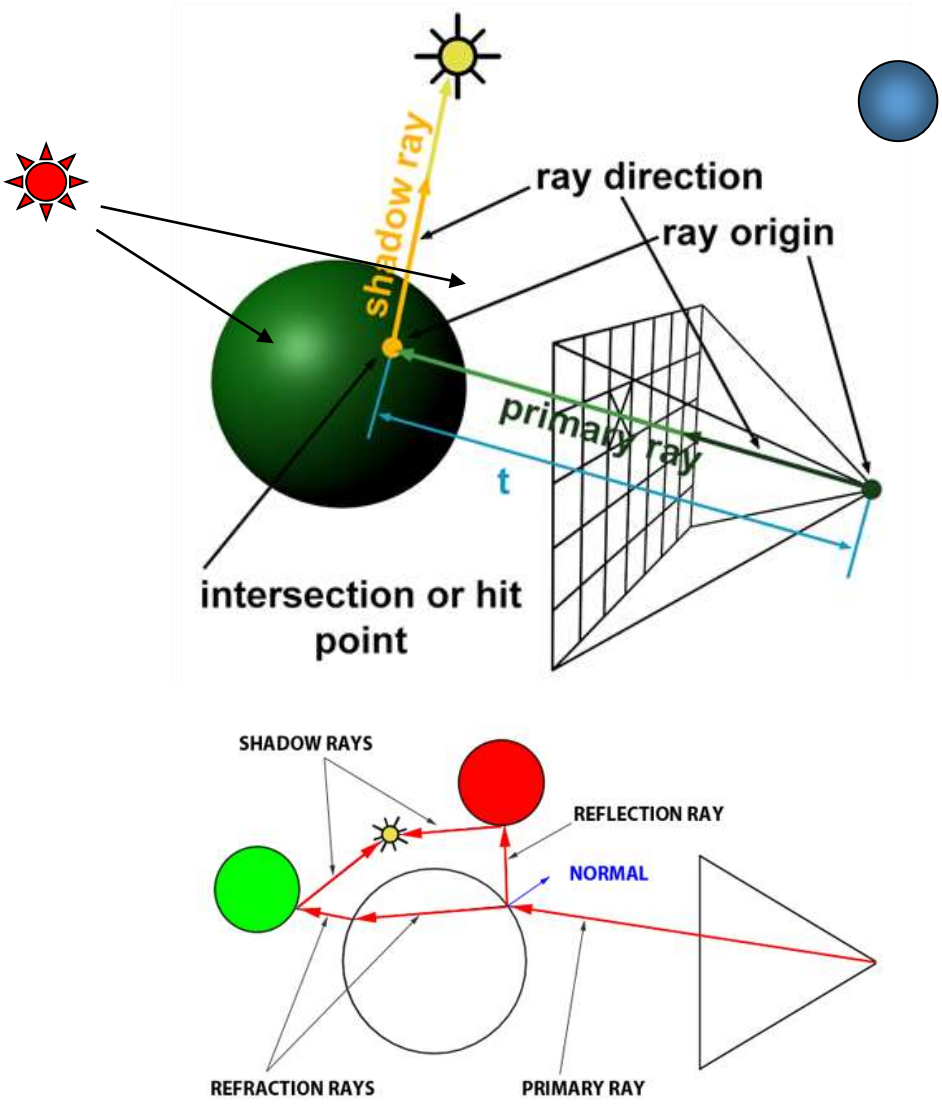
Лабораторная работа №3. «Трассировка лучей»

1. Что такое Трассировка лучей (Ray Tracing)

В контексте данной лабораторной Ray Tracing – это алгоритм построения изображения трёхмерных моделей в компьютерных программах, при которых отслеживается обратная траектория распространения луча (от экрана к источнику).

На рисунке ниже показано прохождение луча из камеры через экран и полупрозрачную сферу. При достижении лучом сферы луч раздваивается, и один из новых лучей отражается, а другой преломляется и проходит сквозь сферу. При достижении лучами диффузных объектов вычисляется цвет, цвет от обоих лучей суммируется и окрашивает цвет пикселя.



В алгоритме присутствует несколько важных вычислительных этапов:

a. Вычисление цвета для диффузной поверхности;

b. Вычисление направления отражения;
c. Вычисление направления преломления;

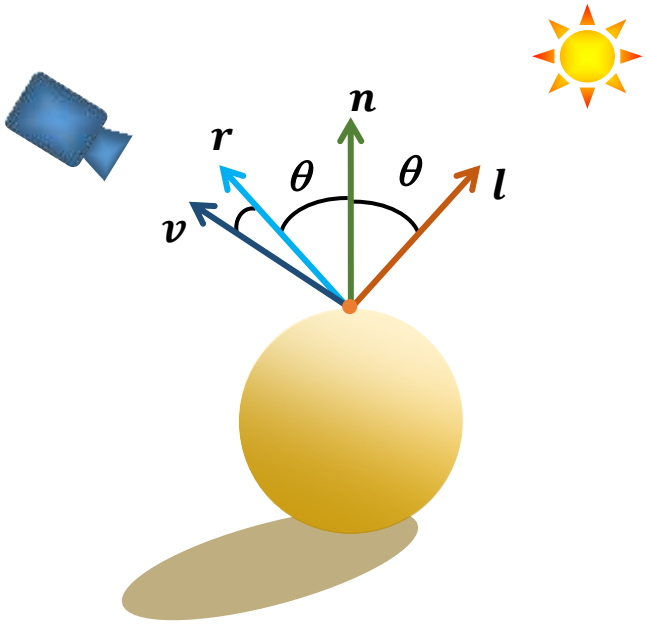
Вычисление цвета для диффузной поверхности: освещение по формуле Фонга:

$C_{out} = C \cdot k_a + C \cdot k_d \cdot max\left((\vec{n}, \vec{l}), 0\right) + L \cdot k_s \cdot \max((\vec{v}, \vec{r}), 0)^p$, где

$r = reflect(-v, n)$

C – цвет материала

L – цвет блика



Вычисление цвета точки P по формуле Фонга.

Вычисление направления отражения. Если поверхность обладает отражающими свойствами, то строится вторичный луч отражения. Направление луча определяется по закону отражения (геометрическая оптика):

$$r = i - 2 \cdot n \cdot (n \cdot i)$$

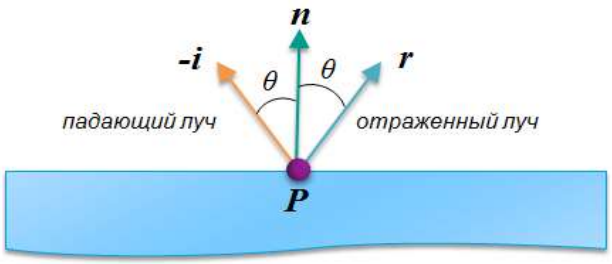


Рисунок 3. Вычисление отраженного луча.

Вычисление направления преломления. Если поверхность прозрачна, то строится еще и вторичный луч прозрачности (transparency ray). Для

определения направления луча используется закон преломления (геометрическая оптика):

$$\sin(\alpha) / \sin(\beta) = \eta_2 / \eta_1$$

$$\boldsymbol{t} = (\eta_1 / \eta_2) \cdot \boldsymbol{i} - [\, \cos(\beta) + (\eta_1 / \eta_2) \cdot (\boldsymbol{n} \cdot \boldsymbol{i}) \,] \cdot \boldsymbol{n},$$
$$\cos(\beta) = \mathrm{sqrt}[\, 1 - (\eta_1 / \eta_2)^2 \cdot (1 - (\boldsymbol{n} \cdot \boldsymbol{i})^2) \,]$$

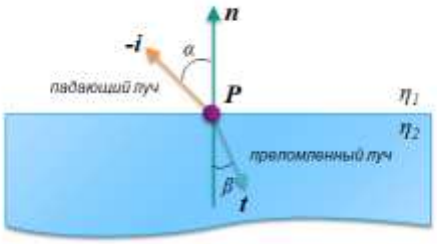


Рисунок 4. Вычисление преломленного луча.

2. О шейдерных программах

Шейдер – многозначное слово, шейдерами могут называть и программу, выполняющуюся на видеокарте, и вычислительные блоки самой видеокарты, и шейдерную программу. Шейдеры – это части шейдерной программы, которые обеспечивают исполнение определенных шагов трехмерного графического конвейера, реализуемого через посредство межплатформенного стандарта OpenGL или через MS DirectX. Они выполняются непосредственно на GPU и в параллельных потоках. Шейдерные программы предназначены для замены определенной части шагов Стандартного трехмерного графического конвейера (с фиксированной функциональностью). С версии GPU, соответствующей OpenGL 3.1, фиксированная функциональность программируемой части графического конвейера была удалена и шейдеры стали обязательными. Шейдеры для OpenGL пишутся на специализированном C-подобном языке — для стандарта OpenGL этот язык называется GLSL (для MS DirectX – HLSL). В OpenGL программа в GLSL компилируется перед использованием в команды эквивалентные операциям и загружается в GPU для исполнения.

Шейдерная программа объединяет набор шейдеров. В простейшем случае шейдерная программа состоит из двух шейдеров: вершинного и фрагментного.

Вершинный шейдер вызывается для каждой вершины. Его выходные данные интерполируются и поступают на вход фрагментного шейдера. Обычно, работа вершинного шейдера состоит в том, чтобы перевести координаты вершин из пространства сцены в пространство экрана и выполнить вспомогательные расчёты для фрагментного шейдера.

Фрагментный шейдер вызывается для каждого графического фрагмента (пикселя растеризованной геометрии, попадающего на экран). Выходом фрагментного шейдера, как правило, является цвет фрагмента, идущий в буфер цвета.

Виды шейдеров: вершинный, тесселяции, геометрический, фрагментный (рисунок 5).

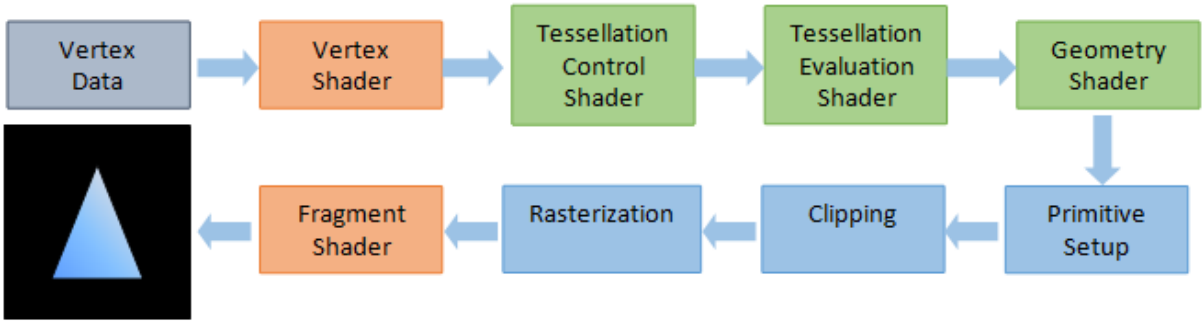


Рисунок 5. Графический конвейер OpenGL 4.3

3. Создание шейдерной программы.

В этой лабораторной работе мы будем программировать вершинный и фрагментный шейдеры. Шейдеры – это два текстовых файла. Их нужно загрузить с диска, скомпилировать и слинковать в шейдерную программу. Создайте два пустых текстовых файла «raytracing.vert» для вершинного шейдера и «raytracing.frag» – для фрагментного. По стандарту OpenGL шейдеры после загрузки компилируются основной CPU программой. Первым этапом лабораторной работы является создание, компиляция и подключение простейшей шейдерной программы.

Как и в прошлой лабораторной работе, создаём класс ShaderWidget, наследника QOpenGLWidget. Секции public и protected остаются такими же как в предыдущей лабораторной работе:

```
protected:
	void initializeGL() override;
	void resizeGL(int nWidth, int nHeight) override;
	void paintGL() override;

public:
	ShaderWidget(QWidget *parent = 0);
	~ShaderWidget();
```

В секции private появляются другие необходимые переменные, и прежде всего переменная типа QOpenGLShaderProgram.

Этот класс позволяет компилировать шейдеры и компоновать шейдерные программы, написанные на специальных языках OpenGL (GLSL) или OpenGL/ES (GLSL/ES), а также активировать шейдерную программу в текущем контексте QOpenGLContext путем вызова QOpenGLShaderProgram::bind().

QOpenGLShaderProgram содержит следующие обертки для функций OpenGL.

QOpenGLShaderProgram	OpenGL
addShaderFromSourceFile()/ addShaderFromSourceCode()	glGetShaderSource()
link()	glLinkProgram(), glGetProgramiv() (this will do

	some work if things haven't been initialized)
bind()	glUseProgram() (bind() is kind of a funny name, but it makes it consistent with the other classes)

Основная часть API предназначена для установки и получения переменных шейдера.

Другие необходимые переменные – массив вершин и целое число, в котором мы сохраним расположение этого массива в пределах списка параметров шейдерной программы.

```
private:
	QOpenGLShaderProgram m_program;

	GLfloat* vert_data;
	int vert_data_location;
```

В конструкторе ShaderWidget инициализируем массив вершин. И заполняем его координатами вершин квада (-1,-1, 0), (1,-1, 0), (1,1, 0), (-1,1, 0). В деструкторе освобождаем память, выделенную для массива.

В initializeGL() пишем код, компилирующий шейдеры и компонующий из них шейдерную программу.

```
void ShaderWidget::initializeGL()
{
	glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

	QOpenGLShader vShader(QOpenGLShader::Vertex);
	vShader.compileSourceFile("raytracing.vert");

	QOpenGLShader fShader(QOpenGLShader::Fragment);
	fShader.compileSourceFile("raytracing.frag");

	m_program.addShader(&vShader);
	m_program.addShader(&fShader);
	if (!m_program.link())
	{
		qWarning("Error link programm shader");
		return;
	}

	vert_data_location = m_program.attributeLocation("vertex");

	qDebug() << QString("Log programm");
	qDebug() << m_program.log();
}
```

Класс QOpenGLShader умеет загружать исходный код шейдера с диска компилировать его. В конструктор необходимо передать какой тип шейдера мы собираемся конструировать. Скомпилированные шейдеры нужно добавить в шейдерную программу с помощью функций addShader(). Такое разделение сделано потому, что один шейдер может быть использован в нескольких шейдерных программах.

Функция link() компонует добавленные шейдеры в шейдерную программу и возвращает true если компоновка прошла успешно. На этапе

компоновки производится стыковка входных переменных одного шейдера с выходными переменными другого, а также стыковка входных/выходных переменных шейдеров с соответствующими областями памяти в окружении OpenGL. После этого можно сказать, что шейдеры будут успешно добавлены в конвейер OpenGL.

С помощью функции attributeLocation() получаем расположение нашего массива вершин в пределах списка параметров шейдерной программы, если имя не найдется в этом списке, то функция вернет -1.

В resizeGL() остается настройка вьюпорта:

```
void ShaderWidget::resizeGL(int nWidth, int nHeight)
{
	glViewport(0, 0, nWidth, nHeight);
}
```

4. Настройка буферных объектов

Для того, чтобы начать работать (синтезировать изображение) нужно нарисовать квадрат (GL_QUAD), заполняющий окно визуализации. Можно рисовать его классическим методом, можно используя буферный объект. Рассмотрим способ с использованием буферных объектов. Использовать буферные объекты для отрисовки можно как при обычном конвейере OpenGL, так и при использовании шейдеров.

Входные переменные вершинного шейдера называются вершинными атрибутами. Существует максимальное количество вершин, которое можно передать в шейдер, такое ограничение накладывается возможностями аппаратного обеспечения. OpenGL гарантирует возможность передачи по крайней мере 16 4-х компонентных вершин, иначе говоря в шейдер можно передать как минимум 64 значения. Максимальное количество входных переменных-вершин, передаваемых в шейдер, можно узнать обратившись к атрибуту GL_MAX_VERTEX_ATTRIBS.

GLint nrAttributes;

glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);

Vertex Buffer Object

Vertex Buffer Object (VBO) — это такое средство OpenGL, позволяющее загружать вершинные атрибуты в память GPU для не оперативного режима рендеринга. VBO дали существенный прирост производительности над непосредственным режимом визуализации, в первую очередь, потому что данные находятся в памяти видеоустройства, а не в оперативной памяти.

Пример загрузки координат треугольника:

```
static const GLfloat coords[] =
{ -1.0f, -1.0f, 0.0f,  1.0f, -1.0f, 0.0f,  0.0f, 1.0f, 0.0f };

GLuint vbo;
```

```
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(coords), coords, GL_STATIC_DRAW);
```

Функция glGenBuffers() возвращает заданное количество неиспользуемых в настоящее время идентификаторов VBO. Создания буферов в этот момент не происходит.

Функция glBindBuffer() связывает (bind) полученное на предыдущем шаге имя с определенной точкой связывания (binding point). Мы будем хранить в буфере координаты вершин, поэтому должны использовать для этого точку связывания GL_ARRAY_BUFFER. Здесь снова ни о каком создании буферов речи пока не идет.

Наконец, при вызове glBufferData() происходит выделение памяти и загрузка в нее данных. Последний аргумент процедуры называется usage и задает предполагаемый сценарий использования буфера. Часть STATIC говорит о том, что мы не собираемся модифицировать данные, а часть DRAW — о том, что данные будут использованы для отрисовки. Этот аргумент не приводит к каким-то действительным ограничениям на использование буфера. Но он является подсказкой для реализации OpenGL, и может существенно влиять на производительность приложения. Так что, лучше указывать в нем что-то правдоподобное. Заметьте, что первым аргументом указывается binding point, а не конкретный VBO. То есть, выбор самого VBO был сделан на предыдущем шаге.

Когда VBO больше не нужен, его можно удалить следующим образом:

glDeleteBuffers(1, &vbo);

Vertex Arrays Object

Vertex Arrays Object (VAO) — это способ сказать OpenGL, какую часть VBO следует использовать в последующих командах. Чтобы было понятнее, представьте, что VAO представляет собой массив, в элементах которого хранится информация о том, какую часть некого VBO использовать, и как эти данные нужно интерпретировать. Таким образом, один VAO по разным индексам может хранить координаты вершин, их цвета, нормали и прочие данные. Переключившись на нужный VAO мы можем эффективно обращаться к данным, на которые он «указывает», используя только индексы.

То, что лежит в VAO по каким-то индексам, правильно называется «vertex attribute array». Название часто сокращают до «vertex attributes» или просто «attributes». Соответственно, отдельный элемент массива называется атрибутом (attribute, без s на конце). При этом каждый атрибут состоит из компонетов (components). Например, координаты вершины (один атрибут) задаются тремя координатами (три компонента типа float). Чтобы нарисовать треугольник, нужно задать три вершины, поэтому нужен массив из трех атрибутов. Усложняется ситуация тем, что авторы туториалов могут использоваться свою собственную терминологию, вроде attribute list или подобную.

Создание и удаление VAO происходит по аналогии с VBO:

```
GLuint vao;
glGenVertexArrays(1, &vao);

glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
glBindBuffer(GL_ARRAY_BUFFER, 0); // unbind VBO
glBindVertexArray(0); // unbind VAO

glDeleteVertexArrays(1, &vao);
```

Функция glVertexAttribPointer() говорит, откуда брать данные для массива атрибутов, а также в каком формате эти данные находятся. Номер массива атрибутов, переданный первым аргументом этой процедуре, еще понадобится нам чуть ниже, а также когда мы дойдем до работы с шейдерами. Второй аргумент задает размер компонента. В данном случае — три float'а на одну вершину. Третий аргумент задает тип компонента. Оставшиеся аргументы называются normalized, stride и pointer. Сейчас они нам не очень интересны.

В функции отрисовки программы должно быть:

```
glBindVertexArray(vao);
glEnableVertexAttribArray(0);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDisableVertexAttribArray(0);
```

Как видите, в нем выбирается интересующий нас VAO при помощи уже знакомой нам функции glBindVertexArray().

Вызываемая далее функция glEnableVertexAttribArray() «включает» указанный vertex attribute array. На предыдущем шаге мы решили, что у него будет номер ноль, поэтому здесь используем тот же номер. Как можно без труда догадаться, функция glDisableVertexAttribArray() совершает обратное действие. Нужно это включение и выключение для оптимизации. Отключая ненужные атрибуты, мы не передаем лишних данных шейдерам (о них ниже). По умолчанию все атрибуты находятся в выключенном состоянии.

Наконец, glDrawArrays() отвечает за рисование примитивов. Первым аргументом передается тип примитива. В данном случае рисуется треугольник. Второй аргумент определяет, с вершины под каким номером нужно начать. Последний аргумент задает количество вершин, которое следует использовать.

```
#version 330 core

layout(location = 0) in vec3 vertexPos;
```

```
void main() {
  gl_Position.xyz = vertexPos;
  gl_Position.w = 1.0;
}
```

В части location = 0 используется тот самый номер, который мы передавали первым аргументом при вызове glVertexAttribPointer. То есть, это номер массива атрибутов, в котором мы решили хранить вершины примитива. Шейдер говорит, что входную вершину он будет брать оттуда. Соответственно, атрибуты должны быть включены вызовом glEnableVertexAttribArray.

Операции с буферами.

Основной принцип: OpenGLShaderProgram оборачивает функции ‘glGet<buffer type>Location(), и gl<buffer type><type>() в одну.

QOpenGLShaderProgram	OpenGL
VBO	
bindAttributeLocation()	glBindAttributeLocation()
setAttributeValue()	glVertexAttrib<type>()
setAttributeArray()	glVertexAttribPointer()
read()	glGetBufferSubData(). (reads *output* data from the vertex shader), although some of the other functions aren’t wrapped in the class?
VAO	
setAttributeBuffer()	glVertexAttribPointer()
setVertexAttribPointer()	glVertexAttribPointer()
enableAttributeArray()	glEnableVertexAttribArray()

Other stuff

Some things aren’t wrapped in a class, but are “directly” accessible in QOpenGLFunctions. And still other things aren’t even wrapped in QOpenGLFunctions. I’ll assume that the Qt OpenGL API implementors did that for a reason and if you call those, you really have to know what you’re doing.

Common QOpenGLFunction calls

glDrawElements, glDrawArrays, glDrawArraysInstanced, glDrawElementsInstanced

glVertexAttribFormat() [and related]

```
void ShaderWidget::paintGL()
{
	glClear(GL_COLOR_BUFFER_BIT);

	if (!m_program.bind())
		return;

	m_program.enableAttributeArray(vert_data_location);
	m_program.setAttributeArray(vert_data_location, vert_data, 3);

	glDrawArrays(GL_QUADS, 0, 4);

	m_program.disableAttributeArray(vert_data_location);

	m_program.release();
}
```

Вершинный шейдер

Вершинный шейдер может быть самым простым – перекладывать интерполированные координаты вершин в выходную переменную, и тогда генерировать луч надо во фрагментном шейдере, а может быть более сложным – с генерацией направления луча.

```
#version 430

//Входные переменные vertex - позиция вершины
in vec3 vertex;
out vec3 interpolated_vertex;

void main (void)
{
    gl_Position = vec4(vertex, 1.0);
    interpolated_vertex = vertex;
}
```

Переменные, отдаваемые вершинным шейдером дальше по конвейеру объявлены со спецификатором out. Венршинный шейдер больше изменяться не будет.

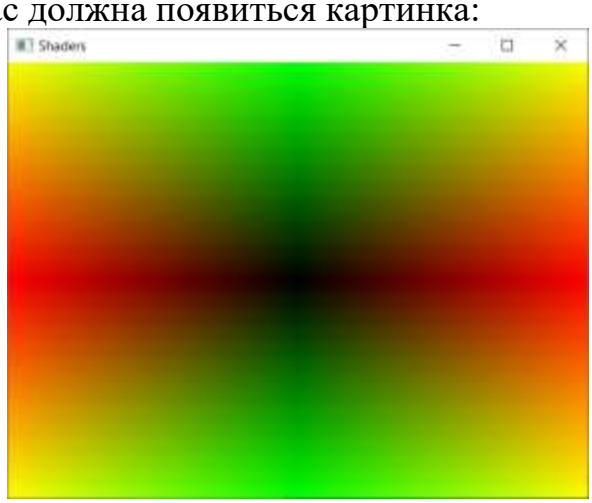
Фрагментный шейдер

Простейший фрагментный шейдер для того, чтобы запустить шейдерную программу может состоять в заполнении единственной выходной переменной FragColor – пиксель экрана.

```
#version 430

in vec3 interpolated_vertex;
out vec4 FragColor;

void main ( void )
{
	FragColor = vec4 ( abs(interpolated_vertex.xy), 0, 1.0);
}
```

У этого шейдера единственная выходная переменная: FragColor. Если выходная переменная одна, значит она соответствует пикселю в буфере экрана. Единственная входная переменная соответствует выходной переменной вершинного шейдера. Функция main записывает интерполированные координаты в выходной буфер цвета. Функция abs (модуль) применяется потому, что компонента цвета не может быть отрицательной, а наши интерполированные значения лежат в диапазоне от -1 до 1.

После запуска у вас должна появиться картинка:



Левый нижний угол соответствует координатам (-1,-1), правый верхний – (1,1).

Если в коде шейдеров содержится ошибка и компилятор не смог его скомпилировать, то после запуска программы у вас будет пустой экран, а в окно вывода напечатается лог с указанием ошибки компиляции. Например,

```
0(8) : error C1068: too much data in type constructor
```

Это значит, что в восьмой строке был передан лишний параметр в конструктор.

5. Генерация первичного луча

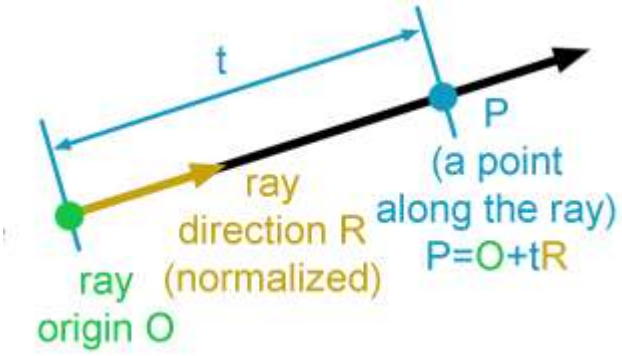
http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays

Для моделирования наблюдателя предназначена камера. В обратной трассировке лучей (Ray Tracing) через каждый пиксель окна изображения должен быть выпущен луч в сцену. Обозначим новый раздел “DATA STRUCTURES” и создадим две структуры для камеры и для луча:

```
/**********DATA STRUCTURES**********/

struct Camera
{
	vec3 position;
	vec3 view;
	vec3 up;
	vec3 side;
```

```
};

struct Ray
{
	vec3 origin;
	vec3 direction;
};
```

Первичный луч - луч, который исходит из камеры. Чтобы правильно вычислить луч для каждого пикселя, нужно вычислить его начало и его направление. Координаты всех точек на луче $r = o + dt, t \epsilon [0, \infty)$.



Добавляем функцию генерации луча:

```
Ray GenerateRay(Camera camera)
{
	vec2 coords = interpolated_vertex.xy * normalize(scale);
	vec3 direction = camera.view + camera.side * coords.x + camera.up * coords.y;
	return Ray(camera.position, normalize(direction));
}
```

Передавать координаты камеры будем через uniform переменную. Для этого создадим переменную camera в фрагментном шейдере. И uniform переменную, отвечающую за соотношение сторон исходного окна. Умножение на scale необходимо, чтобы изображение не деформировалось при изменении размеров окна (см. рисунок).

```
uniform Camera camera;
// отношение сторон выходного изображения
uniform vec2 scale;
```

Изменяем main()

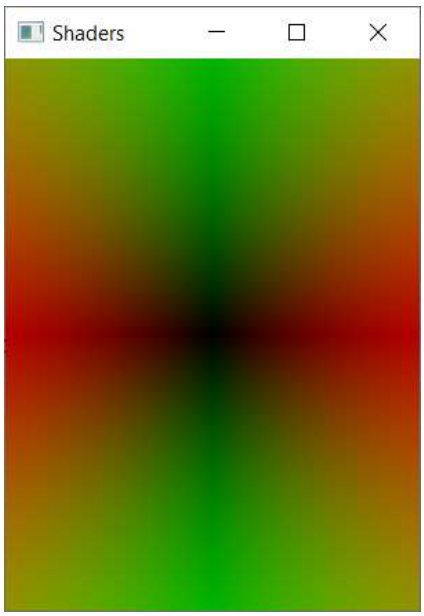
```
void main ( void )
{
	Ray ray = GenerateRay(camera);
	frag_color = vec4 (abs(ray.direction.xy), 0, 1.0);
}
```

Задавать начальные значения камеры имеет смысл в функции initializeGL:

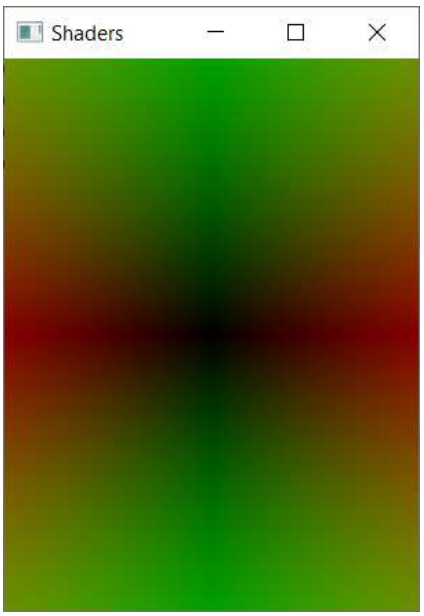
```
if (!m_program.bind())
```

```
        qWarning("Error bind programm shader");

m_program.setUniformValue("camera.position", QVector3D(0.0, 0.0, -10));
m_program.setUniformValue("camera.view",     QVector3D(0.0, 0.0, 1.0));
m_program.setUniformValue("camera.up",       QVector3D(0.0, 1.0, 0.0));
m_program.setUniformValue("camera.side",     QVector3D(1.0, 0.0, 0.0));

m_program.setUniformValue("scale", QVector2D(width(), height()));

m_program.release();
```

Перед передачей uniform переменной в шейдер обязательно необходимо сначала сделать нужную шейдерную программу текущей. Аналогичную строчку, устанавливающую scale прописываем в resizeGL().



Без умножения на scale.



После умножения на scale.

6. Добавление структур данных сцены и источников света.

Объявление типов данных

Большая часть кода будет написана в фрагментном шейдере. Для нашей программы понадобятся структуры для следующих объектов: камера, источник света, луч, пересечение, сфера, треугольник, материал.

После того, как появилась функция, генерирующая луч, можно начать этот луч пересекать с геометрией. Для геометрии тоже введем структуры.

Объекты

Объекты сцены можно задать в самом фрагментном шейдере, а можно передать с хоста во фрагментный шейдер с помощью ещё одного типа буферных блоков – shader storage buffer objects (SSBO).

Эти хранилища обладают двумя важными свойствами: 1) шейдер может как читать из них, так и писать в них, изменяя их содержимое; 2) их размер может быть установлен непосредственно перед рендерингом, а не во время компиляции или компоновки. Переменные буфера объявляются в шейдерах путем помещения их в блок интерфейса, который, в свою очередь, объявляется с использованием ключевого слова buffer.

Например, в шейдере вот такой код:

```
// create a read-writeable buffer
buffer BufferObject
{
	// preamble members
	int mode;
	// last member can be unsized array
	vec4 points[];
};
```

Если в массиве points выше не указан размер в шейдере, то его размер может быть установлен приложением до рендеринга, после компиляции и компоновки. Шейдер может вызвать у массива метод length(), чтобы найти размер во время выполнения рендеринга.

О спецификаторах памяти (например, coherent) и атомарных операциях, применяемых к буферным блокам, можно прочитать главе 11 «Память» спецификации OpenGL 4.3 [ссылка на спецификацию].

Из спецификации: Если вам не нужно записывать в буфер, используйте унифицированный блок, так как ваше устройство может иметь меньше доступных ресурсов для блоков буфера, чем для унифицированных блоков. ???

Чтобы передать массив сфер с хоста в шейдер необходимо и в шейдере, и в программе C++ создать структуру, описывающую сферу.

C++

```
struct Sphere
{
	QVector3D position;
	float radius;
	QVector3D color;
	int material_idx;
};
```

GLSL

```
struct Sphere
{
	vec3 center;
	float radius;
	vec3 color;
	int material_idx;
};
```

Создайте на хосте массив сфер all_spheres, которые вы хотите отрисовать, обратите внимание на расположение сфер относительно положения наблюдателя (камеры).

Для того, чтобы передать массив на видеокарту воспользуемся SSBO, который является доступной функциональностью OpenGL с версии 4.3. Создаём экземпляр класса QOpenGLFunctions_4_3_Core для того, чтобы получить доступ к нужным функциям.

```
QOpenGLFunctions_4_3_Core * functions = QOpenGLContext::currentContext()->
versionFunctions<QOpenGLFunctions_4_3_Core>();
GLuint ssbo = 0;
functions->glGenBuffers(1, &ssbo);
functions->glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
functions->glBufferData(GL_SHADER_STORAGE_BUFFER, size * sizeof(Sphere), all_spheres,
GL_DYNAMIC_COPY);
// Now bind the buffer to the zeroth GL_SHADER_STORAGE_BUFFER binding point
functions->glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, ssbo);
```

Аналогично работе с текстурами, сначала берем незанятый идентификатор буфера, связываем его с псевдонимом GL_SHADER_STORAGE_BUFFER, дальше настраиваем параметры через псевдоним.
Важным параметром для glBufferData() является параметр *usage*. Тип использования должен быть одним из стандартных токенов, таких как GL_STATIC_DRAW или GL_DYNAMIC_COPY. Обратите внимание, что имя токена состоит из двух частей: первая - STATIC, DYNAMIC или STREAM, а вторая - DRAW, READ или COPY.

Модификатор	Содержимое хранилища данных
STATIC_	будет изменено один раз и использовано много раз.
DYNAMIC_	будет неоднократно изменяться и использоваться много раз.
STREAM__	будет изменено один раз и использовано много-много раз.
_DRAW	изменяется приложением и используется в качестве источника для команд рисования OpenGL и спецификации изображения.
_READ	изменяется путем чтения данных из OpenGL и используется для возврата этих данных при запросе приложением.
_COPY	изменяется путем чтения данных из OpenGL и используется в качестве источника для команд рисования OpenGL и спецификации изображения.

Точная спецификация параметра *usage* важна для достижения оптимальной производительности. Этот параметр передает информацию OpenGL о том, как вы планируете использовать буфер. В случае GL_STREAM OpenGL может даже не копировать ваши данные в быструю графическую память, если он может получить к ним доступ на месте. Этот модификатор следует использовать для таких приложений, как физическое моделирование, запущенное на ЦП, где новый набор данных представлен в каждом кадре.

После передачи параметров в шейдер массив можно удалить.

На стороне шейдеров объявляем буфер:

```
layout(std430, binding = 0) buffer SphereBuffer
{
    Sphere sphere_data[];
};
```

Квалификатор, std430, указывает, что схема памяти блока должна соответствовать стандарту std430. Layout std430 описан в Приложении I, «Схемы буферизованных объектов», и аналогичен макету std140, используемому для однородных блоков, но немного экономичнее с использованием памяти.

Второй квалификатор binding = 0 указывает, что блок должен быть связан с привязкой GL_SHADER_STORAGE_BUFFER с нулевым индексом. Объявление блока интерфейса с помощью ключевого слова buffer указывает на то, что блок должен храниться в памяти и поддерживаться буферным объектом.

Обратите внимание, что буферы подвергаются выравниванию, прочитать подробнее можно в спецификации OpenGL Приложение I [ссылка на спецификацию].

Теперь во всём фрагментном шейдере можно обращаться к массиву sphere_data. Длину массива можно взять с помощью функции sphere_data.length().

Материал и освещение

В структурах есть переменные material_idx, которые будут содержать индекс в массиве материалов. Пока можно задать все индексы нулевыми и использовать один материал по умолчанию. Структура материала будет подробно рассмотрена в разделе «Освещение».

```
struct Material
{
	float ambient;
	float diffuse;
	float specular;
	float specular_power;
};
```

Далее зададим материал «по умолчанию». И позицию источника освещения.

```
Material material = {0.4, 0.9, 0.0, 512.0};
vec3 light_pos = vec3(1, 0, -8);
```

7. Пересечение луча с объектами

Для того, чтобы отрисовать сцену, необходимо реализовать пересечение луча с объектами сцены - треугольниками и сферами.

Пересечение луча со сферой

Есть несколько алгоритмов для пересечения луча со сферой, здесь воспользуемся аналитическим решением:

Уравнение луча: $r = o + d \cdot t$

Уравнение сферы: $x^2 + y^2 + z^2 = R$ или $P^2 - R^2 = 0$ Где x, y и z - координаты декартовой точки и R – это радиус сферы, *центрированной в начале координат*.

Подставляем $(o + d \cdot t)^2 - R^2 = 0$

Раскрываем скобки и получаем квадратное уравнение относительно t:

$$d^2t^2 + 2o \cdot d \cdot t + o^2 - R^2 = 0$$

Т.к. d – единичный вектор, то уравнение упрощается:

$$t^2 + 2o \cdot d \cdot t + o^2 - R^2 = 0$$

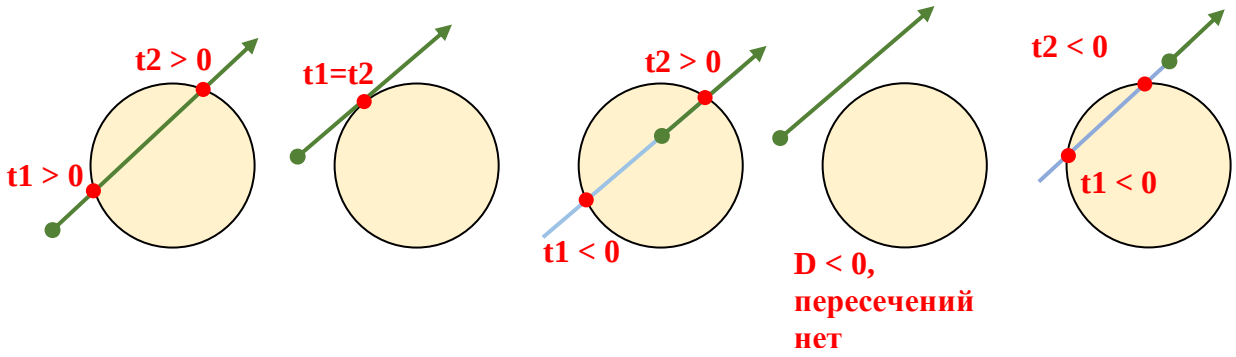
Корни уравнения:

$$t_{1,2} = \frac{-b \mp \sqrt{b^2 - 4*c}}{2} = \frac{-2(o,d) \mp \sqrt{4(o,d)^2 - 4*(o^2 - R^2)}}{2} = -(o,d) \pm \sqrt{(o,d)^2 - (o^2 - R^2)}$$

Если D > 0, то луч пересекает сферу в двух местах, нам нужно ближайшее пересечение, при $t > 0$. Т.к. если $t < 0$, значит луч пересекает сферу до его начала.

Если D = 0, то точка пересечения (касания) одна.

Если D < 0, то точек пересечения нет.



Прежде чем реализовать этот алгоритм необходимо решить проблему, когда сфера не центрирована в начале координат. Перепишем уравнение:

$$(P - Sc)^2 - R^2 = 0$$

где Sc - местоположение центра сферы в трехмерном пространстве. Снова подставляем уравнение луча:

$$(o + d \cdot t - Sc)^2 - R^2 = 0$$

Раскрываем скобки:

$$d^2t^2 + 2(o - Sc)dt + (o - Sc)^2 - R^2 = 0$$

Другими словами, это означает, что мы можем вычесть из начала луча центр сферы и проверить этот преобразованный луч на сфере, как если бы она был центрирована в начале координат.

Реализация функции IntersectSphere() представлена ниже:

```
bool IntersectSphere(Sphere sphere, Ray ray, out float time)
{
	ray.origin -= sphere.center;
	float B = dot(ray.direction, ray.origin);
	float C = dot(ray.origin, ray.origin) - sphere.radius * sphere.radius;
	float D = B * B - C;

	if (D > 0.0)
	{
```

```
		D = sqrt(D);

		float t1 = -B - D;
		float t2 = -B + D;

		float min_t = min(t1, t2);
		float max_t = max(t1, t2);

		if (max_t < 0)
			return false;

		if (min_t < 0)
		{
			time = max_t;
			return true;
		}
		time = min_t;
		return true;
	}
	return false;
}
```

Функция Intersect() пересекает луч со всеми примитивами сцены и возвращает ближайшее пересечение.

Создаём структуру для хранения пересечения. В структуре предусмотрены поля для хранения цвета и материала текущей точки пересечения.

```
struct Intersection
{
	float time;
	vec3 point;
	vec3 normal;
	vec3 color;
	int material_idx;
};
```

Также задаём какое-то очень большое число, означающее выход луча за пределы сцены:

```
#define BIG  1000000.0
```

Создаём функцию, пересекающую луч со всей геометрией сцены.

```
bool Intersect(Ray ray, float start, float final, inout Intersection intersect)
{
	bool result = false;
	float time = start;
	intersect.time = final;

	for(int i = 0; i < sphere_data.length(); i++)
	{
		if (IntersectSphere(sphere_data[i], ray, time) && time < intersect.time)
		{
			intersect.time = time;
			intersect.point = ray.origin + ray.direction * time;
			intersect.normal = normalize(intersect.point - sphere_data[i].center);
```

```
                intersect.color = sphere_data[i].color;
                intersect.material_idx = sphere_data[i].material_idx;
                result = true;
            }
        }
        return result;
}
```

Нормаль точки на сфере может быть просто вычислена как позиция точки минус центр сферы (не забудьте нормализовать результирующий вектор):

$$N = ||P - Sc||$$

Функция Raytrace() трассирует луч:

```
vec4 Raytrace(Ray primary_ray)
{
        vec4  resultColor = vec4(0, 0, 0, 0);
        Ray ray = primary_ray;

        Intersection intersect;
        intersect.time = BIG;
        float start = 0;
        float final = BIG;

        if (Intersect(ray, start, final, intersect))
        {
            resultColor = vec4(1, 0, 0, 0);
        }
        return resultColor;
}
```

Модифицируйте функцию main следующим образом:

```
void main ( void )
{
        // ---------- trace ------------//
        Ray ray = GenerateRay(camera);
        frag_color = Raytrace(ray);
}
```

Теперь на экране должны появиться красные силуэты геометрии сцены.

8. Настройка освещения

Чтобы объекты имели собственную тень и отбрасывали падающую в нашу модель необходимо добавить источник света. Источник света может быть сложным – иметь свой собственный цвет свечения и ограничения по направлению лучей (прожектор), самый простой источник освещения – точечный источник белого света. Для его моделирования нам необходимо задать только его позицию в сцене:

```
        vec3 light_pos = vec3(1, 0, -8);
```

Можно выбрать любую из моделей освещения: Ламберт, Фонг, Блинн, Кук-Торренс. Мы будем реализовывать шейдинг (закрашивание) по Фонгу. Это локальная модель освещения, т.е. она учитывает только свойства заданной точки и источников освещения, игнорируя эффекты рассеивания, линзирования, отражения от соседних тел. Расчёт освещения по Фонгу требует вычисления цветовой интенсивности трёх компонент освещения: фоновой (ambient), рассеянной (diffuse) и глянцевых бликов (specular). Фоновая компонента — грубое приближение лучей света, рассеянных соседними объектами и затем достигших заданной точки; остальные две компоненты имитируют рассеивание и зеркальное отражение прямого излучения.

$I = k_a I_a + k_d(\vec{n}, \vec{l}) + k_s \cdot \max((\vec{v}, \vec{r}), 0)^p$, где

$\vec{n} -$ вектор нормали к поверхности в точке

$\vec{l} -$ направление на источник света

$\vec{v} -$ направление на наблюдателя

$k_a -$ коэффициент фонового освещения

$k_d -$ коэффициент диффузного освещения

$k_s -$ коэффициент зеркального отражения, $p -$ коэффициент резкости бликов

$$r = reflect(-v, n)$$

Для расчета кроме цвета предмета нам потребуются коэффициенты k_a, k_d, k_s, и p.

Вспомним базовую структуру, заданную для хранения коэффициентов материала $k_a - ambient$, $k_d - diffuse$, $k_s - specular$, и $p - specular\_power$. Цвет геометрии передается вместе с геометрией (поле sphere.color). При реализации зеркального отражения и преломления в структуру добавим поля, отвечающие за эти свойства материалов.

```
struct Material
{
	float ambient;
	float diffuse;
	float specular;
	float specular_power;
};
```

Теперь напишем функцию Phong.

```
vec3 Phong(Intersection intersect, vec3 pos_light, float shadow)
{
	vec3  light = normalize(pos_light - intersect.point);
	float diffuse = max(dot(light, intersect.normal), 0.0);
	vec3  view = normalize(camera.position - intersect.point);
	vec3  reflected = reflect(view, -intersect.normal);
	float specular = pow(max(dot(reflected, light), 0.0), material.specular_power);

	return material.ambient * intersect.color +
		shadow *
	       (material.diffuse * diffuse * intersect.color +
		material.specular * specular * vec3(1,1,1));
}
```

Умножение на вектор из единиц означает умножение на белый цвет источника, на цвет освещения, а не объекта умножается потому, что блик – это отражение источника света.

Чтобы смоделировать падающие тени необходимо выпустить, так называемые, теневые лучи. Из каждой точки пересечения, для которой рассчитываем освещение выпускается луч на источник света, если отрезок этого луча между нашей точкой пересечения и источником света пересекается каким-нибудь объектом сцены, значит точка находится в тени и она освещена только ambient компонентой.

```
float Shadow(vec3 pos_light, vec3 point)
{
	float light = 1.0; // Point is lighted

	vec3 direction = normalize(pos_light - point);
	Ray shadow_ray = Ray(point + direction * EPSILON, direction);

	Intersection intersect;
	intersect.time = distance(pos_light, point);

	if (Intersect(shadow_ray, 0, intersect.time, intersect))
	{
		// this light source is invisible in the intercection point
		light = 0.0;
	}
	return light;
}
```

Обратите внимание, что для вычисления пересечения используется та же функция Intersect().

Этот вычисленный коэффициент необходимо передать параметром в функцию Phong и изменить вычисление цвета в функции Raytrace() следующим образом:

```
if (Intersect(ray, start, final, intersect))
{
	float shadowing = Shadow(light_pos, intersect.point);
	resultColor += vec4(Phong(intersect, light_pos, shadowing), 0);
}
```

На этом этапе синтезированное изображение должно быть похоже на рисунок: