UNIWERSYTET ŚLĄSKI
WYDZIAŁ NAUK ŚCISŁYCH I TECHNICZNYCH
INSTYTUT INFORMATYKI

Gajos Kamil, Dróżdż Michał

Silnik do tworzenia gier

(Projekt zaliczeniowy z programowania warstwy wizualnej gry)

Sosnowiec 2024

Spis treści

Wstęp

Celem projektu było stworzenie narzędzia umożliwiającego tworzenie gier. Z racji przedmiotu programowanie warstwy wizualnej gry skupiliśmy się na stworzeniu narzędzia umożliwiającego developerowi szybkie przeglądnięcie efektów zaimplementowanych w dwóch przykładach. Modularność w kodzie dzięki czemu silnik do tworzenia gier może być rozwijany dalej i wprowadzać kolejne funkcjonalności jakich tylko zapragniemy. Przyświecała nam wizja, że w przyszłości wykorzystamy ten silnik do stworzenia pełnoprawnego tytułu lub udostępnimy go innym osobą, które go stworzą. Z racji aktualnych standardów, którymi rządzi się branża silników gier gdzie wielu producentów zaczęło podnosić dla użytkowników użytkujących ich silnik koszty utylizacji komercyjnego czy też niekomercyjnego chcieliśmy spróbować swoich sił w nauce interfejsu programowania aplikacji (ang. Application Programming Interface, API) OpenGL oraz języka bliźniego czyli języka cieni (ang. OpenGL Shading Language, GLSL) Z racji, że mamy już rok czasu za sobą jeśli chodzi o naukę OpenGL udało nam się połączyć funkcjonalności innych wykorzystanych bibliotek w jeszcze lepszy sposób dzięki czemu program oferuję więcej efektów wizualnych jak i całe sterowanie zostało ulepszone. Oprócz jednej podstawowej sceny teraz mamy możliwość dodawania kilkudziesięciu poziomów w grze oraz mieszać nie ożywione obiekty z żywymi. Słowem kończącym wstęp aktualnym kierunkiem przyszłościowym rozwoju projektu jest zapoznanie się z kolizjami w 3D czyli algorytmem dystansu Gilberta–Johnsona–Keerthiego. (ang. Gilbert–Johnson–Keerthi distance algorithm, GJK).

1. Teoria i algorytmy

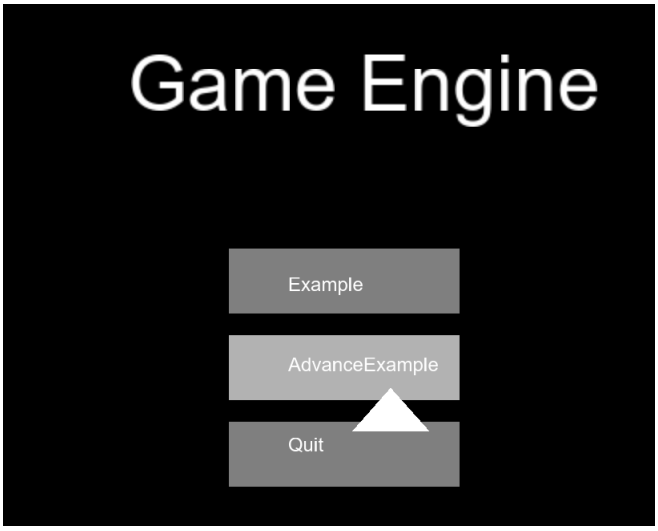
1.1. Rozmycie gaussowskie

Zaimplementowane w silniku rozmycie polega na przerzucaniu między bufforami klatek aktualnie wcześniej wy-renderowanej sceny następnie poprzez dwa różne przejścia czyli horyzontalne oraz wertykalne następnie w shaderze następuję dodanie wag z wykładni Gaussa. Takie zachowanie nazywamy pingpongie gdyż oby dwie klatki odbijają przysłowiową piłeczkę między sobą do momentu zatrzymania. Im dłużej będą robić tym więcej zostanie stworzonej poświaty wokoło najjaśniejszych elementów. Efekt ten jest najczęściej widoczny jest gdy wykorzystujemy (ang. high dynamic range, HDR) a nie LDR (ang. Low Dynamic Range, LDR) gdy mamy większą możliwość do osiągnięcia bardzo jasny stanów klasyfikujących się ponad określony warunek 1.1. Dodatkowe informację jak dokładnie wylicza się rozmycie można znaleźć tutaj [2].

Rysunek 1.1. Przykład rozmycia gaussa

1.2. Kolizje z osiami układu współrzędnego

Z angielskiego (ang. Axes Aligned Bounding Boxes, AABB) polega na sprawdzaniu między dwoma obiektami w 2D czy ich maksymalne obrysy na siebie zachodzą. My wykorzystujemy tą podstawową mechanikę przy sprawdzaniu czy gracz w menu najechał na przycisk, ale ma ona dużo zastosowań i jej wielką zaletą jest to jaka ona jest szybka o czym pisał Gino van den Bergen w artykule Journal of Graphics Tools [4] 1.2 widać, że nasz kursor przeszedł mały tunning i zmienił się w prosty trójkąt, który wskazują naszą pozycję myszy poprzez umieszczenie go na czubku wierzchołka. Dodatkowo widać, że AABB zadziałało z racji małe rozświetlenia przycisku.



Rysunek 1.2. Przykład działania AABB

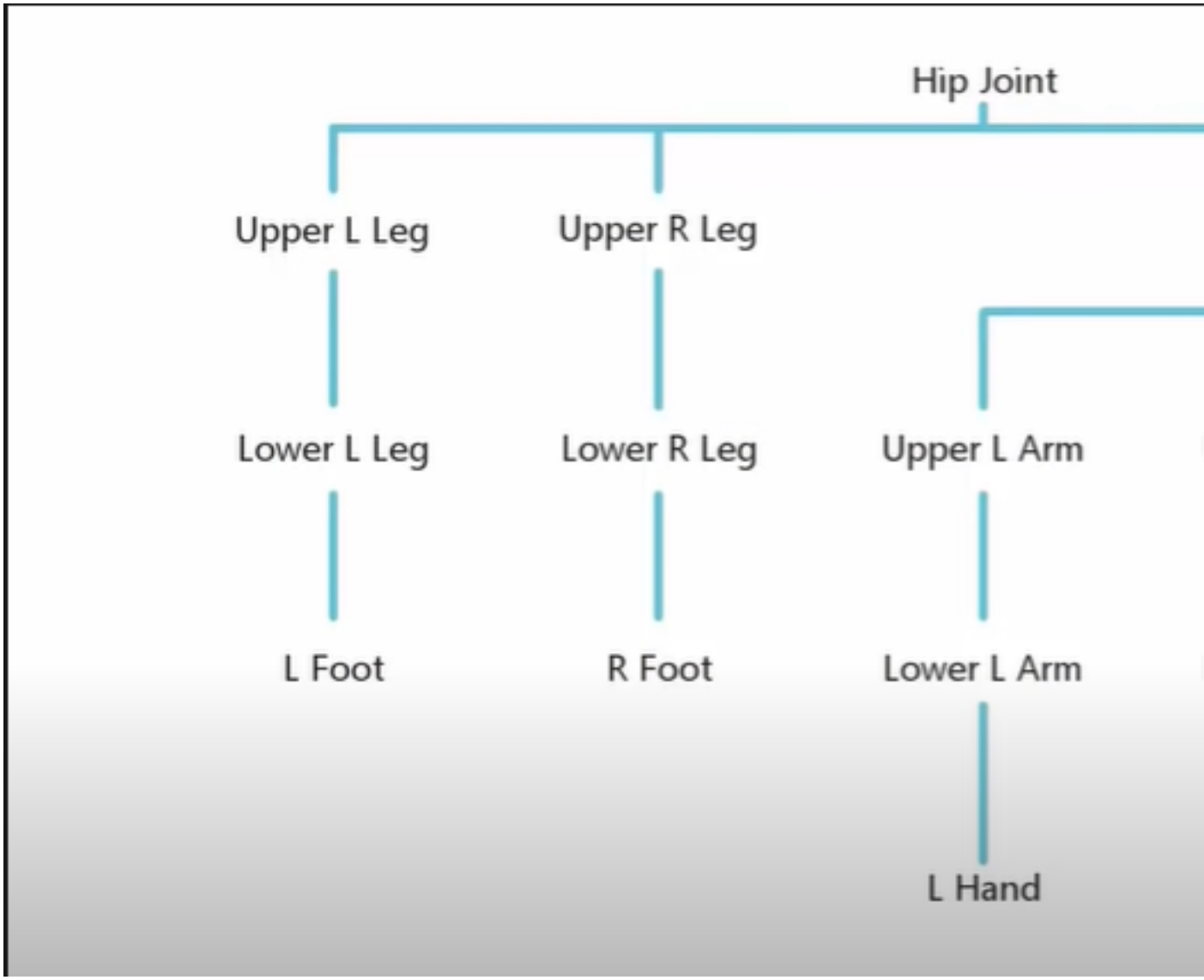
1.3. Interpolacja

Najbardziej powszechna forma animacji polegająca na płynnym przejściu pomiędzy dwoma wartościami. By skutecznie interpolować potrzebujemy znać czas między klatkami by nasze animacje były niezależne od sprzętu gracza. By wyliczyć interpolację musimy znać czas całej animacji oraz jak szybko w tikach na sekundę lub innej mierze czasowe ma być ona odtwarzana oczywiście uwzględniając wcześniej deltę między kolejnymi klatkami. Po czym wyliczamy w jakiej przestrzeni jesteśmy od finalnego efektu czyli następnego stanu określonego. Potem mieszamy oba stany w proporcji czasowej im bliżej następnej klatki.

1.4. Animacja szkieletowa

Podobna do interpolacji z jedno zasadniczą różnicą posiada dedykowany układ nazywany układem kości. W tym układzie następują transformację takie jak odby-

wają się w OpenGL na przykład w transformacji z Clip Space do NDSC czy lokalnego na światowy. Problem polega na tym, że jak w przypadku interpolacji bawiliśmy się jendym obiektem tutaj każda kość jest ze sobą połączona 1.3 Wzór na obliczanie in-



Rysunek 1.3. Relacja rodzic-dziecko

terpolacji to a = a * (1 - t) + b * t gdzie t to czas a to aktualna pozycja a b to pozycja końcowa. Szkopuł w tym, że teraz musimy sprawić by kości na siebie oddziaływały by cała ręka poszła w ruch a nie tylko dłoń. Dlatego musimy przejść przez wszystkie kości końcowe aż dojdziemy do góry z transformacjami i wrócimy do normalnego OpenGL i przestrzeni lokalnej by potem gdy chcemy połączyć szkieletową animację z interpolacją wykorzystujemy kolejne przekształcenia, które nie mają już wpływy na kości i animację szkieletową 1.4.

Więcej na powyższy temat można znaleźć tutaj [3].

Rysunek 1.4. Przykładowa animacja szkieletowa

1.5. Mapowanie tonów Reinharda

Gdy mamy do czynienia z większą ilością widzialnych barw znajdujących się poza zasięgiem zdefiniowanym w OpenGL czyli 0,0f-1.0f musimy wykorzystać coś co pozwoli nam odróżnić światło, które przekracza ten zakres. Do tego wykorzystujemy tonowanie a dzięki temu prostemu wzorowi jesteśmy wstanie zamienić obraz z HDR znów w nasze LDR hdrColor / (hdrColor + vec3(1.0f)). Warto tutaj nadmienić, że istnieją też inne metody tonowania o którcyh można się dowiedzieć więcej tutaj [1]

2. Opis programu

Silnik został stworzony przy pomocy języka programistycznego C++ wersja 14 międzynarodowy standard (ang. International Standard, ISO) oraz wykorzystuję OpenGL w wersji rdzennej z 2017 czyli 4.6 przy użyciu generatora ładowania zależności (ang. Loader-Generator, glad). Dodatkowo wykorzystuję otwartą bibliotekę importowania zasobów (ang. Open Asset Import Library, assimp), pojedynczy plik publicznej domeny bibliotek dla języka C i C++ (ang. single-file public domain libraries for C/C++, stb) graficzną bibliotekę struktury okna (ang. Graphics Library Framework, GLFW) darmową czcionkę (ang. FreeType, FT) matematykę do OpenGL (ang. OpenGL mathematics, GLM).

Do tworzenia modeli i tekstur przedstawionych w przykładach został wykorzystany Blender w wersji 4.0 oraz Photopea. Do programowania wykorzystaliśmy Visual Studio w wersji 17 z 2022 społecznością wersję (ang. Community, VS). Oprócz tego wykorzystywany był Cmake do stworzenia zależności dla assimp oraz freetype.

2.1. Możliwości programu

Program jest wstanie renderować sceny i przełączać się między nimi dzięki czemu w dowolnym momencie można rozszerzyć silnik o kolejne sceny. Sceny mogą być zarówno 2D jak i 3D. Oprócz nich mamy możliwość poruszania się podczas każdej ze scen i bieżące modyfikowanie ich stanów za pomocą klawiszy funkcyjnych.

Animacja szkieletowa to nasze największe osiągnięcie z którego jesteśmy dumni. Dzięki assimp jesteśmy wstanie załadować animacje dowolnego modelu wykorzystującego maksymalnie 4 kości opisujące jeden wierzchołek i przekształcać załadowane kości modelu wedle uznania tworząc zjawiskowe ciągi wydarzeń takie jak zaprezentowane w przykładach.

Assimp również umożliwił nam stworzenie klasy model umożliwiającej nam załadowanie modelu z dwoma teksturami jedną odpowiedzialną za kolor światła rozpraszającego (ang. diffuse) a drugi za światło odbijające (ang. specular).

Dzięki detekcji kolizji AABB jesteśmy wstanie w bardzo efektywny sposób zweryfikować czy zachodzi kolizja między kursorem gracza a przyciskami dzięki czemu możemy się przełączać między scenami.

Wykorzystując kilka różnych modułów cieniujących (ang. shader) możemy płynnie przełączać się między różnymi efektami takimi jak widok 2D i 3D czy też emitować

światło, wyświetlać tekst, włączać większy dynamiczny zasięg barw (ang. high dynamic range, HDR) czy rozmycie gaussa (ang. gausian blur)

GLFW umożliwia nam płynną obsługę klawiatury i myszy dzięki czemu możemy latać i poruszać się na scenach jak i włączać i wyłączać różne efekty wizualne, które udało nam się zaimplementować używając OpenGL czy przełączać się między scenami powracając do menu z załadowanej sceny.

Jesteśmy wstanie tworzyć wiele elementów i zmieniać ich położenie na scenie poprzez rotację przesunięcie czy też powiększanie i zmniejszanie. Elementy te dzięki wielu modułą cieniującym mogą być statyczne bądź dynamiczne.

2.2. Obsługa aplikacji

W celu przełączania się między scenami wykorzystywana jest myszka po naciśnięciu ESC zawsze jesteśmy przenoszeni do głównego menu. Po przejściu resetowana jest pozycja myszki ale zapamiętywana pozycja ustawionej kamery w obu scenach.

Pod klawiszem F została umieszczona latarka, którą gracz może w dowolnym momencie włączyć i wyłączyć. Pod klawiszem G mamy gamma corection, który działa na tej samej zasadzie. Oprócz tego mamy również możliwosć włączenia HDR za pomocą klawisza H oraz Blooma za pomocą klawisza B.

By poruszać się na wybranej scenie używamy myszki oraz klawiszów WSAD. Gdzie W porusza nas do przodu a S do tyłu. AD służą do poruszania się na boki a do sterowania aktualnym nachyleniem wykorzystywana jest myszka.

3. Opis Kodu

3.1. main

W mainie mamy odwołania z GLFW, które obsługują między innymi klikanie myszą jej ruch czy też naciskanie klawiszy, które umożliwia przełączanie między scenami czy uruchamiać dodatkowych funkcji silnika takie jak bloom czy HDR. W głównej pętli **while**(!glfwWouldShouldClose(window)) następuję wybranie aktualnie renderowanej sceny oraz wyliczenie deltaTime, który służy przede wszystkim do poprawnego wyświetlania ruchu w zależności od czasu pomiędzy klatkami.

```
1  #include"SceneManager.h"
2
3  bool firstMousePositionChange = true;
4  float lastPositionX = 0.0f, lastPositionY = 0.0f;
5  int offsetX = 0, offsetY = 0;
6
7  SceneManager sceneManager;
8
9  void framebufferSizeCallback(GLFWwindow* window, int width, int height);
10 void cursorPositionCallback(GLFWwindow* window, double posX, double ↙
       ↳ PosY);
11 void keyCallback(GLFWwindow* window, int key, int scancode, int ↙
       ↳ action, int mods);
12 void MouseButtonCallback(GLFWwindow* window, int button, int action, ↙
       ↳ int mods);
13
14 int main()
15 {
16     glfwInit();
17
18     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
19     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
20     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
21
22     GLFWwindow* window = glfwCreateWindow(sceneManager.width, ↙
           ↳ sceneManager.height, "Game␣Engine", NULL, NULL);
23     if (window == NULL)
24     {
25         std::cout << "Failed␣To␣initialize␣window" << std::endl;
```

```
26          glfwTerminate();
27          return -1;
28      }
29      glfwMakeContextCurrent(window);
30
31      glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
32
33      glfwSetFramebufferSizeCallback(window, framebufferSizeCallback);
34      glfwSetCursorPosCallback(window, cursorPositionCallback);
35      glfwSetKeyCallback(window, keyCallback);
36      glfwSetMouseButtonCallback(window, MouseButtonCallback);
37
38      gladLoadGL();
39
40      glEnable(GL_DEPTH_TEST);
41
42      float deltaTime = 0, lastTime = 0;
43
44      sceneManager = SceneManager(0);
45
46      while (!glfwWindowShouldClose(window))
47      {
48          float currentTime = static_cast<float>(glfwGetTime());
49          deltaTime = currentTime - lastTime;
50          lastTime = currentTime;
51
52          glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
53          glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
54
55          sceneManager.Render(window, deltaTime);
56
57          glfwSwapBuffers(window);
58
59          glfwPollEvents();
60      }
61
62      glfwDestroyWindow(window);
63      glfwTerminate();
64      return 0;
65  }
66
67  void framebufferSizeCallback(GLFWwindow* window, int Width, int Height)
68  {
69      sceneManager.width = Width;
70      sceneManager.height = Height;
71      glViewport(0, 0, sceneManager.width, sceneManager.height);
72  }
```

```
73
74  void cursorPositionCallback(GLFWwindow* window, double posX, double ↙
        ↳ posY)
75  {
76      if (sceneManager.ID == 0)
77      {
78          sceneManager.scenes[0]->mouseX = posX;
79          sceneManager.scenes[0]->mouseY = posY;
80      }
81      if (sceneManager.ID == 1 || sceneManager.ID == 2)
82      {
83          float xPosition = static_cast<float>(posX);
84          float yPosition = static_cast<float>(posY);
85
86          if (firstMousePositionChange)
87          {
88              lastPositionX = xPosition;
89              lastPositionY = yPosition;
90              firstMousePositionChange = false;
91          }
92
93          float xMouseOffset = xPosition - lastPositionX;
94          float yMouseOffset = lastPositionY - yPosition;
95
96          lastPositionX = xPosition;
97          lastPositionY = yPosition;
98
99          sceneManager.scenes[sceneManager.ID]->cameras[0].HandleMouseMovment(xMouseOf
                ↳ yMouseOffset);
100         sceneManager.scenes[sceneManager.ID]->cameras[0].Recalculate();
101     }
102 }
103
104 void keyCallback(GLFWwindow* window, int key, int scancode, int ↙
        ↳ action, int mods)
105 {
106     if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
107     {
108         double mouseX, mouseY;
109
110         glfwGetCursorPos(window, &mouseX, &mouseY);
111
112         sceneManager.scenes[sceneManager.ID]->mouseX = mouseX;
113         sceneManager.scenes[sceneManager.ID]->mouseY = mouseY;
114
115         sceneManager.ID = 0;
116         glfwSetCursorPos(window, 0, 0);
```

```
117             cursorPositionCallback(window, 0, 0);
118         }
119 
120         if (sceneManager.ID == 1 || sceneManager.ID == 2)
121         {
122             if (key == GLFW_KEY_B && action == GLFW_PRESS)
123             {
124                 switch (sceneManager.scenes[sceneManager.ID]->bloom)
125                 {
126                     case true:
127                         sceneManager.scenes[sceneManager.ID]->bloom = false;
128                         break;
129                     case false:
130                         sceneManager.scenes[sceneManager.ID]->bloom = true;
131                         break;
132                 }
133             }
134             if (key == GLFW_KEY_G && action == GLFW_PRESS)
135             {
136                 switch ↙
                    ↳ (sceneManager.scenes[sceneManager.ID]->gammaCorrection)
137                 {
138                     case true:
139                         sceneManager.scenes[sceneManager.ID]->gammaCorrection ↙
                            ↳ = false;
140                         break;
141                     case false:
142                         sceneManager.scenes[sceneManager.ID]->gammaCorrection ↙
                            ↳ = true;
143                         break;
144                 }
145             }
146             if (key == GLFW_KEY_F && action == GLFW_PRESS)
147             {
148                 switch (sceneManager.scenes[sceneManager.ID]->flashlight)
149                 {
150                     case true:
151                         sceneManager.scenes[sceneManager.ID]->flashlight = ↙
                            ↳ false;
152                         break;
153                     case false:
154                         sceneManager.scenes[sceneManager.ID]->flashlight = ↙
                            ↳ true;
155                         break;
156                 }
157             }
158             if (key == GLFW_KEY_H && action == GLFW_PRESS)
```

```
159         {
160             switch (sceneManager.scenes[sceneManager.ID]->hdr)
161             {
162                 case true:
163                     sceneManager.scenes[sceneManager.ID]->hdr = false;
164                     break;
165                 case false:
166                     sceneManager.scenes[sceneManager.ID]->hdr = true;
167                     break;
168             }
169         }
170     }
171 }
172
173 void MouseButtonCallback(GLFWwindow* window, int button, int action, ↙
        ↳ int mods)
174 {
175     std::vector<Button> buttons = ↙
          ↳ sceneManager.scenes[sceneManager.ID]->buttons;
176
177     for (int i = 0; i < buttons.size(); i++)
178     {
179         if (buttons[i].state == HOVER && i == 0 && button == ↙
              ↳ GLFW_MOUSE_BUTTON_LEFT)
180         {
181             sceneManager.ID = 1;
182             lastPositionX = ↙
                  ↳ sceneManager.scenes[sceneManager.ID]->mouseX;
183             lastPositionY = ↙
                  ↳ sceneManager.scenes[sceneManager.ID]->mouseY;
184             glfwSetCursorPos(window, ↙
                  ↳ sceneManager.scenes[sceneManager.ID]->mouseX, ↙
                  ↳ sceneManager.scenes[sceneManager.ID]->mouseY);
185         }
186         if (buttons[i].state == HOVER && i == 1 && button == ↙
              ↳ GLFW_MOUSE_BUTTON_LEFT)
187         {
188             sceneManager.ID = 2;
189             lastPositionX = ↙
                  ↳ sceneManager.scenes[sceneManager.ID]->mouseX;
190             lastPositionY = ↙
                  ↳ sceneManager.scenes[sceneManager.ID]->mouseY;
191             glfwSetCursorPos(window, ↙
                  ↳ sceneManager.scenes[sceneManager.ID]->mouseX, ↙
                  ↳ sceneManager.scenes[sceneManager.ID]->mouseY);
192         }
```

```
193             if (buttons[i].state == HOVER && i == 2 && button == ↙
                    ↳ GLFW_MOUSE_BUTTON_LEFT)
194             {
195                 glfwSetWindowShouldClose(window, 1);
196             }
197         }
198     }
```

3.2. SceneManager

W nim posiadamy opcję zamknięcia wywoływaną przez main w momencie naciśnięcia przycisku oraz informację jako ID, która identyfikuję scenę aktualnie wybraną i która będzie wyświetlana. W konstruktorze tworzymy pozostałe sceny co warte zaznaczenia każda scena dziedziczy po generycznej scenie dzięki czemu możemy stworzyć jedną tablicę w której posiadamy wszystkie możliwe sceny, które mogą posiadać dodatkowe funkcję, które działają tylko na jej obszarze.

```
1   #include "SceneManager.h"
2
3   void Quit(GLFWwindow* window)
4   {
5       glfwSetWindowShouldClose(window, 1);
6   }
7
8   SceneManager::SceneManager(GLuint currentScene)
9   {
10      ID = currentScene;
11
12      Menu* menu = new Menu();
13      Example* example = new Example();
14      AdvanceExample* advanceExample = new AdvanceExample();
15
16      scenes.push_back(menu);
17      scenes.push_back(example);
18      scenes.push_back(advanceExample);
19  }
20
21  void SceneManager::Render(GLFWwindow* window, float deltaTime)
22  {
23      switch (ID)
24      {
25          case 0:
26              scenes[0]->Render(window, deltaTime);
27              break;
```

```
28          case 1:
29              scenes[1]->Render(window, deltaTime);
30              break;
31          case 2:
32              scenes[2]->Render(window, deltaTime);
33              break;
34      }
35  }
```

3.3. Shader

Zacznijmy od funkcji typu void, których nazwy zaczynają się na Set odpowiadają one za ustawienie wartości na karcie graficznej w jednostce cieniującej wywołanej przez ten shader. Oprócz tego mamy odczytanie plików .vert oraz .frag po czym następuję ich złączenie i utworzenie programu wykorzystywanego później do renderowania na karcie graficznej sceny.

```
1   #include "Shader.h"
2
3   Shader::Shader(const char* vertexFile, const char* fragmentFile)
4   {
5       std::string vertexCode = getFileContents(vertexFile);
6       std::string fragmentCode = getFileContents(fragmentFile);
7
8       const char* vertexSource = vertexCode.c_str();
9       const char* fragmentSource = fragmentCode.c_str();
10
11      GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
12      GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
13
14      glShaderSource(vertexShader, 1, &vertexSource, NULL);
15      glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
16
17      glCompileShader(vertexShader);
18      compileErrors(vertexShader, "VERTEX");
19      glCompileShader(fragmentShader);
20      compileErrors(fragmentShader, "FRAGMENT");
21
22      ID = glCreateProgram();
23
24      glAttachShader(ID, vertexShader);
25      glAttachShader(ID, fragmentShader);
26
27      glLinkProgram(ID);
```

```
28      compileErrors(ID, "PROGRAM");
29
30      glDeleteShader(vertexShader);
31      glDeleteShader(fragmentShader);
32  }
33
34  void Shader::Activate()
35  {
36      glUseProgram(ID);
37  }
38
39  void Shader::Delete()
40  {
41      glDeleteProgram(ID);
42  }
43
44  void Shader::SetInt(const std::string& name, int value)
45  {
46      glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
47  }
48
49  void Shader::SetFloat(const std::string& name, float value)
50  {
51      glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
52  }
53
54  void Shader::SetVec3(const std::string& name, glm::vec3& value)
55  {
56      glUniform3fv(glGetUniformLocation(ID, name.c_str()), 1, &value[0]);
57  }
58
59  void Shader::SetVec3(const std::string& name, float v1, float v2, ↙
        ↳ float v3)
60  {
61      glUniform3f(glGetUniformLocation(ID, name.c_str()), v1, v2, v3);
62  }
63
64  void Shader::SetVec2(const std::string& name, glm::vec2& value)
65  {
66      glUniform2fv(glGetUniformLocation(ID, name.c_str()), 1, &value[0]);
67  }
68
69  void Shader::SetVec2(const std::string& name, float v1, float v2)
70  {
71      glUniform2f(glGetUniformLocation(ID, name.c_str()), v1, v2);
72  }
73
```

```
74 void Shader::SetMat4(const std::string& name, glm::mat4& value)
75 {
76     glUniformMatrix4fv(glGetUniformLocation(ID, name.c_str()), 1, ↙
       ↳ GL_FALSE, &value[0][0]);
77 }
78
79 std::string Shader::getFileContents(const char* filename)
80 {
81     std::ifstream fileInput(filename, std::ios::binary);
82     if (fileInput)
83     {
84         std::string contents;
85         fileInput.seekg(0, std::ios::end);
86         contents.resize(fileInput.tellg());
87         fileInput.seekg(0, std::ios::beg);
88         fileInput.read(&contents[0], contents.size());
89         fileInput.close();
90         return contents;
91     }
92     throw(errno);
93 }
94
95 void Shader::compileErrors(unsigned int shader, const char* type)
96 {
97     GLint hasCompiled;
98     char infoLog[1024];
99     if (type != "PROGRAM")
100    {
101        glGetShaderiv(shader, GL_COMPILE_STATUS, &hasCompiled);
102        if (hasCompiled == GL_FALSE)
103        {
104            glGetShaderInfoLog(shader, 1024, NULL, infoLog);
105            std::cout << "SHADER_COMPILATION_ERROR␣for:" << type << ↙
               ↳ "\n" << infoLog << std::endl;
106        }
107    }
108    else
109    {
110        glGetProgramiv(shader, GL_LINK_STATUS, &hasCompiled);
111        if (hasCompiled == GL_FALSE)
112        {
113            glGetProgramInfoLog(shader, 1024, NULL, infoLog);
114            std::cout << "SHADER_LINKING_ERROR␣for:" << type << "\n" ↙
               ↳ << infoLog << std::endl;
115        }
116    }
117 }
```

3.4. Model

W modelu mamy wykorzystanie assimpa na zasadzie załadowania pliku podczas tworzenia konstruktora klasy następnie musimy przejść przez strukturę assimpa która jest zrobiona na zasadzie wskaźników oraz hierarchi zaczynającej się od root node'a oraz sceny. W scenie jest dużo potrzebnych informacji jednak te możemy otrzymać tylko po otrzymaniu odpowiednich wskaźników. Traverse Tree node służy właśnie do rekursywnego odczytywania danych z załadowanego formatu pliku. Warto zaznaczyć, że flagi przy ładowaniu importera są bitmaskami dzięki czemu możemy je łączyć za pomocą pojedynczego operatora bitowego |. W Draw mamy przeniesie kompetencji na poszczególne meshes, który możemy być więcej w jednym modelu jeśli na przykład grafik nie złączył modeli w blenderze przed exportem. Weights posiada dwa przypisania standardowe oraz te faktyczne z pliku jest to zastosowane w celu przejścia shadera przez maksymalnie zdefiniowaną ilości kości w modelu ponieważ shader nie posiada dynamicznych tablic tak jak C++ lub inny język programistyczny. W setupie tekstur należy zwrócić uwagę z jakim plikiem mamy do czynienia gdy używamy spakowanego .fbx posiadamy już dane w tym pliku i nie potrzebujemy wczytywać z innych plików gdyż .fbx posiada węwnątrz siebie wszystkie potrzebne informację dotyczące wielkości oraz danych o pixelach w teksturze. Kolejne warte zaznaczenie jest faktu, że musimy zmienić dane wewnętrzne tekstury podczas wpisywania danych do bufora by poprawnie przeprowadzić gamma correction.

```
1  #include"Model.h"
2
3  Model::Model(std::string path)
4  {
5      Assimp::Importer importer;
6
7      const aiScene* scene = importer.ReadFile(path, aiProcess_FlipUVs | ↙
           ↳ aiProcess_Triangulate);
8
9      if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || ↙
           ↳ !scene->mRootNode)
10     {
11         std::cout << "Error␣Assimp:␣" << importer.GetErrorString() << ↙
               ↳ std::endl;
12         return;
13     }
14
15     diretory = path.substr(0, path.find_last_of('/'));
16
17     traverseNodes(scene->mRootNode, scene);
18 }
19
```

```
20 void Model::Draw(Shader& shader, bool gammaCorrected)
21 {
22     for (int i = 0; i < meshes.size(); i++)
23     {
24         meshes[i].Draw(shader, gammaCorrected);
25     }
26 }
27
28 void Model::SetVertexBoneWeightToDefault(Vertex& vertex)
29 {
30     for (int i = 0; i < MAX_BONE_INFLUENCE; i++)
31     {
32         vertex.boneIds[i] = -1;
33         vertex.weights[i] = 0.0f;
34     }
35 }
36
37 void Model::ExtractBoneWeights(std::vector<Vertex>& vertices, aiMesh* ↙
    ↳ mesh, const aiScene* scene)
38 {
39     for (int i = 0; i < mesh->mNumBones; i++)
40     {
41         int boneID = -1;
42         std::string boneName = mesh->mBones[i]->mName.C_Str();
43         if (bonesInfo.find(boneName) == bonesInfo.end())
44         {
45             BoneInfo boneInfo;
46             boneInfo.ID = currentBone;
47             boneInfo.offset = ↙
                ↳ glm::mat4(mesh->mBones[i]->mOffsetMatrix.a1, ↙
                ↳ mesh->mBones[i]->mOffsetMatrix.b1, ↙
                ↳ mesh->mBones[i]->mOffsetMatrix.c1, ↙
                ↳ mesh->mBones[i]->mOffsetMatrix.d1,
48                 mesh->mBones[i]->mOffsetMatrix.a2, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.b2, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.c2, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.d2,
49                 mesh->mBones[i]->mOffsetMatrix.a3, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.b3, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.c3, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.d3,
50                 mesh->mBones[i]->mOffsetMatrix.a4, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.b4, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.c4, ↙
                    ↳ mesh->mBones[i]->mOffsetMatrix.d4);
51             bonesInfo[boneName] = boneInfo;
52             boneID = currentBone;
```

```
53              currentBone++;
54          }
55          else
56          {
57              boneID = bonesInfo[boneName].ID;
58          }
59          if (boneID == -1)
60          {
61              std::cout << "ERROR::BONE:␣Extracton␣of␣data␣failed!";
62              abort();
63          }
64          aiVertexWeight* weights = mesh->mBones[i]->mWeights;
65          GLuint numberOfWeights = mesh->mBones[i]->mNumWeights;
66
67          for (int j = 0; j < numberOfWeights; j++)
68          {
69              unsigned int vertexID = weights[j].mVertexId;
70              float weight = weights[j].mWeight;
71
72              for (int k = 0; k < MAX_BONE_INFLUENCE; k++)
73              {
74                  if (vertices[vertexID].boneIds[k] < 0)
75                  {
76                      vertices[vertexID].boneIds[k] = boneID;
77                      vertices[vertexID].weights[k] = weight;
78                      break;
79                  }
80              }
81          }
82      }
83  }
84
85  void Model::traverseNodes(aiNode* node, const aiScene* scene)
86  {
87      for (int i = 0; i < node->mNumMeshes; i++)
88      {
89          aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
90
91          processMesh(mesh, scene);
92      }
93
94      for (int i = 0; i < node->mNumChildren; i++)
95      {
96          traverseNodes(node->mChildren[i], scene);
97      }
98  }
99
```

```
100 void Model::processMesh(aiMesh* mesh, const aiScene* scene)
101 {
102     std::vector<Vertex> vertices;
103     std::vector<GLuint> indicies;
104     glm::vec3 position;
105     glm::vec3 color;
106     glm::vec3 normal;
107     glm::vec2 textureCoordinates;
108 
109     for (int i = 0; i < mesh->mNumVertices; i++)
110     {
111         Vertex vertex;
112 
113         SetVertexBoneWeightToDefault(vertex);
114 
115         position.x = mesh->mVertices[i].x;
116         position.y = mesh->mVertices[i].y;
117         position.z = mesh->mVertices[i].z;
118 
119         if (mesh->HasVertexColors(i))
120         {
121             color.x = mesh->mColors[i]->r;
122             color.y = mesh->mColors[i]->g;
123             color.z = mesh->mColors[i]->b;
124             vertex.color = color;
125         }
126 
127         if (mesh->HasNormals())
128         {
129             normal.x = mesh->mNormals[i].x;
130             normal.y = mesh->mNormals[i].y;
131             normal.z = mesh->mNormals[i].z;
132             vertex.normal = normal;
133         }
134 
135         if (mesh->HasTextureCoords(0))
136         {
137             textureCoordinates.x = mesh->mTextureCoords[0][i].x;
138             textureCoordinates.y = mesh->mTextureCoords[0][i].y;
139             vertex.textureCoordinates = textureCoordinates;
140         }
141 
142         vertex.position = position;
143 
144         vertices.push_back(vertex);
145     }
146 
```

```
147     for (GLuint i = 0; i < mesh->mNumFaces; i++)
148     {
149         aiFace face = mesh->mFaces[i];
150         indicies.push_back(face.mIndices[0]);
151         indicies.push_back(face.mIndices[1]);
152         indicies.push_back(face.mIndices[2]);
153     }
154 
155     aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
156 
157     setupTextures(aiTextureType_DIFFUSE, material, scene);
158     setupTextures(aiTextureType_SPECULAR, material, scene);
159 
160     ExtractBoneWeights(vertices, mesh, scene);
161 
162     meshes.push_back(Mesh(vertices, indicies, textures, ↙
        ↳ gammaCorrectedTextures));
163 }
164 
165 void Model::setupTextures(aiTextureType type, aiMaterial* material, ↙
    ↳ const aiScene* scene)
166 {
167     for (int i = 0; i < material->GetTextureCount(type); i++)
168     {
169         Texture texture;
170         GLuint id, id2;
171 
172         aiString path;
173         material->GetTexture(type, i, &path);
174 
175         int width, height, numberOfChannels;
176 
177         unsigned char* imageData = stbi_load((diretory + "/" + ↙
            ↳ path.C_Str()).c_str(), &width, &height, ↙
            ↳ &numberOfChannels, 0);
178 
179         const aiTexture* embendedTexture = ↙
            ↳ scene->GetEmbeddedTexture(path.C_Str());
180         if (embendedTexture)
181         {
182             imageData = stbi_load_from_memory((const ↙
                ↳ stbi_uc*)embendedTexture->pcData, ↙
                ↳ embendedTexture->mWidth, &width, &height, ↙
                ↳ &numberOfChannels, 0);
183         }
184 
185         GLenum dataFormat;
```

```
186         switch (numberOfChannels)
187         {
188         case 1:
189             dataFormat = GL_RED;
190             break;
191         case 3:
192             dataFormat = GL_RGB;
193             break;
194         case 4:
195             dataFormat = GL_RGBA;
196             break;
197         }
198 
199         glGenTextures(1, &id);
200         glBindTexture(GL_TEXTURE_2D, id);
201         glGenerateMipmap(GL_TEXTURE_2D);
202 
203         glTexImage2D(GL_TEXTURE_2D, 0, dataFormat, width, height, 0, ↙
                ↳ dataFormat, GL_UNSIGNED_BYTE, imageData);
204 
205         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, ↙
                ↳ GL_LINEAR);
206         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, ↙
                ↳ GL_LINEAR);
207 
208         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
209         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
210 
211         texture.ID = id;
212         texture.type = std::string(path.C_Str()).find_last_of('.');
213 
214         textures.push_back(texture);
215 
216         GLenum internalFormat;
217         switch (dataFormat)
218         {
219             case GL_RGB:
220                 internalFormat = GL_SRGB;
221                 break;
222             case GL_RGBA:
223                 internalFormat = GL_SRGB_ALPHA;
224                 break;
225         }
226 
227         glGenTextures(1, &id2);
228         glBindTexture(GL_TEXTURE_2D, id2);
229         glGenerateMipmap(GL_TEXTURE_2D);
```

```
230
231         glTexImage2D(GL_TEXTURE_2D, 0, internalFormat, width, height, ↙
                ↳ 0, dataFormat, GL_UNSIGNED_BYTE, imageData);
232
233         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, ↙
                ↳ GL_LINEAR);
234         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, ↙
                ↳ GL_LINEAR);
235
236         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
237         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
238
239         texture.ID = id2;
240         texture.type = std::string(path.C_Str()).find_last_of('.');
241
242         gammaCorrectedTextures.push_back(texture);
243
244         stbi_image_free(imageData);
245     }
246 }
```

3.5. Mesh

Posiada konstruktor, który najczęściej wywołuje klasa model w celu przepisania danych do OpenGL. Pamiętać należy o kolejności bindowania elementów. Najpierw VAO potem EBO i VBO dlatego też tylko VAO jest w pliku nagłówkowym klasy z racji iż później w klasie rysującej musimy się odwołać do niego przez wywołanie funkcji `glDrawElements`. Ostatnią funkcją jest wyliczanie dla sceny Menu danych dotyczących myszy gdyż potrzebujemy jak było wspomniane w teorii dwóch punktów o maksymalnych wartościach w dwóch osiach.

```
1  #include "Mesh.h"
2
3  Mesh::Mesh(std::vector<Vertex> Vertices, std::vector<GLuint> Indices, ↙
       ↳ std::vector<Texture> Textures, std::vector<Texture> ↙
       ↳ GammaCorrectedTextures)
4      :vertices(Vertices), indices(Indices), textures(Textures), ↙
           ↳ gammaCorrectedTextures(GammaCorrectedTextures)
5  {
6      GLuint VBO, EBO;
7      glGenVertexArrays(1, &VAO);
8      glBindVertexArray(VAO);
9
10     glGenBuffers(1, &VBO);
```

```
11      glBindBuffer(GL_ARRAY_BUFFER, VBO);
12
13      glGenBuffers(1, &EBO);
14      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
15
16      glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex), ↙
            ↳ &vertices[0], GL_STATIC_DRAW);
17
18      glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * ↙
            ↳ sizeof(GLuint), &indices[0], GL_STATIC_DRAW);
19
20      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), ↙
            ↳ (void*) 0);
21      glEnableVertexAttribArray(0);
22
23      glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), ↙
            ↳ (void*)offsetof(Vertex, color));
24      glEnableVertexAttribArray(1);
25
26      glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), ↙
            ↳ (void*)offsetof(Vertex, normal));
27      glEnableVertexAttribArray(2);
28
29      glVertexAttribPointer(3, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), ↙
            ↳ (void*)offsetof(Vertex, textureCoordinates));
30      glEnableVertexAttribArray(3);
31
32      glVertexAttribIPointer(4, 4, GL_INT, sizeof(Vertex), ↙
            ↳ (void*)offsetof(Vertex, boneIds));
33      glEnableVertexAttribArray(4);
34
35      glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), ↙
            ↳ (void*)offsetof(Vertex, weights));
36      glEnableVertexAttribArray(5);
37
38      glBindVertexArray(0);
39  }
40
41  void Mesh::Draw(Shader& shader, bool gammaCorrected)
42  {
43      switch (gammaCorrected)
44      {
45          case true:
46              for (int i = 0; i < gammaCorrectedTextures.size(); i++)
47              {
48                  glActiveTexture(GL_TEXTURE0 + i);
```

```
49                  shader.SetInt("material." + ↙
                        ↳ gammaCorrectedTextures[i].type, i);
50
51                  glBindTexture(GL_TEXTURE_2D, ↙
                        ↳ gammaCorrectedTextures[i].ID);
52              }
53              break;
54          case false:
55              for (int i = 0; i < textures.size(); i++)
56              {
57                  glActiveTexture(GL_TEXTURE0 + i);
58                  shader.SetInt("material." + textures[i].type, i);
59
60                  glBindTexture(GL_TEXTURE_2D, textures[i].ID);
61              }
62              break;
63      }
64      glBindVertexArray(VAO);
65      glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
66      glBindVertexArray(0);
67  }
68
69  float Mesh::CalculateCursorOffsetY()
70  {
71      float sumOfVerticesY = 0, max = -INFINITY;
72
73      for (int i = 0; i < vertices.size(); i++)
74      {
75          if (max < vertices[i].position.y) max = vertices[i].position.y;
76          sumOfVerticesY += vertices[i].position.y;
77      }
78
79      return max - sumOfVerticesY / 2;
80  }
81
82  float Mesh::GetMaxPositionX()
83  {
84      float max = -INFINITY;
85
86      for (int i = 0; i < vertices.size(); i++)
87      {
88          if (max < vertices[i].position.x) max = vertices[i].position.x;
89      }
90
91      return max;
92  }
93
```

```
94  float Mesh::GetMinPositionX()
95  {
96      float min = INFINITY;
97
98      for (int i = 0; i < vertices.size(); i++)
99      {
100         if (min > vertices[i].position.x) min = vertices[i].position.x;
101     }
102
103     return min;
104 }
105
106 float Mesh::GetMaxPositionY()
107 {
108     float max = -INFINITY;
109
110     for (int i = 0; i < vertices.size(); i++)
111     {
112         if (max < vertices[i].position.y) max = vertices[i].position.y;
113     }
114
115     return max;
116 }
117
118 float Mesh::GetMinPositionY()
119 {
120     float min = INFINITY;
121
122     for (int i = 0; i < vertices.size(); i++)
123     {
124         if (min > vertices[i].position.y) min = vertices[i].position.y;
125     }
126
127     return min;
128 }
```

3.6. Button

Przyciski dostępne w menu ich aktualne stany i aktualizacja stanów wraz z renderowaniem.

```
1  #include "Button.h"
2
3  Button::Button(glm::vec3 Position)
4  {
```

```
5       state = UNHOVER;
6       position = Position;
7       color = glm::vec3(0.5f, 0.5f, 0.5f);
8       buttonModel = Model("Models/Button/button.obj");
9
10      maxX = buttonModel.meshes[0].GetMaxPositionX();
11      minX = buttonModel.meshes[0].GetMinPositionX();
12      maxY = buttonModel.meshes[0].GetMaxPositionY();
13      minY = buttonModel.meshes[0].GetMinPositionY();
14  }
15
16  void Button::UpdateState(glm::vec2 normalizeMousePosition)
17  {
18      if (normalizeMousePosition.x >= minX + position.x &&
            ↪ normalizeMousePosition.x <= maxX + position.x &&
19          -normalizeMousePosition.y >= minY + position.y &&
                ↪ -normalizeMousePosition.y <= maxY + position.y)
20      {
21          state = HOVER;
22          return;
23      }
24      state = UNHOVER;
25  }
26
27  void Button::Render(Shader shader)
28  {
29      glm::mat4 model = glm::translate(glm::mat4(1.0f), position);
30
31      shader.SetMat4("model", model);
32
33      switch (state)
34      {
35          case HOVER:
36              shader.SetVec3("color", 0.7f, 0.7f, 0.7f);
37              break;
38          case UNHOVER:
39              shader.SetVec3("color", 0.5f, 0.5f, 0.5f);
40              break;
41      }
42
43      buttonModel.Draw(shader, false);
44  }
```

3.7. Camera

W niej znajduję się manipulacja światem, która ma sprawie wrażenie że jesteśmy bezcielesną kamerą gdy w zasadzie za pomocą matmy jesteśmy w stanie na bieżąco z mieniać w świecie pozycje wszystkich obiektów dzięki czemu wydaje nam się, że to my się ruszamy gdy w zasadzie rusza się świat.

```
#include "Camera.h"

Camera::Camera(glm::vec3 Position)
{
    position = Position;
}

void Camera::Update()
{
    view = glm::lookAt(position, position + front, up);
}

void Camera::HandleMouseMovment(float xOffset, float yOffset)
{
    xOffset *= sensivity;
    yOffset *= sensivity;

    yaw += xOffset;
    pitch += yOffset;

    if (pitch > 89.0f) pitch = 89.0f;
    if (pitch < -89.0f) pitch = -89.0f;
}

void Camera::Recalculate()
{
    glm::vec3 Front;
    Front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    Front.y = sin(glm::radians(pitch));
    Front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));

    front = glm::normalize(Front);
    up = glm::normalize(glm::cross(glm::cross(front, worldUp), front));
}

void Camera::HandleEvents(GLFWwindow* window, float deltaTime)
{
    float movment = speed * deltaTime;
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    {
```

```
41          position += movment * front;
42      }
43      if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
44      {
45          position -= movment * front;
46      }
47      if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
48      {
49          position += glm::normalize(glm::cross(front, up)) * movment;
50      }
51      if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
52      {
53          position -= glm::normalize(glm::cross(front, up)) * movment;
54      }
55  }
```

3.8. Bone

Kość wczytanego modelu posiadająca wyliczenia interpolacji.

```
1   #include "Bone.h"
2
3   Bone::Bone(std::string Name, int id, const aiNodeAnim* channel)
4       : name(Name), ID(id), localTransformation(1.0f)
5   {
6       numberOfPositions = channel->mNumPositionKeys;
7
8       for (int i = 0; i < numberOfPositions; ++i)
9       {
10          aiVector3D aiPosition = channel->mPositionKeys[i].mValue;
11          float timeStamp = channel->mPositionKeys[i].mTime;
12          KeyPosition data;
13          data.position.x = aiPosition.x;
14          data.position.y = aiPosition.y;
15          data.position.z = aiPosition.z;
16          data.timeStamp = timeStamp;
17          positions.push_back(data);
18      }
19
20      numberOfRotations = channel->mNumRotationKeys;
21
22      for (int i = 0; i < numberOfRotations; ++i)
23      {
24          aiQuaternion aiOrientation = channel->mRotationKeys[i].mValue;
25          float timeStamp = channel->mRotationKeys[i].mTime;
```

```
26          KeyRotation data;
27          data.orientation.w = aiOrientation.w;
28          data.orientation.x = aiOrientation.x;
29          data.orientation.y = aiOrientation.y;
30          data.orientation.z = aiOrientation.z;
31          data.timeStamp = timeStamp;
32          rotations.push_back(data);
33      }
34
35      numberOfScales = channel->mNumScalingKeys;
36
37      for (int i = 0; i < numberOfScales; ++i)
38      {
39          aiVector3D aiPosition = channel->mScalingKeys[i].mValue;
40          float timeStamp = channel->mScalingKeys[i].mTime;
41          KeyScale data;
42          data.scale.x = aiPosition.x;
43          data.scale.y = aiPosition.y;
44          data.scale.z = aiPosition.z;
45          data.timeStamp = timeStamp;
46          scales.push_back(data);
47      }
48  }
49
50  void Bone::Update(float animationTime)
51  {
52      glm::mat4 translation = InterpolatePosition(animationTime);
53      glm::mat4 rotation = InterpolateRotation(animationTime);
54      glm::mat4 scale = InterpolateScale(animationTime);
55      localTransformation = translation * rotation * scale;
56  }
57
58  float Bone::getScaleFactor(float lastTimeStamp, float nextTimeStamp, ↙
        ↳ float animationTime)
59  {
60      float scaleFactor = 0.0f;
61      float midWayLength = animationTime - lastTimeStamp;
62      float frameDiffrence = nextTimeStamp - lastTimeStamp;
63      scaleFactor = midWayLength / frameDiffrence;
64      return scaleFactor;
65  }
66
67  unsigned int Bone::getPositionIndex(float animationTime)
68  {
69      for (int i = 0; i < numberOfPositions - 1; ++i)
70      {
71          if (animationTime < positions[i + 1].timeStamp) return i;
```

```
72      }
73  }
74
75  unsigned int Bone::getRotationIndex(float animationTime)
76  {
77      for (int i = 0; i < numberOfRotations - 1; ++i)
78      {
79          if (animationTime < rotations[i + 1].timeStamp) return i;
80      }
81
82  }
83
84  unsigned int Bone::getScaleIndex(float animationTime)
85  {
86      for (int i = 0; i < numberOfScales - 1; ++i)
87      {
88          if (animationTime < scales[i + 1].timeStamp) return i;
89      }
90  }
91
92  glm::mat4 Bone::InterpolatePosition(float animationTime)
93  {
94      if (numberOfPositions == 1) return glm::translate(glm::mat4(1.0f), ↙
            ↳ positions[0].position);
95      int startingPositionIndex = getPositionIndex(animationTime);
96      int endingPositionIndex = startingPositionIndex + 1;
97      float scaleFactor = ↙
            ↳ getScaleFactor(positions[startingPositionIndex].timeStamp, ↙
            ↳ positions[endingPositionIndex].timeStamp, animationTime);
98      glm::vec3 finalPosition = ↙
            ↳ glm::mix(positions[startingPositionIndex].position, ↙
            ↳ positions[endingPositionIndex].position, scaleFactor);
99      return glm::translate(glm::mat4(1.0f), finalPosition);
100 }
101
102 glm::mat4 Bone::InterpolateRotation(float animationTime)
103 {
104     if (numberOfRotations == 1)
105     {
106         glm::quat rotation = glm::normalize(rotations[0].orientation);
107         return glm::toMat4(rotation);
108     }
109     int startingRotationIndex = getRotationIndex(animationTime);
110     int endingRotationIndex = startingRotationIndex + 1;
111     float scaleFactor = ↙
            ↳ getScaleFactor(rotations[startingRotationIndex].timeStamp, ↙
            ↳ rotations[endingRotationIndex].timeStamp, animationTime);
```

```
112     glm::quat finalRotation = ↙
            ↳ glm::normalize(glm::slerp(rotations[startingRotationIndex].orientation, ↙
            ↳ rotations[endingRotationIndex].orientation, scaleFactor));
113     return glm::toMat4(finalRotation);
114 }
115
116 glm::mat4 Bone::InterpolateScale(float animationTime)
117 {
118     if (numberOfScales == 1) return glm::scale(glm::mat4(1.0f), ↙
            ↳ scales[0].scale);
119     int startingScaleIndex = getScaleIndex(animationTime);
120     int endingScaleIndex = startingScaleIndex + 1;
121     float scaleFactor = ↙
            ↳ getScaleFactor(scales[startingScaleIndex].timeStamp, ↙
            ↳ scales[endingScaleIndex].timeStamp, animationTime);
122     glm::vec3 finalScale = glm::mix(scales[startingScaleIndex].scale, ↙
            ↳ scales[endingScaleIndex].scale, scaleFactor);
123     return glm::scale(glm::mat4(1.0f), finalScale);
124 }
```

3.9. Animator

Kontroler animacji polega na przekazaniu pracy odpowiednim podklasą takim jak animacjami to w nim mamy informację o aktualnej animacji i to w nim rekursywnie odwracamy z przestrzeni kości na lokalnej o czym było wspomniane w teorii.

```
1  #include "Animator.h"
2
3  Animator::Animator(Animation Animation)
4      :currentTime(0.0f)
5  {
6      animation = Animation;
7      for (int i = 0; i < 100; i++)
8      {
9          finalBoneMatrices.push_back(glm::mat4(1.0f));
10     }
11 }
12
13 void Animator::UpdateAnimation(float deltaTime)
14 {
15     if (&animation)
16     {
17         currentTime += animation.ticksPerSecond * deltaTime;
18         currentTime = fmod(currentTime, animation.duration);
```

```
19          CalculateFinalBoneMatrices(&animation.rootNode, ↙
                ↳ glm::mat4(1.0f));
20      }
21  }
22
23  void Animator::CalculateFinalBoneMatrices(const AssimpNodeData* node, ↙
        ↳ glm::mat4 parentMatrice)
24  {
25      std::string nodeName = node->name;
26      glm::mat4 nodeTransform = node->transformation;
27
28      Bone* Bone = animation.FindBone(nodeName);
29
30      if (Bone)
31      {
32          Bone->Update(currentTime);
33          nodeTransform = Bone->localTransformation;
34      }
35
36      glm::mat4 globalTransformation = parentMatrice * nodeTransform;
37
38      std::map<std::string, BoneInfo> boneInfoMap = animation.boneInfo;
39
40      if (boneInfoMap.find(nodeName) != boneInfoMap.end())
41      {
42          int ID = boneInfoMap[nodeName].ID;
43          glm::mat4 offset = boneInfoMap[nodeName].offset;
44          finalBoneMatrices[ID] = globalTransformation * offset;
45      }
46
47      for (int i = 0; i < node->childrenCount; i++)
48          CalculateFinalBoneMatrices(&node->children[i], ↙
                ↳ globalTransformation);
49  }
50
51  void Animator::PlayAnimation(Animation Animation)
52  {
53      animation = Animation;
54      currentTime = 0.0f;
55  }
```

3.10. Animation

Animacja punkt w którym czytam z pliku assimpa a następnie interpretujemy otrzymane dane warto zaznaczyć że biblioteka do matmy glm oraz assimp mają różne

formaty zapisu quaternionów oraz matryc przez co należy je w odpowiedni sposób przekonwertować. Dodatkowo to w niej mamy odwołanie Animatora jeśli chodzi o pojedyncze kości, które należy zmodyfikować w animacji szkieletowej.

```
1  #include "Animation.h"
2
3  Animation::Animation(std::string animationPath, Model* model)
4  {
5      Assimp::Importer importer;
6      const aiScene* scene = importer.ReadFile(animationPath, ↙
           ↳ aiProcess_Triangulate);
7      aiAnimation* animation = scene->mAnimations[0];
8      duration = animation->mDuration;
9      ticksPerSecond = animation->mTicksPerSecond;
10     aiMatrix4x4 globalTransformation = ↙
           ↳ scene->mRootNode->mTransformation;
11     globalTransformation = globalTransformation.Inverse();
12     ReadHierarchy(rootNode, scene->mRootNode);
13     ReadMissingBones(animation, *model);
14 }
15
16 Bone* Animation::FindBone(std::string name)
17 {
18     std::vector<Bone>::iterator iterator = std::find_if(bones.begin(), ↙
           ↳ bones.end(), [&](const Bone& Bone)
19         {
20             return Bone.name == name;
21         }
22     );
23     if (iterator == bones.end()) return nullptr;
24     else return &(*iterator);
25 }
26
27 void Animation::ReadHierarchy(AssimpNodeData& destination, const ↙
       ↳ aiNode* source)
28 {
29     destination.name = source->mName.data;
30     destination.transformation = glm::mat4(source->mTransformation.a1, ↙
           ↳ source->mTransformation.b1, source->mTransformation.c1, ↙
           ↳ source->mTransformation.d1,
31                                            source->mTransformation.a2, ↙
                                                   ↳ source->mTransformation.b2, ↙
                                                   ↳ source->mTransformation.c2, ↙
                                                   ↳ source->mTransformation.d2,
32                                            source->mTransformation.a3, ↙
                                                   ↳ source->mTransformation.b3, ↙
```

```
                                                    ↳ source->mTransformation.c3, ↙
                                                    ↳ source->mTransformation.d3,
33                                                source->mTransformation.a4, ↙
                                                    ↳ source->mTransformation.b4, ↙
                                                    ↳ source->mTransformation.c4, ↙
                                                    ↳ source->mTransformation.d4);
34      destination.childrenCount = source->mNumChildren;
35      for (int i = 0; i < source->mNumChildren; i++)
36      {
37          AssimpNodeData data;
38          ReadHierarchy(data, source->mChildren[i]);
39          destination.children.push_back(data);
40      }
41  }
42  void Animation::ReadMissingBones(aiAnimation* animation, Model& model)
43  {
44      int size = animation->mNumChannels;
45
46      for (int i = 0; i < size; i++)
47      {
48          aiNodeAnim* channel = animation->mChannels[i];
49          std::string boneName = channel->mNodeName.data;
50
51          if (model.bonesInfo.find(boneName) == model.bonesInfo.end())
52          {
53              model.bonesInfo[boneName].ID = model.currentBone;
54              model.currentBone++;
55          }
56          bones.push_back(Bone(channel->mNodeName.data, ↙
                ↳ model.bonesInfo[channel->mNodeName.data].ID, channel));
57      }
58      boneInfo = model.bonesInfo;
59  }
```

3.11. InGameAnimation

Klasa służąca do animacji interpolacyjny w świecie z lokalnego na świat. Bardzo podobna struktura jak w `Bone.cpp` z wyłączeniem nie posiadania przejścia i czytania z hierarchi oraz plików gdyż wszystko w aktualnej wersji jest pisane po stronie kodu a nie po stronie plików .txt czy .json czy też innych.

```
1  #include "InGameAnimation.h"
2
3  InGameAnimation::InGameAnimation(float Duration, unsigned int ↙
       ↳ TicksPerSecond, std::vector<KeyPositionInGame> Positions, ↙
```

```
    ↪ std::vector<KeyRotationInGame> Rotations, ↙
    ↪ std::vector<KeyScaleInGame> Scales)
    :ticksPerSecond(TicksPerSecond), duration(Duration), positions{ ↙
        ↪ Positions }, rotations{ Rotations }, scales{Scales}, ↙
        ↪ currentTime(0.0f) {}

glm::mat4 InGameAnimation::UpdateAnimation(float deltaTime)
{
    currentTime += ticksPerSecond * deltaTime;
    animationTimePlaying += ticksPerSecond * deltaTime;
    if (animationTimePlaying > duration) endAnimation = true;
    currentTime = fmod(currentTime, duration);
    glm::mat4 translate = InterpolatePosition(currentTime);
    glm::mat4 rotation = InterpolateRotation(currentTime);
    glm::mat4 scale = InterpolateScale(currentTime);
    return translate * rotation * scale;
}

float InGameAnimation::getScaleFactor(float lastTimeStamp, float ↙
    ↪ nextTimeStamp, float animationTime)
{
    float scaleFactor = 0.0f;
    float midWayLength = animationTime - lastTimeStamp;
    float frameDiffrence = nextTimeStamp - lastTimeStamp;
    scaleFactor = midWayLength / frameDiffrence;
    return scaleFactor;
}

unsigned int InGameAnimation::getPositionIndex(float animationTime)
{
    for (int i = 0; i < positions.size() - 1; ++i)
    {
        if (animationTime < positions[i + 1].timeStamp) return i;
    }
}

unsigned int InGameAnimation::getRotationIndex(float animationTime)
{
    for (int i = 0; i < rotations.size() - 1; ++i)
    {
        if (animationTime < rotations[i + 1].timeStamp) return i;
    }
}

unsigned int InGameAnimation::getScaleIndex(float animationTime)
{
    for (int i = 0; i < scales.size() - 1; ++i)
```

```
46      {
47          if (animationTime < scales[i + 1].timeStamp) return i;
48      }
49  }
50
51  glm::mat4 InGameAnimation::InterpolatePosition(float animationTime)
52  {
53      if (positions.size() == 1) return glm::translate(glm::mat4(1.0f), ↙
            ↳ positions[0].position);
54      int startingPositionIndex = getPositionIndex(animationTime);
55      int endingPositionIndex = startingPositionIndex + 1;
56      float scaleFactor = ↙
            ↳ getScaleFactor(positions[startingPositionIndex].timeStamp, ↙
            ↳ positions[endingPositionIndex].timeStamp, animationTime);
57      glm::vec3 finalPosition = ↙
            ↳ glm::mix(positions[startingPositionIndex].position, ↙
            ↳ positions[endingPositionIndex].position, scaleFactor);
58      return glm::translate(glm::mat4(1.0f), finalPosition);
59  }
60
61  glm::mat4 InGameAnimation::InterpolateRotation(float animationTime)
62  {
63      if (rotations.size() == 1)
64      {
65          glm::quat rotation = glm::normalize(rotations[0].orientation);
66          return glm::toMat4(rotation);
67      }
68      int startingRotationIndex = getRotationIndex(animationTime);
69      int endingRotationIndex = startingRotationIndex + 1;
70      float scaleFactor = ↙
            ↳ getScaleFactor(rotations[startingRotationIndex].timeStamp, ↙
            ↳ rotations[endingRotationIndex].timeStamp, animationTime);
71      glm::quat finalRotation = ↙
            ↳ glm::normalize(glm::slerp(rotations[startingRotationIndex].orientation, ↙
            ↳ rotations[endingRotationIndex].orientation, scaleFactor));
72      return glm::toMat4(finalRotation);
73  }
74
75  glm::mat4 InGameAnimation::InterpolateScale(float animationTime)
76  {
77      if (scales.size() == 1) return glm::scale(glm::mat4(1.0f), ↙
            ↳ scales[0].scale);
78      int startingScaleIndex = getScaleIndex(animationTime);
79      int endingScaleIndex = startingScaleIndex + 1;
80      float scaleFactor = ↙
            ↳ getScaleFactor(scales[startingScaleIndex].timeStamp, ↙
            ↳ scales[endingScaleIndex].timeStamp, animationTime);
```

```
81      glm::vec3 finalScale = glm::mix(scales[startingScaleIndex].scale, ↙
            ↳ scales[endingScaleIndex].scale, scaleFactor);
82      return glm::scale(glm::mat4(1.0f), finalScale);
83  }
```

3.12. GenericScene

Jedyna klasa wirtualna będąca nie jako prostym prototypem sceny niż własnoręczną sceną to z niej dziedziczą trzy sceny zawarte w tej wersji silnika.

```
1   #ifndef GENERIC_SCENE_H
2   #define GENERIC_SCENE_H
3
4   #include"../Button.h"
5   #include"../Camera.h"
6   #include"../Animator.h"
7
8   struct Flashlight
9   {
10      glm::vec3 ambient = glm::vec3(0.0f, 0.0f, 0.0f);
11      glm::vec3 diffuse = glm::vec3(10.0f, 10.0f, 10.0f);
12      glm::vec3 specular = glm::vec3(1.0f, 1.0f, 1.0f);
13      float constant = 1.0f;
14      float linear = 0.0014f;
15      float quadratic = 0.000007f;
16  };
17
18  struct PointLight
19  {
20      glm::vec3 position;
21      glm::vec3 ambient;
22      glm::vec3 diffuse;
23      glm::vec3 specular;
24      float constant = 1.0f;
25      float linear;
26      float quadratic;
27  };
28
29  struct Object
30  {
31      glm::mat4 model;
32
33      float strengthOfRotation;
34      glm::vec3 position;
35      glm::vec3 rotation;
```

```
36      glm::vec3 scale;
37  };
38
39  class GenericScene
40  {
41      public:
42          std::vector<Camera> cameras;
43          std::vector<Button> buttons;
44          std::vector<PointLight> pointLights;
45          Flashlight flashlightObject;
46          bool flashlight, gammaCorrection, hdr, bloom;
47          int width = 1920, height = 1080;
48          double mouseX, mouseY;
49
50          GenericScene() {};
51
52          virtual void Render(GLFWwindow* window, float deltaTime) {};
53      protected:
54          std::vector<Animator> animators;
55          std::vector<Shader> shaders;
56          std::vector<Model> models;
57          glm::mat4 projection;
58          GLuint pingpongFBO[2], pingpongColorBuffers[2];
59          GLuint HDR, colorBuffers[2], VAO = 0;
60
61          void SetupPostProcessing()
62          {
63              glGenFramebuffers(1, &HDR);
64              glBindFramebuffer(GL_FRAMEBUFFER, HDR);
65
66              glGenTextures(2, colorBuffers);
67
68              for (int i = 0; i < std::end(colorBuffers) - ↙
                    ↳ std::begin(colorBuffers); i++)
69              {
70                  glBindTexture(GL_TEXTURE_2D, colorBuffers[i]);
71                  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, width, ↙
                        ↳ height, 0, GL_RGBA, GL_FLOAT, NULL);
72                  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, ↙
                        ↳ GL_LINEAR);
73                  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, ↙
                        ↳ GL_LINEAR);
74                  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, ↙
                        ↳ GL_CLAMP_TO_EDGE);
75                  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, ↙
                        ↳ GL_CLAMP_TO_EDGE);
76
```

```
77                  glFramebufferTexture2D(GL_FRAMEBUFFER, ↙
                        ↳ GL_COLOR_ATTACHMENT0 + i, GL_TEXTURE_2D, ↙
                        ↳ colorBuffers[i], 0);
78              }
79
80              GLuint renderDepthObject;
81              glGenRenderbuffers(1, &renderDepthObject);
82              glBindRenderbuffer(GL_RENDERBUFFER, renderDepthObject);
83              glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, ↙
                    ↳ width, height);
84
85              glFramebufferRenderbuffer(GL_FRAMEBUFFER, ↙
                    ↳ GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, ↙
                    ↳ renderDepthObject);
86
87              GLuint attachments[2] = { GL_COLOR_ATTACHMENT0, ↙
                    ↳ GL_COLOR_ATTACHMENT1 };
88
89              glDrawBuffers(2, attachments);
90
91              if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != ↙
                    ↳ GL_FRAMEBUFFER_COMPLETE)
92              {
93                  std::cout << "ERROR::FRAMEBUFFER:␣Main␣framebuffer␣not␣↙
                        ↳ completed!" << std::endl;
94                  abort();
95              }
96
97              glBindFramebuffer(GL_FRAMEBUFFER, 0);
98
99              glGenFramebuffers(2, pingpongFBO);
100             glGenTextures(2, pingpongColorBuffers);
101
102             for (int i = 0; i < std::end(pingpongFBO) - ↙
                    ↳ std::begin(pingpongFBO); i++)
103             {
104                 glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[i]);
105                 glBindTexture(GL_TEXTURE_2D, pingpongColorBuffers[i]);
106                 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, width, ↙
                        ↳ height, 0, GL_RGBA, GL_FLOAT, NULL);
107                 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, ↙
                        ↳ GL_LINEAR);
108                 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, ↙
                        ↳ GL_LINEAR);
109                 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, ↙
                        ↳ GL_CLAMP_TO_EDGE);
```

```
110                 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, ↙
                        ↳ GL_CLAMP_TO_EDGE);
111                 glFramebufferTexture2D(GL_FRAMEBUFFER, ↙
                        ↳ GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, ↙
                        ↳ pingpongColorBuffers[i], 0);
112                 if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != ↙
                        ↳ GL_FRAMEBUFFER_COMPLETE)
113                 {
114                     std::cout << "ERROR::FRAMEBUFFER:␣Pingpong␣↙
                            ↳ framebuffer␣not␣completed!" << std::endl;;
115                 }
116             }
117
118             glBindFramebuffer(GL_FRAMEBUFFER, 0);
119         };
120
121         void RenderQuadFullScreen()
122         {
123             if (VAO == 0)
124             {
125                 float vertices[] =
126                 {
127                     -1.0f,  1.0f, 0.0f, 0.0f, 1.0f,
128                     -1.0f, -1.0f, 0.0f, 0.0f, 0.0f,
129                      1.0f,  1.0f, 0.0f, 1.0f, 1.0f,
130                      1.0f, -1.0f, 0.0f, 1.0f, 0.0f
131                 };
132
133                 GLuint VBO;
134
135                 glGenVertexArrays(1, &VAO);
136                 glBindVertexArray(VAO);
137
138                 glGenBuffers(1, &VBO);
139                 glBindBuffer(GL_ARRAY_BUFFER, VBO);
140
141                 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), ↙
                        ↳ &vertices, GL_STATIC_DRAW);
142
143                 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * ↙
                        ↳ sizeof(float), (void*)0);
144                 glEnableVertexAttribArray(0);
145
146                 glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * ↙
                        ↳ sizeof(float), (void*)(3 * sizeof(float)));
147                 glEnableVertexAttribArray(1);
148
```

```
149                 glBindVertexArray(0);
150             }
151             glBindVertexArray(VAO);
152             glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
153             glBindVertexArray(0);
154         };
155     private:
156 };
157
158 #endif
```

3.13. Menu

Scena 2D w której następuję renderowanie tekstu sprawdzanie kolizji AABB na zasadzie zdobycia informacji o pozyci myszy i pozycji obiektów przycisków zaczerpniętych z meshów poprzez odwołanie się do modela będącego pośrednikiem.

```
1  #include "Menu.h"
2
3  #include<ft2build.h>
4  #include FT_FREETYPE_H
5
6  Menu::Menu()
7  {
8      Button buttonExample(glm::vec3(0.0f, 0.2f, 0.0f));
9      Button buttonAdvanceExample(glm::vec3(0.0f, 0.0f, 0.0f));
10     Button buttonQuit(glm::vec3(0.0f, -0.2f, 0.0f));
11
12     buttons.push_back(buttonExample);
13     buttons.push_back(buttonAdvanceExample);
14     buttons.push_back(buttonQuit);
15
16     Shader cursorShader("Shaders/Cursor.vert", "Shaders/Cursor.frag");
17     Shader buttonShader("Shaders/Button.vert", "Shaders/Button.frag");
18     Shader textShader("Shaders/Text.vert", "Shaders/Text.frag");
19
20     Model cursorModel("Models/Cursor/cursor.obj");
21
22     FT_Library ft;
23     if (FT_Init_FreeType(&ft))
24     {
25         std::cout << "ERROR::FREETYPE:␣Could␣not␣init␣FreeType␣↙
               ↳ Library" << std::endl;
26         abort();
27     }
```

```
28
29      FT_Face face;
30      if (FT_New_Face(ft, "fonts/arial.ttf", 0, &face))
31      {
32          std::cout << "ERROR::FREETYPE:␣Failed␣to␣load␣font" << ↙
                ↳ std::endl;
33          abort();
34      }
35
36      FT_Set_Pixel_Sizes(face, 0, 48);
37
38      textShader.Activate();
39
40      glm::mat4 projection = glm::ortho(0.0f, static_cast<float>(width), ↙
            ↳ 0.0f, static_cast<float>(height));
41      textShader.SetMat4("projection", projection);
42
43      glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
44
45      for (unsigned char c = 0; c < 128; c++)
46      {
47          if (FT_Load_Char(face, c, FT_LOAD_RENDER))
48          {
49              std::cout << "ERROR:FREETYPE:␣Failed␣to␣load␣Glyph" << ↙
                    ↳ std::endl;
50              abort();
51          }
52
53          GLuint texture;
54          glGenTextures(1, &texture);
55          glBindTexture(GL_TEXTURE_2D, texture);
56
57          glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, ↙
                ↳ face->glyph->bitmap.width, face->glyph->bitmap.rows, 0, ↙
                ↳ GL_RED, GL_UNSIGNED_BYTE, face->glyph->bitmap.buffer);
58
59          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, ↙
                ↳ GL_LINEAR);
60          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, ↙
                ↳ GL_LINEAR);
61          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, ↙
                ↳ GL_CLAMP_TO_EDGE);
62          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, ↙
                ↳ GL_CLAMP_TO_EDGE);
63
64          Character ch =
65          {
```

```
66              texture,
67              glm::ivec2(face->glyph->bitmap.width, ↙
                    ↳ face->glyph->bitmap.rows),
68              glm::ivec2(face->glyph->bitmap_left, ↙
                    ↳ face->glyph->bitmap_top),
69              static_cast<GLuint>(face->glyph->advance.x)
70          };
71          characters.insert(std::pair<char, Character>(c, ch));
72      }
73      glBindTexture(GL_TEXTURE_2D, 0);
74
75      glGenVertexArrays(1, &VAO);
76      glBindVertexArray(VAO);
77
78      glGenBuffers(1, &VBO);
79      glBindBuffer(GL_ARRAY_BUFFER, VBO);
80
81      glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 6 * 4, NULL, ↙
            ↳ GL_DYNAMIC_DRAW);
82
83      glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(float), ↙
            ↳ 0);
84      glEnableVertexAttribArray(0);
85
86      glBindBuffer(GL_ARRAY_BUFFER, 0);
87      glBindVertexArray(0);
88      glPixelStorei(GL_UNPACK_ALIGNMENT, 0);
89
90      FT_Done_Face(face);
91      FT_Done_FreeType(ft);
92
93      cursorOffsetY = cursorModel.meshes[0].CalculateCursorOffsetY();
94
95      shaders.push_back(cursorShader);
96      shaders.push_back(buttonShader);
97      shaders.push_back(textShader);
98
99      models.push_back(cursorModel);
100 }
101
102 void Menu::Render(GLFWwindow* window, float deltaTime)
103 {
104     float normalizeMouseX = mouseX / (double)width;
105     float normalizeMouseY = mouseY / (double)height;
106
107     glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT);
108
```

```
109     shaders[0].Activate();
110
111     glm::mat4 model = glm::translate(glm::mat4(1.0f), ↙
            ↳ glm::vec3(normalizeMouseX, -(-0.01f + cursorOffsetY + ↙
            ↳ normalizeMouseY), -0.2f));
112
113     shaders[0].SetMat4("model", model);
114
115     models[0].Draw(shaders[0], false);
116
117     shaders[1].Activate();
118
119     for (int i = 0; i < std::end(buttons) - std::begin(buttons); i++)
120     {
121         buttons[i].UpdateState(glm::vec2(normalizeMouseX, ↙
                ↳ normalizeMouseY));
122         buttons[i].Render(shaders[1]);
123     }
124
125     glEnable(GL_CULL_FACE);
126     glEnable(GL_BLEND);
127     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
128
129     RenderText(shaders[2], "Game␣Engine", glm::vec2(width / 3 + 50, ↙
            ↳ height / 3 + 500.0f), 2.0f, glm::vec3(1.0f, 1.0f, 1.0f));
130     RenderText(shaders[2], "Example", glm::vec2(width / 3 + 250.0f, ↙
            ↳ height / 3 + 275.0f), 0.5f, glm::vec3(1.0f, 1.0f, 1.0f));
131     RenderText(shaders[2], "AdvanceExample", glm::vec2(width / 3 + ↙
            ↳ 250.0f, height / 3 + 175.0f), 0.5f, glm::vec3(1.0f, 1.0f, ↙
            ↳ 1.0f));
132     RenderText(shaders[2], "Quit", glm::vec2(width / 3 + 250.0f, ↙
            ↳ height / 3 + 75.0f), 0.5f, glm::vec3(1.0f, 1.0f, 1.0f));
133
134     glDisable(GL_BLEND);
135     glDisable(GL_CULL_FACE);
136 }
137
138 void Menu::RenderText(Shader& shader, std::string value, glm::vec2 ↙
        ↳ position, float scale, glm::vec3 color)
139 {
140     shader.Activate();
141
142     shader.SetVec3("textColor", color);
143
144     glActiveTexture(GL_TEXTURE0);
145     glBindVertexArray(VAO);
146
```

```
147     float positionX = 0, positionY = 0;
148     float PositionX = position.x, PositionY = position.y;
149 
150     std::string::const_iterator c;
151     for (c = value.begin(); c != value.end(); c++)
152     {
153         Character ch = characters[*c];
154 
155         positionX = position.x + ch.offsetFromTopLeft.x * scale;
156         positionY = position.y - (ch.size.y - ch.offsetFromTopLeft.y) ↙
                ↳ * scale;
157 
158         float width = ch.size.x * scale;
159         float height = ch.size.y * scale;
160 
161         float vertices[6][4] =
162         {
163             {positionX, positionY + height, 0.0f, 0.0f},
164             {positionX, positionY, 0.0f, 1.0f},
165             {positionX + width, positionY, 1.0f, 1.0f},
166             {positionX, positionY + height, 0.0f, 0.0f},
167             {positionX + width, positionY, 1.0f, 1.0f},
168             {positionX + width, positionY + height, 1.0f, 0.0f}
169         };
170 
171         glBindTexture(GL_TEXTURE_2D, ch.textureID);
172         glBindBuffer(GL_ARRAY_BUFFER, VBO);
173         glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
174         glBindBuffer(GL_ARRAY_BUFFER, 0);
175         glDrawArrays(GL_TRIANGLES, 0, 6);
176         position.x += (ch.horizontalOffset >> 6) * scale;
177     }
178 
179     glBindVertexArray(0);
180     glBindTexture(GL_TEXTURE_2D, 0);
181 }
```

3.14. Example

Scena z Boxami dostępna pod przyciskiem Example z Menu w niej odbywa się przygotowanie sceny w konstruktorze a następnie wyrenderowanie z postprocessing w funkcji render.

```
1 #include "Example.h"
2 
```

```
3  Example::Example()
4      : boxesModel{ {glm::mat4(1.0f), 0.0f, glm::vec3(0.0f, 0.0f, 0.0f), ↙
           ↳ glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(1.0f, 1.0f, 1.0f)},
5                    {glm::mat4(1.0f), 0.0f, glm::vec3(3.0f, 0.0f, 0.0f), ↙
                        ↳ glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(0.5f, ↙
                        ↳ 0.5f, 0.5f)},
6                    {glm::mat4(1.0f), 30.0f, glm::vec3(-3.0f, 0.0f, ↙
                        ↳ 0.0f), glm::vec3(0.0f, 0.0f, 1.0f), ↙
                        ↳ glm::vec3(0.8f, 0.8f, 0.8f)},
7                    {glm::mat4(1.0f), 0.0f, glm::vec3(0.0f, 0.0f, 3.0f), ↙
                        ↳ glm::vec3(0.0f, 0.0f, 1.0f), glm::vec3(1.0f, ↙
                        ↳ 1.0f, 1.0f)}}
8  {
9      Shader shader("Shaders/Mesh.vert", "Shaders/Mesh.frag");
10     Shader light("Shaders/Light.vert", "Shaders/Light.frag");
11     Shader postProcessing("Shaders/PostProcessing.vert", ↙
           ↳ "Shaders/PostProcessing.frag");
12     Shader blur("Shaders/Blur.vert", "Shaders/Blur.frag");
13
14     Camera camera(glm::vec3(0.0f, 0.0f, 0.0f));
15
16     Model box("Models/Box/box.obj");
17
18     PointLight pointLight;
19
20     pointLight.position = glm::vec3(0.0f, 5.0f, -3.0f);
21     pointLight.ambient = glm::vec3(0.1f, 0.1f, 0.1f);
22     pointLight.diffuse = glm::vec3(10.0f, 10.0f, 10.0f);
23     pointLight.specular = glm::vec3(1.0f, 1.0f, 1.0f);
24     pointLight.linear = 0.09f;
25     pointLight.quadratic = 0.032f;
26
27     projection = glm::perspective(glm::radians(45.0f), (float)width / ↙
           ↳ (float)height, 0.1f, 100.0f);
28
29     SetupPostProcessing();
30
31     blur.Activate();
32
33     blur.SetInt("image", 0);
34
35     postProcessing.Activate();
36
37     postProcessing.SetInt("hdrBuffer", 0);
38     postProcessing.SetInt("bloomBlur", 1);
39
40     light.Activate();
```

```
41
42      light.SetVec3("lightColor", pointLight.diffuse);
43
44      shaders.push_back(shader);
45      shaders.push_back(light);
46      shaders.push_back(postProcessing);
47      shaders.push_back(blur);
48
49      cameras.push_back(camera);
50
51      models.push_back(box);
52
53      pointLights.push_back(pointLight);
54
55      for (int i = 0; i < std::end(boxesModel) - std::begin(boxesModel); ↙
            ↳ i++)
56      {
57          boxesModel[i].model = glm::translate(boxesModel[i].model, ↙
                ↳ boxesModel[i].position);
58          boxesModel[i].model = glm::rotate(boxesModel[i].model, ↙
                ↳ glm::radians(boxesModel[i].strengthOfRotation), ↙
                ↳ boxesModel[i].rotation);
59          boxesModel[i].model = glm::scale(boxesModel[i].model, ↙
                ↳ boxesModel[i].scale);
60      }
61  }
62
63  void Example::Render(GLFWwindow* window, float deltaTime)
64  {
65      glBindFramebuffer(GL_FRAMEBUFFER, HDR);
66
67          glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
68
69          cameras[0].HandleEvents(window, deltaTime);
70          cameras[0].Update();
71
72          shaders[0].Activate();
73
74          shaders[0].SetFloat("material.shininess", 64.0f);
75
76          shaders[0].SetVec3("directionalLight.direction", 0.0f, 0.0f, ↙
                ↳ 0.0f);
77          shaders[0].SetVec3("directionalLight.ambient", 0.0f, 0.0f, ↙
                ↳ 0.0f);
78          shaders[0].SetVec3("directionalLight.diffuse", 0.0f, 0.0f, ↙
                ↳ 0.0f);
```

```
79          shaders[0].SetVec3("directionalLight.specular", 0.0f, 0.0f, ↙
                ↳ 0.0f);
80
81          shaders[0].SetVec3("pointLight[0].position", ↙
                ↳ pointLights[0].position);
82          shaders[0].SetVec3("pointLight[0].ambient", ↙
                ↳ pointLights[0].ambient);
83          shaders[0].SetVec3("pointLight[0].diffuse", ↙
                ↳ pointLights[0].diffuse);
84          shaders[0].SetVec3("pointLight[0].specular", ↙
                ↳ pointLights[0].specular);
85          shaders[0].SetFloat("pointLight[0].constant", gammaCorrection ↙
                ↳ ? 2 * pointLights[0].constant : pointLights[0].constant);
86          shaders[0].SetFloat("pointLight[0].linear", gammaCorrection ? ↙
                ↳ 2 * pointLights[0].linear : pointLights[0].linear);
87          shaders[0].SetFloat("pointLight[0].quadratic", gammaCorrection ↙
                ↳ ? 2 * pointLights[0].quadratic : ↙
                ↳ pointLights[0].quadratic);
88
89          shaders[0].SetInt("spotLight.on", flashlight);
90          shaders[0].SetVec3("spotLight.position", cameras[0].position);
91          shaders[0].SetVec3("spotLight.direction", cameras[0].front);
92          shaders[0].SetFloat("spotLight.cutOff", ↙
                ↳ glm::cos(glm::radians(12.5f)));
93          shaders[0].SetFloat("spotLight.outerCutOff", ↙
                ↳ glm::cos(glm::radians(15.0f)));
94          shaders[0].SetVec3("spotLight.ambient", ↙
                ↳ flashlightObject.ambient);
95          shaders[0].SetVec3("spotLight.diffuse", ↙
                ↳ flashlightObject.diffuse);
96          shaders[0].SetVec3("spotLight.specular", ↙
                ↳ flashlightObject.specular);
97          shaders[0].SetFloat("spotLight.constant", gammaCorrection ? 2 ↙
                ↳ * flashlightObject.constant : flashlightObject.constant);
98          shaders[0].SetFloat("spotLight.linear", gammaCorrection ? 2 * ↙
                ↳ flashlightObject.linear : flashlightObject.linear);
99          shaders[0].SetFloat("spotLight.quadratic", gammaCorrection ? 2 ↙
                ↳ * flashlightObject.quadratic : ↙
                ↳ flashlightObject.quadratic);
100
101         shaders[0].SetVec3("viewPosition", cameras[0].position);
102
103         for (int i = 0; i < std::end(boxesModel) - ↙
                ↳ std::begin(boxesModel); i++)
104         {
105             boxesModel[i].model = glm::rotate(boxesModel[i].model, ↙
                    ↳ glm::radians(0.05f), glm::vec3(0.0f, 1.0f, 0.0f));
```

```
106
107             shaders[0].SetMat4("model", boxesModel[i].model);
108             shaders[0].SetMat4("view", cameras[0].view);
109             shaders[0].SetMat4("projection", projection);
110
111             models[0].Draw(shaders[0], gammaCorrection);
112         }
113
114         shaders[1].Activate();
115         glm::mat4 model = glm::mat4(1.0f);
116
117         model = glm::translate(model, pointLights[0].position);
118
119         glm::mat4 mvp = projection * cameras[0].view * model;
120
121         shaders[1].SetMat4("mvp", mvp);
122
123         models[0].Draw(shaders[1], gammaCorrection);
124
125     glBindFramebuffer(GL_FRAMEBUFFER, 0);
126
127     bool horizontal = true, firstIteration= true;
128     unsigned int amount = 10;
129
130     shaders[3].Activate();
131
132     glActiveTexture(GL_TEXTURE0);
133     for (GLuint i = 0; i < amount; i++)
134     {
135         glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[horizontal]);
136         shaders[3].SetInt("horizontal", horizontal);
137         glBindTexture(GL_TEXTURE_2D, firstIteration ? colorBuffers[1] ↙
                ↳ : pingpongColorBuffers[!horizontal]);
138         RenderQuadFullScreen();
139         horizontal = !horizontal;
140         if (firstIteration) firstIteration = false;
141     }
142
143     glBindFramebuffer(GL_FRAMEBUFFER, 0);
144
145     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
146
147     shaders[2].Activate();
148     glActiveTexture(GL_TEXTURE0);
149     glBindTexture(GL_TEXTURE_2D, colorBuffers[0]);
150     glActiveTexture(GL_TEXTURE1);
151     glBindTexture(GL_TEXTURE_2D, pingpongColorBuffers[!horizontal]);
```

```
152     shaders[2].SetInt("hdr", hdr);
153     shaders[2].SetInt("gammaCorrection", gammaCorrection);
154     shaders[2].SetInt("bloom", bloom);
155     RenderQuadFullScreen();
156 }
```

3.15. AdvanceExample

Scena ze pszczoła po krótce najważniejsze elementy to inicjalizacja 50 pozycji, skal oraz rotacji dla kwiatów przesłanie informacji do animatora w celu animacji szkieletowej pszczoły. Wykorzystanie prostej interpolacji z klasy InGameAnimation w celu ruchu pszczoły do kwiatka. Uwaga dodatkowo została dodana randomizacja rozmieszczenia kwiatów dzięki czemu pszczoła zna zawsze transę na środek kwiatka ale przy każdym uruchomieniu mamy innych układ z 50 możliwych opcji do którcyh poleci kwiatek. Mamy również dodany prosty cykl dnia i nocy na zasadzie tików bez uwzględnienia delta time oraz postprocessing. Scena ma pokazywać możliwości openGL które udało nam się zbadać przez okres nauki.

```
1  #include "AdvanceExample.h"
2
3  AdvanceExample::AdvanceExample()
4      :flowersModel{{glm::mat4(1.0f), 0.0f, glm::vec3(0.0f, 0.0f, ↙
           ↳ -5.0f), glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(1.0f, 1.0f, ↙
           ↳ 1.0f)},
5                     {glm::mat4(1.0f),  0.0f, glm::vec3(0.0f, 0.0f, ↙
                          ↳ 5.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                          ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
6                     {glm::mat4(1.0f),  0.0f, glm::vec3(5.0f, 0.0f, ↙
                          ↳ 0.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                          ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
7                     {glm::mat4(1.0f),  0.0f, glm::vec3(-5.0f, 0.0f, ↙
                          ↳ 0.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                          ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
8                     {glm::mat4(1.0f), 0.0f, glm::vec3(3.0f, 0.0f, 2.0f), ↙
                          ↳ glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(1.0f, ↙
                          ↳ 1.0f, 1.0f)},
9                     {glm::mat4(1.0f),  0.0f, glm::vec3(4.0f, 0.0f, ↙
                          ↳ 6.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                          ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
10                    {glm::mat4(1.0f), 0.0f, glm::vec3(17.0f, 0.0f, ↙
                          ↳ 4.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                          ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
```

```
11                  {glm::mat4(1.0f), 0.0f, glm::vec3(12.0f, 0.0f, ↙
                        ↳ 8.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
12                  {glm::mat4(1.0f), 0.0f, glm::vec3(9.0f, 0.0f, ↙
                        ↳ 14.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
13                  {glm::mat4(1.0f), 0.0f, glm::vec3(-8.0f, 0.0f, ↙
                        ↳ -5.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
14                  {glm::mat4(1.0f),  0.0f, glm::vec3(-10.0f, 0.0f, ↙
                        ↳ -7.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
15                  {glm::mat4(1.0f),  0.0f, glm::vec3(-11.0f, 0.0f, ↙
                        ↳ -13.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
16                  {glm::mat4(1.0f),  0.0f, glm::vec3(-5.0f, 0.0f, ↙
                        ↳ 13.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
17                  {glm::mat4(1.0f),  0.0f, glm::vec3(0.0f, 0.0f, ↙
                        ↳ 18.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
18                  {glm::mat4(1.0f), 0.0f, glm::vec3(2.0f, 0.0f, ↙
                        ↳ -19.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
19                  {glm::mat4(1.0f), 0.0f, glm::vec3(6.0f, 0.0f, ↙
                        ↳ -21.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
20                  {glm::mat4(1.0f), 0.0f, glm::vec3(-9.0f, 0.0f, ↙
                        ↳ 15.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
21                  {glm::mat4(1.0f), 0.0f, glm::vec3(18.0f, 0.0f, ↙
                        ↳ 14.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
22                  {glm::mat4(1.0f), 0.0f, glm::vec3(-10.0f, 0.0f, ↙
                        ↳ 11.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
23                  {glm::mat4(1.0f), 0.0f, glm::vec3(-17.0f, 0.0f, ↙
                        ↳ 13.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
24                  {glm::mat4(1.0f),  0.0f, glm::vec3(-17.0f, 0.0f, ↙
                        ↳ 4.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
25                  {glm::mat4(1.0f),  0.0f, glm::vec3(-13.0f, 0.0f, ↙
                        ↳ 20.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
```

```
26                  {glm::mat4(1.0f),  0.0f, glm::vec3(14.0f, 0.0f, ↙
                        ↳ -7.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
27                  {glm::mat4(1.0f),  0.0f, glm::vec3(16.0f, 0.0f, ↙
                        ↳ -11.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
28                  {glm::mat4(1.0f), 0.0f, glm::vec3(12.0f, 0.0f, ↙
                        ↳ -19.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
29                  {glm::mat4(1.0f), 0.0f, glm::vec3(-18.0f, 0.0f, ↙
                        ↳ 1.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
30                  {glm::mat4(1.0f), 0.0f, glm::vec3(-10.0f, 0.0f, ↙
                        ↳ 3.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
31                  {glm::mat4(1.0f), 0.0f, glm::vec3(-2.7f, 0.0f, ↙
                        ↳ 4.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
32                  {glm::mat4(1.0f), 0.0f, glm::vec3(-10.0f, 0.0f, ↙
                        ↳ 6.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
33                  {glm::mat4(1.0f), 0.0f, glm::vec3(-6.0f, 0.0f, ↙
                        ↳ 7.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
34                  {glm::mat4(1.0f),  0.0f, glm::vec3(12.0f, 0.0f, ↙
                        ↳ -10.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
35                  {glm::mat4(1.0f),  0.0f, glm::vec3(14.0f, 0.0f, ↙
                        ↳ -13.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
36                  {glm::mat4(1.0f),  0.0f, glm::vec3(7.0f, 0.0f, ↙
                        ↳ -5.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
37                  {glm::mat4(1.0f),  0.0f, glm::vec3(12.0f, 0.0f, ↙
                        ↳ -4.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
38                  {glm::mat4(1.0f), 0.0f, glm::vec3(19.0f, 0.0f, ↙
                        ↳ -7.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
39                  {glm::mat4(1.0f), 0.0f, glm::vec3(-5.0f, 0.0f, ↙
                        ↳ -13.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
40                  {glm::mat4(1.0f), 0.0f, glm::vec3(7.0f, 0.0f, ↙
                        ↳ -18.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                        ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
```

```
41                      {glm::mat4(1.0f), 0.0f, glm::vec3(15.0f, 0.0f, ↙
                            ↳ 0.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
42                      {glm::mat4(1.0f), 0.0f, glm::vec3(-8.0f, 0.0f, ↙
                            ↳ -2.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
43                      {glm::mat4(1.0f), 0.0f, glm::vec3(-18.0f, 0.0f, ↙
                            ↳ -4.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
44                      {glm::mat4(1.0f),  0.0f, glm::vec3(1.5f, 0.0f, ↙
                            ↳ 7.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
45                      {glm::mat4(1.0f),  0.0f, glm::vec3(-8.0f, 0.0f, ↙
                            ↳ -21.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
46                      {glm::mat4(1.0f),  0.0f, glm::vec3(5.0f, 0.0f, ↙
                            ↳ -9.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
47                      {glm::mat4(1.0f),  0.0f, glm::vec3(5.5f, 0.0f, ↙
                            ↳ -13.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
48                      {glm::mat4(1.0f), 0.0f, glm::vec3(0.0f, 0.0f, ↙
                            ↳ -13.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
49                      {glm::mat4(1.0f), 0.0f, glm::vec3(-3.3f, 0.0f, ↙
                            ↳ -22.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
50                      {glm::mat4(1.0f), 0.0f, glm::vec3(6.0f, 0.0f, ↙
                            ↳ 18.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
51                      {glm::mat4(1.0f), 0.0f, glm::vec3(23.0f, 0.0f, ↙
                            ↳ 3.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
52                      {glm::mat4(1.0f), 0.0f, glm::vec3(-22.0f, 0.0f, ↙
                            ↳ 1.2f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)},
53                      {glm::mat4(1.0f), 0.0f, glm::vec3(-2.4f, 0.0f, ↙
                            ↳ -19.0f), glm::vec3(0.0f, 1.0f, 0.0f), ↙
                            ↳ glm::vec3(1.0f, 1.0f, 1.0f)}},
54      beesModel{ {glm::mat4(1.0f), glm::radians(0.0f), glm::vec3(-2.9f, ↙
            ↳ 4.0f, -9.7f), glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(0.09f, ↙
            ↳ 0.09f, 0.09f)}},
55      pathToFlowersAnimationDistribution(0, ↙
            ↳ std::end(selectedFlowersModel) - ↙
            ↳ std::begin(selectedFlowersModel) - 1),
56      generator(std::random_device{}()),
```

```
57      hives{{glm::mat4(1.0f), glm::radians(-90.0f), glm::vec3(-2.5f, ↙
            ↳ 4.15f, -10.0f), glm::vec3(1.0f, 0.0f, 0.0f), glm::vec3(1.0f, ↙
            ↳ 1.0f, 1.0f)}},
58      diffuse(2.0f, 2.0f, 2.0f)
59  {
60      Shader animatedMesh("Shaders/AnimatedMesh.vert", ↙
            ↳ "Shaders/AnimatedMesh.frag");
61      Shader mesh("Shaders/Mesh.vert", "Shaders/Mesh.frag");
62      Shader light("Shaders/Light.vert", "Shaders/Light.frag");
63      Shader postProcessing("Shaders/PostProcessing.vert", ↙
            ↳ "Shaders/PostProcessing.frag");
64      Shader blur("Shaders/Blur.vert", "Shaders/Blur.frag");
65
66      Camera camera(glm::vec3(0.0f, 0.0f, 0.0f));
67
68      Model bee("Models/Bee/bee.fbx");
69      Model tree("Models/Tree/tree.fbx");
70      Model ground("Models/Ground/ground.fbx");
71      Model hive("Models/Hive/hive.fbx");
72      Model flower("Models/Flower/flower.fbx");
73      Model sunAndMoon("Models/SunAndMoon/sunAndMoon.fbx");
74
75      Animation animation("Models/Bee/bee.fbx", &bee);
76
77      Animator animator(animation);
78
79      PointLight pointLight
80      {
81          glm::vec3 (0.0f, -30.0f, 0.0f),
82          glm::vec3 (0.0f, 0.0f, 0.0f),
83          glm::vec3 (0.0f, 0.0f, 0.0f),
84          glm::vec3 (0.0f, 0.0f, 0.0f),
85          1.0f,
86          0.09f,
87          0.032f
88      };
89
90      projection = glm::perspective(glm::radians(45.0f), (float)width / ↙
            ↳ (float)height, 0.1f, 100.0f);
91
92      SetupPostProcessing();
93
94      blur.Activate();
95
96      blur.SetInt("image", 0);
97
98      postProcessing.Activate();
```

```
99
100     postProcessing.SetInt("hdrBuffer", 0);
101     postProcessing.SetInt("bloomBlur", 1);
102
103     light.Activate();
104
105     light.SetVec3("lightColor", pointLight.diffuse);
106
107     animators.push_back(animator);
108
109     shaders.push_back(animatedMesh);
110     shaders.push_back(mesh);
111     shaders.push_back(light);
112     shaders.push_back(postProcessing);
113     shaders.push_back(blur);
114
115     cameras.push_back(camera);
116
117     models.push_back(bee);
118     models.push_back(tree);
119     models.push_back(ground);
120     models.push_back(hive);
121     models.push_back(flower);
122     models.push_back(sunAndMoon);
123
124     pointLights.push_back(pointLight);
125
126     std::uniform_int_distribution<std::size_t> distribution(0, ↙
        ↳ std::end(flowersModel) - std::begin(flowersModel) - 1);
127
128     for (std::size_t i = 0; i < std::end(flowersModel) - ↙
        ↳ std::begin(flowersModel); i++)
129     {
130         std::size_t value = distribution(generator);
131         if (std::find(std::begin(selectedFlowersModel), ↙
            ↳ std::end(selectedFlowersModel), value) != ↙
            ↳ std::end(selectedFlowersModel))
132         {
133             --i;
134             continue;
135         }
136         selectedFlowersModel[i] = value;
137         flowersModel[value].model = ↙
            ↳ glm::translate(flowersModel[value].model, ↙
            ↳ flowersModel[value].position);
138         flowersModel[value].model = ↙
            ↳ glm::rotate(flowersModel[value].model, ↙
```

```
            ↳ flowersModel[value].strengthOfRotation, ↙
            ↳ flowersModel[value].rotation);
139         flowersModel[value].model = ↙
            ↳ glm::scale(flowersModel[value].model, ↙
            ↳ flowersModel[value].scale);
140     }
141
142     animationIndex = pathToFlowersAnimationDistribution(generator);
143
144     std::vector<KeyPositionInGame> positions;
145     std::vector<KeyRotationInGame> rotations;
146     std::vector<KeyScaleInGame> scales;
147
148     for (int i = 0; i < std::end(pathToFlowers) - ↙
        ↳ std::begin(pathToFlowers); i++)
149     {
150         positions = {};
151         rotations = {};
152         scales = {};
153
154     positions.push_back({ glm::vec3(beesModel[0].position), 0.0f});
155     positions.push_back({ glm::vec3(flowersModel[i].position.x, 4.05f, ↙
        ↳ flowersModel[i].position.z), 20.0f});
156     positions.push_back({ glm::vec3(flowersModel[i].position.x, 4.05f, ↙
        ↳ flowersModel[i].position.z), 40.0f });
157     positions.push_back({ glm::vec3(beesModel[0].position), 60.0f });
158     positions.push_back({ glm::vec3(beesModel[0].position), 80.0f });
159     rotations.push_back({glm::angleAxis(glm::radians(beesModel[0].strengthOfRotation)
        ↳ glm::vec3(0.0f, 1.0f, 0.0f)), 0.0f});
160     rotations.push_back({ ↙
        ↳ glm::angleAxis(glm::radians(beesModel[0].strengthOfRotation), ↙
        ↳ glm::vec3(0.0f, 1.0f, 0.0f)), 20.0f });
161     rotations.push_back({ glm::angleAxis(glm::radians(180.0f), ↙
        ↳ glm::vec3(0.0f, 1.0f, 0.0f)), 40.0f });
162     rotations.push_back({ glm::angleAxis(glm::radians(180.0f), ↙
        ↳ glm::vec3(0.0f, 1.0f, 0.0f)), 60.0f });
163     rotations.push_back({ ↙
        ↳ glm::angleAxis(glm::radians(beesModel[0].strengthOfRotation), ↙
        ↳ glm::vec3(0.0f, 1.0f, 0.0f)), 80.0f });
164     scales.push_back({ beesModel[0].scale, 0.0f });
165
166     pathToFlowers[i] = new InGameAnimation(80.0f, 5, positions, ↙
        ↳ rotations, scales);
167     }
168
169     positions = { {glm::vec3(0.0f, 5.0f, 0.0f), 0.0f}};
```

```
170     rotations = { {glm::angleAxis(glm::radians(0.0f), glm::vec3(1.0f, ↙
            ↳ 0.0f, 0.0f)), 0.0f},
171                   {glm::angleAxis(glm::radians(180.0f), ↙
                          ↳ glm::vec3(1.0f, 0.0f, 0.0f)), 200.0f},
172                   {glm::angleAxis(glm::radians(360.0f), ↙
                          ↳ glm::vec3(1.0f, 0.0f, 0.0f)), 400.0f}};
173     scales = { {glm::vec3(1.0f, 1.0f, 1.0f)}};
174
175     SunAndMoon = new InGameAnimation(400.0f, 30, positions, rotations, ↙
            ↳ scales);
176
177     for (int i = 0; i < std::end(beesModel) - std::begin(beesModel); ↙
            ↳ i++)
178     {
179         beesModel[i].model = glm::translate(beesModel[i].model, ↙
                ↳ beesModel[i].position);
180         beesModel[i].model = glm::rotate(beesModel[i].model, ↙
                ↳ beesModel[i].strengthOfRotation, beesModel[i].rotation);
181         beesModel[i].model = glm::scale(beesModel[i].model, ↙
                ↳ beesModel[i].scale);
182     }
183
184     for (int i = 0; i < std::end(hives) - std::begin(hives); i++)
185     {
186         hives[i].model = glm::translate(hives[i].model, ↙
                ↳ hives[i].position);
187         hives[i].model = glm::rotate(hives[i].model, ↙
                ↳ glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
188         hives[i].model = glm::rotate(hives[i].model, ↙
                ↳ hives[i].strengthOfRotation, hives[i].rotation);
189         hives[i].model = glm::scale(hives[i].model, hives[i].scale);
190     }
191 }
192
193 void AdvanceExample::Render(GLFWwindow* window, float deltaTime)
194 {
195     counter++;
196     glBindFramebuffer(GL_FRAMEBUFFER, HDR);
197
198         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
199
200         cameras[0].HandleEvents(window, deltaTime);
201         cameras[0].Update();
202
203         shaders[0].Activate();
204
205         animators[0].UpdateAnimation(deltaTime);
```

```
206
207         for (int i = 0; i < animators[0].finalBoneMatrices.size(); ++i)
208         {
209             shaders[0].SetMat4("finalBonesMatrices[" + ↙
                    ↳ std::to_string(i) + "]", ↙
                    ↳ animators[0].finalBoneMatrices[i]);
210         }
211
212         if ↙
                ↳ (pathToFlowers[selectedFlowersModel[animationIndex]]->endAnimation)
213         {
214             pathToFlowers[animationIndex]->endAnimation = false;
215             pathToFlowers[animationIndex]->currentTime = 0.0f;
216             animationIndex = ↙
                    ↳ pathToFlowersAnimationDistribution(generator);
217         }
218
219         glm::mat4 model = ↙
                ↳ pathToFlowers[selectedFlowersModel[animationIndex]]->UpdateAnimation(de
220
221         glm::mat4 mvp = projection * cameras[0].view * model;
222
223         shaders[0].SetMat4("mvp", mvp);
224
225         models[0].Draw(shaders[0], gammaCorrection);
226
227         shaders[1].Activate();
228
229         shaders[1].SetFloat("material.shininess", 64.0f);
230
231         if (counter < 800)
232         {
233             diffuse.x -= 0.01f;
234             diffuse.y -= 0.01f;
235             diffuse.z -= 0.01f;
236         }
237         if (counter > 800 && counter < 1600)
238         {
239             diffuse.x += 0.01f;
240             diffuse.y += 0.01f;
241             diffuse.z += 0.01f;
242         }
243         if (counter > 1600)
244         {
245             counter = 0;
246         }
247
```

```
248         shaders[1].SetVec3("directionalLight.direction", -0.2f, -1.0f, ↙
                ↳ -0.3f);
249         shaders[1].SetVec3("directionalLight.ambient", 0.1f, 0.1f, ↙
                ↳ 0.1f);
250         shaders[1].SetVec3("directionalLight.diffuse", diffuse);
251         shaders[1].SetVec3("directionalLight.specular", 1.0f, 1.0f, ↙
                ↳ 1.0f);
252
253         shaders[1].SetVec3("pointLight[0].position", ↙
                ↳ pointLights[0].position);
254         shaders[1].SetVec3("pointLight[0].ambient", ↙
                ↳ pointLights[0].ambient);
255         shaders[1].SetVec3("pointLight[0].diffuse", ↙
                ↳ pointLights[0].diffuse);
256         shaders[1].SetVec3("pointLight[0].specular", ↙
                ↳ pointLights[0].specular);
257         shaders[1].SetFloat("pointLight[0].constant", gammaCorrection ↙
                ↳ ? 2 * pointLights[0].constant : pointLights[0].constant);
258         shaders[1].SetFloat("pointLight[0].linear", gammaCorrection ? ↙
                ↳ 2 * pointLights[0].linear : pointLights[0].linear);
259         shaders[1].SetFloat("pointLight[0].quadratic", gammaCorrection ↙
                ↳ ? 2 * pointLights[0].quadratic : ↙
                ↳ pointLights[0].quadratic);
260
261         shaders[1].SetInt("spotLight.on", flashlight);
262         shaders[1].SetVec3("spotLight.position", cameras[0].position);
263         shaders[1].SetVec3("spotLight.direction", cameras[0].front);
264         shaders[1].SetFloat("spotLight.cutOff", ↙
                ↳ glm::cos(glm::radians(12.5f)));
265         shaders[1].SetFloat("spotLight.outerCutOff", ↙
                ↳ glm::cos(glm::radians(15.0f)));
266         shaders[1].SetVec3("spotLight.ambient", ↙
                ↳ flashlightObject.ambient);
267         shaders[1].SetVec3("spotLight.diffuse", ↙
                ↳ flashlightObject.diffuse);
268         shaders[1].SetVec3("spotLight.specular", ↙
                ↳ flashlightObject.specular);
269         shaders[1].SetFloat("spotLight.constant", gammaCorrection ? 2 ↙
                ↳ * flashlightObject.constant : flashlightObject.constant);
270         shaders[1].SetFloat("spotLight.linear", gammaCorrection ? 2 * ↙
                ↳ flashlightObject.linear : flashlightObject.linear);
271         shaders[1].SetFloat("spotLight.quadratic", gammaCorrection ? 2 ↙
                ↳ * flashlightObject.quadratic : ↙
                ↳ flashlightObject.quadratic);
272
273         shaders[1].SetVec3("viewPosition", cameras[0].position);
274
```

```
275         model = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, ↙
                ↳ -10.0f));
276
277         shaders[1].SetMat4("model", model);
278         shaders[1].SetMat4("view", cameras[0].view);
279         shaders[1].SetMat4("projection", projection);
280
281         models[1].Draw(shaders[1], gammaCorrection);
282
283         shaders[1].SetFloat("material.shininess", 12.0f);
284
285         model = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, ↙
                ↳ -0.001f, 0.0f));
286         model = glm::scale(model, glm::vec3(2.5f, 2.5f, 2.5f));
287
288         shaders[1].SetMat4("model", model);
289
290         models[2].Draw(shaders[1], gammaCorrection);
291
292         shaders[1].SetFloat("material.shininess", 8.0f);
293
294         shaders[1].SetMat4("model", hives[0].model);
295
296         models[3].Draw(shaders[1], gammaCorrection);
297
298         shaders[1].SetFloat("material.shininess", 60.0f);
299
300         for (int i = 0; i < std::end(selectedFlowersModel) - ↙
                ↳ std::begin(selectedFlowersModel); i++)
301         {
302             model = flowersModel[selectedFlowersModel[i]].model;
303
304             shaders[1].SetMat4("model", model);
305
306             models[4].Draw(shaders[1], gammaCorrection);
307         }
308
309         model = SunAndMoon->UpdateAnimation(deltaTime);
310
311         shaders[1].SetMat4("model", model);
312
313         models[5].Draw(shaders[1], gammaCorrection);
314
315         shaders[2].Activate();
316
317         model = glm::mat4(1.0f);
318
```

```
319         model = glm::translate(model, pointLights[0].position);
320 
321         mvp = projection * cameras[0].view * model;
322 
323         models[4].Draw(shaders[2], gammaCorrection);
324 
325     glBindFramebuffer(GL_FRAMEBUFFER, 0);
326 
327     bool horizontal = true, firstIteration = true;
328     unsigned int amount = 10;
329 
330     shaders[4].Activate();
331 
332     glActiveTexture(GL_TEXTURE0);
333     for (GLuint i = 0; i < amount; i++)
334     {
335         glBindFramebuffer(GL_FRAMEBUFFER, pingpongFBO[horizontal]);
336         shaders[4].SetInt("horizontal", horizontal);
337         glBindTexture(GL_TEXTURE_2D, firstIteration ? colorBuffers[1] ↙
                ↳ : pingpongColorBuffers[!horizontal]);
338         RenderQuadFullScreen();
339         horizontal = !horizontal;
340         if (firstIteration) firstIteration = false;
341     }
342 
343     glBindFramebuffer(GL_FRAMEBUFFER, 0);
344 
345     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
346 
347     shaders[3].Activate();
348     glActiveTexture(GL_TEXTURE0);
349     glBindTexture(GL_TEXTURE_2D, colorBuffers[0]);
350     glActiveTexture(GL_TEXTURE1);
351     glBindTexture(GL_TEXTURE_2D, pingpongColorBuffers[!horizontal]);
352     shaders[3].SetInt("hdr", hdr);
353     shaders[3].SetInt("gammaCorrection", gammaCorrection);
354     shaders[3].SetInt("bloom", bloom);
355     RenderQuadFullScreen();
356 }
357 
358 glm::vec3 AdvanceExample::InterpolateColor(float deltaTime, unsigned ↙
    ↳ int ticksPerSecond, float duration)
359 {
360     currentColorAnimationTime += ticksPerSecond * deltaTime;
361     currentColorAnimationTime = fmod(currentColorAnimationTime, ↙
        ↳ duration);
362 
```

```
363         return glm::vec3();
364     }
```

4. Opis modułów cieniujących

4.1. AnimatedMesh

```
#version 460 core

layout (location = 0) in vec3 position;
layout (location = 2) in vec3 normal;
layout (location = 3) in vec2 TextureCordinates;
layout (location = 4) in ivec4 boneIds;
layout (location = 5) in vec4 weights;

const int MAX_BONES = 100;
const int MAX_BONE_INFLUENCE = 4;

uniform mat4 mvp;
uniform mat4 finalBonesMatrices[MAX_BONES];

out vec2 textureCordinates;

void main()
{
    vec4 totalPosition = vec4(0.0f);
    for (int i = 0; i < MAX_BONE_INFLUENCE; i++)
    {
        if (boneIds[i] == -1)
            continue;
        if (boneIds[i] >= MAX_BONES)
        {
            totalPosition = vec4(position, 1.0f);
            break;
        }
        vec4 localPosition = finalBonesMatrices[boneIds[i]] * ↙
            ↳ vec4(position, 1.0f);
        totalPosition += localPosition * weights[i];
        vec3 localNormal = mat3(finalBonesMatrices[boneIds[i]]) * ↙
            ↳ normal;
    }
    gl_Position = mvp * totalPosition;
    textureCordinates = TextureCordinates;
}
```

```
#version 460 core

struct Material
{
    sampler2D diffuse;
};

out vec4 FragColor;

in vec2 textureCordinates;

uniform Material material;

void main()
{
    FragColor = texture(material.diffuse, textureCordinates);
}
```

4.2. Button

```
#version 460 core

layout (location = 0) in vec3 position;

uniform mat4 model;

void main ()
{
    gl_Position = model * vec4(position, 1.0f);
}
```

```
#version 460 core

out vec4 FragColor;

uniform vec3 color;

void main ()
{
    FragColor = vec4(color, 1.0f);
}
```

4.3. Cursor

```
#version 460 core

layout (location = 0) in vec3 position;

uniform mat4 model;

void main ()
{
    gl_Position = model * vec4(position, 1.0f);
}
```

```
#version 460 core

out vec4 FragColor;

void main ()
{
    FragColor = vec4(1.0f);
}
```

4.4. Light

```
#version 460 core

layout (location = 0) in vec3 position;

uniform mat4 mvp;

void main ()
{
    gl_Position = mvp * vec4(position, 1.0f);
}
```

```
#version 460 core

layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;

uniform vec3 lightColor;

```

```
8  void main ()
9  {
10     FragColor = vec4(lightColor, 1.0f);
11     BrightColor = vec4(0.0f, 0.0f, 0.0f, 1.0f);
12
13     float brightness = dot(FragColor.rgb, vec3(0.2126f, 0.7152f, ↙
           ↳ 0.0722f));
14     if (brightness > 1.0f)
15     {
16         BrightColor = FragColor;
17     }
18 }
```

4.5. Mesh

```
1  #version 460 core
2
3  layout (location = 0) in vec3 position;
4  layout (location = 2) in vec3 normal;
5  layout (location = 3) in vec2 textureCoordinates;
6
7  out vec3 FragmentPosition;
8  out vec3 Normal;
9  out vec2 TextureCoordinates;
10
11 uniform mat4 model;
12 uniform mat4 view;
13 uniform mat4 projection;
14
15 void main()
16 {
17     FragmentPosition = vec3(model * vec4(position, 1.0f));
18
19     Normal = mat3(transpose(inverse(model))) * normal;
20     TextureCoordinates = textureCoordinates;
21
22     gl_Position = projection * view * vec4(FragmentPosition, 1.0f);
23 }
```

```
1  #version 460 core
2
3  layout (location = 0) out vec4 FragColor;
4  layout (location = 1) out vec4 BrightColor;
5
```

```
6  struct Material
7  {
8      sampler2D diffuse;
9      sampler2D specular;
10     float shininess;
11 };
12
13 struct DirectionalLight {
14     vec3 direction;
15
16     vec3 ambient;
17     vec3 diffuse;
18     vec3 specular;
19 };
20
21 struct PointLight
22 {
23     vec3 position;
24
25     vec3 ambient;
26     vec3 diffuse;
27     vec3 specular;
28
29     float constant;
30     float linear;
31     float quadratic;
32 };
33
34 struct SpotLight
35 {
36     bool on;
37
38     vec3 position;
39     vec3 direction;
40     float cutOff;
41     float outerCutOff;
42
43     vec3 ambient;
44     vec3 diffuse;
45     vec3 specular;
46
47     float constant;
48     float linear;
49     float quadratic;
50 };
51
52 in vec3 FragmentPosition;
```

```
53 in vec3 Normal;
54 in vec2 TextureCoordinates;
55
56 #define NUMBER_OF_POINT_LIGHTS 1
57
58 uniform vec3 viewPosition;
59 uniform Material material;
60 uniform PointLight pointLight[NUMBER_OF_POINT_LIGHTS];
61 uniform SpotLight spotLight;
62 uniform DirectionalLight directionalLight;
63
64 vec3 calculatePointLight(PointLight pointLight, vec3 Normal, vec3 ↙
       ↳ FragmentPosition, vec3 viewDirection);
65 vec3 calculateSpotLight(SpotLight spotLight, vec3 Normal, vec3 ↙
       ↳ FragmentPosition, vec3 viewDirection);
66 vec3 calculateDirectionalLight(DirectionalLight directionalLight, vec3 ↙
       ↳ Normal, vec3 FragmentPosition, vec3 viewDirection);
67
68 void main()
69 {
70     vec3 normal = normalize(Normal);
71     vec3 viewDirection = normalize(viewPosition - FragmentPosition);
72
73     vec3 result = calculateDirectionalLight(directionalLight, normal, ↙
           ↳ FragmentPosition, viewDirection);
74     for(int i = 0; i < NUMBER_OF_POINT_LIGHTS; i++)
75     {
76         result += calculatePointLight(pointLight[i], normal, ↙
               ↳ FragmentPosition, viewDirection);
77     }
78     if (spotLight.on)
79     {
80         result += calculateSpotLight(spotLight, normal, ↙
               ↳ FragmentPosition, viewDirection);
81     }
82     float brightness = dot(result, vec3(0.2126f, 0.7152f, 0.0722f));
83     BrightColor = vec4(0.0f, 0.0f, 0.0f, 1.0f);
84     if (brightness > 1.0f)
85     {
86         BrightColor = vec4(result, 1.0f);
87     }
88     FragColor = vec4(result, 1.0f);
89 }
90
91 vec3 calculateDirectionalLight(DirectionalLight directionalLight, vec3 ↙
       ↳ Normal, vec3 FragmentPosition, vec3 viewDirection)
92 {
```

```
93      vec3 ambient = directionalLight.ambient * ↙
            ↳ texture(material.diffuse, TextureCoordinates).rgb;
94
95      vec3 lightDirection = normalize(-directionalLight.direction);
96      float diffrence = max(dot(Normal, lightDirection), 0.0);
97      vec3 diffuse = directionalLight.diffuse * diffrence * ↙
            ↳ texture(material.diffuse, TextureCoordinates).rgb;
98
99      vec3 halfwayDirection = normalize(lightDirection + viewDirection);
100     float spec = pow(max(dot(viewDirection, halfwayDirection), 0.0f), ↙
            ↳ material.shininess);
101     vec3 specular = directionalLight.specular * spec * ↙
            ↳ texture(material.specular, TextureCoordinates).rgb;
102
103     return (ambient + diffuse + specular);
104 }
105
106 vec3 calculatePointLight(PointLight pointLight, vec3 Normal, vec3 ↙
        ↳ FragmentPosition, vec3 viewDirection)
107 {
108     vec3 ambient = pointLight.ambient * texture(material.diffuse, ↙
            ↳ TextureCoordinates).rgb;
109
110     vec3 lightDirection = normalize(pointLight.position - ↙
            ↳ FragmentPosition);
111     float diffrence = max(dot(Normal, lightDirection), 0.0f);
112     vec3 diffuse = pointLight.diffuse * diffrence * ↙
            ↳ texture(material.diffuse, TextureCoordinates).rgb;
113
114     vec3 halfwayDirection = normalize(lightDirection + viewDirection);
115     float spec = pow(max(dot(Normal, halfwayDirection), 0.0f), ↙
            ↳ material.shininess);
116     vec3 specular = pointLight.specular * spec * ↙
            ↳ texture(material.specular, TextureCoordinates).rgb;
117
118     float distance = length(pointLight.position - FragmentPosition);
119     float attenaution = 1.0f / (pointLight.constant + ↙
            ↳ pointLight.linear * distance + pointLight.quadratic * ↙
            ↳ (distance * distance));
120
121     ambient *= attenaution;
122     diffuse *= attenaution;
123     specular *= attenaution;
124
125     return (ambient + diffuse + specular);
126 }
127
```

```
128 vec3 calculateSpotLight(SpotLight spotLight, vec3 Normal, vec3 ↙
    ↳ FragmentPosition, vec3 viewDirection)
129 {
130     vec3 lightDirection = normalize(spotLight.position - ↙
        ↳ FragmentPosition);
131 
132     vec3 ambient = spotLight.ambient * texture(material.diffuse, ↙
        ↳ TextureCoordinates).rgb;
133     vec3 diffuse = vec3(0.0f, 0.0f, 0.0f);
134     vec3 specular = vec3(0.0f, 0.0f, 0.0f);
135 
136     float diffrence = max(dot(Normal, lightDirection), 0.0f);
137     diffuse = spotLight.diffuse * diffrence * ↙
        ↳ texture(material.diffuse, TextureCoordinates).rgb;
138 
139     vec3 halfwayDirection = normalize(lightDirection + viewDirection);
140     float spec = pow(max(dot(Normal, halfwayDirection), 0.0f), ↙
        ↳ material.shininess);
141     specular = spotLight.specular * spec * texture(material.specular, ↙
        ↳ TextureCoordinates).rgb;
142 
143     float theta = dot(lightDirection, normalize(-spotLight.direction));
144     float epsilon = (spotLight.cutOff - spotLight.outerCutOff);
145     float intensity = clamp((theta - spotLight.outerCutOff) / epsilon, ↙
        ↳ 0.0f, 1.0f);
146     diffuse *= intensity;
147     specular *= intensity;
148 
149     float distance = length(spotLight.position - FragmentPosition);
150     float attenaution = 1.0f / (spotLight.constant + spotLight.linear ↙
        ↳ * distance + spotLight.quadratic * (distance * distance));
151 
152     diffuse *= attenaution;
153     specular *= attenaution;
154 
155     return (ambient + diffuse + specular);
156 }
```

4.6. Text

```
1 #version 460 core
2 
3 layout (location = 0) in vec4 vertex;
4 
```

```
out vec2 TextureCoordinates;

uniform mat4 projection;

void main ()
{
    gl_Position = projection * vec4(vertex.xy, 0.1f, 1.0f);
    TextureCoordinates = vertex.zw;
}
```

```
#version 460 core

in vec2 TextureCoordinates;

out vec4 FragColor;

uniform sampler2D tex;
uniform vec3 textColor;

void main()
{
    vec4 sampled = vec4(1.0f, 1.0f, 1.0f, texture(tex, ↙
        ↳ TextureCoordinates).r);
    FragColor = vec4(textColor, 1.0f) * sampled;
}
```

4.7. Blur

```
#version 460 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec2 TextureCoordinates;

out vec2 textureCoordinates;

void main ()
{
    gl_Position = vec4(position, 1.0f);
    textureCoordinates = TextureCoordinates;
}
```

```
#version 460 core

```

```
3  out vec4 FragColor;
4
5  in vec2 textureCoordinates;
6
7  uniform sampler2D image;
8
9  uniform bool horizontal;
10 uniform float weigths[5] = float[] (0.2270270270f, 0.1945945946f, ↙
       ↳ 0.1216216216f, 0.0540540541f, 0.0162162162f);
11
12 void main ()
13 {
14     vec2 texelSize = 1.0f / textureSize(image, 0);
15     vec3 result = texture(image, textureCoordinates).rgb * weigths[0];
16     if (horizontal)
17     {
18         for (int i = 1; i < 5; ++i)
19         {
20             result += texture(image, textureCoordinates + ↙
                   ↳ vec2(texelSize.x * i, 0.0f)).rgb * weigths[i];
21             result += texture(image, textureCoordinates - ↙
                   ↳ vec2(texelSize.x * i, 0.0f)).rgb * weigths[i];
22         }
23     }
24     else
25     {
26         for (int i = 1; i < 5; ++i)
27         {
28             result += texture(image, textureCoordinates + vec2(0.0f, ↙
                   ↳ texelSize.y * i)).rgb * weigths[i];
29             result += texture(image, textureCoordinates - vec2(0.0f, ↙
                   ↳ texelSize.y * i)).rgb * weigths[i];
30         }
31     }
32     FragColor = vec4(result, 1.0f);
33 }
```

4.8. PostProcessing

```
1  #version 460 core
2
3  layout (location = 0) in vec3 position;
4  layout (location = 1) in vec2 TextureCoordinates;
5
```

```
out vec2 textureCoordinates;

void main ()
{
    textureCoordinates = TextureCoordinates;
    gl_Position = vec4(position, 1.0f);
}
```

```
#version 460 core

out vec4 FragColor;

in vec2 textureCoordinates;

uniform sampler2D hdrBuffer;
uniform sampler2D bloomBlur;
uniform bool hdr;
uniform bool gammaCorrection;
uniform bool bloom;

void main ()
{
    const float gamma = 2.2f;
    vec3 hdrColor = texture(hdrBuffer, textureCoordinates).rgb;
    vec3 bloomValue = texture(bloomBlur, textureCoordinates).rgb;
    vec3 result = hdrColor;
    if (bloom)
    {
         hdrColor += bloomValue;
    }
    if (hdr && gammaCorrection)
    {
        result *= 1.0f;
        result = hdrColor / (hdrColor + vec3(1.0f));

        result = pow(result, vec3(1.0f / gamma));
    }
    if(hdr)
    {
        result *= 1.0f;
        result = hdrColor / (hdrColor + vec3(1.0f));
    }
    if (gammaCorrection)
    {
        result = pow(hdrColor, vec3(1.0f / gamma));
    }
```

```
39      FragColor = vec4(result, 1.0f);
40  }
```

Zakończenie

Tworzenie silnika to wyzwanie, które wymaga niesamowitych nakładów pracy oraz umiejętności. Na każdym progu gdy wprowadzaliśmy kolejne zmiany napotykaliśmy problemy lub kilka rozwiązań, które charakteryzowały się różnorodnością czasu potrzebnego do ich zaimplementowania. Sztuka jest wybrać najpotrzebniejsze elementy i je zaimplementować trzeba też pamiętać, że z racji, że to nasz silnik powinniśmy myśleć przyszłościowo niż wybierać opcję, które będą szybkie do zaimplementowania ale trudne do długoterminowego rozwijania. Nawet osoby, które mają większe doświadczenie od nas wciąż wzbogacają swoje silniki do tworzenia gier o nowe rzeczy dlatego pracy nie widać końca. Aktualny projekt miał tylko pokazać wyzwania związane z zagadaniem tworzenia własnego silnika gier oraz pozytywów wiążących się z decyzją stworzenia własnego silnika gier.

Bibliografia

[1] HOBGARSKI, M. Przegl æad metod mapowania tonÛw obrazÛw wysokokontrastowych, 2005.

[2] LEARNOPENGL. Bloom. . Dostęp: 2023-01-22.

[3] OGDEV. Skeletal animation tutorial. . Dostęp: 2023-01-22.

[4] VAN DEN BERGEN, G. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools* (1997).

Spis rysunków

Listings