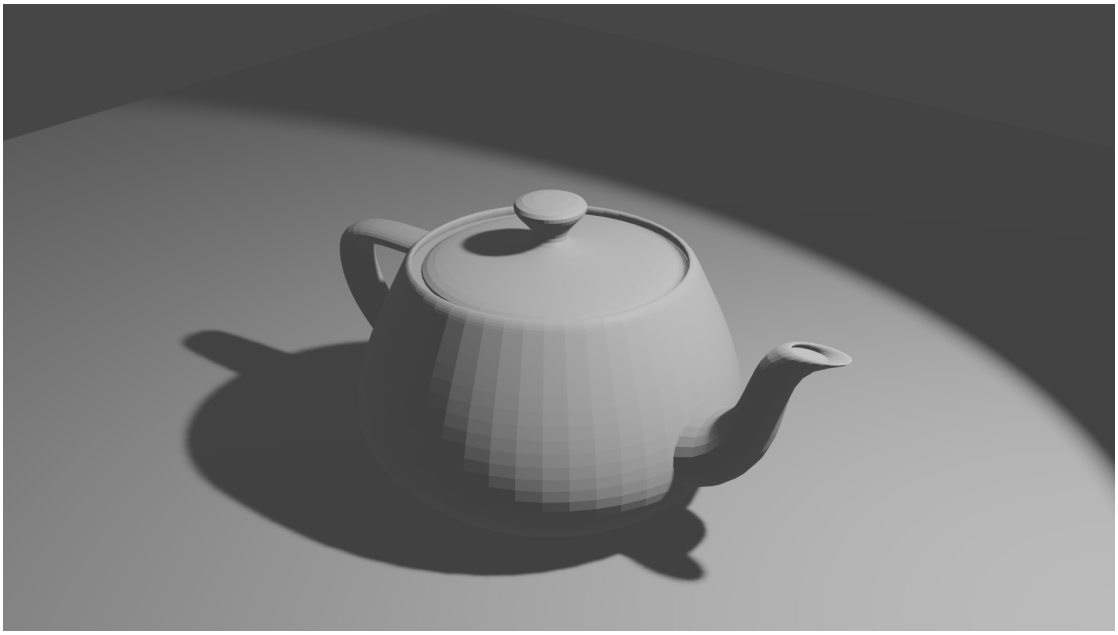
CS324 Computer Graphics
WebGL Labs

© Saad Bashir, Rossella Suma,
Abhir Bhalerao & Shan Raza

October, 2023



Version 3.1
2023-10-16

Contents

1 Lab1: Introduction

What is WebGL?

WebGL (Web-based Graphics Language) is a new and exciting programming language derived from OpenGL (Figure 1.1) that lets you create wonderful and powerful graphics in 2D and 3D within a web browser. It uses the JavaScript API for interactions with GPU (Graphics Processing Unit) and CPU (Central Processing Unit) and servers as a binder between them for drawing graphics. Released for the first time in March 2011, WebGL (Web Graphics Library) is a royalty-free Javascript cross-platform API that allows the use of the GPU in a web browser window to render real-time 3D computer graphics. WebGL is widely accepted for standard graphics application development because its easy to learn and contains all the capabilities of previous versions. Let's learn some basics of OpenGL first as WebGL is derived from OpenGL.

What is OpenGL?

OpenGL stands for Open Graphics Library and is a device independent API which allows the writing of graphics applications by accessing the functionality of graphics hardware, such as GPUs. It is designed to be efficient and models hardware functionality. Because of this, it sometimes feels very low-level. Surprisingly, it consists of only about 200 functions.

OpenGL is based on a stable standard and includes abilities to access the programmable elements of GPUs (shaders) by the GLSL (GL Shading Language). The functions in OpenGL (e.g., `glColor3f()` which sets the current drawing colour) can be divided into following groups:

- Primitive functions (**primitives** in computer graphics are the graphics objects). There are functions to specify geometric primitives such as points, line and polygons, but also bit maps (texture maps). You will learn more about primitive(s) drawing in the lectures.

- Attribute functions which control the appearance of primitives. These are functions to set the colours and widths of lines for example. In 3D, we can set the surface properties of polygons.

- Viewing functions specify the **camera(s)** from which the graphics scene is viewed. Cameras have positions and directions and specify to OpenGL precisely how vertices are projected onto the output window. This will be discussed in detail when we cover transformations and viewing topic in lectures.

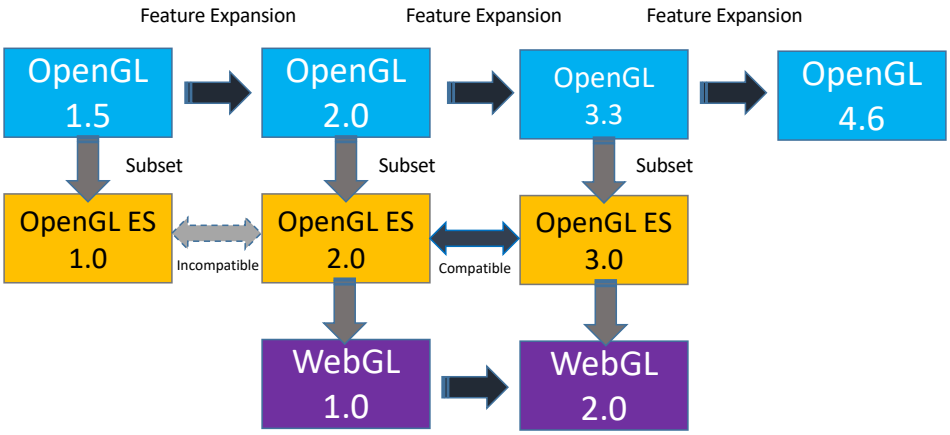


Figure 1.1: Relationship between different versions of OpenGL, OpenGL ES and WebGL.[1]

- Control functions which enable and disable certain OpenGL features, such as hidden line removal or blending.
- Query functions which allow OpenGL state variables to be read or the capabilities/conformance of the particular OpenGL implementation you are using to be determined.
- Input and Window functions. These are the functions which are not part of OpenGL but are essential to allow interactive use of OpenGL, e.g., take mouse and keyboard input from an OpenGL window and position and resize the window. These are in an auxiliary library called GLUT

Figure 1.1 shows the relationship between WebGL and various versions of OpenGL. In 2003 the first version of OpenGL ES 1.0 was released (the "ES" stands for "embedded systems"). This was a simpler version of OpenGL as there was a demand of handling graphics on devices with less computational power. This version was based on OpenGL 1.5.

[1] http://learnwebgl.brown37.net/the_big_picture/webgl_history.html

Why WebGL?

We will use WebGL in these labs. Although WebGL has less functionalities when compared to the latest version of OpenGL, it shares all the basic properties. WebGL is fully shader based, whereas OpenGL requires GLSL support for shaders. The disadvantage of OpenGL is that it does not have a definitive way to interface to an operating system's user interface. The main advantages of WebGL are its cross-browser and cross-platform compatibility, its integration with HTML content (the canvas can occupy part or the whole of the web page), the availability of the standard HTML event handling mechanisms and the automatic implementation of the *double buffering*. Our next step will be to explore the application side of graphics programming. We use the WebGL API, which is powerful, is supported on modern browsers, and has a distinct architecture that will allow us to use it to understand how computer graphics works, from an application program to a final image on a display.

WebGL - Installation?

WebGL is a web-based 3D Graphics API, so no installation is needed! :) However, to test your programs you must run a local web server on your computer. Many features of a WebGL program will not execute in your browser if files are stored locally, therefore we need to load the files from a (local) web server. More on this in the following labs.

WebGL Pipeline

We can represent the WebGL graphic pipeline as a state machine, each of the different stages will preform specific actions on the WebGL application program input and produce a visible output (i.e., pixel colours that are ready to be visualised onto the screen). The input to the state-machine comes from the application program where it passes through different alterations to produce a visible output. A simplified state-machine is shown in Figure 1.2. Input data to the pipeline, includes model vertices (i.e., location, direction and color) as geometric primitive or an image as raster/texture primitive. These primitives pass through the geometric pipeline where they are subjected to a series of geometric operations that determine whether a primitive needs to be clipped, where on the display it appears if it is visible, and the rasterisation of the primitive into pixels in the framebuffer. Because geometric primitives exist in a two or three-dimensional space, they can be manipulated by operations such as rotation and translation.

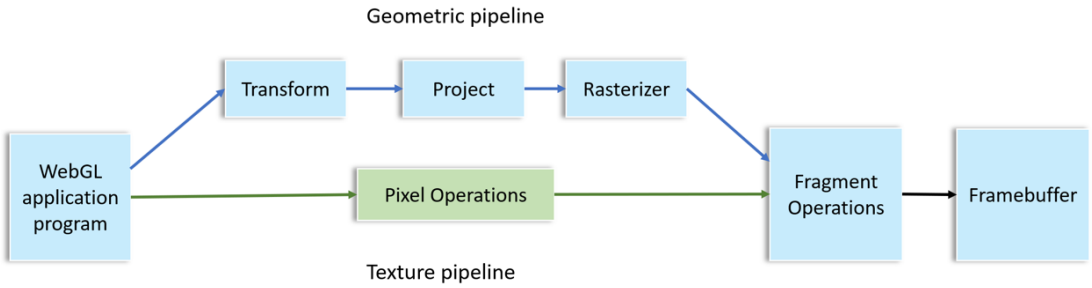


Figure 1.2: Simplified WebGL pipeline. [Angel 7th Edition]

WebGL Application Anatomy

The typical components of a webpage that contains 3D computer graphics generated by a WebGL program are:

1. An HTML (Hypertext Markup Language) description of the web page.
2. A CSS (Cascading Style Sheet) describing how each element of the HTML description is formatted.
3. An HTML *canvas* element. This will indicate the rectangular area in which 3D computer graphics will be rendered.
4. Graphical data that defines the 3D objects to be rendered (generally, but not exclusively contained in an .obj file).
5. One or more JavaScript programs that load your graphical data, configure your graphical data, render your graphical data, and code that responds to user events.
6. The OpenGL Shader programs that perform the graphical rendering.

All of these basic parts initially reside on a web server. When a user (a client) requests a web page that contains 3D graphics, the data described above has to be transferred to the client's computer, stored in correct places, and then processed by appropriate processors (i.e., CPU(s) and GPU(s)).

The first two components are html and CSS description. It is out of the scope of this course to focus on HTML or CSS. Our interaction with these components will be reduced to a minimum. However, in the following subsection there will be a very short introduction to them. For those, with some previous web-development knowledge, please feel free to skip to the next section.

HTML and CSS files

The HTML (hypertext markup language) and CSS (cascading style sheets) files are used to describe web pages. The HTML file is used to describe the content of a webpage, while the CSS describes how the elements contained in the webpage will be represented. As discussed before we will be focusing mainly on the WebGL part while giving a short introduction to HTML and CSS.

HTML stands for Hyper Text Markup Language and is the markup language used to indicate to a web-browser how to display webpages' content. It is characterised by a series of elements. Each element is defined by a start tag, some content, and an end tag e.g., the `<html>` tag signals the beginning of the document and the `</html>` signals its end.

Here follows a minimal skeleton of an HTML file that we will need for our labs, please look at the comments (e.g., `<!-- some text -->`) for further explanation of the different elements.

```
<!doctype html>
<!--This is a comment. Comments are not displayed in the
    browser-->
<!-- Example HTML file -->
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="description" content="WebGL tutorials -
       an example">
    <meta name="author" content="Rossella Suma">
    <title>WebGL Getting Started Page</title>
    <!--Here is how to link a .css file to control the
       appearance of your page-->
    <link rel="stylesheet" href="style.css" />
</head>

<body>
    <!--This tag indicates a heading -->
    <h1>This is a heading</h1>
    <!--This tag indicates a paragraph -->
    <p>An example of a paragraph.</p>

    <!--The canvas window for rendering 3D graphics -->
    <canvas id="MY_canvas" class="canvas3D">
        Please use a browser that supports "canvas"
```

```
    </canvas>
    <br>

    <!-- Load the JavaScript libraries -
    note the path might change depending on the location
        of this folder
  -->
    <script src="/lib/learn_webgl.js"></script>

    <!-- Load the JavaScript data files for the WebGL
       rendering -->
    <script src="main.js"></script>

</body>

</html>
```

All HTML documents must start with a declaration of the document type: `<!doctype html>`. The visible part of the HTML document is between `<body>` and `</body>`. In the `<head>` element we have indicated the page title and provide the link to the .css file.

The .css file will specify the appearance of the elements in the webpage. In the following example, the page background, the heading and the paragraph font colour is specified respectively.

```
body {
  background-color: powderblue;
}
h1 {
  color: blue;
}
p {
  color: red;
}
}
```

Try this: *Change the colour value for h1, p and background colour in the style.css file and check if it is reflected when you reload the html file.*

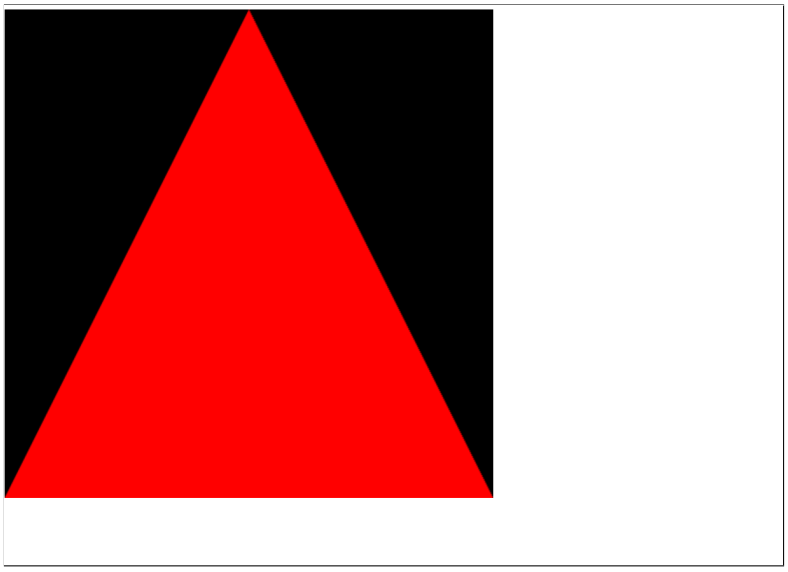


Figure 1.3: Output of Lab-1 triangle.html a red Triangle using WebGL programming.

1.1 Introduction to WebGL Programming

Download the Lab1_WebGL.zip from the `/modules/cs324` folder or from Moodle under the *Practical Sessions/Labs* section.

A good IDE is an essential tool for developing software. However, you can use any text editor to modify the files. In the Labs, you can use Visual Studio Code along with Chrome - or any other browser that allows you to be in developer mode - to visualise and debug your code. Open the developer mode with the shortcut `Ctrl+Shift+I` or Press `Command+Option+I` (Mac).

Unzip the folder and move to the subfolder. You can verify whether everything is working by opening the `triangle.html` file. You can click on the file, or you can just type in Linux terminal:

```
> chrome triangle.html
```

You should be able to see something like what is showed in Figure 1.3 in your web browser. Opening the Developer Mode (`Ctrl+Shift+I`) of your browser will reveal the folder structure and the two main files responsible for creating the graphics. In the following sections, we will describe step by step how this is done.

1.1.1 Languages and Tools

The following are the steps that any WebGL application must perform in order to be able to render on screen no matter how complex or simple the scene is:

- Set up a canvas to render onto
- Generate data application
- Create shader programs
- Create buffer objects and load data into them
- Connect every bit together: data location with shader variables
- Render

We will have a look at each one of them today.

1. The folder called `Common` that contains some useful functionalities - we will explore these later on;
2. an html file called `triangle.html`;
3. a JavaSctipt file called `triangle.js`.

Let's start with the `tiangle.html` file. This is our entry point.

1.1.2 The HTML file

As seen in the introductory section of the lab, the HTML file is structured by using tags. The `<script>` tag, will indicate an asynchronous process in which the JavaScript files and the shaders will be read in order. After this the browser will enter an execution phase that is event driven (e.g., it will be waiting whether the user clicks somewhere on the page).

Vertex Shader Right after the `<html>` tag we encounter the description of the vertex shader and fragment shader. In this lab, we will briefly discuss the shaders. However, we will discuss these in detail with interactivity in the next lab. We are giving an `id` to shaders, so that we can refer to them in the application file (the JavaScript file).

```
<script id="vertex-shader" type="x-shader/x-vertex">
    attribute vec4 vPosition;

void
main()
{
  gl_Position = vPosition;
}
```

This is a simplified version of a vertex shader and in this case we are assuming that the vertex passed will be written in the correct coordinate system (all coordinates values will be between -1 and 1)

Fragment Shader The output, `gl_Position`, of the vertex shader is passed onto the next stage in the graphics pipeline that converts the vertex position to a pixel location in the rendered image. This pixel location (and its associated rendering data) is passed to the next stage, the Rasteriser, that determines the pixels covered by an object. This varies based on whether the vertex is part of a point, a line, or a triangle primitive. The rasterizer will output a list of all the pixels that will need to be coloured to represent an object. Each pixel and its associated rendering data is a called fragment.

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

void
main()
{
  gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>
```

Loading JavaScript Files Finally in our `html` we find the request to the browser to load the JavaScript files.

```
<script type="text/javascript" src="Common/webgl-utils.js">
   </script>
<script type="text/javascript" src="Common/initShaders.js">
   </script>
<script type="text/javascript" src="triangle.js"></script>
```

At this stage, we will ignore the details of the first two files (although nothing stops you to have a browse!). The most important thing to know is that the `webgl-utils.js` contains utilities needed to create the WebGL canvas object, while the `initShaders.js` contains the code for reading, compiling and loading the shaders.

The third file `triangle.js` is the application file which we will explore in detail in the next paragraph. Last but not the least, we find the definition of the `canvas`, which is the area in the web page where we will be displaying our graphics. Note that here, as before, we have assigned an `id` to the canvas object, this is necessary

so that we can call this object from our JavaScript file. If there was an error in the creation of the canvas, an error message will be displayed.

```
<canvas id="gl-canvas" width="512" height="512">
  Oops ... your browser does not support the HTML5
     canvas element
</canvas>
```

One last thing to notice is that in the `<body>` tag we have specified: `<body onload="main()">` indicating the entry point for our code in the application file.

1.1.3 The application file

Now we look at the `triangle.js` file. Once the browser has entered the execution phase the main() function will be called. In this file, we find the definition of our application. Essentially `triangle.js` is a JavaScript program that draws a red triangle on the drawing area defined by the `<canvas>` element.

1 - Set up canvas: The main three steps required to draw 2D graphics on the canvas are:

1. Obtain the `<canvas>` element

   The HTML file specifies a `<canvas>` of the requested size. The request of the `canvas` element is made through using the `id` specified before (`gl-canvas` in our case).

   ```
   var canvas = document.getElementById("gl-canvas");
   ```

2. Request the rendering context from the element

   To request the rendering context we are using a function offered in the `webgl-utils.js` file.

   ```
   gl = WebGLUtils.setupWebGL(canvas);
   if (!gl) {
       alert("WebGL isn't available");
   }
   ```

   This is just a utility that instead of setting up a context manually will also check for success or failure and on failure it will present an appropriate message to the user. The most important operation performed by the `WebGLUtils.setupWebGL` is to obtain the rendering context:

   ```
   var context = canvas.getContext('webgl');
   ```

2 - Generate and draw 2D data: Draw the 2D graphics using the methods that the context supports.

Once the rendering context is obtained we can start drawing. At this stage, we are only aiming at drawing in 2D, so we will indicate only (x, y) tuples. The three 2D coordinates the vertices of our triangle: (-1,-1), (0, 1), (1,-1) are stored in a JavaScript typed array of 32-bit floating point numbers.

```
var vertices = new Float32Array([-1, -1, 0, 1, 1, -1]);
```

Finally, we get ready to draw on our canvas: we call `gl.viewport(x,y,w,h)` and `gl.clearColor()`.

```
gl.viewport(0, 0, canvas.width, canvas.height);
gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

A viewport is a rectangular area of the display window **in pixels**. It accepts 4 parameters, the first two are the lower-left coordinates of the viewport (x,y) and `w` and `h` indicate the width and height. When you first create a WebGL context, the size of the viewport will match the size of the canvas.

The `gl.clearColor()` function specifies the color values used when clearing color buffers.

Try this: *Change the values of `gl.clearColor()` to `(0.0, 0.0, 1.0, 1.0)` in your text editor, reload the webpage and see what happens.*

3 - Load shader programs: Next we will load the shaders, we declared in a program object.

We perform this task by taking the `id` of the two shaders previously declared and using the WebGL context. This is sort of a container that includes all of the shaders (you can have more than just two!). One thing that you may want to do in your programs is to set up different program objects and switch between them.

```
var program = initShaders(gl, "vertex-shader", "fragment
    -shader");
gl.useProgram(program);
```

4 - Create & bind buffer objects: At this point, we want to send our data to the GPU.

```
var bufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, bufferId);
gl.bufferData(gl.ARRAY_BUFFER,vertices, gl.STATIC_DRAW);
```

We start creating a vertex buffer object (**VBO**). A buffer object is a contiguous block of memory in the GPU that shaders can access quickly when executing on the GPU. Buffer objects store the data that shader programs need for rendering. The contents of a buffer object are always a one-dimensional array. The `gl.createBuffer()` will simply reserve an id for our buffer, while `gl.bindBuffer()` will make the buffer object the active buffer. Typically there will be many object buffers and only one of them is the "active buffer". Any command on buffer objects will be manipulating the active buffer.

Note that buffer objects reside on the GPU memory, but they are created and managed using the WebGL API from JavaScript code. The `gl.bufferData()` function will finally copy data from your JavaScript program into the GPU's buffer object. If the buffer object already contained data this will be deleted and the new data will be added.

The last parameter passed to `gl.bufferData()` is called *usage*. This suggests how a buffer object's data store will be accessed. *usage* can be broken down into two parts: first, the frequency of access (modification and usage) and second, the nature of that access. In our case, we are specifying `STATIC`, which means that the data store contents will be modified once and used many times. The other option is `DRAW`, which indicates that the data store contents are modified by the application, and used as the source for GL drawing and image specification commands.

5 - Connect all the components: The last step before rendering is to associate shader variables with our data buffer.

```
var vPosition = gl.getAttribLocation(program, "vPosition
    ");
gl.vertexAttribPointer(vPosition, // Attribute location
                       2, // Elements per attribute
                       gl.FLOAT, // Type of elements
                       false, // Normalization required
                       0, // Individual vertex size
                       0  // Offset from the beginning
                       );
gl.enableVertexAttribArray(vPosition);
```

`gl.getAttribLocation` returns the location of the attribute variable such as the vPosition variable in our vertex shader. We then describe the form of the data in the vertex array and we enable the vertex attributes that are in the shader.

6 - Render: At this point, we call the `render()` function.

The `render()` function will simply clear the frame buffer and then render

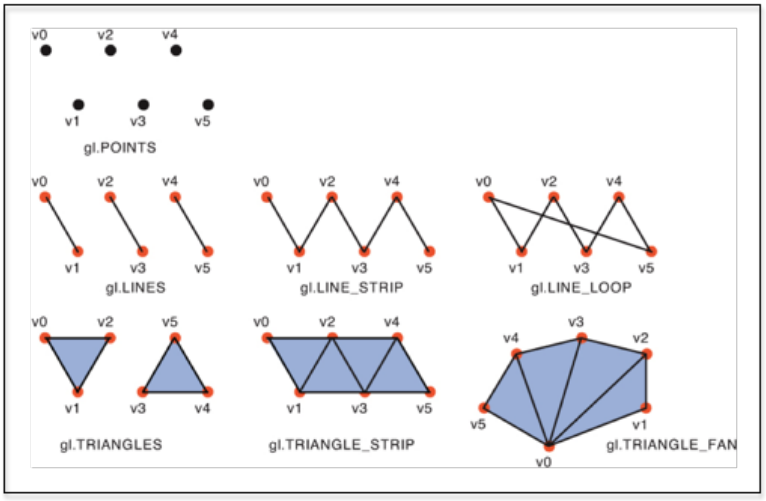


Figure 1.4: WebGL geometric primitives. Primitive type objects are used to draw basic shapes, such as triangle sets (arrays of triangles), triangle strips (described shortly), points, and lines. Primitives use arrays of data, called buffers, which define the positions of the vertices to be drawn. *Image courtesy of Wayne Brown*

the vertex data that is currently loaded on the GPU. In `gl.drawArrays(mode, first, count)` we specify the type of geometric primitive (i.e., gl.TRIANGLES), the starting index in the array of vector points (i.e., 0) and the number of indices to be rendered (i.e., 3).

```
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

1.1.4 Hello Rectangle

In the first exercise, we have seen how to draw a simple triangle. Using the knowledge you have gained try to modify your code to draw a Rectangle.

Try this: *Starting from the code of* ***`triangle.js`****, you need to add an extra vertex coordinate. See Figure 1.5. Pay attention to the order of vertices; otherwise, the draw command will not execute correctly.*

Since you have added an extra vertex, you'll need to modify the ***`gl.drawArrays()`****. However this is not enough. Try experimenting with different geometric primitives (e.g.* ***`gl.TRIANGLE_FAN`*** *or* ***`gl.TRIANGLE_STRIP`****).* What happens?

Both `gl.TRIANGLE_FAN` and `gl.TRIANGLE_STRIP` are primitives based on group

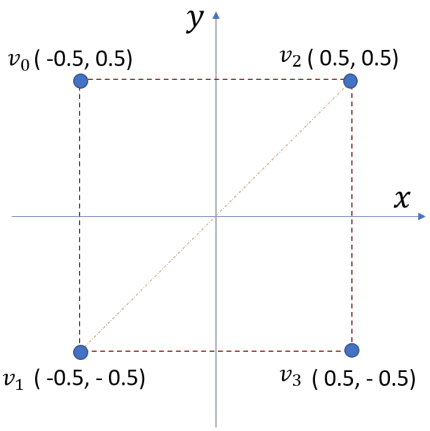


Figure 1.5: Rectangle coordinates.

of triangles that shares vertices and edges. The `gl.TRIANGLE_STRIP` combines each additional vertex with the previous two vertices to define a new triangle. The `gl.TRIANGLE_FAN` is, instead, based on a single (fixed) point and the next two points are used to form the first triangle. Subsequent triangles are formed from one new point (e.g., v_3), the previous point (e.g., v_2) and the first (fixed) one (i.e., v_0). See Figure 1.4).

Why do we limit ourselves to triangles? Polygons are in general problematic as a set of vertices might not lie in the same plane. Such problems do not arise with triangles, as long as the three vertices are not collinear, the triangle is flat and convex and therefore easy to render. In order to deal with more complex geometries we perform a process known as **triangulation**.

Try this *As WebGL does not natively support the creation lines of variable thickness, we have to approximate them by using a triangle strip. Try creating a thick horizontal straight line with the use of the* ***`gl.TRIANGLE_STRIP`****. See Figure 1.6 and observe the pattern of the vertices coordinates. Try to modify the* ***`thickLine.js`*** *file in the* ***`ThickLines`*** *folder.*

You can manually enter each of the points or write a simple routine that takes as input the coordinates of the starting and ending point and the line thickness and returns a vector of coordinates. Once again be mindful of the order in which the points are give is important!

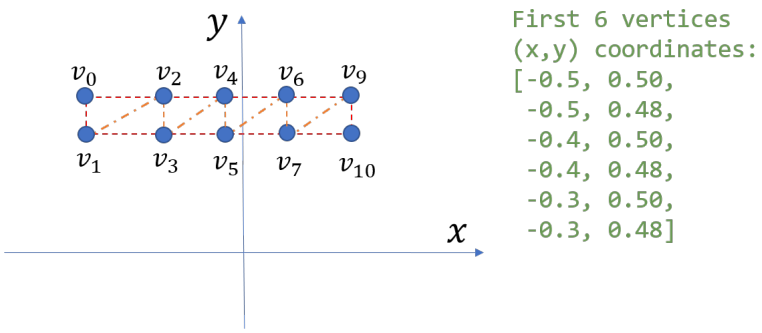


Figure 1.6

1.2 Exercises

1. In Figure 1.4, you can find the geometric primitives available in WebGL. Try to modify the `triangle.js` file and use these primitive in the `gl.drawArrays()` and look at the results in your web-browser. What happens?
   Hint: In `gl.POINTS` you need to modify the vertex shader code by simply adding the following line: `gl_PointSize = 10.0;` as single pixels won't be very much visible.

2. Create a triangle using `gl.TRIANGLE_FAN`.

3. Create a Hexagonal shape using `gl.TRIANGLE_FAN` as shown in the figure 1.7.

4. Try changing the points coordinates - be mindful of to keep them within the canonical viewing volume ($x = \pm 1$, $y = \pm 1$, $z = \pm 1$).

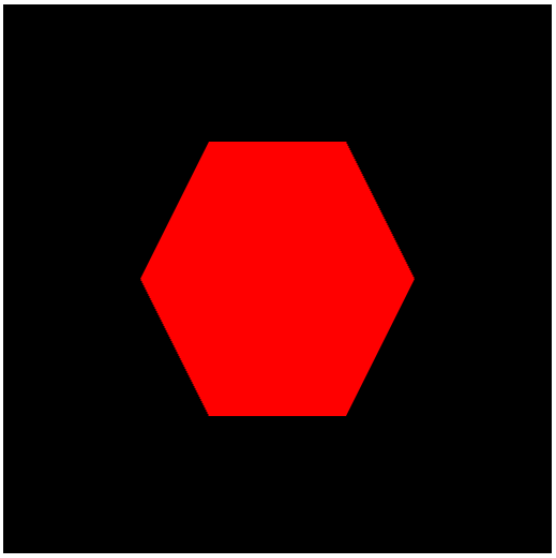


Figure 1.7: Hexagon

References

We've used a number of source references for these tutorials.

The labs are mixture of examples and exercised available in:

1. *Interactive Computer Graphics, A top-down approach with WebGL* by Edward Angel and Dave Shreiner, 2020.
2. *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL* by Kouichi Matsuda and Rodger Lea, 2013.
3. The tutorials made available by Dr. Wayne Brown, Associate Professor of Computer Science at the United States Air Force Academy `http://learnwebgl.brown37.net`.
4. The full documentation for WebGL can be found at `https://www.khronos.org/webgl/`.
5. *Learn Three.js* by Jos Dirksen, 2018.
6. *WebGL: Up and Running: Building 3D Graphics for the Web* by Toni Parisi, 2012.
7. The full documentation for Three.js can be found at `https://threejs.org/`.

For more information on web-development there is plenty of material online, however here there is a couple of references to get you started:

1. *Fundamentals of web development* by Randy Connolly and Ricardo Hoar, 2015.

2. *Web development and design foundations with HTML5* by Terry Felke-Morris, 2019.

2 Lab2: Shaders and Interactivity

In the previous lab, while drawing a triangle the position of the triangle vertices and colors drawn on the screen were hard-coded. In this lab, we will add a little more flexibility by allowing the user to manipulate color and position with the help of shaders. We will also learn to get interactive input using keyboard and mouse in this lab.

Download the Lab2_WebGL.zip file from the `/modules/cs324` folder or from Moodle under the *Practical Sessions/Labs* section.

2.1 Shaders

Figure 2.1 shows the role of shaders in the WebGL pipeline where vertex buffer objects are sent to vertex shader, which is responsible for setting the position of each vertex and some additional data calculations. The output of the vertex shader is then used by the GPU to draw primitives. Next in line is the fragment shader which is responsible for pixel-wise interpolation, coloring of the pixels and clipping anything outside the view-port as discussed in Lab-1 section 1.1.3. We will discuss clipping in detail during the lectures.

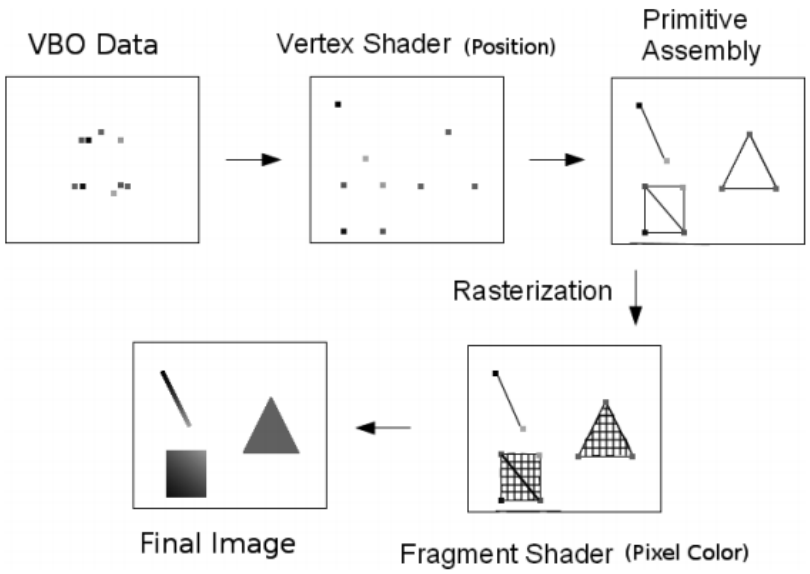


Figure 2.1: WebGL rendering process overview using shaders [Angel 7th Edition]

Shaders are essentially programs/functions that are used to draw something on the screen and run on the GPU. They may run on CPU depending on your hardware and video drivers or graphics API implementation. Using GPUs for

shader functions have a number of benefits over CPU number crunching due to their efficient matrix multiplication powers. There are several kinds of shaders but the two most commonly used - and the only two available are vertex shaders and fragment shaders.

2.1.1 Vertex Shaders

The Vertex Shader is the programmable Shader stage in the rendering pipeline that handles the processing of individual vertices [2]. Vertex shaders are responsible for manipulating the pixel coordinates called vertex using the position variable `gl_Position`. This is a special global variable used for storing the position of the current vertex. In the previous lab, we used a simple vertex shader right after the `<html>` tag where, we passed the vertices without any modification to the pipeline as:

```
<script id="vertex-shader" type="x-shader/x-vertex">
    attribute vec4 vPosition;

void
main()
{
 gl_Position = vPosition;
}
```

The `type` here instructs the browser that what follows is a vertex shader written in the GLSL language [3]. GLSL is a C-like language that expands on the C data types including vectors and matrix types. What follows is the bare minimum description for a vertex shader (this is called a pass-through shader). In this case, we are using a `vec4` type, which is the C equivalent of an array made of four floating points [4].

The `attribute` keyword indicates that this is an expected input to the shader from the application when this is initiated. Each shader is a complete program with the `main()` as its entry point. This specific shader simply takes in a position called `vPosition` and outputs it as a position for the vertex. The `gl_position` is a

[2] `https://www.khronos.org/opengl/wiki/Vertex_Shader`

[3] There are many ways more or less complex (and/or elegant) in which we can handle shaders, however, since this a computer graphics course, we prefer to use a simple approach and let you focus on understanding the fundamental concepts without to having to deal too much with issues related to web-development (e.g., using jQuery). In this instance, shaders will be created as null-terminated strings of characters and included in the HTML file. The WebGL application code will be in a separate JavaScript file.

[4] A single vertex is described using four components in homogeneous coordinates - we will see what this exactly means soon in the coming lectures

built-in state variable (so no need to declare it) for the shader and is the position passed to the rasterizer and MUST be output for every vertex shader. In this case, we are also assuming that the vertex passed will be written in the correct coordinate system (all coordinates values will be between -1 and 1). Let's have a look at one more example of vertex shader.

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

attribute vec4 vertPosition;
attribute vec4 vertColor;
varying vec4 fragColor;

void
main()
{
    fragColor = vertColor;
    gl_Position = vertPosition;
}
```

This shader expects two input `attributes` from the application which are position and color while there is one `varying` type which means the vertex shader will write in it and fragment shader will read it. Apart from `attribute` and `varying` there is another data type commonly used in GLSL: `uniform` (not used in the above example) it represents constant value throughout the program.

2.1.2 Fragment Shader

The fragment shader is responsible for manipulating colours and a single depth value in RGBA (red, green, blue, alpha) format for each pixel being processed using the global variable `gl_FragColor`. The Alpha component determines colour opacity. The value 1.0 indicates a complete opaque colour, whereas 0.0 represents completely transparent opacity value. The output, `gl_Position`, of the vertex shader is passed onto the next stage in the graphics pipeline that converts the vertex's position to a pixel location in the rendered image called rasteriser. The rasteriser determines which pixels should be drawn to draw an object. This varies based on whether the vertex is part of a point, a line, or a triangle primitive as experimented in the previous exercise. The rasteriser will output a list of all the pixels that will need to be coloured to represent an object. Each pixel and its associated rendering data can also be called a fragment.

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

void
main()
{
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
</script>
```

Let's have a look at the example where input from vertex shader is used to change the colors.

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

varying vec4 fragColor;

void
main()
{
    gl_FragColor = fragColor;
}
</script>
```

Note that you must specify precision (i.e., `highp`, `mediump` and `lowp`) for the floating point - this comes from OpenGL ES, and is related to the fact that not all low power devices will support high precision floating points.

Lets look at the steps required to add more colors in our previous example from Lab-1 1.1.3 with solid color as seen in Figure 2.2.

2.1.3 The application file

Now we look at the `triangle.js` file. Once the browser has entered the execution phase the main() function will be called. In this file, we find the definition of our application. Essentially `triangle.js` is a JavaScript program that draws a multicoloured triangle on the drawing area defined by the `<canvas>` element.

1 - Set up canvas: The main three steps required to draw 2D graphics on the canvas are:

1. Obtain the `<canvas>` element

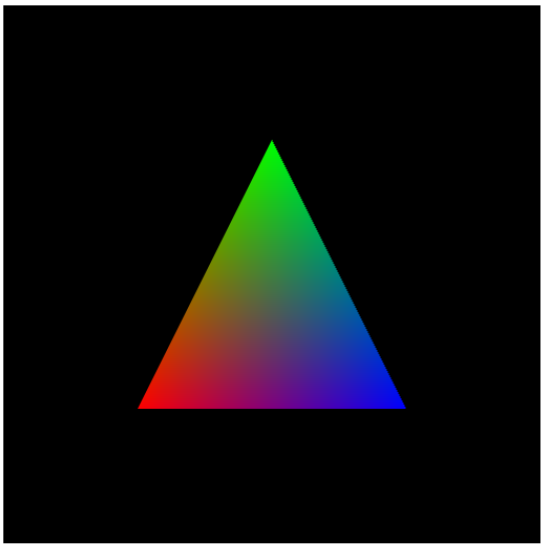


Figure 2.2: Output of colors passed using the vertex shader to fragment shader.

The HTML file specifies a `<canvas>` of the requested size. The request of the `canvas` element is made through using the `id` specified before (`gl-canvas` in our case).

```
var canvas = document.getElementById("gl-canvas");
```

2. Request the rendering context from the element

   To request the rendering context we are using a function offered in the `webGL-utils.js` file.

   ```
   gl = WebGLUtils.setupWebGL(canvas);
   if ( !gl ) {
       alert("WebGL isn't available");
   }
   ```

   This is just a utility that instead of setting up a context manually will also check for success or failure and on failure it will present an appropriate message to the user. The most important operation performed by the `WebGLUtils.setupWebGL` is to obtain the rendering context:

   ```
   var context = canvas.getContext('webgl');
   ```

2 - Generate data: Once obtained the rendering context we can start drawing. At this stage, we are only aiming at drawing in 2D, so we will indicate only

(x, y) tuples. The three 2D coordinates the vertices of our triangle:(-1,-1), (0, 1), (1,-1) are stored in a JavaScript typed array of 32-bit floating point numbers.

```
var vertices =
    new Float32Array([-1, -1, 0, 1, 1, -1]);
```

Add this new color variable as they are made up of (RGBA) Red, Green, Blue, Alpha but we have used simple RGB for this example, so first three indices will exhibit the color for vertex 0, second three for vertex 1 and the third three for vertex 2.

```
var colors =
    new Float32Array([1.0, 0.0, 0.0,   // R, G, B -> v0
                      0.0, 1.0, 0.0,   // R, G, B -> v1
                      0.0, 0.0, 1.0]); // R, G, B -> v2
```

Finally, we get ready to draw on our canvas: we call `gl.viewport(x,y,w,h)` and `gl.clearColor()`.

```
gl.viewport(0, 0, canvas.width, canvas.height);
gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

3 - Load shader programs: Next, we will load the shaders we have declared in a program object, taking the `id` of the two shaders we have previously declared and the WebGL context.

```
var program =
    initShaders(gl, "vertex-shader", "fragment-shader");
gl.useProgram(program);
```

4 - Connect position and color vertex: At this point, we want to send our data to the GPU using the vertex buffer objects so after vertex buffer you need to create another buffer object for color and bind it to the GPU.

```
// Load the data into the GPU
var bufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, bufferId);
gl.bufferData(
    gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

// Associate position shader variables with data buffer
var vPosition =
    gl.getAttribLocation(program, "vertPosition");
gl.vertexAttribPointer(
    vPosition, 2, gl.FLOAT, false, 0, 0);
```

gl.getAttribLocation returns the location of the attribute variable such as the vertPosition and vertColor variable in our vertex shader. We then describe the form of the data in the vertex array and enable the vertex attributes that are in the shader.

```
var colorbufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorbufferId);
gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STATIC_DRAW);

// Associate color shader variables with data buffer
var vColor =
    gl.getAttribLocation(program, "vertColor");
gl.vertexAttribPointer(
    vColor, 3, gl.FLOAT, false, 0, 0);
```

5 - Enable Variables: The last step before rendering is to associate shader variables with our data buffer:

```
gl.enableVertexAttribArray(vPosition);
gl.enableVertexAttribArray(vColor);
```

6 - Render: At this point, we call the `render()` function which will simply clear the frame buffer and then render the vertex data that is currently loaded on the GPU. In gl.drawArrays(mode, first, count), we specify the type of geometric primitive (i.e., gl.TRIANGLES), the starting index in the array of vector points (i.e., 0) and the number of indices to be rendered (i.e., 3). Finally, you would be able to see the triangle with RGB colors.

```
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Try this: *Draw only the **vertices** of the triangle (without the whole triangle) with the same colour as in the example.*

2.2 Interactivity

Most graphics and window systems use a mechanism called *event processing*. Events are changes detected by the operating system (e.g., user pressing the keyboard or clicking a mouse button). When events occur they are placed in a queue called **event queue**. A program can decide to ignore the event or act upon it.

Within WebGL, we will identify events to react to, through functions called **callbacks** or **event handler**. Each callback is associated with a specific type of

event. In this lab, we will learn how to make our program react to user mouse clicks on the canvas and to key press from the keyboard.

2.2.1 Draw on Mouse Click

Let's see how we can draw a point on the canvas by reacting to mouse clicks. In the `Mouse` folder you'll find the code to accomplish this task using WebGL. You will notice the `html` file has not changed much compared to our previous exercise. Let's have a look at the JavaScript file.

As explained above, here we will be using an event handler. To use an event handler, you need to register the event handler (that is, tell the system what code to run when the event occurs). The HTML5 `<canvas>` element supports special properties for registering event handlers for a specific user input, which you will use here. For example, when you want to handle mouse clicks, you can use the onmousedown property of the `<canvas>` to specify the event handler for a mouse click. We will return to this later in this lab.

```
var v_Position = gl.getAttribLocation(program, "
    vPosition");
if (v_Position < 0) {
   console.log("Failed to get the storage location of
      vPosition");
   return;
}
```

The return value of this method is the storage location of the specified attribute variable. This location is then stored in the JavaScript variable, `v_Position` for later use.

Once we have acquired `v_Position` we register the event handler (i.e., tell the system what code to run when the event occurs). We specify the event handler for the mouse click as follows:

```
canvas.onmousedown = function(ev) {
            click(ev, gl, canvas, v_Position);
   };
```

This mechanism is called an anonymous function. As its name suggests, it is a convenient mechanism when you define a function that does not need to have a name [5]. When drawing a point, three variables are needed: `gl`, `canvas`, and `v_Position`. These variables are local variables that are prepared in the function

[5]For those of you unfamiliar with **anonymous functions** there are a couple of differences to canonical functions to highlight: while a named function can be accessed anywhere in a program, an anonymous function assigned to a variable only exists and can only be called after the program executes the assignment. The second difference is that you can change the value of a variable

`main()` in the JavaScript program. However, when a mouse click occurs, the browser will automatically call the function that is registered to the `<canvas>`'s `onmousedown` property with a predefined single parameter (i.e., an event object that contains information about the mouse press). Using an anonymous function is a quite flexible way to avoid the use of global variables and allows us to pass `gl`, `canvas`, and `v_Position` to the `click()` function as well as the event information `ev` (i.e., the mouse position on the canvas).

At this point, the `click()` function is called, let's look in detail at the code and notice that in the JavaScript file we have declared a global variable: `var g_points = [];`.

```
1  function click(ev, gl, canvas, v_Position) {
2    var x = ev.clientX; // x coordinate of a mouse pointer
3    var y = ev.clientY; // y coordinate of a mouse pointer
4
5    // Transform the mouse coordinates from <canvas> to
       WebGL on <canvas>
6    var rect = ev.target.getBoundingClientRect();
7
8    x = ((x - rect.left) - canvas.width/2)/(canvas.width
       /2);
9    y = (canvas.height/2 - (y - rect.top))/(canvas.height
       /2);
10
11   // Store the coordinates to g_points array
12   g_points.push(x); g_points.push(y);
13
14   // Clear <canvas>
15   gl.clear(gl.COLOR_BUFFER_BIT);
16
17   var len = g_points.length;
18   for(var i = 0; i < len; i += 2) {
19     // Pass the position of a point to v_Position
         variable
20     gl.vertexAttrib3f(v_Position, g_points[i], g_points[
         i+1], 0.0);
21
22     // Draw
```

and assign a different function to it at any point making anonymous functions more flexible than named ones.

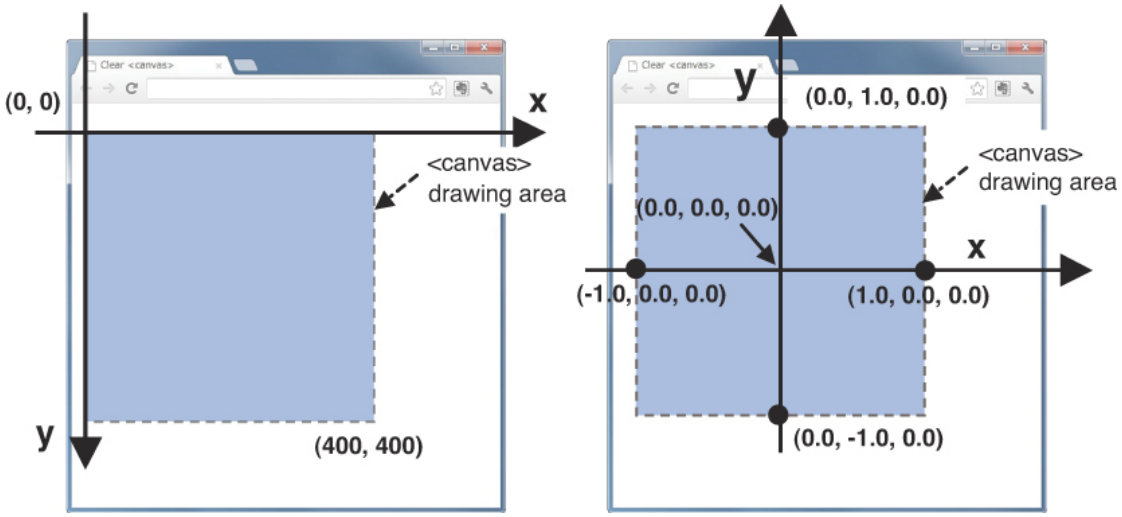


Figure 2.3: `<canvas>` coordinate system (left) vs WebGL on `<canvas>` (right). *Image courtesy of R. Lea and K. Matsuda*

```
23      gl.drawArrays(gl.POINTS, 0, 1);
24    }
25  }
```

The `var g_points = []` variable will store the position of each of the mouse press and you can get the coordinates by using `ev.clientX` and `ev.clientY`, as shown in line 2 and 3.

However, there are some manipulations to perform before we can use these coordinates. As illustrated in Figure 2.3, `ev.clientX` and `ev.clientY` are in fact indicating the position of the mouse in window coordinates, not in the `<canvas>`. We need to first obtain the coordinates of the `canvas`. We can do so by calling

```
ev.target.getBoundingClientRect().
```

Moreover, we need to be aware that the `canvas`'s coordinate system is different from WebGL in terms of their origin and axis directions. The `x` and `y` values are therefore calculated in line 8 and 9 and subsequently pushed into the `g_points` array (i.e., every time a mouse click occurs, the position of the click is **appended** to the array).

The lines that follow will iterate on the `g_points` vector extracting the `x` and `y` values and draw on the canvas each point. The function `gl.vertexAttrib3f (location, v0, v1, v2)` assigns the values `v0`, `v1` and `v2` to the attribute variable specified by `location`. This will allow to pass the `x` and `y` values obtained through the mouse click to the vertex shader. Note the last value `v3` is set to `0.0` as we are operating in 2D.

Try this: *Using the* `Ctrl+Shift+I` *(Windows + Linux) or* `Command+Option+I` *(Mac), open the* `console` *and use method the* `log()` *as* `console.log('Hello WebGL Logging')`*, to output the mouse coordinates in both reference systems:* `<canvas>` *and WebGL on* `<canvas>` *on the console.* [6]

2.2.2 Keyboard Interaction

We can also use the keyboard to control our interaction. Open the `Keyboard` folder and then open `rotatingTriangle.html` file in your browser. By using the `ArrowUp` you will be able to rotate the triangle clockwise.

Let's have a look at the code. Reading through the `rotatingTriangle.js` file you will discover that we are using a `keydown` event. However, this event is a window event rather than a canvas event. To do this, we use the global object `window` which is defined by the browser and is available to all JavaScript files. The call to the window's addeventListener() method contains, again a declaration of an anonymous function.

```
window.addEventListener('keydown', function(e) {
  // Code describing the behaviour
});
```

Depending on the key pressed (i.e., `e.key`) we can drive the behaviour. In this case, we are altering a value `theta` that is used by the vertex shader to modify the position of each of the triangle vertices.

One last thing to notice is the use of the requestAnimationFrame() in the `render` function. Animating with standard loops have inherit flaws due to the difference in compute powers of low-end and high-end devices, as same animation will behave differently on different devices. `requestAnimationFrame()` calls the right time to sync the animation with the screen and it also only calls when the tab is visible. This allows us to call the render function recursively and helps us execute animation related JavaScript code that make changes to the user's screen in an efficient, optimized manner. In `webgl-utils.js` there is an overloaded function `requestAnimFrame()` which has been used in these labs and it calls either the browser specific function e.g., `webkitrequestAnimationFrame()` or standard `requestAnimationFrame()` for the animations.

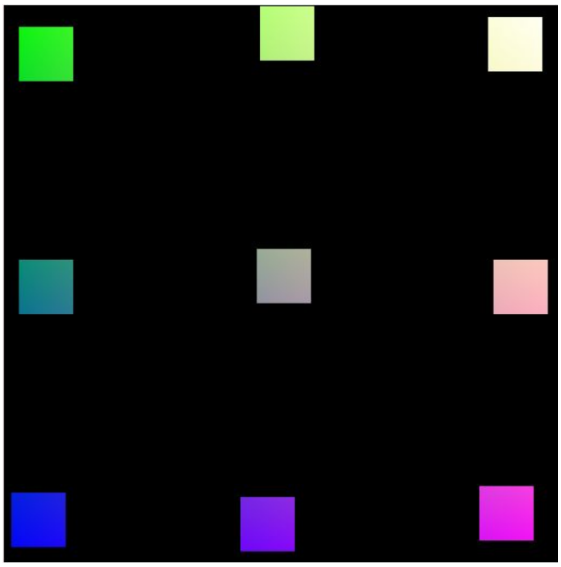


Figure 2.4: Demo of point color based on the position.

2.3 Exercises

1. Modify JavaScript file in the `Keyboard` folder by adding the following behaviours: counter-clockwise rotation when the user presses the `ArrowDown` key, reset the triangle rotation when the user presses the `Enter` key.

2. Open the `Mouse2` folder and modify the JavaScript file so that the position of the triangle changes according to where the user clicks on the canvas. ***Tip:*** Notice that the `VertexShader` in the `html` exposes another variable called `m_Increment` for holding the values of x and y. Your task is to update this variable with the mouse coordinates.

3. Using the `Keyboard` folder modify it to add additional keyboard functionality to change the color of triangle on key press, where 'r' corresponds to red color, 'g' corresponds to green color and 'b' corresponds to blue color. Also add the mouse click listener where on each mouse click a random color for the triangle will be pick along with a random rotation.

4. Add the additional functionality on top of previous question where on mouse click you move the triangle to the location and then perform a random rotation and color change.

[6]To access console output, right click on the canvas, select *inspect* (on Google Chrome and Microsoft Edge Chromium) or *inspect element* (on FireFox) and then *console* or press `Ctrl+Shift+I` (Windows + Linux) or `Command+Option+I` (Mac) and then *console*. Use the link to explore more on how to print using console function

5. Repeat the previous task with a square now where on mouse click you move the square to the location and then perform a random rotation and color change.

6. Extend the `MousePointColor` point click lab demo with additional color integration using the fragment shader instead of using the same solid color for the points. ***Tip:*** There are multiple built-in variables exposed by the WebGL in fragment shader which can be used to assign colors e.g., the Fig. 2.4 uses the `gl_FragCoord` for different colors.

3 Lab3: Transformations

Computer graphics is all about representing things and visualising them as realistically as we see them in the real world. Each of our two eyes works like a **pin-hole camera**, where light reflected from objects in the world goes in straight lines through a **pupil** (the lens), and is focused on our **retina**. If you have ever made a pin-hole camera with a box and a piece of tracing paper as the imager, you might remember that the pictures are **projected** up-side down. This is because the **projection plane** or **view plane** is behind the pin hole, the **centre of projection**: all straight lines from points in the scene project through the pin-hole, onto the view plane.

It may be obvious, but remember that although the world is three dimensional (3D), our view of it is two dimensional, 2D. The imager (or retina) is a 2D surface.

If you are an artist and hold up an empty picture frame in front of you, you can see a window on the world, and light from objects being **projected** on to an imaginary **view plane** in *front of* the centre of projection. This example is exactly how you should think about how computer graphics calculates a 2D projection of objects or the graphics application models. In fact, it's all a process of calculating exactly where **vertices** (3D points) project onto a given 2D surface (the **view plane**).

In this lab, we will learn the fundamentals of setting up a virtual "pin-hole" camera model and getting it to project 3D primitives (3D vertices, lines in three space) onto a view plane.

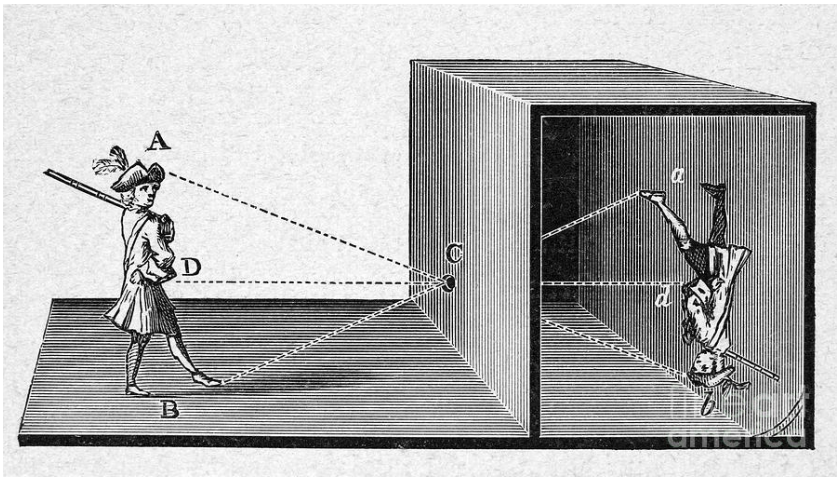


Figure 3.1: A pin-hole camera: view plane behind centre of projection. This sketch depicts a **camera obscura**, which means hidden room and was famously used by the Dutch artist Vermeer. That's where we get the name **camera** from.

We take for granted that when you perform projection, there is the effect of **perspective**; things further away look smaller than things closer to you. We use this fact to estimate distances, and work out what is in front of what. There are other cues too, like **parallax**, **shading**, **shadows**, **texture**, **depth of focus**, **fog** and so on as discussed in topic 02 in lectures. In computer graphics, these cues can be simulated, but for now, we will focus on simple **wireframe** modelling, which is often used in computer aided design (CAD).

Before we look at perspective projection, we will use **parallel** or **orthographic projection**. This has the property that all projection lines run parallel onto the view plane. This sounds dumb, but it is roughly what happens when we look at objects close up and is used in for example 3D CAD drawings and medical image visualisation because it allows us to take measurements of sizes off the image: there is no size distortion despite the projection. See Figure 3.2 for an illustration.

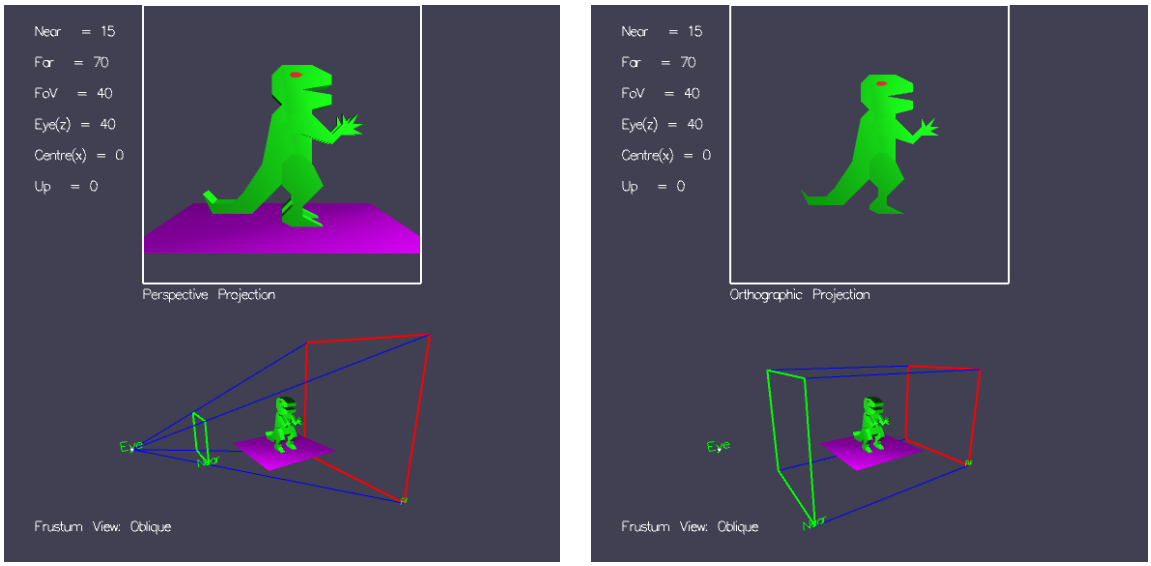


Figure 3.2: (left) Perspective and (right) Orthographic Projection. Note that the **view volume** changes shapes. The projection plane in both cases is the FRONT plane of the view volume. Note also how the view volume is a **frustum** (truncated pyramid) in perspective projection and a **right parallelepiped.**

3.1 Orthographic versus Perspective Projection

As seen previously, WebGL is based on a pipeline model, and the first part involves geometric manipulations (i.e., operations on vertices). These can be described as a sequence of transformations or equivalently a sequence of changes of frames for

the objects specified by the application. The frames appear in the pipeline in the following order:

1. model coordinates [7]
2. world coordinates
3. camera coordinates
4. clip coordinates
5. normalized device coordinates
6. window or screen coordinates.

So far in our exercises we have declared points within the clipping coordinates (i.e., x, y and z in the $[-1, 1]$ range) and we "ignored" the presence of the camera as this was positioned at the origin. However, we need more flexibility.

The first operation is to position and orient the camera and this is achieved through the model-view transformation matrix. By applying this transformation to the vertices they will be represented in the camera coordinates frame. The second operation is to apply projection transformations - orthographic or perspective - this will be done through the projection matrix. This operation will allow us to convert a viewing volume specified in camera coordinates to fit inside the viewing cube in clip coordinates.

Orthographic Projection: Orthographic projections are a special case of parallel projections where the projectors are perpendicular (i.e., 90 degrees) to the view plane. An orthographic projection does not modify the relative relationships between the (x, y) values of vertices and therefore an object's size does not change as it moves closer or farther from the camera as seen on right panel in Figure 3.2.

In the Lab3_WebGL.zip open the `TetrahedronOrtho` folder you will find a small demo `tetrahedron.html` that allows you to manipulate several parameters. This demo shows orthographic projections of a tetrahedron (a top view in this case), while in the next exercise we will apply perspective projections. The tetrahedron is centred in the origin in object coordinates. The camera location is described by the `eye` vector specified in the object frame, this vector points at a second point, the so called `at point`. Once we fix the desired `up` direction for the camera we can construct model-view matrix through the `lookAt` function. We will use the Look-At function implemented in the `MV.js` library [8].

[7] we do not use the object frame directly, but rather implicitly, by representing points and vectors in it.

[8] This library is contained in the `Common` folder. Please also have a look at this library as it exposes several useful functions such as vector sum, multiplication or matrix transpose that can be useful in the future.

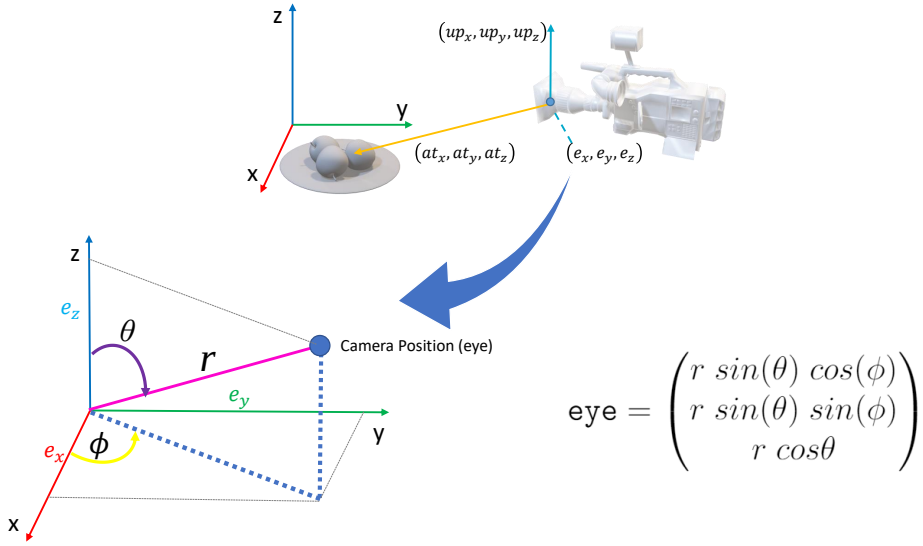


Figure 3.3: Illustration of the camera position in the world coordinates system. The camera is identifyed by *eye*, ***up*** *and* ***at***. The *eye* position is described in polar coordinates.

In this demo, the `eye` point is described in polar coordinates (see Figure 3.3):

$$\texttt{eye} = \begin{pmatrix} r\ sin(\theta)\ cos(\phi) \\ r\ sin(\theta)\ sin(\phi) \\ r\ cos\theta \end{pmatrix}$$

where radius r is the distance from the origin. The sliders in the demo allow you to change these three parameters. In the Javascript file (`tetrahedron.js`) the `up` vector is described as the y direction in object coordinates

```
const at = vec3(0.0, 0.0, 0.0);
const up = vec3(0.0, 1.0, 0.0);
```

allowing us to specify a model-view matrix through the `lookAt` function.

```
modelViewMatrix = lookAt(eye, at , up);
```

Try this: *Your camera is located on top of the tetrahedron (look at the value of θ!). You can modify the values of the sliders so that you can look at the green triangle located at the base of the tetrahedron. Can you think of a way to modify the*

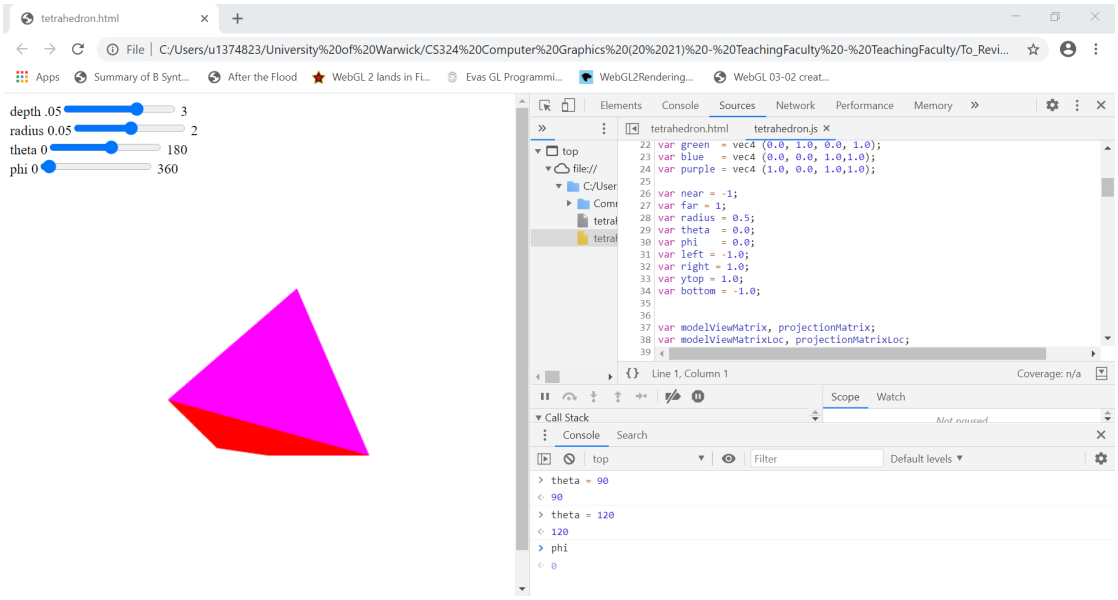


Figure 3.4: As an alternative to using the sliders you can read, set and modify global variables through the console.

`modelViewMatrix` by changing either the `eye` or the `up` vector so that the camera starts off by looking at the fourth triangle instead? To experiment with the effect of the change in values, please use the ϕ, θ and r sliders from the example.

Note: In both demos the values we are changing are declared as global variables. If you want to change them in a different way than using the sliders you can set them through the console - although this won't be reflected in the graphical interface (see Figure 3.4). Example: type `phi` and hit `Enter` to read the current value of the ϕ angle (value is in radians!).

In the `render()` function, we create the `projectionMatrix` by using the `ortho` function[9].

```
projectionMatrix = ortho(left, right, bottom, ytop,
   near, far);
```

Try this: *There is another slider, called `depth` available in the graphical interface. Can you guess what it does?*

[9]Notice that we have used the name `ytop` rather than `top` to avoid naming conflicts with the window object member names.

Perspective Projection: Perspective projection draws the scene like a real world view, where objects farther away from the camera appear to be smaller depicting the idea of depth where all lines project towards the vanishing point as seen on (left) in Figure 3.2.

The `TetrahedronPersp` folder contains a different demo with a perspective projection of the tethreaedron rather than an orthogonal one. The general structure of both the `html` and JavaScript files is very similar to the previous one, however we have two new sliders. One changes the field of view (i.e., the angle between the top and bottom planes) while the other changes the aspect ratio. These are the two parameters necessary, together with the `near` and `far` planes to define the projection matrix using the `perspective` function available in the `MV.js` library.

```
projectionMatrix = perspective(fovy, aspect, near, far);
```

Try this: *What happens if we set* ***`near=far`****? Can you explain why? What are the limits of* ***`far`*** *and* ***`near`****? Can you fix these limits in this code?*

3.2 3D Transformations

Fundamental to all computer graphics is the ability to manipulate objects and to create motion in a scene over time. In this part of the lab, we will learn how to transform an object's location and orientation in 3D. Open the folder `CubeRotation` . If you open the `cube.html` file in your browser you will realise that the program shows an animation of a cube rotating about one of its axis upon clicking on one of the rotate axis buttons.

Let's now dive into the implementation. In computer graphics, we adopt the homogeneous coordinate system. This allows us to represent affine transformations and in general projective transformations as matrix multiplication.

The rotation about the x, y and the z axis are represented by the following matrices:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We already used a multiplication similar to $R_z(\theta)$ in the exercise presented in Lab2 (2.2.2), as a rotation about the origin in 2D becomes a rotation about the `z` axis in 3D. These three matrices are presented in the Vertex Shader.

```
attribute  vec4 vPosition;
attribute  vec4 vColor;
varying vec4 fColor;
uniform vec3 theta;

void main()
{
  // Compute the sines and cosines of theta for each
     of the three axes in one computation.
  vec3 angles = radians( theta );
  vec3 c = cos( angles );
  vec3 s = sin( angles );

  // Remember: these matrices are column-major
  mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
  0.0,  c.x,  s.x, 0.0,
  0.0, -s.x,  c.x, 0.0,
  0.0,  0.0,  0.0, 1.0 );

  mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
  0.0, 1.0,  0.0, 0.0,
  s.y, 0.0,  c.y, 0.0,
  0.0, 0.0,  0.0, 1.0 );


  mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
  s.z,  c.z, 0.0, 0.0,
  0.0,  0.0, 1.0, 0.0,
  0.0,  0.0, 0.0, 1.0 );

  fColor = vColor;
  gl_Position = rz * ry * rx * vPosition;
```

```
    gl_Position.z = -gl_Position.z;
}
```

One thing to be aware of is the fact that WebGL adopts a **column major order**. However, in the folder `Common` you have available the library `MV.js` that will allow you to convert automatically from row-major to column-major by using the `flatten` function. One thing you might have noticed is that we are multiplying different rotation matrices in a specific order, as you may recall matrix multiplication is not commutative. Try experimenting different sequence of rotations through the graphical interface, you should be able to verify this.

As for the `cube.js` file you are already familiar with most of its content, the main difference is that in this case we are creating a cube in the `init` function through a call to `colorCube`.

At line 88 we define the variable `vertices`, that contains a list of each vertex coordinates. However, as you may recall from Lab1 (see Figure 1.4), WebGL only provides a primitive that allows us to draw triangular surfaces and therefore we have to build each of the square faces by using 2 triangles like we did in lab 1 1.2. The `colorCube` function creates 6 squares by using the `quad` function. This function accepts the indexes of the `vertices` variable. Each face of the cube is made by listing the index in the `vertices`. Figure 1.5 illustrates the way in which we obtain a square face by drawing two adjacent triangles - the order in which the vertices are listed is fundamental to obtaining the correct image.

Try this: *What happens if we change the order of the vertices in the* `quad()` *list? (e.g., try changing line* `79` *into* `quad( 3, 0, 0, 2 );`*)*

The interaction in this program is achieved by the use of buttons described in the `html` file.

```
<button id= "xButton">Rotate X</button>
```

Similarly to what we have done in the previous labs, we describe the behaviour in the JavaScript file by registering an event handler on elements in the HTML document.

```
document.getElementById("xButton").onclick = function
    () {
    axis = xAxis;
    stop = false;
};
```

In this case, we retrieve the button element through its `id` (i.e., `xButton` in this example) and react whenever the user has clicked on the specific element (i.e., `onclick`).

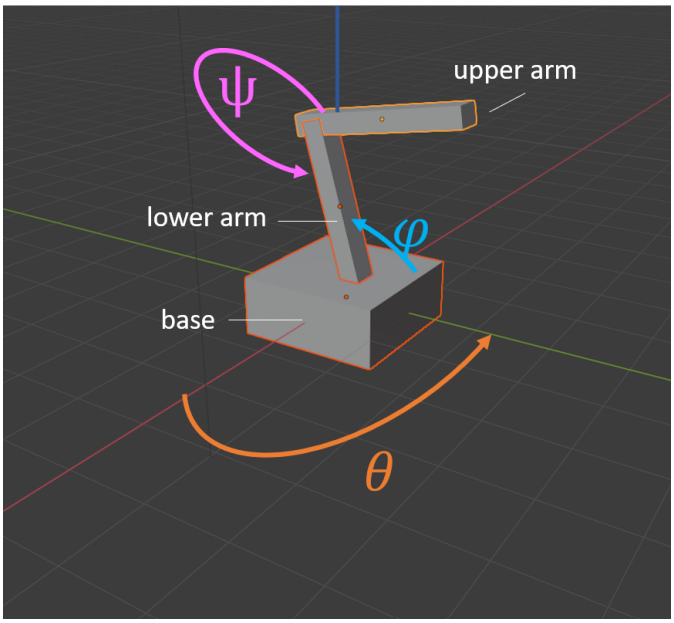


Figure 3.5: Simplified robotic arm.

3.3 Multiple Joint Model Transformation

The folder `MultiJoint` contains a very simplified representation of a robotic arm. The robot arm consists of three parts with three degrees of freedom, two of which can be described by joint angles between components and the third by the angle the base makes with respect to a fixed point on the ground.

In our model, each joint determines how to position a component with respect to the component to which it is attached. Or in case of the base, the joint angle positions relative to the surrounding environment. We can rotate the base about its vertical axes by angle θ (i.e., $R_y(\theta)$). The angles ϕ and ψ indicate the rotation of the other two components. As the angle vary, we can think of the frames of the upper and lower arms as moving relative to the base. By controlling the three angles, we can position the tip of the upper arm in three dimensions.

The program is written so that rather than specifying each part of the robot and its motion independently, we take an incremental approach. The lower arm is rotated about the `z`-axis in its own frame, but this must be shifted to the top of the base by a translation matrix $T(0, h_1, 0)$ where h_1 is the height above the ground to the point where the joint between the base and the lower arm is located. However, if the base has rotated we must also rotate the lower arm using the same $R_y(\theta)$. We can accomplish this by applying $R_y(\theta)T(0, h_1, 0)R_z(\phi)$ to the arm vertices. We can apply the same reasoning to the upper arm.

The `render()` function to display such robot arm uses the array named `theta`[3]

which stores θ, ϕ and ψ respectively and alters the model-view matrix incrementally to display the various parts of the model efficiently.

```
var render = function() {

  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  modelViewMatrix = rotate(theta[Base], 0, 1, 0 );
  base(); // Draw the base

  modelViewMatrix = mult(modelViewMatrix, translate
     (0.0, BASE_HEIGHT, 0.0));
  modelViewMatrix = mult(modelViewMatrix, rotate(theta
     [LowerArm], 0, 0, 1 ));
  lowerArm(); // Draw the lower arm

  modelViewMatrix  = mult(modelViewMatrix, translate
     (0.0, LOWER_ARM_HEIGHT, 0.0));
  modelViewMatrix  = mult(modelViewMatrix, rotate(
     theta[UpperArm], 0, 0, 1) );
  upperArm(); //Draw the upper arm

  requestAnimFrame(render); }
```

3.4 Exercises

1. In the `TetrahedronOrtho` we have added only a slider to control for the `far` and `near` plane (i.e. `depth` slider). Can you create two sliders that change (increase and decrease) the `width` and `height` values? What happens to the tetrahedron?

2. In the `TetrahedronPersp` Can you create a sliders to change (increase and decrease) the aspect ratio value? What happens to the tetrahedron?

3. The cube example was focusing solely on rotations. We want to be able to shrink or expand the size of the cube. Modify the `cube.js` and `cube.html` adding three new buttons. One button to shrink the size of the cube to half and one button to increase to twice its size. Also we would like an extra button that stops the rotation and resets the cube to its original size.**Hint:**

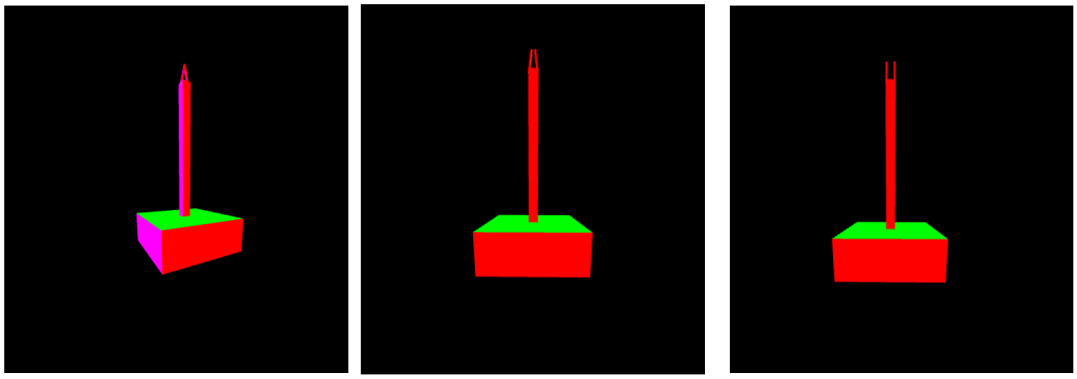


Figure 3.6: Robotic arm with additional clamp in a) close, b) half-open and c) open states.

recall the scale matrix expressed in homogenous coordinates:

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

You may want to modify the Vertex Shader in such a way that you can multiply each vertex for the scale matrix. You can also expose a variable - similar to what we did in Lab 2 - and modify this variable inside the JavaScript file. Does the order of scaling in the transformation matrix matters?

4. In the `MultiJoint` program add clamps on top of the upper arm to implement the grab functionality where using two clamps opening and closing will be performed as seen in Figure 3.6. You can implement this using a slider or a key.

4 Lab4: Lighting and Shading

4.1 Introduction

In real life, we take for granted the effects of light around us. We see the world because surfaces around us reflect light in the scene. Light rays, whether they are from the sun or artificial lights, such as light bulbs, are reflected off surfaces around us and the effects are mainly determined by three things: where the light is coming from, how the surface is angled relative to us, and the **surface properties**. For example, if the light is behind us when we are reading a book, it will be well lit. If a surface is shiny, like a silver pot, we will see bright spots where the the light sources are mirrored in it.

In the natural world, surfaces reflect light from **primary** light sources, such as the sun's rays, or artificial lights, but also secondary light sources, such as other surfaces. In fact, light will continue to "bounce" around the scene until all photons escape or are absorbed. Of course, certain surfaces will **reflect** more light than others: white surfaces reflect much more than the dark ones. A shiny mirror will reflect almost all **incident light**.

This model is known as a **global** model or lighting because we are trying to consider *all* surfaces as light sources, e.g., secondary light source, tertiary light source and so on. To model such interactions would be quite complicated and time-consuming. The model we will see today will offer a simplified approximation of these interactions, modelling only the illumination, i.e., the amount of incident light from the primary light source which is reflected from a surface which goes to the viewer and enters the camera. Furthermore, using **point light sources**: light sources that emit light in all direction equally. For the purposes of light calculations it is assumed that all light rays incident on the surface are parallel to each other (this is a good approximation if the light is far away, like the sun). Mathematically, it is the path from a light source (**a point in space**), to a flat surface and reflected towards the **eye position**. This is known as a **local** lighting model.

Although **local** lighting sounds like a cop-out, the effects it can generate are varied and convincing, despite the fact that the effect of light **scatter** around the scene is being ignored[10]. The key to understanding the local model used in our Lab exercise, is some basic 3D geometry and we need to know three quantities, which we will define here, and is shown diagrammatically in the figure 4.1:

1. **the light vector** this is the direction from the centre of the surface element to the light position. We can denote this vector by **L** and has unit length.

[10] ray-tracing, is a technique to model all light/surface interactions in a scene and uses multiple local models to accumulate the surface to surface reflections

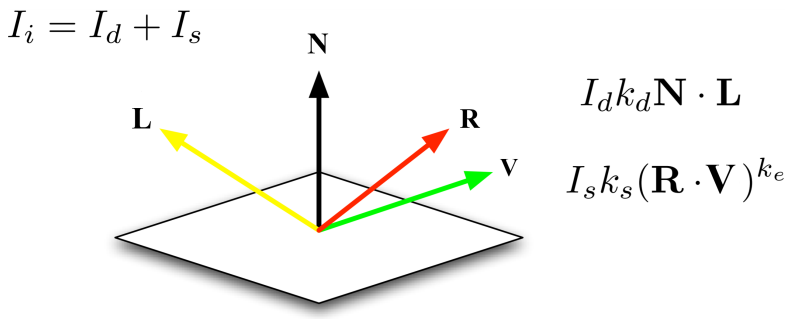


Figure 4.1: A local surface model shows the incident **light vector**, **L**, which is the direction from the surface centre to the light source; the surface's normal vector, **N**; the reflected light vector **R**, which is L mirrored in N; and the **eye vector**, **V**. Note all the vectors point away from the surface.

2. **the eye vector** this is in the direction from the centre of the surface element to the eye position, denoted by **V**. It too has unit length.

3. **the surface normal vector** which is the perpendicular vector of the surface. Remember that the surface is flat, so its tilt is given by its **normal vector**, denoted by **N**.

This lighting model combines the three geometric parameters: **L**, **N**, and **V** and with two properties of a surface which we are allowed to change:

1. **diffuse reflection coefficient** that models how **matt** the surface is: like paper, cloth, wood, plastic or rubber and some dull metals.

2. **specular reflection coefficient** that models how **shiny** the surface is: like silverware, metals, glass, varnished objects, crockery, etc.

A formula devised by Bui Tong Phong, known as the Phong Lighting Model, is used to combine these parameters like this:

$$I_{\lambda,r}(\lambda, \phi) = I_{\lambda,a} k_a(\lambda) + I_{\lambda,i}[k_d(\lambda)(\mathbf{L}^T \mathbf{N}) + k_s(\mathbf{R}^T \mathbf{V})^n]$$

where $I_{\lambda,r}$ is the reflected light energy, $I_{\lambda,a}$ is the ambient light energy, $I_{\lambda,i}$ is the incident light energy. k_a, k_d and k_s are the ambient, diffuse and specular properties of the surface. Here, **L** is the light vector. **R** is the mirror vector i.e., the vector **L** reflected in the surface normal **N**. Note that there is a third "surface" property which can be controlled and is known as the **specular exponent**, k_e, and it controls the size of the specular reflection.

The interesting thing about the Phong model is that: the diffuse contribution to the reflected energy does **not** depend on the eye position [11], whereas, conversely, the specular part entirely does depend on it. This means that matt surfaces look the same, wherever you stand, but shiny ones change as you move. Does that concur with your experience of the world?

Now the Phong Model goes further than this and adds to this an **ambient** lighting term, which is there to mop up all the scatter which is not being modelled and can be written as $k_a I_a$, which is an ambient coefficient times an ambient light energy in the scene, I_a. Generally, this term is kept small relative to the diffuse and specular parts.

4.2 Blinn-Phong model

Download the Lab4_WebGL.zip. The folder shows a sphere as an example to illustrate shading calculation. We will implement the Blinn-Phong model without distance attenuation and with a single light source in the vertex shader, as this is much more efficient for most applications. However, the availability of programmable shaders allows us to implement different shading algorithms from the Blinn-Phong lighting model and also to choose where to apply the lighting model: in the application, in the vertex shader, or in the fragment shader.

The interface allows you to change several parameters - they should be familiar by now as we have already used them in previous Labs. We have two buttons to change the values of the camera position parameters: r, θ and ϕ expressed in polar coordinates. Moreover we can change the number of subdivisions that allows us to describe a sphere by applying **recursive subdivisions** (more on this later).

Lighting Parameters In the JavaScript file we specify the light source colour and location:

```
var lightPosition = vec4(1.0, 1.0, 1.0, 0.0 );
var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0 );
var lightDiffuse = vec4( 1.0, 1.0, 1.0, 1.0 );
var lightSpecular = vec4( 1.0, 1.0, 1.0, 1.0 );
```

We specify the light position as a `vec4`. For a **point light** source, its position will be specified in homogeneous coordinates, for example, as:

```
var lightPosition = vec4(1.0, 1.0, 1.0, 1.0 );
```

[11] **Lambert's Law** states that the intensity of the diffused light is proportional to the cosine of the angle between two vectors, the surface normal and the direction of the light source.

if the fourth component is zero, the source becomes a **directional** source.

A directional light is a light source whose light rays are parallel, hence, it is considered to be at an infinite distance, such as the sun. This light source is considered the simplest, and because its rays are parallel can be specified using only direction and colour.

A point light represents a light source that emits light in all directions from one single point. Light bulbs, lamps, flames, and so on are, for example, represented using this model. This light source is specified by its position and colour. However, the light direction is determined from the position of the light source and the position at which the light strikes a surface. As such, its direction can change considerably within the scene. This type of light actually attenuates which means that it should become weaker the farther it is from the scene. For the sake of simplicity, light is treated as non attenuating in these tutorials.

When we switch on a light in a dark room at any point in the room from which we can see the light source there is a contribution from the light hitting the surfaces directly to the diffuse (`lightDiffuse`) and the specular component (`lightSpecular`). There is also another contribution determined by the light bouncing off multiple surfaces in the room, we call this component ambient light (indirect light) described by `lightAmbient`. Ambient light does not have position and direction and is specified only by its colour.

The diffuse component (i.e., `lightDiffuse`) accounts for the amount of light reflected and is determined by the angle between the light ray and the surface normal vector. While the specular component represents the fact that if a specular reflection light ray goes straight into a camera, it is as if the camera is seeing the light source directly, even though it has bounced off of an object. The camera is seeing the light of the light source, not the colour of the object. If you have a white light source, the specular reflection will be white. If you have a red light source, the specular reflection will be red. Therefore, to model specular reflection you need to specify the colour of the light source in your lighting model.

Try this: *What values do you have to change if you want to emulate a red light source, how does this affect the appearance of the object?*

Material Properties How light is reflected by the surface of an object and thus what colour the surface will become is determined by two things: the type of the light and the type of surface of the object. In this example, we have assumed a single material. The material properties describe information about the surface and include its colour and orientation. The JavaScript code contains the following lines that specify the material properties:

```
var materialAmbient = vec4( 1.0, 0.0, 1.0, 1.0 );
var materialDiffuse = vec4( 0.0, 0.8, 1.0, 0.10 );
var materialSpecular = vec4( 1.0, 1.0, 1.0, 1.0 );
var materialShininess = 20.0;
```

These lines allow us to specify ambient, diffuse and specular reflectivity coefficients (k_a, k_d, and k_s) for each primary primary colour, as RGB or RGBA for the opaque surface. Here we have defined a purple ambient reflectivity and blue diffuse properties and white specular reflections. For the specular component, we also need to specify its shininess (i.e., `materialShininess`). Note that often the specular and diffuse reflectivity are the same.

Try this: *What happens if you increase the* ***`materialShininess`*** *value? If we change the specular reflection to be the same as the* ***`materialDiffuse`*** *value what happens to the highlight?*

We can then calculate the diffuse reflection, the environment (or ambient), and the specular reflection:

```
var ambientProduct = mult(lightAmbient, materialAmbient)
    ;
var diffuseProduct = mult(lightDiffuse, materialDiffuse)
    ;
var specularProduct = mult(lightSpecular,
    materialSpecular);
```

Ambient reflection `ambientProduct` describes the amount of light that is reflected by the surface. In ambient reflection, the light is reflected at the same angle as its incoming angle. Diffuse reflection (`diffuseProduct`) describes the phenomenon of light scattering equally in all directions from where it hits when hits a perfectly diffuse reflector. However, most surfaces are rough like paper, rock, or plastic the light is scattered in random directions from the rough surface. In diffuse reflection, the colour of the surface is determined by the colour and the direction of light and the base colour and orientation of the surface.

The specular component `specularProduct` describes the highlights that we see reflected from shiny object. These highlights usually show a different colour from the reflected ambient and diffuse light. For example, a green pool ball viewed under white light has a white highlight as seen in Figure 4.2.

We can then send these values to the vertex shader in a similar fashion as we have done previously for the projection matrix and the model view matrix.

```
gl.uniform4fv( gl.getUniformLocation(program,
  "ambientProduct"),flatten(ambientProduct) );
```

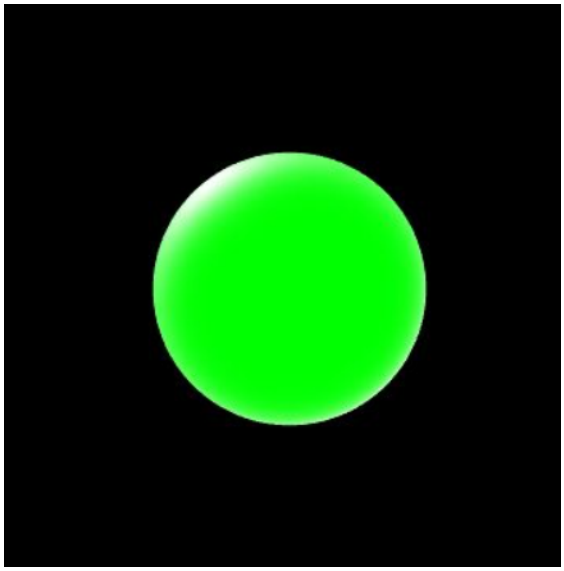


Figure 4.2: A white highlight is on top of the green ball showing the specular component of the object.

```
gl.uniform4fv( gl.getUniformLocation(program,
  "diffuseProduct"),flatten(diffuseProduct) );
gl.uniform4fv( gl.getUniformLocation(program,
  "specularProduct"),flatten(specularProduct) );
gl.uniform4fv( gl.getUniformLocation(program,
  "lightPosition"),flatten(lightPosition) );
gl.uniform1f( gl.getUniformLocation(program,
  "shininess"),materialShininess );
```

Geometry Specification The sphere geometry used in this example is obtained through **recursive subdivisions**, this is a powerful technique for generating approximations of curve surfaces to any desired level of accuracy (see Figure 4.3). The starting shape in our case is a tetrahedron. We specify the four vertices:

```
// tetrahedron  vertices
var va = vec4(0.0, 0.0, -1.0,1);
var vb = vec4(0.0, 0.942809, 0.333333, 1);
var vc = vec4(-0.816497, -0.471405, 0.333333, 1);
var vd = vec4(0.816497, -0.471405, 0.333333,1);
```

We can get a close approximation to a sphere by subdividing each facet of the

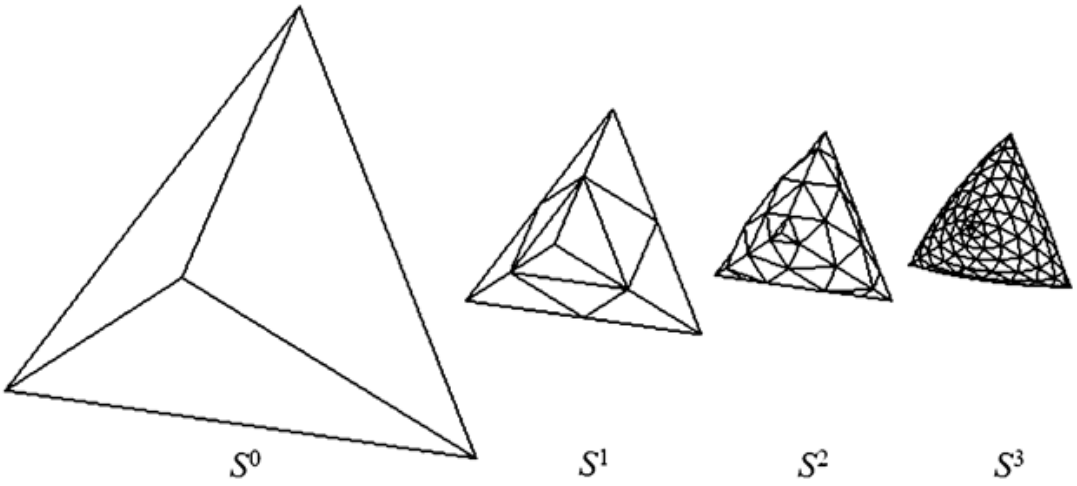


Figure 4.3: Illustration of recursive tethreaedron subdivision. Image courtesy of The-Crankshaft Publishing [13].

tetrahedron into smaller triangles[12]. We compute the bisectors of each side and connect them - to form four equilateral triangles. We initiate the recursion by calling:

```
tetrahedron(va, vb, vc, vd, numTimesToSubdivide);
```

Have a look at the comments in the JavaScript file for further details about the implementation.

Try this: *The interface has two buttons that allow you to increase or decrease the number of subdivision. Observe the effect of changing the subdivisions.*

Vertex Shader: Phong-Blinn Model The vertex shader is described in the `html` file. The shader must output a vertex position in clip coordinates and a vertex colour to the rasterizer. We have already seen in the previous labs how to compute the vertex position (i.e., `gl_Position = projectionMatrix * modelViewMatrix * vPosition;`).

Let's now focus on how we produce the vertex colour. We take as input from the application code the ambient diffuse and specular products and we output a colour - for each vertex - that is the sum of the ambient, diffuse and specular contributions:

[12]Subdivision into triangles ensures the new facets to be flat

[13]`http://what-when-how.com/advanced-methods-in-computer-graphics/mesh-processing-advanced-methods-in-computer-graphics-part-4/`.

```
varying vec4 fColor;
uniform vec4 ambientProduct, diffuseProduct,
    specularProduct;

⋮

fColor = ambient + diffuse + specular;
```

the `ambient` component is simply given by `ambientProduct`:

```
ambient = ambientProduct;
```

The `diffuse` term requires a normal for each vertex, but because triangles are flat, we have sent to the vertex shader the same normal for each vertex (see line 102 of the JavaScript file) and this is used in the vertex shader by declaring:

```
attribute vec4 vNormal;
```

we can now obtain a unit length normal from the `vNormal` through the `normalize` function. The variable `L` describes the direction of the light source (it's again a unit vector obtained through normalising the distance between the light source and the vertex position in eye coordinates). The diffuse term is calculated as:

```
vec3 N = normalize((modelViewMatrix*vNormal).xyz);

vec4  diffuse = max( dot(L, N), 0.0 )*diffuseProduct;
```

The Blinn-Phong model Using the Phong model with specular reflection requires to calculate the dot product between $\mathbf{R}$, the direction of a perfect reflector and $\mathbf{V}$, the direction of the viewer. As seen during lecture we can obtain an interesting approximation by using the unit vector halfway between the viewer vector $\mathbf{V}$ and the light-source vector $\mathbf{L}$:

$$\boldsymbol{h} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \tag{1}$$

$\boldsymbol{h}$ is called the halfway vector and it is calculated in the vertex shader and subsequently used to calculate the specular term:

```
// halfway vector
vec3 H = normalize( L + V );

⋮

vec4  specular = pow( max(dot(N, H), 0.0), shininess )*
   specularProduct:
```

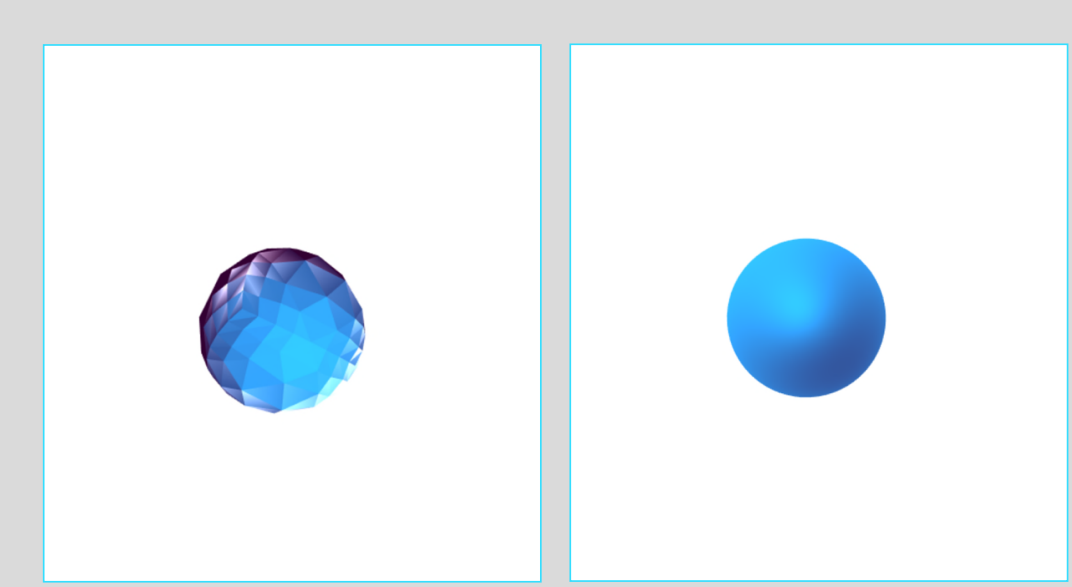


Figure 4.4: Sphere obtained with `numTimesToSubdivide=5`. Left as in the original program. Right improving definition of the normals.

Try this: *Try changing the material property by altering the material diffuse and specular. Which effect do you obtain? why?*

4.3 Exercises

1. Increasing the number of subdivision in the generation of the sphere model is a good way to approximate curve surfaces, however we can still see edges of polygons around the outside of the sphere image.

   This type of outline is called **silhouette edge**. One way to fix this - although it is not a general solution - is to use the actual normals of the sphere for each vertex in our application program. *Hint:* For a sphere centred at the origin, the normal at point $\mathbf{p}$ is simply p.

2. Can you substitute the tetrahedron with a cube? Can you create an animation so that it has three buttons to make it rotate along each axis? Observe the effect of shading.

3. Can you change the diffuse/ambient colour of the sphere?

4. So far you had single light in the environment but in real life there are multiple lights which illuminate the objects and surfaces. Try adding another

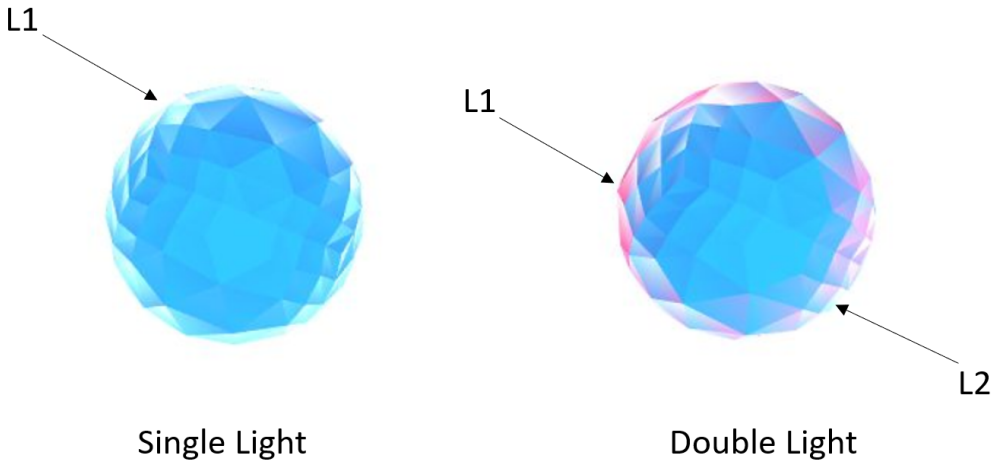


Figure 4.5: Single light source vs Double light source

light in the environment, perhaps with a different colour e.g. red as seen in the Figure 4.5. *Hint: Remember light is additive, so you need to compute the reflections from one light source that includes diffuse reflections and specular reflections and then you compute reflections of the second light source.*

5 Lab5: Texture and Bump Mapping

Texturing or **texture mapping** and bump mapping are powerful ways of adding realism to computer graphics without the need for significantly more vertices and pixels colours being specified. Texture mapping combines both the vertex (geometry) and fragment functions (colours and shading) of the graphics pipeline to allow image values (as pixel colours) to be mapped to fragments when a polygon is rasterised. Bump mapping is a particular way of applying textures that allows to simulate effectively bumps and wrinkles on and object surface in order to achieve a higher level of realism.

5.1 Texture Mapping

Please download the Lab5_WebGL.zip and open the demo available in the `CubeTexture` folder showing a rotating cube. On each face of the cube we have applied a texture as seen in Fig 5.1. Let's have a look at how this texture is loaded and used.

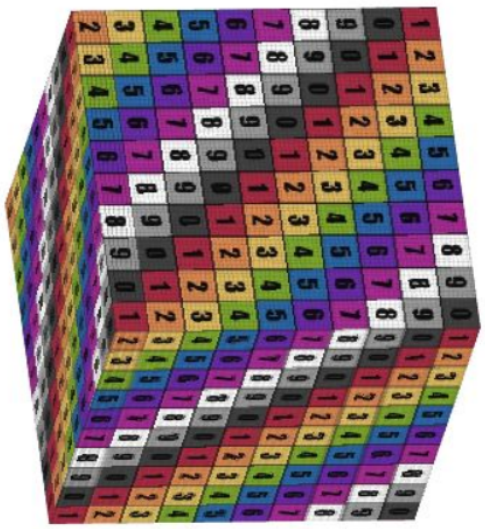


Figure 5.1: Cube texture showing mapping of numerical values from 0-9.

In WebGL texture mapping is done as part of fragment processing. Texture coordinates are handled similarly to normals and colours. They are associated with vertices as an additional vertex attribute and the required texture values are obtained by the rasteriser interpolating the texture coordinates at the vertices

across polygons. Textures can also be generated in one of the shaders (we will see an example later). There are three fundamental steps:

- form a texture image and place it in the texture memory on the GPU.
- assign texture coordinates to each fragment.
- apply the texture to each fragment.

In the `html` file we have added a line that loads the texture file:

```
<img id = "texImage" src = "spirit.jpg" hidden></img>
```

There are many more things to observe in the `textureCube1.html` file, but we will have a look a these later on. For the moment let's focus on the application file i.e., `textureCube1.js`.

The JavaScript file contains the call to `configureTexture()`. This function accepts an image variable and performs the mapping.

We start with creating a texture object (`gl.createTexture()`) and then bind it as two dimensional texture object by calling `gl.bindTexture`. The call to `gl.texImage2D()` assigns a texture and describes to the WebGL system the image characteristics. The prototype of this function is `gl.texImage2D(target, level, internalformat, format, type, image)`. Table 1 describes each parameter.

The key element in applying texture in the fragment shader is the mapping between the location of the fragment and the corresponding location within the texture image. The most common technique is to treat texture coordinates as vertex attributes, similarly as we have done for the vertex colours, we would then pass these coordinates to the vertex shader and let the rasteriser interpolate the vertex coordinates to fragment texture coordinates. Our example is fairly straightforward:

```
var texture;
var texCoord = [
  vec2(0, 0),
  vec2(0, 1),
  vec2(1, 0),
  vec2(1, 1)
];
```

We add an array that holds the texture coordinates and then we fill the texture coordinates at the end of each quadrilateral face in the `quad()` function.

We also need to perform initialisation so that we can pass to the vertex shader the texture coordinates as vertex attribute with the identifier `vTextCoord`.

How to run things locally As we have seen so far, if we simply use procedural geometries and don't load any textures, webpages work straight from the

Parameters	target	Specifies the `gl.TEXTURE_2D` or `gl.TEXTURE_CUBE_MAP`.
	level	Level 0 is the base image level and level n is the nth mipmap reduction level.
	internalformat	Specifies the internal format of the image.
	format	Specifies the format of the texel data. This must be specified using the same value as *internalformat*.
	type	Specifies the data type of the texel data.
	image	Specifies an image object containing an image to be used as a texture.
Return value	None	
Errors	`INVALID_ENUM`	target is none of the above values.
	`INVALID_OPERATION`	No texture object is bound to target.

Table 1: `texImage2D()` function description

file system by simply double-clicking on HTML file. However, if we try to load models or textures from external files, due to browsers' same origin policy security restrictions, loading from a file system will fail with a security exception.

There are two ways to solve this:

1. Run files from a local web server. We **recommend** you to use this option. This allows you to access your page as:
   `http://localhost/yourFile.html`.

2. Change security for local files in a browser. If you use this option, be aware that you may open yourself to some **vulnerabilities** if using the same browser for a regular web surfing. You may want to create a separate browser profile / shortcut used just for local development to be safe. This allows you to access your page as:
   `file:///yourFile.html`

Run a local server: Many programming languages have simple HTTP servers built in. They are not as full featured as production servers such as Apache or NGINX, however they should be sufficient for testing your WebGL application.

On the lab machines there is Python installed and therefore it is enough to run this from a command line (from your working directory!):

```
//Python 2.x
python -m SimpleHTTPServer

//Python 3.x (recommended)
python -m http.server
```

This will serve files from the current directory usually at localhost under port 8000, i.e., in the address bar type:
`http://localhost:8000/`

Shaders The **vertex shader** is unchanged from the other examples we have already seen in previous labs, except for the addition of the `vTexCoord` as input and the `fTexColor` as output. This value is passed to the **fragment shader**. The key to putting everything together is a new type of variable defined in the fragment shader called **sampler**. This variable provides access to a texture object including all its parameters. There are two types of sampler variables supported in WebGL: the two dimensional sampler (`sampler2D`) and the cube map (`samplerCube`). What a sampler does is to return a value or sample of the texture image for the input texture coordinate. We link the texture object created in the application to the sampler in the fragment shader by calling:

```
gl.uniform1i(gl.getUniformLocation(program, "texture"),
    0);
```

where `program` contains the fragment and vertex shaders 1.1.3 and the `texture` is the name of the sampler in the fragment shader. The fragment shader is almost trivial: it uses the texture to fully determine the colour of each fragment.

Try this: *In the fragment shader we can also multiply the vertex colours from the application by the values in the texture image resulting in a blended effect. Try this line of code in the fragment shader to calculate the fragment colour:*

```
gl_FragColor = fColor * texture2D(texture, fTexCoord);
```

and also try to alter the value of the colours that the application chooses for each vertex. What happens to the cube?

Note: The browser tends to cache scripts, so sometimes your modification might not be immediately visible in what the browser is executing. You may need to empty the cache and force a hard reload. In Chrome one of the possible ways to achieve this is to hold down `Ctrl` and click the `Reload` button.

MipMap You may have noticed a call to `gl.generateMipmap` back when we load the texture. What is that for? A mipmap is a collection of progressively smaller images, each one 1/4th the size of the previous one. Generally each smaller level is just a bilinear interpolation of the previous level and that's what `gl.generateMipmap` does. It looks at the biggest level and generates all the smaller levels for you. `gl.generateMipmap` must be called after the texture has been populated with texImage2D and will automatically create a full mipmap chain for the image. Of course you can supply the smaller levels yourself if you prefer [14].

You can choose how WebGL interpolates by setting the texture filtering for each texture. There are 6 modes

- `NEAREST` = choose 1 pixel from the biggest mip
- `LINEAR` = choose 4 pixels from the biggest mip and blend them
- `NEAREST_MIPMAP_NEAREST` = choose the best mip, then pick one pixel from that mip
- `LINEAR_MIPMAP_NEAREST` = choose the best mip, then blend 4 pixels from that mip
- `NEAREST_MIPMAP_LINEAR` = choose the best 2 mips, choose 1 pixel from each, blend them
- `LINEAR_MIPMAP_LINEAR` = choose the best 2 mips. choose 4 pixels from each, blend them

5.2 Bump Mapping

In the `BumpMapping` folder you will find a second demo. Bump mapping is a "texturing" technique that can give the appearance of great complexity to an image without increasing the geometric complexity. Unlike texture mapping, bump mapping will show changes in shading as the light source or the object moves.

Our example is a single square (at $y = 0$) with a light source above the plane that rotates. Let's start with the application code.

The array `data` in the `bumpMap2.js` loads the elevation data that is stored in the `honolulu.js` file. The normal map is then computed starting from this data by taking the difference to approximate the partial derivatives for two of the components and using 1.0 for the third component to form the array `normalst`.

[14] For more on MipMaps see e.g. `https://www.youtube.com/watch?v=8OtOFN17jxM` or `https://www.youtube.com/watch?v=LIqaM66cYHs`, here the original paper.

```
// Bump Map Normals
var normalst = new Array()
  for (var i=0; i<texSize; i++)  normalst[i] = new Array
     ();

  for (var i=0; i<texSize; i++)
    for ( var j = 0; j < texSize; j++)
      normalst[i][j] = new Array();
  for (var i=0; i<texSize; i++)
    for ( var j = 0; j < texSize; j++) {
      normalst[i][j][0] = data[i][j]-data[i+1][j];
      normalst[i][j][1] = data[i][j]-data[i][j+1];
      normalst[i][j][2] = 1;
}
```

The values are then stored into a colour image, that we need to subsequently scale to the interval $(0.0, 1.0)$ (see `//Scale to texture coordinates` in the code). After rescaling the values the `normalst` array is transformed into a `Uint8Array` called `normals`. This array is then sent to the GPU by building a texture object with the same function we have seen in the previous section: `configureTexture()`.

Vertex shader In order to act upon how the object appears we must transform both the eye vector and the light vector. We perform this transformation in the vertex shader. The required transformation matrix is composed by the normal, tangent and binomial vectors. In our case, we have sent through the application a normal that is constant (declared as a uniform variable in the shader) and the tangent that -likewise- is a constant and can be any vector in the same plane as our square.

The normal and tangent are specified in object coordinates and therefore need to be transformed to eye coordinates. We can then use the transformed normal and tangent to give the binormal in eye coordinates and finally use these three vectors to transform the view and light vector to texture space:

```
/* light vector in texture space */

L.x = dot(T, eyeLightPos-eyePosition);
L.y = dot(B, eyeLightPos-eyePosition);
L.z = dot(N, eyeLightPos-eyePosition);

L = normalize(L);

/* view vector in texture space */
```

```
V.x = dot(T, -eyePosition);
V.y = dot(B, -eyePosition);
V.z = dot(N, -eyePosition);

V = normalize(V);

gl_Position = projectionMatrix*modelViewMatrix*vPosition
   ;
```

Fragment Shader At this point, the strategy for the fragment shader is to send normalised perturbed normals as a texture map from the application to the shader and we use it as a normal map. The fragment shader code is as follows:

```
precision mediump float;

varying vec3 L;
varying vec3 V;
varying vec2 fTexCoord;

uniform sampler2D texMap;
uniform vec4 diffuseProduct;

void main()
{
    vec4 N = texture2D(texMap, fTexCoord);
    vec3 NN =  normalize(2.0*N.xyz-1.0);
    vec3 LL = normalize(L);
    float Kd = max(dot(NN, LL), 0.0);
    vec4 ambient = vec4(0.2, 0.2, 0.2, 0.0);
    gl_FragColor = ambient + vec4(Kd*diffuseProduct.xyz,
       1.0);
}
```

The values are first scaled to the interval (-1.0, 1.0) and then we compute fragment lighting by adding the ambient component to the `diffuseProduct`. The diffuse product has been calculated in the application as well and is the product of light component and material component (see line 154 in the application file [15]).

This is a very quick introduction to bump mapping, there is much more to be added to this, you can find more information here.

[15] `var diffuseProduct = mult(lightDiffuse, materialDiffuse)`

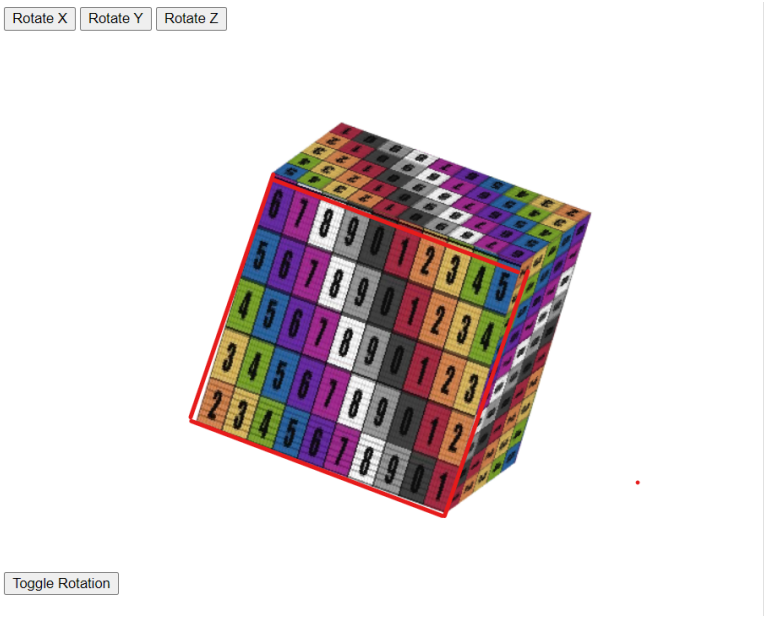


Figure 5.2

5.3 Exercises

1. In the `CubeTexture` demo try changing the texture image with a different one. And also try experimenting with textures having different resolutions and aspect ratio.

2. In the `CubeTexture` demo can you change the texture mapping to use only part of the (u, v) range, for example (0.0, 0.5), like in Figure 5.2?

3. Try manipulating the depth information defined in `honolulu256` and see its effects.

4. Can you import the `cushion.png` image and use that one as bump map in our exercise? *Hint: this will require a bit of work, however the essential steps are: 1. Remove the code for the generation of the normal map - as we will be loading this from file; 2. Create a new texture and bind it to your* ***cushion.png*** *bump map by calling* `texImage2D()` *and; 3. Call* `gl.texParameteri()` *to specify the texture parameters.*

5. *(Optional)* Try to use the `cushion.png` file as a bump map and assign it to each face of a cube.

6 Lab6: Particle Systems

The graphical objects we have worked with so far have limited capabilities in terms of creation of some special effects such as fireworks. When trying to simulate such phenomena the general approach in Computer Graphics is to attempt to reproduce the general quality of motion rather than with precisely controlling the position and orientation of each object in each frame. Such is the case with *physically based animation*. In *physically based animation*, forces are typically used to maintain relationships among geometric elements. Note that these forces may or may not be physically accurate. The concern is not necessarily accuracy, but perception and believability, which is sometimes referred to as being physically realistic.

The quest for the programmer is, nevertheless, to provide a procedure that does as little computation as possible while still providing the desired realistic effects. The advantage of using physical models is that we are relieved of low-level specifications of motions and only need to be concerned with specifying high-level relationships or qualities of the motion.

One particular approach to *physically based animation* are particle systems. A particle system is a collection of a large number of point-like elements. They are often used in computer graphics to model natural phenomena like fireworks, smoke, water, vapour, etc. Particle systems are often animated as a simple physical simulation. Because of the large number of elements typically found in a particle system, simplifying assumptions are used in both the rendering and the calculation of their motions.

A particle system is essentially a set of point masses whose kinematic behaviour has been modelled. Usually particles are given positions and motion vectors, but they may also have mass, have collisions and follow physical rules. We use these rules to write equations that can be solved numerically to obtain the state of these particles at each step. Each particle will be displayed in some way. The simplest thing to do is to show each particle as a point with colour. To produce more convincing effects, particles are usually associated with a quadrilateral and bound to a **texture**. The textures might be quite small, e.g., 64×64, and be given colour and transparency and a lifetime parameter. This allows complex phenomena such as fire and smoke to be modelled. One such phenomenon is illustrated in `ParticleDiffusion` demo in the labs.

Please download the Lab6_WebGL.zip and open the demo available in the `ParticleDiffusion` folder which illustrates a particle system where each particle moves independently from the others in a random direction and leaves a trace behind itself as seen in 7.1. This trace is diffused by the fragment shader, so that at each step the colour starts fading. The effect we have creates is similar to that of a drop of ink diffused in water.

These kind of particle systems are also often called Agent Based Models (ABM)

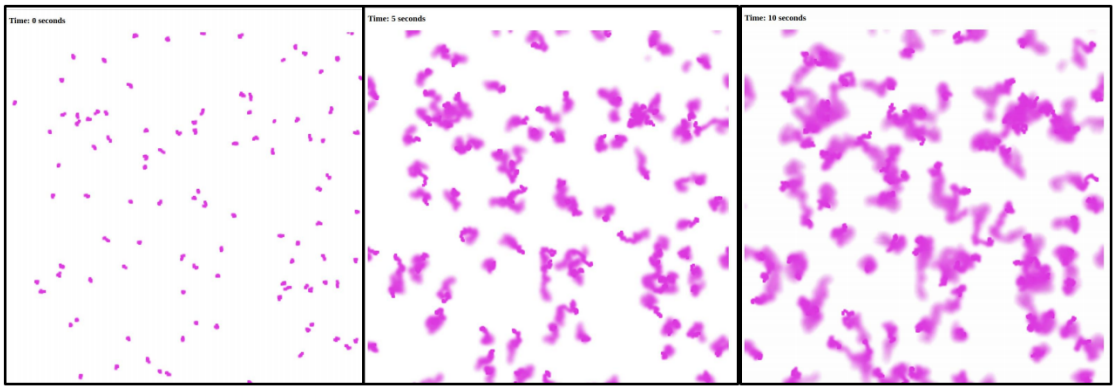


Figure 6.1: Particle diffusion at (left) 0 secs (middle) 5 secs (right) 10 secs

as each of the particles can be programmed with individual behaviours and properties (e.g., different colours, different geometry, different rules etc.), and also key to this kind of simulation is the fact that particles can also interact with each other and the environment. This is one of the most commonly used mechanisms in games to drive the behaviour of non playing characters (NPCs).

In our first example, the agents will not interact with the environment and will simply move randomly. However, we need to remember the previous position of the agent, as it will leave a trace behind it. This will be implemented with a technique called render-to-texture. Essentially, we will render not just on to the canvas, but also save this image in a texture and then use this information in the fragment shader to incrementally create the next frame. Before we dive into the code, there are a couple of concepts we need to introduce:

- rendering to texture.
- buffer ping-ponging.

6.1 Render-to-texture

In a graphics system, the image is formed in the frame buffer. The contents of this frame buffer are displayed on the output device under the control of the window system or, for WebGL, under the control of the browser. In either case the physical display is refreshed at a constant rate, independent of the rate at which the graphics system fills the frame buffer.

With the repetitive clearing and redrawing of an area of the screen and the lack of coupling between when the objects are drawn into the frame buffer and when the frame buffer is re-displayed by the hardware, it could happen that only part of the object may be in the buffer and hence only part of it would be displayed. This model is known as **single buffering**.

Double buffering provides one solution to these problems. Suppose that we have two buffers at our disposal, conventionally called the front and back buffers. The front buffer is always the one displayed, whereas the back buffer is the one into which we draw. WebGL requires double buffering. A typical rendering starts with a clearing of the back buffer, rendering into the back buffer, and finishing with a buffer swap.

In WebGL, the refresh is under the control of the browser. Although the browser will continue to refresh the display at a constant rate, it will not replace the part of the display corresponding to the framebuffer until the application signals that it can do so. One way is when the application finishes executing its JavaScript as in the static examples of Lab 1. While another approach to controlling when the browser displays the framebuffer in dynamic applications using the function `requestAnimationFrame()` as we have been doing since Lab 2 (see 2.2.2).

Even using only the WebGL framebuffer, can be somewhat restrictive. We can create additional off-screen framebuffers and this will open up many new possibilities, including for example creating images in which the textures can be animated, or creating images with shadows. These extra buffers are called Frame Buffer Objects (FBOs).

Frame Buffer Objects allow WebGL programmers to render images to other memory location other than the default framebuffer. An FBO is handled entirely by the graphics system and allows for post-processing of rendered images, rendering a 3D scene to a texture and also operating on the rendered image with another shader. Because FBOs exist in the graphic memory we can have multiple buffers but also transfer among these buffers without incurring in the slow transfer rate between the CPU and GPU, or the overhead of the local window system.

With FBO, you can ask WebGL to render to receivers (other than the monitor) such as:

- Textures
- Render buffer: these are created and used specifically with FBOs. They support fewer operations than textures, but support natively Multisampling (MSAA)[16].

We create and bind frame buffers in the following fashion:

```
var framebuffer;

⋮
```

[16]Here the link to the WebGL Wiki for more information on the use of render buffers `https://www.khronos.org/opengl/wiki/Renderbuffer_Object`.

```
framebuffer = gl.createFramebuffer();
gl.bindFramebuffer( gl.FRAMEBUFFER, framebuffer);
```

At this point, this frame buffer is the one into which the geometry will be rendered. However, due to the fact that is an off-screen buffer, anything rendered onto this buffer will not be displayed. In our demo, we will be rendering onto textures. The first step will be the creation of texture object - similar to what we have done in the previous lab (see 5.1). However, in this case we will specify a `null` image in gl.texImage2D. Nevertheless, this information is sufficient to allocate memory for the texture. We attach this texture object to our FBO via the call:

```
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.
   COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture1, 0);
```

where the parameter `gl.COLOR_ATTACHMENT0` serves to identify the attachment as a colour buffer. We can also verify that the FBO has been set up correctly by calling:

```
var status = gl.checkFramebufferStatus(gl.FRAMEBUFFER);
if(status != gl.FRAMEBUFFER_COMPLETE)
  alert('Framebuffer Not Complete');
```

Once we have completed the rendering onto the texture we can resume rendering on the display by unbinding our FBO (set to `null`):

```
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
```

Buffer ping-ponging We have previously mentioned double buffering as a mechanism to ensure smooth display of content by rendering onto a buffer (back buffer) that is not visible to the user and then swapping this with the buffer that is being displayed (front buffer). With two FBOs, we can accomplish a similar transfer by rendering onto a texture map and then performing a new render-to-texture that uses information available in the texture to generate a new texture and so on and so forth switching (ping-ponging) between the two FBOs. In our implementation, the result will be akin to diffusing the result of the initial rendering over time. This is apparent in our code in the rendering loop, where we define a `flag` variable and we use this variable to decide which of the two textures we will be rendering onto

```
if(flag) {
  gl.bindTexture(gl.TEXTURE_2D, texture1);
  gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.
     COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture2, 0);

}
```

```
else {
  gl.bindTexture(gl.TEXTURE_2D, texture2);
  gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.
     COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture1, 0);
}
```

6.1.1 Shaders

The most interesting part of this program is the shaders in the `html` file. You'll notice that we have defined two sets of shaders!

In the application, we will create two different program objects (`program1` and `program2`) and associate a different set of shader to each. One program will render the diffuse texture, while the other program will be used to draw the points.

The first vertex shader is simply rendering with the texture, so it simply passes the texture coordinates on to the fragment shader.

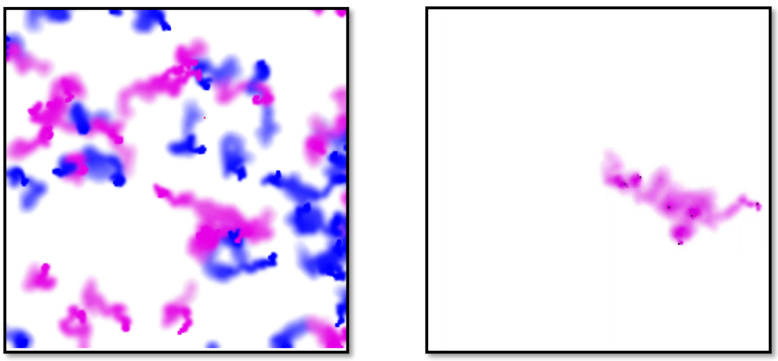


Figure 6.2: Left. Simulation using two different colours. Right. Simulation using a single origin point for spawning particles.

```
<script id="vertex-shader1" type="x-shader/x-vertex">

attribute vec4 vPosition1;
attribute vec2 vTexCoord;

varying vec2 fTexCoord;

void main()
{
  gl_Position = vPosition1;
  fTexCoord = vTexCoord;
```

```
}
</script>
```

The fragment shader will perform the diffusion of the texture:

```
<script id="fragment-shader1" type="x-shader/x-fragment"
   >


precision mediump float;

uniform sampler2D texture;
uniform float d;
uniform float s;

varying vec2 fTexCoord;

void main()
{
  float x = fTexCoord.x;
  float y = fTexCoord.y;
  gl_FragColor = (texture2D( texture, vec2(x+d, y))
  +texture2D( texture, vec2(x, y+d))
  +texture2D( texture, vec2(x-d, y))
  +texture2D( texture, vec2(x, y-d)))/s;

}
</script>
```

This shader will take the texture coordinates received from the rasterizer and calculate the colour for the fragment as the sum of the texture value at four points around the texture coordinates point and scale them by a factor `s`. The value `d` is the distance between two texels (Texture Elements) in the texture - so this depends on the resolution we are using for this application, and `d` is equal to `1/resolution` (this is set in the application).

Try this: *What happens if you change the value of* ***`diffuse`*** *to something like* 3.0*?*

The second sets of shader is not very different from what we have seen so far in the labs. The vertex shader simply takes in the points coordinates and sets the point size:

```
<script id="vertex-shader2" type="x-shader/x-vertex">

attribute  vec4 vPosition2;
uniform float pointSize;

void main()
{
  gl_PointSize = pointSize;
  gl_Position = vPosition2;
}
</script>
```

While the fragment shader is totally trivial as it sets the fragment equal to the colour we have set in the application (magenta in our case).

```
<script id="fragment-shader2" type="x-shader/x-fragment"
   >

precision mediump float;

uniform vec4 color;
void
main()
{
  gl_FragColor = color;

}
</script>
```

6.2 Interacting Particles

In the `ParticleSystem` folder you will find another demo. This demo emulates a simple particle system that can be easily expanded to more complex behaviour. Each particle is defined as:

```
function particle(){

  var p = {};
  p.color = vec4(0, 0, 0, 1);
  p.position = vec4(0, 0, 0, 1);
  p.velocity = vec4(0, 0, 0, 0);
  p.mass = 1;
```

```
  return p;
}
```

and the particle system is described through an array of particles. In this system, we have implemented a collision function that keeps the particles inside the initial axis-aligned box. If the particle has crossed one of the sides we treat this bounce as a reflection, and we will change the sign of the velocity in the normal direction. The variable `coef` indicates the coefficient of restitution and, if less than 1.0 will determine a loss of speed every time the particles hit the side of the box.

The code has also a `forces()` function, emulating forces acting upon the particles. The two buttons in the interface allow to switch these forces on and off.

```
var forces = function( ParticleI)
{
    var ParticleK;
    var force = vec4(0, 0, 0, 0);
    if ( gravity) force[1] = -0.5;/* simple gravity */
    if ( repulsion )
        for ( var ParticleK = 0; ParticleK <
           numParticles; ParticleK++ ) { /* repulsive
           force */
            if ( ParticleK != ParticleI ){
               var t =  subtract(particleSystem[
                  ParticleI].position, particleSystem[
                  ParticleK].position); //vector that
                  represents the distance between
                  particles
               var d2 = dot(t, t); //calculate square of
                   distance
               //Since the projection of a vector on to
                  itself leaves its magnitude unchanged,
               //the dot product of any vector with
                  itself is the square of that vector's
                  magnitude.
               force = add(force, scale(0.01/d2, t));
                }
        }
    return ( force );
}
```

The first force applied to the particle is gravity (in a very simplified fashion!). Assuming that all particles have the same mass, we add a gravitational term along

the y direction. If the *repulsion* option is selected we apply a repulsion force which uses the distance between pairs of points to calculate a force proportional to the inverse square distance.

And when updating the particle system state, we add either the gravity or the repulsive force as follows:

```
var update = function(){
 for (var i = 0; i < numParticles; i++ ) {
            particleSystem[i].position =
            add( particleSystem[i].position, scale(speed
               *dt, particleSystem[i].velocity));
            particleSystem[i].velocity =
            add( particleSystem[i].velocity, scale(speed
               *dt/ particleSystem[i].mass, forces(i)));
        }

⋮

}
```

We use an update function to compute forces and update the particle positions and velocities. Once forces on each particle are computed we apply the Euler integration [17]. The positions are updated using the velocity and the velocity is updated by computing the forces on that particle. After this we use the collision function to keep all particles inside the box.

6.3 Exercises

1. In the `ParticleDiffusion` try changing the code so that only half of the particle gets rendered with the magenta colour and the other half is rendered as a different colour (e.g. blue) to obtain an effect similar to the one in Figure 6.3 left.

2. In `ParticleDiffusion` the initial position of the particles is assigned randomly. Can you create a single spawning point instead? See Figure 6.3, Right. *Hint: You need to change: 1. how the points are generated (starting form the same starting point and also - perhaps - adding an interval between the creation of two subsequent particles) 2. the particle behaviour, it might be appropriate to have a list of the particles that have been released and update only their position (i.e. if a particle hasn't been released yet there is no point in updating its position).*

[17] See this link for more information.

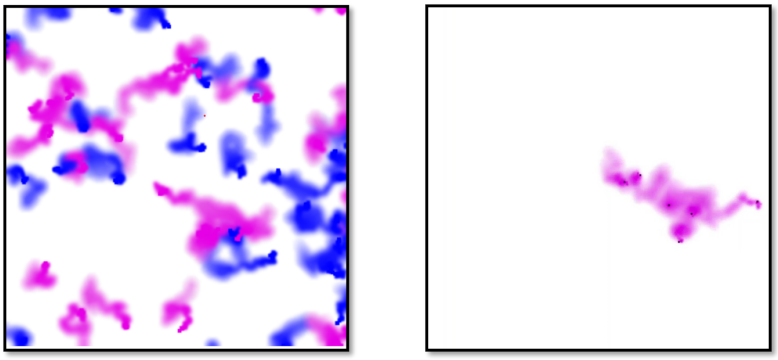


Figure 6.3: Left. Simulation using two different colours. Right. Simulation using a single origin point for spawning particles.

3. *(Optional)* In `ParticleSystem` can you emulate an attractive force, for example having the particles attracted to the mouse position? See `https://www.playfuljs.com/particle-effects-are-easy/` for an example.

4. *(Optional)* In `ParticleSystem` can you emulate a flocking behaviour for the particles? See `https://p5js.org/examples/hello-p5-flocking.html` for an example.

7 Lab7: Introduction to Three.js Library

7.1 Introduction to Three.js

Up to this point we have developed a good understanding of many important aspects of WebGL. However, you might have realised that from one project to another there are a lot of repetitions and, as it is natural, people have come up with ways to save themselves from writing hundreds of lines of code and wrapped their work in libraries that could be used for general purposes. There are quite a few open source libraries built for WebGL that aim at implementing high-level developer friendly features, like for example Babylon.js or A-Frame or Copperlicht. Each one of these has their own way of implementing things. The library we will use in this and the next lab is called **Three.js**. It was first created by Ricardo Cabello Miguel, a programmer based in Barcelona (Spain), and has evolved quite successfully in the last 10 years. This library is one of the most popular open source library for WebGL, it is well maintained and offers several built-in object types and handy utilities.

Among the most salient features Three.js:

- abstracts many details related to the 3D rendering process;
- is object-oriented;
- is feature-rich (i.e., contains many pre-built objects);
- supports interaction;
- provides support for the maths (e.g., matrix multiplication, vector operations, projections etc.)

However, there are also limitations to this library, like for example the absence of support for physics emulation, hence you might need to use different libraries in order to implement this [18].

7.2 Setting Up Three.js

The first thing you will need to do is to get the latest Three.js package from GitHub (available here). You can download the archive, extract this to a directory of your choice. In the zip folder there is a minified version of the JavaScript located in *build/Three.js*, or the full source under *src/*. Feel free to explore the folder - especially the *examples* folder has a lot of interesting examples and useful classes (we will actually use some of the classes in the *examples* folder). It is essential to

[18] Some examples of physics libraries are Oimo.js or enable3d etc.

maintain the library directory structure as references to the same script will only be loaded once as long as their absolute paths are exactly the same.

The full folder for today's lab Lab7_WebGL.zip is available to download the from the `/modules/cs324`. Please notice that we have already added the content of Three.js *build* folder under the *Common* folder. The file might require some time to download. On Moodle under the *Practical Sessions/Labs* section is available a "Light" version of the lab zip, please if you download this version notice that you need to follow the instructions in the `README.txt` file in order to be able to run the demos.

One important note on Three.js: Three.js has a policy of not worrying about backward compatibility. They expect you to download the code and put it in your project. So you might encounter a great number of compatibility issues when looking for online code as its behaviour will probably be related to the specific version of Three.js that the author has used. **These labs have been tested using the r134, if not otherwise specified.** When upgrading to a newer version you can read the migration guide to check if changes are needed.

7.3 Getting started: Three.js Cube

The first demo we will look at today is `SceneSetup`. Although not strictly necessary for this particular scene, we suggest to run the server from the main folder once extracted, so to avoid cross origin errors (as seen in Lab 5.1).

The scene should look quite familiar, we have created a cube, coloured it in red and rendered it. In order to use the Three.js library we need the following elements the application needs:

- to load the Three.js library;
- to create a scene (this is where you'll add objects and lights);
- to add a camera, to allow to view the scene;
- to call the renderer.

We will skip the content of the `.html` file as it should be quite familiar by now. Let's now focus on the content of the `.js` file.

```
import * as THREE from '../Common/build/three.module.js'
    ;

function main() {
  // Get <canvas> element
  const canvas = document.getElementById( "gl-canvas" );
```

The paths in the example code to the default location of three js are wrong atm

```
  // Create a new Three.js scene
  const scene = new THREE.Scene();

  const fov = 75;
  const aspect = 1;
  const near = 0.1;
  const far = 10;
  const camera = new THREE.PerspectiveCamera(fov, aspect
     , near, far);

  camera.position.z = 2;

  // Create the Three.js renderer, add it to our canvas
  const renderer = new THREE.WebGLRenderer({canvas});

  const boxWidth = 1;
  const boxHeight = 1;
  const boxDepth = 1;
  const geometry = new THREE.BoxGeometry(boxWidth,
     boxHeight, boxDepth);

  const material = new THREE.MeshBasicMaterial({color: 0
     xFF0000});
  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  renderer.render(scene, camera);
}
```

The first instruction is used to import the three.js library, and in the following we will be able to call the classes contained in the library by calling `THREE.` followed by the specific class. From version r106 the preferred way to use Three.js is via es6 modules [19]. es6 modules can be loaded via the `import` keyword in a script or inline via a `<script type="module">` tag [20]. Paths must be absolute or relative and references to the same script will only be loaded once as long as their absolute

[19] For a quick overview here.

[20] This can be done in the `html` file as follows:

```
<script type="module">
import * as THREE from './resources/threejs/r132/build/three.module.js';
...
...
</script>
```

paths are exactly the same.

The first action we perform is to retrieve the canvas on which we want to draw, as we have done before. And then we create a new scene by calling `new THREE.Scene()`. A Scene in Three.js is the root of a form of scene graph. In Three.js objects exist in a parent-child hierarchy, and the scene is the top level of this hierarchy. Anything you want Three.js to draw needs to be added to the scene. Following this we set up the values that will be passed to our perspective camera. We have already discussed the perspective camera parameters in Lab 3.1 and here we use the same parameters (i.e., `fov, aspect, near, far`). In this scene, we are also moving the camera along the z axis. We could also use a transformation to set the camera position in the following way:

```
camera.position.set( 0, 0, 2);
```

The call `position.set()` can be made on all objects in a scene. Then, we initialize the Three.js renderer object. The renderer is responsible for all Three.js drawing (via WebGL context, of course).

The mesh in the scene is composed of a geometry object and a material. For geometry, we are using a cube created with the Three.js BoxGeometry object. Our material tells Three.js how to shade the object. In this example, our material is of type MeshBasicMaterial (i.e., just a single simple color with a default value of red). Three.js objects have a default position of 0, 0, 0, so our white rectangle will be placed at the origin.

Finally, we need to render the scene. We do this by calling the renderer's render() method, feeding it a scene and a camera.

Try this: *Try modifying the camera position, to see the cube from a different angle. Try changing the cube colour and the cube position.*

Note how Three.js closely mirrors the graphics concepts introduced so far in the Labs, however here we are working with objects (instead of buffers), viewing them with a camera, moving them with transforms, and defining how they look with materials. The advantage is that in 30 lines of code, we have produced exactly the same graphic as our raw WebGL example which would have probably taken 150 lines.

7.4 Three.js Responsive Design

One thing you might have noticed is that when we resize the browser window, our image does not resize properly. What we need is to change the `camera` parameters when we the browser window is resized.

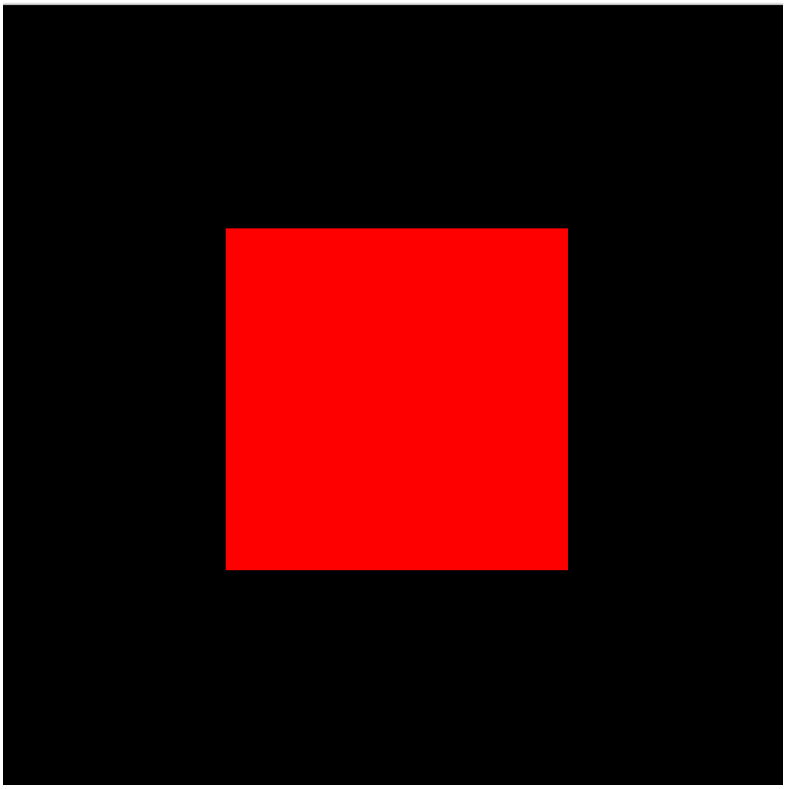


Figure 7.1: Output of Lab-7 `SceneSetup` a red Cube (frontal view) using Three.js

The demo `SceneResize` allows you to do exactly this. The only part that has changed form the previous code is the addition of an event listener.

```
renderer = new THREE.WebGLRenderer({canvas});

⋮

// listen to the resize events
window.addEventListener("resize", onWindowResize);

⋮

function onWindowResize() {
    const aspect = window.innerWidth / window.
       innerHeight;
    camera.aspect = aspect;
    camera.updateProjectionMatrix();
    renderer.setSize( window.innerWidth, window.
       innerHeight );
    renderer.render(scene, camera);
```

```
}
```

In essence, we will create an event listener that will call `onWindowResize()` function (line 42). This function will update the camera aspect ratio, call `updateProjectionMatrix()` - which is a function that we must call every time we change the camera parameter - since we need to re-do the calculation for the projection transformation and finally we call `renderer.setSize(window.innerWidth, window.innerHeight)` to update the renderer parameters and render the scene again.

7.5 Adding Lights and Materials

In order to add a bit more realism to our scene we now need to add lights. Three.js supports several different types of lights. There are ambient lights, which provide constant illumination throughout the scene, regardless of position; point lights, which emanate from a particular position in all directions, and illuminate over a certain distance; spot lights, which emanate from a particular position and in a specific direction, over a certain distance (and look a lot like spotlights you see in the movies); and finally, directional lights, which illuminate in a specific direction but over an infinite distance (and don't have any particular position).

If you want to simulate the sun, for example you might want to select a source that emits in all directions, hence a point light would be appropriate. In our demo `Interaction` you will find that adding lights to a scene with Three.js is fairly simple (in the same demo you will be able also to use the mouse and keyboard for interactivity as explained in the next section).

```
// lights
    // add directional Light
    const dirLight = new THREE.DirectionalLight( 0
       xffffff, 1.1 );
    dirLight.position.set( 0, 10, -5 );
    scene.add( dirLight );
    // add ambient light
    const ambientLight = new THREE.AmbientLight( 0
       xF22222, 0.5 );
    scene.add( ambientLight );
```

We have declared 2 variables: `dirLight` and `ambientLight` and assigned to them a directional and an ambient light respectively. And finally we have added the lights to the scene. The Light object - from which both the ambient and the directional lights are derived, accepts generally 2 optional parameters defining colour (in hexadecimal) and intensity (float).

If you notice the `THREE.DirectionalLight` illuminates our scene from its position (`DirectionalLight.position.set()`).

Just adding the lights is not sufficient to improve our objects' appearance, we need to specify a different material from the one we have seen in the previous demo (i.e., `THREE.MeshBasicMaterial`). We can specify for example `MeshLambertMaterial`, this material takes into account the light sources [21].

Something is still missing to achieve a good level of realism ad this is generating shadows. Shadows are expensive to calculate, so in Three.js these are not calculated by default. We will discuss shadows in the next lab, however at this stage we recommend to experiment with different types of lights and observing their additional properties.

7.6 Interaction/Working with the camera

Our examples so far have made very limited use of interaction. We have used the mouse position as a reference for translating objects, we used controls on the web page to change behaviors in the WebGL scene, but we can go way beyond this. We can click and drag on items in the scene itself, triggering other behaviors. We can also navigate within the scene by manipulating the camera.

You probably take interaction for granted in your experience with games or graphical interfaces in general. However, WebGL is only a drawing system and, as such, has no built-in support for it. You need to build that yourself - as we have seen in previous Labs (see Lab 2.2). Thankfully, Three.js has a number of functions that easily allow us to control the camera throughout a scene.

These controls are located in the Three.js distribution and can be found in the `/examples/jsm/controls` directory. In this section, we'll look in more detail at how to use `TrackBallControls`. Besides this, Three.js also provides a number of additional controls you can use (some examples here). Using these controls, however, is done in a similar manner as the one we will explain here.

Before you can use `TrackballControls`, you first need to include the correct JavaScript file in your application file:

```
import { TrackballControls } from '../Common/examples/
    jsm/controls/TrackballControls.js';
```

And then we have declared a function that will link the camera to the `TrackballControls`, specifies the rotation speed, the zoom speed and the pan speed, and also we have specified specific keys on the keyboard to help controlling better the mouse:

[21] Beside `MeshLambertMaterial`, `MeshPhysicalMaterial`, `MeshStandardMaterial` and `MeshPhongMaterial` are the materials Three.js provides that take light sources into account when rendered.

```
let controls

⋮

function createControls( camera ) {

    controls = new TrackballControls( camera, renderer.
       domElement );

    controls.rotateSpeed = 1.0;
    controls.zoomSpeed = 5;
    controls.panSpeed = 0.8;

    //This array holds keycodes for controlling
       interactions.

    controls.keys = [ 'KeyA', 'KeyS', 'KeyD' ];
}
```

With this object you can control the camera in the following way:

Control	**Action**
Left mouse button and move	Rotate and roll the camera around the scene
Scroll wheel	Zoom in and zoom out
Middle mouse button and move	Zoom in and zoom out
Right mouse button and move	Pan around the scene

And in addition, we can declare a vector that defines the keys for controlling the interactions, `'KeyA','KeyS','KeyD'`. In particular, when the first defined key (i.e., `'KeyA'` in this case) is pressed, all mouse interactions (left, middle, right) perform orbiting; when the second defined key (`'KeyS'`) is pressed, all mouse interactions (left, middle, right) perform zooming and; finally when the third defined key (`'KeyD'`) is pressed, all mouse interactions (left, middle, right) perform panning.

Note that in the `animate()` function we call the `requestAnimationFrame()` and the `renderer.render()` to allow us to swap the front and back buffer and refresh the image on screen. This is necessary every time we want to see the effects of our interaction reflected in the graphics (i.e., every time we modify the camera position we need to render the scene again).

7.7 Exercises

1. Add more than one object to your scene e.g., add a plane and position a cone a sphere and a cube on top of it

2. Create 2 cameras in the previous scene and position an orthographic camera to provide a bird-eye view (top view) of your scene and a perspective one that focuses on your scene from a different angle. Try rendering with either of them. Add an option (i.e., a function) for the user to switch between views using the keyboard and/or mouse.

3. Add a `spotLight` to your `Interaction` scene and try to direct the light source towards one of the side cubes (i.e. left or right). Also you can try making use of another object called `SpotLightHelper` to help you visualise the spotlight cone. *Hint:* ***`THREE.spotLight`*** *has a lot of properties, among them there is one that allows to direct the spotLight towards any* ***`Object3D`*** *that has a* ***`position`*** *property.*

8 Lab8: Loading a 3D model in Three.js and applying illumination and shading techniques

Three.js offers a number of basic geometry objects (e.g., cube, plane, sphere, cone), nevertheless, in many circumstances we might want to create more complex geometries using 3D modelling software e.g., Blender, Maya or 3ds Max. Three.js offers several classes that facilitate importing models in your scene. We will see one of the more popular ones, but depending on the original format of your model, you may need to explore the use of different ones, the list is available here.

Moreover, we will see how to load textures and how to improve on the use of lights seen in the previous lab(s), by adding shadows for increased realism.

The full folder for today's lab Lab8_WebGL.zip is available to download the from the `/modules/cs324`. We have already added the content of Three.js *build* and `examples` and `utils` folders under the *Common* folder. The file might require some time to download. On Moodle under the *Practical Sessions/Labs* section is available a "Light" version of the lab zip, please if you download this version notice that you need to follow the instructions in the `README.txt` file in order to be able to run the demos. Also, you will need to run an http server (see Lab 5.1) from the folder where your demos are located.

8.1 Importing 3D Models in WebGL

A format which is getting more and more attention lately is the glTF format. This format, for which you can find extensive documentation on the KhronosGroup page, focuses on optimizing size and resource usage. A glTF asset may deliver one or more scenes, including meshes, materials, textures, skins, skeletons, morph targets, animations, lights, and/or cameras. This means you can create an entire scene in an external program then load it into three.js.

glTF files come in standard and binary form. These have different extensions:

- Standard .gltf files are uncompressed and may come with an extra .bin data file;
- Binary .glb files include all data in one single file.

Both standard and binary glTF files may contain textures embedded in the file or may reference external textures. Since binary .glb files are considerably smaller, it's best to use this type. On the other hand, uncompressed .gltf are easily readable in a text editor, so they may be useful for debugging purposes.

To use the glTF loader, include the correct JavaScript file:

```
import { GLTFLoader } from '../Common/examples/jsm/
   loaders/GLTFLoader.js';
```

Once you've imported a loader, you're ready to add a model to your scene. The essential steps are: 1) declare a new variable, 2) call the `load()` method indicating the path of your model, and 3) add the model to your scene. In the following code, we have also called other familiar methods - already discussed in Section 7.3 - such as `position, scale` and `rotation`, which allow us to position the imported model in the desired position.

```
const loader = new GLTFLoader();

loader.load( '../GlTF_Loading/minion_alone.glb',
   function ( gltf ) {

    gltf.scene.position.set(-6, 3, -3);
    gltf.scene.scale.set(3, 3, 3);
    gltf.scene.rotation.set(0, Math.PI*(1/4),0);

⋮

    scene.add(gltf.scene);
```

The glTF material system is different from, for example, Blender's own materials or others 3D modelling software. Hence, when importing a model the material imported might not look as you would expect. You could approach this problem in several ways, one option would be to "bake" the material into a texture. Baking in the context of 3D Modelling is when we calculate some data and store the result. For instance, in Blender we can create a very complex material that combines many shaders, image textures, procedural textures, leading to a high number of calculations that the render engine needs to perform. We may also have hundreds of different materials, and each needs to be calculated before we can determine the effects they have on the scene final appearance. However, this becomes very computational intense and almost impossible to achieve in real-time. A common approach is, instead, to "bake" this information beforehand. In the case of materials we can "bake" the final result of the rendering in a texture and load this when rendering in real-time. For example, here the reference to Blender manual on how to achieve this effect.. Another alternative could be to replace the original material with one of the material objects available in Three.js, like for example `THREE.MeshPhongMaterial()`.

In our example, you would **need to add the following lines of code** to our `ModelImport` demo before adding the glTF model to the scene:

```
var myColour = new THREE.Color(0xff00f0); // declaring a
    new colour for the imported object

gltf.scene.traverse( function( node )  {
//this is necessary to traverse the scene
    if (node.isMesh)
      //myColour.copy(node.material.color); //use this
         if you want to preserve the original colour
      var newMaterial = new THREE.MeshPhongMaterial({
         color: myColour});
      node.material = newMaterial;
 });
// after this we can call: scene.add(gltf.scene);
```

You may have noticed that we call the `traverse()` method due to the structure of glTF format [22]. In order to access the material property, we need to traverse its hierarchy. The following lines of code might help you to display on the console the hierarchy for your imported scene:

```
const root = gltf.scene;
scene.add(root);
console.log(dumpObject(root).join('\n'));
```

this functio dumpOb- jects is not defined in Common or any- where else currently

8.2 Loading and Applying a Texture/Bump Map

Adding texture map to a mesh As we have seen in Lab 5, the most basic usage of a texture is when it's set as a map on a material. When we use this material to create a mesh, the mesh will be coloured based on the supplied texture.

Please open the `BasicTexture` demo. In this demo, loading a texture and using it on a mesh is done in the following manner:

```
const texture = new THREE.TextureLoader().load( './
   Resources/Grass002_2K_Color );

⋮

const material = new THREE.MeshBasicMaterial( { map:
   texture} );
```

We first call the texture loader, then we call its load method (providing the path for our texture). This returns a `Texture` object, and subsequently, when

[22] See here for more details on glTF.

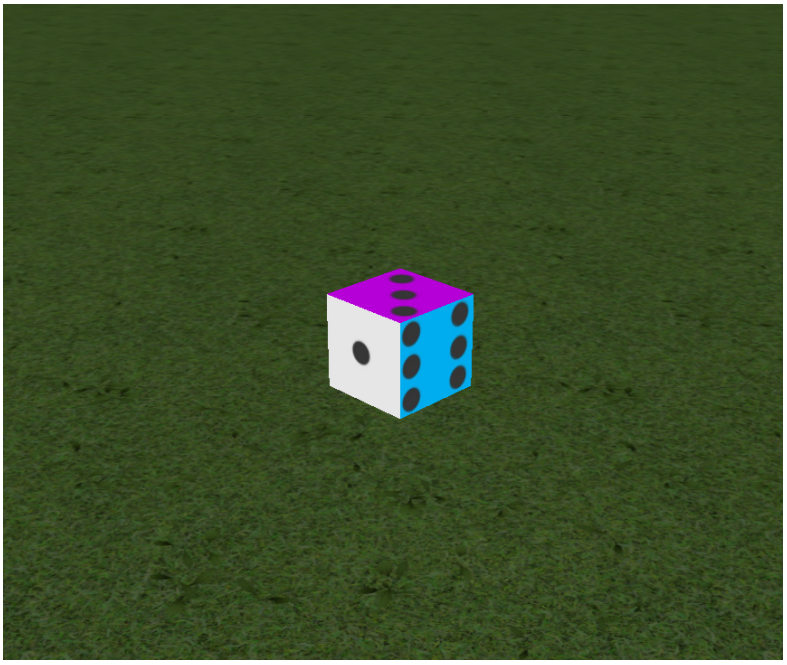


Figure 8.1: Example of importing several textures in Three.js

declaring a new material, we can assign the texture to that material, by specifying its `map` property.

It's important to note that using this method our texture will be transparent until the image is loaded **asynchronously** by Three.js at which point it will update the texture with the downloaded image.

This has the big advantage that we don't have to wait for the texture to load and our page will start rendering immediately. This will be good enough for most scenarios. However, it is possible to wait for the textures to be loaded before creating our Mesh by using a `LoadingManager`, see the Three.js documentation page for more information.

If you would like to upload more than one texture, this is possible depending on the geometry. `BoxGeometry` can use 6 materials one for each face. `ConeGeometry` can use 2 materials, one for the bottom and one for the cone. `CylinderGeometry` can use 3 materials, bottom, top, and side. For other cases you will need to build or load custom geometry and/or modify texture coordinates.

Try This: *There is a cube in the centre of the* `BasicTexture` *demo. You can declare an array of six different materials, that are mapped each to a different texture, and then pass this array as material to your* `cubeMesh`*. You can use*

the dice textures available in the `Resources/cube` *sub-folder as in the example in Figure 8.1.*

Adding Bump and Normal Map As already discussed in Lab 5 bump maps and normal maps are a useful mean through which we can increase realism of our models without altering the object's geometry and keep the vertex count low at the same time. These maps just use the lights from the scene to create fake depth and details. The bump map is generally a grayscale image where the pixel intensity defines the height of the bump. This means that a bump map only contains the relative height of a pixel. To achieve a higher level of realism we can also use normal maps, which represent a map of the alterations we perform on the direction of the normals in our mesh's surface. Three.js offers an option to load either of these.

In the demo `BumpMap` you will find the following lines of code in the javascript file, that allow us to add the bump map to our cube:

```
const cubeSize = 5;
const cubeGeo = new THREE.BoxGeometry(cubeSize, cubeSize
    , cubeSize);

const textureBrick = new THREE.TextureLoader().load('../
    Resources/terracotta/Bricks_Terracotta.jpg')

⋮

 const bumpTexture = new THREE.TextureLoader().load('../
     Resources/terracotta/Bricks_Terracotta_002_height.
     png');

const cubeMaterialBump = new THREE.MeshPhongMaterial({
    map: textureBrick, bumpMap: bumpTexture, bumpScale:
    1});
```

The steps to follow are the following: we declare our geometry, load the texture and also load the bump map in the same way as for the texture. Finally, when declaring a new material we can assign the `bumpTexture` to the `bumpMap` property and we can specify the scale of the bump map with the `bumpScale` property. Notice that a negative number of `bumpScale` will have the effect of inverting the height. Observe the difference between the two cubes in Figure 8.2.

As an alternative to a bump map, as mentioned before, we can also use normal maps. In this case we would have to modify a different property of our `THREE.MeshPhongMaterial`, the `normalMap` property, and similarly to what we have seen for the bump map, we can also adjust the scale of the map through

the `normalScale` property: this will accept a `THREE.Vector2`. See the Three.js documentation page here for more details. **Note that not all materials expose such properties!**

Try this: *In the `./Resources/bricks` folder you will find a normal map for the same brick texture. Try modifying the material properties for the cube on the right: load the `Bricks_Terracotta_002_normal.jpg` file and assign it to the the `normalMap` property.*

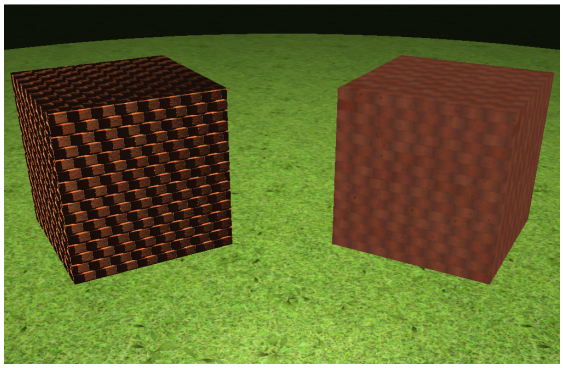


Figure 8.2: Example of adding a bump map in Three.js. The same textures has been applied to both cubes, however a bump map has also been applied to the cube on the left.

8.3 Adding Shadows

Shadows can be a complicated to calculate. In Three.js, one of the solution adopted is to use shadow maps. The way a shadow map works is that for every light that casts shadows, all objects marked to cast shadows are rendered from the point of view of the light. In order to limit the computational load, one common solution is to have multiple lights but only one light being set to generate shadows [23]

In our demo `LightAndShadow`, in order to obtain shadows, besides declaring a light source, we have to add a few more lines of code in our javascript file. First we need to set the `castShadow` property of our light source to true.

```
const color = 0xFFFFFF;
const intensity = 1;
```

[23] Another solution is to use lightmaps and or ambient occlusion maps to pre-compute the effects of lighting offline. This results in static lighting or static lighting hints but at least it's fast.

```
const light = new THREE.PointLight(color, intensity);
light.castShadow = true;
light.position.set(-3, 10, 0);
scene.add(light);
```

Moreover, we need to tell renderer that we want shadows. This can be done by setting the `shadowMap.Enabled` property to true as follows:

```
renderer.shadowMap.enabled = true;
```

And finally, we need to explicitly define which objects cast shadows and which objects receive shadows. In our example, we want the cube and the sphere to cast shadows. You do this by setting the corresponding properties on those objects as done below for the cube:

```
cube.castShadow = true;
cube.receiveShadow = true;
```

This will allow us to add further realism to our scene, as shown in Figure 8.3.

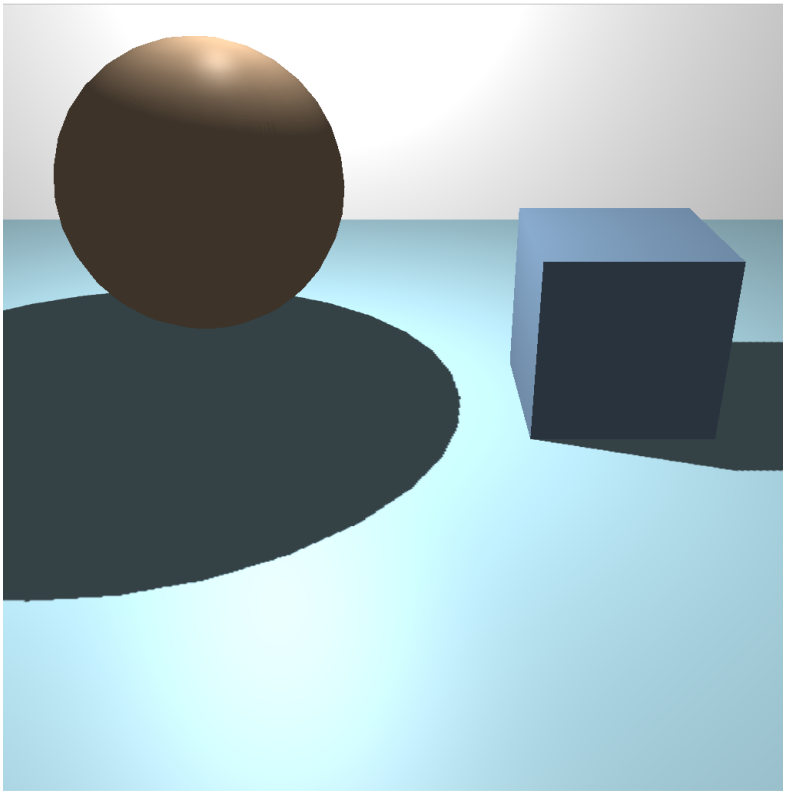


Figure 8.3: Example of casting shadows in Three.js

8.4 Exercises

1. Create or download any Blender model and import it in your WebGL program.

2. It is possible to improve on the `BasicTexture` demo by adding a skybox to our scene - this is a technique often used in games to create the illusion that the game level is larger than it actually is (See Figure 8.4). In the `Resources` folder, we have provided a sub-folder containing 6 images (i.e., `clouds`). *Hint: You will need to load the six textures and assign these to a `CubeTexture`. Then you need to change the `Scene.background` property and assign the `CubeTexture` to it.*

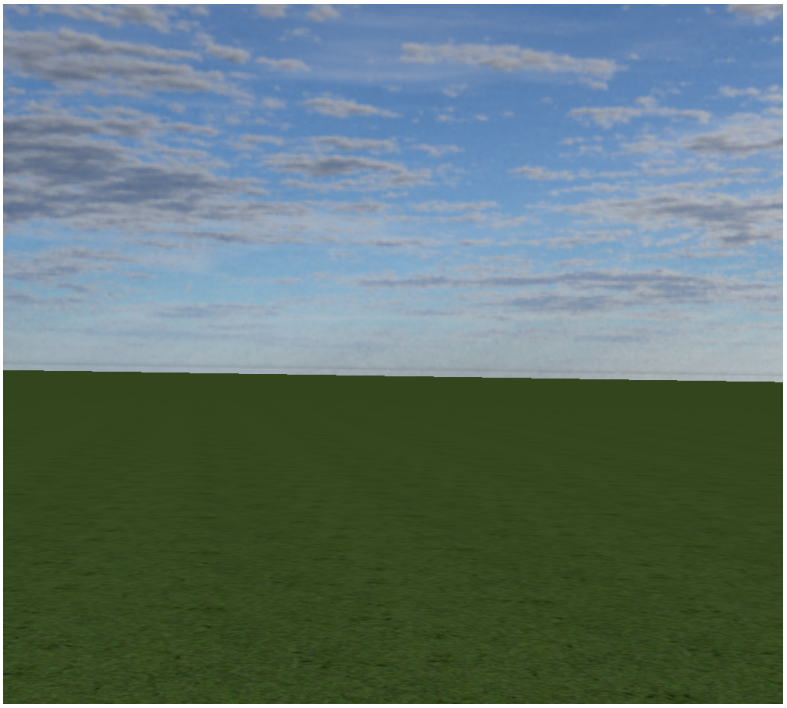


Figure 8.4: Example of using a skybox in Three.js.

3. Modify the `ModelImport` demo, using a spotlight instead of a directional light source, and also add shadows for the imported model.

4. Create a model of the earth by adding a sphere to your scene and then apply the texture and a bump map available in the `Resources/earth` folder. You will also need to use a light source, can you simulate the sun?

Index