Dual Paraboloid Environment Mapping

Whitepaper

Copyright © Imagination Technologies Limited. All Rights Reserved.

This publication contains proprietary information which is subject to change without notice and is supplied 'as is' without warranty of any kind. Imagination Technologies and the Imagination Technologies logo are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.

Filename	:	Dual Paraboloid Environment Mapping.Whitepaper
Version	:	PowerVR SDK REL_16.1@3976567a External Issue
Issue Date	:	17 Mar 2016
Author	:	Imagination Technologies Limited

Contents

1. Introduction 3

2. Technique 4

2.1. Mathematics 4
2.2. Implementations 5
2.2.1. Render to Two Textures 6
2.2.2. Render to Two Faces of a Cube Map Texture 6
2.2.3. Render to Two Halves of a Single Texture 7

3. Glass Example 8

3.1. Shader Flags for Various Effects 9
3.2. Rendering the Skybox 11

4. Contact Details 12

List of Figures

Figure 1. A cube map 3
Figure 2. Reflected rays on a paraboloid surface 4
Figure 3. Dual paraboloid maps 6
Figure 4. Sampling from the skybox separately in the glass ball's fragment shader 8
Figure 5. Rendering the skybox into the paraboloid maps 8
Figure 6. Reflection and refraction with chromatic dispersion 9
Figure 7. Reflection and refraction without chromatic dispersion 10
Figure 8. Reflection only 10
Figure 9. Refraction with chromatic dispersion 11
Figure 10. Refraction without chromatic dispersion 11

1. Introduction

Reflective or refractive surfaces such as glass are often rendered using environment maps. An environment map is typically a texture or set of textures that contains an image of the scene surrounding a given point. This given point is frequently the centre of the reflective or refractive object itself.

Cube maps are commonly used for environment mapping (see Figure 1). These maps contain six textures, one for each face of the cube. Having six faces, however, brings up a problem when environmental conditions change and the environment map needs updating. Rendering a scene six times, once to each face, is a costly operation, and when there is a need to perform this on every frame this can generate a bottleneck.

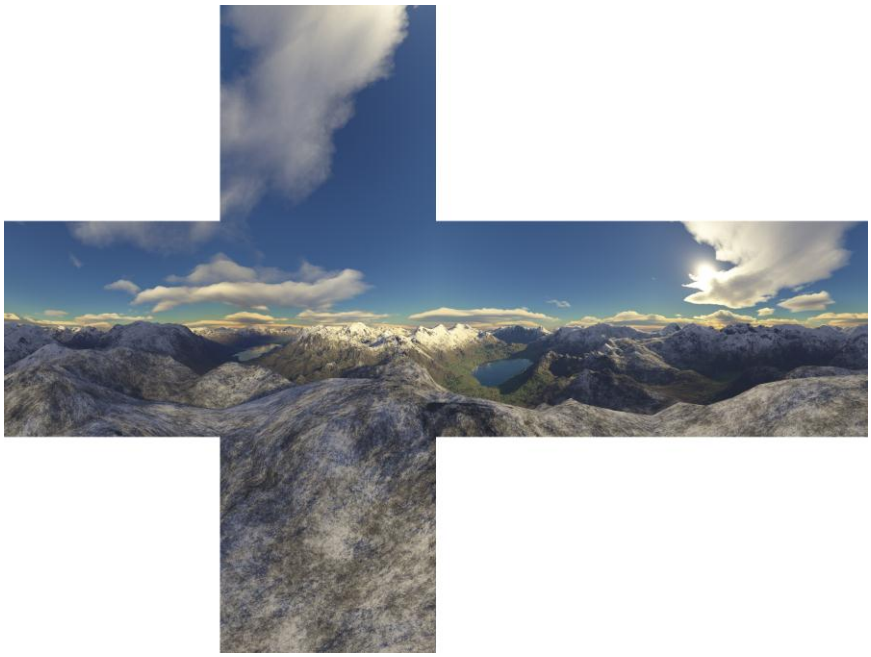

Figure 1. A cube map

Sphere maps are another type of environment map that captures a scene in a single texture from a given point. A sphere map only requires to be rendered once to update the environment map, while cube maps require six renders to update the entire environment map. However, the rapid loss of detail towards the edge of each sphere map is a major disadvantage.

Another technique is dual paraboloid mapping. This technique maps the environment to two paraboloid images, one for each hemisphere. Updating a dual paraboloid map only requires two renders, while a cube map requires six. Using dual paraboloid maps is not as graphically accurate as using cube maps, but the reductions in rendering requirements make it ideal for dynamic environment mapping.

2. Technique

2.1. Mathematics

For a hemisphere around a given origin, there is a paraboloid surface that reflects all incident rays directed at the origin in a constant direction. In Figure 2 next, all incident rays are reflected in the upward direction.

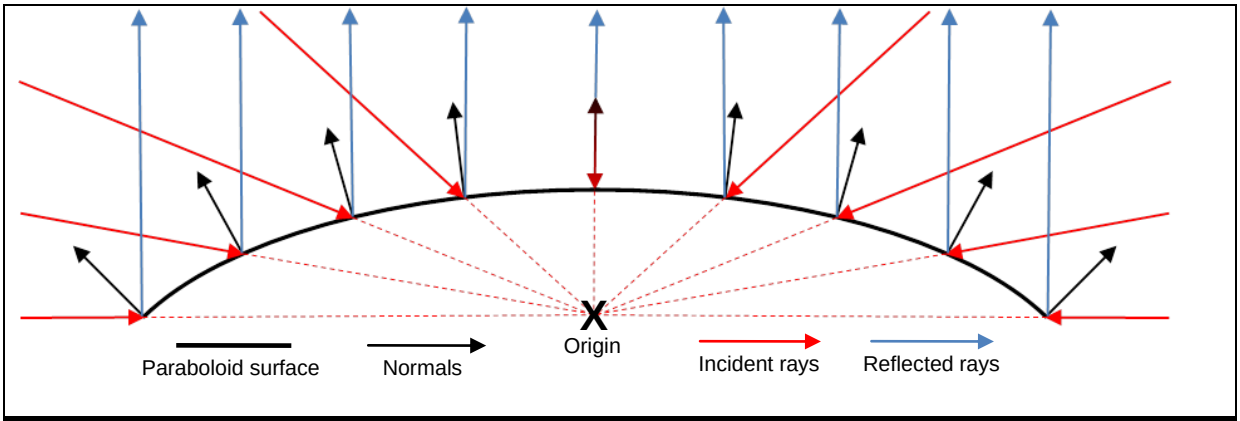

Figure 2. Reflected rays on a paraboloid surface

One such paraboloid, which is used for this technique, has the following equation:

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

Given a particular incident ray directed at the paraboloid's origin, the surface normal at the point of its intersection with the paraboloid can be used to calculate its position in the environment map.

A point on the surface of the paraboloid can be defined as:

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

Given this, the tangent vectors in both the x and y directions can be defined using partial derivatives of the paraboloid function:

$$T_x = \frac{dP}{dx} = \left(1,0,\frac{df(x,y)}{dx}\right) = (1,0,-x)$$

$$T_y = \frac{dP}{dy} = \left(0,1,\frac{df(x,y)}{dy}\right) = (0,1,-y)$$

The cross product of these two tangent vectors defines the normal vector:

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

The x and y values in this normal vector are used to place incident rays in the 2D space of the environment map.

To calculate the normal at the point of intersection, the incident ray can be added to its reflection vector. This produces a vector in the right direction, but with a different magnitude.

$$S_p = I_p + R_p$$
$$S_p = (x_s, y_s, z_s) \Leftrightarrow N_p = (x, y, 1)$$

Dividing this vector by its own z component scales it down to the normal vector with a z value of 1, resulting in the coordinates in the environment map.

$$\frac{1}{z_s}(x_s, y_s, z_s) = \left(\frac{x_s}{z_s}, \frac{y_s}{z_s}, 1\right) = (x, y, 1) = N_p$$

So by adding the incident ray to the reflected ray and dividing the resulting vector by its z value, the mapped coordinate is found. With the reflected ray known for all incident rays that are directed at the paraboloid's origin (as it is always the direction the paraboloid is facing), only the incident ray is needed to perform the mapping.

2.2. Implementations

When rendering to a paraboloid map (see Figure 3), a special vertex shader needs to be used to adjust incoming vertices in the scene into the correct clip space positions.
The view matrix used in the vertex shader is from the point of view of the paraboloid's origin, facing in the direction of the paraboloid. This means the vertices in front of the paraboloid have a negative Z value.

```
// Transform position to the paraboloid's view space
gl_Position = MVMatrix * vec4(inVertex, 1.0);

// Store the distance
highp float Distance = -gl_Position.z;

// Calculate and set the X and Y coordinates
gl_Position.xyz = normalize(gl_Position.xyz);
gl_Position.xy /= 1.0 - gl_Position.z;

// Calculate and set the Z and W coordinates
gl_Position.z = (Distance / Far) * 2.0 - 1.0;
gl_Position.w = 1.0;
```

The vertex shader works by following the maths described previously. The incoming vertex is transformed into the paraboloid's space using the model view matrix. After this, it is normalised to produce the incident ray.
The reflected ray is the direction of the paraboloid, which is along the z axis. As the x and y coordinates of the reflected ray are 0, the x and y coordinates of the summed incident and reflection ray are just the same as those in the incident ray. This means that only the x and y coordinates of the incident ray need to be divided by the z coordinate of the combined incident and reflection ray.
As the z coordinate for a vertex in front of the paraboloid is negative, it is subtracted from 1 instead of added to 1 to produce positive results. This value is then used to divide the x and y coordinates of the incident ray to produce the x and y clip space coordinates, both between -1 and 1.

Figure 3. Dual paraboloid maps

With just a single paraboloid map detailing half a scene from a given point, a solitary 2D texture can be bound to a single frame buffer. However, with dual paraboloids detailing the whole scene, various approaches can be taken to handle this effectively.

2.2.1. Render to Two Textures

The obvious way to render the two halves of the scene is into two separate 2D textures. To do this, two frame buffers are created and each one is bound to its own 2D texture. At the start of each frame, the two frame buffers are bound in turn and the scene rendered to each, with the view matrix set in opposite directions from the origin of the environment mapping (the centre of the object that is to reflect or refract light).
This technique is fine when it comes to creating and updating the two paraboloid maps, but when it comes to sampling from them in the fragment shader of a reflective or refractive surface, the textures need to be sampled according to a branch, which can double the cost of shading.

```
ReflectDir.xy /= abs(ReflectDir.z) + 1.0;
ReflectDir.xy = ReflectDir.xy * 0.5 + 0.5;

if (ReflectDir.z > 0.0)       {
        gl_FragColor = texture2D(sFront, ReflectDir.xy);
} else {
        gl_FragColor = texture2D(sBack, ReflectDir.xy);
}
```

To avoid the disadvantage of this method, the two paraboloid maps need to be placed inside a single texture so that it can be sampled in one texture read without any branching.

2.2.2. Render to Two Faces of a Cube Map Texture

As OpenGL ES 2.0 supports cube mapping, the two maps can be packed into two faces of a cube map. To do this, the two frame buffers can be bound to two opposing faces of a cube map and in the fragment shader of the reflective or refractive surface the correct 3D texture coordinates can be generated in order to sample the right point of the correct face.

```
ReflectDir.xy /= abs(ReflectDir.z) + 1.0;
ReflectDir.z = step(0.0, ReflectDir.z) * 2.0 - 1.0;
ReflectDir.xy *= -1.0;

gl_FragColor = textureCube(sParaboloids, ReflectDir);
```

One issue with using a cube map to store the two paraboloid maps is that four of the faces are effectively wasted. Space needs to be allocated for all of the faces of a cube map, so this technique requires a lot of memory, especially for large texture sizes.

2.2.3. Render to Two Halves of a Single Texture

Another method with similar performance to using a cube map is rendering twice to a single 2D texture. Instead of using a square power-of-two texture, a rectangular power-of-two texture can be used, with each paraboloid map placed into one half. This means that only one frame buffer is needed instead of two, with the viewport changing between scene renders.

```
ReflectDir.xy /= abs(ReflectDir.z) + 1.0;
ReflectDir.xy = ReflectDir.xy * 0.5 + 0.5;
ReflectDir.x *= 0.5;
ReflectDir.x += sign(-ReflectDir.z) * 0.25 + 0.25;

gl_FragColor = texture2D(sParaboloids, ReflectDir.xy);
```

The fragment shader code is similar to that of the cube map method, but with calculations to work out the correct 2D texture coordinates to sample from instead of using 3D coordinates.

3. Glass Example

The glass example in the SDK features a floating faceted glass ball, reflecting and refracting the surrounding dynamic environment (Figure 4 and Figure 5). It makes use of dual paraboloid mapping, rendering the maps to two halves of a single 2D texture as previously described.

Figure 4. Sampling from the skybox separately in the glass ball's fragment shader

Figure 5. Rendering the skybox into the paraboloid maps

Four directions are calculated at each vertex of the glass ball inside its vertex shader. One is the direction of the reflected ray, while the other three are the directions of the red, green and blue refracted rays. These four directions are interpolated across the surface of the glass ball and each one is used in the fragment shader to generate a texture coordinate to sample the dual paraboloid texture. This alone results in four texture reads per fragment of the glass ball.

However, the glass example also features a cube map representing a skybox. As the skybox is already an environment map, it does not need to be rendered into the dual paraboloid maps and can be sampled independently using the four ray directions. This doubles the number of texture reads in the fragment shader to eight.

An alternative to doing this is to render the skybox into the dual paraboloid maps, so that only the four texture reads are needed in the glass ball's fragment shader. One of the major disadvantages of this method is that skybox detail is lost and the quality of the overall reflection and refraction are reduced, especially when the paraboloid map texture is significantly smaller than the skybox's static cube map.

This may also be more expensive if the size of the paraboloid maps are significantly larger than the size of the glass ball in terms of fragments shaded. If the number of fragments shaded with the skybox in the two paraboloid maps is four times the number of fragments in the glass ball, the number of texture reads is the same.

To correctly combine the colour values from the eight texture reads, the paraboloid maps need a clear colour with an alpha value of zero. This allows alpha blending between the paraboloid and skybox colour values for each of the four rays.

As the colours from the three refraction rays are for the three colour channels, they can be combined into one refraction colour. After that, the refraction and the reflection colours can be blended to form the final colour output using a reflection factor determined by a Fresnel approximation.

The shader code to combine the various texture samples into the final fragment colour is next displayed.

```
// Combine skybox reflection colour with paraboloid reflection colour
Reflection.rgb = mix(SkyReflection.rgb, Reflection.rgb, Reflection.a);

// Combine skybox refraction colours with paraboloid refraction colours
Refraction.r = mix(SkyRefraction.r, RedRefraction.r, RedRefraction.a);
Refraction.g = mix(SkyRefraction.g, GreenRefraction.g, GreenRefraction.a);
Refraction.b = mix(SkyRefraction.b, BlueRefraction.b, BlueRefraction.a);

// Combine reflection and refraction colours for final colour
gl_FragColor.rgb = mix(Refraction.rgb, Reflection.rgb, ReflectFactor);
```

3.1. Shader Flags for Various Effects

By default, the glass ball reflects light and refracts light with chromatic dispersion as previously described. However, it also allows for certain effects to be disabled. In total there are five modes for the glass ball (Figure 6 to Figure 10).

Figure 6. Reflection and refraction with chromatic dispersion

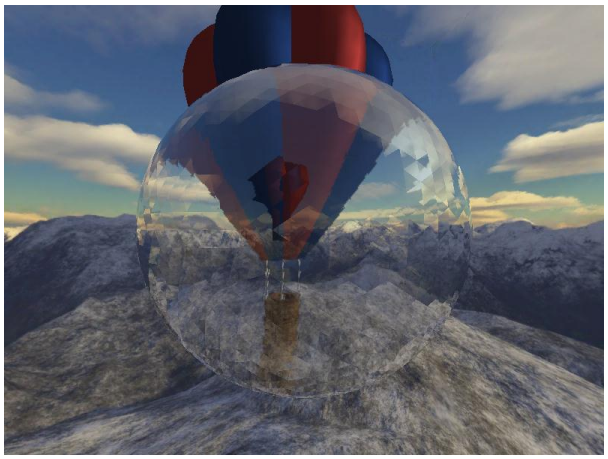

Figure 7. Reflection and refraction without chromatic dispersion

The first two modes are the full effect and one without the chromatic dispersion. This second effect is significantly faster because only two rays need to be calculated in the glass ball's vertex shader and only four texture reads need to be performed in the fragment shader (two for the paraboloid maps and two for the skybox's cube map).

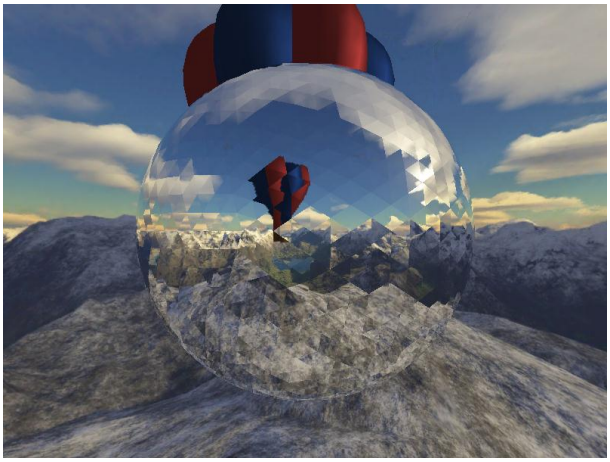

Figure 8. Reflection only

The last two modes are for refraction only, with and without chromatic dispersion. The effect without chromatic dispersion only requires two texture reads, the same as reflection only. The effect with chromatic dispersion requires six texture reads, two for each of the three refraction rays.

Instead of writing separate shaders for each of these effects, the all-inclusive shader (for the first effect) can be created with pre-processor directives to change which code is compiled each time. With this in place, the one shader can be compiled five times with various `#define` statements declared.

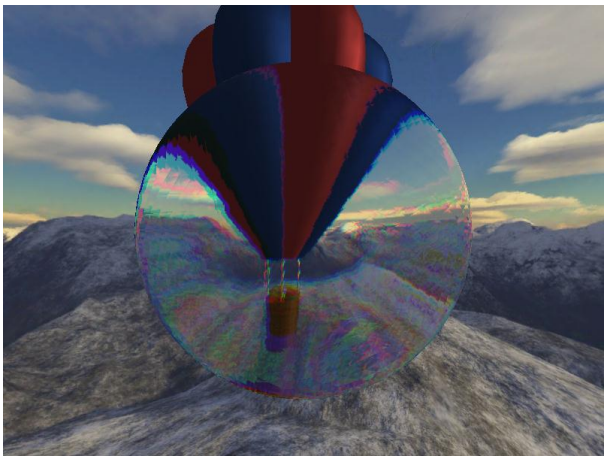

Figure 9. Refraction with chromatic dispersion

Figure 10. Refraction without chromatic dispersion

3.2. Rendering the Skybox

A full screen quad with vertex positions in clip space is used to render the skybox to the screen. In its vertex shader, the direction to the vertex in world space is calculated by multiplying the clip space coordinates by the inverse view projection matrix and subtracting the world space eye position. This direction is normalised and interpolated across the quad.

```
// Set position
gl_Position = vec4(inVertex, 1.0);

// Calculate world space vertex position
vec4 WorldPos = InvVPMatrix * gl_Position;
WorldPos /= WorldPos.w;

// Calculate ray direction
RayDir = normalize(WorldPos.xyz - EyePos);
```

In the fragment shader, this direction is used to sample the skybox's cube map to get the output colour.

4. Contact Details

For further support, visit our forum:
http://forum.imgtec.com

Or file a ticket in our support system:
https://pvrsupport.imgtec.com

To learn more about our PowerVR Graphics SDK and Insider programme, please visit:
http://www.powervrinsider.com

For general enquiries, please visit our website:
http://imgtec.com/corporate/contactus.asp

Imagination Technologies, the Imagination Technologies logo, AMA, Codescape, Ensigma, IMGworks, I2P, PowerVR, PURE, PURE Digital, MeOS, Meta, MBX, MTX, PDP, SGX, UCC, USSE, VXD and VXE are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.