Программирование графических приложений

Тема 11
Освещение

Цель изучения темы. Изучение методов работы с освещением в WebGL.

11.1. Основы освещения

Освещение в значительной степени влияет на реалистичность сцены. Рассмотрим основы освещения WebGL и проведем небольшое сравнение подходов в реализации освещения без применения библиотек.

Мы видим объект, потому что он отражает свет (желтые стрелки на рисунке). Количество отраженного света зависит от направленности поверхности, которая выражена через нормаль (синие стрелки). Нормали, в свою очередь, позволяют нам определить направление отраженного света. Если отраженный свет не направлен в сторону камеры (как в случае с левой гранью), то поверхность будет темной.

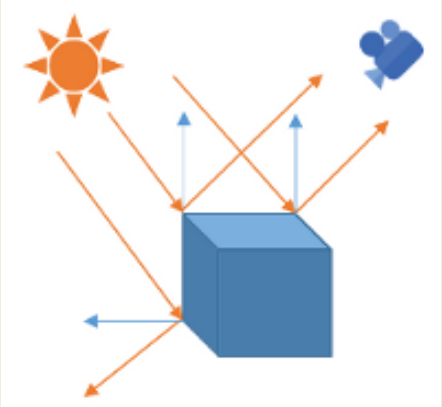

Рис. 11.1

Здесь показано направленное освещение, то есть освещение, источник которого не имеет координат и, соответственно, не находится внутри сцены (как свет Солнца), лучи которого для всех объектов имеют один угол падения.

Типы освещения

Выделяют три основных типа освещения трехмерной сцены:

- Ambient light: окружающее естественное освещение, рассеянный свет;
- Directional light: направленный свет, например, солнечный свет;
- Point light: точечный свет, например, свет лампы.

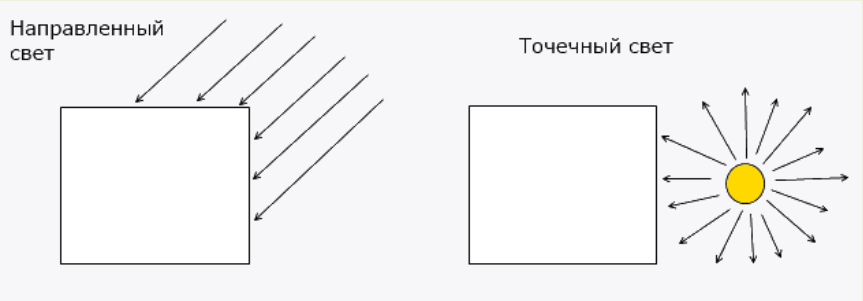

Рис. 11.2

Материалы

Материалы - это то, что составляет поверхность объекта. В программе на WebGL материалы могут быть представлены как совокупность параметров, описывающих цвет, используемые текстуры и т.д. В реальной жизни взаимодействие света и материала объекта может давать различные эффекты: освещение металла будет отличаться от освещения деревянных материалов; поэтому концепция материала важна для имитации освещения, близкого к реальному.

Нормали

Нормаль — это вектор, который перпендикулярен поверхности, которую мы хотим осветить. Нормали определяют ориентацию поверхности и поэтому занимают важное место при расчете освещения. Каждая вершина имеет нормаль. Предположим, что у нас есть треугольник с вершинами p0, p1 и p2. Нормаль в вершине p0 будет равна векторному произведению векторов v1 (p0, p1) и v2 (p0, p2). Такие вычисления производятся для каждой вершины.

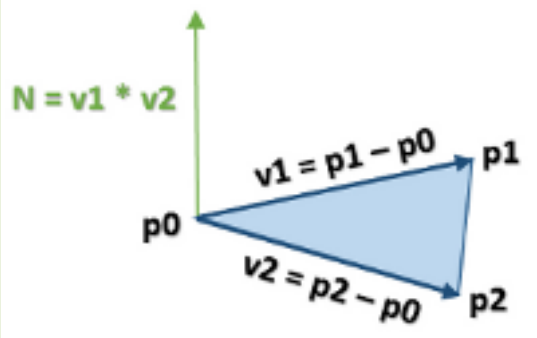

Рис. 11.3

То есть результатом векторного произведения векторов и является вектор нормали, перпендикулярный поверхности. Треугольники, которые образуются векторами, участвующими в произведении векторов, как раз представляют собой треугольники, из которых мы строим объекты в WebGL.

Если вершина принадлежит нескольким поверхностям, нормаль каждой поверхности внесет свой вклад в результирующую нормаль. Рассмотрим эту ситуацию. Предположим, что у нас есть вершина, одновременно принадлежащая поверхности P_1 с вычисленной нормалью N_1, и поверхности P_2 с нормалью N_2. Итоговая нормаль вершины в этом случае будет равняться сумме векторов нормалей N_1 и N_2. В итоге формулу для нормали вершины, которая принадлежит i поверхностям, можно определить как $N = N_1 + N_2 + \ldots + N_i$.

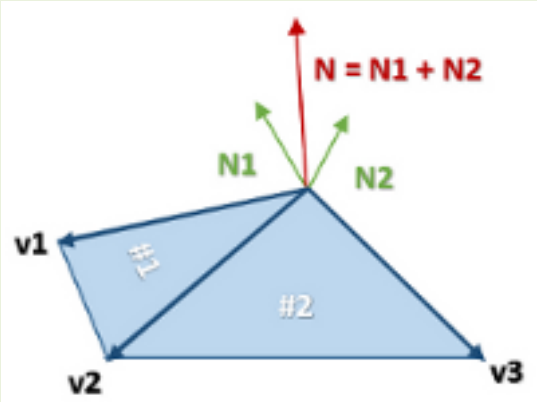

Рис. 11.4

Использование библиотеки glMatrix в программах на WebGL облегчает нам расчеты, так как там есть встроенная функция

```
vec3.cross(vector1, vector2, normal);
```

где normal - это вектор нормали, получающийся в результате произведения векторов vector1 и vector2.

Также перед использованием вектор нормали надо нормализовать, то есть привести к единичному виду. Для этого в языке шейдеров WebGL определена специальная функция normalize.

Если при создании трехмерных моделей используются специализированные программы (например, Blender или 3DSmax), то при экспорте объекта вместе с ним экспортируются также и векторы нормалей и их можно использовать для созданного в Blender или 3DSmax объекта (см. тему 16).

Модели освещения и модели затенения

Для построения реалистического изображения нам надо выбрать модель освещения (lighting model) (также называется моделью отражения) и модель затенения (shading model). Модель затенения представляет определенный тип интерполяции, позволяющей получить различное значение цвета объекта в зависимости от освещения, а модель освещения определяет способ взаимодействия материалов и света для получения итогового значения цвета объекта.

Существует несколько моделей затенения: модель Фонга, модель Гуро, плоское затенение и др. То же касается и моделей освещения.

11.2. Модели отражения света

Модель отражения света (или модель освещения) определяет, *каким образом* будет вычисляться результирующий цвет. Моделей отражения довольно много, но чаще всего используются две из них — модель Ламберта и модель Фонга.

Работу моделей отражения проиллюстрируем на примере графического объекта - чайника, который был выгружен из Blender (см. тему 16).

Модель отражения Ламберта

Эта модель базируется на законе Ламберта. Такое освещение называют *диффузным*, и его смысл заключается в том, что падающий свет отражается во всех направлениях (а не только в одном направлении, как в случае расчета бликов, например).

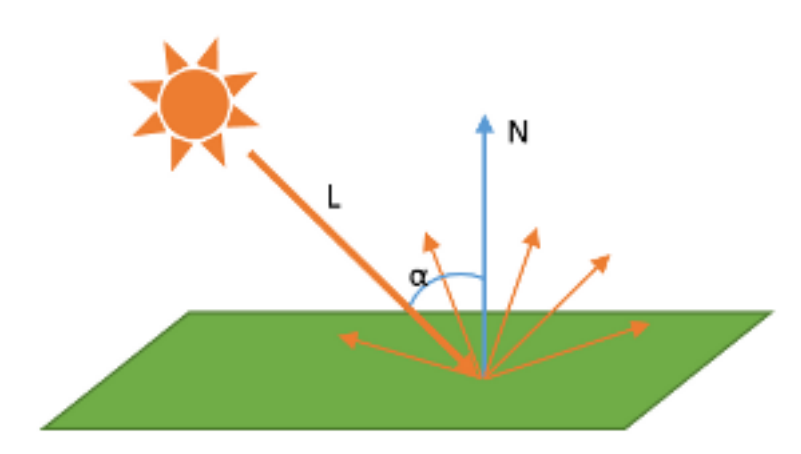

Рис. 11.5

Сила освещения зависит исключительно от угла α между вектором падения света L и вектором нормали N. Максимальная сила света будет при перпендикулярном падении света на поверхность и будет убывать с увеличением угла α. Итоговый цвет равен скалярному произведению векторов N и -L (минус говорит о том, что вектор «разворачивается» и идет от поверхности материала к источнику освещения), умноженному на цвет материала. Сюда же добавляется умножение на цвет диффузного освещения, но в нашем случае мы будем использовать просто белый цвет и его можно игнорировать в уравнении:

$$F = Cm * (-L * N)$$

где Cm — цвет материала, а L и N — вектора направления освещения и нормали соответственно.

Модель освещения в WebGL реализуется вершинным и фрагментным шейдерами. Рассмотрим их для модели освещения Ламберта.

Вершинный шейдер:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
varying vec4 vVertex;
varying vec3 vNormal;
void main(void) {
   // рассчитываем нормаль и положение вершины с учетом трансформаций
   vNormal = normalize(uNMatrix * aVertexNormal);
   vVertex = uMVMatrix * vec4(aVertexPosition, 1.0);
   gl_Position = uPMatrix * vVertex;
}
```

Фрагментный шейдер:

```
uniform vec3 uColor;
varying vec4 vVertex;
varying vec3 vNormal;
uniform mat4 uLightMatrix;
uniform vec3 uPointLightLocation;
void main(void) {
   // получаем вектор направления освещения
   vec4 lightLocation = uLightMatrix * vec4(uPointLightLocation, 1.0);
   vec3 lightDirection = lightLocation.xyz - vVertex.xyz;
   // нормализованный вектор направления освещения
   vec3 L = normalize(lightDirection);
   // нормализуем нормаль, переданную из вершинного шейдера
   vec3 N = normalize(vNormal);
   // находим силу света по Ламберту
   float lambertComponent = max(dot(N, -L), 0.0);
   // получаем итоговый цвет из цвета объекта и освещения
   vec3 diffuseLight = uColor * lambertComponent;
   gl_FragColor = vec4(diffuseLight, 1.0);
}
```

Результат:

Рис. 11.6

Модель отражения Фонга

Если модель освещения Ламберта учитывает только одну составляющую - диффузное отражение, то модель Фонга является более сложной и состоит из трех компонент: фоновое освещение, диффузное освещение и блики. То есть в модели Фонга отраженный свет представлен как сумма рассеянного, диффузного и зеркального отражений.

Фоновое освещение присутствует в любом уголке сцены и никак не зависит от каких-либо источников света. В реальной жизни сторона объекта, на которую не падают прямые солнечные лучи, все равно не будет полностью темной. За счет многократного отражения лучей света от окружающих объектов какая-то часть освещения попадет и на теневую сторону. В 3D графике рассчитывать все отраженные лучи — очень затратно, поэтому есть просто фоновое освещение, которое равномерно «освещает» каждый объект сцены со всех сторон.

Фоновое отражение света (ambient light) представляет собой отражение при естественном освещении. Его можно выразить формулой

$$I = K_a * I_a,$$

где K_a - цветовые параметры материала в виде значений RGB, I_a – интенсивность цветовых компонент фонового света также в виде значений RGB.

Диффузное освещение — тот свет, который зависит от угла **α** между вектором света и нормали и отражается от поверхности во всех направлениях. На эту роль отлично подходит рассмотренная ранее модель Ламберта.

При диффузном отражении света (diffuse light) лучи отражаются под несколькими углами, а не под одним, как при зеркальном отражении. Диффузное отражение характерно прежде всего для неровных шершавых поверхностей.

Диффузное отражение света высчитывается по формуле

$$I = K_d * I_d * \max(\cos \boldsymbol{\alpha}, 0).$$

Здесь кроме коэффициента диффузии материала K_d и интенсивности цветовых компонент освещения I_d присутствует дополнительный параметр, который учитывает направление направленного на поверхность луча. Угол **α** представляет собой угол между нормалью поверхности и вектором направления луча света. Наличие функции максимума, которая выбирает максимальное число из косинуса угла и нуля позволяет обнулить отрицательные значения.

Параметр $\cos(\boldsymbol{\alpha})$ определяет скалярное произведение векторов N (вектор нормали) и L (вектор, представляющий луч света). Таким образом, зная эти вектора, мы можем рассчитать диффузное отражение (для определения скалярного произведения в языке шейдеров есть специальная функция dot).

Зеркальное отражение света (specular light) характеризуется тем, что падающий луч света отражается под одним углом. Зеркальное отражение рассчитывается по формуле

$$I = K_s * I_s \max(\cos \theta, 0)^a.$$

В данном случае K_s определяет материал, а I_s - цвет зеркального отражения. Угол θ здесь представляет угол между вектором, направленным от точки к наблюдателю, и вектором отражаемого луча. Степень **a** (коэффициент яркости) указывает на блеск материала.

Схематично формулу можно представить себе так:

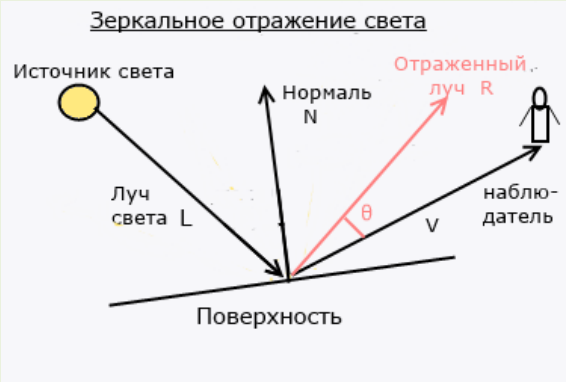

Рис. 11.7

Вновь cos θ определяет скалярное произведение векторов R (вектор отраженного луча) и V (вектор, направленный к наблюдателю).

Язык шейдеров имеет встроенную функцию reflect, которая позволяет найти вектор отраженного луча по нормали и направлению падающего света.

Блики имеют наибольшую силу, когда отраженный свет попадает точно в наблюдателя. Представим себе зеркало. Свет будет ослепляющим, когда будет отражаться точно нам в глаза. Но стоит немного изменить наклон зеркала — и света не будет видно совсем. Как и в случае с зеркалом сила бликов тоже значительно зависит от угла α между вектором отражения света R и вектором наблюдателя V.

В случае, когда нормализованные вектора R и V совпадают, их скалярное произведение (то есть cos θ) будет равняться единице и яркость бликов будет максимальной. С увеличением угла θ скалярное произведение векторов начинает стремиться к нулю, а коэффициент яркости дополнительно уменьшает значение произведения. Получается, что чем больше коэффициент яркости, тем меньше по площади и четче будут наши блики.

Ниже приведен код программы для модели отражения Фонга.

Вершинный шейдер:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
varying vec3 vNormal;
varying vec4 vVertex;
void main(void) {
   // рассчитываем нормаль и положение вершины с учетом трансформаций
   vNormal = normalize(uNMatrix * aVertexNormal);
   vVertex = uMVMatrix * vec4(aVertexPosition, 1.0);
   gl_Position = uPMatrix * vVertex;
}
```

Фрагментный шейдер:

```
uniform vec3 uColor;
varying vec4 vVertex;
```

```
varying vec3 vNormal;
varying vec4 vFinalColor;
uniform mat4 uLightMatrix;
uniform vec3 uPointLightLocation;
const float shiness = 30.0;
// фоновое освещение остается постоянным
const vec3 ambientLightColor = vec3(0.1, 0.1, 0.1);
void main(void) {
   // получаем вектор освещения
   vec4 lightLocation = uLightMatrix * vec4(uPointLightLocation, 1.0);
   vec3 lightDirection = lightLocation.xyz - vVertex.xyz;
   // освещения по Ламберту в качестве диффузного освещения
   vec3 N = normalize(vNormal);
   vec3 L = normalize(lightDirection);
   float lambertComponent = max(dot(N, -L), 0.0);
   vec3 diffuseLight = uColor * lambertComponent;
   // рассчитываем блики
   vec3 eyeVec = -vec3(vVertex.xyz);
   vec3 R = normalize(eyeVec);
   vec3 E = reflect(L, N);
   float specular = pow(max(dot(E, R), 0.0), shiness);
   vec3 specularLight = uColor * specular;
   // итоговый цвет
   vec3 sumColor = ambientLightColor + diffuseLight + specularLight;
   gl_FragColor = vec4(sumColor, 1.0);
}
```

Результат:

Рис. 11.8

Модель отражения Блинна

Также известная как модель Блинна-Фонга, она является разновидностью модели Фонга. Вместо вычисления скалярного произведения между отраженным лучом и наблюдателем, в модели Блинна находится скалярное произведение среднего вектора между наблюдателем и источником света и вектора наблюдателя. А так как найти средний вектор между наблюдателем и источником света (фактически это просто нормализованная сумма двух векторов) с точки зрения вычислений проще, чем найти вектор отражения, модель Блинна более распространена в применении и используется во многих графических системах по умолчанию.

Приведем код программы. Здесь основное отличие от модели Фонга заключается в вычислении halfwayVector.

Вершинный шейдер:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
varying vec3 vNormal;
varying vec4 vVertex;
void main(void) {
   // рассчитываем нормаль и положение вершины с учетом трансформаций
   vNormal = normalize(uNMatrix * aVertexNormal);
   vVertex = uMVMatrix * vec4(aVertexPosition, 1.0);
   gl_Position = uPMatrix * vVertex;
}
```

Фрагментный шейдер:

```
uniform vec3 uColor;
varying vec4 vVertex;
varying vec3 vNormal;
varying vec4 vFinalColor;
uniform mat4 uLightMatrix;
uniform vec3 uPointLightLocation;
const float shiness = 30.0;
// фоновое освещение остается постоянным
const vec3 ambientLightColor = vec3(0.1, 0.1, 0.1);
void main(void) {
   // получаем вектор освещения
   vec4 lightLocation = uLightMatrix * vec4(uPointLightLocation, 1.0);
   vec3 lightDirection = lightLocation.xyz - vVertex.xyz;
   // освещение по Ламберту в качестве диффузного освещения
   vec3 N = normalize(vNormal);
   vec3 L = normalize(lightDirection);
   float lambertComponent = max(dot(N, -L), 0.0);
   vec3 diffuseLight = uColor * lambertComponent;
   // находим средний вектор между освещением и наблюдателем
   vec3 eyeVec = -vec3(vVertex.xyz);
   vec3 R = normalize(eyeVec);
   vec3 halfwayVector = normalize(-L + R);
   float specular = pow(max(dot(halfwayVector, N), 0.0), shiness);
   vec3 specularLight = uColor * specular;
   // итоговый цвет
   vec3 sumColor = ambientLightColor + diffuseLight + specularLight;
   gl_FragColor = vec4(sumColor, 1.0);
}
```

Блики выглядят немного более размыто по сравнению с Фонгом (исправляется бо́льшим значением shiness):

Рис. 11.9

Сэл-шейдерная модель

Можно также встретить название тун-шейдерная модель. Эта модель дает изображение, имитирующее рисование вручную. Использование этой модели можно встретить, например, в мультфильмах с упрощенным рисованием персонажей. Идея шейдера довольно простая: мы берем за основу модель Ламберта, а затем вводим резкие переходы в освещении от порога к порогу. Например, если посмотреть в коде фрагментного шейдера ниже, при значениях коэффициента Ламберта от 0.5 до 0.95 цвет умножается на коэффициент 0.7 - таким образом мы убираем плавное сглаживание в освещении.

Вершинный шейдер:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
varying vec4 vVertex;
varying vec3 vNormal;
void main(void)
{
   // рассчитываем нормаль и положение вершины с учетом трансформаций
   vNormal = normalize(uNMatrix * aVertexNormal);
   vVertex = uMVMatrix * vec4(aVertexPosition, 1.0);
   gl_Position = uPMatrix * vVertex;
}
```

Фрагментный шейдер:

```
uniform vec3 uColor;
varying vec4 vVertex;
varying vec3 vNormal;
uniform mat4 uLightMatrix;
uniform vec3 uPointLightLocation;
void main(void)
{
   // получаем вектор направления освещения
   vec4 lightLocation = uLightMatrix * vec4(uPointLightLocation, 1.0);
   vec3 lightDirection = lightLocation.xyz - vVertex.xyz;
```

```
    // нормализованный вектор направления освещения
    vec3 L = normalize(lightDirection);
    // нормализуем нормаль, переданную из вершинного шейдера
    vec3 N = normalize(vNormal);
    // находим силу света по Ламберту
    float lambertComponent = max(dot(N, -L), 0.0);
    // делаем резкие границы между порогами освещения
    vec3 diffuseLight;
    if (lambertComponent > 0.95) {
        diffuseLight = uColor;
    } else if (lambertComponent > 0.5) {
        diffuseLight = uColor * 0.7;
    } else if (lambertComponent > 0.2) {
        diffuseLight = uColor * 0.2;
    } else {
        diffuseLight = uColor * 0.05;
    }
    gl_FragColor = vec4(diffuseLight, 1.0);
}
```

Результат:

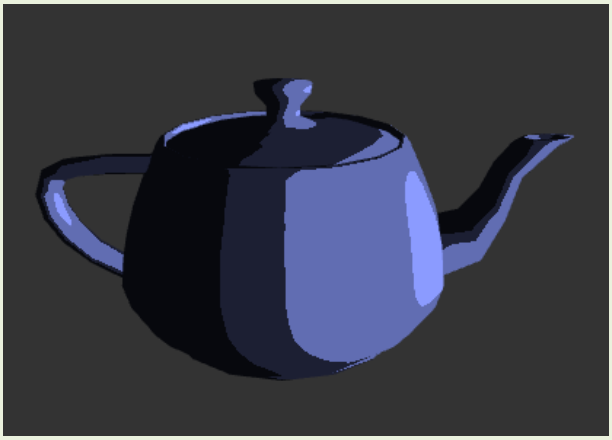

Рис. 11.10

Модель Minnaert

Изначально модель создавалась для моделирования Луны (поэтому иногда можно встретить название «лунный шейдер»), но также подходит для других поверхностей, имеющих корпускулярную или губчатую поверхность.

Вершинный шейдер:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
varying vec4 vVertex;
varying vec3 vNormal;
void main(void) {
    // рассчитываем нормаль и положение вершины с учетом трансформаций
    vNormal = normalize(uNMatrix * aVertexNormal);
```

```
    vVertex = uMVMatrix * vec4(aVertexPosition, 1.0);
    gl_Position = uPMatrix * vVertex;
}
```

Фрагментный шейдер:

```
uniform vec3 uColor;
varying vec4 vVertex;
varying vec3 vNormal;
uniform mat4 uLightMatrix;
uniform vec3 uPointLightLocation;
const float k = 0.7;
void main(void) {
    // получаем вектор направления освещения
    vec4 lightLocation = uLightMatrix * vec4(uPointLightLocation, 1.0);
    vec3 lightDirection = lightLocation.xyz - vVertex.xyz;
    // нормализованный вектор направления освещения
    vec3 L = normalize(lightDirection);
    // нормализуем нормаль, переданную из вершинного шейдера
    vec3 N = normalize(vNormal);
    // вектор наблюдателя
    vec3 eyeVec = -vec3(vVertex.xyz);
    vec3 R = normalize(eyeVec);
    // находим коэффициенты согласно формуле Minnaert
    float p1 = pow(max(dot(N, -L), 0.0), 1.0 + k);
    float p2 = pow(1.0 - dot(N, R), 1.0 - k);
    vec3 diffuseLight = uColor * p1 * p2;
    gl_FragColor = vec4(diffuseLight, 1.0);
}
```

Результат:

Рис. 11.11

Программный код, реализующий все перечисленные модели с выбором цвета материала и модели отражения – в файле **ex11_01.html**

Результат:

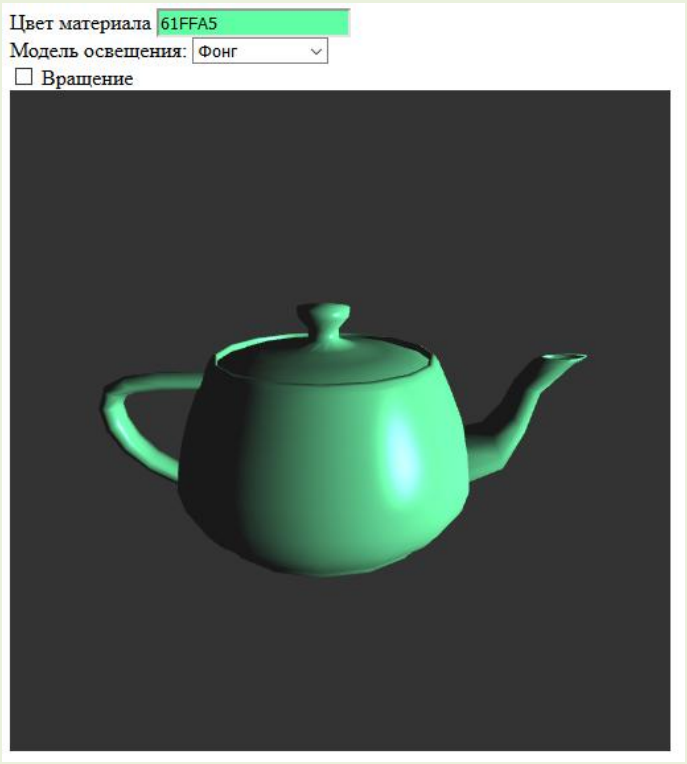

Рис. 11.12

11.3. Методы интерполяции света

Если модель отражения света описывает, *как* будет рассчитываться свет на основании нормалей, материала и источника освещения, то модель интерполяции света (или модель затенения) описывает, *где* будет происходить расчет.

Модель интерполяции Гуро

В модели интерполяции Гуро расчет цвета (с учетом освещения) происходит в вершинном шейдере — то есть итоговый цвет вычисляется для каждой вершины, но не для каждого пикселя. В итоге, так как вся работа по вычислению цвета происходит в вершинном шейдере, фрагментный шейдер принимает лишь одну переменную - итоговый цвет, который равномерно интерполируется между вершинами для каждого пикселя. Интерполяция производится автоматически и не требует никаких действий со стороны программиста.

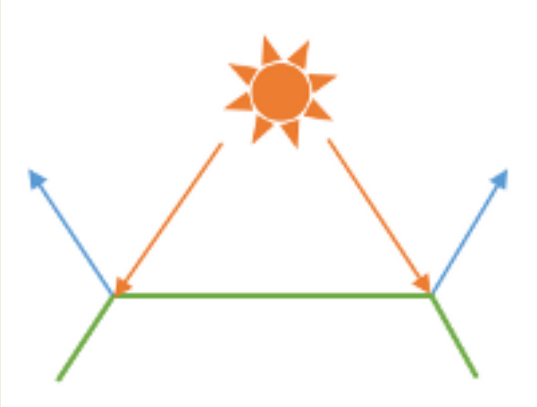

Рис. 11.13

Модель интерполяции Фонга

Кроме модели отражения Фонга, существует и модель интерполяции Фонга. В этой модели цвет вычисляется для каждого пикселя. В итоге освещение получается более качественным и реалистичным, но и требует, соответственно, больше ресурсов со стороны

видеокарты. Нормали по-прежнему рассчитываются в вершинном шейдере, затем равномерно интерполируются для использования в каждом пикселе вершинного шейдера. Кроме нормалей, все остальные вычисления для получения итогового цвета производятся в каждом пикселе, за счет чего и достигается высокое качество освещения.

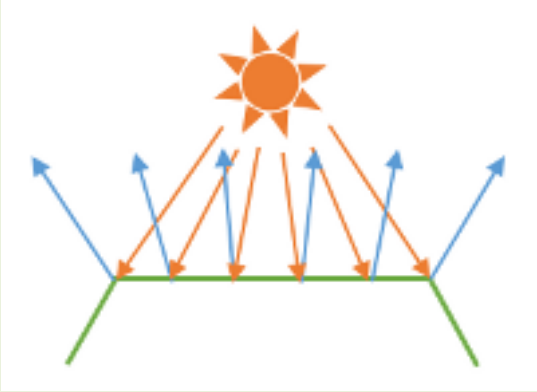

Рис. 11.14

11.4. Освещение объектов в WebGL

Мы добавим освещение к тору, который был выгружен из Blender (см. тему 16). Будем использовать две модели отражения и две модели интерполяции. Тогда мы получим четыре варианта освещения: Ламберт-Гуро, Ламберт-Фонг, Фонг-Гуро, Фонг-Фонг. Рассмотрим каждый из них по порядку.

Так как мы будем использовать разные наборы шейдеров, нам понадобится несколько программ. В коде обратите внимание на функции createProgram и changeProgram.

Модель отражения Ламберта + модель интерполяции Гуро

Это один из самых простых случаев освещения. Основная работа происходит в вершинном шейдере — его и рассмотрим:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
uniform vec3 uColor;
varying vec4 vFinalColor;
// направление освещения
const vec3 vLightDirection = vec3(2.0, -1.0, -1.0);
void main(void) {
   // получаем нормаль с учетом вращения фигуры
   vec3 N = normalize(uNMatrix * aVertexNormal);
   // нормализованный вектор освещения
   vec3 L = normalize(vLightDirection);
   // находим силу света по Ламберту
   float lambertComponent = max(dot(N, -L), 0.0);
   // получаем итоговый цвет из цвета объекта и освещения
   vec3 diffuseLight = uColor * lambertComponent;
   vFinalColor = vec4(diffuseLight, 1.0);
   gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
```

lambertComponent — это и есть искомый коэффициент освещения. На случай, когда произведение векторов возвращает отрицательное значение, компонент устанавливается в значение 0. После того, как мы рассчитали итоговый цвет с учетом освещения, мы передаем его во фрагментный шейдер в varying-переменной vFinalColor. Код фрагментного шейдера в данном случае выглядит совсем просто:

```
varying vec4 vFinalColor;
void main(void) {        gl_FragColor = vFinalColor;     }
```

Модель отражения Ламберта + модель интерполяции Фонга

Отличие этой модели в том, что мы рассчитываем освещение в каждом пикселе. На этот раз вершинный шейдер получается простым и содержит лишь расчет нормалей, которые затем передаются во фрагментный шейдер в переменной vNormal:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
varying vec3 vNormal;
void main(void) {
   // получаем нормаль с учетом вращения фигуры
   vNormal = normalize(uNMatrix * aVertexNormal);
   gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
```

Вершинный шейдер теперь содержит логику расчета освещения:

```
uniform vec3 uColor;
varying vec3 vNormal;
// направление освещения
const vec3 vLightDirection = vec3(2.0, -1.0, -1.0);
void main(void) {
   // нормализованный вектор освещения
   vec3 L = normalize(vLightDirection);
   // нормализуем нормаль, переданную из вершинного шейдера
   vec3 N = normalize(vNormal);
   // находим силу света по Ламберту
   float lambertComponent = max(dot(N, -L), 0.0);
   // получаем итоговый цвет из цвета объекта и освещения
   vec3 diffuseLight = uColor * lambertComponent;
   gl_FragColor = vec4(diffuseLight, 1.0);
}
```

Длина вектора vNormal в ходе интерполяции может не равняться единице, поэтому нужно нормализовать его перед использованием.

Хотя освещение рассчитывается для каждого пикселя, на реальных объектах разница будет не очень заметной по сравнению с вершинным освещением - разве что границы света и тени станут немного чётче. Разница между вершинным и попиксельным освещением будет хорошо заметна на модели отражения Фонга, о которой дальше и пойдет речь.

Модель отражения Фонга + модель интерполяции Гуро

К вершинному шейдеру добавлено множество комментариев, чтобы была понятна каждая строка. Фоновое освещение установлено нулевым - то есть фактически у нас нет фонового освещения и неосвещенные участки будут черными. Можете изменить это значение и посмотреть, как будет выглядеть сцена при этом. Код вершинного шейдера:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
uniform vec3 uColor;
varying vec4 vFinalColor;
// направление освещения
const vec3 vLightDirection = vec3(2.0, -1.0, -1.0);
// блеск материала
const float shiness = 20.0;
void main(void) {
   vec4 vertex = uMVMatrix * vec4(aVertexPosition, 1.0);
   // фоновое освещение
   vec3 ambientLight = vec3(0.0, 0.0, 0.0);
   // блок расчета освещения по Ламберту - диффузное освещение
   vec3 N = normalize(uNMatrix * aVertexNormal);
   vec3 L = normalize(vLightDirection);
   float lambertComponent = max(dot(N, -L), 0.0);
   vec3 diffuseLight = uColor * lambertComponent;
   // рассчитываем блики...
   // так как наша камера в центре системы координат,
   // вектор камеры - просто обратный вектор до точки
   vec3 eyeVec = -vec3(vertex.xyz);
   vec3 R = normalize(eyeVec);
   // функция reflect рассчитает за нас отраженный вектор
   vec3 E = reflect(L, N);
   // сила бликов согласно формуле
   float specular = pow(max(dot(E, R), 0.0), shiness);
   // наконец, получаем последний компонент - блики
   vec3 specularLight = uColor * specular;
   // итоговый цвет = фоновое + диффузное + блики
   vec3 sumColor = ambientLight + diffuseLight + specularLight;
   vFinalColor = vec4(sumColor, 1.0);
   gl_Position = uPMatrix * vertex;
}
```

Фрагментный шейдер в модели Гуро совсем прост:

```
varying vec4 vFinalColor;
void main(void) {       gl_FragColor = vFinalColor;     }
```

Модель отражения Фонга + модель интерполяции Фонга

В вершинном шейдере мы рассчитываем нормали и положение вершины с учетом трансформаций, которые в дальнейшем пойдут во фрагментный шейдер. У нас нет возможности рассчитать эти значения во фрагментном шейдере, так как расчет использует

атрибуты вершин aVertexNormal и aVertexPosition, которые доступны только в вершинном шейдере:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
varying vec3 vNormal;
varying vec4 vVertex;
void main(void) {
   // уже привычная нормаль
   vNormal = normalize(uNMatrix * aVertexNormal);
   // вершина с учетом трансформаций - можем получить только в вершинном
       шейдере
   vVertex = uMVMatrix * vec4(aVertexPosition, 1.0);
   gl_Position = uPMatrix * vVertex;
}
```

Фрагментный шейдер для этой модели:

```
precision mediump float;
uniform vec3 uColor;
varying vec4 vVertex;
varying vec3 vNormal;
varying vec4 vFinalColor;
// направление освещения
const vec3 vLightDirection = vec3(2.0, -1.0, -1.0);
const float shiness = 20.0;
void main(void) {
   // фоновое освещение
   vec3 ambientLight = vec3(0.0, 0.0, 0.0);
   // блок расчета освещения по Ламберту - диффузное освещение
   vec3 N = normalize(vNormal);
   vec3 L = normalize(vLightDirection);
   float lambertComponent = max(dot(N, -L), 0.0);
   vec3 diffuseLight = uColor * lambertComponent;
   // рассчитываем блики...
   // так как наша камера в центре системы координат,
   // вектор камеры - просто обратный вектор до точки
   vec3 eyeVec = -vec3(vVertex.xyz);
   vec3 R = normalize(eyeVec);
   // функция reflect рассчитает за нас отраженный вектор
   vec3 E = reflect(L, N);
   // сила бликов согласно формуле
   float specular = pow(max(dot(E, R), 0.0), shiness);
   // наконец, получаем последний компонент - блики
   vec3 specularLight = uColor * specular;
   // итоговый цвет = фоновое + диффузное + блики
   vec3 sumColor = ambientLight + diffuseLight + specularLight;
   gl_FragColor = vec4(sumColor, 1.0);
}
```

В модели отражения Фонга очень хорошо заметно разницу между вершинным и попиксельным освещением. Можно сравнить картинку слева, где расчет идет в вершинах согласно модели Гуро, и справа, где расчет идет в каждом пикселе:

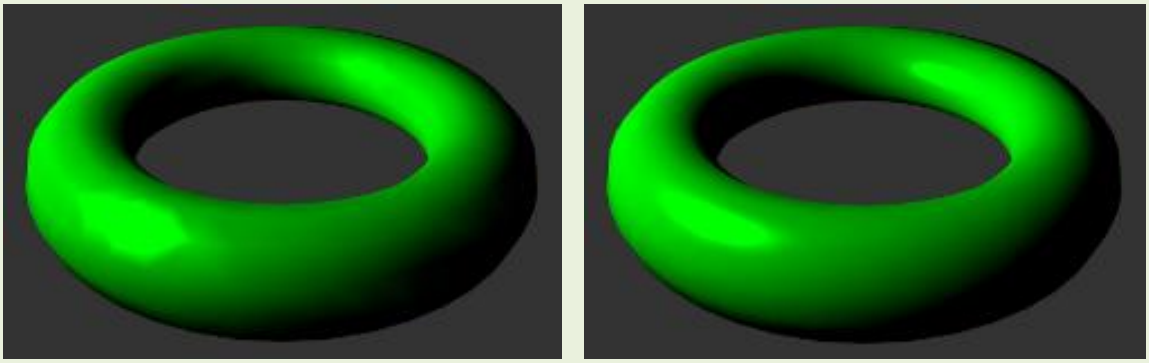

Рис. 11.15

Программный код, реализующий перечисленные сочетания моделей отражения и интерполяции, с выбором цвета материала– в файле **ex11_02.html**

Результат:

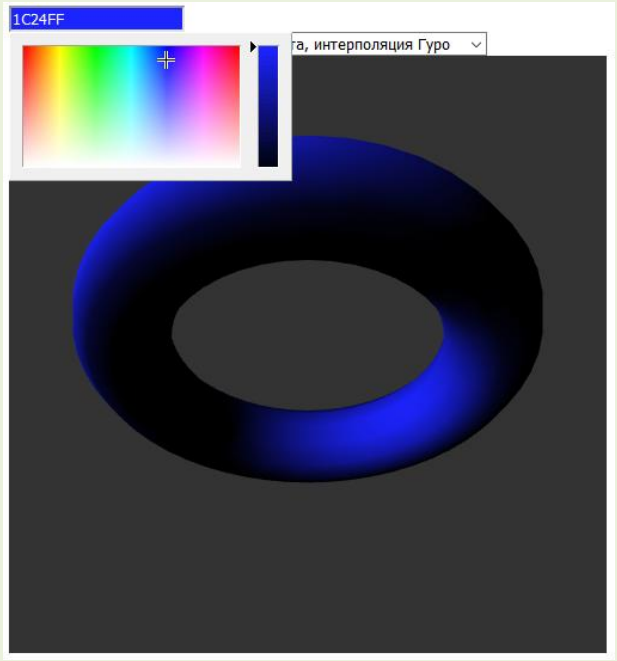

Рис. 11.16

Контрольные вопросы

1. Модели освещения Фонга и Ламберта
2. Модели затенения Фонга и Гуро
3. Использование материалов при расчете освещения в WebGL.