Лабораторная работа №2

«Знакомство с WebGL»

Оглавление

Первая WebGL-программа: очистка области рисования

Начнем знакомство с миром WebGL с самой короткой программы, которая просто очищает область рисования, определяемую тегом `<canvas>`. На рис. 1 показан результат загрузки программы, очищающей прямоугольную область простой заливкой ее черным цветом.

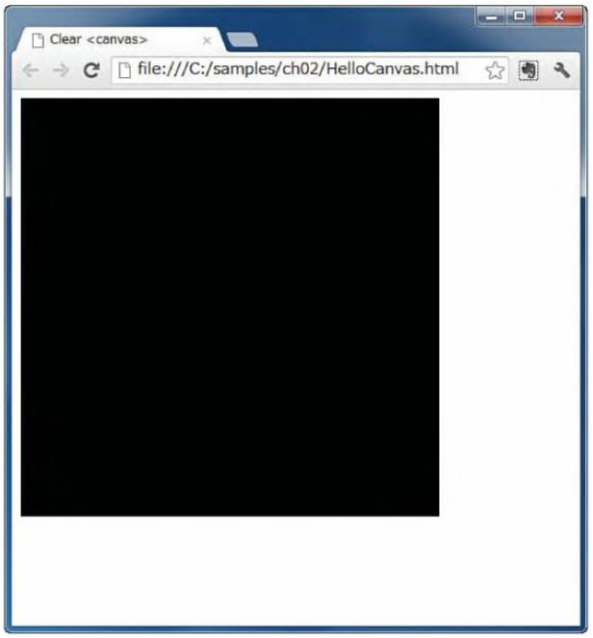

Рис. 1. Результат работы программы

Файл HTML (`2.html`)

Рассмотрим файл `2.html`. Он имеет простую структуру и начинается с определения области рисования в виде элемента `<canvas>`, и затем импортирует файл `2.js`.

Рассмотрим теперь WebGL-программу в файле `2.js`.

Программа на JavaScript (`2.js`)

В начале файла импортируется несколько файлов JavaScript, в которых определяются удобные вспомогательные функции, упрощающие программирование графики WebGL. Подробнее эти файлы будут описываться позже, а пока будем просто считать их библиотеками.

Файл `2.js` выполняет те же 3 этапа, которые выполнялись в лабораторной работе №1: получение ссылки на элемент `<canvas>`, получение контекста отображения и рисование.

Как и в лабораторной работе №1, здесь определена единственная функция `main()`, которая запускается по событию `onload` класса `window`.

На рис. 2 показано, как действует функция `main()` в нашей WebGL-программе, выполняющая четыре шага, которые обсуждаются далее по очереди.

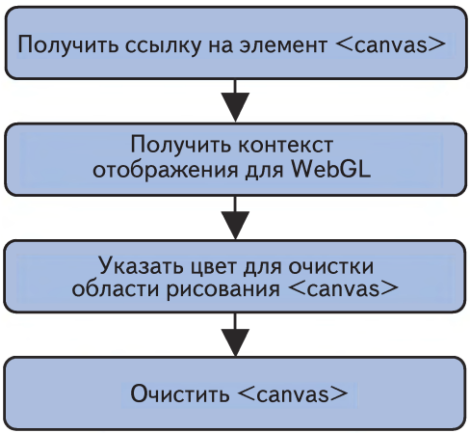

Рис. 2. Порядок работы функции `main()`

Получить ссылку на элемент <canvas>

Сначала функция `main()` получает ссылку на элемент `<canvas>` в файле HTML. Как описывалось в лабораторной работе №1, для этого вызывается метод `document.getElementById()`, которому в качестве аргумента передается идентификатор элемента `'mycanvas'`. Если взглянуть на содержимое файла `2.html`, можно увидеть, что этот идентификатор определен в атрибуте `id` тега `<canvas>`.

Значение, возвращаемое этим методом, сохраняется в переменной `canvas`.

Получить контекст отображения для WebGL

На следующем шаге программа использует переменную `canvas` с целью получения контекста отображения для WebGL. Для получения контекста WebGL, используется метод `canvas.getContext()`, как описывалось выше. В этот метод передается значение `'webgl2'`, т.к. на данный момент актуальной является версия WebGL 2.0.

Теперь контекст поддерживает возможность рисования трехмерной графики вместо двухмерной, то есть предоставляет доступ к методам WebGL. Для сохранения контекста будем использовать переменную `gl`. Это сделает вызовы методов WebGL больше похожими на вызовы методов, определяемых спецификацией OpenGL ES 2.0, которая является основой для спецификации WebGL. Например, сравните вызов `gl.clearColor()` в нашей программе с именем метода `glClearColor()` из OpenGL ES 2.0 или OpenGL.

Указать цвет для очистки области рисования <canvas>

После получения контекста отображения для WebGL, следующим шагом является установка цвета для очистки области рисования, определяемой элементом `<canvas>`. Для этого вызывается метод `gl.clearColor()`, устанавливающий цвет в формате RGBA.

`gl.clearColor(red, green, blue, alpha)`
Устанавливает цвет для очистки (заливки) области рисования.

Параметры:

`red`	Определяет значение красной составляющей цвета (от 0.0 до 1.0).
`green`	Определяет значение зеленой составляющей цвета (от 0.0 до 1.0).
`blue`	Определяет значение синей составляющей цвета (от 0.0 до 1.0).
`alpha`	Определяет значение альфа-канала (прозрачности) цвета (от 0.0 до 1.0). Значение 0.0 соответствует полностью прозрачному цвету, значение 1.0 – полностью непрозрачному.

Любые значения меньше 0.0 и больше 1.0 в этих параметрах усекаются до 0.0 и 1.0, соответственно.

Возвращаемое значение: нет

Ранее в лабораторной работе №1 значения составляющих цвета определялись числами в диапазоне от 0 до 255. Однако, так как технология WebGL основана на OpenGL, в ней используются значения в диапазоне от 0.0 до 1.0, традиционные для OpenGL. Чем больше значение, тем выше интенсивность составляющей цвета. Аналогично, более высокое значение параметра `alpha` (четвертый параметр) соответствует меньшей прозрачности (то есть, большей непрозрачности).

После того, как программа установит цвет для очистки, он сохраняется в системе WebGL и не изменяется, пока не будет указан другой цвет вызовом `gl.clearColor()`.

Очистить <canvas>

Наконец, чтобы очистить область рисования выбранным цветом, вызывается метод `gl.clear()`. Обратите внимание, что аргументом этого метода является значение `gl.COLOR_BUFFER_BIT`. Метод `gl.clear()` основывается на спецификации OpenGL, которая использует более сложную модель, чем простые области рисования, манипулируя множеством внутренних буферов. Один такой буфер – буфер цвета – используется в данном примере. Указав значение `gl.COLOR_BUFFER_BIT`, мы сообщили системе WebGL, что для очистки области рисования он должен использовать буфер цвета. Помимо буфера цвета, в WebGL имеется еще буфер глубины и буфер трафарета. С буфером глубины мы будем знакомиться далее при изучении трехмерной графики. Буфер трафарета мы рассматривать не будем, потому что он редко используется на практике.

Очистка буфера цвета фактически вынуждает WebGL очистить область рисования `<canvas>` на веб-странице.

`gl.clear(buffer)`
Очищает указанный буфер предварительно определенными значениями. В случае буфера цвета, значение (цвет) устанавливается вызовом метода `gl.clearColor()`.

Параметры:

`buffer`	Определяет очищаемый буфер. С помощью оператора `|` (поразрядная операция «ИЛИ») можно указать несколько буферов.
`gl.COLOR_BUFFER_BIT`	Определяет буфер цвета.
`gl.DEPTH_BUFFER_BIT`	Определяет буфер глубины.
`gl.STENCIL_BUFFER_BIT`	Определяет буфер трафарета.

Возвращаемое значение: нет

Ошибки: `INVALID_VALUE`

Эта ошибка возникает, когда аргумент `buffer` имеет значение, отличное от любого из трех, указанных выше.

Если цвет не был указан, то есть, если метод `gl.clearColor()` еще не вызывался, используются значения по умолчанию, перечисленные в табл. 1.

Табл. 1. Значения по умолчанию для очистки буферов и соответствующие методы установки значений очистки

Буфер	Значение по умолчанию	Метод установки
Буфер цвета	(0.0, 0.0, 0.0, 0.0)	`gl.clearColor(red, green, blue, alpha)`
Буфер глубины	1.0	`gl.clearDepth(depth)`
Буфер трафарета	0	`gl.clearStencil(s)`

Эксперименты с примером программы

Попробуйте изменить цвет очистки следующим образом:

```
gl.clearColor(0.0, 0.0, 1.0, 1.0);
```

После перезагрузки `2.html` в браузере, файл `2.js` также перезагрузится, затем будет вызвана функция `main()`, которая очистит область рисования, окрасив ее в синий цвет.

Попробуйте указать другие цвета и посмотрите, что из этого получится. Например, после вызова `gl.clearColor(0.5, 0.5, 0.5, 1.0)` область будет залита серым цветом.

Рисование точки (версия 1)

Мы научились инициализировать контекст WebGL и использовать некоторые его методы. Сделаем еще один шаг вперед и рассмотрим программу рисования самой простой фигуры – точки. Программа будет рисовать красную точку размером `10` пикселей в позиции с координатами (0.0,0.0,0.0). Так как система WebGL предназначена для работы с трехмерной графикой, позиция точки определяется тремя координатами. С описанием системы координат мы познакомимся позже, а пока просто будем иметь в виду, что точка с координатами (0.0,0.0,0.0) соответствует центру области рисования `<canvas>`.

Как показано на рис. 3, программа рисует красную точку (прямоугольник) в центре области рисования `<canvas>`, которая очищается черным цветом. Некоторые браузеры (в частности Firefox) могут неправильно отображать примеры без использования буферных объектов. Поэтому, если возникают проблемы, попробуйте воспользоваться другим браузером. В качестве точек мы будем использовать закрашенные прямоугольники вместо кругов, потому что рисование прямоугольника выполняется быстрее.

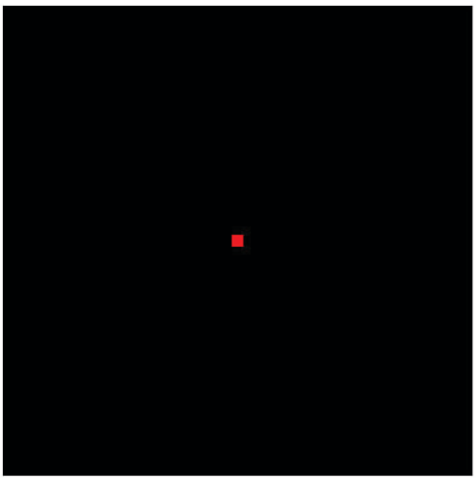

Рис. 3. Красная точка в центре области рисования

Как и в случае с очисткой области рисования в предыдущем разделе, цвет точки требуется указать в формате RGBA. Чтобы получить красный цвет, составляющая R должна иметь значение 1.0, составляющая G – значение 0.0, B – 0.0 и A – 1.0. Как вы наверняка помните, в лабораторной работе №1 сначала выполняется установка цвета, а затем осуществляется рисование прямоугольника:

```
ctx.fillStyle='rgba(0, 255, 0, 1.0)';
ctx.fillRect(120, 10, 150, 150);
```

Соответственно, можно предположить, что в WebGL-приложении подобные операции выполняются некоторым похожим образом, например:

```
gl.drawColor(1.0, 0.0, 0.0, 1.0);
gl.drawPoint(0, 0, 0, 10); // Координаты центра и размеры точки
```

К сожалению, это не так. WebGL опирается на механизм рисования нового типа, который называется **шейдером** (shader), обладающий большей гибкостью и широтой возможностей при

рисовании двух- и трехмерных объектов, и который должен использоваться всеми WebGL-приложениями. Однако шейдеры – не только более мощный, но также и более сложный механизм, и при его применении невозможно просто использовать элементарные операции рисования.

Шейдеры являются критически важным механизмом в программировании с применением технологии WebGL.

Дополним содержимое файла `2.js` двумя шейдерами.

Что такое шейдер?

Шейдеры необходимы, когда требуется что-то нарисовать с помощью WebGL.

Для рисования с помощью WebGL требуется два типа шейдеров:

- **вершинный шейдер (vertex shader)** – это программа, описывающая характеристики вершины (координаты, цвет и другие), а вершина – это точка в двух- или трехмерном пространстве, например, угол или вершина двух- или трехмерной фигуры;
- **фрагментный шейдер (fragment shader)** – это программа, реализующая обработку фрагментов изображений, например, определение освещенности, где под фрагментом подразумевается простейший элемент изображения, своего рода «пиксель».

Наша цель состоит в том, чтобы нарисовать точку размером 10 пикселей. Для этого используются два шейдера:

- вершинный шейдер определяет координаты точки и ее размер; в данном примере указаны координаты (0.0,0.0,0.0) и размер 10.0;
- фрагментный шейдер определяет цвет фрагментов точки; в данном примере выбран красный цвет (1.0,0.0,0.0,1.0).

Структура WebGL-программы, использующей шейдеры

Текст программ вершинного и фрагментного шейдера написан на языке шейдеров OpenGL ES (GLSL ES).

Инициализация шейдеров

Подготовка и передача в систему WebGL шейдеров, выполняется с помощью вспомогательной функции `initShaders()`, которая определена в файле `cuon-util.js`:

`initShaders(gl, vshader, fshader)`
Инициализирует шейдеры и передает их системе WebGL.

Параметры:

`gl`	Контекст отображения.
`vshader`	Вершинный шейдер (строка).
`fshader`	Фрагментный шейдер (строка).

Возвращаемое значение:

`true`	Шейдеры успешно инициализированы
`false`	Инициализация не увенчалась успехом.

Как уже упоминалось, для рисования точки нам необходимо определить три ее характеристики: координаты, размер и цвет:

- вершинный шейдер определяет координаты точки и ее размер; в данном примере точка имеет координаты (0.0,0.0,0.0) и размер 10.0;

- фрагментный шейдер определяет цвет точки; в данном примере точка имеет красный цвет (1.0,0.0,0.0,1.0).

Вершинный шейдер

Вершинный шейдер содержит определение единственной функции `main()`, напоминающее определения аналогичных функций в языках, подобных C. Ключевое слово `void` перед именем `main()` указывает, что эта функция не имеет возвращаемого значения. Кроме того, отсутствует возможность передачи каких-либо аргументов функции `main()`.

Так же, как и в языке C, для присваивания значений переменным, в шейдере можно использовать оператор =. Координаты точки присваиваются переменной `gl_Position`, а размер точки присваиваются переменной `gl_PointSize`. Это встроенные переменные, доступные только внутри вершинного шейдера и имеющие специальное значение: `gl_Position` определяет координаты вершины (в данном случае - координаты точки), а `gl_PointSize` определяет размеры точки (см. табл. 2).

Табл. 2. Встроенные переменные, доступные в вершинном шейдере

Тип и имя переменной	Описание
`vec4 gl_Position`	Определяет координаты вершины
`float gl_PointSize`	Определяет размер точки (в пикселях)

Вершинный шейдер всегда должен присваивать значение переменной `gl_Position`. Если этого не сделать, поведение шейдера (в зависимости от реализации) может оказаться отличным от ожидаемого. Присваивать значение переменной `gl_PointSize`, напротив, требуется только при рисовании точек. Допустимые значения размеров точки зависят от оборудования.

В отличие от JavaScript, язык GLSL ES является типизированным; то есть он требует указания типов данных у переменных. В табл. 3 перечислены типы, поддерживаемые языком GLSL ES и используемые в этой лабораторной работе.

Табл. 3. Типы данных в языке GLSL ES

Тип	Описание
`float`	Вещественное число (число с плавающей точкой).
`vec4`	Вектор с четырьмя вещественными числами `float` `float` `float` `float`

При попытке присвоить переменной данные несоответствующего типа, генерируется ошибка. Например, переменная `gl_PointSize` имеет тип `float`, соответственно ей можно присвоить только вещественное значение. То есть, если изменить строку:

```
gl_PointSize = 10.0;
```

на

```
gl_PointSize = 10;
```

будет сгенерирована ошибка, просто потому, что число `10` интерпретируется как целочисленное значение, в отличие от числа `10.0`, которое в языке GLSL ES считается вещественным значением.

Встроенная переменная `gl_Position`, определяющая координаты вершины, имеет тип `vec4` – вектор, состоящий из четырех вещественных значений. Однако, все, что у нас имеется - это три отдельных значения (0.0,0.0,0.0), представляющих координаты X, Y и Z. Поэтому нужен какой-то механизм, с помощью которого можно было бы преобразовать их в вектор типа `vec4`. К счастью такой механизм имеется – встроенная функция `vec4()`, которая преобразует свои аргументы в вектор `vec4` и возвращает его, то есть как раз то, что нам нужно!

`vec4 vec4(v0, v1, v2, v3)`
Конструирует объект типа `vec4` из четырех значений `v0`, `v1`, `v2` и `v3`

Параметры:

`v0`, `v1`, `v2`, `v3`	Вещественные числа.

Возвращаемое значение:

объект `vec4`, включающий значения `v0`, `v1`, `v2`, `v3`.

В четвертом элементе вектора, который присваивается переменной `gl_Position`, содержится значение 1.0. Такие четырехкомпонентные координаты называют однородными координатами и часто используются в компьютерной графике для эффективной обработки трехмерной информации. Несмотря на то, что однородные координаты являются четырехкомпонентными, если четвертый компонент однородной координаты равен 1.0, такая координата соответствует той же позиции, которая может быть описана аналогичной трехкомпонентной координатой. То есть, определяя координаты вершины, указывайте в четвертом компоненте значение 1.0.

Фрагментный шейдер

После определения координат и размера точки необходимо также с помощью фрагментного шейдера определить ее цвет. **Фрагмент** – это пиксель, отображаемый на экране, который, технически, определяется позицией, цветом и другой информацией.

Фрагментный шейдер – это программа, обрабатывающая данную информацию в процессе подготовки фрагмента к отображению на экране. Фрагментный шейдер так же как вершинный шейдер начинает выполнение с функции `main()`.

Задача шейдера состоит в том, чтобы установить цвет для каждого фрагмента. Переменная `gl_FragColor` – это встроенная переменная, доступная только во фрагментном шейдере; она определяет цвет фрагмента, как показано в табл. 4.

Табл. 4. Встроенное значение, доступное во фрагментном шейдере

Тип и имя переменной	Описание
`vec4 gl_FragColor`	Определяет цвет фрагмента (в формате RGBA).

Когда мы присваиваем значение цвета встроенной переменной, фрагмент отображается с использованием этого цвета. Так же как координаты в вершинном шейдере, значение цвета имеет тип `vec4` и состоит из четырех вещественных чисел - значений RGBA. В данном примере точка будет окрашена в красный цвет, потому что переменной присваивается значение (1.0,0.0,0.0,1.0).

Операция рисования

После установки шейдеров остается только выполнить операцию рисования фигуры – в нашем случае точки. После очистки области рисования можно нарисовать точку с помощью `gl.drawArrays()`. `gl.drawArrays` – мощная функция, которая способна рисовать различные простейшие фигуры.

gl.drawArrays(mode, 0, 1)
Выполняет вершинный шейдер, чтобы нарисовать фигуры, определяемые параметром mode.

Параметры:

mode	Определяет тип фигуры. Допустимыми значениями являются следующие константы: gl.POINTS, gl.LINES, gl.LINE_STRIP, gl.LINE_LOOP, gl.TRIANGLES, gl.TRIANGLE_STRIP и gl.TRIANGLE_FAN.
first	Определяет номер вершины, с которой должно начинаться рисование (целое число).
count	Определяет количество вершин (целое число).

Возвращаемое значение:

нет

Ошибки:

INVALID_ENUM	В параметре mode передано значение, не являющееся ни одной из констант, перечисленных выше.
INVALID_VALUE	Отрицательное значение first и/или count.

В данном примере мы рисуем точку, поэтому в первом параметре mode передается значение gl.POINTS. Во втором параметре передается 0, потому что рисование начинается с первой вершины. В третьем параметре count передается 1, потому что программа рисует только одну точку.

Теперь, когда программа вызовет gl.drawArrays(), вершинный шейдер выполнится count раз, по одному разу для каждой вершины. В данном примере шейдер выполняется только один раз (параметр count имеет значение 1), потому что определена только одна вершина: наша точка. При вызове шейдера выполняется его функция main() – строка за строкой, – в результате чего переменной gl_Position присваивается значение (0.0,0.0,0.0,1.0) и переменной gl_PointSize – значение 10.0.

Вслед за вершинным шейдером вызывается фрагментный шейдер и выполняется его функция main(), которая в этом примере присваивает переменной gl_FragColor значение, соответствующее красному цвету. Как результат, в позиции с координатами (0.0,0.0,0.0,1.0) выводится красная точка размером 10 пикселей, то есть, в центре области рисования (см. рис. 3).

К данному моменту у вас должно сложиться представление о назначении вершинного и фрагментного шейдеров и об особенностях их работы. Давайте теперь посмотрим, как WebGL описывает позиции фигур с применением системы координат.

Система координат WebGL

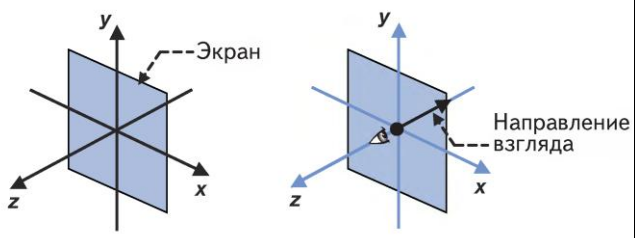

Рис. 4. Система координат в WebGL

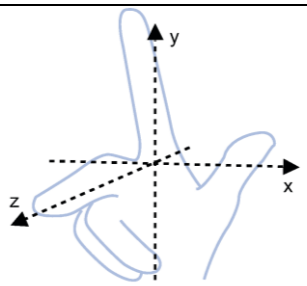

Рис. 5. Правосторонняя система координат

Поскольку система WebGL работает с трехмерной графикой, она использует трехмерную систему координат с тремя осями: *x*, *y* и *z*. Эта система координат проста и понятна, потому что окружающий нас мир тоже имеет три измерения: ширину, высоту и глубину. В любой системе координат направление осей играет важную роль. Вообще, в WebGL, когда вы сидите перед плоскостью экрана монитора, ось *x* простирается по горизонтали, (слева направо), ось *y* - по вертикали (снизу вверх) и ось *z* – в направлении от плоскости экрана к пользователю (рис. 4). Глаз пользователя находится в начале координат (0.0,0.0,0.0) и смотрит вдоль оси *z*, в сторону отрицательных значений – за плоскость экрана (рис. 4). Такая система координат является правосторонней, потому что ее можно изобразить пальцами правой руки (см. рис. 5), и она же обычно используется при работе с WebGL.

Как вы уже знаете из лабораторной работы №1, область рисования, которая определяется элементом `<canvas>`, имеет свою систему координат, отличную от системы координат WebGL, поэтому необходим некоторый механизм отображения одной системы в другую. По умолчанию, как показано на рис. 6, WebGL выполняет такое отображение следующим образом:

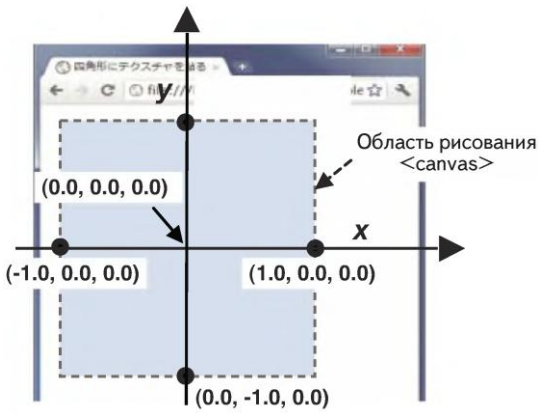

Рис. 6. Область рисования `<canvas>` и система координат WebGL

- центр области рисования `<canvas>`: (0.0,0.0,0.0);
- две вертикальные границы области рисования `<canvas>`: (-1.0,0.0,0.0) и (1.0,0.0,0.0);
- две горизонтальные границы области рисования `<canvas>`: (0.0,-1.0,0.0) и (0.0,1.0,0.0).

Это отображение выполняется по умолчанию. Однако, в WebGL имеется возможность переопределить систему координат по умолчанию. Кроме того, пока мы изучаем 2d-графику, в примерах программ будут использоваться только оси *x* и *y*, а координата глубины (ось *z*) всегда будет иметь значение 0.0.

Эксперименты с примером программы

Для начала попробуем изменить координаты точки, чтобы получить более ясное представление о системе координат WebGL. Например, давайте изменим координату *x*, подставив значение 0.5 вместо 0.0, как показано ниже:

```
gl_Position = vec4(0.5, 0.0, 0.0, 1.0);
```

Сохраните измененный файл и щелкните на кнопке Reload (Обновить) браузера, чтобы перезагрузить его. Вы должны увидеть, что точка сместилась и теперь находится в правой половине области рисования.

Теперь изменим координату *y*, чтобы сместить точку вверх, как показано ниже:

```
gl_Position = vec4(0.0, 0.5, 0.0, 1.0);
```

Сохраните измененный файл и перезагрузите его. На этот раз вы должны увидеть, что точка находится в верхней половине области рисования.

Проведем еще один эксперимент: попробуем изменить цвет точки с красного на зеленый:

```
gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
```

Рисование точки (версия 2)

Мы исследовали порядок рисования точки и необходимые для этого функции шейдеров. Теперь, когда получено некоторое представление об основных особенностях WebGL-программ, мы займемся исследованием механизма передачи данных между JavaScript и шейдерами. Сейчас программа всегда рисует точку в одном и том же месте, жестко определенном в вершинном шейдере. Это делает пример проще для понимания, но лишает его гибкости. Модифицируем нашу программу так, чтобы она могла передавать координаты вершины из JavaScript в вершинный шейдер, и затем рисовать точку в новом местоположении.

Использование `attribute`-переменных

Наша цель – передать координаты точки из программного кода на JavaScript в вершинный шейдер. Сделать это можно двумя способами: с помощью `attribute`-переменных (переменных со спецификатором `attribute`) и `uniform`-переменных (переменных со спецификатором `uniform`). Выбор того или иного спецификатора зависит от природы данных. `attribute`-переменные передают данные, уникальные для каждой вершины, тогда как `uniform`-переменные передают одни и те же данные для всех вершин. В данном примере мы будем использовать `attribute`-переменную, потому что обычно все вершины имеют уникальные координаты.

Чтобы задействовать `attribute`-переменные, необходимо выполнить следующие три шага:

1. Объявить `attribute`-переменную в вершинном шейдере;

2. Присвоить встроенной переменой `gl_Position` значение `attribute`-переменной;

3. Передать данные из основной программы в `attribute`-переменную шейдера.

Объявление `attribute`-переменной осуществляется в шейдере. Ключевое слово `attribute` называется спецификатором класса хранения и указывает, что следующая за ним переменная (в данном случае `a_Position`) является `attribute`-переменной. Эта переменная должна быть объявлена глобальной, потому что данные в нее записываются за пределами шейдера. Объявление должно следовать стандартному шаблону:

```
<Storage Qualifier> <Type> <Variable Name>
```

как показано на рис. 7.:

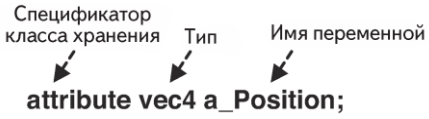

Рис. 7. Объявление `attribute`-переменной

Переменная `a_Position` объявляется как `attribute`-переменная с типом `vec4`, потому что ее значение будет присваиваться встроенной переменной `gl_Position`, которая, как было показано в табл. 2, всегда имеет тип `vec4`.

Мы будем использовать соглашение об именовании переменных, в соответствии с которым имена `attribute`-переменных должны начинаться с префикса `a_`, а имена `uniform`-переменных – с префикса `u_`, чтобы при просмотре программного кода было проще определять типы переменных по их именам.

После объявления переменной `a_Position`, ее значение присваивается встроенной переменной `gl_Position`.

На этом подготовку шейдера к приему данных извне можно считать законченной. Следующий шаг – передача данных в `attribute`-переменную из программы на JavaScript.

Получение ссылки на `attribute`-переменную

Как мы уже знаем, вершинный шейдер передается в систему WebGL с помощью вспомогательной функции `initShaders()`. Когда осуществляется передача вершинного шейдера в WebGL, система анализирует его, обнаруживает `attribute`-переменную и выделяет область памяти для хранения ее значения. Чтобы передать данные в переменную `a_Position`, находящуюся внутри шейдера, необходимо использовать значение местоположения (location) этой переменной.

Присваивание значения `attribute`-переменной

Зная местоположения (location) переменной, ей необходимо присвоить значение. Эта операция выполняется с помощью метода `gl.vertexAttrib3f()`.

В первом аргументе методу передается местоположение (location) переменной шейдера. Во втором, третьем и четвертом аргументах передаются вещественные значения, представляющие координаты *x*, *y* и *z* точки. После вызова метода эти три значения будут записаны в `attribute`-переменную `a_Position` внутри вершинного шейдера.

Затем значение переменной `a_Position` присваивается встроенной переменной `gl_Position` внутри вершинного шейдера. В результате этого координаты *x*, *y* и *z*, которые передаются из JavaScript через `attribute`-переменную в шейдер, оказываются в переменной `gl_Position`. То есть, данная программа дает тот же результат, что и предыдущая программа, где координаты точки определяются переменной `gl_Position`. Однако на этот раз значение переменной `gl_Position` устанавливается динамически, из JavaScript.

В завершение выполняется очистка области рисования `<canvas>` вызовом `gl.clear()` и рисование точки вызовом `gl.drawArrays()`, точно так же, как в предыдущей программе.

И последнее замечание. Как можно видеть переменная `a_Position` объявлена с типом `vec4`. Однако в вызов метода передается только три значения (координаты *x*, *y* и *z*), а не четыре.

Дело в том, что этот метод автоматически подставит значение 1.0 вместо отсутствующего четвертого значения.

Эксперименты с примером программы

Теперь, когда у нас имеется программа, передающая координаты точки из кода на JavaScript в вершинный шейдер, можно попробовать поэкспериментировать с этими координатами. Например, если вы пожелаете отобразить точку в позиции с координатами (0.5,0.0,0.0), вы могли бы изменить программу, как показано ниже:

```
gl.vertexAttrib3f(a_Position, 0.5 , 0.0, 0.0);
```

Для решения той же задачи можно было бы использовать другие методы семейства `gl.vertexAttrib3f()`:

```
gl.vertexAttrib1f(a_Position, 0.5);
gl.vertexAttrib2f(a_Position, 0.5, 0.0);
gl.vertexAttrib4f(a_Position, 0.5, 0.0, 0.0, 1.0);
```

Задание для самостоятельной работы

- Используя `attribute`-переменные, попробуйте применить тот же подход к изменению размера точки из программы на JavaScript.
- Визуализируйте несколько точек в разных местах и разного размера.