GPU Christmas Tree Rendering

Evan Hart
ehart@nvidia.com

February 2007

Document Change History

Version	Date	Responsible	Reason for Change
0.9	2/20/2007	Ehart	Betarelease

Beta Release

This is the beta version of the Christmas tree rendering whitepaper. A final version will be released in a later SDK with additional information and diagrams. Please visit the NVIDIA developer web page for updates:

http://developer.nvidia.com

Abstract

The xmas_tree sample in the NVIDIA OpenGL SDK demonstrates how to render a Christmas tree on a GeForce 8800 or later GPU. The sample brings together several techniques in order to effectively render the tree, including deferred shading and geometry shaders.

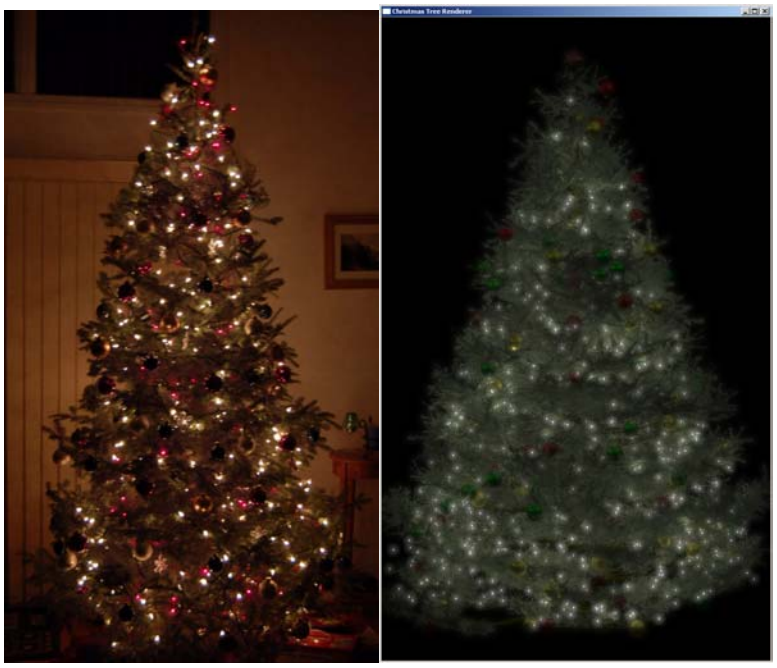

Figure 1. Real tree on left, with rendered tree on right.

Motivation

While GPU based rendering has achieved great advances over the past several years with the introduction of shaders and floating-point computations, many everyday objects are still not commonly rendered in real-time. This can be due to the difficulty of the rendering the object, or the lack of the need in a game or simulation environment. One such object that is ubiquitous for nearly one tenth of the year in a large fraction of the world is a Christmas tree. Both its complexity and uniqueness within a rendered environment make it an interesting object to pursue.

The complexity of rendering a good Christmas tree arises from many factors. First, one must render a convincing looking tree, since it will be the focus. This problem is encountered frequently, and many programmers have produced good solutions for GPU rendering of trees. One such solution is SpeedTree™, who generously allowed the use of their frasir fir texture for this sample. In this case, fractal generation of the branches was chosen to make the placement of lights and ornaments easier. The next order of complexity on a Christmas tree is the number of objects and varied types of materials found on it. A typical tree has dozens (often more than 100) ornaments with varying geometric and shading complexity. Reflective balls, satin balls, crystal snow flakes, velvet ribbons, wooden nutcrackers, and candy canes all require unique shaders. Finally, all these different materials are placed in a very complex illumination environment with no less than fifty lights and typically several hundred lights. This complex illumination requires high dynamic range techniques to handle it effectively. In the end, all these complexities come together to make the Christmas tree an interesting object to render.

More information on SpeedTree can be obtained from the website:

http://www.speedtree.com/

How Does It Work?

Unlike many other SDK samples, this sample is about using several effects together to reach a goal, rather than implementing a single effect. The key effects used in creating the tree are deferred shading to handle the massive number of lights, a set of tricks used in ornament rendering, and tone mapping.

Deferred Shading

As powerful as modern GPUs are, lighting every fragment of a Christmas tree with 1000 or more lights at thirty frames per second is too much to ask. Additionally, breaking the tree into smaller pieces that are only impacted a given set of lights is an impossible task due to the amount of overlap in light influence. This leads to deferred shading as the best option for illuminating the tree. Each light can be rendered as a quad that extends to the limits of the light's noticeable influence in the frame buffer. To prepare for the deferred shading, all objects rendered into the scene must store their position, normal, and material into the frame buffer. On the lighting passes, these will all be read to allow the light sources to compute illumination on a per-fragment basis.

The deferred shading process starts by rendering the tree geometry into a framebuffer object. The framebuffer object contains attachments for eye-space normals, eye-space positions, and materials. After the tree is rendered, the ornaments are rendered into the FBO as well. As the ornaments are all simple glass balls, no real geometry is needed. Instead, the balls are created by expanding a point in the geometry shader, and deriving the eye-space position, and normal in the fragment shader. The material value comes from a vertex attribute, allowing for many different ornament materials to all be rendered in the same draw call.

Once the FBO has been filled with the geometric and material information, the attachments are bound as textures, and a second FBO is bound as the render target. This second FBO has a single color buffer with a high dynamic range format, and it shares its depth buffer with the first FBO. By sharing the depth buffer, the lighting operations are accelerated by the depth cull hardware. The first light applied to the tree is environmental lighting; in this case, it is a directional light. This light is applied by rendering a full-screen quad with a lighting shader. Next, the lights local to the tree are rendered, with the total illumination being accumulated into the frame buffer via additive alpha blending. As mentioned previously, each light is represented as a single point, with expansion occurring in the shader. Next, reflections are applied to all the ornaments in the frame buffer. This technique is similar to the environmental lighting, except all pixels lacking the material properties that require reflection are discarded. Finally, the lights themselves are accumulated into the frame buffer. Again, each light starts as a point that is expanded in the shaders. The light then lands in the frame buffer as a hot spot that will be utilized in the tone mapping phase.

Tone Mapping

To display the high dynamic range rendering of the Christmas tree, the buffer must be tone mapped to bring the colors to a displayable range. The tone mapping algorithm used in this sample, while quite simple, is highly effective. The algorithm works by combining a blurred image with the original image, then applying spatially varying terms, adjusting for exposure, and finally transforming into sRGB space.

The tone mapping algorithm starts by downsampling the HDR buffer with a simple box filter. Conveniently, automatic mipmap generation does just this. Next, the reduced resolution HDR buffer undergoes a Gaussian blur with a large filter kernel (7x7 or larger). Acceptable performance is achieved in this stage of the rendering through a few tricks. First, for each level the image is reduced via box-filtering, the number of pixels to filter is reduced by a quarter. As the blurred image does not need much definition, using a one half resolution image is acceptable, and it reduces the pixels to process to one quarter. Second, the Gaussian blur is a separable operation, so it can be done in two stages by blurring horizontally, then blurring vertically. This cuts the number of values accessed in the convolution from the square of the width of the kernel to two times the width of the kernel. Finally, reducing the image size has also made the effective width of the convolution larger.

This allows the use of a narrow convolution kernel. While these are great optimizations, they are somewhat limited in this example. Since the lights on the tree are essentially points, the reduction in resolution can lead to some aliasing and loss of definition. A more continuous scene would likely be well suited with a lower resolution buffer for filtering.

Once the blurred version of the scene has been computed, a shader is used to combine the blurred image with the original and write the result to the frame buffer. The amount of blur included in the output image is controlled by a simple user editable parameter that is constant across the image. Next, the intensity of the fragment is scaled down by the vignette function. This darkens portions of the image closer to the edges of the scene, approximating an effect that occurs in vision. Next, the fragment is scaled by an adjustable exposure constant. Finally, the linear color space fragment is converted into sRGB space to more closely match the color reproduction of most monitors.

Implementation Details

The sample is broken down into code for creating and managing the trees, utility code, and code for managing the render loop and user input. The code for tree generation and geometry management is all contained in the tree.h and tree.cpp files. They define a class that fractally generates a tree based on several parameters, then places lights and ornaments on it randomly. The render loop and user controls are handled in xmas_tree.cpp. All the FBO creation, rendering, user interaction, and display code is in this file. Finally, there is a single file GPU_timer.h that implements a simple GPU based timing system with timer queries. These allow the application to monitor the performance of different portions of the rendering loop.

The source code is divided into two `cpp` files:

- `xmas_tree.cpp`: This file contains the main program and the rendering code.
- `tree.cpp`: This file contains the code for fractal tree generation.

Additionally, important code lies in two header files:

- `tree.h`: This file describes the interface for the fractal tree.
- `GPU_timer.h`: This file contains code for a simple GPU based stopwatch. This code is used to measure the relative performance of different aspects of the rendering without idling the hardware.

Finally, the demo uses shaders contained in several separate files:

- `basic_vertex.glsl`: This file contains the vertex shader used for rendering the tree geometry into the deferred shader FBO.
- `basic_fragment.glsl`: This file contains the fragment shader used for rendering the tree geometry into the deferred shading FBO.

- `def_shade_fragment.glsl`: This file contains the fragment shader used for applying the directional light while executing the deferred shading pass.
- `def_shade_vertex.glsl`: This file contains the vertex shader used for applying the directional light while executing the deferred shading pass.
- `def_shade_pt_fragment.glsl`: This file contains the fragment shader used for applying the point lights while executing the deferred shading pass. It computes a simple phong illumination model with the normals, positions, and materials from the deferred shading buffer.
- `def_shade_pt_vertex.glsl`: This file contains the vertex shader used for applying the point lights while executing the deferred shading pass.
- `def_shade_pt_geometry.glsl`: This file contains the geometry shader used for applying the point lights while executing the deferred shading pass. This shader expands the point to cover the volume of light influence, and clamps it to the near plane.
- `display_fragment.glsl`: This fragment shader handles the tone mapping operations to display the object in the viewport.
- `filter_fragment.glsl`: This fragment shader applies a 1-dimensional Gaussian blur. The direction and width of the blur is based on a uniforms.
- `light_vertex.glsl`: This is the vertex shader for the light points. It handles the attenuation and size of the lights.
- `light_geometry.glsl`: This is the geometry shader for the light points. It handles the expansion of the light points.
- `light_fragment.glsl`: This is the fragment shader for the light points. It shapes the light points into a circle.
- `ornament_vertex.glsl`: This is the vertex shader file for ornaments.
- `ornament_fragment.glsl`: This is the fragment shader file for ornaments. It uses discard to shape the ornament. It also computes the surface normal and position.
- `ornament_geometry.glsl`: This is the geometry shader file for ornaments. It expands the ornament point into a quad, and sets edge coordinates.
- `pp_vertex.glsl`: This is the post-processing vertex shader. It is used for drawing a fullscreen quad at the front of the depth range, with texture coordinates ranging from (0,0) to (1,1). It is used by the filtering, tone mapping, directional lighting, and reflection shaders.
- reflection_fragment.glsl: This shader is used to apply a cubic environment map to any pixels with a specular material. It biases the lod based on the specular exponent to approximate roughness.

Running the Sample

The sample should be run from the bin directory, so it can find its shaders and textures. The program can optionally take a filename as its only argument. This file will be parsed as a tree definition file. The tree definition file can be used to change the size and shape of the tree, along with the color and number of ornaments and lights.

Once launched, the Christmas tree sample will print several status messages to the command line, and then an overview of the available keyboard commands. In addition to the keyboard commands, the right mouse button brings up a menu of the commands, and a selector for the HDR format to use.

Navigation in the sample is controlled only by keyboard. The viewpoint can be moved in the Y plane by using the w, a, s, and d keys, and it can be moved up and down with the PageUp and PageDn keys. Additional information demonstrating the phases of the rendering, the time of the different phases, and the parameters used in the tone mapping stage can be displayed by using the i, t, and v keys respectively. The phase of rendering displayed in the intermediate panel is controlled by the x key. Finally, ornaments, branches, needles, lighting, and reflections can all be toggled by keys.

Controls

Control	**Action**
Right Mouse Button	Application menu
Escape	Quit the sample
W	Move viewer forward
A	Move viewer left
S	Move viewer back
D	Move viewer right
+	Increase exposure
-	Decrease exposure
[	Decrease sigma
]	Increase sigma
<	Decrease blur amount
>	Increase blur amount
;	Decrease blur buffer size
‘	Increase blur buffer size
T	Toggle the display of the timer graph
V	Toggle display of current HDR parameters
I	Toggle display of intermediate results
B	Toggle drawing the tree branches
F	Toggles drawing the needles
O	Toggle drawing the ornaments
R	Toggle drawing the reflections
L	Toggle drawing the lights

X	Change the current intermediate displayed
Space	Toggle continuous animation of the object

Performance

The sample runs at 150 frames per second on a GeForce 8800 GTX with a window size of 600 by 800 with the eye position zoomed out to allow the entire tree to be viewed. When the sample is maximized to cover a 1920x1200 display and the viewpoint is drawn so that the entire viewport is covered by the tree, the performance drops to 20 frames per second. The major bottlenecks for this sample are the rendering of the tree into the multiple render target FBO, and the lighting of the scene with the 1200 lights. The tree bottleneck is fill rate and bandwidth related, so choosing the smallest possible render target formats helps it. The lighting stage has a similar bottleneck, and using packed floats instead of 16-bit or 32-bit floats provides a significant performance advantage.

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