Graphical Processing System Project
Documentation

Victor-Alexandru Padurean

January 2020

Contents

1 Subject Specification

The subject of the project consists in the photo-realistic presentation of 3D objects using OpenGL library. The user directly manipulates by mouse and keyboard inputs the scene of objects. [3]

- visualization of the scene: scaling, translation, rotation, camera movement
  - using keyboard and mouse
  - using animation
- specification of light sources (minimum two different lights)
- viewing solid, wireframe objects, polygonal and smooth surfaces
- texture mapping and materials
  - textures quality and level of detail
  - textures mapping on objects
- exemplify shadow computation
- exemplify animation of object components
- photo-realism, scene complexity, detailed modeling, algorithms development and implementation (objects generation, collision detection, shadow generation, fog, rain, wind), animation quality, different types of light sources (global, local, spotlights)

2 Scenario

2.1 Scene and Objects Description

The scene represents a large forest with two buffaloes in it. An ufo hover over the forest. The forest contains trees that grow quite randomly, but that are not too crowded, like real ones. The two buffalos stand one facing another, and they stand near the origin of the scene. There is a large, red, floating cube which represent a checkpoint and the spaceship can interact with it. If they intersect, a scene presentation animation will commence. The window has incorporated another perspective, not only the third person view of the ufo. It also shows what the spaceship has beneath it.

The 3D models will be presented below.

1. Trees
   There are two types of trees. The first one is smaller, more branched, with a broader choronal and more pale leaves, presented in Figure 1.

Figure 1: First Tree Type

The other is bolder, with a grander stature and greener leaves, presented in Figure 2.

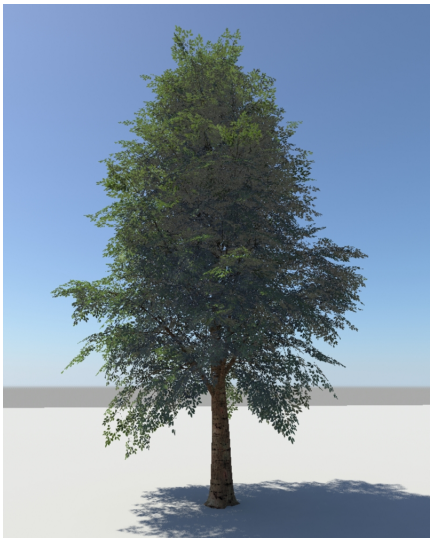

Figure 2: Second Tree Type

2. Buffaloes
   The two buffaloes are the only inhabitants of the forest and they stay near each other, scared of the ufo. They don't do much else, only marking the origin of the scene. The appearance of a buffalo is presented in Figure 3.

Figure 3: Bison

3. Ufo
   The ufo is the main protagonist of the animation, as the user controls it. It constantly rotates around its y axis, generating its thrust force. It has green, point light source inside it, which is visible at night. The appearance of the ufo is presented in Figure 4.

Figure 4: Ufo

4. Ground
   The ground is an object necessary for all the trees and buffaloes to stand. It is also necessary for shadow casting.

5. Light Cube
   The light cube stands over the forest, with a red glow (it is rendered using a different shader).

All the models come from [3] and [1].

2.2 Functionalities

There are various functionalities present, giving the user a better, more convincing experience.

- The user can control the camera, the ufo always remaining in the front of the camera, like a third person view.
- The collision with the red cube will start an automatic presentation of the scene.

- The intensity for light source of the ufo is controlled using the keyboard.
- The scene is rendered dynamically, depending on the position of the camera.
- The directional light source revolves, giving the impression of a day-night cycle.
- Second camera perspective. There is another frame, which shows the scene from the ufo looking down on the ground (bombing view perspective).

3 Implementation Details

3.1 Special Functions and Algorithms

3.1.1 Environment Generation

Solution The ground was generated by placing more ground object one after another. Their number depends on two global parameters. The trees were generated using a special function, presented in Listing 1. It first computes the number of needed trees given the size of the environment, then it randomly generates attributes for each tree: type, position on x and z coordinates, size and, rotation.

```
1  void genTrees() {
2
3    treePosx.clear();
4    treePosz.clear();
5    treeType.clear();
6    treeSize.clear();
7    treeRot.clear();
8
9    for (int i = 0; i < trees; i++) {
10     for (int j = 0; j < trees; j++) {
11       int x = rand() % 100 + 50;
12       float size = x / 100.0;
13       int type = rand() % 2;
14       int posx = rand() % 5;
15       int posz = rand() % 5;
16       int rot = rand() % 628;
17       float rotation = rot / 100;
18
19       posx = -(10 + 20 * (environmentSizex/2)) + i * ((20 *
     environmentSizex) / trees) + posx;
20       posz = -(10 + 20 * (environmentSizez/2)) + j * ((20 *
     environmentSizez) / trees) + posz;
21
22       treePosx.push_back(posx);
23       treePosz.push_back(posz);
24       treeType.push_back(type);
25       treeSize.push_back(size);
26       treeRot.push_back(rotation);
```

```
    }
  }
}
```

Listing 1: Method generating trees

Motivation The solution was chosen because it generates the environment only once, at the beginning, then it is held in the memory.

3.1.2 Environment Rendering

Possibilities There are two possible ways of discarding unneeded fragments: inside the GPU or inside the CPU.

Solution Because the environment is complex, the rendering must be done carefully, with only what is needed or the animation will run to slowly. As the numbers of ground objects and tree objects are the highest, these models must be sent to the GPU and rendered only if necessary. Figure 2 and Figure 3 present the way of selecting which objects will be rendered.

```
for (int i = 0; i < environmentSizex; i++) {
    for (int j = 0; j < environmentSizez; j++) {

      model = glm::mat4(1.0f);

      model = glm::translate(model, glm::vec3((-(10 + 20 * (
    environmentSizex / 2)) + 20 * i), 0, (-(10 + 20 * (
    environmentSizez / 2)) + 20 * j)));

      view = myCamera.getViewMatrix();

      glm::vec4 pos = view * model*glm::vec4(0, 0, 0, 1);

      pos.x = pos.x / pos.w;
      pos.z = pos.z / pos.w;

      if (pos.x < -40 || pos.z < -80 || pos.x > 40 || pos.z > 20)
        continue;

      glUniformMatrix4fv(glGetUniformLocation(shader.shaderProgram,
     "model"), 1, GL_FALSE, glm::value_ptr(model));

      ground.Draw(shader);
    }
  }
```

Listing 2: Method selecting which ground objects to render and rendering them

```
void renderTrees(gps::Shader shader, bool down)
{

```

```
4   //initialize the view matrix
5   view = myCamera.getViewMatrix();
6   //send view matrix data to shader
7   glUniformMatrix4fv(glGetUniformLocation(shader.shaderProgram, "
      view"), 1, GL_FALSE, glm::value_ptr(view));
8
9   //create normal matrix
10  normalMatrix = glm::mat3(glm::inverseTranspose(view*model));
11  //send normal matrix data to shader
12  glUniformMatrix3fv(glGetUniformLocation(shader.shaderProgram, "
      normalMatrix"), 1, GL_FALSE, glm::value_ptr(normalMatrix));
13
14  for (int i = 0; i < treePosx.size(); i++) {
15
16    model = glm::mat4(1.0f);
17
18    model = glm::translate(model, glm::vec3(treePosx[i], 0,
      treePosz[i]));
19    model = glm::rotate(model, glm::radians(treeRot[i]), glm::vec3
      (0, 1, 0));
20    model = glm::scale(model, glm::vec3(treeSize[i]));
21
22    view = myCamera.getViewMatrix();
23
24    glm::vec4 pos = view*model*glm::vec4(0,0,0,1);
25
26    pos.x = pos.x / pos.w;
27    pos.z = pos.z / pos.w;
28
29    if (pos.x < -20 || pos.z < -40 || pos.x > 20 || pos.z > 2)
30      continue;
31
32    glUniformMatrix4fv(glGetUniformLocation(shader.shaderProgram, "
      model") , 1, GL_FALSE, glm::value_ptr(model));
33
34    if (treeType[i] == 0)
35      tree.Draw(shader);
36    else
37      tree2.Draw(shader);
38  }
```

Listing 3: Method selecting which tree objects to render and rendering them

Motivation If the objects which are not needed are discarded as a whole, inside the CPU, there will be less for the GPU to process and the animation will run smoother. Discarding objects that are too far, with respect to x and z axis, is the most straight forward way.

3.1.3 Collision Detection

Alternatives There are two alternatives regarding collision detection using bounding boxes: bounding boxes with edges parallel to the coordinates axis and bounding boxes with edges non parallel with the axis.

Solution One may check if a given point is inside a random given box using the algorithm present in Listing 4. It uses dot products to check whether the point is between the necessary points. Help came from [2]

```
bool checkIfPointInsideBox(glm::vec3 point, glm::vec3 mins, glm::
    vec3 maxs)
{
  glm::vec3 a = glm::vec3(mins.x, mins.y, maxs.z);
  glm::vec3 b = glm::vec3(maxs.x, mins.y, maxs.z);
  glm::vec3 c = glm::vec3(maxs.x, maxs.y, maxs.z);
  glm::vec3 d = glm::vec3(mins.x, maxs.y, maxs.z);
  glm::vec3 e = glm::vec3(mins.x, mins.y, mins.z);
  glm::vec3 f = glm::vec3(maxs.x, mins.y, mins.z);
  glm::vec3 g = glm::vec3(maxs.x, maxs.y, mins.z);
  glm::vec3 h = glm::vec3(mins.x, maxs.y, mins.z);
  glm::vec3 u = a - e;
  glm::vec3 v = a - b;
  glm::vec3 w = a - d;
  bool ok5 = (glm::dot(u, e) < glm::dot(u, point)) && (glm::dot(u,
    point) < glm::dot(u, a));
  bool ok6 = (glm::dot(v, b) < glm::dot(v, point)) && (glm::dot(v,
    point) < glm::dot(v, a));
  bool ok7 = (glm::dot(w, d) < glm::dot(w, point)) && (glm::dot(w,
    point) < glm::dot(w, a));
  if (ok5 && ok6 && ok7)
    return true;
  return false;
}
```

Listing 4: Method that check if a point is inside a box

Then, using the function above, one can check if two bounding boxes intersect by checking if one of their vertices, or its center, is inside the other, as in Listing 5.

```
bool checkCollisions(glm::vec3 mins1, glm::vec3 maxs1, glm::vec3
    mins2, glm::vec3 maxs2, glm::mat4 model1, glm::mat4 model2)
{
  mins1 = glm::vec3(model1 * glm::vec4(mins1, 1.0));
  maxs1 = glm::vec3(model1 * glm::vec4(maxs1, 1.0));
  mins2 = glm::vec3(model2 * glm::vec4(mins2, 1.0));
  maxs2 = glm::vec3(model2 * glm::vec4(maxs2, 1.0));
  glm::vec3 mins = mins1;
  glm::vec3 maxs = maxs1;
  glm::vec3 a1 = glm::vec3(mins.x, mins.y, maxs.z);
  if (checkIfPointInsideBox(a1, mins2, maxs2))
    return true;
  glm::vec3 b1 = glm::vec3(maxs.x, mins.y, maxs.z);
  if (checkIfPointInsideBox(b1, mins2, maxs2))
    return true;
  glm::vec3 c1 = glm::vec3(maxs.x, maxs.y, maxs.z);
  if (checkIfPointInsideBox(c1, mins2, maxs2))
    return true;
  glm::vec3 d1 = glm::vec3(mins.x, maxs.y, maxs.z);
  if (checkIfPointInsideBox(d1, mins2, maxs2))
```

```
20      return true;
21    glm::vec3 e1 = glm::vec3(mins.x, mins.y, mins.z);
22    if (checkIfPointInsideBox(e1, mins2, maxs2))
23      return true;
24    glm::vec3 f1 = glm::vec3(maxs.x, mins.y, mins.z);
25    if (checkIfPointInsideBox(f1, mins2, maxs2))
26      return true;
27    glm::vec3 g1 = glm::vec3(maxs.x, maxs.y, mins.z);
28    if (checkIfPointInsideBox(g1, mins2, maxs2))
29      return true;
30    glm::vec3 h1 = glm::vec3(mins.x, maxs.y, mins.z);
31    if (checkIfPointInsideBox(h1, mins2, maxs2))
32      return true;
33    glm::vec3 i1 = glm::vec3((mins.x + maxs.x) / 2, (mins.y + maxs.y)
        / 2, (mins.z + maxs.z) / 2);
34    if (checkIfPointInsideBox(i1, mins2, maxs2))
35      return true;
36    mins = mins2;
37    maxs = maxs2;
38    glm::vec3 a2 = glm::vec3(mins.x, mins.y, maxs.z);
39    if (checkIfPointInsideBox(a2, mins1, maxs1))
40      return true;
41    glm::vec3 b2 = glm::vec3(maxs.x, mins.y, maxs.z);
42    if (checkIfPointInsideBox(b2, mins1, maxs1))
43      return true;
44    glm::vec3 c2 = glm::vec3(maxs.x, maxs.y, maxs.z);
45    if (checkIfPointInsideBox(c2, mins1, maxs1))
46      return true;
47    glm::vec3 d2 = glm::vec3(mins.x, maxs.y, maxs.z);
48    if (checkIfPointInsideBox(d2, mins1, maxs1))
49      return true;
50    glm::vec3 e2 = glm::vec3(mins.x, mins.y, mins.z);
51    if (checkIfPointInsideBox(e2, mins1, maxs1))
52      return true;
53    glm::vec3 f2 = glm::vec3(maxs.x, mins.y, mins.z);
54    if (checkIfPointInsideBox(f2, mins1, maxs1))
55      return true;
56    glm::vec3 g2 = glm::vec3(maxs.x, maxs.y, mins.z);
57    if (checkIfPointInsideBox(g2, mins1, maxs1))
58      return true;
59    glm::vec3 h2 = glm::vec3(mins.x, maxs.y, mins.z);
60    if (checkIfPointInsideBox(h2, mins1, maxs1))
61      return true;
62    glm::vec3 i2 = glm::vec3((mins.x + maxs.x) / 2, (mins.y + maxs.y)
        / 2, (mins.z + maxs.z) / 2);
63    if (checkIfPointInsideBox(i2, mins1, maxs1))
64      return true;
65    return false;
66  }
```

Listing 5: Method that checks if two bounding boxes intersect

Motivation A general approach is the best, yet by keeping only the minimums and maximums of the bounding edges, only axis parallel bounding boxes can be generated. The algorithm works for this project's needs as the objects are symmetrical.

3.1.4 Light Sources

Possibilities There are three main algorithms: Gourarnd, Phong and Blin-Phong. The first is computing the light influence using the vertex normal, whereas the second uses the fragment normal. So, the first does the computation in the vertex shader, whereas the second, in the fragment shader. The last algorithm resembles the second, yet the reflection is computed using half vector, not the reflection technique. This is described in [3].

Solution and Motivation The Blinn-Phong model was used as it looks more real than the Gourard algorithm, even though the latter is faster. In addition, it is slightly faster than the Phong model. It was used for both directional and point light sources.

3.1.5 Shadow Computation

Solution The shadow computation was done using shadow mapping. The scene is rendered twice. It first renders the scene from the light source's point of view, not onto the screen, but into a texture, keeping track of the closest object to the light source. That texture will then be used in the second render, to compute which object is closer to the light source. [3]

Motivation This is the solution that can be understood with much ease.

3.1.6 Camera Animation

Possibilities The camera movement can be done linearly, or by using Bezier Curves. The first approach can be done by interpolating the camera's position and target with respect to time. The second uses the deCasteljau algorithm.

Solution Both algorithms were implemented, but the one using Bezier Curves is used, as it runs more smoothly. Its implementation is presented in Listing 6. Inspiration came from [4].

```
1  std::vector<glm::vec3> deCasteljau(std::vector<glm::vec3> points,
       int degree, float t) {
2      float *pointsQ = new float[(degree + 1) * 3];
3      float *pointsR = new float[(degree + 1) * 3];
4      int Qwidth = 3;
5      for (int j = 0; j <= degree; j++) {
6        pointsQ[j*Qwidth + 0] = points[j].x;
7        pointsQ[j*Qwidth + 1] = points[j].y;
8        pointsQ[j*Qwidth + 2] = points[j].z;
9        pointsR[j*Qwidth + 0] = points[j].x;
10       pointsR[j*Qwidth + 1] = points[j].y;
11       pointsR[j*Qwidth + 2] = points[j].z;
12     }
13     for (int k = 1; k <= degree; k++) {
14       for (int j = 0; j <= degree - k; j++) {
```

```
15          pointsQ[j*Qwidth + 0] = (1 - t) * pointsQ[j*Qwidth + 0] + t
    * pointsQ[(j + 1)*Qwidth + 0];
16          pointsQ[j*Qwidth + 1] = (1 - t) * pointsQ[j*Qwidth + 1] + t
    * pointsQ[(j + 1)*Qwidth + 1];
17          pointsQ[j*Qwidth + 2] = (1 - t) * pointsQ[j*Qwidth + 2] + t
    * pointsQ[(j + 1)*Qwidth + 2];
18          pointsR[j*Qwidth + 0] = (1 - (t + 0.1)) * pointsR[j*Qwidth
  + 0] + (t + 0.1) * pointsR[(j + 1)*Qwidth + 0];
19          pointsR[j*Qwidth + 1] = (1 - (t + 0.1)) * pointsR[j*Qwidth
  + 1] + (t + 0.1) * pointsR[(j + 1)*Qwidth + 1];
20          pointsR[j*Qwidth + 2] = (1 - (t + 0.1)) * pointsR[j*Qwidth
  + 2] + (t + 0.1) * pointsR[(j + 1)*Qwidth + 2];
21        }
22    }
23    std::vector<glm::vec3> result;
24    glm::vec3 resultPos;
25    glm::vec3 resultTarg;
26    resultPos.x = pointsQ[0];
27    resultPos.y = pointsQ[1];
28    resultPos.z = pointsQ[2];
29    resultTarg.x = pointsR[0];
30    resultTarg.y = pointsR[1];
31    resultTarg.z = pointsR[2];
32    delete[] pointsQ;
33    delete[] pointsR;
34    result.push_back(resultPos);
35    result.push_back(resultTarg);
36    return result;
37  }
38
39  glm::vec3 Camera::interpolateBezier(std::vector<glm::vec3> points
    , double elapsedTime, double totalTime) {
40    float t = elapsedTime / totalTime;
41    std::vector<glm::vec3> result = deCasteljau(points, points.size
    () - 1, t);
42    cameraPosition = result[0];
43    cameraTarget = result[1];
44    cameraDirection = glm::normalize(cameraTarget - cameraPosition)
    ;
45    cameraRightDirection = glm::normalize(glm::cross(
    cameraDirection, glm::vec3(0, 1, 0)));
46    return cameraPosition;
47  }
```

Listing 6: Methods that computes the location of the camera on the Bezier curve

3.2 Graphics Model

OpenGL is considered an Application Programming Interface (API) and is used to develop graphical applications by accessing features available in the graphics hardware.However, OpenGL is a specificationdeveloped and maintained by the Khronos Group. The OpenGL specification describes what is the desired resultor output of each function. The actual implementation can be

different between various OpenGL libraries. These libraries are implemented mainly by the graphics card manufacturers. [3]

It has a client-server architecture: the client is the programmer's application, and the server is the OpenGL implementation present in the computer graphics hardware.

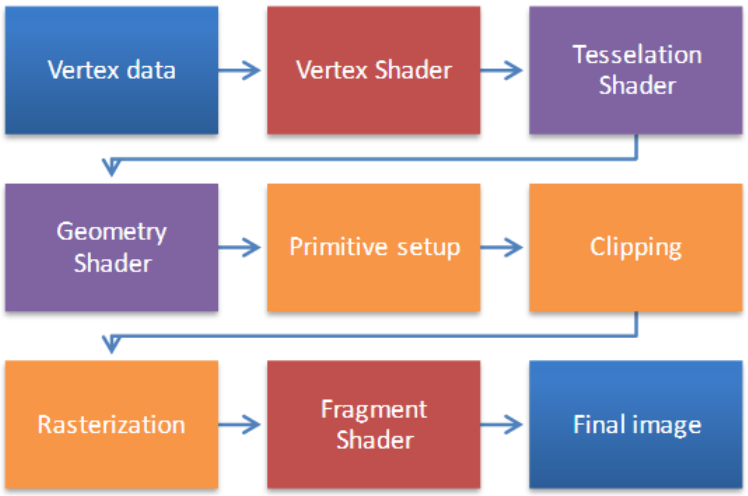

Figure 5: OpenGL Pipeline

3.3 Data Structures

The data structures are either classes or C structures.

1. Classes
   (a) Model3D - is used for reading the object from a file, reading the related textures and drawing the object using a shader; it holds information about an object.
   (b) Mesh - holds information about a mesh (polygon) and the structures that make it up; it has methods for setting up and drawing meshes.
   (c) Camera - keeps camera position, direction and target; it is responsible for moving the camera and rotating it; it also has methods for moving the camera for presenting the scene.
   (d) Shader - holds information about the shader, used for reading, compiling and linking the shaders.
   (e) SkyBox - used for reading and drawing the skybox
2. Structures
   (a) Vertex - holds position, normal and texture coordinates

(b) Texture - holds the id, the type (ambient, diffuse or specular) and the path

(c) Material - has three vectors describing how it behaves with ambient, diffuse and specular light

3.4 Class Hierarchy

The number of classes is reduced, as the majority of the application logic is done inside the application class.

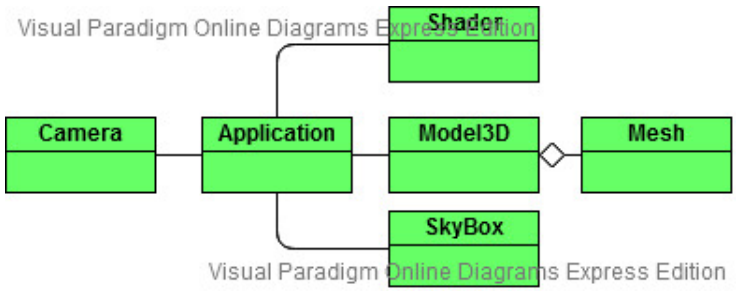

Figure 6: Class Diagram

4 User Manual

The user has as inputs the keyboard and the mouse. They can use the mouse to change the camera view direction and the W, A, S, D, Space and Ctrl keys for camera movement, the classical, game like options. They can control the intensity of the ufo's point light with the Q and E keys. K and L keys are used to change between wireframe and smooth surface representations.

5 Conclusions and Further Developments

The project was a good way of learning and understanding OpenGL (as it can be seen on the results of the lab quizzes too). The mathematics behind Bezier curves and collision detection, as well as shadow casting (which was, by far, the most difficult part) and lighting, are of good use. The camera controls implementation will also be useful in the future.

There are several ways of making the project more interesting, such as:

- Having more buffaloes or other types of animals and animating them.
- Providing object collision for the ufo with all the other object.
- Add an animation in which the ufo kidnaps some buffaloes.

- Add other types of trees.
- Add more diverse soil.

References

[1] Free3d.

[2] Stack exchange, mathematics, point inside 3d box.

[3] Graphical processing systems, moodle, computer science, utc-n, 2019.

[4] yuany90. Beziercurve.