExit.cpp
SceneExit.h	SceneGame.cpp
SceneGame.h	SceneLoad.cpp
SceneMenu.h	SceneManager.cpp
SceneMulti.h	SceneMenu.cpp
SceneMultiplayerGame.h	SceneMulti.cpp
Shader.h	SceneMultiPlayerGame.cpp
shaders.h	Shader.cpp
Skybox.h	shaders.cpp
Sound.h	Skybox.cpp
SoundManager.h	Sound.cpp
SpriteRenderer.h	SoundManager.cpp
SpriteTexture.h	SpriteTexture.cpp
Terrain.h	Terrain.cpp
TextManager.h	TextManager.cpp
TextureClass.h	TextureClass.cpp
vertexBufferObject.h	vertexBufferObject.cpp
	tutorial.cpp

## Użyte biblioteki

W projekcie użyliśmy następujące biblioteki zewnętrzne:

- SDL2 (+\_net, \_image, \_mixer) - obsługa okna, sieci, ładowania obrazów i dźwięku
- Freetype2 - generowanie tekstu z plików .ttf
- Glew - dostarcza funkcje do obsługi OpenGl w najnowszej wersji
- GLM - operacje matematyczne na macierzach, wektorach i przekształcenia przydatne przy pracy z OpenGL

- Assimp - ładowanie modeli 3D z dysku

## Klasy

### Bullet

**Bullet.h**

```
class Bullet
{
public:
    Bullet(glm::vec3 pos1, glm::vec3 vector);
    Bullet(float verticesArray[6]);
    ~Bullet();
    static void setShader(Shader* shader);
    void draw(const glm::mat4 &projectionMatrix, const glm::mat4& viewMatrix,
const glm::vec3& color);
    float vertices[6];

private:
    unsigned int vao;
    unsigned int vbo;
    static Shader* shaderRef;
    bool needInitialising = false;
};
```

Klasa ta symbolizuje pocisk

### Konstruktory

**Bullet(glm::vec3 pos1, glm::vec3 vector)**

Tworzy pocisk w oparciu o pozycję startową i wektor kierunku lotu pocisku. Parametry:

- *pos1* – wektor określający pozycję kamery
- *vector* – wektor określający kierunek lotu pocisku

**Bullet(float verticesArray[6])**

Tworzy pocisk w oparciu o dwa wierzchołki. Parametry:

- *verticesArray* – macierz zawierająca pozycję dwóch wierzchołków definiujących pocisk

### Funkcje

**static void setShader(Shader\* shader)**

Funkcja ustawia shader, przy pomocy którego pocisk zostanie wyświetlony. Parametry:

- *shader* – wskaźnik na shader

**void draw(const glm::mat4 &projectionMatrix, const glm::mat4& viewMatrix, const glm::vec3& color)**

Funkcja ustawia shader, przy pomocy którego pocisk zostanie wyświetlony. Parametry:

- *projectionMatrix* – macierz perspektywy
- *viewMatrix* – macierz widoku
- *color* – wektor przechowujący informację o kolorze pocisku

## Camera

**Camera.h**

```
class Camera
{
public:
    typedef std::bitset<4> Movement;
    enum MovementBits {
```

```

        MOVE_FORWARD = 0,
        MOVE_BACKWARD = 1,
        STRAFE_LEFT = 2,
        STRAFE_RIGHT = 3
    };
    Camera(glm::vec3 position, glm::vec3 front, glm::vec3 up, float
sensitivity, float moveSpeed);
    ~Camera();
    glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
    glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
    glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
    glm::vec3 cameraRight;
    glm::vec3 movementFront = glm::vec3(0.0f, 0.0f, 1.0f);
    float yaw = -90.0f;
    float pitch = 0.0f;
    float sensitivity = 0.1f;
    float moveSpeed = 1.0f;
    void turnCamera(SDL_MouseMotionEvent &event);
    void moveCamera(Movement move, float deltaTime);
    glm::mat4 getViewMatrix() const;
    void updateCameraVectors();
};

Klasa odpowiada za obsługę kamery w przestrzeni 3D

```

### Konstruktory

```
Camera(glm::vec3 position, glm::vec3 front, glm::vec3 up, float sensitivity, float
moveSpeed);
```

Standardowy konstruktor. Parametry:

- *position*- wektor określający pozycję kamery
- *front*-wektor określający kierunek, w którym spogląda kamera
- *up*-wektor wskazujący na górę kamery
- *sensitivity*-czułość kamery na ruchy myszki
- *moveSpeed*-prędkość poruszania się kamery w świecie gry

### Typy wyliczeniowe

```
enum MovementBits {
    MOVE_FORWARD = 0,
    MOVE_BACKWARD = 1,
    STRAFE_LEFT = 2,
    STRAFE_RIGHT = 3
};
```

Typ wyliczeniowy określający bit, który należy zmienić w polu Movement, aby uaktywnić bądź dezaktywować ruch w określonym kierunku.

### Funkcje

```
void turnCamera(SDL_MouseMotionEvent &event);
```

Funkcja obraca kamerę w oparciu o ruch myszki. Parametry:

- *event*-wydarzenie, w oparciu o które ma zostać zmieniony punkt, na który ma patrzeć kamera

```
void moveCamera(Movement move, float deltaTime);
```

Funkcja wykonuje ruch kamery (kamera chodzi). Parametry:

- *move*-kierunek, w którym ma podążać kamera
- *deltaTime*-czas od poprzedniej klatki

```
glm::mat4 getViewMatrix() const;
```

Funkcja zwraca macierz widoku, w oparciu o którą przeprowadzane są transformacje na widzialnych obiektach.

**Wartość zwracana:** macierz 4x4 będąca macierzą widoku

```
void updateCameraVectors();
```

Funkcja aktualizuje wektor kierunku patrzenia kamery w oparciu o najnowsze dane.

## Entity

Entity.h

```
class Entity
{
public:
    Entity(Model& model, glm::vec3 position, float rotation, glm::vec3 scale);
    ~Entity() = default;
    void rotateX(float radians);
    void rotateY(float radians);
    void rotateZ(float radians);
    void setRotationX(float radians);
    void setRotationY(float radians);
    void setRotationZ(float radians);
    void setShader(Shader& shader);
    void setShader(Shader* shader);
    virtual void Draw(glm::mat4 &projectionMatrix, glm::mat4& viewMatrix);
    virtual void Draw(glm::mat4 &projectionMatrix, glm::mat4& viewMatrix, const
LightObject& lightObject, const Camera& camera);
    Shader* GetShader();
    glm::vec3 getPosition();
    void setPosition(const glm::vec3& pos);
protected:
    Model& model;
    void prepareModelMatrix();
    glm::mat4 modelMatrix;
    glm::vec3 position;
    glm::vec3 scale;
    glm::vec3 rotation;
    glm::vec3 rotationAngle;
    Shader* shader;
};
```

Klasa reprezentuje podstawowy obiekt w świecie gry

## Konstruktory

```
Entity(Model& model, glm::vec3 position, float rotation, glm::vec3 scale);
```

Standardowy konstruktor. Parametry:

- *model*- referencja na załadowany model 3D obiektu
- *position*-wektor reprezentujący pozycję obiektu w świecie gry
- *rotation*-rotacja obiektu względem osi Y
- *scale*-wektor pozwalający na przeskalowanie obiektu

## Funkcje

```
void rotateX(float radians);
```

Funkcja obraca model względem osi X. Parametry:

- *radians*-kąt obrotu wyrażony w radianach

```
void rotateY(float radians);
```

Funkcja obraca model względem osi Y. Parametry:

- *radians*-kąt obrotu wyrażony w radianach

```
void rotateZ(float radians);
```

Funkcja obraca model względem osi Z. Parametry:

- *radians*-kąt obrotu wyrażony w radianach

```
void setRotationX(float radians);
```

Funkcja ustawia rotację obiektu względem osi X. Parametry:

- *radians*-kąt obrotu wyrażony w radianach

```
void setRotationY(float radians);
```

Funkcja ustawia rotację obiektu względem osi Y. Parametry:

- *radians*-kąt obrotu wyrażony w radianach

```
void setRotationZ(float radians);
```

Funkcja ustawia rotację obiektu względem osi Z. Parametry:

- *radians*-kąt obrotu wyrażony w radianach

```
void setShader(Shader& shader);
```

Funkcja przypisuje do obiektu shader, który ma zostać użyty do wyświetlenia obiektu. Parametry:

- *shader*-referencja na obiekt typu Shader

```
void setShader(Shader* shader);
```

Funkcja przypisuje do obiektu shader, który ma zostać użyty do wyświetlenia obiektu. Parametry:

- *shader*-wskaźnik na obiekt typu Shader

```
virtual void Draw(glm::mat4 &projectionMatrix, glm::mat4& viewMatrix);
```

Funkcja wyświetlająca obiekt na ekranie. Parametry:

- *projectionMatrix*-macierz określająca perspektywę
- *viewMatrix*-macierz określająca pozycję obiektu w świecie gry

```
virtual void Draw(glm::mat4 &projectionMatrix, glm::mat4& viewMatrix, const LightObject& lightObject, const Camera& camera);
```

Funkcja wyświetlająca obiekt na ekranie, biorąca pod uwagę oświetlenie. Parametry:

- *projectionMatrix*-macierz określająca perspektywę
- *viewMatrix*-macierz określająca pozycję obiektu w świecie gry
- *lightObject*-obiekt reprezentujący światło
- *camera*-obiekt reprezentujący używaną kamerę, potrzebny do obliczenia odbić światła

```
Shader* GetShader();
```

Funkcja zwracająca aktualnie używany shader.

**Wartość zwracana:** wskaźnik na aktualnie używany shader.

```
glm::vec3 getPosition();
```

Funkcja zwracająca aktualną pozycję.

**Wartość zwracana:** wektor zawierający pozycję obiektu w świecie gry.

```
void setPosition(const glm::vec3& pos);
```

Funkcja ustawia pozycję obiektu. Parametry:

- *pos*-wektor zawierający pozycję obiektu w świecie gry

```
void prepareModelMatrix();
```

Funkcja przygotowuje macierz modelu do wyświetlenia, biorąc pod uwagę jego rotację.

## Image2D

Image2D.h

```
class Image2D
{
public:
    Image2D(float xMin, float xMax, float yMin, float yMax);
    void SetTexture(unsigned int textureId);
    void Draw(Shader &shader);
    ~Image2D();

private:
    unsigned int textureId;
    bool textureSet = false;
    float vertices[20];
    unsigned int vao;
    unsigned int vbo;
    unsigned int ebo;
    unsigned int indices[6] = {
        0, 1, 3,
        1, 2, 3
    };
};
```

Jedna z klas przeznaczona do wyświetlania tekstur na ekranie. Obiekowi ustala się pozycje na ekranie, a następnie można zmieniać mu wyświetlane tekstury.

### Konstruktory

```
Image2D(float xMin, float xMax, float yMin, float yMax);
```

Standardowy konstruktor. Parametry:

- *xMin*-pozycja lewej krawędzi tekstury
- *xMax*-pozycja prawej krawędzi tekstury
- *yMin*-pozycja dolnej krawędzi tekstury
- *yMax*-pozycja górnej krawędzi tekstury

### Funkcje

```
void SetTexture(unsigned int textureId);
```

Funkcja przypisuje teksturę do obiektu. Parametry:

- *textureId* - tekstura, którą należy wyświetlić

```
void Draw(Shader &shader);
```

Funkcja wyświetla teksturę na ekranie. Parametry:

- *shader* - shader, który ma zostać użyty do wyświetlenia tekstury

## LightObject

LightObject.h

```

class LightObject : public Entity
{
public:
    LightObject (Model& model, glm::vec3 position, float rotation, glm::vec3 scale, Shader *lightShader, glm::vec3 lightColor);
    ~LightObject ();
    void Draw(glm::mat4 &projectionMatrix, glm::mat4& viewMatrix) override;
    glm::vec3 GetPosition() const;
    glm::vec3 GetColor() const;
private:
    glm::vec3 lightColor;
};

```

Klasa dziedzicząca z klasy Entity, symbolizuje obiekt będący źródłem światła.

### **Konstruktory**

```
LightObject (Model& model, glm::vec3 position, float rotation, glm::vec3 scale,
Shader *lightShader, glm::vec3 lightColor)
```

Standardowy konstruktor. Parametry:

- *model*- referencja na załadowany model 3D obiektu
- *position*-wektor reprezentujący pozycję obiektu w świecie gry
- *rotation*-rotacja obiektu względem osi Y
- *scale*-wektor pozwalający na przeskalowanie obiektu
- *shader* - wskaźnik na shader, który ma zostać użyty do wyświetlenia obiektu
- *lightColor* - wektor zawierający kolor źródła światła

### **Funkcje**

```
void Draw(glm::mat4 &projectionMatrix, glm::mat4& viewMatrix)
```

Funkcja wyświetlająca świecący obiekt na ekranie. Parametry:

- *projectionMatrix*-macierz określająca perspektywę
- *viewMatrix*-macierz określająca pozycję obiektu w świecie gry

```
glm::vec3 GetPosition()
```

Funkcja zwracająca pozycję obiektu.

**Wartość zwracana:** wektor zawierający pozycję obiektu w świecie gry

```
glm::vec3 GetColor()
```

Funkcja zwracająca kolor, na który świeci obiekt.

**Wartość zwracana:** wektor zawierający kolor świecenia

### **Loader**

Loader.h

```

class Loader {
public:
    static bool loadModels();
    static bool loadShaders();
    static bool unloadShaders();
    static bool unloadModels();
    static bool loadTextures2D();
    static bool unloadTextures2D();
    enum LoadedModels {
        CUBE,
        GUN,
        PLAYER,
        MODELS_COUNT
    };
}
```

```

enum LoadedShaders {
    OUR,
    LIGHT,
    BOUNDING_BOX,
    SIMPLE,
    BULLET,
    IMAGE,
    SKYBOX,
    SHADER2D,
    SHADER2D2,
    MENUBOX,
    SHADERS_COUNT
};

enum LoadedTextures2D {
    JA_WON,
    PRZECIWNIK_WON,
    TEXTURES2D_COUNT
};

static Model& getModel(enum LoadedModels modelId);
static Shader& getShader(enum LoadedShaders shaderId);
static unsigned int getTexture2D(enum LoadedTextures2D textureId);

private:
    Loader() = default;
    ~Loader() = default;
    static std::string modelPath[MODELS_COUNT];
    static std::string vertexShaderPath[SHADERS_COUNT];
    static std::string fragmentShaderPath[SHADERS_COUNT];
    static std::string textures2DPath[TEXTURES2D_COUNT];
    static std::string textures2DName[TEXTURES2D_COUNT];

    static std::vector<Model> models;
    static std::vector<Shader> shaders;
    static std::vector<unsigned int> textures2D;
};

Klasa ładująca do pamięci tekstury, shadery i modele 3D. Jest to klasa statyczna

```

### *Typy wyliczeniowe*

```

enum LoadedModels {
    CUBE,
    GUN,
    PLAYER,
    MODELS_COUNT
};

```

Typ wyliczeniowy definiujący załadowane modele

```

enum LoadedShaders {
    OUR,
    LIGHT,
    BOUNDING_BOX,
    SIMPLE,
    BULLET,
    IMAGE,
    SKYBOX,
    SHADER2D,
    SHADER2D2,
    MENUBOX,
    SHADERS_COUNT
};

```

Typ wyliczeniowy definiujący załadowane shadery

```

enum LoadedTextures2D {
    JA_WON,

```

```
    PRZECIWNIK_WON,  
    TEXTURES2D_COUNT
```

```
};
```

Typ wyliczeniowy definiujący załadowane tekstury

## Funkcje

```
static bool loadModels()
```

Funkcja ładująca modele zdefiniowane w odpowiednim typie wyliczeniowym

```
static bool loadShaders()
```

Funkcja ładująca shadery zdefiniowane w odpowiednim typie wyliczeniowym

```
static bool loadTextures2D()
```

Funkcja ładująca tekstury zdefiniowane w odpowiednim typie wyliczeniowym

```
static bool unloadModels()
```

Funkcja zwalniająca pamięć, w której znajdowały się modele

```
static bool unloadShaders()
```

Funkcja zwalniająca pamięć, w której znajdowały się shadery

```
static bool unloadTextures2D()
```

Funkcja zwalniająca pamięć, w której znajdowały się tekstury

```
static Model& getModel(enum LoadedModels modelId)
```

Funkcja zwraca model o danym id. Parametry:

- *modelId*-id modelu do załadowania, zdefiniowane w typie wyliczeniowym

**Wartość zwracana:** referencja na obiekt typu Model zawierający informacje o modelu

```
static Shader& getShader(enum LoadedShaders shaderId)
```

Funkcja zwraca shader o danym id. Parametry:

- *shaderId*-id shadera do załadowania, zdefiniowane w typie wyliczeniowym

**Wartość zwracana:** referencja na obiekt typu Shader

```
static unsigned int getTexture2D(enum LoadedTextures2D textureId)
```

Funkcja zwraca teksturę o danym id. Parametry:

- *textureId*-id tekstury do załadowania, zdefiniowane w typie wyliczeniowym

**Wartość zwracana:** wartość typu unsigned int, wskazująca na pozycję tekstury w pamięci

## Mesh

Mesh.h

```
struct Vertex {  
    glm::vec3 Position;  
    glm::vec3 Normal;  
    glm::vec2 TexCoords;  
    glm::vec3 Tangent;  
    glm::vec3 Bitangent;
```

```

};

struct Texture {
    unsigned int id;
    std::string type;
    std::string path;
};

struct MaterialProperties {
    glm::vec3 ambient = glm::vec3(0.0f);
    glm::vec3 diffuse = glm::vec3(0.0f);
    glm::vec3 specular = glm::vec3(0.0f);
    float shininess;
    float specularTexture = 1.0f;
};

class Mesh {
public:

    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    std::vector<Texture> textures;
    Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices,
        std::vector<Texture> textures, MaterialProperties materialProperties);
    ~Mesh();
    void clearBuffers();
    void Draw(Shader shader);
private:
    MaterialProperties materialProperties;
    unsigned int VAO, VBO, EBO;
    void setupMesh();
};

```

Klasa definiująca siatkę modelu 3D

## Konstruktory

```
Mesh(std::vector<Vertex> vertices, std::vector<unsigned int> indices,
    std::vector<Texture> textures, MaterialProperties materialProperties)
```

Standardowy konstruktor. Parametry:

- *vertices* – wektor zawierający wierzchołki siatki
- *indices* – wektor zawierający numery kolejno rysowanych wierzchołków
- *textures* – wektor zawierający informacje o teksturach, których używa ta siatka
- *materialProperties* – właściwości materiału, posiadanego przez siatkę (np. stopień odbicia światła)

## Struktury

```

struct Vertex {
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
    glm::vec3 Tangent;
    glm::vec3 Bitangent;
};

```

Struktura przechowująca podstawowe informacje o wierzchołku

```

struct Texture {
    unsigned int id;
    std::string type;
    std::string path;
};

```

Struktura przechowująca podstawowe informacje o teksturze

```

struct MaterialProperties {
    glm::vec3 ambient = glm::vec3(0.0f);

```

```

        glm::vec3 diffuse = glm::vec3(0.0f);
        glm::vec3 specular = glm::vec3(0.0f);
        float shininess;
        float specularTexture = 1.0f;
    };

```

Struktura przechowująca podstawowe informacje o właściwościach materiału, z którego “zrobiona jest” siatka

## Funkcje

`void clearBuffers()`

Funkcja zwalniająca pamięć, w której przechowywane były informacje o siatce. Nie można tego zrobić w konstruktorze, ponieważ obiekt zawierający informacje o siatce powinien być przekazywany do funkcji wyświetlających przez wartość.

`void Draw(Shader shader)`

Funkcja wyświetla siatkę na ekranie wykorzystując podany shader. Parametry:

- *shader*-obiekt reprezentujący shader, przy pomocy którego ma zostać wyświetlona siatka

`void setupMesh()`

Funkcja inicjująca struktury openGL danymi o siatce.

## Model

Model.h

```

unsigned int TextureFromFile(const char *path, const std::string &directory, bool
gamma = false);
class Model
{
public:
    std::vector<Texture> textures_loaded;
    std::vector<Mesh> meshes;
    std::string directory;
    bool gammaCorrection;
    Model(std::string const &path, bool gamma = false);
    ~Model();
    void Draw(Shader shader);

private:
    void loadModel(std::string const &path);
    void processNode(aiNode *node, const aiScene *scene);
    Mesh processMesh(aiMesh *mesh, const aiScene *scene);
    std::vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType
type, std::string typeName);
};

```

Klasa definiująca model 3D, czyli zbiór siatek (klasa Mesh). Ponadto, zdefiniowana jest tutaj funkcja ładująca teksturę z pliku na dysku do pamięci

## Konstruktory

`Model(std::string const &path, bool gamma = false)`

Standardowy konstruktor. Parametry:

- *path*-zmienna typu *string* zawierająca ścieżkę do pliku z modelem na dysku
- *gamma*-zmienna określająca, czy podczas ładowania trzeba znormalizować współczynnik gamma

## Funkcje

`void Draw(Shader shader)`

Funkcja wyświetla model (a właściwie siatki, z których składa się model) na ekranie wykorzystując podany shader. Parametry:

- *shader*-obiekt reprezentujący shader, przy pomocy którego ma zostać wyświetlony model

```
void loadModel(std::string const &path)
```

Funkcja ładowająca model z pliku na dysku. Parametry:

- *path* - zmienna typu *string* zawierająca ścieżkę do pliku z modelem na dysku

```
void processNode(aiNode *node, const aiScene *scene)
```

Funkcja przechodząca przez drzewistą strukturę sceny, zimportowanej biblioteką Assimp. Parametry:

- *node* - wskaźnik na kolejny węzeł w strukturze sceny
- *scene* - wskaźnik na przetwarzaną scenę

```
Mesh processMesh(aiMesh *mesh, const aiScene *scene)
```

Funkcja odczytująca ze struktury sceny informacje o siatce. Parametry:

- *mesh* - wskaźnik na siatkę, zdefiniowany w węźle sceny
- *scene* - wskaźnik na przetwarzaną scenę

```
std::vector<Texture> loadMaterialTextures(aiMaterial *mat, aiTextureType type,
                                             std::string typeName)
```

Funkcja ładowająca tekstury skojarzone z danym materiałem. Parametry:

- *mat* - wskaźnik na aktualnie przetwarzany materiał
- *type* - wskaźnik na typ materiału, który jest ładowany
- *typeName* - nazwa na dany typ materiału, używana później przy ładowaniu danych do shaderów

**Wartość zwracana:** wektor zawierający tekstury

```
unsigned int TextureFromFile(const char *path, const std::string &directory, bool gamma = false);
```

Funkcja ładowająca teksturę z pliku do pamięci. Parametry:

- *path* - nazwa pliku z tekstem
- *directory* - ścieżka/katalog zawierający plik z tekstem
- *gamma* - wartość określająca, czy należy znormalizować współczynnik gamma przy ładowaniu

**Wartość zwracana:** wartość typu *unsigned int* wskazująca na miejsce w pamięci, w którym znajduje się tekstura

## Networking

Networking.h

```
class Networking
{
public:
    enum MessageType {
        POSITION_FOLLOW,
        QUIT,
        REQ_DATA_SIZE_CHANGE,
        HIT,
        NO_HEALTH,
        SHOOT
    };
    static bool startServer(Uint16 port);
    static bool stopServer();
    static bool connect(const std::string hostname, Uint16 port);
    static bool closeConnection();
    static void setDataSize(int newSize);
    static bool sendControlMsg(enum MessageType msg);
    static bool sendData(void* data);
```

```

        static bool recvControlMsg(enum MessageType* msg);
        static bool recvData(void* data);
        static bool acceptConnection();

private:
    Networking();
    ~Networking();
    static int dataSize;
    static bool serverStarted;
    static bool isConnected;
    static IPaddress ip;
    static TCPsocket serverSock;
    static TCPsocket connectionSock;
};

Klasa służąca do obsługi sieci. Jest to klasa statyczna (bez względu na ilość wątków, obsługa sieci powinna być realizowana przez jeden obiekt).

```

### **Typy wyliczeniowe**

```

enum MessageType {
    POSITION_FOLLOW,
    QUIT,
    REQ_DATA_SIZE_CHANGE,
    HIT,
    NO_HEALTH,
    SHOOT
};

```

Typ wyliczeniowy definiujący komunikaty kontrolne, wysyłane między klientami podczas gry.

### **Funkcje**

```
static bool startServer(Uint16 port);
```

Funkcja uruchamiająca serwer nasłuchujący na danym porcie. Parametry:

- *port* - nr portu, na którym ma nasłuchiwać serwer

**Wartość zwracana:** wartość typu *bool* informująca, czy serwer udało się uruchomić

```
static bool stopServer();
```

Funkcja zatrzymująca serwer.

**Wartość zwracana:** wartość typu *bool* informująca, czy serwer udało się zatrzymać

```
static bool connect(const std::string hostname, Uint16 port);
```

Funkcja nawiązująca połączenie z serwerem. Parametry:

- *hostname*- adres ip/nazwa serwera
- *port* - nr portu, na którym nasłuchuje serwer

**Wartość zwracana:** wartość typu *bool* informująca, czy udało się połączyć

```
static bool closeConnection();
```

Funkcja zamkająca trwające połączenie.

**Wartość zwracana:** wartość typu *bool* informująca, czy połączenie udało się zamknąć

```
static void setDataSize(int newSize);
```

Funkcja ustawia wielkość paczki danych wysyłanej/odbieranej. Wielkość ta powinna być taka sama po obydwóch stronach połączenia. Parametry:

- *newSize*- nowy rozmiar paczki (w bajtach)

```
static bool sendControlMsg(enum MessageType msg);
```

Funkcja wysyłająca wiadomość kontrolną. Parametry:

- *msg*- wiadomość kontrolna, zdefiniowana w typie wyliczeniowym

**Wartość zwracana:** wartość typu *bool* informująca, czy komunikacja przebiega bez błędów

```
static bool sendData(void* data);
```

Funkcja wysyłająca dane. Parametry:

- *data*- wskaźnik na dane, które chcemy wysłać

**Wartość zwracana:** wartość typu *bool* informująca, czy komunikacja przebiega bez błędów

```
static bool recvControlMsg(enum MessageType* msg);
```

Funkcja odbierająca wiadomość kontrolną. Parametry:

- *msg* - wskaźnik na zmienną, do której ma zostać zapisana wiadomość

**Wartość zwracana:** wartość typu *bool* informująca, czy komunikacja przebiega bez błędów

```
static bool recvData(void* data);
```

Funkcja odberająca dane. Parametry:

- *data*- wskaźnik na zmienną, do której mają zostać zapisane odebrane dane

**Wartość zwracana:** wartość typu *bool* informująca, czy komunikacja przebiega bez błędów

```
static bool acceptConnection();
```

Funkcja sprawdzająca, czy na serwerze znajduje się połączenie oczekujące. Jeżeli takie istnieje, zostaje ono nawiązane.

**Wartość zwracana:** wartość typu *bool* informująca, czy zostało nawiązane połączenie

## Scene

Scene.h

```
class Scene
{
public:
    Scene(glm::mat4 projectionMatrix, Camera* camera);
    ~Scene();
    LightObject* SetLight(LightObject* light);
    LightObject* GetLight();
    Entity* addObject(Entity* object);
    void removeObject(Entity* object);
    Camera* getCamera();
    glm::mat4& GetViewMatrix();
    glm::mat4& GetProjectionMatrix();
    Terrain* addTerrain(Terrain* ptrTerrain);
    void DrawObjects();
    void movePlayer(Camera::Movement move, float deltaTime);
    void playerJumpStart();
    virtual void InitScene();
    virtual void UnInitScene() = 0;
    virtual void handleEvents(SDL_Event& e);
    virtual void render();
    bool run = true;
private:
    std::forward_list<Entity*> objects; // docelowo lista list dla kazdego
znanego typu
    LightObject* light;
    Camera* camera;
    glm::mat4 viewMatrix;
    glm::mat4 projectionMatrix;
    Terrain* terrain;
```

```

        float jumpTime;
        bool notJumping = true;
        bool notFalling = true;
        float jumpStartPos;
        float jumpHeight = 6.0f;
    };

```

Klasa definiująca scenę. Stanowi podstawę, która powinna być dziedziczona przez sceny obecne w grze.

### **Konstruktory**

```
Scene(glm::mat4 projectionMatrix, Camera* camera)
```

Standardowy konstruktor. Parametry:

- *projectionMatrix*- macierz perspektywy
- *camera*- wskaźnik na kamerę używaną w scenie

### **Funkcje**

```
LightObject* SetLight(LightObject* light)
```

Funkcja dodaje do sceny oświetlenie. Parametry:

- *light*- wskaźnik na obiekt typu *LightObject*, definiujący oświetlenie

**Wartość zwracana:** wskaźnik na dodane oświetlenie

```
LightObject* GetLight();
```

Funkcja zwracająca obiekt definiujący oświetlenie sceny.

**Wartość zwracana:** wskaźnik na obiekt definiujący oświetlenie sceny

```
Entity* addObject(Entity* object);
```

Funkcja dodaje do sceny obiekt. Parametry:

- *object*- wskaźnik na obiekt

**Wartość zwracana:** wskaźnik na dodany obiekt

```
void removeObject(Entity* object);
```

Funkcja usuwa ze sceny obiekt. Parametry:

- *object*- wskaźnik na obiekt

```
Camera* getCamera();
```

Funkcja zwracająca kamerę używaną w scenie.

**Wartość zwracana:** wskaźnik na kamerę używaną w scenie

```
glm::mat4& GetViewMatrix();
```

Funkcja zwracająca macierz widoku.

**Wartość zwracana:** referencja na macierz będącą macierzą widoku

```
glm::mat4& GetProjectionMatrix();
```

Funkcja zwracająca macierz perspektywy.

**Wartość zwracana:** referencja na macierz będącą macierzą perspektywy

```
Terrain* addTerrain(Terrain* ptrTerrain);
```

Funkcja dodaje do sceny teren. Parametry:

- *ptrTerrain*- wskaźnik na teren

**Wartość zwracana:** wskaźnik na dodany teren

```
void DrawObjects();
```

Funkcja wyświetlająca na ekranie wszystkie obiekty dodane do sceny.

```
void movePlayer(Camera::Movement move, float deltaTime);
```

Funkcja przesuwa gracza, czyli pozycję kamery. Parametry:

- *move* - pole bitowe określające kierunek, w którym porusza się kamera
- *deltaTime* - czas, który upłynął od poprzedniej klatki

```
void playerJumpStart();
```

Funkcja rozpoczynająca skok gracza (jeżeli jest na ziemi).

```
virtual void InitScene();
```

Funkcja inicjująca scenę (przed przełączeniem się na nią). Każda scena powinna definiować własną wersję tej funkcji.

```
virtual void UnInitScene()
```

Funkcja zwalniająca pamięć po scenie, jest używana podczas przełączania do innej sceny. Każda scena powinna definiować własną wersję tej funkcji.

```
virtual void handleEvents(SDL_Event& e);
```

Funkcja obsługująca interakcję użytkownika w obrębie tej sceny. Parametry:

- *e* - referencja na zmienną przechowującą informacje o pojedynczym zdarzeniu
- Każda scena powinna definiować własną wersję tej funkcji.

```
virtual void render();
```

Funkcja renderująca zawartość sceny na ekran. Każda scena powinna definiować własną wersję tej funkcji.

## SceneMultiplayerGame

SceneMultiplayerGame.h

```
class SceneMultiplayerGame :  
    public Scene  
{  
public:  
    SceneMultiplayerGame(glm::mat4 projectionMatrix, Camera* camera);  
    ~SceneMultiplayerGame();  
    virtual void InitScene();  
    virtual void UnInitScene();  
    virtual void handleEvents(SDL_Event& e);  
    virtual void render();  
    void setServer(bool value);  
private:  
    float ostatniWystrzal = 0.0f;  
    bool quit = false;  
    float deltaTime = 0.0f;  
    float lastFrame = 0.0f;  
    float currentFrame = 0.0f;  
    Camera::Movement nextMove;  
    bool isServer = false;
```

```

Entity * player2;
void serverConnectionHandlerThread();
void connectionHandlerThread();
std::mutex loopLock;
std::mutex posLock;
std::mutex playerPosLock;
std::mutex quitLock;
float pos[4] = { 0.0f, 0.0f, 0.0f, 0.0f };
float playerPos[4] = { 0.0f };
std::thread* connectionThread;
std::thread* tmpThreadObject;
std::mutex bulletLock;
std::deque<Bullet*> bulletContainer;
void deleteBullet();
bool sendBullet = false;
bool hit;
int health;
Bullet* tmpBulletPtr;
bool scoreBoard;
Image2D* imgPtr;
SoundManager* SoundM;
Sound* SoundMgr;
HUD *life1, *life2, *life3;
std::vector<reference_wrapper<const Shader>> SH;
CSkybox MainSkybox;

};


```

Scena, w której odbywa się rywalizacja graczy.

### Konstruktory

```
SceneMultiplayerGame(glm::mat4 projectionMatrix, Camera* camera)
```

Standardowy konstruktor. Parametry:

- *projectionMatrix*- macierz perspektywy
- *camera*- wskaźnik na kamerę używaną w scenie

### Funkcje

```
virtual void InitScene()
```

Funkcja inicjująca scenę (przed przełączeniem się na nią).

```
virtual void UnInitScene()
```

Funkcja zwalniająca pamięć po scenie, jest używana podczas przełączania do innej sceny.

```
virtual void handleEvents(SDL_Event& e)
```

Funkcja obsługująca interakcję użytkownika w obrębie tej sceny. Parametry:

- *e* - referencja na zmienną przechowującą informacje o pojedynczym zdarzeniu

```
virtual void render()
```

Funkcja renderująca zawartość sceny na ekran.

```
void serverConnectionHandlerThread()
```

Funkcja obsługująca połączenie z klientem. Jest uruchamiana na osobnym wątku.

```
void connectionHandlerThread()
```

Funkcja obsługująca połączenie z serwerem. Jest uruchamiana na osobnym wątku.

## Shader

### Shader.h

```
class Shader
{
public:
    unsigned int ID;
    Shader() { }
    Shader &Use();
    Shader(const char* vertexPath, const char* fragmentPath);
    ~Shader() = default;
    void use();
    void setBool(const std::string &name, bool value) const;
    void setInt(const std::string &name, int value) const;
    void setFloat(const std::string &name, float value) const;
    void setVec2(const std::string &name, const glm::vec2 &value) const;
    void setVec2(const std::string &name, float x, float y) const;
    void setVec3(const std::string &name, const glm::vec3 &value) const;
    void setVec3(const std::string &name, float x, float y, float z) const;
    void setVec4(const std::string &name, const glm::vec4 &value) const;
    void setVec4(const std::string &name, float x, float y, float z, float w);
    void setMat2(const std::string &name, const glm::mat2 &mat) const;
    void setMat3(const std::string &name, const glm::mat3 &mat) const;
    void setMat4(const std::string &name, const glm::mat4 &mat) const;
    void Compile(const char* vertexPath, const char* fragmentPath);
    void SetInteger(const char *name, int value, bool useShader = false);
    void SetMatrix4(const char *name, const glm::mat4 &matrix, bool useShader
= false);
    void SetVector3f(const char *name, const glm::vec3 &value, bool useShader
= false);
private:
    void checkCompileErrors(GLuint shader, std::string type);
};
```

Klasa służąca do obsługi shaderów

## Konstruktory

### Shader()

Standardowy konstruktor. Tworzy pusty shader, który należy ręcznie skompilować. Parametry:

- *path*-zmienna typu *string* zawierająca ścieżkę do pliku z modelem na dysku
- *gamma*-zmienna określająca, czy podczas ładowania trzeba znormalizować współczynnik gamma

### Shader(const char\* vertexPath, const char\* fragmentPath)

Tworzy i kompliuje shader. Parametry:

- *vertexPath*- zmienna określająca ścieżkę do pliku z kodem vertex shadera
- *fragmentPath*- zmienna określająca ścieżkę do pliku z kodem fragment shadera

## Funkcje

### void use()

Funkcja ustawiająca shader jako aktualnie używany.

### Shader &Use()

Funkcja ustawiająca shader jako aktualnie używany i zwracająca go.

**Wartość zwracana:** referencja na aktywny shader

### void checkCompileErrors(GLuint shader, std::string type)

Funkcja wypisująca błędy zgłoszone podczas komplikacji shadera (o ile istnieją). Parametry:

- *shader* - zmienna typu *unsigned int* wskazująca na miejsce shadera w pamięci
- *type* - zmienna określająca typ badanego shadera

```
void setBool(const std::string &name, bool value)
```

Funkcja ustawiająca wartość zmiennej typu *bool* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość
- *value* - wartość przypisywana do zmiennej

```
void setInt(const std::string &name, int value)
```

Funkcja ustawiająca wartość zmiennej typu *int* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość
- *value* - wartość przypisywana do zmiennej

```
void setFloat(const std::string &name, float value)
```

Funkcja ustawiająca wartość zmiennej typu *float* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość
- *value* - wartość przypisywana do zmiennej

```
void setVec2(const std::string &name, const glm::vec2 &value)
```

Funkcja ustawiająca wartość zmiennej wektorowej *x,y* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość
- *value* - wartość przypisywana do zmiennej

```
void setVec3(const std::string &name, const glm::vec3 &value)
```

Funkcja ustawiająca wartość zmiennej wektorowej *x,y,z* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość
- *value* - wartość przypisywana do zmiennej

```
void setVec4(const std::string &name, const glm::vec4 &value)
```

Funkcja ustawiająca wartość zmiennej wektorowej *x,y,z,w* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość
- *value* - wartość przypisywana do zmiennej

```
void setMat2(const std::string &name, const glm::mat2 &mat)
```

Funkcja ustawiająca wartość zmiennej macierzowej *2x2* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość
- *mat* - wartość przypisywana do zmiennej

```
void setMat3(const std::string &name, const glm::mat3 &mat)
```

Funkcja ustawiająca wartość zmiennej macierzowej *3x3* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość
- *mat* - wartość przypisywana do zmiennej

```
void setMat4(const std::string &name, const glm::mat4 &mat)
```

Funkcja ustawiająca wartość zmiennej macierzowej *4x4* w shaderze. Parametry:

- *name* - nazwa zmiennej, której przypisujemy wartość

- *mat*- wartość przypisywana do zmiennej

```
void setVec2(const std::string &name, float x, float y)
void setVec3(const std::string &name, float x, float y, float z)
void setVec4(const std::string &name, float x, float y, float z, float w)
```

Powyższe funkcje przeładowywują funkcje o analogicznej nazwie, przyjmując jako wartość zmiennej nie wektor, a kolejne składowe wektora.

```
void SetInteger(const char *name, int value, bool useShader = false)
void SetMatrix4(const char *name, const glm::mat4 &matrix, bool useShader = false)
void SetVector3f(const char *name, const glm::vec3 &value, bool useShader = false)
```

Powyższe funkcje mają działanie takie same jak analogiczne, wyżej zdefiniowane funkcje. Ponadto, pozwalają na uruchomienia shadera w ich ciele.

## Terrain

### Terrain.h

```
class Terrain
{
public:
    Terrain() = default;
    ~Terrain();
    void loadTerrain(std::string modelPath, std::string heightMapPath, Shader& shader);
    float getHeight(glm::vec3 pos);
    void Draw(glm::mat4& projectionMatrix, glm::mat4& viewMatrix, const LightObject& lightObject, const Camera& camera);

private:
    Model* model = nullptr;
    Entity* entity = nullptr;
    SDL_Surface* heightMap = nullptr;
};
```

Klasa służąca do obsługi terenu.

## Funkcje

```
void loadTerrain(std::string modelPath, std::string heightMapPath, Shader& shader)
```

Funkcja ładowająca teren do pamięci. Parametry:

- *modelPath*- ścieżka do pliku zawierającego model terenu
- *heightMapPath* - ścieżka do pliku zawierającego informacje o wysokości terenu (heightmapa)
- *shader* - shader, który ma zostać użyty do wyświetlenia terenu

```
float getHeight(glm::vec3 pos);
```

Funkcja zwracająca wysokość terenu na danej pozycji. Parametry:

- *pos* - wektor zawierający badaną pozycję

**Wartość zwracana:** wartość typu float, będąca wysokością terenu na badanej pozycji

```
void Draw(glm::mat4& projectionMatrix, glm::mat4& viewMatrix, const LightObject& lightObject, const Camera& camera)
```

Funkcja wyświetlająca teren na ekranie. Parametry:

- *projectionMatrix* - macierz perspektywy
- *viewMatrix* - macierz widoku
- *lightObject* - oświetlenie, któremu podlega teren

- *camera* - kamera, przy użyciu której ma zostać wyświetlony teren

## Collision

```
Collision.h
class Collision
{
protected:
    virtual float TerrainCollide(float WorldX, float WorldZ) = 0;

    float Collide;

};
```

Klasa abstrakcyjna posiadająca metodę wirtualną do określania kolizji terenu

## Funkcje

```
virtual float TerrainCollide (float WorldX, float WorldZ) = 0
```

Parametry:

- *WorldX* - aktualna pozycja na osi x gracza na mapie
- *WorldZ* - aktualna pozycja na osi z gracza na mapie

## DirectionalLight

```
DirLight.h
class DirectionalLight
{
public:
    glm::vec3 vColor;
    glm::vec3 vDirection;

    float fAmbient;

    void SetUniformData(ShaderProgram* spProgram, std::string sLightVarName);

    CDirectionalLight();
    CDirectionalLight(glm::vec3 Color, glm::vec3 Direction, float fAmbient);
};
```

Klasa przekazująca dane o atrybutach światła przekazywanych do shaderów. Ustawia parametry światła kierunkowego.

## Funkcje

```
void SetUniformData (ShaderProgram* spProgram, std::string sLightVarName)
```

Funkcja ładująca dane o parametrach oświetlenia do shaderów. Parametry:

- *spProgram* - wksaźnik na aktualnie wybrany obiekt klasy ShaderProgram o odpowiednim shaderze
- *sLightVarName* - nazwa struktury, która zawarta jest w shaderze. Dostajemy się do składowych struktury

## Konstruktory

```
CDirectionalLight (glm::vec3 Color, glm::vec3 Direction, float fAmbient)
```

Konstruktor przekazujący dokładne wartości do shadera oświetlenia Parametry:

- *Color* - ustawienie koloru oświetlenia w przestrzeni RGB.
- *Direction* - kierunek skąd pada światło względem macierzy przekształceń
- *fAmbient* - wartość rozproszenia przestrzennego światła

```
CDirectionalLight(glm::vec3 Color, glm::vec3 Direction, float fAmbient)
```

Konstruktor przekazujący dokładne wartości do shadera oświetlenia Parametry:

- *Color*- ustawienie koloru oświetlenia w przestrzeni RGB.
- *Direction* -kierunku skąd pada światło względem macierzy przekształceń
- *fAmbient* - wartość rozproszenia przestrzennego światła

```
CDirectionalLight()
```

Konstruktor bezparametrowy. Ustawia domyślne wartości dla koloru,rozproszenia i kierunku światła,kąt stożka, osłabienie światła,

## SpotLight

```
SpotLight.h
class SpotLight
{
public:
    glm::vec3 vColor;
    glm::vec3 vPosition;
    glm::vec3 vDirection;

    int bOn;
    float fConeAngle;
    float fLinearAtt;

    void SetUniformData(ShaderProgram* spProgram, string sLightVarName);

    CSpotLight();
    CSpotLight(glm::vec3 vColor, glm::vec3 vPosition, glm::vec3 vDirection, int bOn, float fConeAngle, float
fLinearAtt);
private:
    float fConeCosine;
};
```

Klasa przekazująca dane o atrybutach światła przekazywanych do shaderów. Ustawia parametry światła kierunkowego.

## Funkcje

```
void SetUniformData(ShaderProgram* spProgram, string sLightVarName)
```

Funkcja ładująca dane o parametrach oświetlenia punktowego do shaderów. Parametry:

- *spProgram* - wskaźnik na aktualnie wybrany obiekt klasy ShaderProgram o odpowiednim shaderze
- *sLightVarName*- nazwa struktury, która zawarta jest w shaderze.Dostajemy się do składowych struktury

## Konstruktory

```
CSpotLight()
```

Konstruktor bezargumentowy. Ustawia domyślne wartości dla parametrów latarki.Są to: kolor, pozycja oraz kierunek światła. Przekazujemy także informacje o tym czy latarka jest włączona oraz kąt padania światła.

```
CSpotLight(glm::vec3 vColor, glm::vec3 vPosition, glm::vec3 vDirection, int bOn,
float fConeAngle, float fLinearAtt);
```

Konstruktor parametryczny. Ustawia wartości dla parametrów latarki. Wymienione powyżej w konstruktorze bezargumentowym.

## Fog

```
Fog.h
class Fog
{
public:
    Fog();
    Fog(glm::vec4 FogColor, float FogDensity, int Equation);
    ~Fog() = default;
    void SetUniformData(ShaderProgram* Program, string FogName);
    void SetUniformData(ShaderProgram* Program, glm::mat4 ModelMatrix, glm::mat4
ProjectionMatrix,glm::vec3 CameraPosition);

private:
    glm::vec4 _FogColor;
    float _FogDensity;
    float _AmbientIntensity;
    int _Equation;
    glm::vec3 _Direction;

};
```

Klasa tworząca mgłę w scenie gry. Otacza mapę terenu stanowiąc również obszar graniczny terenu gry.

## Konstruktory

```
Fog()
```

Konstruktor bezargumentowy. Ustawia dla shaderów wartość domyślną dla koloru, gęstości oraz intensywności otoczenia mgły. Również ustalamy rodzaj interpolacji wykładniczej mgły i jej kierunek.

```
Fog(glm::vec4 FogColor, float FogDensity, int Equation)
```

Konstruktor argumentowy. Za pomocą zmiennych:

- *FogColor* - kolor mgły
- *FogDensity* - gęstość mgły
- *Equation* - rodzaj interpolacji mgły

## Funkcje

```
void SetUniformData(ShaderProgram* Program, string FogName)
```

Ustalanie dla wskaźnika podanego obiektu klasy ShaderProgram ustawia dla parametru FogName wartości shadera mgły. FogName jest nazwą struktury w shaderze i przekazywane są wartości parametrów mgły ustalonych wcześniej w konstruktorze klasy mgła.

```
void SetUniformData(ShaderProgram* Program, glm::mat4 ModelMatrix, glm::mat4
ProjectionMatrix,glm::vec3 CameraPosition)
```

Funkcja dla wskaźnika na obiekt klasy ShaderProgram ustawia :

- ModelMatrix - macierz modelu
- ProjectionMatrix - macierz perspektywy

## Geometry

```
Geometry.h
```

```
extern glm::vec3 vCubeVertices[36];
extern glm::vec2 vCubeTexCoords[6];
```

```
extern glm::vec3 vCubeNormals[6];
extern glm::vec3 vGround[6];
```

Plik zawierający zewnętrzne 2/3-elementowe wektory. Wykorzystywane są do przechowywania informacji o wartościach normalnych dla mgły, konkretnych współrzędnych względem terenu. Mgła ustalana jest na siatce sześciennych kostek.

- *vCubeVertices[36]* - wartość wierzchołków dla siatki mgły,
- *vCubeTexCoords[6]* - współrzędne kostki sześciennej, do ustalenia tekstury mgły
- *vCubeNormals[6]* - wartość normalnych kostki,
- *vGround[6]* - wartość współrzędnych podstawy mgły

## HeightMap

```
HeightMap.h
class HeightMap:protected Collision
{
public:
    static bool LoadTerrainShaderProgram();
    static void ReleaseTerrainShaderProgram();
    glm::mat4 GetScaleMatrix();
    bool LoadMapFromImage(const char *path, const std::string &directory);
    void ReleaseHeightmap();
    virtual float TerrainCollide(float WorldX, float WorldZ);
    virtual
    float BarryCentric(glm::vec3 p1, glm::vec3 p2, glm::vec3 p3, glm::vec2 pos);
    void RenderHeightmap(glm::mat4 projection);

    void SetRenderSize(float fQuadSize, float fHeight);
    void SetRenderSize(float fRenderX, float fHeight, float fRenderZ);
    void SetHeightsData();
    vector<vector<glm::vec3>> GetVertexData();
    void SetVertexData();
    vector<vector<glm::vec2>> GetCoordsData();
    void getHeightOfTerrain(float CameraX, float CameranY, float CameraZ);
    void SetCoordsData();
    int GetNumHeightmapRows();
    int GetNumHeightmapCols();
    int counter = 0;
    static ShaderProgram* GetShaderProgram();

    HeightMap();

private:
    unsigned int uiVAO;

    bool bLoaded;
    bool bShaderProgramLoaded;
    int Rows;
    int Columns;
    int width, height;
    glm::vec3 vRenderScale;
    vector<vector<float>> Heihgts;
    vector<vector<glm::vec3>> DataVer;
    vector<vector<glm::vec2>> DataCoordinate;
    HeightMapBuffer vboHeightmapData;
    HeightMapBuffer vboHeightmapIndices;
    HeightMap *Heightmap;
    glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
    static ShaderProgram spTerrain;
    static CShader shTerrainShaders[NUMTERRAINSHADERS];
};
```

```
};
```

Klasa tworząca teren wykorzystując mapę wysokości. Stosowana jest tutaj technika wieloteksturowania. Dziedziczona jest również klasa Collision w celu implementacji kolizji dla terenu.

### Funkcje

```
static bool LoadTerrainShaderProgram()
```

Funkcja ładująca odpowiednie shadery dla mapy wysokości wykorzystywana do wieloteksturowania terenu.

**Wartość zwracana:** bool sprawdzający czy poprawnie załadowano shadery.

```
static void ReleaseTerrainShaderProgram()
```

Funkcja zwalniająca shadery w momencie kiedy usuwamy teren z gry. Usuwa niepotrzebne shadery terenu.

```
glm::mat4 GetScaleMatrix()
```

Getter na skalowanie terenu mapy. Domyślna skala to 1.0.

```
bool LoadMapFromImage(const char *path, const std::string &directory)
```

Główna funkcja klasy ładująca z pliku teren mapy. Ustawia ona współrzędne terenu oraz jego wierzchołków i normalnych w zależności od zadanego pliku z wysokościami. Do załadowania pliku i odczytania wartości o danych wykorzystujemy SDL\_Image. Parametry:

- *path* - ścieżka do pliku
- *directory* - nazwa folderu zawierającego dany plik

**Wartość zwracana:** bool sprawdzający poprawność załadowania obrazka

```
void ReleaseHeightmap()
```

Zwala zasoby związane z utworzeniem i renderowaniem terenu pod koniec gry.

```
virtual float TerrainCollide(float WorldX, float WorldZ)
```

Wirtualna funkcja implementująca kolizje terenu z graczem w zależności od parametrów:

- *WorldX* - pozycja na osi X gracza względem terenu
- *WorldZ* - pozycja na osi Z gracza względem terenu

**Wartość zwracana:** float będący wysokością terenu względem mapy

```
virtual float TerrainCollide(float WorldX, float WorldZ)
```

Barycentryczna funkcja wykorzystywana do aproksymowania pozycji gracza w określonym obszarze mapy. Tworzy na podstawie 3 punktów prawdopodobną pozycję gracza. Z podanych 3 punktów możemy określić położenie na mapie. Domyślnie ustala wysokość gracza. Parametry:

- *glm::vec3 p1* - skrajny lewy punkt
- *glm::vec3 p2* - skrajny prawy punkt
- *glm::vec3 p3* - dolny punkt
- *glm::vec2 pos* - współrzędna punktu w 2D dla przypuszczalnej pozycji wstępnej

**Wartość zwracana:** Wartość zwracana: float będący wysokością gracza względem mapy

```
void RenderHeightmap(glm::mat4 projection)
```

Funkcja renderująca teren gry wykorzystując przekazane w buforze współrzędne o rozlokowaniu zbioru trójkątów na mapie i ich sklejenie w jeden teren. Parametr:

- *projection* - ustala nam wstępna skalę mapy przekazywaną do wewnętrznej funkcji

```
void SetRenderSize(float fQuadSize, float fHeight)
```

Funkcja ustalająca rozmiar renderowania dla wektora vRenderScale. Pozwala na zachowanie proporcji mapy. Uwzględnia ilość kolumn i wierszy mapy określonych na wcześniejszym etapie tworzenia mapy. Parametry:

- *fQuadSize* - rozmiar pojedynczego trójkąta tworzącego teren
- *fHeight* - skala wysokości mapy

```
void SetRenderSize(float fRenderX, float fHeight, float fRenderZ)
```

Modyfikacja funkcji powyżej pozwalająca na ustalenie rozmiaru renderowania dla wektora vRenderScale nie uwzględniająca ilości wierszy i kolumn. Domyślne parametry to (1.0,1.0,1.0) Parametry:

- *fRenderX* - ustalanie skali mapy dla współrzędnej x
- *fHeight* - ustalenie skali mapy dla współrzędnej y
- *fRenderZ* -ustalenie skali mapy dla współrzędnej z

```
void SetHeightsData()
```

Funkcja ustawiająca rozmiar wektora vectorów floatów wysokości Heights w zależności od ilości wierszy i kolumn.

```
vector< vector< glm::vec3> > GetVertexData()
```

Getter vectora vectorów wierzchołków danych

```
void SetVertexData()
```

Funkcja ustawiająca rozmiar wektora vectorów floatów wierzchołków w zależności od ilości wierszy i kolumn.

```
vector< vector< glm::vec2> > GetCoordsData()
```

Getter vectora vectorów współrzędnych danych mapy.

```
void getHeightOfTerrain(float CameraX, float CameraY, float CameraZ)
```

Funkcja pobierająca wysokość terenu. Parametry:

- *CameraX* - współrzędna x dla kamery gracza
- *CameraY* - współrzędna y dla kamery gracza
- *CameraZ* - współrzędna z dla kamery gracza

```
void SetCoordsData()
```

Funkcja ustawiająca rozmiar wektora wektora 2-elementowych wektorów Współrzędnych danych dla tekstur terenu.

```
int GetNumHeightmapRows ()  
Getter zwracający ilość wierszy.
```

```
static ShaderProgram* GetShaderProgram ()  
Getter na shader programu Terrain.
```

```
int GetNumHeightmapCols ()  
Getter zwracający ilość kolumn.
```

## HUD

```
HUD.h  
class HUD  
{  
public:  
  
    struct Calculate {  
        void operator() ( Shader Program) {  
            Program.use();  
            Program.setInt("texture2", 1);  
        }  
    }StructShader;  
  
    HUD();  
    HUD(vector<reference_wrapper<const Shader>>LifeShader,const string &heart,const string &container,float Rotation,int Lifeamount);  
    ~HUD();  
    void InitBuffer();  
    unsigned int LoadTexture(const std::string & path, const std::string & directory, bool gamma);  
    void RotationIntensity();  
    void InitHUD(vector<reference_wrapper<const Shader>>_LifeShader);  
    void Incrementvalue();  
    void RenderHUD(vector<reference_wrapper<const Shader>>_LifeShader);  
    void SetIndices();  
    void SetVertices();  
    void SetData();  
private:  
    int _Lifeamount;  
    static int lifenumber;  
    vector<float> vertices;  
    vector <int> indices;  
    vector<int> HealthID;  
    vector<reference_wrapper<const Shader>>_LifeShader;  
    vector<reference_wrapper<const Shader>>::iterator Lifeliterator;  
    static float _incrementvalue;  
    unsigned int texture1, texture2, texture3, texture4;  
    unsigned int VBO, VAO, EBO;  
    static int HP; // time before image disappear  
    float _gamma;  
    bool loaded;  
};
```

Klasa implementująca 3 życia w dolnym lewym rogu.

## Konstruktory

```
HUD ()  
Konstruktor bezparametryowy. Ustawia informacje o załadowaniu wartości.
```

```
HUD(vector<reference_wrapper<const Shader>>LifeShader, const string &heart, const string &container, float Rotation, int Lifeamount)
```

Konstruktor parametryowy. Przeprowadza inicializacje HUDu gry. Parametry:

- *LifeShader* - vector shaderów
- *heart* - referencje na obrazek życia
- *container* - referencja na obrazek z kontenerem za życiem
- *Rotation* - prędkość rotacji obrazka
- *Lifeamount* - ilość życia w obiekcie

## Funkcje

```
void InitBuffer()
```

Funkcja rezerwująca miejsce w OpenGL na podane obrazki.

```
unsigned int LoadTexture(const std::string & path, const std::string & directory, bool gamma)
```

Funkcja ładowająca tekstury z pliku. Parametry:

- *path* - ścieżka do pliku
- *directory* - folder zawierający plik
- *gamma* - czy ustawiono gamme

```
void RotationIntensity()
```

Funkcja ustawiająca rotacje obrazków.

```
void InitHUD(vector<reference wrapper<const Shader>> LifeShader);
```

Funkcja przekazująca zawartość vectora Shaderów, do vectora odpowiedzialnego za tworzenie shadera HUDu. Parametry:

- *\_LifeShader* - vector Shaderow

```
void Incrementvalue()
```

Funkcja zwiększająca zawartość informacyjną o ilości życia w grze. Niezbędna do określenia pozycji życia.

```
void RenderHUD(vector<reference wrapper<const Shader>> LifeShader)
```

Funkcja renderująca HUD wykorzystującą OpenGL. Parametry:

- *LifeShader* - vector Shaderow

```
void SetIndices()
```

Funkcja ustawiająca indeksy obrazów 2D na ekranie.

```
void SetVertices()
```

Funkcja ustawiająca pozycje wierzchołków w obrazkach 2D na ekranie.

```
void SetData()
```

Funkcja ustawiająca współrzędne, kolory obrazów 2D.

## Skybox

```
Skybox.h
class CSkybox
{
public:
    CSkybox();
    CSkybox(Shader&Program, string Right, string Left, string Top, string Bottom, string Front, string Back);
    void LoadSkyBoxVector(vector<string> vec);
    void LoadCubeMap(vector<string> faces);
    void BindBuffer(Shader&Program);
    void RenderSkybox(Shader&Program, glm::mat4 ViewMatrix, glm::mat4 ProjectionMatrix);
    void SetDeltatime(float deltatime);
    double DegreetoRadians(double degree);

private:
    vector<string> ReserveVector(int size);
    bool Skyloaded;
    bool LoadCube;
    unsigned int skyboxVAO, skyboxVBO;
    vector<string> faces;
    array<float, 109> SkyBoxVertices;
    unsigned int skyboxID;
    float r, g, b;
    static float roatationspeed;
    float _deltatime;
    double pi = 3.14;

};

};
```

Klasa odpowiedzialna za tworzenie CubeMapy oraz jej rotację w trakcie trwania gry.

## Konstruktory

```
CSkybox()
```

Konstruktor bezargumentowy. Rezerwuje w wektorze 6 miejsce na cubemapie.

```
CSkybox(Shader&Program, string Right, string Left, string Top, string Bottom, string Front, string Back)
```

Konstruktor wieloargumentowy. Przypisujący konkretne nazwy obrazka dla wektora przytrzymującego te informacje. Parametry:

- *Right* - prawa część na siatce sześcianu(+X)
- *Left* - lewa część na siatce sześcianu(-X)
- *Top* - górna część na siatce sześcianu (+Y)
- *Bottom* - dolna część na siatce sześcianu(-Y)
- *Front* - przednia część na siatce sześcianu(+Z)
- *Back* - tylna część na siatce sześcianu (-Z)

## Funkcje

```
void LoadSkyBoxVector(vector<string> vec)
```

Funkcja kopiuje zawartość wektora stringów do vectora odpowiedzialnego za ładowanie skyboxa. Parametr:

- *vec* - wektor przechowujący informacje o kolejności ładowania teksturow w cubemapie

```
void BindBuffer(Shader&Program)
```

Funkcja przypisująca shader Skyboxa do programu. W ten sposób informujemy OpenGL o wykorzystywaniu danego shadera. Parametry:

- Program - shader skyboxa

```
void RenderSkybox(Shader& program, glm::mat4 ViewMatrix, glm::mat4 ProjectionMatrix);
```

Funkcja odpowiedzialna za renderowanie Skyboxa w scenie gry. Parametry:

- *Program* - shader skyboxa
- *ViewMatrix* - macierz widoku
- *ProjectionMatrix* - macierz perspektywy

```
void SetDeltatime(float deltatime)
```

Funkcja ustawiająca czas detla, czyli różnicę czasu pomiędzy klatką poprzednią i aktualną gry do zaimplementowania animacji poruszania się skyboxa. Parametry:

- *deltatime* - różnica czasu pomiędzy 2 klatkami czasu gry

```
double DegreetoRadians(double degree)
```

Funkcja przekształcająca stopnia na radiany. Wymagana do shaderów, które nie przyjmują stopni tylko radiany.

**Wartość zwracana:** double, będący wartością radianów.

```
vector<string> ReserveVector(int size)
```

Funkcja rezerwująca wektor o podanym rozmiarze. Domyślnie 6, odpowiada ona ilości elementów z których składa się siatka figury.

**Wartość zwracana:** vector<string> stanowiący ustwiony wektor w funkcji

## Sound

```
Sound.h
```

```
class Sound
{
private:
    static Sound* sInstance;

    std::map<std::string, Mix_Music*> Music;
    std::map<std::string, Mix_Chunk*> SFX;

    Sound();
    ~Sound();

public:
    static Sound* Instance();
    static void Release();

    Mix_Music* GetMusic(std::string filename);
    Mix_Chunk* GetSFX(std::string filename);

};
```

Singleton tworzy instancję klasy Sound z możliwością uzyskania dostępu z muzyką albo SFX. Ścisłe powiązany z SoundManager.

### Konstruktory

`Sound()`

Konstruktor bezparametrowy.

### Funkcje

`static Sound* Instance()`

Tworzenie instancji Sound.

`static void Release()`

Usuwanie obiektu Sound.

`Mix Music* GetMusic(std::string filename)`

Pobranie muzyki getterem. Parametry:

- `filename` - nazwa pliku

Wartość zwracana: wskaźnik na obiekt klasy Mix\_Music.

`Mix Chunk* GetSFX(std::string filename)`

Pobranie SFX getterem. Parametry:

- `filename` - nazwa pliku

Wartość zwracana: wskaźnik na obiekt klasy Mix\_Chunk.

## SoundManager

`SoundManager.h`

`class SoundManager // Singleton`

`{`

`private:`

`static SoundManager* sInstance;`

`Sound* SoundMgr;`

`public:`

`static SoundManager* Instance();  
static void Release();`

`void PlayMusic(std::string filename, int loop = -1);  
void HaltMusic();  
void ResumeMusic();  
void VolumeMusic(int volume = 64);`

`void PlaySFX(std::string filename, int loop = 0, int channel = 0);  
void VolumeSFX(int channel, int volume = 64);  
void HaltSFX(int channel);`

```
private:
```

```
    SoundManager();
    ~SoundManager();
};
```

Singleton tworzy instancję klasy SoundManager zarządza muzyką oraz SFX.

## Konstruktory

```
SoundManager()
```

Konstruktor bezparametrowy. Ustawia kanał przesyłanego dźwięku oraz próbkowanie dla każdego obiektu tej klasy.

## Funkcje

```
void PlayMusic(std::string filename, int loop = -1);
```

Funkcja aktywująca muzykę. Parametry:

- *filename* - nazwa pliku
- *loop* - odtwarzanie w pętli. Domyślnie -1 odtwarza w pętli.

```
void HaltMusic()
```

Zatrzymuje muzykę dla danego obiektu. Jeżeli usuniemy dany obiekt to zostaje usunięta z mapy odtwarzania.

```
void ResumeMusic()
```

Odtwarza zatrzymany wcześniej fragment muzyczny.

```
void VolumeMusic(int volume = 64)
```

Funkcja ustawia głośność muzyki. Domyślnie 64. Parametry:

- *volume* - głośność muzyki

```
void PlaySFX(std::string filename, int loop = 0, int channel = 0)
```

Funkcja aktywująca SFX dla danego obiektu klasy SoundManager. Parametry:

- *filename* - nazwa pliku

```
void VolumeSFX(int channel, int volume = 64)
```

Funkcja ustawia głośność SFX. Domyślnie 64. Należy podać kanał muzyczny. Parametry:

- *volume* - głośność SFX
- *channel* - wybrany kanał muzyczny

```
void HaltSFX(int channel)
```

Zatrzymuje SFX dla danego obiektu. Jeżeli usuniemy dany obiekt to zostaje usunięta z mapy odtwarzania. Parametry:

- *channel* - kanał muzyczny

```
static void Release()
```

Usuwanie instancji klasy SoundManager.

```
static SoundManager* Instance()
```

Tworzenie instancji klasy SoundManager.

**Wartość zwracana:** wskaźnik na instancje klasy SoundManager.

## SpriteRenderer

```
SpriteRenderer.h
```

```
{  
public:  
    SpriteRenderer( Shader &shaders);  
    ~SpriteRenderer();  
    void DrawSprite(SpriteTexture &texture, glm::vec2 position, glm::vec2 size = glm::vec2(10, 10), GLfloat rotate = 0.0f, glm::vec3 color = glm::vec3(1.0f));  
private:  
    Shader shader;  
    GLuint quadVAO;  
    void initRenderData();  
};
```

Klasa SpriteRenderer zarządza tworzeniem spritów.

## Konstruktory

```
SpriteRenderer( Shader &shaders)
```

Konstruktor jednoargumentowy. Przypisuje shadera do programu oraz tworzy bufor danych dla OpenGL.

Parametr:

- *shaders* - obiekt klasy Shader, przekazuje referencje na aktualny shader

## Funkcje

```
void DrawSprite(SpriteTexture &texture, glm::vec2 position, glm::vec2 size = glm::vec2(10, 10), GLfloat rotate = 0.0f, glm::vec3 color = glm::vec3(1.0f))
```

Funkcja rysująca spritz dla danych parametrów wejściowych:

- *texture* - obiekt przechowujący sprity klasy SpriteTexture
- *position* - pozycja sprita 2D
- *size* - rozmiar sprita
- *rotate* - rotacja
- *color* - kolor sprita

## SpriteTexture

```
SpriteTexture.h
```

```
class SpriteTexture  
{  
public:  
    unsigned int ID;  
    unsigned int Width, Height;  
    unsigned int InternalFormat;  
    unsigned int ImageFormat;  
  
    unsigned int WrapS;  
    unsigned int WrapT;  
    unsigned int FilterMin;  
    unsigned int FilterMax;
```

```

SpriteTexture();
~SpriteTexture();

void Generate(unsigned int width, unsigned int height, SDL_Surface*image);

void Bind() const;
}

```

Singleton tworzy instancję klasy SoundManager zarządza muzyką oraz SFX.

### Konstruktory

`SpriteTexture ()`

Konstruktor bezargumentowy. Tworzy bufor danych dla OpenGL.

### Funkcje

`void Generate(unsigned int width, unsigned int height, SDL_Surface*image)`

Funkcja generuje teksturę uwzględniając podane parametry oraz ustawia filtrowanie dla spritów. Parametry:

- *width* - szerokość sprita
- *height* - wysokość sprita
- *image* - wskaźnik na obraz otwarty w `SDL_Image`

`void Bind() const`

Funkcja przypisująca dany ID tekstury do odpowiedniej funkcji w OpenGL.

## TextManager

```

TextManager.h
class TextManager
{
public:
    TextManager(GLuint width, GLuint height);
    void RenderText(string text, GLfloat x, GLfloat y, GLfloat scale, glm::vec3 color = glm::vec3(1.0f));
    void Load(string font, unsigned int fontSize);
    std::map<GLchar, Character> Characters;
    Shader TextShader;
private:
    unsigned int VAO, VBO;

};

struct Character {
    GLuint TextureID;
    glm::ivec2 Size;
    glm::ivec2 Bearing;
    int Advance;
};

```

Klasa odpowiedzialna za ładowanie fontów oraz renderowanie glifów tekstu. Wykorzystywana jest także struktura `Character`, która ułatwia zapisywanie informacji o słowach tekstu.

### Konstruktory

`TextManager(GLuint width, GLuint height)`

Konstruktor 2-argumentowy. Odpowiedzialny za tworzenie tekstu dla żądanej rozdzielczości ekranu. Parametry:

- *width* - szerokość ekranu
- *height* - wysokość ekranu

## Funkcje

```
void RenderText(string text, GLfloat x, GLfloat y, GLfloat scale, glm::vec3 color = glm::vec3(1.0f))
```

Funkcja odpowiedzialna za renderowanie napisów w OpenGL. Parametry:

- *text* - treść jaka ma zostać wypisana
- *x* - pozycja tekstu dla współrzędnej x
- *y* - pozycja tekstu dla współrzędnej y
- *scale* - skala tekstu
- *color* - kolor tekstu

```
void Load(string font, unsigned int fontSize)
```

Funkcja ładująca zadany font, który będzie wykorzystywany do pisania tekstu. Parametry:

- *font* - nazwa fontu
- *fontSize* - rozmiar fontu

## TextureClass

```
TextureClass.h
class CTexture
{
public:
    void CreateEmptyTexture(int Width, int Height, GLenum format);
    void CreateFromData(BYTE* bData, int Width, int Height, int BPP, GLenum format, bool bGenerateMipMaps = false);

    bool LoadTexture2D(string path, const string &directory, bool bGenerateMipMaps);
    void BindTexture(int iTextureUnit = 0);

    void SetFiltering(int fMagnification, int fMinification);

    void SetSamplerParameter(GLenum parameter, GLenum value);

    int GetMinificationFilter();
    int GetMagnificationFilter();

    int GetWidth();
    int GetHeight();
    int GetBPP();

    UINT GetTextureID();

    string GetPath();

    void DeleteTexture();

    CTexture();
private:
    int iWidth, iHeight, iBPP; // Texture width, height, and bytes per pixel
    UINT uiTexture; // Texture name
    UINT uiSampler; // Sampler name
    bool bMipMapsGenerated;

    int tfMinification, tfMagnification;
    string sPath;
};
```

```

enum ETextureFiltering
{
    TEXTURE_FILTER_MAG_NEAREST = 0, // Nearest criterion for magnification
    TEXTURE_FILTER_MAG_BILINEAR, // Bilinear criterion for magnification
    TEXTURE_FILTER_MIN_NEAREST, // Nearest criterion for minification
    TEXTURE_FILTER_MIN_BILINEAR, // Bilinear criterion for minification
    TEXTURE_FILTER_MIN_NEAREST_MIPMAP, // Nearest criterion for minification, but on closest mipmap
    TEXTURE_FILTER_MIN_BILINEAR_MIPMAP, // Bilinear criterion for minification, but on closest mipmap
}

```

Klasa odpowiedzialna za ustawianie tekstur oraz ich renderowanie . Przeznaczona tylko dla terenu z heightmapą.

## **Konstruktory**

`CTexture ()`

Konstruktor bezargumentowy. Ustawia flagę załadowania tekstur.

## **Funkcje**

`void CreateEmptyTexture (int Width, int Height, GLenum format)`

Tworzenie pustego bufora na załadowanie tekstury.Parametry:

- *Width* - szerokość
- *Height* - wysokość
- *format* - model przestrzeni barw

`void CreateFromData (BYTE* bData, int Width, int Height, int iBPP, GLenum format, bool bGenerateMipMaps = false)`

Tworzenie pustego bufora na załadowanie tekstury w przypadku gdyby nie był wcześniej inicjalizowany.Parametry:

- *bData* - wskaźnik na dane o teksturowe
- *Width* - szerokość
- *Height* - wysokość
- *iBPP* - ilość bajtów na piksel w obrazie
- *format* - model przestrzeni barw

`bool LoadTexture2D (string path, const string &directory, bool bGenerateMipMaps)`

Funkcja ładująca plik jako teksturę wyłącznie dla terenu oraz sprawdza czy została załadowana poprawnie.Parametry:

- *path* - ścieżka do pliku
- *directory* - nazwa folderu zawierającego plik
- *bGenerateMipMaps* - tworzenie mipmap

**Wartość zwracana:** bool, sprawdzający załadowanie tekstury.

`void SetFiltering (int tfMagnification, int tfMinification)`

Funkcja ustawiająca filtrowanie dla zadanej ładowanej tekstury.Parametry:

- *tfMagnification* - parametr powiększenia dla tekstury
- *tfMinification* - parametr pomniejszenia dla tekstury

`void SetSamplerParameter (GLenum parameter, GLenum value)`

Funkcja ustawiająca sampler.Parametry:

- *parameter* - zadana zmienna samplera
- *value* - zadana wartość samplera

```
int GetMinificationFilter()
```

Getter na parametr pomniejszenia tekstury.

```
int GetMagnificationFilter()
```

Getter na parametr powiększenia tekstury.

```
int GetWidth()
```

Getter na pobranie szerokości tekstury.

```
int GetHeight()
```

Getter na pobranie wysokości tekstury.

```
int GetBPP()
```

Getter na pobranie ilości bajtów na piksel obrazu.

```
UINT GetTextureID()
```

Getter na ID tekstury.

```
string GetPath()
```

Getter na ścieżkę tekstury.

```
void DeleteTexture()
```

Funkcja usuwająca tekstury z terenu.

```
extern void LoadAllTextures()
```

Funkcja ładująca wszystkie tekstury zadane przez użytkownika.

## HeightMapBuffer

```
vertexBufferObject.h
class HeightMapBuffer
{
public:
    void CreateVBO(int Size = 0);
    void DeleteVBO();

    void* MapBufferToMemory(int iUsageHint);
    void UnmapBuffer();

    void BindVBO(int iBufferType = GL_ARRAY_BUFFER);
    void UploadDataToGPU(int iUsageHint);
```

```

void* GetDataPointer();
unsigned GetBufferID();

HeightMapBuffer();

private:
    unsigned uiBuffer;
    int iSize;
    int iCurrentSize;
    int iBufferType;
    std::vector<unsigned char> data;

    bool bDataUploaded;
};

Klasa służąca jako bufor dla klasy HeightMap.

```

### Konstruktory

`HeightMapBuffer ()`

Konstruktor bezparametrowy. Tworzy flagę na ładowanie tekstury.

### Funkcje

`void CreateVBO (int Size = 0)`

Funkcja tworząca bufor geometrii VBO. Parametry:

- *Size* - rozmiar bufora

`void DeleteVBO ()`

Funkcja usuwająca bufor geometrii VBO.

`void* MapBufferToMemory (int iUsageHint)`

Funkcja tworząca bufor map. Przekazane parametry:

- *iUsageHint* - ilość elementów w buforze

`void UnmapBuffer ()`

Funkcja usuwająca bufor map.

`void BindVBO (int iBufferType = GL_ARRAY_BUFFER)`

Funkcja przypisująca bufor geometrii. Parametry:

- *iBufferType* - typ bufora, domyślnie GL\_ARRAY\_BUFFER

`void UploadDataToGPU (int iUsageHint)`

.Funkcja przekazująca dane do procesora graficznego. Parametry:

- *iUsageHint* - ilość przekazanych danych

`void* GetDataPointer ()`

Funkcja pobierająca załadowane dane z tablicy dane.

```
unsigned GetBufferID ()
```

Getter na ID bufora.

## ResourceManager

```
ResourceManager.h
class ResourceManager
{
public:
    static map<string, Shader > Shaders;
    static map<string, SpriteTexture> Textures;
    static Shader LoadShader(const char *vShaderFile, const char*fShaderFile, string name);
    static Shader GetShader(string name);
    static SpriteTexture LoadTexture(const char *file, bool alpha, string name);
    static SpriteTexture GetTexture(string name);
    static void Clear();
private:
    ResourceManager();
    ~ResourceManager();
    static Shader loadShaderFromFile(const char *vShaderFile, const char *fShaderFile);
    static SpriteTexture loadTextureFromFile(const char *file, bool alpha);
};
```

Klasa odpowiedzialna za zarządzanie tworzeniem obrazów 2D dla menu. Korzysta z klas SpriteTexture,SpriteRenderer.

## Konstruktory

```
ResourceManager ()
```

Konstruktor bezargumentowy.

## Funkcje

```
static Shader LoadShader(const char *vShaderFile, const char*fShaderFile, string name)
```

Funkcja statyczna ładująca shader z obiektu klasy Shader. Parametry:

- *vShaderFile* - nazwa plik vertex shader
- *fShaderFile* - nazwa pliku fragment shader

**Wartość zwracana:** obiekt klasy Shader

```
static Shader GetShader(string name)
```

Getter obiektu typu Shader. Parametry:

- *name* - nazwa shadera

```
static SpriteTexture LoadTexture(const char *file, bool alpha, string name)
```

Funkcja ładująca teksturę z klasy SpriteTexture. Parametry:

- *file* - nazwa pliku, który wykorzystuje shader
- *alpha* - ustawić przezroczystość
- *name* - nazwa , pod jaką chcemy zapisać teksturę

**Wartość zwracana:** obiekt klasy SpriteTexture

```
static SpriteTexture GetTexture(string name)
```

Getter obiektu klasy SpriteTexture. Parametry:

- *name* - nazwa pliku

```
static void Clear()
```

Funkcja usuwająca nieużywane obiekty klasy Shader, SpriteTexture.

```
static Shader loadShaderFromFile(const char *vShaderFile, const char *fShaderFile)
```

Funkcja ładująca z pliku shader klasy Shader. Parametry:

- *vShaderFile* - nazwa pliku vertex shader
- *fShaderFile* - nazwa pliku fragment shader

**Wartość zwracana:** obiekt klasy Shader

```
static SpriteTexture loadTextureFromFile(const char *file, bool alpha)
```

Funkcja ładująca teksturę z obiektu klasy SpriteTexture z pliku . Parametry:

- *file* - nazwa pliku
- *alpha* - ustawienie przezroczystości

## CShader

```
shaders.h
class CShader
{
public:
    bool LoadShader(string sFile, int a_iType);
    void DeleteShader();

    bool GetLinesFromFile(string sFile, bool bIncludePart, vector<string>* vResult);

    bool IsLoaded();
    unsigned int GetShaderID();

    CShader();

private:
    unsigned uiShader; // ID of shader
    int iType; // GL_VERTEX_SHADER, GL_FRAGMENT_SHADER...
    bool bLoaded; // Whether shader was loaded and compiled
};
```

Klasa odpowiedzialna za ładowanie odpowiedniej, zadanej shadera dla heightmapy.

### Konstruktory

```
CShader()
```

Konstruktor bezargumentowy. Ustawienie flagi załadowania shadera.

### Funkcje

```
bool LoadShader(string sFile, int Type)
```

Funkcja statyczna ładująca shader z obiektu klasy Shader. Parametry:

- *sFile*- nazwa plik
- *Type*- typ pliku vertex lub fragment

**Wartość zwracana:** bool, sprawdzenie poprawności załadowania shadera.

```
void DeleteShader()
```

Funkcja usuwająca zbędny shader.

```
bool GetLinesFromFile(string sFile, bool bIncludePart, vector<string>* vResult)
```

Funkcja wczytująca linie tekstu z shaderów. Następnie są one przekazywane do programu. Parametry:

- *sFile* - nazwa pliku
- *bIncludePart* - sprawdzenie czy w pliku shader są #include
- *vResult* - przekazuję wczytanie linie tekstu do vectora

**Wartość zwracana:** bool, sprawdzenie poprawności odczytu linii z shadera.

```
bool IsLoaded()
```

Funkcja sprawdzająca czy załadowano poprawnie shader.

**Wartość zwracana:** bool, sprawdzenie poprawności załadowania shadera.

## Wydruk najważniejszych funkcji

### Główna pętla programu

tutorial.cpp

```
int main(int argc, char *argv[])
{
    if (initSDL() < 0) return -1;
    SDL_Window* window;
    SDL_GLContext context;
    initOpenGL(window, context);

    Loader::loadModels();
    Loader::loadShaders();
    Loader::loadTextures2D();

    SceneManager::GetInstance().SelectScene(SceneManager::Scenes::SCENE_MENU);
    SDL_Event e;
    while(SceneManager::GetInstance().currentScene != SceneManager::SCENE_QUIT)
    {
        SceneManager::GetInstance().getScene()->handleEvents(e);
        if(SceneManager::GetInstance().getScene())
            SceneManager::GetInstance().getScene()->render();

        SDL_GL_SwapWindow(window); // zostanie w glownej petli aplikacji, nie ma sensu sie powtarzac
    }
    SoundManager::Release();
    Loader::unloadShaders();
    Loader::unloadModels();
    Loader::unloadTextures2D();
    SDL_GL_DeleteContext(context);
    SDL_DestroyWindow(window);
    Mix_Quit();
    SDLNet_Quit();
    IMG_Quit();
    SDL_Quit();
    return 0;
}
```

## Scena rozgrywki

### Inicjalizacja sceny

SceneMultiplayerGame.cpp

```
void SceneMultiplayerGame::InitScene()
{
    run = true;
    scoreBoard = false;
    imgPtr = new Image2D(-0.5f, 0.5f, -0.5f, 0.5f);
    health = 3;
    ostatniWystrzał = 0.0f;
    lastFrame = SDL_GetTicks() / 1000.0f;
    Terrain* terrain = new Terrain();
    terrain->loadTerrain("res/models/teren_org/teren.obj", "res/models/teren_org/height.png",
    Loader::getShader(Loader::LoadedShaders::LIGHT));
    addTerrain(terrain);
    SetLight(new LightObject(Loader::getModel(Loader::LoadedModels::CUBE), glm::vec3(0.0f, 10.0f, 0.0f), 0.0f,
    glm::vec3(0.2f), &Loader::getShader(Loader::LoadedShaders::SIMPLE), glm::vec3(1.0f, 1.0f, 1.0f)));
    player2 = addObject(new Entity(Loader::getModel(Loader::LoadedModels::PLAYER), glm::vec3(20.0f, 0.0f, 7.0f),
    0.0f, glm::vec3(1.0f)));
    player2->setShader(Loader::getShader(Loader::LoadedShaders::LIGHT));
    Networking::setDataSize(sizeof(pos));
    SDL_SetRelativeMouseMode(SDL_TRUE);
    SoundMgr = Sound::Instance();
    SoundM = SoundManager::Instance();
    SoundM->PlayMusic("DeathMatch_Boss_Theme.ogg", -1);
    SoundM->VolumeMusic(15);
    MainSkybox = CSkybox(Loader::getShader(Loader::LoadedShaders::SKYBOX), "res/img/right.jpg",
    "res/img/left.jpg", "res/img/top.jpg", "res/img/bottom.jpg", "res/img/front.jpg", "res/img/back.jpg");
    SH.push_back(Loader::getShader(Loader::LoadedShaders::SHADER2D));
    SH.push_back(Loader::getShader(Loader::LoadedShaders::SHADER2D));
    life1 = new HUD(SH, "heart.png", "sand.jpg", 0.01f, 1);
    life2 = new HUD(SH, "heart.png", "sand.jpg", 0.01f, 1);
    life3 = new HUD(SH, "heart.png", "sand.jpg", 0.01f, 1);
    Bullet::setShader(&Loader::getShader(Loader::LoadedShaders::SIMPLE));
    if (isServer) {
        connectionThread = new std::thread([=] {serverConnectionHandlerThread(); });
        getCamera()->cameraPos = glm::vec3(25.0f, 0.85f, 5.0f);
    }
    else {
        connectionThread = new std::thread([=] {connectionHandlerThread(); });
        getCamera()->cameraPos = glm::vec3(25.0f, 0.85f, 45.0f);
    }
}
```

### Wyświetlanie sceny

SceneMultiplayerGame.cpp

```
void SceneMultiplayerGame::render()
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    DrawObjects();
    //rysowanie skyboxa
    MainSkybox.SetDeltatime(currentFrame);
    MainSkybox.RenderSkybox(Loader::getShader(Loader::LoadedShaders::SKYBOX), this->GetViewMatrix(),
    this->GetProjectionMatrix());
```

```

//rysowanie pociskow
bulletLock.lock();
for (Bullet* bullet : bulletContainer) {
    bullet->draw(GetProjectionMatrix(), getCamera()->getViewMatrix(), glm::vec3(1.0f, 0.0f, 0.0f));
}
bulletLock.unlock();
//rysowanie broni
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, getCamera()->cameraPos + getCamera()->cameraFront);
model = glm::rotate(model, glm::radians(180.0f - getCamera()->yaw), glm::vec3(0.0f, 1.0f, 0.0f));
model = glm::rotate(model, glm::radians(-getCamera()->pitch), glm::vec3(0.0f, 0.0f, 1.0f));
model = glm::translate(model, glm::vec3(0.35f, -0.15f, 0.0f));
if (ostatniWystrzal != 0.0) {
    if (currentFrame >= ostatniWystrzal) {
        ostatniWystrzal = 0.0;
    }
    else {
        model = glm::rotate(model, -glm::sin((ostatniWystrzal - currentFrame) / 0.5f) / 3.0f,
        glm::vec3(0.0f, 0.0f, 1.0f));
    }
}
model = glm::scale(model, glm::vec3(0.03f, 0.03f, 0.03f));
Shader& lightShader = Loader::getShader(Loader::LoadedShaders::LIGHT);
lightShader.use();
lightShader.use();
lightShader.setMat4("model", model);
lightShader.setMat4("projection", GetProjectionMatrix());
lightShader.setMat4("view", GetViewMatrix());
lightShader.setVec3("lightColor", GetLight()->GetColor());
lightShader.setVec3("lightPos", GetLight()->GetPosition());
lightShader.setVec3("viewPos", getCamera()->cameraPos);
glClear(GL_DEPTH_BUFFER_BIT);
Loader::getModel(Loader::LoadedModels::GUN).Draw(lightShader);
if (health >= 1) {
    life1->RenderHUD(SH);
    life1->Incrementvalue();
    life1->RotationIntensity();
    if (health >= 2) {
        life2->RenderHUD(SH);
        if(health == 3)
            life3->RenderHUD(SH);
    }
}
if (scoreBoard) {
    nextMove.reset();
    glClear(GL_DEPTH_BUFFER_BIT);
    imgPtr->Draw(Loader::getShader(Loader::LoadedShaders::IMAGE));
}
}

```

## Obsługa połączenia sieciowego (serwer)

### SceneMultiplayerGame.cpp

```

void SceneMultiplayerGame::serverConnectionHandlerThread() {
    bool connection = true;
    Networking::MessageType ctrl;
    float posTemp[4];
    float bulletTmp[6];
    std::thread* tmpPtr;
    if (connection) {
        connection = Networking::sendControlMsg(Networking::MessageType::POSITION_FOLLOWS);
        playerPosLock.lock();
        posTemp[0] = playerPos[0];
    }
}

```

```

        posTemp[1] = playerPos[1];
        posTemp[2] = playerPos[2];
        posTemp[3] = playerPos[3];
        playerPosLock.unlock();
        connection = Networking::sendData(posTemp);
    }
    while (connection && run) {
        bulletLock.lock();
        if (sendBullet) {
            sendBullet = false;
            if (hit)
                connection = Networking::sendControlMsg(Networking::MessageType::HIT);
            else
                connection = Networking::sendControlMsg(Networking::MessageType::SHOOT);
            memcpy(bulletTmp, bulletContainer.back()->vertices, sizeof(float) * 6);
            posTemp[0] = bulletTmp[0];
            posTemp[1] = bulletTmp[1];
            posTemp[2] = bulletTmp[2];
            connection = Networking::sendData(posTemp);
            posTemp[0] = bulletTmp[3];
            posTemp[1] = bulletTmp[4];
            posTemp[2] = bulletTmp[5];
            connection = Networking::sendData(posTemp);
        }
        bulletLock.unlock();
        if (health <= 0) {
            connection = Networking::sendControlMsg(Networking::MessageType::NO_HEALTH);
            connection = Networking::sendControlMsg(Networking::MessageType::NO_HEALTH);
            connection = Networking::sendControlMsg(Networking::MessageType::NO_HEALTH);
            scoreBoard = true;
            imgPtr->SetTexture(Loader::getTexture2D(Loader::LoadedTextures2D::PRZECIWNICK_WON));
            //Networking::closeConnection();
            return;
        }
        connection = Networking::recvControlMsg(&ctrl);
        if (connection && (ctrl == Networking::MessageType::SHOOT || ctrl ==
Networking::MessageType::HIT)) {
            if (ctrl == Networking::MessageType::HIT){
                SoundM->PlaySFX("ScreamA.wav");
                health--;
            }
            else {
                SoundM->PlaySFX("laser.wav");
            }
            connection = Networking::recvData(posTemp);
            bulletTmp[0] = posTemp[0];
            bulletTmp[1] = posTemp[1];
            bulletTmp[2] = posTemp[2];
            connection = Networking::recvData(posTemp);
            if (!connection)
                break;
            bulletTmp[3] = posTemp[0];
            bulletTmp[4] = posTemp[1];
            bulletTmp[5] = posTemp[2];
            bulletLock.lock();
            bulletContainer.push_back(new Bullet(bulletTmp));
            bulletLock.unlock();
            tmpPtr = new std::thread([=] {deleteBullet(); });
            tmpPtr->detach();
        }
        else if (connection && ctrl == Networking::MessageType::POSITION_FOLLOW) {
            connection = Networking::recvData(posTemp);
            if (connection) {
                posLock.lock();
                pos[0] = posTemp[0];
                pos[1] = posTemp[1];

```

```

        pos[2] = posTemp[2];
        pos[3] = posTemp[3];
        posLock.unlock();
        connection =
Networking::sendControlMsg(Networking::MessageType::POSITION_FOLLOWS);
        playerPosLock.lock();
        posTemp[0] = playerPos[0];
        posTemp[1] = playerPos[1];
        posTemp[2] = playerPos[2];
        posTemp[3] = playerPos[3];
        playerPosLock.unlock();
        connection = Networking::sendData(posTemp);
    }
    else break;
}
else if (connection && ctrl == Networking::MessageType::NO_HEALTH) {
    scoreBoard = true;
    imgPtr->SetTexture(Loader::getTexture2D(Loader::LoadedTextures2D::JA_WON));
    Networking::closeConnection();
    return;
}
else if (!connection || ctrl == Networking::MessageType::QUIT) {
    break; //zakonczenie polaczenia
}
}
connection = false;
Networking::closeConnection();
quitLock.lock();
run = false;
quitLock.unlock();
}

```

## Pobieranie wysokości terenu

Terrain.cpp

```

float Terrain::getHeight(glm::vec3 pos)
{
    if (!heightMap || pos.x < 0.0f || pos.z < 0.0f) return 0.0f;
    int rowLength = heightMap->pitch;
    int column = static_cast<int>((pos.x / (map_x)) * rowLength);
    int row = static_cast<int>((pos.z / (map_z)) * rowLength);
    Uint8 pixel = *((Uint8*)heightMap->pixels + row * rowLength + column * heightMap->format->BytesPerPixel);
    Uint8 r, g, b;
    SDL_GetRGB(pixel, heightMap->format, &r, &g, &b);
    return (r / 255.0f) * map_y;
}

```

## Ladowanie modelu

Model.cpp

```

void Model::loadModel(const std::string &path)
{
    Assimp::Importer importer;
    //ReadFile zwraca wskaznik read only i jest usuwany podczas destrukcji importera
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs |
aiProcess_CalcTangentSpace);
    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) // if is Not Zero
    {
        std::cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << std::endl;
        return;
    }
}

```

```

        directory = path.substr(0, path.find_last_of('/'));

    processNode(scene->mRootNode, scene);
}

```

## Tworzenie i komplikacja shadera

### Shader.cpp

```

Shader::Shader(const char* vertexPath, const char* fragmentPath)
{
    std::string vertexCode;
    std::string fragmentCode;
    std::ifstream vShaderFile;
    std::ifstream fShaderFile;
    vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
    fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
    try
    {
        vShaderFile.open(vertexPath);
        fShaderFile.open(fragmentPath);
        std::stringstream vShaderStream, fShaderStream;
        vShaderStream << vShaderFile.rdbuf();
        fShaderStream << fShaderFile.rdbuf();
        vShaderFile.close();
        fShaderFile.close();
        vertexCode = vShaderStream.str();
        fragmentCode = fShaderStream.str();
    }
    catch (std::ifstream::failure e)
    {
        std::cout << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
    }
    const char* vShaderCode = vertexCode.c_str();
    const char * fShaderCode = fragmentCode.c_str();
    unsigned int vertex, fragment;
    vertex = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex, 1, &vShaderCode, NULL);
    glCompileShader(vertex);
    checkCompileErrors(vertex, "VERTEX");
    fragment = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment, 1, &fShaderCode, NULL);
    glCompileShader(fragment);
    checkCompileErrors(fragment, "FRAGMENT");

    ID = glCreateProgram();
    glAttachShader(ID, vertex);
    glAttachShader(ID, fragment);
    glLinkProgram(ID);
    checkCompileErrors(ID, "PROGRAM");
    glDeleteShader(vertex);
    glDeleteShader(fragment);
}

```

## Renderowanie HUDu gry

### HUD.cpp

```

void HUD::RenderHUD(vector<reference_wrapper<const Shader>> _LifeShader)
{

```

```

Shader Program;
for (LifeIterator = _LifeShader.begin(); LifeIterator != _LifeShader.end(); ++LifeIterator) {
    auto i = distance(_LifeShader.begin(), LifeIterator);
    glActiveTexture(GL_TEXTURE0+i);
    glBindTexture(GL_TEXTURE_2D, HealthID[i]);
}

RotationIntensity();

for (LifeIterator = _LifeShader.begin(); LifeIterator != _LifeShader.end(); ++LifeIterator) {
    auto i = distance(_LifeShader.begin(), LifeIterator);
    Program = _LifeShader[i];
    Program.use();
    glUniform1f(glGetUniformLocation(Program.ID, "gamma"), _gamma);
    Program.setInt("texture2", 1);
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
}
}

```

## Wyświetlanie Skyboxa

### Skybox.cpp

```

void CSkybox::RenderSkybox(Shader& Program, glm::mat4 ViewMatrix, glm::mat4 ProjectionMatrix)
{
    glDepthFunc(GL_LEQUAL);

    // change depth function so depth test passes when values are equal to depth buffer's content
    Program.use();

    float rotation += 5.0f * roatationspeed * _deltatime;
    glm::mat4 view = ViewMatrix;
    view[3][0] = 0;
    view[3][1] = 0;
    view[3][2] = 0;
    float Radian = static_cast<float>(DegreeToRadians(rotation));
    glm::mat4 view2 = glm::rotate(ViewMatrix, glm::radians(rotation), glm::vec3(0, 1, 0));
    glm::mat4 view3 = glm::mat4(glm::mat3(view2));
    //glm::rotate(DegreeToRadians(rotation), glm::vec3(0, 1, 0), ViewMatrix, ViewMatrix);
    glm::vec3 fogColour = { r = 0.7f, g = 0.7f, b = 0.7f }; // remove translation from the view matrix
    Program.setMat4("view", view3);
    Program.setMat4("projection", ProjectionMatrix);
    Program.setVec3("fogColour", fogColour);
    // skybox cube
    glBindVertexArray(skyboxVAO);
    glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxID);
    glActiveTexture(GL_TEXTURE0);
    glDrawArrays(GL_TRIANGLES, 0, 36);
    glBindVertexArray(0);
    glDepthFunc(GL_LESS);
}

```

## Pobieranie utworów muzycznych (SFX i Muzyka)

### Sound.cpp

```

Mix_Music * Sound::GetMusic(std::string filename)
{
    std::string Path = "res/img/" + filename;

    if (Music[Path] == nullptr) {

        Music[Path] = Mix_LoadMUS(Path.c_str());

        if (Music[Path] == NULL)
            std::cout << "ERROR LOADING MUSIC: FILE:" << filename.c_str() << " ERROR TYPE: "
<< Mix_GetError();
    }
    return Music[Path];
}

Mix_Chunk * Sound::GetSFX(std::string filename)
{
    std::string Path = "res/img/" + filename;
    //Path.append("res/img/" + filename);

    if (SFX[Path] == nullptr) {
        SFX[Path] = Mix_LoadWAV(Path.c_str());

        if (SFX[Path] == NULL)
            std::cout << "ERROR LOADING SFX: FILE:" << filename.c_str() << " ERROR TYPE: " <<
Mix_GetError();
    }
    return SFX[Path];
}

```

## Ladowanie tekstur do HeightMapy

### HeightMap.cpp

```

bool HeightMap::LoadTerrainShaderProgram()
{
    bool Loaded = true;
    Loaded &= shShaders[0].LoadShader("res\\shaders\\terrain.vert", GL_VERTEX_SHADER);
    Loaded &= shShaders[1].LoadShader("res\\shaders\\terrain.frag", GL_FRAGMENT_SHADER);
    Loaded &= shShaders[2].LoadShader("res\\shaders\\dirLight.frag", GL_FRAGMENT_SHADER);
    //bOK &= shShaders[3].LoadShader("res\\shaders\\fog1.frag", GL_FRAGMENT_SHADER);
    spTerrain.CreateProgram();

    for (int i = 0; i < NUMTERRAINSHADERS; ++i)
        spTerrain.AddShaderToProgram(&shShaders[i]);

    spTerrain.LinkProgram();

    return Loaded;
}

```

## Testowanie

Program testowany był pod względem poprawności działania w następujących sytuacjach wyjątkowych:

1. Szybkie przechodzenie po menu
2. Jako serwer
  - a. poprawne otworzenie portu
  - b. brak uprawnień do otwarcia portu
  - c. port aktualnie zajęty
3. Jako klient
  - a. Poprawne dane, nieudane połączenie
  - b. Niepoprawne dane
  - c. Wpisanie danych za pomocą klawiatury ekranowej
4. Wielokrotna zmiana roli Klient <-> Serwer
5. Podczas gry
  - a. próby wyjścia poza mapę
  - b. Strzelanie przez ściany
  - c. Strzelanie blisko przeciwnika, nie trafiając go
  - d. Przerywanie połączenia
  - e. Wychodzenie kombinacją alt+F4
6. Wychodzenie alt+F4 podczas nawiązywania połączenia

Dla każdej z powyższych sytuacji program zachował się stabilnie, informując użytkownika o zaistniałej niedogodności, kontynuując dalszą pracę

## Wycieki pamięci

Program był również testowany pod kątem wycieków pamięci w debuggerze Visual Studio (korzystając z dostępnych narzędzi Heap Profiler oraz Visual Leak Detector).

Po kilku zmianach program został skompilowany na systemie Arch Linux (zaktualizowany 3.06.2019), gdzie został przetestowany programem Valgrind. Zostały znalezione dwa wycieki w bibliotece Xlib:

```

112 (8 direct, 104 indirect) bytes in 1 blocks are definitely lost in loss record 1,902 of 1,951
at 0x48323BB: realloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
by 0x4E84EFE: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E853F6: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E86B39: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E8734D: _XlcCreateLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4EA5032: _XlcDefaultLoader (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E8EF5D: _XopenLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E8F0B6: _XlcCurrentLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E8F0F9: XSetLocaleModifiers (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x49077C4: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
by 0x490E6AB: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
by 0x48F4E81: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)

980 (68 direct, 912 indirect) bytes in 1 blocks are definitely lost in loss record 1,931 of 1,951
at 0x48323BB: realloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
by 0x4E84EFE: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E853F6: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E86B39: ??? (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E8734D: _XlcCreateLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4EA9000: _XlcUtf8Loader (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E8EF5D: _XopenLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E8F0B6: _XlcCurrentLC (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4E8F0F9: XSetLocaleModifiers (in /usr/lib/i386-linux-gnu/libX11.so.6.3.0)
by 0x4907879: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
by 0x490E6AB: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)
by 0x48F4E81: ??? (in /usr/lib/i386-linux-gnu/libSDL2-2.0.so.0.8.0)

```

Okazuje się, że wyciek powstaje podczas inicjalizacji biblioteki SDL2 i nie powiększa się. Analizując forum SDL okazało się, że wyciek biblioteki Xlib jest znany i nie jest w żaden sposób niebezpieczny.

Mając na uwadze powyższe oraz fakt, że nasza aplikacja projektowana była w założeniu na system Windows, postanowiliśmy dalej nie analizować wątku wycieku w bibliotece xlib.

## Wnioski

### Jakub Klimek

Praca nad programem była dla mnie przyjemnym doświadczeniem. Mój podstawowy cel, tj. poznanie podstaw technologii OpenGl, uważam za spełniony. Była to dla mnie pierwsza styczność z projektem tych rozmiarów, zawierającym tak wiele bibliotek dołączanych. Stanowiło to pewne wyzwanie logistyczne, z którym pomógł nam program CMake generujący pliki projektu Visual Studio.

Dodając kolejne elementy do programu zrozumiałem, jak istotne są zasady hermetyzacji oraz dekompozycji dla zachowania logicznej struktury kodu. Przy małych programach wydaje się to nieistotne, jednak pracując w grupie, należy kłaść szczególny nacisk na strukturę programu, by był on zrozumiały dla wszystkich członków zespołu.

### Michał Jankowski

Dany projekt semestralny pozwolił mi uzyskać ogromną ilość wiedzy. Jest on najbardziej skomplikowanym i czasochłonnym projektem jaki kiedykolwiek wykonałem. Dlatego też praca przy nim początkowo była dla mnie wielkim wysiłkiem intelektualnym, ponieważ musiałem zrozumieć zagadnienia trójwymiarowości w grach oraz (co okazało się bardziej skomplikowane) w jaki sposób je zaimplementować. Poznanie zaawansowanych aspektów grafiki komputerowej oraz algorytmów jakie nimi sterują była ciekawym doświadczeniem. OpenGL pozwala przedstawić (w skomplikowany sposób) rysowanie trójkątów oraz zależności między nimi, co pozwala na osiągnięcie takich efektów jak na przykład sześcian. Jednakowo wykorzystywanie kilkunastu bibliotek SDLowych w projekcie okazało się również wyzwaniem ze strony logistycznej, aby każdy miał podobną wersję oprogramowania. Rozwiążaniem okazało się tworzenie plików makefile tworzących solucję w Visual Studio z podanego kodu źródłowego oraz stosowanie wzorców projektowych np. Singleton

Dzięki faktowi, że projekt był tworzony we współpracy 2- osobowej również poznałem ciekawe narzędzie kontroli wersji takie jak Git Extensions. Kooperacja pozwoliła nam na podział obowiązków oraz także przypomniała, iż należy tworzyć modularny kod umożliwiający jego bezproblemowe dołączenie do reszty projektu. Oczywiście brak stosowania się do utrzymywania spójnej struktury kodu oraz zasad dekompozycji skutkowało błędami, które były rozwiązywane przy tzw. scalaniu wersji. Doświadczenie w tworzeniu gry 3D pozwala zrozumieć jak trudne okazuje się stworzenie wbrew pozorom prostych rzeczy.