CS 580 – Final Project – Realistic Real-Time Skies

Shyam Guthikonda – guthikon@usc.edu
Da Vis Linder – dlinder@usc.edu
Ashok Meena – meena@usc.edu
Sung Yi – sungyi@usc.edu

Introduction

For any game or real-time simulation that has an outdoor area, the sky plays an integral part for conveying the mood and tone of the environment. Games utilize a variety of methods for implementing skies. One such method is the use of a sky box. The images mapped to the cube are static textures. The problem with this method is the static nature of the clouds – it is not very realistic. The clouds never move, and their color is always constant. Another method is the animated texture technique. This could be done with a sky plane or sky dome, and a single cloud texture mapping. The cloud texture can then be animated over time, which provides the effect of moving clouds. While this adds more realism to the simulation, it is far from perfect. The cloud patterns will repeat endlessly, which is something that does not happen in nature. Secondly, the clouds themselves are static. Cloud formation is not utilized, which detracts from the realism. Lastly, in order to change the density of clouds or the color, an entirely new texture must be created. This can be a time consuming task for an artist.

The method proposed in this paper will eliminate all of the aforementioned problems, through the use of a flexible real-time procedural cloud generation technique. We will show how the generated texture can be mapped to a sky dome, or combined with other sky rendering techniques. We will also add some extra features, such as lens flares, that serve to add to the overall scene realism.

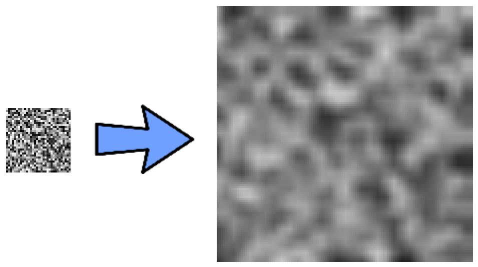
I. Procedural Clouds

The goal for the procedural cloud implementation was to do as much work as possible on the GPU through shaders, in order to get as much performance as possible. 60 Frames Per Second or greater was the target number throughout the development process.

Cloud Generation

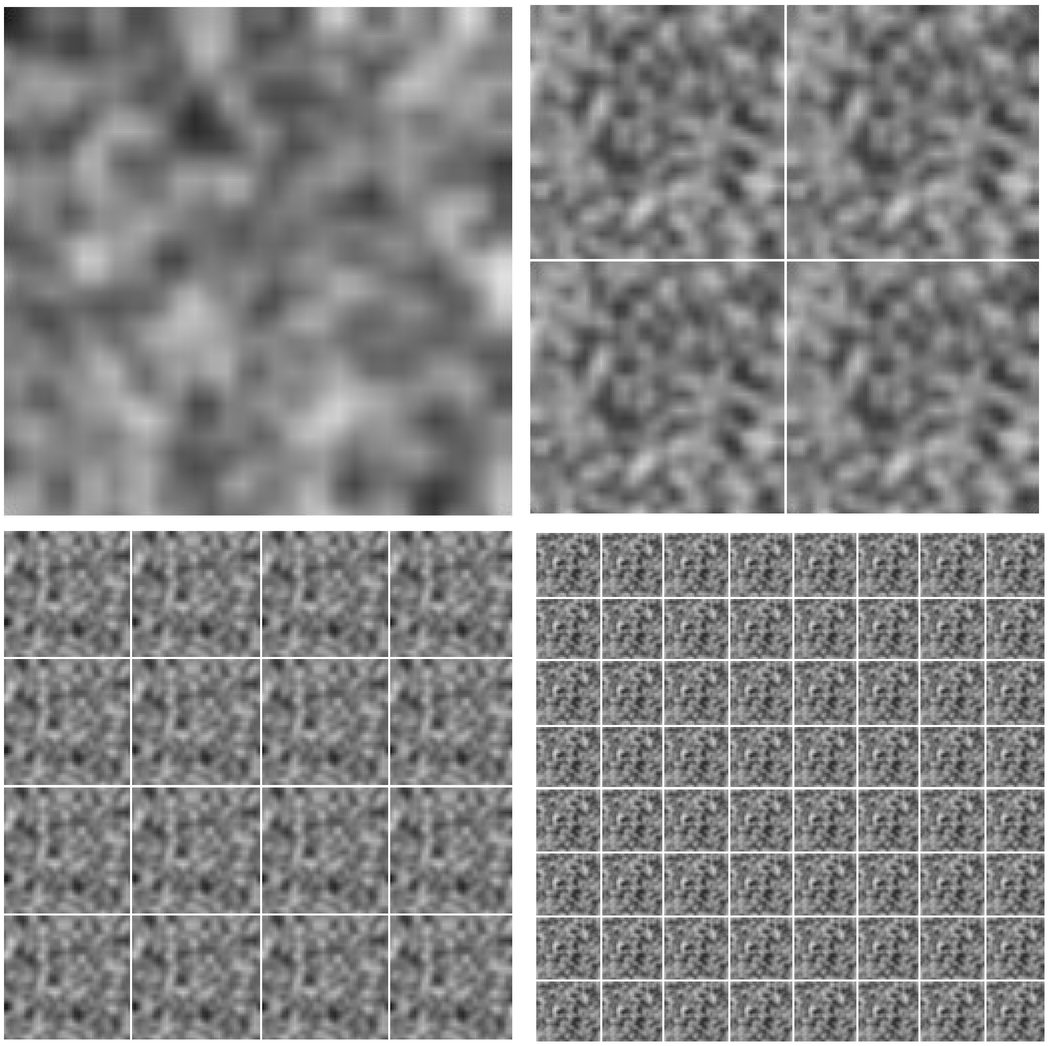
The basic technique for creating realistic-looking clouds involves creating 'octaves' or 'layers' of random noise, and blending these layers together. To create a single octave of noise, we use the standard C++ pseudo-random number generator, rand(). This PRNG is seeded with the current time at the start of the application. This could be replaced by a better PRNG technique, such as Perlin Noise. However, this adds computational expense, while not doing much in terms of realism. We found the rand() function of C++ to be sufficient.

For the first octave, the noise is generated and stored in a 32x32 texture. This noise is then re-sampled up to 256x256. Finally, a Gaussian filter is applied to smooth the texture even further. The following two images show the before and after textures:



This process is repeated to construct four octaves. The octaves are all generated with the same method, but there are a few parameters that change between them: the up-sampling factor, and the tiling factor.

Octave 1 is up-sampled to 256x256 and tiled by a factor of 1. Octave 2 is up-sampled to 128x128 and tiled by a factor of 2. Octave 3 is up-sampled to 64x64, and tiled by a factor of 4. Finally, octave 4 is left at 32x32, but tiled by a factor of 8. By utilizing this technique, it means that octave 1 supplies the largest details, and each successive octave adds finer and finer details. This allows the user to have control over the detail vs performance, as more and more octaves can be added to achieve even more realism. One of the referenced papers utilized eight octaves. For our purposes, four octaves was deemed sufficient. The following images illustrate the up-sampling and tiling process for octaves 1 (top left) to octave 4 (bottom right):



The 32x32 texture noise arrays are generated in the application with C++ code. These textures are then sent to the shader program, where the Gaussian filter is applied, and their UV coordinates are altered to achieve the tiling and re-sampling. The 4 octaves are then added together to create a final 256x256 noise texture.

Cloud Movement

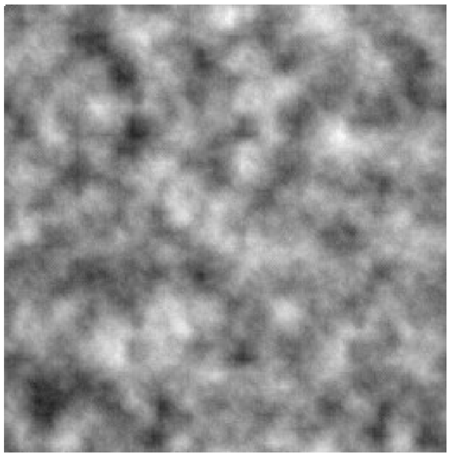
Utilizing the above technique, we arrive at realistic clouds. However, they are still static. We split cloud movement into two categories: scrolling and formation. Scrolling is the constant effect of clouds moving across the sky. This is easily implemented by applying some time-dependent offset to the final cloud UV coordinates of the texture. Cloud formation is a little more involved.

Cloud formation describes a cloud 'morphing' into another shape or form. This is necessary, because clouds are dynamic and not static. To achieve this effect, we maintain two texture arrays for each cloud octave. These arrays are called 'previous' and 'current'. We introduce a variable called *cloudFormationInterval*. This represents, in seconds, the time between our update intervals. When the time interval is reached, new cloud noise octaves are generated and stored in the 'current' octave, while the old octave is copied to the 'previous' octave. We then linearly interpolate, per-frame, between the 'previous' and 'current' octaves, based on *cloudFormationInterval* and the time that has elapsed since the previous update. This gives us a smooth cloud formation effect.

Added Realism

At this point, we have a procedurally generated, animated texture of random noise. The texture does not quite resemble clouds yet. We must first exponentiate the texture image, as shown below, which gives us a more natural-looking distribution of noise. In this formula, *cloudColor* is a value from 0-1, so we must multiply by 255 to achieve a more usable RGB range. *cloudSharpness* is a variable from 0-1 that allows us to alter the appearance of the clouds. “Sharp” clouds, where *cloudSharpness* = 1, produces very abruptly ending clouds, whereas “dull” clouds, where *cloudSharpness* = 0, produces very fuzzy clouds. By applying this function, we get a much more 'cloud-like' texture:

$$1-(1-cloudSharpness)^{cloudColor*255} \Rightarrow$$



There is still a single problem with the above cloud texture: there is no empty space. In a real sky, the clouds often only take over a portion of the sky. The blue color of the sky should be visible. To achieve this effect, we add in a variable called *cloudCover*, which ranges from 0-1 (0 = no clouds, 1 = full clouds). The following equation is then used:

$$max(cloudColor-(1-cloudCover),0) \Rightarrow$$

This is now the final cloud texture. To achieve a sky color, simply add the generated cloud texture directly to a sky color texture – they will combine appropriately. The texture is now ready to be mapped to any surface.

Unexpected and Unique Results

While the clouds look acceptable in their current state, they are still just a flat texture. Since they are not volumetric clouds, it is sometimes apparent that the clouds themselves are flat. This detracts from their realism.

One way to greatly increase the realism of this cloud technique is to utilize alpha. Instead of applying a solid sky color, we can apply full transparency to the areas where there are no clouds, and partial transparency to the areas where there are clouds (dependent on the cloud density). With alpha, we can create a parallax effect, where we have multiple layers of these clouds, each applied to different sky domes that are offset from each other. This creates cloud depth and layers (high clouds, middle clouds, and low clouds), something present in the real world.

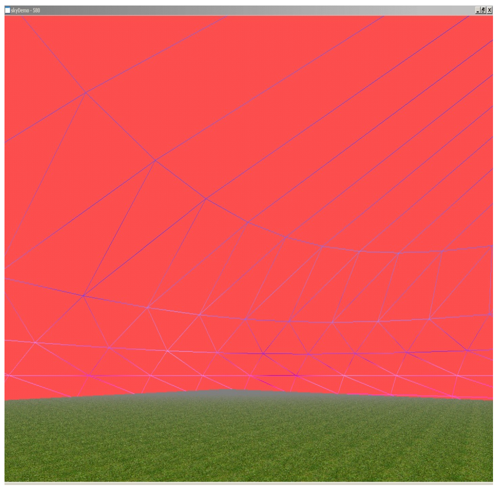
The clouds can also be used to compliment other existing techniques such as sky boxes. As mentioned previously, the primary drawback with sky boxes is that they are static. They can look very nice and realistic, however, because any arbitrary amount of color can be added by the artist to the static textures. Combining the color and detail of a sky box with an animated, dynamic layer of clouds can produce a very convincing sky:

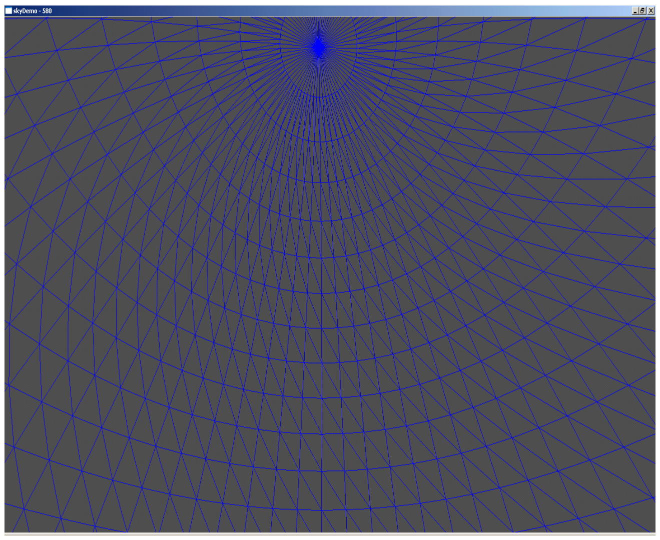
II. Sky Dome

In order to utilize the cloud layer, we must have a surface to map the texture onto. The two most commonly used methods are the sky plane and the sky dome. For clouds, a sky dome is often the best choice, especially for a wide-open scene.

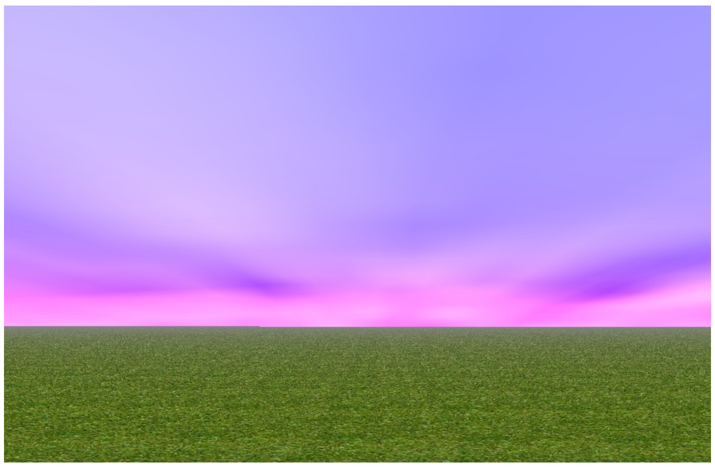
A sky dome is a semi-spherical surface which is used within graphics applications to depict the existence of a sky overhead though the use of simple or complex texturing methods. The dome resembles the natural curvature of the earth, which allows for a more realistic simulation, and is why it is preferred over the sky plane.

To create a sky dome, we calculate where the vertex points of a half sphere are located, based on the size of the radius of the circle and the mathematical properties of a sphere. To find each vertex point, we move around the sphere in ten degree increments. At each point, we determine the (x,y,z) coordinates. This process continues in a very systematic fashion. After the first triangle has been generated (3 points), each additional triangle is determined with only a single point, due to our use of the GL_TRIANGLE_STRIP triangle primitive OpenGL setting. Once we have completed the active circle, we move up ten degrees, and the process is repeated until we reach ninety degrees. Following this procedure, the following mesh is generated:





While calculating the vertex points, the corresponding u,v coordinates are generated using standard spherical texture mapping techniques. This allows us to map the rectangular cloud texture to the dome mesh with minimal visible distortion. The following results are achieved:



III. Lens Flare

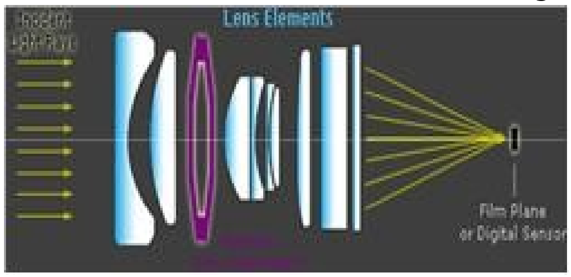
Although not part of the actual sky, lens flares are a commonly seen effect in scenes containing a sky and a sun. By adding this effect to our simulation, we attempt to achieve a higher level of realism.

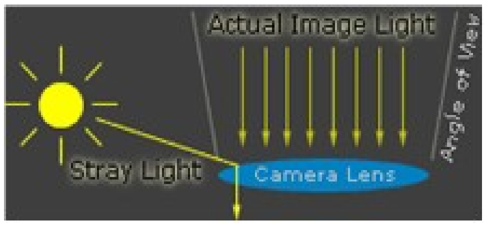
What It Is

Lens Flare is some polygonal- or circular-shaped light reflected, refracted and scattered internally through the multiple lens system, when some light source enters the lens and hits the camera's film or digital sensor.

How It Works

Although some lens has some kind of anti-reflective coating to minimize flare, multi-element lens usually cannot remove flare completely. Those polygonal shapes are from the lights that reflect off the lens aperture (diaphragm), shown on the top two figures. Usually flares are caused by very intense light sources, such as the Sun, artificial bright light, or a full moon. Even in cases where the light source is not in the photo (not within the angle of view of the camera), the stray light could reflect and travel the path to enter the film or sensor, as shown on the right figures.



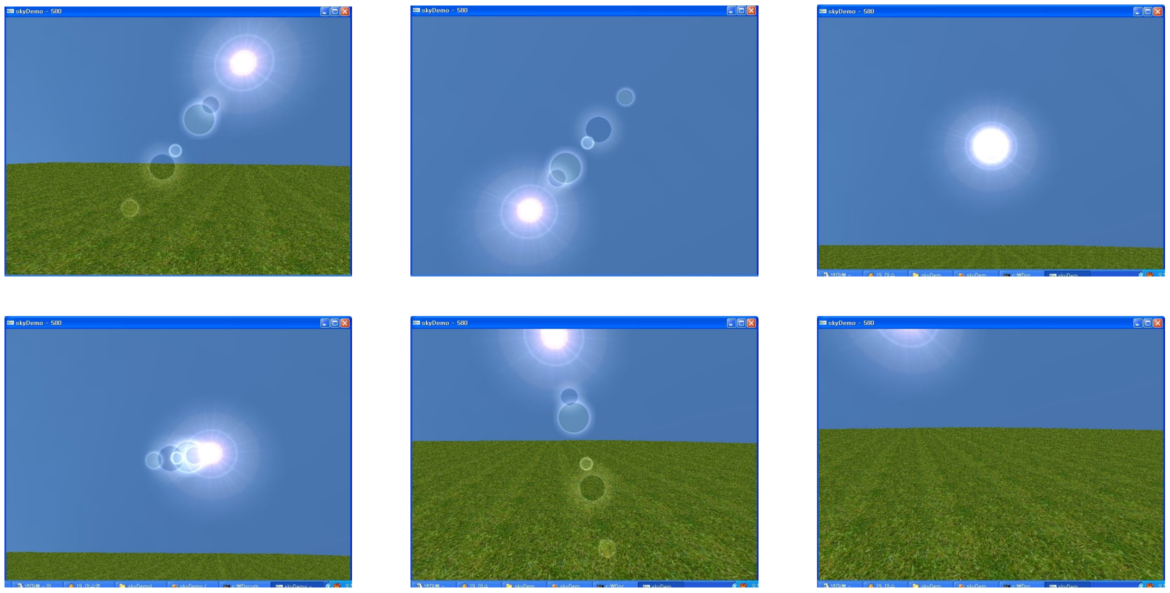


Assumptions

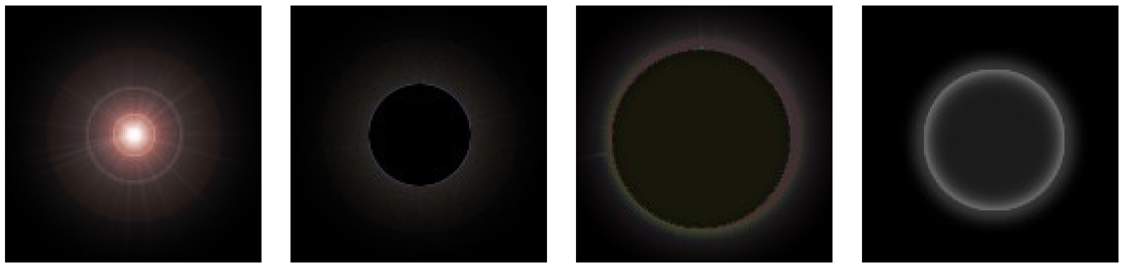
For simplicity, we've left out those lens flares with polygonal shapes and those appearing in the camera when the light source is not within the camera angle of view. In our project demo, the lens flare effect will only appear when the light source (the Sun) is within the camera view. Moreover, the flare will only include circular shapes, assuming we are using very naïve lens system without aperture. The final image should look something like the picture on the right.

What We Did and How We Did It

We put the light source (the Sun) in the sky and positioned each of the five lens flares along the line that connects the center of the camera and the light source. We first put the light source in the 3D space appropriately. Then, we retracted the 2D screen coordinate of the light source from the 3D world coordinate using OpenGL function called gluProject(). Each flares' distance from the light source may vary depending on implementations; we used the distance ratios as 0.5, 0.33333, -0.05, -0.25 and -0.75 respectively, from closest flare to the light source to farthest flare to the light source. After trying various ratios, this combination of ratios seemed to generate the most realistic scenery. As shown below, the light source and each flare blend in to the background scene transparently. This is achieved from OpenGL function called glBlendFunc(GL_SRC_ALPHA, GL_ONE), which blends in the source image to the destination background image, using the source's alpha values. As we move the camera, each flare’s distance to the light source varies. As seen in the last screen shot, the flares disappear as the light source disappears from the camera.



The textures that we used for the light source and each flare are shown below. The complete black parts of each texture are gone transparent and the remaining non-black parts blend in with the background color. The black part of each texture has very low alpha value (i.e. zero; absolutely transparent), and the remaining part has slightly higher alpha value (i.e. around 0.1 to 0.2). Below textures’ alpha value is preset, and they are available at http://www.gamedev.net/reference/articles/article813.asp.



The concepts and algorithms for lens flare were found via the above website (gamedev.net) as well. All major OpenGL function was found in the OpenGL documentation page in http://www.opengl.org.

Conclusion

Realistic skies are a welcomed addition to any outdoor game. Many games utilize other methods. These methods do not provide great flexibility for parameter adjustment, or they may put a heavy strain on artists for texture creation. While volumetric clouds are nice, they may be completely unnecessary for many games. Our implementation shows how a realistic sky can be procedurally generated in real-time, and how other methods can be combined with it to further enhance the realism.

References

Clouds
"Realistic Cloud Rendering on Modern GPUs", Game Programming Gems 5
"Generating Procedural Clouds Using 3D Hardware", Game Programming Gems 2
"Cloud Cover", http://freespace.virgin.net/hugo.elias/models/m_clouds.htm
"Virtual Terrain Project", http://www.vterrain.org/

Sky Dome
http://www.flipcode.com/articles/article_skydomes.shtml
http://www.spheregames.com/index.php?p=templates/pages/tutorials

Lens Flare
http://www.gamedev.net/reference/articles/article813.asp
http://www.cambridgeincolour.com/tutorials/lens-flare.htm
http://en.wikipedia.org/wiki/Lens_flare
http://upload.wikimedia.org/wikipedia/commons/thumb/3/3d/Lens_Flare.JPG/800px-Lens_Flare.JPG
http://www.ghostresearch.org/ghostpics/fake/Lensfl.jpg
http://www.opengl.org