Normal & Parallax Mapping Tutorial

Introduction

This tutorial assumes a basic understanding of OpenGL and GLSL, as well as the ability to load and initialise shaders and to draw geometry using **vertex buffer objects**, under the OpenGL core profile. You should also know how to perform lighting calculations in the fragment shader.

What You'll Learn

By the end of this tutorial, you will have learned:

- What model space and tangent space are.
- How to convert between model space and tangent space.
- What a normal map is, and how it works.
- How to use a normal map in lighting calculations for more detailed lighting.

Model Space

Since you have some experience writing shaders, you should have heard of the term "model space", or should at least have heard of the model view matrix. How exactly does the graphics card know how to draw vertex data? Well, you usually tell it how to do so in the form of a vertex shader, something like this:

```
vec4 modelVertex = u_modelViewMatrix * in_vertex;
gl_Position = u_projectionMatrix * modelVertex;
```

To get the vertex into a usable form - one which represents where it is relative to the camera at (0, 0, 0) - we need to convert it to model space. Model space means exactly what it sounds like - a programmer-defined model of 3D space. Once we have the vertex in this form, we can use it in calculations such as lighting which involve other vertices in the same form.

The model view matrix simply defines a model of 3D space, and by multiplying a matrix by a vertex, we're converting that vertex into the space defined by the matrix.

What Are We Doing?

There is another defintion of 3D space usually called tangent space. Before I get into what tangent space is, I should explain what exactly we'll be doing.

First, we're going to take a textured 2D surface and apply lighting calculations to it as if it's actually a 3D object. After that, we're going to actually make it look 3D based on what angle we're viewing it from. We can do this by displacing the texture further away, in the same direction as the eye vector.

Confusing, right? Let's focus on just the lighting for now.

Normal Maps

Lighting calculations require a normal to indicate which way the surface is facing. If we're using a single 2D surface, the entire surface is facing one way. So how can we possibly do lighting calculations as if it's a 3D object... if we only have one normal?

We'll use something called a normal map.

A normal map describes itself perfectly – it's a map of normals. In addition to the texture we're using, we'll use another texture the same size, but instead of mapping colour pixels, we'll be mapping the normal at each pixel.

I'll say that again. For each colour pixel, we have a normal associated with that point on the texture. This effectively lets us do lighting calculations on the texture as if each pixel is a separate surface.

When we used normals in the past in lighting calculations, we couldn't just use the normal vector. We had to do something like this in the vertex shader:

```
    ex_normal = normalize(modelViewMatrix3x3 * normalize(in_normal));
```

Look familiar? By definition, the normal needs to be normalized, which is why we use `normalize` so much, but other than that, it's easy to see that we're just converting it to model space using the 3x3 equivalent of the model view matrix.

Seems simple enough, so we just do this in the fragment shader, right?

```
    vec3 normal = texture(u_normalMap, ex_texCoord).xyz;
    vec3 finalNormal = normalize(modelViewMatrix3x3 * normalize(normal));
```

Not quite. When we previously converted the normal to model space, we were using the normal of the surface in model space coordinates. However, the normals at each pixel in our texture have no concept of position or orientation in model space. So if we can't represent our normals in model space, how do we do it?

Tangent Space

What if we represented the normals relative to the flat texture they're pointing out from? For example, where the texture is flat, the normal would be pointing directly out from the texture, and so it would be (0, 0, 1). In fact, the normals will very much be biased in the z direction.

By representing the normals in this way, we're actually representing them in a new space, relative to the 2D surface. So how do we go about defining this new space? A space is defined by three directions: Left, Up, and Forward. In fact, a better way to describe these is just x, y, and z. But what do we use? Actually, we already have our z direction – it's just the ***normal*** of the 2D surface.

Figuring out x and y is a bit more involved, but you should already have a general idea of what they'll be. If z is pointing out of the surface, then x and y will point perpendicular to each other and to z. That is to say, x and y will form a right angle on the surface itself.

Any vector which is perfectly parallel and aligned with a surface is called a tangent. You may have heard this term used to describe a line which goes past a circle, not intersecting it but touching it exactly once. This is why tangent space is called tangent space.

What we want are two tangents which are perpendicular to each other. But which tangents should we use? On a circle, each point has exactly one tangent, but in 3D space, each surface has an infinite number of tangents!

Well, it doesn't matter which tangents we use, as long as they're perpendicular. It's quite simple to get our first ***tangent*** – we just subtract one vertex from another vertex, on our surface. The other tangent, which we can call the ***bitangent***, is just the cross product of our ***tangent*** and ***normal***.

Model Space ↔ Tangent Space

Now that we know the definition of our tangent space, let's define it in GLSL as a matrix. In addition to the ***normal*** we already pass into the shader, the ***tangent*** and ***bitangent*** are now also required. This is done in the vertex shader.

```
mat3 tangentToWorld = MV3x3 * mat3(normalize(in_tangent),
                                   normalize(in_bitangent),
                                   normalize(in_normal));
```

We convert the matrix to model view space because that's ultimately where we would have to do our fragment calculations anyway. By doing this, we combine the tangent space with the model view space so we can use it as a single matrix.

It's easy to create a matrix which converts the opposite way – we just ***transpose*** it.

```
mat3 worldToTangent = transpose(tangentToWorld);
```

Implementing Normal Mapping

As with previous lighting shaders, the eye vector needs be passed from the vertex shader to the fragment shader.

```
vec3 eyeVec = -modelVertex;
ex_eyeVec = eyeVec;
```

Wrong. Our normals will be in tangent space, so we need to do our lighting calculations there too.

```
vec3 eyeVec = -modelVertex;
ex_eyeVec = worldToTangent * eyeVec;
```

Let's do the same for the light vector.

```
ex_lightVec = worldToTangent * (in_lightVec – eyeVec);
```

We use a standard lighting shader for the fragment shader. The only addition is this line, which looks suspiciously like the earlier incorrect line.

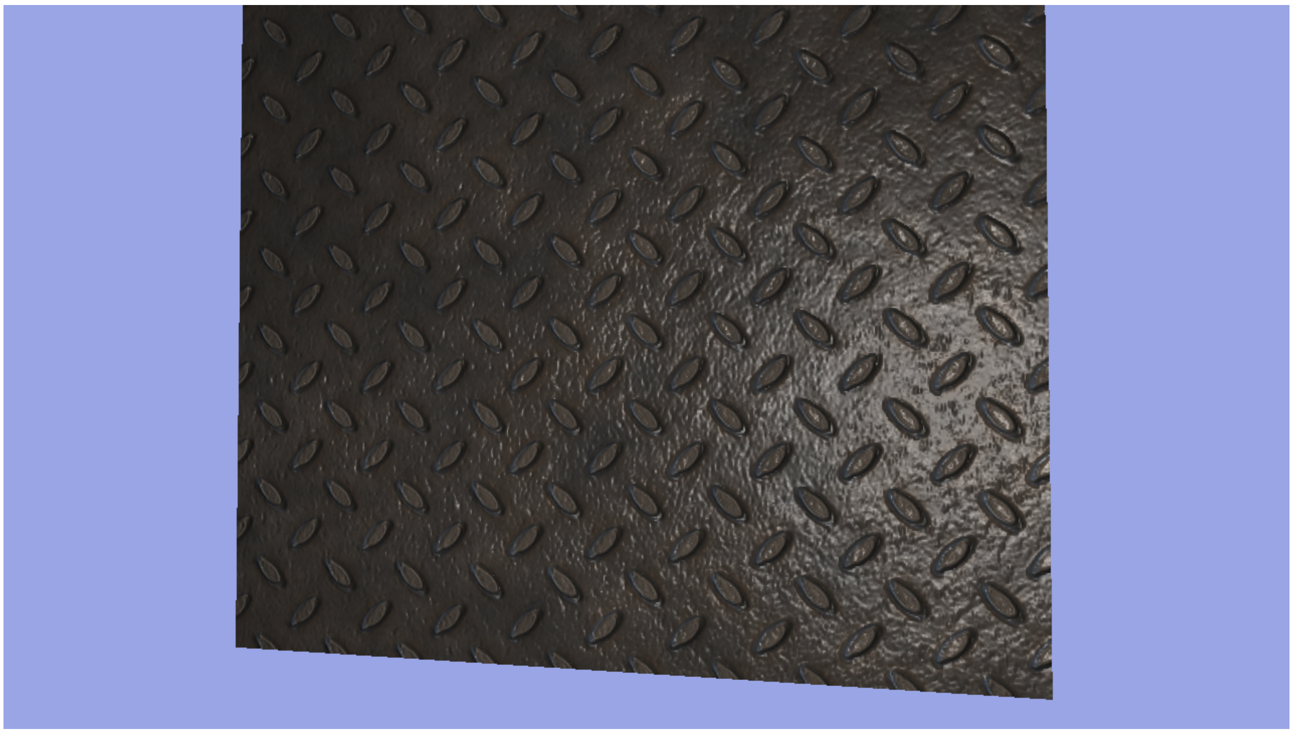
```
    vec3 normal = texture(u_normalMap, ex_texCoord).rgb * 2.0 – 1.0;
```

Take a moment to read that line again.

Why is it being multiplied and subtracted, you ask? Well, the components of a normal must be in the range -1 ↔ 1. If the magnitude of any one component was greater than 1, the length of the normal will definitely be greater than 1.

This calculation takes the rgb values of the map, which are each in the range 0 ↔ 1, multiplies them by 2 to scale to the range 0 ↔ 2, and then subtracts one so they are in the range -1 ↔ 1.

This is how the normals are stored in a normal map, because it allows any possible normal, to a certain precision, to be stored as a single rgb pixel.



Notice how the light – particularly noticeable with specular light – reflects off different parts of the texture in different ways. The result is a much more realistic, lit, textured surface.