ОГЛАВЛЕНИЕ

СОВЕТЫ ПЕРЕД НАЧАЛОМ

Для лабораторных работ рекомендуется использовать VisualStudio 2010 или более поздние версии.

Для подсветки синтаксиса GLSL рекомендуем использовать расширение для студии NShader (http://www.horsedrawngames.com/shader-syntax-highlighting/)

В качестве дополнительного материала могут послужить сайты:

1. http://www.scratchapixel.com/
2. http://ray-tracing.ru
3. http://www2.sscc.ru/Seminars/Docum/NVIDIA%20CUDA-2012/Lec_6_Novsb.pdf

Версия GLSL, используемая в примере 4.3. Скачать спецификацию можно на сайте www.opengl.org.

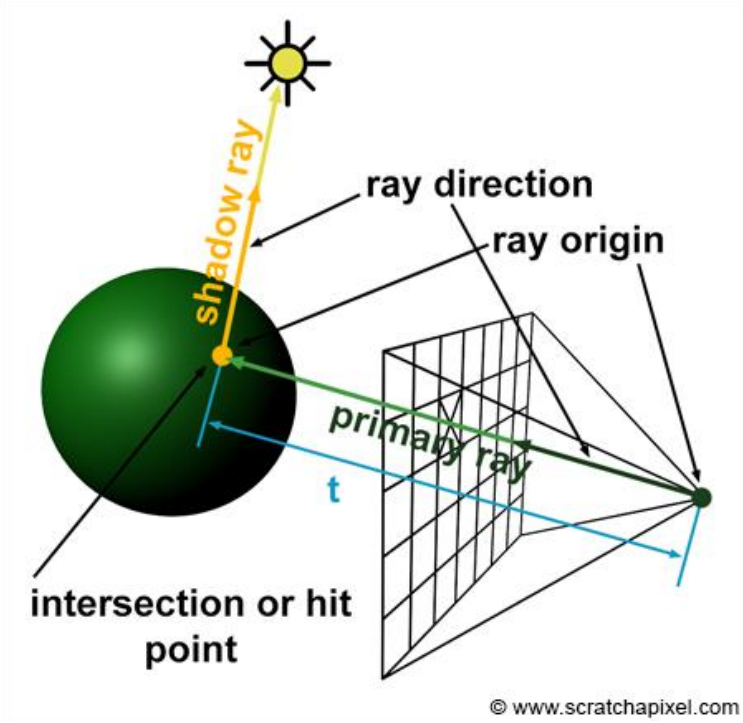
ВВОДНАЯ ЧАСТЬ

1. ЧТО ТАКОЕ 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 1 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.



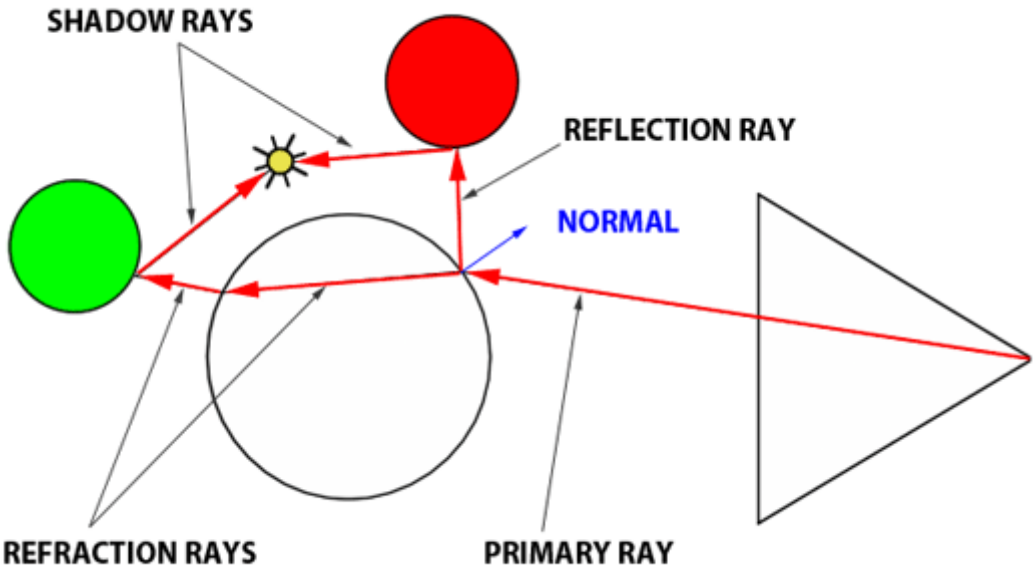


Рисунок 1. Распространение одного луча в сцене.

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

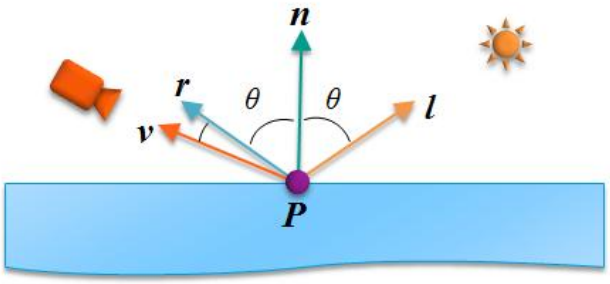


Рисунок 2. Вычисление цвета точки P по формуле Фонга.

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

$$\boldsymbol{r} = \boldsymbol{i} - 2 \cdot \boldsymbol{n} \cdot (\boldsymbol{n} \cdot \boldsymbol{i})$$

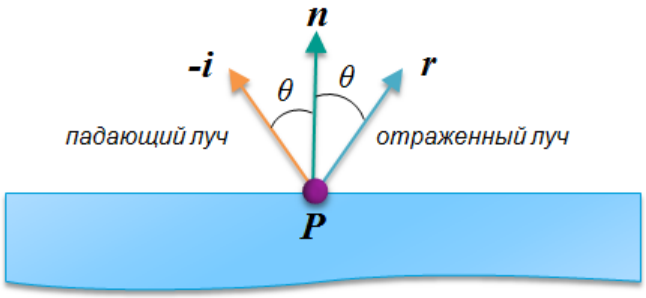


Рисунок 3. Вычисление отраженного луча.

Вычисление направления преломления. Если поверхность прозрачна, то строится еще и вторичный луч прозрачности (transparency ray). Для определения направления луча используется закон преломления (геометрическая оптика):

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

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

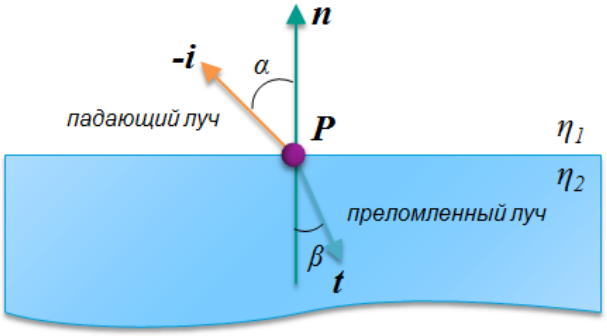


Рисунок 4. Вычисление преломленного луча.

2. О ШЕЙДЕРНЫХ ПРОГРАММАХ

Шейдеры – это компонент шейдерной программы, мини-программа, которая выполняется в рамках отдельного этапа в общем конвейере OpenGL, выполняются непосредственно на GPU и действуют параллельно. Шейдерные программы предназначены для замены части архитектуры OpenGL, которую называют конвейером с фиксированной функциональностью. С версии OpenGL 3.1 фиксированная функциональность была удалена и шейдеры стали обязательными. Шейдер — специальная подпрограмма, выполняемая на GPU. Шейдеры для OpenGL пишутся на специализированном C-подобном языке — GLSL. Они компилируются самим OpenGL перед использованием.

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. На фрагментный шейдер обычно ложится основная часть расчёта освещения.

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

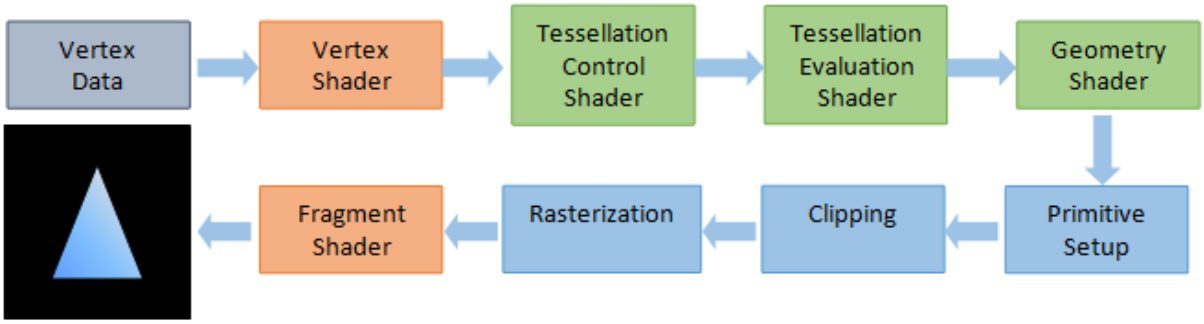


Рисунок 5. Графический конвейер OpenGL 4.3

ПРАКИЧЕСКАЯ ЧАСТЬ

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

1. ПРОСТЕЙШАЯ ШЕЙДЕРНАЯ ПРОГРАММА.

Для более упорядоченного вида программы функции, связанные с OpenGL и шейдерами, можно вынести в отдельный класс View, как это было сделано в предыдущей лабораторной работе.

ЗАГРУЗКА ШЕЙДЕРОВ

```
void loadShader(String filename, ShaderType type, uint program, out uint address)
{

    address = GL.CreateShader(type);
    using (System.IO.StreamReader sr = new StreamReader(filename))
    {
        GL.ShaderSource(address, sr.ReadToEnd());
    }
    GL.CompileShader(address);
    GL.AttachShader(program, address);
    Console.WriteLine(GL.GetShaderInfoLog(address));
}
```

glCreateShader создаёт объект шейдера, её аргумент определяет тип шейдера, возвращает значение, которое можно использовать для ссылки на объект шейдера (дескриптор, то есть в нашем случае это идентификаторы uint VertexShader и uint FragmentShader, которые в данную функцию передаются через аргумент address).

glShaderSource загружает исходный код в созданный шейдерный объект.

Далее надо скомпилировать исходный шейдер, делается это вызовом функции glCompileShader() и передачей ей дескриптора шейдера, который требуется скомпилировать.

Перед тем, как шейдеры будут добавлены в конвейер OpenGL их нужно скомпоновать в шейдерную программу с помощью функции glAttachShader(). 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.

ИНИЦИАЛИЗАЦИЯ ШЕЙДЕРНОЙ ПРОГРАММЫ

В функции `InitShaders()` теперь необходимо создать объект шейдерной программы и вызвать ранее реализованную функцию `loadShader()`, чтобы создать объекты шейдеров, скомпилировать их и скомпоновать в объекте шейдерной программы:

```
BasicProgramID = GL.CreateProgram();  // создание объекта программы
loadShader("..\\..\\raytracing.vert", ShaderType.VertexShader,   BasicProgramID,
             out BasicVertexShader);
loadShader("..\\..\\raytracing.frag", ShaderType.FragmentShader, BasicProgramID,
             out BasicFragmentShader);
GL.LinkProgram(BasicProgramID);
// Проверяем успех компоновки
int status = 0;
GL.GetProgram(BasicProgramID, GetProgramParameterName.LinkStatus, out status);
Console.WriteLine(GL.GetProgramInfoLog(BasicProgramID));
```

НАСТРОЙКА БУФЕРНЫХ ОБЪЕКТОВ

Для того, чтобы что-то нарисовать нужно нарисовать квадрат (GL_QUAD), заполняющий весь экран. Можно рисовать его классическим методом, можно используя буферный объект. Код для буферного объекта ниже.

Сначала создайте член класса `int vbo_position` для хранения дескриптора объекта массива вершин и массив вершин.

```
vertdata = new Vector3[] {
new Vector3(-1f, -1f, 0f),
new Vector3( 1f, -1f, 0f),
new Vector3( 1f,  1f, 0f),
new Vector3(-1f,  1f, 0f) };

GL.GenBuffers(1, out vbo_position);

GL.BindBuffer(BufferTarget.ArrayBuffer, vbo_position);

GL.BufferData<Vector3>(BufferTarget.ArrayBuffer, (IntPtr)(vertdata.Length *
                       Vector3.SizeInBytes), vertdata, BufferUsageHint.StaticDraw);
GL.VertexAttribPointer(attribute_vpos,  3,  VertexAttribPointerType.Float,  false,  0,
0);
```

```
GL.Uniform3(uniform_pos, campos);
GL.Uniform1(uniform_aspect, aspect);

GL.UseProgram(BasicProgramID);

GL.BindBuffer(BufferTarget.ArrayBuffer, 0);
```

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.

```
in vec3 vPosition; //Входные переменные vPosition - позиция вершины
out vec3 glPosition;

void main (void)
{
    gl_Position = vec4(vPosition, 1.0);
    glPosition = vPosition;
}
```

Переменные, отдаваемые вершинным шейдером дальше по конвейеру объявлены со спецификатором out.

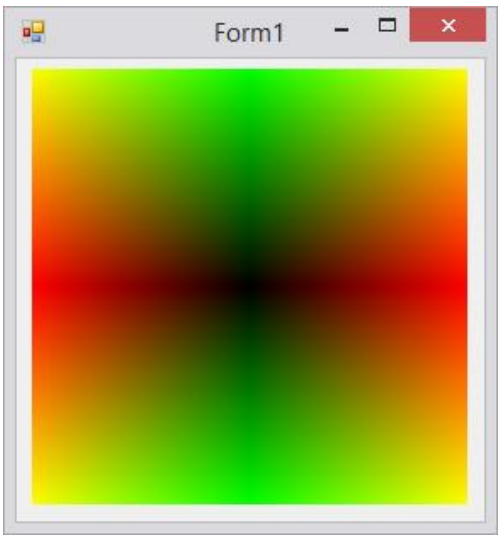
ФРАГМЕНТНЫЙ ШЕЙДЕР

Во фрагментном шейдере будет написан весь содержательный код. Разобьем эту огромную работу на этапы. И начнем с простейшего фрагментного шейдера.

```
#version 430

out vec4 FragColor;
in vec3 glPosition;

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

У этого шейдера единственная выходная переменная: FragColor. Система сама, что если выходная переменная одна, значит она соответствует пикселу 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:

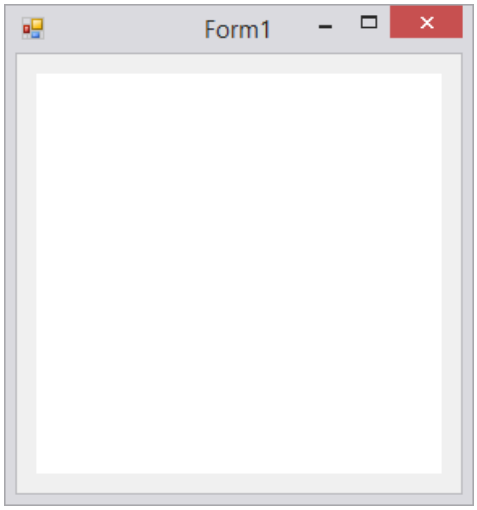


Левый нижний угол соответствует координатам (-1,-1), правый верхний – (1,1).

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.



2. ГЕНЕРАЦИЯ ПЕРВИЧНОГО ЛУЧА

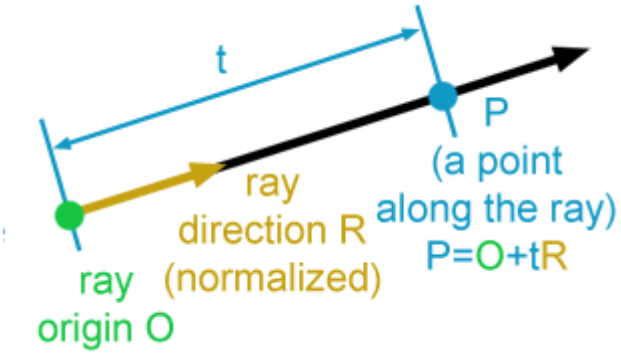
http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays

Для моделирования наблюдателя у нас есть камера. В обратном рейтрейсинге через каждый пиксель выходного изображения должен быть выпущен луч в сцену. Обозначим новый раздел “DATA STRUCTURES” и создадим две структуры для камеры и для луча:

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

```
struct SCamera
{
    vec3 Position;
    vec3 View;
    vec3 Up;
    vec3 Side;
    // отношение сторон выходного изображения
    vec2 Scale;
};

struct SRay
{
    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:

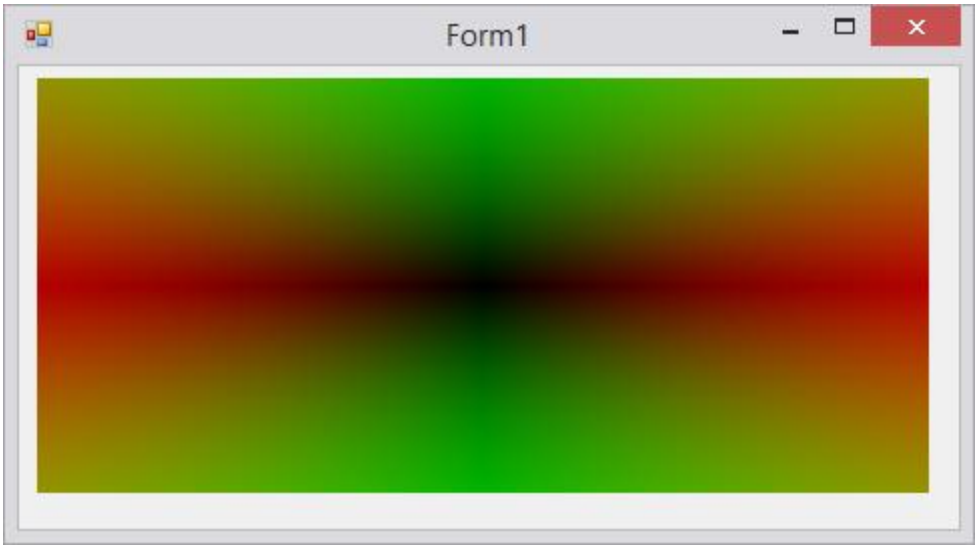
```
SRay GenerateRay ( SCamera uCamera )
{
      vec2 coords = glPosition.xy * uCamera.Scale;
      vec3 direction = uCamera.View + uCamera.Side * coords.x + uCamera.Up *
coords.y;
      return SRay ( uCamera.Position, normalize(direction) );
}

SCamera initializeDefaultCamera()
{
    //** CAMERA **//
    camera.Position = vec3(0.0, 0.0, -8.0);
    camera.View = vec3(0.0, 0.0, 1.0);
    camera.Up = vec3(0.0, 1.0, 0.0);
    camera.Side = vec3(1.0, 0.0, 0.0);
    camera.Scale = vec2(1.0);
}
```

Изменяем main()

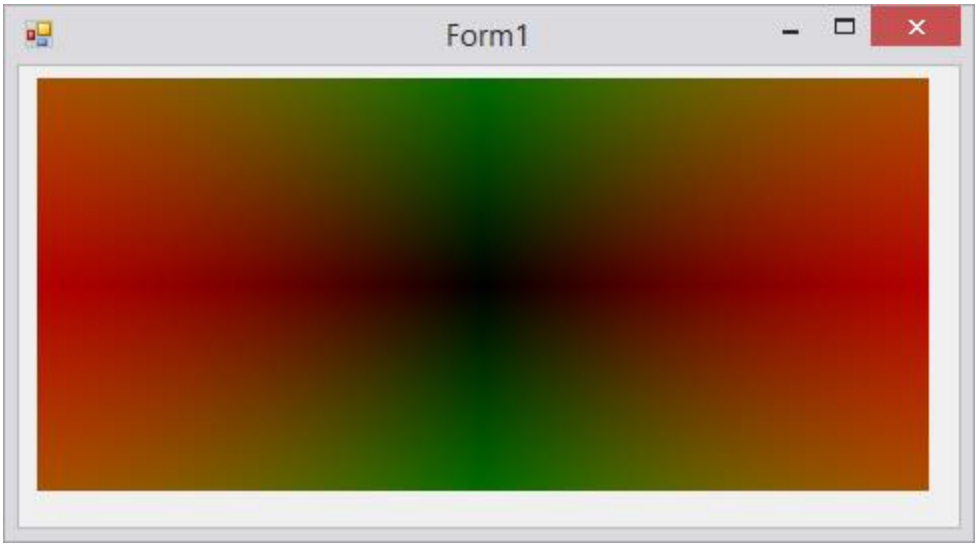
```
void main ( void )
{
```

```
    SCamera uCamera = initializeDefaultCamera();
    SRay ray = GenerateRay( uCamera);
    FragColor = vec4 ( abs(ray.Direction.xy), 0, 1.0);
}
```

Умножение на uCamera.Scale необходимо, чтобы изображение не деформировалось при изменении размеров окна:



Без умножения на uCamera.Scale.



После умножения на uCamera.Scale.

3. 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.

```
#version 430
#define EPSILON = 0.001
#define BIG = 1000000.0
const int DIFFUSE = 1;
const int REFLECTION = 2;
const int REFRACTION = 3;

struct SSphere
{
    vec3 Center;
    float Radius;
    int MaterialIdx;
};
struct STriangle
{
    vec3 v1;
    vec3 v2;
    vec3 v3;
    int MaterialIdx;
};
```

ОБЪЯВЛЕНИЕ И ИНИЦИАЛИЗАЦИЯ ДАННЫХ

В фрагментном шейдере объявите глобальные массивы сфер и треугольников. В структурах есть переменные MaterialIdx, которые будут содержать индекс в массиве материалов. Структура материала будет рассмотрена в следующем разделе. Пока можно задать все индексы нулевыми.

```
STriangle triangles[10];
SSphere spheres[2];
```

Создайте функцию *initializeDefaultScene()*, которая будет инициализировать переменные данными по умолчанию.

```
void initializeDefaultScene(out STriangle triangles[], out SSphere spheres[])
{
      /** TRIANGLES **/
      /* left wall */
      triangles[0].v1 = vec3(-5.0,-5.0,-5.0);
      triangles[0].v2 = vec3(-5.0, 5.0, 5.0);
      triangles[0].v3 = vec3(-5.0, 5.0,-5.0);
      triangles[0].MaterialIdx = 0;

      triangles[1].v1 = vec3(-5.0,-5.0,-5.0);
      triangles[1].v2 = vec3(-5.0,-5.0, 5.0);
      triangles[1].v3 = vec3(-5.0, 5.0, 5.0);
      triangles[1].MaterialIdx = 0;

      /* back wall */
      triangles[2].v1 = vec3(-5.0,-5.0, 5.0);
      triangles[2].v2 = vec3( 5.0,-5.0, 5.0);
      triangles[2].v3 = vec3(-5.0, 5.0, 5.0);
      triangles[2].MaterialIdx = 0;

      triangles[3].v1 = vec3( 5.0, 5.0, 5.0);
```

```
        triangles[3].v2 = vec3(-5.0, 5.0, 5.0);
        triangles[3].v3 = vec3( 5.0,-5.0, 5.0);
        triangles[3].MaterialIdx = 0;

        /* Самостоятельно добавьте треугольники так, чтобы получился куб */

        /** SPHERES **/
        spheres[0].Center = vec3(-1.0,-1.0,-2.0);
        spheres[0].Radius = 2.0;
        spheres[0].MaterialIdx = 0;

        spheres[1].Center = vec3(2.0,1.0,2.0);
        spheres[1].Radius = 1.0;
        spheres[1].MaterialIdx = 0;
}
```

Вызовите вашу функцию после объявленных переменных.

```
initializeDefaultScene ( triangles, spheres );
```

Проверьте, что ваш шейдер компилируется и при запуске вашей программы Output не содержит ошибок.

3. 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 леча со сферой, можно воспользоваться аналичтическим решением:

Уравнение луча: $r = o + dt$

Уравнение сферы: $x^2 + y^2 + z^2 = R$ или $P^2 - R^2 = 0$

Подставляем $(o + dt)^2 - R^2 = 0$

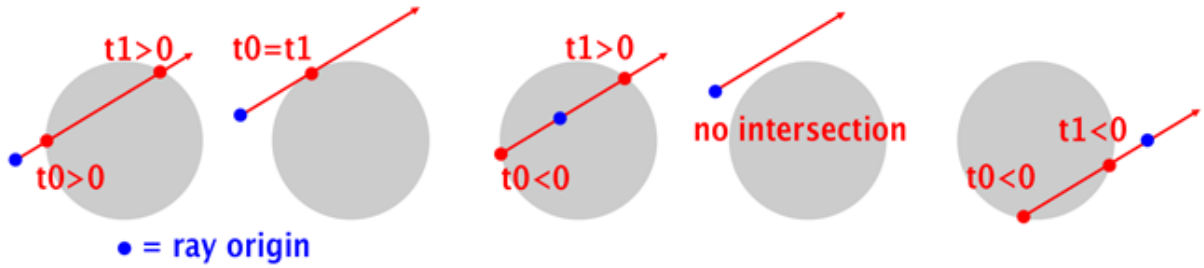
Раскрываем скобки и получаем квадратное уравнение относительно t:

$$d^2t^2 + 2odt + o^2 - R^2 = 0$$

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

Если дискриминант < 0, то точек пересечения нет.



Реализация функции IntersectSphere представлена ниже:

```
bool IntersectSphere ( SSphere sphere, SRay ray, float start, float final, out float
time )
{
    ray.Origin -= sphere.Center;
    float A = dot ( ray.Direction, ray.Direction );
    float B = dot ( ray.Direction, ray.Origin );
    float C = dot ( ray.Origin, ray.Origin ) - sphere.Radius * sphere.Radius;
    float D = B * B - A * C;
    if ( D > 0.0 )
      {
       D = sqrt ( D );
       //time = min ( max ( 0.0, ( -B - D ) / A ), ( -B + D ) / A );
            float t1 = ( -B - D ) / A;
       float t2 = ( -B + D ) / A;
       if(t1 < 0 && t2 < 0)
                   return false;

            if(min(t1, t2) < 0)
            {
           time = max(t1,t2);
           return true;
       }
            time = min(t1, t2);
       return true;
      }
      return false;
}
```

Способы пересечения треугольника с лучом можно найти в презентации
http://www2.sscc.ru/Seminars/Docum/NVIDIA%20CUDA-2012/Lec_6_Novsb.pdf
Реализация ниже:

```
bool IntersectTriangle (SRay ray, vec3 v1, vec3 v2, vec3 v3, out float time )
{
// // Compute the intersection of ray with a triangle using geometric solution
// Input: // points v0, v1, v2 are the triangle's vertices
// rayOrig and rayDir are the ray's origin (point) and the ray's direction
// Return: // return true is the ray intersects the triangle, false otherwise
// bool intersectTriangle(point v0, point v1, point v2, point rayOrig, vector rayDir)
{
// compute plane's normal vector
time = -1;
vec3 A = v2 - v1;
vec3 B = v3 - v1;
// no need to normalize vector
```

```
vec3 N = cross(A, B);
// N
// // Step 1: finding P
// // check if ray and plane are parallel ?
float NdotRayDirection = dot(N, ray.Direction);
if (abs(NdotRayDirection) < 0.001)
	return false;
// they are parallel so they don't intersect !
// compute d parameter using equation 2
float d = dot(N, v1);
// compute t (equation 3)
float t = -(dot(N, ray.Origin) - d) / NdotRayDirection;
// check if the triangle is in behind the ray
if (t < 0)
	return false;
// the triangle is behind
// compute the intersection point using equation 1
vec3 P = ray.Origin + t * ray.Direction;
// // Step 2: inside-outside test //
vec3 C;
// vector perpendicular to triangle's plane
// edge 0
vec3 edge1 = v2 - v1;
vec3 VP1 = P - v1;
C = cross(edge1, VP1);
if (dot(N, C) < 0)
	return false;
// P is on the right side
// edge 1
vec3 edge2 = v3 - v2;
vec3 VP2 = P - v2;
C = cross(edge2, VP2);
if (dot(N, C) < 0)
	return false;
// P is on the right side
// edge 2
vec3 edge3 = v1 - v3;
vec3 VP3 = P - v3;
C = cross(edge3, VP3);
if (dot(N, C) < 0)
	return false;
// P is on the right side;
time = t;
return true;
// this ray hits the triangle
}
```

Функция Raytrace 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, материалы будут рассмотрены в следующем разделе, пока заполним их нулями.

```
struct SIntersection
{
    float Time;
    vec3 Point;
    vec3 Normal;
    vec3 Color;
```

```
    // ambient, diffuse and specular coeffs
    vec4 LightCoeffs;
    // 0 - non-reflection, 1 - mirror
    float ReflectionCoef;
    float RefractionCoef;
    int MaterialType;
};
```

Создаём функцию трассирующую луч. Пока у вас нет материалов, просто присвойте любой цвет.

```
bool Raytrace ( SRay ray, SSphere spheres[], STriangle triangles[], SMaterial
materials[], float start, float final, inout SIntersection intersect )
{
bool result = false;
float test = start;
intersect.Time = final;
//calculate intersect with spheres
for(int i = 0; i < 2; i++)
{
    SSphere sphere = spheres[i];
    if( IntersectSphere (sphere, ray, start, final, test ) && test < intersect.Time )
    {
        intersect.Time = test;
        intersect.Point = ray.Origin + ray.Direction * test;
        intersect.Normal = normalize ( intersect.Point - spheres[i].Center );
        intersect.Color = vec3(1,0,0);
        intersect.LightCoeffs =   vec4(0,0,0,0);
        intersect.ReflectionCoef = 0;
        intersect.RefractionCoef = 0;
        intersect.MaterialType =   0;
        result = true;
    }
}
//calculate intersect with triangles
for(int i = 0; i < 10; i++)
{
    STriangle triangle = triangles[i];

    if(IntersectTriangle(ray, triangle.v1, triangle.v2, triangle.v3, test)
       && test < intersect.Time)
      {
        intersect.Time = test;
        intersect.Point = ray.Origin + ray.Direction * test;
        intersect.Normal =
             normalize(cross(triangle.v1 - triangle.v2, triangle.v3 - triangle.v2));
        intersect.Color = vec3(1,0,0);
        intersect.LightCoeffs =   vec4(0,0,0,0);
        intersect.ReflectionCoef = 0;
        intersect.RefractionCoef = 0;
        intersect.MaterialType =   0;
        result = true;
    }

}
return result;
}
```

Модифицируйте функцию main следующим образом:

```
#define BIG 1000000.0

void main ( void )
{
    float start = 0;
    float final = BIG;

    SCamera uCamera = initializeDefaultCamera();
    SRay ray = GenerateRay( uCamera);
    SIntersection intersect;
    intersect.Time = BIG;
    vec3 resultColor = vec3(0,0,0);
    initializeDefaultScene(triangles, spheres);
    if (Raytrace(ray, spheres, triangles, materials, start, final, intersect))
    {
        resultColor = vec3(1,0,0);
    }
    FragColor = vec4 (resultColor, 1.0);
}
```

Теперь на экране должны появиться красные силуэты геометрии сцены.

ОСВЕЩЕНИЕ

Чтобы объекты имели собственную тень и отбрасывали падающую в нашу модель необходимо добавить источник света.

```
struct SLight
{
    vec3 Position;
};
```

Можно выбрать любую из моделей освещения: Ламберт, Фонг, Блинна, Кук-Торренс. Мы будем реализовывать затенение по Фонгу (http://compgraphics.info/3D/lighting/phong_reflection_model.php). 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 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 освещения, $p-$ коэффициент резкости бликов

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

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

Создаем структуру для хранения коэффициентов материала. Для расчета освещения по Фонгу нам потребуются первые два поля, остальные поля потребуются при реализации зеркального отражения и преломления.

```
struct SMaterial
{
    //diffuse color
    vec3 Color;
    // ambient, diffuse and specular coeffs
    vec4 LightCoeffs;
    // 0 - non-reflection, 1 - mirror
    float ReflectionCoef;
    float RefractionCoef;
    int MaterialType;
};
```

Добавляем как глобальные переменные источник освещения и массив материалов, а также функцию, задающую им значения по умолчанию. Вызовите эту функцию в main.

```
SLight light;
SMaterial materials[6];

void initializeDefaultLightMaterials(out SLight light, out SMaterial materials[])
{
    //** LIGHT **//
    light.Position = vec3(0.0, 2.0, -4.0f);

    /** MATERIALS **/
    vec4 lightCoefs = vec4(0.4,0.9,0.0,512.0);
    materials[0].Color = vec3(0.0, 1.0, 0.0);
    materials[0].LightCoeffs = vec4(lightCoefs);
    materials[0].ReflectionCoef = 0.5;
    materials[0].RefractionCoef = 1.0;
    materials[0].MaterialType = DIFFUSE;

    materials[1].Color = vec3(0.0, 0.0, 1.0);
    materials[1].LightCoeffs = vec4(lightCoefs);
    materials[1].ReflectionCoef = 0.5;
    materials[1].RefractionCoef = 1.0;
    materials[1].MaterialType = DIFFUSE;
}
```

Теперь можно назначить геометрии различные материалы, и копировать значения материалов в функции Raytrace при пересечении в переменную ближайшего пересечения SIntersect intersect (там, где раньше мы поставили нули).

Теперь напишем функцию Phong.

```
vec3 Phong ( SIntersection intersect, SLight currLight)
{
    vec3 light = normalize ( currLight.Position - intersect.Point );
    float diffuse = max(dot(light, intersect.Normal), 0.0);
    vec3 view = normalize(uCamera.Position - intersect.Point);
    vec3 reflected= reflect( -view, intersect.Normal );
    float specular = pow(max(dot(reflected, light), 0.0), intersect.LightCoeffs.w);
    return intersect.LightCoeffs.x * intersect.Color +
           intersect.LightCoeffs.y * diffuse * intersect.Color +
           intersect.LightCoeffs.z * specular * Unit;
}
```

Чтобы «нарисовать» падающие тени необходимо выпустить, так называемые теневые лучи. Из каждой точки, для которой рассчитываем освещение выпускается луч на источник света, если этот луч пересекает какую-нибудь ещё геометрию сцены, значит точка в тени и она освещена только ambient компонентой.

```
float Shadow(SLight currLight, SIntersection intersect)
{
    // Point is lighted
    float shadowing = 1.0;
    // Vector to the light source
    vec3 direction = normalize(currLight.Position - intersect.Point);
    // Distance to the light source
    float distanceLight = distance(currLight.Position, intersect.Point);
    // Generation shadow ray for this light source
    SRay shadowRay = SRay(intersect.Point + direction * EPSILON, direction);
    // ...test intersection this ray with each scene object
    SIntersection shadowIntersect;
    shadowIntersect.Time = BIG;
    // trace ray from shadow ray begining to light source position
    if(Raytrace(shadowRay, spheres, triangles, materials, 0, distanceLight,
                shadowIntersect))
    {
        // this light source is invisible in the intercection point
        shadowing = 0.0;
    }
    return shadowing;
}
```

Обратите внимание, что для вычисления пересечения используется та же функция Raytrace.

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

```
return intersect.LightCoeffs.x * intersect.Color +
          intersect.LightCoeffs.y * diffuse * intersect.Color * shadow +
          intersect.LightCoeffs.z * specular * Unit;
```

Добавьте в функцию main вычисление освещения:

```
if (Raytrace(ray, start, final, intersect)) // Tracing primary ray
{

    float shadowing = Shadow(uLight, intersect);
    resultColor += contribution * Phong ( intersect, uLight, shadowing );
}
```

ЗЕРКАЛЬНОЕ ОТРАЖЕНИЕ

До этого моделировались исключительно объекты из материала диффузно рассеивающего свет. С помощью рейтрейсинга можно моделировать также зеркально отражающие объекты и прозрачные объекты.

Если в сцене есть зеркальный объект, это значит, что он не имеет собственного цвета, а отражает окружающие его объекты. Чтобы узнать какой итоговый цвет мы получим, необходимо выпустить из точки пересечения луч в сцену, согласно закону отражения (угол падения равен углу отражения).

Введем типы материалов: диффузное отражение и зеркальное отражение.

```
const int DIFFUSE_REFLECTION = 1;
const int MIRROR_REFLECTION = 2;
```

Если луч пересекается с диффузным объектом, то вычисляется цвет объекта, а если с зеркальным, то создается новый зеркальный луч, который снова трассируется в сцену. Если зеркальных объектов в сцене много, то луч может переотразиться не один раз прежде чем пересечется с диффузным объектом. Т.к. в шейдерах запрещена рекурсия, введем стек для хранения лучей. На тот случай, если в сцене очень много зеркальных объектов и мало диффузных, настолько, что алгоритм имеет шанс зациклится введем ограничение на количество переотражений. Это ограничение называется глубиной трассировки.

Итак, создадим структуру TracingRay, содержащую луч, число depth, означающую номер переотражения, после которого этот луч был создан и contribution, для хранения вклада луча в результирующий цвет.

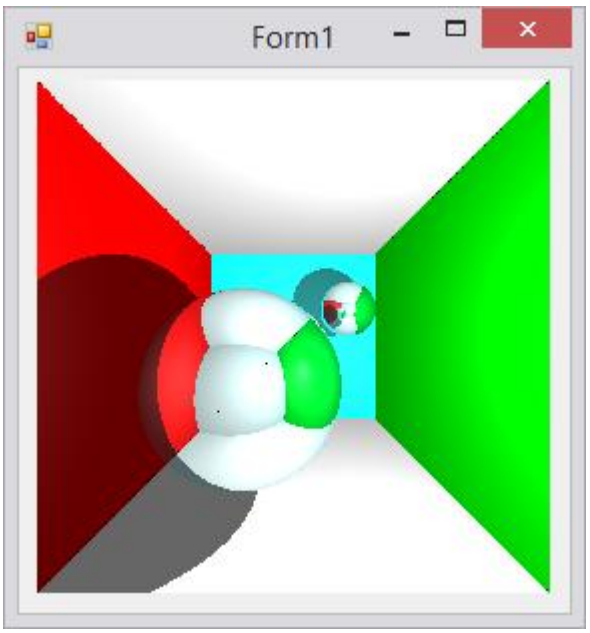
```
struct STracingRay
{
	SRay ray;
	float contribution;
	int depth;
};
```

Создайте стек на основе массива, который умеет класть луч на стек (pushRay), брать луч со стека (popRay) и проверять есть ли ещё элементы в стеке (isEmpty).

Модифицируем функцию main. Первичный луч превращаем в луч, пригодный для стека и оборачиваем функцию Raytrace в цикла while.

```
STracingRay trRay = STracingRay(ray, 1, 0);
pushRay(trRay);
while(!isEmpty())
{
    STracingRay trRay = popRay();
    ray = trRay.ray;
    SIntersection intersect;
    intersect.Time = BIG;
    start = 0;
    final = BIG;
    if (Raytrace(ray, start, final, intersect))
    {
        switch(intersect.MaterialType)
        {
        case DIFFUSE_REFLECTION:
        {
            float shadowing = Shadow(uLight, intersect);
            resultColor += trRay.contribution * Phong ( intersect, uLight, shadowing );
            break;
        }
       case MIRROR_REFLECTION:
       {
           if(intersect.ReflectionCoef < 1)
           {
                float contribution = trRay.contribution * (1 - intersect.ReflectionCoef);
                float shadowing = Shadow(uLight, intersect);
                resultColor +=  contribution * Phong(intersect, uLight, shadowing);
           }
           vec3 reflectDirection = reflect(ray.Direction, intersect.Normal);
           // creare reflection ray
           float contribution = trRay.contribution * intersect.ReflectionCoef;
           STracingRay reflectRay = STracingRay(
                    SRay(intersect.Point + reflectDirection * EPSILON, reflectDirection),
                    contribution, trRay.depth + 1);
            pushRay(reflectRay);
            break;
       }

       } // switch
    } //  if (Raytrace(ray, start, final, intersect))
} // while(!isEmpty())
```

Запускаем, при должной расстановке материалов должно получиться что-то подобное:



ПОЛЕЗНЫЕ ССЫЛКИ

1. http://www.scratchapixel.com/
2. www.opengl.org