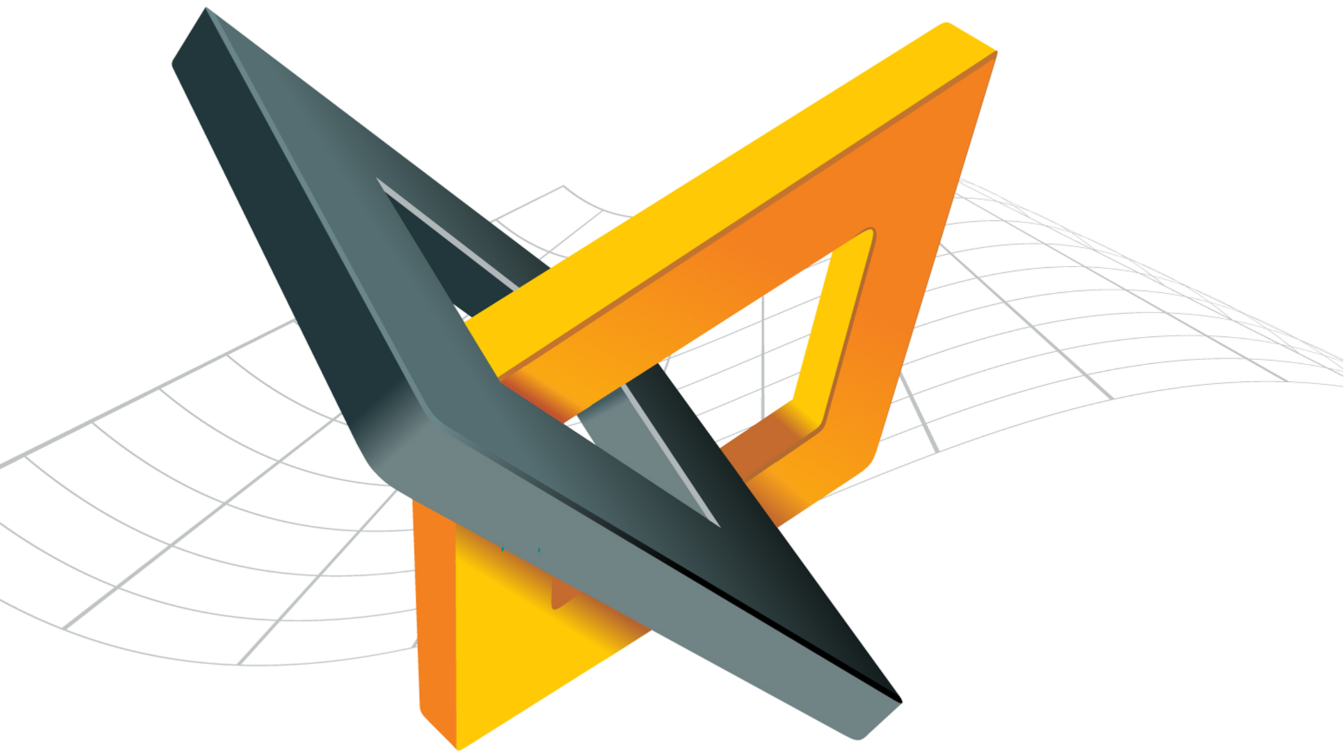
WARREN MOORE



METAL
BY EXAMPLE

High-performance graphics and
data-parallel programming for iOS

Contents

Chapter 4

Drawing in 3D

Building on what we learned about the rendering pipeline in the previous chapter, we will now begin our coverage of rendering in three dimensions.

Specifying Geometry in 3D

Cube Geometry

The object we will render in this chapter is a simple cube. It is easy to write the vertices of a cube in code, avoiding the complexity of loading a 3D model for now. Here are the vertices for the cube mesh:

```
const MBEVertex vertices[] =
{
    { .position = { -1,  1,  1, 1 }, .color = { 0, 1, 1, 1 } },
    { .position = { -1, -1,  1, 1 }, .color = { 0, 0, 1, 1 } },
    { .position = {  1, -1,  1, 1 }, .color = { 1, 0, 1, 1 } },
    { .position = {  1,  1,  1, 1 }, .color = { 1, 1, 1, 1 } },
    { .position = { -1,  1, -1, 1 }, .color = { 0, 1, 0, 1 } },
    { .position = { -1, -1, -1, 1 }, .color = { 0, 0, 0, 1 } },
    { .position = {  1, -1, -1, 1 }, .color = { 1, 0, 0, 1 } },
    { .position = {  1,  1, -1, 1 }, .color = { 1, 1, 0, 1 } }
};
```

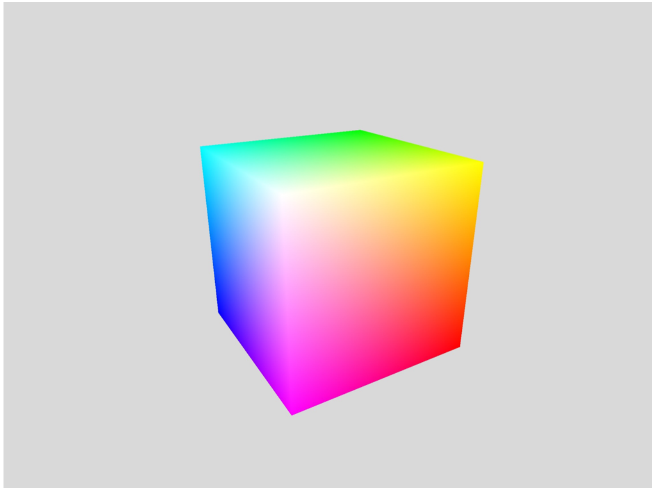


Figure 4.1: The cube rendered by the sample app

We reuse the same `MBEVertex` struct from the previous chapter, which has a position and color for each vertex. Since we have not introduced lighting yet, giving each vertex a distinct color provides an important depth cue. As before, we create a buffer to hold the vertices:

```
vertexBuffer = [device newBufferWithBytes:vertices
    length:sizeof(vertices)
    options:MTLResourceOptionCPUCacheModeDefault];
```

Index Buffers

In the previous chapter, we stored the vertices of our triangle in the order they were to be drawn, and each vertex was used only once. In the case of a cube, each vertex belongs to several triangles. Ideally, we would reuse those vertices instead of storing additional copies of each vertex in memory. As models grow in size, vertex reuse becomes even more important.

Fortunately, like most graphics libraries, Metal gives us the ability to provide an *index buffer* along with our vertex buffer. An index buffer contains a list of indices into the vertex buffer that specifies which vertices make up each triangle.

First, we define a couple of `typedef`s that will simplify working with indices:

```
typedef uint16_t MBEIndex;
const MTLIndexType MBEIndexType = MTLIndexTypeUInt16;
```

Starting out, we will use 16-bit unsigned indices. This allows each mesh to contain up to 65536 distinct vertices, which will serve our purposes for quite a while. If we need to accommodate more vertices in the future, we can change these definitions, and our code will adapt to the larger index size. Metal allows 16- and 32-bit indices.

Each square face of the cube is broken up into two triangles, comprising six indices. We specify them in an array, then copy them into a buffer:

```
const MBEIndex indices[] =
{
    3, 2, 6, 6, 7, 3,
    4, 5, 1, 1, 0, 4,
    4, 0, 3, 3, 7, 4,
    1, 5, 6, 6, 2, 1,
    0, 1, 2, 2, 3, 0,
    7, 6, 5, 5, 4, 7
};


indexBuffer = [device newBufferWithBytes:indices
    length:sizeof(indices)
    options:MTLResourceOptionCPUCacheModeDefault];
```

Now that we have defined some geometry to work with, let's talk about how to render a 3D scene.

Dividing Work between the View and the Renderer

In the previous chapter, we gave the `MBEMetalView` class the responsibility of rendering the triangle. Now, we would like to move to a more sustainable model, by fixing the functionality of the view, and offloading the job of resource management and rendering to a separate class: the renderer.

Responsibilities of the View Class

The view class should only be concerned with getting pixels onto the screen, so we remove the command queue, render pipeline, and buffer properties from it. It retains the responsibility of listening to the display link and managing the texture(s) that will be attachments of the render pass.

The new `MBEMetalView` provides properties named `currentDrawable`, which vends the `CAMetalDrawable` object for the current frame, and `currentRenderPassDescriptor`, which vends a render pass descriptor configured with the drawable's texture as its primary color attachment.

The Draw Protocol

In order to do drawing, we need a way for the view to communicate with us that it's time to perform our draw calls. We decouple the notion of a view from the notion of a renderer through a protocol named `MBEMetalViewDelegate` which has a single required method: `-drawInView:`.

This draw method will be invoked once per display cycle to allow us to refresh the contents of the view. Within the delegate's implementation of the method, the `currentDrawable` and `currentRenderPassDescriptor` properties can be used to create a render command encoder (which we will frequently call a *render pass*) and issue draw calls against it.

Responsibilities of the Renderer Class

Our renderer will hold the long-lived objects that we use to render with Metal, including things like our pipeline state and buffers. It conforms to the `MBEMetalViewDelegate` protocol and thus responds to the `-drawInView:` message by creating a command buffer and command encoder for issuing draw calls. Before we get to that, though, we need to talk about the work the draw calls will be doing.

Transforming from 3D to 2D

In order to draw 3D geometry to a 2D screen, the points must undergo a series of transformations: from object space, to world space, to eye space, to clip space, to normalized device coordinates, and finally to screen space.

From Object Space to World Space

The vertices that comprise a 3D model are expressed in terms of a local coordinate space (called *object space*). The vertices of our cube are specified about the origin, which lies at the cube's center. In order to orient and position objects in a larger scene, we need to specify a transformation that scales, translates, and rotates them into *world space*.

You may recall from linear algebra that matrices can be multiplied together (*concatenated*) to build up a single matrix that represents a sequence of linear transformations. We will call the matrix that gathers together the sequence of transformations that move an object into world space the *model matrix*. The model matrix of our cube consists of a scale transformation followed by two rotations. Each of these individual transformations varies with time, to achieve the effect of a pulsing, spinning cube.

Here is the code for creating the sequence of transformations and multiplying them together to create the world transformation:

```
float scaleFactor = sinf(5 * self.time) * 0.25 + 1;
vector_float3 xAxis = { 1, 0, 0 };
vector_float3 yAxis = { 0, 1, 0 };
matrix_float4x4 xRot = matrix_float4x4_rotation(xAxis, self.rotationX);
matrix_float4x4 yRot = matrix_float4x4_rotation(yAxis, self.rotationY);
matrix_float4x4 scale = matrix_float4x4_uniform_scale(scaleFactor);
matrix_float4x4 modelMatrix = matrix_multiply(matrix_multiply(xRot, yRot),
    scale);
```

We use the convention that matrices are applied from right-to-left to column vectors, so `modelMatrix` scales a vertex, then rotates it about the Y axis, then rotates it about the X axis. The `rotationX`, `rotationY` and `time` properties are updated each frame so that this transformation is animated.

From World Space to View Space

Now that we have our scene (the scaled, rotated cube) in world space, we need to position the entire scene relative to the eye point of our virtual camera. This transformation is called the *view space* (or, equivalently, the *eye space* or *camera space*) transformation. Everything that will eventually be visible on screen is contained in a pyramidal shape called the *view frustum*, illustrated below. The position of the virtual camera's eye is the apex of this viewing volume, the point behind the middle of the near plane of the viewing frustum.

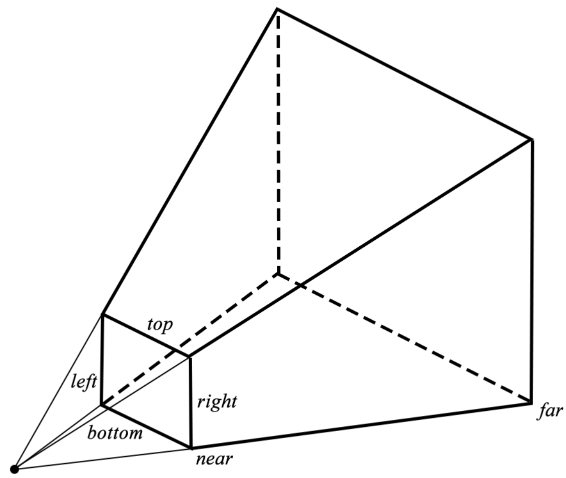


Figure 4.2: The view frustum. The apex of the frustum coincides with the eye of the virtual camera.

Constructing the transformation for the camera requires us to think backwards: moving the camera farther back is equivalent to moving the scene deeper into the screen, and rotating the scene counterclockwise about the Y axis is equivalent to the camera orbiting the scene clockwise.

In our sample scene, we want to position the camera a few units back from the cube. We use the convention that world space is "right-handed," with the Y axis pointing up, meaning that the Z axis points out of the screen. Therefore, the correct transformation is a translation that moves each vertex a *negative* distance along the Z axis. Equivalently, this transformation moves the camera a positive distance along the Z axis. It's all relative.

Here is the code for building our view matrix:

```
vector_float3 cameraTranslation = { 0, 0, -5 };
matrix_float4x4 viewMatrix = matrix_float4x4_translation(cameraTranslation);
```

From View Space to Clip Space

The projection matrix transforms view space coordinates into *clip space* coordinates.

Clip space is the 3D space that is used by the GPU to determine visibility of triangles within the viewing volume. If all three vertices of a triangle are outside the clip volume, the triangle is not rendered at all (it is *culled*). On the other hand, if one or more of the vertices is inside the volume, it is *clipped* to the bounds, and one or more modified triangles are used as the input to the vertex shader.

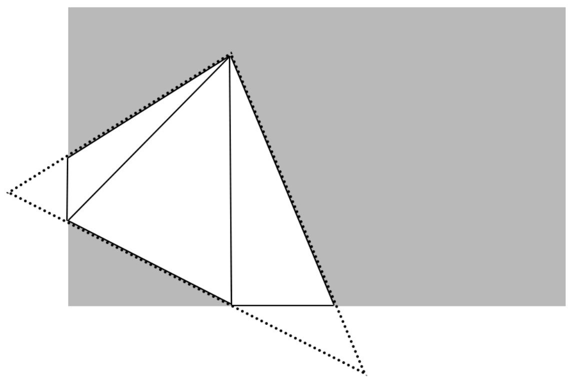


Figure 4.3: An illustration of a triangle being clipped. Two of its vertices are beyond the clipping bounds, so the face has been clipped and retriangulated, creating three new triangles.

The perspective projection matrix takes points from view space into clip space via a sequence of scaling operations. During the previous transformations, the w component remained unchanged and equal to 1, but the projection matrix affects the w component in such a way that if the absolute values of the x, y, or z component is greater than the absolute value of the w component, the vertex lies outside the viewing volume and is clipped.

The perspective projection transformation is encapsulated in the `matrix_float4x4_perspective` utility function. In the sample code, fix a vertical field of view of about 70 degrees, choose a far and near plane value, and select an aspect ratio that is equal to the ratio between the current drawable width and height of our Metal view.

```
float aspect = drawableSize.width / drawableSize.height;
float fov = (2 * M_PI) / 5;
float near = 1;
float far = 100;
matrix_float4x4 projectionMatrix = matrix_float4x4_perspective(aspect, fov,
    near, far);
```

We now have a sequence of matrices that will move us all the way from object space to clip space, which is the space that Metal expects the vertices returned by our vertex shader to be in. Multiplying all of these matrices together produces a *model-view-projection* (MVP) matrix, which is what we will actually pass to our vertex shader so that each vertex can be multiplied by it on the GPU.

The Perspective Divide: From Clip Space to NDC

In the case of perspective projection, w component is calculated so that the perspective divide produces foreshortening, the phenomenon of farther objects being scaled down more.

After we hand a projected vertex to Metal from our vertex function, it divides each component by the w component, moving from clip-space coordinates to *normalized device coordinates* (NDC), after the relevant clipping is done against the viewing volume bounds. Metal's NDC space is a cuboid $[-1, 1] \times [-1, 1] \times [0, 1]$, meaning that x and y coordinates range from -1 to 1, and z coordinates range from 0 to 1 as we move *away from the camera*.

The Viewport Transform: From NDC to Window Coordinates

In order to map from the half-cube of NDC onto the pixel coordinates of a view, Metal does one final internal transformation by scaling and biasing the normalized device coordinates such that they cover the size of the *viewport*. In all of our sample code, the viewport is a rectangle covering the entire view, but it is possible to resize the viewport such that it covers only a portion of the view.

3D Rendering in Metal

Uniforms

A *uniform* is a value that is passed as a parameter to a shader that does not change over the course of a draw call. From the point of view of a shader, it is a constant.

In the following chapters, we will bundle our uniforms together in a custom structure. Even though we only have one such value for now (the MVP matrix), we will establish the habit now:

```
typedef struct
{
    matrix_float4x4 modelViewProjectionMatrix;
} MBEUniforms;
```

Since we are animating our cube, we need to regenerate the uniforms every frame, so we put the code for generating the transformations and writing them into a buffer into a method on the renderer class named `-updateUniforms`.

The Vertex Shader

Now that we have some of the foundational math for 3D drawing in our toolbox, let's discuss how to actually get Metal to do the work of transforming vertices.

Our vertex shader takes a pointer to an array of vertices of type `Vertex`, a struct declared in the shader source. It also takes a pointer to a uniform struct of type `Uniforms`. The definition of these types are:

```
struct Vertex
{
    float4 position [[position]];
    float4 color;
};

struct Uniforms
{
    float4x4 modelViewProjectionMatrix;
};
```

The vertex shader itself is straightforward. In order to find the clip-space coordinates of the vertex, it multiplies the position by the MVP matrix from the uniforms and assigns the result to the output vertex. It also copies the incoming vertex color to the output vertex without modification.

```
vertex Vertex vertex_project(device Vertex *vertices [[buffer(0)]],
                             constant Uniforms *uniforms [[buffer(1)]],
                             uint vid [[vertex_id]])
{
    Vertex vertexOut;
    vertexOut.position = uniforms->modelViewProjectionMatrix *
         vertices[vid].position;
    vertexOut.color = vertices[vid].color;

    return vertexOut;
}
```

Notice that we are using two different address space qualifiers in the parameter list of the function: `device` and `constant`. In general, the `device` address space should be used when indexing into a buffer using per-vertex or per-fragment offset such as the parameter attributed with `vertex_id`. The `constant` address space is used when many invocations of the function will access the same portion of the buffer, as is the case when accessing the uniform structure for every vertex.

The Fragment Shader

The fragment shader is identical to the one used in the previous chapter:

```
fragment half4 fragment_flatcolor(Vertex vertexIn [[stage_in]])
{
    return half4(vertexIn.color);
}
```

Preparing the Render Pass and Command Encoder

Each frame, we need to configure the render pass and command encoder before issuing our draw calls:

```
    [commandEncoder setDepthStencilState:self.depthStencilState];
    [commandEncoder setFrontFacingWinding:MTLWindingCounterClockwise];
    [commandEncoder setCullMode:MTLCullModeBack];
```

The `depthStencilState` property is set to the previously-configured stencil-depth state object.

The front-face *winding order* determines whether Metal considers faces with their vertices in clockwise or counterclockwise order to be front-facing. By default, Metal considers clockwise faces to be front-facing. The sample data and sample code prefer counterclockwise, as this makes more sense in a right-handed coordinate system, so we override the default by setting the winding order here.

The *cull mode* determines whether front-facing or back-facing triangles (or neither) should be discarded (culled). This is an optimization that prevents triangles that cannot possibly be visible from being drawn.

Issuing the Draw Call

The renderer object sets the necessary buffer properties on the render pass' argument table, then calls the appropriate method to render the vertices. Since we do not need to issue additional draw calls, we can end encoding on this render pass.

```
[renderPass setVertexBuffer:self.vertexBuffer offset:0 atIndex:0];

NSUInteger uniformBufferOffset = sizeof(MBEUniforms) * self.bufferIndex;
[renderPass setVertexBuffer:self.uniformBuffer offset:uniformBufferOffset
     atIndex:1];

[renderPass drawIndexedPrimitives:MTLPrimitiveTypeTriangle
    indexCount:[self.indexBuffer length] / sizeof(MBEIndex)
    indexType:MBEIndexType
    indexBuffer:self.indexBuffer
    indexBufferOffset:0];

[renderPass endEncoding];
```

As we saw in the previous chapter, the first parameter tells Metal what type of primitive we want to draw, whether points, lines, or triangles. The rest of the parameters tell Metal the count, size, address, and offset of the index buffer to use for indexing into the previously-set vertex buffer.

The Sample Project

The sample code for this chapter is in the 04-DrawingIn3D directory. Bringing together everything we learned above, it renders a pulsing, rotating cube in glorious 3D.