Chapter 6

Real-Time Atmospheric Effects in Games

Carsten Wenzel[8]
Crytek GmbH

6.1 Motivation

Atmospheric effects, especially for outdoor scenes in games and other interactive applications, have always been subject to coarse approximations due to the computational expense inherent to their mathematical complexity. However, the ever increasing power of GPUs allows more sophisticated models to be implemented and rendered in real-time. This chapter will demonstrate several ways how developers can improve the level of realism and sense of immersion in their games and applications. The work presented here heavily takes advantage of research done by the graphics community in recent years and combines it with novel ideas developed within Crytek to realize implementations that efficiently map onto graphics hardware. In that context, integration issues into game production engines will be part of the discussion.

6.2 Scene depth based rendering

Scene depth based rendering can be described as a hybrid rendering approach borrowing the main idea from deferred shading [Hargreaves04], namely providing access to the depth of each pixel in the scene to be able to recover its original position in world space. This does not imply that deferred shading is a requirement. Rendering in *CryEngine2* for example still works in the traditional sense (i.e. forward shading) yet applies a lot of scene depth based rendering approaches in various scenarios as will be demonstrated in this chapter. What is done instead is decoupling of the actual shading of (opaque) pixels from later application of atmospheric effects, post processing, etc. This allows complex models to be applied while keeping the shading cost relatively moderate as features are implemented in separate shaders. This limits the chances of running into of current hardware shader limits and allows broader use of these effects as they can often be mapped to older hardware as well.

One reoccurring problem in the implementation of scene depth based rendering effects is handling of alpha transparent objects. Just like in deferred shading, you run into the problem of not actually knowing what a pixel's color / depth really is since generally only one color / depth pair is stored but pixel overdraw is usually greater

[8] carsten@crytek.de

than one and potentially unbounded (pathologic case but troublesome for hardware designers). Essentially it comes down to the problem of order independent transparency (OIT) for which to date no solution exists on consumer hardware. Possible approaches like A-Buffers are very memory intensive and not programmable. At this point, it is up to the developer to work around the absence of OIT depending on the effect to be implemented (see below).

To make per-pixel depth available for rendering purposes, there are several options. Ideally, depth is laid out in an early Z-Pass (encouraged by the IHVs in order to improve efficiency of early Z-Culling which can tremendously cut down subsequent pixel shading cost) filling the Z-Buffer. Since scene depth based rendering approaches don't modify scene depth, the Z-Buffer could be bound as a texture to gain access to a pixel's depth (though this would require a remapping of the Z-Buffer value fetched from post perspective space into eye space) or more conveniently it could be an input to a pixel shader automatically provided by the GPU. Due to limitations in current APIs, GPUs and shading models, this is unfortunately not possible for the sake of compatibility but should be a viable, memory saving option in the near future. For the time being, it is necessary to explicitly store scene depth in an additional texture. In *CryEngine2* this depth texture is stored as linear, normalized depth (0 at camera, 1 at far clipping plane) which will be important later when recovering the pixel's world space position. The format can be either floating point or packed RGBA8. On DirectX9 with no native packing instructions defined in the shader model, the use of RGBA8 although precision-wise comparable to floating point, is inferior in terms of rendering speed due to the cost of encoding / decoding depth. However, it might be an option on OpenGL where vendor specific packing instructions are available. An issue arises in conjunction with multisample anti-aliasing (MSAA). In this case, laying out depth requires rendering to a multi-sampled buffer. To be able to bind this as a texture, it needs to be resolved. Currently there's no way to control that down-sampling process. As a result depth values of object silhouettes will merge with the background producing incorrect depth values which will later be noticeable as seams.

Given the depth of a pixel, recovering its world space position is quite simple. A lot of deferred shading implementations transform the pixel's homogenous coordinates from post perspective space back into world space. This takes three instructions (three `dp4`'s or `mul` / `mad` assembly instructions) since we don't care about w (it's 1 anyway). However there's often a simpler way, especially when implementing effects via full screen quads. Knowing the camera's four corner points at the far clipping plane in world space, a full screen quad is set up using the distance of each of these points from the camera position as input texture coordinates. During rasterization, this texture coordinate contains the current direction vector from the camera to the far clipping plane. Scaling this vector by the linear, normalized pixel depth and adding the camera position yields the pixel's position in world space. This only takes one mad instruction and the direction vector comes in for free thanks to the rasterizer.

A lot of the techniques described in this chapter exploit the availability of scene depth at any stage after the actual scene geometry has been rendered to map pixels back to their source location in world space for various purposes.

6.3 Sky light rendering

Probably the most fundamental part of rendering outdoor scenes is a believable sky that changes over time. Several methods with varying quality and complexity have

been developed over the years. To accurately render sky light in *CryEngine2*, the model proposed in [Nishita93] was implemented as it enables rendering of great looking sunsets and provides several means for artist controllability over the output. Unfortunately, it is also one of the most computationally expensive models around. [O'Neil05] presents an implementation for the general case (flight simulator) which runs entirely on the GPU which required several simplifications and tradeoffs to make it work. Using these, it was possible to squeeze Nishita's model into the limits of current hardware but it came at the price of rendering artifacts (color gradients occasionally showed "layer-like" discontinuities).

The goal for *CryEngine2* was to get the best quality possible at reasonable runtime cost by trading in flexibility in camera movement using the following assumption. The viewer is always on the ground (zero height) which is fairly reasonable for any type of game where it's not needed to reach into upper atmosphere regions. This means the sky only ever needs to update when time changes and the update is completely independent of camera movement. The implementation used the acceleration structures suggested by the original paper. That is, we assume sun light comes in parallel and can hence build a 2D lookup table storing the angle between incoming sunlight and the zenith and the optical depth for the height of a given atmosphere layer (exponentially distributed according to characteristics of atmosphere). A mixed CPU / GPU rendering approach was chosen since solving the scattering integral involves executing a loop to compute intermediate scattering results for the intersections of the view ray with *n* atmosphere layers for each point sampled on the sky hemisphere. On the CPU, we solve the scattering integral for 128x64 sample points on the sky hemisphere using the current time of day, sunlight direction as well as Mie and Rayleigh scattering coefficients and store the result in a floating point texture. A full update of that texture is distributed over several fames to prevent any runtime impact on the engine. One distributed update usually takes 15 to 20 seconds. On CPU architectures providing a vectorized expression in their instruction set (e.g. consoles such as Xbox360 and PS3) the computation cost can be significantly reduced. This texture is used each frame by the GPU to render the sky. However, since the texture resolution would result in very blocky images, a bit of work is offloaded to the pixel shader to improve quality. By computing the phase function on the fly per pixel (i.e. not pre-baking it into the low resolution texture) and multiplying it to the scattering result for a given sample point on the sky hemisphere (a filtered lookup from the texture) it is possible to remove the blocky artifacts completely even around the sun where luminance for adjacent pixel varies very rapidly.

On GPUs with efficient dynamic branching it might be possible to move the current approach to solving the scattering integral completely over to GPU. Thanks to the 2D lookup table, the code is already quite compact. Initial test of porting the C++ version of the solver over to HLSL showed that, using loops, it would translate to approx. 200 shader instructions. Currently the loop executes $n = 32$ times. This number of exponentially distributed atmosphere layers was found sufficient to produces precise enough integration results to yield a good looking sky even for "stress tests" like rendering a sunset. Considering that approx. $32 * 200 = 6400$ instructions would have to be executed to solve the scattering integral for each sample point on the sky hemisphere, it seems necessary to distribute the update over several frames (e.g. consecutively rendering part of a subdivided quad into the texture updating individual parts). But still the update rate should be significantly shorter than it is right now.

6.4 Global volumetric fog

Even though Nishita's model indisputably produces nice results, it is still way too expensive to be applied to everything in a scene (i.e. to compute in and out scattering along view ray to the point in world space representing the current pixel in order to model aerial perspective).

Still, it was desired to provide an atmosphere model that can apply its effects on arbitrary objects with the scene. This section will first propose the solution implemented in *CryEngine2*. How to make it interact with the way sky light is computed will be described in the next section. It follows the derivation of an inexpensive formula to compute height/distance based fog with exponential falloff.

$$f\left((x,y,z)^T\right) = be^{-cz}$$

$$\vec{v}(t) = \vec{o} + t\vec{d}$$

$$\oint f(\vec{v}(t))dt = \int_0^1 f\left(\left(o_x + td_x, o_y + td_y, o_z + td_z\right)^T\right)\left\|\vec{d}\right\|dt$$

$$= \int_0^1 be^{-co_z - ctd_z}\sqrt{d_x{}^2 + d_y{}^2 + d_z{}^2}\,dt$$

$$= be^{-co_z}\sqrt{d_x{}^2 + d_y{}^2 + d_z{}^2}\int_0^1 e^{-ctd_z}dt$$

$$= be^{-co_z}\sqrt{d_x{}^2 + d_y{}^2 + d_z{}^2}\left[-\frac{e^{-ctd_z}}{cd_z}\right]_0^1$$

$$= be^{-co_z}\sqrt{d_x{}^2 + d_y{}^2 + d_z{}^2}\left[-\frac{e^{-cd_z}}{cd_z} + \frac{1}{cd_z}\right]$$

$$= be^{-co_z}\sqrt{d_x{}^2 + d_y{}^2 + d_z{}^2}\left[\frac{1 - e^{-cd_z}}{cd_z}\right]$$

$$F(\vec{v}(t)) = e^{-\int f(\vec{v}(t))dt}$$

f - fog density distribution function
b - global density
c - height falloff
v - view ray from camera (o) to world space pos of pixel (o+d), t=1
F - fog density along v

This translates to the following piece of HLSL code (Listing 1). Care must be taken in case the view ray looks precisely horizontal into the world (as d_z is zero in that case).

```
float ComputeVolumetricFog( in float3 worldPos, in float3
cameraToWorldPos)
{
  // NOTE:
  // cVolFogHeightDensityAtViewer = exp( -cHeightFalloff *
  //                                     vfViewPos.z );
  float fogInt = length( cameraToWorldPos ) *
                 cVolFogHeightDensityAtViewer;

  const float cSlopeThreshold = 0.01;
  if( abs( cameraToWorldPos.z ) > cSlopeThreshold )
  {
     float t = cHeightFalloff * cameraToWorldPos.z;
     fogInt *= ( 1.0 - exp( -t ) ) / t;
  }

  return exp( -cGlobalDensity * fogInt );
}
```

Listing 1. *Computing volumetric fog in a DirectX 9.0c HLSL shader code block*

The "`if`" condition which translates into a `cmp` instruction after compiling the shader code prevents floating point specials from being propagated which would otherwise wreak havoc at later stages (tone mapping, post processing, etc). The code translates into 18 instructions using the current version of the HLSL compiler and shader model 2.0 as the target.

Calling this function returns a value between zero and one which can be used to blend in fog. For all opaque scene geometry, this model can be applied after the opaque geometry rendering pass by simply drawing a screen size quad setting up texture coordinates as described in Section 7.2 and invoking that function per pixel. What remains to be seen is how to calculate a fog color that is a good match to blend with the sky. This will be topic of the next section.

6.5 Combining sky light and global volumetric fog

We create a slight problem by implementing a separate model for sky light and global volumetric fog. Now we have two models partially solving the same problem: How to render atmospheric fog / haze. The question is whether it is possible for these two models be combined to work together. We certainly wish to achieve halos around the sun when setting up hazy atmosphere conditions, realize nice color gradients to get a feeling of depth in the scene, be able to see aerial perspective (i.e. the color gradient of a mountain in the distance which is partiality in fog should automatically correlate with the colors of the sky for a given time and atmospheric settings). Nishita's model would allow rendering that but is too expensive to be used in the general case. The global volumetric fog model presented in the previous section is suitable for real-time rendering but far more restrictive.

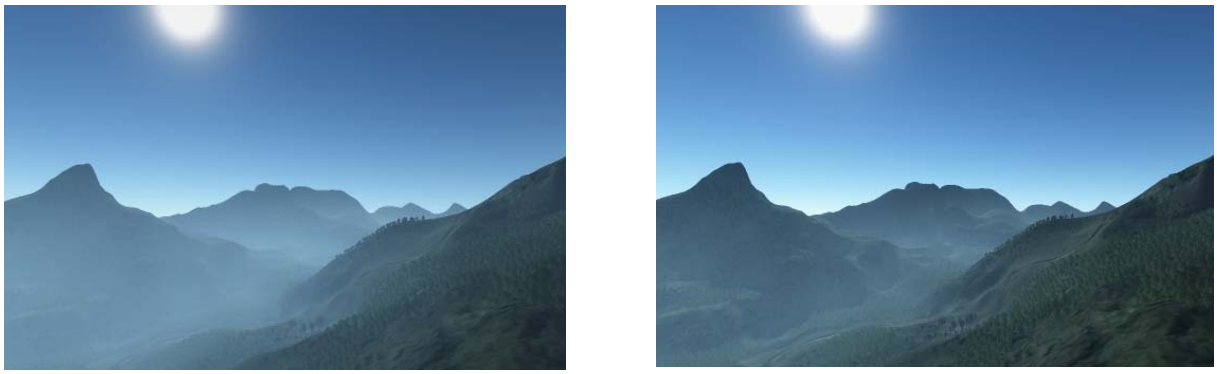
To make the two models cooperate, we need a way to determine a fog color that can be used with accompanied volumetric fog value to blend scene geometry nicely into the sky. For that purpose, Nishita's model was enhanced slightly to allow a low cost per-pixel computation of a fog color matching the sky's color at the horizon for a given direction. To do this, all samples taken during the sky texture update for directions resembling the horizon are additionally averaged and stored for later use in

a pixel shader (using all of the horizon samples to produce a better matching fog color seems tempting but it was found that the difference to the average of all horizon samples is barely noticeable and doesn't justify the additional overhead in shading computations). When rendering the fog, the same code that calculated the final sky color can be used to gain a per-pixel fog color. The phase function result is computed as before but instead of accessing the low resolution texture containing the scattering results, we use the average of the horizon samples calculated on the CPU. Now using the volumetric fog value computed for a given pixel the color stored in the frame buffer can be blended against the fog color just determined. This may not be physically correct but gives pleasing results at very little computational cost.

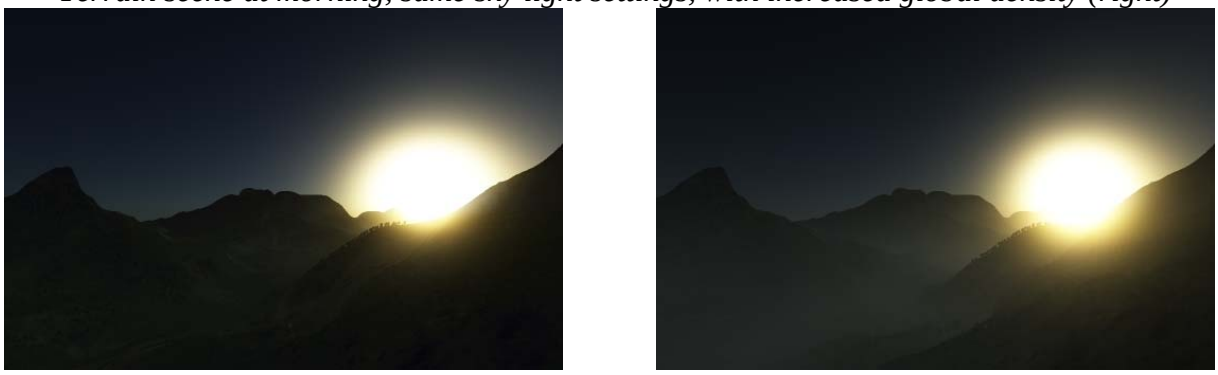
6.6 Locally refined fog via volumes

For game design purposes, it is often necessary to locally hide or disguise certain areas in the game world and global fog is not really suited for that kind of purpose. Instead fog volumes come into play. The implementation goal was to be able to apply the same kind of fog model as described in Section 7.4 to a locally refined area. The model was slightly enhanced to allow the fog gradient to be arbitrarily oriented. We support two types of fog volume primitives, namely ellipsoids and boxes.

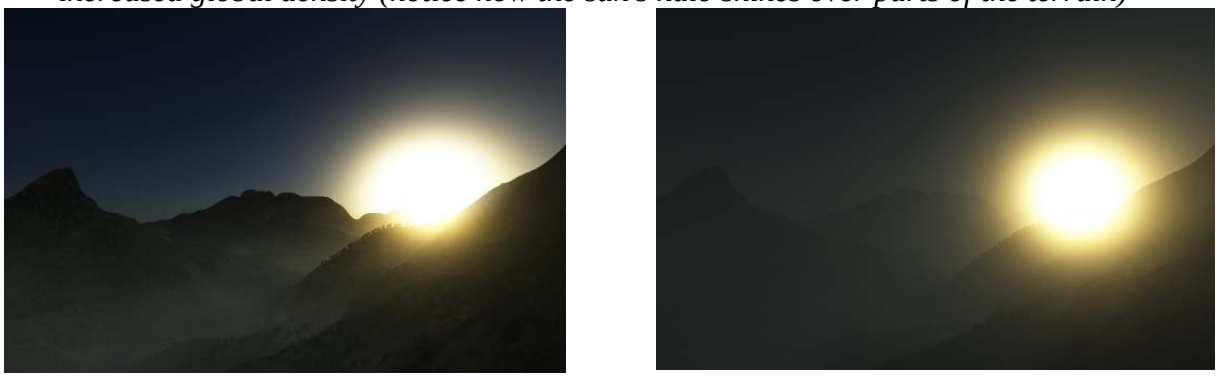
As can be seen in the derivation of the formula for global volumetric fog, a start and end point within the fog volume are needed to solve the fog integral. (Note that the global fog can be thought of as an infinite volume where the start point represents the camera position and end point is equal to the world space position of current pixel.) To determine these two points, we actually render the bounding hull geometry of the fog volume, i.e. a box whose front faces are rendered with Z-Test enabled as long as the camera is outside the fog volume and whose back faces are rendered with Z-Test disabled once the camera is inside the volume. This way it is possible to trace a ray for each pixel to find out if and where the view ray intersects the fog volume. Ray tracing for both primitive types happens in object space for simplicity and efficiency (i.e. transforming the view ray into object space and checking against either a unit sphere or unit cube and transforming the results back into world space). If no intersection occurs, that pixel is discarded via HLSL's `clip` instruction. Doing that has the additional side effect of simplifying the code for shader models and/or hardware not supporting (efficient) branching (i.e. it compiles to less instructions) since it avoids the need to consider "`if / else`" cases for which otherwise all branches would have to be executed and a final result picked. If an intersection occurs, an additional check using the pixel's depth value is necessary to determine if scene geometry is hit before exiting the fog volume. In that case this point overrides the end point. Also, in case we're inside the volume, we need to ensure that the start point is always the camera point. With the start and end point known, these can be plugged into the volumetric fog formula earlier to compute a fog value per pixel.



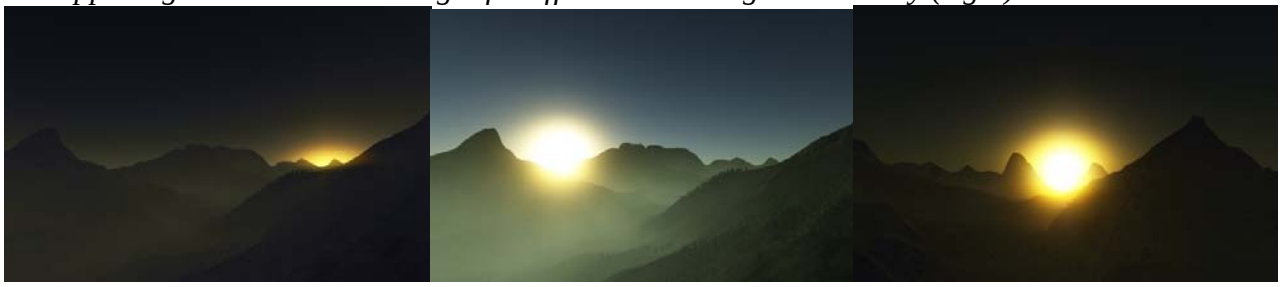
Terrain scene at morning, same sky light settings, with increased global density (right)



Same terrain scene at sun set observed from a different position, (left) default settings for sky light and global volumetric fog, (right) based on settings for upper left but increased global density (notice how the sun's halo shines over parts of the terrain)



Based on settings for upper right but stronger height falloff (left), based on settings for upper right but decreased height falloff and increase global density (right)



Various sunset shots with different setting for global density, height falloff, Mie- and Rayleigh scattering

Figure 1. *Atmospheric scattering effects.*

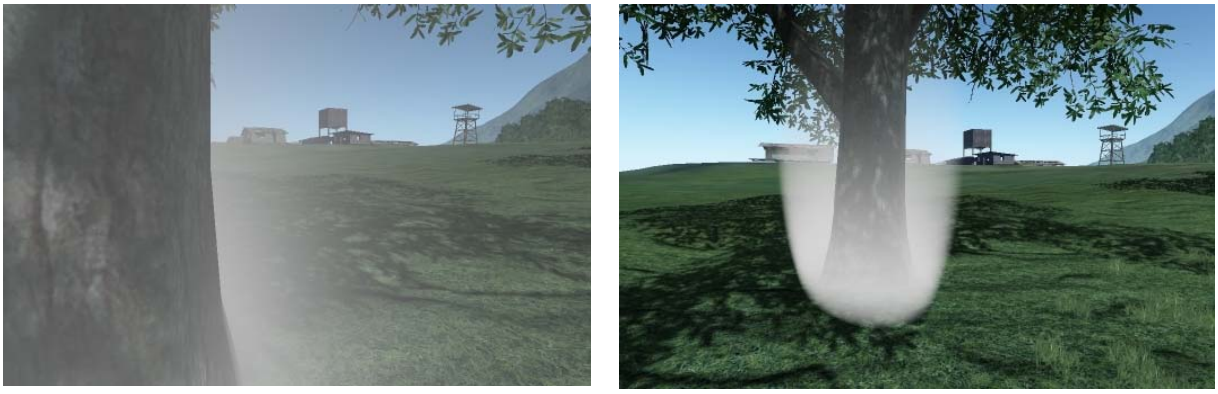


***Figure 2**. GPU ray traced fog volume (ellipsoid), observed from inside (left) and outside (right)*

Other approaches to implementing fog volumes were also considered. Polygonal fog volumes seemed like a good idea at first (depth of all back faces is accumulated and subtracted from depth of all front faces to get length traveled through volume along pixel direction). They also don't suffer from clipping artifacts at the near clip plane since a depth of zero doesn't have any impact while accumulating front / back face depth. Their advantage over the fog volume primitives described above is that they can be arbitrarily complex and convex. In order to do efficient ray tracing one is currently still stuck with rather simplistic primitives like ellipsoids and boxes. However, polygonal fog volumes also exhibit a few disadvantages which have to be taken into account and outweigh their advantages (at least for the purpose of *CryEngine2*). First they really allow depth based fog only. Currently they need to be rendered in two passes to accumulate back face depth and subtract front face depth. Moreover to do so they need additional texture storage and floating point blending support which is kind of prohibitive when fog volume support is required for older hardware as well. Implementations of polygonal fog volumes exist that use clever bit-twiddling to make them work with standard RGBA8 textures but then it is necessary volumes do not exceed a certain depth complexity or size to prevent overflow which creates another limitation and kind of defeats their usefulness in the first place.

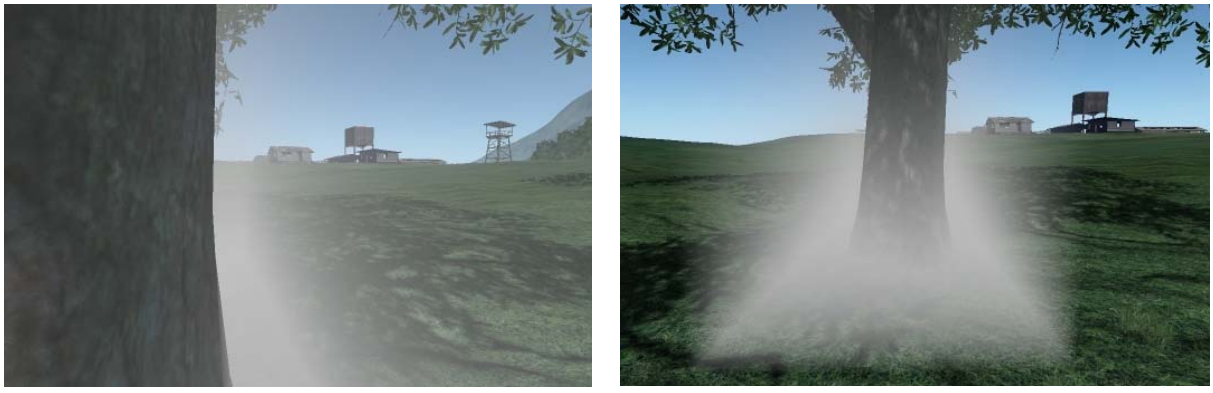


***Figure 3**. GPU ray traced fog volume (box), observed from inside (left) and outside (right)*

6.7 Fogging alpha transparent objects

As mentioned in the introduction, scene depth based rendering approaches often cause problems with alpha transparent objects, in this case the global/local volumetric fog model. Since currently there's no feasible hardware solution available to tackle this problem, it is up to the developer to find suitable workarounds, as will be shown in this section.

6.7.1 Global fog

Global volumetric fog for alpha transparent objects is computed per vertex. Care needs to be taken to properly blend the fogged transparent object into the frame buffer already containing the fogged opaque scene. It proves to be very useful that the entire fog computation (fog density as well as fog color) is entirely based on math instructions and doesn't require lookup tables of any sort. The use of textures would be prohibitive as lower shader models don't allow texture lookups in the vertex shader but need to be supported for compatibility reasons.

6.7.2 Fog volumes

Applying fog volumes on alpha transparent objects is more complicated. Currently, the contribution of fog volumes on alpha transparent objects is computed per object. This appears to be the worst approximation of all but it was found that if designers know about the constraints implied they can work around it. Usually they need to make sure alpha transparent objects don't become too big. Also, the gradient of fog volumes should be rather soft as sharp gradients make the aliasing problem more obvious (i.e. one sample per object). The ray tracing code done per pixel on the GPU can be easily translated back to C++ and invoked for each alpha transparent objected pushed into the pipeline. A hierarchical structure storing the fog volumes is beneficial to reduce the number of ray/volume traces as much as possible. To compute the overall contribution of all fog volumes (partially) in front of an alpha transparent object the ray tracing results are weighted to a single contribution value in back to front order (i.e. farthest fog volumes gets weighted in first). Another approach that was investigated involved building up a volume texture containing fog volume contributions for sample points around or in front of the camera (i.e. world space or camera space respectively). Both a uniform and non-uniform distribution of sampling points was tried but aliasing was just too bad to deem this approach useful.

One potential extension to the computation of the overall contribution of fog volumes on alpha transparent objects is to compute values for all corners of an object's bounding box and in the vertex shader lerp between them based on the vertex' relative position.

6.8 Soft particles

Particles are commonly used to render various natural phenomena like fire, smoke, clouds, etc. Unfortunately, at least on consumer hardware available today (even with MSAA enabled), they usually suffer from clipping artifacts at their intersection with opaque scene geometry. By having the depth of all opaque objects in the world laid

out and accessible in the pixel shader, it is possible to tweak a particle's alpha value per pixel to remove jaggy artifacts as shown in the figure below.

Particles drawn as hard billboards *Soft particles*

Figure 4. *Billboard and soft particle rendering comparison*

Each particle is treated as a screen aligned volume with a certain size. For each pixel, the particle shader determines how much the view ray travels through the particle volume until it hits opaque scene geometry. Dividing that value by the particle volume size and clamping the result to [0, 1] yields a relative percentage of how much opaque scene geometry penetrates the particle volume for the given pixel. It can be multiplied with the original alpha value computed for the current particle pixel to softly fade out the particle wherever it's getting close to touching opaque geometry.

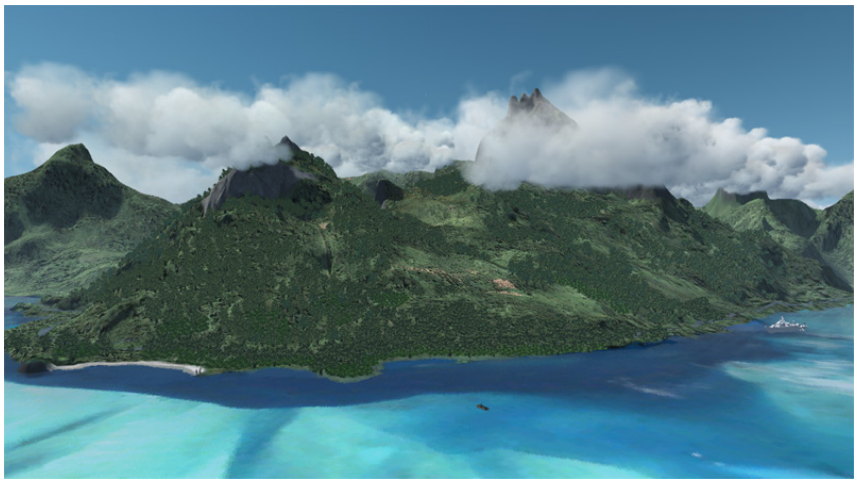
6.9 Other effects benefiting from per pixel depth access

6.9.1 Clouds

The cloud rendering subsystem in *CryEngine2* is based on [Wang03]. Shading is gradient-based and scene depth used to implement soft clipping, especially with terrain (e.g. rain clouds around mountains) and both the near and far clipping plane. It borrows the main idea from the soft particle implementation. Additionally back lighting with respect to the sun was added to achieve the effect of glowing edges when looking at clouds partially covering the sun.

Figure 5*. Backlighting of clouds, showing different values for backlighting threshold (with respect to cloud alpha), and backlighting softness*

Cloud shadows are cast in single full screen pass using the scene depth to recover world space position to be able to transform into shadow map space.



Island scene without cloud shadows

Same scene with cloud shadows. Notice how it breaks up the regularity of shading giving a more natural look

Same scene with cloud shadows viewed from a different position

***Figure 6**. Cloud shadow rendering*

6.9.2 Volumetric lightning

Modeling volumetric lightning is similar to the global volumetric fog model postulated earlier. Only this time it is light emitted from a point falling off radially. The attenuation function needs to be chosen carefully in order to be able to integrate it in a closed form.

$$f\left((x,y,z)^T\right)=\frac{i}{1+a\cdot\left\|\vec{l}-(x,y,z)^T\right\|^2}$$

$$\vec{v}(t)=\vec{o}+t\vec{d}$$

$$\oint f(\vec{v}(t))dt=\int_0^1 f\left(\left(o_x+td_x,o_y+td_y,o_z+td_z\right)^T\right)\left\|\vec{d}\right\|dt$$

$$=\int_0^1\left(\frac{i}{1+a\left\|\vec{l}-\left(\vec{o}+t\vec{d}\right)\right\|^2}\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\right)dt$$

$$\left\langle\vec{c}=\vec{l}-\vec{o}\right\rangle$$

$$=i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\int_0^1\left(\frac{1}{1+a\cdot\left\|\vec{c}-t\cdot\vec{d}\right\|^2}\right)dt$$

$$=i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\int_0^1\left(\frac{1}{1+a\left(\left(c_x-td_x\right)^2+\left(c_y-td_y\right)^2+\left(c_y-td_y\right)^2\right)}\right)dt$$

$$\left\langle u=1+a(\vec{c}\circ\vec{c});v=-2a\left(\vec{c}\circ\vec{d}\right);w=a\left(\vec{d}\circ\vec{d}\right)\right\rangle$$

$$=i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\int_0^1\left(\frac{1}{u+vt+wt^2}\right)dt$$

$$=i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\left[\frac{2\arctan\left(\frac{v+2wt}{\sqrt{4uw-v^2}}\right)}{\sqrt{4uw-v^2}}\right]_0^1$$

$$=2i\sqrt{d_x{}^2+d_y{}^2+d_z{}^2}\left[\frac{\arctan\left(\frac{v+2w}{\sqrt{4uw-v^2}}\right)-\arctan\left(\frac{v}{\sqrt{4uw-v^2}}\right)}{\sqrt{4uw-v^2}}\right]$$

$$=F(\vec{v}(t))$$

f - light attenuation function
i - source light intensity
a - global attenuation control value
v - view ray from camera (o) to world space pos of pixel (o+d), t=1
F - amount of light gathered along v

This lightning model can be applied just like global volumetric fog by rendering a full screen pass. By tweaking the controlling variables a and i volumetric lightning flashes can be modeled. F represents the amount of light emitted from the lightning source that gets scattered into the view ray. Additionally, when rendering scene geometry the lightning source needs to be taken into account when computing shading results.

(a) Scene at night

(b) Same scene rendering as the lightning strikes

Figure 7*. Volumetric lightning rendering.*

6.10 Conclusion

GPUs nowadays offer a lot of possibilities to realize complex visual effects in real-time. This chapter has shown a few examples of atmospheric effects implemented within *CryEngine2,* how scene depth based rendering was utilized in them, what integration issues had to be faced and how they got solved.

6.11 Bibliography

HARGREAVES, S. 2004. Deferred Shading, *Game Developers Conference*, D3D Tutorial Day, March, 2004.

NISHITA, T., SIRAI, T., TADAMURA, K., NAKAMAE, E. 1993. Display of the Earth Taking into Account Atmospheric Scattering, Proceedings of the 20th annual conference on Computer graphics and interactive techniques *(SIGGRAPH '93)*, pp. 175-182.

O'NEAL, S. 2005. Accurate Atmospheric Scattering. GPU Gems 2, Addison Wesley, pages 253-268

WANG, N. 2003. Realistic and Fast Cloud Rendering in Computer Games. Proceedings of the SIGGRAPH 2003 conference. Technical Sketch.

Internal replica of Na Pali Coast on Hawaii (Kauai) - real world photo(left), CryEngine2 shot (right)

Internal replica of Kualoa Ranch on Hawaii - real world photo(left), CryEngine2 shot (right)

Figure 8. *In-game rendering comparisons with the real-world photographs.*