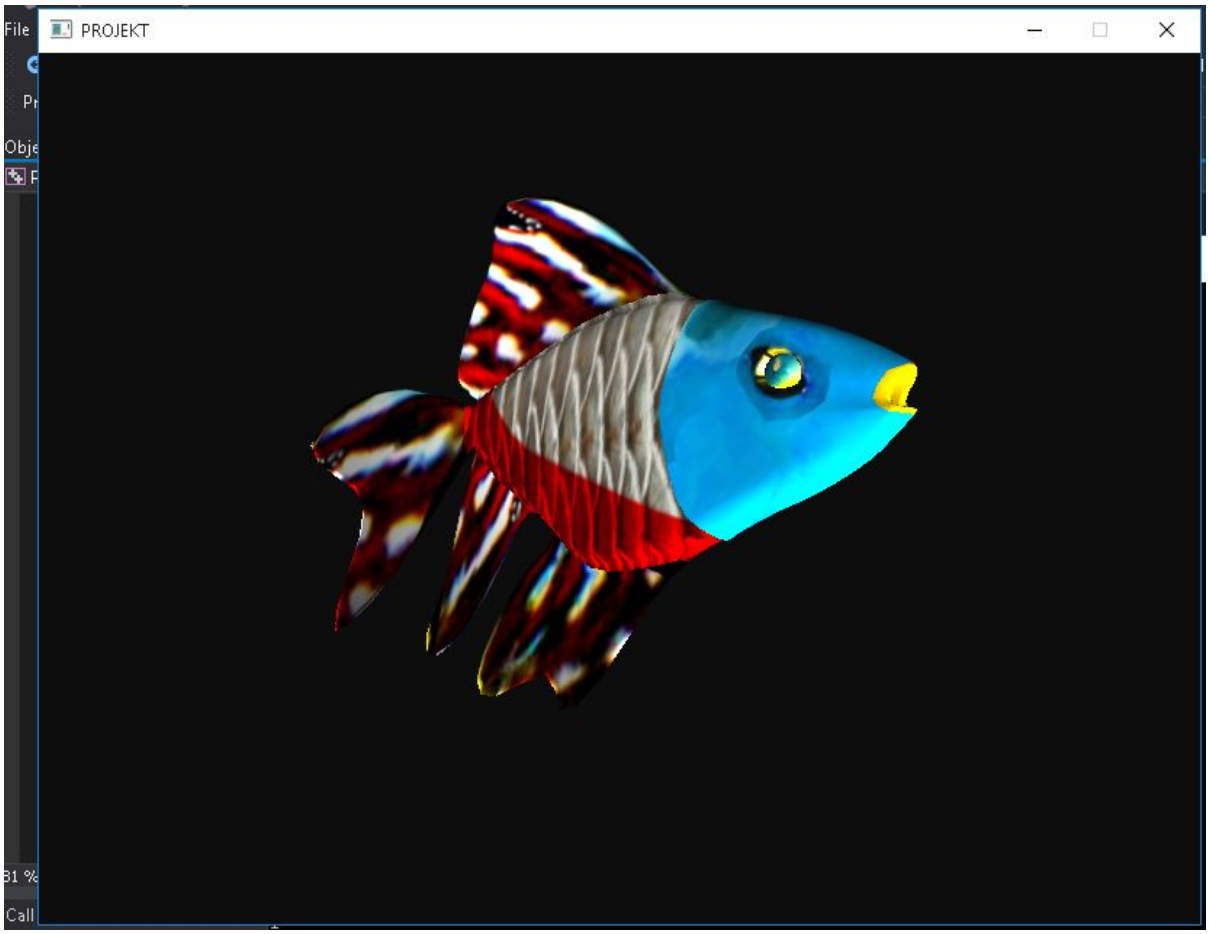
Projekt GRK

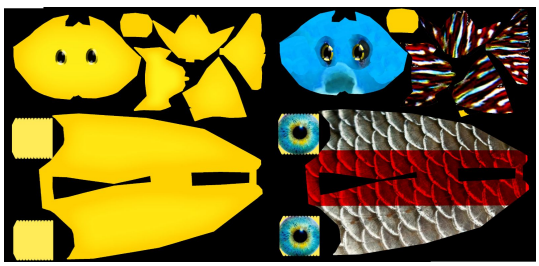
Student: Sergiusz Jańczura

indeks 396338



Opis:

rybka na nasz rozkaz pływa po kwadracie, w momencie gdy płynie po prostej rotuje w lewo i w prawo (kieruje się w lewo i w prawo, 3 razy zmienia swój kąt patrzenia na jednej prostej). Rybka została pobrana ze strony z darmowymi obiektami 3d w formacie .obj. Zmieniłem jej tekstury (wcześniej była złota).



Używane biblioteki:

```
#include "Dependencies\glew\glew.h"
#include "Dependencies\soil\SOIL.h"
#include <iostream>
#include <GLFW\glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <string>
```

Klasy:

```
#include "Shader.h"
#include "Camera.h"
#include "Model.h"
```

Klasa Shader odpowiedzialna jest za wczytanie shaderów (vertex i fragment) z plików, kompilowane one są, zostają połączone z programem i usunięte.

```
this->Program = glCreateProgram();
        glAttachShader(this->Program, vertex);
        glAttachShader(this->Program, fragment);
        glLinkProgram(this->Program);

glDeleteShader(vertex);
glDeleteShader(fragment);
```

Na końcu zdefiniowana jest funkcja Use która jest wywoływana w głównym pliku projektu. Obiekt tej klasy tworzony jest poprzez podanie w argumencie ścieżek. Na przykład:

```
Shader shader("C:/Users/Ser-LT/Desktop/PROJEKT/Project1/VertexShader.vs",
"C:/Users/Ser-LT/Desktop/PROJEKT/Project1/FragmentShader.frag");
```

Klasa Camera odpowiedzialna jest za operacje związane z kamerą. Dzięki niej możemy poruszać się po naszej scenie, obracać myszką czy latać.
Funkcja

```
ProcessKeyboard(Camera_Movement direction, GLfloat deltaTime);
```

jest odpowiedzialna za ruch wywoływany poprzez klawiaturę a

```
void Fly(int q, GLfloat deltaTime);
```

za latanie w górę i w dół.
Zdefiniowane są tam również wartości kamery takie jak czułość czy zoom, szybkość.

Klasa Model odpowiada za załadowanie gotowych obiektów .obj stworzonych z programów typu blender. Zawiera ona w sobie **klasę Mesh**. Ta zaś odpowiada za załadowanie samych siatek obiektów trójwymiarowych (duża ilość wierzchołków i łączenia). Obiekt klasy Model tworzymy podając w argumencie ścieżkę do pliku .obj.

Model zlotaRybka("C:/Users/Ser-LT/Desktop/PROJEKT/Project1/Models/GoldenFish_OBJ/Golden_Fish_OBJ.obj");
Istotną funkcją jest *Draw(Shader shader);*.dzięki której nasze obiekty pojawią się na scenie i jest ona wywoływana w głównej pętli programu.

Source.cpp (main)

Na samym początku programu tworzymy okno. Z pomocą przychodzi biblioteka GLFW. Dokładniejszy opis tworzenia okienka możemy znaleźć w dokumentacji tu http://www.glfw.org/docs/latest/window.html#window_hints
Uważam, że nie ma sensu tego dogłębnie omawiać.

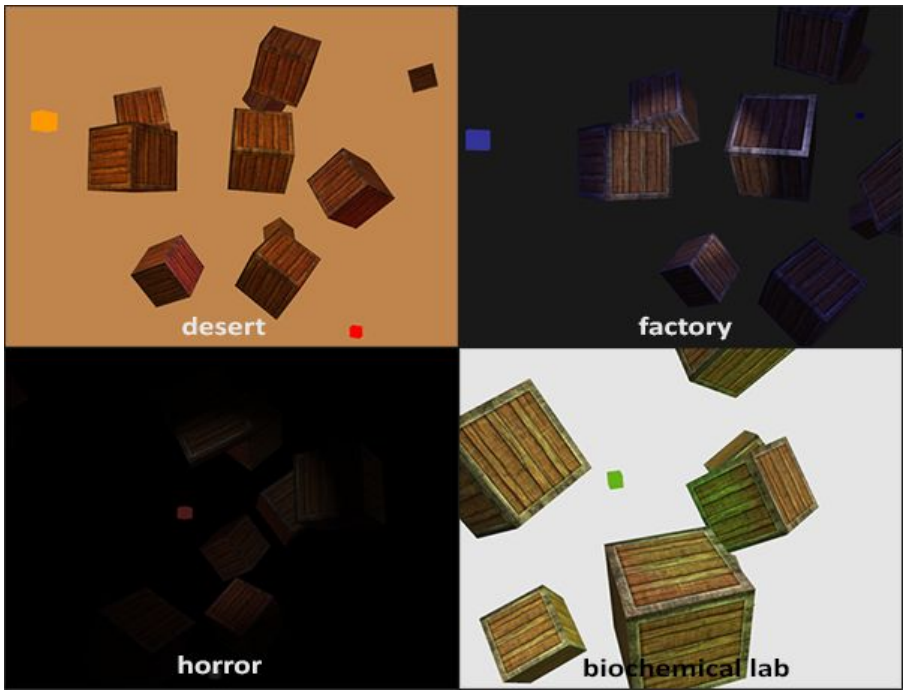
glViewport(0, 0, screenWidth, screenHeight);
funkcja ta służy do dynamicznej modyfikacji obszaru renderingu

glEnable(GL_DEPTH_TEST);
funkcja MUST-HAVE dla programów w technologii 3D, przysłania elementy które są za innymi elementami - powoduje, że nie widzimy np. drzewa posadzonego za budynkiem. W naszym przypadku byśmy widzieli jedną ‘ścianę’ rybki z zewnątrz oraz przeplatającą się wewnętrzną drugą ‘ścianką’.
Istnieją również inne funkcje pomocnicze np *glEnable(GL_POLYGON_SMOOTH)* ale z nich nie korzystałem.

glm::vec3 pointLightPositions[] =
Tu zawarte są wierzchołki oświetlenia

while (!glfwWindowShouldClose(window))
jest główną pętlą gry i trwa dopóki okno programu nie zostanie zamknięte.
W tym miejscu obliczany jest wszelki ruch dziejący się na scenie, ustalany jest kolor tła, czyszczone są zawartości buforów funkcją *glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);*
Nadawane są natężenia dla poszczególnych świateł, oraz ich pozycje na przykład dla światła otoczenia o indeksie 0:
glUniform3f(glGetUniformLocation(shader.Program, "pointLights[0].ambient"), 0.05f, 0.05f, 0.05f);
Inne rodzaje oświetleń:
rozproszone
glUniform3f(glGetUniformLocation(shader.Program, "pointLights[0].diffuse"), 1.0f, 1.0f, 1.0f);
odbijane
glUniform3f(glGetUniformLocation(shader.Program, "pointLights[0].specular"), 1.0f, 1.0f, 1.0f);
Te trzy rodzaje tworzą oświetlenie phonga (ale nie są to jedyne wartości którymi możemy operować by zmieniać oświetlenie) i są przekazywane do shaderów.
Podając różne wartości w argumentach np 0.05f, 1.0f, 1.0f spowodowalibyśmy zmianę barwy światła na przykład na bardziej niebieską.

Dzięki temu możemy eksperymentować i nadawać naszym scenom pewien unikalny klimat jak na obrazkach obok (obrazek ze strony podanej na końcu dokumentacji).



Wykonywane są translacje, skalowania oraz rotacje wczytanych modeli np.:
```
model = glm::translate(model, glm::vec3(rybx, ryby, rybz));
model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
model = glm::rotate(model, 20.0f, glm::vec3(0.6f, 0.1f, 0.3f));
```
i przekazywane są do shaderów po czym wyświetlane na ekranie.

```
glBindVertexArray(0);
```
‘sprząta’ po nas

później następuje jeszcze [podwójne buforowanie](#)
```
glfwSwapBuffers(window);
```

na koniec (już poza główną pętlą programu) `glfwTerminate()` niszczy wszystkie okienka i zwalnia zasoby.

FragmentShader

Istotnym elementem tutaj jest przedewszystkim funkcja
vec3 CalcPointLight(PointLight light, Material mat, vec3 normal, vec3 fragPos, vec3 viewDir);
funkcja ta bierze odpowiednie argumenty i zwraca vec3 reprezentujacy dane światło/barwę na danym fragmencie:
return (ambient + diffuse + specular);

```
for(int i = 0; i < NR_POINT_LIGHTS; i++)
        result += CalcPointLight(pointLights[i], material, norm, fragPosition, viewDir);

color = vec4(result, 1.0f);
```

Taki rodzaj zapisania tego kodu w ten sposób ma zaletę - w przypadku wielu źródeł światła nie dublujemy kodu.

Problem 1:

zmiana scen została zaimplementowana na dwóch przyciskach ponieważ na jednym sceny za szybko przeskakiwały i nie byliśmy w stanie obejrzeć ich po kolei. Dzięki rozwiązaniu przedstawionym poniżej działać będzie tylko jeden klawisz : albo F1 albo F2 i oba mają tę samą funkcję - zmieniają wartość zmiennej *scene.* Klawisze te przestają działać po pojedynczym zarejestrowaniu kliknięcia.

```
if (keys[GLFW_KEY_F1] && switcher==0)
   {
        if (switcher == 0)
        {
                switcher = 1;
        }
        else switcher = 0;
        scene++;
        if (scene > maxscenes)
                scene = 1;
   }
   if (keys[GLFW_KEY_F2] && switcher == 1)
   {
        if (switcher == 0)
        {
                switcher = 1;
        }
        else switcher = 0;
        scene++;
        if (scene > maxscenes)
                scene = 0;
   }
```

Problem 2:

rybka pływa po kwadracie w związku z czym mamy 4 różne przypadki płynięcia:
1* dodajemy do zmiennej x: *rybx = 1.0f + 0.01f*ile;*
2* dodajemy do zmiennej z: *rybz = -5.0f + 0.01f*ile;*
3* odejmujemy od zmiennej x: *rybx = lastx - 0.01f*ile;*
4* odejmujemy od zmiennej z: *rybz = lastz - 0.01f*ile;*
Zmienne lastx i lastz są to wartości rybx i rybz po tych 3000 przejściach. Gdyby ich nie było byłby generowany dodatkowy problem….
Przypadki zmieniają się co 3000 przejść głównej pętli gry, wartość ta jest obliczana w zmiennej *ile*.
Zmienne *rybx* i *rybz* używane są później do translacji przed jej wyświetlaniem.
model = glm::translate(model, glm::vec3(rybx, ryby, rybz));
Rybka gdy płynie po linii prostej rotuje. Odpowiedzialna jest za to zmienna *rotater* która zmienia swoje wartości.

Sterowanie:

W,A,S,D - ruch
Spacja - lot w górę
Shift - lot w dół
+myszka - obrót kamerą
9 i 0 - włączenie/wyłączenie pływania rybki
F1 i F2 zmiana scen (rybka, miasto oraz miasto przeskalowane)
ESC - wyłączenie
Scroll od myszki - zoom

Modele pobierane z:
http://tf3dm.com/
http://www.turbosquid.com/index.cfm

Projekt stworzony w oparciu o tutorial ze strony
http://learnopengl.com/
z działów **Getting Started**, **Lighting** (Multiple Lights) oraz **Model Loading**.