# Dokumentacja Silnika 3D ebil

## 1. Wprowadzenie

Niniejszy dokument opisuje strukturę i działanie silnika **3D ebil** stworzonego przy użyciu biblioteki LWJGL 3 (Lightweight Java Game Library) oraz JOML (Java OpenGL Math Library). Silnik ten stanowi podstawę do nauki i dalszej rozbudowy, obejmując kluczowe koncepcje renderowania grafiki czasu rzeczywistego, takie jak zarządzanie oknem, obsługa wejścia, pętla gry, transformacje obiektów, oświetlenie i cieniowanie.

Celem silnika jest demonstracja podstawowych technik i zapewnienie szkieletu, na którym można budować bardziej złożone gry lub aplikacje graficzne.

## 2. Główne Funkcje

* **Zarządzanie Oknem:** Tworzenie i zarządzanie oknem aplikacji przy użyciu GLFW.
* **Kontekst OpenGL:** Inicjalizacja i zarządzanie kontekstem graficznym OpenGL (Core Profile 3.3+).
* **Pętla Gry:** Implementacja standardowej pętli gry z podziałem na fazy wejścia, aktualizacji i renderowania.
* **Obsługa Wejścia:** Śledzenie stanu klawiatury i myszy (pozycja, przyciski, delta ruchu).
* **System Czasu:** Mierzenie czasu między klatkami (deltaTime) dla płynnej animacji i fizyki.
* **Kamera:** Implementacja kamery 3D typu FPS (First-Person Shooter) z kontrolą za pomocą klawiatury i myszy. Obliczanie macierzy widoku i projekcji.
* **Obiekty Gry (GameObject):** Podstawowa reprezentacja obiektów na scenie z transformacjami (pozycja, rotacja, skala).
* **Renderowanie Siatek (Mesh):** Zarządzanie danymi geometrii (wierzchołki, normalne, indeksy) przy użyciu VAO i VBO.
* **Zarządzanie Shaderami (ShaderProgram):** Ładowanie, kompilacja, linkowanie programów shaderowych (GLSL) oraz zarządzanie uniformami.
* **Oświetlenie:** Implementacja prostego oświetlenia kierunkowego (model Phonga/Blinn-Phonga - Ambient, Diffuse, Specular).
* **Cieniowanie (Shadow Mapping):** Implementacja podstawowej techniki mapowania cieni dla światła kierunkowego przy użyciu dedykowanego Framebuffera (FBO) i tekstury głębi.

## 3. Wymagania i Konfiguracja

* **Java Development Kit (JDK):** Wersja 11 lub nowsza.
* **Maven:** System budowania projektu używany do zarządzania zależnościami.
* **System Operacyjny:** Windows, Linux lub macOS (wymaga odpowiedniej konfiguracji natywnych bibliotek LWJGL w pom.xml).

Projekt jest skonfigurowany przy użyciu pliku pom.xml, który zawiera zależności do:

* LWJGL (Core, GLFW, OpenGL)
* JOML (Java OpenGL Math Library)
* Natywne biblioteki LWJGL dla docelowej platformy (natives-windows, natives-linux, natives-macos).

## 4. Struktura Projektu (Główne Komponenty)

Projekt jest zorganizowany w pakiety odzwierciedlające funkcjonalność komponentów:

* **org.example.core**: Podstawowe elementy silnika.
  + Main.java: Punkt wejścia aplikacji, inicjalizacja i główna pętla gry.
  + Window.java: Zarządzanie oknem GLFW, kontekstem OpenGL i podstawowymi zdarzeniami okna. Współpracuje z Input.
  + Input.java: Obsługa wejścia z klawiatury i myszy. Zarządza callbackami GLFW i śledzi stan urządzeń.
  + Timer.java: Obliczanie czasu deltaTime.
* **org.example.graphics**: Komponenty związane bezpośrednio z renderowaniem.
  + Camera.java: Zarządzanie widokiem i projekcją kamery. Obsługuje nawigację.
  + Mesh.java: Reprezentacja geometrii 3D (VAO, VBO, EBO). Zawiera pozycje, normalne.
  + Renderer.java: Główna klasa orkiestrująca proces renderowania. Wykonuje przejście cieniowania (depth pass) i główne przejście renderowania sceny (scene pass).
  + ShaderProgram.java: Ładuje, kompiluje, linkuje i zarządza programami shaderowymi GLSL oraz ich uniformami.
  + light/DirectionalLight.java: Reprezentuje światło kierunkowe, jego właściwości (kolor, kierunek, intensywność) oraz oblicza macierze potrzebne do shadow mappingu.
  + shadow/ShadowMap.java: Zarządza Framebufferem (FBO) i teksturą głębi używaną do generowania mapy cieni.
* **org.example.scene**: Komponenty związane z reprezentacją sceny.
  + GameObject.java: Podstawowy budulec sceny. Posiada pozycję, rotację, skalę, referencję do Mesh oraz kolor.
* **org.example.util**: Klasy pomocnicze.
  + MeshLoader.java: Prosta klasa do generowania podstawowych siatek (np. sześcian, płaszczyzna).
* **src/main/resources/shaders**: Pliki shaderów GLSL.
  + scene\_vertex.glsl / scene\_fragment.glsl: Shadery do renderowania głównej sceny z oświetleniem i cieniami.
  + depth\_vertex.glsl / depth\_fragment.glsl: Shadery do generowania mapy głębi (shadow map) z perspektywy światła.

## 5. Omówienie Kodu

Ta sekcja zawiera szczegółowe omówienie poszczególnych plików źródłowych (Java) oraz shaderów (GLSL) składających się na silnik **3D ebil**.

### 5.1. Komponenty Rdzenia (org.example.core)

Komponenty w tym pakiecie stanowią fundament silnika, zarządzając oknem, pętlą gry, czasem i podstawowym wejściem.

#### 5.1.1. Main.java

* **Przeznaczenie:** Główny punkt wejścia aplikacji. Odpowiada za inicjalizację wszystkich podsystemów silnika, uruchomienie głównej pętli gry oraz zarządzanie procesem czyszczenia zasobów po zakończeniu działania.
* **Kluczowe Pola:** Przechowuje referencje do głównych komponentów: window, input, timer, camera, renderer, a także listę obiektów gry (gameObjects) i światło kierunkowe (directionalLight).
* **Kluczowe Metody:**
  + run(): Metoda startowa, która opakowuje cały cykl życia aplikacji (inicjalizacja, pętla, sprzątanie) w blok try...finally, aby zapewnić zwolnienie zasobów nawet w przypadku błędu.
  + init(): Tworzy instancje klas Input, Timer, Window. Inicjalizuje okno (window.init(), co tworzy kontekst OpenGL). Inicjalizuje Renderer (ładowanie shaderów, tworzenie mapy cieni). Tworzy instancję Camera. Wczytuje siatki (MeshLoader) i tworzy obiekty gry (GameObject), dodając je do listy gameObjects. Inicjalizuje DirectionalLight.
  + loop(): Główna pętla gry, która działa dopóki window.windowShouldClose() zwraca false. W każdej iteracji:
    1. Aktualizuje timer (timer.update()).
    2. Przetwarza wejście użytkownika (processInput()).
    3. Aktualizuje logikę gry (updateGameLogic()).
    4. Renderuje scenę (renderer.render()).
    5. Aktualizuje okno (zamiana buforów, przetwarzanie zdarzeń GLFW - window.update()).
    6. Aktualizuje stan wewnętrzny Input (np. obliczenie delty myszy - input.update()). Kolejność wywołań window.update() i input.update() jest istotna.
  + processInput(): Pobiera stan wejścia z klasy Input i aktualizuje kamerę (camera.processKeyboard(), camera.processMouseMovement()).
  + updateGameLogic(): Miejsce na logikę specyficzną dla gry. W tej wersji demonstracyjnej obraca niektóre sześciany.
  + cleanup(): Wywołuje metody cleanup() na wszystkich zarządzanych komponentach (renderer, gameObjects (czyszczenie Mesh), window) w celu zwolnienia zasobów OpenGL i GLFW.
* **Interakcje:** Jest to główny "orkiestrator", który tworzy i zarządza cyklem życia większości innych komponentów.

#### 5.1.2. Window.java

* **Przeznaczenie:** Zarządzanie oknem aplikacji przy użyciu biblioteki GLFW. Odpowiada za utworzenie okna, inicjalizację kontekstu OpenGL, obsługę podstawowych zdarzeń okna (zmiana rozmiaru, zamknięcie) oraz ustawienie callbacków wejścia, delegując ich obsługę do klasy Input.
* **Kluczowe Pola:** windowHandle (uchwyt GLFW), width, height, title (tytuł okna), input (referencja do obiektu Input).
* **Kluczowe Metody:**
  + init(Input input): Inicjalizuje GLFW, ustawia "podpowiedzi" dla tworzonego okna (wersja OpenGL, profil Core, niewidoczne na starcie, możliwość zmiany rozmiaru). Tworzy okno (glfwCreateWindow). Rejestruje callbacki GLFW (glfwSetKeyCallback, glfwSetCursorPosCallback, glfwSetMouseButtonCallback), przekazując do nich metody z obiektu Input. Dodatkowo, callback klawiatury obsługuje zamknięcie okna klawiszem ESC. Inicjalizuje obiekt Input (input.init()). Wyśrodkowuje okno na ekranie. Ustawia bieżący kontekst OpenGL (glfwMakeContextCurrent). Włącza V-Sync (glfwSwapInterval(1)). Tworzy powiązania OpenGL (GL.createCapabilities()). Pokazuje okno (glfwShowWindow). Ustawia viewport i callback zmiany rozmiaru okna (glfwSetFramebufferSizeCallback), który aktualizuje viewport OpenGL.
  + isKeyPressed(): Deleguje sprawdzenie stanu klawisza do Input.
  + windowShouldClose(): Zwraca flagę zamknięcia okna GLFW.
  + update(): Zamienia przedni i tylny bufor OpenGL (glfwSwapBuffers) oraz przetwarza zdarzenia systemowe i wejścia (glfwPollEvents).
  + cleanup(): Zwalnia callbacki (input.cleanup() jest wywoływane), niszczy okno (glfwDestroyWindow) i kończy działanie GLFW (glfwTerminate).
* **Interakcje:** Bezpośrednio używa funkcji GLFW. Ściśle współpracuje z Input w zakresie obsługi zdarzeń wejścia. Używana przez Main i Renderer.

#### 5.1.3. Input.java

* **Przeznaczenie:** Abstrakcja nad surowym wejściem GLFW. Śledzi stan klawiatury (wciśnięte klawisze) i myszy (pozycja, wciśnięte przyciski, zmiana pozycji - delta). Zarządza instancjami callbacków GLFW.
* **Kluczowe Pola:** Statyczne tablice keys[] i buttons[] przechowujące stan wciśnięcia. Statyczne zmienne mouseX, mouseY (bieżąca pozycja), prevMouseX, prevMouseY (pozycja z poprzedniej klatki) oraz mouseDelta (obliczona zmiana pozycji). Instancje callbacków GLFW (keyboardCallback, mouseMoveCallback, mouseButtonCallback).
* **Kluczowe Metody:**
  + Konstruktor: Tworzy instancje callbacków, które aktualizują wewnętrzne tablice stanu (keys, buttons) oraz pozycję myszy (mouseX, mouseY).
  + isKeyDown(), isMouseButtonDown(): Statyczne metody publiczne do odpytywania stanu klawiszy/przycisków.
  + getMouseX(), getMouseY(), getMouseDelta(): Statyczne metody publiczne do pobierania informacji o myszy.
  + init(long windowHandle): Ustawia tryb kursora na GLFW\_CURSOR\_DISABLED (ukryty i zablokowany w oknie, typowy dla gier FPS) oraz włącza GLFW\_RAW\_MOUSE\_MOTION, jeśli jest wspierany (dla precyzyjniejszego ruchu niezależnego od akceleracji systemowej). Resetuje początkowy stan delty myszy.
  + update(): Oblicza wektor mouseDelta na podstawie różnicy między bieżącą a poprzednią pozycją myszy. Aktualizuje prevMouseX, prevMouseY. **Musi być wywołana po glfwPollEvents()**.
  + cleanup(): Zwalnia zasoby alokowane przez instancje callbacków (.free()).
  + getKeyboardCallback(), getMouseMoveCallback(), getMouseButtonCallback(): Publiczne gettery zwracające instancje callbacków, używane przez Window podczas rejestracji.
* **Interakcje:** Dostarcza informacji o stanie wejścia do Camera (w Main.processInput()) oraz potencjalnie do logiki gry. Window używa jej callbacków i metody init().

#### 5.1.4. Timer.java

* **Przeznaczenie:** Prosta klasa do mierzenia czasu, który upłynął pomiędzy kolejnymi klatkami renderowania (deltaTime).
* **Kluczowe Pola:** lastLoopTime (czas zakończenia poprzedniej klatki), deltaTime (obliczony czas trwania ostatniej klatki w sekundach).
* **Kluczowe Metody:**
  + init(): Zapisuje początkowy czas.
  + getTime(): Zwraca bieżący czas od inicjalizacji GLFW (glfwGetTime()).
  + getDeltaTime(): Zwraca obliczony deltaTime.
  + update(): Oblicza deltaTime jako różnicę między bieżącym czasem a lastLoopTime, następnie aktualizuje lastLoopTime. Powinna być wywoływana raz na początku każdej klatki pętli gry.
* **Interakcje:** Używana przez Main do uzyskania deltaTime, który jest przekazywany do metod processInput i updateGameLogic w celu zapewnienia płynności ruchu i animacji niezależnej od liczby klatek na sekundę (FPS).

### 5.2. Komponenty Graficzne (org.example.graphics)

Te klasy są bezpośrednio odpowiedzialne za aspekty wizualne i proces renderowania.

#### 5.2.1. Camera.java

* **Przeznaczenie:** Reprezentuje wirtualną kamerę w scenie 3D. Zarządza jej pozycją, orientacją (za pomocą kątów Eulera - yaw i pitch) oraz parametrami (prędkość ruchu, czułość myszy, kąt widzenia - FOV). Oblicza macierze Widoku (View) i Projekcji (Projection) niezbędne do transformacji sceny.
* **Kluczowe Pola:** position, front (kierunek patrzenia), up (kierunek "góry" kamery), right (kierunek "prawo" kamery), worldUp (stały wektor "góry" świata), yaw, pitch, movementSpeed, mouseSensitivity, fov.
* **Kluczowe Metody:**
  + getViewMatrix(): Oblicza i zwraca macierz Widoku przy użyciu metody lookAt z biblioteki JOML, na podstawie position, position + front (punkt docelowy) i up.
  + getProjectionMatrix(float aspectRatio): Oblicza i zwraca macierz Projekcji perspektywicznej (perspective), używając fov (przekonwertowanego na radiany za pomocą java.lang.Math.toRadians i zrzutowanego na float), aspectRatio (stosunek szerokości do wysokości okna) oraz płaszczyzn przycinania zNear (0.1f) i zFar (100.0f).
  + processKeyboard(float deltaTime): Aktualizuje pozycję kamery (position) na podstawie wciśniętych klawiszy (W, A, S, D, Spacja, Shift) i deltaTime. Ruch odbywa się wzdłuż wektorów front, right i worldUp.
  + processMouseMovement(float xoffset, float yoffset, boolean constrainPitch): Aktualizuje kąty yaw i pitch na podstawie zmiany pozycji myszy (xoffset, yoffset) i czułości. Opcjonalnie ogranicza pitch, aby zapobiec "przewrotce" kamery. Wywołuje updateCameraVectors() na końcu.
  + updateCameraVectors(): Prywatna metoda, która przelicza wektory front, right i up na podstawie aktualnych wartości yaw i pitch przy użyciu funkcji trygonometrycznych (java.lang.Math.cos, java.lang.Math.sin, java.lang.Math.toRadians). Zapewnia, że wektory kierunkowe kamery są zawsze aktualne i ortonormalne.
* **Interakcje:** Pobiera stan z Input (poprzez Main). Dostarcza macierze View i Projection oraz pozycję kamery (viewPos) do Renderer. Jej stan jest modyfikowany w Main.processInput.

#### 5.2.2. Mesh.java

* **Przeznaczenie:** Reprezentuje pojedynczą, renderowalną siatkę geometryczną (model 3D). Zarządza obiektami buforów OpenGL: Vertex Array Object (VAO), Vertex Buffer Objects (VBOs) dla pozycji i normalnych oraz Element Buffer Object (EBO/IBO) dla indeksów.
* **Kluczowe Pola:** Identyfikatory obiektów OpenGL: vaoId, posVboId (pozycje), normalVboId (normalne), idxVboId (indeksy), oraz vertexCount (liczba indeksów do narysowania).
* **Kluczowe Metody:**
  + Konstruktor (Mesh(float[] positions, float[] normals, int[] indices)): Przyjmuje tablice z danymi wierzchołków. Tworzy VAO i wiąże je. Tworzy VBO dla pozycji, przesyła dane (glBufferData) i konfiguruje atrybut wierzchołka 0 (glVertexAttribPointer, glEnableVertexAttribArray). Tworzy VBO dla normalnych, przesyła dane i konfiguruje atrybut wierzchołka 1. Tworzy EBO (które jest wiązane z VAO), przesyła dane indeksów. Odwiązuje VAO i VBO. Używa MemoryUtil do alokacji i zwalniania buforów NIO.
  + render(): Wiąże VAO skojarzone z siatką i wywołuje komendę rysowania glDrawElements, która używa danych z aktywnych VBO i EBO powiązanych z VAO. Odwiązuje VAO po zakończeniu.
  + cleanup(): Wyłącza atrybuty wierzchołków, odwiązuje bufory i usuwa wszystkie utworzone obiekty OpenGL (glDeleteBuffers, glDeleteVertexArrays) w celu zwolnienia pamięci na GPU.
* **Interakcje:** Przechowywana przez GameObject. Tworzona (w tym demie) przez MeshLoader. Używana przez Renderer do wykonania faktycznego rysowania.

#### 5.2.3. Renderer.java

* **Przeznaczenie:** Główna klasa odpowiedzialna za proces renderowania całej sceny. Orkiestruje rysowanie, zarządzając shaderami, ustawiając uniformy i wywołując rysowanie siatek. Implementuje dwuprzebiegowe renderowanie dla mapowania cieni.
* **Kluczowe Pola:** Referencje do programów shaderowych (sceneShaderProgram, depthShaderProgram), mapy cieni (shadowMap) oraz okna (window - potrzebne do pobrania rozmiaru dla resetowania viewportu).
* **Kluczowe Metody:**
  + init(): Tworzy instancję ShadowMap. Ładuje, kompiluje, linkuje i tworzy uniformy dla obu programów shaderowych (sceneShaderProgram i depthShaderProgram). Włącza globalne stany OpenGL, takie jak test głębi (GL\_DEPTH\_TEST) i odrzucanie tylnych ścianek (GL\_CULL\_FACE).
  + render(Camera camera, List<GameObject> gameObjects, DirectionalLight light): Publiczna metoda wyzwalająca renderowanie klatki. Wywołuje najpierw renderDepthMap, a następnie renderScene.
  + renderDepthMap(): Prywatna metoda wykonująca pierwszy przebieg (renderowanie mapy cieni). Wiąże FBO mapy cieni (shadowMap.bindForWriting()). Aktywuje depthShaderProgram. Ustawia uniform lightSpaceMatrix (otrzymany z light.getLightSpaceMatrix()). Iteruje przez gameObjects, ustawiając dla każdego uniform model (z go.getModelMatrix()) i wywołując go.getMesh().render(). Odwiązuje FBO (shadowMap.unbindAfterWriting()), przywracając domyślny framebuffer i viewport okna.
  + renderScene(): Prywatna metoda wykonująca drugi, główny przebieg renderowania. Czyści bufory koloru i głębi domyślnego framebuffera (glClear). Aktywuje sceneShaderProgram. Ustawia uniformy kamery (projection, view, viewPos). Ustawia uniformy światła (dirLight.\*). Ustawia uniform lightSpaceMatrix (potrzebny do obliczenia pozycji w przestrzeni światła w shaderze fragmentów). Aktywuje jednostkę teksturującą (np. GL\_TEXTURE0), wiąże teksturę mapy cieni (shadowMap.getDepthMapTexture()) i ustawia uniform shadowMap (sampler) na indeks tej jednostki (0). Ustawia uniform shadowBias. Iteruje przez gameObjects, ustawiając uniformy model i objectColor dla każdego obiektu i wywołując go.getMesh().render(). Odwiązuje shader program.
  + cleanup(): Wywołuje metody cleanup() na zarządzanych shaderach i mapie cieni.
* **Interakcje:** Jest centralnym punktem procesu renderowania. Używa Camera (macierze), GameObject (transformacje, siatki, kolor), Mesh (rysowanie), DirectionalLight (właściwości, macierze światła), ShadowMap (FBO, tekstura głębi), ShaderProgram (aktywacja, ustawianie uniformów), Window (rozmiar viewportu). Wywoływana przez Main.

#### 5.2.4. ShaderProgram.java

* **Przeznaczenie:** Hermetyzuje logikę związaną z programami shaderowymi OpenGL. Odpowiada za ładowanie kodu shadera z plików, jego kompilację, linkowanie vertex i fragment shaderów w kompletny program, walidację oraz zarządzanie (tworzenie i ustawianie) zmiennych uniform.
* **Kluczowe Pola:** programId (identyfikator programu OpenGL), vertexShaderId, fragmentShaderId (tymczasowe identyfikatory shaderów obiektowych), uniforms (mapa przechowująca nazwy uniformów i ich lokalizacje w programie shaderowym - jako cache).
* **Kluczowe Metody:**
  + Konstruktor: Tworzy pusty program shaderowy (glCreateProgram).
  + createVertexShader(), createFragmentShader(): Przyjmują kod shadera jako string i wywołują createShader.
  + createShader(): Tworzy obiekt shadera (glCreateShader), ładuje kod źródłowy (glShaderSource), kompiluje (glCompileShader), sprawdza błędy kompilacji (glGetShaderi, glGetShaderInfoLog) i dołącza shader do programu (glAttachShader).
  + link(): Linkuje skompilowane shadery w finalny program (glLinkProgram), sprawdza błędy linkowania (glGetProgrami, glGetProgramInfoLog), waliduje program (glValidateProgram - dobra praktyka podczas developmentu) i odłącza oraz usuwa shadery obiektowe (glDetachShader, glDeleteShader), które nie są już potrzebne po zlinkowaniu.
  + createUniform(): Pobiera lokalizację uniformu w programie shaderowym (glGetUniformLocation) i zapisuje ją w mapie uniforms. Obsługuje sytuację, gdy uniform nie zostanie znaleziony (np. jest nieużywany w shaderze).
  + setUniform() (przeciążone): Metody do ustawiania wartości uniformów różnych typów (Matrix4f, Vector3f, float, int). Używają odpowiednich funkcji glUniform\*. W przypadku macierzy używa MemoryStack do efektywnego przekazania danych.
  + bind(): Aktywuje program shaderowy (glUseProgram).
  + unbind(): Dezaktywuje bieżący program shaderowy (glUseProgram(0)).
  + cleanup(): Dezaktywuje i usuwa program shaderowy (glDeleteProgram).
  + loadShaderSource(): Statyczna metoda pomocnicza do wczytywania zawartości pliku shadera jako string.
* **Interakcje:** Używana przez Renderer do zarządzania shaderami. Wczytuje pliki .glsl z zasobów.

#### 5.2.5. light/DirectionalLight.java

* **Przeznaczenie:** Reprezentuje źródło światła kierunkowego, które świeci równolegle w całej scenie z nieskończonej odległości. Przechowuje jego właściwości (kolor, kierunek, intensywność) i dostarcza metody do obliczenia macierzy potrzebnych do renderowania mapy cieni z jego perspektywy.
* **Kluczowe Pola:** color, direction (znormalizowany), intensity, shadowPosMult (mnożnik odległości kamery światła), orthoSize (rozmiar rzutowania ortogonalnego dla mapy cieni).
* **Kluczowe Metody:**
  + Konstruktor: Ustawia właściwości światła, normalizując kierunek.
  + Gettery/Settery: Standardowe metody dostępu.
  + getLightProjectionMatrix(): Zwraca macierz rzutowania ortogonalnego (ortho) używaną do renderowania mapy cieni. Rzutowanie ortogonalne jest odpowiednie dla światła kierunkowego.
  + getLightViewMatrix(): Zwraca macierz widoku z perspektywy światła. Używa lookAt, ustawiając pozycję kamery światła w pewnej odległości w kierunku przeciwnym do kierunku światła (direction \* -shadowPosMult), patrząc na środek sceny (uproszczenie) z wektorem "góry" zazwyczaj (0,1,0), chyba że kierunek światła jest zbyt bliski osi Y.
  + getLightSpaceMatrix(): Zwraca połączoną macierz Projekcji \* Widoku dla światła (P \* V), która transformuje współrzędne ze świata do przestrzeni przycinania (clip space) światła. Jest ona używana zarówno w depthShaderProgram (do transformacji wierzchołków), jak i w sceneShaderProgram (do obliczenia współrzędnych próbkowania mapy cieni).
* **Interakcje:** Używana przez Renderer do ustawiania uniformów światła i cieni.

#### 5.2.6. shadow/ShadowMap.java

* **Przeznaczenie:** Zarządza zasobami OpenGL potrzebnymi do techniki mapowania cieni: Framebuffer Object (FBO) oraz teksturą przechowującą wartości głębi sceny widzianej z perspektywy światła.
* **Kluczowe Pola:** SHADOW\_MAP\_WIDTH, SHADOW\_MAP\_HEIGHT (stałe definiujące rozdzielczość mapy cieni), depthMapFBO (identyfikator FBO), depthMapTexture (identyfikator tekstury głębi).
* **Kluczowe Metody:**
  + Konstruktor: Tworzy FBO (glGenFramebuffers). Tworzy teksturę 2D (glGenTextures). Wiąże teksturę i konfiguruje ją (glTexImage2D) jako teksturę głębi (GL\_DEPTH\_COMPONENT16, GL\_DEPTH\_COMPONENT) o zadanej rozdzielczości, bez przesyłania danych (ostatni argument null). Ustawia parametry filtrowania (GL\_NEAREST) i zawijania (GL\_CLAMP\_TO\_EDGE). Wiąże FBO i dołącza do niego stworzoną teksturę głębi jako GL\_DEPTH\_ATTACHMENT (glFramebufferTexture2D). Informuje OpenGL, że FBO nie będzie renderować do żadnego bufora koloru (glDrawBuffer(GL\_NONE), glReadBuffer(GL\_NONE)). Sprawdza, czy FBO zostało poprawnie skonfigurowane (glCheckFramebufferStatus). Odwiązuje FBO.
  + bindForWriting(): Wiąże FBO mapy cieni (glBindFramebuffer), ustawia viewport OpenGL na rozmiar mapy cieni i czyści bufor głębi (glClear(GL\_DEPTH\_BUFFER\_BIT)). Przygotowuje do renderowania *do* mapy cieni.
  + unbindAfterWriting(int windowWidth, int windowHeight): Odwiązuje FBO (wraca do domyślnego framebuffera 0) i przywraca viewport OpenGL do rozmiarów okna aplikacji.
  + getDepthMapTexture(): Zwraca identyfikator tekstury głębi, aby mogła być użyta (próbkowana) w shaderze sceny.
  + cleanup(): Usuwa obiekt FBO (glDeleteFramebuffers) i teksturę głębi (glDeleteTextures).
* **Interakcje:** Używana przez Renderer do przełączania celów renderowania pomiędzy mapą cieni a głównym oknem oraz do uzyskania tekstury głębi.

### 5.3. Komponenty Sceny (org.example.scene)

Klasy reprezentujące elementy składowe wirtualnej sceny.

#### 5.3.1. GameObject.java

* **Przeznaczenie:** Podstawowa klasa reprezentująca obiekt w scenie 3D. Każdy obiekt posiada informacje o swojej transformacji (pozycji, rotacji, skali), referencję do siatki (Mesh), którą ma renderować, oraz podstawowy kolor.
* **Kluczowe Pola:** mesh (referencja do obiektu Mesh), position (Vector3f), rotation (Quaternionf - kwaterniony są preferowane do unikania problemu "gimbal lock"), scale (Vector3f), color (Vector3f - dla uproszczenia, zamiast materiału/tekstury).
* **Kluczowe Metody:**
  + Konstruktor: Inicjalizuje obiekt z domyślnymi transformacjami (pozycja (0,0,0), brak rotacji, skala (1,1,1)) i domyślnym białym kolorem. Wymaga przekazania Mesh.
  + getModelMatrix(): Oblicza i zwraca macierz Modelu (World Matrix) dla tego obiektu, stosując kolejno translację, rotację (z kwaternionu) i skalowanie do macierzy jednostkowej. Ta macierz transformuje obiekt z jego lokalnej przestrzeni do przestrzeni świata.
  + Gettery/Settery: Standardowe metody dostępu do pól, w tym metody do ustawiania pozycji, skali, koloru oraz rotacji (bezpośrednio z kąta i osi lub przez dodanie obrotu rotate).
* **Interakcje:** Przechowuje referencję do Mesh. Jest zarządzany przez Main w liście gameObjects. Używany przez Renderer do pobrania macierzy modelu i siatki do renderowania.

### 5.4. Narzędzia (org.example.util)

Klasy pomocnicze ułatwiające pewne zadania.

#### 5.4.1. MeshLoader.java

* **Przeznaczenie:** Prosta klasa narzędziowa dostarczająca statyczne metody do tworzenia podstawowych obiektów Mesh (w tym demie: sześcianu i płaszczyzny). Definiuje ona "na sztywno" dane wierzchołków (pozycje, normalne) i indeksy dla tych kształtów.
* **Kluczowe Metody:**
  + createCube(): Definiuje tablice positions, normals i indices dla standardowego sześcianu o boku 1, wyśrodkowanego w (0,0,0), a następnie tworzy i zwraca nowy obiekt Mesh z tymi danymi.
  + createPlane(float size): Definiuje tablice positions, normals i indices dla płaskiej płaszczyzny w płaszczyźnie XZ o zadanym rozmiarze, a następnie tworzy i zwraca nowy obiekt Mesh.
* **Interakcje:** Tworzy instancje Mesh. Używana przez Main podczas inicjalizacji sceny.

### 5.5. Shadery (src/main/resources/shaders)

Pliki GLSL (OpenGL Shading Language) definiujące programy wykonywane na karcie graficznej (GPU) podczas renderowania.

#### 5.5.1. depth\_vertex.glsl

* **Przeznaczenie:** Vertex shader używany podczas pierwszego przebiegu renderowania (generowanie mapy cieni). Jego jedynym zadaniem jest przetransformowanie pozycji wierzchołka z przestrzeni modelu do przestrzeni przycinania (clip space) światła.
* **Wejście:** aPos (pozycja wierzchołka w przestrzeni modelu - atrybut 0).
* **Uniformy:** lightSpaceMatrix (połączona macierz Widoku \* Projekcji światła), model (macierz modelu obiektu).
* **Wyjście:** gl\_Position (pozycja wierzchołka w przestrzeni przycinania światła).
* **Logika:** Mnoży pozycję wierzchołka przez macierz modelu, a następnie przez macierz przestrzeni światła: gl\_Position = lightSpaceMatrix \* model \* vec4(aPos, 1.0);.

#### 5.5.2. depth\_fragment.glsl

* **Przeznaczenie:** Fragment shader używany podczas generowania mapy cieni. Ponieważ interesuje nas tylko zapis głębi, a nie koloru, ten shader jest celowo pusty. Wartość głębi jest zapisywana automatycznie do tekstury głębi podłączonej do FBO.
* **Wejście/Wyjście/Uniformy:** Brak.

#### 5.5.3. scene\_vertex.glsl

* **Przeznaczenie:** Vertex shader używany podczas drugiego, głównego przebiegu renderowania sceny. Transformuje wierzchołki do różnych przestrzeni i przekazuje niezbędne dane do shadera fragmentów.
* **Wejście:** aPos (pozycja wierzchołka - atrybut 0), aNormal (normalna wierzchołka - atrybut 1).
* **Uniformy:** projection, view, model (standardowe macierze transformacji), lightSpaceMatrix (macierz przestrzeni światła).
* **Wyjście (do Fragment Shadera):** Blok VS\_OUT zawierający:
  + FragPos: Pozycja fragmentu w przestrzeni świata (vec3(model \* vec4(aPos, 1.0))).
  + Normal: Normalna w przestrzeni świata (transformowana przez macierz normalnych mat3(transpose(inverse(model))) \* aNormal dla poprawnej obsługi niesymetrycznego skalowania).
  + FragPosLightSpace: Pozycja fragmentu przetransformowana do przestrzeni przycinania światła (lightSpaceMatrix \* vec4(vs\_out.FragPos, 1.0)). Potrzebna do próbkowania mapy cieni.
* **Wyjście (Główne):** gl\_Position (pozycja wierzchołka w przestrzeni przycinania kamery: projection \* view \* vec4(vs\_out.FragPos, 1.0)).

#### 5.5.4. scene\_fragment.glsl

* **Przeznaczenie:** Fragment shader używany podczas głównego przebiegu renderowania. Oblicza ostateczny kolor dla każdego fragmentu (piksela) na podstawie oświetlenia, koloru obiektu i informacji o cieniu.
* **Wejście:** Blok fs\_in z danymi z vertex shadera (FragPos, Normal, FragPosLightSpace).
* **Uniformy:** objectColor, viewPos (pozycja kamery), struktura DirLight (zawierająca direction, color, intensity), shadowMap (sampler2D - tekstura mapy cieni), shadowBias (mała wartość do uniknięcia "shadow acne").
* **Wyjście:** FragColor (ostateczny kolor fragmentu - vec4).
* **Logika:**
  1. Normalizuje wejściową normalną (fs\_in.Normal).
  2. **Oświetlenie Otoczenia (Ambient):** Oblicza stały komponent oświetlenia (ambientStrength \* dirLight.color \* dirLight.intensity).
  3. **Oświetlenie Rozproszone (Diffuse):** Oblicza kierunek *do* światła (lightDir). Oblicza współczynnik rozproszenia na podstawie kąta między normalną a kierunkiem do światła (diff = max(dot(normal, lightDir), 0.0)). Oblicza kolor rozproszenia (diff \* dirLight.color \* dirLight.intensity).
  4. **Oświetlenie Lustrzane (Specular):** Oblicza kierunek *do* obserwatora (viewDir). Używa modelu Blinn-Phong: oblicza wektor połówkowy (halfwayDir = normalize(lightDir + viewDir)). Oblicza współczynnik odbicia lustrzanego na podstawie kąta między normalną a wektorem połówkowym, podniesionego do potęgi połyskliwości (spec = pow(max(dot(normal, halfwayDir), 0.0), 32.0)). Oblicza kolor lustrzany (specularStrength \* spec \* dirLight.color \* dirLight.intensity).
  5. **Obliczanie Cienia (CalculateShadow):**
     + Wykonuje dzielenie perspektywiczne na FragPosLightSpace (projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w).
     + Transformuje współrzędne do zakresu [0, 1] (projCoords = projCoords \* 0.5 + 0.5).
     + Pobiera najbliższą głębię zapisaną w mapie cieni dla tych współrzędnych (closestDepth = texture(shadowMap, projCoords.xy).r).
     + Pobiera bieżącą głębię fragmentu z perspektywy światła (currentDepth = projCoords.z).
     + Sprawdza, czy fragment jest poza mapą cieni (projCoords.z > 1.0).
     + Porównuje currentDepth z closestDepth, uwzględniając shadowBias. Jeśli currentDepth - shadowBias > closestDepth, fragment jest w cieniu (shadow = 1.0), w przeciwnym razie jest oświetlony (shadow = 0.0). (Zakomentowany kod pokazuje prosty wariant PCF dla miększych cieni).
  6. **Kolor Końcowy:** Łączy komponenty oświetlenia, mnożąc diffuse i specular przez (1.0 - shadow), aby zastosować cień. Dodaje ambient (który nie jest cieniowany). Mnoży wynikowy kolor oświetlenia przez objectColor. Ustawia FragColor.
* **Interakcje:** Otrzymuje dane z scene\_vertex.glsl. Używa wielu uniformów ustawionych przez Renderer. Próbkuje teksturę shadowMap.

## 6. Shadery GLSL

* **depth\_vertex.glsl**: Transformuje pozycje wierzchołków do przestrzeni światła (przy użyciu lightSpaceMatrix).
* **depth\_fragment.glsl**: Jest pusty, ponieważ w tym przejściu interesuje nas tylko zapis głębi do tekstury (co odbywa się automatycznie).
* **scene\_vertex.glsl**: Transformuje pozycje wierzchołków do przestrzeni świata, widoku i projekcji. Przekazuje pozycję fragmentu, normalną oraz pozycję fragmentu w przestrzeni światła do shadera fragmentów.
* **scene\_fragment.glsl**: Oblicza ostateczny kolor fragmentu, uwzględniając:
  + Kolor obiektu.
  + Oświetlenie otoczenia, rozproszone i lustrzane (Ambient, Diffuse, Specular) na podstawie właściwości światła kierunkowego i normalnej powierzchni.
  + Pozycję kamery (dla oświetlenia lustrzanego).
  + Obliczenie cienia (CalculateShadow) poprzez porównanie głębi fragmentu w przestrzeni światła z wartością z mapy cieni (shadowMap).
  + Zastosowanie cienia do komponentów rozproszonych i lustrzanych oświetlenia.

## 7. Potencjalne Ulepszenia i Dalszy Rozwój

* **Tekstury:** Dodanie obsługi ładowania i renderowania tekstur na obiektach.
* **Materiały:** System materiałów definiujący właściwości powierzchni (kolor bazowy, metaliczność, chropowatość, mapy tekstur).
* **Więcej Typów Świateł:** Światła punktowe, reflektory (spotlights).
* **Wiele Świateł:** Obsługa wielu źródeł światła na scenie.
* **System Sceny:** Bardziej zaawansowana struktura do zarządzania obiektami (np. drzewo sceny).
* **Ładowanie Modeli:** Integracja z bibliotekami do ładowania modeli 3D z plików (np. Assimp przez LWJGL bindings).
* **Fizyka:** Integracja z silnikiem fizyki (np. JBullet).
* **Dźwięk:** Dodanie obsługi dźwięku (np. OpenAL przez LWJGL bindings).
* **GUI:** Implementacja interfejsu użytkownika (np. Nuklear, ImGui przez LWJGL bindings).
* **Optymalizacje:** Frustum Culling, Occlusion Culling, Instanced Rendering, Level of Detail (LOD).
* **Jakość Cieni:** Poprawa jakości cieni (PCF, VSM, ESM).

## 8. Podsumowanie

Silnik **3D ebil** jest prostym, ale funkcjonalnym przykładem wykorzystania LWJGL 3 i JOML do stworzenia aplikacji graficznej 3D. Implementuje on podstawowe mechanizmy niezbędne do renderowania sceny z oświetleniem i cieniami, stanowiąc dobrą bazę do nauki i eksperymentowania z grafiką komputerową w Javie.