WebGL™

Web Graphics Library

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

About the Tutorial

WebGL (Web Graphics Library) is the new standard for 3D graphics on the Web, designed for rendering 2D graphics and interactive 3D graphics.

This tutorial starts with a basic introduction to WebGL, OpenGL, and the Canvas element of HTML-5, followed by a sample application. This tutorial contains dedicated chapters for all the steps required to write a basic WebGL application. It also contains chapters that explain how to use WebGL for affine transformations such as translation, rotation, and scaling.

Audience

This tutorial will be extremely useful for all those readers who want to learn the basics of WebGL programming.

Prerequisites

It is an elementary tutorial and one can easily understand the concepts explained here with a basic knowledge of JavaScript or HTML-5 programming. However, it will help if you have some prior exposure to OpenGL language and matrix operation related to 3D graphics.

Copyright & Disclaimer

Table of Contents

Part 1: WebGL Overview

1. WEBGL– Introduction

A few years back, Java applications – as a combination of applets and JOGL – were used to process 3D graphics on the Web by addressing the GPU (Graphical Processing Unit). As applets require a JVM to run, it became difficult to rely on Java applets. A few years later, people stopped using Java applets.

The Stage3D APIs provided by Adobe (Flash, AIR) offered GPU hardware accelerated architecture. Using these technologies, programmers could develop applications with 2D and 3D capabilities on web browsers as well as on IOS and Android platforms. Since Flash was a proprietary software, it was not used as web standard.

In March 2011, WebGL was released. It is an openware that can run without a JVM. It is completely controlled by the web browser.

The new release of HTML 5 has several features to support 3D graphics such as 2D Canvas, WebGL, SVG, 3D CSS transforms, and SMIL. In this tutorial, we will be covering the basics of WebGL.

What is OpenGL?

OpenGL (Open Graphics Library) is a cross-language, cross-platform API for 2D and 3D graphics. It is a collection of commands. OpenGL4.5 is the latest version of OpenGL. The following table lists a set of technologies related to OpenGL.

API	Technology Used
OpenGL ES	It is the library for 2D and 3D graphics on embedded systems - including consoles, phones, appliances, and vehicles. OpenGL ES 3.1 is its latest version. It is maintained by the Khronos Group www.khronos.org
JOGL	It is the Java binding for OpenGL. JOGL 4.5 is its latest version and it is maintained by jogamp.org.
WebGL	It is the JavaScript binding for OpenGL. WebGL 1.0 is its latest version and it is maintained by the khronos group.

OpenGLSL	**OpenGL Shading Language**. It is a programming language which is a companion to OpenGL 2.0 and higher. It is a part of the core OpenGL 4.4 specification. It is an API specifically tailored for embedded systems such as those present on mobile phones and tablets.

Note: In WebGL, we use GLSL to write shaders.

What is WebGL?

WebGL (Web Graphics Library) is the new standard for 3D graphics on the Web, It is designed for the purpose of rendering 2D graphics and interactive 3D graphics. It is derived from OpenGL's ES 2.0 library which is a low-level 3D API for phones and other mobile devices. WebGL provides similar functionality of ES 2.0 (Embedded Systems) and performs well on modern 3D graphics hardware.

It is a JavaScript API that can be used with HTML5. WebGL code is written within the **<canvas>** tag of HTML5. It is a specification that allows Internet browsers access to Graphic Processing Units (GPUs) on those computers where they were used.

Who Developed WebGL

An American-Serbian software engineer named **Vladimir Vukicevic** did the foundation work and led the creation of WebGL.

- In 2007, Vladimir started working on an **OpenGL** prototype for Canvas element of the HTML document.

- In March 2011, Kronos Group created WebGL.

Rendering

Rendering is the process of generating an image from a model using computer programs. In graphics, a virtual scene is described using information like geometry, viewpoint, texture, lighting, and shading, which is passed through a render program. The output of this render program will be a digital image.

There are two types of rendering:

- **Software Rendering:** All the rendering calculations are done with the help of CPU.

- **Hardware Rendering:** All the graphics computations are done by the GPU (Graphical processing unit).

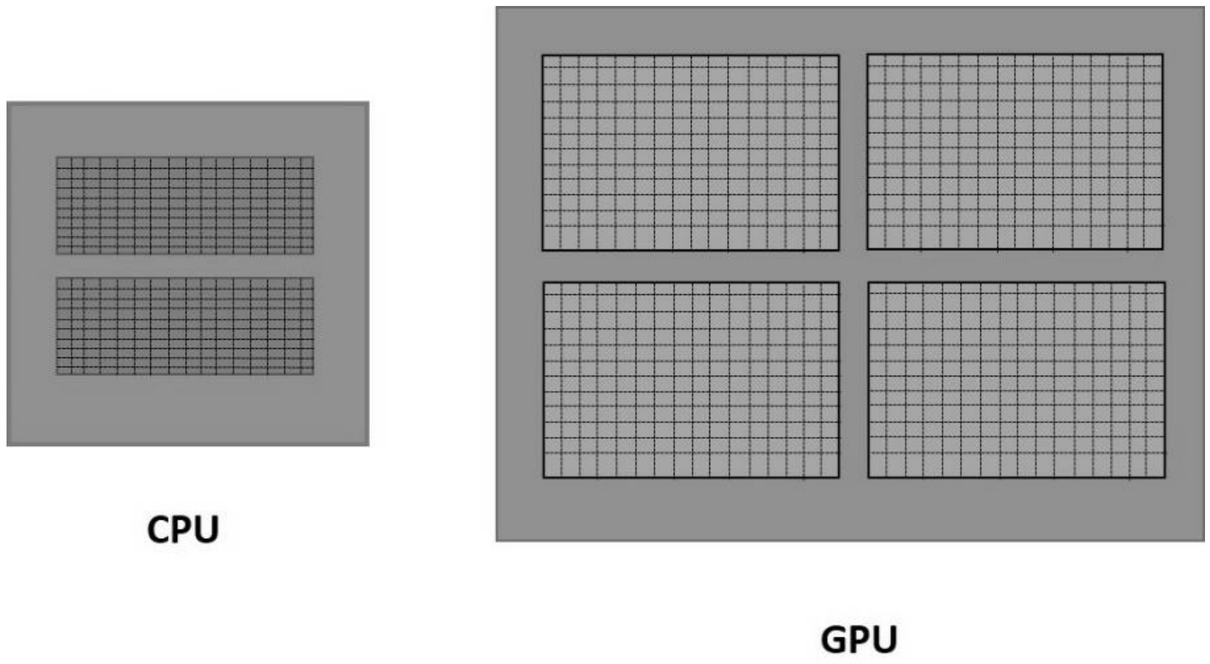
Rendering can be done locally or remotely. If the image to be rendered is way too complex, then rendering is done remotely on a dedicated server having enough of hardware resources required to render complex scenes. It is also called as **server-based rendering**. Rendering can also be done locally by the CPU. It is called as **client-based rendering**.

WebGL follows a client-based rendering approach to render 3D scenes. All the processing required to obtain an image is performed locally using the client's graphics hardware.

GPU

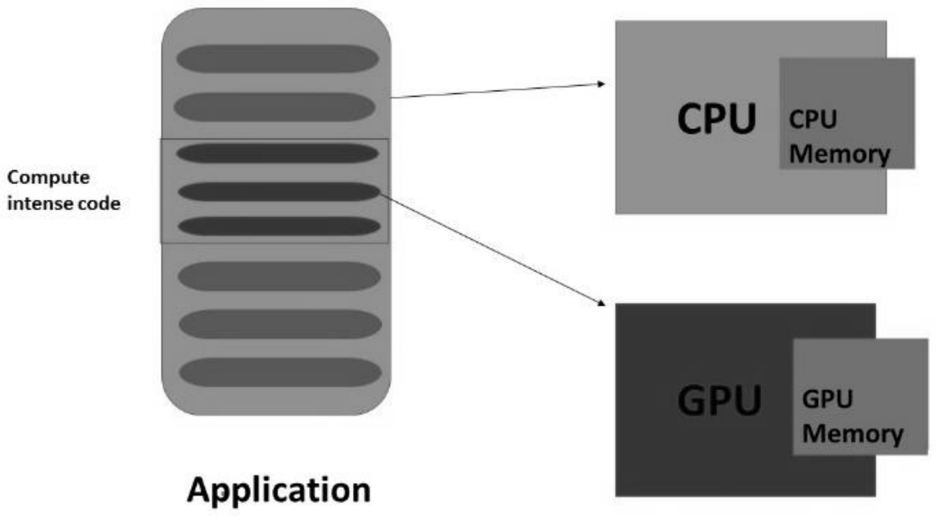
According to NVIDIA, a GPU is "a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines capable of processing a minimum of 10 million polygons per second."

Unlike multi-core processors with a few cores optimized for sequential processing, a GPU consists of thousands of smaller cores that process parallel workloads efficiently. Therefore, the GPU accelerates the creation of images in a frame buffer (a portion of ram which contains a complete frame data) intended for output to a display.



GPU Accelerated Computing

In GPU accelerated computing, the application is loaded into the CPU. Whenever it encounters a **compute-intensive** portion of the code, then that portion of code will be loaded and run on the GPU. It gives the system the ability to process graphics in an efficient way.



GPU will have a separate memory and it runs multiple copies of a small portion of the code at a time. The GPU processes all the data which is in its local memory, not the central memory. Therefore, the data that is needed to be processed by the GPU should be loaded/copied to the GPU memory and then be processed.

In the systems having the above architecture, the communication overhead between the CPU and GPU should be reduced to achieve faster processing of 3D programs. For this, we have to copy all the data and keep it on the GPU, instead of communicating with the GPU repeatedly.

Browsers Supported

The following tables show a list of browsers that support WebGL:

Web Browsers

Browser Name	Version	Support
Internet Explorer	11 and above	Complete support

Google Chrome	39 and above	Complete support
Safari	8	Complete support
Firefox	36 and above	Partial support
Opera	27 and above	Partial support

Mobile Browsers

Browser Name	Version	Support
Chrome for Android	42	Partial support
Android browser	40	Partial support
iOS Safari	8.3	Complete support
Opera Mini	8	Does not support
Blackberry Browser	10	Complete support
IE mobile	10	Partial support

Advantages of WebGL

Here are the advantages of using WebGL:

- **JavaScript programming –** WebGL applications are written in JavaScript. Using these applications, you can directly interact with other elements of the HTML Document. You can also use other JavaScript libraries (e.g. JQuery) and HTML technologies to enrich the WebGL application.

- **Increasing support with mobile browsers –** WebGL also supports Mobile browsers such as iOS safari, Android Browser, and Chrome for Android.

- **Open source –** WebGL is an open source. You can access the source code of the library and understand how it works and how it was developed.

- **No need for compilation –** JavaScript is a half-programming and half-HTML component. To execute this script, there is no need to compile the file. Instead, you can directly open the file using any of the browsers and check the result. Since WebGL applications are developed using JavaScript, there is no need to compile WebGL applications as well.

- **Automatic memory management –** JavaScript supports automatic memory management. There is no need for manual allocation of memory. WebGL inherits this feature of JavaScript.

- **Easy to set up –** Since WebGL is integrated within HTML 5, there is no need for additional set up. To write a WebGL application, all that you need is a text editor and a web browser.

Environment Setup

There is no need to set a different environment for WebGL. The browsers supporting WebGL have their own in-built setup for WebGL.

2. WEBGL– HTML-5 Canvas Overview

To create graphical applications on the web, HTML-5 provides a rich set of features such as 2D Canvas, WebGL, SVG, 3D CSS transforms, and SMIL. To write WebGL applications, we use the existing canvas element of HTML-5. This chapter provides an overview of the HTML-5 2D canvas element.

HTML-5 2D Canvas

HTML-5 **<canvas**> provides an easy and powerful option to draw graphics using JavaScript. It can be used to draw graphs, make photo compositions, or do simple (and not so simple) animations.

Here is a simple **<canvas>** element having only two specific attributes **width** and **height** plus all the core HTML-5 attributes like id, name, and class.

Syntax

The syntax of HTML canvas tag is given below. You have to mention the name of the canvas inside double quotations (" ").

```
<canvas id="mycanvas" width="100" height="100"></canvas>
```

Canvas Attributes

The canvas tag has three attributes namely, id, width, and height.

- **Id:** Id represents the identifier of the canvas element in the *Document Object Model (DOM).*
- **Width:** Width represents the width of the canvas.
- **Height:** Height represents the height of the canvas.

These attributes determine the size of the canvas. If a programmer is not specifying them under the canvas tag, then browsers such as Firefox, Chrome, and Web Kit, by default, provide a canvas element of size 300 × 150.

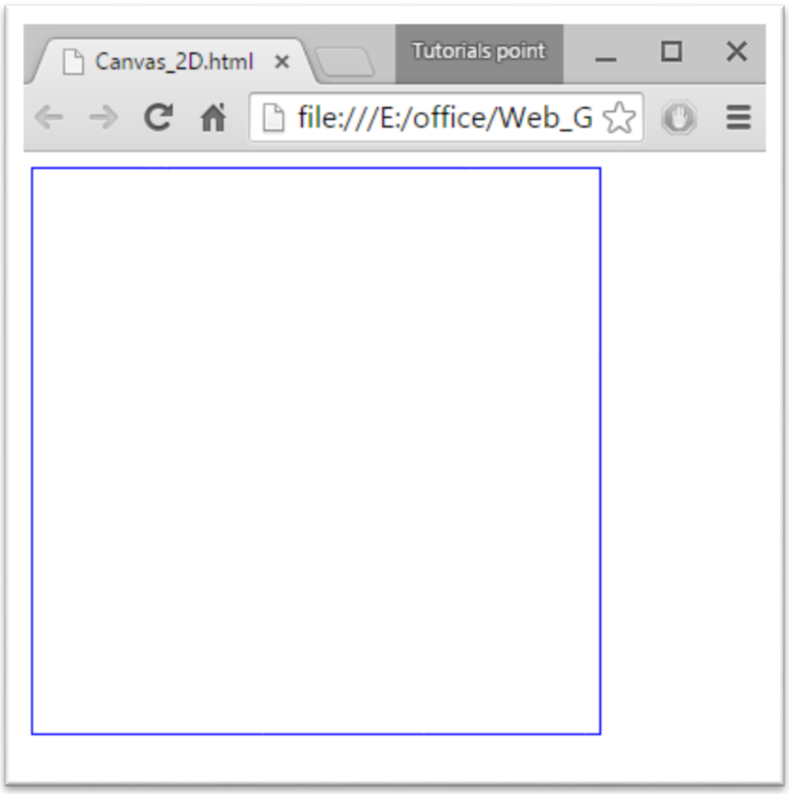
Example – Create a Canvas

The following code shows how to create a canvas. We have used CSS to give a colored border to the canvas.

```
<html>
   <head>
      <style>
         #my canvas{border:1px solid red;}
      </style>
   </head>
   <body>
      <canvas id="mycanvas" width="100" height="100"></canvas>
   </body>
</html>
```

Output

On executing, the above code will produce the following output:



The Rendering Context

The <canvas> is initially blank. To display something on the canvas element, we have to use a scripting language. This scripting language should access the rendering context and draw on it.

The canvas element has a DOM method called **getContext()**, which is used to obtain the rendering context and its drawing functions. This method takes one parameter, the type of context **2d**.

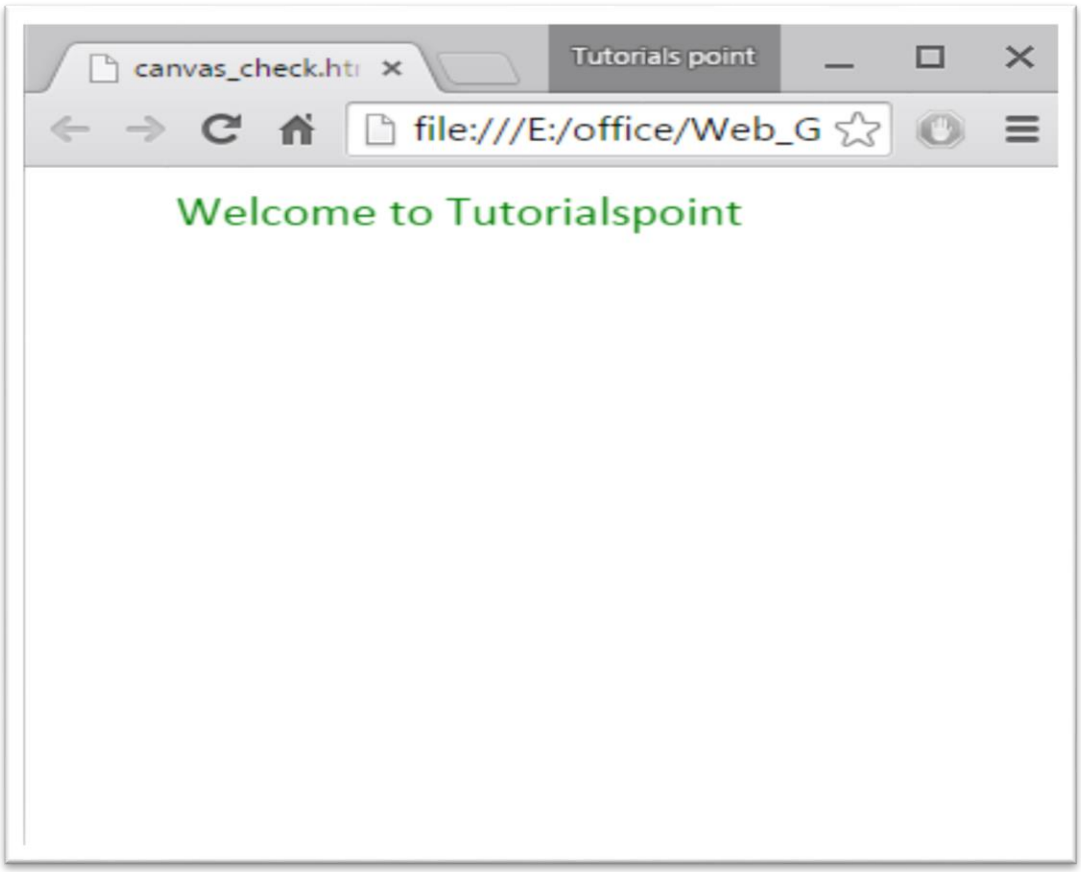
The following code is to be written to get the required context. You can write this script inside the body tag as shown below.

```
<!DOCTYPE HTML>
<html>
   <body>
      <canvas id="mycanvas" width="600" height="200"></canvas>
      <script>
```

```
            var canvas = document.getElementById('mycanvas');
            var context = canvas.getContext('2d');
            context.font = '20pt Calibri';
            context.fillStyle = 'green';
            context.fillText('Welcome to Tutorialspoint', 70, 70);
         </script>
    </body>
</html>
```

Output

On executing, the above code will produce the following output:



Welcome to Tutorialspoint

For more example on HTML-5 2D Canvas, check out the following link http://www.tutorialspoint.com/html5/html5_canvas.htm

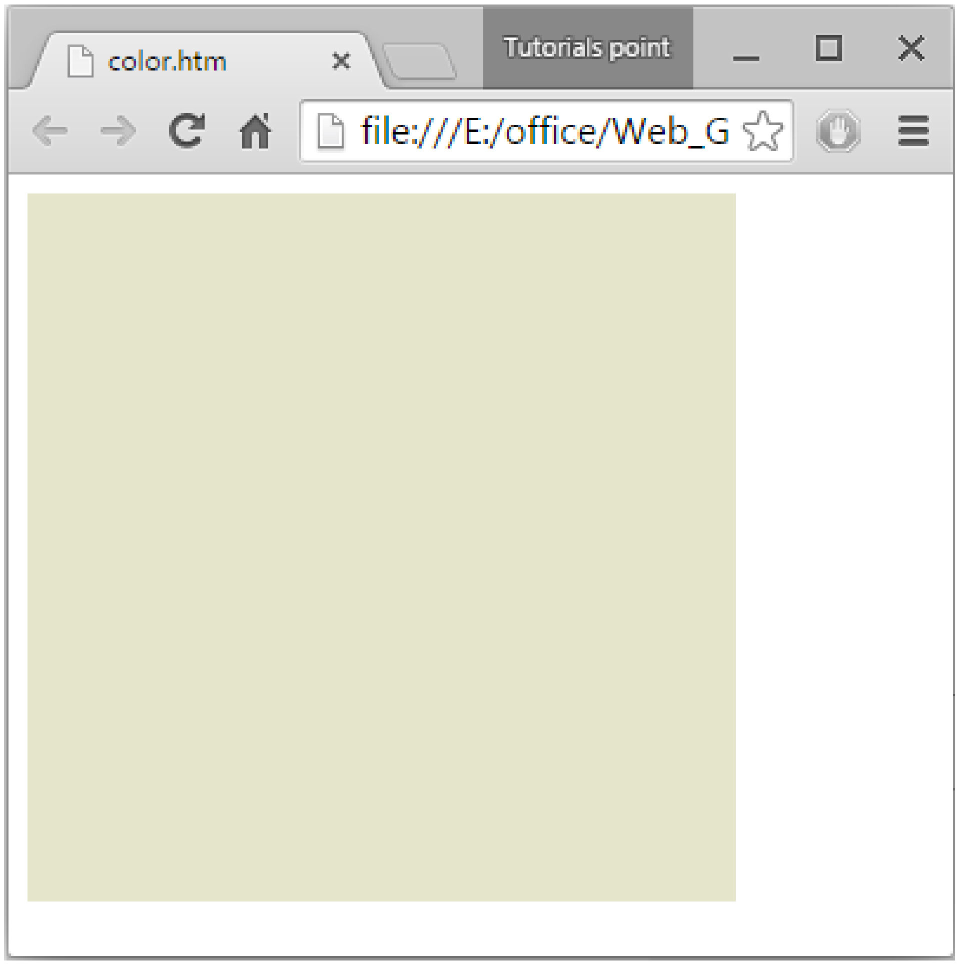
WebGL Context

The canvas tag provided by HTML-5 is also used to write WebGL applications. To create a WebGL rendering context on the canvas element, you should pass the string **experimental-webgl**, instead of **2d** to the **canvas.getContext()** method. Some browsers support only 'webgl'.

```
<!DOCTYPE html>
<html>
     <canvas id='my_canvas'></canvas>
     <script>
            var canvas = document.getElementById('my_canvas');
            var gl = canvas.getContext('experimental-webgl');
            gl.clearColor(0.9,0.9,0.8,1);
            gl.clear(gl.COLOR_BUFFER_BIT);
     </script>
</html>
```

Output

On executing, the above code will produce the following output:


color.htm
Tutorials point
file:///E:/office/Web_G

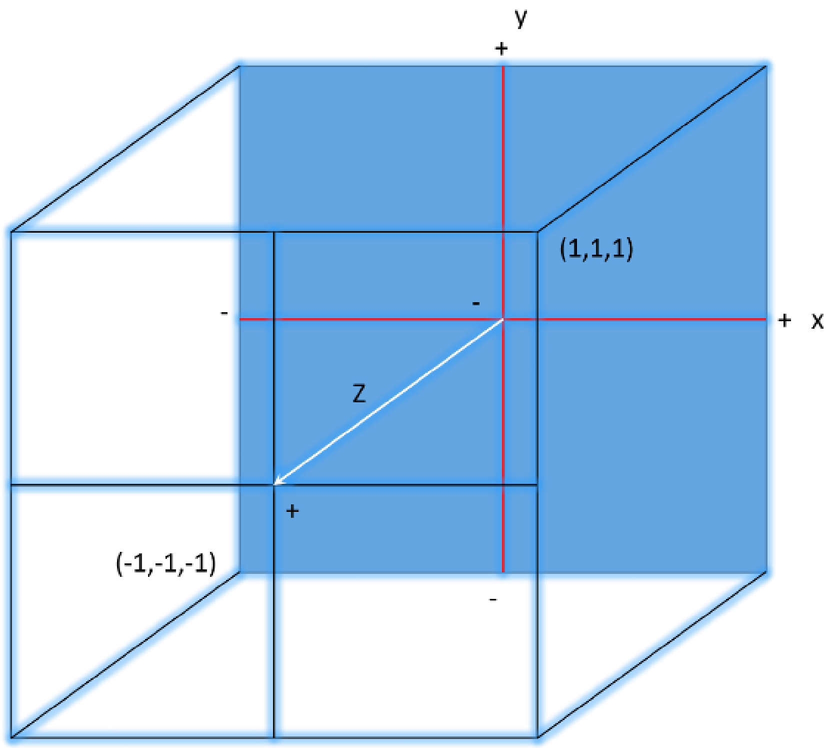
3. WebGL – Basics

WebGL is mostly a low-level rasterization API rather than a 3D API. To draw an image using WebGL, you have to pass a vector representing the image. It then converts the given vector into pixel format using OpenGL SL and displays the image on the screen. Writing a WebGL application involves a set of steps which we would be explaining in this chapter.

WebGL – Coordinate System

Just like any other 3D system, you will have xyz axes in WebGL, where the **z** axis signifies **depth**. The coordinates in WebGL are restricted to (1,1,1) and (-1,-1,-1). It means – if you consider the screen projecting WebGL graphics as a cube, then one corner of the cube will be (1,1,1) and the opposite corner will be (-1,-1,-1). WebGL won't display anything that is drawn beyond these boundaries.

The following diagram depicts the WebGL coordinate system. The z-axis signifies depth. A positive value of z indicates that the object is near the screen/viewer, whereas a negative value of z indicates that the object is away from the screen. Likewise, a positive value of x indicates that the object is to the right side of the screen and a negative value indicates the object is to the left side. Similarly, positive and negative values of y indicate whether the object is at the top or at the bottom portion of the screen.



WebGL Graphics

After getting the WebGL context of the canvas object, you can start drawing graphical elements using WebGL API in JavaScript.

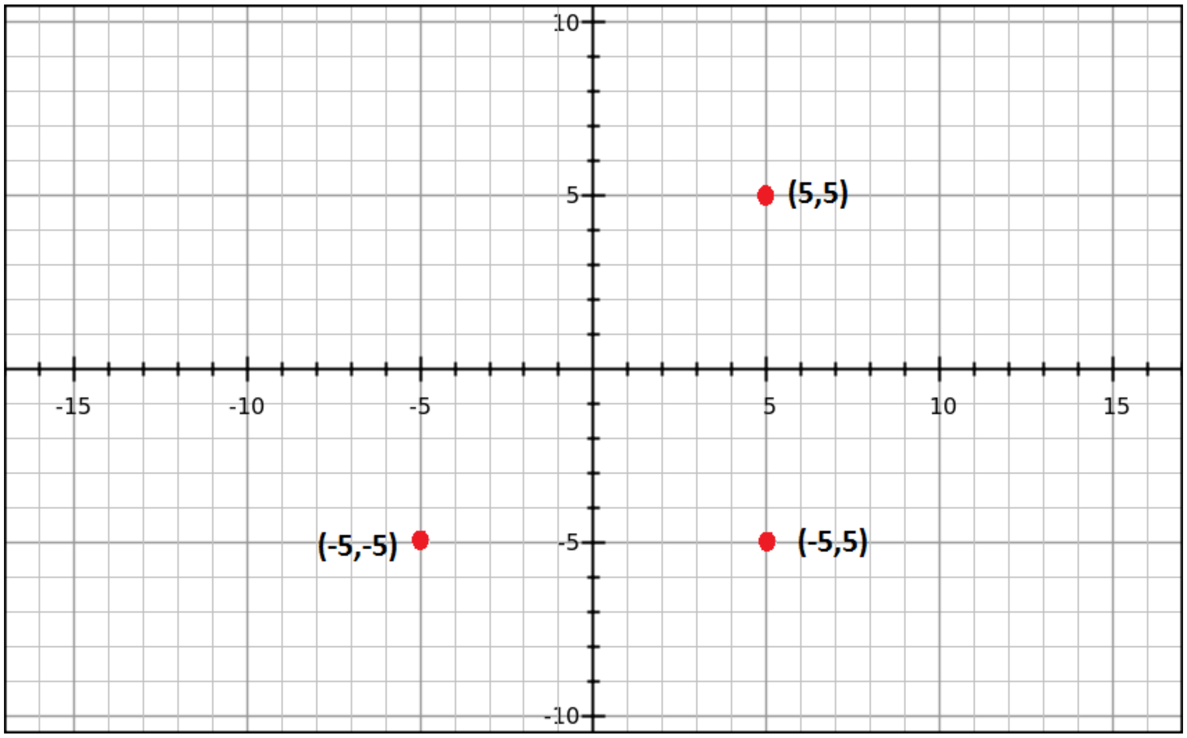
Here are some fundamental terms you need to know before starting with WebGL.

Vertices

Generally, to draw objects such as a polygon, we mark the points on the plane and join them to form a desired polygon. A **vertex** is a point which defines the conjunction of the edges of a 3D object. It is represented by three floating point values each representing x,y,z axes respectively.

Example

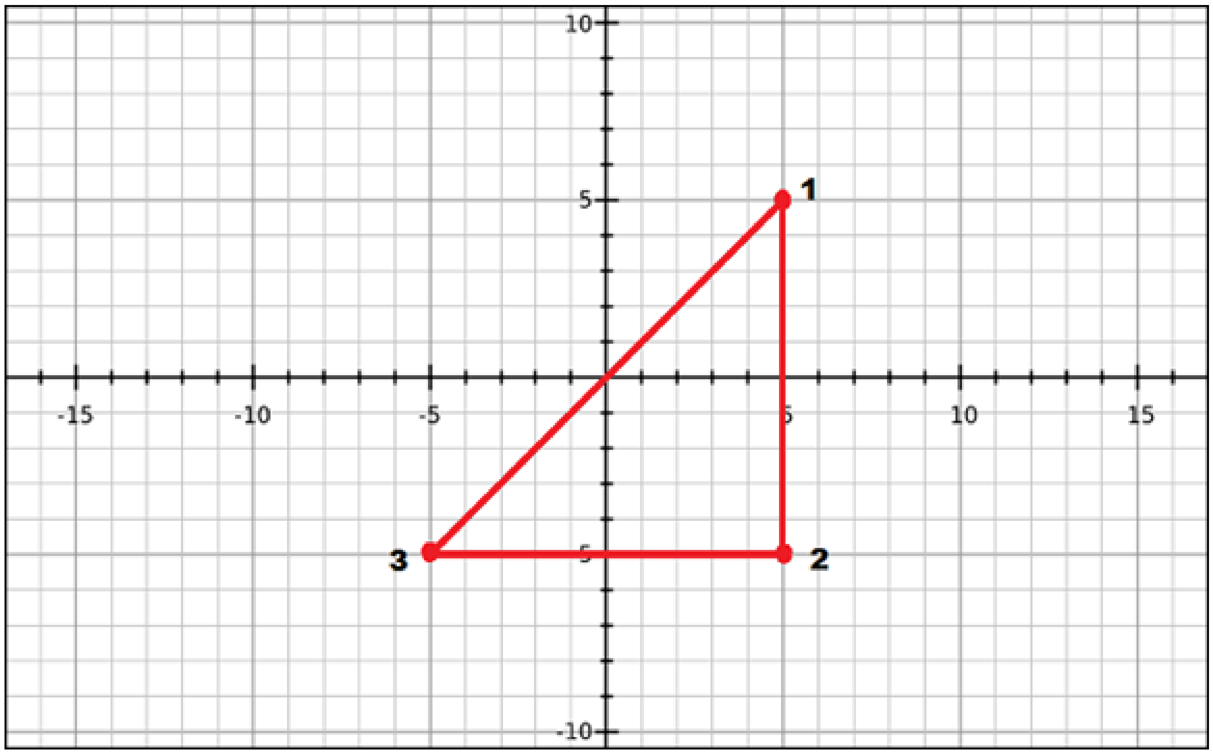
In the following example, we are drawing a triangle with the following vertices: (5,5), (5,-5), (-5,5).



Note: We have to store these vertices manually using JavaScript arrays and pass them to the WebGL rendering pipeline using vertex buffer.

Indices

Generally vertices are labeled using numerical or alphabets. In WebGL, numerical values are used to identify the vertices. These numerical values are known as indices. These indices are used to draw meshes in WebGL.



Note: Just like vertices, we store the indices using JavaScript arrays and pass them to WebGL rendering pipeline using index buffer.

Arrays

Unlike OpenGL and JoGL, there are no predefined methods in WebGL to render the vertices directly. We have to store them manually using JavaScript arrays.

Example

```
var vertices = [ 0.5, 0.5,  0.1,-0.5,  0.5,-0.5]
```

Buffers

Buffers are the memory areas of WebGL that hold the data. There are various buffers namely, drawing buffer, frame buffer, vetex buffer, and index buffer. The **vertex buffer** and **index buffer** are used to describe and process the geometry of the model.

Vertex buffer objects store data about the vertices, while Index buffer objects store data about the indices. After storing the vertices into arrays, we pass them to WegGL graphics pipeline using these Buffer objects.

Frame buffer is a portion of graphics memory that hold the scene data. This buffer contains details such as width and height of the surface (in pixels), color of each pixel, depth and stencil buffers.

Mesh

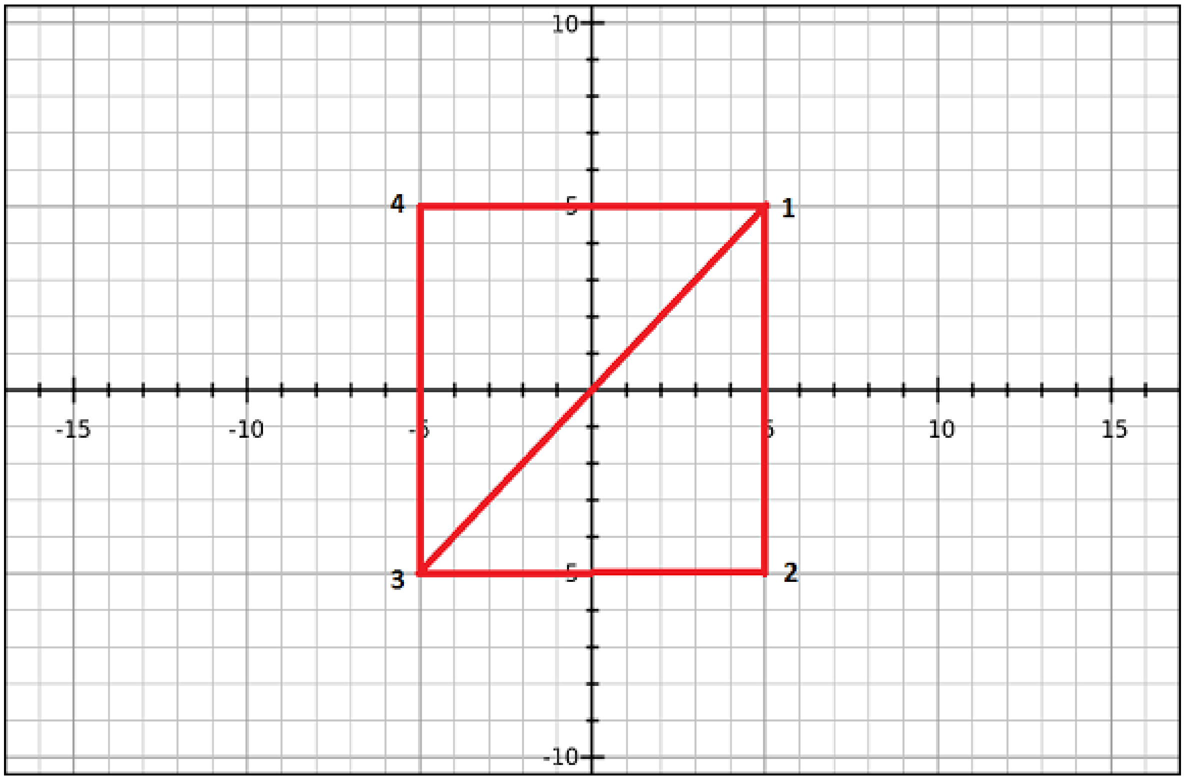
To draw 2D or 3D objects, the WebGL API provides two methods namely, **drawArrays()** and **drawElements()**. These two methods accept a parameter called **mode** using which you can select the object you want to draw, but the options provided by this field are restricted to points, lines, and triangles.

To draw a 3D object using these two methods, we have to construct one or more primitive polygons using points, lines, or triangles. Thereafter, using those primitive polygons, we can form a mesh.

A 3D object drawn using primitive polygons is called a **mesh**. WebGL offers several ways to draw 3D graphical objects, however users normally prefer to draw a mesh.

Example

In the following example, you can observe that we have drawn a square using two triangles: {1, 2, 3} and {4, 1, 3}.



Shader Programs

We normally use triangles to construct meshes. Since WebGL uses GPU accelerated computing, the information about these triangles should be transferred from CPU to GPU which takes a lot of communication overhead.

WebGL provides a solution to reduce the communication overhead. Since it uses ES SL (Embedded System Shader Language) that runs on GPU, we write all the required programs to draw graphical elements on the client system using **shader programs** (the programs which we write using OpenGL ES Shading Language).

These shaders are the programs for GPU; the language used to write shader programs is GLSL. In these shaders, we define exactly how the vertices, transforms, materials, lights, and the camera interact with one another to create a particular image.

In short, it is a snippet that implements algorithms to get the pixels for a mesh. We will discuss more about shaders in later chapters. There are two types of shaders: Vertex Shader and Fragment Shader.

Vertex Shader

Vertext shader is the program code called on every vertex. It is used to transform (move) the geometry (ex: triangle) from one place to another. It handles the data of each vertex (per—vertex data) such as vertex coordinates, normals, colors, and texture coordinates.

In the **ES GL** code of vertex shader, programmers have to define attributes to handle the data. These attributes point to a Vertex Buffer Object written in JavaScript.

The following tasks can be performed using vertex shaders:

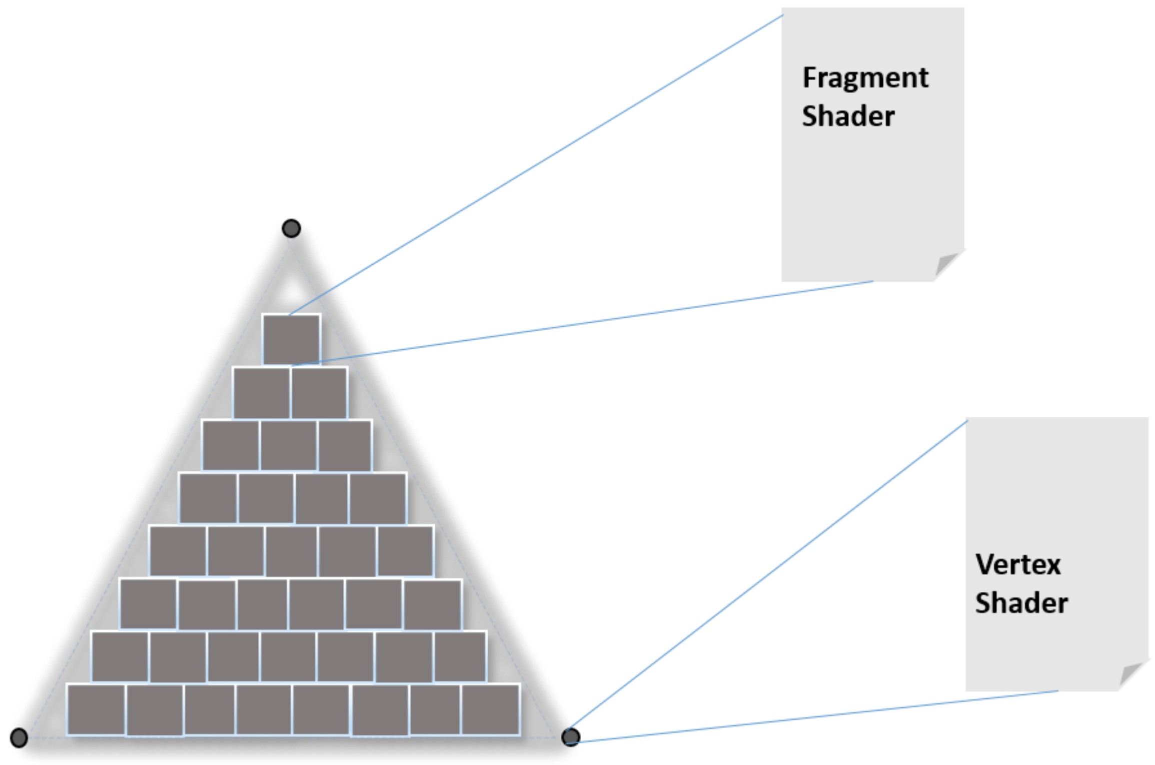
- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

Fragment Shader (Pixel Shader)

A mesh is formed by multiple triangles, and the surface of each of the triangles is known as a **fragment**. Fragment shader is the code that runs on every pixel on each fragment. It is written to calculate and fill the color on *individual pixels*.

The following tasks can be performed using Fragment shaders:

- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color sum



OpenGL ES SL Variables

The full form of **OpenGL ES SL** is OpenGL Embedded System Shading Language. To handle the data in the shader programs, ES SL provides three types of variables. They are as follows:

- Attributes: These are the variables that hold the input values of the vertex shader program. They point to the vertex buffer object which contains per-vertex data. Each time the vertex shader is invoked, the values of these attributes vary.

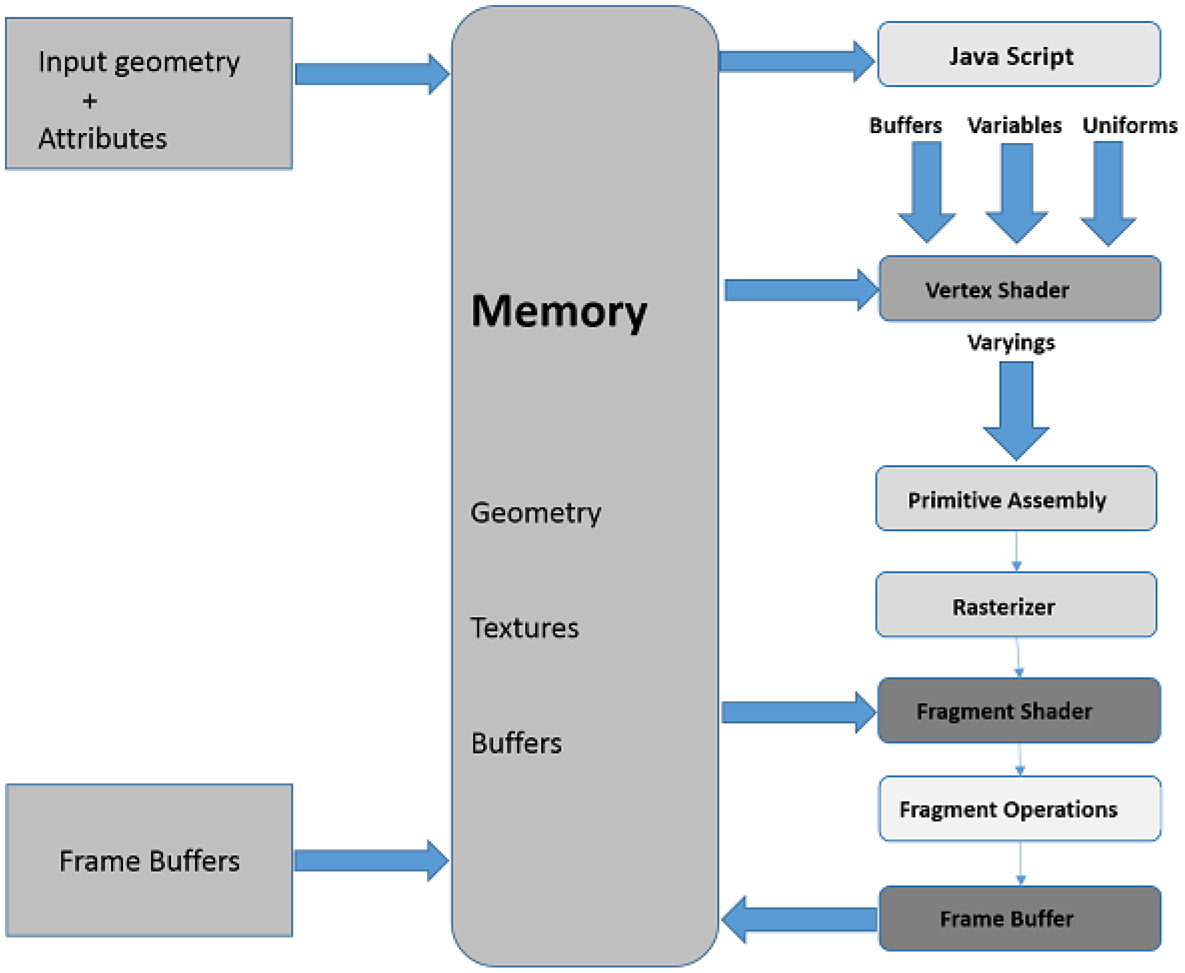
- Uniforms: These are the variables that hold the input data that is common for both vertex and fragment shaders, such as light position, texture coordinates, and color.

- Varyings: These are the variables that are used to pass the data from the vertex shader to the fragment shader.

With this much basics, we will now move on to discuss the Graphics Pipeline.

4. WebGL – Graphics Pipeline

To render 3D graphics, we have to follow a sequence of steps. These steps are known as **graphics pipeline** or **rendering pipeline**. The following diagram depicts WebGL graphics pipeline.



In the following sections, we will discuss one by one the role of each step in the pipeline.

JavaScript

While developing WebGL applications, we write Shader language code to communicate with the GPU. JavaScript is used to write the control code of the program, which includes the following actions:

- **Initialize WebGL:** JavaScript is used to initialize the WebGL context.
- **Create arrays:** We create JavaScript arrays to hold the data of the geometry.
- **Buffer objects:** We create buffer objects (vertex and index) by passing the arrays as parameters.
- **Shaders:** We create, compile, and link the shaders using JavaScript.
- **Attributes:** We can create attributes, enable them, and associate them with buffer objects using JavaScript.
- **Uniforms:** We can also associate the uniforms using JavaScript.
- **Transformation matrix:** Using JavaScript, we can create transformation matrix.

Initially we create the data for the required geometry and pass them to the shaders in the form of buffers. The attribute variable of the shader language points to the buffer objects, which are passed as inputs to the vertex shader.

Vertex Shader

When we start the rendering process by invoking the methods **drawElements()** and **drawArray()**, the vertex shader is executed for each vertex provided in the vertex buffer object. It calculates the position of each vertex of a primitive polygon and stores it in the varying **gl_position**. It also calculates the other attributes such as **color**, **texture coordinates**, and **vertices** that are normally associated with a vertex.

Primitive Assembly

After calculating the position and other details of each vertex, the next phase is the **primitive assembly stage**. Here the triangles are assembled and passed to the rasterizer.

Rasterization

In the rasterization step, the pixels in the final image of the primitive are determined. It has two steps:

- **Culling**: Initially the orientation (is it front or back facing?) of the polygon is determined. All those triangles with improper orientation that are outside the view area are discarded. This process is called culling.
- **Clipping**: If a triangle is partly outside the view area, then the part outside the view area is removed. This process is known as clipping.

Fragment Shader

The fragment shader gets

- data from the vertex shader in varying variables,
- primitives from the rasterization stage, and then
- calculates the color values for each pixel between the vertices.

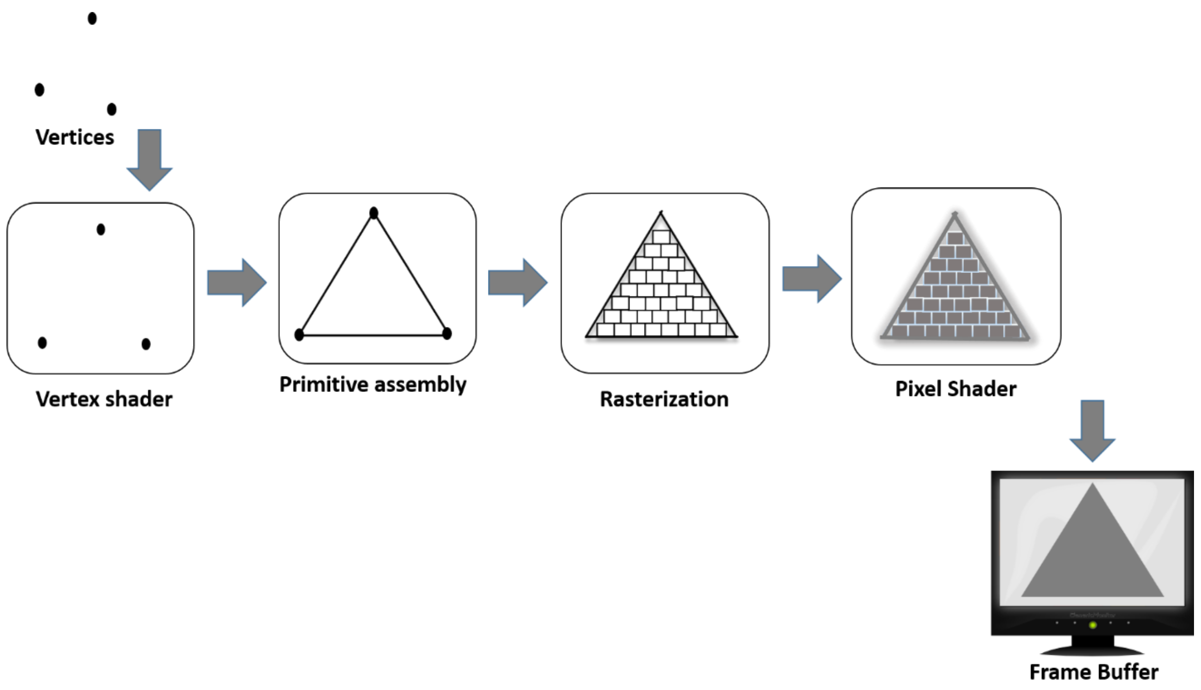
The fragment shader stores the color values of every pixel in each fragment. These color values can be accessed during fragment operations, which we are going to discuss next.

Fragment Operations

Fragment operations are carried out after determining the color of each pixel in the primitive. Fragment operations include the following:

- Depth
- Color buffer blend
- Dithering

Once all the fragments are processed, a 2D image is formed and displayed on the screen. The **frame buffer** is the final destination of the rendering pipeline.



Frame Buffer

Frame buffer is a portion of graphics memory that hold the scene data. This buffer contains details such as width and height of the surface (in pixels), color of each pixel, and depth and stencil buffers.