Politechnika Świętokrzyska w Kielcach

Informatyka – Grafika Komputerowa

Algorytmy Grafiki Komputerowej

Grupa: 1ID23B

Sebastian Wójcik

Temat Projektu: Symulacja przezroczystości na przykładowej scenie 3D

1) Opis zastosowanych bibliotek i algorytmów :

1.1) Biblioteki:

1.1.1) GLEW - Biblioteka pomagająca w ładowaniu rozszerzeń OpenGL

1.1.2) GLM - Biblioteka matematyczna wykorzystywana w programach OpenGL

1.1.3) stb_image – Część biblioteki stb wykorzystywana do ładowania tekstur z pliku graficznego.

1.1.4) OBJLoader – Biblioteka wykorzystywana do ładowania modeli 3D zapisanych w formacie OBJ

1.2) Algorytmy:

1.2.1) Oświetlenie Blinna – Phonga – To uproszczony model symulacji odbicia zwierciadlanego wykorzystującego wektor połówkowy zamiast wektora odbicia

1.2.2) Specular Map (Mapa rozbłysków) – Czyli zasotoswanie dodatkowej tekstury określającej intensywność i kolor rozbłysków światła na powierzchni obiektu.

1.2.3) Discard Transparency – Przeźroczystość wykonywana poprzez zasotosowanie instrukcji **discard** dzięki, której shader natychmiastowow przerywa swoje działanie. W przypadku mojej pracy zostało to zastosowane do usunięcia koloru tła obiektów trawy w celu symulacji pełnej przeźroczystości.

1.2.4) Blending – Czyli technika symulująca w moim przypadku przeźroczystość poprzez odpowiednie mieszanie barw obiektu „przeźroczystego” oraz obiektów za nim względem kanału alfa. W celu odpowiedniej symulacji przeźroczystości wszystkie obiekty wykorzystujące tą technike są generowane względem odległości, od najdalszej do najbliższej.

2) Opis implementacji z fragmentami kodu źródłowego:

2.1) Główna pętla programu:

```
while (!glfwWindowShouldClose(window)) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(0.07f, 0.13f, 0.17f, 1.0f);

    camera.Input(window);
    camera.UpdateMatrix(45.0f, 0.1f, 100.0f);

    database.DrawDatabase(camera);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Jest to główna pętla programu, która wykonuje czyszczenie buforów kolorów oraz głębokości, a następnie aktualizuje pozycje kamery oraz jej macierz widoku na podstawie uzyskanych informacji od użytkownika. Ostatecznie rysuje wszystkie obiekty na ekranie.

2.2) Bazy programowe:

```
class TextureDatabase {
public:
    vector<Texture> furnitureTextures, wallTextures, lampTextures, bowlTextures, floorTextures, potTextures, shelfTextures, flowersTextures, appleTextures, orangeTextures, imageTextures, frameTextures, bookTe

    TextureDatabase() { ... }
};

class VertexDatabase {
public:
    vector<Vertex> lightVertices, window1Vertices, window2Vertices, window3Vertices, window4Vertices;
    vector<GLuint> lightIndices, windowIndices;

    VertexDatabase() { ... }
};

class ModelDatabase {
public:
    vector<Light> lights;
    vector<Mesh> meshes;
    vector<Window> windows;

    ModelDatabase(){}
    ModelDatabase(VertexDatabase vertexDatabase, TextureDatabase textureDatabase, ShaderUseClass& lightShader, ShaderUseClass& basicShader, ShaderUseClass& windowShader, ShaderUseClass& flowerShader) { ... }
};
```

Jest to zbiór kilku klas, które utworzyłem w celu hierarchicznego utrzymania wszystkich w programie oraz tak by łatwo odnaleźć się w strukturze programu. Każda klasa odpowiada za ładowanie poszczególnych obiektów do programu. Każda klasa zostanie opisana dokładniej ponizej.

2.2.1) Texture Database:

```
TextureDatabase() {
    this->furnitureTextures.push_back(Texture("Textures/tableDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));
    this->furnitureTextures.push_back(Texture("Textures/tableSpecular.jpg", "Specular", 1, GL_RED, GL_UNSIGNED_BYTE, 4));

    this->wallTextures.push_back(Texture("Textures/wallDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));
    this->wallTextures.push_back(Texture("Textures/wallSpecular.jpg", "Specular", 1, GL_RED, GL_UNSIGNED_BYTE, 4));

    this->bowlTextures.push_back(Texture("Textures/bowlDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));
    this->bowlTextures.push_back(Texture("Textures/bowlSpecular.jpg", "Specular", 1, GL_RED, GL_UNSIGNED_BYTE, 4));

    this->floorTextures.push_back(Texture("Textures/floorDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));
    this->floorTextures.push_back(Texture("Textures/floorSpecular.jpg", "Specular", 1, GL_RED, GL_UNSIGNED_BYTE, 4));

    this->potTextures.push_back(Texture("Textures/potDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));
    this->potTextures.push_back(Texture("Textures/potSpecular.jpg", "Specular", 1, GL_RED, GL_UNSIGNED_BYTE, 4));

    this->shelfTextures.push_back(Texture("Textures/shelfDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));
    this->shelfTextures.push_back(Texture("Textures/shelfSpecular.jpg", "Specular", 1, GL_RED, GL_UNSIGNED_BYTE, 4));

    this->flowersTextures.push_back(Texture("Textures/grassDiffuse.png", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));
    this->flowersTextures.push_back(Texture("Textures/grassSpecular.png", "Specular", 1, GL_RED, GL_UNSIGNED_BYTE, 4));

    this->lampTextures.push_back(Texture("Textures/lampDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));

    this->appleTextures.push_back(Texture("Textures/appleDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));

    this->orangeTextures.push_back(Texture("Textures/orangeDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));

    this->imageTextures.push_back(Texture("Textures/imageDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));

    this->frameTextures.push_back(Texture("Textures/frameDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));

    this->bookTextures.push_back(Texture("Textures/bookDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));

    this->windowTextures.push_back(Texture("Textures/windowDiffuse.jpg", "Diffuse", 0, GL_RGBA, GL_UNSIGNED_BYTE, 4));
}
```

Klasa ta zawiera jedynie i jej jedynym zadanie jest załadowanie wszystkich tekstur do programu. Wykonywane jest to poprzez zastosowanie kolejnej klasy o nazwie Texture, która reprezentuje pojedyńczą teksturę.

```
class Texture
{

public:
    GLuint textureID;
    const char* type;
    GLuint unit;
    Texture() {};
    Texture(const char* imagePath, const char* textureType, GLuint slot, GLenum format, GLenum pixelType, GLuint channels) {

        int imgWidth, imgHeight, imgChannels;
        this->type = textureType;

        unsigned char* imgBytes = stbi_load(imagePath, &imgWidth, &imgHeight, &imgChannels, channels);
        if (!imgBytes) {
            cout << "Texture Loading Error!" << endl;
            return;
        }
        this->type = textureType;
        glGenTextures(1, &textureID);
        glActiveTexture(GL_TEXTURE0 + slot);
        this->unit = slot;
        glBindTexture(GL_TEXTURE_2D, textureID);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, imgWidth, imgHeight, 0, format, pixelType, imgBytes);
        glGenerateMipmap(GL_TEXTURE_2D);
        stbi_image_free(imgBytes);
        glBindTexture(GL_TEXTURE_2D, 0);
    }
```

Głównym zadanie tej klasy jest załadowanie tekstury. Jest to wykonywane przez załadowanie pliku ze wskazanej scieżki przy pomocy biblioteki stb_image. Tekstura jest generowana oraz aktywowana jest odpowiednia jednostka teksturująca, a następnie ustawiane są dla niej odpowiednie parametry.

```
void Bind() {
    glActiveTexture(GL_TEXTURE0 + this->unit);
    glBindTexture(GL_TEXTURE_2D, textureID);
}

void Unbind() {
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Teksture, można w dowolnym momencie podłączyć lub odłączyć od wybranego obiektu.

2.2.2) Vertex Database:

```
class VertexDatabase {
public:
    vector<Vertex> lightVertices, window1Vertices, window2Vertices, window3Vertices, window4Vertices;
    vector<GLuint> lightIndices, windowIndices;

    VertexDatabase() {
        this->lightVertices = {
            Vertex{vec3(-0.05f, -0.05f,  0.05f)},
            Vertex{vec3(-0.05f, -0.05f, -0.05f)},
            Vertex{vec3(0.05f,  -0.05f, -0.05f)},
            Vertex{vec3(0.05f,  -0.05f,  0.05f)},
            Vertex{vec3(-0.05f,  0.05f,  0.05f)},
            Vertex{vec3(-0.05f,  0.05f, -0.05f)},
            Vertex{vec3(0.05f,   0.05f, -0.05f)},
            Vertex{vec3(0.05f,   0.05f,  0.05f)}
        };
        this->lightIndices = {
            0, 1, 2,
            0, 2, 3,
            0, 4, 7,
            0, 7, 3,
            3, 7, 6,
            3, 6, 2,
            2, 6, 5,
            2, 5, 1,
            1, 5, 4,
            1, 4, 0,
            4, 5, 6,
            4, 6, 7
        };
        this->window1Vertices = { ... }
```

Klasa ta jest wykorzystywana do załadowania oraz przetrzymywania informacji na temat poszczególnych tablic Vertexów oraz Indexów.

```
struct Vertex {
    vec3 position;
    vec3 normal;
    vec3 color;
    vec2 textureUv;
};
```

Czyli obiektów klasy utworzonych w celu łatwiejszego przechowywania informacji opisujących poszczególne obiekty. Vertex zawiera tak informacje jak pozycja, wektor normalny, kolor czy pozycja tekstury dla obiektu. Klasa ta wykorzystywana jest do tworzenia wszystkich obiektów, które nie są łądowane przy pomocy loadera.

2.2.3) Model Database:

```
class ModelDatabase {
public:
    vector<Light> lights;
    vector<Mesh> meshes;
    vector<Window> windows;

    ModelDatabase(){}
    ModelDatabase(VertexDatabase vertexDatabase, TextureDatabase textureDatabase, ShaderUseClass

        Light light(vertexDatabase.lightVertices, vertexDatabase.lightIndices, &lightShader);
        light.SetPosition(vec3(2.7f, 1.5f, -3.6f));
        lights.push_back(light);

        Mesh table("Obj/table.obj", &basicShader, textureDatabase.furnitureTextures);
        table.SetPosition(vec3(2.3f, 0.2f, -4.2f));
        meshes.push_back(table);

        Mesh bowl("Obj/bowl.obj", &basicShader, textureDatabase.bowlTextures);
        bowl.SetPosition(vec3(2.3f, 0.79f, -4.2f));
        bowl.SetScale(vec3(0.1f, 0.1f, 0.1f));
        meshes.push_back(bowl);

        Mesh apples("Obj/apples.obj", &basicShader, textureDatabase.appleTextures, 0.5f);
        apples.SetPosition(vec3(2.3f, 0.79f, -4.2f));
        apples.SetScale(vec3(0.1f, 0.1f, 0.1f));
        meshes.push_back(apples);

        Mesh oranges("Obj/oranges.obj", &basicShader, textureDatabase.orangeTextures, 0.5f);
        oranges.SetPosition(vec3(2.3f, 0.79f, -4.2f));
        oranges.SetScale(vec3(0.1f, 0.1f, 0.1f));
        meshes.push_back(oranges);

        Mesh chair1("Obj/chair.obj", &basicShader, textureDatabase.furnitureTextures);
        chair1.SetPosition(vec3(1.4f, 0.0f, -4.2f));
        meshes.push_back(chair1);
```

Klasa ta wykorzystywana jest w celu załadowania oraz ustawienie wszystkich modeli, które są ładowane z plików OBJ przy pomocy poprzednio wspomnianego loadera, świateł oraz okien złożonych z klasy **Vertex Database**. Do przechowywania wszystkich informacji na temat obiektów oraz ich renderowania jest wykorzystywana klasa **Mesh** oraz jej odpowiedniki, które różnią się od niej w niewielkim stopniu. Są to **Light** oraz **Window**.

```
class Mesh
{
public:
    ShaderUseClass* shader;
    vector<Vertex> vertices;
    vector<Texture> textures;
    VertexArray vertexMeshArray;

    vec3 position = vec3(0.0f, 0.0f, 0.0f);
    vec3 rotation = vec3(0.0f, 0.0f, 0.0f);
    vec3 scale = vec3(1.0f, 1.0f, 1.0f);
    mat4 modelMatrix = mat4(1.0f);

    Mesh() {};
    ~Mesh() {};

    Mesh(const char* filename,ShaderUseClass* shader ,vector<Texture> newtextures, float textureScale = 5.0f){ ... }

    vector<Vertex> loadOBJ(const char* file_name, float textureScale = 5.0f){ ... }

    void DrawMesh(Camera& camera) {
        this->shader->UseShader();
        this->vertexMeshArray.Bind();
        if (!this->textures.empty()) {

            GLuint textureDiffuseNumber = 0;
            GLuint textureSpecularNumber = 0;

            for (int i = 0; i < textures.size(); i++) {
                string textureName;
                string textureType = textures[i].type;
                if (textureType == "Diffuse") {
                    textureName = to_string(textureDiffuseNumber++);
                }
                else if (textureType == "Specular") {
                    textureName = to_string(textureSpecularNumber++);
                }
                textures[i].textureUnit(*shader, ("fragTexture" + textureType + textureName).c_str(), i);
                textures[i].Bind();
            }
        }
        glUniform3f(glGetUniformLocation(shader->shaderID, "fragCameraPosition"), camera.position.x, camera.position.y, camera.position.z);
        camera.Matrix(*shader, "vertCamMatrix");
        glDrawArrays(GL_TRIANGLES, 0, this->vertices.size());
    }
```

Klasa **Mesh** zawiera takie informacje jak np. wykorzystywany shader przez wskazany obiekt, zbiór Vertexów określających kształt obiektu, zbiór tekstur nakładanych na obiekt czy jego pozycja, rotacja lub skala. Przedstawiona jest natomiast metoda odpowiadająca za rysowanie obiektu. Przyjmuje ona obiekt kamery względem, której będą wykonywane przeliczenia jej matrycy widoku. Uruchamiany jest shader, a następnie podpinana jest tablica wierzchołków **VAO** obiektu oraz sprawdzane jest czy obiekt posiada teksturę. Jeżeli tak to sprawdza czy jest to tekstura typu **Speucalr** czy **Diffuse**, a następnie podpina ją do odpowiedniej jednostki teksturujacej i przekazuje do shadera. Ostatecznie wykonuje przeliczenia względem obecnej pozycji kamery oraz rysuje obiekt.

2.2.4) Database:

```
class Database {
public:
    TextureDatabase textureDatabase;
    VertexDatabase vertexDatabase;
    ModelDatabase modelDatabase;

    map<float, Window> drawOrder;

    Database(ShaderUseClass& lightShader, ShaderUseClass& basicShader, ShaderUseClass& windowShader, ShaderUseClass& flowerShader) {
        this->textureDatabase = TextureDatabase();
        this->vertexDatabase = VertexDatabase();
        this->modelDatabase = ModelDatabase(this->vertexDatabase, this->textureDatabase, lightShader, basicShader, windowShader, flowerShader);
    }

    void DrawDatabase(Camera camera) {
        for (int i = 0; i < modelDatabase.lights.size(); i++) {
            modelDatabase.lights[i].SetUniform();
            modelDatabase.lights[i].DrawLight(camera);
        }
        for (int i = 0; i < modelDatabase.meshes.size(); i++) {
            modelDatabase.meshes[i].SetUniform(modelDatabase.lights[0]);
            modelDatabase.meshes[i].DrawMesh(camera);
        }
        for (int i = 0; i < modelDatabase.windows.size(); i++) {
            drawOrder.insert({length(camera.position - modelDatabase.windows[i].position), modelDatabase.windows[i]});
        }
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        for (auto i = drawOrder.rbegin(); i != drawOrder.rend(); i++) {
            i->second.SetUniform(modelDatabase.lights[0]);
            i->second.DrawWindow(camera);
        }
        drawOrder.clear();
        glDisable(GL_BLEND);
    }
};
```

Database jest klasą łączącą wszystkie powyższe klasy w jedną całość. Przechowuje ona informacje na temat wszystkich poprzednio wspomniaych zbiorach oraz dodatkowo pozwala na renderowanie wszystkiego w przestrzeni. Odpowiada za to metoda **DrawDatabase**, która kolejno generuje obiekty znajdujące się w bazie danych modeli z już narzuconymi teksturami. Dodatkowo w procesie generowania obiektów przeźroczystych (w tym wypadku są to obiekty klasy **Window**) wykonuje ona sortowanie obiektów wzlędem odległości od najdalszego do najbliższego z wykorzystaniem mechanizmów zbioru mapy. Pozwala to na jednoznaczne sprawdzenie odległości poszczególnych okien względem pozycji kamery, a następnie odpowiednie renderowanie ich od najdalszego do najbliższego.

2.3) Kamera:

```
void Matrix(ShaderUseClass& shader, const char* uniform) {
    glUniformMatrix4fv(glGetUniformLocation(shader.shaderID, uniform), 1, GL_FALSE, value_ptr(this->cameraMatrix));
}

void UpdateMatrix(float fovDegrees, float near, float far) {
    mat4 view = mat4(1.0f);
    mat4 projection = mat4(1.0f);

    view = lookAt(this->position, this->position + this->orientation, this->upVector);
    projection = perspective(radians(fovDegrees), (float)(this->width / this->height), near, far);

    this->cameraMatrix = projection * view;
}

void Input(GLFWwindow* window)
{
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    {
        this->position += speed * this->orientation;
    }
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    {
        this->position += speed * -glm::normalize(glm::cross(this->orientation, this->upVector));
    }
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {
        this->position += speed * -this->orientation;
    }
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    {
        this->position += speed * glm::normalize(glm::cross(this->orientation, this->upVector));
    }

    if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS) { ... }
    else if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_RELEASE) { ... }
}
```

Kamera **(Camera)** jest obiektem, które reprezentuje widok z oka obserwatora. Występuje tu parę metod, które pozwalają na przekazanie informacji odnośnie obecnej pozycji do shadera **(Matrix)**. **UpdateMatrix**, która jest wykorzystywana w celu odpowiedniego ustawienia kamery względem obecnie występujących ustawień oraz **Input**, które powoduje odpowiednią modyfikację parametrów kamery na podstawie wykorzystanych przycisków. Sterowanie kamerą jest dokładniej opisane w dokumentacji użytkowania oprogramowania.

2.4) IBO, VBO oraz VAO:

Klasy te wykorzystywane są do generowania odpowiednich elementów opisujących obiekty w przestrzeni środowiska OpenGL.

2.4.1) Index Buffer:

```
class IndexBuffer
{

public:
    GLuint ibo;
    IndexBuffer(vector<GLuint>& indices) {
        glGenBuffers(1, &this->ibo);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->ibo);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(GLuint), indices.data(), GL_STATIC_DRAW);
    }

    void Bind() {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->ibo);
    }

    void Unbind() {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }

    void Delete() {
        glDeleteBuffers(1, &this->ibo);
    }
};
```

Klasa ta jest wykorzystywana do generowania bufora indeksów wierzchołków na podstawie uzyskanego zbioru wartości.

2.4.2) Vertex Buffer:

```
class VertexBuffer
{

public:
    GLuint vbo;
    VertexBuffer(vector<Vertex> &vertices) {
        glGenBuffers(1, &this->vbo);
        glBindBuffer(GL_ARRAY_BUFFER, this->vbo);
        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), vertices.data(), GL_STATIC_DRAW);

    }

    void Bind() {
        glBindBuffer(GL_ARRAY_BUFFER, this->vbo);
    }

    void Unbind() {
        glBindBuffer(GL_ARRAY_BUFFER, 0);
    }

    void Delete() {
        glDeleteBuffers(1, &this->vbo);
    }
};
```

Klasa ta jest wykorzystywana do generowania bufora z tabelą współrzędnych na podstawie uzyskanego zbioru Vertexów. Używany jako atrybuty wierzchołków prymitywów.

2.4.3) Vertex Array:

```
class VertexArray
{

public:
    GLuint vao;
    VertexArray() {
        glGenVertexArrays(1, &this->vao);
    }

    void LinkAttribute(VertexBuffer& vbo, GLuint layout, GLuint numComponents, GLenum type, GLsizeiptr stride, void* offset) {
        vbo.Bind();
        glVertexAttribPointer(layout, numComponents, type, GL_TRUE, stride, offset);
        glEnableVertexAttribArray(layout);
        vbo.Unbind();
    }
    void Bind() {
        glBindVertexArray(this->vao);
    }
    void Unbind() {
        glBindVertexArray(0);
    }

    void Delete() {
        glDeleteVertexArrays(1, &this->vao);
    }
};
```

Klasa ta jest wykorzystywana do generowania struktury przechowywującej bufor tablicy **VBO** wraz z danymi wierzchołków prymitywów, takimi jak ich współrzędne, kolor czy normalne.

2.5) Klasy obsługujące Shadery:

Są to klasy pozwalające na tworzenie oraz obsługę obiektów shaderów.

2.5.1) Shader Load Class:

```
static class ShaderLoadClass
{
public:
    static string readFile(const string& filePath) {
        ifstream fs(filePath, ios::in);
        stringstream buffer;
        string line;
        while (getline(fs, line)) {
            buffer << line << "\n";
        }
        fs.close();
        return buffer.str();
    }

    static GLuint compileShader(GLuint type, const string& source)
    {
        GLuint hShader = glCreateShader(type);
        GLint result;
        const char* src = source.c_str();

        glShaderSource(hShader, 1, &src, nullptr);
        glCompileShader(hShader);
        glGetShaderiv(hShader, GL_COMPILE_STATUS, &result);

        return hShader;
    }
};
```

Jest to klasa statyczna pozwalająca na załadowanie zawartości shadera z pliku oraz jego przekompilowania względem wskazanego typu.

2.5.2) Shader Use Class:

```
class ShaderUseClass
{
public:
    GLuint shaderID;
    ShaderUseClass(const string& vertexShader, const string& fragmentShader) {
        this->CreateShader(vertexShader, fragmentShader);
    }
    void CreateShader(const string& vertexShader, const string& fragmentShader) {
        GLint result;
        GLuint fs = ShaderLoadClass::compileShader(GL_FRAGMENT_SHADER, fragmentShader);
        GLuint vs = ShaderLoadClass::compileShader(GL_VERTEX_SHADER, vertexShader);
        if (vs == 0 || fs == 0) {
            return;
        }
        this->shaderID = glCreateProgram();
        glAttachShader(shaderID, vs);
        glAttachShader(shaderID, fs);
        glLinkProgram(shaderID);
        glGetShaderiv(shaderID, GL_COMPILE_STATUS, &result);
        glValidateProgram(shaderID);
        glDetachShader(shaderID, vs);
        glDetachShader(shaderID, fs);
    }

    void UseShader() {
        glUseProgram(shaderID);
    }

    void DeleteShader() {
        glDeleteProgram(shaderID);
    }
};
```

Shader Use Class jest natomiast klasą pozwalającą na obsługę utworzonego już shadera na rzecz programu. Na samym początku tworzy on obiekt Shadera na podstawie przekazanych scieżek do plików oraz pośrednio wykorzystując klasę **Shader Load Class.** Shadery są kompilowane oraz określany jest ich typ, a następnie wykonywany jest cały proces polegający na utworzeniu dla nich programu, podpięciu ich oraz zwalidowaniu. W dowolnym momęcie, można wywołać funkcję pozwalającą na uzycie wskazanego shadera.

2.6) Shadery:

Wszystkie poszczególne shadery są względem siebie bardzo podobne, różnią się tylko i wyłącznie szczegółami, dlatego też przedstawiona oraz opisana będzie jedynie struktura pojedynczego z nich, a następnie zaprezentowane zostaną elementy wyróżniające każde z nich.

2.6.1) Vertex Shader:

```
#version 460 core

out vec3 fragCurrentPosition;
out vec3 fragNormals;
out vec3 fragColor;
out vec2 fragTexture;

layout (location = 0) in vec3 vertPosition;
layout (location = 1) in vec3 vertNormals;
layout (location = 2) in vec3 vertColor;
layout (location = 3) in vec2 vertTexturePosition;

uniform mat4 vertCamMatrix;
uniform mat4 vertModel;

void main(){
    fragCurrentPosition = vec3(vertModel * vec4(vertPosition, 1.0f));
    gl_Position = vertCamMatrix * vec4(fragCurrentPosition, 1.0f);

    fragColor = vertColor;
    fragTexture = vertTexturePosition;
    fragNormals = vertNormals;
};
```

Głównym zadanie, każdego z tego typu shaderów w przypadku mojego kodu jest przyjęcie poszczególnych wartości opisujących pozycje modeli na scenie oraz przeliczenie ich pozycji względem oka obserwatora. Dodatkowo przekazuje ona kolejne wartości do **fragment** shadera.

2.6.2) Fragment Shader:

```
#version 460 core

out vec4 FragColor;

in vec3 fragCurrentPosition;
in vec3 fragNormals;
in vec3 fragColor;
in vec2 fragTexture;

uniform sampler2D fragTextureDiffuse0;
uniform sampler2D fragTextureSpecular0;
uniform vec4 fragLightColor;
uniform vec3 fragLightPos;
uniform vec3 fragCameraPosition;

vec4 PointLight(float ambientValue, float specularValue, float a, float b){
    vec3 lightVector = fragLightPos - fragCurrentPosition;
    float distance = length(lightVector);
    float intensity = 1.0f / (a * pow(distance,2) + b * distance + 1.0f);
    float ambient = ambientValue;

    vec3 normalizedNormals = normalize(fragNormals);
    vec3 lightDirection = normalize(fragLightPos - fragCurrentPosition);
    float diffuse = max(dot(normalizedNormals, lightDirection), 0.0f);

    float specular = 0.0f;
    if(diffuse != 0.0f){
        float specularLight = specularValue;
        vec3 viewDirection = normalize(fragCameraPosition - fragCurrentPosition);
        vec3 halfLength = normalize(viewDirection + lightDirection);
        vec3 reflectionDirection = reflect(-lightDirection, normalizedNormals);
        float specAmount = pow(max(dot(normalizedNormals, halfLength), 0.0f), 32);
        specular = specAmount * specularLight;
    };
    return (texture(fragTextureDiffuse0, fragTexture) * (diffuse * intensity + ambient ) + texture(fragTextureSpecular0, fragTexture).r * specular * intensity) * fragLightColor;
}

void main(){
    FragColor = PointLight(0.2f, 0.5f, 1.0f, 0.5f);
};
```

W progmie przy pomocy shadera został zaimplementowany algorytm cieniowania Blinna – Phonga z dodatkowym zastosowaniem map rozbłysku.

2.6.3) Pixel Discard:

```
if(texture(fragTextureDiffuse0, fragTexture).a < 0.1f){
    discard;
}
```

W przypadku roślin została wykorzystana funkcja shaderów pozwalająca na porzucenia piksela w procesie jego renderowania. Z tego też powodu wskazane piksele są usuwane jeżeli ich współczynnik alfa jest mniejszy niż wskazana wartość.

2.6.4) Kolorowanie wynikowego obiektu:

```
void main(){
    FragColor = PointLight(0.2f, 0.5f, 1.0f, 0.5f) * vec4(fragColor, 0.8f);
};
```

Jeżeli chodzi o okna to w celu lepszego zaprezentowania procesu blendingu, wynikowy kolor obiektu z narzuconymi efektami oświetlenia jest dodatkowo kolorowany na wskazany kolor ze współczynnikiem alfa równym wybranej wartości.

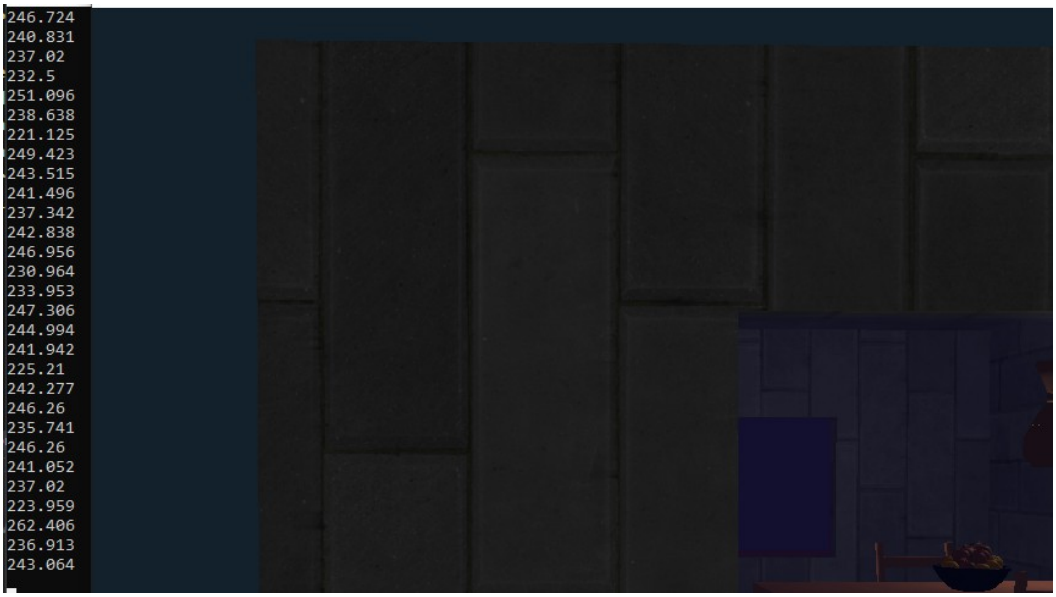
3) Testy wydajnościowe oraz jakościowe:

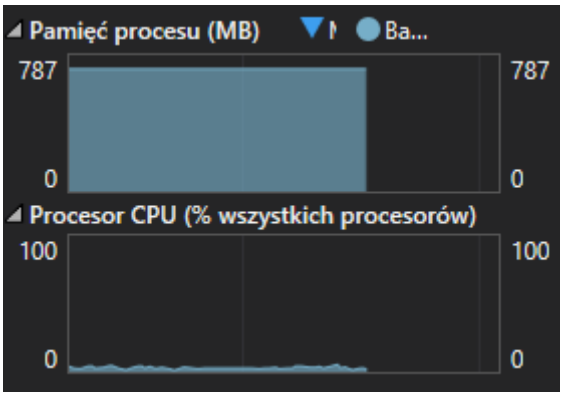
W moim przypadku w procesie testowania wykorzystałem narzędzia wbudowane w programie Visual Studio. Dzięki nim mogłem przetestować program pod kątem obciążenia karty graficznej oraz procesora. Dodatkowo zaimplementowałem funkcję pozwalającą na mierzenie ilości klatek na sekundę.

3.1) Testy programu (Wartości bazowe):

W celu policzenia ilości klatek na sekundę zastosowałem poniższą funkcję. Pozwala ona na policzeniu ilości czasu, który upłynął pomiędzy odświeżeniami programu, a następnie na przeliczenie tej wartości na ilość klatek. Poniżej zdjęcie kodu, screen z osiąganymi wartościami FPS (Ekran ma ustawione odświeżanie 240Hz, stąd też taki wynik) na moim komputerze oraz wykorzystane ilości pamięci na procesorach (GeForce RTX 2060 oraz Ryzen 5 3600).

```
static float CountFps(float lastTime) {
    float currTime = static_cast<float>(glfwGetTime());
    float dt = currTime - lastTime;
    float newLastTime = currTime;
    float fps = 1 / dt;
    std::cout << fps << std::endl;
    return newLastTime;
}
```





3.2) Testy programu (Pod obciążeniem):

W celu dodatkowego przetestowania programu obciążyłem go znacznie większą ilością obiektów (Dokładnie 10 razy więcej). Jak można zaobserwować na poniższym zdjęciu, na którym zawarłem od razu wszystkie możliwe wartości pomiarów. Program mimo większego wykorzystanjia pamięci karty graficznej, dalej działa identycznie.