МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Московский Авиационный Институт»
(Национальный Исследовательский
Университет)

Институт: №8 «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Реферат
по курсу «Фундаментальная информатика»
I семестр
Тема:
«Язык программирования C++ на примере разработки графического приложения для Windows»

Группа	М8О-109Б-22
Студент	Любарский И.В.
Преподаватель	Сысоев М.А.
Оценка	
Дата	

Москва, 2022

Содержание

Язык программирования C++

C++ — компилируемый, статически типизированный язык программирования общего назначения.

Поддерживает такие парадигмы программирования, как процедурное программирование, объектно-ориентированное программирование, обобщённое программирование.

Язык имеет богатую стандартную библиотеку, которая включает в себя распространенные контейнеры и алгоритмы, ввод-вывод, регулярные выражения, поддержку многопоточности и другие возможности. C++ сочетает свойства как высокоуровневых, так и низкоуровневых языков. В сравнении с его предшественником — языком C — наибольшее внимание уделено поддержке объектно-ориентированного и обобщенного программирования.

Синтаксис C++ унаследован от языка C. Изначально одним из принципов разработки было сохранение совместимости с C. Тем не менее C++ не является в строгом смысле надмножеством C; множество программ, которые могут одинаково успешно транслироваться как компиляторами C, так и компиляторами C++, довольно велико, но не включает все возможные программы на C.

Существует множество реализаций языка C++, как бесплатных, так и коммерческих и для различных платформ. Например, на платформе x86 это GCC, Visual C++, Intel C++ Compiler, Embarcadero (Borland) C++ Builder и другие. C++ оказал огромное влияние на другие языки программирования, в первую очередь на Java и C#.

Применение

C++ широко используется для разработки программного обеспечения, являясь одним из самых популярных языков программирования. Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также компьютерных игр.

Разработка игр и игрового движка

В инди-играх часто применяют высокоуровневые языки программирования и, разумеется, игровые движки (C# + Unity, Java + jMonkeyEngine). Однако на том же Unreal Engine вы можете успешно писать игры и на C++. Мало того, множество игр AAA-класса создаются именно на C++ и вот почему:

• топовые игры требуют хорошей оптимизации, а C++ довольно гибок для этого. Также на нем удобно писать в ООП-стиле, не спускаясь при этом до низкого уровня;

• применение своего движка для контроля любых игровых механик. Так, может, делают и не всегда, но делают часто. В основном это связано с наличием определенных проблем с лицензиями. Не стоит забывать и о необходимости делить прибыль с продаж;

• кроссплатформенность: создаваемая игра должна прекрасно работать на всех поддерживаемых устройствах. Пусть многие движки по умолчанию и заявляют такую возможность, но на практике она не всегда эффективна: где-то игра работает быстрее, где-то медленнее. На C++ код пишется таким образом, чтобы оптимизировать компиляцию в бинарные файлы на различных платформах.

Графика

OpenGL и Nvidia CUDA — это 2 платформы, которые позволяют запускать C++-код на графическом процессоре. GPU включает сотни небольших вычислительных ядер, способных одновременно выполнять простые математические операции. Если вы напишете код для GPU, распараллеливающий сложные вычисления, вы сможете добиться повышенной скорости и производительности. И C++ для этого прекрасно подходит.

Возможные применения:

• **майнинг биткоинов**. Код, написанный на C++, позволяет майнить быстрее;

• **обучение нейросетей**. Большинство Deep Learning-библиотек применяют C/C++-код на CUDA либо OpenCL, что обеспечивает более высокую скорость обучения и работы нейронных сетей. Причем многие функции доступны и на C++, и на других, более высокоуровневых языках;

• **шейдеры для игр и графического программного обеспечения**. Шейдеры — это небольшие участки кода, которые параллельно запускаются на GPU и выполняют сложные графические задачи (к примеру, трассировку лучей).

Операционные системы

Изначально, данный язык создавался для системного программирования. Поэтому нет ничего удивительного в том, что он активно используется при разработке новых операционных систем и различного программного обеспечения. C++ может работать с кодом низкого уровня, поэтому он практически идеален для разработки ОС. При этом, использование этого языка позволяет впоследствии проводить гибкую настройку операционной системы.

Разработка встроенных систем

Язык C++ отлично подходит для программирования встроенных систем. В первую очередь то связано с тем, что он обладает высокой производительностью и при этом простотой использования.

Такой инструмент экономичен с точки зрения использования ресурсов. Это позволяет выполнять любые программы с высокой скоростью. В результате, встроенные системы могут работать без замедления в режиме реального времени.

К таким встроенным системам можно отнести, например, управление беспилотными автомобилями, сенсорами, умными часами. При этом, C++ или C может работать с ресурсами памяти и за счет этого возможно внесение изменений в любую часть кода.

Десктопные и кроссплатформенные приложения

Это еще одна сфера, где активно применяется язык C++. Причем он используется здесь повсеместно, так как позволяет создавать кроссплатформенное ПО. Также у C++ есть масса интересных библиотек, которые делают работу программиста еще более гибкой.

Из наиболее известных можно назвать такие приложения, как Photoshop, Illustrator и Adobe Premiere. Facebook частично мигрировал с PHP на C++.

Преимущества языка программирования C++

1. Высокая скорость. Можно открыть любой тестер скоростей языков программирования, и вы увидите, что C++ является одним из наиболее высокоскоростных. При этом, можно использовать любой язык для решения локальных задач. Но если необходимо написать все приложение на одном языке, с этой задачей отлично справится C++.

2. Универсальность. Компиляторы языка C++ есть в любой операционной системе. При этом, написанные на этом языке программы могут без проблем исполняться на любой платформе.

3. Широкая сфера применения. Язык C++ можно использовать для разработки буквально всего от интернета вещей до умных часов, беспилотных транспортных средств и игр.

4. Большое сообщество. Язык постоянно обновляются и сюда внедряются различные полезные новшества. Но и это не все. C++ дополняется библиотеками и шаблонами, которые могут пригодиться как опытным программистам, так и начинающим разработчикам. Помимо этого, под C++ написано множество полезных книг и самоучителей, которые помогут быстрее освоить тонкости языка.

5. Принципы C++ заложены во многие современные языки программирования. Поэтому те. Кто его досконально изучат, смогут без труда освоить Java, JavaScript или C#, например. Тот же Java будет сложно понять, если не изучить хотя бы основы C++.

Графическое приложение (движок) на C++

Цель разработки данного приложения - создание, вывод на экран трехмерных объектов и управление над ними.

Необходимо собрать базовые трехмерные объекты приложения (примитивы): настроить их создание, буфферизацию на графический процессор и управление в коде и приложении.

Также важно создать код расчета положения и фрагментации (управление цвета пикселя) объектов и их вывода на экран. С этим работает OpenGL, производя вычисления на графическом процессоре.

Для удобного использования в коде нужно создать систему управления объектами внутри программного кода. В нее входят: объекты примитивов, создания и загрузки шейдера (см. OpenGL), загрузки сторонних файлов.

В итоге конечный пользователь будет иметь возможность свободно перемещаться в трехмерном пространстве и наблюдать за трехмерными объектами и их перемещениями на экране монитора при помощи ввода клавиатуры и мыши.

При разработки были использованы сторонние библиотеки GLFW, GLAD, GLM, std_image, кроссплатформенная утилита CMake.

Разработка производилась в IDE Microsoft Visual Studio.

OpenGL

OpenGL - спецификация, определяющая платформонезависимый программный интерфейс для написания приложений, использующих двумерную и трехмерную компьютерную графику.

Спецификация OpenGL описывает каким будет результат выполнения каждой конкретной функции и что она должна делать. Реализация же этих спецификаций занимается непосредственно разработчик.

OpenGL по своей сути — это большой конечный автомат: набор переменных, определяющий поведение OpenGL. Под состоянием OpenGL в основном имеется ввиду контекст OpenGL.

GLFW

GLFW — это библиотека, написанная на C, специально нацеленная для предоставления OpenGL самого необходимого для отрисовки контента на экран. Она позволяет создать контекст, определить параметры окна и работать с пользовательским вводом

GLAD

Поскольку OpenGL — это лишь спецификация — то реализация ложится на плечи разработчиков видеокарт. По этой причине, поскольку существует множество реализаций OpenGL реальное расположение OpenGL функций не доступно на этапе компиляции и их приходится получать на этапе исполнения. Фактически получение адресов функций ложится на плечи программиста.

Существуют библиотеки реализующие эту динамическую линковку и одной из самых популярных библиотек является GLAD.

GLM

GLM это аббревиатура от Open**GL** **M**athematics. При ее помощи становиться возможными программная реализация матриц, векторов и их вычисления.

CMake

CMake - то кроссплатформенная утилита, обладающая возможностями автоматизации сборки программного обеспечения из исходного кода. Сам CMake не занимается непосредственно сборкой, а лишь генерирует файлы сборки из предварительно написанного скрипт-файла «CMakeLists.txt» и предоставляет простой единый интерфейс управления.

Создание окна

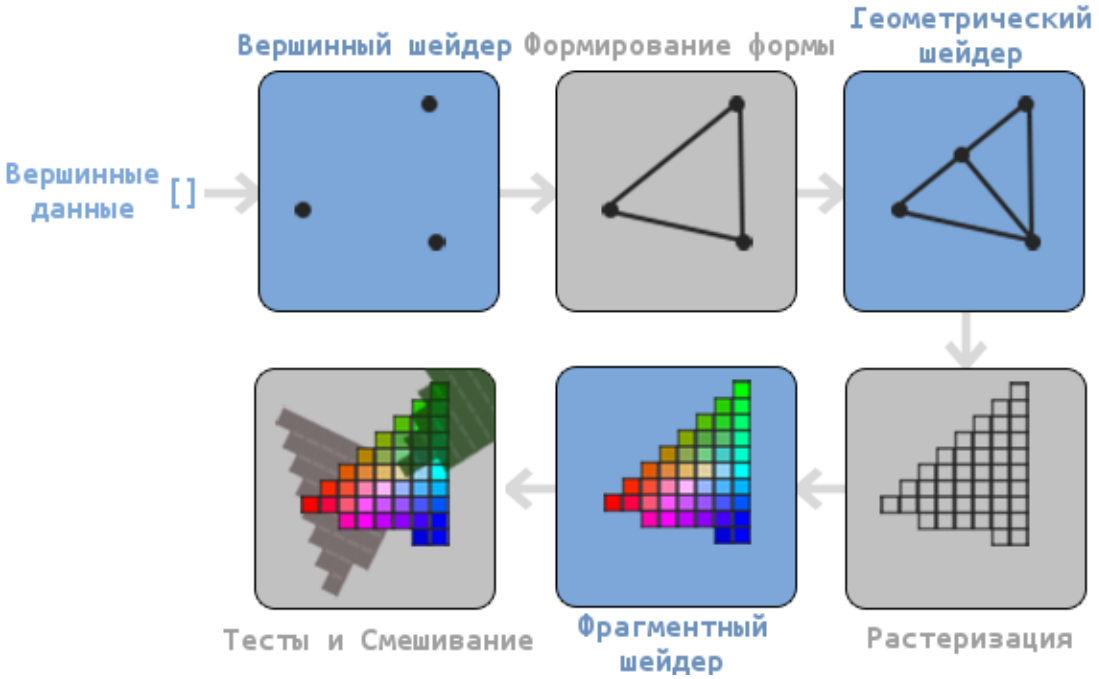
Для начала работы с графикой требуется инициализировать библиотеки GLAD и GLEW, создать окно и контекс, закрепить функции ввода пользователя.

```
if (!glfwInit()) return -1;

glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 6);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_COMPAT_PROFILE);

GLFWwindow* window;
window = glfwCreateWindow(windowSizeX, windowSizeY, "Engine", NULL, NULL);
if (!window) { glfwTerminate(); return -1; }

glfwSetWindowSizeCallback(window, glfwWindowSizeCallBack);
glfwSetKeyCallback(window, glfwKeyCallBack);

glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
glfwSetCursorPosCallback(window, mouse_callback);

glfwSetScrollCallback(window, scroll_callback);

glfwMakeContextCurrent(window);

if (!gladLoadGL()) return -1;
std::cout << "Renderer : " << glGetString(GL_RENDERER) << std::endl;
std::cout << "OpenGL version " << glGetString(GL_VERSION) << std::endl;
```

Шейдеры

В OpenGL все находится в 3D пространстве, но при этом экран и окно — это 2D матрица из пикселей. Поэтому большая часть работы OpenGL — это преобразование 3D координат в 2D пространство для отрисовки на экране. Процесс преобразования 3D координат в 2D координаты управляется графическим конвейером OpenGL.

Шейдеры — это небольшие программы выполняемые на графическом процессоре. Эти программы выполняются для каждого конкретного участка графического конвейера.



Некоторые из этих шейдеров могут настраиваться разработчиком, что позволяет нам писать собственные шейдеры для замены стандартных. Это дает гораздо больше возможностей тонкой настройки специфичных мест конвейера, и именно из за того, что они работают на GPU, позволяет сохранить процессорное время. Шейдеры пишутся на OpenGL Shading Language (GLSL). Это язык является полностью C подобным со встроенными функциями обработки графики.

Для реализации отображения положения объектов в пространстве и их цвета были написаны вершинный и фрагментный шейдера соответсвенно:

```
#version 460
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_normal;

out vec3 normal;
out vec3 fragPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;


void main(){
   fragPos = vec3(model * vec4(vertex_position, 1.0f));
   gl_Position = projection * view * model * vec4(vertex_position, 1.0);
   normal = mat3(transpose(inverse(model))) * vertex_normal;
}
```

```
#version 460

in vec3 fragPos;
in vec3 normal;

out vec4 frag_color;

uniform vec3 cudeColor;
uniform vec3 lightColor;
uniform vec3 lightPos;
uniform vec3 viewPos;
uniform int specInt;

void main(){

    float ambientStrenght = 0.1f;
    vec3 ambient = lightColor * ambientStrenght;

    vec3 norm = normalize(normal);
    vec3 lightDir = normalize(lightPos - fragPos);
    float diff = max(dot(norm, lightDir), 0.0f);
    vec3 diffuse = diff * lightColor;

    float specularStrenght = 0.5f;
    vec3 viewDir = normalize(viewPos - fragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), specInt);
    vec3 specular = specularStrenght * spec * lightColor;

    frag_color = vec4((ambient + diffuse + specular) * cudeColor, 1.0f);
}
```

Для получения данных в шейдер существует два способа:

- Использование буффера данных (VBO и VAO);
- uniform переменные;

VBO и VAO являются массивами значений (обычно float), под которые выделяется определенное место в памяти.

uniform переменные возможно задать внутри программы, найдя их местоположение внутри уже созданной шейдерной программы.

Для использования, создания и загрузки шейдерной программы были созданы отдельные классы самой шейдерной программы и ее загрузчика.

Класс шейдерной программы

```
class ShaderProgramm {
public:
    ShaderProgramm(const std::string& vertexShader, const std::string& fragmentShader);
    ~ShaderProgramm();
    bool isCompiled() { return is_Compiled; }
    void use();
    GLuint getID() { return ID; }
private:
    bool createShader(const std::string& source, const GLenum shaderType, GLuint& shaderID);
    bool is_Compiled = false;
    GLuint ID = 0;
};
```

В этом классе были реализованы конструктор (создание шейдерной программы при вводе строк вершинного и фрагментного шейдера) и деструктор (непосредственное удаление):

```cpp
ShaderProgramm::ShaderProgramm(const std::string& vertexShader, const std::string& fragmentShader) {

    GLuint vertexShaderID;
    if (!createShader(vertexShader, GL_VERTEX_SHADER, vertexShaderID)) {

        std::cerr << "VERTEX SHADER COMPILE ERROR" << std::endl;

    }

    GLuint fragmentShaderID;
    if (!createShader(fragmentShader, GL_FRAGMENT_SHADER, fragmentShaderID)) {

        std::cerr << "FRAGMENT SHADER COMPILE ERROR" << std::endl;
        glDeleteShader(vertexShaderID);

    }

    ID = glCreateProgram();
    glAttachShader(ID, vertexShaderID);
    glAttachShader(ID, fragmentShaderID);
    glLinkProgram(ID);

    GLint success;
    glGetProgramiv(ID, GL_LINK_STATUS, &success);
    if(!success){

        GLchar infoLog[1024];
        glGetShaderInfoLog(ID, 1014, nullptr, infoLog);
        std::cerr << "ERROR::SHADER: LINK TIME ERROR: \n" << infoLog << std::endl;

    }
    else { is_Compiled = true; }

    glDeleteShader(vertexShaderID);
    glDeleteShader(fragmentShaderID);

}
```

```cpp
ShaderProgramm::~ShaderProgramm() { glDeleteProgram(ID); }
```

Также в этом классе присутствуют методы использования шейдерной программы и создание шейдера:

```cpp
bool ShaderProgramm::createShader(const std::string& source, const GLenum shaderType, GLuint& shaderID) {

    shaderID = glCreateShader(shaderType);
    const char* code = source.c_str();
    glShaderSource(shaderID, 1, &code, NULL);
    glCompileShader(shaderID);

    GLint success;
    glGetShaderiv(shaderID, GL_COMPILE_STATUS, &success);
    if (!success) {

        GLchar infoLog[1024];
        glGetShaderInfoLog(shaderID, 1014, nullptr, infoLog);
        std::cerr << "ERROR::SHADER: COMPILE TIME ERROR: \n" << infoLog << std::endl;
        return false;

    }
    return true;
}
```

```cpp
void ShaderProgramm::use(){ glUseProgram(ID); }
```

Класс загрузчика шейдеров

```
class ResourceManager {
    public:
        ResourceManager(const std::string& executablePath);
        ~ResourceManager() = default;

        std::string getPath();
        std::shared_ptr<Shader::ShaderProgramm> loadShader(const std::string& shaderName,
                                                           const std::string& vertexPath,
                                                           const std::string& fragmentPath);
        std::shared_ptr<Shader::ShaderProgramm> getShader(const std::string shaderName);
    private:
        std::string getFileString(const std::string& relativePath) const;

        typedef std::map<const std::string, std::shared_ptr<Shader::ShaderProgramm>> ShaderProgrammsMap;
        ShaderProgrammsMap shaderProgramms;

        std::string path;
};
```

В классе загрузчика шейдеров также были реализованы конструктор и деструктор:

```
ResourceManager::ResourceManager(const std::string& executablePath) {

    size_t found = executablePath.find_last_of("/\\");
    path = executablePath.substr(0, found);

    found = path.find_last_of("/\\");
    path = path.substr(0, found);
    found = path.find_last_of("/\\");
    path = path.substr(0, found);

}
```

Главной частью этого класса является методы загрузки кода шейдера из отдельного файла:

```
std::string ResourceManager::getFileString(const std::string& relativePath) const {

    std::ifstream f;
    f.open(path + '/' + relativePath.c_str(), std::ios::in | std::ios::binary);
    if (!f.is_open()) {

        std::cerr << "failed to open file : " << relativePath << std::endl;
        return std::string{};

    }

    std::stringstream buffer;
    buffer << f.rdbuf();
    return buffer.str();

}
```

```
std::shared_ptr<Shader::ShaderProgramm> ResourceManager::loadShader(const std::string& shaderName,
                                                                    const std::string& vertexPath, const std::string& fragmentPath) {

    std::string vertexString = getFileString(vertexPath);
    if (vertexString.empty()) { std::cerr << "No vertex shader!" << std::endl; }
    std::string fragmentString = getFileString(fragmentPath);
    if (fragmentString.empty()) { std::cerr << "No fragment shader!" << std::endl; }

    std::shared_ptr<Shader::ShaderProgramm>& newShader = shaderProgramms.emplace(shaderName,
                                                                                 std::make_shared<Shader::ShaderProgramm>(vertexString,
                                                                                                                          fragmentString)).first->second;
    if (newShader->isCompiled()) { return newShader; }
    std::cerr << "Can`t load shader programm : \n" << "Vertex : " << vertexPath << "\n Fragment : " << fragmentPath << std::endl;
    return nullptr;
}
```

Готовая шейдерная программа сохраняется в map для дальнейшего доступа.

Базовые трехмерные объекты (примитивы) движка

Примитивы движка были реализованы в отдельных классах при помощи константных моделей (набора вершин и нормалей).

Константные модели являются трехмерными объектами, координаты вершин которых ограничены пространством от -1 до 1 по каждой оси.

Пример константной модели пирамиды (3 координаты на каждую вершину + 3 координаты вектора нормали для каждой вершины):

```
const GLfloat piramidpoints[] = {
    -0.5f, -0.5f,  0.5f, 0.0f, -1.0f,  0.0f,
     0.5f, -0.5f,  0.5f, 0.0f, -1.0f,  0.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, -1.0f,  0.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, -1.0f,  0.0f,
     0.5f, -0.5f,  0.5f, 0.0f, -1.0f,  0.0f,
     0.5f, -0.5f, -0.5f, 0.0f, -1.0f,  0.0f,

    -0.5f, -0.5f, 0.5f, 0.0f,  0.5f, 1.0f,
     0.5f, -0.5f, 0.5f, 0.0f,  0.5f, 1.0f,
     0.0f,  0.5f, 0.0f, 0.0f,  0.5f, 1.0f,

    -0.5f, -0.5f,  0.5f, -1.0f, 0.5f, 0.0f,
    -0.5f, -0.5f, -0.5f, -1.0f, 0.5f, 0.0f,
     0.0f,  0.5f,  0.0f, -1.0f, 0.5f, 0.0f,

     0.5f, -0.5f,  0.5f, 1.0f, 0.5f, 0.0f,
     0.5f, -0.5f, -0.5f, 1.0f, 0.5f, 0.0f,
     0.0f,  0.5f,  0.0f, 1.0f, 0.5f, 0.0f,

    -0.5f, -0.5f, -0.5f, 0.0f, 0.5f, -1.0f,
     0.5f, -0.5f, -0.5f, 0.0f, 0.5f, -1.0f,
     0.0f,  0.5f,  0.0f, 0.0f, 0.5f, -1.0f
};
```

Каждый класс объекта имеет собственные переменные буфферов и положения в мировом пространстве. В каждом реализованы конструктор и деструктор класса, а также метод его отрисовки.

Пример класса:

```
class Piramid {
    public:
        Piramid();
        ~Piramid() = default;
        void Draw(glm::vec4 material, glm::vec3 cords, glm::vec3 angle, std::shared_ptr<Shader::ShaderProgramm> Shad);
    private:
        GLuint VBO, VAO;
        glm::mat4 model;
};
```

Конструктор (создание буффера и отправка на графический процессор):

```
Piramid::Piramid() {

    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(piramidpoints), piramidpoints, GL_STATIC_DRAW);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
    glEnableVertexAttribArray(0);

    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
    glEnableVertexAttribArray(1);

    glBindVertexArray(0);

}
```

Отрисовка (при помощи созданного буффера VBO в VAO):

```
void Piramid::Draw(glm::vec4 material, glm::vec3 cords, glm::vec3 angle, std::shared_ptr<Shader::ShaderProgramm> Shad) {

    GLint cudeColor = glGetUniformLocation(Shad->getID(), "cudeColor");
    GLuint specularLoc = glGetUniformLocation(Shad->getID(), "specInt");
    GLuint modelLoc = glGetUniformLocation(Shad->getID(), "model");

    model = glm::mat4(1.0f);
    model = glm::translate(model, cords);
    model = glm::rotate(model, glm::radians(45.0f * (GLfloat)glfwGetTime()), angle);

    glUniform3f(cudeColor, material.x, material.y, material.z);
    glUniform1i(specularLoc, material.w);
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 18);

    glBindVertexArray(0);

}
```

Пользовательское взаимодействие (Камера)

Для наблюдения за работой движка и базовыми объектами разработчику и конечному пользователю требуется возможность оглядеть каждый объект со всех сторон. Для этого требуется “камера”, которая будет изменять свое местоположение и направление в зависимости от ввода.

Для реализации этого требуется создание матриц преобразования и вектора координатной системы мира и камеры.

Данные матрицы будут отравляться в шейдеры в качестве uniform переменных (см. Пункт Шейдеры):

```
glm::vec3 cameraPos = glm::vec3(6.0f, 0.0f, 9.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
```

```
glm::mat4 model = glm::mat4(1.0f);
glm::mat4 view = glm::mat4(1.0f);
glm::mat4 projection = glm::mat4(1.0f);
```

```
glUniform3f(viewPosition, cameraPos.x, cameraPos.y, cameraPos.z);
```

```
view = glm::mat4(1.0f);
view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

projection = glm::mat4(1.0f);
projection = glm::perspective(glm::radians(fov), windowSizeX / windowSizeY, 0.1f, 100.0f);
```

Матрица model преобразует положение трехмерного объекта относительно его координат, а также относительно мирового пространства.

Матрица view преобразует данные об вершинах объекта в соответствии с положением камеры.

Матрица projection преобразует полученные данные после всех других преобразований и создает перспективу, которая потом выводиться на экран.

Все управление было реализовано в данных функциях, которые позже были закреплены в main:

```
void glfwWindowSizeCallBack(GLFWwindow* window, int x, int y);
void glfwKeyCallBack(GLFWwindow* window, int key, int scancode, int action, int mode);
void do_movement();
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
```

```
void glfwWindowSizeCallBack(GLFWwindow* window, int x, int y) {

    windowSizeX = x;
    windowSizeY = y;
    glViewport(0, 0, windowSizeX, windowSizeY);

}
```

```
void glfwKeyCallBack(GLFWwindow* window, int key, int scancode, int action, int mode) {

    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) { glfwSetWindowShouldClose(window, GL_TRUE); }

    if (action == GLFW_PRESS)
        keys[key] = true;

    else if (action == GLFW_RELEASE)
        keys[key] = false;

}
```

```
void do_movement() {

    GLfloat cameraSpeed = 5.0f * deltaTime;
    if (keys[GLFW_KEY_W])
        cameraPos += cameraSpeed * cameraFront;
    if (keys[GLFW_KEY_S])
        cameraPos -= cameraSpeed * cameraFront;
    if (keys[GLFW_KEY_A])
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (keys[GLFW_KEY_D])
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (keys[GLFW_KEY_LEFT_SHIFT])
        cameraPos += glm::vec3(0.0f, -cameraSpeed, 0.0f);
    if (keys[GLFW_KEY_SPACE])
        cameraPos += glm::vec3(0.0f, cameraSpeed, 0.0f);

}
```

```
void mouse_callback(GLFWwindow* window, double xpos, double ypos) {
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    GLfloat xoffset = xpos - lastX;
    GLfloat yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

    GLfloat sensitivity = 0.05;
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;

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

    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));
    cameraFront = glm::normalize(front);
}
```

```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset) {
    if (fov >= 1.0f && fov <= 45.0f)
        fov -= yoffset;
    if (fov <= 1.0f)
        fov = 1.0f;
    if (fov >= 45.0f)
        fov = 45.0f;
}
```

Главный “игровой” цикл

Так как отрисовка объектов приложения динамическая, нужно создать бесконечный цикл с возможностью выхода, где будут происходить изменения и постоянная отрисовка существующих трехмерных моделей.

Алгоритм каждой итерации таков:

- Очистка прошлого отображения;
- Нахождение uniform переменных;
- Запись некоторых uniform переменных;
- Отрисовка примитивов;
- Запись матриц преобразований (матрица model записывается для каждого объекта отдельно в его методе);
- Смена буффера;

```
while (!glfwWindowShouldClose(window))
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


    pDefaultShaderProgramm->use();

    GLint lightPosLoc = glGetUniformLocation(pDefaultShaderProgramm->getID(), "lightPos");
    GLint viewPosition = glGetUniformLocation(pDefaultShaderProgramm->getID(), "viewPos");
    GLint lightColor = glGetUniformLocation(pDefaultShaderProgramm->getID(), "lightColor");
    glUniform3f(lightPosLoc, lightPos.x, lightPos.y, lightPos.z);
    glUniform3f(viewPosition, cameraPos.x, cameraPos.y, cameraPos.z);
    glUniform3f(lightColor, 1.0f, 1.0f, 1.0f);

    GLuint modelLoc = glGetUniformLocation(pDefaultShaderProgramm->getID(), "model");
    GLuint viewLoc = glGetUniformLocation(pDefaultShaderProgramm->getID(), "view");
    GLuint projectionLoc = glGetUniformLocation(pDefaultShaderProgramm->getID(), "projection");
    GLuint specularLoc = glGetUniformLocation(pDefaultShaderProgramm->getID(), "specInt");
```

```
    for (int i = 0; i < 5; i++) {

        p.Draw(glm::vec4(0.33f * ((i + 1) % 4), 0.33f * ((i + 1) % 3), 0.33f * ((i + 1) % 2), 50 * i + 2), glm::vec3(3.0f * i, 3.0f, 0.0f),
            glm::vec3(0.33f * ((i + 1) % 4), 0.33f * ((i + 1) % 3), 0.33f * ((i + 1) % 2)), pDefaultShaderProgramm);

        t.Draw(glm::vec4(0.33f * ((i + 1) % 4), 0.33f * ((i + 1) % 2), 0.33f * ((i + 1) % 3), 50 * i + 2), glm::vec3(3.0f * i, -3.0f, 0.0f),
            glm::vec3(0.33f * ((i + 1) % 4), 0.33f * ((i + 1) % 2), 0.33f * ((i + 1) % 3)), pDefaultShaderProgramm);

        c.Draw(glm::vec4(0.33f * ((i + 1) % 2), 0.33f * ((i + 1) % 3), 0.33f * ((i + 1) % 4), 50 * i + 2), glm::vec3(3.0f * i, 0.0f, 0.0f),
            glm::vec3(0.33f * ((i + 1) % 2), 0.33f * ((i + 1) % 3), 0.33f * ((i + 1) % 4)), pDefaultShaderProgramm);

    }
```

```
view = glm::mat4(1.0f);
view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);

projection = glm::mat4(1.0f);
projection = glm::perspective(glm::radians(fov), windowSizeX / windowSizeY, 0.1f, 100.0f);

glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, glm::value_ptr(projection));
```

```
glfwSwapBuffers(window);
```

Итог

Было создано графическое приложения на основе OpenGL с модульной структурой. Это позволяет в дальнейшем легко дорабатывать и продолжать разрабатывать данную программу (в особенности добавлять новые примитивы).

Были написаны шейдеры для реализации преобразований вершин примитивов и отображения на экране, обработки данных на графическом процессоре.

Для разработки потребовалось досконально изучить возможности C++, а именно: управление указателями и ссылками, классы и объекты, разделение на заголовочные файлы.

Также немаловажным является приобретение навыка поиска и исправления ошибок в коде. Множественные ошибки линковки и компиляции заставляли по часу или более сидеть и искать пропущенную запятую.

Источники

- https://ru.wikipedia.org/wiki/C%2B%2B - информации о C++
- https://otus.ru/nest/post/1967/ - информация о применении C++
- https://cmake.org - сайт CMake
- https://ru.wikipedia.org/wiki/OpenGL - информация об OpenGL
- https://glm.g-truc.net/0.9.8/index.html - сайт glm
- https://glad.dav1d.de - сайт для генерации glad
- https://www.glfw.org - сайт GLFW