POLITECHNIKA RZESZOWSKA
im. IGNACEGO ŁUKASIEWICZA

WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Dokumentacja projektu z przedmiotu Grafika Komputerowa

Microcar Flex Furgon

Cezary Bober
Michał Kruczek

Rzeszów, 2019

Spis treści

1 Opis projektu

Celem projektu było stworzenie interaktywnego modelu prostego samochodu korzystając z biblioteki OpenGL. Dzięki wykonaniu projektu poszerzyliśmy naszą wiedzę dotyczącą zarówno tworzenia modeli 3D jak i z pozostałych zagadnień składających się na dziedzinę jaką jest grafika komputerowa.

Stworzony pojazd z pozwala na jazdę do przodu oraz do tyłu, zaimplementowane zostały także pewne elementy interaktywne takie jak otwieranie i zamykanie klapy bagażnika oraz maski. Samochód posiada także kręcący się wał napędowy napędzający potężny silnik.

1.1 Wykorzystane technologie

Projekt został napisany przy wykorzystaniu języka C++ w wersji 11. Przy tworzeniu projektu wykorzystywaliśmy bibliotekę graficzną *OpenGL*. Oprócz niej wykorzystana została biblioteka *OpenGL Mathematics*, która pozwala na proste wykorzystanie bardziej skomplikowanych narzędzi aparatu matematycznego. Kolejną biblioteką była *GLFW*, która zapewnia proste API do tworzenia okien, kontekstów i powierzchni, odbierania danych wejściowych i zdarzeń. Załadowanie tych bibliotek było możliwe dzięki bibliotece *OpenGL Extension Wrangler Library*, która zapewnia wydajne mechanizmy uruchamiania w celu określenia, które rozszerzenia OpenGL są obsługiwane na platformie docelowej.

Modele poszczególnych części samochodu zostały stworrzone w narzędziu *Blender*, który jest otwartym oprogramowaniem do modelowania i renderowania obrazów oraz animacji trójwymiarowych. Następnie dzięki bibliotece *Open Asset Import Library* zostały one umieszczone w programie.

2 Inspiracja

Inspiracją do stworzenia omawianego modelu samochodu był posiadany przez jednego z autorów projektu samochód *Microcar MC1* przedstawiony na zdjęciu poniżej.

Umieszczona w projekcie wersja jest modelem zmodyfikowanym na potrzeby handlu towarowego. Niewielkie rozmiary i mała waga powodują, że jest to bardzo dobry pojazd do ciasnych i zatłoczonych miast.

Rysunek 1: Microcar MC1 *"Szerszeń"*

Rysunek 2: Microcar Flex Furgon

3 Powstawanie projektu

3.1 Plik główny

Main.cpp

```
1 #pragma once
2
3 #include <stdio.h>
```

```
#include <string.h>
#include <math.h>
#include <vector>
#include <sstream>
#include <GL\glew.h>
#include "Window.h"

GLfloat deltaTime = 0.0f;
GLfloat lastTime = 0.0f;
GLfloat timeCounter = 0.0f;
int frameCount = 0;

static const char* vShader = "Shaders/shader.vert";
static const char* fShader = "Shaders/shader.frag";

void computeFPS(GLFWwindow* pWindow);

int main()
{
	Window mainWindow(1200, 800);
	mainWindow.Initialize();

	while (!mainWindow.getShouldClose())
	{
		computeFPS(mainWindow.mainWindow);

		glfwPollEvents();

		glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
		glClear(GL_COLOR_BUFFER_BIT |
		 ↪  GL_DEPTH_BUFFER_BIT);

		glUseProgram(0);
		mainWindow.swapBuffers();
	}

	return 0;
}
```

Pierwszym plikiem jaki został stworzony podczas tworzenia programu był plik Main.cpp wraz z główną funkcją. Funkcja ta jest samym centrum programu. W pętli głównej będą wywoływane główne funkcje odpowiedzialne za obsługę zdarzeń, obsługę shaderów oraz renderowanie poszczególnych elementów.

```
GLfloat deltaTime = 0.0f;
GLfloat lastTime = 0.0f;
GLfloat timeCounter = 0.0f;
```

```
4  int frameCount = 0;
5
6  static const char* vShader = "Shaders/shader.vert";
7  static const char* fShader = "Shaders/shader.frag";
8
9  void computeFPS(GLFWwindow* pWindow)
10 {
11         GLfloat now = glfwGetTime();
12         deltaTime = now - lastTime;
13         lastTime = now;
14         timeCounter += deltaTime;
15         frameCount++;
16
17         if (timeCounter >= 1.0) {
18                 double fps = double(frameCount) / timeCounter;
19
20                 std::stringstream ss;
21                 ss << " [" << fps << " FPS]";
22
23                 glfwSetWindowTitle(pWindow, ss.str().c_str());
24
25                 frameCount = 0;
26                 timeCounter = 0;
27         }
28 }
```

Pierwszą funkcje jaka została zainmplementowana w programie to był licznik kaltek na sekundę, który następnie został umieszczony w tytule okna. Dzięki niemu w łatwy sposób mogliśmy określić wydajność naszej aplikacji. Licznik ten działa w oparciu o ilość czasu który upłynął pomiędzy kolejnymi iteracjami programu.

Aby uniknąć nadmiarowej ilośći kodu w głownym pliku aplikacji, całość kodu odpowiedzialnego za inicjalizacje oraz konfigurację okna został umieszczony w osobnym pliku o nazwie Window.cpp

```
1  #include "Window.h"
2
3  Window::Window()
4  {
5          width = 800;
6          height = 600;
7          xChange = 0;
8          yChange = 0;
9          mouseFirstMoved = true;
10         for (size_t i = 0; i < 1024; i++)
11         {
12                 keys[i] = false;
13         }
```

```
14  }
15
16  Window::Window(GLint windowWidth, GLint windowHeight)
17  {
18          width = windowWidth;
19          height = windowHeight;
20          xChange = 0;
21          yChange = 0;
22          mouseFirstMoved = true;
23          for (size_t i = 0; i < 1024; i++)
24          {
25                  keys[i] = false;
26          }
27  }
28
29  int Window::Initialize()
30  {
31          if (!glfwInit())
32          {
33                  printf("Error Initialising GLFW");
34                  glfwTerminate();
35                  return 1;
36          }
37
38          glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
39          glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
40
41          glfwWindowHint(GLFW_OPENGL_PROFILE,
            ↪  GLFW_OPENGL_CORE_PROFILE);
42
43          glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
44
45          glfwWindowHint(GLFW_REFRESH_RATE, 60);
46
47          const GLFWvidmode* mode =
            ↪  glfwGetVideoMode(glfwGetPrimaryMonitor());
48      glfwGetPrimaryMonitor(), NULL);
49          mainWindow = glfwCreateWindow(width, height, "Microcar
            ↪  Flex Furgon", NULL, NULL);
50          width = mode->width;
51          height = mode->height;
52          if (!mainWindow)
53          {
54                  printf("Error creating GLFW window!");
55                  glfwTerminate();
56                  return 1;
57          }
58
```

```
59          glfwGetFramebufferSize(mainWindow, &bufferWidth,
            ↪ &bufferHeight);
60
61          glfwMakeContextCurrent(mainWindow);
62
63          createCallbacks();
64          glfwSetInputMode(mainWindow, GLFW_CURSOR,
            ↪ GLFW_CURSOR_DISABLED);
65
66          glewExperimental = GL_TRUE;
67
68          GLenum error = glewInit();
69          if (error != GLEW_OK)
70          {
71                  printf("Error: %s", glewGetErrorString(error));
72                  glfwDestroyWindow(mainWindow);
73                  glfwTerminate();
74                  return 1;
75          }
76
77          glEnable(GL_DEPTH_TEST);
78
79          glViewport(0, 0, bufferWidth, bufferHeight);
80
81          glfwSetWindowUserPointer(mainWindow, this);
82  }
83
84  void Window::createCallbacks()
85  {
86          glfwSetKeyCallback(mainWindow, handleKeys);
87          glfwSetCursorPosCallback(mainWindow, handleMouse);
88  }
89
90  GLfloat Window::getXChange()
91  {
92          GLfloat theChange = xChange;
93          xChange = 0.0f;
94          return theChange;
95  }
96
97  GLfloat Window::getYChange()
98  {
99          GLfloat theChange = yChange;
100         yChange = 0.0f;
101         return theChange;
102 }
103
```

```
104 void Window::handleKeys(GLFWwindow* window, int key, int code, int
    ↪ action, int mode)
105 {
106         Window* theWindow =
            ↪ static_cast<Window*>(glfwGetWindowUserPointer(window));
107
108         if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
109         {
110                 glfwSetWindowShouldClose(window, GL_TRUE);
111         }
112         else if (key >= 0 && key < 1024)
113         {
114                 if (key == GLFW_KEY_B && action == GLFW_PRESS)
115                 {
116                         theWindow->keys[key] =
                            ↪ !theWindow->keys[key];
117                 }
118                 else if (key == GLFW_KEY_B && action ==
                    ↪ GLFW_RELEASE)
119                         return;
120                 else if (action == GLFW_PRESS)
121                 {
122                         theWindow->keys[key] = true;
123                 }
124                 else if (action == GLFW_RELEASE)
125                 {
126                         theWindow->keys[key] = false;
127                 }
128         }
129 }
130
131 void Window::handleMouse(GLFWwindow* window, double xPos, double
    ↪ yPos)
132 {
133         Window* theWindow =
            ↪ static_cast<Window*>(glfwGetWindowUserPointer(window));
134
135         if (theWindow->mouseFirstMoved)
136         {
137                 theWindow->lastX = xPos;
138                 theWindow->lastY = yPos;
139                 theWindow->mouseFirstMoved = false;
140         }
141
142         theWindow->xChange = xPos - theWindow->lastX;
143         theWindow->yChange = theWindow->lastY - yPos;
144
145         theWindow->lastX = xPos;
```

```
146         theWindow->lastY = yPos;
147 }
148
149 Window::~Window()
150 {
151         glfwDestroyWindow(mainWindow);
152         glfwTerminate();
153 }
```

Klasa Window, posiada dwa konstruktory. Jeden z nich służy do utworzenia okna o zadanym rozmiarze, natomiast drugo tworzy okno o rozmiarze 800x600 [px]. Można znaleść również destruktor tej klasy który uwalnia pamięć, która była wykorzystywana przez ten obiekt. ****** TODO: Co to jest z tą myszką klawiaturą i callback ******

3.2 Utworzenie podłoża

Pierwszą rzeczą jaką postanowiliśmy utworzyć w naszym programie było podłoże na którym będzie stał nasz obiekt.

```
1  #include "Floor.h"
2
3  Floor::Floor()
4  {
5
6  }
7
8  void Floor::render(Camera camera, glm::mat4 projection, GLuint
   ↪  uniformProjection, GLuint uniformView, GLuint uniformModel)
9  {
10         glm::mat4 model(1.0f);
11         model = glm::mat4(1.0f);
12         model = glm::translate(model, glm::vec3(0.0, -0.35f,
           ↪  0.0f));
13         glUniformMatrix4fv(uniformProjection, 1, GL_FALSE,
           ↪  glm::value_ptr(projection));
14         glUniformMatrix4fv(uniformView, 1, GL_FALSE,
           ↪  glm::value_ptr(camera.ComputeCameraMatrix()));
15         glUniformMatrix4fv(uniformModel, 1, GL_FALSE,
           ↪  glm::value_ptr(model));
16 }
```

Podłoga została stworzona bez żadnego problemu, ale nie posiada żandej tekstury przez co jest niewidoczna. W celu dodania tekstury do obeikty wymagane było stworzenie meshu oraz funkcji odpowiedzialnych za ładowanie tekstur oraz przydzielanie ich do obiektów. Podczas realizacji projektu został wybrany mesh

po to, aby w łątwiejszy sposób można było dodać teksturę do obeiktu jaki właśnie stworzyliśmy.Kod po wprowadzonych zmianach zyskał następujące rzeczy:

```
#include "Floor.h"

Floor::Floor(GLfloat size)
{
        unsigned int floorIndices[] = {
                0,2,1,
                1,2,3
        };

        GLfloat floorVertices[] = {
                -size,  0.0f, -size,   0.0f,  0.0f,   0.0f,  0.0f,
                 ↪  0.0f,
                 size,  0.0f, -size,  size,  0.0f,        0.0f,
                  ↪  0.0f,  0.0f,
                -size,  0.0f,  size,   0.0f,  size,        0.0f,
                 ↪  0.0f,  0.0f,
                 size,  0.0f,  size,  size,  size,  0.0f,  0.0f,
                  ↪  0.0f
        };

        mesh.CreateMesh(floorVertices, floorIndices, 32, 6);

        texture.LoadTextureA();
}

void Floor::render(Camera camera, glm::mat4 projection, GLuint
 ↪  uniformProjection, GLuint uniformView, GLuint uniformModel)
{
    .
    .
    .

        texture.UseTexture();
        mesh.RenderMesh();
}
```

Należy pamiętać również o tym, że jeżeli podłoga ma się pojawić w naszym programie to musi zostać dodana do głównej pętli programu.

```
        .
        .
        .

        Floor floor(20.0f);
```

```
        .
        .
        .
        while (!mainWindow.getShouldClose())
        {
        .
        .
        .

                floor.render(camera, projection,
                 ↪  uniformProjection, uniformView, uniformModel);

        .
        .
        .
        }
```

3.3 Teksturowanie

Każdy obiekt stworzony w naszym programie posiada własną teksturę. W tym celu została stworozny cały odrępny plik pod nazwą Texture.cpp. Posiada on funkcje ładowania tekstury z pliku jak i jej czyszczenie.

```
#include "Texture.h"

Texture::Texture()
{
        textureID = 0;
        width = 0;
        height = 0;
        bitDepth = 0;
        fileLocation = "";
}

Texture::Texture(const char* fileLoc)
{
        textureID = 0;
        width = 0;
        height = 0;
        bitDepth = 0;
        fileLocation = fileLoc;
}
bool Texture::LoadTexture()
{
        unsigned char *texData = stbi_load(fileLocation, &width,
         ↪  &height, &bitDepth, 0);
        if (!texData)
```

```
24          {
25                  printf("Failed to fint: %s\n", fileLocation);
26                  return false;
27          }
28
29          glGenTextures(1, &textureID);
30          glBindTexture(GL_TEXTURE_2D,textureID);
31
32          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
            ↪  GL_REPEAT);
33          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
            ↪  GL_REPEAT);
34          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            ↪  GL_LINEAR);
35          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
            ↪  GL_LINEAR);
36
37          glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
            ↪  GL_RGB, GL_UNSIGNED_BYTE, texData);
38          glGenerateMipmap(GL_TEXTURE_2D);
39
40          glBindTexture(GL_TEXTURE_2D, 0);
41
42          stbi_image_free(texData);
43
44          return true;
45  }
46  bool Texture::LoadTextureA()
47  {
48      .
49      .
50      .
51  }
52  void Texture::UseTexture()
53  {
54          glActiveTexture(GL_TEXTURE0);
55          glBindTexture(GL_TEXTURE_2D, textureID);
56  }
57
58  void Texture::ClearTexture()
59  {
60          glDeleteTextures(1, &textureID);
61          textureID = 0;
62          width = 0;
63          height = 0;
64          bitDepth = 0;
65          fileLocation = "";
66
```

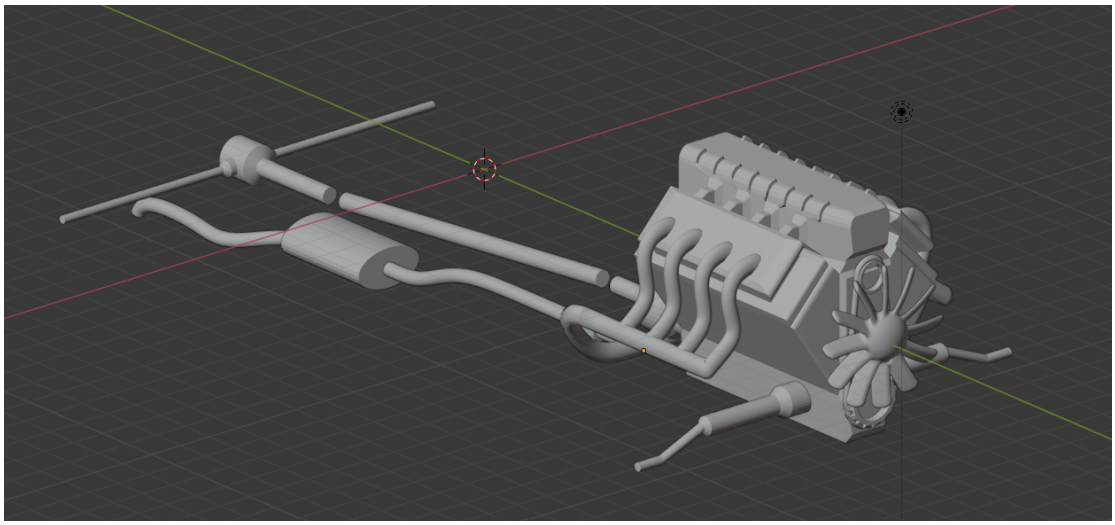
```
67 }
68
69 Texture::~Texture()
70 {
71         ClearTexture();
72 }
```

Klasa posiada dwa typy funkcji LoadTexture, różnica polega na tym, że funkcja LoadTextureA posiada kanał alpha w teksturze. Aby dodać taką funkcjonalnośc do programu należało zmienić jedną linijkę kodu.

```
1         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
          ↪  GL_RGBA, GL_UNSIGNED_BYTE, texData);
```

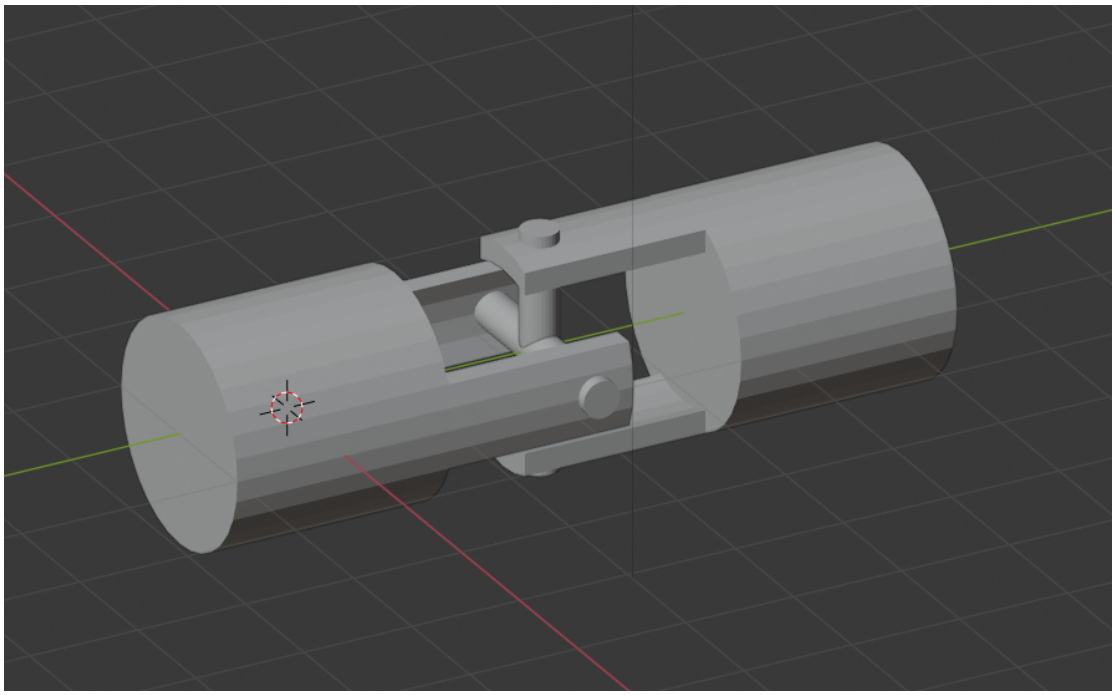
3.4 Stworzenie modeli 3D

Tak jak to było wspomnianne we wstępie do dokumentacji, cały samochód został wykonany za pomocą programu Blender. Na początku jednak samochód był tworzony ręcznie, za pomocą trójkątów. Taki sposób tworzenia samochodu był bardzo pracochłonny oraz generował bardzo duże trudności podczas dodawania kolejnych elmentów auta, dlatego też autorzy zdecydowali się na utworzenie modelów w programie Blender, a następnie wyeksportowanie do takeigo formatu, aby nie było porblemu z ich implementacją w kodzie. Na samymm początku było tworzone podwozie pojazdu.



Rysunek 3: Podwozie samochodu

Jak widać na rysunku 3 podwozie składa się z kilku prostych elementów. Na samym początku znajduje się silnik wraz z przegubami, następnie wał kardana wędruje do tylnej osi. Poswtał również cały układ wydechowy w samochodzie. Model



Rysunek 4: Przegub Cardana

ten nie posiada samych krzyżaków na wale, a także kół, gdyż aby te elementy były możliwe do zaanimowania musiały być tworzone jako osobne elementy.

Na rysunku 4 został zaprezentowany przegub Cardana, który jest istotnym elementem każdego wału napędowego w samochodzie. Przeguby w układach napędowych służą do przekazywania momentu obrotowego z jednego wału na inny, gdy osie obrotu tych wałów są do siebie nachylone. Stworzenie takeigo elementu w programie Blender nie było trudne. Taki przegub składa się z dwóch rzeczy: krzyżaka oraz jego uchwytu.

W podobny sposób zostały wykonane koła dla samochodu. Zostało stworzone jedno koło, które następnie zostało wielokrotnie wykorzystane w procesie implementacji modelu.

3.5 Implementacja modeli w programie

3.6 Ożywienie samochodu

4 Obsługa programu

5 Podsumowanie

Jak widać na rysunku 5 efekt naszej pracy jest bardzo bliski temu co staraliśmy się odwzorować. Nie był to łątwy projekt. Podczas jego realizacji napotkaliśmy wiele problemów. Niektóre z nich były łatwe w rozwiązaniu, np. problem z ruchem

Rysunek 5: Porównanie oryginału z utworzonym modelem

kamery oraz prawidłowym ładowaniu tekstur, a niektóre zajęły czasami tygodnie. Najtrudniejszym zadaniem okazało się poprawne umieszczenie stworzonego obiektu w Blenderze oraz jego zaanimowanie. Autorzy projektu nigdy wcześniej nie mieli styczności z takim typem projektu. Wymagało to od nich nabycia zupełnie nowej wiedzy, aby zrealizować ten projekt bez problemów.

Podczas realizacji projektu niezwykle pomocnym okazały się wykłady prowadzone przez Pana Dr. Ryszarda Leniowskiego. Dzięki tym wykładą autorzy projektu mogli zrozumieć charakterysytkę projektu jaki sobie postawili za cel do zrealizowania, a także dogłębnie zorzumieć jak działają biblioteki graficzne.

Literatura

[1] T. Gałaj, *Learn OpenGL*, https://shot511.github.io/pages/learnopengl/