



SDK White Paper

Rainbows and Fogbows Adding Natural Phenomena

WP-01410-001-v01
July 2004

NVSDK

Abstract

This document describes a method to render rainbows, coronas, fogbows, and halos realistically with a 3D scene in real-time using pre-calculated lookup textures. A summary of the basic physics of rainbows and other atmospheric phenomena is also provided.

Rainbows, fogbows, corona (around the sun) and halos (around the moon) are each created when small water droplets scatter light in a particular way as it travels to your eye.

This whitepaper tells you how to create this effect in a 3D scene without computing a complex equation for every pixel on the screen. It explains the basic optics behind rainbows, fogbows, coronas and halos and tells you where to find out all the gritty details behind the physics. It lists the steps to render a rainbow using lookup textures. It discusses how to combine these light effects with the rest of your scene and describes directions for future work. After reading this whitepaper you should have all the info you need to add a realistic rainbow, fogbow, corona, or halo to your game engine.



Clint Brewer
devrelfeedback@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

The Optics of Rainbows

Everyone since Aristotle has been trying to explain how rainbows work. For recent treatments of the topic, look to *The Rainbow Bridge*, a book that covers just about everything known and unknown about rainbows. This whitepaper concerns itself with rendering realistic rainbows in an interactive environment, it does not provide all the physical details. It provides enough information about the optics of rainbows to make it possible to add believable-looking rainbow effects to games.

Rainbows appear when sunlight is separated into its component colors by spherical droplets of water. Different wavelengths of light (red, orange, yellow, green, blue, and violet) are refracted by different amounts as they enter and exit the water droplet. Other aspects of rainbows include the following:

- ❑ Rainbows are circles of colored light centered on the anti-solar point. If you are the viewer, then the anti-solar point is the shadow of your own head.
- ❑ Every person sees a unique rainbow. You can think of it as a cone of light focused on your eye.
- ❑ Spherical water droplets refract and reflect light back to your eye
- ❑ Different wavelengths of light are refracted by different amounts
- ❑ Very tiny spherical water droplets reflect and diffract light causing fogbows

Figure 1 illustrates how a ray of sunlight is refracted by a water droplet, is reflected internally a single time, and is refracted on the way out as it goes to the viewer. The figure shows the path of a ray of sunlight as it passes through the water droplet. The ray refracts into the droplet at point **a**, reflects internally at point **b**, and refracts out of the droplet at point **c**. Violet rays are not refracted as much as the other colors so violet comes out on the bottom of a rainbow, while red is refracted the most and comes out on top of a rainbow. This process shown in Figure 1 causes the bright primary rainbow.

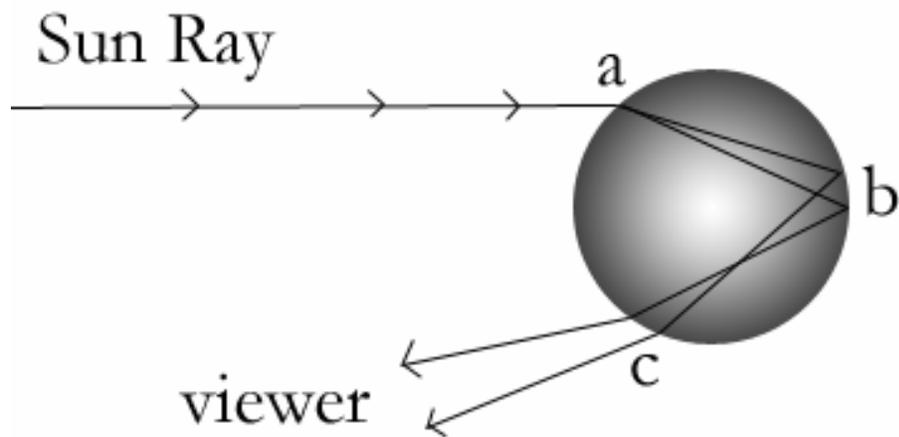


Figure 1. Primary Rainbow

Figure 2 illustrates how a ray of sunlight is refracted by a water droplet, is reflected internally two times, and is refracted on the way out as it goes to the viewer. In the figure, a ray of sunlight refracts into the water droplet at point **d**, reflects internally once at point **c** and again at point **b**, then finally refracts out of the droplet at point **a**. This two-reflection phenomenon causes the lighter secondary rainbow which appears above the primary rainbow on occasion. Note that the colors of the secondary rainbow are reversed from those of the primary rainbow. That is violet on top and red on the bottom instead of the other way around for a primary rainbow (see Figure 8).

The secondary rainbow is a rare event (even more rare than a primary rainbow). Even though this paper does not address how to code the secondary rainbow effect, the techniques discussed in this paper could be used to develop code that simulates the secondary rainbow as well as the primary rainbow.

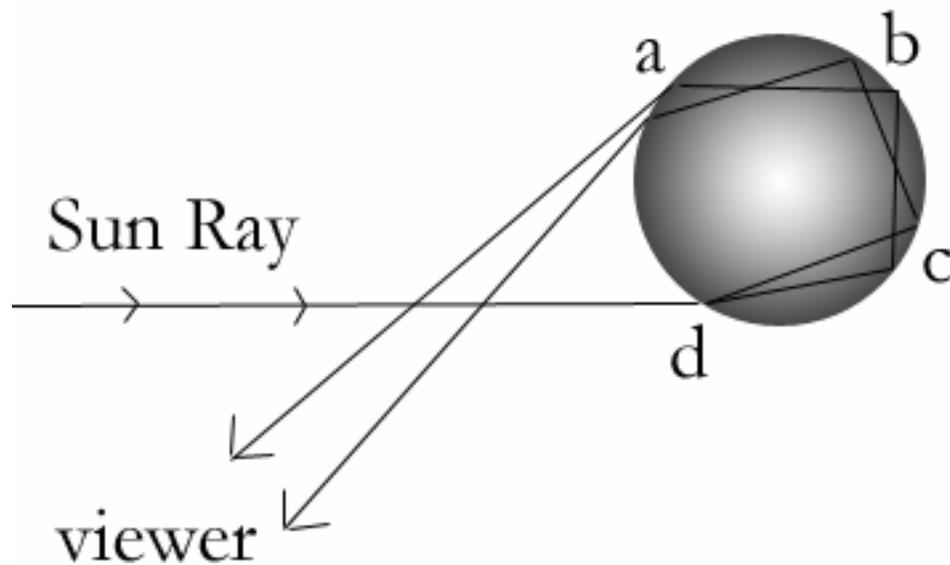


Figure 2. Secondary Rainbow

You can see from the refraction of the sun ray in Figure 1 and Figure 2, the color of light the viewer sees will depend on which color gets bent just enough to hit the viewer's eye. This depends on the viewer's position, the radius and position of the water drop, and the position and angle the sun ray intersects the drop.

We can simplify the problem to be a function of two values: the angle of deviation and the radius of the water droplet. Figure 3 shows the *angle of deviation* which is the total angle that light is bent back to the viewer.

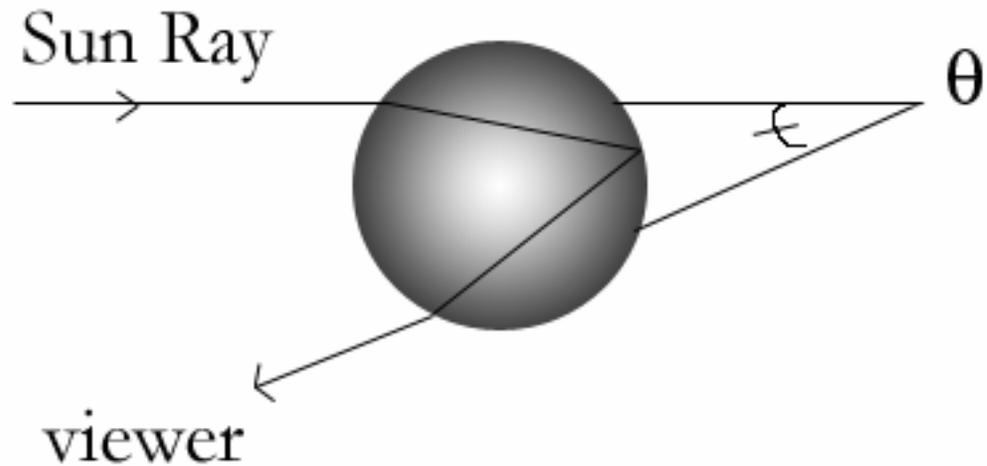


Figure 3. Angle of Deviation

Integrating a Rainbow into a 3D Scene

Distinctive shape and color are well-known characteristics of rainbows. However, rainbows rarely appear perfectly uniform rainbow in reality. Usually we see half a rainbow, or just the top, or see it fade in and out as we watch it. To create a realistic rainbow, these features of rainbows must be included.

This section describes the following characteristics of rainbows and how to integrate them into a scene. Coronas, rings around the sun, are similar and are described in the comments in the sample code found in the section called, “Source Code for Rainbow and Corona Effects.”

- ❑ Color
- ❑ Moisture in the air
- ❑ Light added back to the background
- ❑ Light source color

Color

Rainbow light is caused by water drops in the atmosphere reflecting and refracting sunlight in part of the sky. The previous section explains what causes the color of the rainbow in very simple terms, but this section describes how to color a rainbow in more detail.

The Airy light scattering equations in the article by Raymond Lee, “Mie Theory Airy Theory and the Natural Rainbow,” and the details provided in the article by Phillip Lavin, “The Optics of a Water Drop,” provide helpful models of water drops and light form the colors of a rainbow.

Texture Lookup

To create a color lookup texture for a sample rainbow, we will use a freely available program Lavin created, called MiePlot. The program calculates the results of the Airy light scattering equations that can be used in a graphics application.

MiePlot generates Lee diagrams. These diagrams show how the color of a rainbow changes as the radius of a water droplet changes. Lee diagrams are 2D images in which each pixel represents the color of scattered light corresponding to a given angle of deviation on one axis and radius of water droplet on the other axis.

Figure 4 shows an example of a Lee diagram generated by Mr. Laven's MiePlot.

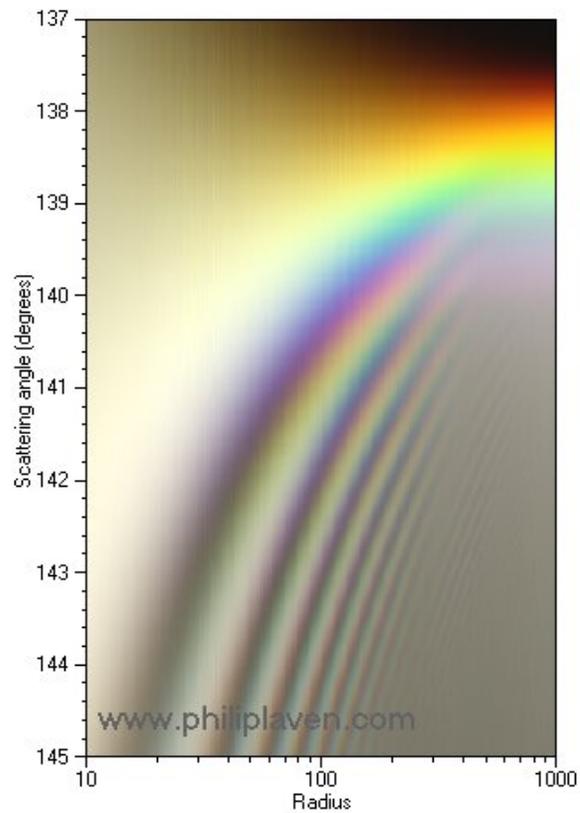


Figure 4. Lee Diagram Generated by MiePlot

In order to add a rainbow in a scene, we need to compute a function of two values:

- Angle of deviation
- Radius of a water droplet.

The Lee diagram gives provides exactly this information when the texture is indexed with the coordinates *[radius, angle of deviation]*.

Some manipulation of the Lee diagram is needed to create a useable lookup texture. Since the primary rainbow is caused by an internal reflection, limit the angle of deviation to be between 90 degrees and 180 degrees. If the angle of deviation is less than 90 degrees then the light ray would not have reflected internally, but refracted out of the water droplet at point **b** in Figure 1. If the angle of deviation is greater than 180, then it mirrors the colors from 90 to 180.

The radii of water droplets varies, depending on the phenomenon. Water droplets in rainbows are large compared to the water droplets in fogbows. To build a model that can render both rainbows and fogbows, use a range 5 to 800 microns.

Lee, in his paper on Mie and Airy Theory, shows that the simplified Airy equations produce very realistic results, even if they are not as accurate as Mie Theory. The Airy Theory can be calculated relatively quickly in MiePlot. To get the final lookup texture, set up MiePlot to generate a Lee diagram using Airy Theory with the desired range of radius and angle of deviation. Since MiePlot does not provide a way to save the texture, take a screenshot of the results and save the Lee diagram portion of the screen.

To achieve more distinctive, brighter rainbow colors, make the bands of color wider, as shown in Figure 5.

In summary, the image in Figure 5, generated by MiePlot, has been hand-modified to enhance width of color bands. The vertical axis is the angle of deviation from 180 to 90. The horizontal axis is the radius of a water droplet from 5 microns to 800 microns. It supplies the rainbow texture lookup.



Figure 5. Hand-Modified Lee Diagram

Calculate Pixel Values

The next problem to solve is – given what we know about a 3D scene – how can we calculate the color of the rainbow at a pixel on the screen?

Because the GPU is set up to access this texture with coordinates in the range $[0..1]$, we need to map angle of deviation and radius to these values. If for each pixel on the screen, we know what the view vector is that passes through that pixel and what the sun lights direction vector is, then we can use the dot product to get the cosine of the angle between the vectors. The cosine is in the range $[1..0]$ when the angle is between 0 and 90 degrees and in the range $[0..-1]$ when the angle is between 90 and 180 degrees.

In all of the previous figures, the sun light vector points at the water droplet and the view vector points at the viewer which is common in mathematical models of rainbows. For this calculation, use the computer graphics convention that the light vector points away from the water droplet towards the light source and the view vector still points towards the viewer. Instead of the vertical axis ranging from 180 to 90 degrees, it ranges from 90 to 0 degrees. This change maps nicely onto the dot product which ranges from 0 to 1 for an angle between 90 and 0 degrees.

To render the rainbow, use a screen-aligned quad and compute the color of scattered light at each pixel on the screen.

Note: The screen-aligned quad view approach is not an efficient way to add rainbows to a game. Options to improve the speed are discussed in the section called, “Improvements and Other Uses.”

Given a screen-aligned quad, we need to find a view vector and sun ray vector at each pixel. We know that all sun rays are roughly parallel at the earth’s surface, so the direction of the sunlight is constant at each pixel. Use a vertex shader to calculate the view vector at each vertex of the screen aligned quad and let it interpolate across the triangles for each pixel. Then, use a pixel shader to calculate the texture coordinates to index into the Lee diagram lookup texture for each pixel on the screen.

To calculate the view vector per vertex, render the screen-aligned quad in homogeneous clip space (which is a common way to render a full screen quad). First, transform the vertex position by the inverse projection matrix to find the position of the quad in eye space. Once the position of the vertex is in eye space, the position of the eye in eye space is $\langle 0, 0, 0 \rangle$. Simply subtract the two points to get an unnormalized view vector at each vertex.

To compute the cosine of the angle of deviation using the dot product, normalize the eye vector. Let the unnormalized vector interpolate. To get the eye space sun ray vector, take the world-space sunlight vector and transform it by the view matrix.

Note: If you normalize the eye vector at the vertex level, the interpolated vector will not be normalized and you will have to renormalize it in the pixel shader.

Every pixel has an unnormalized view vector and a normalized sun light vector. To get a value in the range $[-1 \dots 1]$, normalize the view vector and compute its dot product with the sun light vector. For a rainbow, only use the range $[0 \dots 1]$. Set the texturing hardware to CLAMP the coordinate in the range $[0 \dots 1]$. Use the result of the dot product as the t texture coordinate. Specify the radius of water droplets s coordinate, then sample the Lee diagram lookup texture. This results in the color of a perfect rainbow at each screen pixel. (See the section called, “Source Code for Rainbow and Corona Effects” for a complete listing of vertex and pixel shaders.)

Moisture in the Air

The quality of a rainbow also depends on the amount of moisture in the sky. A thick sheet of moisture will have more water droplets to refract and reflect more color back to the viewer’s eye than a thin sheet of moisture will. More moisture produces a brighter rainbow and the less moisture produces a dimmer rainbow. Use a separate *moisture texture* when rendering the scene. Use the red color component to store the amount of moisture in the scene at each pixel. The rainbow color can be multiplied by this moisture factor (which ranges from $[0..1]$) to fade the rainbow in and out.

To calculate the moisture factor, render the fog amount to the texture, and then rendered the main skybox’s alpha component to represent the moisture/sun interaction in the skybox. Finally, render another skybox with a noise texture scrolling downward to simulate the far away sheets of rain and add a little dynamic motion to the rainbow. The better the model the interaction of sunlight and moisture in the atmosphere, the better the rainbow will look.

A model that encodes the water droplet radius into the moisture texture to simulate clouds and rain in the scene that may cause different types of rainbows at the same time is not effective. It does not work because, when the radius smoothly fades from one value to another, the rainbow colors curve unrealistically. To achieve this effect, it would be necessary to render the rainbow in multiple passes once for each possible water droplet radius.

Light Added to the Background

Rainbows add to the light in the background. To computer added rainbow light, set the source blend factor to one and use alpha blending, and set the destination blend factor to `invSrcColor`. With a true high-dynamic range lighting engine, both blend factors could be merged. However, a destination blend factor of `invSrcColor` handles over-saturation better and the resulting rainbow looks appealing.

Light Source Color

Finally, apply the rainbow color from the Lee diagram lookup texture by the sunlight color, since that light effects the possible rainbow color.

Render the Rainbow

Figure 6 shows the important parts of rendering the rainbow.

- ❑ The top left shows the plain rainbow color as rendered on a full-screen quad.
- ❑ The top middle shows the moisture texture. Objects in the world are black and the clouds in the sky are rendered in greyscale.
- ❑ The top right shows the moisture texture multiplied by the rainbow color.
- ❑ The bottom shows the 3D scene rendered normally without any rainbow
- ❑ The bottom middle, finally, combines of the 3D scene with the rainbow.



Figure 6. Integrate the Rainbow into a 3D Scene

Improvements and Other Uses

This section covers the following topics:

- ❑ Optimizations and improvements to the rainbow demonstration
- ❑ Additional atmospheric water and light-based phenomena
- ❑ Other applications

Optimizations and Improvements

Before using rainbows in a game, the technique described needs to become less expensive to execute. The dependant texture cannot be read in real-time for every single pixel of a 1600 x 1200 resolution screen. To use rainbows, fogbows, coronas and halos in a game, we need an optimized version.

One optimization is to design the rainbow texture beforehand, then render it into a cube-map texture. At runtime, simply render another skybox using this rainbow cube-map and combine the rainbow color with the moisture texture as described. As long as the water drop radius does not change, this base rainbow color cube-map need not change. When the sun direction changes, you can rotate the rainbow skybox so that the rainbow remains centered on the anti-solar point.

Another optimization is to render a half dome facing away from the sun with the top of the dome at the anti-solar point. Then, map the texture coordinates to a 1D slice of the Lee diagram lookup texture that corresponds to water drop radius. This technique uses normal texture wrapping to smear the 1D slice in a circle. The angle of deviation would not need to be computed at runtime and you could still change the radius of water droplets causing the rainbow. The dome would need to be well-tessellated in order to look good, but thanks to vertex processing, it would not create a bottleneck.

The technique presented to add rainbows to 3D scenes uses simplified model of a rainbow that has a constant intensity. In reality, the intensity varies greatly. MiePlot graphs the intensity data in addition to the color. To add greater realism, this high-dynamic range lighting data should be included in the model.

Another interesting improvement would be to port the Mie and Airy Theory simulation onto the GPU to generate the Lee diagram lookup texture. This would help it generate plots more quickly.

Other Water- and Light-Based Phenomena

The technique for rendering a rainbow can be used to render a corona around the sun, a halo around a moon, and some of the halos caused by ice crystals around the sun. The sample application included in this document implements both rainbows and sun coronas. Coronas are based on the same optical effects effect as rainbows. The difference is that, instead of being interested in the $[0..1]$ range of the dot product result, the corona is interested in the $[0..-1]$ range that is around the sun. Unfortunately Airy Theory could not be used to create a corona Lee diagram lookup texture because it does not take that effect into account. However, Mie does include that case. Use MiePlot to render a 1D lookup texture for the Corona.

Other Phenomena

Some odd results show up in the rainbow technique when the angle of deviation used is less than 90 degrees and the rainbow light bends around as it goes from one radius to another. The result looks surprisingly similar to the effect caused by abalone and oyster shells. The internal lining of these shells is made up of tiny crystals of calcite that reflect and refract light. Because the layers of calcite have varying thickness, you could correlate the thickness to water droplet radius thus using the same lookup textures to render various pearlescent materials.

Conclusion

Source Code for Rainbow and Corona Effects

```
/*
Rainbow.fx

rainbow simulation using precomputed light scattering and
interference.

*/

texture tRainbowLookup : DiffuseMap
<
  string name = "rainbow_Scatter_FakeWidet.tga";
  //I've manually tweaked this texture to widen the color bands,
  //not perfectly realistic, but looked better to me.
>;

texture tCoronaLookup : DiffuseMap
<
  string name = "rainbow_plot_i_vs_a_diffract_0_90_1024.tga";
>;

texture tMoisture : DiffuseMap
<
  string name = "env3_rainbow.bmp";
>;

float4x4 View : View;
float4x4 ProjInv: ProjectionInverse;

float3 LightVec : Direction
<
  string UIObject = "DirectionalLight";
  string Space = "World";
> = {1.0f, -1.0f, 1.0f};

half dropletRadius : Radius
<
  string UIType = "slider";
  float UIMin = 0.01;
  float UIMax = 0.99;
  float UIStep = 0.01;
  string UIName = "rainbow: droplet radius";
```

```

> = 0.81;

half  rainbowIntensity : Intensity
<
    string UIType = "slider";
    float UIMin = 0.0;
    float UIMax = 5.0;
    float UIStep = 0.1;
    string UIName = "rainbow: intensity";
> = 1.3;

struct VS_INPUT {
    float3 Position : POSITION;
    float4 vTexCoord : TEXCOORD0;
};

struct VS_OUTPUT {
    float4 vPosition : POSITION;
    half4 vTexCoord : TEXCOORD0; // quad texture coordinates
    float3 vEyeVec : TEXCOORD1; // eye vector
    float3 vLightVec: TEXCOORD2; // light vector
};

VS_OUTPUT VS_rainbow(VS_INPUT IN)
{
    VS_OUTPUT OUT;

    OUT.vTexCoord = IN.vTexCoord;
    // our input is a full screen quad in homogeneous-clip space
    OUT.vPosition = float4(IN.Position,1.0);

    //we need to unproject the position
    half4 tempPos = float4(IN.Position,1.0);
    tempPos = mul(tempPos, ProjInv);

    //while in homogenous clip space, the eye is at 0,0,0
    //vector from vertex to eye, no need to normalize here since we
    //will be normalizing in the pixel shader

    OUT.vEyeVec = float3(0.0, 0.0, 0.0 ) - tempPos;

    //transform light into eyespace
    float4 tempLightDir;
    tempLightDir = float4(-LightVec , 0.0);
    OUT.vLightVec = normalize(mul(tempLightDir, View ).xyz);

    return OUT;
}

sampler LookupMap = sampler_state
{
    Texture = <tRainbowLookup>;
}

```

```

    MinFilter = LINEAR;
    MagFilter = LINEAR;
    MipFilter = NONE;
    AddressU = CLAMP;
    AddressV = CLAMP;
};

sampler CoronaLookupMap = sampler_state
{
    Texture    = <tCoronaLookup>;
    MinFilter  = LINEAR;
    MagFilter  = LINEAR;
    MipFilter  = NONE;
    AddressU   = CLAMP;
    AddressV   = CLAMP;
};

sampler MoistureMap = sampler_state
{
    Texture    = <tMoisture>;
    MinFilter  = LINEAR;
    MagFilter  = LINEAR;
    MipFilter  = NONE;
    AddressU   = CLAMP;
    AddressV   = CLAMP;
};

void CalculateRainbowColor(VS_OUTPUT IN, out float d, out half4
scattered, out half4 moisture )
{
/*
    notes about rainbows

    -the lookuptexture should be blurred by the suns angular size 0.5
degrees.
    this should be baked into the texture

    -rainbow light blends additively to existing light in the scene.
    aka current scene color + rainbow color
    aka alpha blend, one, one

    -horizontal thickness of moisture,
    a thin sheet of rain will produce less bright rainbows than a
thick sheet
    aka rainbow color * water ammount, where water ammount ranges
from 0 to 1

    -rainbow light can be scattered and absorbed by other atmospheric
particles.
    aka simplified..rainbow color * light color

*/

```

```

        d = dot(
                    IN.vLightVec,                //this can be
normalized per vertex
                    normalize(IN.vEyeVec ) //this must be normalized
per pixel to prevent banding
                );

        //d will be clamped between 0 and 1 by the texture sampler
        // this gives up the dot product result in the range of [0 to 1]
        // that is to say, an angle of 0 to 90 degrees
        scattered = tex2D(LookupMap, float2( dropletRadius, d));
        moisture = tex2D(MoistureMap, IN.vTexCoord.xy);

    }

float4 PS_rainbowOnly(VS_OUTPUT IN) : COLOR
{
    //note: I can use a half for d here, since there are no
corruptions
    half d;
    half4 scattered;
    half4 moisture;
    CalculateRainbowColor(IN, d, scattered, moisture );
    return scattered*rainbowIntensity*moisture.x;
}

half4 PS_rainbowAndCorona(VS_OUTPUT IN) : COLOR
{
    /*
    Same as rainbow shader, but adds corona arround sun.
    */

    float d; //note: I use a float for d here, since a half corrupts
the corona
    half4 scattered;
    half4 moisture;

    CalculateRainbowColor(IN, d, scattered, moisture );

    //(1 + d) will be clamped between 0 and 1 by the texture sampler
    // this gives up the dot product result in the range of [-1 to
0]
    // that is to say, an angle of 90 to 180 degrees
    half4 coronaDiffracted = tex2D(CoronaLookupMap,
float2(dropletRadius, 1 + d));

    return (coronaDiffracted +
scattered)*rainbowIntensity*moisture.x;
}

```

technique Rainbow

```
{  
  
    pass P0  
    <  
        string geometry = "fullscreenquad";  
    >  
    {  
        // Shaders  
        VertexShader = compile vs_1_1 VS_rainbow();  
        PixelShader   = compile ps_2_0 PS_rainbowOnly();  
  
        // Render states:  
        lighting     = false;  
        zenable      = false;  
        alphablendenable = true;  
        srcblend      = one;  
        destblend    = invsrccolor;  
    }  
  
}  
  
technique RainbowAndCorona  
{  
  
    pass P0  
    <  
        string geometry = "fullscreenquad";  
    >  
    {  
        // Shaders  
        VertexShader = compile vs_1_1 VS_rainbow();  
        PixelShader   = compile ps_2_0 PS_rainbowAndCorona();  
  
        // Render states:  
        lighting     = false;  
        zenable      = false;  
        alphablendenable = true;  
        srcblend      = one;  
        destblend    = invsrccolor;  
    }  
  
}
```

Figure 7 shows a fogbow composed of large white bands. Fogbows are sometimes also called cloudbows.

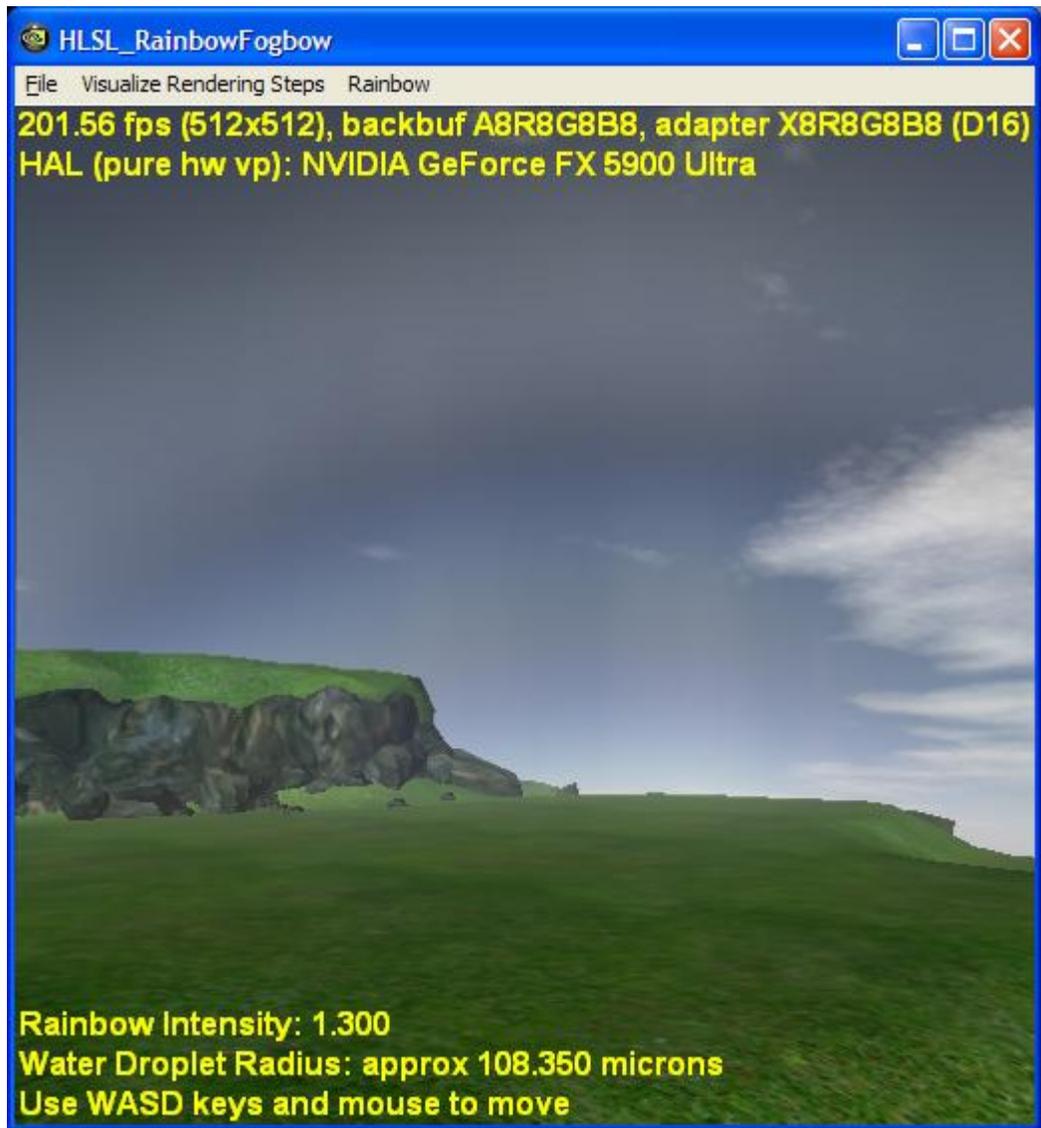


Figure 7. Simulation of Fogbow

The photograph in Figure 8 shows primary and secondary rainbows in waterfall mist in Iceland. Note the inversed colors of the secondary rainbow. Thanks to Orion Elenzil who provided the photograph.



Figure 8. Primary and Secondary Rainbows in Waterfall Mist

Bibliography

- Cowley, L. 2004. "Atmospheric Optics." Retrieved July 19, 2004.
<http://www.sundog.clara.co.uk/atoptics/phenom.htm>
- Dimasi, E., Jordan-Sweet, J. L., and Sarikaya M. 2000. "Orientation of Microcrystals in Abalone Shell near the Nacre-Prismatic Boundary." Retrieved on July 19, 2004.
<http://www.solids.bnl.gov/~dimasi/bones/abalone/>
- Laven, P. 2004. "The Optics of a Water Drop: Mie Scattering and the Debye Series"
Retrieved July 19, 2004. <http://www.philiplaven.com/index1.html>
- Lee, R. L. and Fraser, A. B. Fraser. 2001. *The Rainbow Bridge: Rainbows in Art Myth, and Science*.
University Park, Pennsylvania: Pennsylvania State University Press.
- Lee, Raymond L. 1998. "Mie Theory Airy Theory and the Natural Rainbow." *Applied Optics*,
37(9).
- Wynn, C. 2004. "Real-Time BRDF-Based Lighting Using Cube-Maps."
Retrieved on July 19, 2004. <http://developer.nvidia.com/object/brdfs.html>



Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2004 NVIDIA Corporation. All rights reserved



NVIDIA.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com