Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

Л. К. Титова, С. О. Ломако

ПРОГРАММИРОВАНИЕ ГРАФИКИ В WEB

ПРАКТИКУМ
по выполнению лабораторных работ по одноименной дисциплине для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» дневной формы обучения

Гомель 2021

УДК 004.738.52(075.8)
ББК 32.973.4я73
Т45

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 10 от 01.06.2020 г.*)

Рецензент: доц. каф. «Промышленная электроника» ГГТУ им. П. О. Сухого
канд. техн. наук *А. В. Ковалев*

Титова, Л. К.

Т45 Программирование графики в Web : практикум по выполнению лаборатор. работ по одноим. дисциплине для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» днев. формы обучения / Л. К. Титова, С. О. Ломако. – Гомель : ГГТУ им. П. О. Сухого, 2021. – 149 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Содержит теоретические сведения и задания для выполнения студентами лабораторных работ по учебной дисциплине «Программирование графики в Web», способствующие усвоению пройденного теоретического материала. Темы лабораторных работ ориентированы на получение студентами навыков проектирования графики для Web-сайтов с использованием современных Web-технологий, а также помогут овладеть практическими приемами Web-конструирования и Web-программирования.

Для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» дневной формы обучения.

УДК 004.738.52(075.8)
ББК 32.973.4я73

ВВЕДЕНИЕ

Индустрия услуг и развлечений в сети Интернет стремительно развивается, ведущие разработчики программного обеспечения улучшают поддержку двухмерной и трехмерной графики в своих продуктах. Традиционно ее поддержка ограничивалась высокопроизводительными компьютерами или специализированными игровыми консолями, а программирование требовало применения сложных алгоритмов. Однако благодаря росту производительности персональных компьютеров и расширению возможностей браузеров стало возможным создание и отображение высококаественной графики с применением веб-технологий.

С развитием *HTML,* разработчики получили возможность создавать все более сложные веб-приложения. На заре своего развития язык *HTML* предлагал только возможность отображения статического контента, но с добавлением поддержки *JavaScript* стало возможным реализовывать более сложные взаимодействия элементов и отображения динамического контента. Внедрение стандарта *HTML5* позволило использовать новые возможности, включая поддержку двухмерной графики в виде тега *canvas*. Создание технологии *WebGL* позволило отображать и манипулировать трехмерной графикой на веб-страницах с помощью *JavaScript*. При помощи *WebGL* разработчики могут создавать совершенно новые пользовательские интерфейсы, трехмерные игры и использовать трехмерную графику для визуализации различной информации. Несмотря на внушительные возможности, *WebGL* отличается от других технологий доступностью и простотой использования, что способствует ее быстрому распространению.

Учебный план специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» предполагает изучение дисциплины «Программирование графики в Web». Учебная дисциплина знакомит студентов с разработкой интерактивных графических приложений, возможностями перехода от *2D*-графики к *3D*-графике и подходами к разработке гипертекстовых документов с графикой, предназначенных для публикации в глобальной компьютерной сети *Internet*. Цель дисциплины – получение теоретических и практических знаний использования современных *web*-технологий, овладение практическими приемами *Web*-

конструирования и *Web*-программирования. Изучение дисциплины состоит из двух разделов:

1. *JavaScript*: Рисование на холсте.

2. *WebGL* – технология для рисования и отображения интерактивной *2D*- и *3D*-графики в веб-браузерах.

Canvas – элемент *HTML5*, предназначенный для создания растрового двухмерного изображения при помощи скриптов на языке *JavaScript*. Используется, как правило, для отрисовки графиков для статей и игрового поля в некоторых браузерных играх, но также может использоваться для встраивания видео в страницу и создания полноценного плеера. Также используется в *WebGL* для аппаратного ускорения *3D* графики. Преимущества *Canvas:* в отличие от *SVG* гораздо удобнее иметь дело с большим числом элементов; имеет аппаратное ускорение; можно манипулировать каждым пикселем, применять фильтры обработки изображений; есть много библиотек.

WebGL (библиотека веб-графики) – это *API*-интерфейс *JavaScript* для рендеринга высокопроизводительной интерактивной *3D* и *2D*-графики в любом совместимом веб-браузере без использования плагинов. WebGL делает это, вводя *API*, полностью соответствующий *OpenGL ES* 2.0, который можно использовать в *canvas* элементах *HTML5*. Это соответствие позволяет *API* использовать преимущества аппаратного ускорения графики, предоставляемого устройством пользователя.

1 *Canvas* – элемент *HTML5*, позволяющий рисовать графику на *JavaScript*

1.1 Введение в *Canvas*

Разработчики или дизайнеры могут создавать интерфейсы с помощью технологий, основанных на стандартах *HTML5*. Это позволяет существенно улучшить взаимодействие с пользователем, поскольку установка подключаемых модулей не требуется. В настоящее время за графику, как правило, отвечает браузер.

Одной из интересных возможностей нового стандарта *HTML5* является элемент *canvas*, или холст. *HTML5 Canvas* – веб-технология, которая позволяет использовать высококачественную графику в браузерах.

Холст предназначен для создания растровых изображений на странице средствами графического движка браузера. *Canvas* способен изображать не только статические, но и динамические изображения (анимацию).

До появления *canvas* для вставки анимации могли использоваться *gif*- изображения, *flash*-анимация или, основанные на скриптах или других подключаемых модулях, другие решения (в частности *Silverlight, Java Applets, ActiveX* и другие). Каждое из этих решений имеет ряд недостатков. Например, плохое качество анимации *gif*-изображений, большой размер загружаемых модулей *Java*, несоответствие версий *Flash* проигрывателя, *ActiveX* работает исключительно в *Internet Explorer* и многое другое. Но главным фактором появления анимации и графики на основе ‹*canvas*› является высокий рост мобильного сегмента Интернет, особенно устройств под управлением операционных систем *Android* и *iOS*. Установить плагин на них нет возможности, а *i*-устройства не поддерживают *Flash*, а начиная с версии 4.0, его также не поддерживают устройства под управлением *Android*. Поэтому для поддержки огромной доли рынка мобильных устройств с возможностью подключения к сети Интернет анимацию начали создавать при помощи *JavaScript*. Для этого обычно используют библиотеки, как например *jQuery* или *Prototype*. С введением в действие стандарта *CSS3* часть анимаций возможно создавать с помощью каскадных таблиц стилей. Однако самыми широкими возможностями по созданию изображений и анимаций пользуется стандарт *HTML5* и его новый элемент ‹*canvas*›.

Элемент *‹canvas›* – это новый элемент *HTML5*, который позволяет создавать изображения на сайте с помощью *JavaScript*. Область использования холстов довольно широкая. Чаще всего его можно увидеть при создании деловой графики (чарты, диаграмы, графики), а также для рендеринга браузерных игр (чаще всего встречаются в социальных сетях). У *‹canvas›* есть только два атрибута – ширина и высота.

Элемент *‹canvas›* создает контекст отрисовки, на котором в будущем можно создавать и манипулировать объектами *JavaScript*. Другими словами, *‹canvas›* представляет собой прямоугольную область, в которой с помощью *javascript* можно «рисовать».

На сегодняшний день стандарт полностью описывает работу двумерных контекстов (для плоской графики, 2*D*). Элемент *‹canvas›* также используется технологией *WebGL* для отрисовки аппаратно-ускоренной *3D*-графики на вебстраницах.

Для размещения элемента на странице HTML достаточно указать:

```
‹canvas width=600 height=250› ‹/canvas›
```

После помещения на страницу, элементом *‹canvas›* можно манипулировать различными способами: помещать на него текст, рисовать графические элементы и линии, выполнять заливку, добавлять анимацию. Данные манипуляции совершаются при помощи команд javascript. Чтобы использовать холст программным путем, необходимо получить доступ к его контексту. После этого выполняются все необходимые действия с контекстом и только тогда результат подтверждается и выводится на холст. Сначала изображение создается программно, а потом результат выводится визуально.

Так как не все браузеры поддерживают *HTML5*, то на данное время воспользоваться *‹canvas›* можно только в следующих браузерах (по информации *caniuse.com*): *Internet Explorer 9+; Firefox 2.0+; Chrome 4+; Safari 3.1+; Opera 9.0+; iOS 3.2+; Android 2.1+.*

Холст представлен двумерной сеткой. Начало координат (0,0) находится в левом верхнем углу холста. Если двигаться вправо вдоль оси *X*, значение координаты *X* будет увеличиваться. Аналогично, если двигаться вниз вдоль оси *Y*, будет увеличиваться значение координаты *Y*.

С помощью *canvas* можно рисовать как простейшие графические примитивы – линии, фигуры, текст, так и создавать сложные графические игры. В дополнение, *canvas* позволяет манипулировать изображениями и даже видео.

Как правило, для элемента *canvas* задаются ширина и высота с помощью атрибутов *width* и *height*. Если не указать эти атрибуты, то по умолчанию ширина будет составлять *300*, а высота – *150* пикселей.

1.2 Получение контекста рисования

Все рисование на *canvas* производится с помощью кода *JavaScript*. Чтобы начать рисовать на *canvas*, необходимо получить его контекст, то есть экземпляр объекта *CanvasRenderingContext2D*. Сделать это можно следующим образом:

```
var canvas = document.getElementById("Canvas");
var context = canvas.getContext("2d");
```

В первой строке получается сам холст, а во второй, с помощью вызова единственного метода объекта холста *getContext(),* получается контекст этого холста. Параметр *2D* указывает на то, что получаемый контекст будет создавать плоское изображение (экземпляр объекта *CanvasRenderingContext2D*).

Для получения контекста используется функция *getContext("2d")*.

1.3 Рисование прямоугольников

Для рисования простейших фигур – прямоугольников, могут понадобиться три метода:

– *clearRect(x,y,w,h)* – очищает определенную прямоугольную область, верхний левый угол которой имеет координаты *x* и *y*, ширина равна *w*, а высота равна *h;*

– *fillRect(x, y, w,h)* – заливает цветом прямоугольник, верхний леый угол которого имеет координаты *x* и *y*, ширина равна *w*, а высота равна *h;*

– *strokeRect(x,y,w,h)* – рисует контур прямоугольника без заливки его каким-то определенным цветом.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="350" height="200"
  style="background-color:#eee; border: 1px solid red;">
            Ваш браузер не поддерживает Canvas
        </canvas>
     <script>
         var canvas = document.getElementById("Canvas"),
           context = canvas.getContext("2d");
         context.strokeRect(50, 40, 100, 100);
         context.fillRect(200, 40, 100, 100);
        </script>
    </body>
</html>
```

Результат работы в браузере отображён на рисунке 1.1:

Рисунок 1.1 – Результат рисования двух прямоугольников

При рисовании в функции *strokeRect* и *fillRect* передаются координаты *x* и *y* верхнего левого угла прямоугольника относительно элемента *canvas*, а также ширина и высота прямоугольника. При этом началом координат считается верхний левый угол элемента canvas, ось *X* направлена вправо, а ось *Y* направлена вниз (рисунок 1.2):

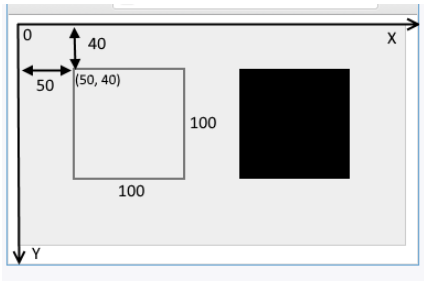


Рисунок 1.2 – Отображение размеров и смещений у прямоугольников относительно начала координат

В отличие от *strokeRect* и *fillRect* метод *clearRect* очищает определенную область.

Фактически эта область приобретатет тот цвет, который у нее был бы, если бы к ней не применялись функции *strokeRect* и *fillRect*. Например:

```
var canvas = document.getElementById("Canvas"),
        context = canvas.getContext("2d");
        context.fillRect(50, 40, 100, 100);
        context.clearRect(50, 40, 40, 40);
```

В данном случае очищается небольшая часть прямоугольника в левом верхнем углу (рисунок 1.3).

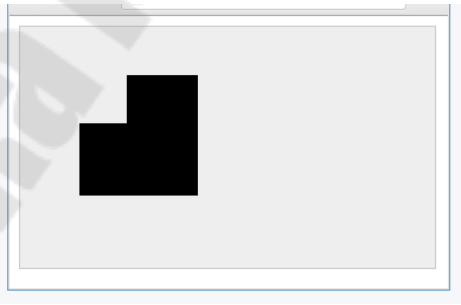


Рисунок 1.3 – Результат очищения выделенной области

1.4 Настройка рисования

Контекст элемента *canvas* предоставляет ряд свойств, с помощью которых можно настроить отрисовку на *canvas*. К подобным свойствам относятся следующие:

– *strokeStyle:* устанавливает цвет линий или цвет контура. По умолчанию установлен черный цвет;

– *fillStyle:* устанавливает цвет заполнения фигур. По умолчанию установлен черный цвет;

– *lineWidth:* устанавливает толщину линий. По умолчанию равно *1.0;*

– *lineJoin:* устанавливает стиль соединения линий;

– *globalAlpha:* устанавливает прозрачность отрисовки на *canvas;*

– *setLineDash:* создает линию из коротких черточек.

Чтобы рисовать какие-то видимые фигуры, необходимо задать цвет. Установить цвет можно разными способами. Во-первых, можно задать цвет контура или границы фигур с помощью свойства *strokeStyle*:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="350" height="200"
                style="background-color:#eee;
                border: 1px solid #ccc;">
            Ваш браузер не поддерживает Canvas
        </canvas>
   <script>
     var canvas = document.getElementById("Canvas"),
          context = canvas.getContext("2d");
         context.strokeStyle = "red";
         context.fillStyle = "blue";
         context.strokeRect(50, 40, 100, 100);
         context.fillRect(200, 40, 100, 100);
     </script>
  </body>
</html>
```

Результат работы данного кода отображен на рисунке 1.4:

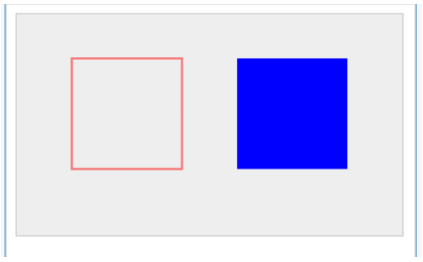


Рисунок 1.4 – Работа с цветом

В качестве значения свойства *strokeStyle* и *fillStyle* получают название цвета в виде строки, либо в виде шестнадцатиричного значения цвета (например, *"#00FFFF"*), либо в виде значений *rgb ("rgb(0, 0, 255)")* и *rgba ("rgba(0, 0, 255, 0.5)")*.

Свойство *lineWidth* позволяет установить толщину линии:

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
    context.strokeStyle = "red";
    context.fillStyle = "blue";
    context.lineWidth = 6.5;
    context.strokeRect(50, 40, 100, 100);
    context.fillRect(50, 40, 100, 100);
```

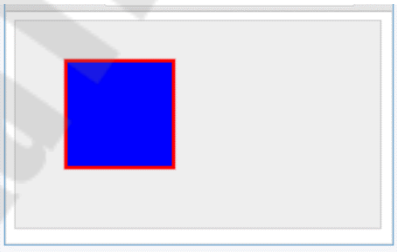


Рисунок 1.5 – Демонстрация работы свойств *strokeStyle* и *fillStyle*

Метод *setLineDash()* в качестве параметра принимает массив чисел, которые устанавливают расстояния между линиями.

Например:

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

```
            <meta charset="utf-8">
            <title>Canvas </title>
          </head>
        <body>
        <canvas id="Canvas" width="450" height="200"
         style="background-color:#eee;  border:  1px  solid
#ccc;">
                  Ваш браузер не поддерживает Canvas
         </canvas>

     <script>
          var canvas = document.getElementById("Canvas"),
              context = canvas.getContext("2d");
                 context.strokeStyle = "red";
                 context.setLineDash([15, 5]);
                 context.strokeRect(40, 40, 100, 100);
                 context.strokeStyle = "blue";
                 context.setLineDash([2, 5, 6]);
                 context.strokeRect(180, 40, 100, 100);
                 context.strokeStyle = "green";
                 context.setLineDash([2]);
                 context.strokeRect(320, 40, 100, 100);
     </script>
     </body>
     </html>
```

Результат работы в браузере (рисунок 1.6):

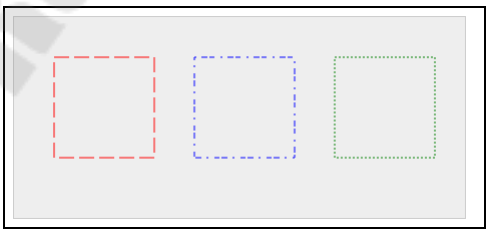


Рисунок 1.6 – Демонстрация работы метода *setLineDash()*

Свойство *lineJoin* отвечает за тип соединения линий в фигуре. Оно может принимать следующие значения:

– *miter:* прямые соединения, которые образуют прямые углы. Это значение по умолчанию;

– *round:* закругленные соединения;

– *bevel:* конические соединения.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="450" height="200"
                style="background-color:#eee;
                 border: 1px solid #ccc;">
            Ваш браузер не поддерживает Canvas
        </canvas>
        <script>
            var canvas =
                document.getElementById("Canvas"),
                context = canvas.getContext("2d");
            context.strokeStyle = "red";
            context.fillStyle = "blue";
            context.lineWidth = 15;
            context.lineJoin = "miter";
            context.strokeRect(40, 40, 100, 100);
            context.lineJoin = "bevel";
            context.strokeRect(180, 40, 100, 100);
            context.lineJoin = "round";
            context.strokeRect(310, 40, 100, 100);
        </script>
    </body>
</html>
```

Результат работы в браузере (рисунок 1.7):

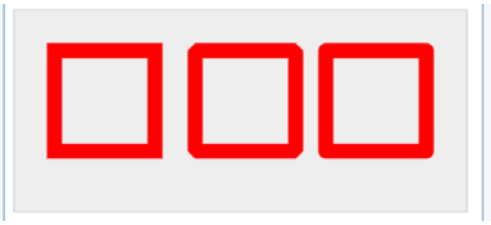


Рисунок 1.7 – Демонстрация работы свойства *lineJoin*

Свойство *globalAlpha* задает прозрачность отрисовки. Оно может принимать в качестве значения число от 0 (полностью прозрачный) до 1.0 (не прозрачный):

```
var canvas = document.getElementById("Canvas"),
            context = canvas.getContext("2d");
            context.fillStyle = "blue";
            context.fillRect(50, 50, 100, 100);
            context.globalAlpha = 0.5;
            context.fillStyle = "red";
            context.fillRect(100, 100, 100, 100);
```

Здесь на *canvas* выводятся два прямоугольника: синий и красный.

Но до вывода красного прямоугольника установлена полупрозрачность отрисовки, поэтому сквозь красный прямоугольник мы сможем увидеть и синий (рисунок 1.8):

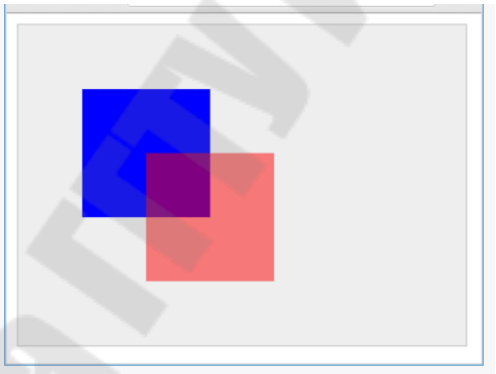


Рисунок 1.8 – Демонстрация работы свойства *globalAlpha*

1.5 Фоновые изображения

Вместо конкретного цвета для заливки фигур, например, прямоугольников, можно использовать изображения. Для этого у кконтекста canvas имеется функция *createPattern*(), которая принимает два параметра: изображение, которое будет использоваться в качестве фона, и принцип повторения изображения. Последний параметр играет роль в том случае, если размер изображения у нас меньше, чем размер фигуры на *canvas*. Этот параметр может принимать следующие значения:

– *repeat*: изображение повторяется для заполнения всего пространства фигуры;
– *repeat-x*: изображение повторяется только по горизонтали;
– *repeat-y*: изображение повторяется только по вертикали;
– *no-repeat*: изображение не повторяется.

Нарисуем прямоугольник и выведем в нем изображение:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="300" height="200"
                style="background-color:#eee;
border:1px solid #ccc;">
            Ваш браузер не поддерживает Canvas
        </canvas>
        <script>
            var canvas =
document.getElementById("Canvas"),
                context = canvas.getContext("2d");

            var img = new Image();
            img.src = "list.jpg";
          img.onload = function() {
          var  pattern  =  context.createPattern(img,
"repeat");
                context.fillStyle = pattern;
                context.fillRect(10, 10, 150, 150);
                context.strokeRect(10, 10, 150, 150);
            };
        </script>
    </body>
</html>
```

Чтобы использовать изображение, необходимо создать элемент Image и установить источник изображения – локальный файл или ресурс в сети:

```
var img = new Image();
   img.src = "list.jpg";
```

Однако загрузка изображения может занять некоторое время, особенно если файл изображения берется из сети интернет. Поэтому, чтобы быть уверенными, что изображение уже загрузилось, все действия по его использованию производятся в методе *img.onload*, который вызывается при завершении загрузки изображения:

```
            img.onload = function() {
            var    pattern    =    context.createPattern(img,
"repeat");
                    context.fillStyle = pattern;
                    context.fillRect(10, 10, 150, 150);
                    context.strokeRect(10, 10, 150, 150);
                };
```

Метод *createPattern()* возвращает объект, который устанавливается в качестве стиля заполнения фигуры:

```
context.fillStyle = pattern;
```

Отрисовка прямоугольника остается той же (рисунок 1.9).

Рисунок 1.9 – Демострация работы метода *createPattern()*

1.6 Создание градиента

Элемент *Canvas* позволяет использовать градиент в качестве фона.

Для этого применяется объект *CanvasGradient*, который можно создать либо с помощью метода *createLinearGradient()* (линейный градиент), либо с помощью метода *createRadialGradient()* (радиальный градиент).

Линейный градиент. Метод *createLinearGradient(x0, y0, x1, y1),* где x0 и y0 – это начальные координаты градиента относительно верхнего левого угла canvas, а x1 и y1 – координаты конечной точки градиента.

```
var gradient =
          context.createLinearGradient(50,30,150,150);
```

Также для создания градиента необходимо задать опорные точки, которые определяют цвет. Для этого у объекта *CanvasGradient* применяется метод *addColorStop(offset, color)*, где *offset* – это смещение точки градиента, а *color* – ее цвет. Например:

```
gradient.addColorStop(0, "blue");
```

Смещение представляет значение в диапазоне от 0 до 1. Смещение 0 представляет начало градиента, а 1 – его конец.

Цвет задается либо в виде строки, либо в виде шестнадцатиричного значения, либо в виде значения *rgb/rgba*.

Применим градиент:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
 <body>
    <canvas id="Canvas" width="300" height="200"
            style="background-color:#eee;
            border:1px solid #ccc;">
            Ваш браузер не поддерживает Canvas
    </canvas>

  <script>
     var canvas = document.getElementById("Canvas"),
```

```
            context = canvas.getContext("2d"),


gradient=context.createLinearGradient(50,30,150,150);
          gradient.addColorStop(0, "blue");
          gradient.addColorStop(1, "white");
          context.fillStyle = gradient;
          context.fillRect(50, 30, 150, 150);
          context.strokeRect(50, 30, 150, 150);
        </script>
      </body>
    </html>
```

В данном случае получаем диагональный линейный градиент (рисунок 1.10).

Рисунок 1.10 – Создание линейного градиента

Управляя координатами, можно добиться горизонтального или вертикального градиента.

Совпадение *x*-координат начальной и конечной точек создает горизонтальный градиент:

```
gradient = context.createLinearGradient(50,30,50,150);
```

А совпадение *y*-координат начальной и конечной точек создает горизонтальный градиент (рисунок 1.11):

```
gradient = context.createLinearGradient(50,30,150,30);
```

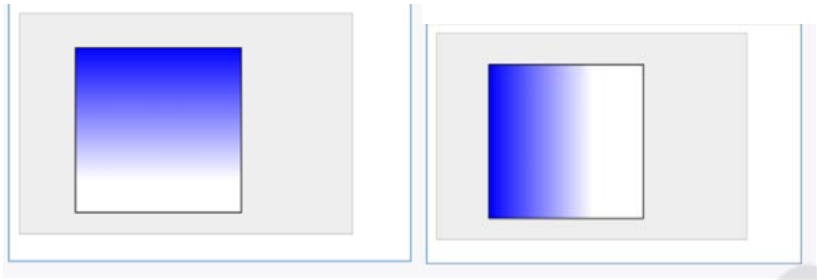


Рисунок 1.11 – Горизонтальный и вертикальный градиенты

Радиальный градиент создается с помощью метода *createRadialGradient(x0, y0, r0, x1, y1, r1)*, который принимает следующие параметры:

– *x0* и *y0*: координаты центра первой окружности;
– *r0*: радиус первой окружности;
– *x1* и *y1*: координаты центра второй окружности;
– *r1*: радиус второй окружности.

Например:

```
var gradient =
context.createRadialGradient(120,100,100,120,100,30);
```

Для радиального градиента необходимо задать опорные цветовые точки с помощью метода *addColorStop()*(рисунок 1.12).

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="300" height="200"
style="background-color:#eee; border:1px solid #ccc;">
            Ваш браузер не поддерживает Canvas
        </canvas>

        <script>
     var canvas = document.getElementById("Canvas"),
                context = canvas.getContext("2d"),
```

```
                gradient =
context.createRadialGradient(120,100,100,120,100, 30);
                gradient.addColorStop(0, "blue");
                gradient.addColorStop(1, "white");
                context.fillStyle = gradient;
                context.fillRect(50, 30, 150, 150);
                context.strokeRect(50, 30, 150, 150);
            </script>
        </body>
    </html>
```

Рисунок 1.12 – Радиальный градиент

1.7 Рисование текста

Наряду с геометрическими фигурами и изображениями *canvas* позволяет выводить текст. Для этого необходимо установить у контекста *canvas* свойство *font*:

```
var canvas = document.getElementById("Canvas"),
context = canvas.getContext("2d");
context.font = "22px Verdana";
```

Свойство *font* в качестве значения принимает определение шрифта. В данном случае это шрифт *Verdana* высотой 22 пикселя. В качестве шрифтов используются стандартные шрифты (рисунок 1.13).

Далее выводится некоторый текст с помощью метода *fillText()*:

```
<!DOCTYPE html>
<html>
    <head>
      <meta charset="utf-8">
      <title>Canvas </title>
```

```
        </head>
        <body>
         <canvas   id="Canvas"   width="300"   height="200"
style="background-color:#eee; border:1px solid #ccc;">
                  Ваш браузер не поддерживает Canvas
          </canvas>
        <script>
          var canvas = document.getElementById("Canvas"),
              context = canvas.getContext("2d");
              context.font = "22px Verdana";
              context.fillText("Hello HTML5!", 20, 50);
             </script>
          </body>
       </html>
```

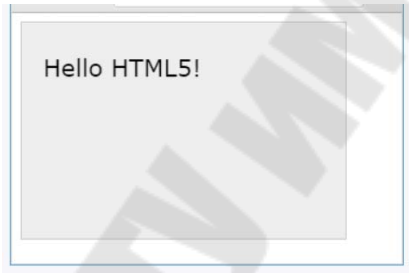


Рисунок 1.13 – Демонстрация работы свойства *font*

Метод *fillText(text, x, y)* принимает три параметра: выводимый текст, *x* и *y* координаты точки, с которой выводится текст.

Для вывода текста можно также применять метод *strokeText()*, который создает границу для выводимых символов:

```
var canvas = document.getElementById("Canvas"),
          context = canvas.getContext("2d");
          context.font = "30px Verdana";
          context.strokeStyle = "red";
          context.strokeText("Hello HTML5!", 20, 50);
```

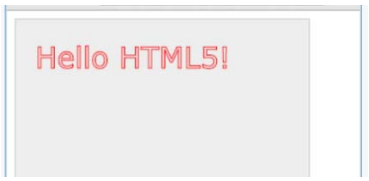


Рисунок 1.13 – Демонстрация работы метода *fillText()*

Свойство *textAlign* позволяет выровнить текст относительно одной из сторон. Это свойство может принимать следующие значения:

– *left*: текст начинается с указанной позиции;

– *right*: текст завершается до указанной позиции;

– *center*: текст располагается по центру относительно указанной позиции;

– *start*: значение по умолчанию, текст начинается с указанной позиции;

– *end*: текст завершается до указанной позиции.

```
var canvas = document.getElementById("Canvas"),
           context = canvas.getContext("2d");
           context.font = "22px Verdana";
           context.textAlign = "right";
           context.fillText("Right Text", 120, 30);
           context.textAlign = "left";
           context.fillText("Left Text", 120, 60);
           context.textAlign = "center";
           context.fillText("Center Text", 120, 90);
           context.textAlign = "start";
           context.fillText("Start Text", 120, 120);
           context.textAlign = "end";
           context.fillText("End Text", 120, 150);
```

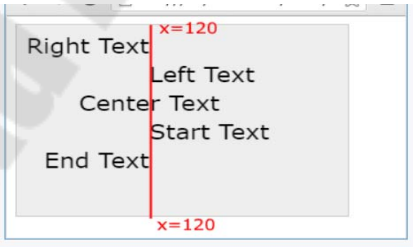


Рисунок 1.14 – Выравнивание текста

Свойство *lineWidth* задает ширину линии текста (рисунок 1.15):

```
var canvas = document.getElementById("Canvas"),
           context = canvas.getContext("2d");
           context.font = "30px Verdana";
           context.strokeStyle = "red";
           context.lineWidth = 2;
           context.strokeText("Hello HTML5!", 20, 50);
```

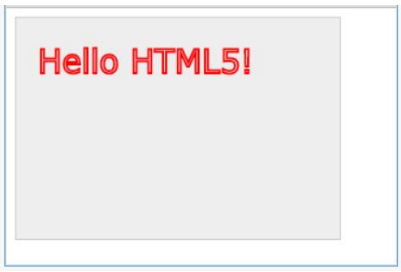


Рисунок 1.15 – Демонстрация работы свойства *lineWidth*

Свойство *textBaseline* задает выравнивание текста по базовой линии. Оно может принимать следующие значения (рисунок 1.16):

– *top;*
– *middle;*
– *bottom;*
– *alphabetic;*
– *hanging;*
– *ideographic.*

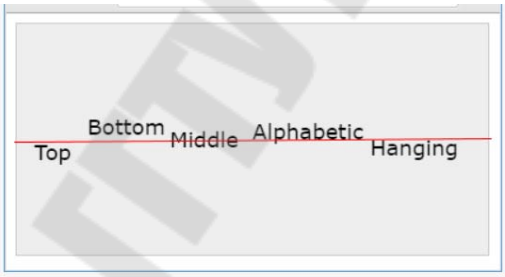


Рисунок 1.16 – Демонстрация работы свойства *textBaseline*

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
            context.font = "18px Verdana";
            context.textBaseline = "top";
            context.fillText("Top", 15, 100);
            context.textBaseline = "bottom";
            context.fillText("Bottom", 60, 100);
            context.textBaseline = "middle";
            context.fillText("Middle", 130, 100);
            context.textBaseline = "alphabetic";
            context.fillText("Alphabetic", 200, 100);
            context.textBaseline = "hanging";
            context.fillText("Hanging", 300, 100);
```

С помощью метода *measureText()* можно определить ширину текста на canvase:

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
    context.font = "18px Verdana";
   var text = context.measureText("Hello HTML5");
        alert(text.width);
```

1.8 Рисование фигур

Кроме прямоугольников *canvas* позволяет рисовать и более сложные фигуры. Для оформления сложных фигур используется концепция геометрических путей, которые представляют набор линий, окружностей, прямоугольников и других более мелких деталей, необходимых для построения сложной фигуры.

Для создания нового пути надо вызвать метод ***beginPath()***, а после завершения пути вызывается метод ***closePath()***:

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
            context.beginPath();
            // здесь инструкции по созданию фигур
            context.closePath();
```

Между вызовами методов *beginPath()* и *closePath()* находятся методы, непосредственно создающие различные участки пути. Это методы *moveTo()* и *lineTo()*.

Метод ***moveTo(x, y)*** – перемещает нас на точку с координатами *x* и *y*.

Метод ***lineTo(x, y)*** – рисует линию от текущей позиции до точки с координатами *x* и *y*.

Нарисуем ряд линий:

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
            context.beginPath();
            context.moveTo(30, 20);
            context.lineTo(100, 80);
            context.lineTo(150, 30);
            context.closePath();
```

Здесь устанавливаем начало пути в точку (30, 20), затем от нее рисуем линию до точки (100, 80) и далее рисуем еще одну линию до точки (150, 30).

Хотя нарисовали несколько линий, пока их не увидим, так как их надо отобразить на экране (рисунок 1.17). Для отображения пути надо использовать метод ***stroke()***:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
    <canvas id="Canvas" width="300" height="200"
style="background-color:#eee; border:1px solid #ccc;">
            Ваш браузер не поддерживает Canvas
        </canvas>
    <script>
     var canvas = document.getElementById("Canvas"),
                context = canvas.getContext("2d");
        context.beginPath();
        context.moveTo(30, 20);
        context.lineTo(150, 140);
        context.lineTo(250, 30);
        context.closePath();
        context.strokeStyle = "red";
        context.stroke();
        </script>
    </body>
</html>
```

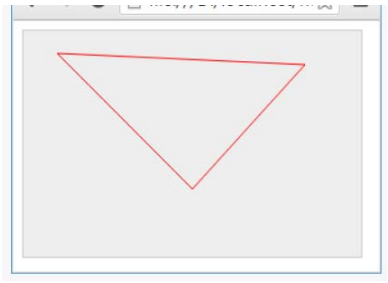


Рисунок 1.17 – Отрисовка треугольника

Не смотря на то, что нарисовали две линии, но по факту видим три линии, которые оформляют треугольник. Дело в том, что вызов метода ***context.closePath()*** завершает путь, соединяя последнюю точку с первой. И в результате образуется замкнутый контур (рисунок 1.18).

Если не надо замыкание пути, то можно удалить вызов метода *context.closePath()*:

Рисунок 1.18 – Рисование треугольника без замыкания путей

Метод *rect()* создает прямоугольник (рисунок 1.19). Он имеет следующее определение:

```
rect(x, y, width, height),
```

где *x* и *y* – это координаты верхнего левого угла прямоугольника относительно canvas, а width и height – соответственно ширина и высота прямоугольника. Нарисуем, к примеру, следующий прямоугольник:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
    <canvas id="Canvas" width="300" height="200"
    style="background-color:#eee;   border:1px   solid
#ccc;">
            Ваш браузер не поддерживает Canvas
    </canvas>
        <script>
```

```
            var                    canvas                    =
document.getElementById("Canvas"),
                    context = canvas.getContext("2d");
                    context.beginPath();
                    context.rect(30, 20, 100, 90);
                    context.closePath();
                    context.strokeStyle = "red";
                    context.stroke();
            </script>
        </body>
    </html>
```

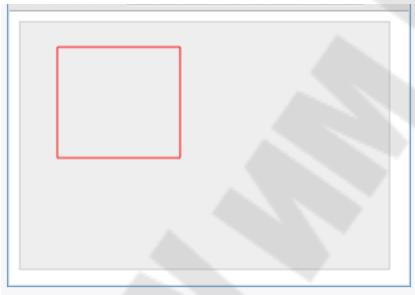


Рисунок 1.19 – Демострация работы метода *Rect()*

Стоит отметить, что такой же прямоугольник могли бы создать из линий:

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
        context.beginPath();
        context.moveTo(30, 20);
        context.lineTo(130, 20);
        context.lineTo(130, 110);
        context.lineTo(30, 110);
        context.closePath();
        context.strokeStyle = "red";
        context.stroke();
```

Метод *fill()* заполняет цветом все внутреннее пространство нарисованного пути:

```
var canvas = document.getElementById("Canvas"),
```

```
context = canvas.getContext("2d");
        context.beginPath();
        context.rect(30, 20, 100, 90);
        context.closePath();
        context.strokeStyle = "red";
        context.fillStyle = "blue";
        context.fill();
        context.stroke();
```

С помощью свойства *fillStyle* опять же можно задать цвет заполнения фигуры. В данном случае это синий цвет (рисунок 1.20).

Рисунок 1.20 – Демострация работы свойства *fillStyle*

Метод *clip()* позволяет вырезать из *canvas* определенную область, а все, что вне этой области, будет игнорироваться при последующей отрисовке (рисунок 1.21).

Для понимания этого метода сначала нарисуем два прямоугольника:

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
        // рисуем первый красный прямоугольник
        context.beginPath();
        context.moveTo(30, 20);
        context.lineTo(130, 20);
        context.lineTo(130, 110);
        context.lineTo(30, 110);
        context.closePath();
        context.strokeStyle = "red";
        context.stroke();
```

```
        // рисуем второй зеленый прямоугольник
        context.beginPath();
        context.rect(10, 50, 180, 70);
        context.closePath();
        context.strokeStyle = "green";
        context.stroke();
```

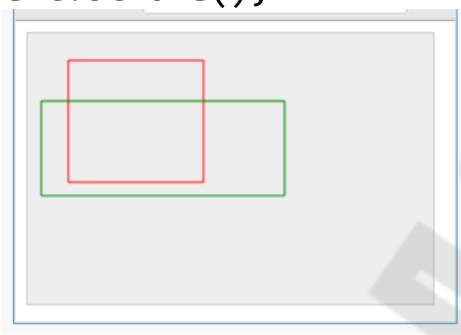


Рисунок 1.21 – Демострация работы метода *Clip()*

Теперь применим метод *clip()* для ограничения области рисования только первым прямоугольником (рисунок 1.22):

```
    var canvas = document.getElementById("Canvas"),
      context = canvas.getContext("2d");
            // рисуем первый красный прямоугольник
            context.beginPath();
            context.moveTo(30, 20);
            context.lineTo(130, 20);
            context.lineTo(130, 110);
            context.lineTo(30, 110);
            context.closePath();
            context.strokeStyle = "red";
            context.stroke();
          context.clip();
    //вырезать из canvas определенную область, а все, что вне
этой //области, будет игнорироваться при последующей отрисовке.
            context.beginPath();//рисуем          зеленый
прямоугольник
            context.rect(10, 50, 180, 70);
            context.closePath();
            context.strokeStyle = "green";
            context.stroke();
```

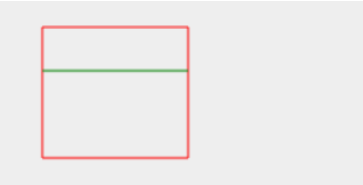


Рисунок 1.22 – Демострация работы метода *Clip()* при ограничении области рисования одним прямоугольником

Поскольку вызов метода *clip()* идет после первого прямоугольника, то из второго прямоугольника будет нарисована только та часть, которая попадает в первый прямоугольник (рисунок 1.23).

Метод *arc()* добавляет к пути участок окружности или арку. Он имеет следующее определение:

```
arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

Здесь используются следующие параметры:

x и *y*	*x* и *y* – координаты, в которых начинается арка
Radius	радиус окружности, по которой создается арка
startAngle *endAngle*	начальный и конечный угол, которые усекают окржность до арки.
anticlockwise	направление движения по окружности при отсечении ее части, ограниченной начальным и конечным углом. При значении *true* направление против часовой стрелки, а при значении *false* – по часовой стрелке.

В качестве единицы измерения для углов применяются радианы.

Например, полная окружность – это 2π радиан. Если, к примеру, необходимо нарисовать полный круг, то для параметра *endAngle* можно указать значение 2π. В *JavaScript* эту величину можно получить с помощью выражения *Math.PI * 2*.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
     <canvas id="Canvas" width="420" height="200"
```

```
        style="background-color:#eee;   border:1px   solid
#ccc;">
                Ваш браузер не поддерживает Canvas
          </canvas>
            <script>
         var canvas = document.getElementById("Canvas"),
             context = canvas.getContext("2d");

           context.strokeStyle = "red";
           context.beginPath();
           context.moveTo(20, 90);
           context.arc(20, 90, 50, 0, Math.PI / 2, false);
           context.closePath();
           context.stroke();

          context.beginPath();
           context.moveTo(130, 90);
           context.arc(130, 90, 50, 0, Math.PI, false);
           context.closePath();
           context.stroke();

    context.beginPath();
         context.moveTo(240, 90);
         context.arc(240,  90,  50,  0,  Math.PI  *  3  /  2,
false);
         context.closePath();
         context.stroke();

          context.beginPath();
          context.arc(350, 90, 50, 0, Math.PI * 2, false);
          context.closePath();
          context.stroke();

            </script>
        </body>
    </html>
```

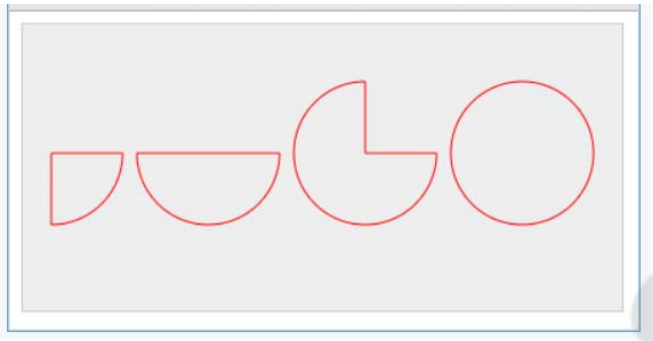


Рисунок 1.23 – Демонстрация работы метода *Arc()*

Последний параметр *anticlockwise* играет важную роль, так как определяет движение по окружности, и в случае изменения *true* на *false* и наоборот, можно получить совершенно разные фигуры (рисунок 1.24):

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
    context.strokeStyle = "red";
    context.beginPath();
    context.moveTo(80, 90);
    context.arc(80, 90, 50, 0, Math.PI/2, false);
    context.closePath();
    context.stroke();
```

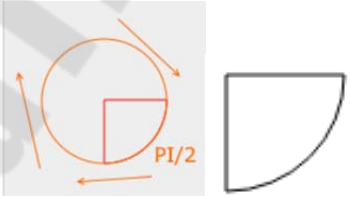


Рисунок 1.24 – Вариации с параметром *anticlockwise*

Метод *arcTo()* также рисует дугу (рисунок 1.25). Он имеет следующее определение:

```
arcTo(x1, y1, x2, y2, radius)
```

где *x1* и *y1* – координаты первой контрольной точки, *x2* и *y2* – координаты второй контрольной точки, а *radius* – радиус дуги.

```
context.beginPath();
```

```
context.moveTo(0, 150);
context.arcTo(0, 0, 150, 0, 140)
context.closePath();
context.stroke();
```

Рисунок 1.25 – Демонстрация метода *arcTo*()

Здесь перемещаемся вначале на точку (0, 150), и от этой точки до первой контрольной точки (0, 0) будет проходить первая касательная. Далее от первой контрольной точки (0, 0) до второй (150, 0) будет проходить вторая касательная. Эти две касательные оформляют дугу, а 140 служит радиусом окружности, на которой усекается дуга.

Метод *quadraticCurveTo()* создает квадратичную кривую (рисунок 1.26). Он имеет следующее определение:

```
quadraticCurveTo(x1, y1, x2, y2),
```

где *x1* и *y1* – координаты первой опорной точки, а *x2* и *y2* – координаты второй опорной точки.

```
var canvas = document.getElementById("Canvas"),
context = canvas.getContext("2d");
context.strokeStyle = "red";
context.beginPath();
context.moveTo(20, 90);
context.quadraticCurveTo(130, 0, 280, 90)
context.closePath();
context.stroke();
```

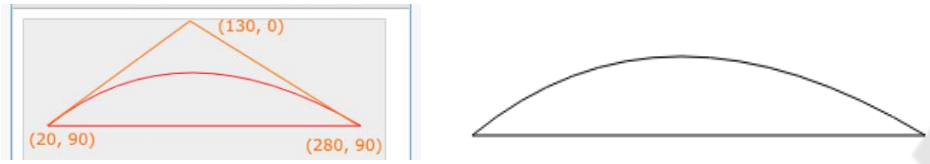


Рисунок 1.26 – Демонстрация работы метода *quadraticCurveTo*()

Метод *bezierCurveTo()* рисует кривую **Безье** (рисунок 1.27). Он имеет следующее определение:

```
bezierCurveTo(x1, y1, x2, y2, x3, y3)
```

где *x1* и *y1* – координаты первой опорной точки,
x2 и *y2* – координаты второй опорной точки,
x3 и *y3* – координаты третьей опорной точки.

```
var canvas = document.getElementById("Canvas"),
        context = canvas.getContext("2d");
        context.strokeStyle = "red";
        context.beginPath();
        context.moveTo(30, 100);
        context.bezierCurveTo(110,0,190,200,270,100);
        context.closePath();
        context.stroke();
```

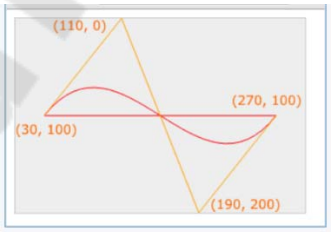


Рисунок 1.27 – Демонстрация работы метода *bezierCurveTo()*

Комплексные фигуры. Объединим несколько фигур вместе и нарисуем более сложную двухмерную сцену:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
```

```
        </head>
        <body>
            <canvas id="Canvas" width="350" height="250"
      style="background-color:#eee; border: 1px solid #ccc;
margin:10px;">
                Ваш браузер не поддерживает Canvas
            </canvas>
        <script>
         var canvas = document.getElementById("Canvas");
              if (canvas.getContext) {
                var ctx = canvas.getContext("2d");
                ctx.beginPath();
                ctx.fill();
                ctx.fillStyle = "yellow";
                ctx.beginPath();
                ctx.arc(160, 130, 100, 0, 2 * Math.PI);
                ctx.fill();

                ctx.beginPath();
                ctx.moveTo(100, 160);
                ctx.quadraticCurveTo(160, 250, 220, 160);
                ctx.closePath();
                ctx.fillStyle = "red";
                 ctx.fill();
                 ctx.lineWidth = 2;
                 ctx.strokeStyle = "black";
                 ctx.stroke();

                    ctx.fillStyle = "#FFFFFF";
                    ctx.fillRect(140, 160, 15, 15);
                    ctx.fillRect(170, 160, 15, 15);
                    //
                    ctx.beginPath();
                    ctx.arc(130, 90, 20, 0, 2 * Math.PI);
                    ctx.fillStyle = "#333333";
                    ctx.fill();
                    ctx.closePath();

                    ctx.beginPath();
                    ctx.arc(190, 90, 20, 0, 2 * Math.PI);
                    ctx.fillStyle = "#333333";
```

```
                ctx.fill();
                ctx.closePath();
            }
        </script>
    </body>
</html>
```

Результат работы в браузере (рисунок 1.28):

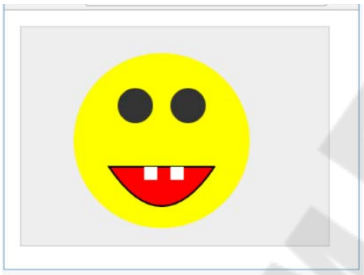


Рисунок 1.28 – Смайлик в canvas, реализованный посредством комплексных фигур

1.9 Изображения на *Canvas*

Для вывода изображения на *canvas* (рисунок 1.29) применяется метод *drawImage()*:

```
context.drawImage(image, x, y).
```

Здесь параметр *image* передает выводимое изображение, *x* и *y* – координаты верхнего левого угла изображения. Например:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="600" height="400"
style="background-color:#eee; border:1px solid #ccc;">
            Ваш браузер не поддерживает Canvas
        </canvas>
```

```
<script>
    var canvas = document.getElementById("Canvas"),
               context = canvas.getContext("2d");

    var img = new Image();
           img.src = "4.jpg";
           img.onload = function () {
               context.drawImage(img, 0, 0);
           };
       </script>
   </body>
</html>
```

Рисунок 1.29 – Отображение изображения с помощью метода *drawImage()*

При выводе изображения надо быть уверенными, что изображение уже загружено браузером и готово к использованию, поэтому метод отрисовки изображения помещается в обработчик загрузки изображения *img.onload*.

Другая версия метода позволяет дополнительно задать ширину и высоту выводимого изображения:

```
var canvas = document.getElementById("Canvas"),
  context = canvas.getContext("2d");
      var img = new Image();
      img.src = "dubi.png";
      img.onload = function () {
          context.drawImage(img, 20, 40, 110, 90);
          context.drawImage(img, 160, 40, 110, 90);
      }
```

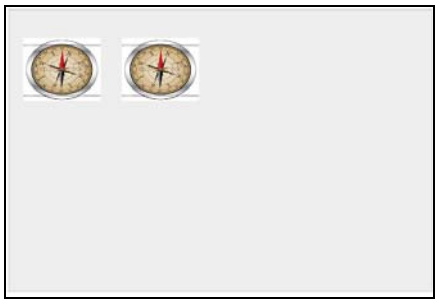


Рисунок 1.30 – Изменение пропорций исходного изображения

Метод *drawImage()* имеет еще и третью форму (рисунок 1.31):

drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight),

где *sx* и *sy* – координаты на изображении, с которого начиется обрезка изображения;

sWidth u sHeight – ширину и высоту выреза относительно координат *sx* и *sy;*

dx u dy – координаты отрисовки обрезанного изображения на *canvas*;

dWidth и *dHeight* – ширина и высота изображения на *canvas.*

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
    <canvas id="Canvas" width="600" height="500"
    style="background-color:#eee;   border:1px   solid
#ccc;">
            Ваш браузер не поддерживает Canvas
    </canvas>
  <script>
    var canvas = document.getElementById("Canvas"),
        context = canvas.getContext("2d");

          var img = new Image();
          img.src = "4.jpg";
```

```
            img.onload = function () {

context.drawImage(img, 24,8, 400,400, 20,30, 300,250);
            };
  </script>
    </body>
</html>
```

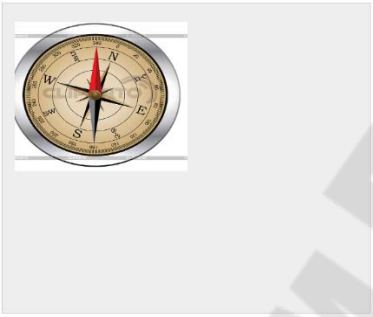


Рисунок 1.31 – Третий вариант задания параметров у *drawImage()*

Одной из замечательных функциональностей элемента *canvas* является возможность **захвата изображения с другого элемента**, например, элемента *video* или другого элемента canvas (рисунок 1.32). Например:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>

<video id="myVideo" src="cats.mp4" width="300"
                    height="200" controls ></video>

<canvas id="Canvas" width="300" height="200"
    style="background-color:#eee;   border:1px   solid
#ccc;">
            Ваш браузер не поддерживает Canvas
     </canvas>
        <div>
```

```
            <button id="snap">  F O T O  </button>
        </div>
        <script>
    var canvas = document.getElementById("Canvas"),
        context = canvas.getContext("2d");
    var video = document.getElementById("myVideo");

document.getElementById("snap").onclick = function (e)

   {
          context.drawImage(video, 0, 0, 300, 200);
    }
        </script>
    </body>
</html>
```

Рисунок 1.32 – Вставка видео в *canvas*

По нажатии на кнопку, *canvas* будет получать текущий кадр воспроизводимого видео и фиксировать его в качестве изображения. При этом в метод *drawImage* в качестве первого параметра передается сам элемент, используемый как источник изображения.

1.10 Добавление теней

Элемент canvas поддерживает добавление теней к нарисованным объектам (рисунок 1.33). Для создания теней применяются следующие свойства:

– *shadowOffsetX*: горизонтальное смещение в пикселях справа (или слева при отрицательном значении);

– *shadowOffsetY*: вертикальное смещение в пикселях снизу (или сверху при отрицательном значении);
– *shadowBlur*: число пикселей ля установки размыти тени;
– *shadowColor*: цвет тени.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
  <body>
  <canvas id="Canvas" width="300" height="200"
    style="background-color:#eee;   border:1px   solid
#ccc;">
                Ваш браузер не поддерживает Canvas
   </canvas>
   <script>
     var canvas = document.getElementById("Canvas"),
            context = canvas.getContext("2d");

            context.font = "16px Verdana";
            context.fillStyle = "#222";
            context.shadowOffsetX = 3;
            context.shadowOffsetY = 3;
            context.shadowBlur = 3;
            context.shadowColor = "#AAA";
            context.fillText("Тени в HTML5", 80, 30);

            var img = new Image();
            img.src = "dubi.png";

              img.onload = function () {
             .shadowOffsetX = 8;
              context.shadowOffsetY = 8;
              context.shadowBlur = 5;
              context.shadowColor = "#333";
              context.drawImage(img, 80, 70, 128, 96);
            };
    </script>
   </body>
</html>
```

Рисунок 1.33 – Эффект тени к изобраению

1.11 Редактирование пикселей

HTML5 предоставляет встроенную функциональность для редактирования изображения и установки значения конкретных пикселей на *canvas*.

В частности, можно изменить цветовые значения пикселя, его прозрачность. Для этого предназначены такие методы, как:

- *getImageData(),*
- *putImageData(),*
- *createImageData().*

Метод *getImageData()* позволяет извлечь из *canvas* какую-либо часть изображеня.

Он имеет следующее определение:

```
getImageData(sx, sy, sw, sh),
```

где *sx* и *sy* – координаты верхнего левого угла области, из которой извлекаются данные на *canvas*,

sw и *sh* – соотвественно ширина и высота этой области.

Данные из определенной этими параметрами области извлекаются в виде объекта *ImageData*, который потом используется для манипуляции пикселями.

Пример использования:

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");

var img = new Image();
```

```
    img.src = "dubi.png";
    img.onload = function() {

        context.drawImage(img, 0, 0);
        var  imageData  =  context.getImageData(0,0,  100,
100);
    };
```

Все данные об изображении в объекте *ImageData* хранятся в массиве *data*.

Каждый пиксель на canvas характеризуется четырьмя компонентами в формате *RGBA*: красной, зеленой, синей компонентой, которые устанавливают цвет, и альфа-компонентой, которая устанавливает прозрачность.

Каждая компонента принимает значени от 0 до 255.

Чтобы получить значения цвета для самого первого пикселя в *ImageData*, надо последовательно получить четыре значения из массива *data*:

```
var canvas = document.getElementById("Canvas"),
                context = canvas.getContext("2d");
  var img = new Image();
  img.src = "dubi.png";
  img.onload = function() {

  context.drawImage(img, 0, 0);
  var imageData = context.getImageData(0,0, 100, 100);
  var red = imageData.data[0];// компонента красного цвета
  var green = imageData.data[1];// компонента зелен.цвета
  var blue = imageData.data[2]; // компонента синего цвета
  var alpha = imageData.data[3];// компонента прозрачности
    };
```

В данном случае получаем информацию о самом первом пикселе, который находится в самом верхнем левом углу, то есть имеет координаты x=0 и y=0.

Чтобы получить информацию о втором пикселе, который имеет координаты x=1 и y=0, необходимо получить следующие четыре значения из массива *data*:

imageData.data[4]; // компонента красного цвета

imageData.data[5]; // компонента зеленого цвета
imageData.data[6]; // компонента синего цвета
imageData.data[7]; // компонента прозрачности
И так далее можем получить информацию обо всех пикселях.

Метод *putImageData()* устанавливает на canvas новые данные. Этот метод имеет следеющее определение:

putImageData(imageData, dx, dy)

Параметры *dx* и *dy* указывают координаты верхнего левого угла условного прямоугольника *imageData*, в который размещается на *canvas*.

Используем методы *getImageData()* и *putImageData()* для преобразования изображения:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
    <canvas id="Canvas" width="530" height="200"
    style="background-color:#eee;   border:1px   solid
#ccc;">
                Ваш браузер не поддерживает Canvas
    </canvas>
    <script>
     var canvas = document.getElementById("Canvas"),
         context = canvas.getContext("2d");

            var img = new Image();
            img.src = "list.png";
            img.onload = function () {
          context.drawImage(img, 0, 0);
      var imageData = context.getImageData(0, 0,
                                  img.width, img.height);
      var red, green, blue, greyscale;
```

```
        for (var i = 0; i < imageData.data.length; i += 4)
{
       red = imageData.data[i];
                          // получаем компоненту красного
                  цвета
       green = imageData.data[i + 1];
                          // получаем компоненту зеленого
цвета
       blue = imageData.data[i + 2];
                        // получаем компоненту синего цвета
       grayscale = red * 0.3 + green * 0.59 + blue * 0.11;
                        // получаем серый фон
                          imageData.data[i] = grayscale;
                             // установка серого цвета
                          imageData.data[i + 1] = grayscale;
                          imageData.data[i + 2] = grayscale;
                      }
            context.putImageData(imageData, img.width + 10,
0);
                  };
        </script>
       </body>
     </html>
```

Ключевым участком кода здесь является цикл, в котором и происходит преобразование изображения в серое:

```
     var red, green, blue, greyscale;
       for (var i = 0; i < imageData.data.length; i += 4) {
            red = imageData.data[i];
                    // получаем компоненту красного цвета
            green = imageData.data[i + 1];
                    // получаем компоненту зеленого цвета
            blue = imageData.data[i + 2];
                    // получаем компоненту синего цвета
        grayscale = red * 0.3 + green * 0.59 + blue * 0.11;
                                 // получаем серый фон
                    imageData.data[i] = grayscale;
                              // установка серого цвета
                    imageData.data[i + 1] = grayscale;
                    imageData.data[i + 2] = grayscale;
                }
```

Здесь перемещаемся по всему массиву *imageData.data*, обрабатывая за раз четыре элемента, которые и представляют отдельный пиксель. При этом учитываются только три элемента, поскольку компонента прозрачности в данном случае нас не интересует.

Сначала получаем компоненты *RGB*. Затем, применяя математическую формулу, преобразуем значения *RGB* в серый цвет. И в конце серый цвет устанавливается для элементов пикселя.

Примечание. Если попробуем просто кинуть файл веб-страницы с выше описанным кодом в браузер *Google Chrome* или попытаемся открыть файл по двойному клику, то *Google Chrome* не отобразит преобразованное серое изображение в связи с политикой браузера. Хотя в других браузерах, как *Firefox* или *Microsoft Edge* все может быть нормально. Дело в том, что в *Google Chrome* не позволяет манипулировать изображением сайта из одного домена пользователю из другого домена.

По сути, когда загружаем файл по протоколу *file://* (просто кинув файл в браузер или по двойному клику), браузер рассматривает пользователя и открытую веб-страницу как разные домены.

Если веб-страница будет расположена на веб-сервере и загружена по протоколу *http*, как это обычно бывает, то, конечно, проблемы не возникнет, так как пользователь и сайт будут работать в рамках одного домена.

Но для тестирования опять же либо придется использовать веб-сервер, либо изменить соответствующим образом настройки браузера *Google Chrome*.

Метод *createImageData()* создает новый объект *ImageData*, который затем может использоваться на canvas.

Метод *createImageData()* имеет две формы:

createImageData(width, height);
createImageData(imagedata).

Первая форма принимает параметры *width* и *height*, которые устанавливают соотвественно ширину и высоту создаваемого объекта *ImageData.*

Вторая форма принимает в качестве параметра другой объект *ImageData*, по которому будет создан новый объект *ImageData*.

Пример использования:

```
var canvas = document.getElementById("Canvas"),
  context = canvas.getContext("2d");

var img = new Image();
img.src = "list.png";
img.onload = function () {

context.drawImage(img, 0, 0);
var    imageData=context.getImageData(0,0,img.width,
img.height);
var                  newImageData                  =
context.createImageData(imageData);
for (var i = 0; i < newImageData.data.length; i++) {
   newImageData.data[i] = imageData.data[i];
                    // если это альфа-компонента
        if ((i + 1) % 4 === 0) {
           newImageData.data[i] = 120;
             }
        }
  context.putImageData(newImageData, img.width + 10,
0);
           };
```

В данном случае создаем новый объект *newImageData*, в этот объект копируем все данные из текущего *imageData*, который представляет изображение на *canvas*. При этом при копировании значения альфа-компоненты, которая отвечает за прозрачность, устанавливаем ей значение 120, то есть делаем пиксель полупрозрачным (рисунок 1.34).

Рисунок 1.34 – Работа с методом *createImageData()*

1.12 Трансформации

Элемент *canvas* поддерживает трансформации – перемещение, вращение, масштабирование.

Перемещение осуществляется с помощью метода *translate():*

```
translate(x, y)
```

Первый параметр указывает на смещение по оси *X*, а второй параметр – по оси *Y*.

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");

            context.fillStyle = "blue";
            context.fillRect(50, 50, 100, 100);
            context.translate(100, 50);
      // смещение на 100 пикселей вправо и 50px вниз
            context.globalAlpha = 0.5;
            context.fillStyle = "red";
            context.fillRect(50, 50, 100, 100);
```

Здесь на одной позиции отрисовываются два равных прямоугольника: синий и красный. Однако к красному прямоугольнику применяется трансформация перемещения (рисунок 1.35):

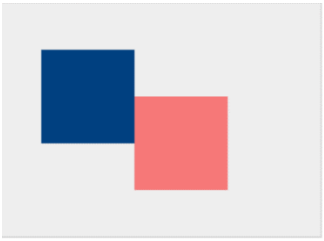


Рисунок 1.35 – Демонстрация перемещения с помощью метода *translate()*

Вращение. Для поворота фигур на canvase применяется метод ***rotate()***:

```
rotate(angle).
```

В этот метод в качестве параметра передается угол поворота в радианах относительно точки с координатами (0, 0) (рисунок 1.36).

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
            context.fillStyle = "blue";
            context.fillRect(50, 50, 100, 100);

            context.rotate(.52);
             // поворот на 0.52 радиан или 30 градусов
            context.globalAlpha = 0.5;
            context.fillStyle = "red";
            context.fillRect(50, 50, 100, 100);
```

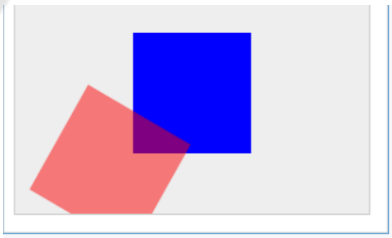


Рисунок 1.36 – Демонстрация поворота с помощью метода *rotate()*

Масштабирование. Для масштабирования фигур применяется метод ***scale()***:

```
scale(xScale, yScale)
```

Параметр *xScale* указывает на масштабирование по оси *X,* а *yScale* – по оси *Y* (рисунок 1.37).

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
            context.fillStyle = "blue";
            context.fillRect(30, 30, 100, 100);
            context.scale(1.5, 1.3);
//растяжение по ширине в 1.5 раза и сжатие по высоте в 1.3
раза
            context.globalAlpha = 0.5;
            context.fillStyle = "red";
            context.fillRect(50, 30, 100, 100);
```

Рисунок 1.37 – Демонстрация масштабирования с помощью метода *scale()*

Матрица преобразований. При необходимости будем применять последовательно несколько преобразований (рисунок 1.38):

ctx.scale(1.5, 1.3);
ctx.translate(100, 150);
ctx.rotate(0.34);

Но контекст элемента canvas также предоставляет метод *transform()*, который позволяет задать матрицу преобразования:

```
transform(a, b, c, d, e, f)
```

Все параметры этого метода последовательно представляют элементы матрицы преобразования:

– *a:* масштабирование по оси *X;*
– *b:* поворот вокруг оси *X;*

– *c:* поворот вокруг оси *Y;*
– *d:* масштабирование по оси *Y;*
– *e:* горизонтальное смещение;
– *f:* вертикальное смещение.

Например:

```
context.fillStyle = "blue";
context.fillRect(100, 50, 100, 100);

context.transform(Math.cos(Math.PI / 6),
                  Math.sin(Math.PI / 6),
                  -1 * Math.sin(Math.PI / 6),
                  Math.cos(Math.PI / 6), 0, 0);

   context.globalAlpha = 0.5;
   context.fillStyle = "red";
   context.fillRect(100, 50, 100, 100);
```

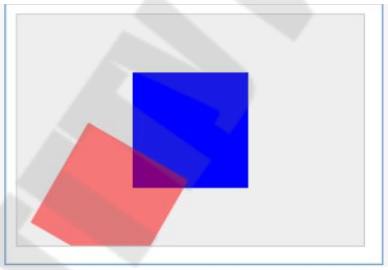


Рисунок 1.38 – Демонстрация нескольких преобразований

Замена трансформации. При последовательном применении разных трансформаций они просто последовательно применяются к фигурам.

Однако может возникнуть ситуация, когда надо применить трансформацию не вместе со другими, а вместо других, то есть заменить трансформацию.

Для этого применяется метод *setTransform()* (рисунок 1.39):

```
setTransform(a, b, c, d, e, f)
```

Его параметры представляют матрицу преобразования, и в целом его применение аналогично применению метода *transform()*. Например:

```
var canvas = document.getElementById("Canvas"),
    context = canvas.getContext("2d");
            var k = 0;
            for (var x = 0; x < 30; x++) {
                k = Math.floor(255 / 34 * x);
context.fillStyle="rgb("+ k + "," + k + "," + k + ")";
            context.fillRect(50, 50, 200, 100);
            context.setTransform(1, 0, 0, 1, x, x);
            }
```

Рисунок 1.39 – Демонстрация работы метода *setTransform()*

Сброс трансформаций. При применении трансформаций вся последующая отрисовка фигур подвергается данным трансформациям. Но возможна ситуация, когда после одиночного применения трансформации нам больше не нужно ее применение. И для всей последующей отрисовки можно сбросить трансформации с помощью метода:

```
var canvas = document.getElementById("Canvas"),
  context = canvas.getContext("2d");
            context.fillStyle = "blue";
            context.fillRect(50, 50, 100, 100);
            context.translate(100, 50);
            // дальше применяется трансформация
            context.globalAlpha = 0.5;
            context.fillStyle = "red";
            context.fillRect(50, 50, 100, 100);
            context.resetTransform();
            // трансформация больше не применяется
            context.fillStyle = "green";
            context.fillRect(0, 0, 100, 100);
```

Рисунок 1.40 – Сброс всех трансформаций

1.12 Рисование мышью

Ранее рассматривали в основном статическую графику на canvas. Можно создавать фигуры динамически, просто рисуя указателем мыши.

Для этого определим следующую страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Canvas </title>
    </head>
    <body>
        <canvas id="Canvas" width="350" height="250"
         style="background-color:#eee; border: 1px
                              solid #ccc; margin:10px;">
            Ваш браузер не поддерживает Canvas
        </canvas>
    <script>
     var canvas = document.getElementById("Canvas"),
                context = canvas.getContext("2d"),
                w = canvas.width,
                h = canvas.height;
            var mouse = { x: 0, y: 0 };
            var draw = false;
   canvas.addEventListener("mousedown", function (e)
            {
                mouse.x = e.pageX - this.offsetLeft;
                mouse.y = e.pageY - this.offsetTop;
                draw = true;
                context.beginPath();
                context.moveTo(mouse.x, mouse.y);
```

```
                });
        canvas.addEventListener("mousemove", function (e)
{
                    if (draw == true) {
                     mouse.x = e.pageX - this.offsetLeft;
                     mouse.y = e.pageY - this.offsetTop;
                     context.lineTo(mouse.x, mouse.y);
                     context.stroke();
                    }
                });
            canvas.addEventListener("mouseup", function (e)
{
                    mouse.x = e.pageX - this.offsetLeft;
                    mouse.y = e.pageY - this.offsetTop;
                    context.lineTo(mouse.x, mouse.y);
                    context.stroke();
                    context.closePath();
                    draw = false;
                });
            </script>
        </body>
    </html>
```

Для обработки движения мыши для элемента canvas определены три обработчика – нажатия мыши, перемещения и отпускания мыши.

При нажатии мыши устанавливаем переменную *draw* равным *true*. То есть идет рисование. Также при нажатии фиксируем точку, с которой будет идти рисование.

При перемещении мыши получаем точку, на которую переместился указатель, и рисуем линию. При отпускании указателя закрываем графический путь методом *context.closePath()* и сбрасываем переменную *draw* в *false* (рисунок 1.41).

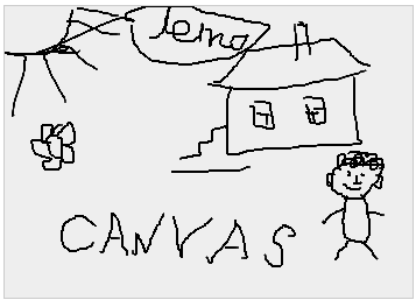


Рисунок 1.41 – Рисунок мышью

2 *WebGL (Web-based Graphics Library)* – кроссплатформенный *API* для *3D*-графики в браузере

2.1 Введение в *WebGL*

WebGL представляет собой технологию, базирующуюся на *OpenGL ES* 2.0 и предназначенную для рисования и отображения интерактивной 2*D*- и 3*D*-графики в веб-браузерах. При этом для работы с данной технологией не требуются сторонние плагины или библиотеки. Вся работа веб-приложений с использованием *WebGL* основана на коде *JavaScript*, а некоторые элементы кода – шейдеры могут выполняться непосредственно на графических процессорах на видеокартах, благодаря чему разработчики могут получить доступ к дополнительным ресурсам компьютера, увеличить быстродействие. Таким образом, для создания приложений разработчики могу использовать стандартные для веб-среды технологии *HTML*/*CSS*/*JavaScript* и при этом также применять аппаратное ускорение графики.

Если создание настольных приложений работающих с 2*D* и 3*D*-графикой нередко ограничивается целевой платформой, то здесь главным ограничением является только поддержка браузером технологии *WebGL*. А сами веб-приложения, построенные с использованием данной платформы, будут доступны в любой точке земного шара при наличии сети интернет вне зависимости от используемой платформы: то ли это десктопы с ОС *Windows*, *Linux*, *Mac*, то ли это смартфоны и планшеты, то ли это игровые консоли.

WebGL возник из экспериментов над *Canvas* 3*D* американского разработчика сербского происхождения Владимира Вукичевича из компании *Mozilla* в 2006 году. Впоследствии разработчики браузеров *Opera* и *Mozilla* стали создавать свои реализации *WebGL*. А впоследствии была организована рабочая группа с участием крупнейших разработчиков браузеров *Apple*, *Google*, *Mozilla*, *Opera* для работы над спецификацией технологии. И в 3 марта 2011 года была представлена спецификация *WebGL* 1.0.

В настоящий момент *WebGL* поддерживается десктопными и мобильными браузерами.

Десктопные браузеры:

- *Mozilla Firefox* (с 4-й версии);
- *Google Chrome* (с 9-й версии);

– *Safari* (с 6-й версии, по умолчанию поддержка *WebGL* отключена)

– *Opera* (с 12-й версии, по умолчанию поддержка *WebGL* отключена);

– *IE* (с 11-й версии, для других версий можно воспользоваться сторонними плагинами, например, *IEWebGL*).

Мобильные браузеры и платформы:

– *Android-браузер* (поддерживает *WebGL* только на некоторых устройствах, например, на смартфонах *Sony Ericsson Xperia* и некоторых смартфонах *Samsung);*

– *Opera Mobile* (начинаю с 12-й версии и только для ОС *Android);*

– *IOS;*

– *Firefox for mobile;*

– *Google Chrome* для *Android.*

Преимущества использования *WebGL:*

– кроссбраузерность и отсутствие привязки к определенной платформе. *Windows*, *MacOS*, *Linux* - все это не важно, главное, чтобы ваш браузер поддерживал *WebGL;*

– использование языка *JavaScript*, который достаточно распространен;

– автоматическое управление памятью. В отличие от *OpenGL,* в *WebGL* не надо выполнять специальные действия для выделения и очистки памяти;

– поскольку *WebGL* для рендеринга графики использует графический процессор на видеокарте (*GPU*), то для этой технологии характерна высокая производительность.

На рисунке 2.1 представлены спецификации, поддерживающие шейдеры.

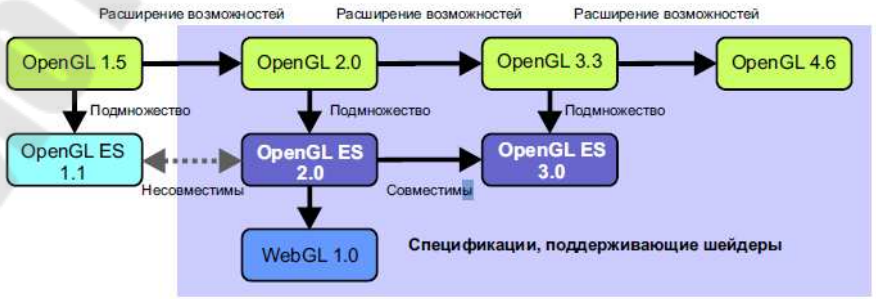


Рисунок 2.1 – Спецификации, поддерживающие шейдеры

2.2 Основы работы с WebGL

Для создания приложения на *WebGL*, во-первых, необходим элемент *canvas*.

```
<canvas id="canvas3D" width="400" height="300">
      Ваш браузер не поддерживает элемент canvas
</canvas>
```

Этот элемент и будет тем полотном, на котором затем произойдет отрисовка сцены. Далее необходимо получить контекст *WebGL*.

Все действие по использование функциональности *WebGL* будет происходить в функции *JavaScript*, которая будет срабатывать при загрузке страницы.

Есть два способа: либо повесить функцию на обработчик *onload* элемента *body*, либо использовать *window.onload*.

Итак, создадим перед закрывающим тегом *body* элемент *script* и изменим страницу следующим образом:

```
<!DOCTYPE html>
<html>
<head>
<title> WebGL</title>
<meta charset="utf-8" />
</head>
<body>
<canvas id="canvas3D" width="400" height="300">
      Ваш браузер не поддерживает элемент
canvas</canvas>
<script type="text/JavaScript">
var gl=null; // контекст WebGL

window.onload=function(){
    // получаем элемент canvas
    var canvas = document.getElementById("canvas3D");
    try {
    // Сначала пытаемся получить стандартный контекст
  //  Если не получится, обращаемся к эксперимент.
контексту
```

```
            gl     =     canvas.getContext("webgl")     ||
canvas.getContext("experimental-webgl");
        }
        catch(e) {}

      //  Если  контекст  не  удалось  получить,  выводим
сообщение
          if (!gl) {
            alert("Ваш браузер не поддерживает WebGL");
          }
        if(gl){
            // покрасим в серый цвет фон
            gl.clearColor(0.5, 0.5, 0.5, 1.0);
            gl.clear(gl.COLOR_BUFFER_BIT);
        }
    }
    </script>
    </body>
    </html>
```

Вначале объявляем глобальную переменную *var gl=null;* – это и есть контекст *WebGL*, через который будет идти рисование.

Далее идет функция, срабатывающая при загрузке страницы. Для начала устанавливаем контекст:

```
gl = canvas.getContext("webgl") ||
     canvas.getContext("experimental-webgl");
```

Элемент *canvas* поддерживает два контекста: 2*D*(используется для простого рисования) и *webgl*.

Необходимо получить контекст *webgl*. Однако некоторые браузеры пока поддерживают экспериментальный контекст - *experimental-webgl*, поэтому в этом случае необходимо получить его, если контекст *webgl* не установлен.

Далее идет закраска.

Метод *gl.clearColor*(0.5, 0.5, 0.5, 1.0) принимает значения *ARGB* для цвета, которым будет закрашено полотно. Чтобы осуществить закраску, вызываем метод *gl.clear*. (Если быть точнее, очищаем цветовой буфер, используя константу *gl.COLOR_BUFFER_BIT*).

Конвейер *WebGL*

Прежде чем приступить к рисованию и созданию объектов, рассмотрим, что собой представляет конвейер *WebGL*. Схематично его можно представить следующим образом (рисунок 2.2):

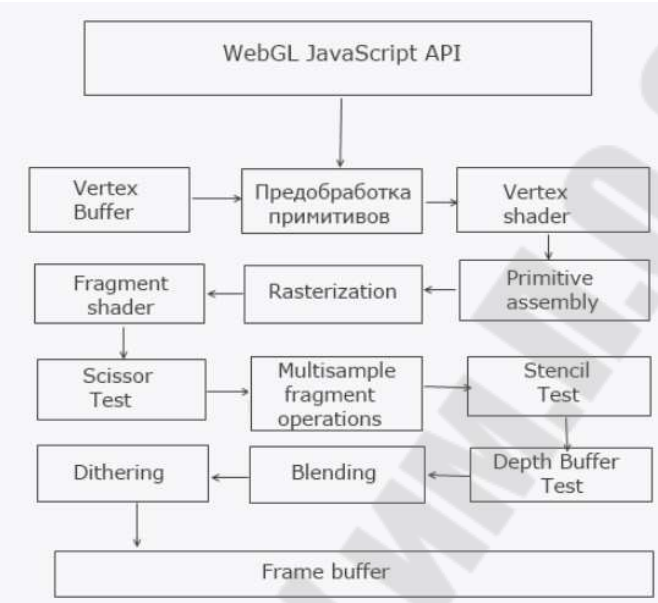


Рисунок 2.2. – Графическое представление конвейера *WebGL*

Разберем поэтапно.

1. Вначале создаем набор вершин в буфере вершин (*Vertex Buffer*). По этим вершинам впоследствии будут составлены геометрические примитивы, а из примитивов – объекты. И проводим некоторую предобработку.

2. 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. На следующем этапе (*Primitive Assembly*) конвейер получает результат вершинного шейдера и пытается измененные вершины сопоставить в отдельные примитивы – линии, треугольники, спрайты. Также на этом этапе определяется, входит ли примитив в видимое пространство. Если нет, то он обрезается. Оставшиеся примитивы передаются на следующий этап конвейера.

4. Далее на этапе растеризации (*Rasterization*) полученные примитивы преобразуются в фрагменты, которые можно представить как пиксели, которые затем будут отрисованы на экране

5. И затем – фрагментный шейдер (*Fragment shader*). (В технологиях *Direct3D*, *XNA* прямым аналогом является пиксельный шейдер). Фрагментый шейдер производит преобразования с цветовой составляющей примитивов, наполняет их цветом, точнее окрашивает пиксели, и в качестве вывода передает на следующий этап измененные фрагменты.

Следующий этап представляет собой ряд преобразований над полученными с фрагментного шейдера фрагментами. Собственно он состоит из нескольких подэтапов:

1) *Scissor Test*: на этом этапе 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.

2) *Multisample Fragment Operations*: на данном этапе у каждого фрагмента изменяются цветовые составляющие, производится сглаживание (*anti-alising*), чтобы объект выглядел более плавно на экране.

3) *Stencil Test*: 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.

4) *Depth Buffer Test* – тест буфера глубины. В буфере глубины (*depth buffer*, а также называется, *z-buffer*) сравнивается *z*-компонента фрагмента, и если она больше значения в буфере глубины, то, следовательно, данный фрагмент расположен к смотрящему на трехмерную сцену 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.

5) *Blending*: на данном этапе происходите небольшое смешение цветов, например, для создания прозрачных объектов.

6) *Dithering*: здесь происходит смешение цветов, для создания тонов и полутонов.

6. *Frame Buffer*: здесь полученные после предобработки фрагменты превращаются в пиксели на экране.

Данные этапы рисуют некоторый алгоритм действий, от которого затем будет отталкиваться. В реальности, создание программы разбивается также на некоторые этапы:

1. создание и настройка шейдеров;
2. создание и настройка буфера вершин, которые в последствии образуют геометрическую фигуру;
3. отрисовка фигуры.

Рассмотим программу отрисовки треугольника.

```
<!DOCTYPE html>
<html>
<head>
<title> WebGL</title>
</head>
<body>
<canvas  id="canvas3D"  width="400"  height="300">Ваш
браузер не поддерживает элемент canvas</canvas>
<!-- фрагментный шейдер -->
<script id="shader-fs" type="x-shader/x-fragment">
  void main(void) {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
  }
</script>
<!-- вершинный шейдер -->
<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() {
    // получаем шейдеры
    var fragmentShader = getShader(gl.FRAGMENT_SHADER,
'shader-fs');
```

```
        var vertexShader = getShader(gl.VERTEX_SHADER,
'shader-vs');
        //создаем объект программы шейдеров
        shaderProgram = gl.createProgram();
        // прикрепляем к ней шейдеры
        gl.attachShader(shaderProgram, vertexShader);
        gl.attachShader(shaderProgram, fragmentShader);
        // связываем программу с контекстом webgl
        gl.linkProgram(shaderProgram);

        if (!gl.getProgramParameter(shaderProgram,
gl.LINK_STATUS)) {
            alert("Не удалсь установить шейдеры");
        }

        gl.useProgram(shaderProgram);
        // установка атрибута программы
        shaderProgram.vertexPositionAttribute =
gl.getAttribLocation(shaderProgram, "aVertexPosition");
        // делаем доступным атрибут для использования

gl.enableVertexAttribArray(shaderProgram.vertexPositionAtt
ribute);
    }
    // Функция создания шейдера по типу и id источника в
структуре DOM
    function getShader(type,id) {
        var source =
document.getElementById(id).innerHTML;
        // создаем шейдер по типу
        var shader = gl.createShader(type);
        // установка источника шейдера
        gl.shaderSource(shader, source);
        // компилируем шейдер
        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 буфера вершин
    function initBuffers() {
     // установка буфера вершин
      vertexBuffer = gl.createBuffer();
      gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
      // массив координат вершин объекта
      var triangleVertices = [
            0.0,  0.5,  0.0,
           -0.5, -0.5,  0.0,
            0.5, -0.5,  0.0
      ];
      gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(triangleVertices), gl.STATIC_DRAW);
      // указываем кол-во точек
      vertexBuffer.itemSize = 3;
      vertexBuffer.numberOfItems = 3;
    }
    // отрисовка
    function draw() {
        // установка области отрисовки
        gl.viewport(0, 0, gl.viewportWidth,
gl.viewportHeight);

        gl.clear(gl.COLOR_BUFFER_BIT);

        // указываем, что каждая вершина имеет по три
координаты (x, y, z)

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribu
te,
                               vertexBuffer.itemSize,
gl.FLOAT, false, 0, 0);
        // отрисовка примитивов - треугольников
        gl.drawArrays(gl.TRIANGLES, 0,
vertexBuffer.numberOfItems);
    }
```

```
    window.onload=function(){
        // получаем элемент canvas
        var canvas = document.getElementById("canvas3D");
        try {
            // Сначала пытаемся получить стандартный
контекст WegGL
            // Если не получится, обращаемся к
экспериментальному контексту
            gl = canvas.getContext("webgl") ||
canvas.getContext("experimental-webgl");
        }
        catch(e) {}

        // Если контекст не удалось получить, выводим
сообщение
          if (!gl) {
            alert("Ваш браузер не поддерживает WebGL");
          }
        if(gl){
            // установка размеров области рисования
            gl.viewportWidth = canvas.width;
            gl.viewportHeight = canvas.height;
            // установка шейдеров
            initShaders();
            // установка буфера вершин
            initBuffers();
            // покрасим фон в бледно-розовый цвет
            gl.clearColor(1.0, 0.0, 0.0, 0.5);
            // отрисовка сцены
            draw();
        }
    }
```

В данной программе по отрисовке треугольника видим, что сначала срабатывают функция *initShaders*(), производящая инициализацию шейдеров и их настройку. Шейдеры являются обязательным звеном в конвейере *WebGL*, поэтому без них сложно будет построить программу.

Затем в дело вступает функция *initBuffers*(), устанавливающая буфер точек, по которым идет отрисовка.

И далее происходит отрисовка в функции *draw*() – при помощи шейдеров буфер вершин превращается в геометрическую фигуру.

Настройка буфера вершин и буфер индексов

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

Так, в предыущей программе создание буфера вершин происходило в следующем участке кода:

```
function initBuffers() {
 // установка буфера вершин
  vertexBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  // массив координат вершин объекта
  var triangleVertices = [
         0.0,  0.5,  0.0,
        -0.5, -0.5,  0.0,
         0.5, -0.5,  0.0
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(triangleVertices), gl.STATIC_DRAW);
  // указываем кол-во точек
  vertexBuffer.itemSize = 3;
  vertexBuffer.numberOfItems = 3;
}
```

Итак, есть глобальная переменная *vertexBuffer*, которая будет хранить буффер вершин, но для начала необходимо его создать:

```
vertexBuffer = gl.createBuffer();
```

Далее 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 создаем буфер вершин, поэтому используется значение *gl.ARRAY_BUFFER*.

Впоследствии можно 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 *Float*32*Array*, который создает представление массива в виде набора чисел с плавающей точкой.

При этом нельзя передать просто переменную *triangleVertices*, представляющую массив координат, необходимо ее типизировать.

Но объект *Float*32*Array* не единственный, который можем использовать для типизации. Также можно использовать следующие объекты: *Int*8*Array*, *Uint*8*Array*, *Int*16*Array*, *Uint*16*Array*, *Int*32*Array*, *Uint*32*Array*, *Float*64*Array*. Но в отличие от *Float32Array* перечисленные типы будут создавать набор целых чисел соответственно занимающих 8, 16, 32 и 64 бита, как указано в их названии.

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

Строки:

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

используются для последующего применения при отрисовке.

Установка атрибута для буфера вершин

В методе отрисовки ***draw*** использовали следующее выражение:

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

С помощью данного метода устанавливаем атрибут или указатель на чтение из буфера вершин. Данный атрибут создаем в функции *initShaders()*:

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

Поскольку здесь определяем для атрибута имя "*aVertexPosition*", то в вершинном шейдере используем переменную с подобным именем:

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

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. Можем установить различные атрибуты, например, один для вершин, другой для цветов и т.д. В данном случае устанавливаем атрибут только для вершин.

И чтобы эти все атрибуты должным образом установить, используем метод

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

Этот метод принимает следующие параметры:

- *index*: индекс атрибута, который сопоставляем с буфером вершин;

– *size*: число значений для каждой вершины, которые хранятся в буфере. Например, у нас три координаты на вершину, поэтому и число передаваемое в качестве данного параметра будет равно 3;

– *type*: тип значений, который хранятся в буфере. В качестве значения параметр может принимать следующие константы: *FIXED*, *BYTE*, *UNSIGNED_BYTE*, *FLOAT*, *SHORT* и *UNSIGNED_SHORT*;

– *norm*: представляет булевое значение. Данный параметр управляет числовыми преобразованиями;

– *stride*: шаг. При установке в качестве значения 0 (нулевой шаг) элементы будут последовательно обрабатываться;

– *offset*: смешение – позиция в буфере, с которой начинается обработка. Обычно устанавливается нулевое смещение, то есть все элементы обрабатываются с самого начала.

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

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

После этого можем передать каждую вершину в вершинный шейдер через переменную *attribute vec*3 *aVertexPosition*.

В программе 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];
```

Итак, сначала 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 необходимо установить указатель на чтение из буфера вершин:

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

Так, как 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 можем получить переданную через атрибут вершину и провести над ней трансформации через переменную *aVertexPosition*:

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

В данном случае никакой трансформации не происходит, и конечная вершина устанавливается как есть.

Отрисовка в WebGL

Для отрисовки фигур в *WebGL* используется следующие методы:

- *gl.drawArrays*()
- *gl.drawElements*()

Метод *gl.drawArrays*() отрисовывает объекты последовательно по вершинам в буфере вершин.

Данный метод принимает три параметра:

gl.drawArrays(mode, index, count)

- *mode* – указывает на отрисуемый примитив и может принимать следующие значения: *gl.POINTS*, *gl.LINES*, *gl.LINE_LOOP*, *gl.LINE_STRIP*, *gl.TRIANGLES*, *gl.TRIANGLE_STRIP*, *gl.TRIANGLE_FAN*;

– *index* – указывает, какой номер вершины в буфере вершин будет первой для примитива;

– *count* – указывает, сколько вершин будет использоваться для отрисовки.

В *WebGL* определены следующие виды примитивов:

– *gl.LINES* – набор линий, при этом все линии рисуются отдельно, если у двух линий определена общая точка, тогда 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 получается 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;

– *gl.POINTS* – набор обычных точек, не соединенных между собой.

Метод *gl.drawElements*() работает с буфером индексов. Он имеет следующую сигнатуру:

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 иметь значение *UNSIGNED_BYTE* или *UNSIGNED_SHORT;*

– *offset* – смещение, с какого индекса будет проводиться отрисовка.

Например:

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

gl.TRIANGLES. Несмотря на то, что треугольник в принципе обычная фигура на плоскости, однако всю трехмерную фигуру можно представить как набор обычных треугольников.

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.

gl.TRIANGLE_STRIP. Данный примитив используется для создания набора соединенных треугольников, которые последовательно соединяют определенные в буфере вершины. Особенно удобно его использовать в тех случаях, когда есть наборы соединенных треугольников. Это позволит уменьшить количество используемых вершин по сравнению, например, с *gl.TRIANGLES*, что приведет к уменьшению объема данных при обработке.

Имея определенное количество вершин, можно подсчитать общее количество треугольников: *count-2*, где *count* – это и есть количество вершин в буфере вершин. То есть, чтобы построить три треугольника нам потребуется 3+2=5 вершин.

gl.TRIANGLE_FAN создает набор треугольников по типу веера, которые имеют одну общую точку.

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

Рисование линий во многом аналогично созданию треугольников, только теперь каждый примитив создается из двух вершин, что облегчает создание.

Другой способ создания линий представляют примитивы *gl.LINE_LOOP* и *gl.LINE_STRIP*.

В отличие от *gl.LINES*, можно обойтись меньшим количеством вершин и индексов, поскольку в *gl.LINE_LOOP* линиии последовательно соединяются, образуя в конце замкнутый контур.

В *gl.LINE_STRIP* вершины также последовательно соединяются линиями, только в конце не происходит соединения последней вершины с первой.

Установка *Viewport*

Отрисовка примитивов с помощью методов *gl.drawElements* или *gl.drawArrays* еще не гарантирует, что эти примитивы сможем увидеть на экране. Для рисования нужна некая область, где будет происходить отрисовка. Наличие просто элемента *canvas* еще недостаточно, чтобы произвести отрисовку.

Для установки области рисования надо настроить *viewport* контекста *WebGL*. В предыдущем примере настройка *viewport* была следующая: сначала получаем 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, система знала, какую часть элемента *canvas* надо использовать для рисования, а *gl.viewport* указывает на левый нижний угол этой области, а также ее ширину и высоту.

Это распространенный способ установки *viewport*. Однако, это не значит, что надо обязательно им пользоваться. Можно задать и другую область рисования: *gl.viewport*(150, 200, 50, 50); :

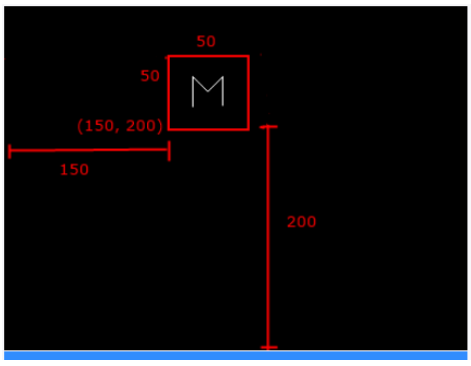


Рисунок 2.3 – Область рисования gl.viewport

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

2.3 Шейдеры. Введение в шейдеры

Шейдеры являются одним из базовых элементов любой программы на *WebGL*. На вход в графический процессов передается лишь набор вершин.

Но благодаря вершинному и фрагментному шейдерам этот набор сначала превращается в набор примитивов, а затем окрашивается, и в итоге видим какие-нибудь трехмерные модели.

Если посмотреть на ранее использовавшиеся шейдеры (все они пока однотипны), то увидите, что в нем используется особый язык с Си-подобным синтаксисом. Например, код вершинного шейдера:

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

Этот язык называется *GLSL*, а если точнее, то язык шейдеров *OpenGL ES Shading Language*.

В свою очередь *GLSL* основан на *C++*.

GLSL довольно распространен.

Этот язык используется в двух частях программы *WebGL* – в вершинном шейдере и фрагментном шейдере.

Для написания кода на *OpenGL ES Shading Language* используются выражения языка *JavaScript*.

Вершинный шейдер

Практически в самом начале работы конвейера *WebGL* буфер вершин передается в вершинный шейдер. Вершинный шейдер проходит по всем переданным вершинам и выполняет определенные преобразования, которые применил в программе вершинного шейдера разработчик.

Именно вершинный шейдер отвечает за матричные преобразования координат, их смещения и т.д.

На выходе он генерирует финальные координаты вершины и передает ее для дальнейшей обработки дальше. До сих пор использовалось простейшее определение шейдера:

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

Атрибут *aVertexPosition*, имеющий тип *vec*3, как раз и передает координаты вершины из буфера вершин. И, так как каждая вершина представлена тремя координатами *x, y, z*, то для передачи вершины используется трехмерный вектор *vec3*.

Как и каждая программа на C/C++, шейдер имеет основную процедуру *main*, в которой происходит генерация окончательных координат вершины, только уже в виде четырехмерного вектора. А *gl_Position* – это и есть преобразованные координаты вершины.

В данном случае у нас никаких преобразований и трансформаций не производится, и финальная вершина будет иметь те же координаты, что и переданная в шейдер.

Фрагментный шейдер

Фрагментный шейдер наполняет набор примитивов цветом, раскрашивает их. Фрагменный шейдер никак не влияет на координаты вершины, он влияет только на цветовую составляющую, преобразуя вершины уже в пиксели или фрагменты.

Ранее были использованы простейшие определения фрагментных шейдеров, например:

```
void main(void) {
  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Здесь также основная программа *main*, которая устанавливает финальный цвет для пикселя, представленного вершиной.

Так как представление цвета состоит из четырех элементов – *RGBA*, то также используется для установки цвета тип четырехмерного вектора *vec4*. В данном случае каждый пиксель примитивов окрашивается в белый цвет.

А если бы захотели покрасить примитивы в зеленый цвет, то могли бы задать следующее определение цвета:

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

Рассмотрим две функции: ***initShaders*() и *getShader*().**

Поскольку создание обоих шейдеров предусматривает однотипные операции, все эти операции были вынесены в отдельную функцию ***getShader*()**.

В эту функцию передаются два параметра: *type (тип шейдера)* и *id элемента*, который содержит код шейдера.

Поскольку метод *gl.createShader*() может создавать шейдеры обоих типов, то в этот метод в качестве параметра и передаем тип создаваемого шейдера. В качестве типа используем встроенные константы:

gl.FRAGMENT_SHADER и *gl.VERTEX_SHADER.*

В функции *initShaders()* создаем оба шейдера и инициализируем программу шейдеров *shaderProgram* и затем устанавливаем атрибут. Далее в функции отрисовке через программу шейдеров используем ранее созданные шейдеры. Таков общий механизм использования шейдеров.

Основы *GLSL*. Базовые моменты синтаксиса *GLSL*

В *GLSL* определены следующие примитивные типы:

- *void*: функция не возвращает никакого значения;
- *bool*: логические значения *true* или *false;*
- *int*: целочисленные значения;
- *float*: числовые значения с плавающей точкой;
- *vec*2, *vec*3, *vec*4: двух-, трех- и четырехмерные векторы соответственно, которые содержат объекты типа *float;*
- *ivec*2, *ivec*3, *ivec*4: двух-, трех- и четырехмерные векторы соответственно, которые содержат объекты типа *int;*
- *bvec*2, *bvec*3, *bvec*4: двух-, трех- и четырехмерные векторы соответственно, которые содержат объекты типа *bool;*
- *mat*2, *mat*3, *mat*4: матрицы размера 2x2, 3x3 и 4x4 соответственно, которые содержат объекты типа *float;*

– *sampler2D*, *samplerCube*: специальные типы – семплеры для работы с текстурами. С помощью сэмплеров во фрагментном шейдере можем получить цветовые значения текстур и передать их примитиву.

Как и *C/C++* в *GLSL* можно также создавать сложные типы – структуры, которые содержать наборы примитивных типов:

```
struct someStruct{
    int someInt;
    vec4 someVec;
}
```

В *GLSL* можно не просто объявить переменную, но и добавить к ней определенное поведение.

Это делается с помощью модификаторов (квалификаторов). Используются следующие модификаторы:

– *attribute*: атрибут или часть описания вершины, которое передается из программы на *WebGL* в вершинный шейдер;

– *const*: константы, эти переменные определяют свое значение только один раз и в процессе программы его уже не меняют;

– *uniform*: по сути то же переменные с константными значениями, только эти значения задаются для всего примитива;

– *varying*: переменная, которая задается в вершинном шейдере и затем передается во фрагментный шейдер, где может быть использована.

Ранее в примерах модификатор в вершинном шейдере применется следующим образом: *attribute vec3 aVertexPosition;*

Здесь *aVertexPosition* представляет описание вершины и имеет тип трехмерного вектора, поскольку используем вершину в трехмерном пространстве.

Квалификаторы для чисел с плавающей точкой

Кроме различных модификаторов также можно использовать квалификаторы для чисел с плавающей точкой – то есть для переменных типа *float*:

– *highp*: число с плавающей точкой сохраняет максимальную точность;

– *mediump*: число со средней степенью точности;

– *lowp*: диапазон плавающей запятой от -2 до 2.

Например, *varying highp vec4 vColor;* – здесь каждое число *float* в векторе *vec*4 имеет высокую точность.

Встроенные глобальные переменные *GLSL*

Кроме задаваемых разработчиком переменных *GLSL* имеет также небольшой набор встроенных глобальных переменных, которые могут использоваться в вершинном или фрагментном шейдерах:

- *gl_Position* – переменная имеет тип *vec*4 и указывает на положение вершины. Используется в вершинном шейдере в качестве выходного параметра;
- *gl_PointSize* – имеет тип *float* и содержит размер точки. Используется в вершинном шейдере в качестве выходного параметра;
- *gl_FragCoord* – имеет тип *vec*4 и указывает на положение фрагмента в буфере фреймов. Используется во фрагментном шейдере в качестве входного параметра;
- *gl_FontFacing* – имеет тип *bool* и определяет, принадлежит ли фрагмент лицевому примитиву. Используется во фрагментном шейдере в качестве входного параметра;
- *gl_PointCoord* – имеет тип *vec*2 и указывает на позицию фрагмента внутри точки. Используется во фрагментном шейдере в качестве входного параметра;
- *gl_FragColor* – имеет тип *vec*4 и указывает на итоговый цвет фрагмента. Используется во фрагментном шейдере в качестве выходного параметра;
- *gl_FragData[n]* – имеет тип *vec*4 и указывает на цвет фрагмента для прикрепления цвета *n*. Используется во фрагментном шейдере в качестве выходного параметра.

Например, в вершинном шейдере устанавливаем переменную *gl_Position* для установки финальной позиции вершины:

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

Можем, например, добавить сжатие по оси *X*, разделив значение координаты X вершины на какой-нибудь коэффициент:

```
attribute vec3 aVertexPosition;
```

```
    const float k=2.0;
     void main(void) {
        float x = aVertexPosition.x / k;
        gl_Position = vec4(x, aVertexPosition.y,
aVertexPosition.z, 1.0);
    }
```

В итоге фигура сожмется в два раза, так как у нас коэффициент 2.0.

Переменная *aVertexPosition* представляет вектор {*x*, *y*, *z*}, поэтому можно обратиться к каждой составляющей отдельно: *aVertexPosition.x* / *k*, затем установить все составляющие вектора *gl_Position*.

Втроенные функции

GLSL обладает рядом встроенных функций. Некоторые из них:

- *dot*(*x*, *y*) – возвращает скалярное произведение векторов *x* и *y;*
- *cross*(*x*, *y*) – возвращает векторное произведение векторов *x* и *y;*
- *matrixCompMult*(*mat x*, *mat y*): возвращает произведение матриц *x* и *y*, которые должны быть одной размерности;
- *normalize*(*x*) – возвращает нормализованный вектор *x*, то есть такой вектор, у которого длина равна 1;
- *reflect*(*t*, *n*) – отражает вектор *t* вдоль вектора *n;*
- *sin*(*angle*) – возвращает синус угла *angle;*
- *cos*(*angle*) – возвращает косинус угла *angle;*
- *pow*(*x*, *y*) – возвращает *x* в степени *y;*
- *max*(*x*, *y*) – возвращает максимальное из двух значений;
- *min*(*x*, *y*) – возвращает минимальное из двух значений.

2.4 Цвета в WebGL. Установка цвета вершины

Чтобы установить для каждой вершины цвета, необходимо задать буфер цветов для вершины, а также определить дополнительный атрибут, который будет передавать данные цветов в шейдеры. Разберем, как это происходит.

Вначале рассмотрим основную программу *WebGL*, а потом шейдеры.

Во-первых, поскольку в шейдеры будут передаваться наборы из двух буферов – цвета и вершин, то создаем два атрибута – для каждого типа:

```
    ashaderProgram.vertexPositionAttribute                    =
gl.getAttribLocation(shaderProgram, "aVertexPosition");
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribu
te);
    shaderProgram.vertexColorAttribute                        =
gl.getAttribLocation(shaderProgram, "aVertexColor");
   gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
```

Затем в функции *initBuffers()* задаем два буфера. Задание буфера цветов во многом похоже на буфер вершин:

```
    var colors = [
            1.0, 0.0, 0.0,
            0.0, 0.0, 1.0,
            0.0, 1.0, 0.0
        ];
    colorBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
    gl.bufferData(gl.ARRAY_BUFFER,  new  Float32Array(colors),
gl.STATIC_DRAW);
```

Значения 1.0, 0.0, 0.0 передает цвет первой вершины, в данном случае это красный цвет. Следующая тройка 0.0, 0.0, 1.0 окрашивает вторую вершину в синий цвет, а третья тройка – в зеленый.

Если при определении буфера вершин каждая тройка обозначает соответственно координаты *x*, *y*, *z*, то при определении буфера цветов три компоненты отвечают за значения *rgb* – то есть цветовые компоненты.

И далее похожим образом создаем буфер цветов и связываем с контекстом *WebGL*.

Далее, в функции отрисовки *draw()* возникает проблема, что надо установить два атрибута – для передачи вершин и цветов. Поэтому производим установку последовательно: сначала для одного, потом для другого.

А общий принцип одинаков для всех.

Теперь перейдем к шейдерам, где эти атрибуты используются.

В вершинном шейдере получаем атрибуты, установленные в основной программе *WebGL*:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexColor;
```

Переменная *varying highp vec4 vColor;* будет передавать цвет из вершинного шейдера во фрагментный. Поэтому у нее используется модификатор *varying*.

Затем в самой программе устанавливаются значения:

gl_Position = *vec*4(*aVertexPosition*, 1.0);
vColor = *vec*4(*aVertexColor*, 1.0);

И во фрагментном шейдере используем для установки финального цвета именно цвет, переданный через переменную *vColor*.

2.5 Матрицы и создание 3D

Первый 3*D*-объект

Все предыдущие примеры фактически рисовались на плоскости, не принимая в учет *z*-координаты. Перейдем к созданию 3*D*-объектов.

Создадим куб. Куб формируют 8 точек, связанных между собою линиями. Итак, у нас есть 8 точек, которые соединены между собой линиями. Теперь можем запустить и увидим наш куб (визуально будем видеть обычный прямоугольник). Однако повернув или изменив точку обзора, можно увидеть трехмерную проекцию куба, но для этого нам понадобятся матрицы

Введение в матрицы.

Матрица ***Model-View Matrix***, которую условно можно назвать матрицей модели, объединяет мировую матрицу и матрицу вида.

Мировая матрица переводит локальные координаты объекта в глобальную систему координат с учетом различных преобразований, а матрица вида переводит глобальное пространство, в котором находятся объекты, в видимое пространство камеры.

Камера в данном случае фактически и есть наш взгляд на трехмерную сцену. И поскольку, условно говоря, камера направлена определенным образом, мы видим не куб, а просто прямоугольник. Но стоит переместить камеру, и увидим куб.

Projection Matrix – матрица проекции преобразует трехмерную систему координат объекта в двухмерные для отображения их на экране. Матрица проекции бывает двух типов: ортогональная и перспективная.

При ортогональной проекции вне зависимости от значения координаты *Z* все объекты будут отрисовываться как есть с их текущими шириной и высотой.

При перспективной проекции будет создаваться видимость глубины или отдаления: близлежащие объекты будут казаться больше, а те, которые находятся дальше, будут меньше. Это будет создавать некоторую иллюзию реалистичности сцены.

Использование матриц *glMatrix* для создания 3*D*

Поскольку *JavaScript* не имеет встроенных средств для работы с матрицами и векторами, то воспользуемся сторонними библиотеками.

Одной из наиболее используемых подобных библиотек является ***glMatrix***. Ее можно найти на официальном сайте разработчика по следующему адресу: *http://glmatrix.net/*. Она относительно небольшая – минимизированная версия весит около 27 кБ, полная – 105 кБ.

Это не единственная библиотека *JavaScript*, которую можно использовать в *WebGL* для работы с матрицами. Есть и другие, например, *Sylvester*, *WebGL-mjs*, но в данном случае будем использовать *glMatrix*. Обратим внимание, что *API* данной библиотеки может меняться, а примеры с ее использованием могут не совсем работать. В этом случае необходимо обратиться к документации. Будем ориентироваться на последнюю версию glMatrix 2.0

Первым делом подключим библиотеку ***glMatrix***, в данном случае используем минимизированную версию.

Сначала определяем матрицы модели и проекции:

```
var mvMatrix = mat4.create();       // матрица модели
var pMatrix = mat4.create();        // матрица проекции
```

Метод *mat4.create()* – представляет метод библиотеки *glMatrix*, используемый для создания матрицы 4x4.

В функции установки шейдеров *initShaders()* создаем две матрицы – для матрицы модели и проекции. Эти матрицы будут передаваться в вершинный шейдер и там применяться к координатам объекта:

```
shaderProgram.MVMatrix =
      gl.getUniformLocation(shaderProgram, "uMVMatrix");
shaderProgram.ProjMatrix =
```

```
gl.getUniformLocation(shaderProgram, "uPMatrix");
```

Матрицы определяются как свойства в объекте *shaderProgram* с помощью метода *gl.getUniformLocation()*, который одинаков для обеих матриц.

Далее, в отличие от предыдущих примеров, добавляем две новых функции: *setupWebGL*() и *setMatrixUniforms*(), которые затем вызываем в основной функции программы.

Функция *setupWebGL*() содержит код по настройке трехмерной сцены, например, установку матриц и т.д.

Вначале устанавливается матрица проекции:

*mat*4.*perspective*(*pMatrix*, 1.04, *gl.viewportWidth* / *gl.viewportHeight*, 0.1, 100.0);

Первый параметр функции – это и есть выше созданная матрица проекции *pMatrix*, которая затем будет передаваться в шейдер.

Второй параметр – 1.04 – это угол обзора в радианах.

Третий параметр *gl.viewportWidth* / *gl.viewportHeight* задает аспектное соотношение ширины к длине.

Четвертый и пятый параметр задают размеры видимой области – ближайшую и самую дальнюю ее точку.

Далее устанавливаем матрицу модели. Сначала для нее устанавливаем матрицу идентичности: *mat*4.*identity*(*mvMatrix*);. А затем применяем к ней перемещение и вращение:

*mat*4.*translate*(*mvMatrix*,*mvMatrix*,*[*0, 0, -2.0*]*);
*mat*4.*rotate*(*mvMatrix*,*mvMatrix*, 1.9, *[*0, 1, 0*]*);

Метод *mat*4.*translate* принимает в качестве первого параметра выходную матрицу, а в качестве второго – входную. Так как преобразуем одну матрицу, то данные параметры у совпадают.

Третьим параметром идет вектор, на который выполняется перемещение, то есть в данном случае идет перемещение на 2 единицы по оси *Z* в сторону от нас.

Метод *mat*4.*rotate* также в качестве первого и второго параметра принимает выходную и входную матрицу. Третьим параметром идет угол в радианах, на который выполняется вращение, а четвертый параметр – вектор, указывающий вдоль какой оси выполнять вращение.

Поскольку в данном случае для оси *Y* указана единичка, а для других нули, то вращение будет идти только по оси *Y*.

И, наконец, метод *setMatrixUniforms*() выполняет передачу матриц в вершинный шейдер:

```
gl.uniformMatrix4fv(shaderProgram.ProjMatrix,  false, pMatrix);
gl.uniformMatrix4fv(shaderProgram.MVMatrix, false, mvMatrix);
```

Фактически, связываем матрицу *shaderProgram.ProjMatrix* с матрицей проекции *pMatrix*, а матрицу *shaderProgram. MVMatrix* с *mvMatrix*.

После этого значения матриц попадут в вершинный шейдер, где будут применены для определения конечных координат вершин:

```
attribute vec3 aVertexPosition;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
void main(void) {
  gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
```

Обратите внимание на порядок преобразований: матрица проекции умножается на матрицу модели, а не наоборот, и потом все умножается на вектор координат вершин.

Дополнительно о матрицах в *WebGL.* Установка камеры

Чтобы прочувствовать, что получился не прямоугольник, а куб, в предыдущем примере пришлось повернуть этот самый куб на определенное количество градусов и еще сместить его по оси *Z*.

Но существует еще другой способ.

Можно не куб поворачивать или перемещать, а устанавливать камеру в определенное место и направлять ее определенным образом, вращать и т.д. То есть можно менять не свойства объекта, а свойства точки обзора.

Для установки камеры в библиотеке *glMatrix* используется функция:

```
mat4.lookAt(matrix, eye, center, up)
```

Она принимает следующие параметры:

- *matrix* – матрица модели, которая меняется от изменения свойств камеры
- *eye* – позиция камеры;
- *center* – точка, на которую направлена камера;
- *up* – вектор вертикальной ориентации.

Посмотрим на примере. Рассмотрим функцию *setupWebGL*():

```
function setupWebGL()
{
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.viewport(0, 0, gl.viewportWidth,
gl.viewportHeight);
    mat4.perspective(pMatrix, Math.PI/2,
gl.viewportWidth / gl.viewportHeight, 0.1, 100.0);
    mat4.identity(mvMatrix);
    mat4.lookAt(mvMatrix, [2, 0,-2], [0,0,0], [0,1,0]);
}
```

Здесь никакого вращения или перемещения. Камера устанавливается в точку пространства с координатами (2, 0, -2) и направлена на точку пространства (0, 0, 0). Направить камеру можно на любую точку пространства. И затем устанавливается вектор вертикальной ориентации. В результате получим трехмерную проекцию куба.

Операции с матрицами

Для перемещения применяется метод:

*mat*4.*translate*(*output*, *input*, *vec*),

где *output* – итоговая выходная матрица, которая получается после перемещения матрицы *input* на трехмерный вектор *vec*. Например, *mat*4.*translate*(*mvMatrix*, *mvMatrix*,*[*0,0,-2.0*]*); – перемещение матрицы *mvMatrix* на 2.0 назад по оси *Z*.

Для вращения применяется метод:

mat4.rotate(output, input, rad, axis),

где *output* - также итоговая матрица, которая получается поворотом матрицы *input* на угол *rad* (в радианах) вокруг оси *axis*. Например, *mat4.rotate(mvMatrix, mvMatrix, Math.PI/2, [1, 0, 0])* – поворачиваем матрицу *mvMatrix* на 90 градусов вокруг оси *X*.

Можно не указывать вектор, а воспользоваться специальными методами:

*mat*4.*rotateX*(*output*, *input*, *rad*),
*mat*4.*rotateY*(*output*, *input*, *rad*) и
*mat*4.*rotateZ*(*output*, *input*, *rad*)

Кроме этих операций можно применить **масштабирование** с помощью метода:

mat4.scale(output, input, vec).

Вектор *vec* указывает масштаб, на который изменяются значения матрицы *input*. Например, *mat4.scale(mvMatrix, mvMatrix, [0.5, 1, 1])* – сжимает значения объекта по оси X, то есть уменьшаем ширину в 2 раза.

Окрашивание куба

По сути окрашивание трехмерных предметов мало чем отличается от окрашивания двухмерных. Однако все же тут есть некоторые моменты, на которые надо обратить внимание.

Во-первых, в отличие от предыдущих примеров, куб создам из треугольников. Одна сторона куба состоит из двух треугольников, поэтому определяем индексы в буфере индексов для создания 12 треугольников, образующих 6 поверхностей: лицевую, заднюю, верх, низ и две боковых.

При этом количество вершин остается тем же, что и в предыдущих примерах.

Затем определяем для каждой вершины цвет:

```
function initBuffers() {
     var vertices =[  // -------лицевая часть
                  -0.5, -0.5, 0.5,
                  -0.5, 0.5, 0.5,
                   0.5, 0.5, 0.5,
                   0.5, -0.5, 0.5,
                  -0.5,  -0.5,  -0.5,        //--  задняя
часть
                  -0.5, 0.5, -0.5,
                   0.5, 0.5, -0.5,
                   0.5, -0.5, -0.5
                   ];
```

```
        var indices = [ // лицевая часть
                    0, 1, 2,
                    2, 3, 0,
                    //нижняя часть
                    0, 4, 7,
                    7, 3, 0,
                    // левая боковая часть
                    0, 1, 5,
                    5, 4, 0,

                    // правая боковая часть
                    2, 3, 7,
                    7, 6, 2,
                    // верхняя часть
                    2, 1, 6,
                    6, 5, 1,
                    // задняя часть
                    4, 5, 6,
                    6, 7, 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;

        // установка цветов для каждой вершины
        var colors = [
            0.0, 0.0, 0.3,
```

```
            0.0, 0.0, 1.0,
            0.0, 1.0, 0.0,
            0.0, 0.3, 0.0,


            0.0, 0.0, 0.3,
            0.0, 0.0, 1.0,
            0.0, 1.0, 0.0,
            0.0, 0.3, 0.0,
        ];
        colorBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
        gl.bufferData(gl.ARRAY_BUFFER,
        new Float32Array(colors), gl.STATIC_DRAW);
    }
```

Также устанавливаем атрибут цвета и передаем его в вершинный шейдер, а через него – во фрагментный.

В общем общий принцип будет таким же, как и при создании цветной двухмерной фигуры.

И самый важный момент, о котором не надо забывать: перед самой отрисовкой необходимо включить глубину, тем самым скрыв те поверхности, которые не должны быть отображены.

gl.enable(gl.DEPTH_TEST);
gl.drawElements(gl.TRIANGLES,
indexBuffer.numberOfItems, gl.UNSIGNED_SHORT,0);

2.6 Анимация и пользовательский ввод

Анимация объектов. Вращающийся куб

Для анмации необходимо добавить переменную *angle*, которая будет представлять угол поворота. С каждым интервалом времени данная переменная будет изменяться, и соответственно будет меняться угол поворота куба. Так же необходмо добавить функцию *requestAnimationFrame* для создания анимации.

Использование *requestAnimationFrame*

Для создания анимации раньше разработчики могли использовать специальные функции *JavaScript*:

- *setTimeout*(*callback*, *timeoutInMilliseconds*)
- *setInterval*(*callback*, *timeoutInMilliseconds*)

Обе функции в качестве параметра *callback* принимают функцию, которая срабатывала через определенное время *timeoutInMilliseconds*.

Разработчики браузеров в последних версиях стали внедрять новое решение – метод *requestAnimationFrame*, который является рекомендуемым способом для создания скриптовой анимации.

Здесь, в отличие от использования методов *setInterval*() и *setTimeout*(), браузер сам оптимизирует анимацию.

Если в случае с *setInterval*() устанавливается интервал анимации пользователем, то при использовании *requestAnimationFrame*() браузер сам определяет оптимальный интервал – ведь в одно и тоже время в браузере могут выполняться сразу несколько анимаций, которые влияют друг на друга.

Поэтому для создания более плавного эффекта браузер может замедлять анимации, эффективно определяя нужный интервал.

В данном случае мы задействуем *requestAnimationFrame*, и в качестве параметра данный метод получает функцию, выполняемую перед перерисовкой.

```
window.requestAnimFrame = (function(){
      return  window.requestAnimationFrame       ||
              window.webkitRequestAnimationFrame ||
              window.mozRequestAnimationFrame    ||
              window.oRequestAnimationFrame      ||
              window.msRequestAnimationFrame     ||
            function(callback, element) {
            return window.setTimeout(callback, 1000/60);
            };
})();
```

Это кроссбраузерное определение метода.

Далее в функции настройки контекста *setupWebGL*() увеличиваем угол на некоторое число и применяем его при повороте матрицы:

```
  angle += 0.01;
  //.........................
```

```
mat4.rotate(mvMatrix,mvMatrix, angle, [0, 1, 0]);
```

В главной функции используем следующую конструкцию, в которую выносим функции по настройке матриц и отрисовке:

```
(function animloop(){
    setupWebGL();
    setMatrixUniforms();
    draw();
    requestAnimFrame(animloop, canvas);
})();
```

Функция *requestAnimFrame*(*animloop*, *canvas*) как раз обращается к *window.requestAnimationFrame*, передавая в параметрах функцию обратного вызова и элемент, который содержит всю анимацию, то есть в данном случае *canvas*.

Подобным образом можно задать анимацию перемещения или масштабирования. Либо анимировать перемещение камеры, а не куба.

Обработка пользовательского ввода

Обработка пользовательского ввода предполагает обработку событий клавиатуры и мыши и не сильно отличается от общей модели, что используется в *JavaScript*, например, при создании игр с использованием элемента *canvas*.

Вначале для фиксации перемещения задаем две переменных:

```
var angle = 2.0;          // угол поворота в радианах
var zTranslation = -2.0;      // смещение по оси Z
```

Дальше, если задан контекст *WebGL*, регистрируем обработчик события нажатия клавиши:

document.addEventListener('*keydown*', *handleKeyDown*, *false*);.

Соответственно добавляем в конце функцию обработчика:

```
function handleKeyDown(e){
    switch(e.keyCode)
    {
```

```
            // изменяем угол поворота
            case 39:  // стрелка вправо
                angle+=0.1;
                break;
            case 37:  // стрелка влево
                angle-=0.1;
                break;
            // изменяем смещение по оси Z
            case 40:  // стрелка вниз
                zTranslation+=0.1;
                break;
            case 38:  // стрелка вверх
                zTranslation-=0.1;
                break;
        }
    }
```

С помощью свойства *e.keyCode* можно узнать код нажатой клавиши и в зависимости от этого выполнить определенные действия. А так как задействована функция анимации *requestAnimFrame*, то в цикле функции для установки матриц получают новые значения и, таким образом, изменяют положение объекта.

2.7 Текстурирование

Текстурирование – нанесение на поверхность объекта изображений, которые помогут имитировать реальность.

Для начала рассмотрим процесс текстурирования на примере двухмерного объекта – прямоугольника.

Однозначно должна быть в памяти загруженная текстура. Для этого создается глобальная переменная *var texture*;.

Основное отличие от предыдущих примеров – функция *setTextures*(). Эта функция затем будет вызываться в главной функции вместо метода отрисовки *draw*().

Рассмотрим функцию *setTextures*().

Для начала создаем текстуру:

texture = gl.createTexture();

Загрузка текстуры

Для установки изображения в качестве текстуры необходим элемент *img*. Изображение, установленное для данного элемента, и будет устанавливаться в качестве текстуры.

Существует два способа:

1. можем заранее определить в структуре *DOM* веб-страницы элемент *img* и с помощью его атрибута *src* установить какое-либо изображение.

2. можем динамически в коде *JavaScript* создать данный элемент, как продемонстрировано в данном примере.

Поскольку текстура не сразу загружается, то используем обработку события *onload*:

```
image.onload = function() {
    handleTextureLoaded(image, texture);
    setupWebGL();
    draw();
}
image.src = "pic.png";
```

Также устанавливаем некоторую картинку. В нашем случае это изображение 128x128 *pic.png*.

Сама установка текстуры происходит в функции *handleTextureLoaded*(*image*, *texture*);.

Когда все настройки текстурирования сделаны, происходит отрисовка.

Функция *handleTextureLoaded*() выполняет важную роль по настройке всех параметров текстурирования.

```
function handleTextureLoaded(image, texture) {
     gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, image);
    gl.texParameteri(gl.TEXTURE_2D,
gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D,
gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    }
```

Прежде чем перейти к использованию текстуры, ее надо связать с объектом текстуры *texture*: *gl.bindTexture*(*gl.TEXTURE_2D*, *texture*);.

Этот метод действует аналогично вызову *gl.bindBuffer*() при связывании буфера вершин.

Используемый далее метод *gl.pixelStorei*() указывает методу *gl.texImage2D*()**,** как текстура должна позиционироваться.

Так, в данном случае передаем в качестве параметра значение *gl.UNPACK_FLIP_Y_WEBGL* – этот параметр указывает методу *gl.texImage2D*(), что изображение надо перевернуть относительно горизонтальной оси.

Это нужно для того, что стандартный объект *Image*, который в данном случае выбран в качестве источника для поставки текстуры, имеет другую систему координат:

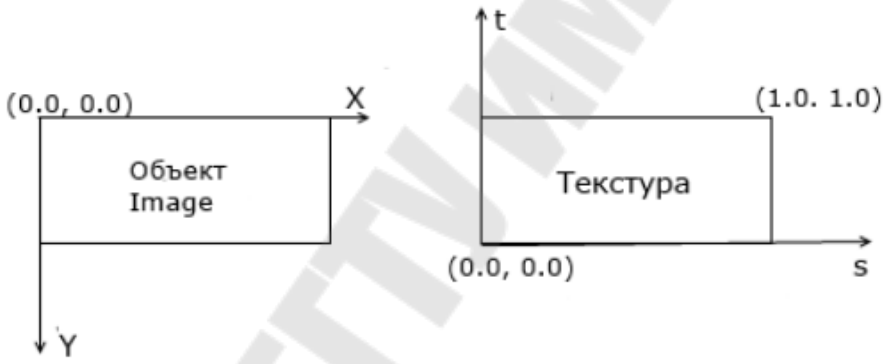


Рисунок 2.4 – Система координат объектов *Image* и *Texture*

Поэтому необходимо совместить две координатные системы, чтобы в будущем было проще работать с текстурой.

Затем можно уже загрузить в текстуру изображение. Это делается с помощью метода *gl.texImage2D* и изображение из элемента ***image*** загружается в текстуру.

Метод *gl.texParameteri* выполняет настройку параметров текстурирования.

Первый вызов этого метода устанавливает значение для параметра *gl.TEXTURE_MAG_FILTER* – тем самым определяем рендеринг текстуры, если она будет увеличена.

Второй вызов метода *gl.texParameteri*, наоборот, определяет поведение рендеринг текстуры, если она будет уменьшена.

Использование текстуры в шейдере

Чтобы использовать текстуру в шейдере и там ее применить к объекту, нанеобходимо в конце функции *setTextures* установить семплер:

shaderProgram.samplerUniform =
gl.getUniformLocation(*shaderProgram*, "*uSampler*");
*gl.uniform*1*i*(*shaderProgram.samplerUniform*, 0);

Семплер будет использоваться для забора из текстуры текселей и совмещения их с пикселями объекта на экране. (**Тексель** представляет собой пиксель на текстуре.) В вершинном шейдере используем следующий код:

```
attribute vec3 aVertexPosition;
varying vec2 vTextureCoords;
void main(void) {
   gl_Position = vec4(aVertexPosition, 1.0);
   vTextureCoords =
     vec2(aVertexPosition.x+0.5,aVertexPosition.y+0.5);
 }
```

Вектор *vTextureCoords* затем будет передан во фрагментный шейдер. Здесь же устанавливаем координаты:

vTextureCoords = vec2(aVertexPosition.x+0.5,aVertexPosition.y+0.5);.

Поскольку вершины реального объекта расположены в пределах от (-0.5, -0.5) до (0.5, 0.5) (так определили в нашем буфере вершин), то к координатам вершины, передаваемой через атрибут *aVertexPosition*, прибавляем 0.5.

Таким образом, в последствии сможем совместить объект с текстурой, у которой все очки находятся в пределах от (0.0, 0.0) до (1.0, 1.0).

Далее используются координаты самой текстуры.

Во фрагментном шейдере приводится в действие семплер:

```
precision highp float;
uniform sampler2D uSampler;
```

```
varying vec2 vTextureCoords;
  void main(void) {
    gl_FragColor = texture2D(uSampler, vTextureCoords);
  }
```

Суммируя все выше сказанное, можно определить следующие этапы текстурирования:

1. Определение и создание объекта текстуры с помощью метода *gl.createTexture*().
2. Создание элемента *html*, который будет выступать источником для текстуры, например, *Image*, и установка его атрибута *src*.
3. Определение метода *onload*, который будет содержать логику, срабатывающую при загрузке изображения в элемент *Image*.
4. Привязка текстуры с помощью метода *gl.bindTexture*().
5. Переворачивание текстуры для совмещения ее с координатной системой объекта *Image* с помощью метода *gl.pixelStorei*(*gl.UNPACK_FLIP_Y_WEBGL*, *true*);
6. Загрузка текстуры в *GPU* с помощью метода *gl.texImage2D*().
7. Установка параметров текстуры с помощью метода *gl.texParameteri*().

Работа с координатами текстуры

Используя координаты текстуры можно более точно проецировать ее на поверхность объекта.

В данном случае сопоставление текстуры с объектом будет идти по координатам.

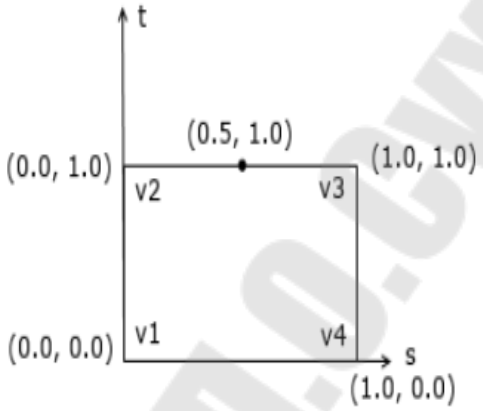
В конце функции инициализации буферов у нас есть такой код:

```
var textureCoords = [       // Координаты текстуры
            0.0, 0.0,
            0.0, 1.0,
            1.0, 1.0,
            1.0, 0.0
        ];
// Создание буфера координат текстуры
textureCoordsBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
gl.bufferData(gl.ARRAY_BUFFER,
      new Float32Array(textureCoords), gl.STATIC_DRAW);
```

```
textureCoordsBuffer.itemSize=2;
   // каждая вершина имеет две координаты
```

Каждая объявленная в массиве *textureCoords* точка на текстуре соответствует вершине двухмерного объекта.

А создание буфера координат текстуры аналогично созданию буфера вершин.



Функция *setTextures*(), которая выполняет загрузку и настройку текстуры осталась та же, за тем исключением, что теперь из нее вынесены методы *setupWebGL*() и *draw*() в функцию анимации.

После создания буфера координат текстуры необходимо его содержание передать в шейдер. Чтобы сделать это, нам, во-первых, надо создать атрибут (это делается в функции настройки шейдеров *initShaders*):

shaderProgram.vertexTextureAttribute = *gl.getAttribLocation*(*shaderProgram*, "*aVertexTextureCoords*");

Далее в функции отрисовки *draw* подобно тому, как создаем указатель на атрибут для вершинного буфера, то же самое проделываем с буфером координат текстуры:

```
gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexTextureAttribute,
          textureCoordsBuffer.itemSize, gl.FLOAT, false, 0, 0);
  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, texture);
```

То есть также создаем указатель на атрибут и подключаем его и далее делаем активной текстуру (*gl.TEXTURE*0) и связываем ее.

WebGL поддерживает работу с несколькими текстурами одновременно, а использование *gl.TEXTURE*0 отсылает нас к первой текстуре.

И в завершении необходимо получить в шейдере атрибут *aVertexTextureCoords* и его использовать.

Для этого в вершинном шейдере заводим переменную для передачи значений текстуры во фрагментный шейдер, а данные она берет как раз из атрибута *aVertexTextureCoords*:

```
attribute vec3 aVertexPosition;
attribute vec2 aVertexTextureCoords;
varying vec2 vTextureCoords;

  void main(void) {
    gl_Position = vec4(aVertexPosition, 1.0);
    vTextureCoords = aVertexTextureCoords;
  }
```

И во фрагментном шейдере так же проходим семплером:

```
gl_FragColor = texture2D(uSampler, vTextureCoords);
```

Настройка текстурирования. Функция *gl.texImage2D*

Данный метод загружает текстуру в *GPU* (графический процессор на видеокарте). Он имеет следующий синтаксис:

texImage2D(*target*, *level*, *internalformat*, *format*, *type*, *elem*)

- *target* – указывает целевой объект для загрузки текстуры
- *level* – уровень множественного отображения текстуры
- *internalformat* и *format* – формат и внутренний формат. В *WebGL* должны иметь одно и то же значение. Так, формат *gl.RGBA*, к примеру, показывает, что для каждого текселя на текстуре должны быть установлены цветовые каналы для красного, зеленого и синего цветов, а также альфа-канал.
- *type*: тип данных, которых сохраняет все данные текселей текстуры. Например, *gl.UNSIGNED_BYTE* указывает, что для каждого цветового канала в *gl.RGBA* для сохранения данных выделяется один байт.
- *elem*: указывает на элемент, который содержит источник текстурирования. Это может быть элемент *img* или *Image*. Это также может быть элемент *HTML5 video* или *canvas*.

Все возможные сочетания форматов и типов:

Формат	***Тип***
gl.RGBA	*gl.UNSIGNED_BYTE*
gl.RGB	*gl.UNSIGNED_BYTE*
gl.RGBA	*gl.UNSIGNED_SHORT_4_4_4_4*
gl.RGBA	*gl.UNSIGNED_SHORT_5_5_5_1*
gl.RGB	*gl.UNSIGNED_SHORT_5_6_5*
gl.LUMINANCE_ALPHA	*gl.UNSIGNED_BYTE*
gl.LUMINANCE	*gl.UNSIGNED_BYTE*
gl.ALPHA	*gl.UNSIGNED_BYTE*

Формат *gl.RGBA* – каждый тексель текстуры имеет канал красного, зеленого и синего цветов, а также альфа-канал.

Формат *gl.RGB* – то же самое, только без альфа-канала.

Формат *gl.LUMINANCE_ALPHA* имеет канал яркости и альфа-канал.

Формат *gl.LUMINANCE* имеет только канал яркости, а формат *gl.ALPHA* - только альфа-канал.

Тип *gl.UNSIGNED_BYTE* предоставляет по одному байту на каждый канал.

Тип *gl.UNSIGNED_SHORT*_4_4_4_4 предоставляет для каждого канала в формате *RGBA* по четыре байта.

Тип *gl.UNSIGNED_SHORT*_5_5_5_1 предоставляет для каждого каналов красного, зеленого и синего цветов в формате *RGBA* по пять байт, а для альфа-канала – один байт.

Тип *gl.UNSIGNED_SHORT*_5_6_5 предоставляет для каналов красного и синего цветов по пять байт и для зеленого цвета - шесть байт в формате *RGB*.

Определение параметров текстуры

Метод *gl.texParameteri*() позволяет определить параметры текстуры. Он имеет следующий формальный синтаксис:

texParameteri(*target*, *pname*, *param*).

Сочетания параметров бывают разными и могут влиять на используемые значения.

– *target* – в зависимости от направления текстурирования может принимать значения *gl.TEXTURE_2D*, либо *gl.TEXTURE_CUBE_MAP*.

– *pname* – указывает на фильтр, который хотим установить. Может принимать следующие значения:

gl.TEXTURE_MAG_FILTER, *gl.TEXTURE_MIN_FILTER*,

gl.TEXTURE_WRAP_S и *gl.TEXTURE_WRAP_T*

– *param* – предоставляет значение для фильтра *pname*. То есть в выражении

gl.texParameteri(*gl.TEXTURE_2D*, *gl.TEXTURE_MAG_FILTER*, *gl.NEAREST*);

фильтру текстуры *gl.TEXTURE_MAG_FILTER* устанавливается значение *gl.NEAREST*.

Значения, передаваемые параметром *param*, разнообразны и позволяют создавать определенные эффекты.

В реальности текстуры имеют определенные размеры, например, 128x128. Однако поверхность объекта, на которую накладывается текстура, может иметь как большие, так и меньшие размеры.

Использование фильтра *gl.TEXTURE_MAG_FILTER* фактически помогает определить рендеринг текстуры, если она меньше размера объекта, то ее надо увеличить.

Фильтр *gl.TEXTURE_MIN_FILTER*, наоборот, указывает, каким образом надо проводить рендеринг, если размеры поверхности объекта меньше размеров текстуры.

gl.NEAREST – данное значение позволяет семплеру взять из текстуры цвет того текселя, центр которого находится ближе всего к точке, с которой семплер берет цветовые значения. Это значение может быть установлено как для фильтра *gl.TEXTURE_MIN_FILTER*, так и для фильтра *gl.TEXTURE_MAG_FILTER*.

gl.LINEAR – данный фильтр возвращает средневзвешенное значение соседних четырех пикселей, центры которых находятся ближе всего к точке, с которой семплер берет цветовые значения. Это обеспечивает цветовую плавность, плавное смешивание цветов.

В то же время, поскольку здесь для определения цвета нужны значения четырех пикселей, то и работать данный фильтр будет медленне, чем *gl.NEAREST*, но при этом более качественней.

Это значение может быть установлено как для фильтра *gl.TEXTURE_MIN_FILTER*, так и для фильтра *gl.TEXTURE_MAG_FILTER*.

***Mip*-текстурирование**

Концепция *mip*-текстурирования предполагает использование нескольких копий одной текстуры, но с разной детализацией. Это позволяет увеличивать качество отображения, например, при удалении от объекта.

Mip-текстурирование в *WebGL* использует ряд фильтров. Подобные фильтры могут использоваться только в качестве значения для фильтра *gl.TEXTURE_MIN_FILTER*:

- *gl.NEAREST_MIPMAP_NEAREST* – фильтр использует одну копию текстуры, которая наиболее подходит под размеры текстуры на экране. Выборка семплером значений происходит по алгоритму *NEAREST*. Самый быстрый способ текстурирования, но при этом менее качественный
- *gl.LINEAR_MIPMAP_NEAREST* – фильтр использует одну копию текстуры, которая наиболее подходит под размеры текстуры на экране. Выборка семплером значений происходит по алгоритму *LINEAR*
- *gl.NEAREST_MIPMAP_LINEAR* – фильтр использует две копии текстуры, которые наиболее подходят под размеры текстуры на экране. Выборка семплером значений происходит по алгоритму *NEAREST*. Выборка цвета пикселя идет параллельно сразу из двух копий, а финальное значение цвета представляет средневзвешенное значение двух выборок
- *gl.LINEAR_MIPMAP_LINEAR* – фильтр использует две копии текстуры, которые наиболее подходят под размеры текстуры на экране. Выборка семплером значений происходит по алгоритму *LINEAR*. Выборка цвета пикселя идет параллельно сразу из двух копий, а финальное значение цвета представляет средневзвешенное значение двух выборок. Наиболее медленный способ, но при этом дающий наибольшее качество

Само использование этих значений для фильтров еще предполагает, что будет использоваться *mip*-текстурирование.

Перед этим надо сгенерировать мипмапы, то есть копии текстуры, с помощью метода *gl.generateMipmap*(*gl.TEXTURE_2D*);.

Этот метод должен вызываться после метода *gl.texImage2D*().

То есть, возьмем функцию *handleTextureLoaded,* которую использоваль ранее, и изменим ее так, чтобы использовались мипмапы:

```
function handleTextureLoaded(image, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, image);
    gl.generateMipmap(gl.TEXTURE_2D);
    gl.texParameteri(gl.TEXTURE_2D,
gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D,
gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);
}
```

Это и даст требуемый результат.

Отметим, что мип-текстурирование имеет некоторые ограничения: используемые изображения должны иметь размеры, которые равны степени двойки. Например, 16*px*, 32*px*, 64*px*, 128*px* и т.д. При этом необязательно, чтобы высота и ширина были равные, главное, чтобы их значения были равны степени двойки.

Texture wrapping

Еще один способ текстурирования называется ***texture wrapping***. Этот термин можно перевести как обертывание текстурой.

То есть данный способ определяет поведение семплера при отборе цветов пикселей с текстуры, если заданные координаты текстуры находятся вне диапазона [0.0, 1.0].

В данном случае потребуется установить значения для фильтров *gl.TEXTURE_WRAP_S* и *gl.TEXTURE_WRAP_T*, которые отвечают за рендеринг текстуры вдоль осей *s* и *t*.

Например, у нас определены следующие координаты текстуры в буфере координат текстуры:

```
    // Координаты текстуры
    var textureCoords = [
                0.0, 0.0,
                0.0, 2.0,
                2.0, 2.0,
                2.0, 0.0,

                0.0, 0.0,
                0.0, 2.0,
                2.0, 2.0,
                2.0, 0.0,

                0.0, 0.0,
                0.0, 2.0,
                2.0, 2.0,
                2.0, 0.0,

                0.0, 0.0,
                0.0, 2.0,
                2.0, 2.0,
                2.0, 0.0
            ];
```

А функция *handleTextureLoaded* выглядела бы следующим образом:

```
    function handleTextureLoaded(image, texture) {
        gl.bindTexture(gl.TEXTURE_2D, texture);
        gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, image);
        gl.generateMipmap(gl.TEXTURE_2D);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
gl.REPEAT);
        gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
gl.CLAMP_TO_EDGE);
    }
```

Для параметров можно использовать следующие значения:

– *gl.CLAMP_TO_EDGE* – все координаты текстуры, которые больше 1 и меньше 0, сжимаются до диапазона *[*0, 1*];*

– *gl.REPEAT* – происходит повторение текстуры после выхода вне диапазона *[*0, 1*];*

– *gl.MIRRORED_REPEAT* – повторение текстуры с зеркальным отображением.

Можно комбинировать данные значения, как в вышеприведенном примере, где одновременно используются *gl.REPEAT* и *gl.CLAMP_TO_EDGE*.

Текстурирование 3*D*-объектов

Фактически все будет так же, как и для двухмерных, только добавляетс больше вершин для имитации 3*D*, и, конечно же, матрицы. Но есть и свои ньюансы.

Совместим текстурирование с созданием трехмерного объекта.

Есть восемь вершин, по которым создается куб. И восемь координат текстуры, которые соответствуют вершинам. Однако, если запустим страничку, то можем увидеть не совсем ожидаемый результат.

Дело в том, что, так как определили всего восемь вершин и восемь соответствующих им координат текстур, то и текстурирование идет только для двух сторон куба, образуемых этими восемью вершинами.

Чтобы оттекстурировать остальные стороны куба, необходимо добавить дополнительные вершины и соответствующие им координаты текстуры. Добавим вершины и координаты еще для двух боковых сторон, изменив функцию *initBuffers* следующим образом:

```
function initBuffers() {
    var vertices =[    // лицевая часть
                 -0.5, -0.5, 0.5,
                -0.5, 0.5, 0.5,
                 0.5, 0.5, 0.5,
                 0.5, -0.5, 0.5,
                // задняя часть
                -0.5, -0.5, -0.5,
                -0.5, 0.5, -0.5,
                 0.5, 0.5, -0.5,
                 0.5, -0.5, -0.5,
                    // левая боковая часть
```

```
                        -0.5, -0.5, 0.5,
                       -0.5, 0.5, 0.5,
                       -0.5, 0.5, -0.5,
                       -0.5, -0.5, -0.5,
                               // правая боковая часть
                        0.5, -0.5, 0.5,
                       0.5, 0.5, 0.5,
                        0.5, 0.5, -0.5,
                        0.5, -0.5, -0.5
                        ];
          var indices = [ // лицевая часть
                       0, 1, 2,
                       2, 3, 0,
                        4, 5, 6,     // задняя часть
                       6, 7, 4,
                        8, 9, 10,    //левая боковая часть
                       10, 11, 8,
                       // правая боковая часть
                       12, 13, 14,
                       14, 15, 12
                    ];
         vertexBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(vertices), gl.STATIC_DRAW);
        vertexBuffer.itemSize = 3;

          indexBuffer = gl.createBuffer();
          gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
indexBuffer);
          gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
Uint16Array(indices), gl.STATIC_DRAW);
          indexBuffer.numberOfItems = indices.length;
       // Координаты текстуры
        var textureCoords = [
                       0.0, 0.0,
                       0.0, 1.0,
                       1.0, 1.0,
                       1.0, 0.0,

                       0.0, 0.0,
```

```
                    0.0, 1.0,
                    1.0, 1.0,
                    1.0, 0.0,

                    0.0, 0.0,
                    0.0, 1.0,
                    1.0, 1.0,
                    1.0, 0.0,

                    0.0, 0.0,
                    0.0, 1.0,
                    1.0, 1.0,
                    1.0, 0.0
                ];
        // Создание буфера координат текстуры
        textureCoordsBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, textureCoordsBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(textureCoords), gl.STATIC_DRAW);
          textureCoordsBuffer.itemSize=2;
    }
```

Так как координаты повторяются, можно сократить код программы, использовав метод *push* и цикл для добавления координат текстуры:

```
    for (var i=0; i<4; i++)  {
      textureCoords.push(0.0,0.0,0.0,1.0,1.0,1.0,1.0,0.0);
    }
```

В итоге будут оттекстурированы все стороны куба.

Множественное текстурирование

При построении трехмерных сцен вряд ли будем ограничиваться одним объектом и одной текстурой. Возникает вопрос, как совместить в проекте использование сразу нескольких текстур для разных объектов?

Один из способов состоит в последовательной отрисовке объектов и использовании текстур.

Необходимо определить для двух объектов свой набор буферов вершин, индексов и координат текстуры.

Переменная ***wallTexture*** будет хранить в себе изображение №1, а переменная ***roofTexture*** – изображение №2.

Для инициализации буферов изображения №1 используется функция *initWallBuffers*, а для инициализации буферов изображеия №2 – функция *initRoofBuffers*.

Далее необходимо два раза инициализировать текстуры и загрузить в них изображения.

Это делается в функции *setupTextures()*:

```
function setupTextures() {
    wallTexture = gl.createTexture();
    setTexture("pic1.png", wallTexture);

    roofTexture = gl.createTexture();
    setTexture("pic2.jpg", roofTexture);
}
```

После срабатывания этой функции сможем использовать текстуры.

Для самой отрисовки каждого объекта создаем две функции:

wallDraw() (отрисовка изображения №1) и

roofDraw() (отрисовка изображения №2).

А в главной функции последовательно вызываем настройку буферов объектов и их отрисовку:

```
initShaders();
        initRoofBuffers();
        initWallBuffers();
        setupTextures();
        (function animloop(){
            setupWebGL();
            setMatrixUniforms();
            wallDraw();
            roofDraw();
            requestAnimFrame(animloop, canvas);
        })();
```

2.8 Введение в освещение

Теперь знаем, как создавать трехмерные объекты, окрашивать их разными цветами и текстурировать. Но в реальности объекты

редко выглядят таковыми. В первую очередь из-за освещения, которое немного искажает отображение объекта, делая его то светлее, то темнее.

Существует несколько различных моделей освещения, различных теорий. Рассмотрим теорию для создания трехмерных сцен.

Типы освещения

Выделяют три типа освещения:

– ***Ambient light*** – окружающее естественное освещение, рассеянный свет, который можем видеть вокруг в повседневной жизни;

– ***Directional light*** – направленный свет, например, солнечный свет;

– ***Point light*** – точечный свет, например, свет лампы.

Материалы

Материалы – по сути, это то, что составляет поверхность объекта.

В программе на *WebGL* материалы могут быть представлены как совокупность параметров, описывающих цвет, используемые текстуры и т.д.

В реальной жизни взаимодействие света и материала объекта может давать различные эффекты: освещение металла будет отличаться от освещения деревянных материалов. Поэтому концепция материала в программе *WebGL* очень важна для имитации освещения, близкого к реальному.

Нормали

Для создания освещения не обойтись без нормалей. Нормали представляют собой векторы, перпендикулярные освещаемой поверхности. Каждая вершина объекта имеет свою связанную с ней нормаль.

Для определения нормалей используется векторное произведение векторов (рисунок 2.5).

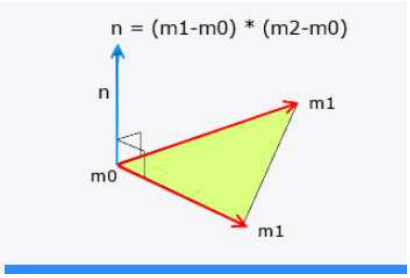


Рисунок 2.5 – Векторное произведение векторов

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

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

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

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

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

Если при создании трехмерных моделей используем специальные программы, например, *Blender*, то там при экспорте объекта вместе с ним экспортируются также и векторы нормалей, и их можно использовать для созданного в *Blender* объекта.

Модели освещения и модели затенения

При создании эффекта освещения необходимо выбрать модель освещения (*lighting model*) (также называются модели отражения), а также модель затенения (*shading model*).

Модель затенения представляет определенный тип интерполяции, позволяющей получить конечное значение цвета объекта в зависимости от освещения.

Модель освещения определяет способ взаимодействия материалов и света для получения финального значения цвета объекта.

Существует разное количество моделей затенения:

– затенение Фонга;
– модель Гуро;
– плоское затенение.

То же самое касается и моделей освещения.

Создание освещенного объекта по модели Фонга. Шейдеры

Основная работа по созданию освещения объекта выполняется в шейдерах в *GPU*, в стандартном коде *JavaScript* от нас требуется только установить ряд объектов и передать их в шейдеры. Это следующие объекты:

- матрицу нормалей;
- нормали вершины;
- направления света и световые точки;
- цвета освещения.

Перейдем к шейдерам, чтобы рассмотреть принцип создания освещенной трехмерной сцены.

Итак, оба шейдера буду выглядеть так:

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec3 aVertexNormal;
  attribute vec2 aVertexTextureCoords;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  uniform mat3 uNMatrix;
  uniform vec3 uLightPosition;
  uniform vec3 uAmbientLightColor;
  uniform vec3 uDiffuseLightColor;
  uniform vec3 uSpecularLightColor;

  varying vec2 vTextureCoords;
  varying vec3 vLightWeighting;

  const float shininess = 16.0;

  void main() {
    // установка позиции наблюдателя сцены
    vec4 vertexPositionEye4 =
        uMVMatrix * vec4(aVertexPosition, 1.0);
```

```
        vec3 vertexPositionEye3 =
      vertexPositionEye4.xyz / vertexPositionEye4.w;

        // получаем вектор направления света
        vec3 lightDirection =
       normalize(uLightPosition - vertexPositionEye3);

        // получаем нормаль
        vec3 normal =
             normalize(uNMatrix * aVertexNormal);

        //   получаем   скалярное   произведение   векторов
нормали и направления света
        float diffuseLightDot =
             max(dot(normal, lightDirection), 0.0);

    //получаем вектор отраженного луча и нормализуем его
        vec3 reflectionVector =
       normalize(reflect(-lightDirection, normal));

        // установка вектора камеры
        vec3 viewVectorEye =
                  -normalize(vertexPositionEye3);

        float specularLightDot =
    max(dot(reflectionVector, viewVectorEye), 0.0);

        float specularLightParam =
                pow(specularLightDot, shininess);
         /*   отраженный   свет   равен   сумме   фонового,
диффузного и зеркального отражений света */
        vLightWeighting = uAmbientLightColor +
           uDiffuseLightColor * diffuseLightDot +
         uSpecularLightColor * specularLightParam;

         // Finally transform the geometry
         gl_Position = uPMatrix *
            uMVMatrix * vec4(aVertexPosition, 1.0);
         vTextureCoords = aVertexTextureCoords;
      }
    </script>
```

```
<script id="shader-fs" type="x-shader/x-fragment">
  precision mediump float;

  varying vec2 vTextureCoords;
  varying vec3 vLightWeighting;
  uniform sampler2D uSampler;

  void main() {
    vec4 texelColor = texture2D(uSampler,
vTextureCoords);
    gl_FragColor = vec4(vLightWeighting.rgb *
texelColor.rgb, texelColor.a);
  }
</script>
```

Все переменные типа *attribute* и *uniform* передаются из основной программы в шейдер. Эти переменные и задают параметры освещения.

В модели отражения Фонга отраженный свет представлен как сумма фонового, диффузного и зеркального отражений.

В шейдерах за хранение параметров отраженного света отвечает переменная *varying vec3 vLightWeighting*.

В вершинном шейдере находим эту сумму и затем передаем переменную во фрагментный шейдер для окончательной установки цвета фрагмента.

Все цвета для нахождения параметров отраженного света передаются через переменные:

uniform vec3 uAmbientLightColor,
uniform vec3 uDiffuseLightColor u
uniform vec3 uSpecularLightColor.

Также передаем в шейдер положение источника света через переменную *uniform vec3 uLightPosition*.

Физический смысл модели Фонга и разбор кода шейдеров

Первым делом необходимо модифицировать координаты вершины и ее нормаль, чтобы правильно провести вычисления по нахождению отражения света, поскольку на входе в шейдер и

вершина, и нормаль пока не учитывают преобразования с матрицами – вращения, перемещения и т.д.:

```
    // установка позиции наблюдателя сцены
    vec4
vertexPositionEye4=uMVMatrix*vec4(aVertexPosition,1.0);
    vec3 vertexPositionEye3 = vertexPositionEye4.xyz /
vertexPositionEye4.w;

    // получаем вектор направления света
    vec3 lightDirection = normalize(uLightPosition -
vertexPositionEye3);

    // получаем нормаль
    vec3 normal = normalize(uNMatrix * aVertexNormal);
```

Вершину преобразуем с помощью матрицы, для установки нормалей применяем матрицу нормалей, которую также устанавливаем в коде *JavaScript*. А также вычисляем вектор от вершины до источника света.

Получение отражений света

Фоновое отражение света (*ambient light*) представляет собой отражение при естественном освещении. Его можно выразить формулой $I = Ka*Ia$, где *Ka* – цветовые параметры материала в виде значений *RGB*, а *Ia* – это цвет фонового света также в виде значений *RGB*.

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

Диффузное отражение света высчитывается по формуле:

$$I = Kd * Id * max(\cos \theta, 0).$$

Здесь кроме диффузного материала *Kd* и цвета освещения *Id* присутствует дополнительный параметр. Этот параметр учитывает направление направленного на поверхность луча. А угол θ – представляет собой угол между нормалью поверхности и вектором направления луча света. Наличие функции максимума, которая

выбирает максимальное число из косинуса угла и нуля позволяет отсечь отрицательные значения и свести их к нулю.

Схематично диффузное отражение представлено на рисунке 2.6.

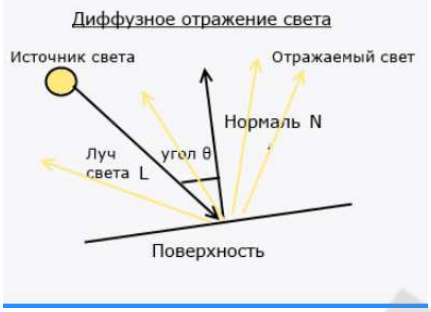


Рисунок 2.6 – Диффузное отражение света

В данном случае *cos θ* – это скалярное произведение векторов *N* (вектор нормали) и *L* (вектор, представляющий луч света). И, таким образом, зная эти вектора, можно рассчитать диффузное отражение (для определения скалярного произведения в языке шейдеров есть специальная функция *dot*):

float diffuseLightDot = max(dot(normal, lightDirection), 0.0);

т.е.получаем скалярное произведение векторов нормали и направления света. Затем получаем зеркальное отражение света (*specular light*). Подобное отражение характеризуется тем, что падающий луч света отражается под одним углом. Зеркальное отражение рассчитывается по формуле $I = K_s * I_s \, max(cos\ \theta, 0)^a$.

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

Схематично формула представлена на рисунке 2.7.

Рисунок 2.7 – Зеркальное отражение света

В данном случае *cos θ* – это скалярное произведение векторов *R* (вектор отраженного луча) и *V* (вектор, направленный к наблюдателю). А, получив на предыдущем шаге векторы нормали и направления падающего луча, можем получить вектор луча отражения *R* и вычислить значение зеркального отражения: *R*=2(*L* * *N*)**N*-*L*.

Однако язык шейдеров имеет встроенную функцию *reflect*, которая позволяет найти вектор отраженного луча по нормали и направлению падающего света. Собственно эта функция и применяется.

Но поскольку при зеркальном отражении вектор луча падающего света направлен в противоположную сторону в отличие от вектора, который применяется при подсчете диффузного отражения света, поэтому используем знак минуса:

vec3 reflectionVector = normalize(reflect(-lightDirection, normal));

Затем применяем скалярное произведение и возводим в степень *shininess*. Параметр *shininess* задается произвольно: чем он выше, тем более блестящим будет казаться отблекс.

vec3 viewVectorEye = -normalize(vertexPositionEye3);
float specularLightDot = max(dot(reflectionVector, viewVectorEye), 0.0);
float specularLightParam = pow(specularLightDot, shininess);

И в конце собираем все световые значения и получаем общее значение отраженного света, которое затем передаем во фрагментный шейдер и далее применяем полученные значения к цветам текстуры:

*gl_FragColor = vec4(vLightWeighting.rgb * texelColor.rgb, texelColor.a);*

В итоге получаем имитацию освещения объекта.

Модель Фонга – не единственная модель отражения света, которая есть и которую можно использовать в *WebGL*. Существуют различные техники создания освещения. Однако она довольно популярная и позволяет учесть различные аспекты при освещении.

Использование материалов

Применение материалов в *WebGL* по сути не привносит ничего сложного,т.е. определяем для каждого типа отражения свой материал и передаем его в вершинный шейдер.

В плане кода, настройка, передача в шейдер и использование материалов, ничем не отличается от использования цветов освещения. Можно убрать весь код, связанный с текстурами. В принципе он не нужен, так как цвет фигуры будут задавать цвета материалов. Поэтому необходимо задать цвет материалов:

```
    // установка материалов
    function setupMaterials() {
gl.uniform3fv(shaderProgram.uniformAmbientMaterialColor,
[0.0, 1.0, 0.0]);

gl.uniform3fv(shaderProgram.uniformDiffuseMaterialColor,
[0.7, 0.7, 0.7]);

gl.uniform3fv(shaderProgram.uniformSpecularMaterialColor,
[1.0, 1.0, 1.0]);
    }
```

И в вершинном шейдере можем их получить и использовать при настройке освещения по модели отражения Фонга:

*vLightWeighting = uAmbientMaterialColor * uAmbientLightColor + uDiffuseMaterialColor * uDiffuseLightColor * diffuseLightDot + uSpecularMaterialColor * uSpecularLightColor * specularLightParam*;

Освещение Ламберта является еще одной моделью освещения. Модель Ламберта намного проще модели Фонга, так как учитывает только одну составляющую – диффузное отражение.

Собственно на уровне кода веб-страницы реализация освещения будет похожей за тем исключением, что в ней будем использовать только цвет диффузного отражения и цвет диффузной поверхности. Финальный цвет отражения в итоге будет равен цвету отражения диффузного света:

```
vLightWeighting =
   uDiffuseMaterialColor*uDiffuseLightColor * lambertTerm;
```

Таким образом, *WebGL* и *Canvas* объединяют *HTML5* и *JavaScript*. Оба инструмента направлены на создание и демонстрацию графических составляющих *HTML5*. Это могут быть растровые изображения, фоны сайтов, анимированная графика, динамические заставки, а также браузерные 2*D* и 3*D* игры.

WebGL обеспечивает возможности 3*D* графики библиотеки *OpenGL*. Он также преобразует геометрическую модель в пиксельный формат, используя шейдеры.

Отметим, что для реализации своего проекта можно пользоваться массой различных библиотек и фреймворков.

3 ЛАБОРАТОРНЫЕ РАБОТЫ

Отчеты по лабораторным работам должны содержать следующие элементы:

- титульный лист;
- номер и название работы;
- цель;
- задание;
- краткие теоретические сведения;
- текст программы с комментариями;
- экранные копии вида страниц сайта в браузере;
- содержательный вывод (копия цели выводом не является).

3.1 Лабораторная работа 1. *Canvas*. Основы

Цель работы: ознакомление с элементом <*canvas*>, стилизацией <*canvas*> и создание простейших *JavaScript*-функций для генерации изображений в рамках элемента <*canvas*>.

Задание:

Самостоятельно нарисовать изображения, приведенные в задании.

1. Разместить элемент <*canvas*> на странице.
2. Создать файл стилей для <*canvas*>.
3. При помощи *JavaScript*-функции реализовать генерацию следующих изображений на странице (рисунок 3.1):

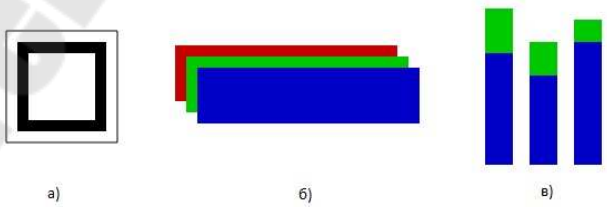


Рисунок 3.1 – Пример заданий для выполнения лабораторной работы

Указания для выполнения лабораторной работы

Размещение <*canvas*> на странице:

```
<canvas id="Lab1" width="150" height="150"
</canvas>
```

Атрибуты *width* и *height* не являются обязательными. По-умолчанию, высота и ширина холста равны 150 и 300 пикселям соответственно. При явном задании размеров холста его картинка масштабируется соответствующим образом.

По аналогии, с такими новыми тегами, как <*audio*> и <*video*>, элементы, размещенные в рамках парного тега <*canvas*>, будут отображаться на странице в том случае, если используемый пользователем браузер поддерживает <*canvas*>.

Стилизация <*canvas*>

При стилизация холста могут быть заданы величины отступа, выравнивания, рамки и т.п. Поскольку не создана функция по отрисовке содержимого холста, то и в браузере ничего не увидим. Создадим стиль для элемента <*canvas*>, задающий параметры границы вокруг элемента:

```
canvas { border: 3px double black;}
```

Прикрепив файл стилей к странице и открыв ее в браузере, получим рамку вокруг элемента <*canvas*> (рисунок 3.2).

Рисунок 3.2 – Рамка вокруг элемента <*canvas*>

Рисование на холсте

Элемент <*canvas*> создаёт поверхность для рисования. Элемент <*canvas*> изначально пустой, и для того, чтобы что-либо отобразить, скрипту необходимо получить контекст отрисовки и рисовать уже на нём. Элемент *canvas* имеет *DOM*-метод, называемый *getContext*, и предназначенный для получения контекста отрисовки вместе с его функциями рисования. *getContext*() принимает один параметр – тип

контекста. Осуществляется все вышесказанное следующим образом, в рамках *JavaScript*-функции:

```
var canvas = document.getElementById("Lab1"),
        context = canvas.getContext("2d");
```

Первая строка создает переменную, фактически являющуюся созданным холстом. Вторая – получает доступ к контексту рисования.

Создадим изображение – прямоугольник багрового цвета.

Для этого, во-первых, создадим *JavaScript* – функцию (не забыв "привязать" файл веб-сценариев к *html*-странице):

```
function createImage() {
var canvas = document.getElementById("Lab1"),
          context = canvas.getContext("2d");
}
```

Прямоугольную закрашенную область можно нарисовать при помощи функции *fillRect* (*x*,*y*,*w*,*h*), где *x* и *y* – координаты левой верхней вершины прямоугольника (по горизонтали и вертикали соответственно), а *w* и *h* – значения ширины и высоты прямоугольника, соответственно.

Функция *strokeRect* (*x*,*y*,*w*,*h*) рисует границы прямоугольника, *clearRect*(*x*,*y*,*w*,*h*) – очищает заданную прямоугольную область.

Функция *fillStyle* задает цвет рисования.

Таким образом, итоговая функция будет выглядеть так:

```
function createImage()
  {
 var canvas = document.getElementById("Lab1");
  var ctx = canvas.getContext("2d");
     ctx.fillStyle = "rgb(100,0,0)"
     ctx.fillRect (20, 10, 50, 70);
     }
```

Для того, чтобы изображение отрисовывалось в момент загрузки страницы, добавим в тег *<body>* вызов функции *createImage* при загрузке. *Html*-код в данном случае будет выглядеть следующим образом:

```
<body onload="createImage();">
<canvas id="lab1" width="150" height="150"> /canvas>
</body>
```

Результат работы в браузере представлен на рисунке 3.3.

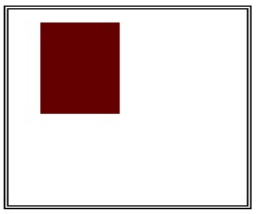


Рисунок 3.3 – Результат выполнения функции *createImage*

Дополнительные указания для выполнения лабораторной работы – в разделе 1.

3.2 Лабораторная работа 2. Canvas. Рисование фигур

При выполнении данной лабораторной работы будем работать на основе файлов, созданных на прошлом занятии *(.html*, *.css* и *.js)*.

Цель: формирование базовых навыков рисования простых фигур на холсте.

Задание: Нарисовать следующие фигуры:

1. Рисование путей

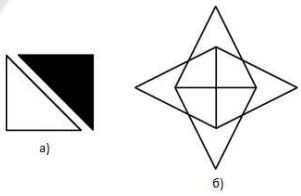


Рисунок 3.4 – Задания к выполнению часть 1

2. Дуги

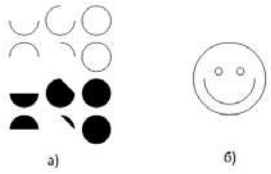


Рисунок 3.5 – Задания к выполнению часть 2

3. Кривые Безье

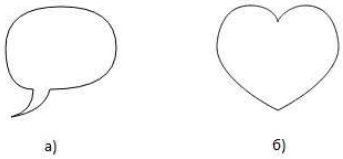


Рисунок 3.6 – Задания к выполнению часть 3

4. Комплексное задание

Рисунок 3.7 – Задание к выполнению часть 4

Указания для выполнения лабораторной работы

Опишем ход выполнения только одного задания в частях 1 – 3. Остальные задания необходимо выполнить самостоятельно.

1. Рисование путей

Рассмотрим задание а).

Обозначим стороны треугольников следующим образом (рисунок 3.8):

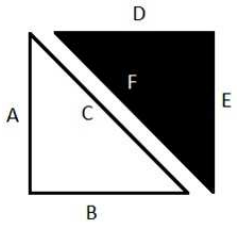


Рисунок 3.8 – Требуемое изображение

Каждый треугольник может быть нарисован последовательным вызовом трех методов *lineTo*.

Создадим путь для отрисовки первого треугольника:

```
function draw(){
  var canvas = document.getElementById("lab2");
  var ctx = canvas.getContext("2d");
  ctx.beginPath();
  ctx.moveTo(20,20);  //задаем  начальную  точку  для
"пера"
  ctx.lineTo(20, 120); // сторона A
  ctx.lineTo(120,120); // сторона B
  ctx.lineTo(20,20); // сторона C
  ctx.stroke(); // рисуем незаполненную фигуру
```

Отрисовка первого треугольника закончена:

Создание залитого треугольника отличается от предыдущего шага незначительно, а именно, вызовом метода *fill*, вместо *stroke*. Добавим соответствующий код в функцию *draw*:

```
function draw(){
  var canvas = document.getElementById("lab2");
  var ctx = canvas.getContext("2d");
  ctx.beginPath();
  ctx.moveTo(20,20); // задаем начальную точку для "пера"
  ctx.lineTo(20, 120); // сторона A
  ctx.lineTo(120,120); // сторона B
  ctx.lineTo(20,20); // сторона C
  ctx.stroke(); // рисуем незаполненную фигуру

  ctx.beginPath();
  ctx.moveTo(40,20); // задаем начальную точку для
                     //отрисовки второго треугольника
  ctx.lineTo(140, 20); // сторона D
  ctx.lineTo(140,120); // сторона E
  ctx.lineTo(40,20); // сторона F
  ctx.fill(); // рисуем залитую цветом фигуру
```

В результате в браузере получаем требуемое изображение (рисунок 3.9).

Рисунок 3.9 – Результат выполнения задания а) части 1

2. Рисование дуг

Эксперименты с рисованием дуг с различными заполнениями и углами – на самостоятельное рассмотрение. Для задания координат для дуг необходимо использовать циклы.

Рассмотрим подробнее задание б).

Можно выделить всего четыре дуги, которые необходимо нарисовать (рисунок 3.10):

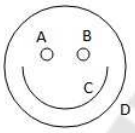


Рисунок 3.10 – Требуемое изображение

Основная сложность этого задания в переносе «пера» в начальные позиции для рисования, поэтому обратите внимание на координаты, указанные в методах *moveTo* и *arc*:

```
    function draw(){
    var canvas = document.getElementById("lab2");
      var ctx = canvas.getContext("2d");
      ctx.beginPath();
      ctx.moveTo(65,65); // начальная позиция для рисование
      ctx.arc(60,65,5,0,Math.PI*2,true); // дуга A
      ctx.moveTo(95,65);  // перенос "пера" в позицию для
отрисовки дуги B
      ctx.arc(90,65,5,0,Math.PI*2,true); // дуга B
      ctx.moveTo(110,75);  // перенос "пера" для отрисовки
дуги C
      ctx.arc(75,75,35,0,Math.PI,false); // дуга C
      ctx.moveTo(125,75);  // перенос "пера" для отрисовки
дуги D
      ctx.arc(75,75,50,0,Math.PI*2,true); // дуга D
      ctx.stroke(); // отрисовка незаполненной фигуры
```

Результат вызова метода *draw* получаем результат выполнения задания б) части 2 (рисунок 3.10).

3. Кривые Безье

Оставим задание б) на самостоятельное выполнение. Отметим только, что для отрисовки каждой "половинки сердца" используется три кривые (метод *bezierCurveTo*).

Рассмотрим задание а). Всего понадобится шесть кривых для выполнения задания:

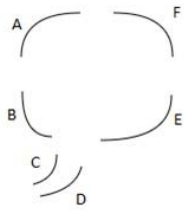


Рисунок 3.11 – Требуемое изображение

Если рисовать кривые последовательно от *A* к *F*, то понадобится только один вызов функции *moveTo* для задания начальных координат для первой кривой (*A*). Как и в предыдущем примере, нужно уделить внимание задаваемым координатам для кривых:

```
function draw(){
var canvas = document.getElementById("lab2");
  var ctx = canvas.getContext("2d");
  ctx.beginPath();
  ctx.moveTo(75,25); // задание начальных координат
  ctx.quadraticCurveTo(25,25,25,62.5); // кривая A
  ctx.quadraticCurveTo(25,100,50,100); // кривая B
  ctx.quadraticCurveTo(50,120,30,125); // кривая C
  ctx.quadraticCurveTo(60,120,65,100); // кривая D
  ctx.quadraticCurveTo(125,100,125,62.5); // кривая E
  ctx.quadraticCurveTo(125,25,75,25); // кривая F
  ctx.stroke(); // отрисовка незаполненной фигуры
```

В результате в браузере получим следующее изображение результат выполнения задания а) части 3 (рисунок 3.11).

Рассмотрены все необходимые функции, оставшиеся задания необходимо выполнить самостоятельно.

Дополнительные указания для выполнения лабораторной работы – в разделе 1.

3.3 Лабораторная работа 3. *Canvas*. Работа с изображениями

Цель: ознакомление с методами импортирования изображений, масштабирование и резка изображений.

Задание:

Разместить на *HTML*-странице кнопки, при нажатии на которые будут выполняться следующие пункты задания:

1. Разместить внешнее изображение на холсте.
2. Создать изображение холста, использовав внешнюю картинку в качестве фона.
3. Замостить холст внешним изображением (масштабирование изображения).
4. Отобразить на холсте часть внешнего изображения.
5. Создать кнопку очистки.

Указания для выполнения лабораторной работы

Для того, чтобы импортировать изображение, необходимо получить ссылку на объект *JavaScript Image*. При этом доступны следующие варианты:

1. В случае, если изображение находится на той же странице, то можно обратиться к нему через *document.images*, *document.getElementByTagName* или *document.getElementById*.

2. При наличии на странице нескольких элементов *canvas*, можно указать один из них в качестве источника изображения для другого посредством *document.getElementByTagName* или *document.getElementById*.

3. Создание изображения в рамках веб-сценария. Объект *JavaScript Image* создается непосредственно в коде.

4. Вложение изображения при помощи *data*: *url.Data urls* позволяют полностью определить изображение непосредственно в коде как *Base*64-кодированную строку. Преимущество данного метода в том, что изображение будет доступно немедленно без каких-либо дополнительных запросов к серверу.

Размещение внешнего изображения на холсте

Разместим холст на странице. В файле веб-сценариев напишем следующий код:

```
    function createImage() {
      var                          ctx                          =
document.getElementById(("lab3").getContext("2d");
      var img = new Image(); // создание нового изображения
      img.src = "img.bmp"; // путь к источнику для img
      img.onload = function() // объявление функции,
                //котороя будет вызываться при загрузке img
      {
       ctx.drawImage(img,0,0);
          // вызов метода отрисовки изображения на холсте
      }
```

Открыв веб-страницу в браузере, получим размещение внешнего изображения на холсте (рисунок 3.12).

Рисунок 3.12 – Результат размещения внешнего изображения на холсте

Создание изображения холста, при использовании в качестве обоев внешнего изображения – для самостоятельного выполнения (необходимо только дописать код, формирующий изображение холста, после вызова *drawImage*).

Масштабирование изображения

Для масштабирования изображения используется следующий вызов *drawImage*:

drawImage(*i*, *x*, *y*, *w*, *h*),

где:

– *i* – объект *JavaScript Image*;
– *x* – начальная координата (по-горизонтали) для отрисовки изображения;
– *y* – начальная координата (по-вертикали) для отрисовки изображения;
– *w* – ширина изображения на холсте;
– *h* – высота изображения на холсте.

Для того, чтобы замостить холст изображениями, очевидно, необходимо *n*-ое количество раз вызвать метод *drawImage*(*i*, *x*, *y*, *w*,*h*) с указанием разных начальных координат для размещения изображения, для чего разумно будет использовать два цикла – счетчика, соответственно для координаты по горизонтали и по вертикали:

```
function createImage()
  {
  var ctx =
document.getElementById("lab3").getContext("2d");
  var img = new Image(); // создание нового
изображения
  img.src = "img.bmp"; // путь к источнику для img
  img.onload = function()
              // функция, вызываемая при загрузке img
  {
  for (h=0;h<3;h++) /* цикл для задания координат
                    (по-вертикали) для изображения */
  {
    for (w=0;w<5;w++) /* цикл для задания координат
                      (по-горизонтали) для изображения */
    {
     ctx.drawImage(img,w*200,h*160,200,160);
     /* вызов метода для отрисовки изображения на
холсте с переменными начальными координатами */
    }
   }
  }
```

Получим следующий результат (рисунок 3.13):

Рисунок 3.13 – Результат масштабирования и
размещения нескольких внешних изображений на холсте

Резка изображения

Третий вариант вызова *drawImage* позволяет отображать только часть внешнего изображения на холсте:

drawImage(*i*, *srcx*, *srcy*, *srcw*, *srch*, *x*, *y*, *w*, *h*),

где:

- *i* – объект *JavaScript Image*;
- *srcx* – координата (по-горизонтали) верхнего левого угла вырезаемого изображения, относительно исходного;
- *srcy* – координата (по-вертикали) верхнего левого угла вырезаемого изображения, относительно исходного;
- *srcw* – ширина вырезаемого изображения;
- *srch* – высота вырезаемого изображения;
- *x* – начальная координата (по-горизинтали) для отрисовки изображения на холсте;
- *y* – начальная координата (по-вертикали) для отрисовки изображения на холсте;
- *w* – ширина изображения на холсте;
- *h* – высота изображения на холсте.

Ниже приведена функция, иллюстрирующая вырезку части изображения и размещения его на холсте:

```
    function createImage()
      {
      var ctx =
document.getElementById("lab3").getContext("2d");
      var img = new Image(); // создание нового изображения
      img.src = "img.bmp";   // путь к источнику для img
      img.onload = function()
```

```
            // функция, вызываемая при загрузке img
{
  ctx.drawImage(img,300,250,100,250,0,0,100,250);
      /* вызов метода для вырезания части изображения
                        и отрисовки его на холсте */
}
```

Открыв *html*-документ в браузере, получим следующее изображение (рисунок 3.14).

Рисунок 3.14 – Результат резки изображения и размещения вырезанной части на холсте

Дополнительные указания для выполнения лабораторной работы – в разделе 1.

3.4 Лабораторная работа 4. *Canvas*. Цвет. Вставка текста. Работа с тенями.

Цель: демонстрация способов работы с цветом *fillStyle*, *strokeStyle* и текстом *strokeText*, *fillText*. Добавление теней.

Задание:

Нарисовать картинку, используя команды работы с цветом, текстом, прозрачностью, тенями (рисунок 3.15).

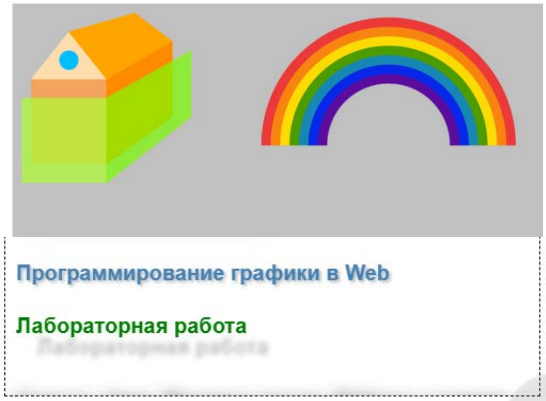


Рисунок 3.15 – Пример выполнения лабораторной работы

Указания для выполнения лабораторной работы

По аналогии с методами *fill* и *stroke*, *fillStyle* задает цвет заливки фигуры, а *strokeStyle* – цвет обводки фигуры.

Допустимыми значениями методов задания цветов могут быть строки – значения цветов *CSS*3.

К примеру, могут быть использованы непосредственно названия цветов. Рассмотрим на примере – создадим массив строк, обозначающих цвета в *CSS* для отрисовки цветов радуги:

```
 function createImage()
      {
      var ctx =
document.getElementById("lab4").getContext("2d");
      var colors = ["red","orange","yellow", "green",
"lightblue", "blue","violet"];
      for (i=0; i<7; i++)
      {
        ctx.fillStyle=colors[i];
        ctx.fillRect(20*i,20,20,200);
      }
      }
```

С результатом можно ознакомиться, открыв *HTML*-страницу (рисунок 3.16).

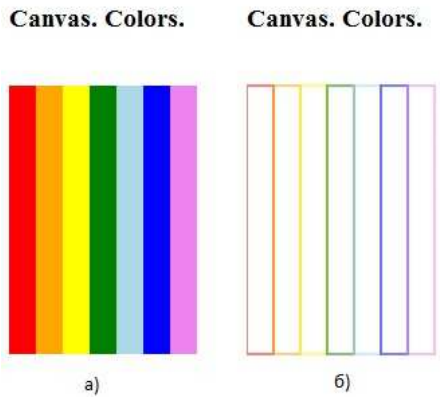


Рисунок 316 – Пример отображения цветов:
а) – при использовании *fillStyle* и *fillRect*
б) – при использовании *strokeStyle* и *strokeRect*

Можно для задания цветов использовать *HEX*-строки – приведенный результат на рисунке 1 не изменится в случае определения массива цветов следующим образом:

```
var colors = ["#FF0000", "#FFA500", "#FFFF00",
"#008000", "#ADD8E6", "#0000FF","#EE82EE"];
```

Помимо указанных способов определения цветов могут быть указаны их *rgb*-обозначения – посредством задания строки вида *rgd*(*r*,*g*,*b*):

```
var colors = ["rgb(255,0,0)", "rgb(255,165,0)",
"rgb(255,255,0)", "rgb(0,255,0)", "rgb(173,216,230)",
"rgb(0,0,255)", "rgb(238,130,238)"];
```

Прозрачность

На холсте можно выводить полупрозрачные фигуры. Значение прозрачности определяется так называемым значением *globalAppha* и находится в пределах [0,0; 1,0]. При этом 1.0 – непрозрачное значение, заданное по-умолчанию.

Для установки прозрачности цвета в методах *fillStyle* и *strokeStyle* используются строки вида "*rgba*(*r*,*g*,*b*,*a*)", где параметр *a* задает значение прозрачности цвета. Иными словами, следующий код *rgba*(255,0,0,0.5) задаст красный цвет средней прозрачности.

Рассмотрим на примере, схожем с предыдущим. Вместо одного прямоугольника, залитого цветом очередного цвета радуги, будем

отрисовывать по четыре прямоугольника одного цвета, но разных степеней прозрачности.

Для хранения *rgb* кодов цветов будем использовать хэш (см. "*JavaScript*. Работа с функциями и массивами"):

```
  function createImage()
  {
    var                          ctx                          =
document.getElementById("lab4").getContext("2d");
    var     clr     =     ["red","orange","yellow","green",
"lightblue", "blue", "violet"];
    /* этот массив понадобится для корректности ссылок
            на элементы хэша */
    var colors = {}; /* объявляем хэш, в котором будем
                        хранить rgb значение цветов*/
    colors["red"] = [255,0,0];
                       /* rgb код красного цвета и т.д.*/
    colors["orange"] = [255,165,0];
    colors["yellow"] = [255,255,0];
    colors["green"] = [0,255,0];
    colors["lightblue"] = [173,216,230];
    colors["blue"] = [0,0,255];
    colors["violet"] = [238,130,238];
    var x=0;
   /*вспомогательная переменная, для задания координаты по
горизонтали очередного залитого цветом прямоугольника*/
    for (i=0; i<7; i++)
    /* цикл для обращения к элементам хэша,
             хранящим массив rgb кода */
    {
      for(j=1; j<5; j++)
      {
       ctx.fillStyle="rgba("+colors[clr[i]][0]+","+
colors[clr[i]][1]+", "+colors[clr[i]][2]+ ","+(j*0.2)+")";
       /*задание параемтров цвета заливки*/
       ctx.fillRect(x,20,20,100);
                    /*отрисовка прямоугольника*/
       x=x+20;
       /*   приращение   переменной   координаты   по   -
горизонтали, для того, чтобы следующий прямоугольник не
накладывался на предыдущий */
```

```
        }
    }
}
```

В результате получим следующее (рисунок 3.17).

Рисунок 3.17 – Пример задания прозрачности для цветов

Вставка текста

Существует два метода и несколько свойств для вывода и форматирования текста. Для вывода текста используют следующие методы:

– *strokeText*(«*text*», *x*, *y*, *width*) – выводит на холст текст без заливки. Здесь «*text*» выводимая строка, *x* и *y* – координаты верхнего левого угла блока с текстом на холсте, *width* – максимальная ширина блока с текстом. Если выводимый текст получается шире, холст выводит его либо шрифтом с уменьшенной шириной символов (если данный шрифт поддерживает такое начертание), либо шрифтом меньшего размера.

– *fillText*(«*text*», *x*, *y*, *width*) – выводит на холст текст без контура, только заливкой. Все параметры повторяют аналогичные у метода *strokeText*(). Для форматирования текста есть несколько различный свойств:

– *font* – свойство позволяет установить все возможные параметры выводимого шрифта;

– *textAlign* – свойство позволяет выравнивать текст относительно блока вывода (точки, которая задается координатами *x* и *y*);

– *textBaseline* – свойство позволяет задать вертикальное выравнивание текста относительно базовой линии.

Рассмотрим пример:

```
<!DOCTYPE HTML>
```

```
<html>
<body>
    <canvas width=600 height=250 id='canvas' > </canvas>
    <script>
        function CreateImage() {
        var cnvs = document.getElementById("canvas");
        var ctxt = cnvs.getContext("2d");
        ctxt.strokeStyle = "black";
        ctxt.font = "bold 30pt sans-serif";
        ctxt.textAlign = "center";
        ctxt.strokeText("HTML5", 75, 70, 120);
        ctxt.fillStyle = "orange";
        ctxt.font = "italic 28pt serif";
        ctxt.textAlign = "center";
        ctxt.fillText("Canvas", 75, 100, 120);
        ctxt.stroke();
        }
        CreateImage();
    </script>
</body>
</html>
```

Результат работы в браузере представлен на рисунке 3.18.

Рисунок 3.18 – Пример вывода текстового изображения

Для добавления теней к фигуре, тексту, изображению используются следующие методы:

– *shadowColor* – устанавливает цвет тени. Можно установить черную или цветную тень, но обычно лучше всего делать ее полусерой. Другой хороший подход – использовать полупрозрачные тени, чтобы можно было видеть содержимое под ними. Чтобы отключить тени, необходимо присвоить атрибуту альфа свойства *shadowColor* нулевое значение;

– *shadowBlur* – устанавливает степень размытия теней. Нулевое значение этого свойства определяет четкую, резкую тень, которая выглядит как силуэт исходного изображения. А значение 20 дает тень в виде размытой дымки, и можно установить еще большее значение.

– *shadowOffsetX* и *shadowOffsetY* – определяют положение тени относительно содержимого, которому она принадлежит. Например, если присвоить каждому свойству значение 5, тень будет расположена на 5 пикселов вправо и 5 пикселов вниз от исходного содержимого. Отрицательные значения сдвигают тень в противоположном направлении – влево и вверх.

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<title> Canvas</title>
    <style>
        canvas {
        border: 1px dashed black;
        }
    </style>
    <script>
        window.onload = function() {
        var canvas =
document.getElementById("drawingCanvas");
        var context = canvas.getContext("2d");
        context.rect(20, 20, 200, 100);
        context.fillStyle = "#8ED6FF";
        context.shadowColor = "#bbbbbb";
        context.shadowBlur = 20;
        context.shadowOffsetX = 15;
        context.shadowOffsetY = 15;
        context.fill(); context.textBaseline = "top";
        context.font = "bold 20px Arial";
        context.shadowBlur = 3;
        context.shadowOffsetX = 2;
        context.shadowOffsetY = 2;
        context.fillStyle = "steelblue";
        context.fillText("Программирование графики в
Web", 10, 175);
```

```
            context.shadowBlur = 5;
            context.shadowOffsetX = 20;
            context.shadowOffsetY = 20;
            context.fillStyle = "green";
            context.fillText("Лабораторная работа", 10, 225);
            context.shadowBlur = 15;
            context.shadowOffsetX = 0;
            context.shadowOffsetY = 0; context.shadowColor =
"black";
            context.fillStyle = "white";
            context.fillText("Canvas. Цвет. Вставка текста.
Работа с тенями", 10, 300);
            }
      </script>
  </head>
  <body>
  <canvas id="drawingCanvas" width="500" height="300">
  </canvas>
  </body>
  </html>
```

Результат работы в браузере представлен на рисунке 3.19.

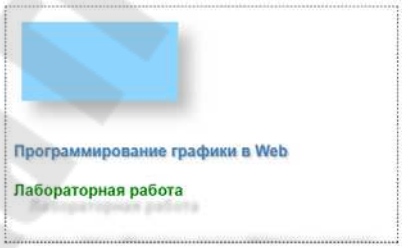


Рисунок 3.19 – Пример вывода текстового изображения

Дополнительные указания для выполнения лабораторной работы – в разделе 1.

Дополнительные ссылки по теме:

http://true-coder.ru/javascript/delaem-mir-canvas-yarche-primenenie-stilej-cvetov-i-tenej.html

3.5 Лабораторная работа 5. *Canvas*. Анимация

Цель: углубленное изучение рисования и знакомство с простейшей анимацией в *canvas* (холст).

Задание:

Нарисовать картинку (домик, человечка) и анимировать его.

Пример картинки представлен на рисунке 3.20.

Рисунок 3.20 – Пример для выполнения лабораторной работы 5

Указания для выполнения лабораторной работы

Чтобы нарисовать кадр, необходимо сделать следующее:

1. Отчистить холст. Самый простой способ сделать это – метод *clearRect*.
2. Сохранить состояние холста. Если меняете любые параметры (стили, преобразования и т.д.), которые влияют на состояние холста и хотите, чтобы исходное состояние использовалось каждый раз, когда кадр рисуется, необходимо его сохранить.
3. Нарисовать анимированые фигуры. Фактически, это шаг создания кадра.
4. Восстановление состояния холста. Если сохранили состояние холста, восстановите его, прежде чем делать новые кадры.

Рассмотрим два несложных примера по отображению анимированного изображения на холсте.

Пример 1. Для начала, просто изобразим движущийся квадрат. Функция *createImage* вызывается при загрузке тела *HTML* документа. В самой функции также определена дополнительная функция *animation*, которая, непосредственно, отрисовывает "кадры", рекурсивно, вызывая саму себя.

В рамках функции *animation* осуществляется следующая последовательность действий:

1. Очистка холста.
2. Отрисовка закрашенного квадрата.
3. Изменение координаты левого верхнего угла квадрата по горизонтали, для "смещения" квадрата на следующем шаге.
4. В случае, если наш квадрат дойдет до края холста, то начальное значение координаты *x* вернется к нулю.
5. Рекурсивный вызов *animation*, посредством метода *setTimeout* (см. http://javascript.ru/setTimeout).

```
function createImage() {
    var ctx =
document.getElementById("lab5").getContext("2d");
    var x = 0;
     function animation() {
        ctx.clearRect(0,0, 1000, 1000);
        ctx.fillRect(x, 10, 10, 10);
        x = x+1;
        if (x > 999) { x = 0;}
          setTimeout(animation, 50);
        }  animation();
}
```

Результат работы в браузере в разный момент времени представлен на рисунке 3.21.

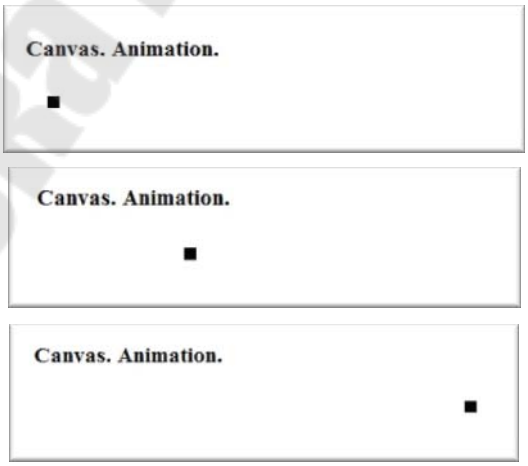


Рисунок 3.21 – Результат выполнения в момент времени 1, 2, 3

Пример 2. Во втором примере рассмотрим отрисовку двух графиков. В отличие от предыдущего примера, не будем очищать холст, для того, чтобы графики формировались в реальном времени.

```
   function createImage()
    {
    var ctx =
document.getElementById("lab5").getContext("2d");
    var x = 0;
    function animation()
    {
     ctx.fillStyle="blue";
     ctx.fillRect(x*100, (Math.sin(x)*100)+100, 3, 3);
     ctx.fillStyle="red";
     ctx.fillRect(x*100, (Math.cos(x)*100)+100, 3, 3);
     x = x+0.05;
     setTimeout(animation, 50);
    }
    animation();
   }
```

Результат выполнения представлен на рисунке 3.22.

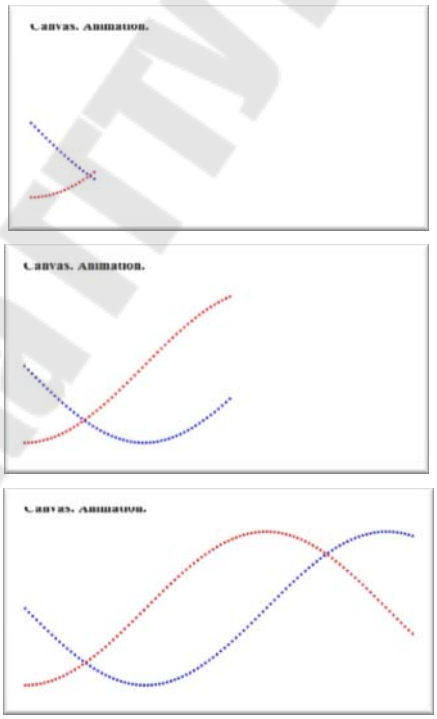


Рисунок 3.22 – Результат выполнения в момент времени 1, 2, 3

Очевидно, что основной сложностью при работе с анимацией является создание алгоритма отрисовки очередного кадра. С более сложными примерами анимации рекомендуется ознакомиться по ссылкам приведенным ниже.

Дополнительные указания для выполнения лабораторной работы – в разделе 1.

Дополнительные ссылки по теме:

1. *http://msdn.microsoft.com/ru-ru/library/gg589516%28v=vs.85%29.aspx*
2. *http://habrahabr.ru/post/119585/*

3.6 Лабораторная работа 6. Введение в *WebGL*. Основы работы с *WebGL*

***Цель работы**:* ознакомление с *WebGL*, основными принципами и методами отрисовки линий, фигур и изображений.

Задание:

4. Написать инициалы с помощью линий.
5. Нарисовать трехмерный тетраэдр, используя библиотеку *three.js*.

Указания для выполнения лабораторной работы – в разделе 2 подраздел 2.1-2.2.

3.7 Лабораторная работа 7. *WebGL*. Использование шейдеров в программе.

***Цель работы**:* получение навыков по отрисовке линий с помощью функций *initShaders()* и *getShader()*.

Задание:

Написать свою Фамилию И.О. с помощью линий, используя функции: *initShaders()* и *getShader()*.

Пример выполнения лабораторной работы представлен на рисунке 3.23.

Рисунок 3.23 – Результат выполнения лабораторной работы 7

Указания для выполнения лабораторной работы – в разделе 2 подраздел 2.3.

3.8 Лабораторная работа 8. *WebGL*. Установка цвета вершины.

***Цель работы**:* ознакомление с цветом в *WebGL*, а также основными принципами для работы с цветом.

Задание:

1. Изучить основные принципы работы с цветом в *WebGL*.
2. Отрисовать треугольник с вершинами разных цветов.
3. Написать функцию, случайным образом меняющую цвет каждой вершины треугольнка (пример выполнения представлен на рисунке 3.24).

Рисунок 3.24 – Результат выполнения лабораторной работы 8

Указания для выполнения лабораторной работы – в разделе 2 подраздел 2.4.

3.9 Лабораторная работа 9. *WebGL*. Матрицы и создание *3D*.

***Цель работы**:* ознакомление с матрицами *WebGL*, создание простейших 3*D* объектов.

Задание:

1. Изучить основные принципы работы с матрицами и объемными объектами.
2. Написать программу, выполняющую изменение цвета куба.

3. Изменение цвета куба организовать в видее выпадающего списка (пример представлен на рисунке 3.25).

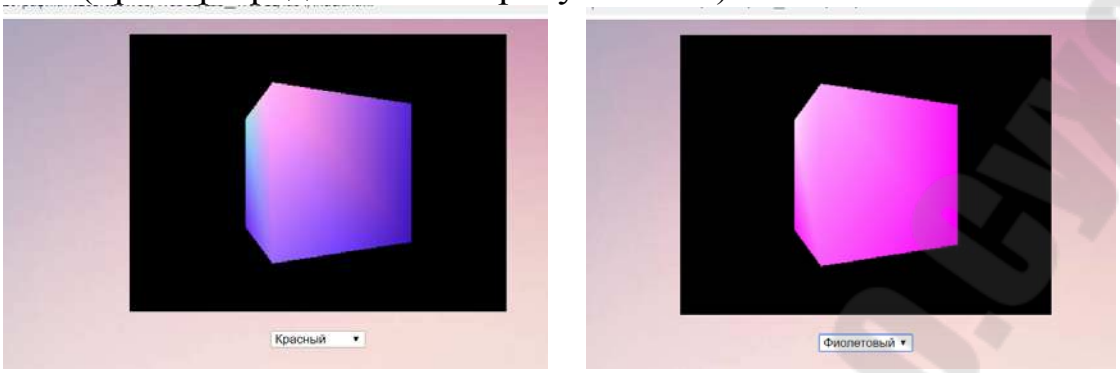


Рисунок 3.25 – Результат выполнения лабораторной работы 9

Указания для выполнения лабораторной работы – в разделе 2 подраздел 2.5.

3.10 Лабораторная работа 10. *WebGL*. Анимация и пользовательский ввод

***Цель работы**:* знакомство с анимацией 3D моделей в WebGL, основными принципами анимирования объемных изображений и принципами работы с пользовательским вводом.

Задание:

1. Нарисовать пирамиду и добавить кнопки для управления анимацией (пример представлен на рисунке 3.26).

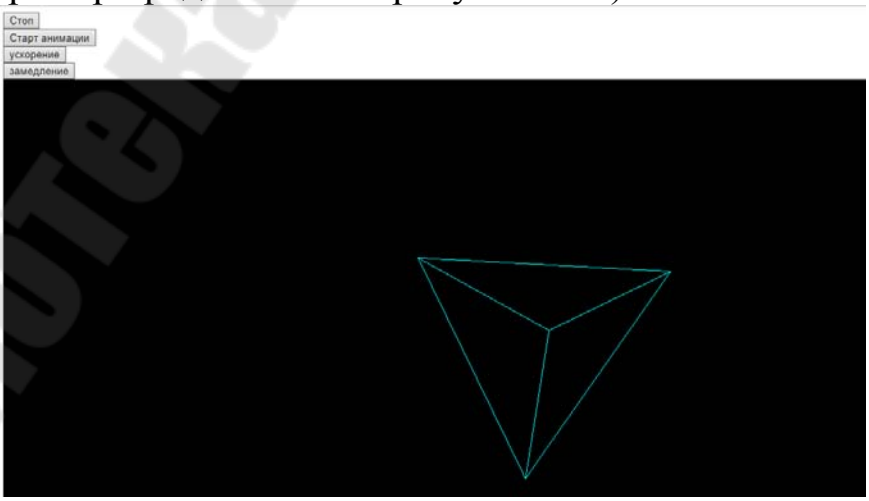
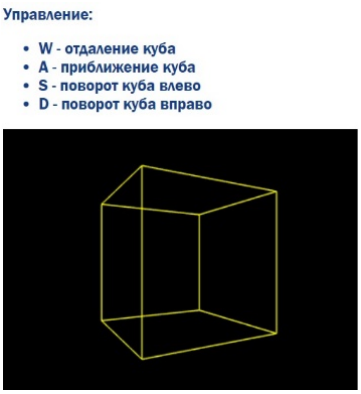


Рисунок 3.26 – Результат выполнения первой части лабораторной работы 10

2. Добавить кнопки для управления положением куба аналогично управлением стрелками (пример представлен на рисунке 3.27).



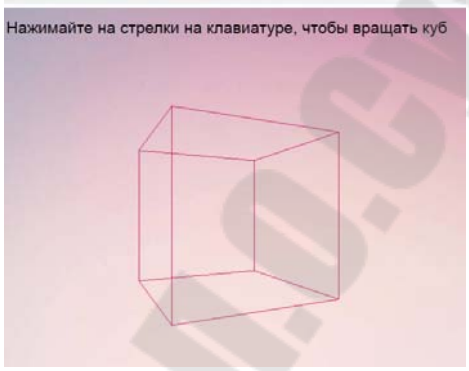
или


Рисунок 3.27 – Результат выполнения лабораторной работы 10

Указания для выполнения лабораторной работы – в разделе 2 подраздел 2.6.

3.11 Лабораторная работа 11. *WebGL*. Текстурирование.

***Цель работы**:* получение навыков по текстурированию 3*D* фигуры с *WebGL.*

Задание:

Для каждой стороны куба наложить отдельную текстуру (пример выполнения представлен на рисунке 3.28).

Рисунок 3.28 – Результат выполнения лабораторной работы 11

Указания для выполнения лабораторной работы – в разделе 2 подраздел 2.7.

3.12 Лабораторная работа 12. WebGL. Освещение.

***Цель работы**:* знакомство с освещением объектов в *WebGL*, его типами и способом задания.

Задание:

Написать программу, позволяющую задавать цвет освещения в модели *RGB*. Для этого добавить 3 поля ввода *RGB*, значения которых меняли бы цвет освещения для куба (пример выполнения представлен на рисунке 3.29).

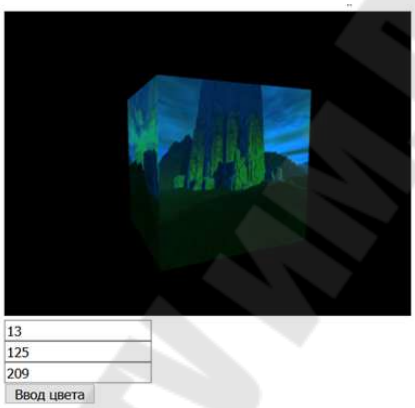


Рисунок 3.29 – Результат выполнения лабораторной работы 12

Указания для выполнения лабораторной работы – в разделе 2 подраздел 2.8.

4 Примерный перечень контрольных вопросов по курсу

1. Доступ к *Canvas* и рисование прямоугольников
2. *Canvas*. Настройка рисования
3. *Canvas*. Фоновые изображения
4. *Canvas*. Создание градиента
5. *Canvas*. Рисование текста
6. *Canvas*. Рисование фигур
7. *Canvas*. Рисование изображений
8. *Canvas*. Добавление теней
9. *Canvas*. Редактирование пикселей
10. *Canvas*. Трансформации
11. *Canvas*. Рисование мышью
12. *WebGL*. Создание контекста
13. *WebGL*. Конвейер *WebGL*
14. *WebGL*. Настройка буфера вершин и буфер индексов
15. *WebGL*. Установка атрибута для буфера вершин
16. *WebGL*. Отрисовка в *WebGL*
17. *WebGL*. *gl.TRIANGLES*
18. *WebGL*. *gl.TRIANGLE_STRIP*
19. *WebGL*. *gl.TRIANGLE_FAN*
20. *WebGL*. Рисование линий
21. *WebGL*. Установка *Viewport*
22. В *WebGL*. Введение в шейдеры
23. *WebGL*. Использование шейдеров в программе
24. *WebGL*. Основы *GLSL*
25. *WebGL*. Установка цвета вершины
26. *WebGL*. Первый 3*D*-объект
27. *WebGL*. Использование матриц *glMatrix* для создания 3*D*
28. *WebGL*. Дополнительно о матрицах в *WebGL*
29. *WebGL*. Окрашиваем куб
30. *WebGL*. Анимация объектов. Вращающийся куб
31. *WebGL*. Обработка пользовательского ввода
32. *WebGL*. Введение в текстурирование
33. *WebGL*. Работа с координатами текстуры
34. *WebGL*. Настройка текстурирования
35. *WebGL*. Текстурирование 3*D*-объектов
36. *WebGL*. Множественное текстурирование
37. *WebGL*. Введение в освещение

38. *WebGL*. Создание освещенного объекта по модели Фонга. Шейдеры

39. *WebGL*. Модель отражения Фонга. Код *JavaScript*

40. *WebGL*. Использование материалов

41. *WebGL*. Освещение Ламберта

Список рекомендуемой литературы по курсу

1. Web Technologies: HTML, Java Script, PHP, Java, JSP, XML and AJAX = Сетевые технологии: HTML, Java Script, PHP, Java, JSP, XML and AJAX. – Delhi : Dreamtech Press, 2010 – 1354p.

2. Дакетт, Д. Основы веб-программирования с использованием HTML, XHTML и CSS / Д. Дакетт. – Москва: Эксмо, 2010 – 767 с.

3. Евсеев, Д.А. Web-дизайн в примерах и задачах: учебное пособие / Д. А. Евсеев, В. В. Трофимов; под ред В. В. Трофимова. – Москва: КНОРУС, 2009 – 263 с.

4. Макфарланд, Д. С. Большая книга CSS3 / Дэвид Макфарланд ; пер. с англ. Н Вильчинского. - 3-е изд.. - Санкт-Петербург [и др.] : Питер, 2014. - 608 с.. - (Бестселлеры O'Reilly)

5. Роббинс, Д. HTML5, CSS3 и JavaScript . Исчерпывающее руководство / Дженнифер Роббинс ; [пер. с англ. М. А. Райтман]. - 4-е изд.. - Москва : Эксмо, 2014. - 528 с. + 1 электрон. опт. диск (DVD). - (Мировой компьютерный бестселлер)

6. Фрэйн, Б. HTML5 и CSS3. Разработка сайтов для любых браузеров и устройств / Бен Фрэйн ; пер. с англ. В. Черника. - Санкт-Петербург [и др.] : Питер, 2014. - 298 с. УДК 004.738.1 ББК 32

7. Боресков, В. Разработка и отладка шейдеров (+ CD-ROM) / В. Боресков. - Москва: БХВ-Петербург, 2006. - 488 с.

8. Боресков, В. Расширения OpenGL (+ CD-ROM) / В. Боресков. - Москва: БХВ-Петербург, 2005. - 688 с.

9. Боресков, В. Расширения OpenGL / В. Боресков. - М.: БХВ-Петербург, 2005. - 688 с.

10. Вольф, Дэвид OpenGL 4. Язык шейдеров. Книга рецептов / Дэвид Вольф. – Москва : ДМК Пресс, 2015. - 368 с.

11. Гинсбург, Дэн OpenGL ES 3.0. Руководство разработчика / Дэн Гинсбург , Будирижанто Пурномо. – Москва : ДМК Пресс, 2015. - 448 с.

12. Курушин, Г. А. Применение метода Canvas для формирования деловой модели проекта девелопмента / Курушин Г. А. // Менеджмент в России и за рубежом. - 2013. — № 6. — С. 93—98.

13. Мацуда, Коичи WebGL. Программирование трехмерной графики / Коичи Мацуда , Роджер Ли. – Москва : ДМК Пресс, 2015. - 494 с.

14. http://htmlbook.ru/

15. https://metanit.com/web/html5/

16. https://www.labirint.ru/books/465875/
17. https://webglfundamentals.org/webgl/lessons/ru/
18. Статья «Начало работы с WebGL»:
19. https://msdn.microsoft.com/ru-ru/library/dn385807(v=vs.85).aspx
20. Официальный сайт WebGL: https://www.khronos.org/webgl/
21. Статья на habrahabr «WebGL для всех» :
22. https://habrahabr.ru/company/2gis/blog/273735/
23. https://davidwalsh.name/webgl-demos
24. https://davidwalsh.name/webgl-demo
25. https://www.chromeexperiments.com/webgl

СОДЕРЖАНИЕ

Титова Людмила Константиновна
Ломако Сергей Олегович

ПРОГРАММИРОВАНИЕ ГРАФИКИ В WEB

Практикум
по выполнению лабораторных работ
по одноименной дисциплине для студентов
специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)»
дневной формы обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 31.03.21.

Рег. № 34Е.
http://www.gstu.by