Γραφική

Εισηγητής Δρ. Α.Γ. Μαλάμος

Βιβλίο αναφοράς:WebGL Beginner's Guide, Diego Cantor, Brandon Jones

Requirements

1. no installation

2. web browsers:
   a. Firefox 4.0 or above
   b. Google Chrome 11 or above
   c. Safari (OSX 10.6 or above). WebGL is disabled by default but you can switch it on by enabling the Developer menu and then checking the Enable WebGL option
   d. Opera 12 or above

3. updated list of the Internet web browsers http://www.khronos.org/webgl/wiki/Getting_a_WebGL_Implementation

4. Need graphics card

5. current WebGL configuration, http://get.webgl.org/

Rendering types

1. Rendering resources:
    a. software-based

    b. hardware-based rendering (Graphics Processing Unit (GPU))

        hardware-based rendering is much more efficient

        software-based rendering solution can be more pervasive

2. Rendering location
    a. Server-based rendering

    b. client-based rendering

WEBGL Graphics Tech

WebGL has **a client-based rendering** approach: the elements that make part of the 3D scene are usually downloaded from a server. However, all the processing required to obtain an image is performed locally using **the client's graphics hardware**.

In comparison with other technologies (such as Java 3D, Flash, and The Unity Web Player Plugin), WebGL presents several advantages:

- ✓ JavaScript programming
- ✓ Automatic memory management
- ✓ Pervasiveness
- ✓ Performance

Structure of WEBGL

The components we are referring to are as follows:

- **Canvas:** It is the placeholder where the scene will be rendered. It is a standard HTML5 element and as such, it can be accessed using the Document Object Model (DOM) through JavaScript.

- **Objects:** These are the 3D entities that make up part of the scene. These entities are composed of triangles.

- **Lights:** Nothing in a 3D world can be seen if there are no lights.

- **Camera:** The canvas acts as the viewport to the 3D world. We see and explore a 3D scene through it. Camera may be modeled through matrix operations.

Canvas

```
<!DOCTYPE html>
<html>
<head>
  <title> WebGL Beginner's Guide - Setting up
the canvas </title>
  <style type="text/css">
  canvas {border: 2px dotted blue;}
  </style>
</head>
<body>
<canvas id="canvas-element-id" width="800"
height="600">
Your browser does not support HTML5
</canvas>
</body>
</html>
```

Accessing the WebGL context

```
<script>
var gl = null;
function getGLContext(){
var canvas = document.getElementById("canvas-element-id");
  if (canvas == null){
    alert("there is no canvas on this page");
    return;
  }
var names = ["webgl",
          "experimental-webgl",
          "webkit-3d",
          "moz-webgl"];
  for (var i = 0; i < names.length; ++i) {
     try {
       gl = canvas.getContext(names[i]);
     }
     catch(e) {}
     if (gl) break;
  }
  if (gl == null){
    alert("WebGL is not available");
  }
  else{
alert("Hooray! You got a WebGL context");
  }
}
  </script>
```

Context Properties

```
<Script>
var gl = null;
var c_width = 0;
var c_height = 0;
window.onkeydown = checkKey;
function checkKey(ev){
switch(ev.keyCode){
case 49:{ // 1
gl.clearColor(0.3,0.7,0.2,1.0);
clear(gl);
break;
}
case 50:{ // 2
gl.clearColor(0.3,0.2,0.7,1.0);
clear(gl);
break
}
case 51:{ // 3
var color = gl.getParameter(gl.COLOR_CLEAR_VALUE);
// Don't get confused with the following line. It
// basically rounds up the numbers to one decimal cipher
//just for visualization purposes
alert('clearColor = (' +
Math.round(color[0]*10)/10 +
',' + Math.round(color[1]*10)/10+
',' + Math.round(color[2]*10)/10+')');
window.focus();
break;
} } }
```

```
function getGLContext(){
var canvas = document.getElementById("canvas-element-
id");
if (canvas == null){
alert("there is no canvas on this page");
Return; }
var names = ["webgl",
"experimental-webgl",
"webkit-3d",
"moz-webgl"];
var ctx = null;
for (var i = 0; i < names.length; ++i) {
try {
ctx = canvas.getContext(names[i]); }
catch(e) {}
if (ctx) break; }
if (ctx == null){
alert("WebGL is not available"); }
else{
return ctx; } }
function clear(ctx){
ctx.clear(ctx.COLOR_BUFFER_BIT);
ctx.viewport(0, 0, c_width, c_height);
}
function initWebGL(){
gl = getGLContext();
}

  </script>
```

Rendering Geometry

WebGL renders objects following a "divide and conquer" approach. Complex polygons are decomposed into triangles, lines, and point primitives. Then, each geometric primitive is processed in parallel by the GPU through a series of steps, known as the rendering pipeline, in order to create the final scene that is displayed on the canvas.

Vertices and Indices

Vertices are the points that define the corners of 3D objects. Each vertex is represented by three floating-point numbers that correspond to the x, y, and z coordinates of the vertex.

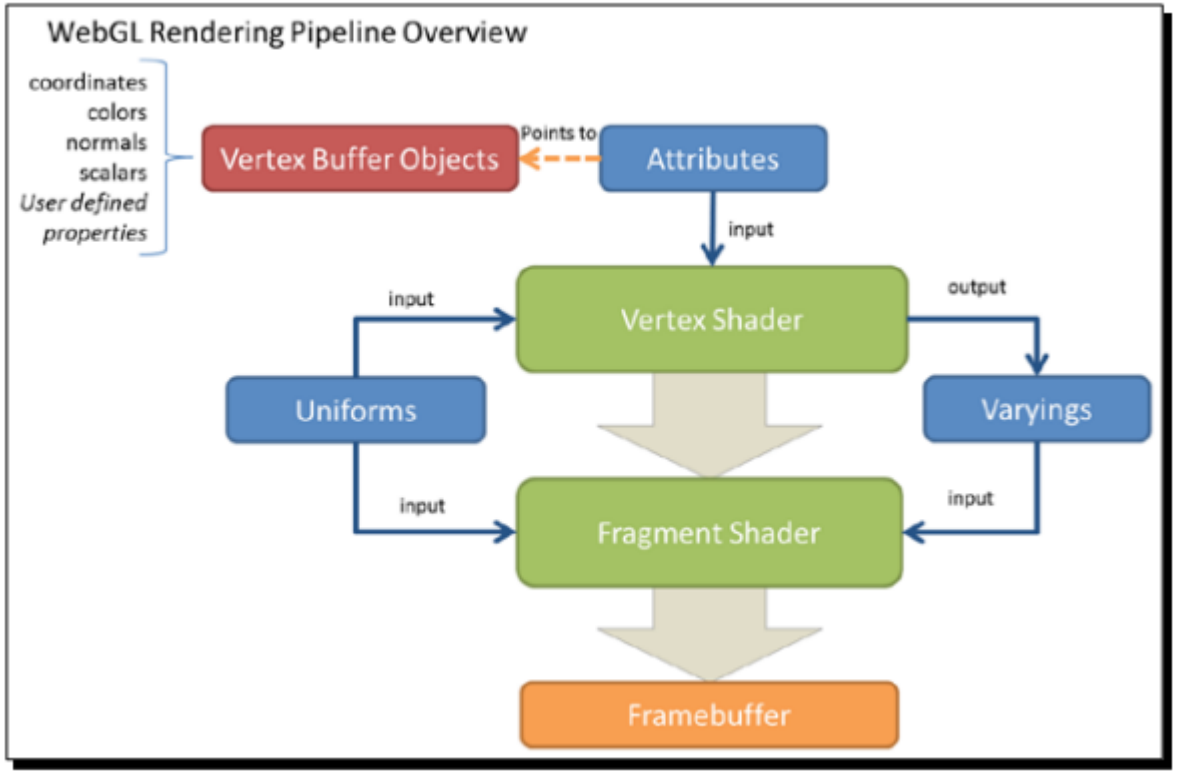
Unlike its cousin, OpenGL, WebGL does not provide API methods to pass independent vertices to the rendering pipeline, therefore we need to write all of our vertices in a JavaScript arrayand then construct a WebGL vertex bufferwith it.

Indices are numeric labels for the vertices in a given 3D scene.

Indices allow us to tell WebGL how to connect vertices in order to produce a surface. Just like with vertices, indices are stored in a JavaScript array and then they are passed along to WebGL's rendering pipeline using a WebGL index buffer.

There are two kind of WebGL buffers used to describe and process geometry: Buffers that contain vertex data are known as ***Vertex Buffer Objects(VBOs)****. Similarly, buffers that contain index data are known as* ***Index Buffer Objects (IBOs).***

WebGL's rendering pipeline



Vertex Buffer Objects (VBOs)

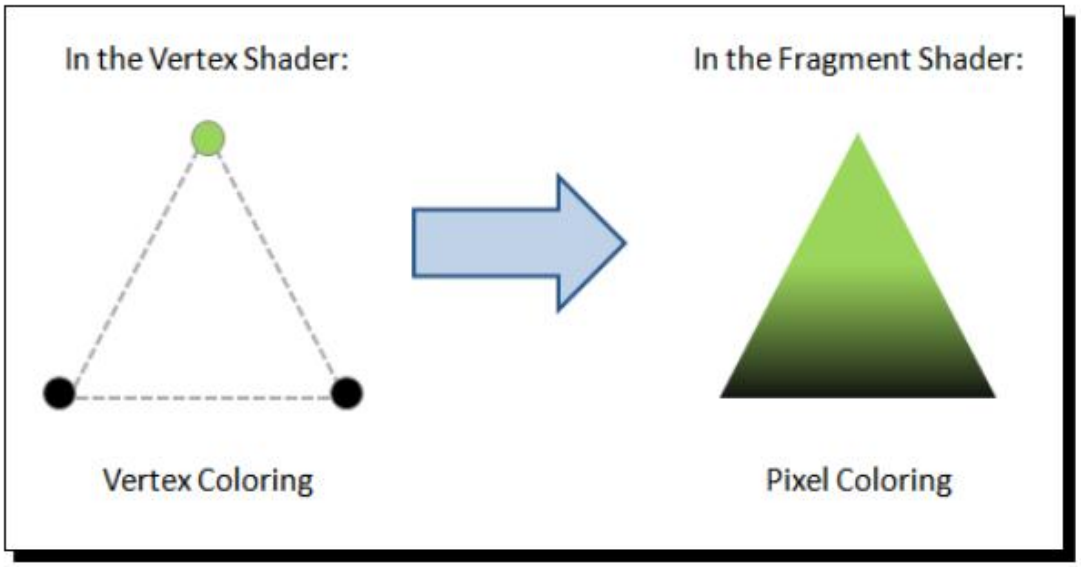
VBOs contain the data that WebGL requires to describe the geometry that is going to be rendered. As mentioned in the introduction, vertex coordinates are usually stored and processed in WebGL as VBOs. Additionally, there are several data elements such as vertex normals, colors, and texture coordinates, among others, that can be modeled as VBOs.

Vertex shader

The vertex shader is called on each vertex. This shader manipulatesper-vertexdata such as vertex coordinates, normals, colors, and texture coordinates. This data is represented by attributes inside the vertex shader. Each attribute points to a VBOfrom where it reads vertex data.

Fragment shader

Every set of three vertices defines a
triangle needs to be assigned a colo
Each surface element is called a fra
going to be displayed on your scree
pixels.The main goal of the fragmer
pixels. The following diagram expla

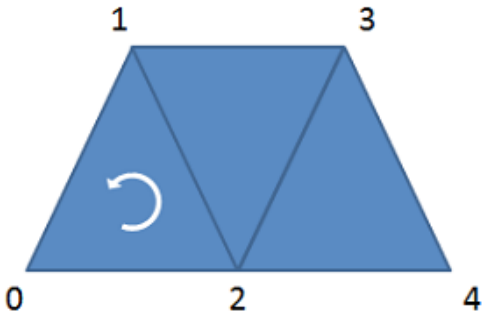


Framebuffer

It is a two-dimensional buffer that
the fragment shader. Once all fragn
and displayed on screen. The frame
pipeline.

Defining a geometry using JavaScript arrays

Vertex and Indices



Index	Vertex Coordinates
0	(0,0)
1	(10,10)
2	(20,0)
3	(30,10)
4	(40,0)

coordinates

Vertex array = [0,0,10,10,20,0,30,10,40,0] → Vertex Buffer

Index array = [0,2,1,1,2,3,2,4,3] → Index Buffer

triangles

Triangles in the index array are *usually* but not necessarily defined counter-clockwise.

Creating WEB GL buffers

```
var vertices = [-50.0, 50.0, 0.0, -50.0,-50.0, 0.0, 50.0,-50.0, 0.0,
50.0, 50.0, 0.0];/* our JavaScript vertex array */
var myBuffer = gl.createBuffer(); /*gl is our WebGL Context*/
```

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

The first parameter is the type of buffer that we are creating. We have two options for this parameter:

- **gl.ARRAY_BUFFER: Vertex data**
- **gl.ELEMENT_ARRAY_BUFFER: Index data**

WebGL will always access the currently bound buffer looking for the data. Therefore, we should be careful and make sure that we have always bound a buffer before calling any other operation for geometry processing. If there is no buffer bound, then you will obtain the error INVALID_OPERATION

Array data types

Once we have bound a buffer, we need to pass along its contents. We do this with the **bufferData** function:

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

Please observe that vertex coordinates can be float, but indices are always integer. Therefore, we will use Float32Array for VBOs and UInt16Array for IBOs throughout the examples of this book. These two types represent the largest typed arrays that you can use in WebGL per rendering call. The other types can be or cannot be present in your browser, as this specification is not yet final at the time of writing the book.

Since the indices support in WebGL is restricted to 16 bit integers, an index array can only be 65,535 elements in length. If you have a geometry that requires more indices, you will need to use several rendering calls.

Unbind buffer

Finally, it is a good practice to unbind the buffer. We can achieve that by calling the following instruction:

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

```
var coneVBO = null; //Vertex Buffer Object
var coneIBO = null; //Index Buffer Object
function initBuffers() {
var vertices = []; //JavaScript Array that populates
coneVBO
var indices = []; //JavaScript Array that populates
coneIBO;
//Vertices that describe the geometry of a cone
vertices =[1.5, 0, 0,
-1.5, 1, 0,
-1.5, 0.809017, 0.587785,
-1.5, 0.309017, 0.951057,
-1.5, -0.309017, 0.951057,
-1.5, -0.809017, 0.587785,
-1.5, -1, 0.0,
-1.5, -0.809017, -0.587785,
-1.5, -0.309017, -0.951057,
-1.5, 0.309017, -0.951057,
-1.5, 0.809017, -0.587785];
```

```
//Indices that describe the geometry of a cone
indices = [0, 1, 2,
0, 2, 3,
0, 3, 4,
0, 4, 5,
0, 5, 6,
0, 6, 7,
0, 7, 8,
0, 8, 9,
0, 9, 10,
0, 10, 1];

coneVBO = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, coneVBO);
gl.bufferData(gl.ARRAY_BUFFER, new
Float32Array(vertices),
gl.STATIC_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
coneIBO = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
coneIBO);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new
Uint16Array(indices),
gl.STATIC_DRAW);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
}
```

Summary

To summarize, for every buffer, we want to:

- Create a new buffer
- Bind it to make it the current buffer
- Pass the buffer data using one of the typed arrays
- Unbind the buffer

Operations to WebGL buffers

Method	Description
`var aBuffer = createBuffer(void)`	Creates the `aBuffer` buffer
`deleteBuffer(Object aBuffer)`	Deletes the `aBuffer` buffer
`bindBuffer(ulong target, Object buffer)`	Binds a buffer object. The accepted values for `target` are: ◆ `ARRAY_BUFFER` (for vertices) ◆ `ELEMENT_ARRAY_BUFFER` (for indices)
`bufferData(ulong target, Object data, ulong type)`	The accepted values for `target` are: ◆ `ARRAY_BUFFER` (for vertices) ◆ `ELEMENT_ARRAY_BUFFER`(for indices) The parameter `type` is a performance hint for WebGL. The accepted values for `type` are: ◆ `STATIC_DRAW`: Data in the buffer will not be changed (specified once and used many times) `DYNAMIC_DRAW`: Data will be changed frequently (specified many times and used many times) ◆ `STREAM_DRAW`: Data will change on every rendering cycle (specified once and used once)

Attributes, uniforms, and varyings

Attributes, uniforms, and varyings are the three different types of variables that you will find when programming with shaders.

Attributes are input variables used in the vertex shader. For example, vertex coordinates, vertex colors, and so on. Due to the fact that the vertex shader is called on each vertex, the attributes will be different every time the vertex shader is invoked.

Uniforms are input variables available for both the vertex shader and fragment shader. Unlike attributes, uniforms are constant during a rendering cycle. For example, lights position.

Varyings are used for passing data from the vertex shader to the fragment shader.

Pointing an attribute to the currently bound VBO

Assume that we have the aVertexPosition attribute and that it will represent vertex coordinates inside the vertex shader.
The WebGL function that allows pointing attributes to the currently bound VBOs is vertexAttribPointer. The following is its api:
gl.vertexAttribPointer(Index,Size,Type,Norm,Stride,Offset);

- **Index:** An attribute's index that we are going to map the currently bound buffer to.
- **Size:** Indicates the number of values per vertex that are stored in the currently bound buffer.
- **Type:** Specifies the data type of the values stored in the current buffer. It is one of the following constants: FIXED, BYTE, UNSIGNED_BYTE, FLOAT, SHORT, or UNSIGNED_SHORT.
- **Norm**: This parameter can be set to true or false. It handles numeric conversions that lie out of the scope of this introductory guide. For all practical effects, we will set this parameter to false.
- **Stride**: If stride is zero, then we are indicating that elements are stored sequentially in the buffer.
- **Offset**: The position in the buffer from which we will start reading values for the corresponding attribute. It is usually set to zero to indicate that we will start reading values from the first element of the buffer.

The drawArrays and drawElements

The functions drawArrays and drawElements are used for writing on the Frame buffers.
drawArrays uses vertex data in the order in which it is defined in the buffer to create the geometry
drawElements uses indices to access the vertex data buffers and create the geometry

Both drawArrays and drawElements will only use enabled arrays. These are the vertex buffer objects that are mapped to active vertex shader attributes.

We will come back after the end of Chapter 2 when we will refer extensively to Attributes and Uniforms

Advanced geometry loading techniques: JavaScript Object Notation (JSON) and AJAX

Let's study a way to load the geometry (vertices and indices) from a file instead of declaring the vertices and the indices every time

To achieve this, we will make asynchronous calls to the web server using AJAX. We will retrieve the file with our geometry from the web server and then we will use the built-in JSON parser to convert the context of our files into JavaScript objects. In our case, these objects will be the vertices and indices array.

JavaScript Object Notation (JSON)

JSON stands for **JavaScript Object Notation**. It is a lightweight, text-based, open format used for data interchange. JSON is commonly used as an alternative to XML.
The JSON format is language-agnostic. This means that there are parsers in many languages to read and interpret JSON objects. Also, JSON is a subset of the object literal notation of JavaScript. Therefore, **we can define JavaScript objects** using JSON.

Let's see how this work. Assume for example that we have the mode lobject with two arrays vertices and indices. Say that these arrays contain the information described in the cone example (ch2_Cone.html) as follows:

```
vertices =[1.5, 0, 0,
-1.5, 1, 0,
-1.5, 0.809017, 0.587785,
-1.5, 0.309017, 0.951057,
-1.5, -0.309017, 0.951057,
-1.5, -0.809017, 0.587785,
-1.5, -1, 0,
-1.5, -0.809017, -0.587785,
-1.5, -0.309017, -0.951057,
-1.5, 0.309017, -0.951057,
-1.5, 0.809017, -0.587785];
indices = [0, 1, 2, 0, 2, 3,
0, 3, 4,
0, 4, 5,
0, 5, 6,
0, 6, 7,
0, 7, 8,
0, 8, 9,
0, 9, 10,
0, 10, 1];
```

Following the JSON notation, we would represent these two arrays as an object, as follows:

```
var model = {
"vertices" : [1.5, 0, 0,
-1.5, 1, 0,
-1.5, 0.809017, 0.587785,
-1.5, 0.309017, 0.951057,
-1.5, -0.309017, 0.951057,
-1.5, -0.809017, 0.587785,
-1.5, -1, 0,
-1.5, -0.809017, -0.587785,
-1.5, -0.309017, -0.951057,
-1.5, 0.309017, -0.951057,
-1.5, 0.809017, -0.587785],
"indices" : [0, 1, 2,0, 2, 3,0, 3, 4,
0, 4, 5,
0, 5, 6,
0, 6, 7,
0, 7, 8,
0, 8, 9,
0, 9, 10,
0, 10, 1]};
```

From the previous example, we can infer the following syntax rules:

- The extent of a JSON object is defined by curly brackets {}
- Attributes in a JSON object are separated by comma ,
- There is no comma after the last attribute
- Each attribute of a JSON object has two parts: a key and a value
- The name of an attribute is enclosed by quotation marks " “
- Each attribute key is separated from its corresponding value with a colon :
- Attributes of the type Array are defined in the same way you would define them in JavaScript

JSON encoding and decoding

Method	Description
`var myText = JSON.stringify(myObject)`	We use `JSON.stringify` for converting JavaScript objects to JSON-formatted text.
`var myObject = JSON.parse(myText)`	We use `JSON.parse` for converting text into JavaScript objects.

JSON example

Open Firefox console (Firefox Tools| Web Developer| Web Console)

- Create a JSON object by typing:
  var model = {"vertices":[0,0,0,1,1,1], "indices":[0,1]};

- Verify that the model is an object by writing:
  typeof(model)

- Now, let's print the model attributes. Write this in the console:
  model.vertices <ENTER>
  model.indices <ENTER>

- Now, let's create a JSON text:
  var text = JSON.stringify(model)

- Now let's convert the JSON text back to an object. Type the following:
  var model2 = JSON.parse(text)
  typeof(model2)
  model2.vertices

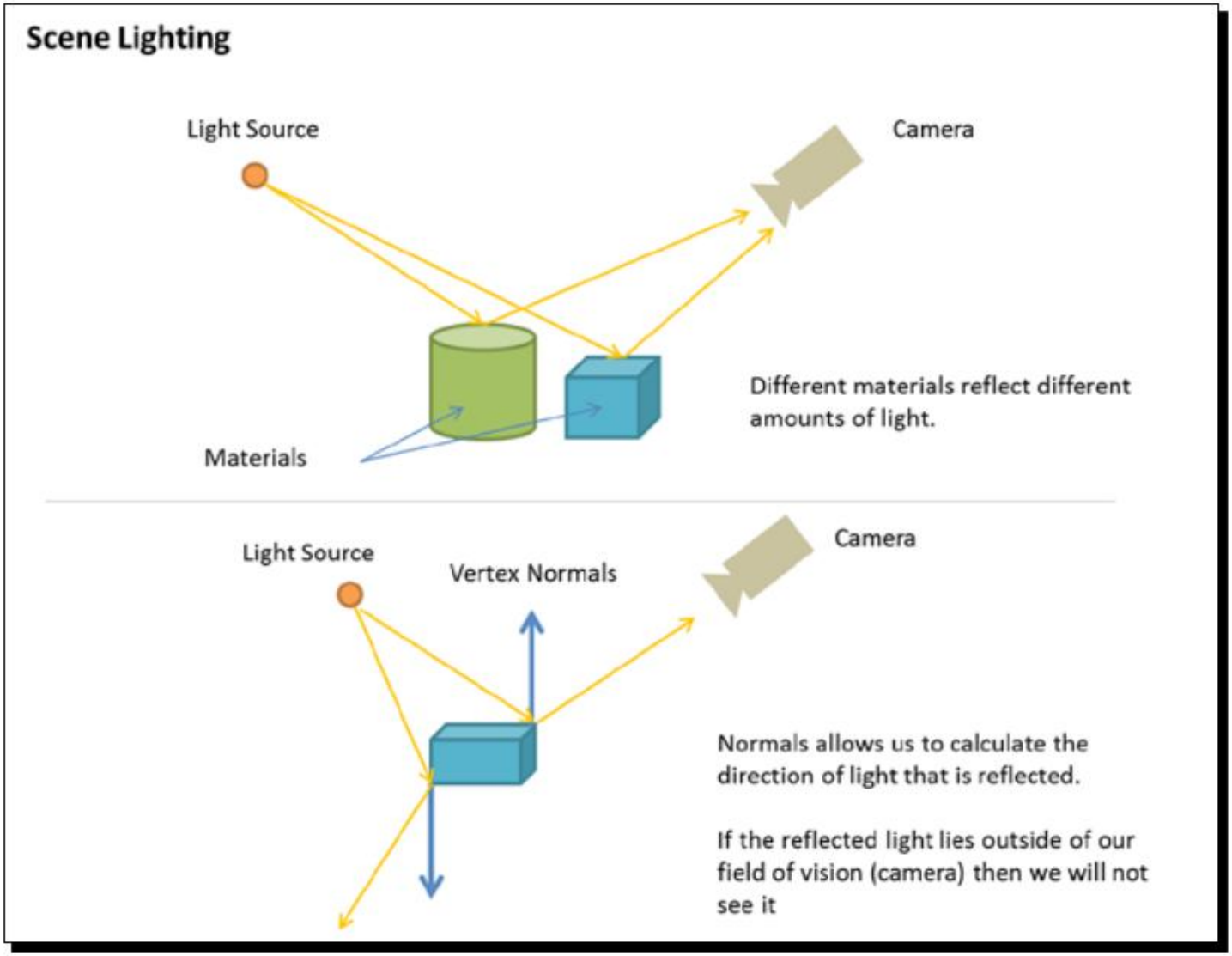
JSON object persists through text!

Write
alert(text)
And see what happens….

What happens when you type text.vertices?
As you can see, you get an error message saying that text.verticesis not defined. This happens because text is not a JavaScript object but a string with the peculiarity of being written according to JSON notation to describe an object. Everything in it is text and therefore it does not have any fields.

The example that we have used is relevant because this is the way we will define our geometry to be loaded from external files.

LIGHTS


Scene Lighting
Light Source
Camera
Materials
Different materials reflect different amounts of light.
Light Source
Vertex Normals
Camera
Normals allows us to calculate the direction of light that is reflected.
If the reflected light lies outside of our field of vision (camera) then we will not see it

positional & directional light

A light source is called **positional** when its location will affect how the scene is lit. For instance, a lamp inside a room falls under this category. Objects far from the lamp will receive very little light and they will appear obscure.

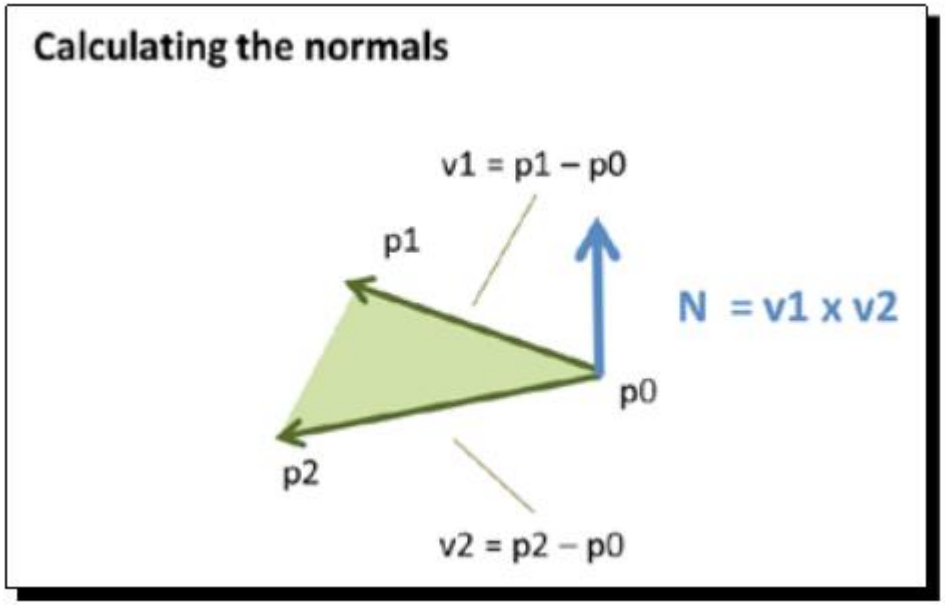
In contrast, **directional** lights refer to lights that produce the same result independent from their position. For example, the light of the sun will illuminate all the objects in a terrestrial scene, regardless of their distance from the sun.

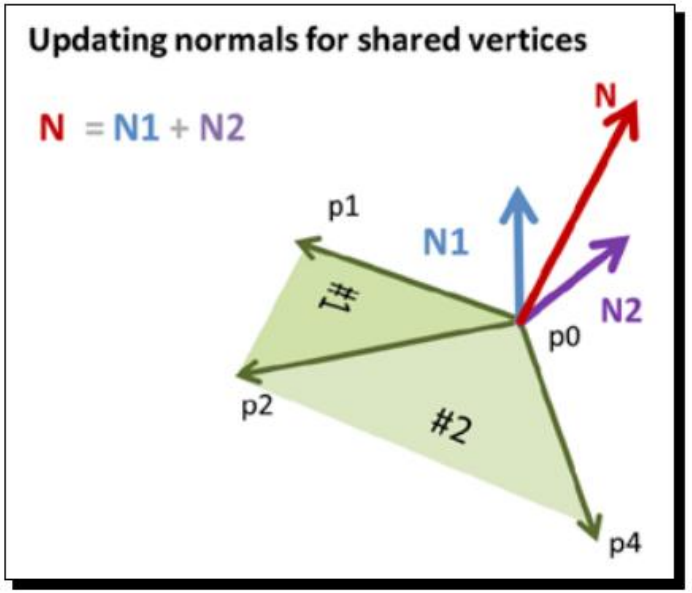
A positional light is modeled by a point in space, while a directional light is modeled with a vector that indicates its direction.

Normals

Normals are vectors that are perpendicular to the surface that we want to illuminate. Normals represent the orientation of the surface and therefore they are critical to model the interaction between a light source and the object. Each vertex has an associated normal vector.

Then the normal is obtained by calculating the cross product v1 x v2.





Materials (in brief)

The material of an object in WebGL can be modeled by several parameters, including its color and its texture. Material colors are usually modeled as triplets in the RGB space (Red, Green, Blue). Textures, on the other hand, correspond to images that are mapped to the surface of the object. This process is usually called Texture Mapping. (more on this in a later course)

Shading and light reflection

Shading refers to the type of interpolation that is performed to obtain the final color for every fragment in the scene.

Lighting determines how the normals, materials, and lights are combined to produce the final color. (also called reflection models)

Shading interpolation methods

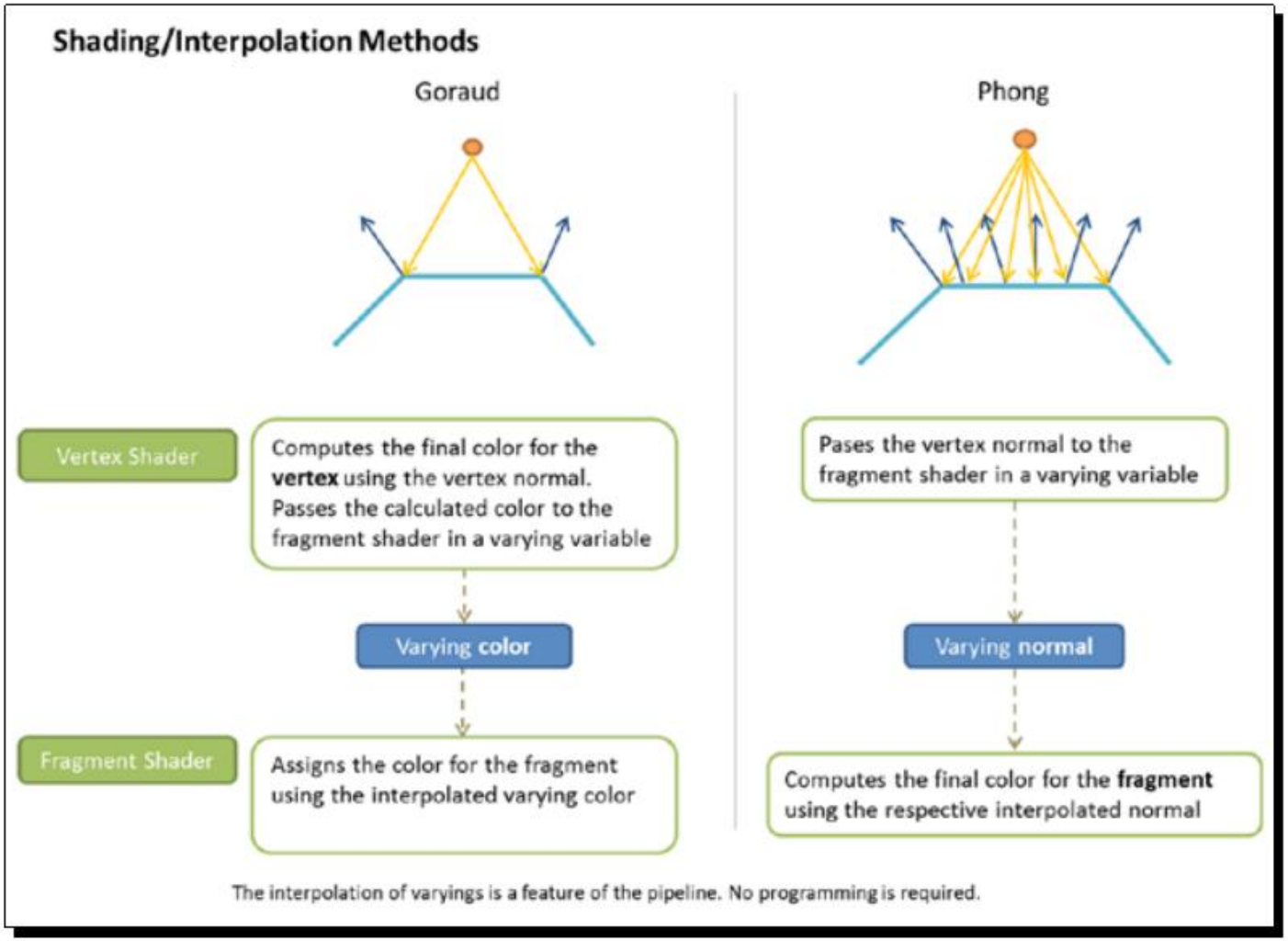
Gouraud interpolation

The Gouraud interpolation method calculates the final color in the vertex shader. The vertex normals are used in this calculation. Gouraud shading is most often used to achieve continuous lighting on triangle surfaces by computing the lighting at the corners of each triangle and linearly interpolating the resulting colours for each pixel covered by the triangle. Gouraud first published the technique in 1971

Phong interpolation

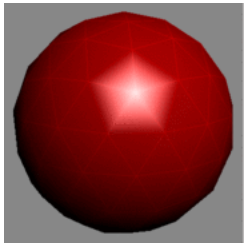
The Phong method calculates the final color in the fragment shader. Phong or normal-vector interpolation shading. Specifically, it interpolates surface normals across rasterized polygons and computes pixel colors based on the interpolated normals and a reflection model.

Gouraud & Phong



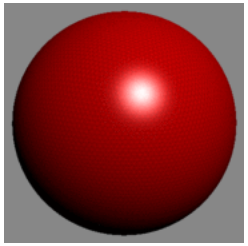
Gouraud vs Phong

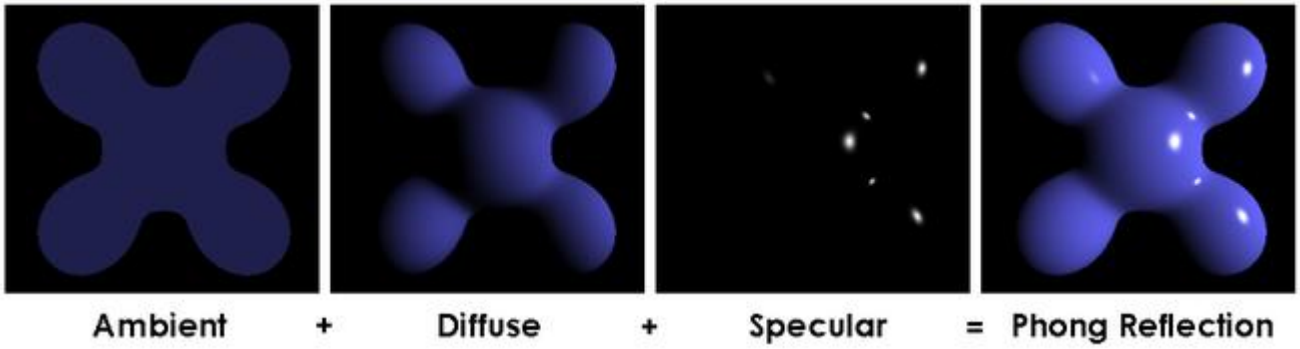
Unlike Gouraud shading, which interpolates colors across polygons, in Phong shading a normal vector is linearly interpolated ***across the surface*** of the polygon from the polygon's vertex normals. The surface normal is interpolated and normalized at each pixel and then used in a reflection model, e.g. the Phong reflection model, to obtain the final pixel color. Phong shading is more computationally expensive than Gouraud shading since the reflection model must be computed at each pixel instead of at each vertex.



Gouraud shading

Source Wikipedia (http://en.wikipedia.org/wiki/Gouraud_shading)





Source: wikipedia (http://en.wikipedia.org/wiki/File:Phong_components_version_4.png)

Light reflection models

Lambertian reflection model

Lambertian reflections are commonly used in computer graphics as a model for diffuse reflections, which are the kind of reflections where an incident light ray is reflected in many angles instead of only in one angle as it is the case for specular reflections.

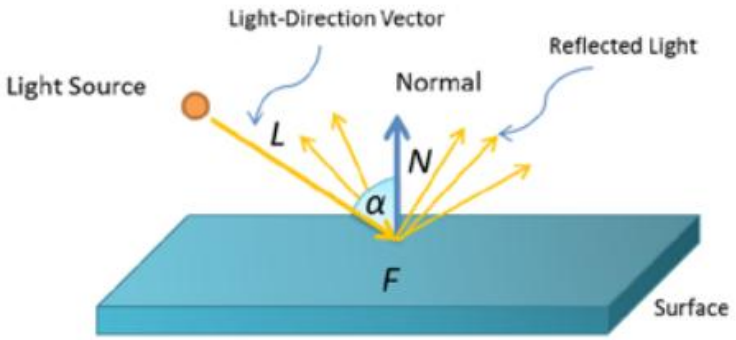
This lighting model is based on the cosine emission law or Lambert's emission law. It is named after Johann Heinrich Lambert, from his Photometria, published in 1760.

Phong reflection model

The Phong reflection model describes the way a surface reflects the light as the sum of three types of reflection: ambient, diffuse, and specular. It was developed by Bui Tuong Phong who published it in his 1973 Ph.D. dissertation

Lambertian reflection

Lambertian Reflectance

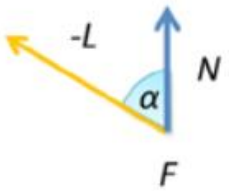


Final Diffuse Color

$$F = C_l C_m(-L \cdot N)$$

Light Diffuse Color Material Diffuse Color

Final diffuse color calculation for fragment F



$$-L \cdot N = |-L||N| \cos\alpha$$

If L and N are normalized then:

$$-L \cdot N = \cos\alpha$$

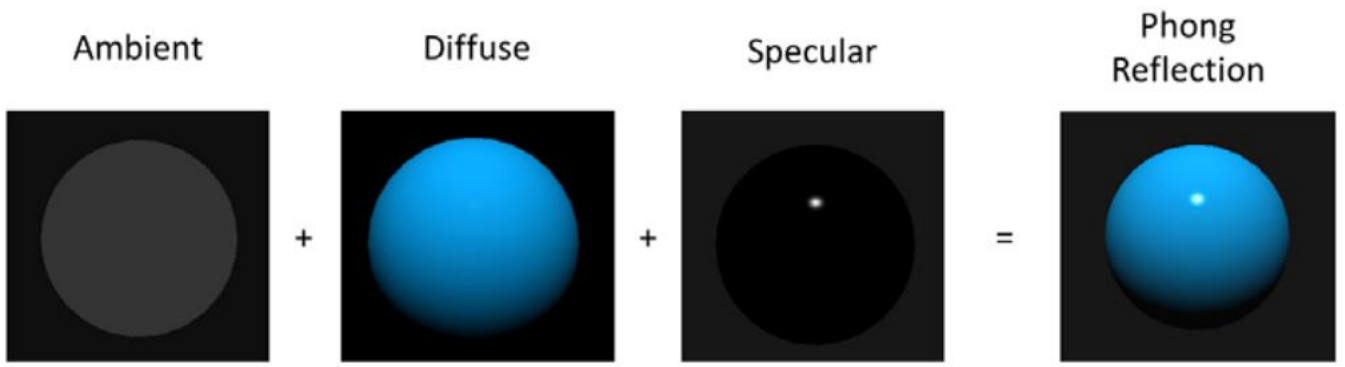
$$F = C_l C_m \cos\alpha$$

A Lambertian surface reflects light in many directions

Phong reflection

Phong Reflection Model

Reflected color is the result of combining three types of light-object interactions:



Amount of light present *everywhere* in the scene. Independent from any light source

The incident light is reflected in *many directions*. It can be modelled by a Lambertian surface.

Mirror-like reflection. The direction of the incoming light and the direction of the reflected outgoing light make *the same angle with respect to the surface normal*.

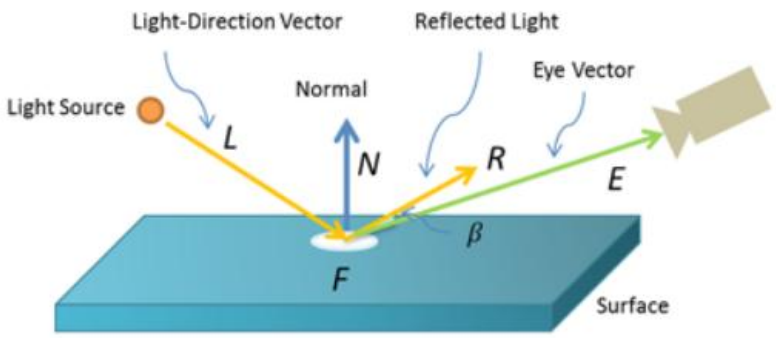
Ambient & Diffuse

The ambient term accounts for the scattered light present in the scene. This term is independent from any light source and it is the same for all fragments.

The diffuse term corresponds to diffuse reflections. Usually a Lambertian model is used for this component.

Specular reflection

Specular Reflection

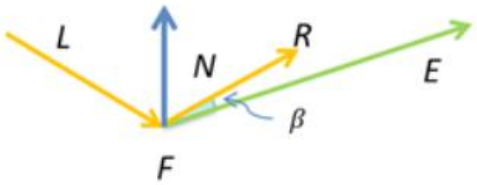


Final Specular Color — Material Shininess

$$F_s = C_l C_m (R \cdot E)^n$$

Light Specular Color — Material Specular Color

Final specular color calculation for fragment F



$$R \cdot E = |R||E| \cos\beta$$

If R and E are normalized then:

$$R \cdot E = \cos\beta$$

$$F = C_l C_m \cos^n \beta$$

The specular reflection reaches its maximum when *R* and E have the same direction.

CAMERA

There is no camera object in the WebGL API, only matrices.
Having matrices instead of a cameras gives WebGL a lot of flexibility

In this chapter, we will:

- Understand the transformations that the scene undergoes from a 3D world to a 2D screen
- Learn about affine transformations
- Work with the Model-View matrix and the Perspective matrix
- Appreciate the value of the Normal matrix
- Create a camera and use it to move around a 3D scene

Some maths

See wiki: http://en.wikipedia.org/wiki/Transformation_matrix

Or Local file

Homogeneous coordinates

Homogeneous coordinates are a key component of any computer graphics program. Thanks to them, it is possible to represent affine transformations (rotation, scaling, shear, and translation) and projective transformations as 4x4 matrices.

In Homogeneous coordinates, vertices have four components: x, y, z, and w. The first three components are the vertex coordinates in Euclidian Space. The fourth is the perspective component. The 4-tuple (x,y,z,w) take us to a new space: The **Projective Space.**

It is easy to convert from Homogeneous coordinates to non-homogeneous, old-fashioned, Euclidean coordinates. All you need to do is divide the coordinate by *w*:

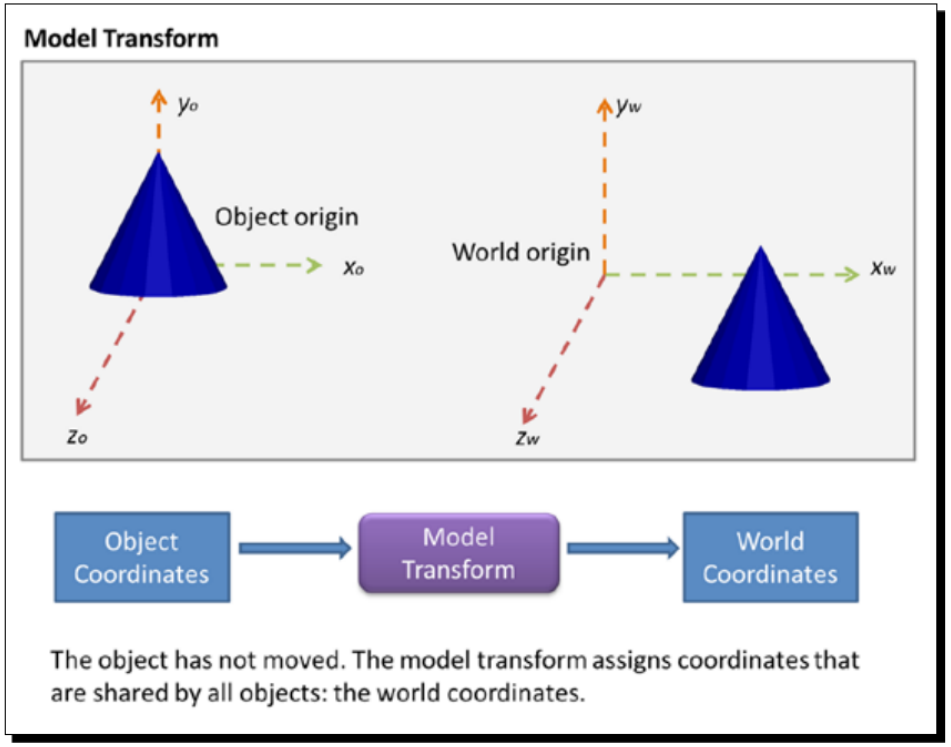
$$h(x,y,z,\ w)=v(x/w,y/w,z/w) \rightarrow v(x,y,z)=h(x,y,z,\ 1)$$

Consequently, if we want to go from Euclidian to Projective space, we just add the fourth component *w* and make it 1.

Model Transform

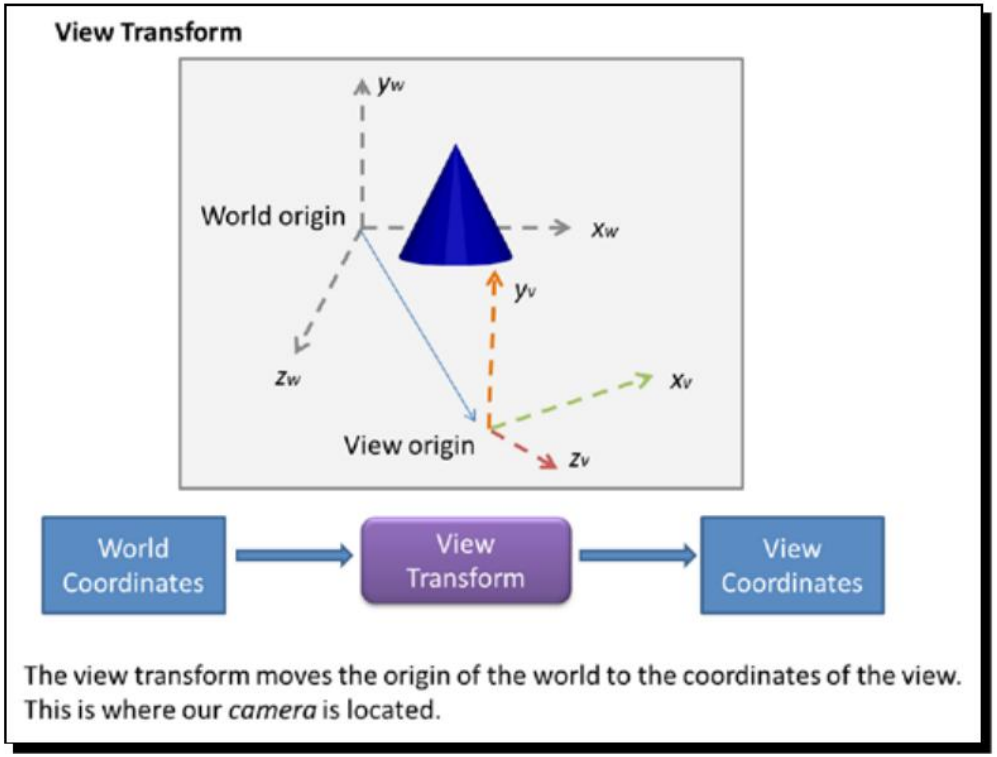
We start our analysis from the object coordinate system. It is in this space where vertex coordinates are specified. Then **if we want to translate or move objects around, we use a matrix that encodes these transformations. This matrix is known as the model matrix.**

Once we multiply the vertices of our object by the model matrix, we will obtain new vertex coordinates. These new vertices will determine the position of the object in our 3D world.



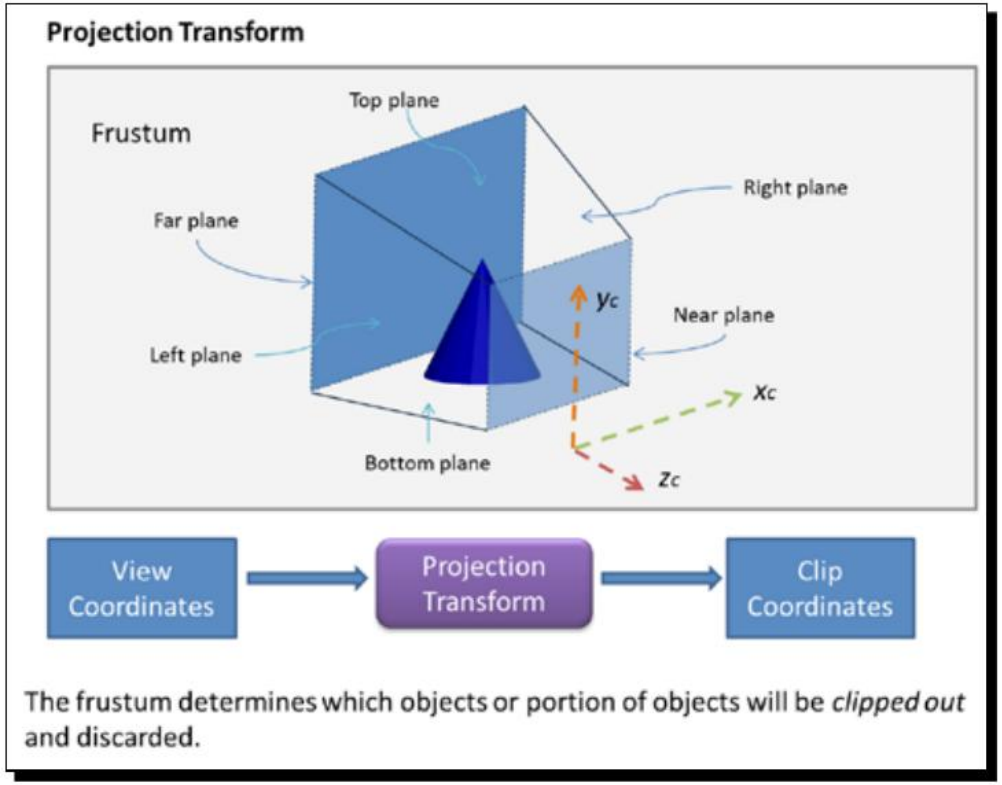
View Transform

The next transformation, **the view transform, shifts the origin of the coordinate system to the view origin**. The view origin is where our eye or camera is located with respect to the world origin. In other words, the view transform switches world coordinates by view coordinates. This transformation is encoded in the **view matrix**.

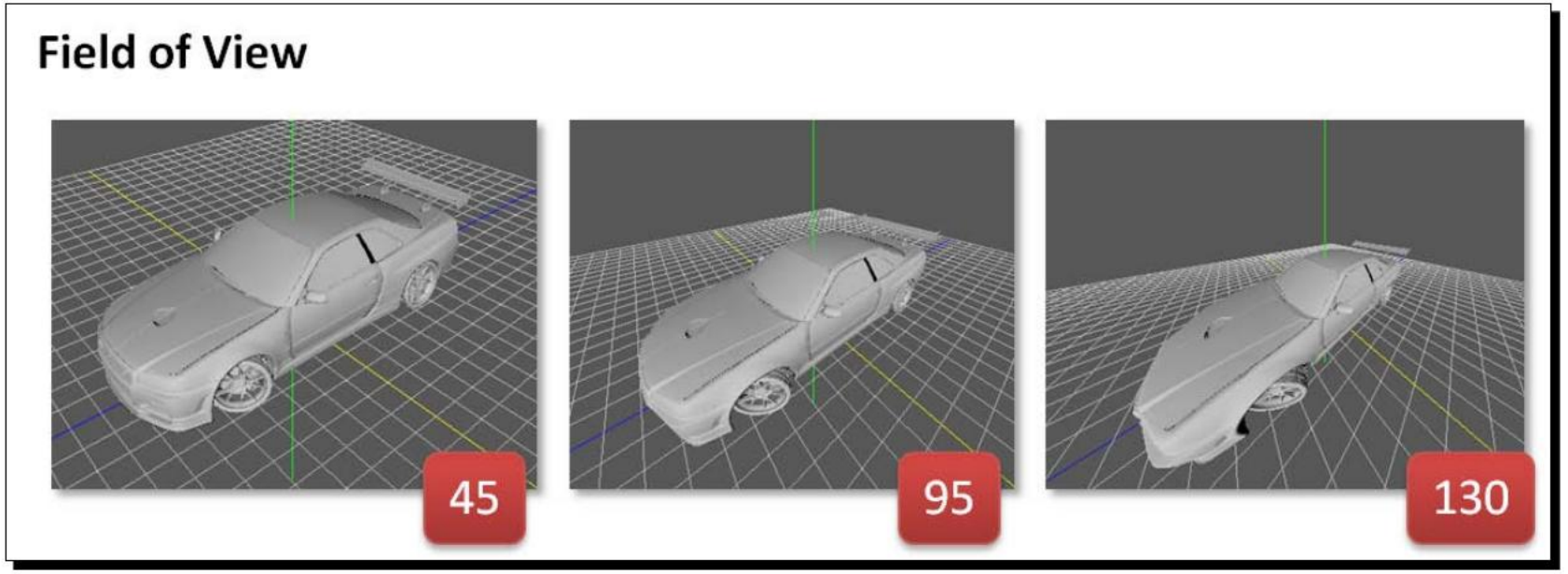


Projection Transform

The next operation is called the projection transform. This operation determines how much of the view space will be rendered and how it will be mapped onto the computer screen.
This region is known as the frustum and it is defined by six planes (near, far, top, bottom, right, and left planes)

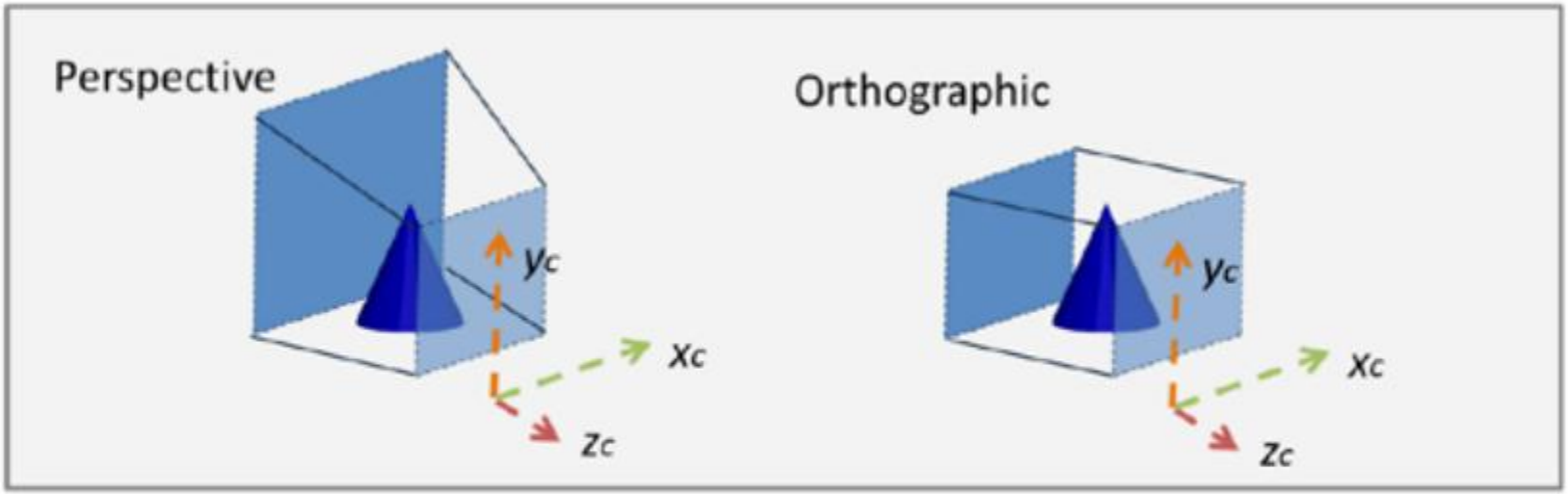


Perspective transform and field of view



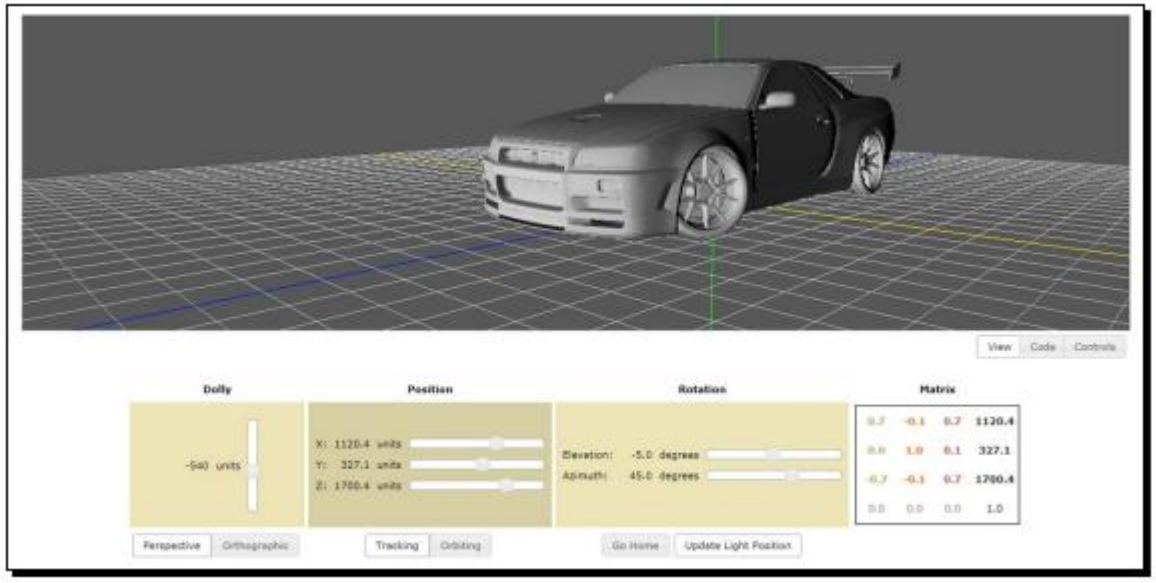
Perspective vs Orthographic

Frustum shape



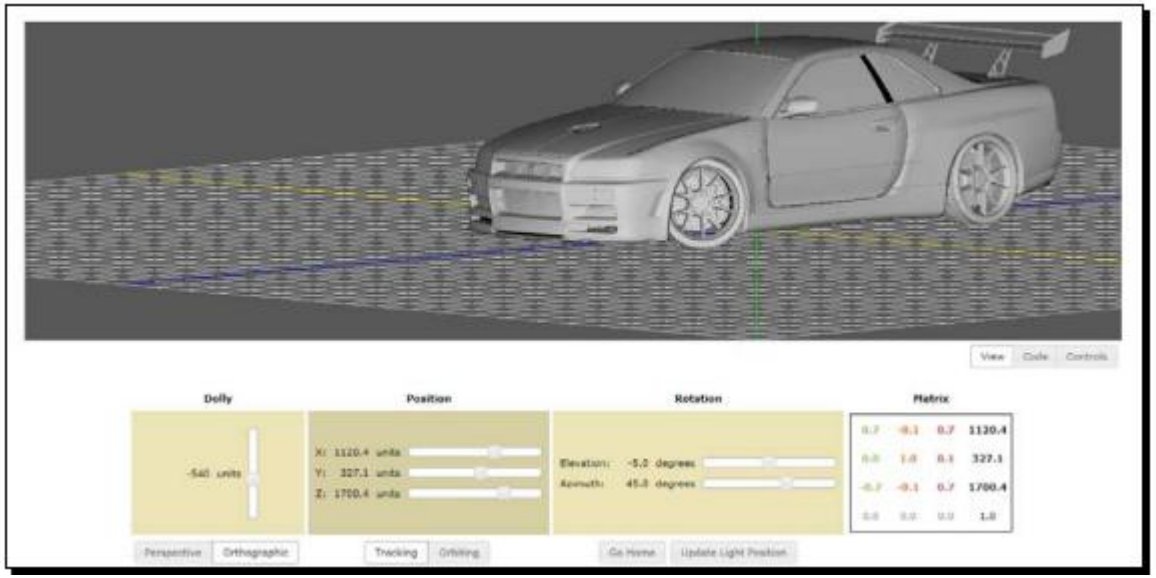
The extent and shape of the frustum determines how much of the 3D view space Is mapped to the screen and the type of 3D to 2D projection that takes place.

Perspective vs Orthographic



perspective

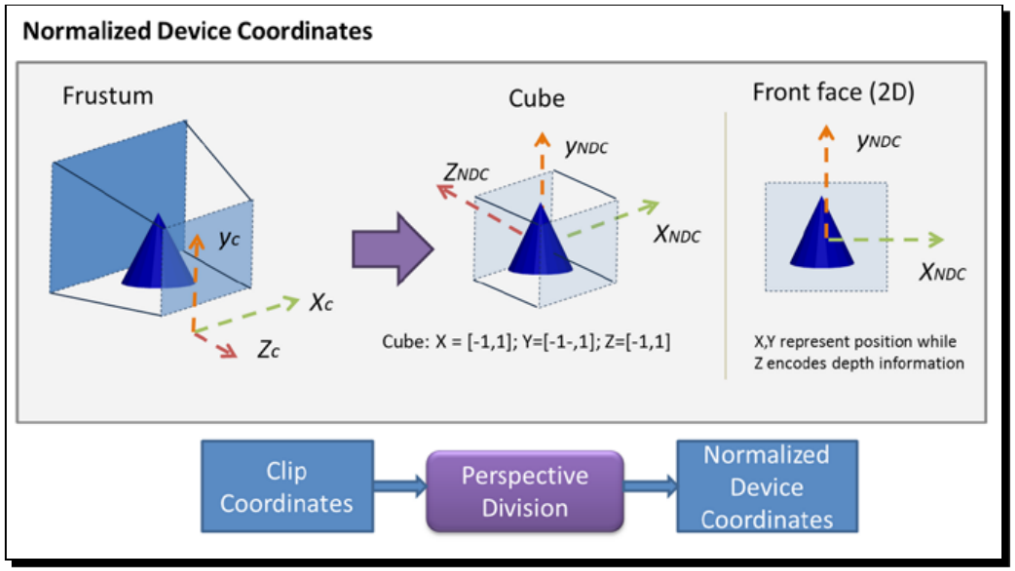
orthographic



Perspective division

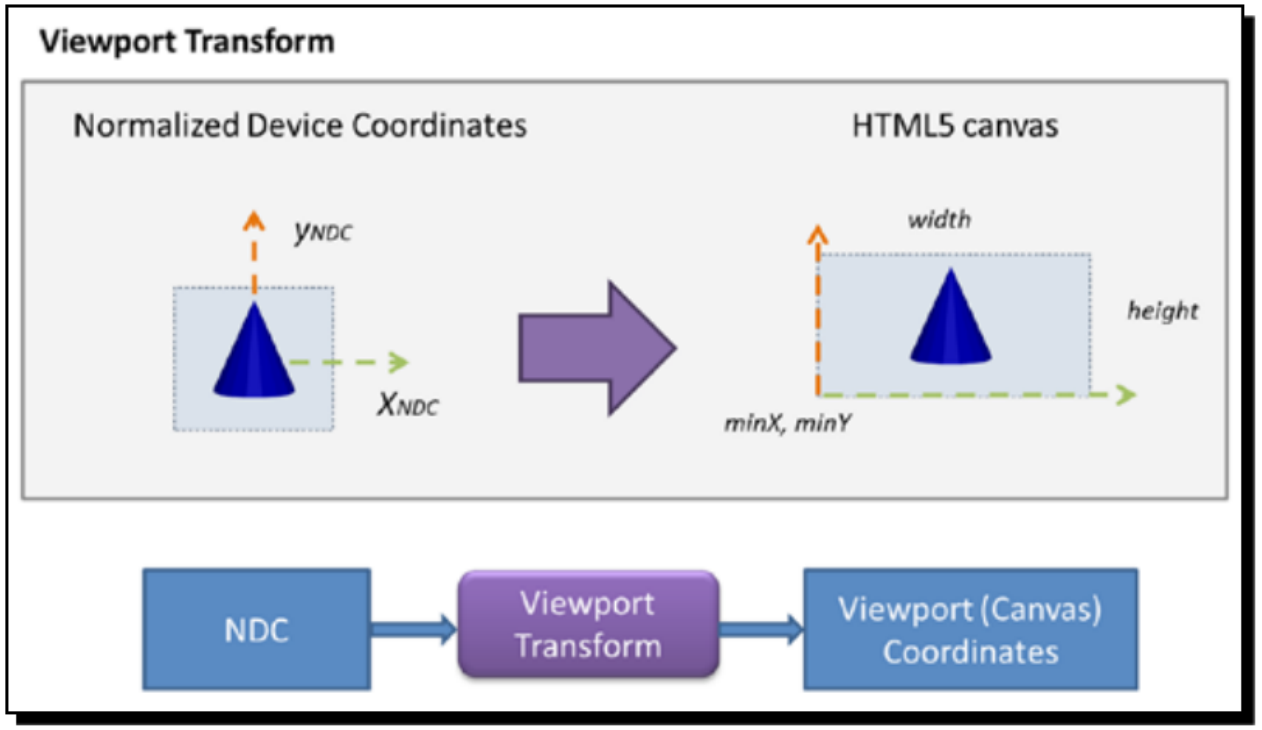
Once it is determined how much of the viewing space will be rendered, the frustum is mapped into the near plane in order to produce a 2D image. The near plane is what is going to be rendered on your computer screen.

Different operative systems and displaying devices can have mechanisms to represent 2D information on screen. To provide robustness for all possible cases, WebGL (also in OpenGL ES) provides an intermediate coordinate system that is independent from any specific hardware. This space is known as the **Normalized Device Coordinates (NDC). Normalized device coordinates are obtained by dividing the clipping coordinates by the w component.** This is the reason why this step is known as perspective division.

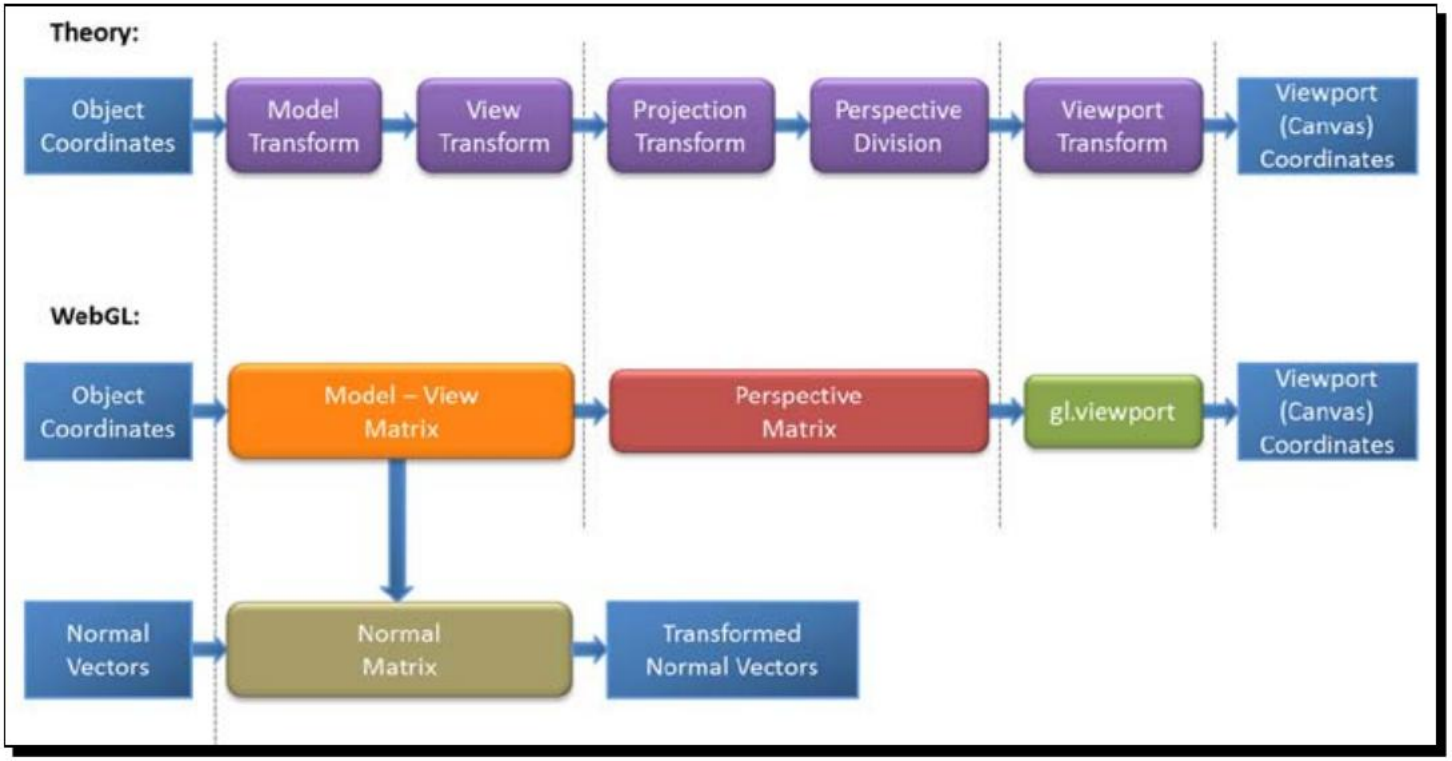


Viewport transform

Finally, NDCs are mapped to viewport coordinates. This step maps these coordinates to the available space in your screen. In WebGL, this space is provided by the HTML5 canvas. Unlike the previous cases, the viewport transform is not generated by a matrix transformation. In this case, we use the WebGL viewport function.



WebGL implementation



WebGL implementation

In WebGL, the five transformations that we apply to object coordinates to obtain viewport coordinates are grouped in three matrices and one WebGL method:

1. **The Model-View matrix** that groups the model and view transform in one single matrix. When we multiply our vertices by this matrix, we end up in view coordinates.

2. **The Normal matrix** is obtained by inverting and transposing the Model-View matrix. This matrix is applied to normal vectors for lighting purposes.

3. The **Perspective matrix** groups the projection transformation and the perspective division, and as a result, we end up in normalized device coordinates (NDC).

4. Finally, we use the operation **gl.viewport** to map NDCs to viewport coordinates:
gl.viewport(minX, minY, width, height);

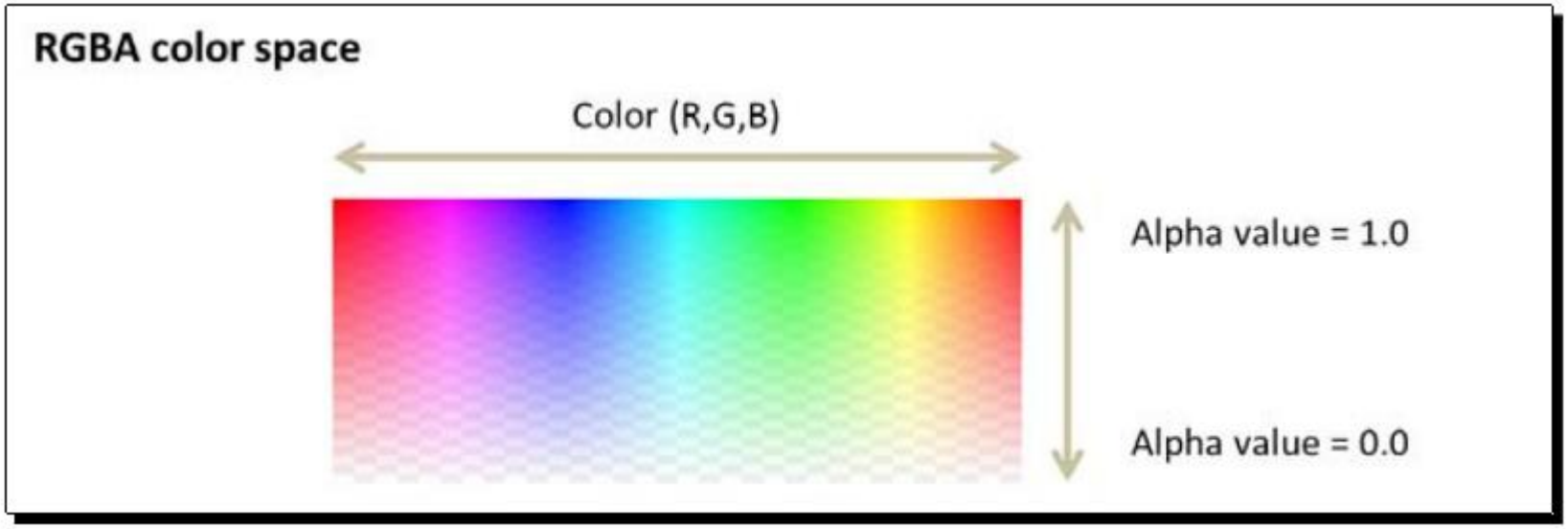
The viewport coordinates have their origin in the **lower-left corner** of the HTML5 canvas.

Colors, Depth test, and Alpha Blending

- Using colors in objects
- Assigning colors to light sources
- The depth test and the z-buffer
- Blending functions and equations
- Creating transparent objects

Using colors in WebGL

WebGL includes a fourth attribute to the RGB model. This attribute is called the alpha channel. The extended model then is known as the **RGBA** model, where **A** stands for alpha. The alpha channel contains values in the range from 0.0 to 1.0, just like the other three channels (red, green, and blue). The following diagram shows the **RGBA** color space. On the horizontal axis you can see the different colors that can be obtained by combining the R, G, and B channels. The vertical axis corresponds to the alpha channel (opacity).



Where do we use color?

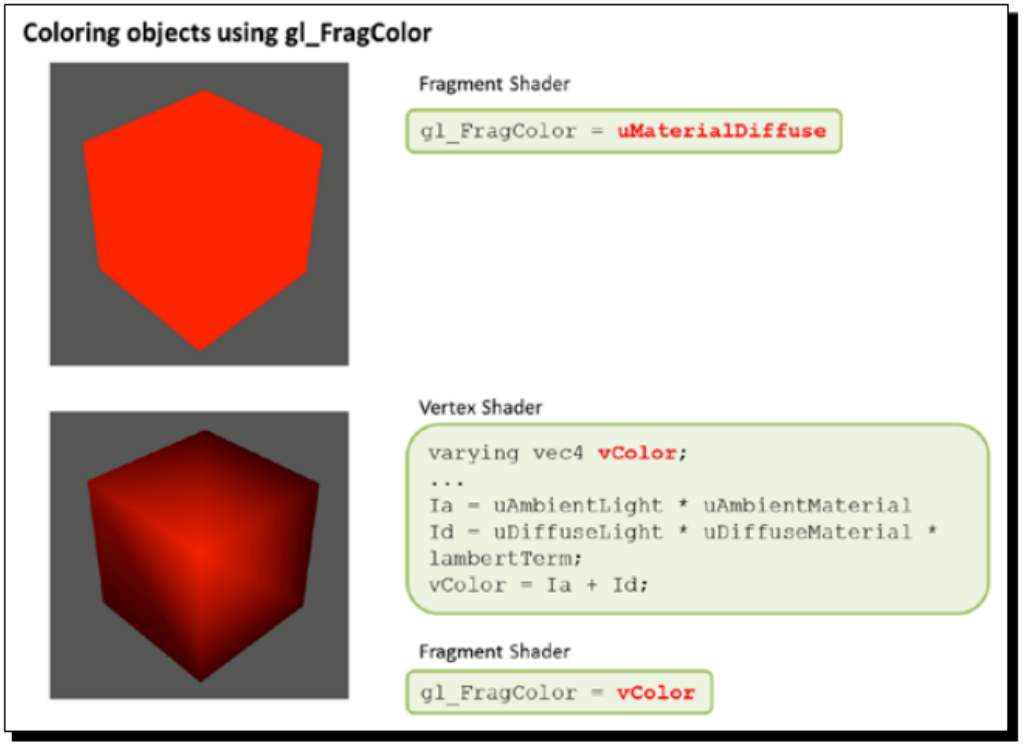
We use colors everywhere in our WebGL 3D scenes:

- **Objects**: 3D objects can be colored selecting one color for every pixel (fragment) of the object, or by selecting the color that the object will have. This would usually be the material diffuse property.
- **Lights**: Though we have been using white lights so far in the book, there is no reason why we can't have lights whose ambient or diffuse properties contain colors other than white.

Scene: The background of our scene has a color that we can change by calling gl.clearColor. Also, as we will see later, there are special operations on objects colors in the scene when we have translucent objects.

Color in objects, Constant color

To obtain a constant color we store the desired color in a uniform that is passed to the fragment shader. This uniform is usually called the object's diffuse material property.
We can also combine object normals and light source information to obtain a Lambert coefficient. We can use **the Lambert coefficient** to proportionally change the reflecting color depending on the angle on which the light hits the object.



Color in objects, Per-vertex coloring

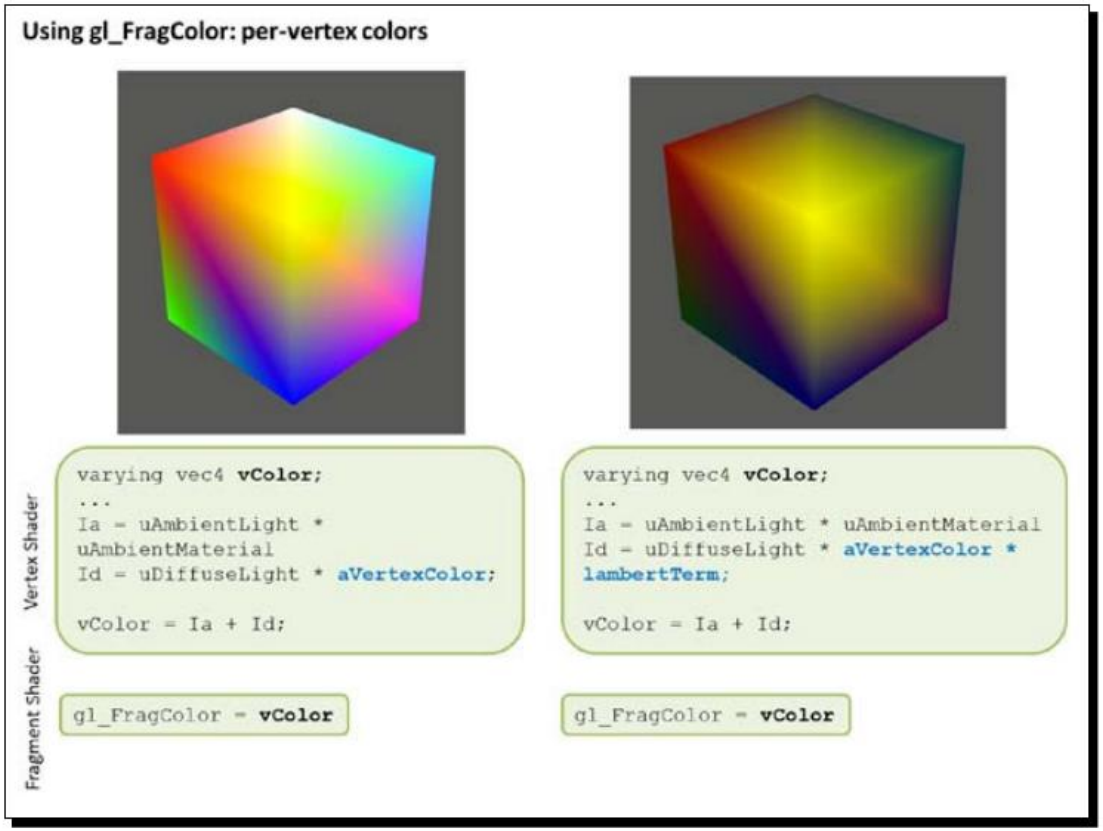
In medical and engineering visualization applications, it is common to find color maps that are associated to the vertices of the models that we are rendering. These maps assign each vertex a color depending on its scalar value. An example of this idea is the temperature charts where we can see cold temperatures as blue and hot temperatures as red overlaid on a map.
To implement per-vertex coloring, we need to define an attribute that stores the color for the vertex in the vertex shader:

attribute vec4 aVertexColor;

The next step is to assign the aVertexColor attributeto a varying so it can be carried into the fragment shader. Remember that varyings are automatically interpolated. Therefore, each fragment will have a color that is the weighted contribution of the vertices surrounding it

Vertex Color + Light

If we want our color map to be sensitive to lighting conditions we can multiply each vertex color by the diffuse component of the light.



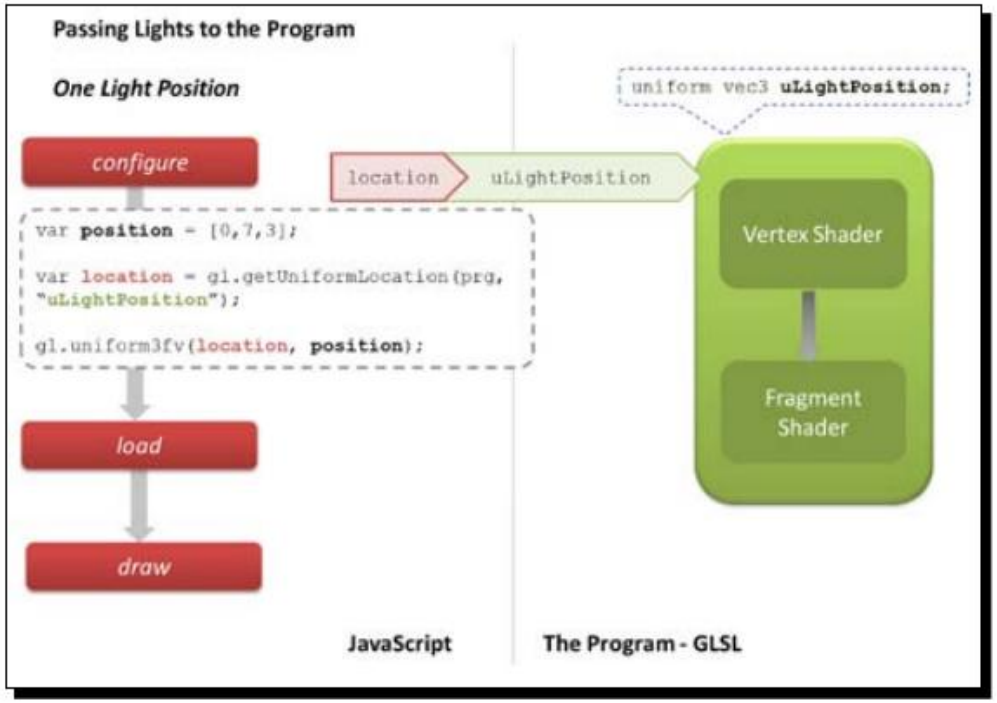
Use of color in lights

Colors are light properties. In Chapter 3, Lights, we saw that the number of light properties depend on the lighting reflection model selected for the scene. For instance, using a Lambertian reflection model we would only need to model one shader uniform: the light diffuse property/color.

In the Phong reflection model were selected, each light source would need to have three properties: the ambient, diffuse, and specular colors.

The light position is usually also modeled as a uniform when the shader needs to know where the light source is. Therefore, a Phong model with a positional light would have four uniforms: ambient, diffuse, specular, and position.
For the case of directional lights, the fourth uniform is the light direction.

Insert light to the program



The two functions we use to pass lights to the shaders are:

- **getUniformLocation**— locates the uniform in the program and returns an index we can use to set the value
- **uniform4fv**— since the light components are RGBA, we need to pass a four-element float vector

Colors in the scene

It is not possible to obtain a translucent object unless alpha blending is activated. Things get a bit more complicated when we have several objects in the scene. We will see here what to do in order to have a consistent scene when we have translucent and opaque objects.

The first approach to obtain transparent objects is to use polygon stippling. This technique consists of discarding some fragments so you can see through the object. Think of it as punching little holes throughout the surface of your object. OpenGL supports polygon stippling through the glPolygonStipple function. This function is not available in WebGL.

Transparency

More commonly, we can use the **alpha channel** information to obtain translucent objects.

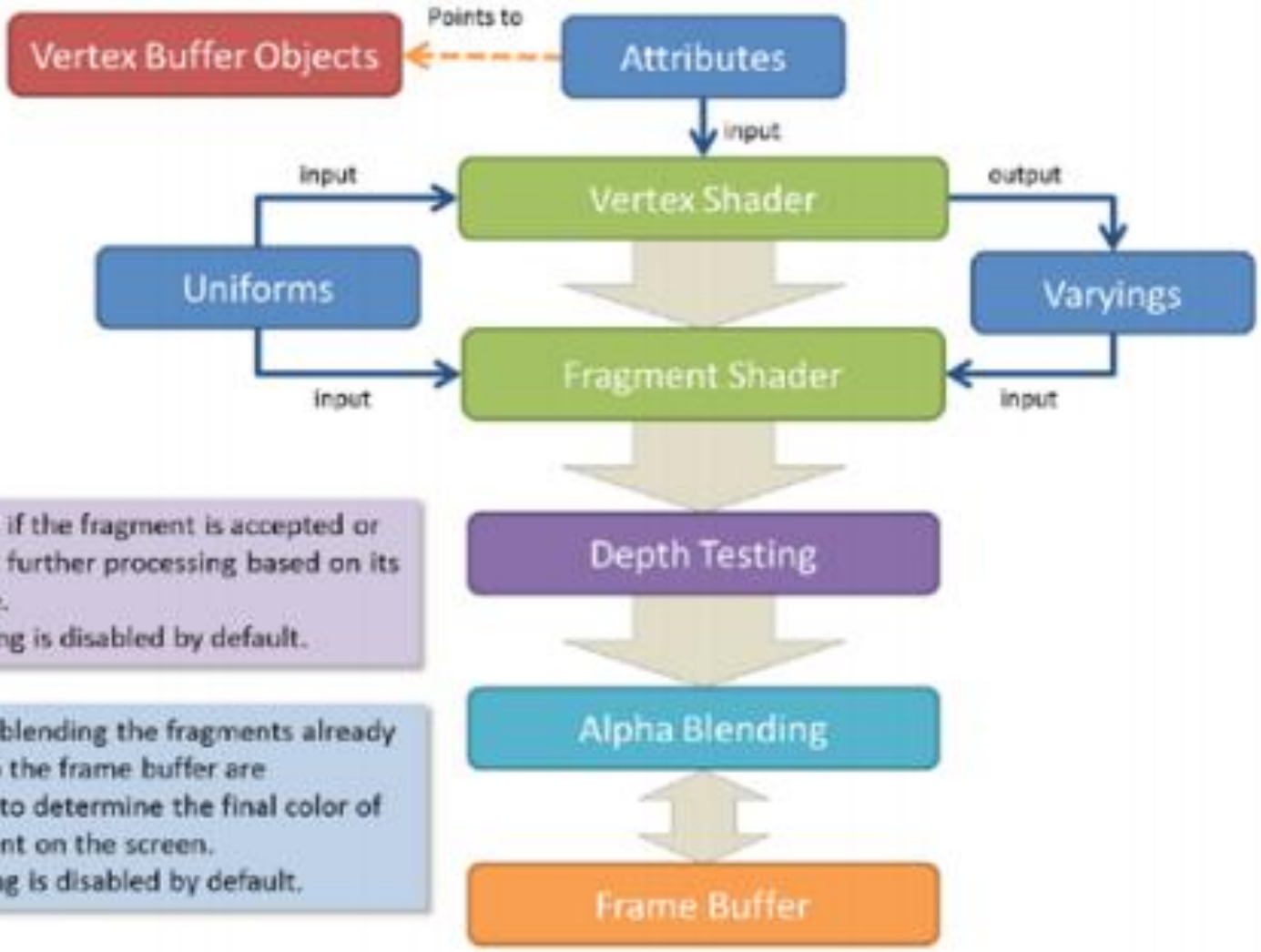
Creating transparencies corresponds to alter the fragments that we have already written to the frame buffer. Think for instance of a scene where there is one translucent object in front of an opaque object (from our camera view). For the scene to be rendered correctly we need to be able to see the opaque object through the translucent object. Therefore, the fragments that overlap between the far and the near objects need to be combined somehow to create the transparency effect.

To implement transparencies, we need to learn about two important WebGL concepts: depth testing and alpha blending.

Use Depth testing in rendering

Depth testing and alpha blending are two optional stages for the fragments once they have been processed by the fragment shader. If the depth test is not activated, all the fragments are automatically available for alpha blending. If the depth test is enabled, those fragments that fail the test will be automatically discarded by the pipeline and will no longer be available for any other operation. This means that discarded fragments will not be rendered.

WebGL Rendering Pipeline: Depth Testing and Alpha Blending



Determines if the fragment is accepted or rejected for further processing based on its depth value.
Depth Testing is disabled by default.

With alpha blending the fragments already written into the frame buffer are considered to determine the final color of that fragment on the screen.
Alpha Testing is disabled by default.

Depth testing

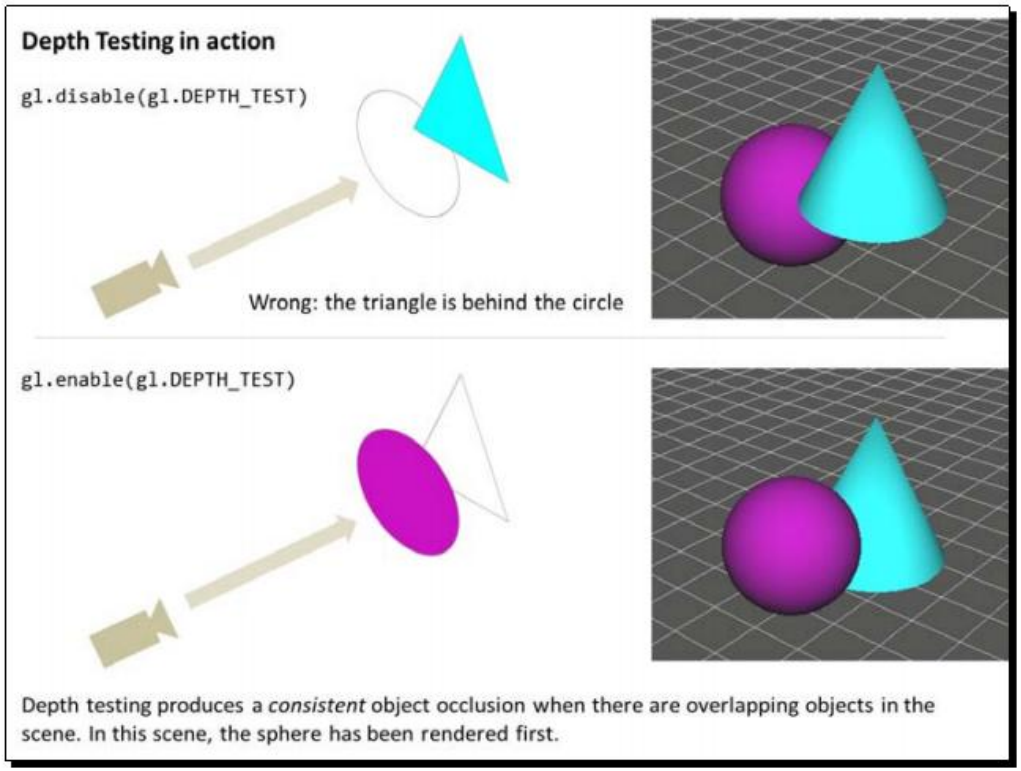
Each fragment that has been processed by the fragment shader carries an associated depth value. Though fragments are two-dimensional as they are going to be displayed on the screen, the depth value keeps the information of how distant the fragment is from the camera (screen). Depth values are stored in a special WebGL buffer named depth bufferor z-buffer. The z comes from the fact that x and y values correspond to the screen coordinates of the fragment while the z value measures distance perpendicular to the screen.
After the fragment has been calculated by the fragment shader, it is eligible for depth testing.
This only occurs if the depth test is enabled. Assuming that gl is the JavaScript variable that contains our WebGL context, we can enable depth testing by writing:

gl.enable(gl.DEPTH_TEST)

The depth test takes into consideration the depth value of a fragment and it compares it to the depth value for the same fragment coordinates already stored in the depth buffer. The depth test determines whether or not that fragment is accepted for further processing in the rendering pipeline.
Only the fragments that pass the depth test will be processed. Otherwise, any fragment that does not pass the depth test will be discarded.

Example on depth testing

When the depth test is enabled, only those fragments with a lower depth value than the corresponding fragments present in the depth buffer will be accepted.
Depth testing is a commutative operation with respect to the rendering order. This means that no matter which object gets rendered first, as long as depth testing is enabled, we will always have a consistent scene.



Depth function

In some applications, we could be interested in changing the default function of the depth-testing mechanism which discards fragments with a higher depth value than those fragments in the depth buffer. For that purpose WebGL provides the ***gl.depthFunc(function)***.

Parameter	Description
gl.NEVER	The depth test always fails
gl.LESS	Only fragments with a depth lower than current fragments on the depth buffer will pass the test
gL.LEQUAL	Fragments with a depth less than or equal to corresponding current fragments in the depth buffer will pass the test
gl.EQUAL	Only fragments with the same depth as current fragments on the depth buffer will pass the test
gl.NOTEQUAL	Only fragments that do not have the same depth value as fragments on the depth buffer will pass the test
gl.GEQUAL	Fragments with greater or equal depth value will pass the test
gl.GREATER	Only fragments with a greater depth value will pass the test
gl.ALWAYS	The depth test always passes

The depth test is disabled by default in WebGL. When enabled, if no depth function is set, the gl.LESS function is selected by default.

Alpha blending

A fragment is eligible for alpha blending if it has passed the depth test. However, when depth testing is disabled, all fragments are eligible for alpha blending. Alpha blending is enabled using the following line of code:
gl.enable(gl.BLEND);

For each eligible fragment the alpha blending operation reads the color present in the frame buffer for those fragment coordinates and creates a new color that is the result of a linear interpolation between the color previously calculated in the fragment shader ***(gl_FragColor)*** and the color already present in the frame buffer.

Blending function

With blending enabled, the next step is to define a blending function. This function will determine how the fragment colors coming from the object we are rendering (source) will be combined with the fragment colors already present in the frame buffer (destination).

We combine source and destination as follows:

Color Output = S * sW + D * dW

S: source color

- **S.rgb**: rgb components of the source color
- **S.a**: alpha component of the source color

D: destination color

- **D.rgb**: rgb components of the destination color
- **D.a**: alpha component of the destination color

sW: source scaling factor

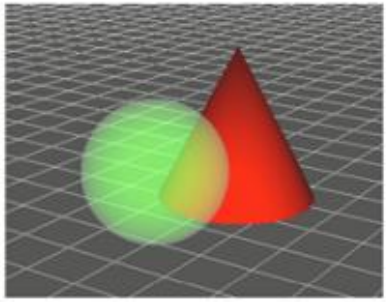
dW: destination scaling factor

Example in alfa blend

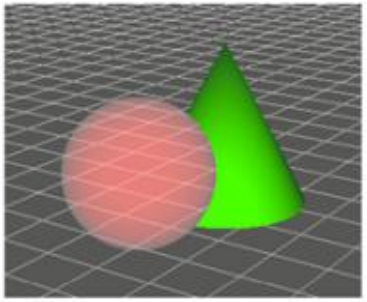
It is very important to notice here that the rendering order will determine what the source and the destination fragments are in the previous equations. Following the example from the previous section, if the sphere is rendered first, then it will become the destination of the blending operation because the sphere fragments will be already stored in the frame buffer when the cone is rendered. In other words, alpha blending is a non-commutative operation with respect to the rendering order.

Rendering order is relevant for blending operations

```
Output Color  = Source * sW + Destination * dW
```



Back to Front order
The cone is rendered first.
The overlapping sphere fragments pass the depth test and are available for blending.



Front to Back order
The sphere is rendered first.
The overlapping cone fragments do not pass the depth test. Blending is not possible.

Separate blending functions

It is also possible to determine how the RGB channels are going to be combined independently from the alpha channels. For that, we use the gl.blendFuncSeparatefunction.
We define two independent functions this way:

Color output = S.rgb * sW.rgb + D.rgb * dW.rgb
Alpha output = S.a * sW.a + D.a * dW.a

Here,
- sW.rgb: source scaling factor (only rgb)
- dW.rgb: destination scaling factor (only rgb)
- sW.a: source scaling factor for the source alpha value
- dW.a: destination scaling factor for the destination alpha value

Then, for example, we could have something as follows:
Color output = S.rgb * S.a + D.rbg * (1 - S.a)
Alpha output = S.a * 1 + D.a * 0

This would be translated into code as:
gl.blendFuncSeparate(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA, gl.ONE, gl.ZERO)

Blend equation

We could have the case where we do not want to interpolate the source and destination fragment colors by scaling them and adding them as shown before. It could be the case where we want to subtract one from the other. In that case, WebGL provides the *gl.blendEquationfunction*. This function receives one parameter that determines the operation on the scaled source and destination fragment colors.

gl.blendEquation(gl.FUNC_ADD) will correspond to:
Color output = S * sW + D *dW

gl.blendEquation(gl.FUNC_SUBTRACT) corresponds to:
Color output = S * sW - D *dW

gl.blendEquation(gl.FUNC_REVERSE_SUBTRACT)
Color output = D* dw – S*sW

As it is expected, it is also possible to define the blending equation separately for the RGB channels and for the alpha channel. For that, we use the l.blendEquationSeparate function.

Blend color

WebGL provides the scaling factors ***gl.CONSTANT_COLOR*** and **gl.ONE_MINUS_CONSTANT_COLOR**.

These scaling factors can be used with ***gl.blendFunc*** and with ***gl.blendFuncSeparate***. However, we need to establish beforehand what the blend color is going to be. We do so by invoking ***gl.blendColor***.

WebGL blending API

WebGL Function	Description
`gl.enable|disable (gl.BLEND)`	Enable/disable blending

`gl.blendFunc (sW, dW)`	Specify pixel arithmetic. Accepted values for *sW* and *dW* are:
	`ZERO`
	`ONE`
	`SRC_COLOR`
	`DST_COLOR`
	`SRC_ALPHA`
	`DST_ALPHA`
	`CONSTANT_COLOR`
	`CONSTANT_ALPHA`
	`ONE_MINUS_SRC_ALPHA`
	`ONE_MINUS_DST_ALPHA`
	`ONE_MINUS_SRC_COLOR`
	`ONE_MINUS_DST_COLOR`
	`ONE_MINUS_CONSTANT_COLOR`
	`ONE_MINUS_CONSTANT_ALPHA`
	In addition, *sW* can also be `SRC_ALPHA_SATURATE`

`gl.blendFuncSeparate(sW_rgb, dW_rgb, sW_a, dW_a)`	Specify pixel arithmetic for RGB and alpha components separately

WebGL Function	Description

`gl.blendEquation(mode)`	Specify the equation used for both the RGB blend equation and the alpha blend equation. Accepted values for mode are:
	`gl.FUNC_ADD`
	`gl.FUNC_SUBTRACT`
	`gl.FUNC_REVERSE_SUBTRACT`

`gl.blendEquationSeparate(modeRGB, modeAlpha)`	Set the RGB blend equation and the alpha blend equation separately
`gl.blendColor ( red, green, blue, alpha)`	Set the blend color

`gl.getParameter(pname)`	Just like with other WebGL variables, it is possible to query blending parameters using `gl.getParameter`.
	Relevant parameters are:
	`gl.BLEND`
	`gl.BLEND_COLOR`
	`gl.BLEND_DST_RGB`
	`gl.BLEND_SRC_RGB`
	`gl.BLEND_DST_ALPHA`
	`gl.BLEND_SRC_ALPHA`
	`gl.BLEND_EQUATION_RGB`
	`gl.BLEND_EQUATION_ALPHA`

Alpha blending modes

Depending on the parameter selection for sW and dW we can create different blending modes.
All blending modes depart from the already known formula:

Color output = S * (sW) + D * dW

Additive blending

Additive blending simply adds the colors of the source and destination fragments, creating a lighter image. We obtain additive blending by writing:
gl.blendFunc(gl.ONE, gl.ONE);

This assigns the weights for source and destination fragments sW and dW to 1. The color output will be:
Color output = S * 1 + D * 1
Color output = S + D
Since each color channel is in the [0, 1]range, this blending will clamp all values over 1.

When all channels are 1 this results in a white color.

Subtractive blending

Similarly, we can obtain subtractive blending by writing:
gl.blendEquation(gl.FUNC_SUBTRACT);
gl.blendFunc(gl.ONE, gl.ONE);

This will change the blending equation to:
Color output = S * (1) - D * (1)
Color output = S – D

Any negative values will be simply shown as zero. When all channels are negative this results in black color.

Multiplicative blending

We obtain multiplicative blending by writing:
gl.blendFunc(gl.DST_COLOR, gl.ZERO);

This will be reflected in the blending equation as:
Color output = S * (D) + D * (0)
Color output = S * D

The result will be always a darker blending.

Interpolative blending

If we set sWto S.a and dW to 1-S.a then:
Color output = S * S.a + D *(1-S.a)

This will create a linear interpolation between the source and destination color using the source alpha color S.a as the scaling factor.
In code, this is translated as:
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);

Interpolative blending allows us to create a transparency effect as long as the destination fragments have passed the depth test. This implies that the objects need to be rendered from back to front.

Textures

Hint. Canvas' coordinate system considers origin to be in the left bottom corner.
This is different from the origin in the left top corner in OPENGL.
Thus, textures should be Y flipped.

gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

Texture Coordinates

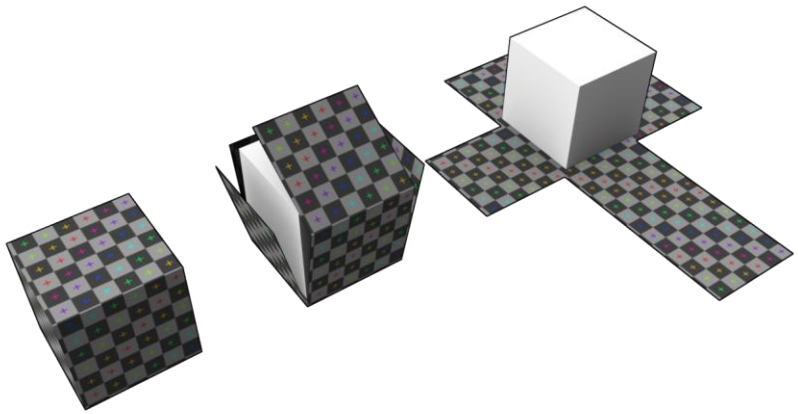
Implementation of a texture into the mesh is doing through a *vertex attribute* named **texture coordinates.**

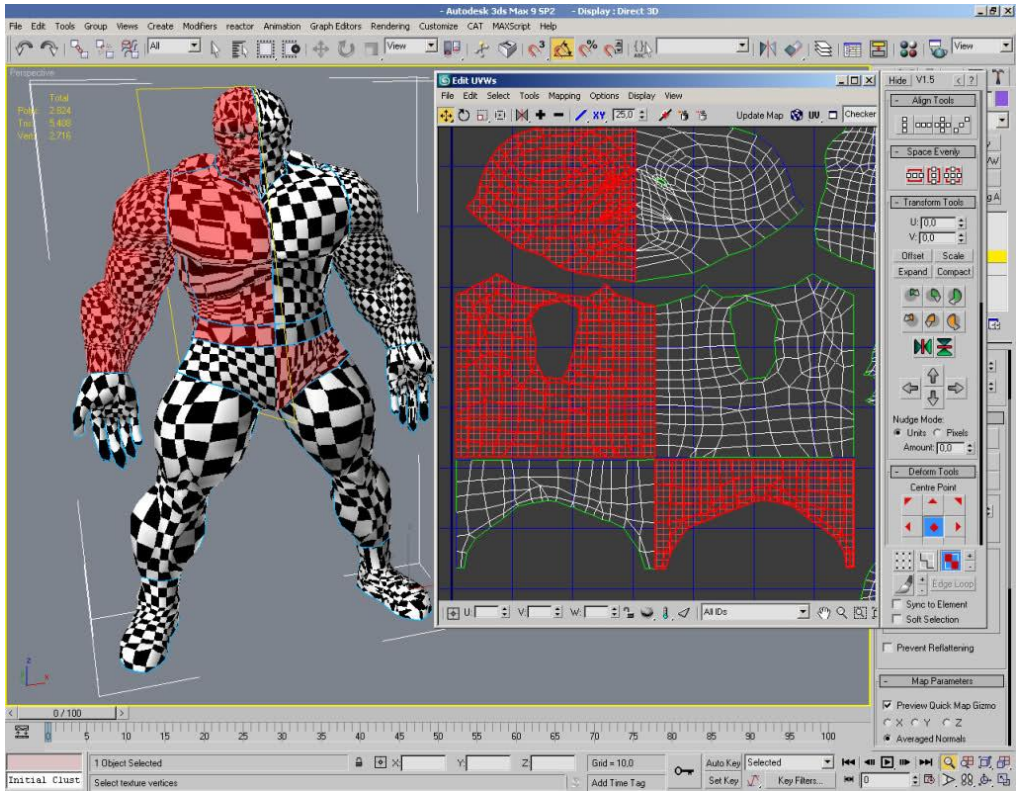
Texture coordinates are two-element float vectors that describe a location on the texture that coincides with that vertex. WebGL forces all the texture coordinates into a 0 to 1 range, where [0, 0] represents the top left-hand side corner of the texture and [1, 1] represents the bottom right-hand side corner, as is shown in the following image:

A vertex to the center of any texture, you would give it a texture coordinate of [0.5, 0.5]. This coordinate system holds true even for non- rectangular textures.

Unwrapping

Figuring out what the texture coordinates for your mesh should be, especially if the mesh is complex, can be one of the trickier parts of creating 3D resources, but fortunately most 3D modeling tools come with excellent utilities for laying out texture coordinates. This process is called **Unwrapping**.





Texture coordinates symbolism

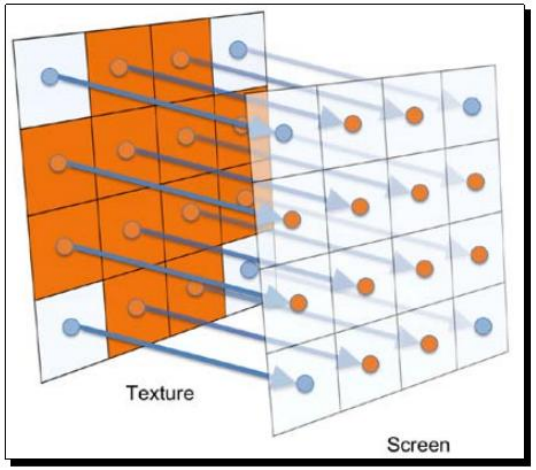
Just like the vertex position components are commonly represented with the characters X, Y, and Z, texture coordinates also have a common symbolic representation. Unfortunately, it's not consistent across all 3D software applications. OpenGL (and therefore WebGL) refers to the coordinates as S and T for the X and Y components respectively. However, DirectX and many popular modeling packages refer to them as U and V. As a result, you'll often see people referring to texture coordinates as "UVs" and Unwrapping as "UV Mapping".

Texture filter modes

if we zoom in on the textured geometry you would see that the texture begins to degrade.

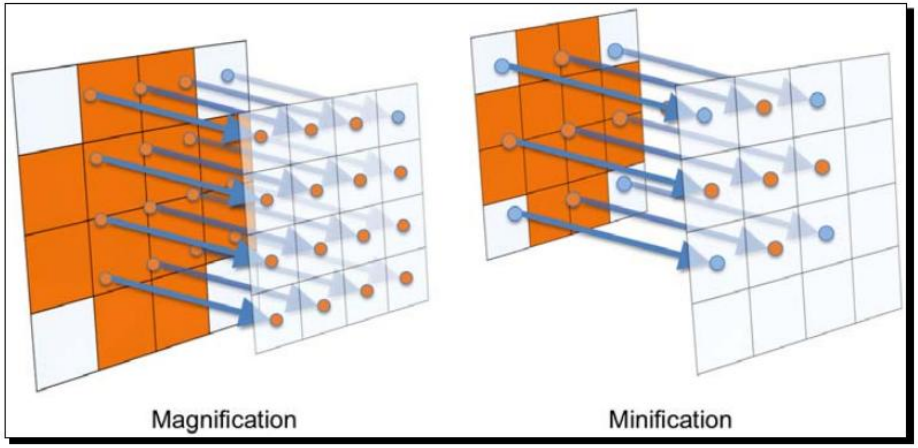
Why?

Vertex colors are interpolated, so that the fragment shader is provided a smooth gradient of color. Texture coordinates are interpolated in exactly the same way, with the resulting coordinates being provided to the fragment shader and used to sample color values from the texture. In a perfect situation, the texture would display at a 1:1 ratio on screen, meaning **each pixel of the texture** (known as **texels**) would take up exactly one pixel on screen. In this scenario, there would be no artifacts.



Magnification and Minification

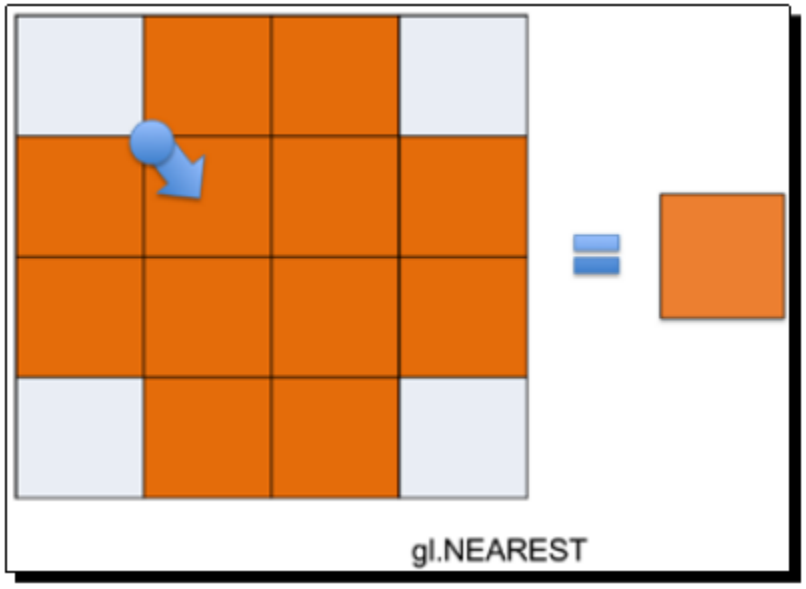
The reality of 3D applications, however, is that the textures are almost never displayed at their native resolution. We refer to these scenarios as **magnification** and **minification**, depending on whether the texture has a lower or higher resolution than the screen space it occupies. (pixels per inch)
When a texture is magnified or minified, there can be some ambiguity about what color the texture sampler should return. For example, consider the following diagram of sample points against a slightly magnified texture:



We need texture filtering to overcome pixelation and noise

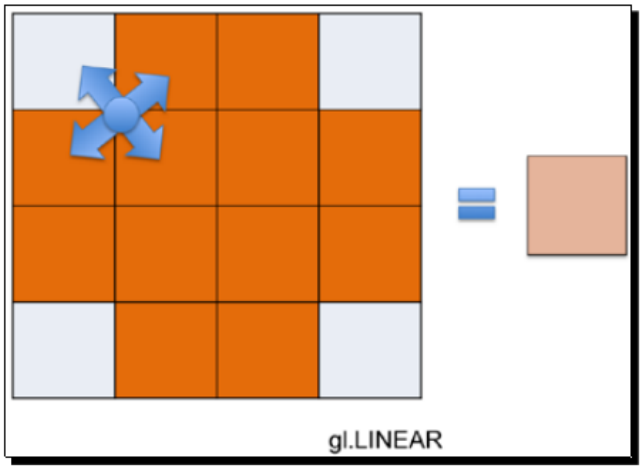
Filters NEAREST

Textures using the NEAREST filter always return the color of the texel whose center is nearest to the sample point. Using this mode textures will look blocky and pixilated when viewed up close, which can be useful for creating "retro" graphics. NEAREST can be used for both MIN and MAG filters.



Filter LINEAR

The LINEAR filter returns the weighted average of the four pixels whose centers are nearest to the sample point. This provides a smooth blending of texel colors when looking at textures close up, and generally is a much more desirable effect. This does mean that the graphics hardware has to read four times as many pixels per fragment, so naturally it's slower than NEAREST, but modern graphics hardware is so fast that this is almost never an issue. LINEAR can be used for both MIN and MAG filters. This filtering mode is also known as **bilinear filtering**.

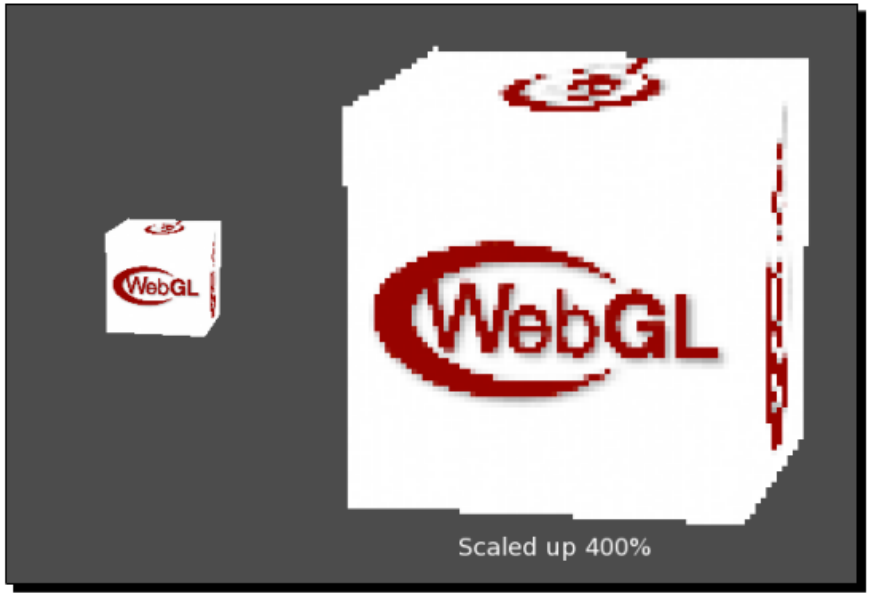


without

with

Mipmapping

Before we can discuss the remaining filter modes that are only applicable to TEXTURE_MIN_FILTER, we need to introduce a new concept: mipmapping.
A problem arises when sampling minified textures; even when using LINEAR filtering where the sample points can be so far apart that we can completely miss some details of the texture. As the view shifts, the texture fragments that we miss changes and the result is a shimmering effect. You can see this in action by setting the MIN filter in the demo to NEAREST or LINEAR, zooming out, and rotating the cube.



Mipmapping cont..

To avoid this, graphics cards can utilize a mipmap chain. Mipmaps are scaled-down copies of a texture, with each copy being exactly half the size of the previous one. If you were to show a texture and all of it's mipmaps in a row, it would look like this:

The advantage is that when rendering, the graphics hardware can choose the copy of the texture that most closely matches the size of the texture on screen and sample from it instead, which reduces the number of skipped texels and the jittery artifacts that accompany it. However, mipmapping is only used if you use the appropriate texture filters. The following TEXTURE_MIN_FILTER modes will utilize mipmaps in some fashion or the other.

Mipmapping filters

NEAREST_MIPMAP_NEAREST
This filter will select the mipmap that most closely matches the size of the texture on screen and sample from it using the NEAREST algorithm.

LINEAR_MIPMAP_NEAREST
This filter selects the mipmap that most closely matches the size of the texture on screen and sample from it using the LINEAR algorithm.

NEAREST_MIPMAP_LINEAR
This filter selects two mipmaps that most closely matches the size of the texture on screen and samples from both of them using the NEAREST algorithm. The color returned is a weighted average of those two samples.

LINEAR_MIPMAP_LINEAR
This filter selects two mipmaps that most closely matches the size of the texture on screen and samples from both of them using the LINEAR algorithm. The color returned is a weighted average of those two samples. This mode is also known as trilinear filtering. (it provides the best quality at the lowest performance)

Texture wrapping

In the previous section, we used texParameteri to set the filter mode for textures, but as you might expect from the generic function name, that's not all that it can do. Another texture behavior that we can manipulate is the texture wrapping mode

Texture wrapping describes the behavior of the sampler when the texture coordinates fall outside the range of 0-1. The wrapping mode can be set independently for both the S and T coordinates

Texture Wrapping Modes

CLAMP_TO_EDGE (stretch)
This wrap mode rounds any texture coordinates greater than 1 down to 1 and lower than 0 up to 0, "clamping" the values to the 0-1 range. Visually, this has the effect of repeating the border pixels of the texture indefinitely once the coordinates go out of the 0-1 range. Note that this is the only wrapping mode that is compatible with NPOT textures.

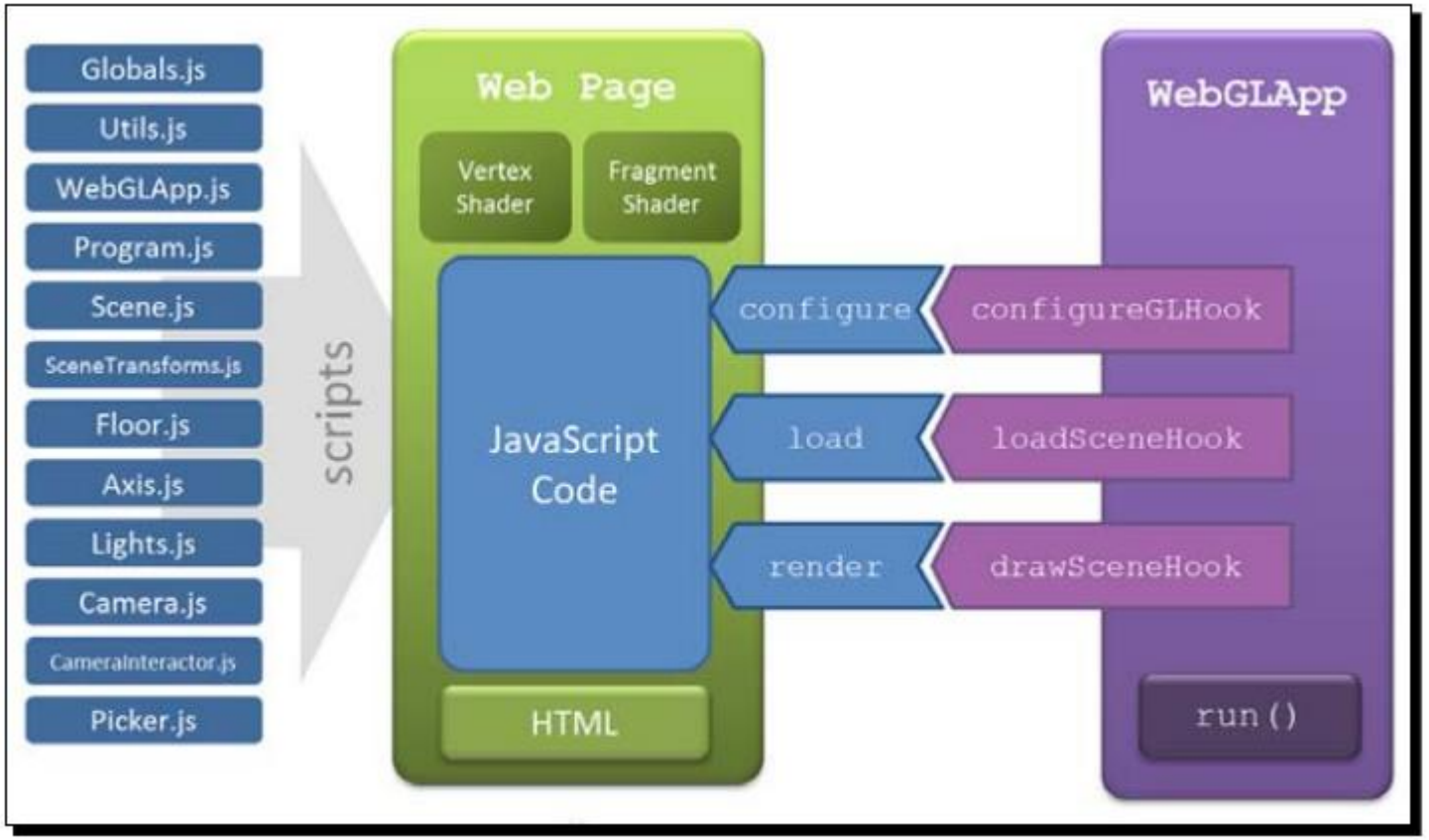
REPEAT
This is the default wrap mode, and the one that you'll probably use most often.
In mathematical terms this wrap mode simply ignores the integer part of the texture coordinate.

Texture Wrapping Modes cont..

MIRRORED_REPEAT
The algorithm for this mode is a little more complicated. If the coordinate's integer portion is even, the texture coordinates will be the same as with REPEAT. If the integer portion of the coordinate is odd, however, the resulting coordinate is 1 minus the fractional portion of the coordinate. This results in a texture that "flip-flops" as it repeats, with every other repetition being a mirror image

Creating an application

The Architecture



The Objects

Globals.js: Defines the global variables gl(WebGL context), prg(ESSL program), and the canvas width (c_width) and height (c_height).

Utils.js: Contains auxiliary functions such as getGLContextwhich tries to create a WebGL context for a given HTML5 canvas.

WebGLApp.js: It provides three function hooks, namely: configureGLHook, loadSceneHook, and drawSceneHookthat define the life cycle of a WebGL application.

configure: Here we create cameras, lights, and instantiate the Program.object.

load: Here we request objects from the web server by calling Scene.loadObject. We can also add locally generated geometry (such as the Floor) by calling Scene. addObject.

render(or draw): This is the function that is called every time when the rendering timer goes off. Here we will retrieve the objects from the Scene, one by one, and we will render them paying attention to their location (applying local transforms using the matrix stack), and their properties (passing the respective uniforms to the Program).

The Objects *cont..*

Program.js: Is composed of the functions that handle programs, shaders, and the mapping between JavaScript variables and ESSL uniforms.

Scene.js: Contains a list of objects to be rendered by WebGL.

SceneTransform.js: Contains the matrices discussed in the book: The Model-View matrix, the Camera matrix, the Perspective matrix, and the Normal matrix. It implements the matrix stack with the operations push and pop.

Lights.js: Simplifies the creation and managing of lights in the scene.

Camera.js: Contains a camera representation. We have developed two types of camera: orbiting and tracking.

CameraInteractor.js: Listens for mouse and keyboard events on the HTML5 canvas that it is being used. It interprets these events and then transforms them into camera actions.

Picker.js*: Provides color-based object picking.*
Floor.js*: Auxiliary object that when rendered appears like a rectangular mesh providing the floor reference for the scene.*
Axis.js*: Auxiliary object that represents the center of the scene.*

A Demo Application

- First of all, we need to define what the graphical user interface (GUI) is going to look like.
- Then, we will be adding WebGL support by creating a canvas element and obtaining the correspondent WebGL context.
- Simultaneously, we need to define and implement the Vertex Shader and Fragment Shader using ESSL.
- After that, we need to implement the three functions that constitute the lifecycle of our application: configure, load, and render.

Context Related Decisions

Complexity of the models

A real-world application is different from a proof of concept demo in that the models that we will be loading are much more detailed than simple spheres, cones, and other geometric figures. Usually, models have lots of vertices conforming very complicated configurations that give the level of detail and realism that people would expect. Also, in many cases, these models are accompanied by one or more textures.

We may import car models created with Blender into a WebGL scene. To do so, we will export them to an intermediary file format called OBJ and then we will parse OBJ files into JSON files.

Shader quality

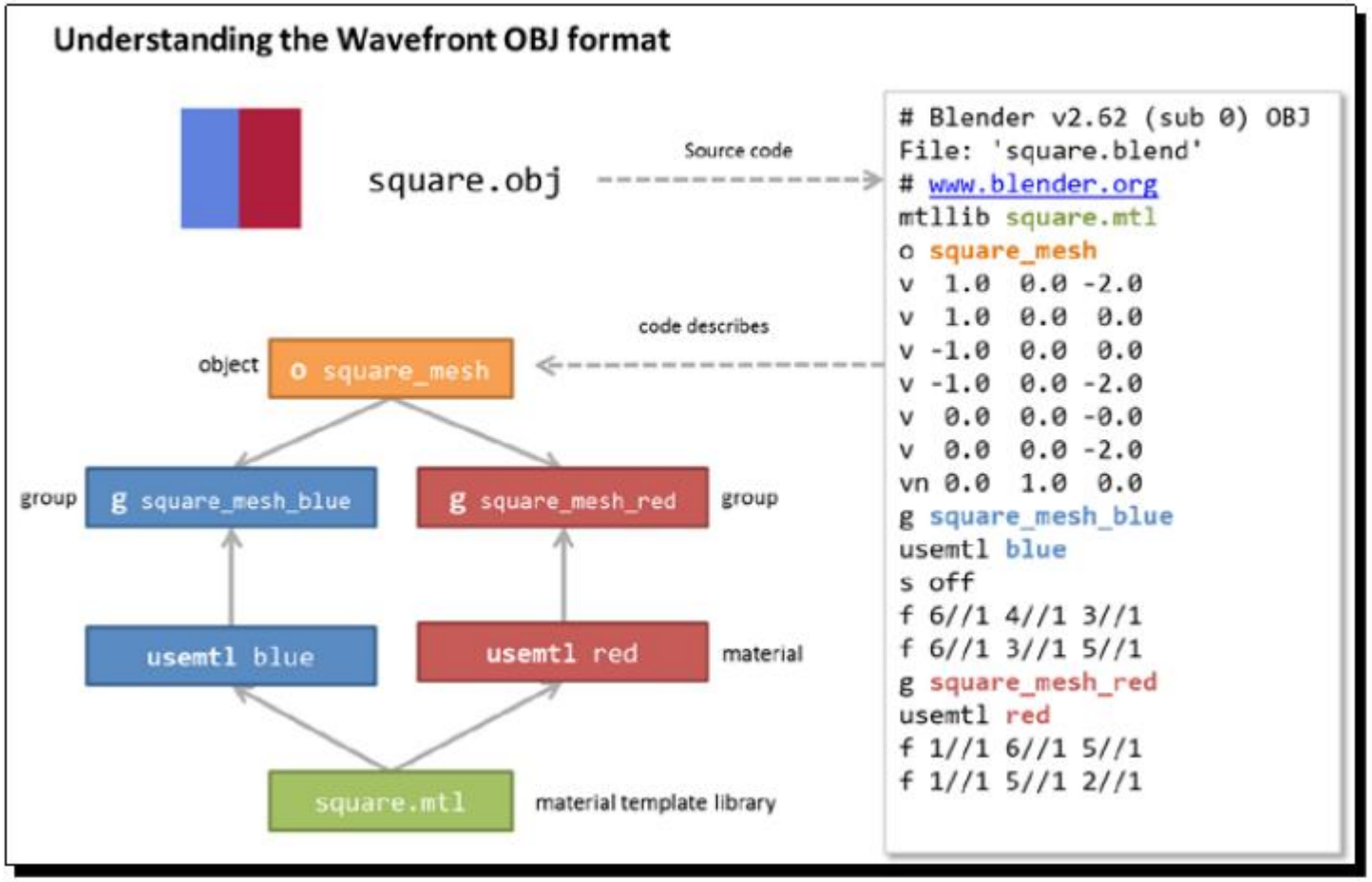
Because we will be using complex models, such as cars, we will see that there is a need to develop shaders that can render the different materials that our models are made of. This is not a big deal for us since the shaders that we previously developed can handle diffuse, specular, and ambient components for materials. In Blender, we will select the option to export materials when generating the OBJ files. When we do so, Blender will generate a second file known as the Material Template Library(MTL). Also, our shaders will use Phong shading, Phong lighting, and will support multiple lights.

Context Related Decisions *cond…*

Network delays and bandwidth consumption

Due to the nature of WebGL, we will need to download the geometry and the textures from a web server. Depending on the quality of the network connection and the amount of data that needs to be transferred this can take a while. There are several strategies that you could investigate, such as geometry compression. Another alternative is background data downloading (using AJAX for example) while the application is idle or the user is busy and not waiting for something to download.

Understanding the OBJ format



referring to the Material Template Library that this OBJ file is
using. Such line will start with the keyword mtllibfollowed by the name of the materials library file: **mtllib square.mtl**

There are several ways in which geometries can be grouped into entities in an OBJ file. We can find lines starting with the prefix **o** followed by the object name; or by the prefix **g**,
followed again by the group name:
o squares_mesh

After an object declaration, the following lines will refer to vertices (v) and optionally to vertex normals (vn) and texture coordinates (vt). It is important to mention that vertices are shared by all the groups in an object in the OBJ format. That is, you will not find lines referring to vertices when defining a group because it is assumed that all vertex data was defined first when the object was defined:

v 1.0 0.0 -2.0
v 1.0 0.0 0.0
v -1.0 0.0 0.0
v -1.0 0.0 -2.0
v 0.0 0.0 0.0
v 0.0 0.0 -2.0
vn 0.0 1.0 0.0

In our case, we have instructed Blender to export group materials. This means that each part of the object that has different set of material properties will appear in the OBJ file as a group. In this example, we are defining an object with two groups (***squares_mesh_blue and squares_mesh_red***) and two corresponding materials (blue and red):

g squares_mesh_blue

If materials are being used, the line after the group declaration will be the material that is being used for that group. Here only the name of the material is required. It is assumed that the material properties for this material are defined in the Material Template Library file that was declared at the beginning of the OBJ file:

usemtl blue

The lines that start with the prefix **s** refer to ***smooth shading across polygons***. We mention it here in case you see it on your files but we will not be using this definition when parsing the OBJ files into JSON files:

s off

The lines that start with **f** refer to ***faces***. There are different ways to represent faces. Let's see them:

Vertex:

f i1 i2 i3...

In this configuration, every face element corresponds to a vertex index. Depending on the number of indices per face, you could have triangular, rectangular, or polygonal faces.

Vertex / Texture Coordinate:

f i1/t1 i2/t2 i3/t3...

In this combination, every vertex index appears followed by a slash sign and a texture coordinate index. You will normally find this combination when texture coordinates are defined at the object level with vt.

Vertex / Texture Coordinate / Normal:

f i1/t1/n1 i2/t2/n2 i3/t3/n3...

Here a normal index has been added as the third element of the configuration. If both texture coordinates and vertex normals are defined at the object level, you most likely see this configuration at the group level.

Vertex // Normal:

There could also be a case where normals are defined but not texture coordinates. In this case, the second part of the face configuration is missing:

f i1//n1 i2//n2 i3//n3...

This is the case for square.obj, which looks like this:

f 6//1 4//1 3//1

f 6//1 3//1 5//1

Please notice that faces are defined using indices. In our example, we have defined a square divided in two parts. Here we can see that all vertices share the same normal identified with index 1.