ОГЛАВЛЕНИЕ

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

Список сокращений:

ЛКМ – левая кнопка мыши.

ПКМ – правая кнопка мыши.

НЕМНОГО ТЕОРИИ

1. ЧТО ТАКОЕ RAY TRACING И ЗАЧЕМ ОН НУЖЕН.

Классический ray tracing, или метод трассировки лучей, предложен Артуром Аппелем (Arthur Appel) в 1968 году и дополнен алгоритмом общей рекурсии, разработанным Whitted в 1980 году. Понадобилось почти 12 лет эволюции вычислительных систем, прежде чем этот алгоритм стал доступен для широкого применения в практических приложениях.

Суть метода: отслеживание траекторий лучей и расчета взаимодействий с лежащими на траекториях объектами, от момента испускания лучей источником света до момента попадания в камеру.

Под взаимодействием луча с объектами понимаются процессы диффузного (в смысле модели локальной освещенности), многократного зеркального отражения от их поверхности и прохождение лучей сквозь прозрачные объекты.

Ray tracing – первый метод расчета глобального освещения, рассматривающий освещение, затенение (расчет тени), многократные отражения и преломления.

2. КРАТКОЕ ОПИСАНИЕ АЛГОРИТМА.

1. В традиционной трассировке лучей лучи света обрабатываются в обратном направлении (backward ray tracing): луч испускается из камеры вглубь сцены через пиксель окна вывода.

   Первичный луч может не столкнуться ни с одним объектом сцены. В этом случае процесс обрывается – пиксель закрашивается цветом фона.

   В остальных случаях луч соударяется с некоторым объектом. Необходимо рассчитать прямое освещение и вторичное освещение, а также выяснить, находится ли точка в тени.

2. Для выяснения того, находится ли точка в тени, следует выпустить вторичный теневой луч из точки к источнику света. Если теневой луч столкнется с объектом сцены, то точка находится в тени. Если лучи от источника света достигают точки, то следует рассчитать вклад от прямого освещения данным

источником (используются различные модели освещения – эмпирические или физически корректные). Наиболее распространенная эмпирическая модель – освещение по Фонгу

$$C_{out} = C \cdot [k_a + k_d \cdot \max(0, (\boldsymbol{n} \cdot \boldsymbol{l}))] + I \cdot k_s \cdot \max(0, (\boldsymbol{v} \cdot \boldsymbol{r}))^p, \quad \boldsymbol{r} = \text{reflect}(-\boldsymbol{v}, \boldsymbol{n})$$

3. Однако точка может получать энергию и от вторичного освещения посредством отраженных и преломленных лучей.
    a. Если поверхность обладает отражающими свойствами, то строится вторичный луч отражения. Направление луча определяется по закону отражения. Для отраженного луча также определяется возможность его взаимодействия с объектами сцены. Если соударений нет, то возвращается цвет фона.
    b. Если же поверхность прозрачна, то строится еще и вторичный луч прозрачности (transparency ray). Для определения направления луча используется закон преломления.

ПРЯМИКОМ К КОДУ!

За основу берем приложение OpenGL с шейдерами из предыдущей темы. 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. Создадим два пустых текстовых файла «`raytracing.vert`» для вершинного шейдера и «`raytracing.frag`» - для фрагментного.

1. УПРАВЛЯЮЩАЯ ПРОГРАММА.

ЗАГРУЗКА ШЕЙДЕРОВ

```
void loadShader(String filename, ShaderType type, int program, out int address)
{
    address = GL.CreateShader(type);
    using (StreamReader sr = new StreamReader(filename))
    {
        GL.ShaderSource(address, sr.ReadToEnd());
    }
    GL.CompileShader(address);
    GL.AttachShader(program, address);
}
```

`glCreateShader` (`GL.CreateShader` в случае OpenTK) 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:

```
private void InitShaders()
{
    BasicProgramID = GL.CreateProgram();
    loadShader("..\\..\\raytracing.vert", ShaderType.VertexShader,   BasicProgramID, out BasicVertexShader);
    loadShader("..\\..\\raytracing.frag", ShaderType.FragmentShader, BasicProgramID, out BasicFragmentShader);
    GL.LinkProgram(BasicProgramID);
}
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

1. Самостоятельно организуйте проверку на успех компиляции шейдерной программы и ее компоновки.

НАСТРОЙКА БУФЕРНЫХ ОБЪЕКТОВ

Создайте массив вершин буферного объекта, задайте позицию камеры.

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

campos = new Vector3(0.0f, 0.0f, 0.8f);
```

Самостоятельно настройте буферные объекты, передайте в шейдерную программу положение камеры, соотношение сторон экрана.

2. ВЕРШИННЫЙ ШЕЙДЕР

В вершинном шейдере определение входных атрибутов выполняется с использованием квалификатора `in`. Объявлены две `uniform`-переменные: отношение сторон экрана `aspect` и позиция камеры-наблюдателя `cameraPosition`.

Переменные, отдаваемые вершинным шейдером дальше по конвейеру объявлены со спецификатором `out`. `origin` вектор - это точка, откуда испускаются лучи, то есть глаз наблюдателя, `direction` вектор - это направление трассирующего луча.

```
#version 430
uniform float aspect;
uniform vec3 campos;
in vec3 vPosition; //Входные переменные vPosition - позция вершины


out vec3 origin, direction;
void main()
{
    gl_Position = vec4(vPosition, 1.0);
    direction = normalize(vec3(vPosition.x*aspect, vPosition.y, -1.0));
    origin = campos;
}
```

3. ФРАГМЕНТНЫЙ ШЕЙДЕР

Во фрагментный шейдер `origin` и `direction` векторы поступают на вход с конвейера, а выходными данными является цвет фрагмента - `outputColor`.

```
#version 430

in vec3 origin, direction;
out vec4 outputColor;
```

ЭТАП 1

Сначала напишем простой шейдер, который будет отрисовывать одну сферу с затенением. Будем считать, что источник света расположен в одной точке с позицией наблюдателя, а материал поверхности сферы имеет только свойство рассеянного отражения. Дальнейший код следует добавить во фрагментный шейдер.

Для описания сущности сферы добавляем структуру Sphere:

```
struct Sphere {
    vec3 position;
    float radius;
    vec3 color;
};
```

Добавляем служебную структуру Ray - непосредственно луч:

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

Структура RayNode - является атомарным узлом, из которого складывается путь луча. В случае, когда испускамый луч из точки наблюдения встречается с объектом и отражается от него, нам необходимо описать оба луча: исходный и отраженный. В случае преломления понадобится еще и преломленный луч. На данном этапе RayNode будет совпадать с испускаемым лучом. (Далее мы рассмотрим более сложные случаи)

```
struct RayNode {
    Ray ray;
    vec3 color;     //накопленный цвет на данном участке пути
    int depth;      //глубина следа луча: показывает сколько раз луч отражался или преломлялся
};

const int Max_Depth = 5;    //максимальная глубина следа луча
const int Max_Nodes = 64;   //максимальное количество компонент луча
RayNode rayNode[Max_Nodes];
```

Структура HitInfo описывает столкновение луча с объектом:

```
struct HitInfo {
    bool hitDetected;       //было ли столкновение
    vec3 hitPoint;          //точка столкновения
    vec3 surfaceNormal;     //вектор нормали в точке столкновения
    float distance;         //расстояние от старта луча до точки столкновения в виде distance*direction
};
```

ЭТАП 1.1

Далее напишем функцию, которая находит пересечение луча со сферой. Любая сфера задается своим уравнением: $(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = radius^2$. Составим уравнение прямой через точку $origin$, параллельно вектору $direction$: $(x\ y\ z)^{\mathrm{T}} = direction * stepCount + origin$. Обозначим $stepCount = t$. Если подставить координаты $x(t), y(t), z(t)$ прямой в уравнение сферы и уравнение будет иметь решение, то прямая пересекает сферу. Можно заметить, что при подстановке мы получим скалярное произведение вида: $< direction * t + origin - position, direction * t + origin - position > = radius^2$.
Воспользовавшись свойствами скалярного произведения получим уравнение, относительно t:

$$< direction * direction > t^2 + 2t * < origin - position, direction > + |origin - position|^2 = radius^2.$$

```
void sphereIntersect(Ray ray, Sphere sphere, inout HitInfo hitInfo) {
    vec3 trackToSphere = ray.origin - sphere.position;
    float a = dot(ray.direction, ray.direction);
    float b = 2 * dot(trackToSphere, ray.direction);
    float c = dot(trackToSphere, trackToSphere) - sphere.radius * sphere.radius;
    float discriminant = b * b - 4.0 * a * c;

    if (discriminant > 0.0) {
        float distance = (-b - sqrt(discriminant)) / (2.0 * a);
        if (distance > 0.0001 && (distance < hitInfo.distance && hitInfo.hitDetected || !hitInfo.hitDetected)) {
            hitInfo.distance = distance;
            hitInfo.hitPoint = ray.origin + ray.direction * hitInfo.distance;
            hitInfo.surfaceNormal = normalize(hitInfo.hitPoint - sphere.position);
            hitInfo.hitDetected = true;
        }
    }
}
```

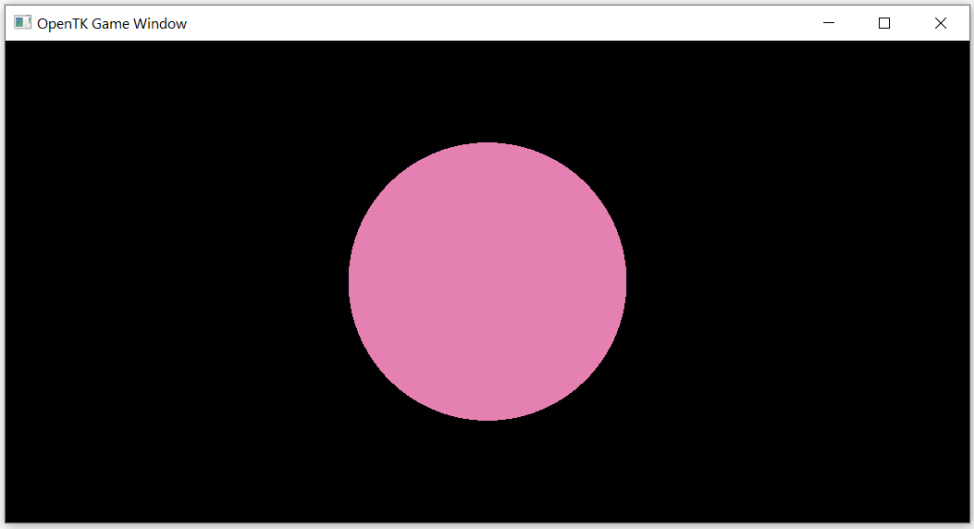
Далее напишем функцию, в которой необходимо создать объект типа сфера и проинициализировать его поля. Далее инициализируем поля первого (и пока единственного) узла луча и в цикле ищем пересечения луча и сферы (нетрудно заметить, что цикл отработает ровно одну итерацию).

```
vec3 iterativeRayTrace(Ray ray) {
    Sphere sphere;
    sphere.position = vec3(0.0, 0.0, -1.0);
    sphere.radius = 0.5;
    sphere.color = vec3(0.9, 0.5, 0.7);

    int numberOfNodes = 1, currentNodeIndex = 0;

    rayNode[currentNodeIndex].ray = ray;
    rayNode[currentNodeIndex].depth = 0;

    while (currentNodeIndex < numberOfNodes) {
        HitInfo hitInfo;
        hitInfo.hitDetected = false;
        sphereIntersect(ray, sphere, hitInfo);

        if (hitInfo.hitDetected) {
            rayNode[currentNodeIndex].color = sphere.color;
        }
        else break;
        currentNodeIndex++;
    }
    return rayNode[0].color;
}
```

Остается написать функцию `main`, в которой происходит создание объекта типа луч, инициализация его полей и вызов ранее написанной функции, которая возвращает вычисленный цвет фрагмента.

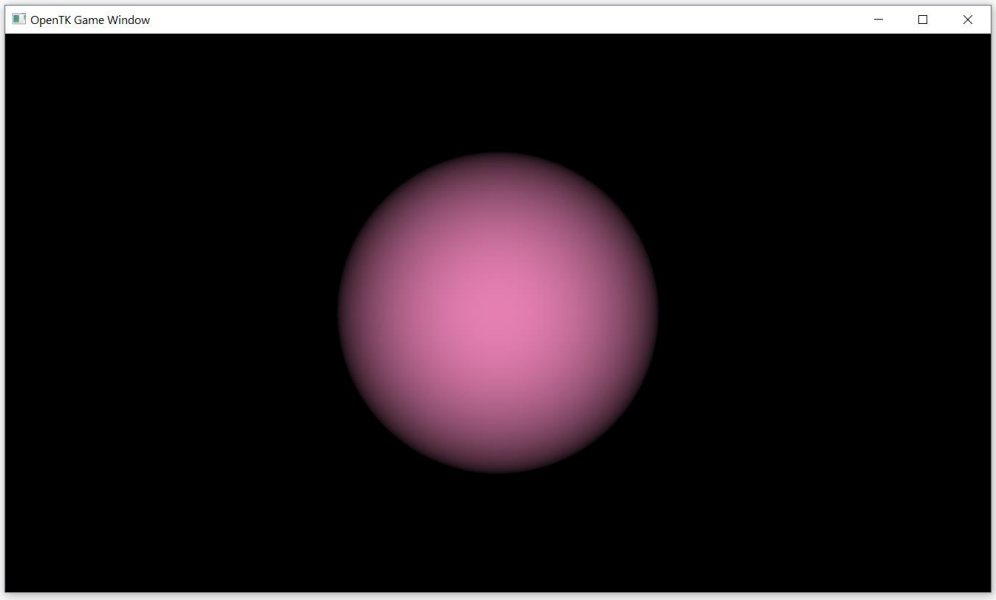
```
void main () {
    Ray ray = Ray(origin, direction);
    outputColor = vec4(iterativeRayTrace(ray), 1);
}
```

В результате получим подобное изображение:



Чтобы сфера выглядела более объемно необходимо добавить затенение. Рассмотрим простой пример, когда результирующий цвет умножается на коэффициент, равный косинусу угла между вектором направления луча и нормалью к поверхности (иначе говоря, расположим источник света непосредственно в точке наблюдения). Перед строкой `rayNode[currentNodeIndex].color = sphere.color` добавьте выражение, вычисляющее косинус угла, а потом умножьте результирующий цвет на этот коэффициент. (Для справки: $cosa = \frac{<vec_1, vec_2>}{|vec_1| * |vec_2|}$)

Должно получиться подобное изображение:



ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

1. Добавьте несколько сфер на сцену с разными цветами. (Совет: глобально объявите массив структур `Sphere` из `sphereNumber` элементов, в функции `sphereIntersect()` замените параметр типа `Sphere` на целое число - идентификатор сферы и вызовите эту функцию `sphereNumber` раз.

```
void sphereIntersect(Ray ray, int objectid, inout HitInfo hitInfo) {
    Sphere sphere = spheres[objectid];
```

При этом в теле самой функции запомните этот идентификатор, если произошло столкновение. Для этого добавьте в структуру `HitInfo` поле `int objectid.`

```
hitInfo.objectid = objectid;
```

Обратите внимание, что алгоритм предусматривает выбор наиближайшей точки пересечения с любым объектом, следовательно в вызов функции `sphereIntersect()` необходимо передавать один и тот же объект структуры `HitInfo`.)

2. Во фрагментном шейдере реализуйте отображение треугольника по трем заданным вершинам. По аналогии создайте структуру `Triangle` с полями `a`, `b` и `c`. Объявите массив структур типа треугольник. Реализуйте функцию `triangleIntersect`. Для того, чтобы проверить, принадлежит ли точка треугольнику можно воспользоваться следующим алгоритмом:

```
vec3 a, b, n;
a = triangle.a - ray.origin;
b = triangle.b - ray.origin;
n = cross(a, b);
float ip1 = dot(ray.direction, n);
a = triangle.b - ray.origin;
b = triangle.c - ray.origin;
n = cross(a, b);
float ip2 = dot(ray.direction, n);
a = triangle.c - ray.origin;
b = triangle.a - ray.origin;
n = cross(a, b);
float ip3 = dot(ray.direction, n);

if (!(((ip1 <= 0) && (ip2 <= 0) && (ip3 <= 0)) || ((ip1 >= 0) && (ip2 >= 0) && (ip3 >= 0))))
    return;
```

Далее останется только найти точку пересечения луча и плоскости, которой принадлежит треугольник. Для реализации этого алгоритма следует решить систему уравнений (способы нахождения нормали к плоскости известны из курса линейной алгебры, а за точку плоскости можно взять любую вершину треугольника):

$$\begin{cases} normal^{\mathrm{T}} * ((x_t\ y_t\ z_t)^{\mathrm{T}} - point) = 0 \\ (x_t\ y_t\ z_t)^{\mathrm{T}} = direction * t + origin \end{cases}$$

Вызовите эту функцию для каждого объявленного треугольника. Для того, чтобы правильно идентифицировать объект, с которым пересекся луч, можно добавить в структуру `HitInfo` поле `int objectType`, и в функциях `sphereIntersect` и `triangleIntersect` при необходимости присваивать ему значения констант `const int TYPE_SPHERE = 1; const int TYPE_TRIANGLE = 2.`

3. (Дополнительно) Во фрагментном шейдере реализуйте отображение плоскости по заданному вектору нормали и точке, через которую проходит плоскость.

ЭТАП 2

На данном этапе мы реализуем более сложный алгоритм, который подразумевает, что на сцене могут находиться не только объекты, у которых материал имеет диффузную составляющую, но и объекты, которые могут отражать свет, преломлять и создавать световые блики от источников света. Источниками света будем считать объекты, у которых задана `emission`-составляющая.

Чтобы описать все составляющие материала, добавим структуру:

```
struct Material {
    vec3 ambient;           //фоновая составляющая
    vec3 diffuse;           //диффузная составляющая
    vec3 reflection;        //зеркальное отражение
    vec3 specular;          //зеркальные блики
    vec3 transparency;      //прозрачность
    vec3 emission;          //самосвечение
    vec3 atenuation;        //затухание
    float refractionCoef;   //коэффициент преломления
    float shiness;          //глянцевость
};
```

В структурах, описывающих сферу и треугольник, замените поле, содержащее цвет объекта, на структуру `Material`.

В структуре `Ray` добавьте поле, которое будет характеризовать тип этого луча. И добавьте константы:

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

const int TYPE_DIFFUSE = 1;
const int TYPE_SHADOW = 2;
const int TYPE_REFLECTION = 3;
const int TYPE_TRANSPARENCY = 4;
```

Далее необходимо изменить структуру `RayNode`. Теперь в ней необходимо хранить не только луч и одну составляющую цвета, но и индекс родительского узла, цвет, который накопили отраженные и преломленные лучи и свойства материала, с которым столкнулся луч.

```
struct RayNode {
    Ray ray;
    vec3 reflectionColor;   //возвращаемый цвет от зеркального отражения
    vec3 refractionColor;   //возвращаемый цвет от преломленного луча
    vec3 diffuseColor;      //возвращаемый цвет от диффузного рассеивания, фона и бликов
    vec3 specular;          //свойство материала в отношении бликов
    vec3 reflection;        //свойство материала в отношении отражения
    vec3 refraction;        //свойство материала в отношении преломления
    int parentIndex;        //индекс родительского луча
    int depth;              //глубина следа луча: показывает сколько раз луч отражался или преломлялся
};
```

На следующем шаге изменим функцию `iterativeRayTrace`, где теперь будем генерировать отраженные и преломленные лучи, а также подсчитывать итоговый цвет. Этот алгоритм можно сравнить с построением дерева, где в корне будет находиться нулевой луч, испущенный из камеры, а сыновьями очередной вершины будут отраженные и преломленные лучи. В алгоритме ray tracing подразумевается, что трассировка обрывается, если встретился объект, который имеет только диффузную составляющую, или по количеству итераций. В цикле добавим инициализацию полей, накапливающих цвет узла.

```
while (currentNodeIndex < numberOfNodes) {
    rayNode[currentNodeIndex].diffuseColor = vec3(0);
    rayNode[currentNodeIndex].reflectionColor = vec3(0);
    rayNode[currentNodeIndex].refractionColor = vec3(0);
```

Как и раньше ищем пересечение луча с объектами

```
    HitInfo hitInfo;
    hitInfo.hitDetected = false;
    for (int i = 0; i < sphereNumber; i++)
        sphereIntersect(rayNode[currentNodeIndex].ray, i, hitInfo);
    for (int i = 0; i < triangleNumber; i++)
        triangleIntersect(rayNode[currentNodeIndex].ray, i, hitInfo);
```

Обработку условия `if (hitInfo.hitDetected)` придется полностью переписать. Для начала необходимо получить материал того объекта, с которым произошло столкновение:

```
    if (hitInfo.hitDetected)
    {
        Material material;
        switch (hitInfo.objectType) {
        case TYPE_SPHERE   : material = spheres[hitInfo.objectid].material;    break;
        case TYPE_PLANE    : material = planes[hitInfo.objectid].material;     break;
        case TYPE_TRIANGLE : material = triangles[hitInfo.objectid].material;  break;
        }
```

Задаем полям RayNode необходимые значения

```
    rayNode[currentNodeIndex].specular   = material.specular;
    rayNode[currentNodeIndex].reflection = material.reflection;
    rayNode[currentNodeIndex].refraction = material.transparency;
```

И далее генерируем зеркальный луч, если это необходимо:

```
    if (length(material.reflection) > 0.0 && rayNode[currentNodeIndex].depth < MAX_DEPTH)
    {
        //генерация зеркального луча
        vec3 reflectionDir = normalize(reflect(rayNode[currentNodeIndex].ray.direction, hitInfo.surfaceNormal));
        vec3 offset = reflectionDir * 0.01;
        rayNode[numberOfNodes].ray = Ray(hitInfo.hitPoint + offset, reflectionDir, TYPE_REFLECTION);
        rayNode[numberOfNodes].parentIndex = currentNodeIndex;
        rayNode[numberOfNodes].depth = rayNode[currentNodeIndex].depth + 1;
        numberOfNodes++;
    }
```

Заметьте, что новый луч испускается не из точки столкновения, а из точки, смещенной по направлению луча на `offset`. Это необходимо для корректной работы алгоритма. В противном случае, из-за округлений или нехватки точности вычислений точка столкновения может оказаться внутри объекта.

Аналогичным образом создается и преломленный луч:

```
if (length(material.transparency) > 0.0 && rayNode[currentNodeIndex].depth < MAX_DEPTH)
{
    //генерация преломленного луча
    vec3 refractionDir = normalize(refract(rayNode[currentNodeIndex].ray.direction, hitInfo.surfaceNormal,
                                material.refractionCoef));
    vec3 offset = refractionDir * 0.01;
    rayNode[numberOfNodes].ray = Ray(hitInfo.hitPoint + offset, refractionDir, TYPE_TRANSPARENCY);
    rayNode[numberOfNodes].parentIndex = currentNodeIndex;
    rayNode[numberOfNodes].depth = rayNode[currentNodeIndex].depth + 1;
    numberOfNodes++;
}
```

Кроме испускания зеркального и преломленного луча необходимо вычислить влияние источников света на объект. Происходит это в функции `calculateColor`, которую мы реализуем позже. Обратите внимание, что здесь больше узлов не генерируется, а значит текущая ветка распространения луча заканчивается в случае, если не было создано зеркального или преломленного луча. То есть данный узел является листом в дереве распространения лучей.

```
if (length(material.ambient) > 0.0 || length(material.diffuse) > 0.0 || length(material.specular) > 0.0
        || rayNode[currentNodeIndex].depth >= MAX_DEPTH)
{
    //считаем влияние других источников света: диффузное рассеивание, блики
    rayNode[currentNodeIndex].diffuseColor = calculateColor(hitInfo);
}
```

Если же столкновения не было, то идем по противной ветке, где присваиваем узлу цвет фона, и переходим к рассмотрению следующего узла

```
else
{
    //если столкновения не было, задаем цвет фона
    rayNode[currentNodeIndex].diffuseColor = vec3(0.0,0.0,0.0);
}
currentNodeIndex++; //переходим к следующему узлу
```

После построения такого дерева необходимо итеративно вернуться к корню, считая цвет на каждом шаге.

```
for (int i = currentNodeIndex - 1; i > 0; i--) {
    //считаем итоговый цвет на текущем луче
    vec3 nodeColor = rayNode[i].diffuseColor + rayNode[i].reflectionColor * rayNode[i].reflection +
                     rayNode[i].refractionColor * rayNode[i].refraction;
    if (rayNode[i].ray.type == TYPE_REFLECTION)
        //если текущий луч -- отраженный, то задаем родительскому лучу цвет от зеркального отражения
        rayNode[rayNode[i].parentIndex].reflectionColor = nodeColor;
    else if (rayNode[i].ray.type == TYPE_TRANSPARENCY)
        //если текущий луч -- преломленный, то задаем родительскому лучу цвет от преломленния
        rayNode[rayNode[i].parentIndex].refractionColor = nodeColor;
}
```

После завершения цикла в нулевом узле будут находиться все необходимые значения цветов, а значит можно вычислить и вернуть результат:

```
return clamp(rayNode[0].diffuseColor + rayNode[0].reflectionColor * rayNode[0].reflection +
             rayNode[0].refractionColor * rayNode[0].refraction, vec3(0), vec3(1));
```

ЭТАП 2.1

Реализуем ранее упомянутую функцию `vec3 calculateColor(HitInfo hitInfo)`. Во-первых, нам необходим материал объекта, с которым столкнулся луч, а так же сама точка столкновения и нормаль к поверхности.

```
Material material;
if (hitInfo.objectType == TYPE_SPHERE)
    material = spheres[hitInfo.objectid].material;
if (hitInfo.objectType == TYPE_TRIANGLE)
    material = triangles[hitInfo.objectid].material;

vec3 hitPoint = hitInfo.hitPoint;
vec3 surfaceNormal = hitInfo.surfaceNormal;

if (length(material.emission) > 0.0)                //если у материала есть emission состовляющая,
    return material.emission + material.diffuse;    //то не будем считать влияние других источников на него
vec3 resultColor = vec3(0);
```

Далее для каждого объекта на сцене проверим: если он источник света, то рассчитаем его влияние на точку, для которой была вызвана функция. (`objectNumber` - константа, показывающая, сколько суммарно объектов на сцене)

```
for (int i = 0; i < objectNumber; i++) {
    Material lightMaterial;
    vec3 lightPosition;                             //позиция центра источника
    int lightType;                                  //тип источника света: сфера или треугольник
    int lightIndex;                                 //индекс в текущем массиве объектов
    if (i < sphereNumber) {
        lightIndex = i;
        lightMaterial = spheres[lightIndex].material;
        lightPosition = spheres[lightIndex].position;
        lightType = TYPE_SPHERE;
    }
    else {
        lightIndex = i - sphereNumber;
        lightMaterial = triangles[lightIndex].material;
        lightPosition = triangles[lightIndex].center;
        lightType = TYPE_TRIANGLE;
    }
```

```
vec3 transparency;
//если объект -- источник света, то нужно найти
// 1 - если источник в прямой видимости от hitPoint, то просчитать его затенение по Фонгу
// 2 - если мы в тени от этого источника, то задать цвет фоного свечения
if ((hitInfo.objectid != lightIndex || hitInfo.objectType != lightType) && length(lightMaterial.emission) > 0.0)
{
    vec3 currentColor = vec3(0);
    if (!isShadowed(hitPoint, lightIndex, lightType, transparency))
    {
        vec3 lightDir = lightPosition - hitPoint;       //направление на источник
        float distance = length(lightDir);              //расстояние до него
        lightDir = normalize(lightDir);

        vec3 eyeDir = normalize(origin - hitPoint);     //направление от камеры до точки столкновения
        vec3 reflectDir = vec3(0);
        if (dot(surfaceNormal, lightDir) > 0.0)
            reflectDir = normalize(reflect(surfaceNormal, -lightDir));  //направление идеального отражения

        currentColor += phongShading(material, lightMaterial, hitPoint, surfaceNormal,
            lightDir, reflectDir, eyeDir, distance) * transparency;
    }
    else
    {
        currentColor += lightMaterial.ambient * material.ambient;
    }
    resultColor += currentColor;
}
```

В качестве результата вернем `resultColor`. Для проверки, находится ли точка в тени от текущего источника, используется функция `isShadowed`. Для просчета затенения по Фонгу -- функция `phongShading`. Обе эти функции будут реализованы далее.

ЭТАП 2.2

Реализуем функцию `isShadowed`. Функция возвращает `true`, если точка в тени, `false` в противном случае. Один из параметров функции со спецификатором `inout vec3 transparency` - это накопленная прозрачность. Если на пути от точки до источника встретится прозрачный объект, точка не считается в тени, если полупрозрачный, то точка также не считается затененной, но интенсивность света от источника будет уменьшена.

Сначала сгенерируем луч от точки до источника, пересечем его с источником и найдем расстояние до него.

```
bool isShadowed(vec3 hitPoint, int lightIndex, int lightType, inout vec3 transparency) {
    HitInfo hitInfoLight;
    hitInfoLight.hitDetected = false;
    Ray ray;
    if (lightType == TYPE_SPHERE) {
        Sphere light = spheres[lightIndex];
        vec3 eps = normalize(light.position - hitPoint) * 0.001;
        ray = Ray(hitPoint + eps, normalize(light.position - hitPoint), TYPE_SHADOW);
        sphereIntersect(ray, lightIndex, hitInfoLight);
    }
    if (lightType == TYPE_TRIANGLE) {
        Triangle light = triangles[lightIndex];
        vec3 eps = normalize(light.center - hitPoint) * 0.001;
        ray = Ray(hitPoint + eps, normalize(light.center - hitPoint), TYPE_SHADOW);
        triangleIntersect(ray, lightIndex, hitInfoLight);
    }

    float distance = hitInfoLight.distance;     //расстояние до источника света
```

Далее для каждого объекта сцены проверим, не загораживает ли он источник света.

```
    HitInfo hitInfo;
    transparency = vec3(1.0);
    for (int i = 0; i < objectNumber; i++) {
        hitInfo.hitDetected = false;
        Material material;
        int type;                               //тип источника света: сфера или треугольник
        int index;                              //индекс в текущем массиве объектов
        if (i < sphereNumber) {
            index = i;
            material = spheres[index].material;
            type = TYPE_SPHERE;
            sphereIntersect(ray, index, hitInfo);
        }
        else {
            index = i - sphereNumber;
            material = triangles[index].material;
            type = TYPE_TRIANGLE;
            triangleIntersect(ray, index, hitInfo);
        }
        //если между точкой столкновения и источником есть объекты,
        //и расстояние до него меньше расстояния до источника,
        //то есть он загораживает источник, тогда входим в тело if
        if ((lightIndex != index || lightType != type) && hitInfo.hitDetected && hitInfo.distance < distance)
        {
            //если объект прозрачный - уменьшить прозрачность
            if (length(material.transparency) > 0) {
                transparency *= material.transparency;
                continue;
            }
            //если объект непрозрачный - точка столкновения в тени от этого источника
            transparency = vec3(0.0);
            return true;
        }
    }
    return false;
```

ЭТАП 2.3

Реализуем функцию `phongShading`, согласно формуле Фонга

```
vec3 phongShading(Material material, Material lightMaterial, vec3 hitPoint, vec3 surfaceNormal,
                  vec3 lightDir, vec3 reflectDir, vec3 eyeDir, float distance)
{
    //рассчитываем затухание света от источника в зависимости от расстояния
    float attenuation = 1.0 / (1.0 + lightMaterial.atenuation.x + distance * lightMaterial.atenuation.y +
                        distance * distance * lightMaterial.atenuation.z);

    //рассчитываем коэффициенты в формуле Фонга
    float diffuseCoef = max(0.0, dot(surfaceNormal, lightDir));
    float specularCoef = pow(dot(eyeDir, reflectDir), material.shiness);

    return  material.ambient * lightMaterial.ambient +
            (material.diffuse * lightMaterial.emission * diffuseCoef +
            material.specular * lightMaterial.emission * specularCoef) * attenuation;
}
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

1. Самостоятельно создайте несколько объектов на сцене и задайте свойства материалов. Обратите внимание, что следует организовать передачу геометрии объектов и свойств материалов из управляющей программы в шейдерную. Рассмотрим примитивный способ. Например, свойства материала типа золото:

```
materials[4].ambient = new Vector3(0.25f, 0.2f, 0.07f);
materials[4].diffuse = new Vector3(0.75f, 0.6f, 0.23f);
materials[4].reflection = new Vector3(0.63f, 0.56f, 0.37f);
materials[4].specular = new Vector3(1.0f, 1.0f, 1.0f);
materials[4].transparency = new Vector3(0.0f, 0.0f, 0.0f);
materials[4].emission = new Vector3(0.0f, 0.0f, 0.0f);
materials[4].atenuation = new Vector3(1.0f, 1.0f, 6);
materials[4].refractionCoef = 0.0f;
materials[4].shiness = 150;
```

Далее следует передать значения свойств материала в шейдерную программу:

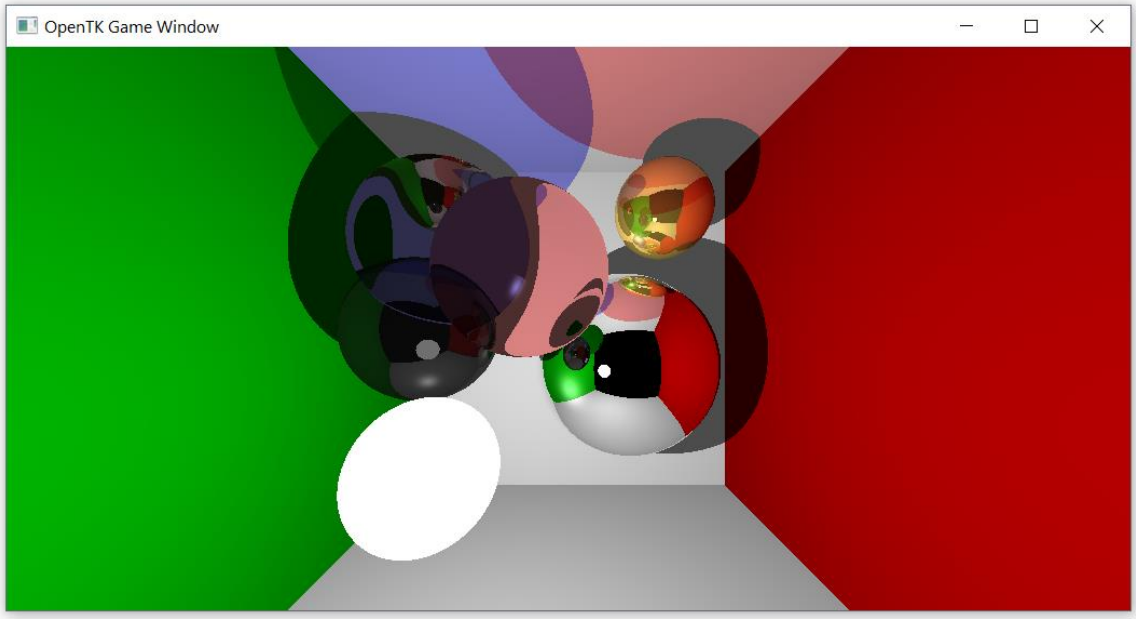
```
for (int i = 0; i < materialNumber; i++)
{
    int ambPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].ambient");
    int difPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].diffuse");
    int refPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].reflection");
    int spcPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].specular");
    int traPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].transparency");
    int emsPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].emission");
    int atnPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].atenuation");
    int refrPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].refractionCoef");
    int shnPos = GL.GetUniformLocation(BasicProgramID, "materials[" + i.ToString() + "].shiness");

    GL.Uniform3(ambPos, materials[i].ambient);
    GL.Uniform3(difPos, materials[i].diffuse);
    GL.Uniform3(refPos, materials[i].reflection);
    GL.Uniform3(spcPos, materials[i].specular);
    GL.Uniform3(traPos, materials[i].transparency);
    GL.Uniform3(emsPos, materials[i].emission);
    GL.Uniform3(atnPos, materials[i].atenuation);
    GL.Uniform1(refrPos, materials[i].refractionCoef);
    GL.Uniform1(shnPos, materials[i].shiness);
}
```

Обратите внимание, что при организации такой передачи данных, необходимо добавить `uniform`-спецификаторы тем переменным, которые инициализируются из управляющей программы.

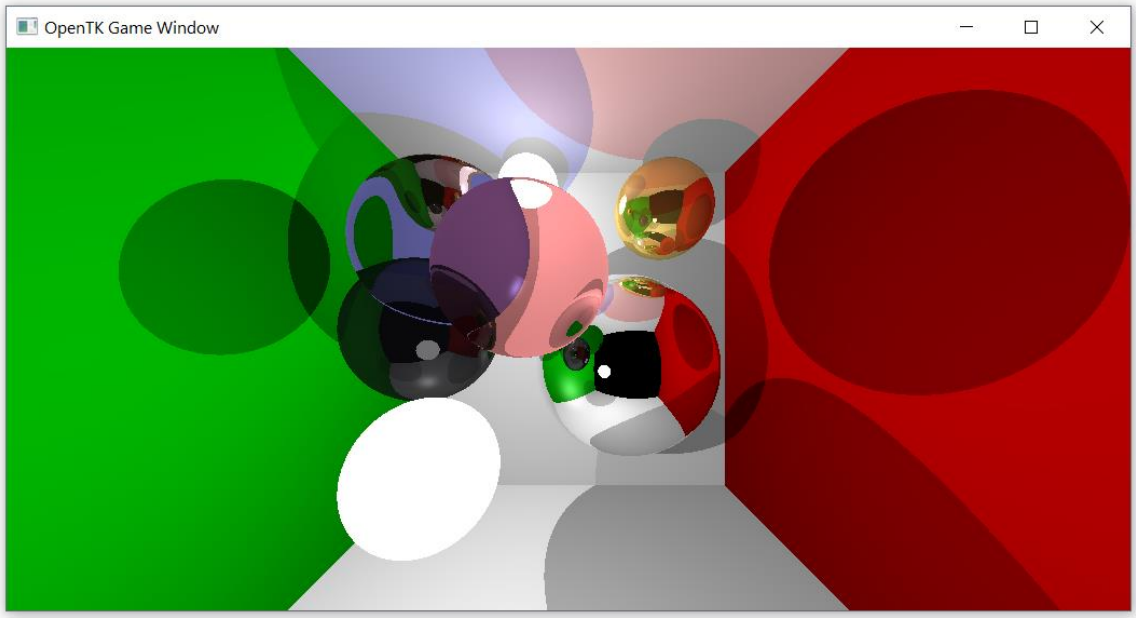
```
uniform int materialNumber;
const int maxMaterialNumber = 20;
uniform Material materials[maxMaterialNumber];
```

Вы также можете найти описание некоторых материалов в Интернете, или экспериментировать самостоятельно.
Аналогичным образом нужно передать значения полей для каждого объекта на сцене.

2. Пользуясь тем, что вы реализовали методы отображения треугольников и сфер, создайте подобие корнуэльской комнаты.



3. Добавьте несколько источников света



4. Реализуйте движение камеры по сцене. Для поворота камеры проще всего использовать матрицы поворота.