Sprawozdanie z projektu	
Politechnika Świętokrzyska	
Informatyka, III rok, 5 semestr	
Grafika Komputerowa - projekt	
Temat:	Gra 3D
Autorzy:	Adam Furmanek Tomasz Gębski

Opis tematyki projektu

Celem projektu było wykonanie gry 3D opartej na świecie stworzonego z sześciennych voxeli. Inspiracją dla pracy była gra Minecraft i podobnie jak w niej, gracz może poruszać się po świecie modyfikując dowolnie teren. Świat jest zbudowany z kilku rodzajów oteksturowanych bloków, które gracz może dowolnie niszczyć i tworzyć, poruszając się przy użyciu myszki i klawiatury. Klawiatura odpowiedzialna jest za chodzenie i skakanie, natomiast myszka za ruch kamery. Gra obsługuje 8 slotów zapisu gry, które można dowolnie wczytywać i nadpisywać.

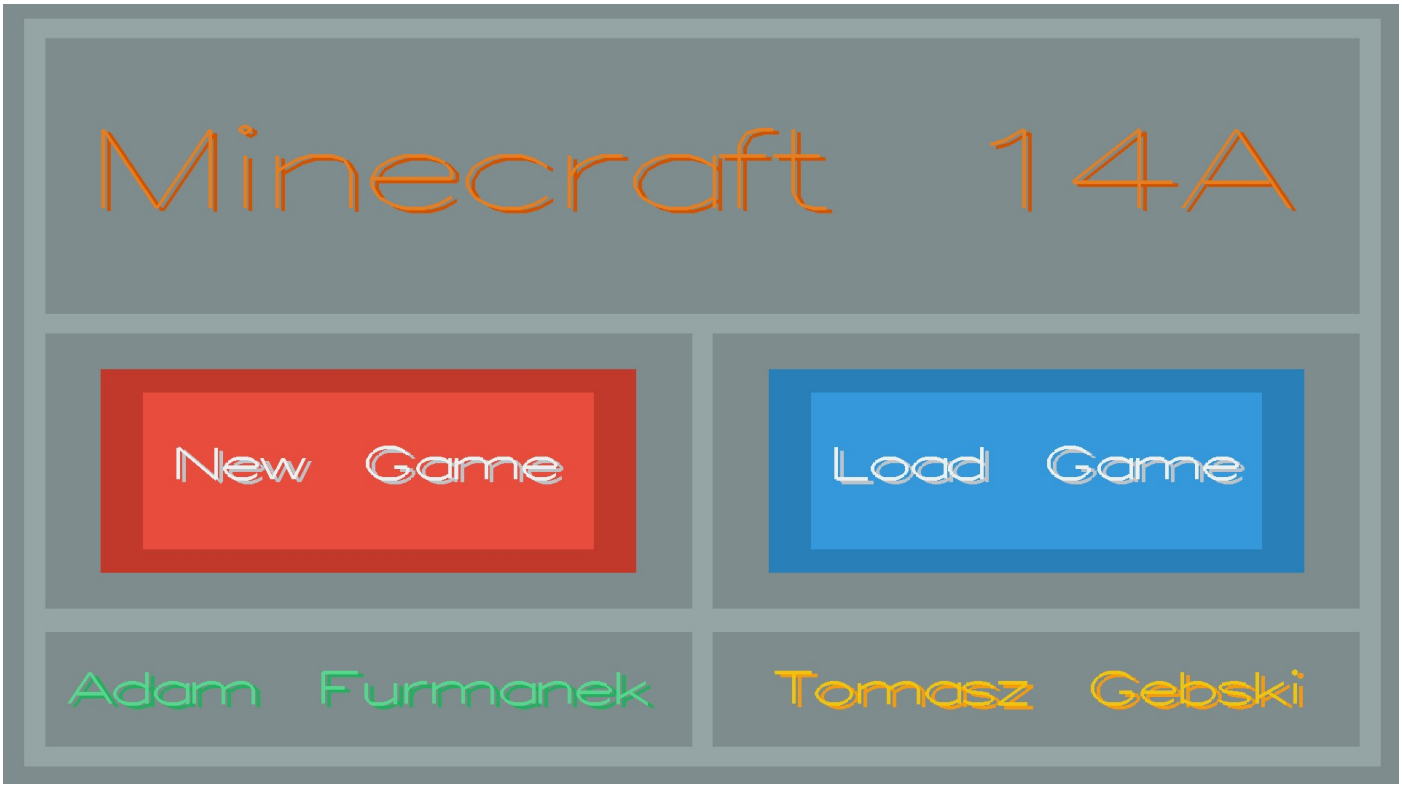
Użyty język programowania, środowisko, biblioteki oraz OS

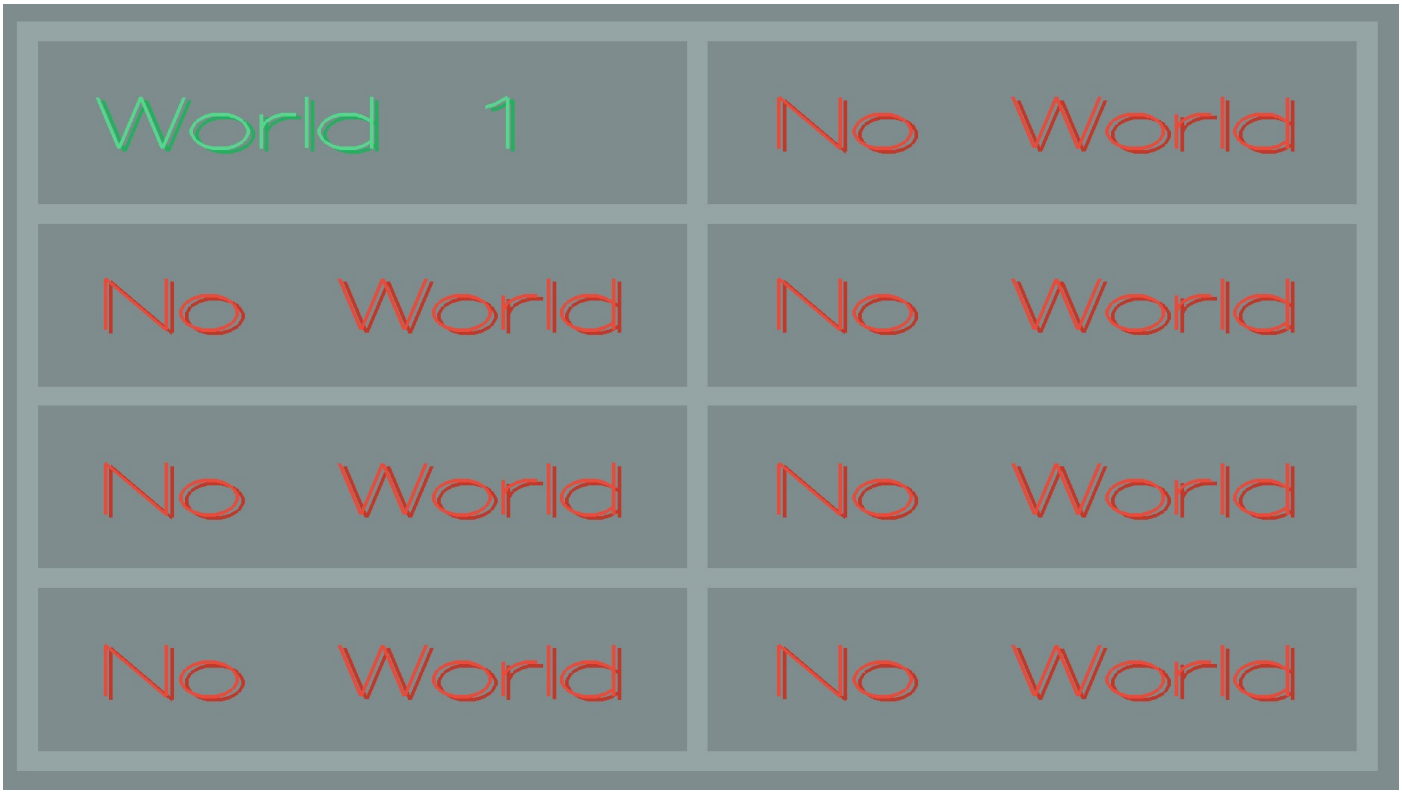
Projekt wykonany jest w środowisku Microsoft Visual Studio w języku C++. Używane biblioteki to OpenGL oraz GLUT, pobrane przy użyciu modułu Nuget Package Manager. Gra działa na systemie operacyjnym Windows.

Instrukcja kompilacji/uruchomienia, opis działania i instrukcja obsługi

Program kompiluje się do pliku .exe i wymaga pliku freeglut.dll oraz folderu Textures w tym samym katalogu. Ewentualne pliki zapisu gry, również będą w tym samym folderze. Uruchamiając program pokazuje się główne menu, pozwalające na stworzenie nowego świata, lub wczytanie istniejącego. Przy wyborze tej drugiej opcji, wyświetlone istniejące zapisy, które można wczytać. Po wczytaniu lub stworzeniu nowej mapy, uruchamia się rozgrywka. Gracz steruje postacią klawiszami WASD (chodzenie do przodu, do tyłu i na boki), skok wykonuje się przy użyciu SPACE. Dodatkowe klawisze to: Q – sprint, F – włączenie/wyłączenie latarki, G – włączenie/wyłączenie mgły, 9/0 – zwiększenie/zmniejszenie widocznego obszaru, -/= - zwiększenie/zmniejszenie obszaru renderowanego świata. Przycisk V pozwala na zapis gry na jednym z ośmiu slotów, natomiast ESC wychodzi do głównego menu. W trakcie rozgrywki myszka reguluje ruch kamery oraz pozwala na usunięcie i stworzenie nowego bloku (w miejscu, w którym wskazuje kursor) oraz pozwala na wybór tworzonego bloku. Bez względu na to czy trwa rozgrywka, czy uruchomione jest menu, klawisze 1, 2, 3 pozwalają na zmianę rozmiaru okna.

Zrzuty ekranu z przykładowym działaniem





Menu główne oraz menu zapisu

Drzewo stworzone przez gracza. Z tyłu widoczna granica świata oraz mgła.

Dom stworzony przez gracza w nocy. Z tyłu widoczny księżyc, gracz ma włączoną latarkę.

Wschód Słońca.

Poranek.

Pozostałe dwa domyślne rozmiary okna (choć gracz może je dowolnie modyfikować).

Zmniejszony zasięg renderowania.

Zwiększony zasięg renderowania.

Zmniejszone pole widzenia.

Zwiększone pole widzenia.

Podział pracy pomiędzy poszczególnymi członkami zespołu

Nie da się dokładnie wyznaczyć który członek zespołu jest odpowiedzialny za którą funkcjonalność, ponieważ członkowie wielokrotnie pracowali nad tymi samymi mechanikami, stale dodając lub zmieniając fragmenty kodu. Można jednak ocenić stosunek wykonanych prac na 60%/40% członków Adam Furmanek/Tomasz Gębski.

Opis użytych algorytmów i najważniejszych fragmentów implementacji

Klasa Window

Ta klasa oddelegowuje wykonanie operacji obiektowi odpowiedniej klasy w zależności od stanu.
Stany: 0 – menu główne, 1 – menu wczytywania mapy, 2 – rozgrywka, 3 – menu zapisu mapy.
Przykład metody „oddelegowującej”:

```
49  void Window::Display() {
50
51      if (state == 0)
52          menu->MenuDisplay();
53      else if (state == 1)
54          menu->SavingMenuDisplay();
55      else if(state == 2)
56          game->GameDisplay();
57      else if(state == 3)
58          menu->SavingMenuDisplay();
59
60      glutPostRedisplay();
61  }
62
```

Display, wywołuje jedynie odpowiednią metodę aby wykonać wyświetlenie obrazu.

Klasa Game

Ta klasa jest odpowiedzialna za rozgrywkę i ma wiele ważnych algorytmów i mechanik. Oto kilka z nich:

```cpp
void Game::GameDisplay() {

    // Czyszczenie bufora koloru i bufora głębi.
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Kolor tła.
    ambient->clearColor();

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60, (glutGet(GLUT_WINDOW_WIDTH) * 1.0 / glutGet(GLUT_WINDOW_HEIGHT)), 0.01, 64);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Rysowanie celownika.
    player->DrawCursor();
    // Rysowanie trzymanego bloku.
    textures->DrawSelectedBlock(interaction->getHandID());

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(player->getViewField(), (glutGet(GLUT_WINDOW_WIDTH) * 1.0 / glutGet(GLUT_WINDOW_HEIGHT)), 0.01, 64);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    camera->LookAt(player->getX(), player->getY(), player->getZ());

    player->Flashlight(camera->getVector());

    ambient->AmbientDisplay(player->getX(), player->getY(), player->getZ(), textures->getViewDistance());

    textures->TexturesDisplay(player->getX(), player->getY(), player->getZ());

    // Rysowanie ramki śledzonego bloku.
    interaction->DrawCubeBorder();
    // Skierowanie poleceń do wykonania.
    glFlush();
    // Zamiana buforów koloru.
    glutSwapBuffers();

}
```

Metoda wyświetlająca: Czyści bufory, ustawia kolor nieba, rysuje prosty interface, ustawia kierunek kamery, wywołuje latarkę, rysuje otoczenie, rysuje świat z sześcianów.

```cpp
void Game::SaveGame(int id) {
    string name = "save1.sav";
    name[4] = (id + 48);

    ofstream plik(name.c_str(), ios::binary);
    char m;
    for (int i = 0; i < 600; i++) {
        for (int j = 0; j < 64; j++) {
            for (int k = 0;k < 600;k++) {
                m = map->get(i, j, k);
                plik.write((const char*)&m, sizeof(char));
            }
        }
    }
    plik << "\n";
    plik << ambient->getTime() << "\n";
    plik << textures->getViewDistance() << "\n";
    plik << camera->getVector()[0] << "\n" << camera->getVector()[1] << "\n" << camera->getVector()[2] << "\n" << camera->getAngleXZ() << "\n";
    plik << player->getX() << "\n" << player->getY() << "\n" << player->getZ() << "\n" << player->getJump() << "\n" << player->getFallingSpeed()
    plik << interaction->getHandID() << "\n";
    plik.close();

}
```

Zapis najważniejszych parametrów gry: mapa, pora dnia, kamera, pozycja gracza

```
188         plik >> time;
189         ambient = new Ambient(time);
190         float viewDistance;
191         plik >> viewDistance;
192         textures = new Textures(map, viewDistance);
193         float x, y, z;
194         float angleXZ;
195         plik >> x;
196         plik >> y;
197         plik >> z;
198         plik >> angleXZ;
199         camera = new Camera(x, y, z, angleXZ);
200         int jump;
201         float fallingSpeed;
202         bool flashlight;
203         float viewField;
204         plik >> x;
205         plik >> y;
206         plik >> z;
207         plik >> jump;
208         plik >> fallingSpeed;
209         plik >> flashlight;
210         cout << flashlight << endl;
211         plik >> viewField;
212         player = new Player(map, x, y, z, jump, fallingSpeed, flashlight, viewField);
213         int handID;
214         plik >> handID;
215         interaction = new Interaction(map, player, handID);
216
217         plik.close();
218     }
219
220     void Game::CreateGame() {
221         map = new Map();
222         ambient = new Ambient();
223         textures = new Textures(map);
224         camera = new Camera();
225         player = new Player(map);
226         interaction = new Interaction(map, player);
227         GameInit();
228     }
```

Porównanie metod odczytu gry i tworzenia nowej gry. Linie do 217 pokazują odczytywanie parametrów i tworzenie nowych obiektów, z których Game będzie korzystał, podając odczytane parametry. Metoda CreateGame tworzy czystą grę, a konstruktory poszczególnych obiektów przyjmują argumenty domyślne.

Klasa Player

```
40  void Player::PressKey(unsigned char key) {
41
42      switch (key) {
43      case'f':
44          if (flashlight)
45              glDisable(GL_LIGHT1);
46          else
47              glEnable(GL_LIGHT1);
48          flashlight = !flashlight;
49          break;
50      case 'a': deltaMoveSides = -0.12f; break;
51      case 'd': deltaMoveSides = 0.12f; break;
52      case 'w': deltaMoveStraight = 0.12f; break;
53      case 's': deltaMoveStraight = -0.12f; break;
54      case 'q':
55          if (deltaMoveStraight == 0.12f) {
56              deltaMoveStraight += 0.07f;
57              viewField += 1.2;
58          }
59          break;
60      case ' ':
61          if (jump == 0)
62              jump = 20;
63          break;
64      case 'r':
65          x = 300;
66          y = 60;
67          z = 300;
68          fallingSpeed = 0;
69          jump = 0;
70          break;
71      }
72  }
```

Ta metoda obsługuje sterowania gracza. Przycisk F włącza/wyłącza latarkę. WASD dodają prędkość w odpowiednim kierunku, Q dodatkowo zwiększa prędkość do przodu, jeśli ta jest zwiększona, SPACE ustawia wartość skoku na 20, R resetuje pozycję gracza.

```
    // Jeśli wykonywany jest skok.
    if (jump > 1) {
        // Potencjalna zmiana wysokości y.
        float deltaY;
        // Zmiana wysokości jest dyktowana tym, w którym momencie lotu jest gracz.
        switch (jump) {
        case 20: case 19: case 18: case 17: case 16:
            deltaY = 0.15f;
            break;
        case 15: case 14: case 13:
            deltaY = 0.14f;
            break;
        case 12: case 11: case 10:
            deltaY = 0.13f;
            break;
        case 9: case 8:
            deltaY = 0.11f;
            break;
        case 7:
            deltaY = 0.09f;
            break;
        case 6:
            deltaY = 0.07f;
            break;
        case 5:
            deltaY = 0.05f;
            break;
        case 4:
            deltaY = 0.02f;
            break;
        case 3: case 2:
            deltaY = 0.00f;
            break;
        }

        // Sprawdzenie kolizji ciała na wysokości głowy z potencjalnie nową wartością y.
        if (!Collision(x, y + 2.7f + deltaY, z))
            // Przypisanie nowej wartości y.
            y += deltaY;

        // Zmniejszenie licznika długości skoku.
        jump -= 1;
    }
```

Fragment metody obliczającej ruch w pionie (skok i spadanie). Jeśli wciąż wykonywany jest skok gracz unosi się do góry, jeśli nie uderzy w jakiś blok ponad nim. Wartość dodanej wysokości jest różna w zależności od momentu skoku, dzięki czemu jest on dobrze wyprofilowany.

```
    // Grawitacja.
    else if (y > 0) {

        // Obliczenie potencjalnie nowej wartości y.
        float newY = y - fallingSpeed;
        // Sprawdzenie kolizji ciała na wysokości butów z potencjalnie nową wartością y.
        if (!Collision(x, newY, z)) {
            // Przypisanie nowej wartości y.
            y = newY;
            // Zwiększenie prędkości spadania.
            fallingSpeed += 0.006f;
        }
        else {
            // Wyrównanie wysokości y do liczby całkowitej.
            y = (int)(y);
            // Zakończenie skoku. Pozwala wykonać nowy.
            jump = 0;
            // Zresetowanie prędkości spadania do podstawowej.
            fallingSpeed = 0.1f;
        }
    }
}
```

Druga część tej samej metody. Jeśli nie jest już wykonywany rozpoczyna się spadanie. Im dłużej trwa spadanie, tym szybsza jest jego prędkość. Prędkość spadania zeruje się przy kontakcie z jakimś blokiem.

```
166  void Player::ComputeMove(float x1, float z1) {
167      // Jeśli wykonano ruch w przód/tył.
168      if (deltaMoveStraight) {
169          // Obliczenie potencjalnie nowej wartości x.
170          float newX = x + (deltaMoveStraight * x1);
171          // Sprawdzenie kolizji ciała na wysokości butów, pasa i głowy z potencjalnie nową wartością x.
172          if (!Collision(newX, y, z) && !Collision(newX, y + 1, z) && !Collision(newX, y + 2, z))
173              // Przypisanie nowej wartości do x.
174              x = newX;
175          // Obliczenie potencjalnie nowej wartości z.
176          float newZ = z + (deltaMoveStraight * z1);
177          // Sprawdzenie kolizji ciała na wysokości butów, pasa i głowy z potencjalnie nową wartością z.
178          if (!Collision(x, y, newZ) && !Collision(x, y + 1, newZ) && !Collision(x, y + 2, newZ))
179              // Przypisanie nowej wartości z.
180              z = newZ;
181
182      }
183      // Jeśli wykonano ruch na bok.
184      if (deltaMoveSides) {
185          // Obliczenie potencjalnie nowej wartości x.
186          float newX = x + (deltaMoveSides * -z1);
187          // Sprawdzenie kolizji ciała na wysokości butów, pasa i głowy z potencjalnie nową wartością x.
188          if (!Collision(newX, y, z) && !Collision(newX, y + 1, z) && !Collision(newX, y + 2, z))
189              // Przypisanie nowej wartości do x.
190              x = newX;
191          // Obliczenie potencjalnie nowej wartości z.
192          float newZ = z + (deltaMoveSides * x1);
193          // Sprawdzenie kolizji ciała na wysokości butów, pasa i głowy z potencjalnie nową wartością z.
194          if (!Collision(x, y, newZ) && !Collision(x, y + 1, newZ) && !Collision(x, y + 2, newZ))
195              // Przypisanie nowej wartości z.
196              z = newZ;
197      }
198  }
199
```

Obliczenie przemieszczenia. Najpierw oblicza się potencjalne przemieszczenie. Jeśli dojdzie do kolizji, ruch nie jest wykonywany. W przeciwnym wypadku ruch zostaje wykonany.

```
200  bool Player::Collision(float x, float y, float z) {
201      int x1, x2, y1, z1, z2;
202      float space = 0.4;
203      x1 = (int)(x + space);
204      x2 = (int)(x - space);
205      y1 = (int)(y);
206      z1 = (int)(z + space);
207      z2 = (int)(z - space);
208      if (map->get(x1, y1, z1) == 0 && map->get(x2, y1, z1) == 0 && map->get(x1, y1, z2) == 0 && map->get(x2, y1, z2) == 0
209          && x1 > 50 && x1<map->getX() - 49 && z1> 50 && z1 < map->getZ() - 49)
210          return false;
211      else
212          return true;
213  }
214
```

Metoda sprawdzająca kolizję. Podaje się współrzędne x z gracz oraz wysokość na której ma zostać sprawdzona kolizja. Metoda sprawdza 4 punkty na podanej wysokości. Punkty te to wierzchołki kwadratu o boku 0.8, o środku w punkcie xy. Dzięki temu, np. wywołując metodę kolizji na trzech różnych wysokościach gracza (stopy, brzuch i czubek głowy), tworzy się siatka punktów, które stanowią o tym czy dojdzie do kolizji gracza.

Klasa Textuers

```
152  void Textures::TexturesDisplay(float x, float y, float z) {
153      glEnable(GL_LIGHTING);
154      glEnable(GL_TEXTURE_2D);
155
156      GLint viewport[4];
157      GLdouble modelview[16];
158      GLdouble projection[16];
159      GLdouble windowX, windowY, windowZ;
160      glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
161      glGetDoublev(GL_PROJECTION_MATRIX, projection);
162      glGetIntegerv(GL_VIEWPORT, viewport);
163
164      int x1, y1, z1;
165      char v;
166      for (y1 = 0;y1 < map->getY();y1++) {
167          for (x1 = x - viewDistance;x1 < x + viewDistance; x1++) {
168              for (z1 = z - viewDistance;z1 < z + viewDistance;z1++) {
169
170                  gluProject(x1+0.5, y1, z1+0.5, modelview, projection, viewport, &windowX, &windowY, &windowZ);
171                  if (windowX<2400 && windowX>-500 && windowY > -700 && windowY < 1200 && windowZ < 1) {
172                      if (map->get(x1, y1, z1) > 0 && (v = map->getV(x1, y1, z1)) != 63) {
173                          glTranslatef(x1, y1, z1);
174
175                          glBindTexture(GL_TEXTURE_2D, TextureID[map->get(x1, y1, z1)]);
176
177                          if (((v >> 0) & 1UL) == 0 && y + 2.5 > y1)
178                              glDrawElements(GL_TRIANGLES, sizeof(wallTop), GL_UNSIGNED_BYTE, wallTop);
179                          if (((v >> 1) & 1UL) == 0 && y + 2.5 < y1)
180                              glDrawElements(GL_TRIANGLES, sizeof(wallBottom), GL_UNSIGNED_BYTE, wallBottom);
181                          if (((v >> 2) & 1UL) == 0 && z > z1)
182                              glDrawElements(GL_TRIANGLES, sizeof(wallZp), GL_UNSIGNED_BYTE, wallZp);
183                          if (((v >> 3) & 1UL) == 0 && z < z1)
184                              glDrawElements(GL_TRIANGLES, sizeof(wallZm), GL_UNSIGNED_BYTE, wallZm);
185                          if (((v >> 4) & 1UL) == 0 && x < x1)
186                              glDrawElements(GL_TRIANGLES, sizeof(wallXm), GL_UNSIGNED_BYTE, wallXm);
187                          if (((v >> 5) & 1UL) == 0 && x > x1)
188                              glDrawElements(GL_TRIANGLES, sizeof(wallXp), GL_UNSIGNED_BYTE, wallXp);
189
190                          glTranslatef(-x1, -y1, -z1);
191                      }
192                  }
193              }
194          }
195      }
196      for (y1 = y - 3;y1 < y + 10;y1++) {
```

Główny fragment metody renderującej świat. Włączane jest światło, Tworzone są zmienne, do których zapisane są poszczególne macierze. Potrójna pętla sprawdza wszystkie bloki w zasięgu viewDistance. Dla każdego bloku sprawdza się czy znajduje się w oknie programu (linia 171, wartości windowX/Y/Z zostały dobrane manualnie [testy pokazały, że liczba renderowanych bloków spada trzykrotnie]) oraz czy blok nie jest pokryty innymi blokami i jeśli tak, sprawdza się, które ściany nie są pokryte blokami by móc je wyrenderować.

Klasa Map

```
55   void Map::checkVisibility(int x1, int y1, int z1) {
56       char code = 0;
57       if (y1 + 1 >= y || map[x1][y1 + 1][z1] > 0)
58           code |= 1UL << 0;
59       if (y1 - 1 < 0 || map[x1][y1 - 1][z1] > 0)
60           code |= 1UL << 1;
61       if (z1 + 1 >= z || map[x1][y1][z1 + 1] > 0)
62           code |= 1UL << 2;
63       if (z1 - 1 < 0 || map[x1][y1][z1 - 1] > 0)
64           code |= 1UL << 3;
65       if (x1 - 1 < 0 || map[x1 - 1][y1][z1] > 0)
66           code |= 1UL << 4;
67       if (x1 + 1 >= z || map[x1 + 1][y1][z1] > 0)
68           code |= 1UL << 5;
69       visibilityMap[x1][y1][z1] = code;
70   }
71
72   void Map::checkFullVisibility() {
73       for (int x = 0;x < getX();x++) {
74           for (int y = 0;y < getY();y++) {
75               for (int z = 0;z < getZ();z++) {
76                   checkVisibility(x, y, z);
77               }
78           }
79       }
80   }
```

Pierwsza metoda sprawdza dla każdego bloku, czy dookoła istnieją bloki czy jest tam powietrze. Zapisuje wynik na 6 bitach zmiennej typu char. Druga metoda wywołuje pierwszą, dla wszystkich możliwych bloków. Ważne jest też to, że przy położeniu/usunięciu bloku, musi zostać wykonane dla niego checkVisibility, aby mapa był aktualna.

Klasa Ambient

W tej klasie najbardziej złożoną mechaniką jest obliczenie bloku i jego ściany na które właśnie patrzy gracz.

```
72  void Interaction::ComputeTracking(float x, float y, float z, float vector[3]) {
73      float x1 = x, y1 = y + 2.5, z1 = z;
74      float x2 = x, y2 = y + 2, z2 = z;
75      float lx1 = vector[0];
76      float ly1 = vector[1];
77      float lz1 = vector[2];
78
79      for (int i = 0;i < 450;i++) {
80          x1 += lx1 * 0.01;
81          z1 += lz1 * 0.01;
82          y1 += ly1 * 0.01;
83          x2 = (int)(x1);
84          y2 = (int)(y1);
85          z2 = (int)(z1);
86          if (y2 > y + 6 || y2 < y - 3 || x2<51 || z2<51 || x2 > map->getX() - 51 || z2 > map->getX() - 51)
87              break;
88          if (map->get(x2, y2, z2) > 0) {
89              //cout << x << "   " << y << "   " << z << endl;
90              //cout << x2 << "   " << y2 << "   " << z2 << endl;
91              followedX = x2;
92              followedY = y2;
93              followedZ = z2;
94              Wall(x1, y1, z1);
95              return;
96          }
97      }
98
99      followedX = NULL;
100     followedY = NULL;
101     followedZ = NULL;
102     followedWall = NULL;
103     //cout << " Koniec petli: " << x1 << "   " << y1 << "   " << z1 << endl;
104 }
105
```

Metoda ta sprawdza kilkaset punktów w linii prostej, w małych odstępach w kierunku kamery. Jeśli po drodze napotka na blok, wywołuje metodę Wall, aby sprawdzić na którą ścianę patrzy gracz. W przeciwnym wypadku gracz nie śledzi żadnego bloku.

```
106 void Interaction::Wall(float x1, float y1, float z1) {
107     x1 = x1 - (int)x1 - 0.5;
108     y1 = y1 - (int)y1 - 0.5;
109     z1 = z1 - (int)z1 - 0.5;
110
111     if (abs(x1) > abs(y1) && abs(x1) > abs(z1)) {
112         if (x1 >= 0)
113             followedWall = 1;
114         else if (x1 < 0)
115             followedWall = 2;
116     }
117     else if (abs(y1) > abs(x1) && abs(y1) > abs(z1)) {
118         if (y1 >= 0)
119             followedWall = 3;
120         else if (y1 < 0)
121             followedWall = 4;
122     }
123     else if (abs(z1) > abs(x1) && abs(z1) > abs(y1)) {
124         if (z1 >= 0)
125             followedWall = 5;
126         else if (z1 < 0)
127             followedWall = 6;
128     }
129
130 }
131
```

Ta metoda oblicza dla której ściany punkt jest najbliżej i wybiera ją jako śledzoną.

Klasa Camera

```
16      void Camera::Move(int x1, int y1) {
17          glutSetCursor(GLUT_CURSOR_NONE);
18          // Ustawienie kursor z powrotem na środek.
19          glutWarpPointer(glutGet(GLUT_WINDOW_WIDTH) / 2, glutGet(GLUT_WINDOW_HEIGHT) / 2);
20          // Obliczenie ruchu myszki i przypisanie do deltaAngleXZ i deltaAngleY.
21          deltaAngleXZ = (x1 - (glutGet(GLUT_WINDOW_WIDTH) / 2)) * 0.0006f;
22          deltaAngleY = (y1 - (glutGet(GLUT_WINDOW_HEIGHT) / 2)) * 0.0006f;
23
24          // Aktualizacja wektora kamery.
25          angleXZ += deltaAngleXZ * 2;
26          vector[0] = sin(angleXZ);
27          vector[2] = -cos(angleXZ);
28
29          if (deltaAngleY > 0)
30              deltaAngleY += vector[1] * vector[1] / 98;
31          else if (deltaAngleY < 0)
32              deltaAngleY -= vector[1] * vector[1] / 98;
33          vector[1] = vector[1] - deltaAngleY;
34          if (vector[1] > 7)
35              vector[1] = 7;
36          if (vector[1] < -7)
37              vector[1] = -7;
38      }
```

Ta metoda ustawia odpowiednio kamerę na podstawie położenia gracza i kierunku w którym patrzy.

Pozostałe algorytmy i metody wszystkich klas zostały opisane w dokumentacji.

Podsumowanie

Jako autorzy poświęciliśmy projektowi bardzo dużo czasu i energii, ale w zamian otrzymaliśmy olbrzymią ilość wiedzy oraz satysfakcji. W trakcie pracy, zmieniono nieznacznie założone wymagania, co jednak przyniosło pozytywne efekty. Pozwoliło to na spełnienie dodatkowych punktów, które podniosły poziom wykonania projektu. Można powiedzieć, że projekt uzyskał ostateczną wersję 1.0, choć ma on potencjał do wielu ulepszeń i w wolnych chwilach na pewno będzie rozwijany hobbystycznie.