Санкт-Петербургский Государственный
Электротехнический Университет

Кафедра МОЭВМ

Задание для лабораторной работы № 3
"3D освещение"

Выполнил: Калмак Д.А.
Факультет: ФКТИ
Группа: 0303
Преподаватель: Герасимова Т.В.

Санкт-Петербург
2025 г.

Цель работы.

- освоить освещение 3D объектов и 3D сцены.
- проанализировать полученное задание, выделить информационные объекты и действия.
- определить трехмерный объект на основе простой геометрической фигуры, добавить нормали к трехмерному объекту, чтобы сделать его подходящим для освещения, добавить второй источник света, добавить позиционный и направленный источник света, попробовать сформировать затенение по Гуро и Фонгу, сравнить их, узнать больше о взаимодействии материалов и свойств освещения и применить это на сцене.

Основные теоретические положения.

Векторные компоненты. Векторы, которые вы отправляете в свои шейдеры, обычно представляют цвета, координаты вершин, нормаль поверхности и координаты текстуры. Поскольку компоненты этих векторов имеют разные значения, GLSL предоставляет специальные средства доступа, которые можно использовать для ссылки на компоненты.

r, g, b, a используется для цветов. красный, зеленый, синий, альфа (коэффициент смешивания)

x, y, z, w используется для пространственных координат, таких как векторы и точки.

s, t, p, q используется для поиска текстур.

Они могут быть добавлены в конец переменной vec для ссылки на один или несколько компонентов или для их смешивания и сопоставления.

Затенение. В WebGL 1.0, который вы изучаете, есть только плавное затенение. При плавном затенении значения цвета интерполируются между вершинами. В

WebGL 2.0, который пока доступен не во всех базарах, вы можете использовать плоскую заливку. Если задано плоское затенение, одна вершина выбирается как репрезентативная для всех вершин; таким образом, весь примитив отображается одним цветом. Для всех примитивов это последняя указанная вершина в каждом многоугольнике или отрезке. Плоское затенение задается ключевым словом flat перед типом данных на выходе из вашего вершинного шейдера и соответствующим входом для вашего фрагментного шейдера.

Ваш вершинный шейдер может выводить больше, чем просто цвет и положение вершин. Любой атрибут вершины может быть интерполирован по мере его отправки в ваш фрагментный шейдер. Если вы решили отправлять и интерполировать только цвета, вы делаете затенение по Гуро. Вы также можете интерполировать нормали и выполнять расчеты освещения в фрагментном шейдере, а не в вершинном шейдере. Это называется затенением Фонг. Не путайте с отражением Фонга, которое вы можете вычислить в вершинном шейдере.

Механизм отражения света от текущей поверхности очень сложен и зависит от многих факторов. Некоторые из них являются геометрическими - например, относительные направления источника света, глаза наблюдателя и нормали к поверхности. Другие факторы относятся к характеристикам поверхности, таким как шероховатость, и к цвету этой поверхности. Модель закрашивания определяет, как свет рассеивается по поверхности или отражается от нее. источники света "сверкают" на различных поверхностях объектов, и падающий свет взаимодействует с поверхностью одним из трех возможных способов:

- некоторая часть поглощается поверхностью и превращается в тепло;
- некоторая часть отражается от поверхности;
- некоторая часть проходит внутрь объекта, как в случае куска стекла.

Если весь падающий свет поглощается, то данный объект воспринимается как черный, поэтому его называют абсолютно черным телом. Если весь свет проходит сквозь объект, то он виден только в результате эффекта рефракции.

Различают два типа отражения падающего света:

- диффузное рассеяние происходит, когда часть падающего света слегка проникает внутрь поверхности и излучается обратно равномерно по всем направлениям. Диффузный свет сильно взаимодействует с поверхностью, поэтому его цвет обычно зависит от природы, из которого сделана эта поверхность;
- зеркальные отражения больше похожи на зеркало и имеют ярко выраженную направленность: падающий свет не поглощается объектом, а отражается прямо от его наружной поверхности. Это порождает блики, и поверхность выглядит блестящей. В простейшей модели зеркального света отраженный свет имеет такой же цвет, что и падающий свет, что делает материал похожим на пластмассу. В более сложной модели цвет отраженного света пробегает интервал бликов, что дает лучшее приближение металлических поверхностей.

Для того чтобы вычислить диффузный и зеркальный компоненты света, необходимо найти три вектора. Углы между этими тремя векторами составляют основу для вычисления интенсивностей освещения. Обычно эти углы вычисляются в мировых координатах, поскольку при некоторых преобразованиях (таких, как перспективное преобразование) углы не сохраняются.

- нормаль m к поверхности в точке P;
- вектор v, соединяющий точку P с глазом наблюдателя;
- вектор s, соединяющий точку P с источником света

Диффузный компонент представляет собой свет, исходящий из одного направления, поэтому он ярче, когда падает прямо на поверхность, чем когда он

просто скользит по поверхности. Как только он сталкивается с поверхностью, он равномерно рассеивается во всех направлениях, так что он кажется равномерно ярким светом, независимо от положения наблюдателя. Любой свет, исходящий из определенного места или направления, должен содержать диффузный компонент. Зеркальный (отраженный) свет исходит из определенного направления и при столкновении с поверхностью отражается в определенном направлении. Реальные объекты не рассеивают свет равномерно во всех направлениях, поэтому к модели закраски добавляются зеркальный компонент. Зеркальное отражение порождает блики, которые могут существенно увеличить реалистичность изображения, заставив объекты блестеть. Фоновое (рассеянное) освещение представляет собой свет, настолько рассеянный в окружающей среде, что его направление невозможно определить.

Уравнения освещения требуют нормалей. Это векторы, которые указывают, в каком направлении обращена поверхность.

Задание.

Разработать программу, реализующую представление куба и пирамиды.

Разработанная программа должна быть пополнена возможностями освещения с учетом нормалей, несколькими источниками света, в том числе направленным и позиционным, затенениями Гуро и Фонг, а также использовать свойства материала и освещения.

Выполнение работы.

Работа выполнена с использованием HTML для веб-страницы, JavaScript и WebGL для логики и рендеринга приложения и gl-matrix – библиотеки для работы с матрицами. Для визуализации разработки использовался браузер Microsoft Edge Версия 135.0.3179.73.

В интерфейсе был создан новый div блок для управлением светом. По умолчанию работает один источник света, но есть checkbox для второго источника света, направленного источника света и позиционного света. Также по умолчанию работает затенение Фонг, но активировав checkbox включится затенение Гуро. Создан также еще один новый div блок для материала фигур: ambient, diffuse, specular и shininess. Причем цвета задают в спектре для удобства, а у shininess подключено численное значение к ползунку, которое меняется.

Были оптимизированы координаты куба для определения корректных нормалей. Также были оптимизированы координаты пирамиды, в том числе основание было разделено на два треугольника. После данных оптимизаций можно использовать функцию makeFlatNormals, которая принимает вершины, индексы начала и конца, а также нормали, которые и будут результатом функции. Поскольку используются не массивы по три значения, а формат подряд идущих чисел, ограничение накладывается на кратность 9, поскольку данная функция работает только с треугольниками. Нормали записаны в массивы cubeNormals и pyramidNormals, и для них созданы буферы cubeNormalBuffer и pyramidNormalBuffer. Определяется местоположение атрибута “normal” в шейдерной программе с помощью метода getAttribLocation(), и включается атрибут enableVertexAttribArray(). Поскольку в пирамиде основание было заменено на два треугольника, в отрисовке используется gl.TRIANGLES. Для куба и пирамиды были также добавлены буферы нормалей.

Поскольку шейдеры дополняются значительным объемом кода, для их компиляции добавлены проверки, чтобы понимать ошибки. Проверка также добавлена для shaderProgram на предмет линковки.

Для направлений всех четырех источников света получаем uniform-переменные из шейдерной программы методом getUniformLocation(). Аналогично для трех источников получаем для цвета uniform-переменные. Затем

получаем uniform-переменную для флага использования затенения Гуро, uniform-переменные для значений материала фигур. В функции updateView() эти переменные все используются. Сначала передается значение положения для первого основного источника света, затем, если checkbox второго света активен, то его цвет и положение, аналогично для направленного и позиционного. Затем передается 1 или 0 значение для флага использования затенения Гуро. Поскольку для удобства пользователя были использованы спектры html color типа input для выбора цвета, значения с помощью функции hexToVec3() переводятся в vec3. Вектора материала: ambient, diffuse, specular и shininess – передаются в шейдеры. Матрицы нормалей были добавлены в предыдущей лабораторной работе.

Чтобы сцена обновлялась в соответствии с checkbox, color и range были добавлены соответствующие обработчики событий, а для shininess также изменение числа рядом со слайдером.

Рассмотрим шейдеры. В вершинном шейдере для обоих затенений переводим координаты в пространство камеры. Преобразуем нормаль в соответствии с матрицей нормали. Далее подходим к двум затенениям: Гуро и Фонг. Если checkbox активирован, то используется затенение Гуро. Их отличия заключаются в том, что у Гуро освещение рассчитывается в вершинном шейдере, а цвет интерполируется между вершинами. У Фонг же интерполируются нормали, а освещение рассчитывается в фрагментном шейдере. Это приводит к повышении точности затенения, но понижении производительности. Поскольку мы рассматриваем вершинный шейдер, то рассмотрим затенение Гуро. У освещения рассматриваем компоненты ambient, diffuse, specular, то есть окружающее освещение, рассеянное освещение и зеркальное освещение, а у материала фигур ambient, diffuse, specular и еще добавляется shininess, то есть блеск. По умолчанию заданы параметры золота, но пользователь через панель управления это может изменить. В сцене может быть четыре источника, поэтому общий цвет получается из суммы. В первом источнике

цвет источника по умолчанию, но используется ambient материала, в других источниках он не используется. У всех источников рассчитываются компоненты рассеянного и зеркального освещения и умножаются на компоненты материала, результат домножается на цвет источника, а затем все суммируется, а затем домножается на цвет вершины. Третий источник реализован в виде направленного – берем вектор от поверхности к свету, а четвертый – в виде позиционного, причем с затуханием – берем вектор от точки на поверхности к источнику. Затухание связано с расстоянием от точки до источника линейно. Таким образом, получен цвет по затенению Гуро и в фрагментном шейдере просто выводится рассчитанный на вершине цвет. Если же checkbox не активирован, то вычисления проводятся уже в фрагментном шейдере для каждого пикселя.

Тестирование.

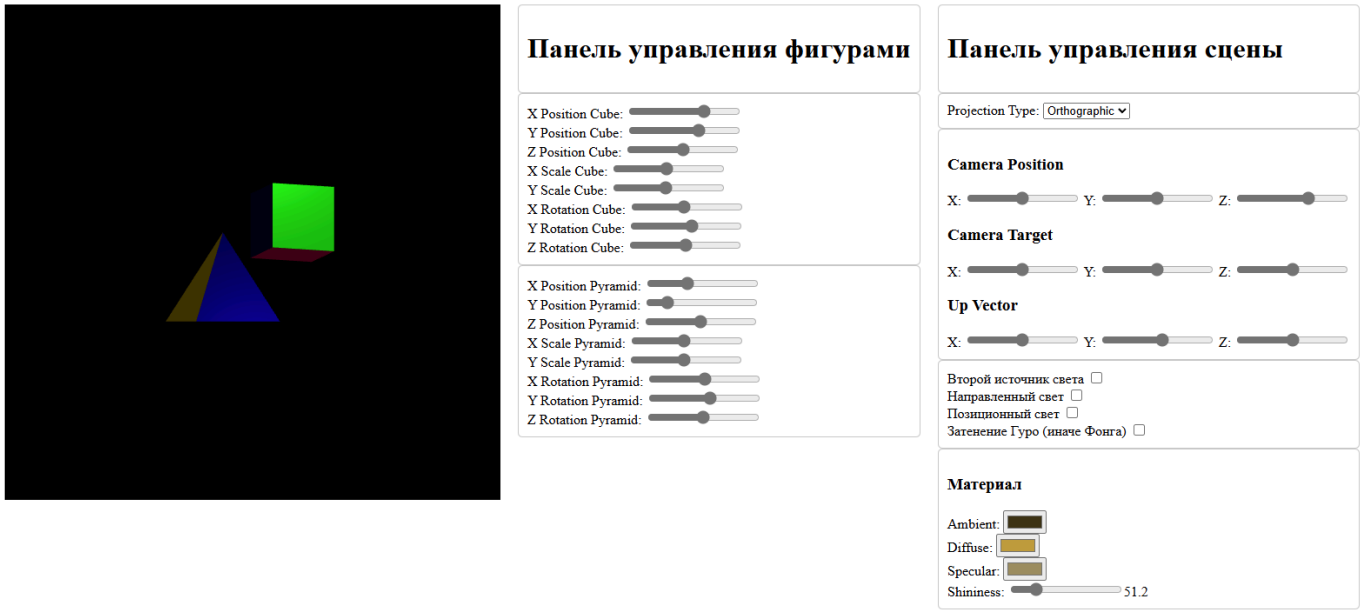

Рисунок 1 – Сцена с кубом и пирамидой с основным освещением

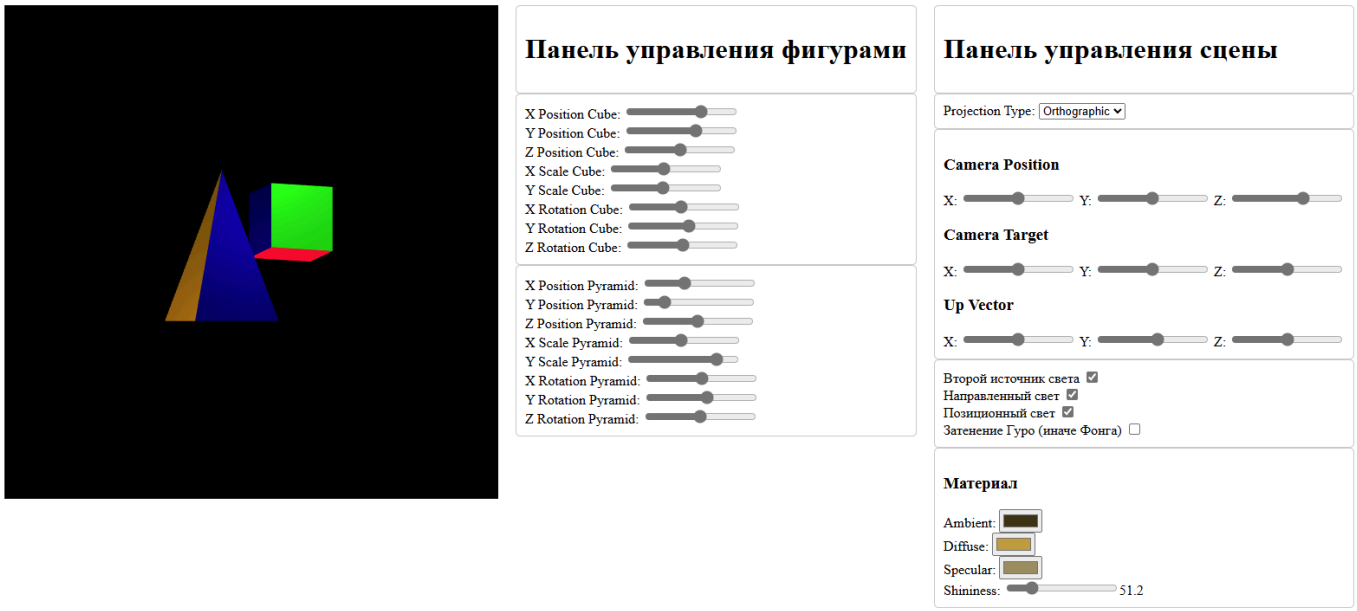

Рисунок 2 – Сцена с кубом и пирамидой со всеми источниками света

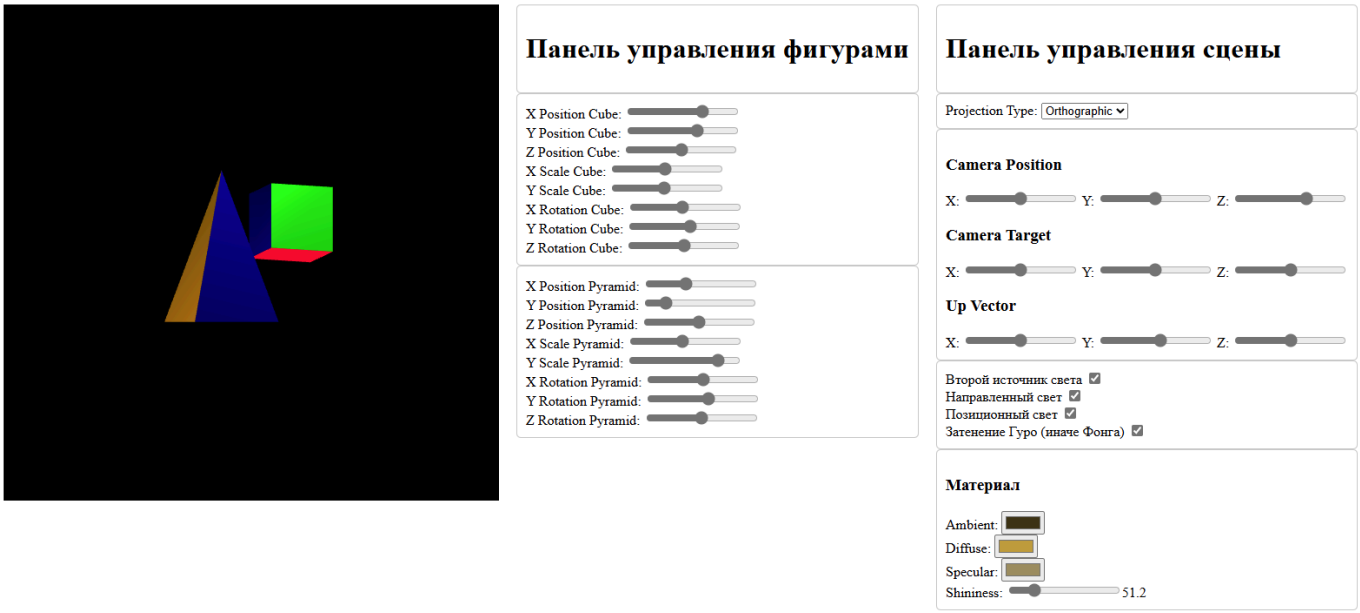

Рисунок 3 – Сцена с кубом и пирамидой со всеми источниками света с затенением Гуро

Вывод.

В результате выполнения лабораторной работы была разработана программа, отображающая геометрические объекты: куб и пирамида вместе с панелями управления фигурами и сценой и освещением. В него входит основной источник света, второй источник света, позиционный и направленный источник света, затенение по Гуро и Фонг, у второго блики получаются красивее, а также использованы материал фигур и разные свойства освещения. Программа работает корректно. При выполнении работы были приобретены навыки работы с графической библиотекой WebGL.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>WebGL CG3D lab3</title>
    </head>
    <body>
        <div class="page">
            <canvas width="570" height="570" id="my_Canvas"></canvas>
            <div class="controls">
                <div class="title-control">
                    <h1>Панель управления фигурами</h1>
                </div>
                <div class="cube-control">
                    <label>X Position Cube:</label>
                    <input type="range" id="xPosCube" min="-1" max="1" step="0.1"
value="0.4">
                    <br>
                    <label>Y Position Cube:</label>
                    <input type="range" id="yPosCube" min="-1" max="1" step="0.1"
value="0.3">
                    <br>
                    <label>Z Position Cube:</label>
                    <input type="range" id="zPosCube" min="-1" max="1" step="0.1"
value="0">
                    <br>
                    <label>X Scale Cube:</label>
                    <input type="range" id="xScaleCube" min="0.1" max="2" step="0.1"
value="1">
                    <br>
                    <label>Y Scale Cube:</label>
                    <input type="range" id="yScaleCube" min="0.1" max="2" step="0.1"
value="1">
                    <br>
                    <label>X Rotation Cube:</label>
                    <input type="range" id="xRotateCube" min="-180" max="180" step="1"
value="-10">
                    <br>
                    <label>Y Rotation Cube:</label>
                    <input type="range" id="yRotateCube" min="-180" max="180" step="1"
value="20">
                    <br>
                    <label>Z Rotation Cube:</label>
                    <input type="range" id="zRotateCube" min="-180" max="180" step="1"
value="0">
                </div>
                <div class="pyramid-control">
                    <label>X Position Pyramid:</label>
                    <input type="range" id="xPosPyramid" min="-1" max="1" step="0.1"
value="-0.3">
                    <br>
                    <label>Y Position Pyramid:</label>
```

```
                    <input type="range" id="yPosPyramid" min="-1" max="1" step="0.1"
value="-0.7">
                    <br>
                    <label>Z Position Pyramid:</label>
                    <input type="range" id="zPosPyramid" min="-1" max="1" step="0.1"
value="0">
                    <br>
                    <label>X Scale Pyramid:</label>
                    <input type="range" id="xScalePyramid" min="0.1" max="2" step="0.1"
value="1">
                    <br>
                    <label>Y Scale Pyramid:</label>
                    <input type="range" id="yScalePyramid" min="0.1" max="2" step="0.1"
value="1">
                    <br>
                    <label>X Rotation Pyramid:</label>
                    <input type="range" id="xRotatePyramid" min="-180" max="180"
step="1" value="0">
                    <br>
                    <label>Y Rotation Pyramid:</label>
                    <input type="range" id="yRotatePyramid" min="-180" max="180"
step="1" value="20">
                    <br>
                    <label>Z Rotation Pyramid:</label>
                    <input type="range" id="zRotatePyramid" min="-180" max="180"
step="1" value="0">
                </div>
            </div>
            <div class="controls">
                <div class="title-control">
                    <h1>Панель управления сцены</h1>
                </div>
                <div class="projection-control">
                    <div class="projection-type">
                        <label>Projection Type:</label>
                        <select id="projectionType">
                            <option value="ortho">Orthographic</option>
                            <option value="perspective">Perspective</option>
                        </select>
                    </div>
                    <div class="perspective-controls">
                        <label>FOV:</label>
                        <input type="range" id="fov" min="0" max="180" step="1"
value="45">
                        <label>Near:</label>
                        <input type="range" id="near" min="0.1" max="10" step="0.1"
value="0.1">
                        <label>Far:</label>
                        <input type="range" id="far" min="1" max="200" step="1"
value="100">
                    </div>
                </div>
                <div class="lookat-control">
                    <h3>Camera Position</h3>
                    <label>X: <input type="range" id="eyeX" min="-15" max="15"
step="0.1" value="0"></label>
```

```
                    <label>Y: <input type="range" id="eyeY" min="-15" max="15"
step="0.1" value="0"></label>
                    <label>Z: <input type="range" id="eyeZ" min="-15" max="15"
step="0.1" value="5"></label>
                    <h3>Camera Target</h3>
                    <label>X: <input type="range" id="centerX" min="-5" max="5"
step="0.1" value="0"></label>
                    <label>Y: <input type="range" id="centerY" min="-5" max="5"
step="0.1" value="0"></label>
                    <label>Z: <input type="range" id="centerZ" min="-5" max="5"
step="0.1" value="0"></label>
                    <h3>Up Vector</h3>
                    <label>X: <input type="range" id="upX" min="-10" max="10"
step="0.1" value="0"></label>
                    <label>Y: <input type="range" id="upY" min="-10" max="10"
step="0.1" value="1"></label>
                    <label>Z: <input type="range" id="upZ" min="-10" max="10"
step="0.1" value="0"></label>
                </div>
                <div class="light-control">
                    <label>Второй источник света <input type="checkbox"
id="enableLight2"></label><br>
                    <label>Направленный свет <input type="checkbox"
id="enableDirLight"></label><br>
                    <label>Позиционный свет <input type="checkbox"
id="enablePointLight"></label><br>
                    <label>Затенение Гуро (иначе Фонга) <input type="checkbox"
id="gouraudToggle"></label>
                </div>
                <div class="material-control" id="material-control">
                    <h3>Материал</h3>
                    <label>Ambient: <input type="color" id="ambientColor"
value="#3F3313"></label><br>
                    <label>Diffuse: <input type="color" id="diffuseColor"
value="#C09B3A"></label><br>
                    <label>Specular: <input type="color" id="specularColor"
value="#A08D5D"></label><br>
                    <label>Shininess: <input type="range" id="shininess" min="1"
max="256" value="51.2"><span id="shininessValue">51.2</span></label>
                </div>
            </div>
        </div>
        <script src="gl-matrix.js"></script>
        <script src="WebGLJavascript.js"></script>
    </body>
    <style>
        .page {
            display: flex;
            align-items: flex-start;
            gap: 20px;
        }

        .title-control, .projection-control, .lookat-control, .light-control,
.material-control, .cube-control, .pyramid-control {
            border: 1px solid #ccc;
            padding: 10px;
            border-radius: 5px;
```

```
        }
    </style>
</html>
```

WebGLJavascript.js

```
let canvas = document.getElementById('my_Canvas');
let gl = canvas.getContext('webgl');

const cubeVertices = [
   // зад (красный)
   -0.33, -0.33, -0.33,  0.33,  0.33, -0.33, 0.33, -0.33, -0.33,
   -0.33, -0.33, -0.33,  -0.33,  0.33, -0.33, 0.33,  0.33, -0.33,

   // перед (зеленый)
   -0.33, -0.33, 0.33,  0.33, -0.33, 0.33,  0.33, 0.33, 0.33,
   -0.33, -0.33, 0.33,  0.33, 0.33, 0.33,  -0.33, 0.33, 0.33,

   // лево (синий)
   -0.33, -0.33, -0.33,  -0.33,  0.33,  0.33,  -0.33,  0.33, -0.33,
   -0.33, -0.33, -0.33,  -0.33, -0.33,  0.33,  -0.33,  0.33,  0.33,

   // право (жёлтый)
   0.33, -0.33, -0.33,   0.33,  0.33, -0.33,   0.33,  0.33,  0.33,
   0.33, -0.33, -0.33,   0.33,  0.33,  0.33,   0.33, -0.33,  0.33,

   // низ (пурпурный)
   -0.33, -0.33, -0.33,   0.33, -0.33, -0.33,   0.33, -0.33,  0.33,
   -0.33, -0.33, -0.33,   0.33, -0.33,  0.33,  -0.33, -0.33,  0.33,

   // верх (голубой)
   -0.33,  0.33, -0.33,   0.33,  0.33,  0.33,   0.33,  0.33, -0.33,
   -0.33,  0.33, -0.33,  -0.33,  0.33,  0.33,   0.33,  0.33,  0.33
];

const cubeColors = [
   // зад (красный)
   1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0, 0.0, 0.0,
   1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0, 0.0, 0.0,

   // перед (зелёный)
   0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0, 1.0, 0.0,
   0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0, 1.0, 0.0,

   // лево (синий)
   0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0, 0.0, 1.0,
   0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0, 0.0, 1.0,

   // право (жёлтый)
   1.0, 1.0, 0.0,  1.0, 1.0, 0.0,  1.0, 1.0, 0.0,
   1.0, 1.0, 0.0,  1.0, 1.0, 0.0,  1.0, 1.0, 0.0,

   // низ (пурпурный)
   1.0, 0.0, 1.0,  1.0, 0.0, 1.0,  1.0, 0.0, 1.0,
   1.0, 0.0, 1.0,  1.0, 0.0, 1.0,  1.0, 0.0, 1.0,

   // верх (голубой)
```

```
   0.0, 1.0, 1.0,  0.0, 1.0, 1.0,  0.0, 1.0, 1.0,
   0.0, 1.0, 1.0,  0.0, 1.0, 1.0,  0.0, 1.0, 1.0
];

const pyramidVertices = [
   // Основание (оранжевый)
   -0.45, 0.0, -0.45,  0.45, 0.0, -0.45,  0.45, 0.0, 0.45,
   -0.45, 0.0, -0.45,  0.45, 0.0, 0.45,  -0.45, 0.0, 0.45,

   // Боковые грани
   // перед (зелёный)
   0.0, 0.9, 0.0,  0.45, 0.0, -0.45, -0.45, 0.0, -0.45,
   // право (красный)
   0.0, 0.9, 0.0,  0.45, 0.0, 0.45, 0.45, 0.0, -0.45,
   // зад (синий)
   0.0, 0.9, 0.0,  -0.45, 0.0, 0.45, 0.45, 0.0, 0.45,
   // лево (жёлтый)
   0.0, 0.9, 0.0,  -0.45, 0.0, -0.45, -0.45, 0.0, 0.45
];

const pyramidColors = [
   // Основание (оранжевый)
   1.0, 0.5, 0.0,  1.0, 0.5, 0.0,  1.0, 0.5, 0.0,
   1.0, 0.5, 0.0,  1.0, 0.5, 0.0,  1.0, 0.5, 0.0,

   // Боковые грани
   // перед (зелёный)
   0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0, 1.0, 0.0,
   // право (красный)
   1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0, 0.0, 0.0,
   // зад (синий)
   0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0, 0.0, 1.0,
   // лево (жёлтый)
   1.0, 1.0, 0.0,  1.0, 1.0, 0.0,  1.0, 1.0, 0.0
];

function makeFlatNormals(vertices, start, num, normals) {
   if (num % 9 !== 0) {
      console.warn("Warning: number of floats is not a multiple of 9");
      return;
   }

   for (let i = start; i < start + num; i += 9) {
      const p0 = vec3.fromValues(vertices[i],   vertices[i+1], vertices[i+2]);
      const p1 = vec3.fromValues(vertices[i+3], vertices[i+4], vertices[i+5]);
      const p2 = vec3.fromValues(vertices[i+6], vertices[i+7], vertices[i+8]);

      const v1 = vec3.create();
      const v2 = vec3.create();
      const n = vec3.create();

      vec3.subtract(v1, p1, p0);
      vec3.subtract(v2, p2, p0);
      vec3.cross(n, v1, v2);
      vec3.normalize(n, n);

      normals.push(...n, ...n, ...n);
```

```
    }
}

const cubeNormals = [];
makeFlatNormals(cubeVertices, 0, cubeVertices.length, cubeNormals);

const pyramidNormals = [];
makeFlatNormals(pyramidVertices, 0, pyramidVertices.length, pyramidNormals);

let matrixStack = [];
function pushMatrix(m) {
    matrixStack.push(mat4.clone(m));
}
function popMatrix() {
    if (matrixStack.length === 0) {
        throw "Ошибка: Стек матриц пуст.";
    }
    return matrixStack.pop();
}
let modelViewMatrix = mat4.create();
let projectionMatrix = mat4.create();
let normalMatrix = mat4.create();

function createBuffer(data) {
    let buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(data), gl.STATIC_DRAW);
    return buffer;
}

let cubeBuffer = createBuffer(cubeVertices);
let pyramidBuffer = createBuffer(pyramidVertices);

let cubeColorBuffer = createBuffer(cubeColors);
let pyramidColorBuffer = createBuffer(pyramidColors);

let cubeNormalBuffer = createBuffer(cubeNormals);
let pyramidNormalBuffer = createBuffer(pyramidNormals);

const vertCode = `
    precision mediump float;

    attribute vec3 coordinates;
    attribute vec3 color;
    attribute vec3 normal;

    varying vec3 vColor;
    varying vec3 vPosition;
    varying vec3 vNormal;
    varying vec3 vLightColorGouraud;

    uniform mat4 modelViewMatrix;
    uniform mat4 projectionMatrix;
    uniform mat4 normalMatrix;

    uniform vec3 light1Direction;
    uniform vec3 light2Color;
```

```
   uniform vec3 light2Direction;
   uniform vec3 dirLightDirection;
   uniform vec3 dirLightColor;
   uniform vec3 pointLightPosition;
   uniform vec3 pointLightColor;

   uniform bool useGouraudShading;

   uniform vec3 materialAmbient;
   uniform vec3 materialDiffuse;
   uniform vec3 materialSpecular;
   uniform float materialShininess;

   void main(void) {
      vec4 pos = modelViewMatrix * vec4(coordinates, 1.0);
      vColor = color;
      vPosition = pos.xyz;
      vec3 transformedNormal = normalize(mat3(normalMatrix) * normal);
      vNormal = transformedNormal;

      if (useGouraudShading) {
         vec3 lightColor = vec3(0.0);
         vec3 viewDir = normalize(-pos.xyz);

         // Источник 1
         vec3 light1 = normalize(light1Direction);
         float diffuse1 = max(dot(transformedNormal, light1), 0.0);
         vec3 reflect1 = reflect(-light1, transformedNormal);
         float spec1 = pow(max(dot(viewDir, reflect1), 0.0), materialShininess);
         lightColor += materialAmbient +
                     materialDiffuse * diffuse1 +
                     materialSpecular * spec1;

         // Источник 2
         vec3 light2Dir = normalize(light2Direction);
         float light2Diffuse = max(dot(transformedNormal, light2Dir), 0.0);
         vec3 light2Reflect = reflect(-light2Dir, transformedNormal);
         float light2Spec = pow(max(dot(viewDir, light2Reflect), 0.0),
materialShininess);
         lightColor += light2Color * (
                        materialDiffuse * light2Diffuse +
                        materialSpecular * light2Spec);

         // Источник 3 — направленный
         vec3 dirDir = normalize(-dirLightDirection);
         float dirDiffuse = max(dot(transformedNormal, dirDir), 0.0);
         vec3 dirReflect = reflect(-dirDir, transformedNormal);
         float dirSpec = pow(max(dot(viewDir, dirReflect), 0.0), materialShininess);
         lightColor += dirLightColor * (
                        materialDiffuse * dirDiffuse +
                        materialSpecular * dirSpec);

         // Источник 4 — позиционный
         vec3 pointDir = normalize(pointLightPosition - pos.xyz);
         float pointDiffuse = max(dot(transformedNormal, pointDir), 0.0);
         vec3 pointReflect = reflect(-pointDir, transformedNormal);
```

```
         float pointSpec = pow(max(dot(viewDir, pointReflect), 0.0),
materialShininess);
         float distance = length(pointLightPosition - pos.xyz);
         float attenuation = 1.0 / (distance);
         lightColor += pointLightColor * attenuation * (
                        materialDiffuse * pointDiffuse +
                        materialSpecular * pointSpec);

         vLightColorGouraud = vColor * lightColor;
         }

      gl_Position = projectionMatrix * pos;
   }
`;

let vertShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertShader, vertCode);
gl.compileShader(vertShader);
if (!gl.getShaderParameter(vertShader, gl.COMPILE_STATUS)) {
   console.error("Vertex Shader Error:\n", gl.getShaderInfoLog(vertShader));
}

const fragCode = `
   precision mediump float;

   varying vec3 vColor;
   varying vec3 vPosition;
   varying vec3 vNormal;
   varying vec3 vLightColorGouraud;

   uniform vec3 light1Direction;
   uniform vec3 light2Color;
   uniform vec3 light2Direction;
   uniform vec3 dirLightDirection;
   uniform vec3 dirLightColor;
   uniform vec3 pointLightPosition;
   uniform vec3 pointLightColor;

   uniform vec3 materialAmbient;
   uniform vec3 materialDiffuse;
   uniform vec3 materialSpecular;
   uniform float materialShininess;

   uniform bool useGouraudShading;

   void main(void) {
      if (useGouraudShading) {
         gl_FragColor = vec4(vLightColorGouraud, 1.0);
      } else {
         vec3 normal = vNormal;
         vec3 viewDir = normalize(-vPosition);
         vec3 lightColor = vec3(0.0);

         // Источник 1
         vec3 light1 = normalize(light1Direction);
         float diffuse1 = max(dot(normal, light1), 0.0);
         vec3 reflect1 = reflect(-light1, normal);
```

```
        float spec1 = pow(max(dot(viewDir, reflect1), 0.0), materialShininess);
        lightColor += materialAmbient +
                    materialDiffuse * diffuse1 +
                    materialSpecular * spec1;

        // Источник 2
        vec3 light2Dir = normalize(light2Direction);
        float light2Diffuse = max(dot(normal, light2Dir), 0.0);
        vec3 light2Reflect = reflect(-light2Dir, normal);
        float light2Spec = pow(max(dot(viewDir, light2Reflect), 0.0),
materialShininess);
        lightColor += light2Color * (
                      materialDiffuse * light2Diffuse +
                      materialSpecular * light2Spec);

        // Источник 3 — направленный
        vec3 dirDir = normalize(-dirLightDirection);
        float dirDiffuse = max(dot(normal, dirDir), 0.0);
        vec3 dirReflect = reflect(-dirDir, normal);
        float dirSpec = pow(max(dot(viewDir, dirReflect), 0.0), materialShininess);
        lightColor += dirLightColor * (
                      materialDiffuse * dirDiffuse +
                      materialSpecular * dirSpec);

        // Источник 4 — позиционный
        vec3 pointDir = normalize(pointLightPosition - vPosition);
        float pointDiffuse = max(dot(normal, pointDir), 0.0);
        vec3 pointReflect = reflect(-pointDir, normal);
        float pointSpec = pow(max(dot(viewDir, pointReflect), 0.0),
materialShininess);
        float distance = length(pointLightPosition - vPosition);
        float attenuation = 1.0 / (distance);
        lightColor += pointLightColor * attenuation * (
                      materialDiffuse * pointDiffuse +
                      materialSpecular * pointSpec);

        vec3 finalColor = vColor * lightColor;
        gl_FragColor = vec4(finalColor, 1.0);
      }
   }
`;

let fragShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragShader, fragCode);
gl.compileShader(fragShader);
if (!gl.getShaderParameter(fragShader, gl.COMPILE_STATUS)) {
   console.error("Fragment Shader Error:\n", gl.getShaderInfoLog(fragShader));
}

let shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertShader);
gl.attachShader(shaderProgram, fragShader);
gl.linkProgram(shaderProgram);
gl.useProgram(shaderProgram);
if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
   console.error("Program Link Error:\n", gl.getProgramInfoLog(shaderProgram));
}
```

```
let coordLocation = gl.getAttribLocation(shaderProgram, "coordinates");
gl.enableVertexAttribArray(coordLocation);
let colorLocation = gl.getAttribLocation(shaderProgram, "color");
gl.enableVertexAttribArray(colorLocation);
let normalLocation = gl.getAttribLocation(shaderProgram, "normal");
gl.enableVertexAttribArray(normalLocation);

let modelViewMatrixLocation = gl.getUniformLocation(shaderProgram, "modelViewMatrix");
let projectionMatrixLocation = gl.getUniformLocation(shaderProgram,
"projectionMatrix");
let normalMatrixLocation = gl.getUniformLocation(shaderProgram, "normalMatrix");

const light1DirectionLocation = gl.getUniformLocation(shaderProgram,
'light1Direction');
const light2ColorLocation= gl.getUniformLocation(shaderProgram, 'light2Color');
const light2DirectionLocation= gl.getUniformLocation(shaderProgram, 'light2Direction');
const dirLightDirectionLocation= gl.getUniformLocation(shaderProgram,
'dirLightDirection');
const dirLightColorLocation= gl.getUniformLocation(shaderProgram, 'dirLightColor');
const pointLightDirectionLocation= gl.getUniformLocation(shaderProgram,
'pointLightPosition');
const pointLightColorLocation= gl.getUniformLocation(shaderProgram, 'pointLightColor');
const useGouraudLocation = gl.getUniformLocation(shaderProgram, "useGouraudShading");
const materialAmbientLocation = gl.getUniformLocation(shaderProgram,
"materialAmbient");
const materialDiffuseLocation = gl.getUniformLocation(shaderProgram,
"materialDiffuse");
const materialSpecularLocation = gl.getUniformLocation(shaderProgram,
"materialSpecular");
const materialShininessLocation = gl.getUniformLocation(shaderProgram,
"materialShininess");

function drawCube(vertexBuffer, colorBuffer, normalBuffer, vertexCount) {
   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
   gl.vertexAttribPointer(coordLocation, 3, gl.FLOAT, false, 0, 0);

   gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
   gl.vertexAttribPointer(colorLocation, 3, gl.FLOAT, false, 0, 0);

   gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
   gl.vertexAttribPointer(normalLocation, 3, gl.FLOAT, false, 0, 0);

   gl.drawArrays(gl.TRIANGLES, 0, vertexCount);
}

function drawPyramid(vertexBuffer, colorBuffer, normalBuffer, vertexCount) {
   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
   gl.vertexAttribPointer(coordLocation, 3, gl.FLOAT, false, 0, 0);

   gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
   gl.vertexAttribPointer(colorLocation, 3, gl.FLOAT, false, 0, 0);

   gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
   gl.vertexAttribPointer(normalLocation, 3, gl.FLOAT, false, 0, 0);

   gl.drawArrays(gl.TRIANGLES, 0, vertexCount);
```

```
}

gl.clearColor(0.0, 0.0, 0.0, 1.0);
gl.enable(gl.DEPTH_TEST);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
gl.viewport(0, 0, canvas.width, canvas.height);

function updateProjection() {
   let aspect = canvas.width / canvas.height;
   let projectionType = document.getElementById('projectionType').value;

   let perspectiveControls = document.querySelector('.perspective-controls');
   perspectiveControls.style.display = (projectionType === 'perspective') ? 'block' :
'none';

   mat4.identity(projectionMatrix);

   if (projectionType === 'perspective') {
      let fov = parseFloat(document.getElementById('fov').value) * Math.PI / 180;
      let near = parseFloat(document.getElementById('near').value);
      let far = parseFloat(document.getElementById('far').value);
      mat4.perspective(projectionMatrix, fov, aspect, near, far);
   } else {
      let distance = parseFloat(document.getElementById('eyeZ').value);
      mat4.ortho(projectionMatrix, -aspect * distance / 2, aspect * distance / 2,
-distance / 2, distance / 2, 0.1, 100.0);
   }

   gl.uniformMatrix4fv(projectionMatrixLocation, false, projectionMatrix);
}

function hexToVec3(hex) {
   const bigint = parseInt(hex.slice(1), 16);
   const r = ((bigint >> 16) & 255) / 255;
   const g = ((bigint >> 8) & 255) / 255;
   const b = (bigint & 255) / 255;
   return vec3.fromValues(r, g, b);
}

function updateView() {
   gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
   gl.viewport(0, 0, canvas.width, canvas.height);

   gl.uniform3fv(light1DirectionLocation, [1.0, 1.0, 1.0]);

   const enablelight2 = document.getElementById('enableLight2').checked;
   const enableDir = document.getElementById('enableDirLight').checked;
   const enablePoint = document.getElementById('enablePointLight').checked;
   const useGouraud = document.getElementById("gouraudToggle").checked;

   // Второй свет
   if (enablelight2) {
      gl.uniform3fv(light2ColorLocation, [0.67, 0.4, 0.8]);
      gl.uniform3fv(light2DirectionLocation, [-1.0, 0.0, -1.0]);
   } else {
      gl.uniform3fv(light2ColorLocation, [0.0, 0.0, 0.0]);
   }
```

```
// Третий — направленный
if (enableDir) {
   gl.uniform3fv(dirLightColorLocation, [1.0, 0.67, 0.2]);
   gl.uniform3fv(dirLightDirectionLocation, [0.0, 1.0, 0.0]);
} else {
   gl.uniform3fv(dirLightColorLocation, [0.0, 0.0, 0.0]);
}

// Четвёртый — позиционный
if (enablePoint) {
   gl.uniform3fv(pointLightDirectionLocation, [0.5, 0.5, 0.0]);
   gl.uniform3fv(pointLightColorLocation, [0.4, 0.8, 1.0]);
} else {
   gl.uniform3fv(pointLightColorLocation, [0.0, 0.0, 0.0]);
}

gl.uniform1i(useGouraudLocation, useGouraud ? 1 : 0);

const ambient = hexToVec3(document.getElementById("ambientColor").value);
const diffuse = hexToVec3(document.getElementById("diffuseColor").value);
const specular = hexToVec3(document.getElementById("specularColor").value);
const shininess = parseFloat(document.getElementById("shininess").value);

gl.uniform3fv(materialAmbientLocation, ambient);
gl.uniform3fv(materialDiffuseLocation, diffuse);
gl.uniform3fv(materialSpecularLocation, specular);
gl.uniform1f(materialShininessLocation, shininess);

updateProjection();

let eye = [
   parseFloat(document.getElementById('eyeX').value),
   parseFloat(document.getElementById('eyeY').value),
   parseFloat(document.getElementById('eyeZ').value)
];

let center = [
   parseFloat(document.getElementById('centerX').value),
   parseFloat(document.getElementById('centerY').value),
   parseFloat(document.getElementById('centerZ').value)
];

let up = [
   parseFloat(document.getElementById('upX').value),
   parseFloat(document.getElementById('upY').value),
   parseFloat(document.getElementById('upZ').value)
];

let viewMatrix = mat4.create();
mat4.lookAt(viewMatrix, eye, center, up);

mat4.identity(modelViewMatrix);
mat4.copy(modelViewMatrix, viewMatrix);

pushMatrix(modelViewMatrix);
```

```
   let xPosCube = parseFloat(document.getElementById('xPosCube').value);
   let yPosCube = parseFloat(document.getElementById('yPosCube').value);
   let zPosCube = parseFloat(document.getElementById('zPosCube').value);
   let xScaleCube = parseFloat(document.getElementById('xScaleCube').value);
   let yScaleCube = parseFloat(document.getElementById('yScaleCube').value);
   let xRotateCube = parseFloat(document.getElementById('xRotateCube').value) * Math.PI
/ 180;
   let yRotateCube = parseFloat(document.getElementById('yRotateCube').value) * Math.PI
/ 180;
   let zRotateCube = parseFloat(document.getElementById('zRotateCube').value) * Math.PI
/ 180;

   mat4.translate(modelViewMatrix, modelViewMatrix, [xPosCube, yPosCube, zPosCube]);
   mat4.rotateX(modelViewMatrix, modelViewMatrix, xRotateCube);
   mat4.rotateY(modelViewMatrix, modelViewMatrix, yRotateCube);
   mat4.rotateZ(modelViewMatrix, modelViewMatrix, zRotateCube);
   mat4.scale(modelViewMatrix, modelViewMatrix, [xScaleCube, yScaleCube, 1]);

   mat4.copy(normalMatrix, modelViewMatrix);
   mat4.invert(normalMatrix, normalMatrix);
   mat4.transpose(normalMatrix, normalMatrix);

   gl.uniformMatrix4fv(modelViewMatrixLocation, false, modelViewMatrix);
   gl.uniformMatrix4fv(normalMatrixLocation, false, normalMatrix);

   drawCube(cubeBuffer, cubeColorBuffer, cubeNormalBuffer, cubeVertices.length / 3);

   modelViewMatrix = popMatrix();

   let xPosPyramid = parseFloat(document.getElementById('xPosPyramid').value);
   let yPosPyramid = parseFloat(document.getElementById('yPosPyramid').value);
   let zPosPyramid = parseFloat(document.getElementById('zPosPyramid').value);
   let xScalePyramid = parseFloat(document.getElementById('xScalePyramid').value);
   let yScalePyramid = parseFloat(document.getElementById('yScalePyramid').value);
   let xRotatePyramid = parseFloat(document.getElementById('xRotatePyramid').value) *
Math.PI / 180;
   let yRotatePyramid = parseFloat(document.getElementById('yRotatePyramid').value) *
Math.PI / 180;
   let zRotatePyramid = parseFloat(document.getElementById('zRotatePyramid').value) *
Math.PI / 180;

   mat4.translate(modelViewMatrix, modelViewMatrix, [xPosPyramid, yPosPyramid,
zPosPyramid]);
   mat4.rotateX(modelViewMatrix, modelViewMatrix, xRotatePyramid);
   mat4.rotateY(modelViewMatrix, modelViewMatrix, yRotatePyramid);
   mat4.rotateZ(modelViewMatrix, modelViewMatrix, zRotatePyramid);
   mat4.scale(modelViewMatrix, modelViewMatrix, [xScalePyramid, yScalePyramid, 1]);

   mat4.copy(normalMatrix, modelViewMatrix);
   mat4.invert(normalMatrix, normalMatrix);
   mat4.transpose(normalMatrix, normalMatrix);

   gl.uniformMatrix4fv(modelViewMatrixLocation, false, modelViewMatrix);
   gl.uniformMatrix4fv(normalMatrixLocation, false, normalMatrix);

   drawPyramid(pyramidBuffer, pyramidColorBuffer, pyramidNormalBuffer,
pyramidVertices.length / 3);
```

```
}


document.getElementById('projectionType').addEventListener('change', updateView);
document.getElementById('enableLight2').addEventListener('change', updateView);
document.getElementById('enableDirLight').addEventListener('change', updateView);
document.getElementById('enablePointLight').addEventListener('change', updateView);
document.getElementById('gouraudToggle').addEventListener('change', updateView);
document.querySelectorAll('input').forEach(input => {
   input.addEventListener('input', updateView);
});
document.querySelectorAll("#material-control input").forEach(input => {
   input.addEventListener("input", () => {
      document.getElementById("shininessValue").textContent =
document.getElementById("shininess").value;
      updateView();
   });
});

updateView();
```