Курсов проект по Компютърна графика

STL ФАЙЛОВ ВИЗУАЛИЗАТОР

ALEXANDER NAUMOV

Съдържание

1. Въведение

1.1. Описание на разработения софтуер

Целта на този проект бе да бъде разработен софтуер, с помощта на който да могат да бъдат визуализирани файлове, съдържащи описанието на геометрични модели създадени посредством методите на компютърната графика, в частност файловете с разширение „STL”. Приложението съчетава основните функционалности, нужни за анализа и визуализацията на 3D модели, като предлага цялостна среда за преглед и оценка на STL файлове с различни допълнителни функционалности като:

- Преглед на обекта в различни цветове в нормален режим – плътен обект – симулира преглеждането в естествена среда
- Преглед на обекта в полупрозрачен режим – помага за оценката на скрити детайли
- Преглед STL мрежата (Mesh) на обекта – помага за оценката на качеството и точността на изготвения STL файл
- Възможности за разглеждане на обекта чрез контролите на компютърната мишка – приближаване, отдалечаване и обикаляне

STL файловете са широкопопулярни във всякакъв вид графичен дизайн с компютърно подпомагане (Computer Aided Design – CAD) и особено в случаите, в които тези модели се използват с цел производство – тези файлове са един от стандартите при 3D принтирането или по – общо – адитивното производство. В този файл обектите се описват чрез взаимносвързани триъгълници или още наречени фасети като всеки триъгълник се дефинира чрез нормален вектор и трите върха на триъгълника (фасета). Важно е да се отбележи, че координатите на всеки от тези параметри са в триизмерното пространство и съответно имат стойности за X, Y и Z координатите. Съществуват две форми на запис на тези файлове – бинарен и ASCII като бинарният е по – широко използван, тъй като е по – компактен и съответно бърз за зареждане.

1.2. Използвани технологии

За целите на проекта са избрани OpenGL, GLFW и GLAD като три основни компонента, които заедно осигуряват пълната инфраструктура за визуализация, работа с прозорци и комуникация с графичния процесор.

OpenGL (Open Graphics Library) е приложно програмен интерфейс (API) от ниско ниво, който осигурява директен достъп до функционалността на видеокартата. Той е избран, тъй като е основен предмет на обучението в курса „Компютърна графика“, но и по множество други причини. OpenGL е надежден, преносим и високопроизводителен, понеже е платформено независим, съществува от над 30 години и е използван в най – различни професионални приложения, поддържан е от всички основни производители, използва графичния процесор директно и е подходящ в ситуации, в който се изисква rendering с десетки или стотици кадри в секунда.

За да може OpenGL софтуерът да се изпълнява в някакъв контекст като прозорец се използва GLFW – библиотека, с която могат да се създават и управляват прозорци с OpenGL контекст и позволява обработката на входни сигнали от устройства като клавиатура, мишка, джойстик и др.

Glad (GL Loader Generator) loader библиотека – той зарежда адресите на OpenGL функциите от драйвера. Тоест, когато бъде извикана например „glClearColor“ – функция от OpenGL API-то, Glad е заредил адреса на функцията от драйвера и извикването всъщност използва указател към тази функция, който сочи към реалната ѝ имплементация в драйвера. Тъй като OpenGL е стандарт, а реалните функции се предоставят от драйвера на всяка конкретна система, те трябва да бъдат заредени динамично по време на изпълнение – за това се грижи Glad.

1.3. Основни компоненти

Разработеният софтуер разполага с множество различни компоненти, центрирани около основните дейности. Това са например модул за четене на STL файла, модули, свързани с обобщаването на информацията в него и изпращането ѝ към графичния процесор, модули свързани с графичния

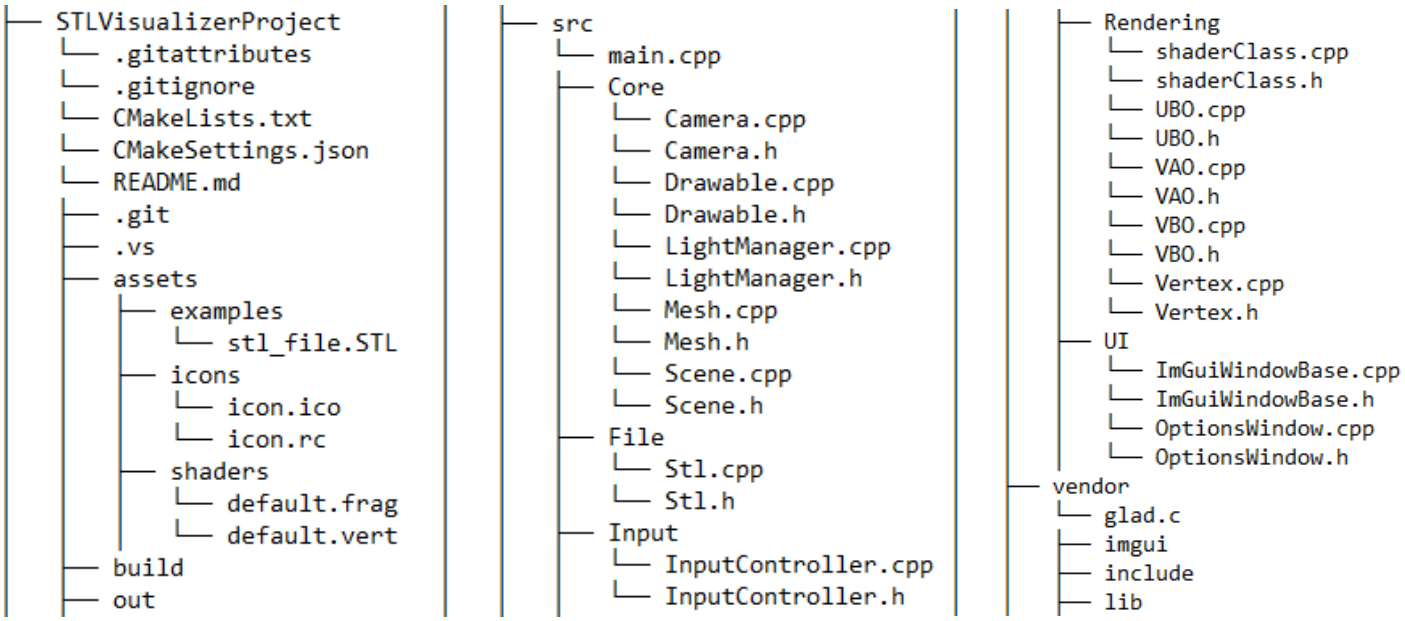
интерфейс, както и останалите участници в сцената като осветлението и камерата. Всеки от тях ще бъде разгледан по – подробно в останалите глави.

2. Архитектура

Архитектурата на проекта е изградена върху принципа на ясно разграничени отговорности между класовете, с цел постигане на модулност, предвидимост и лесна поддръжка. Всеки клас е проектиран да управлява само своята функционална област, без да поема логика, която принадлежи на други компоненти - например входът, осветлението, rendering и потребителският интерфейс са ясно разграничени в отделни модули.

2.1. Файлова структура

За да бъде придобита представа за размера и структурата на проекта е добре първо да бъде разгледана файловата му структура. Тя е разделена по този начин, за да може да центрира прилежащите файлове към проекта в директории, подсказващи тяхната цел (фигура 2.1).



Фиг. 2.1 – Файлова структура на проекта

Основните директории са:

- **/assets** – в тази папка се помещават ресурсите, които са необходими на програмата по време на нейното изпълнение – това са най – вече

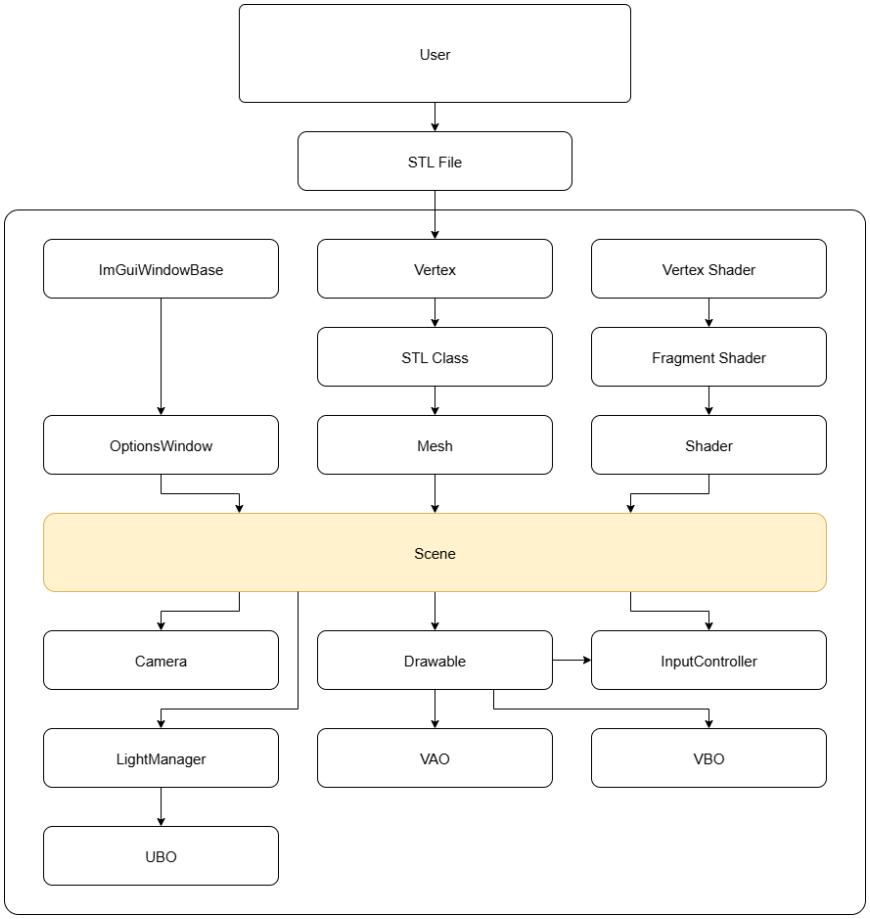
шейдърите (shaders) – програми, които се изпълняват върху графичния процесор в различни моменти от конвейера за обработка на изображението (rendering pipeline). Също така има и примерен STL файл, ако по някаква причина не бъде избран друг за зареждане.

- **/build** – това е папката, в която се съдържа готовият компилиран машинен код под формата на .exe, както и множество други файлове, позволяващи този процес да се случи с помощта на CMake.
- **/src** – тук се съдържа целият код на програмата, тук е реализирана нейната функционалност. Има множество подпапки като например:
  - **/Core** – съдържа основните класове, които дефинират логиката на приложението на високо ниво. Те са предназначени да бъдат използвани директно от разработчика в основната програма (main), като предоставят интуитивни и абстрактни интерфейси за създаване и управление на ключови компоненти като осветление, mesh обекти, камери и други. Целта на тези класове е да обобщят сложната вътрешна логика и да я представят чрез лесни за използване методи, така че разработчикът да може да работи с тях без да навлиза в детайлите на тяхната реализация. Това позволява по-чист и модулен код, в който всяка функционалност е ясно капсулирана.
  - **/File** – съдържа файловете, отговорни за консумацията на данни от файловете, които ще бъдат прочитани и изобразявани на екрана. На този етап това е единствено STL форматът, но е отделен в отделна директория с цел при бъдещо развитие да има отделено място за код с тази цел.
  - **/Input** – съдържа класа, отговорен за менажирането на всички входни сигнали от потребителя като натискания на бутони и влачения на компютърната мишка.
  - **/Rendering** – в тази директория са развити функционалностите от по – ниско ниво, чрез които се осъществяват операциите към видеокартата. Това са класове за различни буфери като VBO (Vertex Buffer Object) – буфер в паметта на графичния процесор, който съдържа данни за върховете на обекта, или пък VAO (Vertex Array Object), който описва как

да се използват тези VBO-та, shaderClass, който помага за използването на shader-ите за създаването на програма чрез glCreateProgram и други.
    - **/UI** – тук са развити класовете, които помагат за създаването на прозорци за целите на графичния потребителски интерфейс с помощта на библиотеката ImGui.
- **/vendor** – тук се съдържат всички външни библиотеки за целите на статичното връзване с разработената програма - локално включени библиотеки, които се компилират заедно с проекта

2.2. Зависимости между модулите на системата

На фигура 2.2 е представена схема, която цели да изобрази схематично зависимостите между отделните модули.



Фиг. 2.2 – Схема на зависимостите между модулите на системата

Ясно се очертава основният и най – главен модул на системата – “Scene”. Това е класът, който притежава състоянието на цялата „сцена“, която се изобразява на екрана. Този клас (освен инициализационните променливи) е единственото, което трябва на един разработчик, за да създаде сцена и успешно да я изобразява на екрана. Това е направено с цел да се създаде максимално ниво на абстракция от функционалностите на ниско ниво.

За да се създаде един “Scene” клас ни трябват няколко неща – “Mesh”, “Window” и “Shader” инстанции. Тези три инстанции, с които се инициализира “Scene”, от неговата гледната точка е аналогично на: „Какво искаш да покажа, къде да го покажа и как да го покажа?“. “Mesh” класа се създава като продукт от извлечените данни от STL файла (подаден от потребителя), които пък се съдържат в клас „Stl”, който предлага интерфейс за получаване на данните за всички върхове под формата на вектор от класа “Vertex”. “Shader” инстанцията се създава от двата съществуващи по подразбиране шейдъра.

По желание може да се подаде и инстанция на класа “Camera”, но ако не бъде подадена ще бъде създадена такава по подразбиране. Освен това след създаване на така наречената сцена, има възможност за добавяне на един или повече прозорци за потребителски интерфейс, които са базирани на класа “ImGuiWindowBase”. В случая съществува само един такъв – “OptionsWindow”.

За да може да работи успешно, използвайки подадените данни, класът “Scene” създава инстанции на класовете “Drawable”, “InputController” и “LightManager”, а пък те – свои инстанции, с които извършват своята работа.

3. Реализация

За максимално лесно проследяване на функционалността на разработената програма, описанията в тази глава ще се движат според представената на фиг. 2.2 диаграма.

В текущото си състояние програма може да стартира всеки един STL файл като с десен бутон на мишката се кликне върху него и се избере „Open with" менюто, а вътре в него – изпълнимият файл на програмата. По този начин името на файла се подава директно като аргументи на програмата. В началото на изпълнение на програмата има инициализационна процедура, развита във функцията "initApplication". В нея се инициализират библиотеките, които са използвани и се извикват техни стартови или конфигурационни функции за настройка на поведението. Следва изпълнението на функцията "getExecutableDir", чрез която се взима директорията, в която се намира изпълнимият файл и се използва като базова директория за достъпване на всички asset-и. След това може да се пристъпи към зареждане на STL файла, избран от потребителя. Реализацията на тази част от програмата е представена на фигура 3.1.

```
20      GLFWwindow* window;
21      ImGuiIO io;
22
23
24      void initApplication();
25      std::string getExecutableDir();
26
27      int main(int argc, char* argv[])
28      {
29          initApplication();
30
31          std::string baseDir = getExecutableDir();
32
33          std::string vertexShaderPath = baseDir + "/assets/shaders/default.vert";
34          std::string fragmentShaderPath = baseDir + "/assets/shaders/default.frag";
35          std::string exampleSTLPath = baseDir + "/assets/examples/stl_file.STL";
36          std::string iconPath = baseDir + "/assets/icons/icon.ico";
37
38
39          StlFile stl = StlFile((argc > 1) ? argv[1] : exampleSTLPath);
40          std::vector<Vertex> verticesVector = stl.verticesConvertVertexArray();
```

Фиг. 3.1 – Инициализация и зареждане на STL

Следва създаването на необходимите за инициализацията на “Scene” променливи. От заредените за STL файла данни се създава “Mesh”, а от shader-ите в папка assets се създава “Shader” (фиг. 3.2). Създаването на този клас създава два шейдъра посредством “glCreateShader”, добавя и компилира код за този шейдър чрез “glShaderSource” и “glCompileShader”, създава програма чрез “glCreateProgram” и обвързва с нея двата новосъздадени и компилирани шейдъра чрез “glAttachShader” и линква програмата. След всички тези операции тя е готова за изпълнение, което ще се случи в момента, в който някой извиква “glUseProgram”, с което ще се избере текущата шейдър програма за rendering.

```
18      // Constructor that build the Shader Program from 2 different shaders
19      Shader::Shader(const char* vertexFile, const char* fragmentFile)
20      {
21          // Read vertexFile and fragmentFile and store the strings
22          std::string vertexCode = get_file_contents(vertexFile);
23          std::string fragmentCode = get_file_contents(fragmentFile);
24
25          // Convert the shader source strings into character arrays
26          const char* vertexSource = vertexCode.c_str();
27          const char* fragmentSource = fragmentCode.c_str();
28
29          // Create Vertex Shader Object and get its reference
30          GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
31          // Attach Vertex Shader source to the Vertex Shader Object
32          glShaderSource(vertexShader, 1, &vertexSource, NULL);
33          // Compile the Vertex Shader into machine code
34          glCompileShader(vertexShader);
35          compileErrors(vertexShader, "VERTEX");
36
37          // Create Fragment Shader Object and get its reference
38          GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
39          // Attach Fragment Shader source to the Fragment Shader Object
40          glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
41          // Compile the Vertex Shader into machine code
42          glCompileShader(fragmentShader);
43          compileErrors(fragmentShader, "FRAGMENT");
44
45          // Create Shader Program Object and get its reference
46          this->ID = glCreateProgram();
47          // Attach the Vertex and Fragment Shaders to the Shader Program
48          glAttachShader(this->ID, vertexShader);
49          glAttachShader(this->ID, fragmentShader);
50          // Wrap-up/Link all the shaders together into the Shader Program
51          glLinkProgram(this->ID);
52          compileErrors(this->ID, "PROGRAM");
53
54          // Delete the now useless Vertex and Fragment Shader objects
55          glDeleteShader(vertexShader);
56          glDeleteShader(fragmentShader);
57      }
```

Фиг. 3.2 – Създаване на обекта Shader

След създаването на “Scene” добавяме и всички желани прозорци за потребителски интерфейс чрез метода “addGuiWindow” и активираме сцената, което всъщност извиква „activate” функцията на Shader-а и избира текущата шейдър програма за rendering. “Scene” прави цялата работа на заден план и благодарение на това main функцията изглежда по този начин (фиг. 3.3):

```
27  int main(int argc, char* argv[])
28  {
29      initApplication();
30
31      std::string baseDir = getExecutableDir();
32
33      std::string vertexShaderPath = baseDir + "/assets/shaders/default.vert";
34      std::string fragmentShaderPath = baseDir + "/assets/shaders/default.frag";
35      std::string exampleSTLPath = baseDir + "/assets/examples/stl_file.STL";
36      std::string iconPath = baseDir + "/assets/icons/icon.ico";
37
38
39      StlFile stl = StlFile((argc > 1) ? argv[1] : exampleSTLPath);
40      std::vector<Vertex> verticesVector = stl.verticesConvertVertexArray();
41
42      Mesh objectMesh(verticesVector);
43      Shader shaderProgram(vertexShaderPath.c_str(), fragmentShaderPath.c_str());
44
45      // What will be drawn, where and how
46      Scene scene = Scene(objectMesh, window, shaderProgram);
47
48      OptionsWindow optionsWindow("Options Chooser", glGetUniformLocation(shaderProgram.ID, "objectColor"), 300, 400, 10.0f, 10.0f);
49      // Let the scene handle the gui window
50      scene.addGuiWindow(&optionsWindow);
51      scene.activate();
52
53
54      // WHILE LOOP
55      while (!glfwWindowShouldClose(window))
56      {
57          glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
58
59          // Clean the back buffer and assign the new color to it
60          glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
61
62
63          if (!io.WantCaptureMouse) {
64              scene.handleInput();
65          }
66
67          scene.draw();
68
69          // Swap the back buffer with the front buffer
70          glfwSwapBuffers(window);
71          // Take care of all GLFW events
72          glfwPollEvents();
73      }
```

Фиг. 3.3 – Main функцията

Както вече беше споменато, “Scene” създава и още допълнителни инстанции на класове, които му помагат да изпълни задълженията си - “Drawable” “InputController” и “LightManager”.

“Drawable” съдържа “Mesh”-а, който описва какво ще се „рисува“, но и “VAO” и „VBO” променливи, с които комуникира всички операции по „рисувания“ обект с графичния процесор. Именно затова тук също се пазят

“rotationQuaternion”, “rotationMatrix” и “scalingMatrix” – посредством тях се извършват всички трансформации върху обекта при подаване на команда за такива от потребителя. Два от най – важните методи на този клас са “draw” и “applyRotation”. “draw” е методът, който се извиква от “Scene” класа на всеки frame, за да „изрисува“ своя “Drawable”. Методът започва с активиране на шейдъра и свързване на съответния VAO, който съдържа конфигурацията на върховете. Обектът се центрира спрямо своя центроид, за да се извърши ротация около центъра, прилага се ротационна трансформация чрез “rotationMatrix”, а след това обектът се връща обратно в оригиналната си позиция. Накрая се прилага мащабиране чрез “scalingMatrix”.

```
19      void Drawable::draw(Shader& shader, Camera& camera) {
20          shader.activate();
21          this->VAO.bind();
22
23          glm::mat4 model = glm::mat4(1.0f);
24
25          // Translate to origin (center the object)
26          model = glm::translate(model, -this->mesh.centroid);
27
28          // Rotate
29          model *= this->rotationMatrix;
30
31          // Translate back to original position
32          model = glm::translate(model, this->mesh.centroid);
33
34          model *= this->scalingMatrix;
35
36          glUniformMatrix4fv(glGetUniformLocation(shader.ID, "model"), 1, GL_FALSE, glm::value_ptr(model));
37
38          glm::vec3 cameraPosition = camera.getPosition();
39          glUniform3f(glGetUniformLocation(shader.ID, "camPos"), cameraPosition.x, cameraPosition.y, cameraPosition.z);
40          camera.uploadMatrix(shader, "camMatrix");
41
42          glDrawArrays(GL_TRIANGLES, 0, this->mesh.vertices.size());
43      }
44
45      void Drawable::applyRotation(float angleDeltaX, float angleDeltaY) {
46          glm::quat yaw = glm::angleAxis(angleDeltaX, glm::vec3(0.0f, 1.0f, 0.0f));
47          glm::quat pitch = glm::angleAxis(angleDeltaY, glm::vec3(1.0f, 0.0f, 0.0f));
48
49          this->rotationQuaternion = glm::normalize(yaw * pitch * this->rotationQuaternion);
50
51          this->rotationMatrix = glm::mat4_cast(this->rotationQuaternion);
52      }
```

Фиг. 3.4 – “draw” и “applyRotation” методите на “Drawable”

“InputController” съдържа два метода, чрез които се извършват действия съответно при натискане на бутон на компютърната мишка – “mouseButtonCallback” и при „скролване“ с мишката – “mouseScrollCallback”. Тези два метода се свързват със събитията, които GLFW библиотеката реализира за тези бутони още при създаването на обекта в неговия

конструктор, чрез използването на private метода “registerInputCallbacks”. Това е демонстрирано на фиг. 3.5.

```
7   InputController::InputController(GLFWwindow* window, Drawable& drawable) : window(window), drawable(drawable) {
8       this->rotationVelocity = glm::vec2(0.0f);
9       this->rotating = false;
10      this->scaling = false;
11
12      registerInputCallbacks();
13  }
39  void InputController::registerInputCallbacks() {
40      glfwSetScrollCallback(this->window, [](GLFWwindow* window, double xoffset, double yoffset) {
41          ImGui_ImplGlfw_ScrollCallback(window, xoffset, yoffset);
42          if (!ImGui::GetIO().WantCaptureMouse) {
43              InputController* controller = static_cast<InputController*>(glfwGetWindowUserPointer(window));
44              if (controller) controller->mouseScrollCallback(window, xoffset, yoffset);
45          }
46      });
47
48      glfwSetMouseButtonCallback(this->window, [](GLFWwindow* window, int button, int action, int mods) {
49          ImGui_ImplGlfw_MouseButtonCallback(window, button, action, mods);
50          if (!ImGui::GetIO().WantCaptureMouse) {
51              InputController* controller = static_cast<InputController*>(glfwGetWindowUserPointer(window));
52              if (controller) controller->mouseButtonCallback(window, button, action, mods);
53          }
54      });
55
56      glfwSetWindowUserPointer(this->window, this);
57  }
```

Фиг. 3.5 – Регистриране на callback функции за user input от компютърната мишка

“LightManager” е класът, в който се реализират методи като “addlight”, “updateLightBuffer”, “enableLights”. Тези методи работят с полетата на класа “lightUBO” от типа “UBO” и вектор от структурата Light, която съдържа в себе си padding полета, което се налага, поради използването на Uniform Buffered Object и съответно нуждата от използване на “layout(std140)” при създаването на “uniform Lights” променливата в шейдъра. Функциите на този клас се свеждат до добавяне на светлинни източници към масива от такива, подаването им към шейдъра и пускането и спирането им, чрез допълнителна променлива в шейдъра “lightsEnabled”.

4. Бъдещи подобрения

4.1. Подобрено стартиране без аргументи

На този етап програмата може да стартира визуализирането на избран от потребителя STL файл единствено чрез използването на “Open With” опцията при натискане на десен бутон върху дадения файл. Една добра добавка към програмата би била, когато приложението се стартира директно (двойно кликване върху .exe) и няма подаден STL файл като аргумент, да се появява диалогов прозорец за избор на файл. Това позволява потребителят да зареди произволен STL файл.

4.2. Инсталация и асоцииране на файлове

Друга полезна функционалност би била да се създаде инсталатор, който да инсталира приложението в системата, да добавя записи в Windows Registry за асоциация със STL файлове, с което да позволява в контекстното меню да излиза подсказка за “Open With” и името на програмата.

4.3. Drag-And-Drop поддръжка

Като допълнение към подобрението в т. 4.1., може да се развие опцията потребителя да има възможност единствено чрез хващане с курсора на мишката и плъзване на поддържан от програмата файл вътре в прозореца на програмата да може автоматично файлът да се зареди и модела да може да бъде разглеждан. Това също включва реализирането на подходящ графичен интерфейс, който уведомява потребителя, че в момента моделът се зарежда.

4.4. Подобрено осветление и Ambient Occlusion

В момента реализацията на Ambient Occlusion е чисто на ниво изчисление на ъгъл спрямо нормалата и view direction векторите. По – добър вариант би бил да се използва SSAO (Screen-Space Ambient Occlusion), който е малко по – тежък изчислително, но по – реалистичен.

В момента приложението използва стандартно Lambert/Phong осветление, което е достатъчно за базова визуализация на STL модели. В бъдеще може да

бъде добавен PBR (Physically Based Rendering) модел, който пресъздава светлината по по-реалистичен и физически коректен начин. Това би позволило материали, които реагират на осветлението по естествен начин – метални повърхности, матови обекти, гладък или груб релеф. Макар STL форматът да няма информация за материали, приложението може да задава базови стойности (металност, грапавина) според нуждите на визуализацията. Това би повишило значително качеството на изображението.

4.5. По – богат UI и настройки

В бъдещи версии може да се разшири графичният интерфейс – повече настройки за осветление, материали, AO параметри, камера, цветови теми и др. Това ще направи инструмента по-богат функционално и по-достъпен за различни типове потребители.

5. Приложения

- https://en.wikipedia.org/wiki/STL_(file_format)
- https://en.wikipedia.org/wiki/GLFW
- https://learnopengl.com/
- https://registry.khronos.org/OpenGL-Refpages
- https://community.khronos.org/
- https://www.youtube.com/watch?v=45MIykWJ-C4