Rays of Light through Stained Glass Windows

Rajarshi Roy (rroy) and Emman Kabaghe (emmank)

Introduction:

We were inspired to pursue the rendering of stained glass windows after a visit to the Stanford memorial church. Since we were interested in learning rasterization shader coding and real-time rendering techniques, we based our project on rasterization based techniques in OpenGL as opposed to raytracing.

The image that originally inspired our scene was the scene in the image below [10].

In this image we see a colored stained glass window with visible colored rays emanating from it and projecting on the floor. Thus we pursued the implementation of real-time volumetric light scattering (or God rays) and projective texture mapping. We added on more techniques such as refraction for the windows and phong shading with light intensity falloff for the walls as the project progressed.

Core Graphics Technical Concepts:

Volumetric Light Scattering ("God Rays"):

To create the "God Rays" or Crepuscular rays, we implemented post process volumetric light scattering. In the real world, the "God Ray" effect is caused by the presence of sufficiently dense mixture of light scattering media in the environment such as gas molecules and dust particles. We based our implementation on the discussion of "God rays" in chapter 13 of GPU gems [1]. The core idea is that we loop through all pixels and create rays (light shafts) from the light source to the pixel all the while accounting for occluding objects in front of the light source. The day light scattering of light through the scene is governed by the following analytic equation:

$$L(s,\theta) = L_0 e^{-\beta_{ex} S} + \frac{1}{\beta_{ex}} E_{sun} B_{sc}(\theta)(1 - e^{-\beta_{ex} S})$$

Where:

L is the radiance
θ is the angle between the ray and the sun
L_0 is the initial radiance at point in space
β_{ex} is the extinction constant composed of light absorption and out-scattering properties
S is the is the distance traveled through the media
E_{sun} is the source illumination from the sun
B_{sc} is the angular scattering term composed of Rayleigh and Mie scattering properties

The first term calculates the amount of light absorbed from the point of emission to the viewpoint and the second term calculates the additive amount due to light scattering into the path of the view ray.

To apply the equation in post processing, sample steps along rays from pixels to the light source. The analytic equation is therefore simplified to the following

$$L(s,\theta) = exposure \times \sum_{i=0}^{n} decay^{i} \times weight \times \frac{L(s_i, \theta_i)}{n}$$

Where:

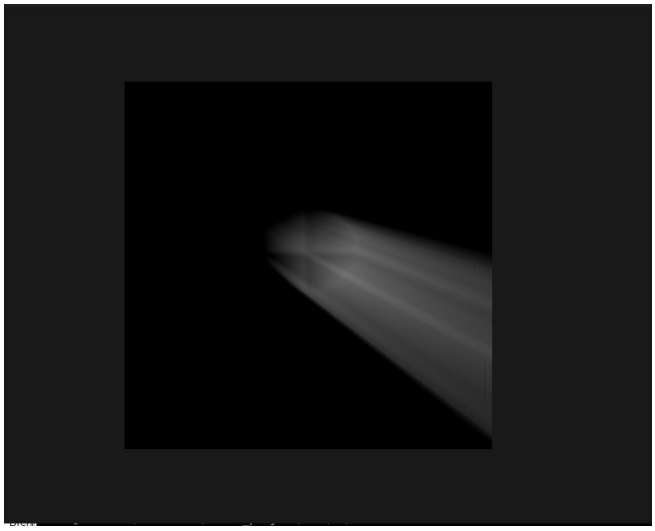
exposure controls the overall intensity of the rays
weight controls the intensity of each sample
$decay^i$ (for the range [0, 1]) dissipates each sample's contribution as the ray progresses away from the light source.

Using the above equation, we implemented our “God Ray” shader which does the following steps per pixel:

1. Calculate vector from pixel to light source in screen space
2. Divide vector by number of samples and scale by control factor
3. Get initial color of the pixel at that location
4. Setup up illumination decay factor
5. Step through the ray by looping over some number of samples
    a. Step along the ray to a new location
    b. Retrieve the sample at the new location
    c. Apply sample attenuation scale/decay factors
    d. Accumulate combined color
    e. Update exponential decay factor

6. Output final color

The “God Ray” shader is applied on a scene with everything else black (walls, roof, floor) apart from the stained glass windows. This way the colors of the light shafts are only affected by the colors on the stained glass windows. The result of the post process is then blended with a rendering of the regular scene. Below is an initial result from our God ray shader. In this image, we added our light source behind a circular window with cross bar occluders. Notice that the actual light source is located on the top left and the window is in the center causing the light rays to shoot through the window at that angle.



Generic Refraction through stained glass:

Refraction and associated caustics is generally better suited to raytracing algorithms. However, since the only refractive objects in our scene are the stained glass windows which are essentially planes in terms of geometry with surface distortions, the refraction calculation can be greatly simplified. The simplification is that the outgoing light ray from a point on the glass plane originates from a incoming light incident on the other side of the glass pane to a point that is offset from the outgoing point by a distance that is proportional to the surface normal. As a result, the fragment shader implementation of stained glass refractions samples the background texture behind the glass pane at an offset based on a normal map [2].

Below are results from our refraction shader implementation with two different normal maps [5]. The normal maps were textures with the normal direction encoded in the red and green channels.

Projective texture mapping:

In order to have the stained glass windows appear on our floor, we implemented projective texture mapping. We created projection and view matrices from the light perspective (as if the camera was at the light source position) and multiplied these new texture projection and view matrices by the position in the vertex shader to give us texture coordinates. In the fragment shader, when rendering the floor, we sampled the window textures using the generated texture coordinates from the vertex shader and applied the texture colors to the floor.

Note that the texture sampled for projective texture mapping was actually the window texture with a gaussian blur applied to emulate blurring due to the refractive nature of the glass and the soft-shadowing of window grilles since the sun is not purely a point source.

Phong shading with falloff:

The shader implemented for lighting the walls of the corridor is the standard phong fragment shader with the addition of intensity falloff from light source as per the inverse square law of light [9]. The image below on the left is rendered without falloff. The image below on the right is rendered with falloff and is more realistic.

Multipass rendering:

In order to use our post processing “God Ray” and “Refraction” shaders, we implemented multi-pass rendering. We had three passes to render to three different textures and a fourth pass to apply the “God Ray” and “Refraction” shaders on the textures from the first three passes. Here is a description of what each pass accomplished:

Pass 1: Render the stained glass windows (with coloured stained glass textures applied) and make everything else in the scene and occluding objects black (walls, floor, roof etc). Store the resulting image as a color texture attached to a framebuffer object.

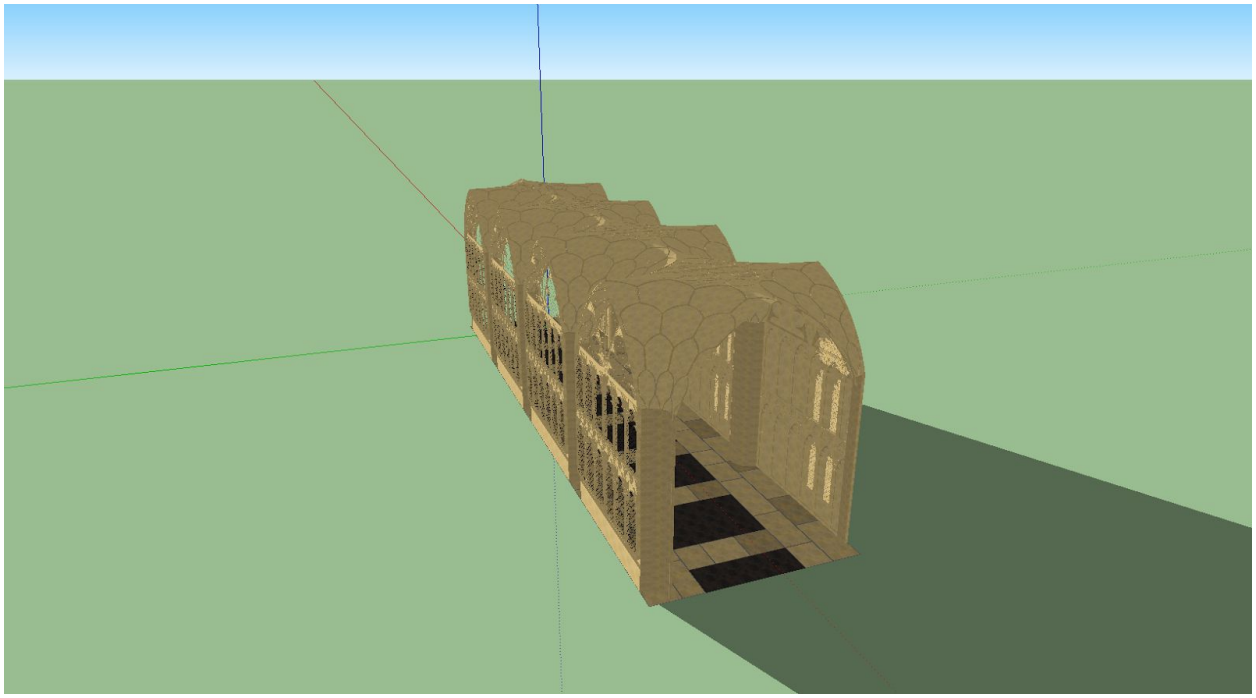
Pass 2: Render the windows (with various normal map textures applied to the windows) and and make everything else in the scene and occluding objects black (walls, floor, roof etc). Store the resulting image as a color texture attached to a framebuffer object.

Pass 3: Render the entire scene regularly (walls, floor, roof all shaded) but make all windows black. Store the resulting image as a color texture attached to a framebuffer object.

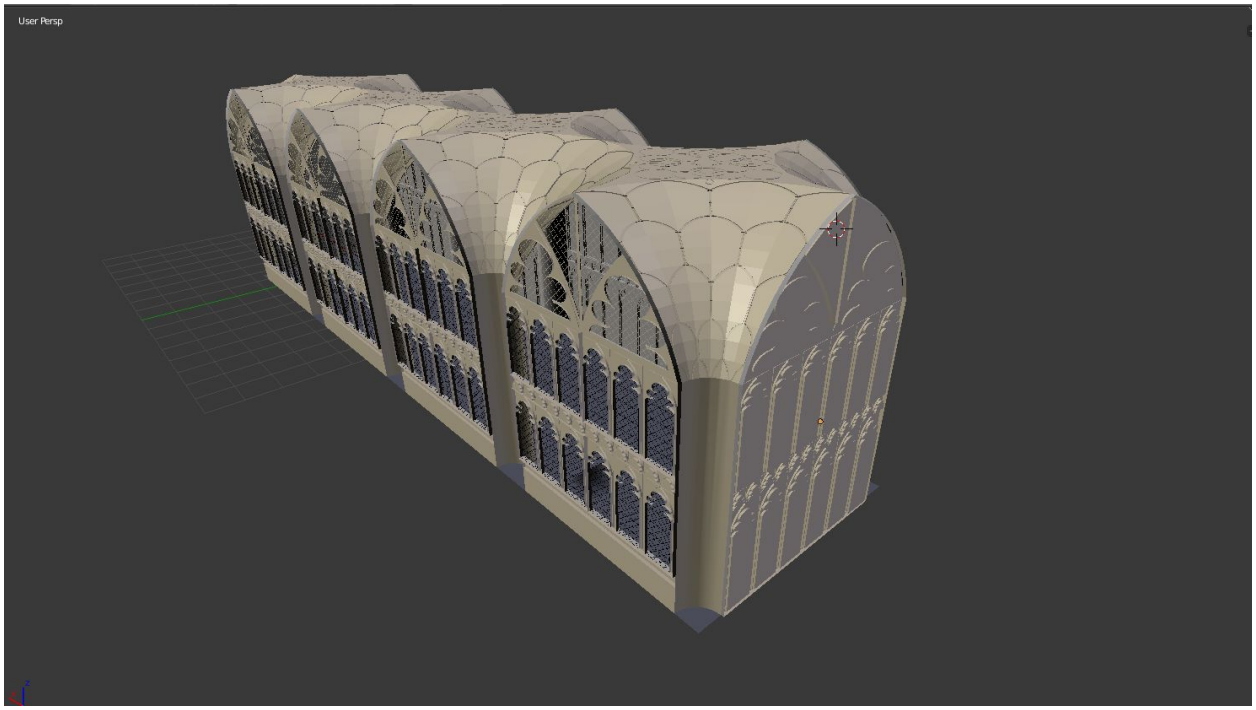
Pass 4: Render textures to a quad that spans the entire screen. First additively blend textures from pass 1 and and pass 2 in addition to a background texture (texture that has our sky + mountains background outside the windows). Then apply the refraction shader to the result of the additive blend. Next, we additively blend the result of the refraction shader processing with the texture from pass 3 to give us our entire scene with refraction through the windows. We then apply the “God ray” shader to the texture from pass 1 and finally blend this with the scene that has refraction to give us our final image with both refraction and God rays.

Scene Geometry:

The basic scene geometry was obtained from a sketchup model [4] which is shown below.



This model was exported using sketchup to the .obj file format. This model was imported into Blender and modified to add an end wall as shown below.



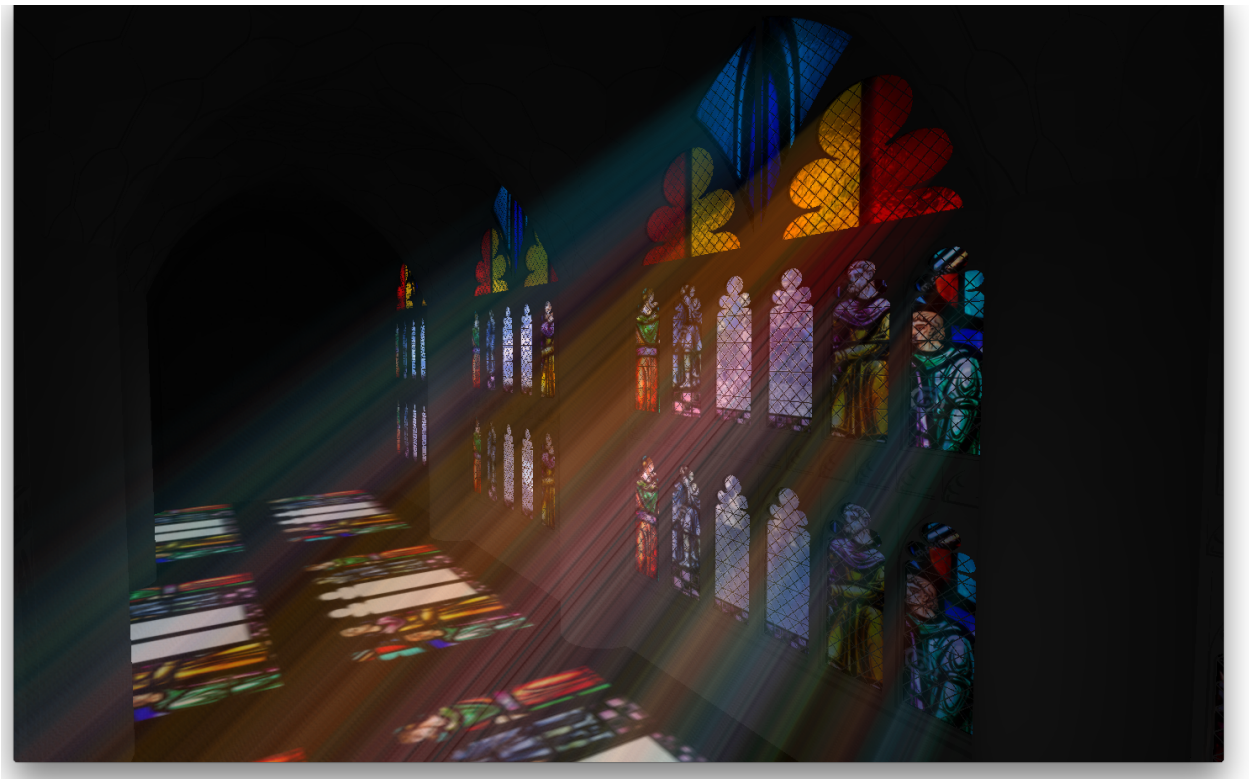
Blender was also used to triangulate faces, correct the vertex normals, and to add quads that would display the window textures. Then the model mesh was broken up into four sections (floor, walls, upper windows, lower windows) and exported as four separate .obj files. Then a C++ script using the STShape library [7] was used to read in each .obj file and print out a .h file containing a C++ array with all the vertex coordinates.
The array format is has three rows per triangle with each row containing a vertex's 9 attributes in the following format:
[position.x, position.y, position.z, normal.x, normal.y, normal.z, texture.s, texture.t, material]
The material attribute was assigned with the mapping {1:floor, 2:walls, 3:upper windows, 4:lower windows}. This generated .h file was then included and the array was loaded into the vertex buffer.

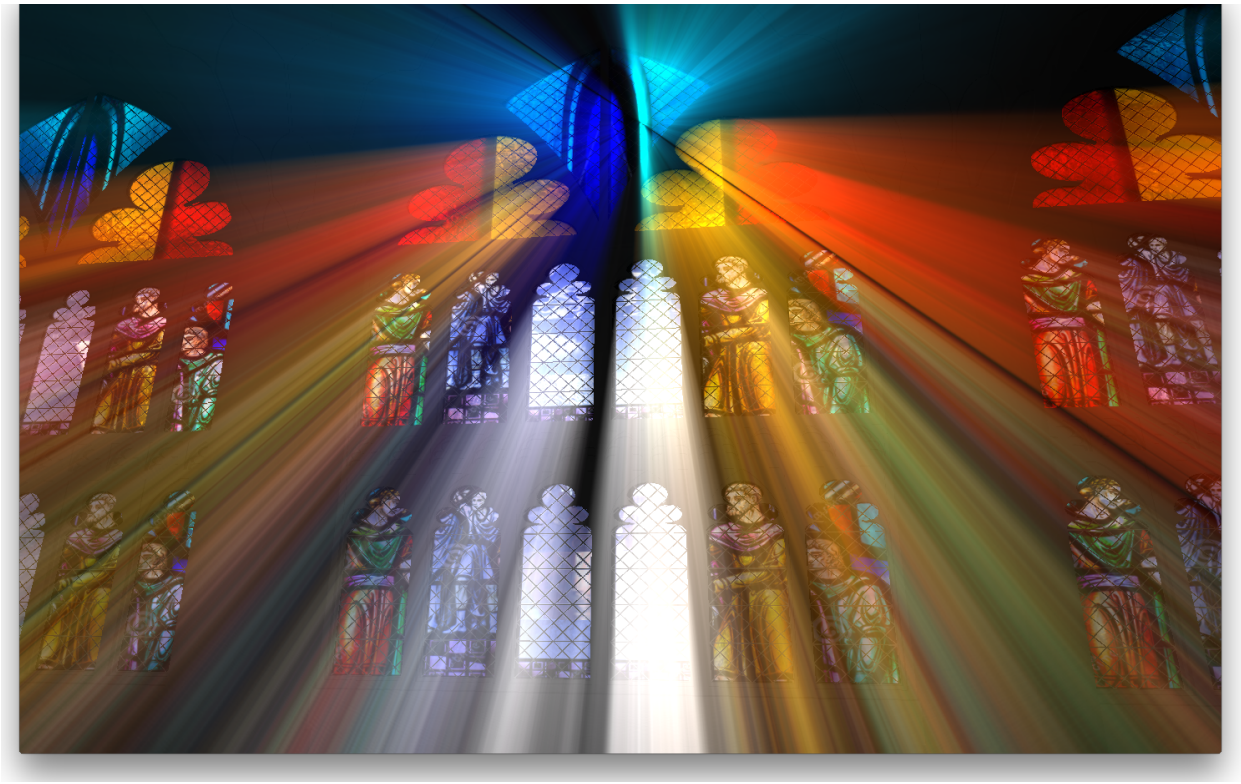
Textures and normal maps were edited in photoshop and loaded into OpenGl using the SOIL library [8]. A base image [6] of a stained glass window was manipulated using photoshop to generate all window stained glass textures. Two normal maps [5] were similarly manipulated to generate window stained glass normal map textures.

Final Results:

A demo video of exploring the scene is hosted here: https://youtu.be/5pHq6C7uAIs.

Below are four images of the scene from various viewpoints.





Challenges:

We faced a challenge with integrating the scene geometry with our render flow. The five primary portions of our scene geometry (floor, walls, upper window glass, lower window glass, background) had to be be rendered differently in different passes. Thus our shaders needed to detect which of these five materials is being processed at the vertex or fragment level. Plumbing this information from designing the scene in Blender to the shader level was a challenge.

Furthermore, a certain caveat in the volumetric light scattering algorithm [1] caused a debug challenge. If the camera direction is close to perpendicular to the vector from the camera to the light source, the light source will be projected onto the screen space near infinity. This breaks the ray-marching algorithm.

Division of Labour:

God rays: Emman implemented the shader. Rajarshi helped debug.
Refraction: Rajarshi implemented the shader.
Projective texture mapping: Emman implemented the shaders.
Phong shading with falloff: Rajarshi implemented the shader.
Multipass rendering: Emman implemented the flow. Both planned out the passes.
Scene geometry: Rajarshi did scene modelling, normal maps and textures, import into OpenGL and material attribute allocation code.

References:

[1] Volumetric light scattering: GPU gems chapter 13.
<http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html>
[2] Generic Refraction through Stained glass: GPU gems chapter 19.
<http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter19.html>
[3] Projective Texture Mapping: GPU gems chapter 9.
<http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html>
[4] Scene model: Sketchup 3D warehouse Hogwarts Corridor.
<https://3dwarehouse.sketchup.com/model.html?id=7213fe7a37b7d6bbd0656aee7124fe30>
[5] Glass normal maps: Filter Forge Artistic Glass Normal Map.
<https://www.filterforge.com/filters/5632-normal.html>
[6] Stained glass image: Dreamstime Stock Images. Church Stained Glass Window with Religious Scene.
<https://www.dreamstime.com/stock-images-church-stained-glass-window-religious-scene-image24525644>
[7] STShape library: CS148 HW5 Starter Code.
[8] Image texture library: SOIL (Simple OpenGL Image Library) library.
<http://www.lonesock.net/soil.html>
[9] Inverse Square Law of Light: PetaPixel. Understanding the Inverse-Square Law of Light.
<http://petapixel.com/2016/06/02/primer-inverse-square-law-light/>
[10] Rays of light through stained glass.
<https://www.123rf.com/stock-photo/stained_glass_church.html>