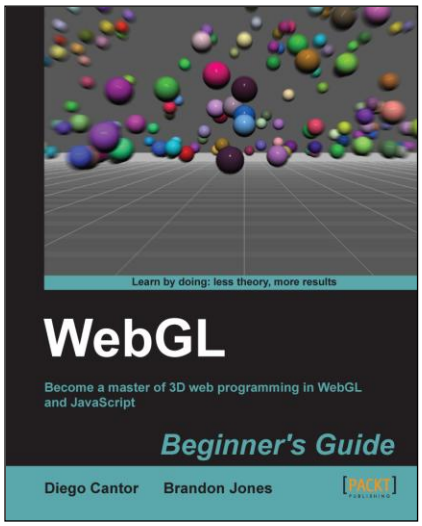
[PACKT] PUBLISHING

WebGL Beginner's Guide

Diego Cantor

Brandon Jones



Chapter No. 4

"Camera"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.4 "Camera"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Diego Hernando Cantor Rivera is a Soft ware Engineer born in 1980 in Bogota, Colombia. Diego completed his undergraduate studies in 2002 with the development of a computer vision system that tracked the human gaze as a mechanism to interact with computers.

Later on, in 2005, he finished his master's degree in Computer Engineering with emphasis in Soft ware Architecture and Medical Imaging Processing. During his master's studies, Diego worked as an intern at the imaging processing laboratory CREATIS in Lyon, France and later on at the Australian E-Health Research Centre in Brisbane, Australia.

Diego is currently pursuing a PhD in Biomedical Engineering at Western University in London, Canada, where he is involved in the development augmented reality systems for neurosurgery.

When Diego is not writing code, he enjoys singing, cooking, travelling, watching a good play, or bodybuilding.

Diego speaks Spanish, English, and French.

Brandon Jones has been developing WebGL demos since the technology first began appearing in browsers in early 2010. He finds that it's the perfect combination of two aspects of programming that he loves, allowing him to combine eight years of web development experience and a life-long passion for real-time graphics.

Brandon currently works with cutting-edge HTML5 development at Motorola Mobility.

I'd like to thank my wife, Emily, and my dog, Cooper, for being very patient with me while writing this book, and Zach for convincing me that I should do it in the first place.

WebGL Beginner's Guide

WebGL is a new web technology that brings hardware-accelerated 3D graphics to the browser without requiring the user to install additional soft ware. As WebGL is based on OpenGL and brings in a new concept of 3D graphics programming to web development, it may seem unfamiliar to even experienced web developers.

Packed with many examples, this book shows how WebGL can be easy to learn despite its unfriendly appearance. Each chapter addresses one of the important aspects of 3D graphics programming and presents different alternatives for its implementation. The topics are always associated with exercises that will allow the reader to put the concepts to the test in an immediate manner.

WebGL Beginner's Guide presents a clear road map to learning WebGL. Each chapter starts with a summary of the learning goals for the chapter, followed by a detailed description of each topic. The book offers example-rich, up-to-date introductions to a wide range of essential WebGL topics, including drawing, color, texture, transformations, framebuffers, light, surfaces, geometry, and more. Each chapter is packed with useful and practical examples that demonstrate the implementation of these topics in a WebGL scene. With each chapter, you will "level up" your 3D graphics programming skills. This book will become your trustworthy companion filled with the information required to develop cool-looking 3D web applications with WebGL and JavaScript.

What This Book Covers

Chapter 1, Getting Started with WebGL, introduces the HTML5 canvas element and describes how to obtain a WebGL context for it. After that, it discusses the basic structure of a WebGL application. The virtual car showroom application is presented as a demo of the capabilities of WebGL. This application also showcases the different components of a WebGL application.

Chapter 2, Rendering Geometry, presents the WebGL API to define, process, and render objects. Also, this chapter shows how to perform asynchronous geometry loading using AJAX and JSON.

Chapter 3, Lights!, introduces ESSL the shading language for WebGL. This chapter shows how to implement a lighting strategy for the WebGL scene using ESSL shaders. The theory behind shading and reflective lighting models is covered and it is put into practice through several examples.

Chapter 4, Camera, illustrates the use of matrix algebra to create and operate cameras in WebGL. The Perspective and Normal matrices that are used in a WebGL scene are also described here. The chapter also shows how to pass these matrices to ESSL shaders so they can be applied to every vertex. The chapter contains several examples that show how to set up a camera in WebGL.

Chapter 5, Action, extends the use of matrices to perform geometrical transformations (move, rotate, scale) on scene elements. In this chapter the concept of matrix stacks is discussed. It is shown how to maintain isolated transformations for every object in the scene using matrix stacks. Also, the chapter describes several animation techniques using matrix stacks and JavaScript timers. Each technique is exemplified through a practical demo.

Chapter 6, Colors, Depth Testing, and Alpha Blending, goes in depth about the use of colors in ESSL shaders. This chapter shows how to define and operate with more than one light source in a WebGL scene. It also explains the concepts of Depth Testing and Alpha Blending, and it shows how these features can be used to create translucent objects. The chapter contains several practical exercises that put into practice these concepts.

Chapter 7, Textures, shows how to create, manage, and map textures in a WebGL scene. The concepts of texture coordinates and texture mapping are presented here. This chapter discusses different mapping techniques that are presented through practical examples. The chapter also shows how to use multiple textures and cube maps.

Chapter 8, Picking, describes a simple implementation of picking which is the technical term that describes the selection and interaction of the user with objects in the scene. The method described in this chapter calculates mouse-click coordinates and determines if the user is clicking on any of the objects being rendered in the canvas. The architecture of the solution is presented with several callback hooks that can be used to implement logic-specific application. A couple of examples of picking are given.

Chapter 9, Putting It All Together, ties in the concepts discussed throughout the book. In this chapter the architecture of the demos is reviewed and the virtual car showroom application outlined in *Chapter 1, Getting Started with WebGL,* is revisited and expanded. Using the virtual car showroom as the case study, this chapter shows how to import Blender models into WebGL scenes and how to create ESSL shaders that support the materials used in Blender.

Chapter 10, Advanced Techniques, shows a sample of some advanced techniques such as post-processing effects, point sprites, normal mapping, and ray tracing. Each technique is provided with a practical example. After reading this WebGL Beginner's Guide you will be able to take on more advanced techniques on your own.

4
Camera

In this chapter, we will learn more about the matrices that we have seen in the source code. These matrices represent transformations that when applied to our scene, allow us to move things around. We have used them so far to set the camera to a distance that is good enough to see all the objects in our scene and also for spinning our Nissan GTS model (Animate button in ch3_Nissan.html). In general, we move the camera and the objects in the scene using matrices.

The bad news is that you will not see a camera object in the WebGL API, only matrices. The good news is that having matrices instead of a camera object gives WebGL a lot of flexibility to represent complex animations (as we will see in *Chapter 5*, *Action*). In this chapter, we will learn what these matrix transformations mean and how we can use them to define and operate a virtual camera.

In this chapter, we will:

- Understand the transformations that the scene undergoes from a 3D world to a 2D screen
- Learn about affine transformations
- Map matrices to ESSL uniforms
- 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

WebGL does not have cameras

This statement should be shocking! How is it that there are no cameras in a 3D computer graphics technology? Well, let me rephrase this in a more amicable way. WebGL does not have a camera object that you can manipulate. However, we can assume that what we see rendered in the canvas is what our camera captures. In this chapter, we are going to solve the problem of how to represent a camera in WebGL. The short answer is we need 4x4 matrices.

Every time that we move our camera around, we will need to update the objects according to the new camera position. To do this, we need to systematically process each vertex applying a transformation that produces the new viewing position. Similarly, we need to make sure that the object normals and light directions are still consistent after the camera has moved. In summary, we need to analyze two different types of transformations: vertex (points) and normal (vectors).

Vertex transformations

Objects in a WebGL scene go through different transformations before we can see them on our screen. Each transformation is encoded by a 4x4 matrix, as we will see later. How do we multiply vertices that have three components (*x,y,z*) by a 4x4 matrix? The short answer is that we need to augment the cardinality of our tuples by one dimension. Each vertex then will have a fourth component called the homogenous coordinate. Let's see what they are and why they are useful.

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**.

Homogeneous coordinates make possible to solve a system of linear equations where each equation represents a line that is parallel with all the others in the system. Let's remember here that in Euclidian Space, a system like that does not have solutions, because there are not intersections. However, in Projective Space, this system has a solution—the lines will intersect at infinite. This fact is represented by the perspective component having a value of zero. A good physical analogy of this idea is the image of train tracks: parallel lines that touch in the vanishing point when you look at them.

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*:

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

$$v(x, y, z) = h(x, y, z, 1)$$

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

As a matter of fact, this is what we have been doing so far! Let's go back to one of the shaders we discussed in the last chapter: the Phong vertex shader. The code looks like the following:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat4 uNMatrix;

varying vec3 vNormal;
varying vec3 vEyeVec;

void main(void) {
     //Transformed vertex position
     vec4 vertex = uMVMatrix * vec4(aVertexPosition, 1.0);

     //Transformed normal position
     vNormal = vec3(uNMatrix * vec4(aVertexNormal, 0.0));

     //Vector Eye
     vEyeVec = -vec3(vertex.xyz);

     //Final vertex position
     gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
```

Please notice that for the `aVertexPosition` attribute, which contains a vertex of our geometry, we create a 4-tuple from the 3-tuple that we receive. We do this with the ESSL construct `vec4()`. ESSL knows that `aVertexPosition` is a `vec3` and therefore we only need the fourth component to create a `vec4`.

To pass from Homogeneous coordinates to Euclidean coordinates, we divide by *w*

To pass from Euclidean coordinates to Homogeneous coordinates, we add *w* =1

Homogeneous coordinates with *w* = 0 represent a point at infinity

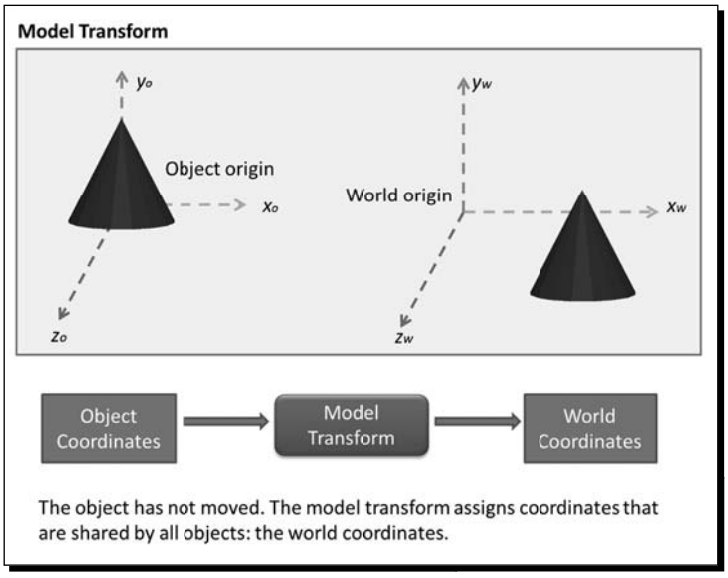
There is one more thing you should know about Homogeneous coordinates—while vertices have a Homogeneous coordinate w = 1, vectors have a Homogeneous coordinate w = 0. This is the reason why, in the Phong vertex shader, the line that processes the normals looks like this:

```
vNormal = vec3(uNMatrix * vec4(aVertexNormal, 0.0));
```

To code vertex transformations, we will be using Homogeneous coordinates unless indicated otherwise. Now let's see the different transformations that our geometry undergoes to be displayed on screen.

Model transform

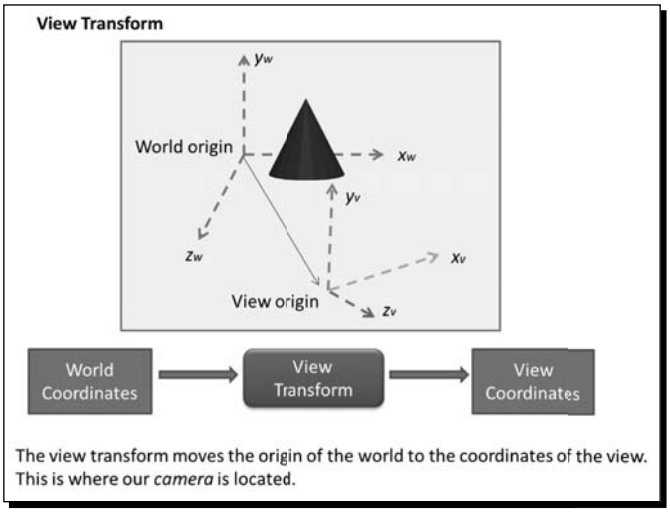
We start our analysis from the object coordinate system. It is in this space where vertex coordinates are specified. Then if we want to translate or move objects around, we use a matrix that encodes these transformations. This matrix is known as the **model matrix**. Once we multiply the vertices of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



While in object coordinates, each object is free to define where its origin is and then specify where its vertices are with respect to this origin, in world coordinates, the origin is shared by all the objects. World coordinates allow us to know where objects are located with respect to each other. It is with the model transform that we determine where the objects are in the 3D world.

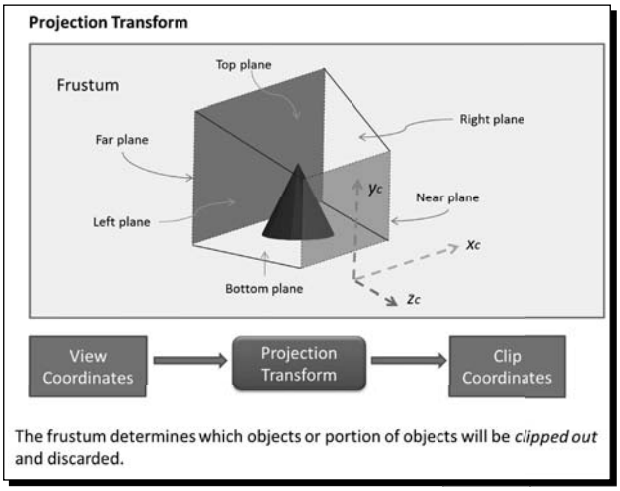
View transform

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**. We multiply this matrix by the vertex coordinates obtained by the model transform. The result of this operation is a new set of vertex coordinates whose origin is the view origin. It is in this coordinate system that our camera is going to operate. We will go back to this later in the chapter.



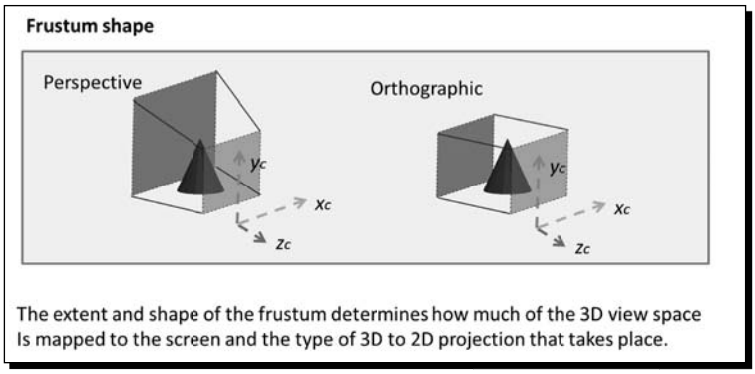
Projection transform

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), as shown in the following diagram:



These six planes are encoded in the **Perspective matrix**. Any vertices lying outside of the frustum after applying the transformation are *clipped out* and discarded from further processing. Therefore, the frustum defines, and the projection matrix that encodes the frustum produces, *clipping coordinates*.

The shape and extent of the frustum determines the type of projection from the 3D viewing space to the 2D screen. If the far and near planes have the same dimensions, then the frustum will determine an *orthographic* projection. Otherwise, it will be a *perspective* projection, as shown in the following diagram:



Up to this point, we are still working with Homogeneous coordinates, so the clipping coordinates have four components: *x*, *y*, *z*, and *w*. The clipping is done by comparing the *x*, *y*, and *z* components against the Homogeneous coordinate *w*. If any of them is more than, *+w*, or less than, *–w* , then that vertex lies outside the frustum and is discarded.

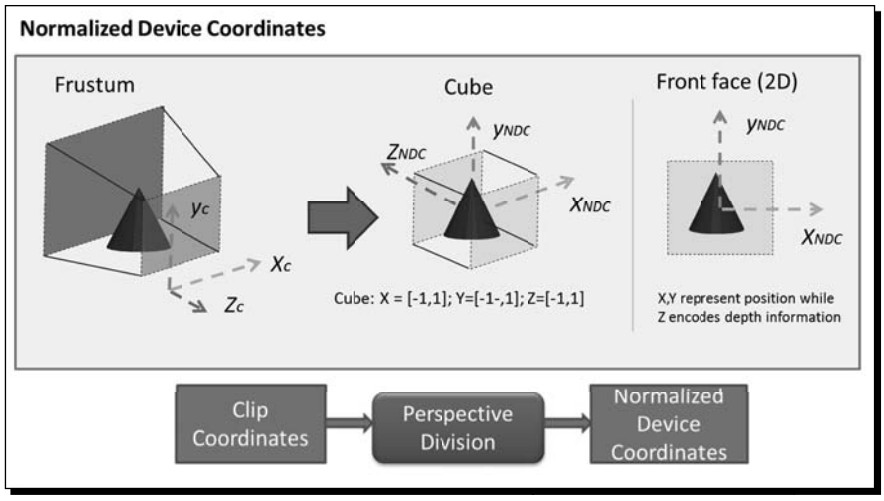
Perspective division

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)**.

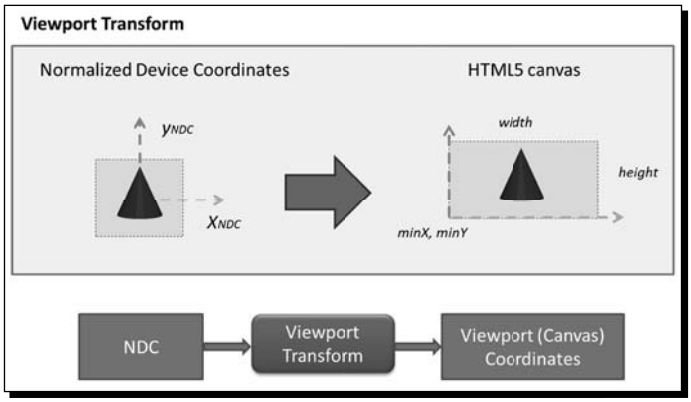
Normalized device coordinates are obtained by dividing the clipping coordinates by the w component. This is the reason why this step is known as *perspective division*. Also, please remember that when you divide by the Homogeneous coordinate, we go from projective space (4-components) to Euclidean space (3-components), so NDC only has three components. In the NDC space, the *x* and *y* coordinates represent the location of your vertices on a normalized 2D screen, while the z-coordinate encodes depth information, which is the relative location of the objects with respect to the near and far planes. Though, at this point, we are working on a 2D screen, we still keep the depth information. This will allow WebGL to determine later how to display overlapping objects based on their distance to the near plane. When using normalized device coordinates, the depth is encoded in the z-component.

The perspective division transforms the viewing frustum into a cube centered in the origin with minimum coordinates [-1,-1,-1] and maximum coordinates [1,1,1]. Also, the direction of the z-axis is inverted, as shown in the following figure:



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, as shown in the following figure:



Unlike the previous cases, the viewport transform is not generated by a matrix transformation. In this case, we use the WebGL `viewport` function. We will learn more about this function later in the chapter. Now it is time to see what happens to normals.

Normal transformations

Whenever vertices are transformed, **normal vectors** should also be transformed, so they point in the right direction. We could think of using the Model-View matrix that transforms vertices to do this, but there is a problem: The Model-View matrix will not always keep the perpendicularity of normals.

This problem occurs if there is a unidirectional (one axis) scaling transformation or a shearing transformation in the Model-View matrix. In our example, we have a triangle that has undergone a scaling transformation on the y-axis. As you can see, the normal N' is not normal anymore after this kind of transformation. How do we solve this?

Calculating the Normal matrix

If you are not interested in finding out how we calculate the Normal matrix and just want the answer, please feel free to jump to the end of this section. Otherwise, stick around to see some linear algebra in action!

Let's start from the mathematical definition of perpendicularity. Two vectors are perpendicular if their dot product is zero. In our example:

$$N.S = 0$$

Here, *S* is the surface vector and it can be calculated as the difference of two vertices, as shown in the previous diagram at the beginning of this section.

Let *M* be the Model-View matrix. We can use *M* to transform *S* as follows:

$$S' = MS$$

This is because *S* is the difference of two vertices and we use *M* to transform vertices onto the viewing space.

We want to find a matrix *K* that allows us to transform normals in a similar way. For the normal *N*, we want:

$$N' = KN$$

For the scene to be consistent after obtaining *N'* and *S'*, these two need to keep the perpendicularity that the original vectors *N* and *S* had. This is:

$$N'.S' = 0$$

Substituting *N'* and *S'*:

$$(KN).(MS) = 0$$

A dot product can also be written as a vector multiplication by transposing the first vector, so we have that this still holds:

$$(KN)^T(MS) = 0$$

The transpose of a product is the product of the transposes in the reverse order:

$$N^TK^TMS = 0$$

Grouping the inner terms:

$$N^T(K^TM)S = 0$$

Now remember that $N.S = 0$ so $N^TS = 0$ (again, a dot product can be written as a vector multiplication). This means that in the previous equation, (K^TM) needs to be the identity matrix *I*, so the original condition of *N* and *S* being perpendicular holds:

$$K^TM = I$$

Applying a bit of algebra:

$K^TMM^{-1} = IM^{-1} = M^{-1}$	multiply by the inverse of M on both sides
$K^T(I) = M^{-1}$	because $MM^{-1} = I$
$(K^T)^T = (M^{-1})^T$	transposing on both sides
$K = (M^{-1})^T$	Double transpose of *K* is the original matrix *K*.

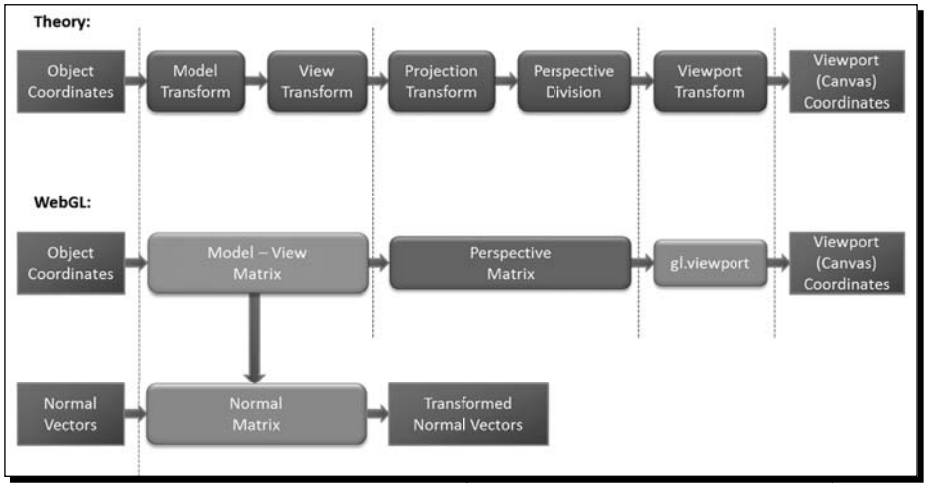
Conclusions:

- *K* is the correct matrix transform that keeps the normal vectors being perpendicular to the surface of the object. We call *K* the **Normal matrix**.

- *K* is obtained by transposing the inverse of the Model-View matrix (M in this example).
- We need to use *K* to multiply the normal vectors so they keep being perpendicular to surface when these are transformed.

WebGL implementation

Now let's take a look at how we can implement vertex and normal transformations in WebGL. The following diagram shows the theory that we have learned so far and it shows the relationships between the steps in the theory and the implementation in WebGL.



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).

   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.

JavaScript matrices

WebGL does not provide its own methods to perform operations on matrices. All WebGL does is it provides a way to pass matrices to the shaders (as uniforms). So, we need to use a JavaScript library that enables us to manipulate matrices in JavaScript. In this book, we have used `glMatrix` to manipulate matrices. However, there are other libraries available online that can do this for you.

We used `glMatrix` to manipulate matrices in this book. You can find more information about this library here: `https://github.com/toji/gl-matrix`. And the documentation (linked further down the page) can be found at: `http://toji.github.com/gl-matrix/doc`

These are some of the operations that you can perform with `glMatrix`:

Operation	Syntax	Description
Creation	`var m = mat4.create()`	Creates the matrix `m`
Identity	`mat4.identity(m)`	Sets `m` as the identity matrix of rank 4
Copy	`mat4.set(origin,target)`	Copies the matrix `origin` into the matrix `target`
Transpose	`mat4.transpose(m)`	Transposes matrix `m`
Inverse	`mat4.inverse(m)`	Inverts `m`
Rotate	`mat4.rotate(m,r,a)`	Rotates the matrix `m` by `r` radians around the axis `a` (this is a 3-element array [*x*,*y*,*z*]).

`glMatrix` also provides functions to perform other linear algebra operations. It also operates on vectors and matrices of rank 3. To get the full list, visit `https://github.com/toji/gl-matrix`

Mapping JavaScript matrices to ESSL uniforms

As the Model-View and Perspective matrices do not change during a single rendering step, they are passed *as uniforms* to the shading program. For example, if we were applying a translation to an object in our scene, we would have to paint the whole object in the new coordinates given by the translation. Painting the whole object in the new position is achieved in exactly one rendering step.

However, before the rendering step is invoked (by calling `drawArrays` or `drawElements`, as we saw in *Chapter 2*, *Rendering Geometry*), we need to make sure that the shaders have an updated version of our matrices. We have seen how to do that for other uniforms such as light and color properties. The method map JavaScript matrices to uniforms is similar to the following:

First, we get a JavaScript reference to the uniform with:

```
var reference= getUniformLocation(Object program, String uniformName)
```

Then, we use the `reference` to pass the matrix to the shader with:

```
gl.uniformMatrix4fv(WebGLUniformLocation reference, bool transpose,
float[] matrix);
```

`matrix` is the JavaScript matrix variable.

As it is the case for other uniforms, ESSL supports 2, 3, and 4-dimensional matrices:

`uniformMatrix[234]fv(ref,transpose,matrix)`: will load 2x2, 3x3, or 4x4 matrices (corresponding to 2, 3, or 4 in the command name) of floating points into the uniform referenced by ref. The type of ref is `WebGLUniformLocation`. For practical purposes, it is an integer number. According to the specification, the transpose value must be set to `false`. The matrix uniforms are always of floating point type (`f`). The matrices are passed as 4, 9, or 16 element vectors (`v`) and are always specified in a column-major order. The matrix parameter can also be of type `Float32Array`. This is one of JavaScript's typed arrays. These arrays are included in the language to provide access and manipulation of raw binary data, therefore increasing efficiency.

Working with matrices in ESSL

Let's revisit the Phong vertex shader, which was introduced in the last chapter. Please pay attention to the fact that matrices are defined as `uniform mat4`.

In this shader, we have defined three matrices:

- `uMVMatrix`: the Model-View matrix
- `uPMatrix`: the Perspective matrix
- `uNMatrix`: the Normal matrix

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;

varying vec3 vNormal;
varying vec3 vEyeVec;

void main(void) {
     //Transformed vertex position
     vec4 vertex = uMVMatrix * vec4(aVertexPosition, 1.0);
```

```
        //Transformed normal vector
        vNormal = uNMatrix * aVertexNormal;

        //Vector Eye
        vEyeVec = -vec3(vertex.xyz);

        //Final vertex position
        gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition,
1.0);
}
```

In ESSL, the multiplication of matrices is straightforward, that is, you do not need to multiply element by element, but as ESSL knows that you are working with matrices, it performs the multiplication for you.

```
gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
```

The last line of this shader assigns a value to the predefined `gl_Position` variable. This will contain the clipping coordinates for the vertex that is currently being processed by the shader. We should remember here that the shaders work in parallel: each vertex is processed by an instance of the vertex shader.

To obtain the clipping coordinates for a given vertex, we need to multiply first by the Model-View matrix and then by the Projection matrix. To achieve this, we need to multiply to the left (because matrix multiplication is not commutative).

Also, notice that we have had to augment the `aVertexPosition` attribute by including the Homogeneous coordinate. This is because we have always defined our geometry in Euclidean space. Luckily, ESSL lets us do this just by adding the missing component and creating a vec4 on the fly. We need to do this because both the Model-View matrix and the Perspective matrix are described in homogeneous coordinates (4 rows by 4 columns).

Now that we have seen how to map JavaScript matrices to ESSL uniforms in our shaders, let's talk about how to operate with the three matrices: the Model-View matrix, the Normal matrix, and the Perspective matrix.

The Model-View matrix

This matrix allows us to perform *affine transformations* in our scene. **Affine** is a mathematical name to describe transformations that *do not change* the structure of the object that undergoes such transformations. In our 3D world scene, such transformations are rotation, scaling, reflection shearing, and translation. Luckily for us, we do not need to understand how to represent such transformations with matrices. We just have to use one of the many JavaScript matrix libraries that are available online (such as `glMatrix`).

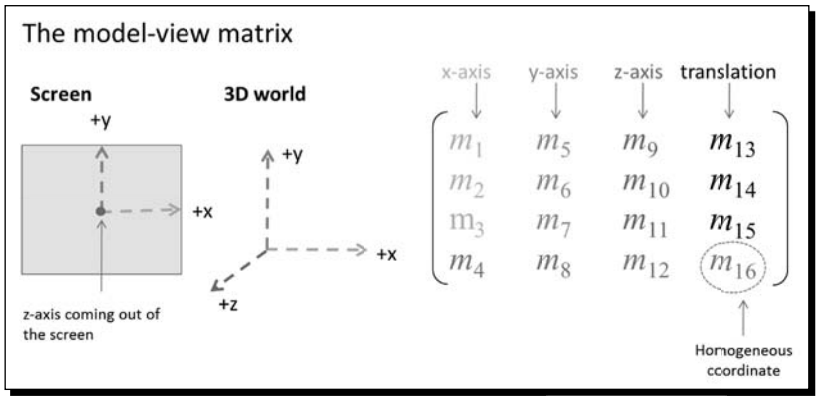
You can find more information on how transformation matrices work in any linear algebra book. Look for *affine transforms in computer graphics*.

Understanding the structure of the Model-View matrix is of no value if you just want to apply transformations to the scene or to objects in the scene. For that effect, you just use a library such as `glMatrix` to do the transformations on your behalf. However, the structure of this matrix could be invaluable information when you are trying to troubleshoot your 3D application.

Let's take a look.

Spatial encoding of the world

By default, when you render a scene, you are looking at it from the origin of the world in the negative direction of the z-axis. As shown in the following diagram, the z-axis is coming out of the screen (which means that you are looking at the negative z-axis).



From the center of the screen to the right, you will have the positive x-axis and from the center of the screen up, you will have the positive y-axis. This is the initial configuration and it is the reference for affine transformations.

In this configuration, the Model-View matrix is the **identity matrix** of rank four.

The first three rows of the Model-View matrix contain information about rotations and translations that are affecting the world.

Rotation matrix

The intersection of the first three rows with the first three columns defines the 3x3 Rotation matrix. This matrix contains information about rotations around the standard axis. In the initial configuration, this corresponds to:

$$[m_1, m_2, m_3] = [1, 0, 0] = \text{x-axis}$$

$$[m_5, m_6, m_7] = [0, 1, 0] = \text{y-axis}$$

$$[m_9, m_{10}, m_{11}] = [0, 0, 1] = \text{z-axis}$$

Translation vector

The intersection of the first three rows with the last column defines a three-component Translation vector. This vector indicates how much the origin, and for the same sake, the world, have been translated. In the initial configuration, this corresponds to:

$$\begin{bmatrix} m_{13} \\ m_{14} \\ m_{15} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$ = origin (no translation)

The mysterious fourth row

The fourth row does not bear any special meaning.

- Elements m_4, m_8, m_{12} are always zero.
- Element m_{16} (the homogeneous coordinate) will always be 1.

As we described at the beginning of this chapter, there are no cameras in WebGL. However, all the information that we need to operate a camera (mainly rotations and translations) can be extracted from the Model-View matrix itself!

The Camera matrix

Let's say, for a moment, that we do have a camera in WebGL. A camera should be able to rotate and translate to explore this 3D world. For example, think of a first person shooter game where you have to walk through levels killing zombies. As we saw in the previous section, a 4x4 matrix can encode rotations and translations. Therefore, our hypothetical camera could also be represented by one such matrix.

Assume that our camera is located at the origin of the world and that it is oriented in a way that it is looking towards the negative z-axis direction. This is a good starting point—we already know what transformation represents such a configuration in WebGL (identity matrix of rank 4).

For the sake of analysis, let's break the problem down into two sub-problems: camera translation and camera rotation. We will have a practical demo on each one.

Camera translation

Let's move the camera to [0 ,0, 4] in world coordinates. This means 4 units from the origin on the positive z-axis.

Remember that we do not know at this point of a matrix to move the camera, we only know how to move *the world* (with the Model-View matrix). If we applied:

```
mat4.translate(mvMatrix, [0,0,4]);
```

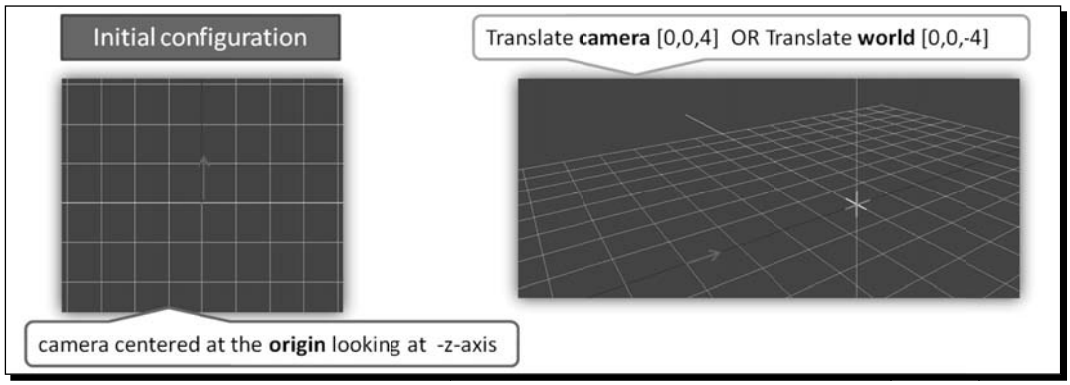
In such a case, the world would be translated 4 units on the positive z-axis and as the camera position has not been changed (as we do not know a matrix to do this), it would be located at [0,0,-4], which is exactly the opposite of what we wanted in the first place!

Now, say that we applied the translation in the opposite direction:

```
mat4.translate(mvMatrix, [0,0,-4]);
```

In such a case, the world would be moved 4 units on the negative z-axis and then the camera would be located at [0,0,4] in the new world coordinate system.

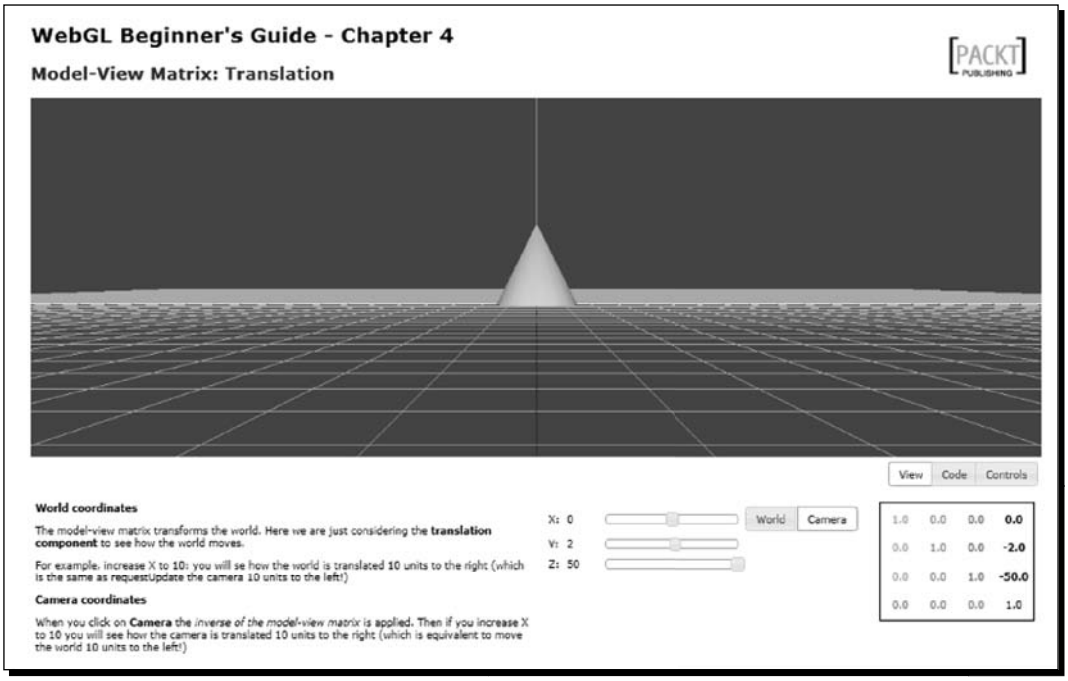
We can see here that translating the camera is *equivalent* to translating the world in the *opposite* direction.



In the following section, we are going to explore translations both in world space and in camera space.

Time for action – exploring translations: world space versus camera space

1. Open `ch4_ModelView_Translation.html` in your HTML5 browser:



2. We are looking from a distance at the positive z-axis at a cone located at the origin of the world. There are three sliders that will allow you to translate either the world or the camera on the x, y, and z axis, respectively. The world space is activated by default.

3. Can you tell by looking at the World-View matrix on the screen where the origin of the world is? Is it [0,0,0]? (Hint: check where we define translations in the Model-View matrix).

4. We can think of the canvas as the image that our camera sees. If the world center is at [0,-2,-50], where is the camera?

5. If we want to see the cone closer, we would have to move the center of the world towards the camera. We know that the camera is far on the positive z-axis of the world, so the translation will occur on the z-axis. Given that you are on world coordinates, do we need to increase or decrease the z-axis slider? Go ahead and try your answer.

6. Now switch to camera coordinates by clicking on the **Camera** button. What is the translation component of this matrix? What do you need to do if you want to move the camera closer to the cone? What does the final translation look like? What can you conclude?

7. Go ahead and try to move the camera on the x-axis and the y-axis. Check what the correspondent transformations would be on the Model-View matrix.

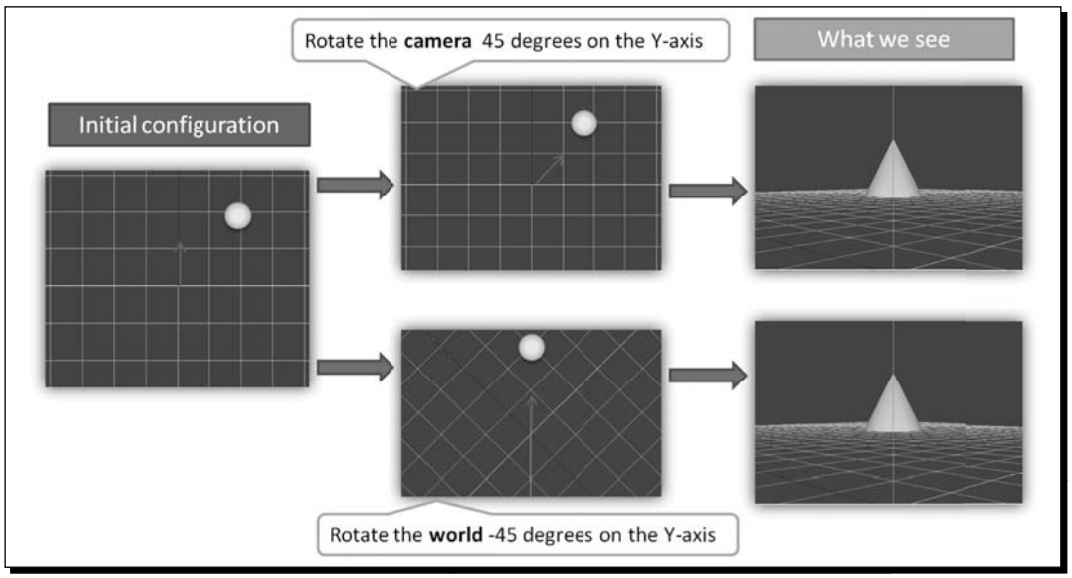
What just happened?

We saw that the camera translation is the inverse of the Model-View matrix translation. We also learned where to find translation information in a transformation matrix.

Camera rotation

Similarly, if we want to rotate the camera, say, 45 degrees to the right, this would be equivalent to rotating the world 45 degrees to the left. Using `glMatrix` to achieve this, we write the following:

```
mat4.rotate(mvMatrix,45 * Math.PI/180, [0,1,0]);
```

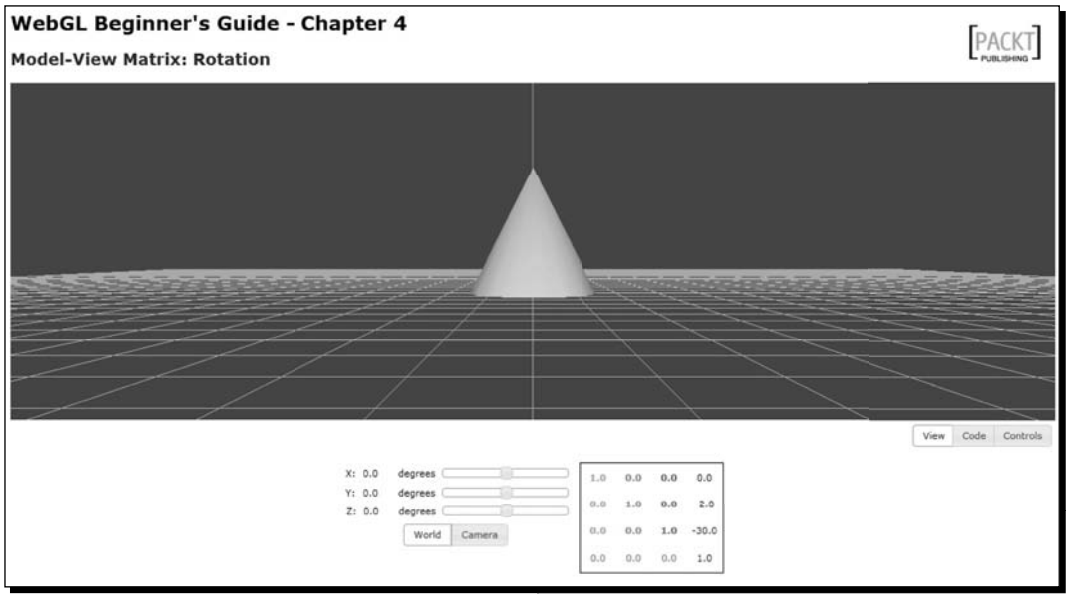


Let's see this behavior in action!

Similar to the previous section where we explored translations, in the following time for action, we are going to play with rotations in both world and camera spaces.

Time for action – exploring rotations: world space versus camera space

1. Open `ch4_ModelView_Rotation.html` in your HTML5 browser:



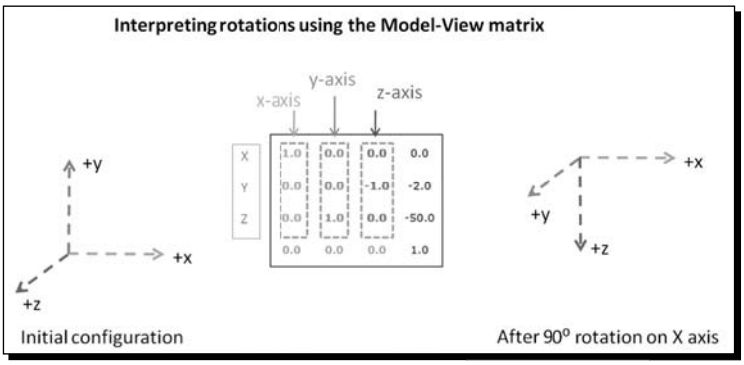
2. Just like in the previous example, we will see:
   - A cone at the origin of the world
   - The camera is located at [0,2,50] in world coordinates
   - Three sliders that will allows us to rotate either the world or the camera
   - Also, we have a matrix where we can see the result of different rotations

3. Let's see what happens to the axis after we apply a rotation. With the **World** coordinates button selected, rotate the world 90 degrees around the x-axis. What does the Model-View matrix look like?

4. Let's see where the axes end up after a 90 degree rotation around the x-axis:
   - By looking at the first column, we can see that the x-axis has not changed. It is still [1,0,0]. This makes sense as we are rotating around this axis.
   - The second column of the matrix indicates where the y-axis is after the rotation. In this case, we went from [0,1,0] , which is the original

configuration, to [0,0,1], which is the axis that is coming out of the screen. This is the z-axis in the initial configuration. This makes sense as now we are looking from above, down to the cone.

- The third column of the matrix indicates the new location of the z-axis. It changed from [0,0,1], which as we know is the z-axis in the standard spatial configuration (without transforms), to [0,-1,0], which is the negative portion of the y-axis in the original configuration. This makes sense as we rotated around the x-axis.



5. As we just saw, understanding the Rotation matrix (3x3 upper-left corner of the Model-View matrix) is simple: the first three columns are always telling us where the axis is.

6. Where are the axis in this transformation:

0.0	-1.0	0.0	0.0
1.0	0.0	0.0	-2.0
0.0	0.0	1.0	-50.0
0.0	0.0	0.0	1.0

Check your answer by using the sliders to achieve the rotation that you believe produce this matrix.

7. Now let's see how rotations work in **Camera** space. Click on the **Camera** button.

8. Start increasing the angle of rotation in the **X** axis by incrementing the slider position. What do you notice?

9. Go ahead and try different rotations in camera space using the sliders.

10. Are the rotations *commutative*? That is, do you get the same result if you rotate, for example, 5 degrees on the **X** axis and 90 degrees on the **Z** axis, compared to the case where you rotate 90 degrees on the **Z** axis and then you rotate 5 degrees on the **X** axis?

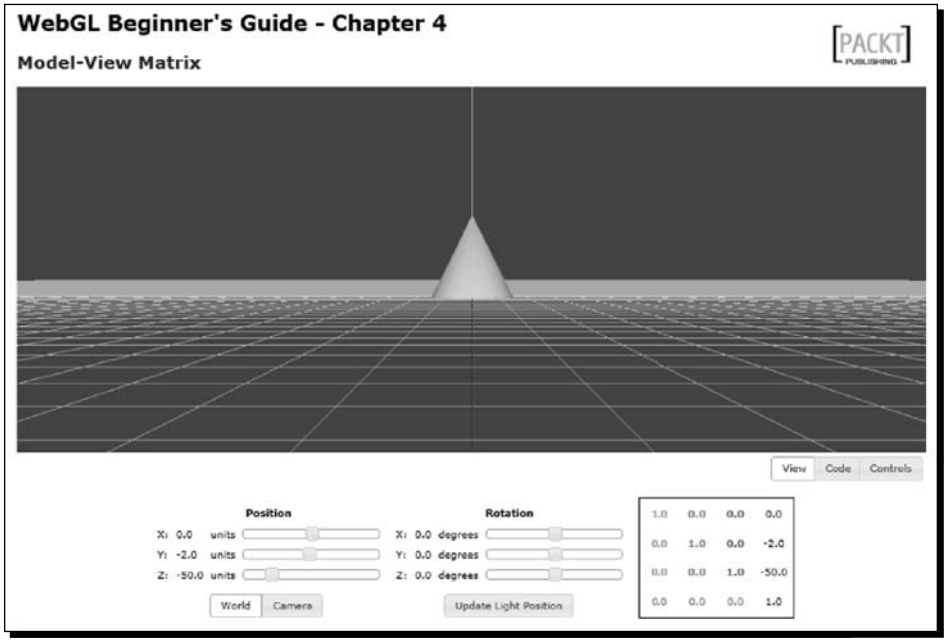
11. Now, go back to **World** space. Please check that when you are in **World** space, you need to reverse the rotations to obtain the same pose. So, if you were applying 5 degrees on the **X** axis and 90 degrees on the **Z** axis. Check that when you apply -5 degrees on the X axis and -90 degrees on the Z axis you obtain the same image as in point 10.

What just happened?

We just saw that the Camera matrix rotation is the inverse of the Model-View matrix rotation. We also learned how to identify the orientation of our world or camera upon analysis of the rotation matrix (3x3 upper-left corner of the correspondent transformation matrix).

Have a go hero – combining rotations and translations

1. The file `ch4_ModelView.html` contains the combination of rotations and translations. When you open it your HTML5 browser, you see something like the following:



2. Try different configurations of rotations and translations in both **World** and **Camera** spaces.

The Camera matrix is the inverse of the Model-View matrix

We can see through these two scenarios that a Camera matrix would require being the exact Model-View matrix opposite. In linear algebra, we know this as the **inverse** of a matrix.

The inverse of a matrix is such that when multiplying it by the original matrix, we obtain the identity matrix. In other words, if *M* is the Model-View matrix and *C* is the Camera matrix, we have the following:

$$MC = I$$

$$M^{-1}MC = M^{-1}$$

$$C = M^{-1}$$

We can create the Camera matrix using `glMatrix` by writing something like the following:

```
var cMatrix = mat4.create();
mat4.inverse(mvMatrix,cMatrix);
```

Thinking about matrix multiplications in WebGL

Please do not skip this section. If you want to, just put a sticker on this page so you remember where to go when you need to debug Model-View transformations. I spent so many nights trying to understand this (sigh) and I wish I had had a book like this to explain this to me.

Before moving forward, we need to know that in WebGL, the matrix operations are written in the *reverse order* in which they are applied to the vertices.

Here is the explanation. Assume, for a moment, that you are writing the code to rotate/move the world, that is, you rotate your vertices around the origin and then you move away. The final transformation would look like this:

$$RTv$$

Here, *R* is the 4x4 matrix encoding pure rotation, *T* is the 4x4 matrix encoding pure translation, and *v* corresponds to the vertices present in your scene (in homogeneous coordinates).

Now, if you notice, the first transformation that we actually apply to the vertices is the translation and then we apply the rotation! Vertices need to be multiplied first by the matrix that is to the left. In this scenario, that matrix is *T*. Then, the result needs to be multiplied by *R*.

This fact is reflected in the order of the operations (here `mvMatrix` is the Model-View matrix):

```
mat4.identity(mvMatrix)
mat4.translate(mvMatrix, position); mat4.rotateX(mvMatrix,rotation[0]
*Math.PI/180);
mat4.rotateY(mvMatrix,rotation[1]*Math.PI/180);
mat4.rotateZ(mvMatrix,rotation[2]*Math.PI/180);
```

Now if we were working in camera coordinates and we wanted to apply the same transformation as before, we need to apply a bit of linear algebra first:

$M = RT$	The Model-View matrix *M* is the result of multiplying rotation and translation together
$C = M^{-1}$	We know that the Camera matrix is the inverse of the Model-View matrix
$C = (RT)^{-1}$	By substitution
$C = T^{-1}R^{-1}$	Inverse of a matrix product is the reverse product of the inverses

Luckily for us, when we are working in camera coordinates in the chapter's examples, we have the inverse translation and the inverse rotation already calculated in the global variables `position` and `rotation`. Therefore, we would write something like this in the code (here `cMatrix` is the Camera matrix):

```
mat4.identity(cMatrix);
mat4.rotateX(cMatrix,rotation[0]*Math.PI/180);
mat4.rotateY(cMatrix,rotation[1]*Math.PI/180);
mat4.rotateZ(cMatrix,rotation[2]*Math.PI/180);
mat4.translate(cMatrix,position);
```

Basic camera types

The following are the camera types that we will discuss in this chapter.

- Orbiting camera
- Tracking camera

Orbiting camera

Up to this point, we have seen how we can generate rotations and translations of the world in the world or camera coordinates. However, in both cases, we are always generating the rotations around the center of the world. This could be ideal for many cases where we are orbiting around a 3D object such as our Nissan GTX model. You put the object at the center of the world, then you can examine the object at different angles (rotation) and then you move away (translation) to see the result. Let's call this type of camera an **orbiting camera**.

Tracking camera

Now, going back to the example of the first person shooting game, we need to have a camera that is able to look up when we want to see if there are enemies above us. Just the same, we should be able to look around left and right (rotations) and then move in the direction in which our camera is pointing (translation). This camera type can be designated as a **first-person** camera. This same type is used when the game follows the main character. Therefore, it is also known as a **tracking camera**.

To implement first-person cameras, we need to set up the rotations on the camera axis instead of using the world origin.

Rotating the camera around its location

When we multiply matrices, the order in which matrices are multiplied is relevant. Say, for instance, that we have two 4x4 matrices. Let *R* be the first matrix and let's assume that this matrix encodes pure rotation; let *T* be the second matrix and let's assume that *T* encodes pure translation. Now:

$$RT \neq TR$$

In other words, the order of the operations affects the result. It is not the same to rotate around the origin and then translate away from it (orbiting camera), as compared to translating the origin and then rotating around it (tracking camera)!

So in order to set the location of the camera as the center for rotations, we just need to invert the order in which the operations are called. This is equivalent to converting from an orbiting camera to a tracking camera.

Translating the camera in the line of sight

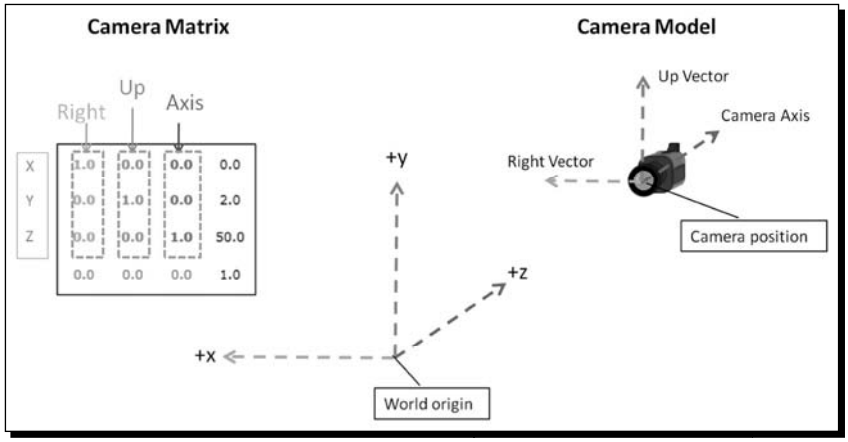
When we have an orbiting camera, the camera will be always looking towards the center of the world. Therefore, we will always use the z-axis to move to and from the object that we are examining. However, when we have a tracking camera, as the rotation occurs at the camera location, we can end up looking to any position in the world (which is ideal if you want to move around it and explore it). Then, we need to know the direction in which the camera is pointing to in world coordinates (camera axis). We will see how to obtain this next.

Camera model

Just like its counterpart, the Model-View matrix, the Camera matrix encodes information about the camera axes orientation. As we can see in the figure, the upper-left 3x3 matrix corresponds to the camera axes:

- The first column corresponds to the x-axis of the camera. We will call it the **Right vector.**
- The second column is the y-axis of the camera. This will be the **Up vector.**
- The third column determines the vector in which the camera can move back and forth. This is the z-axis of the camera and we will call it the **Camera axis**.

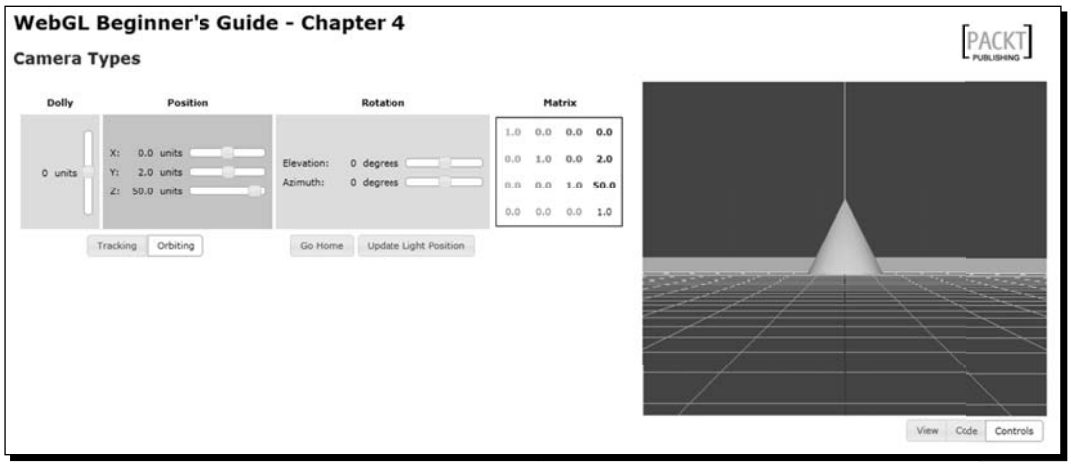
Due to the fact that the Camera matrix is the inverse of the Model-View matrix, the upper-left 3x3 rotation matrix contained in the Camera matrix gives us the orientation of the camera axes in world space. This is a plus, because it means that we can tell the orientation of our camera in world space, just by looking at the columns of this 3x3 rotation matrix (And we know now what each column means).



In the following section, we will play with orbiting and tracking cameras and we will see how we can change the camera position using mouse gestures, page widgets (sliders), and also we will have a graphical representation of the resulting Model-View matrix. In this exercise, we will integrate both rotations and translations and we will see how they behave under the two basic types of cameras that we are studying.

Time for action – exploring the Nissan GTX

1. Open the file `ch4_CameraTypes.html` in your HTML5 browser. You will see something like the following:



2. Go around the world using the sliders in **Tracking** mode. Cool eh?

3. Now, change the camera type to **Orbiting** mode and do the same.

4. Now, please check that besides the slider controls, both in **Tracking** and **Orbiting** mode, you can use your mouse and keyboard to move around the world.

5. In this exercise, we have implemented a camera using two new classes:

   - `Camera`: to manipulate the camera.
   - `CameraInteractor`: to connect the camera to the canvas. It will receive mouse and keyboard events and it will pass them along to the camera.

   If you are curious, you can see the source code of these two classes in `/js/webgl`. We have applied the concepts explained in this chapter to build these two classes.

6. So far, we have seen a cone in the center of the world. Let's change that for something more interesting to explore.

7. Open the file `ch4_CameraTypes.html` in your source code editor.

8. Go to the `load` function. Let's add the car to the scene. Rewrite the contents of this function so it looks like the following:

```
function load(){
    Floor.build(2000,100);
    Axis.build(2000);
    Scene.addObject(Floor);
    Scene.addObject(Axis);
    Scene.loadObjectByParts('models/nissan_gts/pr','Nissan',178);
}
```

You will see that we have increased the size of the axis and the floor so we can see them. We do need to do this because the car is an object much larger than the original cone.

9. There are some steps that we need to take in order to be able to see the car correctly. First we need to make sure that we have a large enough view volume. Go to the `initTransforms` function and update this line:

```
mat4.perspective(30, c_width / c_height, 0.1, 1000.0, pMatrix);
```

With this:

```
mat4.perspective(30, c_width / c_height, 10, 5000.0, pMatrix);
```

10. Do the same in the `updateTransforms` function.

11. Now, let's change the type of our camera so when we load the page, we have an orbiting camera by default. In the `configure` function, change this line:

```
camera = new Camera(CAMERA_TRACKING_TYPE);
```

With:

```
camera = new Camera(CAMERA_ORBIT_TYPE);
```

12. Another thing we need to take into account is the location of the camera. For a large object like this car, we need to be far away from the center of the world. For that purpose, go to the `configure` function and change:

```
camera.goHome([0,2,50]);
```

Add:

```
camera.goHome([0,200,2000]);
```

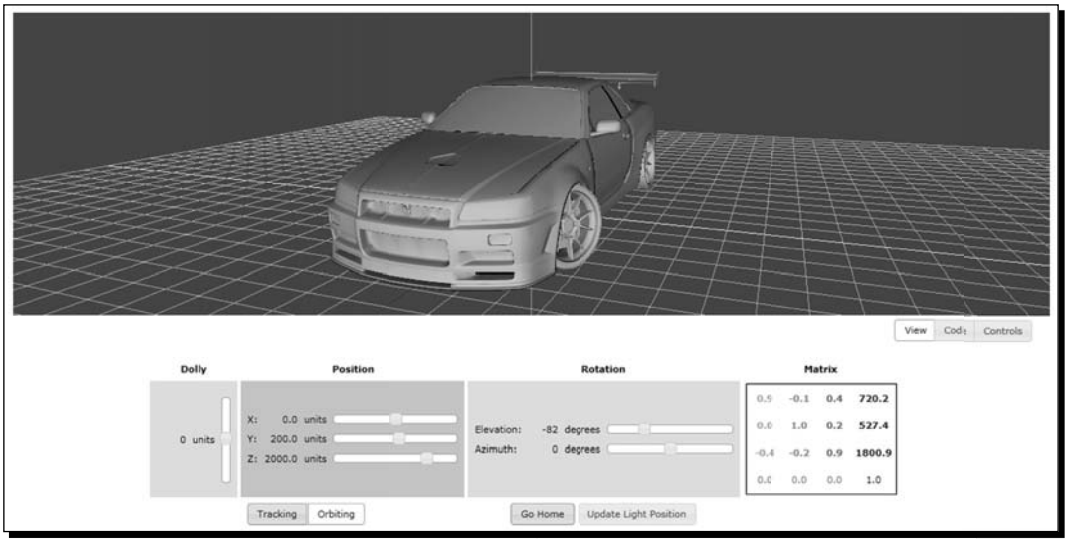
13. Let's modify the lighting of our scene so it fits better in the model we are displaying. In the function `configure` function, right after this line:

```
interactor = new CameraInteractor(camera, canvas);
```

Write:

```
gl.uniform4fv(prg.uLightAmbient,    [0.1,0.1,0.1,1.0]);
gl.uniform3fv(prg.uLightPosition,   [0, 0, 2120]);
gl.uniform4fv(prg.uLightDiffuse,    [0.7,0.7,0.7,1.0]);
```

14. Save the file with a different name and then load this new file in your HTML5 Internet browser. You should see something like the following screenshot:



15. Using the mouse, keyboard, or/and the sliders, explore the new scene. Hint: use orbiting mode to explore the car from different angles.

16. See how the Camera matrix is updated when you move around the scene.

17. You can see what the final exercise looks like by opening the file `ch4_NissanGTR.html`.

What just happened?

We added mouse and keyboard interaction to our scene. We also experimented with the two basic camera types—tracking and orbiting cameras. We modified the settings of our scene to visualize a complex model.

Have a go hero – updating light positions

Remember that when we move the camera, we are applying the inverse transformation to the world. If we do not update the light position, then the light source will be located at the same static point, *regardless* of the final transformation applied to the world.

This is very convenient when we are moving around or exploring an object in the scene. We will always be able to see if the light is located on the same axis of the camera. This is the case for the exercises in this chapter. Nevertheless, we can simulate the case when the camera movement is independent from the light source. To do so, we need to calculate the new light position whenever we move the camera. We do this in two steps:

First, we calculate the light direction. We can do this by simply calculating the difference vector between our target and our origin. Say that the light source is located at [0,2,50]. If we want to direct our light source towards the origin, we calculate the vector [0,0,0] - [0,2,50] (target - origin). This vector has the correct orientation of the light when we target the origin. We repeat the same procedure if we have a different target that needs to be lit. In that case, we just use the coordinates of the target and from them we subtract the location of the light.

As we are directing our light source towards the origin, we can find the direction of the light just by inverting the light position. If you notice, we do this in ESSL in the vertex shader:

```
vec3 L = normalize(-uLightPosition);
```

Now as `L` is a vector, if we want to update the direction of the light, then we need to use the Normal matrix, discussed earlier in this chapter, in order to update this vector under any world transformation. This step is optional in the vertex shader:

```
if(uUpdateLight){
  L = vec3(uNMatrix*vec4(L,0.0));
}
```

In the previous fragment of code, `L` is augmented to 4-components, so we can use the direct multiplication provided by ESSL. (Remember that `uNMatrix` is a 4x4 matrix and as such, the vectors that are transformed by it need to be 4-dimensional). Also, please bear in mind that, as explained in the beginning of the chapter, vectors have their homogeneous coordinate always set to zero, while vertices have their homogeneous coordinate set to one.

After the multiplication, we reduce the result to 3-components before assigning the result back to `L`.

You can test the effects of updating the light position by using the button **Update Light Position**, provided in the files `ch4_NissanGTR.html` and `ch4_CameraTypes.html`.

We connect a global variable that keeps track of the state of this button with the uniform `uUpdateLight`.

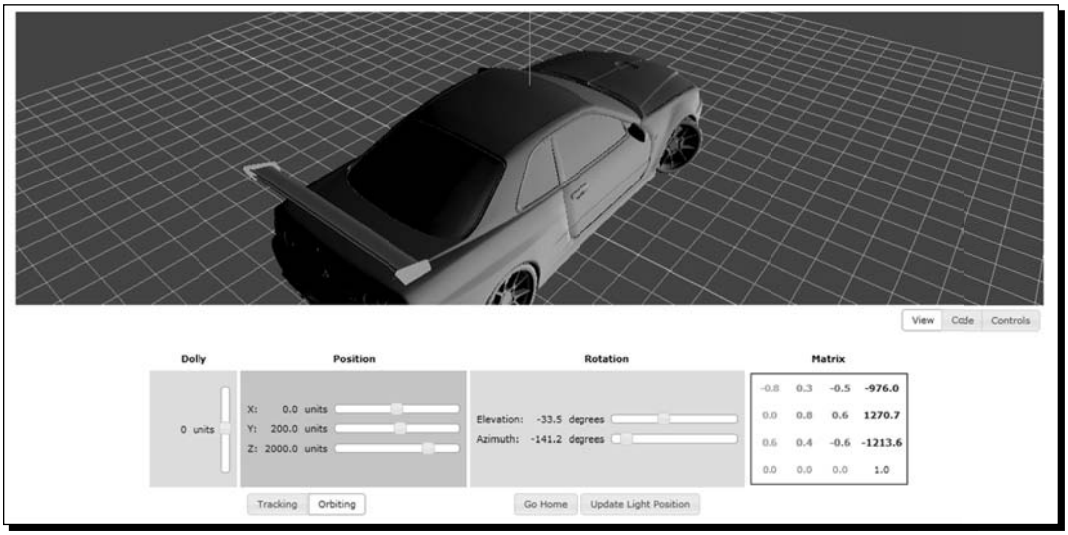
1. Edit `ch4_NissanGTR.html` and set the light position to a different location. To do this, edit the `configure` function. Go to:

   ```
   gl.uniform3fv(prg.uLightPosition,[0, 0, 2120]);
   ```

   Try different light positions:

   - `[2120,0,0]`
   - `[0,2120,0]`
   - `[100,100,100]`

2. For each option, save the file and try it with and without updating the light position (use the button **Update Light Position**).



3. For a better visualization, use an **Orbiting** camera.

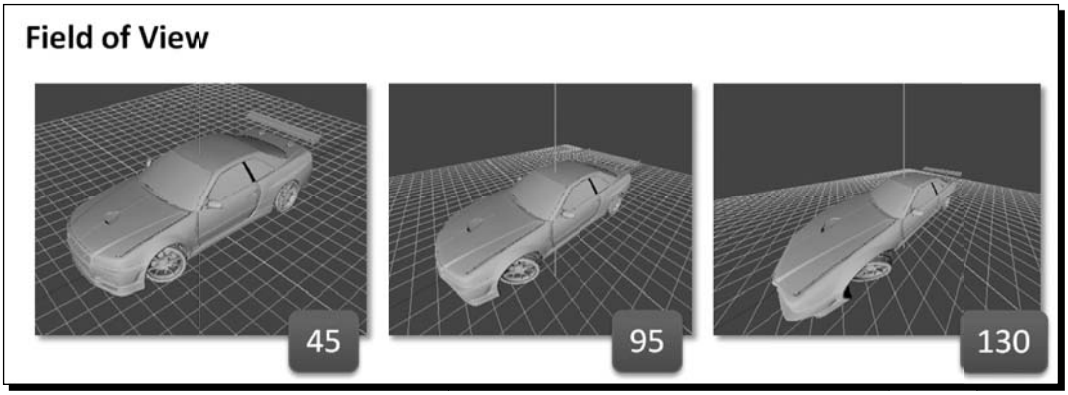
The Perspective matrix

At the beginning of the chapter, we said that the Perspective matrix combines the projection transformation and the perspective division. These two steps combined take a 3D scene and converts it into a cube that is then mapped to the 2D canvas by the viewport transformation.

In practice, the Perspective matrix determines the geometry of the image that is captured by the camera. In a real world camera, the lens of the camera would determine how distorted the final images are. In a WebGL world, we use the Perspective matrix to simulate that. Also, unlike in the real world where our images are always affected by perspective, in WebGL, we can pick a different representation: the orthographic projection.

Field of view

The Perspective matrix determines the **Field of View** (**FOV**) of the camera, that is, how much of the 3D space will be captured by the camera. The field of view is a measure given in degrees and the term is used interchangeably with the term **angle of view**.



Perspective or orthogonal projection

A perspective projection assigns more space to details that are closer to the camera than the details that are farther from it. In other words, the geometry that is close to the camera will appear bigger than the geometry that is farther from it. This is the way our eyes see the real world. Perspective projection allows us to assess the distance because it gives our brain a *depth cue*.

In contrast, an orthogonal projection uses parallel lines; this means that will look the same size regardless of their distance to the camera. Therefore, the depth cue is lost when using orthogonal projection.

Using `glMatrix`, we can set up the perspective or the orthogonal projection by calling `mat4.persective` or `mat4.ortho` respectively. The signatures for these methods are:

Function	Description (Taken from the documentation of the library)
`mat4.perspective(fovy, aspect, near, far, dest)`	Generates a perspective projection matrix with the given bounds Parameters: `fovy` - vertical field of view `aspect` - aspect ratio—typically viewport width/height `near`, `far` - near and far bounds of the frustum `dest` - Optional, `mat4` frustum matrix will be written into Returns: `dest` if specified, a new `mat4` otherwise
`mat4.ortho(left, right, bottom, top, near, far, dest)`	Generates an orthogonal projection matrix with the given bounds: Parameters: `left`, `right` - left and right bounds of the frustum `bottom`, `top` - bottom and top bounds of the frustum `near`, `far` - near and far bounds of the frustum `dest` - Optional, `mat4` frustum matrix will be written into Returns: `dest` if specified, a new `mat4` otherwise.

In the following *time for action* section, we will see how the field of view and the perspective projection affects the image that our camera captures. We will experiment perspective and orthographic projections for both orbiting and tracking cameras.

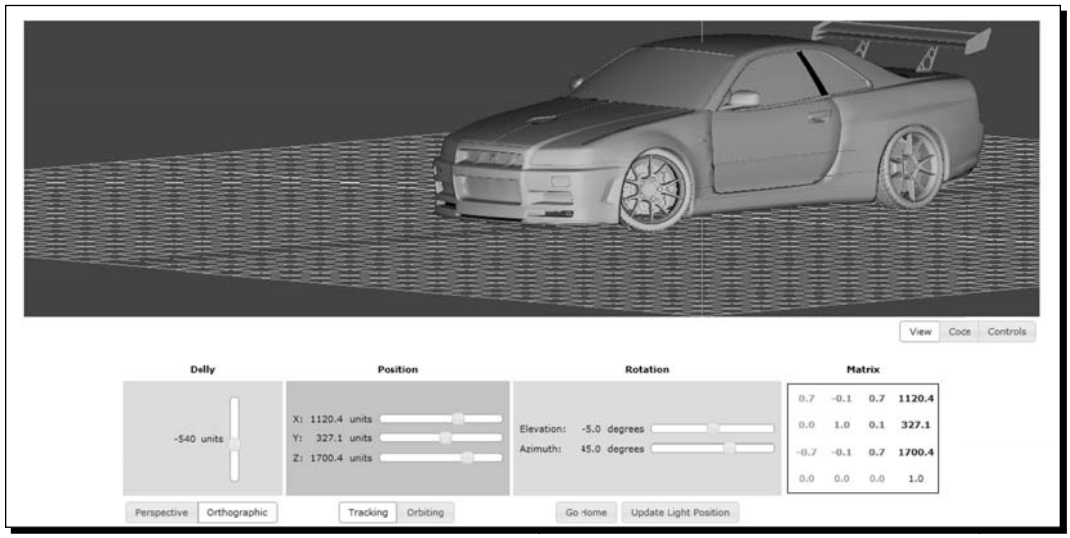
Time for action – orthographic and perspective projections

1. Open the file `ch4_ProjectiveModes.html` in your HTML5 Internet browser.
2. This exercise is very similar to the previous one. However, there are two new buttons: Perspective and Orthogonal. As you can see, Perspective is activated by default.

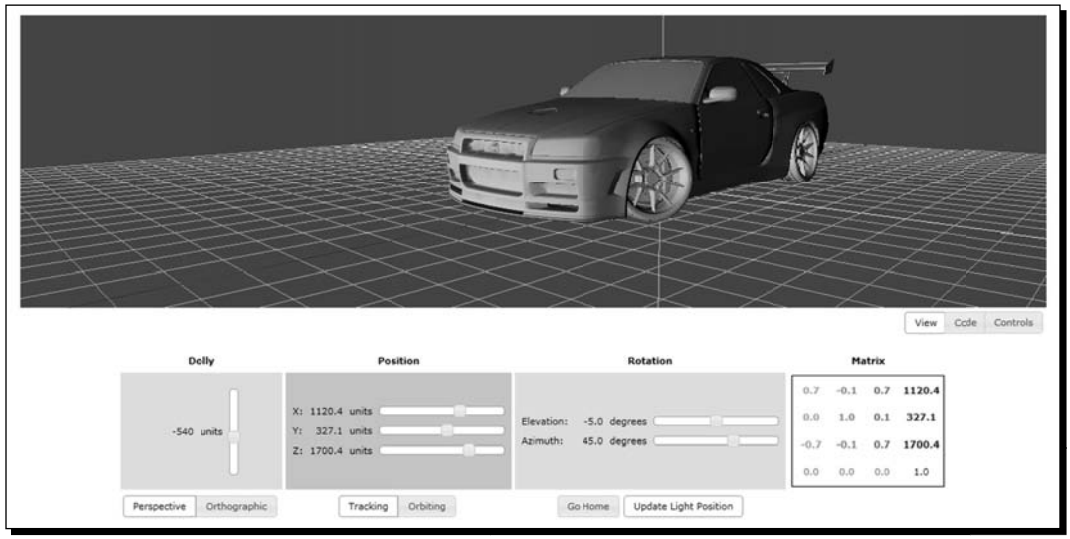
3. Change the camera type to **Orbiting**.

4. Change the projective mode to **Orthographic**.

5. Explore the scene. Notice the lack of depth cues that is characteristic of orthogonal projections:



6. Now switch to **Perspective** mode:



7. Explore the source code. Go to the `updateTransforms` function:

```
function updateTransforms(){
    if (projectionMode == PROJ_PERSPECTIVE){
        mat4.perspective(30, c_width / c_height, 10, 5000,
pMatrix);
    }
    else{
        mat4.ortho(-c_width, c_width, -c_height, c_height, -5000,
5000, pMatrix);
    }
}
```

8. Please take a look at the parameters that we are using to set up the projective view.

9. Let's modify the field of view. Create a global variable right before the `updateTransforms` function:

```
var fovy = 30;
```

10. Let's use this variable instead of the hardcoded value:

Replace:

```
mat4.perspective(30, c_width / c_height, 10, 5000, pMatrix);
```

With:

```
mat4.perspective(fovy, c_width / c_height, 10, 5000, pMatrix);
```

11. Now let's update the camera interactor to update this variable. Open the file `/js/webgl/CameraInteractor.js` in your source code editor.

Append these lines to `CameraInteractor.prototype.onKeyDown` inside `if (!this.ctrl){`:

```
else if (this.key == 87) {  //w
    if(fovy<120) fovy+=5;
    console.info('FovY:'+fovy);
}
else if (this.key == 78) { //n
    if(fovy>15) fovy-=5;
    console.info('FovY:'+fovy);
}
```

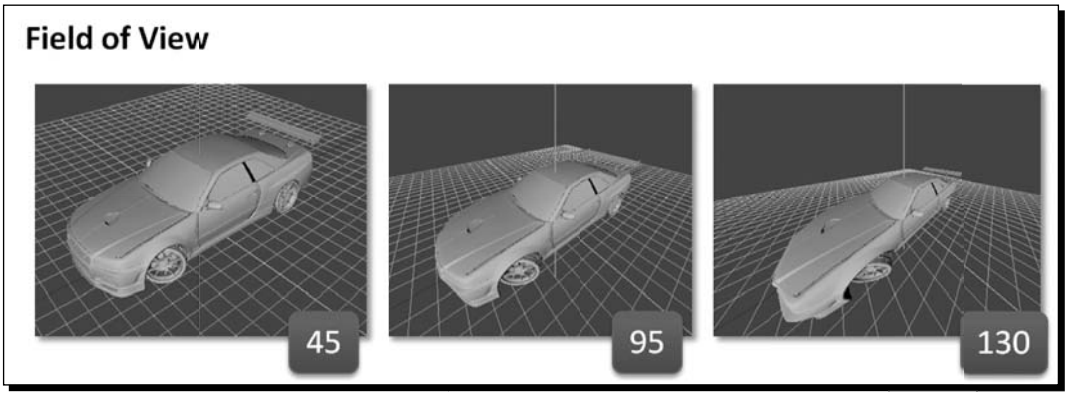
Please make sure that you are inside the `if` section.

If these instructions are already there, do not write them again. Just make sure you understand that the goal here is to update the global `fovy` variable that refers to the field of view in perspective mode.

12. Save the changes made to `CameraInteractor.js`.

13. Save the changes made to `ch4_ProjectiveModes.html`. Use a different name. You can see the final result in the file `ch4_ProjectiveModesFOVY.html`.

14. Open the renamed file in your HTML5 Internet browser. Try different fields of view by pressing `w` or `n` repeatedly. Can you replicate these scenes:



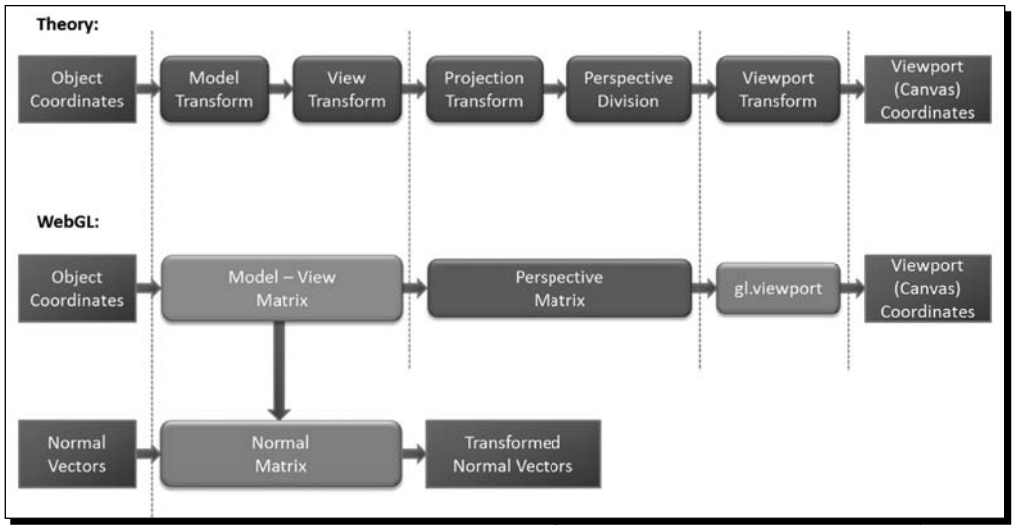
15. Notice that as you increase the field of view, your camera will capture more of the 3D space. Think of this as the lens of a real-world camera. With a wide-angle lens, you capture more space with the trade-off of deforming the objects as they move towards the boundaries of your viewing box.

What just happened?

We experimented with different configurations for the Perspective matrix and we saw how these configurations produce different results in the scene.

Have a go hero – integrating the Model-view and the projective transform

Remember that once we have applied the Model-View transformation to the vertices, the next step is to transform the view coordinates to NDC coordinates:



We do this by a simple multiplication using ESSL in the vertex shader:

```
gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition,1.0);
```

The predefined variable, `gl_Position`, stores the NDC coordinates for each vertex of every object defined in the scene.

In the previous multiplication, we augment the shader attribute, `aVertexPosition`, to a 4-component vertex because our matrices are 4x4. Unlike normals, vertices have a homogeneous coordinate equal to one (w=1).

After this step, WebGL will convert the computed clipping coordinates to normalized device coordinates and from there to canvas coordinates using the WebGL `viewport` function. We are going to see what happens when we change this mapping.

1. Open the file `ch4_NisanGTS.html` in your source code editor.
2. Go to the `draw` function. This is the rendering function that is invoked every time we interact with the scene (by using the mouse, the keyboard, or the widgets on the page).
3. Change this line:

   ```
   gl.viewport(0, 0, c_width, c_height);
   ```

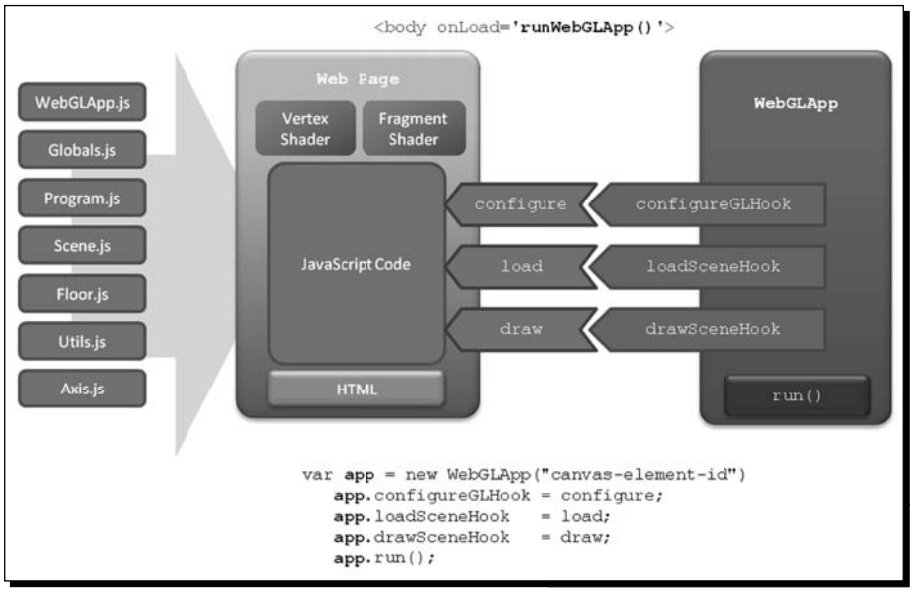
   Make it:

   ```
   gl.viewport(0, 0, c_width/2, c_height/2);
   gl.viewport(c_width/2,c_height/2, c_width, c_height);
   gl.viewport(50, 50, c_width-100, c_height-100);
   ```

4. For each option, save the file and open it on your HTML5 browser.
5. What do you see? Please notice that you can interact with the scene just like before.

Structure of the WebGL examples

We have improved the structure of the code examples in this chapter. As the complexity of our WebGL applications increases, it is wise to have a good, maintainable, and clear design. We have left this section at the end of the chapter so you can use it as a reference when working on the exercises.



Just like in previous exercises, our entry point is the `runWebGLApp` function which is called when the page is loaded. There we create an instance of `WebGLApp`, as shown in the previous diagram.

WebGLApp

This class encapsulates some of the utility functions that were present in our examples in previous chapters. It also declares a clear and simple life cycle for a WebGL application. `WebGLApp` has three function hooks that we can map to functions in our web page. These hooks determine what functions will be called for each stage in the life cycle of the app. In the examples of this chapter, we have created the following mappings:

- `configureGLHook`: which points to the `configure` function in the web page
- `loadSceneHook`: which is mapped to the `load` function in the webpage
- `drawSceneHook`: which corresponds to the `draw` function in the webpage

A function hook can be described as a pointer to a function. In JavaScript, you can write:

```
function foo(){alert("function foo invoked");}
var hook = foo;
hook();
```

This fragment of code will execute `foo` when `hook()` is executed. This allows a pluggable behavior that is more difficult to express in fully typed languages.

`WebGLApp` will use the function hooks to call `configure`, `load`, and `draw` in our page in that order.

After setting these hooks, the `run` method is invoked.

The source code for `WebGLApp` and other supporting objects can be found in `/js/webgl`

Supporting objects

We have created the following objects, each one in its own file:

- `Globals.js`: Contains the global variables used in the example.
- `Program.js`: Creates the program using the shader definitions. Provides the mapping between JavaScript variables (`prg.*`) and program attributes and uniforms.
- `Scene.js`: Maintains a list of objects to be rendered. Contains the AJAX/JSON functionality to retrieve remote objects. It also allows adding local objects to the scene.
- `Floor.js`: Defines a grid on the X-Z plane. This object is added to the `Scene` to have a reference of where the floor is.
- `Axis.js`: Represents the axis in world space. When added to the scene, we will have a reference of where the origin is.

- `WebGLApp.js`: Represents a WebGL application. It has three function hooks that define the configuration stage, the scene loading stage, and the rendering stage. These hooks can be connected to functions in our web page.
- `Utils.js`: Utility functions such as obtaining a `gl` context.

You can refer to `Globals.js` to find the global variables used in this example (the definition of the JavaScript matrices is there) and `Program.js` to find the `prg.*` JavaScript variables that map to attributes and uniforms in the shaders.

Life-cycle functions

The following are the functions that define the life-cycle of a `WebGLApp` application:

Configure

The `configure` function sets some parameters of our `gl` context, such as the color for clearing the canvas, and then it calls the `initTransforms` function.

Load

The `load` function sets up the objects `Floor` and `Axis`. These two locally-created objects are added to the `Scene` by calling the `addObject` method. After that, a remote object (AJAX call) is loaded using the `Scene.loadObject` method.

Draw

The `draw` function calls `updateTransforms` to calculate the matrices for the new position (that is, when we move), then iterates over the objects in the `Scene` to render them. Inside this loop, it calls `setMatrixUniforms` for every object to be rendered.

Matrix handling functions

The following are the functions that initialize, update, and pass matrices to the shaders:

initTransforms

As you can see, the Model-View matrix, the Camera matrix, the Perspective matrix, and the Normal matrix are set up here:

```
function initTransforms(){

    mat4.identity(mvMatrix);
    mat4.translate(mvMatrix, home);
```

```
        displayMatrix(mvMatrix);

        mat4.identity(cMatrix);
        mat4.inverse(mvMatrix,cMatrix);

        mat4.identity(pMatrix);
        mat4.perspective(30, c_width / c_height, 0.1, 1000.0, pMatrix);

        mat4.identity(nMatrix);
        mat4.set(mvMatrix, nMatrix);
        mat4.inverse(nMatrix);
        mat4.transpose(nMatrix);

        coords = COORDS_WORLD;
    }
```

updateTransforms

In `updateTransforms`, we use the contents of the global variables `position` and `rotation` to update the matrices. This is, of course, if the `requestUpdate` variable is set to `true`. We set `requestUpdate` to `true` from the GUI controls. The code for these is located at the bottom of the webpage (for instance, check the file `ch4_ModelView_Rotation.html`).

```
    function updateTransforms(){

        mat4.perspective(30, c_width / c_height, 0.1, 1000.0, pMatrix);
        if (coords == COORDS_WORLD){
                mat4.identity(mvMatrix);
                mat4.translate(mvMatrix, position);
                mat4.rotateX(mvMatrix,rotation[0]*Math.PI/180);
                mat4.rotateY(mvMatrix,rotation[1]*Math.PI/180);
                mat4.rotateZ(mvMatrix,rotation[2]*Math.PI/180);
         }
         else{
                mat4.identity(cMatrix);
                mat4.rotateX(cMatrix,rotation[0]*Math.PI/180);
                mat4.rotateY(cMatrix,rotation[1]*Math.PI/180);
                mat4.rotateZ(cMatrix,rotation[2]*Math.PI/180);
                mat4.translate(cMatrix,position);
         }
    }
```

setMatrixUniforms

This function performs the mapping:

```
function setMatrixUniforms(){

    if (coords == COORDS_WORLD){
         mat4.inverse(mvMatrix, cMatrix);
         displayMatrix(mvMatrix);
         gl.uniformMatrix4fv(prg.uMVMatrix, false, mvMatrix);
    }
    else{
         mat4.inverse(cMatrix, mvMatrix);
         displayMatrix(cMatrix);

     }

     gl.uniformMatrix4fv(prg.uPMatrix, false, pMatrix);
     gl.uniformMatrix4fv(prg.uMVMatrix, false, mvMatrix);
     mat4.transpose(cMatrix, nMatrix);
     gl.uniformMatrix4fv(prg.uNMatrix, false, nMatrix);
}
```

Summary

Let's summarize what we have learned in this chapter:

There is no camera object in WebGL. However, we can build one using the Model-View matrix.

3D objects undergo several transformations to be displayed on a 2D screen. These transformations are represented as 4x4 matrices.

Scene transformations are affine. Affine transformations are constituted by a linear transformation followed by a translation. WebGL groups affine transforms in three matrices: the Model-View matrix, the Perspective matrix, and the Normal matrix and one WebGL operation: `gl.viewport()`.

Affine transforms are applied in projective space so they can be represented by 4x4 matrices. To work in projective space, vertices need to be augmented to contain an extra term, namely, w, which is called the perspective coordinate. The 4-tuple (*x*,*y*,*z*,*w*) is called homogeneous coordinates. Homogeneous coordinates allows representation of lines that intersect on infinity by making the perspective coordinate $w = 0$. Vectors always have a homogeneous coordinate $w = 0$; While points have a homogenous coordinate, namely, $w = 1$ (unless they are at infinity, in which case $w=0$).

By default, a WebGL scene is viewed from the world origin in the negative direction of the z-axis. This can be altered by changing the Model-View matrix.

The Camera matrix is the inverse of the Model-View matrix. Camera and World operations are opposite. There are two basic types of camera—orbiting and tracking camera.

Normals receive special treatment whenever the object suffers an affine transform. Normals are transformed by the Normal matrix, which can be obtained from the Model-View matrix.

The Perspective matrix allows the determining of two basic projective modes, namely, orthographic projection and perspective projection.

Where to buy this book

You can buy WebGL Beginner's Guide from the Packt Publishing website: `http://www.packtpub.com/webgl-javascript-beginners-guide/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/webgl-javascript-beginners-guide/book