Лабораторная работа №6:
Программируемый графический конвейер

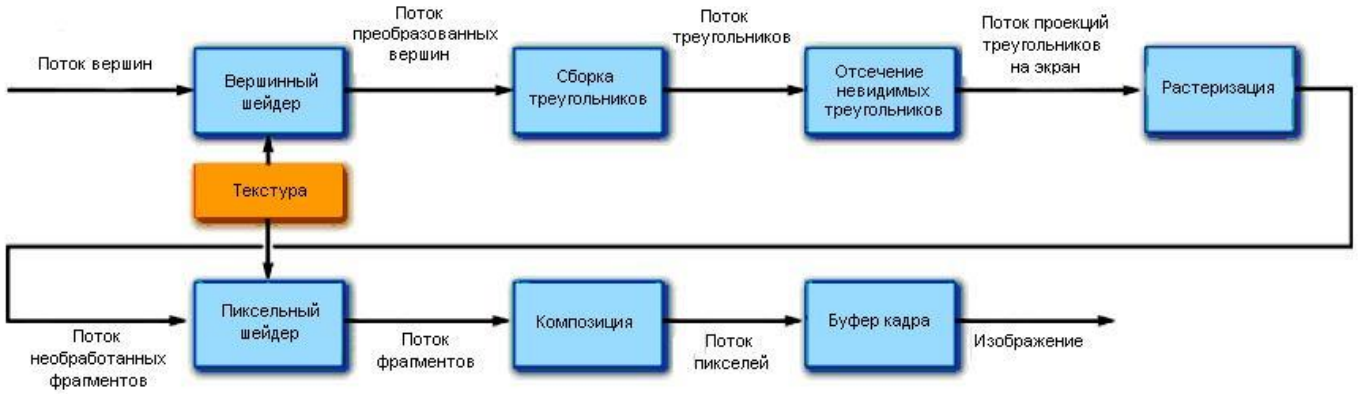
Цель:

Целью данной работы, является знакомство с процессом написания вершинных и фрагментных подпрограмм для графического процессора на языке GLSL.

Справочная информация:
текстуры, используемые в лабораторной работе:
https://github.com/mrdoob/three.js/tree/master/examples/textures/planets
справочник по GLSL:
https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)
редактор GLSL для VSCode:
https://marketplace.visualstudio.com/items?itemName=raczzalan.webgl-glsl-editor

Краткая справка:

Для отображения трёхмерной сцены на экран монитора, необходимо выполнить проекцию объектов этой сцены на плоскость (растеризацию). Процесс, отвечающий за такую проекцию, называется Графическим Конвейером (rendering pipeline). Как показано на рисунке ниже, на вход конвейера передаются вершины и текстуры, а на выходе получается изображение, готовое для отображения на экране:



Для каждого этапа, существует стандартный набор операций, выполняемый по умолчанию. Однако, современные графические API и устройства поддерживают программируемый конвейер, который позволяет самостоятельно описать действия, выполняемые с вершинами, поступающими на вход графического конвейера, и пикселями непосредственно перед формированием выходного изображения.

Шейдеры:

Шейдером, называется программа для графического процессора (GPU).

Существует два основных типа шейдеров:

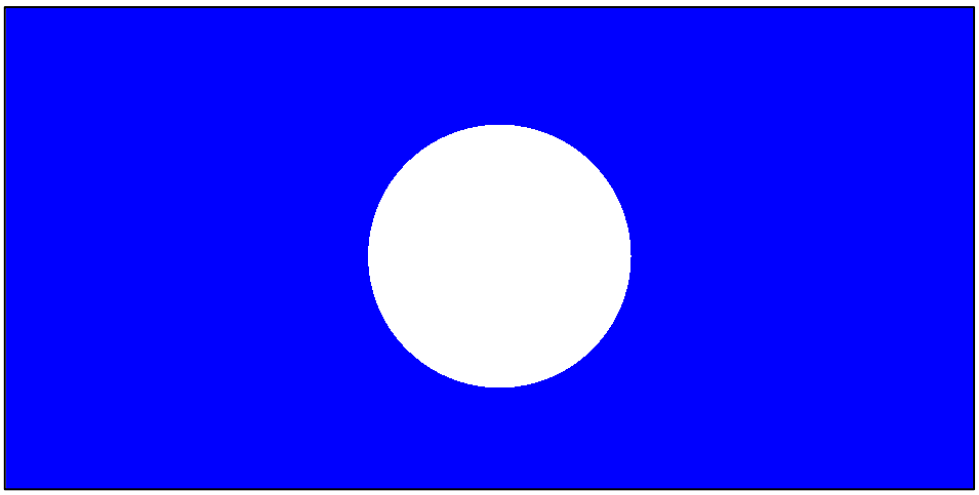
1. Вершинный – отвечает за обработку геометрии сцены, отправленной на растеризацию.
2. Пиксельный/фрагментный – отвечает за обработку пикселей/фрагментов изображения.

В библиотеке Three.js, использовать шейдеры можно при помощи класса ShaderMaterial.

Если создать минимальное приложение Three.js и добавить в сцену сферу:

```
var geometry = new THREE.SphereGeometry( 15, 64, 64 );
var material = new THREE.MeshBasicMaterial( { color: 0xffffff } );
var sphere = new THREE.Mesh( geometry, material );
scene.add( sphere );
```

то на экране должно получиться изображение подобное представленному ниже:



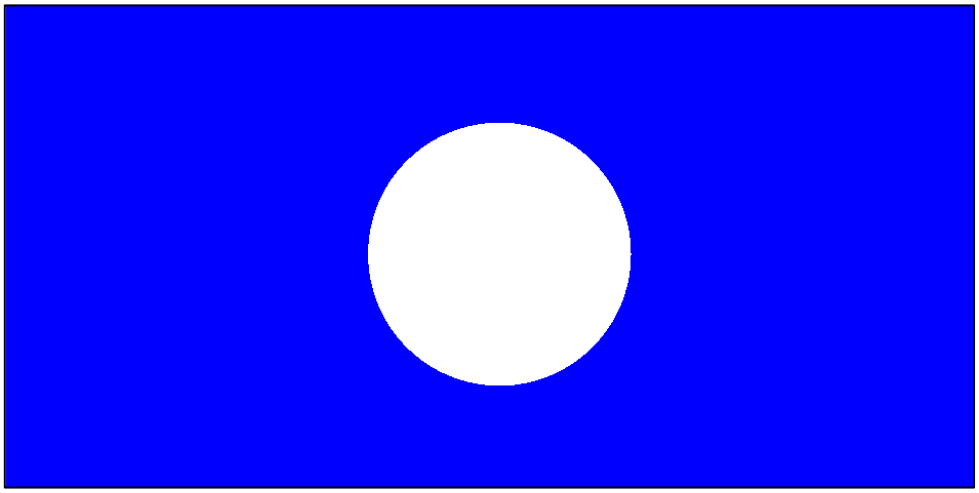
Получить тот же результат, используя программируемый конвейер можно описав шейдеры в html файле:

```
<!-- ============================Simple  Shader======================================== -->
<!-- ============================VertexShader========================================== -->
<script id="vertexShader" type="x-shader/x-vertex">
    void main()
    {
        gl_Position = projectionMatrix * modelViewMatrix * vec4(position,1.0);
    }
</script>
<!-- ============================FragmentShader======================================== -->
<script id="fragmentShader" type="x-shader/x-fragment">
    void main()
    {
        gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
</script>
================================================================================== -->
<!--подключение скрипта с графической программой-->
<script type="module" src="js/main.js"></script>
```

А затем, создав ShaderMaterial и использовать его в качестве материала для сферы:

```
var shaderMaterial = new THREE.ShaderMaterial({
    vertexShader:   document.getElementById('vertexShader').textContent,
    fragmentShader: document.getElementById('fragmentShader').textContent
});

var sphere = new THREE.Mesh( geometry, shaderMaterial );
scene.add( sphere );
```

Если всё было сделано правильно, то на экране будет изображение вида:



Оба шейдера содержат функцию main, а также возвращаемые параметры:

- gl_Position – итоговая позиция вершины в сцене. Получается путём умножения позиции вершины объекта на матрицы мира, вида и проекции. Имеет формат – вектор, состоящий из 4х элементов (x, y, z, w).
- gl_FragColor – цвет пикселя итогового изображения. Имеет формат – вектор, состоящий из 4х элементов (r, g, b, a).

Вершинный шейдер по умолчанию принимает из основной программы следующие параметры вершины:

```
attribute vec3 position;    //позиция вершины
attribute vec3 normal;      //нормаль вершины
attribute vec2 uv;          //текстурные координаты вершины
```

а также набор матриц камеры и объекта:

```
uniform mat4 modelMatrix;       //матрица мира объекта (позиция + поворот + масштаб)
uniform mat4 modelViewMatrix;   //матрица вида * матрица мира
uniform mat4 projectionMatrix;  //матрица проекции
uniform mat4 viewMatrix;        //матрица вида
```

Кроме того, в шейдерах можно объявлять переменные следующих типов:

GLSL type	JavaScript type	Size
float	Number	1
vec2	THREE.Vector2	2
vec3	THREE.Vector3	3
vec3	THREE.Color	3
vec4	THREE.Vector4	4

Кроме того, данные из вершинного шейдера могут передаваться в фрагментный при помощи переменных вида:

```
varying vec3 vNormal;
```

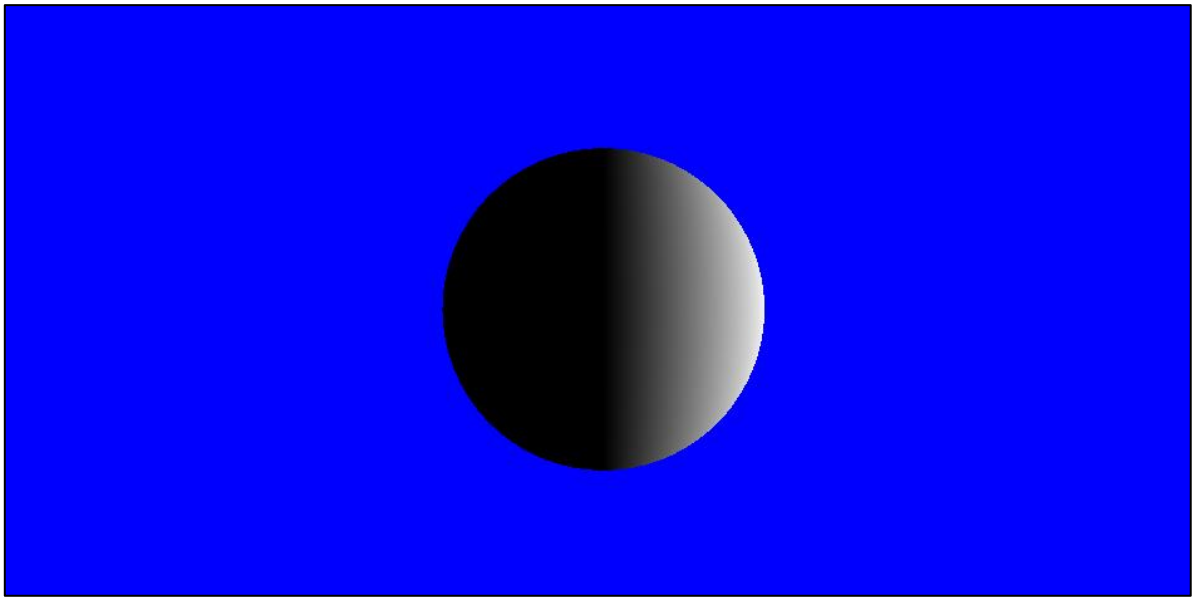
если переменная была объявлена и в вершинном и в пиксельном шейдере, то данные, записанные в неё в первом, будут доступные во втором.

Расчёт освещения:

Использовать такой механизм передачи данных можно, например, для расчёта освещения. Если модифицировать шейдеры следующим образом:

```
<!-- ==========================Lighting Shader======================================= -->
<!-- ============================VertexShader======================================== -->
<script id="vertexShader" type="x-shader/x-vertex">
    varying vec3 vNormal; //переменная для передачи нормали

    void main() {
        //умножение нормали на матрицу мира модели для учёта всех преобразований
        vNormal = normalize(vec3(modelMatrix * vec4(normal, 1.0)));

        gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    }
</script>
<!-- ============================FragmentShader====================================== -->
<script id="fragmentShader" type="x-shader/x-fragment">
    varying vec3 vNormal;

    void main() {
        vec3 lightDir = vec3(1.0, 0.0, 0.0); //направление света
        lightDir = normalize(lightDir);

        float k = max(0.0, dot(vNormal, lightDir)); //коэффициент освещения
        vec3 color = vec3(1.0, 1.0, 1.0) * k; //цвет объекта с учётом освещения
        gl_FragColor = vec4(color, 1.0);
    }
</script>
<!-- ================================================================================== -->
```

Если всё было сделано правильно, должно получиться изображение подобное:



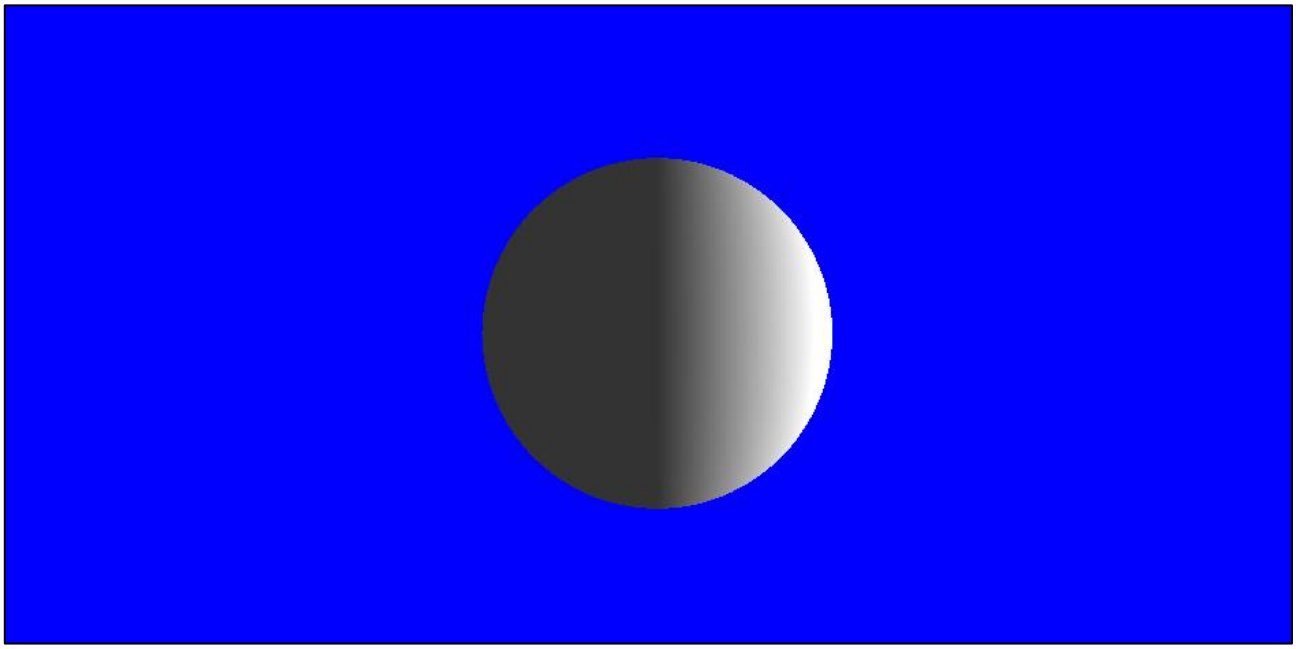
Передать параметры из основной программы в шейдеры можно при помощи переменных типа uniform. Например, можно передать цвет объекта, цвет света, позицию источника освещения и фоновое освещение изменив шейдеры следующим образом:

```
<!-- ============================Lighting Shader======================================= -->
<!-- ============================VertexShader========================================== -->
<script id="vertexShader" type="x-shader/x-vertex">
    uniform vec3 lightPosition; //позиция источника света

    varying vec3 vNormal;
    varying vec3 lightDir;

    void main() {
        vec3 vPos = vec3(modelMatrix * vec4(position, 1.0));    //позиции вершины
        lightDir = lightPosition - vPos;                        //вектор освещения
        lightDir = normalize(lightDir);

        vNormal = normalize(vec3(modelMatrix * vec4(normal, 1.0))); //нормаль

        gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    }
</script>
<!-- ============================FragmentShader======================================== -->
<script id="fragmentShader" type="x-shader/x-fragment">
    uniform vec4 color;         //цвет объекта
    uniform vec4 lightColor;    //цвет света
    uniform vec4 ambientColor;  //цвет фонового освещения

    varying vec3 vNormal;
    varying vec3 lightDir;

    void main() {
      float k = max(0.0, dot(vNormal, lightDir));

      gl_FragColor = color * (lightColor * k + ambientColor);
    }
</script>
<!-- ================================================================================== -->
```

Передать параметры можно добавив в ShaderMaterial блок uniforms и описав переменные соответствующих типов:

```
var shaderMaterial = new THREE.ShaderMaterial({
    uniforms: {
        lightPosition: { value: new THREE.Vector3(10000.0, 0.0, 0.0) },
        color: { value: new THREE.Vector4(1.0, 1.0, 1.0, 1.0) },
        ambientColor: { value: new THREE.Vector4(0.2, 0.2, 0.2, 1.0) },
        lightColor: { value: new THREE.Vector4(1.0, 1.0, 1.0, 1.0) }
    },
    vertexShader:   document.getElementById('vertexShader').textContent,
    fragmentShader: document.getElementById('fragmentShader').textContent
});
```

В результате должно получиться изображение следующего вида:



Изменяя цвет объекта/цвет освещения/цвет фонового освещения, вы сможете изменить цвет итогового изображения.

Изменение геометрии:

На данном этапе, может возникнуть вопрос – “Для чего это нужно, если стандартные материалы делают тоже самое, да ещё и без необходимости писать лишний код?”.

Например, программируемый конвейер позволяет вносить изменения в геометрию объектов прямо на этапе их растеризации. Если к уже существующей сфере, добавить дополнительный атрибут, отвечающий за смещение каждой вершины в произвольном направлении:

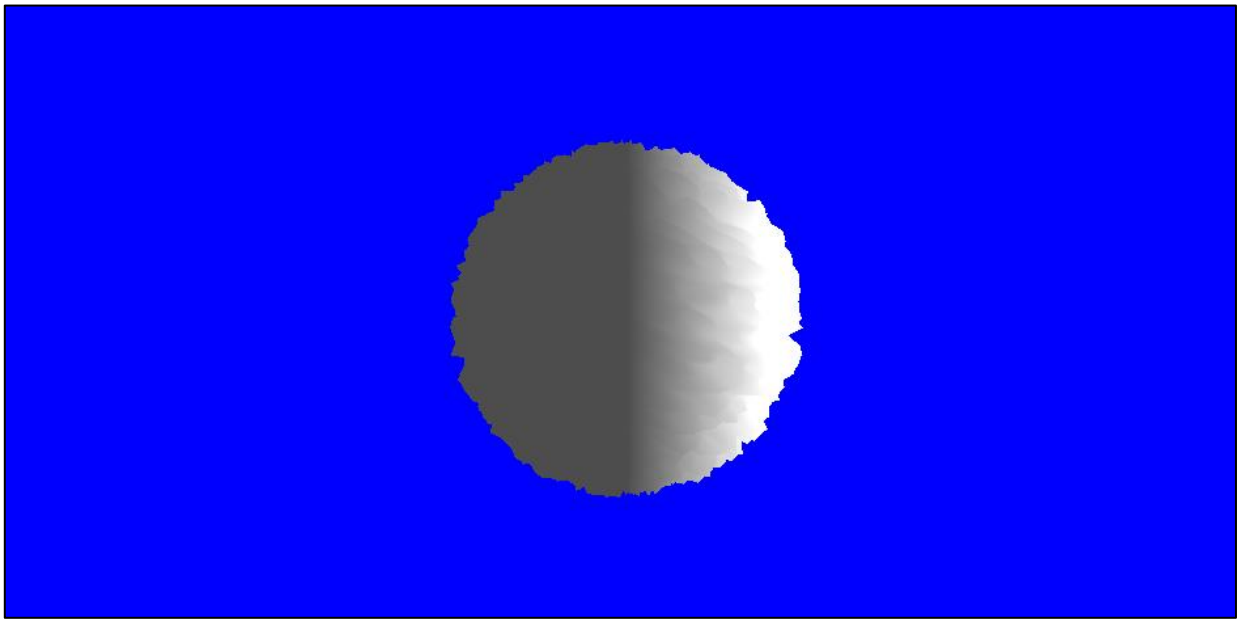
```
//создание списка смещений для вершин
var displacement = new Float32Array(geometry.attributes.position.array.length);

for (var i = 0; i < geometry.attributes.position.array.length; i++)
{
    var rand = (Math.random() * 2) - 1; //случайное значение в диапазоне от -1 до 1
    displacement[i] = rand;
}
//установка списка смещений в качестве атрибута геометрии
geometry.setAttribute("displacement", new THREE.BufferAttribute (displacement, 1));
```

А затем описать этот список атрибутов в вершинном шейдере и добавить смещение позиции вершины:

```
<script id="vertexShader" type="x-shader/x-vertex">
    attribute float displacement;   //смещение для вершины

    uniform vec3 lightPosition; //позиция источника освещения
    varying vec3 vNormal;
    varying vec3 lightDir;

    void main()
    {
```

```
        vNormal = vec3(modelMatrix * vec4(normal, 1.0));
        vNormal = normalize(vNormal);

        vec3 vPos = vec3(modelMatrix * vec4(position, 1.0));    //позиции вершины
        lightDir = lightPosition - vPos;
        lightDir = normalize(lightDir);

        vec3 newPosition = position + normal * vec3(displacement); //смещение вершины

        gl_Position = projectionMatrix * modelViewMatrix * vec4(newPosition,1.0);
    }
</script>
```

То изображение изменится следующим образом:



При желании, смещение можно анимировать, добавив параметр амплитуды изменения в ShaderMaterial:

```
    var shaderMaterial = new THREE.ShaderMaterial({
        uniforms: {
            lightPosition: { value: light.position },
            color: { value: new THREE.Vector3(1.0, 1.0, 1.0) },
            lightColor: { value: new THREE.Vector3(1.0, 1.0, 1.0) },
            ambientColor: { value: new THREE.Vector3(0.3, 0.3, 0.3) },
            amp: { value: 1.0 } //аплитуда
        },
        vertexShader:   document.getElementById('vertexShader').textContent,
        fragmentShader: document.getElementById('fragmentShader').textContent
    });
```

И изменение этого параметра в функции animate:

```
sphere.material.uniforms.amp.value = Math.cos(time)*5;
```

где time – переменная содержащее время, прошедшее с начала работы программы.

Формула расчёта позиции вершины объекта изменится на:

```
vec3 newPosition = position + normal * vec3(displacement*amp); //смещение вершины
```

Смешивание текстур:

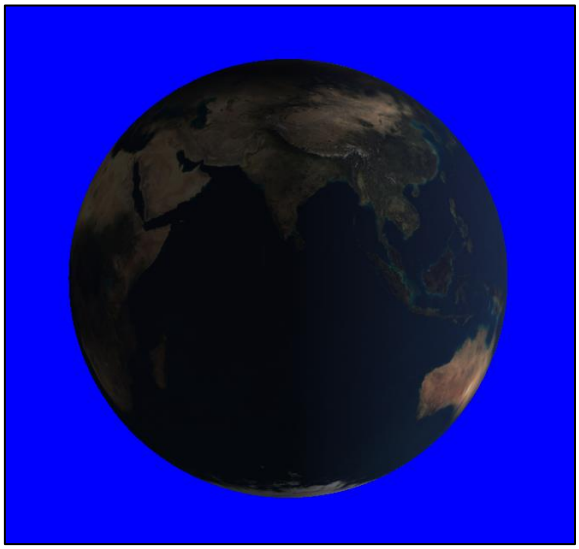
Передать текстуру в шейдер можно загрузив её в переменную и передав в блоке uniform:

```
//загрузка текстуры
var earthTex = new THREE.TextureLoader().load( 'imgs/earth_atmos_2048.jpg' );

var shaderMaterial = new THREE.ShaderMaterial({
    uniforms: {
        dTex: { value: earthTex },  //текстура
        …
    },
});
```

Для получения цвета, необходимо передать в фрагментный шейдер текстурные координаты и использовать функцию texture2D:

```
<!-- ============================VertexShader================================== -->
<script id="vertexShader" type="x-shader/x-vertex">
    varying vec2 vUv; //текстурные координаты
    …
    void main() {
        vUv = uv;
        …
    }
</script>
<!-- ============================FragmentShader================================== -->
<script id="fragmentShader" type="x-shader/x-fragment">
    uniform sampler2D dTex; //текстура
    varying vec2 vUv;
    …
    void main() {
        //получение цвета из текстуры по текстурным координатам
        vec4 color = texture2D( dTex, vUv );
        …
    }
</script>
```

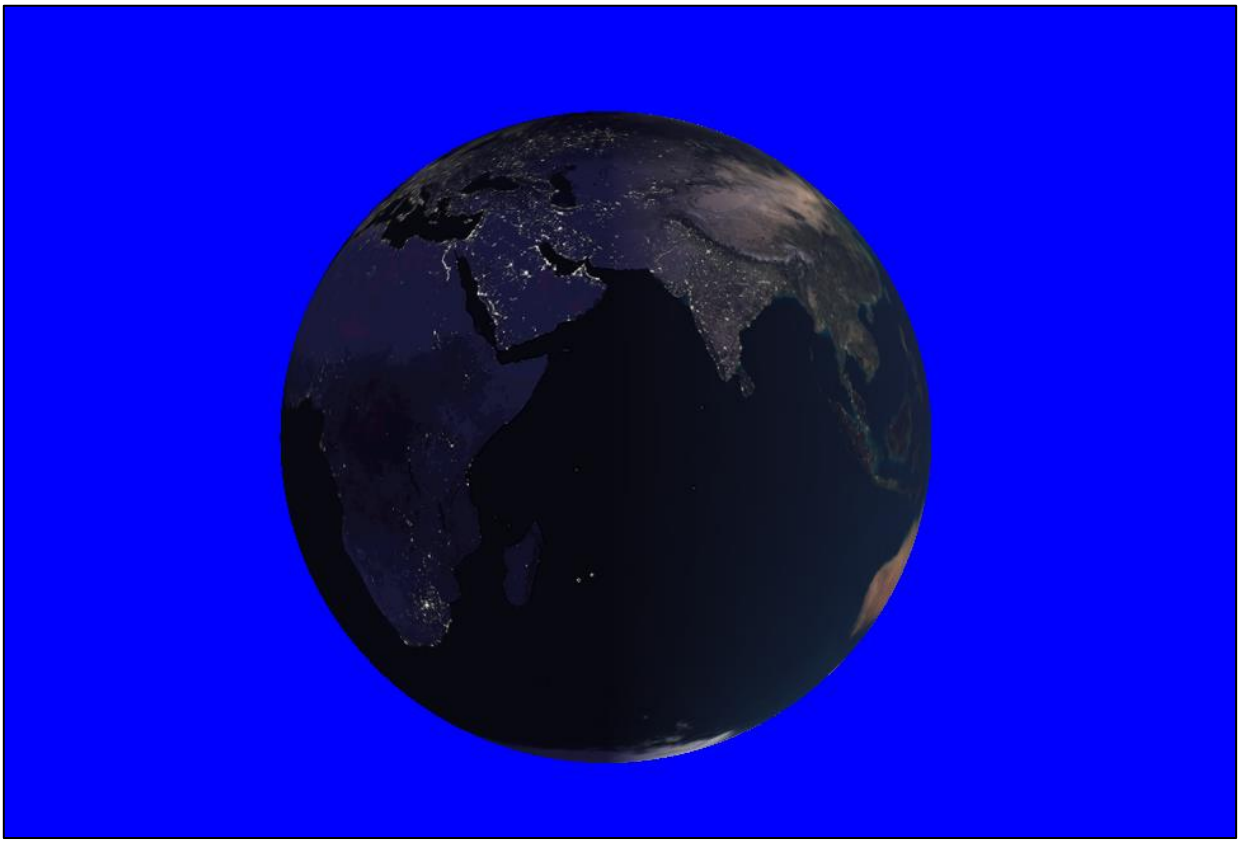
Результат:



По аналогии, можно передать в шейдер вторую текстуру, содержащую карту ночной земли, а затем описать отображение текстур, в зависимости от угла между нормалью и вектором до источника освещения (ночная текстура на тёмной стороне и дневная на светлой):

```
<!-- ============================VertexShader=================================== -->
<script id="vertexShader" type="x-shader/x-vertex">
    varying vec3 vNormal;   //нормаль
    varying vec2 vUv;       //текстурные координаты
    varying vec3 FragPos;   //позиция вершины

    void main() {
        vUv = uv;

        vNormal = vec3(modelMatrix * vec4(normal, 1.0));
        vNormal.y = 0.0;    //устранение влияния отклонения нормалей по оси y
        vNormal = normalize(vNormal);

        FragPos = vec3(modelMatrix * vec4(position, 1.0));

        gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    }
</script>
<!-- ============================FragmentShader================================= -->
<script id="fragmentShader" type="x-shader/x-fragment">
    uniform sampler2D dTex; //текстура для дневной стороны
    uniform sampler2D nTex; //текстура для ночной стороны
    uniform vec3 lightPosition; //позиция источника освещения

    varying vec3 vNormal;
    varying vec2 vUv;
    varying vec3 FragPos;

    void main() {
        vec3 dColor = texture2D( dTex, vUv ).rgb; //цвет дневной текстуры
        vec3 nColor = texture2D( nTex, vUv ).rgb; //цвет ночной текстуры

        //вычисления вектора до источника освещения
        vec3 lightDir = lightPosition - FragPos;
        lightDir = normalize(lightDir);

        //вычисления угла между нормалью и вектором до источника освещения
        float k = max(0.0, dot(vNormal, lightDir));

        //смешивание текстур в зависимости от коэффициента освещения
        vec3 color = (dColor * k) + (nColor * (1.0-k));

        if (k > 0.0) //вывод цвета в зависимости от коэффициента освещения
            gl_FragColor = vec4(color, 1);
        else
            gl_FragColor = vec4(nColor, 1);
    }
</script>
```

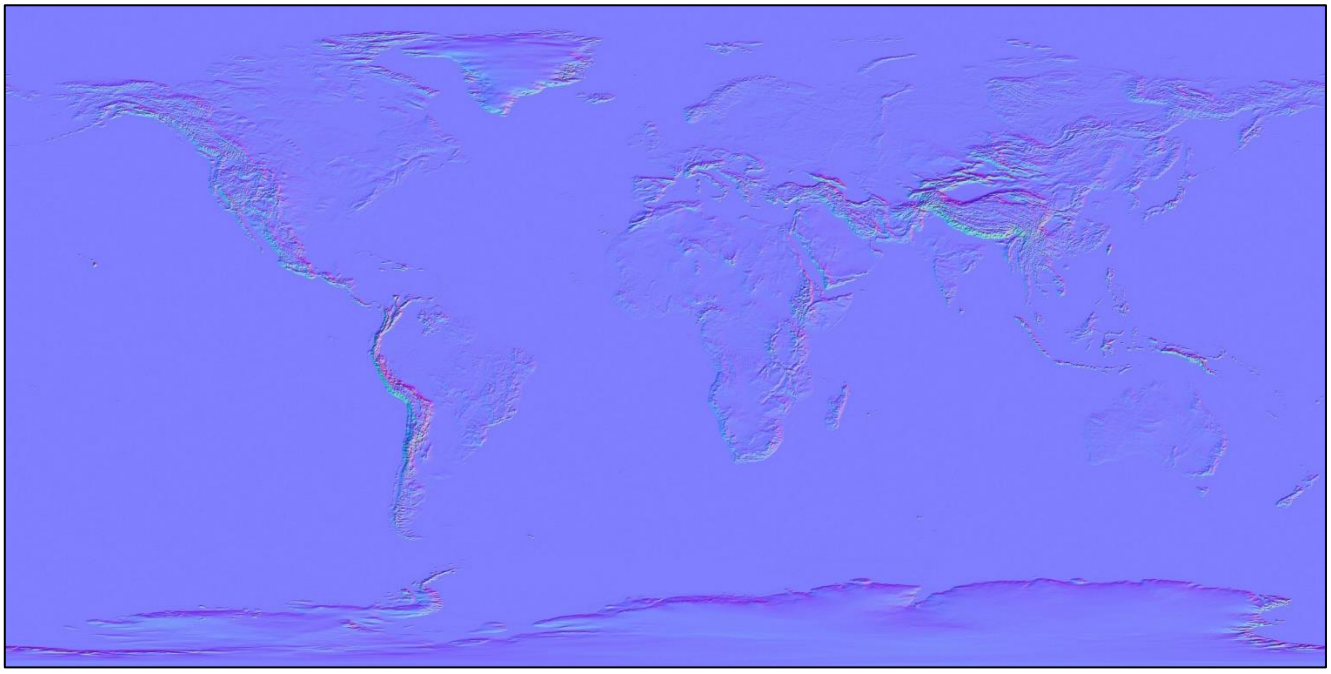
В результате, на тёмной стороне сферы будет отображаться текстура ночной земли, а на освещённой стороне текстура дневная:



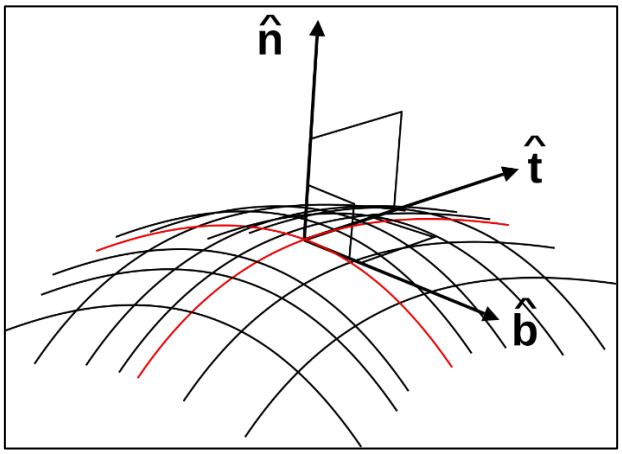
Карты нормалей:

В рассмотренных выше примерах использовались нормали вершин объекта, однако, используя карты нормалей, можно задать нормаль для каждого текселя объекта. Картой нормалей называется текстура, которая вместо значений цвета (rgb) содержит вектора нормалей (xyz). Текселем называется минимальная единица текстуры трёхмерного объекта, пиксел текстуры.

Пример карты нормалей:



При работе с нормалями вершин, преобразования объекта применялись путём перемножения нормалей на матрицу мира этого объекта. В случае с картой нормалей, необходимо построить локальную систему координат для каждого тексела объекта, в которой осью y будет являться нормаль, осью x будет являться тангент, а осью z – бинормаль (битангент):



В библиотеке Three.js, рассчитать тангенты можно используя стандартный метод геометрии:

```
geometry.computeTangents();
```

в случае, если тангенты были рассчитаны в не правильную сторону, можно их развернуть:

```
for (var i = 0; i < geometry.attributes.tangent.array.length; i++)
{
    geometry.attributes.tangent.array[i] = -geometry.attributes.tangent.array[i];
}
```

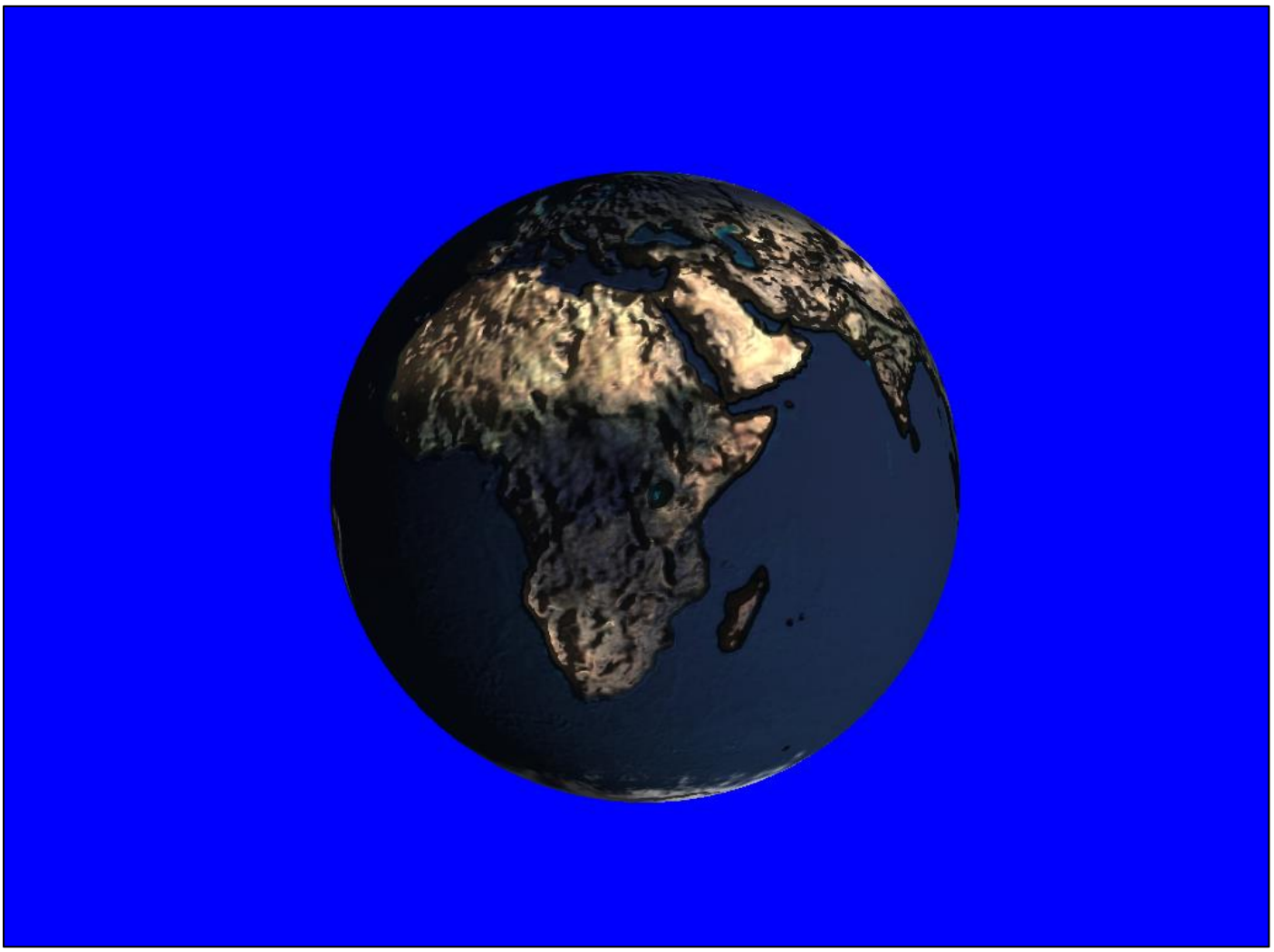
В вершинном шейдере необходимо рассчитать бинормаль и применить преобразования:

```
<!-- ============================VertexShader========================================== -->
<script id="vertexShader" type="x-shader/x-vertex">
    uniform vec3 lightPosition;

    varying vec3 L;   //вектор до источника освещения
    varying vec2 vUv;
    varying mat3 tbn; //матрица тангент/бинормаль/нормаль

    attribute vec3 tangent; //тангент вершины

    void main() {
       vUv = uv;
       vec3 mvPosition = (modelMatrix * vec4(position,1.0)).xyz;
       //вычисление бинормали и применение преобразований к системе координат
       vec3 norm = normalize((modelMatrix * vec4(normal,0.0)).xyz);
       vec3 tang = normalize((modelMatrix * vec4(tangent, 0.0)).xyz);
       vec3 bitang = normalize((modelMatrix * vec4(cross(norm, tang), 0.0)).xyz);
       tbn = mat3( tang, bitang, norm );

       L =  normalize(lightPosition - mvPosition);

       gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
    }
</script>
```

```
<!-- ============================FragmentShader======================================= -->
<script id="fragmentShader" type="x-shader/x-fragment">
    uniform vec3 lightColor;
    uniform vec3 ambientColor;
    uniform sampler2D baseTexture; //текстура
    uniform sampler2D normalMap;   //карта нормалей
    varying vec3 L;
    varying vec2 vUv;
    varying mat3 tbn;

    void main() {
      vec3 totalDiffuse = vec3(0.0);
      //цвет не может быть отрицательным, нормаль может
      vec3 normalMapValue = 2.0 * texture2D(normalMap, vUv).rgb - 1.0;

      //в некоторых случаях, необходимо инвертировать r или g компоненту карты нормалей
      normalMapValue.r = -normalMapValue.r;
      vec3 unitNormal = normalize(normalMapValue);
      unitNormal = unitNormal * tbn; //применение преобразований к нормали

      float nDotl = dot(unitNormal, L);
      float brightness = max(nDotl,0.0); //вычисление коэффициента освещения

      totalDiffuse = ambientColor + (brightness * lightColor);
      vec4 textureColour = texture2D(baseTexture,vUv); //вычисление цвета

      gl_FragColor = vec4(vec3(totalDiffuse),1.0) * textureColour;
    }
</script>
```

Если всё было сделано правильно, на сфере должна появиться иллюзия микрорельефа:

Результат с картой нормалей сгенерированной при помощи https://cpetry.github.io/NormalMap-Online/



Задание:

1. Реализовать примеры, описанные в лабораторной работе.
2. Реализовать приложение, совмещающее и смешивание текстур дневной и ночной земли, и использование карты нормалей для земли.