Algorytmy grafiki komputerowej - projekt

Temat:

Symulacja odbicia otoczenia w płaskich powierzchniach (np. podłoga, lustro) na przykładowej scenie 3D.

Autor:

Bartłomiej Marzec

1. Opis zastosowanych bibliotek i algorytmów:

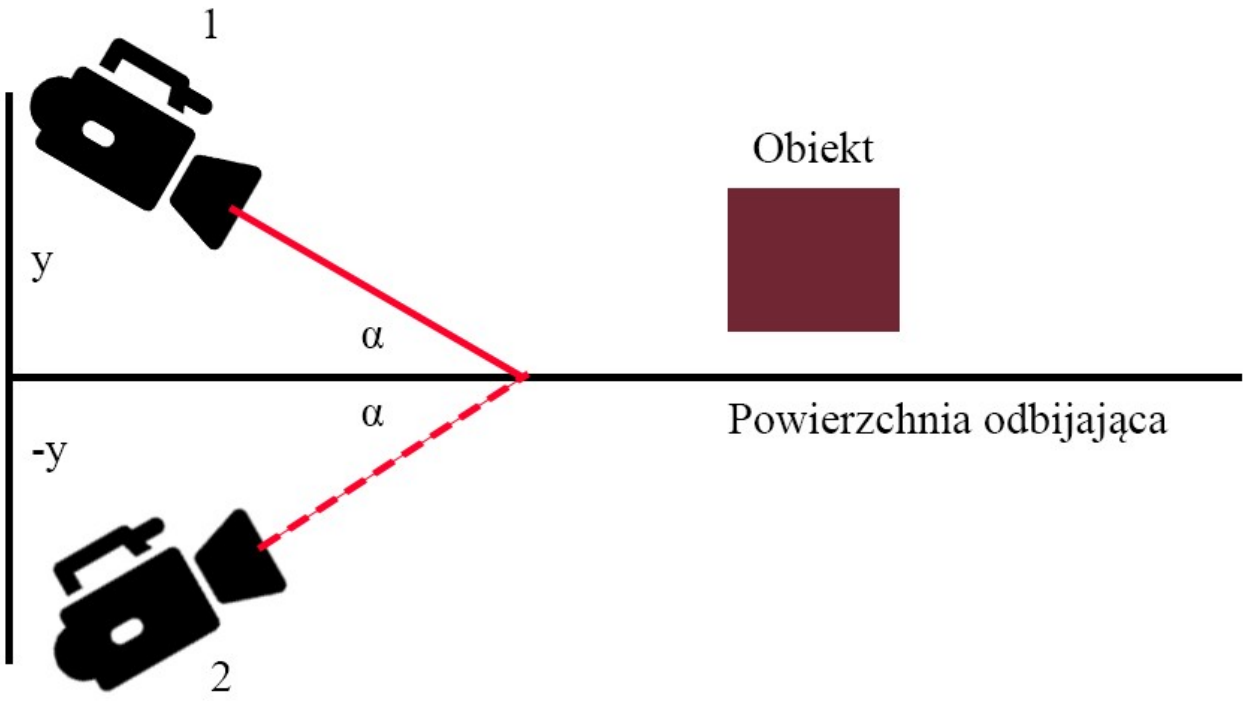
a) Biblioteki:

stb – zestaw bibliotek różnego przeznaczenia. W moim przypadku korzystam jedynie z modułu stb_image w celu wczytania tekstur z pliku.

OBJLoader – biblioteka używana do wczytywania modeli 3D zapisanych w formacje OBJ.

b) Algorytmy:

Algorytm którego użyłem do stworzenia odbić polega na prostej zmianie pozycji kamery na osi y i rotacji względem który ma odbijać scenę. Przykładowo, kiedy kamera jest na wysokości 20 nad lustrem którego y = 0, to algorytm zmienia tą pozycję na wysokość -20. Rotacja kamery również jest odbijana względem obiektu lustra. Ilustracja takiego przekształcenia pokazana została na rysunku 1.1. Następnie przekształcona kamera renderuje scenę do tzw. „frame buffor'a" (buffora ramki). Po zakończeniu procesu renderowania tekstura zostaje nałożona na lustro. Następnie czyszczone są buffory i renderowana jest właściwa scena z oryginalnej pozycji kamery która zostaje wyświetlona na ekranie.



rys. 1.1 Transformacja kamery

2. Opis implementacji:

a) Główna pętla

Główna pętla programu wykonuje ciągle 2 czynności: odświeżanie (funkcja **update**) i renderowanie (funkcja **render**).

```
while (!glfwWindowShouldClose(okno)) {
    update(&shader);
    render(&shader,&cubemapShader,&reflectionShader);
}
```

I) Update():

Funkcja update odświeża stan wszystkich obiektów które tego wymagają (np. pozycję i rotację kamery i obiektów dynamicznych, stan klawiszy oraz pozycję myszy). Odpowiednią prędkość rotacji obiektów zapewnia użycie zmiennej **dt**, która przechowuje czas pomiędzy aktualną i poprzednią klatką. Dzięki pomnożeniu wartości którą chcemy aktualizować przez tą zmienną, zapewniona jest stała prędkość np. obrotu obiektu (linijka 202).

```
194  void update(ShaderObj* shader) {
195      glfwPollEvents();
196      updateKeyboard();
197
198      currTime = static_cast<float>(glfwGetTime());
199      dt = currTime - lastTime;
200      lastTime = currTime;
201
202      renderObjects.at(0)->mesh->rotate(glm::vec3(0.f, 90.f * dt, 0.f));
203
204      if (cameraRotActive) {
205          rotateCamera();
206      }
207      camera.UpdateMatrix(shader);
208
209      mainCubemap.updateMatrix(camera);
210
211  }
```

II) Render():

Funkcja render obsługuje proces renderowania każdego obiektu na scenie. Na początku czyści buffory i resetuje kolor. Następnie w pętli for (l. 155) renderuje obraz do tekstur wszystkich obiektów odbijających. Proces ten składa się z kilku kroków. Na początku (l.156) zostaje obliczony dystans pomiędzy kamerą i obiektem, potem kamera zostaje odpowiednio obrócona (l.157). Następnie następuje aktualizacja matryc kamery i tła. W linijce 160 wywołana zostaje funkcja **bindReflections()**, która sprawia, że kolejne wyrenderowane obiekty będą trafiały do obiektu buffora ramki obiektu odbijającego. Następnie kolejna pętla (l.161) wywołuje funkcję renderującą każdego innego obiektu który nie jest obiektem odbijającym. Potem renderowane jest tło. W linijce 168 na aktualnie iterowanym obiekcie zostaje wywołana zostaje funkcja **unbindReflections()**, która kończy render następnych obiektów do obiektu buffora ramki. W linijce 169 następuję obrócenie kamery do pozycji początkowej, a później aktualizowane są macierze kamery i tła.

W pętli od linijki 174 wykonuje się render właściwy. Każdy obiekt zostaje

wyrenderowany na ekran użytkownika odpowiednim shaderem zależnym od typu obiektu. Po skończeniu pętli (l.180) renderowane jest tło. Następnie funkcja **glfwSwapBuffers()** wyświetla obraz na ekranie.

```
150  void render(ShaderObj *shader, ShaderObj *cubemapShader, ShaderObj *reflectionShader) {
151      glClearColor(0.f, 0.f, 0.f, 1.f);
152      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
153
154      // Render odbić
155      for (size_t j = 0; j < reflectiveRenderObjects.size(); j++) {
156          glm::vec3 distance = (camera.position - reflectiveRenderObjects.at(j)->position) * 2.f;
157          camera.flip(distance);
158          camera.UpdateMatrix(shader);
159          mainCubemap.updateMatrix(camera);
160          reflectiveRenderObjects.at(j)->bindReflections();
161          for (size_t i = 0; i < renderObjects.size(); i++)
162          {
163              if (!renderObjects.at(i)->reflective)
164                  renderObjects.at(i)->render(shader);
165          }
166
167          mainCubemap.render(cubemapShader);
168          reflectiveRenderObjects.at(j)->unbindReflections(rozmiarOkna);
169          camera.flip(-distance);
170          camera.UpdateMatrix(shader);
171          mainCubemap.updateMatrix(camera);
172      }
173      // Render głowny
174      for (size_t i = 0; i < renderObjects.size(); i++)
175      {
176          if (!renderObjects.at(i)->reflective)
177              renderObjects.at(i)->render(shader);
178          else renderObjects.at(i)->render(reflectionShader,&camera);
179      }
180      mainCubemap.render(cubemapShader);
181
182      //std::cout << camera.orientation.x << " : " << camera.orientation.y << " : " << camera.orientation.z << endl;
183
184      // Koniec renderu
185      glfwSwapBuffers(okno);
186      glFlush();
187
188      glBindVertexArray(0);
189      glUseProgram(0);
190      glActiveTexture(0);
191      glBindTexture(GL_TEXTURE_2D, 0);
192  }
```

b) klasa RenderObject

W tej klasie przechowywane są informację o każdym obiekcie na scenie. Najważniejszymi jej funkcjami są:

I) render():

W tej funkcji następuję render obiektów. Na początku (l.93) wskazany przez parametr shader jest aktywowany. Następnie sprawdzane jest czy obiekt ma teksturę oraz jednocześnie czy nie jest obiektem odbijającym, jeśli stwierdzenie daję wynik **true**, to przypisywana jest domyślna tekstura (obiekty odbijające w polu **texture** nie mają przypisanej żadnej wartości). Następnie sprawdzany jest typ obiektu, jeśli nie jest to obiekt odbijający to do renderu przypisywana jest tekstura z pola **texture**. Jeśli jest to obiekt odbijający to przypisywana jest tekstura z pola **reflectionTexture**, zapełnionego przez proces renderu odbić z pętli głównej. Następnie do shadera odbić trafia tekstura oraz matryca kamery. W linijce 109 następuję render obiektu wczytanego z pliku (pole **mesh**) oteksturowanego odpowiednimi teksturami. Następnie resetowane są pola przypisanej do renderu tekstury i shadera.

```
92      void render(ShaderObj* shader, Camera *camera = NULL) {
93          shader->Use();
94          if (!this->hasTexture && !reflective) {
95              SetTexture(defaultTextureFile);
96          }
97
98          glActiveTexture(GL_TEXTURE0);
99          if (!reflective) {
100             glBindTexture(GL_TEXTURE_2D, texture);
101         }
102         else {
103             glBindTexture(GL_TEXTURE_2D, reflectionTexture);
104             shader->set1i(0, "reflectionTexture");
105             shader->setMat4fv(camera->projection * camera->view, "cameraMatrix");
106         }
107         //shader->set1i(texture, "tex0");
108
109         this->mesh->render(shader);
110         glBindTexture(GL_TEXTURE_2D, 0);
111         shader->Stop();
112     }
```

II) bindReflections(), unbindReflections(), initReflections():

Zestaw funkcji odpowiadających za odbicia obiektu. Pierwsza z nich wywołuję zestaw funkcji OpenGL które umożliwiają render do buffora ramki. Czyści także buffor z pozostałości z poprzednich klatek. Na końcu ustawia odpowiednią rozdzielczość renderu zdefiniowaną przez programistę. Druga funkcja kończy render i przywraca prawidłową rozdzielczość renderu przypisaną do kamery. Ostatnia funkcja inicjuję obiekt odbijający. Na początku tworzy nowy obiekt buffora ramki, następnie generuje teksturę do której będzie zapisywany obraz odbić, potem tworzy obiekt buffora głębokości, a na końcu ustawia zmienną **reflective** na **true**, co oznacza że od teraz, ten obiekt jest traktowany jako odbijający.

```
56      void bindReflections() {
57          glBindTexture(GL_TEXTURE_2D, 0);
58          glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
59          glClearColor(0.f, 0.f, 0.f, 1.f);
60          glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
61          glViewport(0, 0, widthFB, heightFB);
62      }
63
64      void unbindReflections(glm::vec2 rozmiarOkna) {
65          glBindFramebuffer(GL_FRAMEBUFFER, 0);
66          glViewport(0, 0, rozmiarOkna.x, rozmiarOkna.y);
67      }
68
69
70      void initReflections(ShaderObj* shader = NULL) {
71          glGenFramebuffers(1, &frameBuffer);
72          glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
73          glDrawBuffer(GL_COLOR_ATTACHMENT0);
74
75          glGenTextures(1, &reflectionTexture);
76          glBindTexture(GL_TEXTURE_2D, reflectionTexture);
77          glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, this->widthFB, this->heightFB, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
78          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
79          glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
80          glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, reflectionTexture, 0);
81
82          glGenRenderbuffers(1, &depthBuffer);
83          glBindRenderbuffer(GL_RENDERBUFFER, depthBuffer);
84          glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, widthFB, heightFB);
85          glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuffer);
86
87          reflective = true;
88
89      }
```

c) Klasa Camera

W tej klasie znajdują się pola i metody obsługujące kamerę.

I) UpdateMatrix():

Funkcja ma na celu odświeżenie wartości zmiennych **view** i **projection** kamery, oraz przekazanie ich iloczynu do shadera (część „VP” w modelu MVP, „M” jest przekazywane przez każdy renderowany obiekt).

```
45    void UpdateMatrix(ShaderObj* shader) {
46
47        shader->Use();
48
49        view = glm::lookAt(position, position + orientation, up);
50        projection = glm::perspective(glm::radians(45.0f), windowSize.x / windowSize.y, 0.01f, 100.0f);
51
52        glUniformMatrix4fv(glGetUniformLocation(shader->ID, uniformLocation), 1, GL_FALSE, glm::value_ptr(projection * view));
53        shader->Stop();
54    }
```

II) Flip(), Move(), Rotate():

Zestaw funkcji transformujących kamerę. Pierwsza odpowiada za przekształcenie kamery w przypadku obiektu odbijającego, czyli na procesie opisanym w rozdziale 1.b. Druga odpowiada za ruch kamery w świecie. Ostatnia obsługuję rotację kamery na podstawie przekazanych wartości (pozycji myszy i zmiennej **dt**).

```
56    void flip(glm::vec3 distance) {
57        this->position.y -= distance.y;
58        orientation = glm::rotate(orientation, (float)glm::radians(180.f), up);
59        orientation = glm::rotate(orientation, (float)glm::radians(180.f), glm::normalize(glm::cross(orientation, up)));
60
61    }
62
63    void Move(const int direction, const float& dt) {
64        switch (direction) {
65        case FORWARD:
66            this->position += this->speed * this->orientation * dt;
67            break;
68        case RIGHT:
69            this->position += this->speed * dt * glm::normalize(glm::cross(orientation, up));
70            break;
71        case BACK:
72            this->position += this->speed * dt * (-orientation);
73            break;
74        case LEFT:
75            this->position += this->speed * dt * (-glm::normalize(glm::cross(orientation, up)));
76            break;
77        case UP:
78            this->position += this->speed * dt * up;
79            break;
80        case DOWN:
81            this->position += this->speed * dt * -up;
82            break;
83        }
84    }
85
86    void Rotate(const glm::vec2 mouseRot, const float& dt) {
87        glm::vec3 newOrint = glm::rotate(orientation, glm::radians(-(mouseRot.x * (sensitivity*dt))), glm::normalize(glm::cross(orientation, up)));
88
89        if (!((glm::angle(newOrint, up) <= glm::radians(5.0f)) || (glm::angle(newOrint, -up) <= glm::radians(5.0f)))) {
90            orientation = newOrint;
91        }
92        orientation = glm::rotate(orientation, glm::radians(-(mouseRot.y * (sensitivity*dt))), up);
93    }
94};
```

d) Klasa Mesh

W klasie mesh przechowywane są informacje na temat siatki z której składa się obiekt.

I) initVao()

Funkcja na początku (l.35-40) inicjuje odpowiednie elementy języka openGL potrzebne do opisania obiektu 3D na scenie. Są to kolejno: tablica wierzchołków (VAO), oraz buffor tablicy (VBO). VBO

tworzony jest na podstawie ilości wierzchołków wczytanego obiektu. Następnie od linijki 42 w obiekcie VBA wskazywane są odpowiednie miejsca z których shadery będą mogły odczytywać zmienne.

```
33    void initVAO()
34    {
35        glGenVertexArrays(1, &this->VAO);
36        glBindVertexArray(this->VAO);
37
38        glGenBuffers(1, &this->VBO);
39        glBindBuffer(GL_ARRAY_BUFFER, this->VBO);
40        glBufferData(GL_ARRAY_BUFFER, this->nrOfVertices * sizeof(Vertex), this->vertexArray, GL_STATIC_DRAW);
41
42        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (GLvoid*)offsetof(Vertex, position));
43        glEnableVertexAttribArray(0);
44        glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (GLvoid*)offsetof(Vertex, texcoord));
45        glEnableVertexAttribArray(1);
46
47        glBindVertexArray(0);
48    }
```

II) Render()

Funkcja obsługująca render obiektu. Na początku w linijce 110 odświeżana zostaje matryca modelu na podstawie jego pozycji, rotacji i skali. Nastepnie do shadera przekazana zostaje zmienna **ModelMatrix** (część „M” w modelu MVP). Potem aktywowana zostaje tablica VAO zawierające m.in. informację o zmiennych które shader odczytuje. W linijce 115 wywołana zostaje funkcja openGL’a **glDrawArrays()** która rysuje dane w postaci trójkątów odczytane z tablicy VAO.

```
108   void render(ShaderObj* shader)
109   {
110       this->updateModelMatrix();
111       shader->setMat4fv(this->ModelMatrix, "ModelMatrix");
112       shader->Use();
113       glBindVertexArray(this->VAO);
114
115       glDrawArrays(GL_TRIANGLES, 0, this->nrOfVertices);
116
117       glBindVertexArray(0);
118       glActiveTexture(0);
119       glBindTexture(GL_TEXTURE_2D, 0);
120   }
```

e) Shadery

I) Shader obsługujący renderowanie obiektów (shader1)

I.a) VertexShader

Shader na podstawie zmiennej odczytanej z tablic VAO obiektu, oblicza koordynaty tekstury (**texCoord**) (a ze względu na sposób przekazania, oś y musi zostać pomnożona przez -1, inaczej tekstura byłaby do góry nogami). Następnie do zmiennej **gl_Position** (określającej pozycje obecnie przetwarzanego wierzchołka) zostaje przypisany iloczyn macierzy kamery, modelu i wektora pozycji obiektu.

```glsl
#version 440

layout(location = 0) in vec3 aPos;
layout(location = 1) in vec2 aTex;

out vec2 texCoord;

uniform mat4 cameraMatrix;
uniform mat4 ModelMatrix;

void main(){
    texCoord = vec2(aTex.x, aTex.y*-1.f);
    gl_Position =  cameraMatrix*ModelMatrix*vec4(aPos, 1.0);
}
```

I.b) FragmentShader

Shader fragmentów ustawia kolor piksela z tekstury odczytanej ze zmiennej tex0 o koordynatach przekazanych przez shader wierzchołków.

```glsl
#version 440
out vec4 color;

in vec2 texCoord;
uniform sampler2D tex0;

void main(){
    color = texture(tex0, texCoord);
}
```

II) Shader obsługujący renderowanie obiektów odbijających (odbicie)

I.a) VertexShader

Shader oblicza tą samą pozycje co w przypadku poprzedniego shadera wierzchołków, zamiast przekazywać koordynaty tekstury, to przekazana zostaje właśnie ta pozycja.

```glsl
#version 440

layout(location = 0) in vec3 aPos;

out vec4 position;

uniform mat4 cameraMatrix;
uniform mat4 ModelMatrix;

void main(){
    position = cameraMatrix*ModelMatrix*vec4(aPos, 1.0);
    gl_Position =  position;
}
```

I.b) FragmentShader
Shader fragmentów oblicza koordynaty odbitej tekstury, a następnie przypisuję do zmiennej color odpowiedni fragment tekstury. W linijce 8 do równania została dodana wartość 0.5, ponieważ bez niej, odbicie było przesunięte do środka, co ilustruje rysunek 2.b. Dodatkowo należało obrócić cały powstały wynik na osi y, tak jak w przypadku pierwszego shadera.

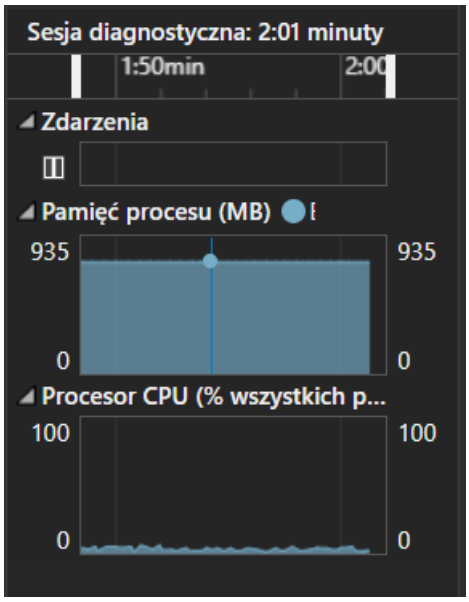
```
1  #version 440
2  out vec4 color;
3
4  in vec4 position;
5  uniform sampler2D reflectionTexture;
6
7  void main(){
8      vec2 coords = (position.xy/position.w)/2 +0.5;
9      color = texture(reflectionTexture, vec2(coords.x, -coords.y));
10 }
```

rys. 2.b

3. Testy wydajnościowe i jakościowe:

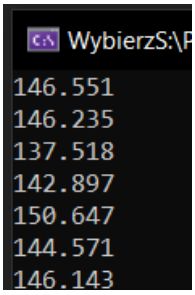
Pamięć:
Program nie wykazuję żadnych wycieków pamięci. Potwierdza to poniższy obraz z sesji diagnostycznej programu Visual Studio 2022. Można również zauważyć znikome użycie procesora. Wykorzystanie pamięci zależy od ilości i wielkości obiektów i tekstur wykorzystanych w

programie.



Klatki:

Program na domyślnej scenie (około 700MB pamięci wykorzystanej przez tekstury i obiekty) osiąga około 144 klatek, co jest moim odświeżaniem monitora.



Przy wykorzystaniu ponad 3GB pamięci na obiekty, program dalej osiąga 144 klatek na sekundę.