Dual Paraboloid Mapping In the Vertex Shader

By Jason Zink

Introduction

I had heard about ‘dual-paraboloid mapping’ some time ago and always wanted to look into how the algorithm worked. The name itself begs to be investigated, and the paraboloid mapping could be applied to environment maps as well as shadow maps so it seemed like a good idea to look into it.

When I recently found some spare time to read up on it I found several different academic papers on the topic explaining the theory behind the algorithm. However, after several rounds with Google I realized that there weren’t many good resources on how to implement the mapping scheme. So I decided that I would investigate the topic and try to provide an easy to understand guide to the theory as well as a reference to the implementation of paraboloid mapping in the modern programmable rendering pipeline.

The implementation is written in DirectX 9 HLSL. The effect files that are developed in this exercise are provided with the article and may be used in whatever manner you desire.

Algorithm History

Before diving into the algorithm, I want to point out where the idea came from and why it was developed. The original motivation behind paraboloid mapping was presented in the paper “View-Independent Environment Maps” by Heidrich and Seidel. This new parameterization was intended to provide an alternative to spherical and cubical parameterizations used in environment mapping.

Spherical mapping suffers from wildly non-uniform sampling rates across its surface, making it difficult to use as a general-purpose environment mapping strategy. On the other hand, cube mapping provides a much more uniform sampling rate, but also requires six rendering passes to generate a complete environment of the current scene. This is quite an expensive algorithm to implement in a real-time rendering scenario.

Paraboloid mapping provides a good balance between these two methods. It produces a representation of the current scene about a single point with two surfaces, one for the forward facing hemisphere and one for the backward facing hemisphere. Its sampling rate is also more uniform than spherical mapping.

Another paper introduced the paraboloid parameterization to shadow maps: “Shadow Mapping for Hemispherical and Omnidirectional Light Sources” by Stefan Brabec. This paper provided the framework for building shadow maps that could cover an entire half of a scene with a single map or the entire scene with two maps. As opposed to standard cubical shadow mapping, the paraboloid parameterization saves a significant amount of work.

Algorithm Overview

So what exactly is paraboloid mapping? First let’s look at the shape of a paraboloid to understand how it works. Figure 1 shows a paraboloid.

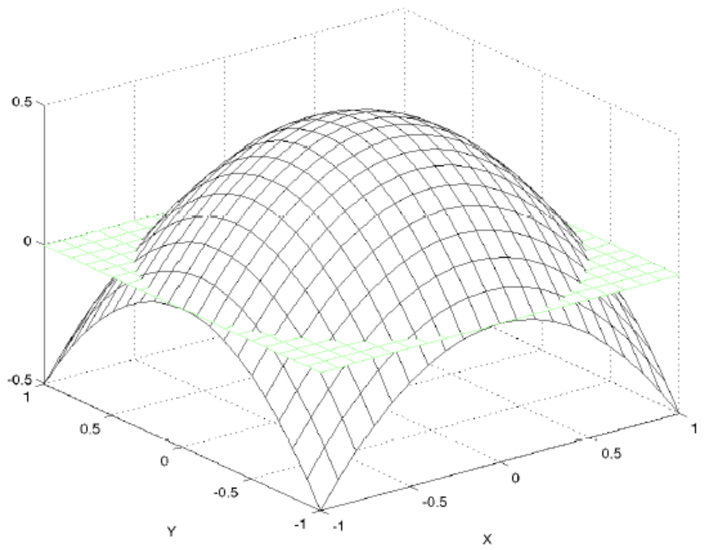

Figure 1: Surface plot of a paraboloid (image taken from Brabec)

The basic idea behind it is relatively simple. For a given point P in a 3D scene we can divide the scene into two hemispheres, we’ll call them forward and backward. For each of these hemispheres there exists a paraboloid surface that will focus any ray traveling toward point P into the direction of that hemisphere. Here is a 2D image that shows this idea.

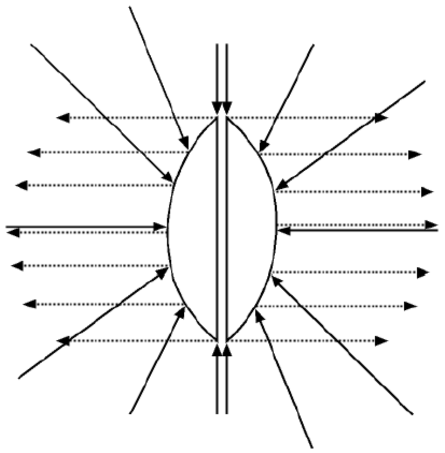
Figure 2: Dual paraboloids reflecting rays into parallel directions (image taken from Brabec)

Figure 2 shows both hemispheres reflecting rays into a parallel stream away from the point P. This is the key to the paraboloid mapping – environment mapping encodes the light surrounding a point into a grid of values to be stored in a map while shadow mapping does the same for depth values.

This encoding is based on the incoming rays being reflected about the paraboloid surface's normal vector. As we will see in the Mathematic Background section, this idea is the basis of the implementation of paraboloid mapping.

Mathematic Background

The mathematic definition of the paraboloid that we will be working with is:

$$f(x, y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2) \text{ for } x^2 + y^2 \leq 1$$

Look at Figure 1 to see how this equation is evaluated at various values of x and y. This equation is going to be used throughout this section to provide a mathematic basis for the implementation that we will be doing later on.

The first task at hand is to be able to find the normal vector at any point on the paraboloid surface. We will use the relatively well-known method of finding two vectors that are tangential to the surface and taking their cross product to produce the normal vector. The two tangent vectors are found by taking the partial derivatives of the function with respect to x and y. So here is the mathematic version:

$$P = (x, y, f(x, y))$$

$$T_x = \frac{\partial P}{\partial x} = \left(1,0,\frac{\partial f(x,y)}{\partial x}\right) = (1,0,-x)$$

$$T_y = \frac{\partial P}{\partial y} = \left(0,1,\frac{\partial f(x,y)}{\partial y}\right) = (0,1,-y)$$

$$N_P = T_x \times T_y = (x,y,1)$$

With the normal vector now known for the entire paraboloid surface, we can solve for the x and y coordinates of the intersection of an incoming ray and the paraboloid surface. This result will later be used to determine the x and y texture coordinates for both generating and accessing the paraboloid maps.

To find the intersection point on the paraboloid, we must have the direction of the incident ray as well as the direction of the reflected ray. Since the paraboloid reflects all rays to the forward direction for the forward hemisphere, then we know the reflected direction – it is the same for all reflections in that hemisphere! Examine figure 2 if this is not clear to you, it is crucial to understand. So the forward hemisphere's reflection vector is always going to be (0,0,1) and the backward hemisphere's reflection vector is always going to be (0,0,-1).

Now that we have both the incident vector and the reflected vector, we can calculate the normal vector that caused the reflection. The sum of the incident and reflected vectors should have the same direction as the normal vector, although with a different magnitude (assuming the vectors are the same length!). Again, in math form:

$$N_P \Leftrightarrow V_{incident} + V_{reflected}$$

Then from our earlier derivation of the normal vector:

$$N_P = (x,y,1) \Leftrightarrow V_{incident} + V_{reflected} = V_{sum}$$

This relationship shows that if we divide the result of the sum of the two vectors by their z component, then the resulting x and y components are the x and y components of the paraboloid surface that reflected the vector.

$$N_P = \frac{1}{z_{sum}}\left(x_{sum}, y_{sum,} z_{sum}\right) = \left(\frac{x_{sum}}{z_{sum}}, \frac{y_{sum}}{z_{sum}}, 1\right)$$

The relationships that we have defined in this section will allow us to implement the paraboloid mapping in HLSL.

Generating Paraboloid Maps

Now that we have a better understanding of the inner workings of the paraboloid parameterization, it is time to put our newly acquired knowledge to use. First we will look at how to generate a dual paraboloid environment map, and then we will look at how to access the map for use in the various shaders that can benefit from it. The sample effect files are written in HLSL, but the implementation should apply to other shading languages as well.

To begin, the application must calculate the transformation matrices to be used in our rendering pipeline. I will quickly discuss the three basic transforms usually used: the world, view, and projection matrices.

The world matrix remains the same as with standard rendering – it just positions the object in world space and is generated in the normal fashion.

The view matrix is generated in the normal fashion as well, but the ‘camera’ that will be used to create it from is really the point P. So to create the view matrix, the world space position of P will be used as the translation part, and an appropriate orientation has to be selected to provide the rotation part. The orientation that is used will determine what will end up being the forward and backward directions for the two paraboloids.

The projection matrix is actually not needed to generate the paraboloid map. The input geometry will be transformed into camera space and its position will be directly manipulated in the vertex shader. In this example the projection matrix is simply set to the identity matrix.

Once all three matrices have been calculated, we are ready to start rendering our geometry. The majority of the work in producing a paraboloid map is done in the vertex shader. You can think of the overall algorithm that we are trying to implement as requiring us to place each vertex into a location in the paraboloid map that can later be looked up according to the mathematic rules that we defined earlier.

The first operation needed is to transform the incoming vertices with the combined world, view, and projection matrix object space to post projection space and divide by the w coordinate to produce a homogenous position in camera space.

```
	OUT.position = mul( float4(IN.position, 1), mWVP );
	OUT.position = OUT.position / OUT.position.w;
```

To find the x and y coordinates of the map that we should store this vertex at, we will need to have a vector from the point P to the transformed vertex. In the paraboloid basis (set by the view matrix earlier) point P is actually located at (0,0,0) so finding a vector from P to the vertex is as simple as dividing P by the length of the vector from (0,0,0) to P.

```
	float L 		= length( OUT.position.xyz );
	OUT.position 	= OUT.position / L;
```

The resulting vector represents a ray from the vertex toward point P. Now we must find the x and y coordinates of the point where this ray intersects the paraboloid surface. From the earlier findings, the normal vector at any point on the paraboloid surface can be found by adding the incident and reflected vectors, and then dividing that result by its z component. We know that the reflected vector is always (0,0,1) due to the properties of the paraboloid (see **Figure 2**). So to compute the normal vector, we only have to add 1 to the z component and then divide the x and y components by this new z value.

```
OUT.position.z = OUT.position.z + 1;
OUT.position.x = OUT.position.x / OUT.position.z;
OUT.position.y = OUT.position.y / OUT.position.z;
```

This x and y coordinates represent the positions in the output texture that this vertex maps to in the paraboloid parameterization. To have proper depth testing, I have set the z coordinate back to the distance from P to the vertex in world space scaled by the viewing distance. In addition, the w component is set to 1 to finalize the x, y, and z values output by the vertex shader.

```
OUT.position.z = (L - 0.1)/(500.0-0.1);
OUT.position.w = 1;
```

The only change necessary between rendering the front map and the back map is to multiply the z coordinate by –1 before dividing by the distance to the vertex. Also, back face culling is disabled in the sample file so that the same effect can be used on both maps (by negating z the triangle winding would be reversed). This could also be done with two different techniques, but I thought it would be simpler this way.

To see a complete listing of the vertex shader that we have built over the last few paragraphs as used in generating a two texture terrain, see the provided effect file: ***dual_paraboloid_terrain.fx***
Here are some sample paraboloid maps generated from a Terragen terrain file:

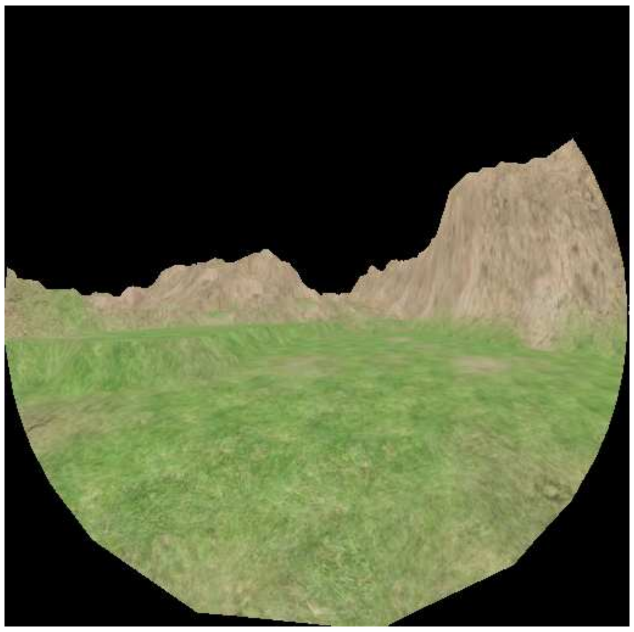
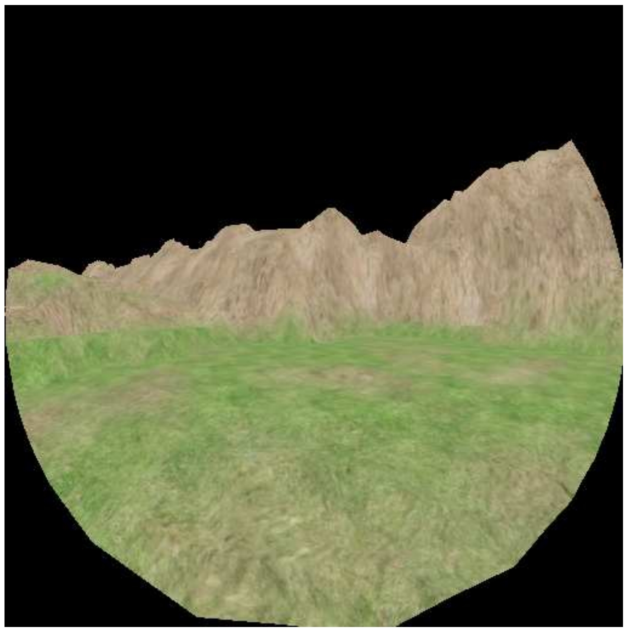

Figure 3: Sample Front and Back Paraboloid Maps

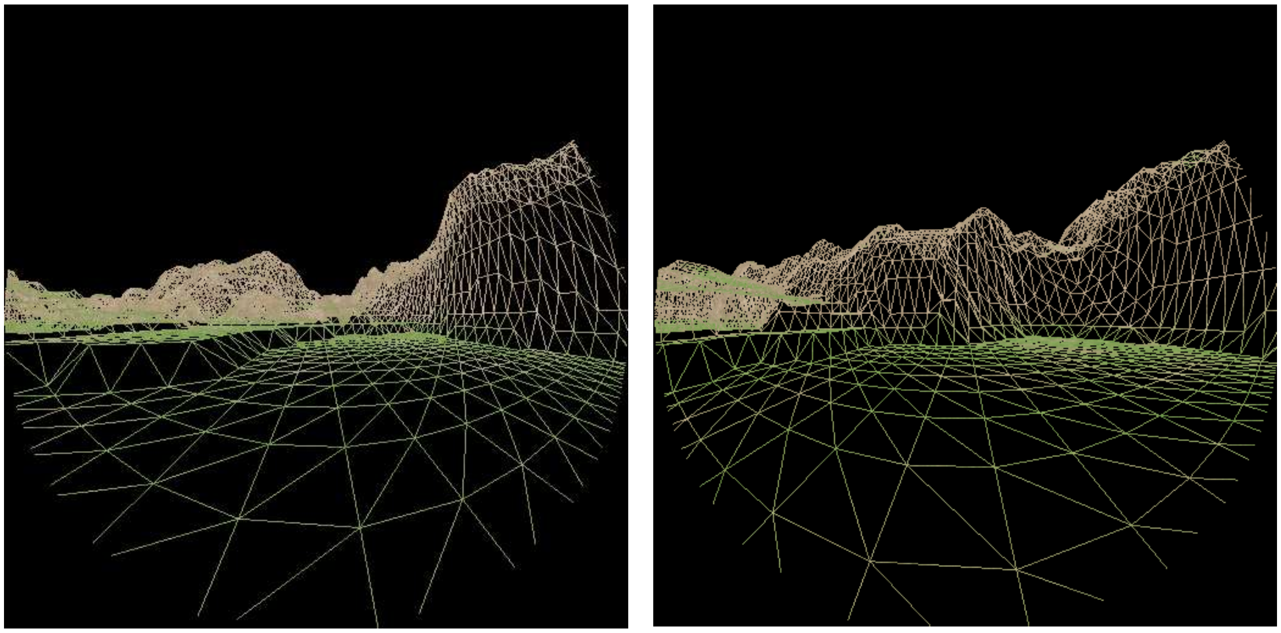

Figure 4: Wireframe Views of Sample Front and Back Paraboloid Maps

Accessing Paraboloid Maps

Now that we have generated the paraboloid map, we need to be able to access it to put it to good use. How the data is accessed depends on what you are using the maps for, but the general idea applies to any paraboloid map access. The process of calculating the texture coordinates to access the paraboloid maps is quite similar to the method used to generate the map.

As an overview of the algorithm, we will follow these steps:

1) Find a vector from P to the desired object
2) Use this vector to calculate u and v coordinates (one set for each hemisphere)
3) Sample both paraboloid maps with the coordinates
4) Do something with the sampled values

In the case of environment mapping, the vector from P to the desired object is a vector from the camera to the point on the environment-mapped object, which is then reflected about that point's normal vector. So the camera sees what is reflected off of the environment-mapped object. So the first step is to generate the reflected vector:

```
	float3 N = normalize( IN.normal );
	float3 E = normalize( IN.eye );
```

```
        float3 R = reflect( E, N );
```

This reflected vector is now in world space. It is necessary to transform this vector into the paraboloid map's basis. This is essentially done by multiplying by the view transform from the generation phase:

```
        R = mul( R, (float3x3)ParaboloidBasis );
```

With the reflected vector now available, the texture coordinates are calculated in the same way as when generating the paraboloid map. The resulting values are then scaled and biased to account for the texture addressing modes of Direct3D:

```
        float2 front;
        front.x = (R.x / (2*(1 + R.z))) + 0.5;
        front.y = 1-((R.y / (2*(1 + R.z))) + 0.5);

        float2 back;
        back.x = (R.x / (2*(1 - R.z))) + 0.5;
        back.y = 1-((R.y / (2*(1 - R.z))) + 0.5);
```

These coordinates are then used to sample the two paraboloid maps:

```
        float4 forward = tex2D( FrontSampler, front );
        float4 backward = tex2D( BackSampler, back );
```

I chose to use border texture addressing with a black color outside of the texture so that I could simply take the max of the two sampled values to get the environment color at that point. This is shown here:

```
        OUT.color = max(forward, backward);
```

The resulting color should be the reflected environment color at that point. Here is a screen shot from rendering a poorly tessellated sphere using the dual paraboloid environment map:

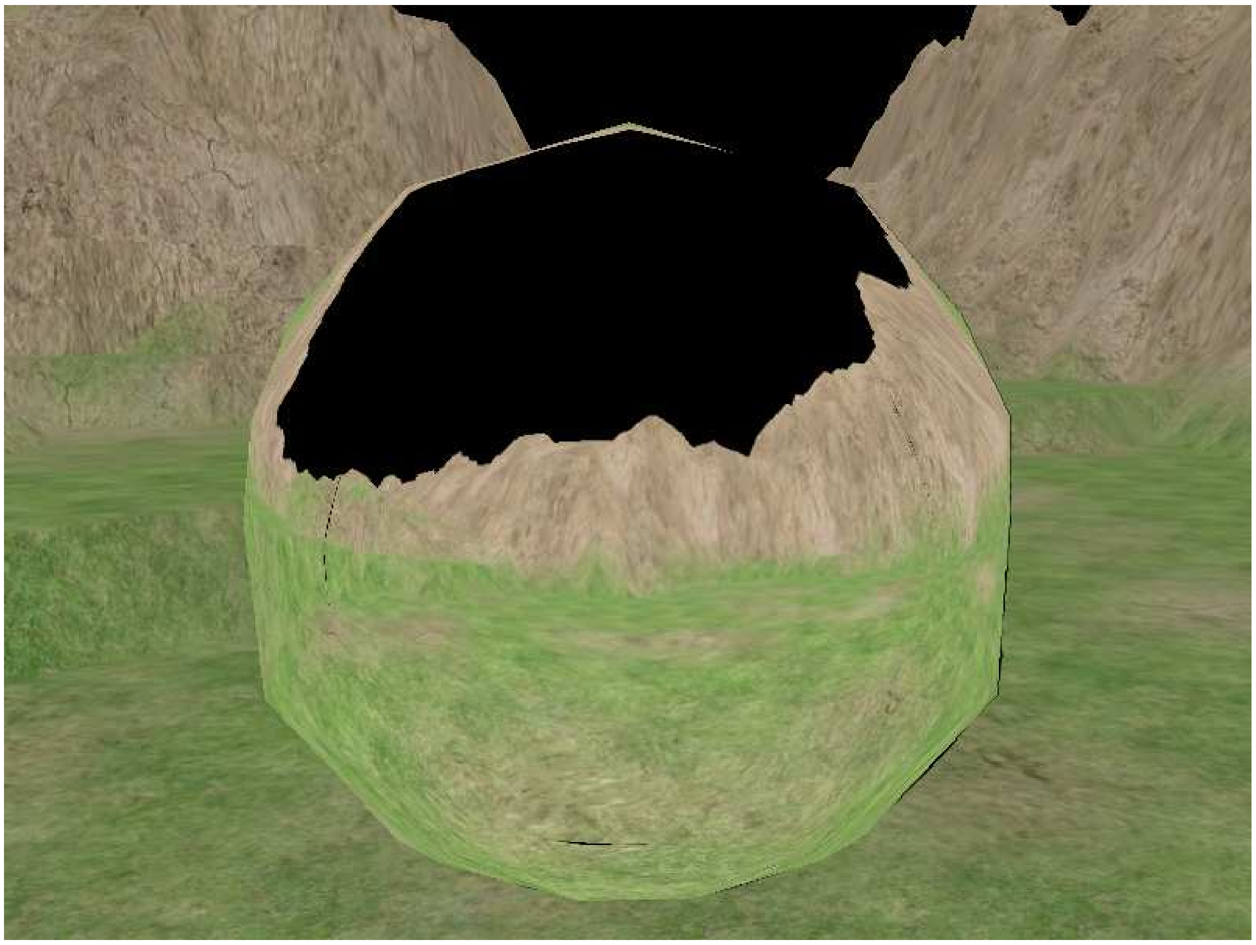
Figure 5: Dual Paraboloid Environment Mapped Sphere

For a complete effect file that implements this dual paraboloid environment mapped effect, see the included file: ***dual_paraboloid_environment_map.fx***

Conclusion

Now that we understand how paraboloid mapping works, lets consider some of the down sides associated with it. Paraboloid mapping is not perfect. If you examine the wire frame views in Figure 4, you will see that the triangle edges are still straight lines. This is because we apply the paraboloid warping to the vertices of a model when generating the map. The triangle is still rasterized in the same manner – a line segment from vertex to vertex in clip space forms each edge. At first this does not seem to be that big of a deal.

However, if you look closely at Figure 5 you will see a couple of areas where artifacts are introduced into the reflected image. This is the area where the front and back maps meet. Around the edges of the paraboloid map shown in Figure 3, there are small areas that appear to be cut into straight lines. This is caused by the scene not being tessellated highly enough to counteract the rasterization effects we just described. Even so, the paraboloid parameterization is still a useful tool to have at your disposal. When used properly it is capable of providing a significant performance increase as opposed to cube mapping, and a quality increase as opposed to spherical mapping.

I decided to write this article due to the lack of implementation details available on the web. Hopefully this guide has provided you with some insight into the world of paraboloid mapping. If you feel that this document could be improved or have questions or comments on it please feel free to contact me as ‘Jason Z’ at gamedev.net.