УДК 004.921

ТРЕХМЕРНАЯ КОМПЬЮТЕРНАЯ ГРАФИКА И РЕНДЕРИНГ НА ПРИМЕРЕ OPENGL/C++

[1]Лашукевич К. Д., [2]Кондратёнок Е. В.
[1]Белорусский национальный технический университет,
Минск, Беларусь, samprise228@gmail.com,
[2] Белорусский национальный технический университет,
Минск, Беларусь, elena_kondr@tut.by

Аннотация. Рассмотрены основные определения компьютерной графики. Приведена реализация OpenGL/C++ с помощью GLAD и GLFW на примере отрисовки Солнца, Земли и Луны с моделью затенения по Фонгу.

Ключевые слова: компьютерная графика, 3D графика, графический API, рендеринг, модель затенения, освещение.

Abstract. The basic definitions of computer graphics are considered. An implementation of OpenGL/C++ using GLAD and GLFW is given using the example of rendering the Sun, Earth and Moon with the Phong shading model.

Key words: computer graphics, 3D graphics, graphics API, rendering, shading model, lighting.

Введение.

Трехмерная графика или 3D-графика – это область компьютерной графики, набор техник и инструментов, позволяющих создавать трехмерные объекты с использованием тестур и цвета. Трехмерная графика отличается от двумерных изображений процессом создания геометрической проекции трехмерной модели сцены (виртуального пространства) в 2D. Это выполняется с помощью специального ПО [1].

Графический API – это набор функций и процедур, которые разработчик может использовать для создания и управления графикой [2; 3].

В настоящее время для разработки компьютерных игр используют три основные API:

1. DirectX – это набор, разработанный Microsoft для программирования под Microsoft Windows [4].

2. Vulkan – это набор, разработанный Khronos Group для программирования под мультиплатформу. Изначально был известен как «новое поколение OpenGL» или просто «glNext», но после анонса компания отказалась от этих названий в пользу названия Vulkan [5].

3. OpenGL – спецификация, определяющая платформонезависимый (независимый от языка программирования) программный интерфейс для написания приложений, использующих двумерную и трехмерную компьютерную

графику, разработанная Silicon Graphics, затем Khronos Group для программирования под мультиплатформу [6].

Все вышеперечисленные графические API умеют работать с 2D и 3D графикой. Разница между Vulkan и DirectX состоит в том, что Vulkan находится на более низком уровне абстракции и позволяет лучше контролировать аппаратные ресурсы системы, и вследствие чего, выдает лучшую производительность в сложных графических приложениях.

OpenGl является более старым и намного проще в освоении новичкам. При знакомстве с программированием компьютерной графики лучше всего будет начать изучение именно с OpenGL [1].

Рендеринг – это преобразование информации о трехмерных объектах в битовую карту, которая может быть отображена. Рендеринг требует значительной памяти и вычислительной мощности [1].

Основы трехмерной компьютерной графики.

Полигоны.

Все современные графические приложения строятся на основе полигонов (рисунок 1), а уже из полигонов состоят графические модели различной сложности.

Полигон – это множества точек, называемых вершинами, которые соединены ребрами.

Полигоны имеют различные формы, такие как треугольники, четырехугольники и многоугольники. Чаще всего используются треугольники, так как у треугольного полигона имеется свойство компланарности (все вершины полигона лежат в одной плоскости).

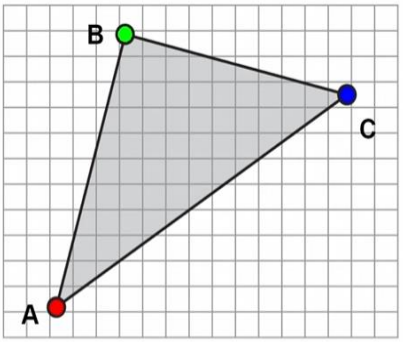

Рисунок 1 – Полигон с тремя вершинами

Вершины полигона имеют координаты в 3-мерной системе координат. В дальнейшем они используются для матриц перемещения, скалирования и поворота.

Vertex	**Position (XYZ)**
A	(2.1, 1.2, 0.2)
B	(5.3, 11.8, –4.6)
C	(14.8, 9.5, 2.1)

Шейдеры.

Шейдер – это компьютерная программа, предназначенная для исполнения процессорами видеокарты.

Для написания шейдерных программ используется специальный язык шейдеров GLSL, который схож с языком программирования C и имеет удобные функции для работы с векторами и матрицами.

Для обработки полигона используются вершинный и фрагментный шейдеры (рис. 2). Вершинный шейдер параллельно обрабатывает каждую вершину в полигоне и передает нужные параметры на вход фрагментного шейдера, а фрагментный шейдер параллельно обрабатывает каждый пиксель попавший в область полигона. Все исполнение происходит на графических ядрах видеоускорителя [7].

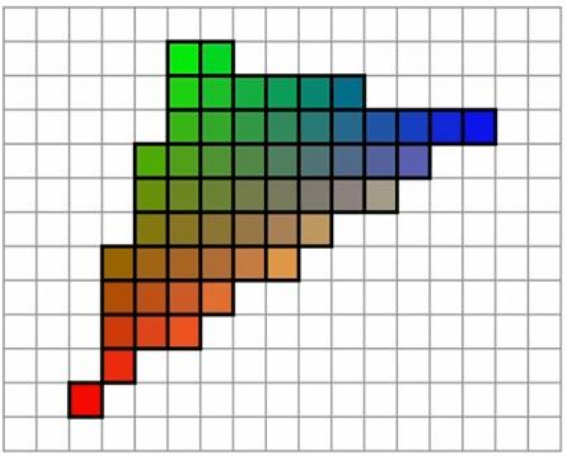

Рисунок 2 – Полигон после обработки фрагментарным шейдером

Буферы.

Для того чтобы каждый раз не передавать координаты и параметры цвета, существует специальный вершинный буфер. Этот буфер передается один раз из оперативной памяти в память видеоускорителя и хранится там пока он актуален.

Чтобы собрать модель из полигонов часто используют индексный буфер. Он позволяет сократить количество вершин тем, что полигоны отрисовываются не по координатам, а по индексам где лежат координаты.

Таким образом можно отрисовать фигуру из четырех вершин и двух полигонов, вместо 6 вершин и двух полигонов (рис. 3).

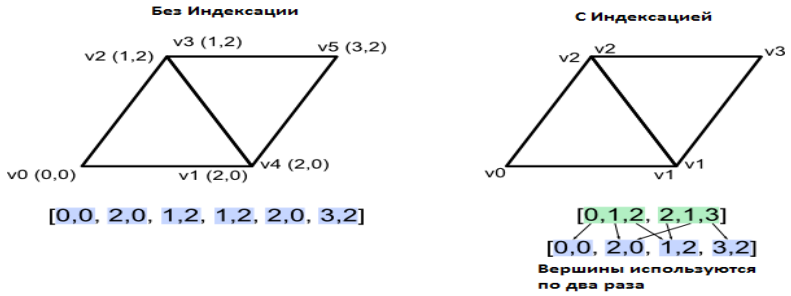

Рисунок 3 – Применение индексного буфера

Текстуры и материалы.

Также в компьютерной графике используются модели освещения, текстуры и материалы.

Пиксели в двумерной графике имеют свойства цвета, яркости и положения. Трехмерные пиксели, кроме этого, имеют свойство глубины, которое указывает, где точка находится на воображаемой оси Z. При объединении множества трехмерных пикселей, где каждый имеет свое значение глубины, получается трехмерная поверхность, называемая текстурой [1].

Текстуры для модели обрабатываются таким же способом, как и цвет, только вместо интерполирования цвета, пиксель полигона принимает цвет пикселя текстуры по координатам.

Материалы используются для просчета освещения. Материалы могут отражать, поглощать или пропускать свет.

Модели освещения.

Модель освещения – это расчет освещения трехмерных объектов, в том числе полигональных моделей и примитивов, а также метод интерполяции освещения по всему объекту.

Самая простая для реализации модель – это модель отражения Фонга (рис. 4). Она представляет собой эмпирическую модель локального освещения точек на поверхности. Основана на том, что блестящие поверхности имею небольшие интенсивные зеркальные блики, а тусклые поверхности имеют большие блики, которые исчезают более постепенно. Включает в себя такие этапы как [8]:

1. Фоновое освещение.
2. Рассеянное освещение.
3. Блики.

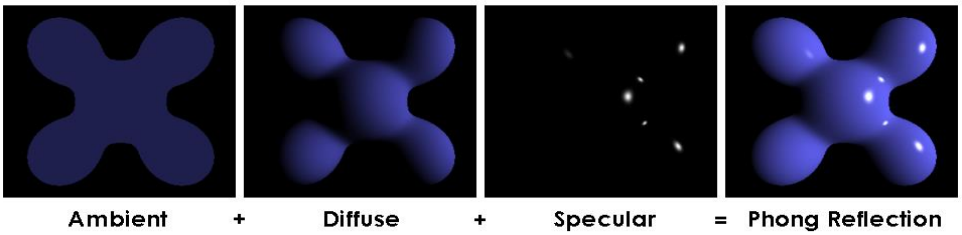

Рисунок 4 – Затенение по Фонгу

Камера.

Камера предназначена для отображения того, что видит пользователь. Она имеет координаты и проекцию.

Существует два вида проекций:

– перспективная;

– ортогональная.

Для реализации необходима специальная фигура «фрустум» (рис. 5). Все что попадает в эту фигуру будет отрисовано, а что не попадает будет отсечено. Имея разные фигуры «Фрустума», будет происходить разная проекция.

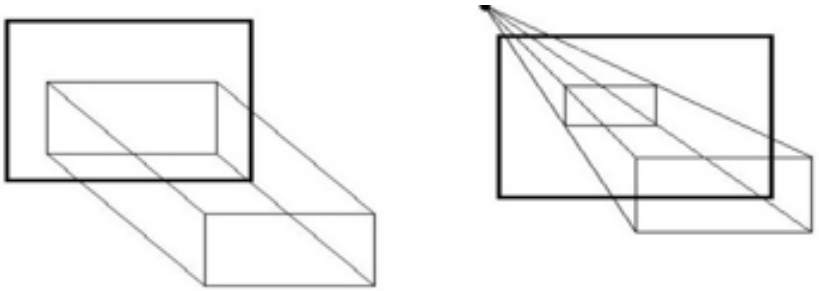

Рисунок 5 – Ортогональный и перспективный фрустум

Для ортогональной проекции используется прямоугольник, при этом виде проекции невозможно увидеть удаление объекта от камеры, он всегда будет одного размера.

Для перспективной проекции используется усеченная пирамида, при этом виде проекции предмет, который удален дальше от камеры будет меньше.

Реализация OpenGL/C++ с помощью GLAD и GLFW.

Так как OpenGL является спецификацией, то нам нужен специальный интерфейс для взаимодействия с ним. В C++ есть специальная библиотека GLAD, она позволяет нам вызывать функции для отрисовки графики с использованием OpenGL [9].

Для отображения нам необходимо окно, которое можно получить с помощью библиотеки GLFW.

GLAD и GLFW можно бесплатно скачать с GitHub и подключить к своему проекту [10; 11].

В качестве примера были отрисованы Солнце, Земля и Луна с моделью затенения по Фонгу [8].

Для отрисовки сферы нам необходимо знать координаты ее вертексов, координаты нормалей, координаты текстур, индексы и написать шейдерную программу.

Чтобы отрисовать какой-либо объект используется вертексный массив буферов, который содержит буферы с необходимыми параметрами. Для каждого уникального объекта свой вертексный массив буферов будет уникальным.

Для полной отрисовки используется пользовательский класс который хранит в себе нужные параметры.

_vertex_array_object = new VertexArray();

_vertex_buffer_position = new VertexBuffer(Planet::getInterleavedVertices(),
Planet::getInterleav-
edVertexSize(),
_planet_layout);

_vertex_buffer_normal = new VertexBuffer(Planet::getNormals(),
Planet::getNormalSize(),
_planet_layout);

_vertex_buffer_texCoords = new VertexBuffer(Planet::getTexCoords(),
Planet::getTexCoordSize(),
_planet_layout);

_index_buffer = new IndexBuffer(Planet::getIndices(), Planet::get-
IndexSize());

*_vertex_array_object->addVertexBuffer(*_vertex_buffer_position);*
*_vertex_array_object->addVertexBuffer(*_vertex_buffer_normal);*
*_vertex_array_object->addVertexBuffer(*_vertex_buffer_texCoords);*
*_vertex_array_object->setIndexBuffer(*_index_buffer);*

Так происходит заполнение массива вертексных буферов для дальнейшей отрисовки.

Далее мы создаем объект класса и назначаем ему текстуры, которые загружены в формате .bmp.

Planet earth(1.0f, 36, 18, true, 3);
earth.setTexture("./textures/earth.bmp");

Так же делаем и с другими объектами.

Далее нам нужны шейдерные программы для объекта и источника света.

const char vertex_shader = R"(*
#version 460
// uniforms
uniform mat4 model_matrix;
uniform mat4 view_projection_matrix;
// vertex attribs (input)
layout(location=0) in vec3 vertex_position;
layout(location=1) in vec3 vertex_normal;
layout(location=2) in vec2 vertex_tex_coord;
// varyings (output)
out vec3 frag_normal;
out vec3 frag_position;

```
	out vec2 texCoord;

	void main()
	{
		frag_normal = mat3(transpose(inverse(model_matrix))) * vertex_nor-
mal;
		vec4 world_vertex_position = model_matrix * vec4(vertex_position, 1.0);
		frag_position = world_vertex_position.xyz;
		texCoord = vertex_tex_coord;
		gl_Position = view_projection_matrix * world_vertex_position;
	}
	)";

	const char* fragment_shader = R"(
		#version 460
		// uniforms
		uniform vec3 light_color;
		uniform vec3 light_position;
		uniform float ambient_factor;
		uniform float diffuse_factor;
		layout(binding=0) uniform sampler2D in_texture;
		// varyings (input)
		in vec3 frag_normal;
		in vec3 frag_position;
		in vec2 texCoord;
		// output
		out vec4 frag_color;

		void main() {
		//ambient
		vec3 ambient = ambient_factor * light_color;
		//diffuse
		vec3 normal = normalize(frag_normal);
		vec3 ligth_direction = normalize(light_position - frag_position);
		vec3 diffuse = diffuse_factor * light_color * max(dot(normal, ligth_di-
rection), 0.0);
		//specular
		vec3 specular = vec3(0.0f);
		frag_color = texture(in_texture, texCoord) * vec4(ambient + diffuse +
specular, 1.0);
		}
	)";
```

```
const char* ligth_vertex_shader = R"(
#version 460
// uniforms
uniform mat4 model_matrix;
uniform mat4 normal_matrix;
uniform mat4 view_projection_matrix;
// vertex attribs (input)
layout(location=0) in vec3 vertex_position;
layout(location=1) in vec3 vertex_normal;
layout(location=2) in vec2 vertex_tex_coord;
// varyings (output)
out vec3 esNormal;
out vec3 color;
out vec2 texCoord;

void main()
{
	esNormal = vec3(normal_matrix * vec4(vertex_normal, 1.0));
	texCoord = vertex_tex_coord;
	gl_Position = view_projection_matrix * model_matrix * vec4(ver-
tex_position, 1.0);
}
)";

const char* ligth_fragment_shader = R"(
	#version 460
	// uniforms
	uniform vec3 light_color;
	layout(binding=0) uniform sampler2D in_texture;
	// varyings (input)
	in vec3 esNormal;
	in vec2 texCoord;
	// output
	out vec4 frag_color;

	void main() {
	frag_color = texture(in_texture, texCoord) * vec4(light_color, 1.0f);
	}
)";
```

Теперь эти шейдеры необходимо скомпилировать и собрать в шейдерную программу.

```
shader_program = new ShaderProgram(vertex_shader, fragment_shader);
	if(!shader_program->isCompile())
```

```
{
return -4;
}

ligth_shader_program = new ShaderProgram(ligth_vertex_shader, ligth_fragment_shader);
if(!ligth_shader_program->isCompile())
{
return -4;
}
```

Для того, чтобы видеть глубину картинки и полигоны не накладывались друг на друга, необходимо включить глубину.

glEnable(GL_DEPTH_TEST);

Далее идет передача параметров в шейдерные программы и отрисовка моделей. Так как все объекты являются сферами, то можно использовать одни и те же вершины.

```
while(_is_window_alive)
{
RenderOpenGL::clear();
shader_program->bind();
camera.setProjection(is_perspective_mode ? Camera::ProjectionMode::Perspective : Camera::ProjectionMode::Orthographic);
shader_program->setMatrix4("view_projection_matrix", camera.getProjectionMatrix() * camera.getViewMatrix());
shader_program->setVec3("light_color", glm::vec3(light_color[0], light_color[1], light_color[2]));
shader_program->setFloat("ambient_factor", ambient_factor);
shader_program->setFloat("diffuse_factor", diffuse_factor);
shader_program->setVec3("light_position", glm::vec3(sun.getLocation()[0], sun.getLocation()[1], sun.getLocation()[2]));
earth.getTexture()->bind(0);
earth.updateMatrix();
shader_program->setMatrix4("model_matrix", earth.getModelMatrix());
RenderOpenGL::draw(*space.getVertexArrayObject());
moon.getTexture()->bind(0);
moon.updateMatrix();
shader_program->setMatrix4("model_matrix", moon.getModelMatrix());
RenderOpenGL::draw(*space.getVertexArrayObject());
ligth_shader_program->bind();
sun.getTexture()->bind(0);
sun.updateMatrix();
```

*ligth_shader_program->setMatrix4("view_projection_matrix", camera.getProjectionMatrix() * camera.getViewMatrix());*

ligth_shader_program->setMatrix4("model_matrix", sun.getModelMatrix());

ligth_shader_program->setVec3("light_color", glm::vec3(light_color[0], light_color[1], light_color[2]));

*RenderOpenGL::draw(*space.getVertexArrayObject());*

_window->onUpdate();

}

Для самой отрисовки необходим только вертексный массив буферов

void RenderOpenGL::draw(const VertexArray& vertex_array)

{

vertex_array.bind();

glDrawElements(GL_TRIANGLES, static_cast<GLsizei>(vertex_array.getIndicesCount()), GL_UNSIGNED_INT, nullptr);

}

В результате выполнения кода мы получили три отрисованных объекта и модель освещения с использование GLAD, GLFW, GLSL и C++. Благодаря уникальности массива буферов можно менять параметры объекта независимо от других. С использованием матрицы скалирования был изменен размер каждой модели в примере (рис. 6).

Рисунок 6 – Результат работы программы

Список использованных источников:

1. Шохоева, А. И. Технологии компьютерной графики и визуализация данных [Электронный ресурс]. – Режим доступа: https://cyberleninka.ru/article/n/tehnologii-kompyuternoy-grafiki-i-vizualizatsii-dannyh/viewer. – Дата доступа: 09.11.2023.

2. Джейсон, Г. Игровой движок. Программирование и внутреннее устройство / Г. Джейсон. – Питер, 2021.
3. OpenGL API Documentation [Электронный ресурс]. – Режим доступа: https://docs.gl/. – Дата доступа: 09.11.2023.
4. DirectX [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/DirectX. – Дата доступа: 09.11.2023.
5. Vulkan [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Vulkan. – Дата доступа: 09.11.2023.
6. OpenGL [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/OpenGL. – Дата доступа: 09.11.2023.
7. Документация языка шейдеров GLSL [Электронный ресурс]. – Режим доступа: https://docs.gl/sl4/all. – Дата доступа: 09.11.2023.
8. Модель отражения Фонга [Электронный ресурс]. – Режим доступа: https://en.wikipedia.org/wiki/Phong_reflection_model. – Дата доступа: 09.11.2023.
9. Документация языка C++ [Электронный ресурс]. – Режим доступа: https://en.cppreference.com/w/cpp/header. – Дата доступа: 09.11.2023.
10. Библиотека GLAD [Электронный ресурс]. – Режим доступа: https://github.com/Dav1dde/glad. – Дата доступа: 09.11.2023.
11. Библиотека GLFW [Электронный ресурс]. – Режим доступа: https://github.com/glfw/glfw. – Дата доступа: 09.11.2023.