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 GLfloat deltaTime = 0.0f;
2 GLfloat lastTime = 0.0f;
3 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 int main()
12 {
13         Window mainWindow(1200, 800);
14         mainWindow.Initialize();
15
16         while (!mainWindow.getShouldClose())
17         {
18                 computeFPS(mainWindow.mainWindow);
19
20                 glfwPollEvents();
21
22                 glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
23                 glClear(GL_COLOR_BUFFER_BIT |
                    ↪  GL_DEPTH_BUFFER_BIT);
24
25                 glUseProgram(0);
26                 mainWindow.swapBuffers();
27         }
28
29         return 0;
30 }
```

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. To własnie ta pętla odpowiada za ponowne wyświetlanie wszystkich elementów.

```
1  void computeFPS(GLFWwindow* pWindow)
2  {
3          GLfloat now = glfwGetTime();
4          deltaTime = now - lastTime;
5          lastTime = now;
6          timeCounter += deltaTime;
7          frameCount++;
8
9          if (timeCounter >= 1.0) {
10                 double fps = double(frameCount) / timeCounter;
11
12                 std::stringstream ss;
```

```
                ss << " [" << fps << " FPS]";

                glfwSetWindowTitle(pWindow, ss.str().c_str());

                frameCount = 0;
                timeCounter = 0;
        }
}
```

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

```
int Window::Initialize()
{
        if (!glfwInit())
        {
                printf("Error Initialising GLFW");
                glfwTerminate();
                return 1;
        }

        glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
        glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);

        glfwWindowHint(GLFW_OPENGL_PROFILE,
        ↪  GLFW_OPENGL_CORE_PROFILE);

        glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

        glfwWindowHint(GLFW_REFRESH_RATE, 60);

        const GLFWvidmode* mode =
        ↪  glfwGetVideoMode(glfwGetPrimaryMonitor());
    glfwGetPrimaryMonitor(), NULL);
        mainWindow = glfwCreateWindow(width, height, "Microcar
        ↪  Flex Furgon", NULL, NULL);
        width = mode->width;
        height = mode->height;
        if (!mainWindow)
        {
                printf("Error creating GLFW window!");
                glfwTerminate();
```

```
28                  return 1;
29          }
30
31          glfwGetFramebufferSize(mainWindow, &bufferWidth,
            ↪  &bufferHeight);
32
33          glfwMakeContextCurrent(mainWindow);
34
35          createCallbacks();
36          glfwSetInputMode(mainWindow, GLFW_CURSOR,
            ↪  GLFW_CURSOR_DISABLED);
37
38          glewExperimental = GL_TRUE;
39
40          GLenum error = glewInit();
41          if (error != GLEW_OK)
42          {
43                  printf("Error: %s", glewGetErrorString(error));
44                  glfwDestroyWindow(mainWindow);
45                  glfwTerminate();
46                  return 1;
47          }
48
49          glEnable(GL_DEPTH_TEST);
50
51          glViewport(0, 0, bufferWidth, bufferHeight);
52
53          glfwSetWindowUserPointer(mainWindow, this);
54  }
```

W funkcji inicjalizacyjnej okna ustawiane są wszelkie niezbędne parametry do utworzenia okna aplikacji. Definiowany jest rozmiar okna, tryb pracy biblioteki openGl, a także odświeżanie okna oraz obsługa zdarzeń. W aplikacji została zainmplementowana obsługa zdarzeń, po przez stworzenie jednej wielkiej tablicy z aktualnym stanem klawiszy.

W celu obsługi zdarzeń klawiszy została stworzona funkcja o nazwie handleKeys, która została przedstawiona w listingu kodu poniżej. W funkcji tej wykrywane jest, który klawisz został naciśnięty i odpowiednio zmieniany jest jego stan w tablicy.

```
1   void Window::handleKeys(GLFWwindow* window, int key, int code, int
    ↪  action, int mode)
2   {
3           Window* theWindow =
            ↪  static_cast<Window*>(glfwGetWindowUserPointer(window));
4
```

```
        if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        {
                glfwSetWindowShouldClose(window, GL_TRUE);
        }
        else if (key >= 0 && key < 1024)
        {
                if (key == GLFW_KEY_B && action == GLFW_PRESS)
                {
                        theWindow->keys[key] =
                        ↪  !theWindow->keys[key];
                }
                else if (key == GLFW_KEY_B && action ==
                ↪  GLFW_RELEASE)
                        return;
                else if (action == GLFW_PRESS)
                {
                        theWindow->keys[key] = true;
                }
                else if (action == GLFW_RELEASE)
                {
                        theWindow->keys[key] = false;
                }
        }
}
```

W kolejnym listiningu code przedstawione jest to jak stworzona została funkcja przechwytująca ruchy myszką. Są one tak samo przechowywane w odpowiednich zmiennych, aby można było w łatwy sposób z nich korzystać.

```
void Window::handleMouse(GLFWwindow* window, double xPos, double
↪  yPos)
{
        Window* theWindow =
        ↪  static_cast<Window*>(glfwGetWindowUserPointer(window));

        if (theWindow->mouseFirstMoved)
        {
                theWindow->lastX = xPos;
                theWindow->lastY = yPos;
                theWindow->mouseFirstMoved = false;
        }

        theWindow->xChange = xPos - theWindow->lastX;
        theWindow->yChange = theWindow->lastY - yPos;

        theWindow->lastX = xPos;
        theWindow->lastY = yPos;
```

```
17 }
```

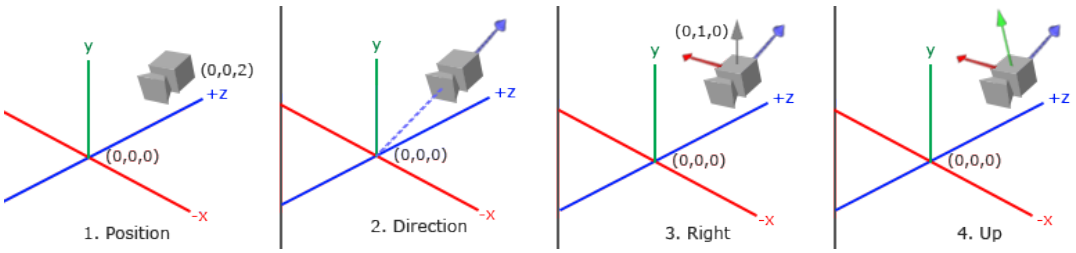
W celu implementacji tych własnoręcznie napisanych funkcji obsługi zdarzeń, zostały one podpięte pod callbacki funkcji OpenGl'owych.

```
1 void Window::createCallbacks()
2 {
3         glfwSetKeyCallback(mainWindow, handleKeys);
4         glfwSetCursorPosCallback(mainWindow, handleMouse);
5 }
```

Klasa Window, posiada również 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.

3.2 Kamera

Aby móc modyfikować to co użytkownik widzi na ekranie napisana została klasa `Camera` implementująca kamerę. Aby zdefiniować kamerę, potrzebujemy jej pozycji w przestrzeni świata, kierunku, w którym patrzy, wektora wskazującego w prawo i wektora skierowanego w górę od kamery. Będziemy tworzyć układ współrzędnych z 3 prostopadłymi do siebie osiami jednostkowymi z pozycją kamery jako początkiem układu.



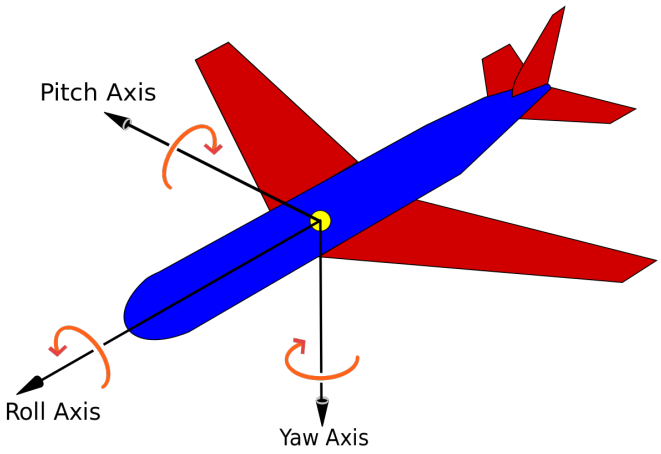
Rysunek 3: Najważniejsze parametry kamery

Jeżeli zdefiniuje się przestrzeń współrzędnych za pomocą 3 prostopadłych (lub nie liniowych) osi, można utworzyć macierz z tymi 3 osiami i wektorem translacji, dzięki czemu można przekształcić dowolny wektor do tego stworzonego układu współrzędnych, mnożąc ten wektor przez tą macierz. Ta zależność znacznie ułatwia dalszą implementację kamery. Opisany wyżej proces jest realizowany przez funkcję `glm::lookAt()`, która wymaga wektora pozycji, punktu, na który ma być skierowana kamera i górnego wektora. Tworzy ona odpowiednią macierz widoku.

Kolejnym krokiem było umożliwienie poruszania kamerą przez użytkownika. W tym celu utworzona została funkcja `MouseControl()`, dzięki której każdorazowe naciśnięcie jednego z klawiszy strzałek spowoduje aktualizację pozycji kamery. Do

poruszania się do przodu lub do tyłu, należy dodać lub odjąć wektor kierunku od wektora położenia. Natomiast do przesuwania się na boki, wykonać iloczyn wektorowy, aby utworzyć prawy wektor i odpowiednio poruszać się wzdłuż tego wektora w prawo.

Do pełnej wygody korzystania z kamery brakowało jeszcze możliwości rozglądania się. Do tego celu wykorzystaliśmy kąty Eulera — *pitch*, *yaw* i *roll*.



Rysunek 4: Rozkład kątów Eulera

Pitch to kąt, który obrazuje, jak bardzo patrzymy w górę lub w dół. Yaw reprezentuje wielkość tego jak bardzo patrzymy w lewo lub w prawo. Roll oznacza, jak bardzo się turlamy, co jest najczęściej używane w statkach kosmicznych i powietrznych. Każdy z kątów Eulera jest reprezentowany przez pojedynczą wartość, a za pomocą kombinacji wszystkich trzech z nich możemy obliczyć dowolny wektor rotacji w 3D.

W naszy systemie wykorzystaliśmy tylko kąty *pitch* i *yaw*, a do ich modyfikacji wykorzystujemy ruch myszki.

3.3 Utworzenie podłoża

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

```
1 void Floor::render(Camera camera, glm::mat4 projection, GLuint
  ↪ uniformProjection, GLuint uniformView, GLuint uniformModel)
2 {
3         glm::mat4 model(1.0f);
4         model = glm::mat4(1.0f);
5         model = glm::translate(model, glm::vec3(0.0, -0.35f,
          ↪ 0.0f));
6         glUniformMatrix4fv(uniformProjection, 1, GL_FALSE,
          ↪ glm::value_ptr(projection));
```

```
        glUniformMatrix4fv(uniformView, 1, GL_FALSE,
        ↪ glm::value_ptr(camera.ComputeCameraMatrix()));
        glUniformMatrix4fv(uniformModel, 1, GL_FALSE,
        ↪ glm::value_ptr(model));
}
```

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:

```
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);
        ...
        }
```

3.4 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. W celu przypisania przygotowanej tekstury do obeiktu wykorzystywana jest funkcja UseTexture, która podmienia aktualną teksturę obiektu na tą zadaną.

```
bool Texture::LoadTexture()
{
        unsigned char *texData = stbi_load(fileLocation, &width,
         ↪  &height, &bitDepth, 0);
        if (!texData)
        {
                printf("Failed to fint: %s\n", fileLocation);
                return false;
        }

        glGenTextures(1, &textureID);
        glBindTexture(GL_TEXTURE_2D,textureID);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
         ↪  GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
         ↪  GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
         ↪  GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
         ↪  GL_LINEAR);

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
         ↪  GL_RGB, GL_UNSIGNED_BYTE, texData);
        glGenerateMipmap(GL_TEXTURE_2D);

```

```
        glBindTexture(GL_TEXTURE_2D, 0);

        stbi_image_free(texData);

        return true;
}
bool Texture::LoadTextureA()
{
    ...
}
void Texture::UseTexture()
{
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, textureID);
}

void Texture::ClearTexture()
{
        glDeleteTextures(1, &textureID);
        textureID = 0;
        width = 0;
        height = 0;
        bitDepth = 0;
        fileLocation = "";

}
```

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 z:

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

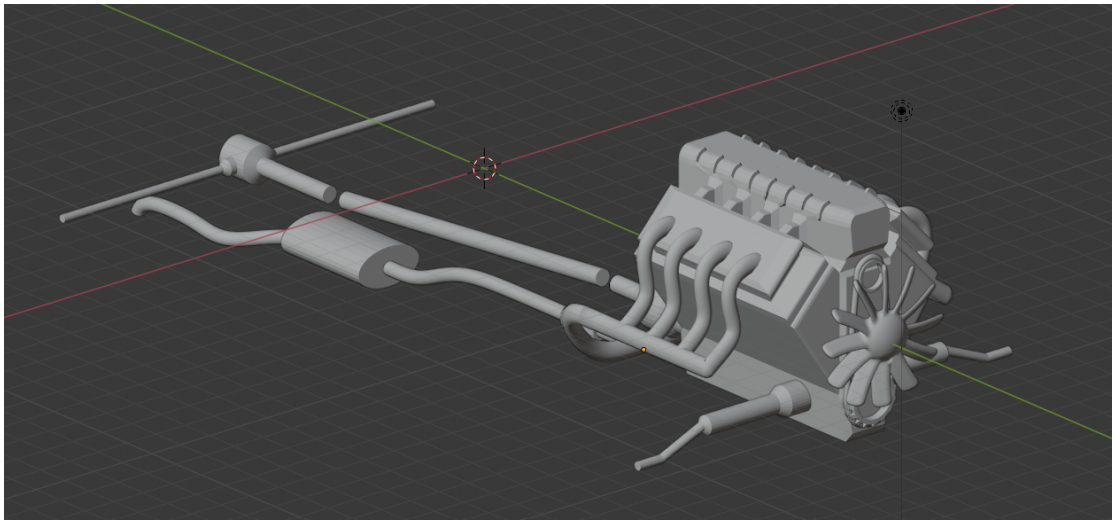
Na

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

3.5 Stworzenie modeli 3D

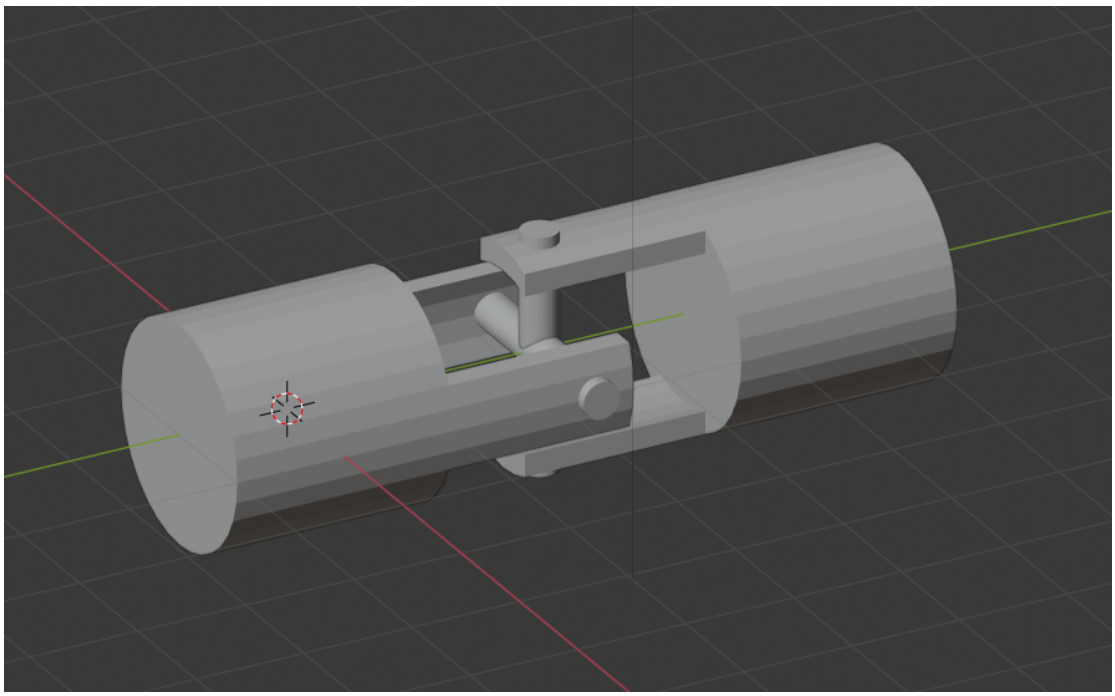
Tak jak to było wspomnianne we wstępie do dokumentacji, cały samochód został wykonany za pomocą programu Blender. Na początku jednak samochód był tworzony ręcznie, za pomocą trójkątów. Taki sposób tworzenia samochodu

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.



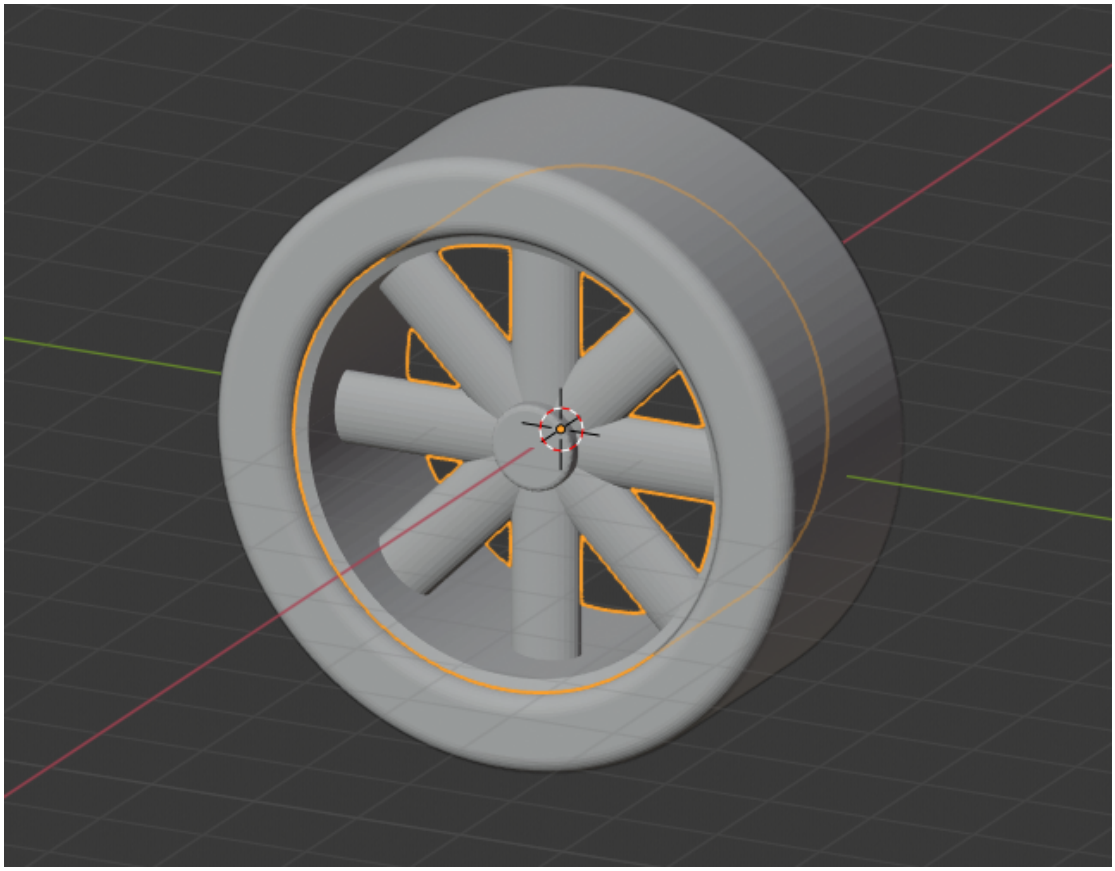
Rysunek 5: Podwozie samochodu

Jak widać na rysunku 5 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 ten nie posiada samych krzyżaków na wale, a także kół, gdyż aby te elementy były możliwe do zaanimowania musiały być tworzone jako osobne elementy.



Rysunek 6: Przegub Cardana

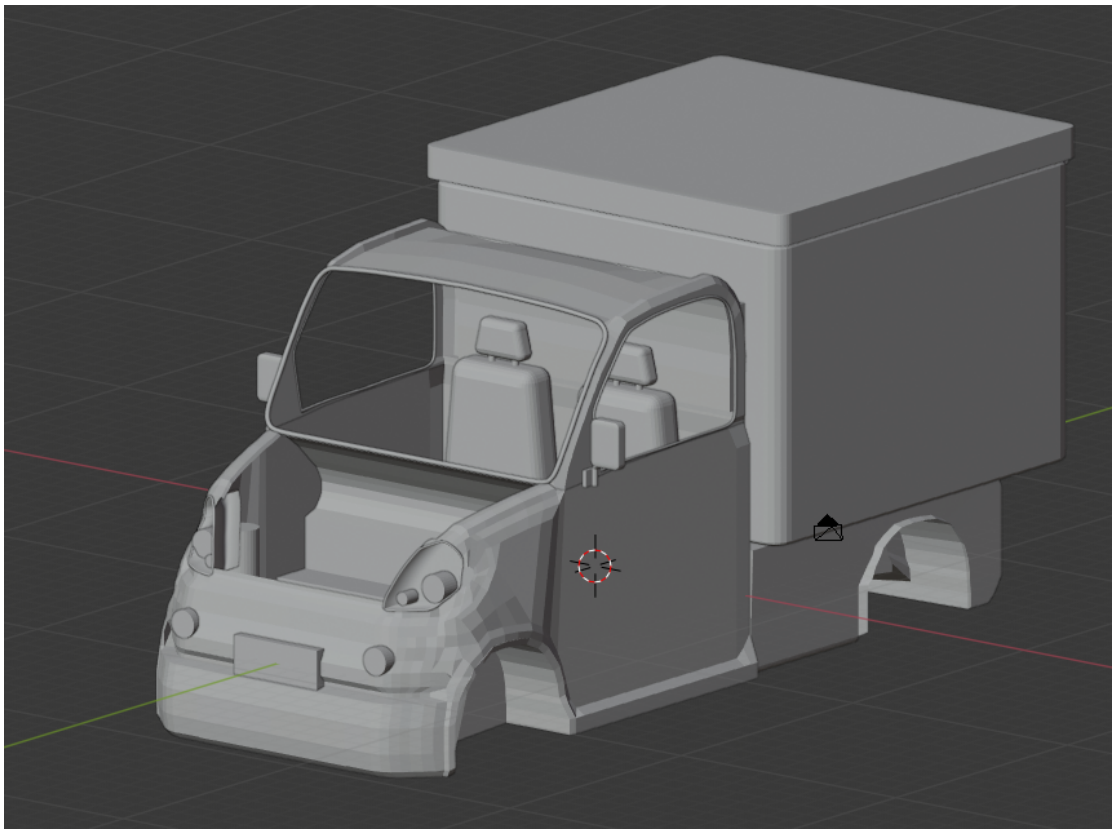
Na rysunku 6 został zaprezentowany przegub Cardana, który jest istotnym elementem każdego wału napędowego w samochodzie. Przeguby w układach napędowych służą do przekazywania momentu obrotowego z jednego wału na inny, gdy osie obrotu tych wałów są do siebie nachylone. Stworzenie takeigo elementu w programie Blender nie było trudne. Taki przegub składa się z dwóch rzeczy: krzyżaka oraz jego uchwytu.



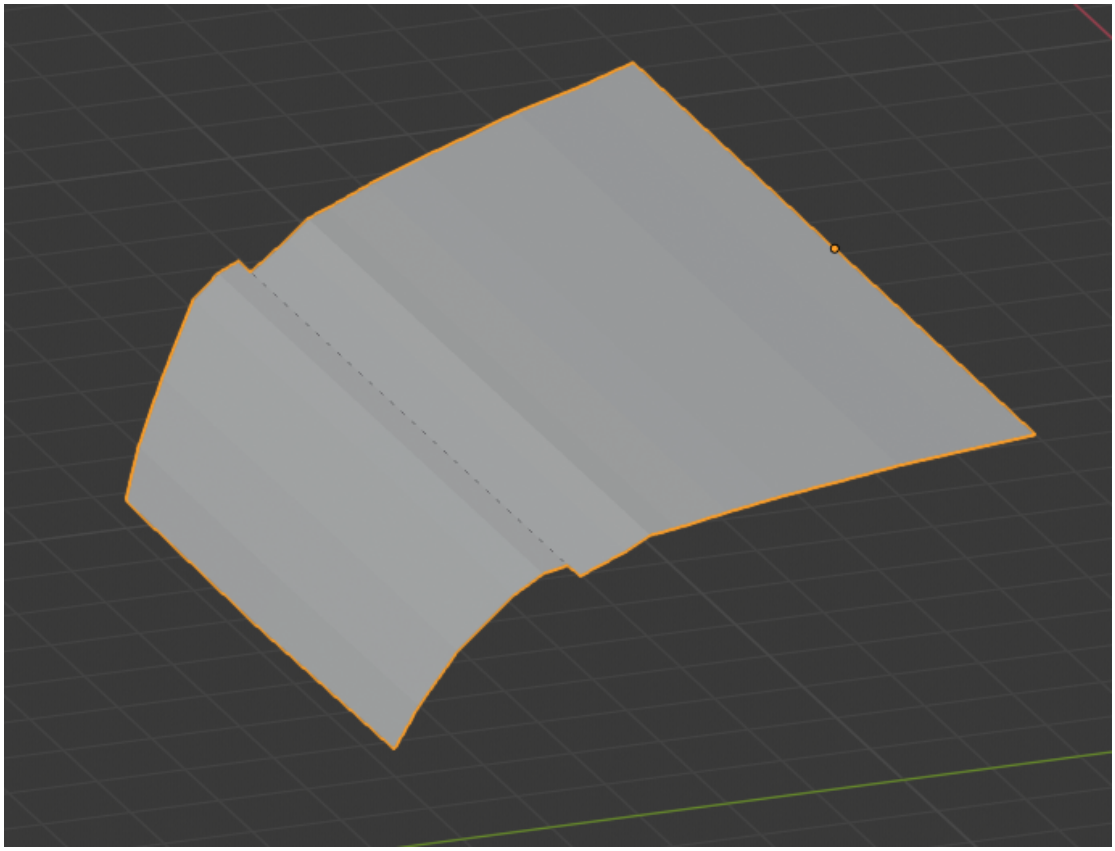
Rysunek 7: Koło samochodu

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

Cała karoseria pojazdu na początku została stworzona jako jednolita bryła, niestety w taki sposób nie istnieje łatwy sposób na zaanimowanie pewnych części karoserii. Jako osobne obiekty zostały wyeksportowane takie lementy jak: maska oraz klapa bagażnika. W ten o to sposób samochód zyskał możliwość otwierania oraz zamykania klaby bagażnika oraz maski.



Rysunek 8: Nadwozie samochodu



Rysunek 9: Maska pojazdu

3.6 Implementacja modeli w programie

Stworzone w Blenderze modele 3D należało następnie dodać do projektu. Aby to zrobić posłużyliśmy się biblioteką *Assimp*. Służy ona do importowania modeli 3D w różnych formatach do OpenGL'a.

Na początku utworzyliśmy klasę `Model`, która będzie służyła jako kontener na meshe, materiały i tekstury naszych obiektów. Klasa ta posiada kilka metod do ładowania poszczególnych części modelu, a także funkcję `RenderModel()`, dzięki której cały model jest renderowany w oknie.

```
1  void Model::LoadModel(const std::string& fileName)
2  {
3          Assimp::Importer importer;
4          const aiScene *scene = importer.ReadFile(fileName,
           ↪  aiProcess_Triangulate | aiProcess_FlipUVs |
           ↪  aiProcess_GenSmoothNormals |
           ↪  aiProcess_JoinIdenticalVertices);
5          LoadNode(scene->mRootNode, scene);
6          LoadMaterials(scene);
7  }
```

Rysunek 10: Funkcja ładująca model

```
1  void Model::RenderModel()
2  {
3          for (size_t i = 0; i < meshList.size(); i++) {
4                  unsigned int materialIndex = meshToTex[i];
5
6                  if (materialIndex < textureList.size() &&
                   ↪  textureList[materialIndex]) {
7                          textureList[materialIndex]->UseTexture();
8                  }
9                  meshList[i]->RenderMesh();
10         }
11 }
```

Rysunek 11: Funkcja renderująca model

3.7 Ożywienie samochodu

W celu łatwiejszego zarządzania pojazdem utworzona została klasa `Car`, w której zawarte zostały najważniejsze parametry pojazdu. Funkcja `computeMovement()` służy do obliczenia pozycji samochodu oraz obrotu kół.

```
void Car::computeMovement(GLfloat deltaTime)
{
        GLfloat angle = (velocity * deltaTime) / 1.0f;
        wheelRotation += glm::degrees(angle);
        wheelRotation = fmod(wheelRotation, 360.0f);

        GLfloat roadLength = (velocity * deltaTime);
        position[0] += roadLength * sin(glm::radians(azymuth *
        ↪  0.5));
        position[1] += roadLength * cos(glm::radians(azymuth *
        ↪  0.5));
}
```

Obliczane w tej funkcji parametry są potem uwzględniane przy renderowaniu modelu.

- Strzałka w górę — zwiększa prędkość pojazdu
- Strzałka w dół — zmniejsza prędkość pojazdu
- Strzałka w lewo — obrót kół w lewo
- Strzałka w prawo — obrót kół w prawo
- Klawisz 1 — otwieranie maski
- Klawisz 2 — zamykanie maski
- Klawisz 3 — otwieranie bagażnika
- Klawisz 4 — zamykanie bagażnika

W funkcji `reactToInput()` modyfikowane są parametry samochodu w zależności od wciśniętych klawiszy. Dzięki temu doszła możliwość otwierania bagażnika oraz maski, a także skręcania kołami.

```
void Car::reactToInput(bool* keys)
{
        if (keys[GLFW_KEY_UP] && velocity < maxSpeed) velocity +=
        ↪  0.1f;
        if (keys[GLFW_KEY_DOWN] && velocity > -maxSpeed) velocity
        ↪  -= 0.1f;
        if (keys[GLFW_KEY_SPACE]) velocity = 0.0f;

        if (keys[GLFW_KEY_1] && hoodDegree < 90.0f) hoodDegree +=
        ↪  1.0f;
        if (keys[GLFW_KEY_2] && hoodDegree > 0.0f) hoodDegree -=
        ↪  1.0f;
```

```
9
10          if (keys[GLFW_KEY_3] && flapDegree < 90.0f) flapDegree +=
            ↪  1.0f;
11          if (keys[GLFW_KEY_4] && flapDegree > 0.0f) flapDegree -=
            ↪  1.0f;
12
13          if (keys[GLFW_KEY_LEFT] && turnAngle < 30.0f) turnAngle +=
            ↪  0.2f;
14          if (keys[GLFW_KEY_RIGHT] && turnAngle > -30.0f) turnAngle
            ↪  -= 0.2f;
15
16          hideBody = keys[GLFW_KEY_B];
17  }
```

4 Obsługa programu

5 Podsumowanie

Jak widać na rysunku 12 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 kamery oraz prawidłowym ładowaniu tekstur, a niektóre zajęły czasami tygodnie. Najtrudniejszym zadaniem okazało się poprawne umieszczenie stworzonego obiektu w Blenderze oraz jego zaanimowanie.

Rysunek 12: Porównanie oryginału z utworzonym modelem

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/