Программирование графических приложений

Тема 2
Работа с базовыми примитивами в WebGL

Цель изучения темы. Изучение принципов построения изображений в WebGL на основе базовых примитивов.

2.1. Конвейер WebGL

Для понимания способов применения базовых примитивов WebGL – точек, прямых линий, треугольников – рассмотрим построение изображений без использования библиотек.

Прежде чем приступить непосредственно к рисованию и созданию объектов, посмотрим, что собой представляет конвейер WebGL. Схематично его можно представить следующим образом:

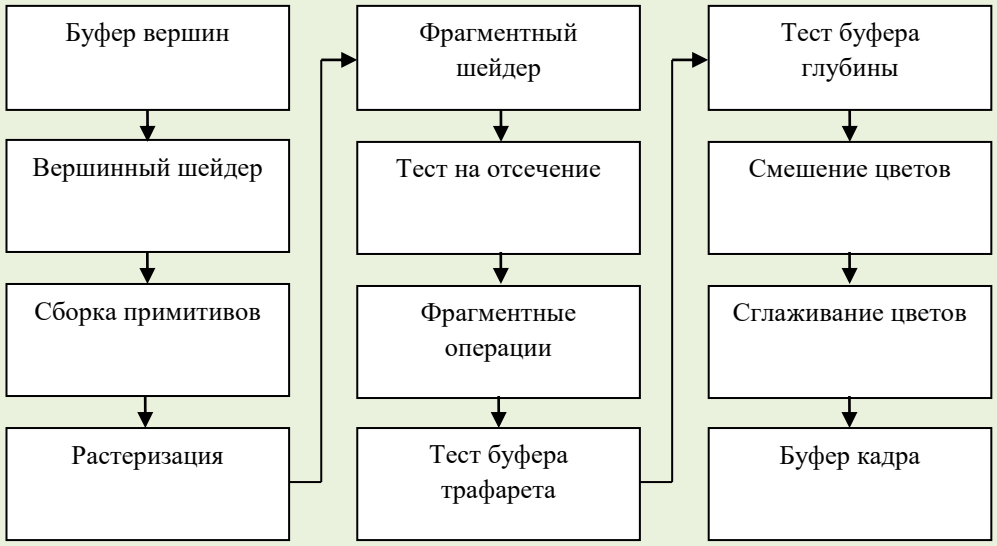

Рис. 2.1

Процесс содержит следующие тапы:

1. *Буфер вершин*. Вначале мы создаем набор вершин в буфере вершин (*Vertex Buffer*). По этим вершинам впоследствии будут составлены геометрические примитивы, а из примитивов - объекты. Здесь же проводится некоторая предварительная обработка примитивов, представленных вершинами из буфера.

2. *Вершинный шейдер*. Затем содержимое буфера вершин поступает на обработку в вершинный шейдер (*Vertex Shader*). Шейдер производит над вершинами некоторые трансформации, например, применяет матрицы преобразования и т.д. Шейдеры пишутся самим разработчиком, поэтому программист может применить различные преобразования по своему усмотрению.

3. *Сборка примитивов*. На этом этапе конвейер получает результат вершинного шейдера и пытается измененные вершины соотнести с отдельными примитивами - линиями, треугольниками, спрайтами. Также на этом этапе определяется, входит ли примитив в видимое пространство. Если нет, то он обрезается. Оставшиеся примитивы передаются на следующий этап конвейера.

4. *Растеризация*. Далее на этапе растеризации полученные примитивы преобразуются в фрагменты - наборы пикселей, которые затем будут отрисованы на экране.

5. *Фрагментный шейдер*. Здесь в дело вступает фрагментный шейдер (*Fragment shader*) (в технологиях Direct3D, XNA прямым аналогом является пиксельный шейдер). Фрагментный шейдер производит преобразования цветовой составляющей примитивов, окрашивая их пиксели, и в качестве вывода передает на следующий этап измененные фрагменты.

Следующие этапы представляют собой ряд преобразований над полученными с фрагментного шейдера фрагментами.

6. *Проверка на отсечение*. Здесь проверяется, находится ли фрагмент в пределах отсекающего прямоугольника. Если фрагмент находится в пределах этого прямоугольника, то он передается на следующий этап. Если же нет, то он отбрасывается и больше не принимает участия в обработке.

7. *Многопараметрические фрагментные операции*. На данном этапе у каждого фрагмента изменяются цветовые составляющие, производится сглаживание (anti-alising), чтобы объект выглядел более плавно на экране.

8. *Проверка буфера трафарета***.** Здесь фрагмент передается в буфер трафаретов (stencil buffer). В этом буфере дополнительно отбрасываются те фрагменты, которые не должны отображаться на экране. Как правило, данный буфер используется для создания различного рода эффектов, например, эффект теней.

9. *Проверка буфера глубины*. В буфере глубины (depth buffer, также называется z-buffer) сравнивается z-компонента фрагмента, и если она больше значения в буфере глубины, то, следовательно, данный фрагмент расположен к наблюдателю трехмерной сцены ближе, чем предыдущий фрагмент, поэтому текущий фрагмент проходит тест. Если же z-компонента больше значения в буфере глубины, то, следовательно, данный фрагмент находится дальше, поэтому он не должен быть виден и отбрасывается.

10. *Смешение цветов*. На этом этапе происходите небольшое смешение цветов, например, для создания прозрачных объектов.

11. *Сглаживание цветов*. Здесь происходит смешение цветов для создания тонов и полутонов.

12. *Буфер кадра*. Здесь полученные после предобработки фрагменты превращаются в пиксели на экране.

Этапы конвейера составляют некоторый типовой алгоритм действий. Разработка самой программы разбивается также на некоторые этапы:

1. Создание и настройка шейдеров
2. Создание и настройка буфера вершин, которые впоследствии образуют геометрическую фигуру
3. Отрисовка фигуры

2.2. Настройка буфера вершин и буфера индексов

Чтобы что-то нарисовать, надо определить точки или вершины, по которым будет рисоваться конкретный примитив, а из примитивов уже будет складываться геометрическая фигура. Поэтому первым делом необходимо настроить буфер вершин.

В программе **ex01_01.html** создание буфера вершин происходило в следующем участке кода:

```
function initBuffers() {
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer (gl.ARRAY_BUFFER, vertexBuffer);
  var triangleVertices =   [ 0.0,  0.5,  0.0,
                             0.0,  0.5,  0.0,
                             0.5,  0.0,  0.0  ];
  gl.bufferData (gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
            gl.STATIC_DRAW);
  vertexBuffer.itemSize = 3;
  vertexBuffer.numberOfItems = 3;
}
```

Итак, у нас есть глобальная переменная vertexBuffer, которая будет хранить *буфер вершин*. И для начала нам надо его создать: vertexBuffer = gl.createBuffer().

Затем выполняем привязку буфера к контексту WebGL:

```
gl.bindBuffer (gl.ARRAY_BUFFER, vertexBuffer);
```

Привязка означает, что все операции с буфером будут происходить именно над буфером vertexBuffer.

В качестве первого параметра метод gl.bindBuffer принимает тип создаваемого буфера и может принимать следующие значения:

gl.ARRAY_BUFFER: данные вершин

gl.ELEMENT_ARRAY_BUFFER: данные индексов

Поскольку в нашем случае мы создаем буфер вершин, то используется значение gl.ARRAY_BUFFER. Впоследствии мы можем отвязать буфер, например, с помощью следующего выражения:

```
gl.bindBuffer (gl.ARRAY_BUFFER, null);
```

Последним шагом является загрузка определенных разработчиком координат в буфер вершин и его типизация:

```
gl.bufferData (gl.ARRAY_BUFFER, new Float32Array(triangleVertices),
        gl.STATIC_DRAW);
```

В данном случае массив координат типизируется конструктором Float32Array, который создает представление массива в виде набора чисел с плавающей точкой.

При этом мы не можем передать просто переменную triangleVertices, представляющую массив координат. Нам обязательно надо ее типизировать. Но объект Float32Array не единственный, который мы можем использовать для типизации. Также мы можем использовать следующие объекты: Int8Array, Uint8Array, Int16Array, Uint16Array, Int32Array, Uint32Array, Float62Array. Только в отличие от Float32Array перечисленные типы будут создавать набор целых чисел соответственно занимающих 8, 16 и 32 бита, как указано в их названии. И только последний тип - Float62Array будет создавать набор 8-байтных чисел с плавающей точкой.

Переменная triangleVertices определяет набор координат, по которым будут создаваться вершины в буфере вершин, а затем по ним будут строиться геометрические примитивы.

Последние сроки

```
vertexBuffer.itemSize=3;
vertexBuffer.numberOfItems=3;
```

используются для последующего применения при отрисовке.

Теперь рассмотрим использование буфера индексов. Изменим программу **ex01_01.html** таким образом, чтобы использовать в ней сразу два буфера (**ex02_01.html**).

```
function initBuffers() {
  vertices = [ -0.5, -0.5, 0.0,
                -0.5,  0.5, 0.0,
                 0.5,  0.5, 0.0,
                 0.5, -0.5, 0.0 ];
 indices = [0, 1, 2, 0, 3, 2];
 // установка буфера вершин
vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
```

```
 // указываем размерность
vertexBuffer.itemSize = 3;
 // создание буфера индексов
indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
            gl.STATIC_DRAW);
 // указываем число линий. это число равно числу индексов
 indexBuffer.numberOfItems = indices.length;
}

function draw() {
   // установка фона
   gl.clearColor(0.0, 0.0, 0.0, 1.0);
   // установка области отрисовки
   gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
   gl.clear(gl.COLOR_BUFFER_BIT);
   gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
              vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
   // отрисовка примитивов - линий
   gl.drawElements(gl.LINE_LOOP, indexBuffer.numberOfItems,
            gl.UNSIGNED_SHORT,0);
}
```

Результатом будет фигура из 5 линий - квадрат с диагональю.

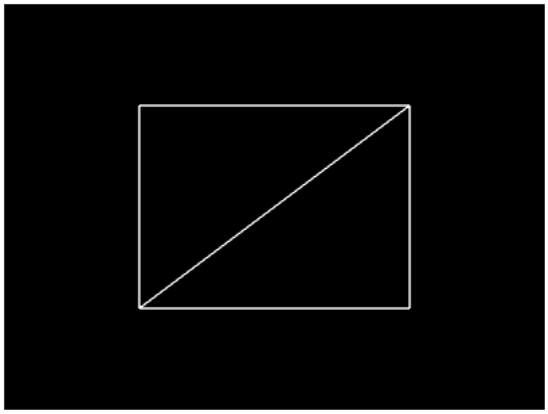

Рис. 2.2

В данном случае нас интересует метод настройки буферов вершин и индексов - initBuffers().

Первым делом мы определяем массивы вершин и индексов. Если в массиве вершин, как и ранее, определяется по три координаты для каждой вершины, то в массиве индексов мы указываем индексы, по которым будут строиться линии. Первым идет индекс 0 - то есть с первой точки в массиве вершин будет начинаться построение фигуры, и от этой вершины мы начинаем линию.

Далее идет индекс 1, который указывает индекс новой точки, до которой будет построена линия и т.д.:

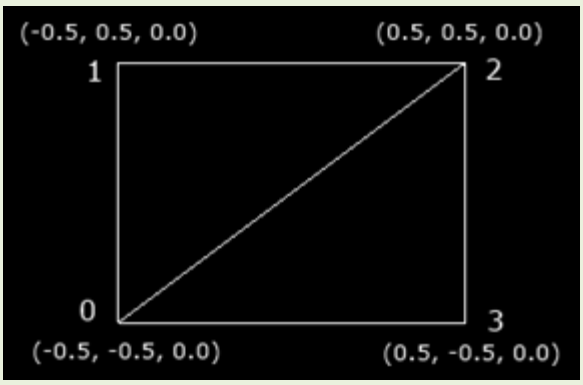

Рис. 2.3

Далее в программе идет создание буферов вершин и индексов: буфер вершин создается уже знакомым способом, а при создании буфера индексов указывается другой тип массива:

```
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
        gl.STATIC_DRAW);
```

В дальнейшем в методе draw производится непосредственно отрисовка с помощью метода

```
gl.drawElements(gl.LINE_LOOP, indexBuffer.numberOfItems,
        gl.UNSIGNED_SHORT,0);
```

Метод gl.drawElements рисует примитивы (в данном случае линии) по индексам, а не по вершинам.

2.3. Установка атрибута для буфера вершин

В предыдущих примерах в методе отрисовки draw мы использовали следующее выражение:

```
gl.vertexAttribPointer (shaderProgram.vertexPositionAttribute,
        vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
```

С помощью метода draw мы устанавливаем атрибут или указатель на чтение из буфера вершин; этот атрибут создается в функции initShaders():

```
shaderProgram.vertexPositionAttribute = gl.getAttribLocation (shaderProgram,
        "aVertexPosition");.
```

Поскольку здесь для атрибута устанавливается имя "aVertexPosition", то в вершинном шейдере мы используем переменную с данным именем:

```
attribute vec3 aVertexPosition;
void main (void) {    gl_Position = vec4(aVertexPosition, 1.0);   }
```

Эта переменная attribute vec3 aVertexPosition представляет собой атрибут вершинного шейдера и в программе WebGL она передает координаты вершины в шейдер. Поскольку в методе gl.getAttribLocation() атрибут shaderProgram.vertexPositionAttribute связывается с именем aVertexPosition, то именно с таким именем и определяется переменная, которая будет передавать в шейдер координаты вершин.

Это не единственный атрибут. В нашем примере мы устанавливаем атрибут только для вершин, но можем установить различные атрибуты (например, один для вершин, другой для цветов), используя метод

gl.vertexAttribPointer (index, size, type, norm, stride, offset).

Этот метод принимает следующие параметры:

- index: индекс атрибута, который сопоставляется с буфером вершин;
- size: число значений для каждой вершины, которые хранятся в буфере (в частности у нас три координаты на вершину, поэтому и число, передаваемое в качестве данного параметра, будет равно 3);
- type: тип значений, который хранятся в буфере; параметр может быть следующих типов: FIXED, BYTE, UNSIGNED_BYTE, FLOAT, SHORT и UNSIGNED_SHORT.
- norm: представляет булево значение, управляет числовыми преобразованиями;
- stride: шаг; при установке в качестве значения 0 (нулевой шаг) элементы будут обрабатываться последовательно;
- offset: смещение - позиция в буфере, с которой начинается обработка; обычно устанавливается нулевое смещение, то есть все элементы обрабатываются с самого начала.

Кроме установки указателя необходимо еще надо включить атрибут с помощью метода gl.enableVertexAttribArray(), в который передается ранее установленный атрибут:

```
gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
```

После этого мы сможем передать каждую вершину в вершинный шейдер через переменную attribute vec3 aVertexPosition.

Теперь посмотрим, что происходит в программе. Возьмем пример **ex02_01.html**. Там набор вершин задан с помощью следующего массива:

```
vertices =  [ -0.5, -0.5, 0.0, -0.5, 0.5, 0.0,  0.5, 0.5, 0.0,  0.5, -0.5, 0.0];
```

Сначала мы осуществляем привязку данного набора координат в качестве буфера вершин:

```
gl.bindBuffer (gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData (gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
```

После этого нам надо установить указатель на чтение из буфера вершин:

```
gl.vertexAttribPointer (shaderProgram.vertexPositionAttribute, 3, gl.FLOAT, false, 0, 0);
```

Так как у нас каждая вершина представлена тремя координатами, то в качестве второго параметра мы устанавливаем число 3. В итоге при передаче в вершинный шейдер атрибут shaderProgram.vertexPositionAttribute будет получать по одной вершине из трех координат.

Например, в самом начале атрибут будет передавать в вершинный шейдер первую вершину: shaderProgram.vertexPositionAttribute=[-0.5, -0.5, 0.0], затем вторую и т.д. Затем в вершинном шейдере мы можем получить переданную через атрибут вершину и провести над ней преобразования через переменную aVertexPosition:

```
attribute vec3 aVertexPosition;
void main(void) { gl_Position = vec4(aVertexPosition, 1.0);  }
```

В данном случае никакого преобразования не происходит, и мы устанавливаем конечную вершину как есть.

2.4. Отрисовка в WebGL

Отрисовка объектов выполняется с помощью примитивов. Для отрисовки фигур в WebGL используется методы gl.drawArrays() и gl.drawElements(). Рассмотрим их подробнее.

Метод **gl.drawArrays()** отрисовывает объекты последовательно по вершинам в буфере вершин. Данный метод принимает три параметра: gl.drawArrays(mode, index, count)

- mode: данный параметр указывает на отрисовываемый примитив и может принимать следующие значения: gl.POINTS, gl.LINES, gl.LINE_LOOP, gl.LINE_STRIP, gl.TRIANGLES, gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN. Ниже приведено более подробное описание примитивов.
- index: второй параметр указывает, какой номер вершины в буфере вершин будет первой для примитива.
- count: третий параметр указывает, сколько вершин будет использоваться для отрисовки.

В WebGL определены следующие виды примитивов:

- gl.POINTS: набор обычных точек, не соединенных между собой;
- gl.LINES: набор линий; при этом все линии рисуются отдельно: если у двух линий определена общая точка, то визуально они соединяются, но фактически это две разные линии;
- gl.LINE_STRIP: набор точек последовательно соединяются линиями, а незамкнутый контур, образуемый линиями, представляет единое целое;
- gl.LINE_LOOP: то же самое, что и gl.LINE_STRIP, только последняя точка в наборе дополнительно еще соединяется с первой, таким образом образуется замкнутый контур;
- gl.TRIANGLES: набор треугольников;
- gl.TRIANGLE_STRIP: набор треугольников, при этом вершины последовательно соединяются в треугольники;
- gl.TRIANGLE_FAN: набор треугольников, при этом для всех треугольников есть одна общая вершина в центре.

Метод **gl.drawElements()** работает с буфером индексов, в отличие от рассмотренного выше метода, имеющего дело с буфером вершин. Он имеет следующее описание: gl.drawElements(mode, count, type, offset):

- mode: режим, указывающий на тип примитива (в качестве примитивов используются те же, что и для метода gl.drawArrays);
- count: число элементов для отрисовки;
- type: тип значений в буфере индексов: может принимать значение UNSIGNED_BYTE или UNSIGNED_SHORT;
- offset: смещение, определяющее, с какого индекса будет проводиться отрисовка;

Например, в одной из программ ранее использовалось: gl.drawElements(gl.LINE_LOOP, indexBuffer.numberOfItems, gl.UNSIGNED_SHORT,0).

Далее будет рассмотрено использование примитивов метода gl.drawArrays().

2.5. Точки и линии

В результате вызова функции drawArrays с параметром gl.POINTS просто отрисовываются те **точки**, которые заданы в ARRAY_BUFFER. Например:

```
(0, 0, 0), (-1, 0, 0), (-0.7, 0.7, 0), (0, 1, 0), (0.7, 0.7, 0), (1, 0, 0)
```

Тогда мы получим следующую картинку (красные цифры дорисованы отдельно):

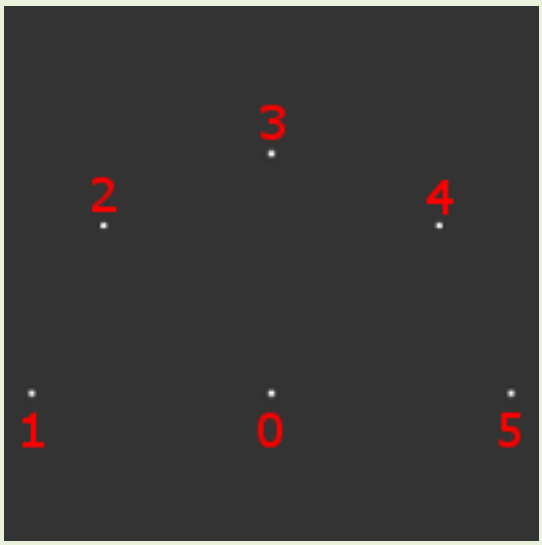

Рис. 2.4

Здесь стоит только обратить внимание на то, что для отрисовки точек необходимо задать их размер в теле вершинного шейдера (иначе точек не будет видно):

```
gl_PointSize = 5.0;
```

Линии содержат сразу три примитива. Разница между ними заключается в том, каким образом они обрабатывают массив вершин. Если количество вершин нечетное (то есть для последней линии нет второй вершины), то последняя линия не формируется.

gl.LINES

В этом режиме для линии берутся вершины 0 и 1, затем 2 и 3 и так далее. В общем случае линии образуются вершинами 2i и 2i+1 (здесь и далее i начинается с нуля). При том же наборе координат

```
(0, 0, 0), (-1, 0, 0), (-0.7, 0.7, 0), (0, 1, 0), (0.7, 0.7, 0), (1, 0, 0)
```

мы получим следующую картинку:

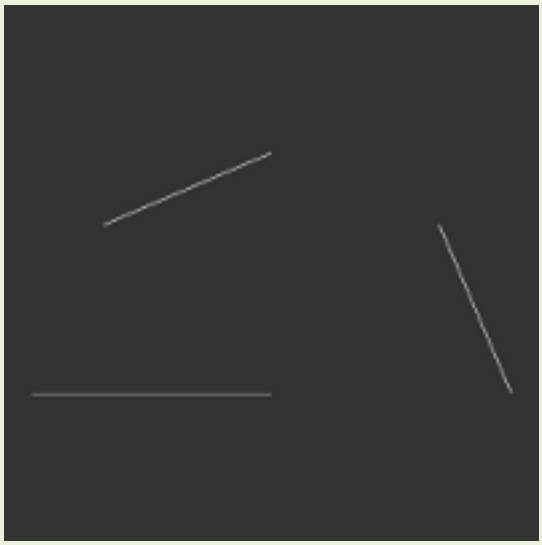
Рис. 2.5

gl.LINE_STRIP

Этот случай можно выразить формулой (i, i + 1). Таким образом, в отличие от gl.LINES, мы получаем непрерывную линию, составленную из отрезков на основании вершин 0 и 1, затем 1 и 2 и так далее. Вот, как это будет выглядеть:

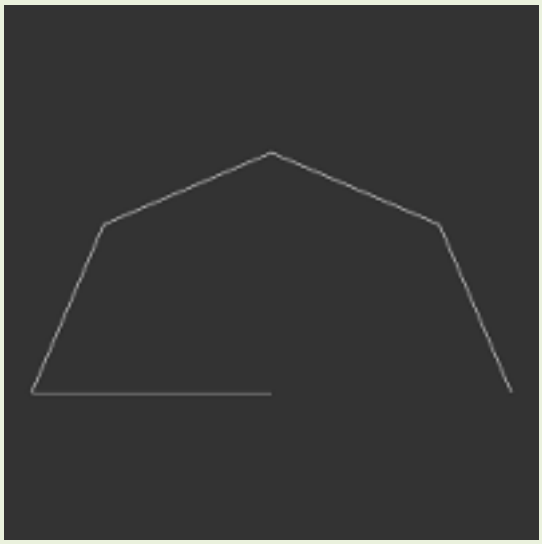

Рис. 2.6

gl.LINE_LOOP

Случай аналогичен предыдущему за тем исключением, что последняя точка замыкается с первой:

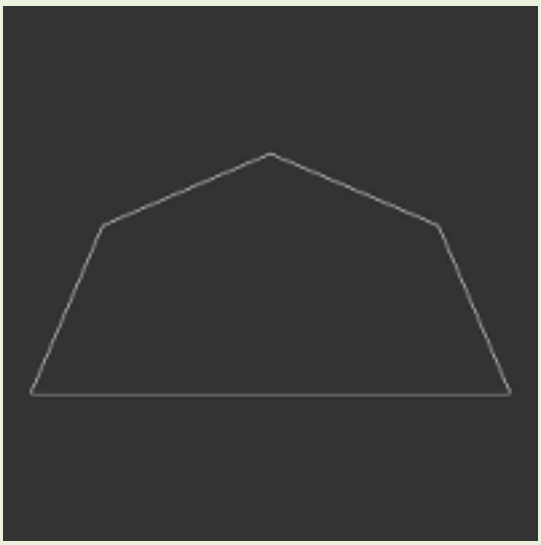

Рис. 2.7

Примеры отрисовки отрезков прямых линий

Рисование отрезков прямых линий во многом аналогично рисованию треугольников, только теперь каждый примитив создается на основе двух последовательных вершин. Два следующих друг за другом отрезка не обязательно должны быть соединены между собой. Создадим набор линий (**ex02_02.html**):

```
<!doctype html>
<html>
<head>
<title>Прямые линии</title>
```

```
<meta content="charset=utf-8">
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<script id="shader-fs" type="x-shader/x-fragment">
 void main(void) {    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
 attribute vec3 aVertexPosition;
 void main(void) {    gl_Position = vec4(aVertexPosition, 1.0);  }
</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer; // буфер вершин
var indexBuffer; //буфер индексов
// установка шейдеров
function initShaders() {
   var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
   var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
   shaderProgram = gl.createProgram();
   gl.attachShader(shaderProgram, vertexShader);
   gl.attachShader(shaderProgram, fragmentShader);
   gl.linkProgram(shaderProgram);
   if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Не удалось установить шейдеры");
   }
   gl.useProgram(shaderProgram);
   shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
         "aVertexPosition");
   gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}
// Функция создания шейдера
function getShader(type,id) {
   var source = document.getElementById(id).innerHTML;
   var shader = gl.createShader(type);
   gl.shaderSource(shader, source);
   gl.compileShader(shader);
   if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
      return null;
   }
   return shader;
}
 // установка буферов вершин и индексов
function initBuffers() {
   vertices =[ -0.5, -0.5, 0.0,
         -0.5, 0.5, 0.0,
         0.0, 0.0, 0.0,
         0.5, 0.5, 0.0,
```

```
            0.5, -0.5, 0.0];
    indices = [0, 1, 1, 2, 2, 3, 3, 4];
  // установка буфера вершин
   vertexBuffer = gl.createBuffer();
   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
   gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
   vertexBuffer.itemSize = 3;
   // создание буфера индексов
   indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
              gl.STATIC_DRAW);
    // указываем число индексов это число равно числу индексов
    indexBuffer.numberOfItems = indices.length;
}
function draw() {
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                      vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
    // отрисовка треугольников
    gl.drawElements(gl.LINES, indexBuffer.numberOfItems, gl.UNSIGNED_SHORT,0);
}
window.onload=function(){
    var canvas = document.getElementById("canvas3D");
    try { gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl"); }
    catch(e) {}
      if (!gl) { alert("Ваш браузер не поддерживает WebGL");   }
    if(gl){
        gl.viewportWidth = canvas.width;
        gl.viewportHeight = canvas.height;
        initShaders();
        initBuffers();
        draw();
    }
}
</script>
</body>
</html>
```

Результат:

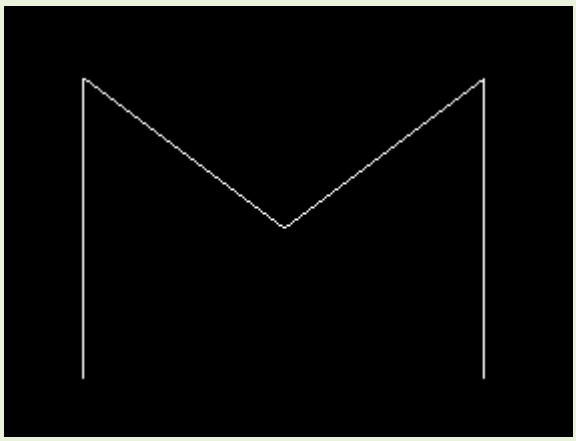

Рис. 2.8

Набор индексов indices = [0, 1, 1, 2, 2, 3, 3, 4]; указывает последовательность вершин, из которых составляется четыре линии: сначала между первой и второй вершиной, потом между второй и третьей и так далее. То есть каждая линия формируется отдельно, несмотря на то, что на рисунке они визуально соединяются.

Другие способы создания линий дают примитивы gl.LINE_LOOP и gl.LINE_STRIP. В отличие от gl.LINES здесь можно обойтись меньшим количеством вершин и индексов, поскольку в gl.LINE_LOOP линии последовательно соединяются, образуя в результате замкнутый контур. В gl.LINE_STRIP вершины также последовательно соединяются линиями, только не происходит соединения последней вершины с первой.

Возьмем предыдущий пример и изменим в нем всего две строки. Во-первых, изменим в функции initBuffers набор индексов на следующий: indices = [0, 1, 2, 3, 4];. Во-вторых, изменим в функции draw способ отрисовки примитивов на следующий:

```
gl.drawElements(gl.LINE_STRIP, indexBuffer.numberOfItems,
                gl.UNSIGNED_SHORT,0);
```

В результате у нас должен получиться тот же эффект, что и в предыдущем примере. Если бы вместо gl.LINE_STRIP мы бы использовали gl.LINE_LOOP, то получили бы замкнутый контур:

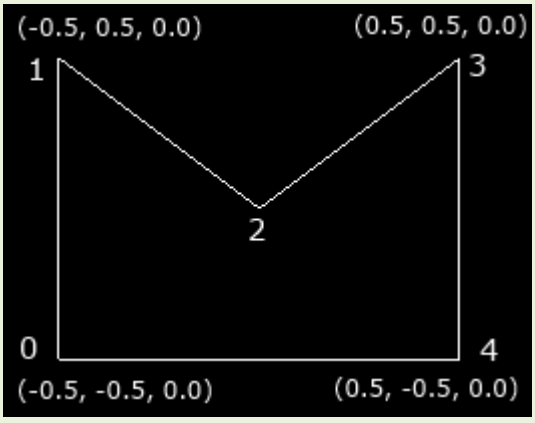

Рис. 2.9

2.6. Треугольники

Отрисовка треугольниками встречается наиболее часто при создании приложений, так как мы видим сам объект, а не просто его «скелет». Если количество точек не кратно трем, то последний треугольник построен не будет, а точки проигнорируются.

Несмотря на то, что треугольник – это всего лишь обычная фигура на плоскости, в webgl он имеет важное значение: любой трехмерный объект мы можем с заданной точностью представить как набор треугольников.

gl.TRIANGLES

Треугольники образуются точками (3i, 3i+1, 3i+2). Первый треугольник образуется вершинами (0, 1, 2), второй (3, 4, 5) и так далее. Выглядит это следующим образом:

Рис. 2.10

Для каждого треугольника в буфере вершин определяется по три вершины. При этом если нам надо отрисовать несколько треугольников, то вершины одного треугольника не используются для другого.

В примере **ex01_01.html** мы уже рассматривали пример отрисовки треугольника, теперь его несколько модифицируем (**ex02_03.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>Треугольники</title>
<meta content="charset=utf-8">
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<script id="shader-fs" type="x-shader/x-fragment">
 void main(void) {   gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
 attribute vec3 aVertexPosition;
 void main(void) {  gl_Position = vec4(aVertexPosition, 1.0);  }
</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer; // буфер вершин
var indexBuffer; //буфер индексов
// установка шейдеров
```

```
function initShaders() {
    var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
    var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);
    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Не удалось установить шейдеры");
    }
    gl.useProgram(shaderProgram);
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
            "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}
// Функция создания шейдера
function getShader(type,id) {
    var source = document.getElementById(id).innerHTML;
    var shader = gl.createShader(type);
    gl.shaderSource(shader, source);
    gl.compileShader(shader);
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
        return null;
    }
    return shader;
}
 // установка буферов вершин и индексов
function initBuffers() {
    vertices =[ -0.5, -0.5, 0.0,
                -0.5, 0.5, 0.0,
                 0.0, 0.0, 0.0,
                 0.5, 0.5, 0.0,
                 0.5, -0.5, 0.0];
    indices = [0, 1, 2, 2, 4, 3];
 // установка буфера вершин
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  // размерность
  vertexBuffer.itemSize = 3;
  // создание буфера индексов
  indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
            gl.STATIC_DRAW);
    // указываем число индексов это число равно числу индексов
    indexBuffer.numberOfItems = indices.length;
}
 // отрисовка
function draw() {
```

```
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                    vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
    // отрисовка треугольников
    gl.drawElements(gl.TRIANGLES, indexBuffer.numberOfItems,
            gl.UNSIGNED_SHORT,0);
}

window.onload=function(){
    var canvas = document.getElementById("canvas3D");
    try { gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");  }
    catch(e) {}
      if (!gl) {    alert("Ваш браузер не поддерживает WebGL");     }
    if(gl){
        gl.viewportWidth = canvas.width;
        gl.viewportHeight = canvas.height;
        initShaders();
        initBuffers();
        draw();
    }
}
</script>
</body>
</html>
```

Результат будет следующий:

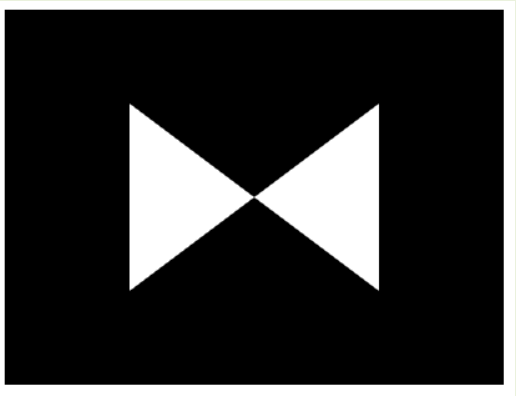

Рис. 2.11

В данном случае у нас есть пять вершин и шесть индексов:

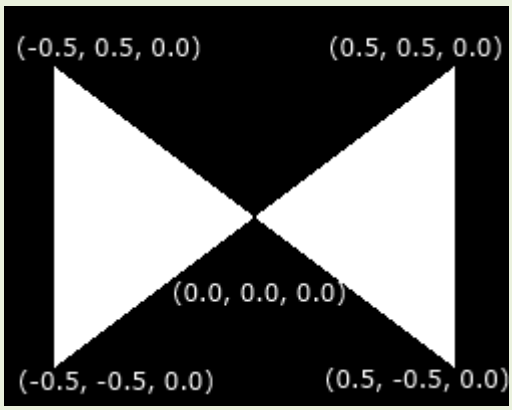

Рис. 2.12

Поскольку построение треугольника идет по индексам через метод gl.drawElements, можно определить пять вершин. Главное, чтобы на каждый треугольник приходилось по три индекса, которые указывают на три разных вершины.

gl.TRIANGLE_STRIP

Треугольники образуются точками (i, i+1, i+2). Первый треугольник образуется вершинами (0, 1, 2), второй (1, 2, 3) и так далее. В нашем случае треугольники будут пересекаться между собой, поэтому на рисунке выделены отдельные треугольники разными цветами для понимания принципа работы:


Рис. 2.13

Данный примитив используется для создания набора соединенных (имеющих одну общую сторону) треугольников, каждый из которых определяется тремя последовательными точками в буфере вершины. Особенно удобно его использовать в тех случаях, когда мы строим триангулированные (разбитые на треугольники) объекты: это позволит уменьшить количество используемых вершин по сравнению, например, с gl.TRIANGLES, что приведет к уменьшению объема данных при обработке.

Имея некоторое количество точек в буфере вершин (count), можно подсчитать общее количество треугольников, которое можно на их основе построить: count-2. То есть чтобы построить три треугольника, нам потребуется 3+2=5 вершин.

Создадим программу, использующую данный примитив (**ex02_04.html**):

```
<!DOCTYPE html>
<html>
```

```
<head>
<title>Соединенные треугольники</title>
<meta content="charset=utf-8">
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<script id="shader-fs" type="x-shader/x-fragment">
 void main(void) { gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
 attribute vec3 aVertexPosition;
 void main(void) { gl_Position = vec4(aVertexPosition, 1.0);  }
</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer; // буфер вершин
// установка шейдеров
function initShaders() {
   var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
   var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
   shaderProgram = gl.createProgram();
   gl.attachShader(shaderProgram, vertexShader);
   gl.attachShader(shaderProgram, fragmentShader);
   gl.linkProgram(shaderProgram);
   if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Не удалось установить шейдеры");
   }
   gl.useProgram(shaderProgram);
   shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
         "aVertexPosition");
   gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}
// Функция создания шейдера
function getShader(type,id) {
   var source = document.getElementById(id).innerHTML;
   var shader = gl.createShader(type);
   gl.shaderSource(shader, source);
   gl.compileShader(shader);
   if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
      return null;
   }
   return shader;
}
 // установка вершинного буфера
function initBuffers() {
   vertices =[ -0.5,  -0.5,  0.0, //m0
     -0.5,  0.5,  0.0, //m1
     0.0,  0.0,  0.0, //m2
```

```
      0.5,  0.5,  0.0, //m3
      0.5,  -0.5,  0.0, //m3
      ];
 // установка буфера вершин
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  // размерность
  vertexBuffer.itemSize = 3;
  // указываем кол-во вершин - 5
  vertexBuffer.numberOfItems=5;
}
// отрисовка
function draw() {
   gl.clearColor(0.0, 0.0, 0.0, 1.0);
   gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
   gl.clear(gl.COLOR_BUFFER_BIT);
   gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                  vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
   // рисуем примитивы gl.TRIANGLE_STRIP
   gl.drawArrays(gl.TRIANGLE_STRIP, 0, vertexBuffer.numberOfItems);
}

window.onload=function(){
   var canvas = document.getElementById("canvas3D");
   try { gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl"); }
   catch(e) {}
    if (!gl) { alert("Ваш браузер не поддерживает WebGL"); }
   if (gl){
      gl.viewportWidth = canvas.width;
      gl.viewportHeight = canvas.height;
      initShaders();
      initBuffers();
      draw();
   }
}
</script>
</body>
</html>
```

В результате определяется пять вершин, которые последовательно соединяются в 3 треугольника:

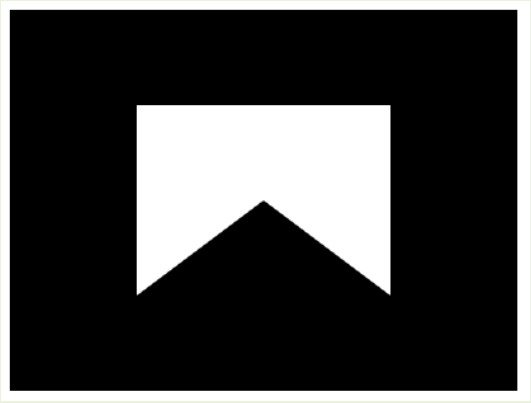

Рис. 2.14

Их параметры:

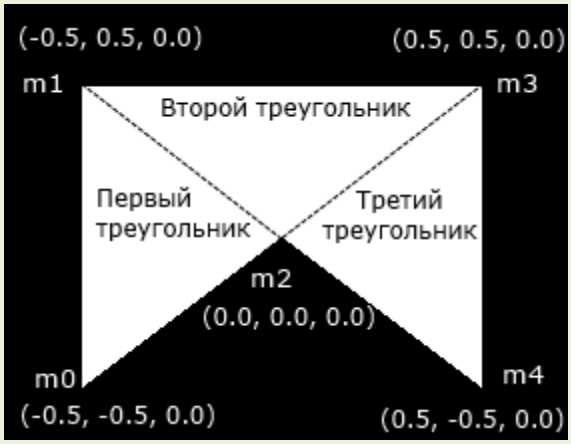

Рис. 2.15

gl.TRIANGLE_FAN

В этом случае в треугольниках всегда участвует первая вершина и формула принимает вид (0, i+1, i+2). Первый треугольник образуется вершинами (0, 1, 2), второй (0, 2, 3) и так далее. Получается «веер», который представлен на следующей картинке:

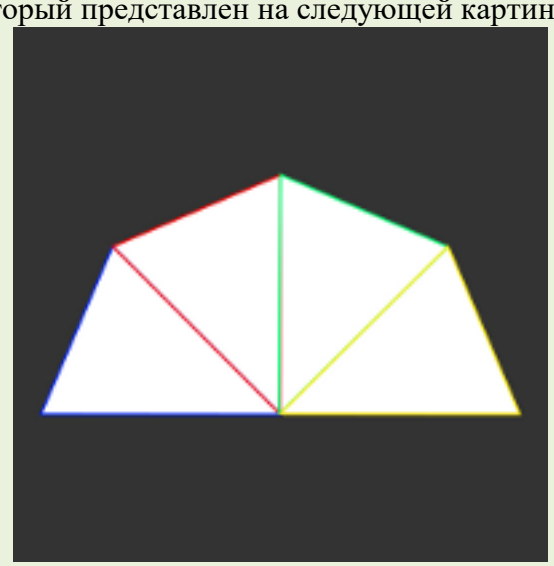

Рис. 2.16

Примитив gl.TRIANGLE_FAN создает набор треугольников по типу веера: каждые два последовательных треугольника имеют общую сторону и все треугольники имеют одну общую вершину (она задается первой).

Так же, как и в случае с gl.TRIANGLE_STRIP, общее количество треугольников будет равно count-2, где count - количество вершин в буфере вершин. То есть чтобы построить три треугольника нам опять потребуется 3+2=5 вершин.

Создадим программу, использующую данный примитив (**ex02_05.html**):

```
<!DOCTYPE html>
<html>
<head>
<title>Веер треугольников</title>
<meta content="charset=utf-8">
</head>
<body>
<canvas id="canvas3D" width="400" height="300">Ваш браузер не поддерживает
        элемент canvas</canvas>
<script id="shader-fs" type="x-shader/x-fragment">
 void main(void) {   gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);  }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
 attribute vec3 aVertexPosition;
 void main(void) {   gl_Position = vec4(aVertexPosition, 1.0);  }
</script>
<script type="text/javascript">
var gl;
var shaderProgram;
var vertexBuffer;
// установка шейдеров
function initShaders() {
   var fragmentShader = getShader(gl.FRAGMENT_SHADER, 'shader-fs');
   var vertexShader = getShader(gl.VERTEX_SHADER, 'shader-vs');
   shaderProgram = gl.createProgram();
   gl.attachShader(shaderProgram, vertexShader);
   gl.attachShader(shaderProgram, fragmentShader);
   gl.linkProgram(shaderProgram);

   if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
      alert("Не удалось установить шейдеры");
   }
   gl.useProgram(shaderProgram);
   shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
         "aVertexPosition");
   gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
}
// Функция создания шейдера
function getShader(type,id) {
   var source = document.getElementById(id).innerHTML;
   var shader = gl.createShader(type);
   gl.shaderSource(shader, source);
   gl.compileShader(shader);
     if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
      alert("Ошибка компиляции шейдера: " + gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
      return null;
```

```
    }
    return shader;
}

function initBuffers() {
    vertices =[ 0.0,  0.5,  0.0, //m0
       -0.5,  -0.4,  0.0, //m1
       -0.2,  -0.5,  0.0, //m2
       0.2,  -0.5,  0.0, //m3
       0.5,  -0.4,  0.0, //m3
       ];
 // установка буфера вершин
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
  // размерность
  vertexBuffer.itemSize = 3;
  // указываем кол-во вершин - 5
  vertexBuffer.numberOfItems=5;
}

function draw() {
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                    vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
     gl.drawArrays(gl.TRIANGLE_FAN, 0, vertexBuffer.numberOfItems);
}

window.onload=function(){
    var canvas = document.getElementById("canvas3D");
    try {  gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");  }
    catch(e) {}
      if (!gl) {  alert("Ваш браузер не поддерживает WebGL");    }
    if(gl){
        gl.viewportWidth = canvas.width;
        gl.viewportHeight = canvas.height;
        initShaders();
        initBuffers();
        draw();
    }
}
</script>
</body>
</html>
```

Результат:

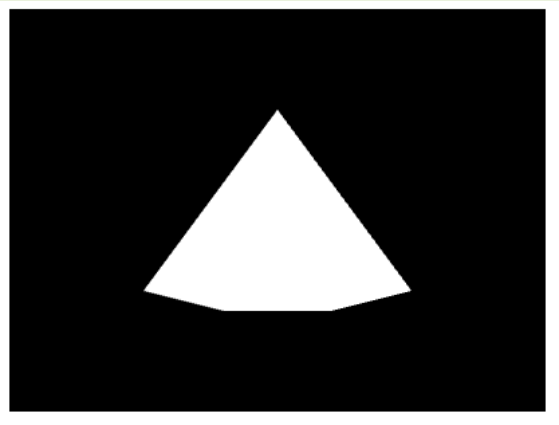

Рис. 2.17

Мы определяем пять вершин, которые последовательно соединяются в 3 треугольника:

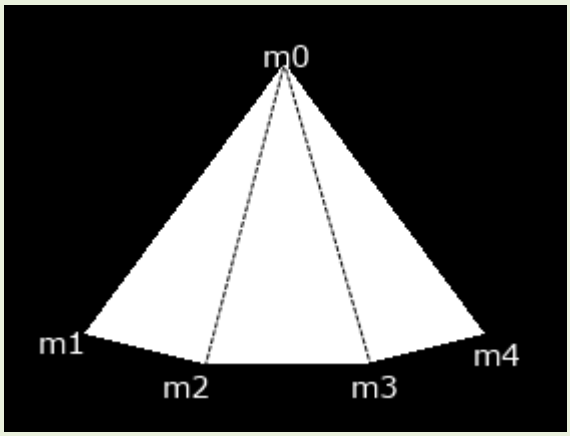

Рис. 2.18

Таким образом, первый треугольник образуется вершинами m0, m1, m2; второй - m0, m2, m3; а третий - m0, m3, m4. В итоге получается подобие веера, где имеется общая точка m0.

2.7. Установка Viewport

Отрисовка примитивов с помощью методов gl.drawElements или gl.drawArrays еще не позволяет нам увидеть их на экране. Для рисования нам нужна некая область, где будет происходить отрисовка (наличие элемента canvas еще недостаточно, чтобы произвести отрисовку).

Для установки области рисования нам надо настроить viewport контекста WebGL. Во всех ранее использованных примерах настройка viewport была однотипна: сначала мы получали размеры области элемента canvas, которой представляет собой полотно рисования:

```
if (gl){
   gl.viewportWidth = canvas.width;
   gl.viewportHeight = canvas.height;
   initShaders();
   //.....................
```

Свойства gl.viewportWidth и viewportHeight позволяли настроить ширину и высоту области рисования. Затем в функции отрисовки draw перед рисованием примитивов происходила установка области рисования:

```
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);.
```

Таким образом определяется какую часть элемента canvas надо использовать для рисования: gl.viewport указывает на левый нижний угол этой области, а также ее ширину и высоту. Обычно область рисования привязывается к началу координат – точке (0,0), но мы можем задать и другую область рисования, например: gl.viewport(150, 200, 50, 50) :

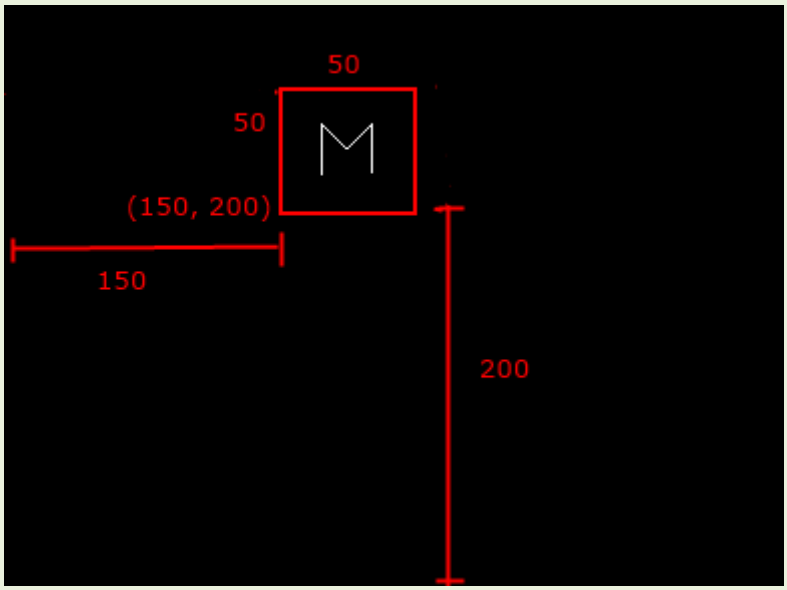

Рис. 2.19

В этом случае вся отрисовка будет идти на указанной нами области.

2.8. Пример программы построения всех примитивов

Код, реализующий построение всех рассмотренных выше базовых примитивов с использованием библиотеки gl-matrix-min.js **(ex02_06.html)**:

```
<!doctype html>
<html lang="ru">
<head>
<title>Примитивы WebGL</title>
<meta content="charset=utf-8">
<script type="text/javascript" src="gl-matrix-min.js"></script>
<script id="shader-fs" type="x-shader/x-fragment">
   precision mediump float;
   void main(void) {     gl_FragColor = vec4(1, 1, 1, 1.0);    }
</script>
<script id="shader-vs" type="x-shader/x-vertex">
   attribute vec3 aVertexPosition;
   uniform mat4 uMVMatrix;
   uniform mat4 uPMatrix;
   void main(void) {
      gl_PointSize = 5.0;
      gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
   }
</script>
<script type="text/javascript">
   var gl;
   function initGL(canvas) {
      try {
```

```
        gl = canvas.getContext("experimental-webgl");
        gl.viewportWidth = canvas.width;
        gl.viewportHeight = canvas.height;
    } catch (e) {  }
    if (!gl) {  alert ("Не поддерживается WebGL");   }
}

function getShader(gl, id) {
    var shaderScript = document.getElementById(id);
    if (!shaderScript) {    return null;   }
    var str = "";
    var k = shaderScript.firstChild;
    while (k) {
        if (k.nodeType == 3) {   str += k.textContent;    }
        k = k.nextSibling;
    }

    var shader;
    if (shaderScript.type == "x-shader/x-fragment") {
        shader = gl.createShader(gl.FRAGMENT_SHADER);
    } else if (shaderScript.type == "x-shader/x-vertex") {
        shader = gl.createShader(gl.VERTEX_SHADER);
    } else {  return null;   }

    gl.shaderSource(shader, str);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert(gl.getShaderInfoLog(shader));
        return null;
    }
    return shader;
}

var shaderProgram;
function initShaders() {
    var fragmentShader = getShader(gl, "shader-fs");
    var vertexShader = getShader(gl, "shader-vs");
    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);
    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Шейдеры не инициализируются");     }
    gl.useProgram(shaderProgram);
    shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram,
        "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
    shaderProgram.pMatrixUniform = gl.getUniformLocation(shaderProgram,
        "uPMatrix");
    shaderProgram.mvMatrixUniform = gl.getUniformLocation(shaderProgram,
        "uMVMatrix");
```

```
    }

    var mvMatrix = mat4.create();
    var pMatrix = mat4.create();
    function setMatrixUniforms() {
        gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, pMatrix);
        gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, mvMatrix);
    }
    var vertexBuffer;
    function initBuffers() {
        vertexBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
        vertices = [  0, 0, 0,  -1, 0, 0,   -0.7, 0.7, 0,   0, 1, 0,     0.7, 0.7, 0,   1, 0, 0  ];
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
            gl.STATIC_DRAW);
        vertexBuffer.itemSize = 3;
        vertexBuffer.numItems = 6;
    }

    function drawScene() {
        gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
        mat4.perspective(pMatrix, 45, gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
        mat4.identity(mvMatrix);
        mat4.translate(mvMatrix, mvMatrix, [0.0, -0.5, -2.0]);
        gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
        gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
            vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);
        setMatrixUniforms();

        var selDrawType = document.getElementById("selDrawType").value;
        var drawType = eval(selDrawType);
        gl.drawArrays(drawType, 0, vertexBuffer.numItems);
    }

    function webGLStart() {
        var canvas = document.getElementById("canvas");
        initGL(canvas);
        initShaders();
        initBuffers();
        gl.clearColor(0.2, 0.2, 0.2, 1.0);
        drawScene();
    }
</script>
</head>
<body onload="webGLStart();">
    <select id="selDrawType" style="margin: 20px;"
            onchange="drawScene(this.value)">
        <option>gl.POINTS</option>
        <option>gl.LINES</option>
        <option>gl.LINE_STRIP</option>
        <option>gl.LINE_LOOP</option>
```

```
    <option>gl.TRIANGLES</option>
    <option>gl.TRIANGLE_STRIP</option>
    <option>gl.TRIANGLE_FAN</option>
  </select>
  <br />
  <canvas id="canvas" width="500" height="500"></canvas>
</body>
</html>
```

Результат полученной страницы:

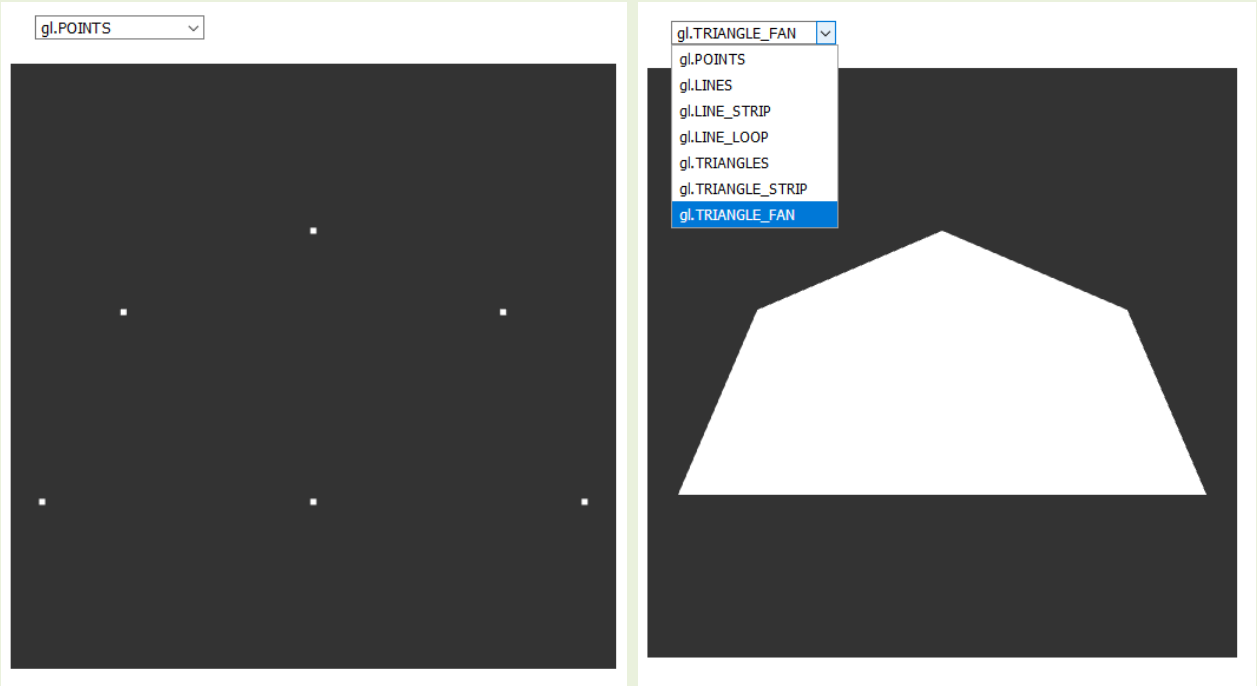

Рис. 2.20

Контрольные вопросы

1. Конвеер WebGL.
2. Построение графических примитивов в WebGL.