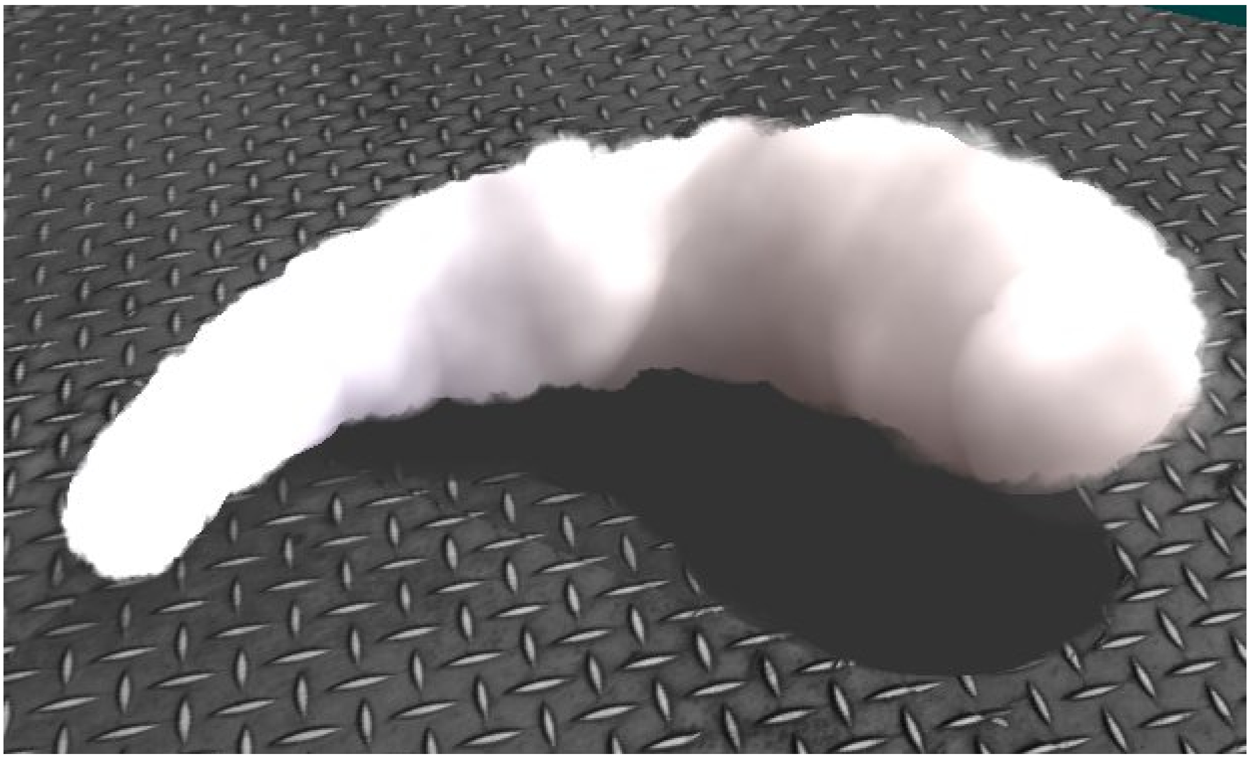
CHALMERS



Interactive Real-Time Smoke Rendering

ROBERT LARSSON

Master's Thesis

Computer Science and Engineering Programme

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2010

Interactive Real-Time Smoke Rendering

Robert Larsson

Examiner: Ulf Assarsson

Chalmers University of Technology

University of Gothenburg

Department of Computer Science and Engineering

SE-412 96 Göteborg

Sweden

Telephone + 46 (0)31-772 1000

The cover shows the final result of the thick smoke implementation discussed in chapter 3.3.

Department of Computer Science and Engineering

Göteborg, Sweden April 2010.

Abstract

There are many different approaches to real-time rendering of smoke. In this work, three different methods were tested and evaluated. Soft particles proved to be a reliable method to render smoke as particles doesn't have visible intersections with the scene. Smoke with volumetric lighting resulted in thick white convincing smoke, but it's too unstable to be used commercially. The soft particles solution were finally successfully implemented into the PC graphics engine at the game development company SimBin.

Content

1 Introduction

This work will compare different methods to render smoke realistic in games and implement the most appropriate method in the engine Simbin uses for PC racing games. The report assumpt that the reader is well knowledged in real-time rendering methods and vocabulary.

1.1 Background

In the video game business, there is an ever going strive to get better graphics in the games. Realistic rendering of smoke have recently got more attention, but so far, no good solution has been found. Smoke is an important visual effect in racing games since they are one of the few dynamic parts of the game. Cars, people and particle effects are what usually is dynamic in a racing games. Most games focus on the cars, and only implements simple solutions for people and special effects like smoke. But to get a visually realistic result, all parts of the scene must be of the same high quality. Therefore, it's important to render smoke and the other special effects in a convincing way.

Smoke is a very hard effect to render. It's made out of many tiny particles, which interact which the light in a complex way. With the advance of hardware, lightening calculations have recently become more and more advanced. But few have tried to improve particle rendering by using these more advanced lighting techniques.

1.1.1 SimBin

SimBin is a Swedish developer and publisher of racing games. They do development for both PC and XBOX platform. They have release titles as “Race 07 – The WTCC Game” and “GTR Evolution”.

Image 1: The logo of the game development company SimBin.

The renderer they use is a bought 3D engine called gMotor 2.0. This engine has been only little improved over the last years. And it comes with neither documentation nor support. The engine will be referenced as gMotor, CUBE engine or Simbin renderer throughout the report.

1.2 Goal

These were the goals of the project:

- Perform a study of current research in the smoke rendering field
- Develop a simple separate test application
- Combine the results into the best solution for today's hardware and rendering pipelines
- Implement the solution in the PC renderer at Simbin

1.2.1 Limitations

The following limitations constrained the project:

- No work will be done in refactoring current code that SimBin has.
- No implementation for XBOX will be done.
- The solution proposed should be targeted for a racing game. So it might not be appropriate for any other type of games.
- The particle system developed should only be able to simulate and render smoke.

1.3 Method

For the different parts of the projects, different methods were used. For research, a study of research papers and relevant books was performed. The combined result were summarized into the theory chapter of this report and proves as a basement for the discussion and implementation.

For application and effect development, a software development process were used. The final smoke effect and it's shaders were developed using the tools NVIDIA FxComposer 2.0 and AMD RenderMonkey 1.6. The development of the test application and the Simbin integration were done in Microsoft Visual Studio, and debugging was done with PIX.

1.3.1 Software development process

Even if this project was a one man project, a software development process were used to structure the work. The software development process used in this project was a Scrum influenced model. At the beginning, a development plan were done containing the overall goals of the project. The work was divided into small tasks with priorities. For each month, the most important task, based on project value, were selected and worked on. The work load and priorities were adjusted for each month.

1.3.2 Software architecture

The project consists of a test application, mostly used for rapid testing and evaluation of the different rendering, the particle system for rendering smoke, and an adapter to connect the particle system to the SimBin game code. Depending on chosen solution, the final implementation with the CUBE renderer might look different then what this architecture shows.

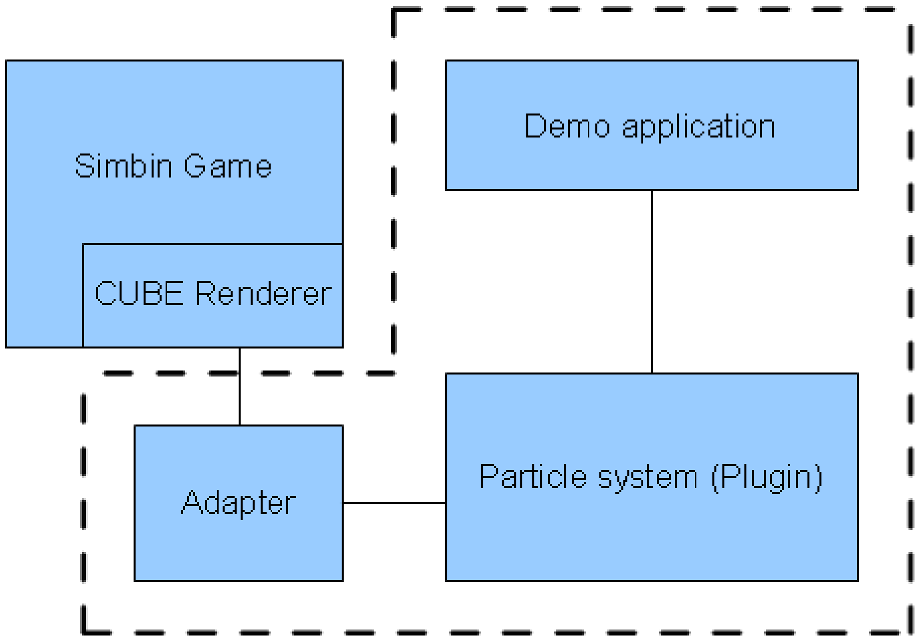
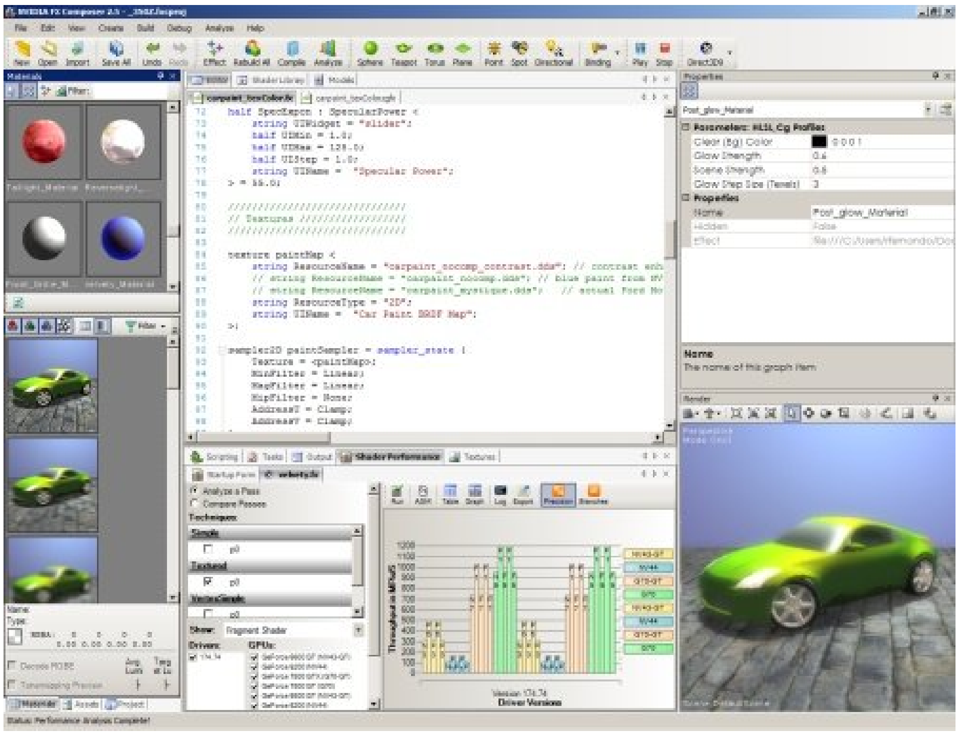


Image 2: The planned architecture for the system developed.

1.3.3 NVIDIA FxComposer 2.0

FxComposer is an IDE for developing effects and shaders in HLSL and CgFx. It focuses more on effect development than on single shaders. It's a useful tool for evaluating, developing, debugging, testing and optimizing shaders. The built-in support for handling DirectX Effect files makes it the first hand choice for many DirectX shader developers. The IDE comes with a large library of pre-made shaders.

This tool was only used for compiling and correcting the DirectX Effect files.



```
half SpecExpon : SpecularPower <
    string UIWidget = "slider";
    half UIMin = 1.0;
    half UIMax = 128.0;
    half UIStep = 1.0;
    string UIName =  "Specular Power";
> = 55.0;

//////////////////////////////
// Textures //////////////////
//////////////////////////////

texture paintMap <
    string ResourceName = "carpaint_nocomp_contrast.dds"; // contrast enh
    // string ResourceName = "carpaint_nocomp.dds"; // blue paint from NV
    // string ResourceName = "carpaint_mystique.dds";   // actual Ford Mo
    string ResourceType = "2D";
    string UIName =  "Car Paint BRDF Map";
>;

sampler2D paintSampler = sampler_state {
    Texture = <paintMap>;
    MinFilter = Linear;
    MagFilter = Linear;
    MipFilter = None;
    AddressT = Clamp;
    AddressY = Clamp;
```

Image 3: This is what it looks like to develop shaders in FxComposer.

1.3.4 RenderMonkey 1.6

RenderMonkey is an IDE for developing shaders in GLSL and HLSL. It's a useful tool for evaluating, developing, debugging, testing and optimizing shaders. The tool simplify render target creation and can easily be used to rapidly test new shader ideas. The drawback is lack of good DirectX Effect support. This IDE also comes with a large library of premade shaders.

This tool was used before FxComposer to test out shader ideas rapidly. It's faster to develop in RenderMonkey because it has better support for render targets.

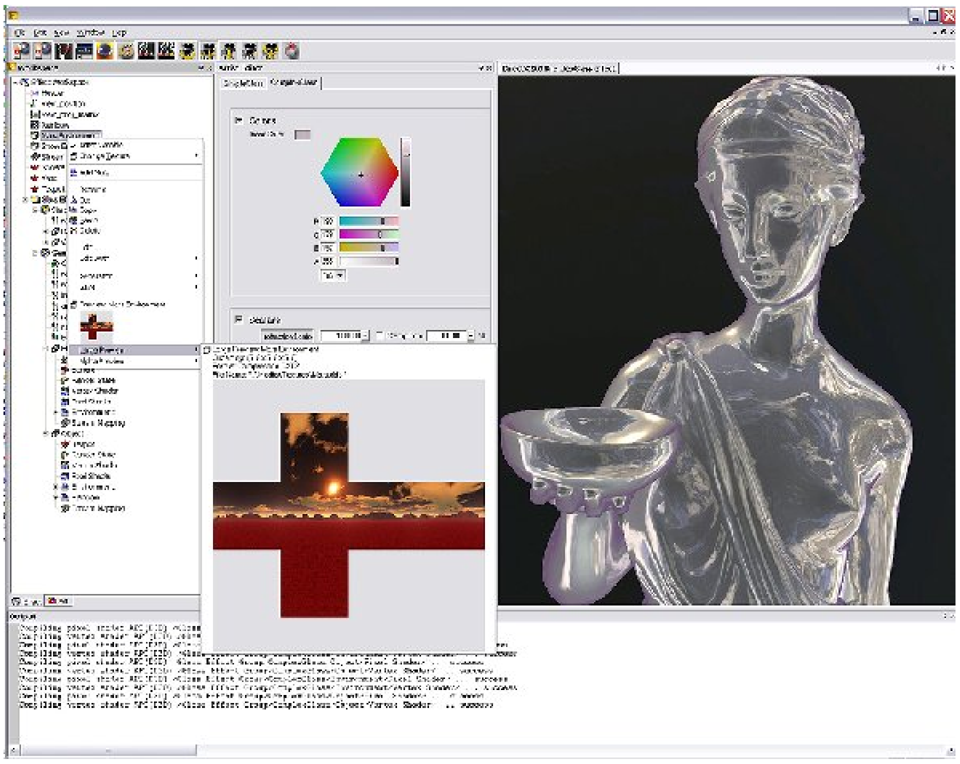


Image 4: An example workspace for developing shaders in RenderMonkey.

1.3.5 Microsoft DirectX9 SDK

The development in this work used the DirectX API for accelerated 3D graphics and therefore, the DirectX9 SKD was used. The DirectX9 SDK includes demos, library files, and header files for compiling projects using DirectX9. The test application is based on the test suite that is included in the SDK.

1.3.6 Microsoft Visual Studio

The Microsoft Visual Studio was used as IDE for developing the C++ code. It's one of the most popular tools for developing products for the Microsoft platform. And the example projects in the DirectX9 SDK includes Visual Studio project files to simplify for users.

1.3.7 PIX

PIX is a tool included in the DirectX SDK for debugging the rendering pipeline of a DirectX application. The main feature is the possibility to debug single pixels on the screen (or on render targets) and track down errors in the shaders by stepping trough the code of them.

2 Theory

The following chapters introduce the theory needed to understand the discussion and results. A good knowledge in computer graphics, computer programming and linear algebra is recommended.

2.1 Rendering pipeline

The graphics rendering pipeline consists of three major stages, application, geometry and rasterization. The goal with the pipeline is to convert a 3D scene into a 2D image.

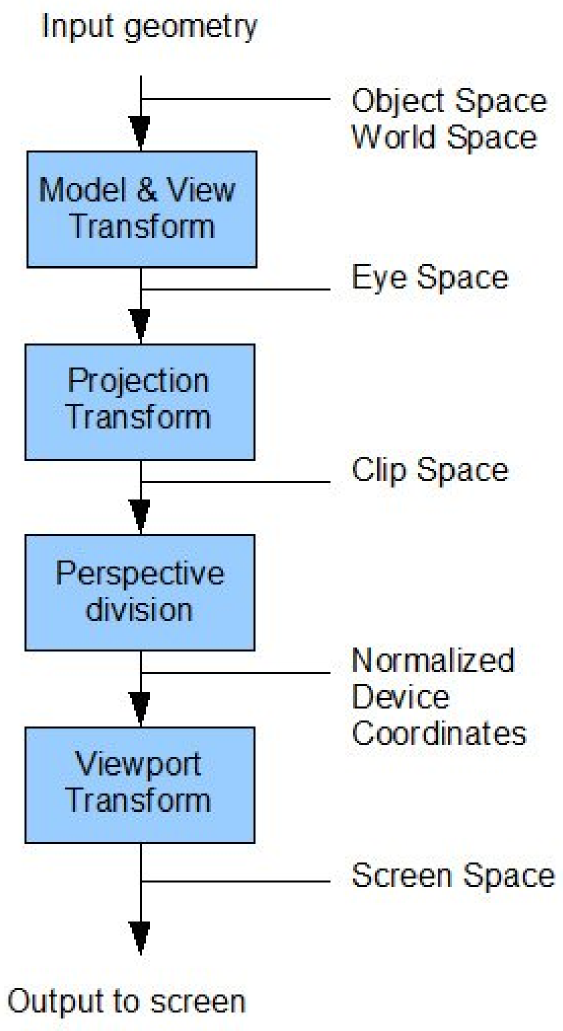


Image 5: This image shows the different transformations needed for the geometry throughout the pipeline.

In the application stage, the application culls and send the models that should be rendered to the graphics card. It also decides what render states to use, shaders and textures. The application stage has the responsibility to set up the matrices used.

In the geometry stage , the vertices are transformed from model space into clip space. In this space, the triangles are clipped against the camera frustum.

The rasterization stage renders the triangles as pixels to the screen.

Since it's a pipeline, the stage which is the bottleneck has the greatest negative impact on the throughput. And therefore, this bottleneck should be main focus of optimization.

2.2 Shaders

Advances in graphics hardware have made it possible for graphic programmers' to program certain parts of the rendering pipeline. These programmable parts of the pipeline are called shaders. There are currently three types, vertex shaders, geometry shaders and pixel shaders. Pixel shaders are sometimes called fragment shaders, for example in OpenGL. The high-level shading language in DirectX is HLSL and in OpenGL it's GLSL. NVIDIA also has their own shading language CgFx, which is very similar to HLSL in syntax but works in both DirectX and OpenGL.

Vertex Shader

Has all the vertices as input and has as the goal to transform them into cull space. Does also often do animation, lighting and more.

The vertex shader is a part of the geometry stage of the rendering pipeline.

Geometry Shader

Has vertices as input and can output more vertices if needed.

The geometry shader is a part of the geometry stage of the rendering pipeline.

Pixel Shader

Does computations for each pixel of the geometry rendered. Will decide exactly what color is sent to the blending stage. Since the pixel shader is executed for each pixel, it will the the shader that is (nearly always) executed the most. And should therefore be optimized the most.

The pixel shader is a part of the rasterization stage of the rendering pipeline.

2.3 Instancing

Geometry instancing is a method to decrease bandwidth usage when rendering many copies of the same geometry. This is done by reducing the overhead of drawing multiple copies of the same vertex buffer. In DirectX this can be setup by using different streams (at least two) when rendering. Each stream can contain data as position, color, uv-coordinates and more. By setting different frequencies on these streams, one or more streams can be reuse many times and therefore saving memory and also increase performance.

2.4 Particle systems

Particle systems have been used for special effects since the dawn of computer graphics. They are often used to render smoke, fire, explosion and more. The main reason to use visualize these effects with particle systems is that these types of effects originally consists of millions of small particles in real life. In a particle system, these small particles are approximated by larger particles following some basic rules. The more particles used, the better result. But the more particles, the slower it is.

Main components of a particle system are, emitters, particles and forces. The emitter decides how creation of particles should be done. For example in which angle, quantity, speed, size and distribution. The particle should be possible to be rendered to the screen. Often as a billboarded quad. Sometimes, forces are used to affect the particle system, for example gravity or wind.

When doing a particle system we must decide what type of geometry to send to the GPU. In other words, what vertices. It's also important to decide where and when to transform the geometry to screen space for rasterization.

It have been standard to use quads (sometimes just single triangles) as representation for particles. These quads are usually billboarded to the screen which requires additional work. There also exist a technique called point sprites, in both DirectX and OpenGL, that allows us to render a simple billboarded image by just sending a single vertex for each particle to the GPU, compared to at least four when doing quads (and maybe also indices). This sprite will be scaled by the distance accordingly to a formula that can be tweaked. It's easy to translate it (it's the position of the vertex itself) but it's not so easy to rotate it. Rotation would need a special vertex or fragment shader that does the rotation manually.

Sprites can reduce the bandwidth usage a lot, at the cost of increased complexity in the shader. Quads on the other hand allows more possibilities, for example stretching. If sprites is faster than quads depends on hardware and where the bottleneck is in the rendering pipeline.

2.5 Billboarding

Billboarding [13] is the technique to rotate a quad to align to the screen. The technique have been popular for a long time in computer games to fake complex geometry. For example, the computer game Doom used billboarded animated sprites as enemy characters. Today, billboarding are used to fake distance geometry by a technique called impostors or used in particle systems.

Billboards are often either viewplane-aligned or viewpoint-oriented. As Image 6 shows, there is a difference in facing of the billboard between the two alignments. The calculation needed to billboard a quad can either be done on the CPU or on the GPU in a vertex shader.

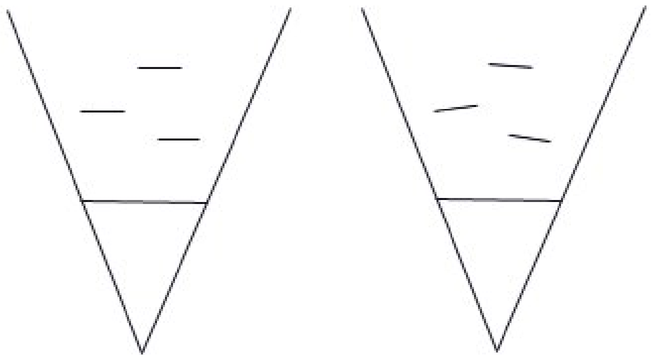


Image 6: Viewplane aligned billboards on the left. Viewpoint-oriented billboards on the right. The billboards on the right will rotate when camera is moved forward, but the left ones will not.

Viewplane-aligned billboards

One technique [14] to do billboarding is to remove the rotation and scaling from the view matrix. This requires that the vertices of the quad are already positioned correctly in view space to be projected. Rotation and scaling can be removed by setting the upper left 3x3 in the projection matrix to the identity matrix.

Another approach to viewplane-aligne billboards is to do the positioning of thee vertices in a vertex shader. The goal with the shader will be to position each corner vertex for the quad so they form a viewplane aligned quad. Since the world space vertex position is calculated in the shader, the quad doesn't need any position for the vertices. It only needs texture coordinates and the center position of the quad. The texture coordinates will be used to identify the four corners of the billboarded quad and offset them correctly from the center position. In this example, the following texture coordinates are used for the four corners:

(0,0)	(0,1)
(1,0)	(1,1)

From the view matrix, we can get the up and right vector of the camera in world space. Combining this with the texture coordinates and the size of the quad is enough to do billboarding.

The following code snippet shows how it can be done in a HLSL vertex shader:

```
VS_OUTPUT vs_main( VS_INPUT Input )
{
  VS_OUTPUT Output;
  float3 rightVector = normalize(float3(matView[0][0], matView[1][0],
matView[2][0] ));
  float3 upVector = normalize(float3(matView[0][1], matView[1][1],
matView[2][1] ));
  float2 offset = (Input.texCoord.xy - 0.5f)*2; // scale from 0..1 to -1..1
  float4 position =  float4(billboardCenterPosition +
billboardSize*(offset.x*rightVector + offset.y*upVector),1);
 // position is in world space
  Output.Position = mul( position, matWorldViewProjection );
  return( Output );
}
```

2.6 Fluid simulation

For realistic simulation and rendering of effects like smoke or water, a fluid simulation and rendering approach is needed. There are currently two popular methods for this:

- Simulate the fluid on the CPU and send the result as particles to the GPU for rendering as billboards. This is often called a particle system. The technique has been around since the dawn of computer graphics.
- Simulate the fluid on the GPU and render the result into textures. This will then be rendered by doing volume ray casting (or ray marching) on the GPU. This technique is new and rather unexplored, and there are few real-life implementations. The result can be very realistic but slow.

Technique one burdens both CPU, bandwidth and GPU. Although in modern solutions, it’s often the bandwidth that’s the bottleneck. The GPU based technique only burdens the GPU (but a lot).

Image 7: This image shows an example of fluid simulated and rendered realistically on the GPU.

2.7 Soft particles

The aim with soft particles is to remove the ugly artifact that appears when the particle quad intersects the scene. There are a lot of different approaches to solve this, some more complicated than others. The simplest formula for soft particles is to fade the particle if it’s getting to close to the scene. To do this, the scene without particles has to be rendered first and the depth saved in a texture. When drawing the particles, the depth of the particle will be compared to the scene depth. The alpha should be increased by a smooth fade by this depth difference. The formula below in HLSL is the simplest possible for soft particles, and works very well. Scene_depth is the sampled depth (in view space) of the scene in the direction of the current pixel. Particle_depth is the depth (in view space) of the current particle pixel. Scale is used to control the “softness” of the intersection between particles and scene.

*fade = saturate((scene_depth – particle_depth) * scale);*

NVIDIA [1] proposes a method that the following fade should be used instead of the linear one described above, to make the fade even smoother.

*float Output = 0.5*pow(saturate(2*((Input > 0.5) ? 1-Input : Input)), ContrastPower);*

Output = (Input > 0.5) ? 1-Output : Output;

Umenhoffer [2] proposes a method called spherical billboards to deal with these problems. In this method, the volume is approximated by a sphere. This method also deals with the near clip-plane problem that particles will instantly disappear if they get to close to the camera.

There is also an idea [3] that the alpha channel can be used to represent the density of the particles. Although this method has the drawback that the textures might need to be redone by the artists.

The method by Microsoft [4] uses a combination of spherical billboards and a texture representation of the volume. But instead of using the alpha channel as a simple texture, they ray march the sphere and sample the density and volume from a 3D noise texture. The result can be seen in Image 8.

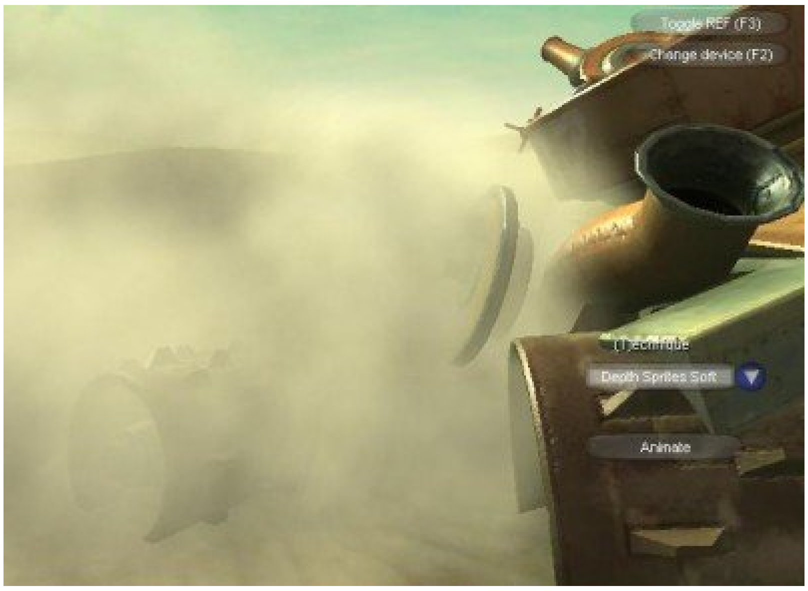


Image 8: This screenshot shows how the volumetric soft particles look like in the DirectX10 SDK.

2.8 Mega Particles

Instead of rendering textured billboards in a particle system, a technique called "Mega Particles" [5] render spheres to an off-screen texture. This texture is then blurred and randomly displaced using a fractal cube. The final result is blended into the scene, taking depth into account as with soft particles. The result is a volumetric cloud that is lit by lighting and look volumetric. The problem is that this technique suffers from the shower-door effect which makes it annoying to view and use in practice. It's also harder for artist to control the final look. Also, going inside the cloud requires special treatment not mentioned in the slides.

2.9 Lighting theory

The very common Phong Lighting model for real-time rendering divides the material lighting equation into diffuse, specular and ambient contributions. These three properties have worked well for solid materials, but they does not correctly approximate the lighting interaction with translucent materials. For example smoke requires a more complex lighting formula that includes the light scattering that happens inside the smoke.

When lighting is scattering many times it's called multi scattering. Multi scattering [15] is the diffuse term in the Phong Lighting, but can also be calculated using the Mie scattering theory. Mie scattering can be approximated with the Henyey-Greenstein phase function [16] seen in equation 1.

$$p_{hg}(\theta)=\frac{1}{2}\frac{1-g^2}{(1-2g\cos(\theta)+g^2)^{\left(\frac{3}{2}\right)}} \quad \textit{Equation 1}$$

2.10 Deferred Rendering

This is a lighting rendering technique [6] [7] [8] that lately has increased a lot in popularity. The normal way of shading is to perform the lighting calculations on a fragment when it is rasterized to the screen. This is often good but requires a lot of calculations if there are many lights. And the bad thing is that this fragment might later on be overwritten by some other fragment so the calculations might be a waste.

In deferred lighting (or deferred shading, or deferred rendering), you save the information about the fragment that is necessary to perform the shading (lighting) by rendering them to textures instead of doing the actual lighting calculation. When all geometry is rendered, the lighting will now be calculated only once per pixel on the screen. So no calculations will be wasted. This can perhaps be seen as a lazy evaluator.

The information saved per fragment is often:

- position (or only depth)
- albedo (the diffuse texture)
- normal
- specular

When all geometry has been rendered and it’s time to perform the lighting, the lights needs to be represented as geometry when sent to rasterization. This geometry should fit the volume reached by the light. A smaller volume is faster to render. Point lights can be drawn either as spheres or just square billboards. Directional lights should be drawn as a full screen rectangle. And spotlights will be cones. Note that this shading technique allows for lights shaped in any form, not just these traditional ones. And sometimes, stenciling can be used to avoid unnecessary shading calculations.

The big reason for using deferred rendering is how well it scales with more lights. Another reason that it has increased in popularity lately is how nice it works with recent post-process effects like Screen Space Ambient Occlusion (SSAO) and Depth of Field (DOF).

The problem areas with deferred lighting are transparent objects and multisampling. If the original scene didn’t used per-pixel lighting (but instead maybe vertex lightning) on the whole scene then the deferred rendering might be slower than traditional rendering.

2.11 Position reconstruction

There are many occasions when the fragment position in world space needs to be reconstructed from a texture holding the scene depth (depth texture). One example of use is in deferred rendering when trying to decrease memory usage by not saving the position but instead only the depth. This will result in one channel of data, instead of three channels needed when saving the whole position.

There are different ways to save the depth. The most popular are view space and screen space. Saving depth in view space instead of screen space gives two advantages. It's faster, and it gives better precision because it's linear in view space.

This is how screen space depth can be rendered in HLSL:

```
struct VS_OUTPUT
{
  float4 Pos: POSITION;
  float4 posInProjectedSpace: TEXCOORD0;
};
// vertex shader
VS_OUTPUT vs_main( float4 Pos: POSITION )
{
  VS_OUTPUT Out = (VS_OUTPUT) 0;
  Out.Pos = mul(Pos,matWorldViewProjection);
  Out.posInProjectedSpace = Out.Pos;
  return Out;
}
// pixel shader
float4 ps_main( VS_OUTPUT Input ) : COLOR
{
  float depth = Input.posInProjectedSpace.z / Input.posInProjectedSpace.w;
  return depth;
}
```

The HLSL pixel shader below shows how the position can be reconstructed from the depth map stored with the code above. Although this is one of the slowest ways of doing position reconstruction since it requires a matrix multiplication.

```
float4 ps_main(float2 vPos : VPOS;) : COLOR0
{
  float depth = tex2D(depthTexture,vPos*fInverseViewportDimensions +
fInverseViewportDimensions*0.5).r;
  // scale it to -1..1 (screen coordinates)
  float2 projectedXY = vPos*fInverseViewportDimensions*2-1;
  projectedXY.y = -projectedXY.y;
  // create the position in screen space
  float4 pos = float4(projectedXY,depth,1);
  // transform position into world space by multiplication with the inverse
view projection matrix
  pos = mul(pos,matViewProjectionInverse);
  // make it homogeneous
  pos /= pos.w; // result will be (x,y,z,1) in world space
  return pos; // for now, just render it out
}
```

To reconstruct depth from view space, a ray from the camera position to the frustum far plane is needed. For a full screen quad, this ray can be precalculated for the four corners and passed to the shader. This is how the computer game Crysis did it [10] . But for arbitrary geometry, as needed in deferred rendering, the ray must be calculated in the shaders [9] .

2.12 Texture atlas

Texture atlas [11] [12] is a technique to group smaller textures into a larger texture. This decreases the number of state switches a renderer needs to do and therefore often increases performance. Texture atlases have been used for a long time in the video game industry for sprite animations. When using texture atlases, the uv-coordinates of the models have to be changed so the original 0..1 map to the texture tiles in the atlas. Grouping of textures can be done manually by texture artists or with tools. The texture coordinate system can be changed to map the new texture in a tool, or in the shader at run-time.

There are some limitations with using texture atlases compared to normal textures. First of all, all texture coordinates must initially be within 0..1 range. So for example, no “free” tiling can be used. The other problem is bleeding between tiles in the atlas when doing filtering, for example when using mipmaps.

2.13 Ray-sphere intersection

The intersection test [13] between a ray and a sphere is useful in raytracing, physics, picking and more. The test has three solutions, no intersection, one intersection, or two intersections.

The ray can be defined as

$x(t) = a + tk$

and the sphere as

$(x – c) (x – c) = r^2$

where c is the center of the sphere and r is the radius.

It's simple to calculate the algebraic solution. But for better performance, a little bit of reorganization needs to be done to only perform squaring when actually needed. The result is the pseudo code below. A small bias, SMALL_BIAS, is needed to deal with inaccuracies in the calculations. This also results in that the solution with only one intersection point is removed. Instead, an intersection is always in two points. In the pseudo code, nearT and farT is the t value for the two intersections.

```
delta = a – c
A =  k dot k
B = 2 (delta dot k)
C = delta dot delta – r^2
disc = (B^2 – 4AC)
if disc <  SMALL_BIAS
   no hit
else
  hit
  sqrtDisc = square( disc )
  nearT = (- B -  sqrtDisc) / (2A)
  farT = (- B + sqrtDisc) / (2A)
```

This is the HLSL code for the same algorithm. It's taken from the Soft Particle example in the DirectX10 SDK.

```
#define DIST_BIAS 0.01
bool RaySphereIntersect( float3 rO, float3 rD, float3 sO, float sR, out float tnear, out float tfar )
{
    float3 delta = rO - sO;
    float A = dot( rD, rD );
    float B = 2*dot( delta, rD );
    float C = dot( delta, delta ) - sR*sR;
    float disc = (B*B - 4.0*A*C);
    if( disc < DIST_BIAS )
    {
        return false;
    }
    else
    {
        float sqrtDisc = sqrt( disc );
        tnear = (-B - sqrtDisc ) / (2*A);
        tfar = (-B + sqrtDisc ) / (2*A);
        return true;
    }
}
```

2.14 Gaussian blur

There are many different filters to blur an image and one of the more popular ones are the Gaussian blur filter. It's based on the Gaussian Function, a bell-shaped curve function. The Gaussian blur filter does blurring by working as a low-pass filter, removing all high-frequencies in the image.

When applying the Gaussian function as a blur filter on an regular 2D image, it has to be applied in both dimensions of the image. By utilizing the linearly separability of the filter, the blurring can be done in one direction at a time which is often much faster than using the 2D Gaussian blur filter.

3 Results and Discussion

This chapter concludes the result from the thesis work. Three different solutions where developed. One based on soft particles, one based on mega particles and finally one based on an experimental particle technique using volumetric lighting. The implementation of the final result into the renderer at Simbin is presented as well.

As a first step, the fluid smoke simulation demo from the DirectX10 SDK was tested. Unfortunately, both speed and quality was unacceptable so the idea was abandoned right away. This approach will probably mature in the next years, but it's too early to implement it into a commercial racing game. There are also still to many DirectX9 only computers among the gaming population. So using a solution that requires DirectX10 or later isn't appropriate yet.

3.1 Soft Particles

The first solution developed was a simple soft particles implementation. It was only tested in RenderMonkey and a comparison with softness enabled or disabled can be seen in Image 9.

The solution used the simple fade described in Chapter 2.7 . Depth was rendered in view-space, and the billboarding was done in the vertex shader (for simple testing). This technique is very robust, works well in all situations and only requires Shader Model 1. Many games use this solution to render large particles.

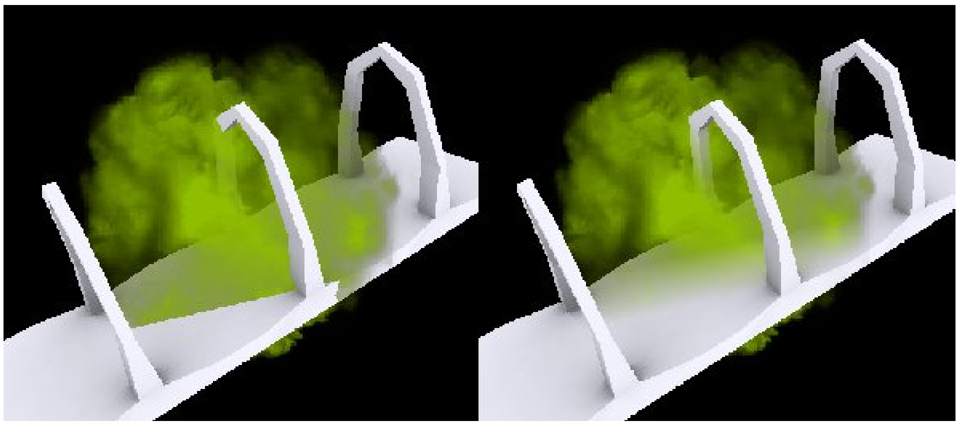


Image 9: The left image shows a scene with a single large green particle. Note the very distinct edge at the intersection of the particle quad and the scene. In the image on the right, the same particle is rendered. But this time, it's using the soft particles shader developed in this project. As seen, the intersection is now much smoother.

One of the test cases, where a box uses the soft particle shader, can be seen in Image 10. As seen, the distance-fade can have more uses than just for particle systems.

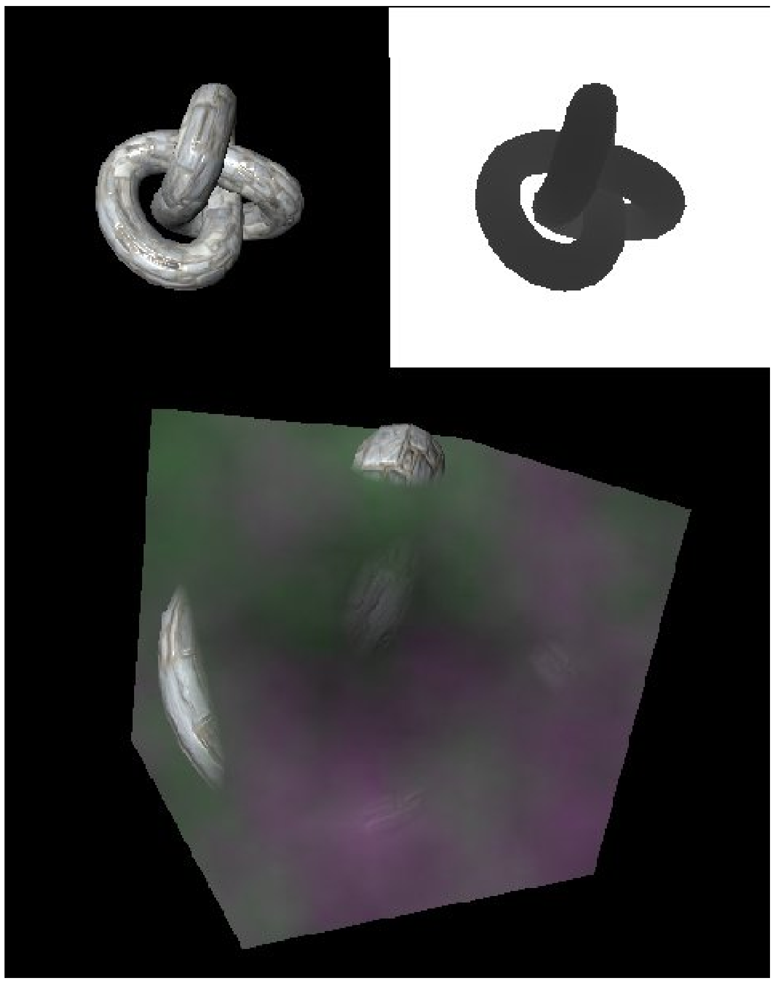


Image 10: ***Top left*** *image shows the first pass that renderer the scene, in this case a textured torus.* ***Top right*** *image shows the second pass, rendering the scene depth. Note that the lighter it is, the further away it is from the viewer. The* ***bottom*** *image shows the final pass where a textured cube is rendered using the soft particle shader. Because of the smooth fade, it's possible to see into the cube. It's behaving like fog. Note that there is no hard intersection between the two objects anywhere.*

3.2 Mega particles

A solution based on the mega particles technique described in chapter 2.8 was developed. This solution was only tested in RenderMonkey and the very noticeable screen-door effect was verified. Since the technique isn't based on any physical formulas or real life observations, the implementation details of the shader and all parameters where decided by testing.

The implemented method resulted in four sequential passes seen in Image 11. First, all particles are rendered as spheres, lit with ordinary Phong Lighting. This is rendered to an off-screen texture with at least three color channels. In the next step, the off-screen texture is blurred using the two-pass Gaussian blur filter.

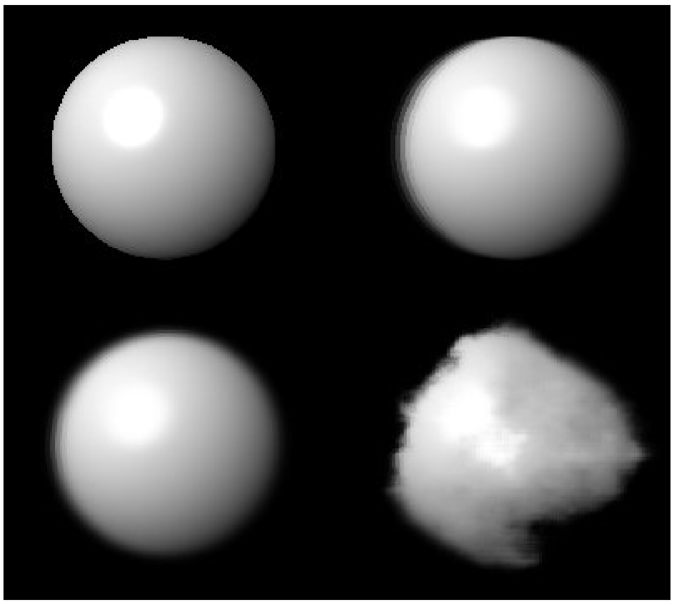


Image 11: These four images shows the steps involved in rendering smoke using the mega particles technique. Top left image is a simple sphere rendered to a texture. Top right image is the sphere blurred in horizontal direction. Bottom left image is the sphere blurred in horizontal direction. And the bottom right image shows the result of using the mega particles shader.

At last, a full screen quad is rendered with the mega particles shader. The vertex shader is a simple shader that stretches the quad over the screen and creates texture coordinates for the vertices. These texture coordinates should map to screen coordinates and are passed to the fragment shader. In the fragment shader, the texture coordinates will together with a time variable create a 3D texture coordinate. This 3D texture coordinate is then used to sample from a 3D noise texture. Because of the use of time in the texture coordinates. The result will vary periodically when time passes. Creating the illusion of an animation as seen in Image 12. The 3D noise sample is used as a seed when some random samples are taken from the previously blurred texture.

The example code below shows how the 3D noise texture is sampled in the fragment shader.

```
float2 coord =
tex3D(NoiseTexture,float3(Input.TexCoord*1.25,fTime/8.0f)).rg;
```

The result was something that looked like a cloud, lit by a light. Some noticeable drawbacks are the difficulty in tweaking the noise sampling, so the final result gets enough detail without looking too unrealistic. Also, since the screen coordinates are used when sampling from the noise texture, when the sphere moves, or the camera, there will be a very noticeably shower-door effect.

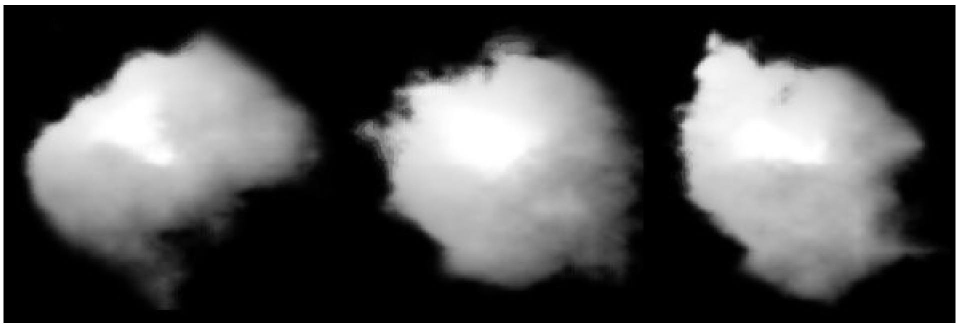


Image 12: The image shows three random frames of a single mega particle.

3.3 Thick smoke with volumetric lighting

This is a new experimental technique based on the recent research in how to render smoke, clouds and snow waves. It's also inspired by the recent innovations in deferred rendering. Focus with the technique was on rendering non-physically correct but visually convincing thick smoke. Few other smoke rendering algorithms deal with thick white smoke.

The basic idea is to render the particles as spheres. All these spheres will create the volume of the smoke. Then, lighting will be applied to this volume by raytracing through it and collecting the contribution in the direction of the light source. Some noise is also applied to add high frequency details and make it more “cloud like”.

To create the volume for raytracing, the spheres must be rendered to two textures. The first texture saves the min and max depth in the view direction for the volume. The second texture saves the min and max depth in the light direction for the volume. These two textures will then approximate the volume of the cloud.

Drawbacks that initially were identified were the limit of only one light source and the difficulty to fade away dying particles.

3.3.1 Test application

The test application was developed to easily test and tweak the parameters of the deferred particle technique in a real DirectX9 application. The code was written to cleanly separate between test application code and the smoke rendering engine code.

The test application code holds all the functionality necessary to launch, load and exit the application. This code is inspired and partly copied from the DirectX9 SDK examples. The GUI controls, for real-time tweaking of variables, are also in this part of the code.

The smoke rendering engine is based on a set of abstract classes, making the foundation of functionality required for rendering of the smoke. The DirectX9 implementation inherits from these abstract classes, and implement them accordingly to DirectX9 specification. This architecture makes it easier to port the engine to other API's. Some of the more important object oriented design patterns used are the factory method (for textures), singleton (for the renderer instance) and adapter (DirectX9 implementation) .

In the test application, the shader parameters of the smoke can be changed and updated in real-time. This enables fast testing and development. Image 13 shows what it looks like when running the test application.

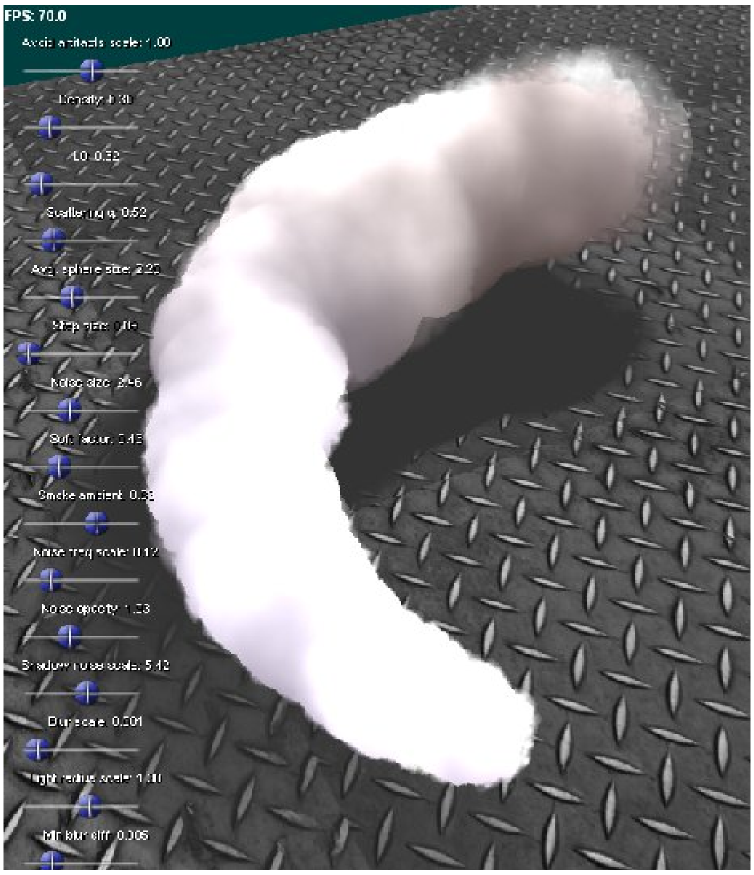

FPS: 70.0
Avoid artifacts scale: 1.00
Density: 0.30
L0: 0.32
Scattering q: 0.52
Avg. sphere size: 2.20
Step size: 0.04
Noise size: 2.46
Soft factor: 0.43
Smoke ambient: 0.33
Noise freq scale: 0.12
Noise opacity: 1.23
Shadow noise scale: 5.42
Blur scale: 0.001
Light radius scale: 1.00
Min blur diff: 0.005

Image 13: This image shows the test application in action with all the tweak-able parameters on the left. They can be changed and the result will be updated in real-time in the scene.

3.3.2 Fake rendering a sphere

Rendering the particles as hundreds of spheres as meshes, with triangles enough for the quality needed, would be slow. Therefore, the spheres are rendered as billboarded quads. A shader is used when rendering these quads and in this shader, an imaginary sphere, with center of the billboard and size of the particle, is raytraced. The view vector is checked against the sphere for intersection, the intersection points will be used when saving the depth to the textures. Image 14 shows the difference between rendering a sphere mesh and faking the sphere. There is some error since the sphere mesh isn't truly spherical because its made up of many tiny flat triangles.

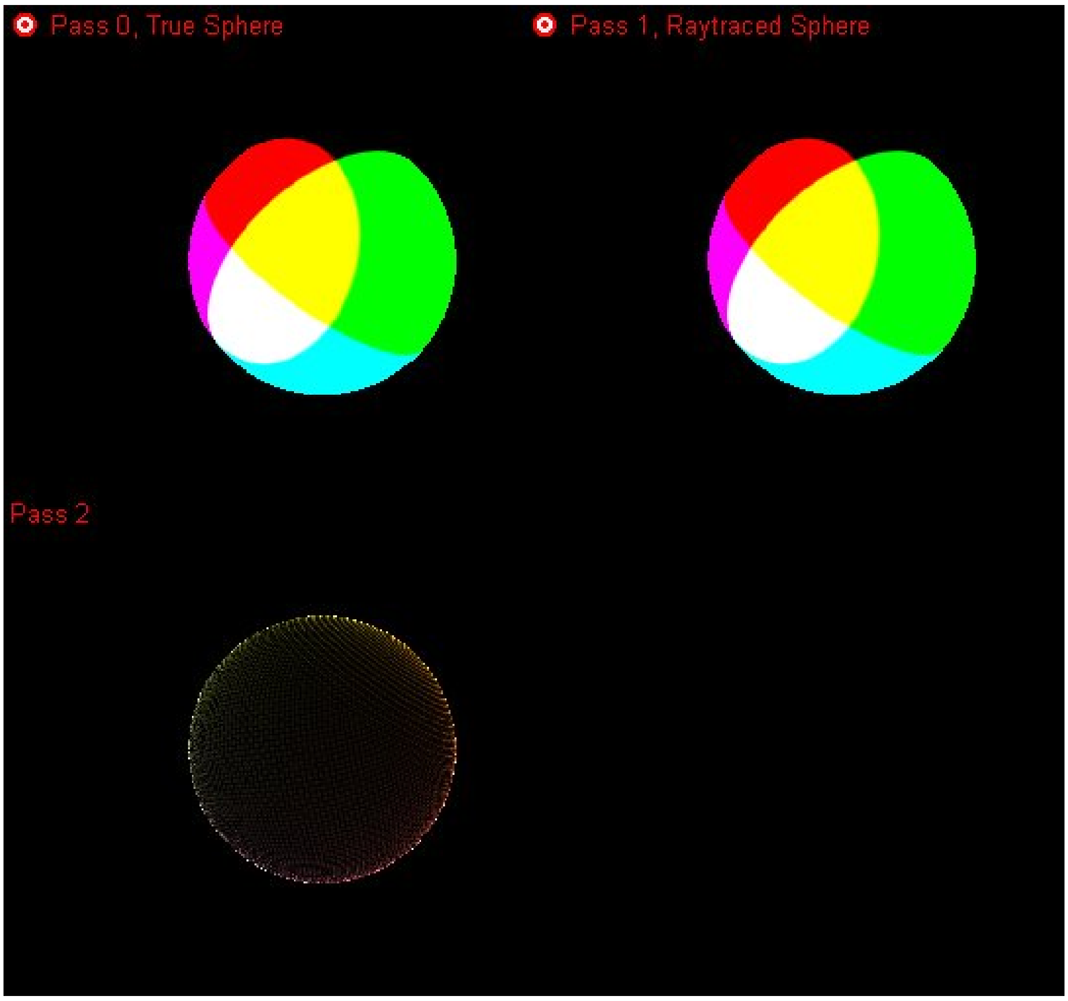


Image 14: In this example, Pass 0 renders a true sphere by doing rasterization of a sphere mesh. In Pass 1, a quad is rendered and an imaginary sphere is raytraced. For both these passes, the world position is written to a texture. In Pass 2, these two world positions are compared and the difference is written to the screen. As seen, most difference is around the edges of the sphere.

The drawback with the method is that no z-culling could be used. Since it would cull wrong in this simple implementation.

3.3.3 Billboarding shader

The particle quadss are rendered by instancing to save bandwidth. The instancing is done in two streams of data. The first stream is shared for all particles. And the second stream is unique for each particle. The second stream is updated each frame.

First stream holds following data:

– **texture coordinates**, for billboarding of the quad

Second stream holds following data:

- **particle position**, the center position of the particle, used for billboarding and in sphere intersection
- **size**, the size of the particle, used in billboarding and in sphere intersection
- **life**, 0..1 range of the life of the particle with zero as dead and one as fully alive. It's used to fade away the particle.
- **texture translation**, two coordinates that select which of the four textures to use for this particle
- **texture rotation**, an angel used to rotate the texture for this particle

The billboarding of the quads were done in the vertex shader as described in chapter 2.5.

3.3.4 Position reconstruction

There are two textures that save the position data.

1. A texture that save the min/max depth in the view direction for the volume.
2. A texture that save the min/max depth in the light direction for the volume.

Since no comparison between current fragment result and previous (in the same render target) can be manually made in the shader, a blend mode must be used to save a min or max value. Trying to save the actual position with a min or max blend mode will fail because of obvious reasons. Therefore, we have to save the depth instead and later reconstruct the position from it as seen in Image 15.

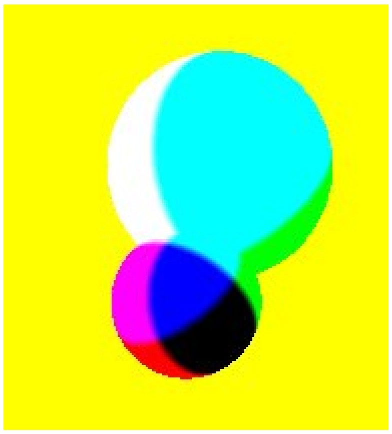


Image 15: This image shows the world position of the pixels of two particles. This world position have been reconstructed from the depth of two raytraced spheres in a previous pass. The position coordinates have simply been written to the screen and naturally colors the spheres. 0,0,0 -> white, 1,1,1 -> black, 1,0,0 -> red, and so on.

In this work, the projected screen space depth is stored in the texture. This is far from optimal and better implementations should prefer view space depth. But for simplicity, this work will use projected screen space depth and reconstruct it the slow way by multiplying it with the inverse view-projection matrix.

Because of the use of screen space depth in this work, 32-bit floating point textures have to be used to store the depth with good enough accuracy. Anything less will result in very noticeably artifacts.

3.3.5 Ambient lighting

Ambient lighting is an approximation of the scattered indirect lighting. This lighting can lit areas in shadows, and also contribute with color bleeding. Therefore , the ambient lighting of the smoke is approximated by sampling from a cube map with a blurred image of the surrounding. This works the same way as reflection mapping. And the normal used is the direction from the center of the smoke to the closest position to the screen of the volume for the current pixel. Image 16 shows the difference with ambient lighting enabled or disabled.

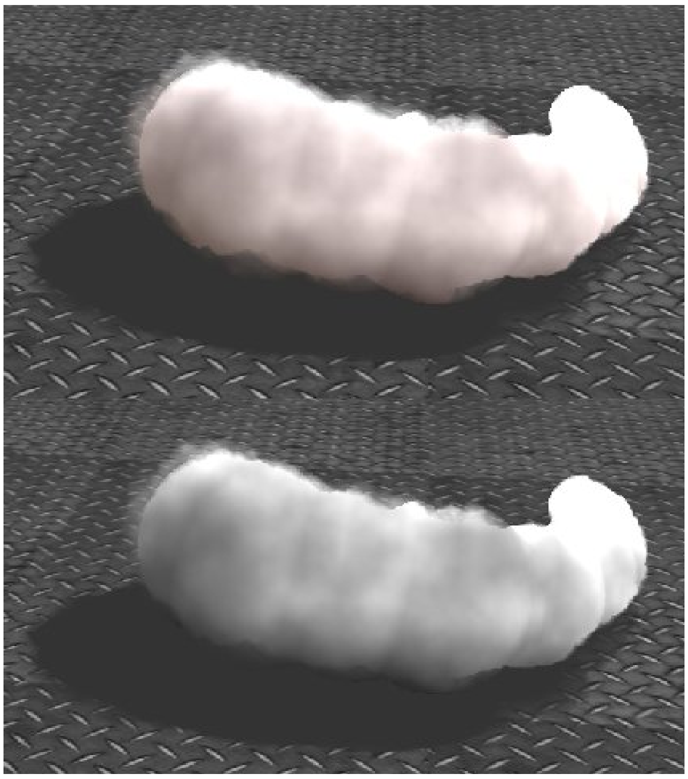


Image 16: The image at the top shows the smoke with ambient lighting, the image at the bottom shows the same smoke without any ambient lighting.

3.3.6 Texturing

Textures were used to symbolize the density of the smoke. The goal with them was to add high frequency details to the smoke.

To simplify the shader, only the four different textures shown in Image 18 were used for the smoke particles, and the limit was hardcoded. These textures were stored in a texture atlas with dimensions 2x2. To improve the visual quality by giving even more diversity, the textures where rotated in the vertex shader in a random angle.

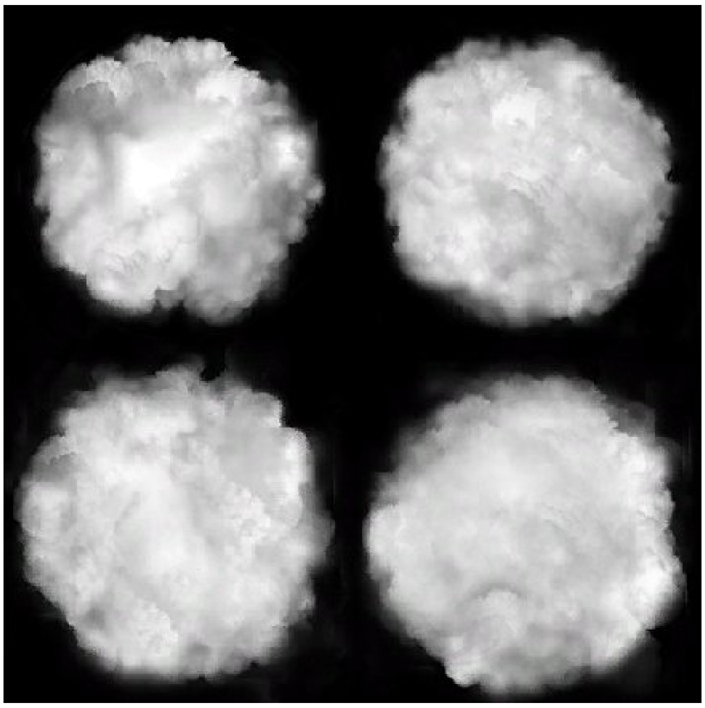


Image 17: This image shows the four textures, combined in a texture atlas, used as density textures for the particles. Their goal was to add high frequency details to the smoke.

Both rotation angle and what texture to use were sent to the shader as particle properties.

The following HLSL snippet of code calculates the rotation and translation matrix used when sampling from the texture atlas.

```
float sin;
float cos;
sincos(angle,sin,cos);
float2x2 rotationMatrix = { cos, -sin, sin, cos };
Input.texCoord = mul(Input.texCoord,rotationMatrix);
Input.texCoord = Input.texCoord*0.25f+Input.texTransform.xy;
```

3.3.7 Shadows

Most particle system implementations lack shadows because shadows would make them to slow. This particle system technique does already have all data needed to do shadows so adding shadows the final image will only decrease performance a little. Since no actual polygon structure exists for the smoke, shadow volumes cannot be used. Therefore, shadow mapping was chosen. The two different parts of shadow mapping technique are depth comparison and texture projection.

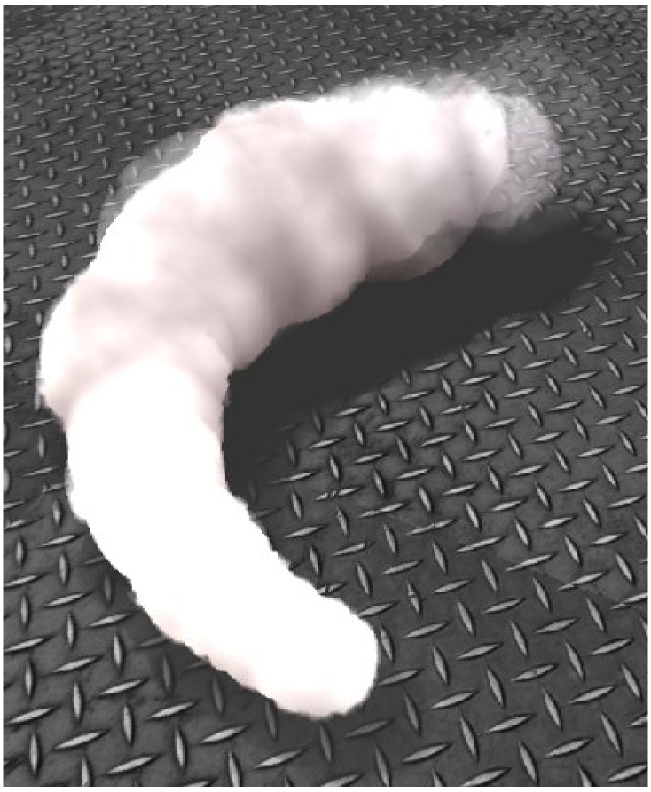


Image 18: This rendering shows the shadows from the smoke in the test application. The light is located to the upper left.

Since this smoke will most likely only be applied to the ground, since it's for a racing game. The comparison was skipped to simplify the solution. Instead, the density texture, in the light direction, were simply projected on the ground. The drawback with this method of doing shadows is that anything between the light and the smoke will get shadowed as well. But since this is not very likely in a racing game, the approximation was accepted.

Because of the used lighting algorithm, the smoke will self shadow with realistic soft shadows. This can be seen in Image 32 in Appendix.

3.3.8 Implementation

The smoke rendering consists of 9 passes.

Pass 1

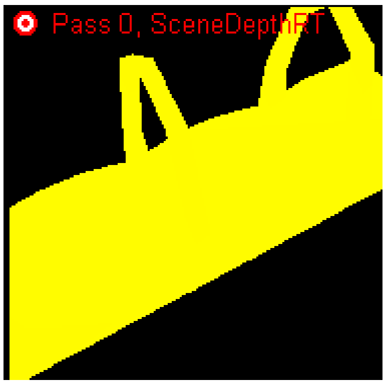


Image 19: This image shows the texture with the scene depth.

The first pass renders the scene depth to a texture. The scene depth is stored in screen space in a 32-bit float texture.

Pass 2

In the second pass, all particles are rendered with the instancing method described earlier. This is combined with billboarding, texture atlas and the fake sphere rendering method. The camera used is a camera that is located at the light position and looking in the light direction. The result is saved in a 32-bit float texture, and the following is the data saved for each fragment:

R-channel: Distance between the particle and near clip plane.

G-channel: Distance between the particle and far clip plane.

B-channel: not used

A-channel: Particle density

This in combination with the following blendmodes, BLENDOP_MIN for color channels and BLENDOP_ADD for alpha channels results in the following data will be stored in the final texture:

R-channel: Smallest depth (closest position) to the smoke volume.

G-channel: Largest depth (furthest position) to the smoke volume.

B-channel: not used

A-channel: Smoke density

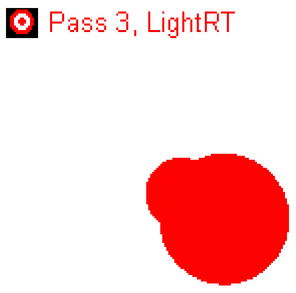


Image 20: This image shows the texture with the depths and density. Rendered from the lights perspective.

Pass 3

Same as Pass three but uses the normal view camera when rendering. Also, channel B is used to store life. Because of the MIN blend mode, the minimum life will be stored for the smoke at each screen position.

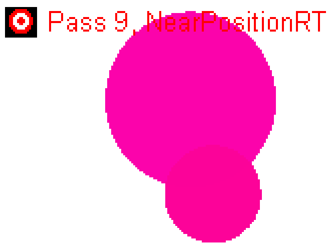


Image 21: This image shows the texture with the depths, density and life. Rendered with the normal view camera.

Pass 4,5

These two passes applies separable Gaussian Blur on the rendered texture from pass 2. This is done do soften the lighting and shadows.

Pass 5, LightRT_2

Image 22: This images shows the result after blurring the texture from pass 2.

Pass 6,7

These two passes applies separable Gaussian Blur on the rendered texture from pass 3. This is done do soften the lighting and shadows.

Pass 8

In pass 8, the scene is rendered normally to the screen. This pass can be rendered anytime during rendering and it can be split up in any number of additional passes.

Image 23: The scene rendered with simple light-mapping.

Pass 9

Shadows are rendered in this pass. Both from the smoke and from the scene. This can of course be separated in a real application.

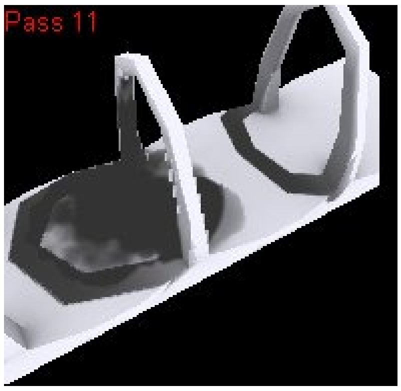


Image 24: Scene with projected shadows from both the smoke and the scene itself.

Pass 10

The final pass is the most important one. It's in this pass that the actual rendering of the smoke to the screen happens.

First, both far and near positions are reconstructed in both world space and view space.

Then, the volume is raytraced and for each sample, the scattering contribution is gathered and added to an accumulating shadowing value. This will give the resulting shadowing and shading of the smoke.

The alpha of the smoke is decided upon the life of the particle, the density, and the distance to the scene. The distance contribution is calculated as in soft particles, it's a simple fade when the volume is to close to the scene, to remove any visible intersections.

A color value for the smoke is sampled from an one dimensional texture, using life as the texture coordinate. This enables artist to easily create any color range for the system. The ambient term, sampled from a cubemap, is finally added to the resulting color.

Image 25: Final scene with the smoke rendered.

3.3.9 Particle simulation

The particles were simulated on the CPU as a basic particle system. The particles where emitted by an cone shaped emitter.

At spawn, all particles where given a random life-time, a random texture, a random speed and a random texture rotation angle.

3.3.10 Performance

Very little optimizations were done but performance was acceptable anyway. Following fps were measured on two different hardware:

GPU	Resolution	Average FPS
NVIDIA GeForce 8800 GT	800x800	85
NVIDIA GeForce 8800 GT	1280x1024	60
NVIDIA GeForce GTX 260	800x800	50
NVIDIA GeForce GTX 260	1280x1024	50

Because of the deferred nature of the algorithm, the amount of smoke on the screen doesn't affect the performance as much as in an ordinary particle system.

What could have been optimized more:

- rewrite shader code in a more optimized way
- use view-space instead of projection space depth
- try to step less times when collecting scattering contribution
- find optimal combination of size and amount of particles

- use the bounding box of the particle system instead of a fullscreen quad when blurring and rendering the final shader
- render the smoke to a texture smaller than the screen

3.3.11 Problem areas

The smoke rendering method described in this chapter have some serious problem areas that prevent an implementation in a real game. The two main problems are that dead particles cannot easily fade away and the camera is not allowed to be inside the smoke.

When the smoke particles die, they should fade away slowly. Fading the density is easy, since it can be multiplied with the life of the particle. But the position of the particle cannot be manipulated. There is no way to smoothly fade away the particle since it will either write the depth or it won't. There is nothing in between. This limitation was the main reason for no further development on this algorithm. The limitation makes the smoke flicker where it's close to the parts that's fading away. A solution not tested in this project, would be to use simply soft particles for the particles with low life and therefore are very transparent.

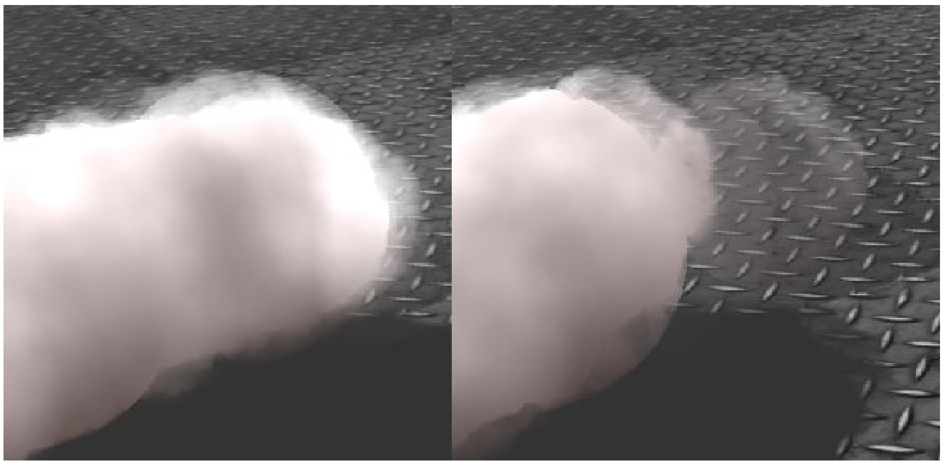


Image 26: These two images shows how the smoke fades away when it dies. There are only half a second between the images.

No work or research have been put into finding a solution to the problem with a camera inside the smoke as can be seen in image 27. The algorithm can currently not deal with this situation. But there are no known limitations that makes it impossible to adopt the algorithm to handle this case in a nice manner.

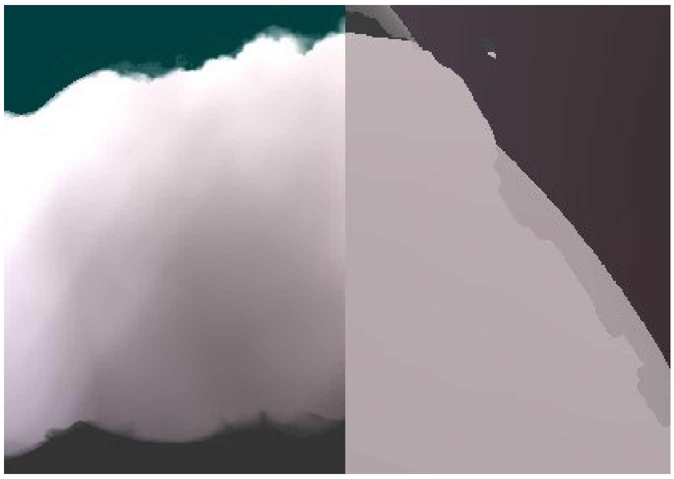


Image 27: The image on the left shows the smoke before moving the camera inside it. The image on the right shows how it looks like when standing in the middle of the smoke.

Another little limitation with the algorithm has to do with the approximation of the smoke volume. Since holes in the volume are filled by the approximation, this can give very wrong visual result sometimes as seen in Image 28. A solution to this is to divide the smoke rendering into many passes. So the smoke in the back is rendered separately from the smoke in the front. The drawback is that this would be slower and more complex.

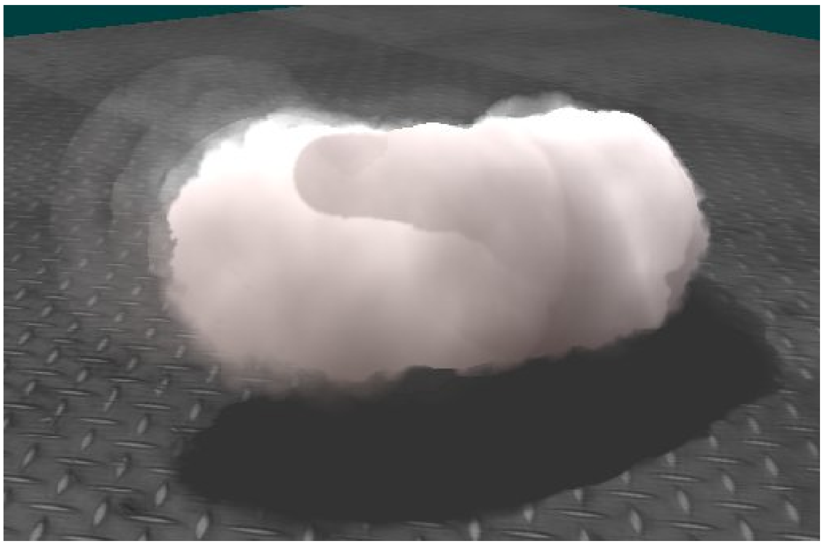


Image 28: Here we can see that the algorithm doesn't correctly handle the case when there is empty space between the smoke in the front and the smoke in the back. Therefore, the smoke in the back will give a dark silhouette.

This can be concluded in that the algorithm is not robust enough for a real implementation.

3.4 The Simbin implementation

Before the actual implementation into the rendering system that Simbin has, an evaluation took place of the tested smoke rendering methods. The best candidate were then tested and implemented in the renderer.

3.4.1 Old smoke rendering system

The old way to render smoke in the CUBE renderer was to use a simple particle system with emitters at the wheels. The particles where textured but this texture was neither rotated nor animated.

The particles were view-plane aligned on the CPU before rendered using either a shader or the fixed-function pipeline. Some particles could be lit by a single directional light. Since the particle system didn't have any soft particles implementation, it used very high alpha on the textures to make them nearly invisible to avoid any artifacts at edges. Also, all particles were spawned above the ground as Image 29 shows, to avoid the intersection with the ground. This cheat made the smoke look less realistic and hard to tweak for the artist, but at the same time avoided what could have been very noticeable artifacts.

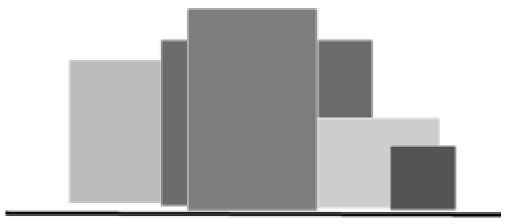


Image 29: The particle sprites (gray rectangles) are carefully placed hovering above the ground (black line) to avoid any intersection because the lack of a soft particles implementation.

The effects are loaded as reaction. A reaction could be smoke, or rain dust from the wheel. These reactions were stored in a simple data file for easy tweaking without recompilation of the source code.

3.4.2 Evaluation

Soft particles is an easy technique to render smoke with good quality and realism. It's stable and work on all modern gaming hardware and it's the method most games are using nowdays.

Mega particles is too unstable and low quality to be used.

The smoke with volumetric lighting is unfortunately to unstable to be used in a real product. There are still to many issues to solve. It's also quite demanding on the hardware.

Soft particles were chosen as smoke rendering method because of it's proven stability and simplicity.

3.4.3 Problems

The largest problem with the implementation was that the CUBE renderer didn't have any built in support for rendering to a texture. The engine did have a similar functionality but it was only developed for rendering sprites to a texture. A more general solution would have been good. Most engines today rely heavily on this functionality shadows, lighting and post-process effects.

Some time had to be spent on implementing a general render to texture functionality into the CUBE renderer.

Since the whole engine and game is undocumented, expect for source code comments, a lot of time had to be spent on searching, understanding and testing the code.

Much code in the the renderer were no longer used. To understand the real pipeline, this dead code were removed as the example below shows. In the example, the case that never does any work is commented out, and an assertion were added instead, to verify that this code never would be used.

```
assert(pScene->mParticleSystem.Begin() == NULL);
/*
if (pView->Flags() & CUBEVIEW_RENDERPARTICLES)
{
        cubeParticleSystem *ps = pScene->mParticleSystem.Begin();
        while (ps)
        {
                ps->Render (pScene, pView);
                ps = pScene->mParticleSystem.Next();
        }
}
*/
```

3.4.4 Result

The resulting implementation seen in Image 30 worked well and made it possible to increase the alpha of particles and make them more visible without ugly intersection artifacts.

The depth texture can in the future be the used for other post-processes like Screen Space Ambient Occlusion, Depth Of Field, Motion Blur, and Soft Shadows.

Only the render to texture functionality were borrowed from the test application.

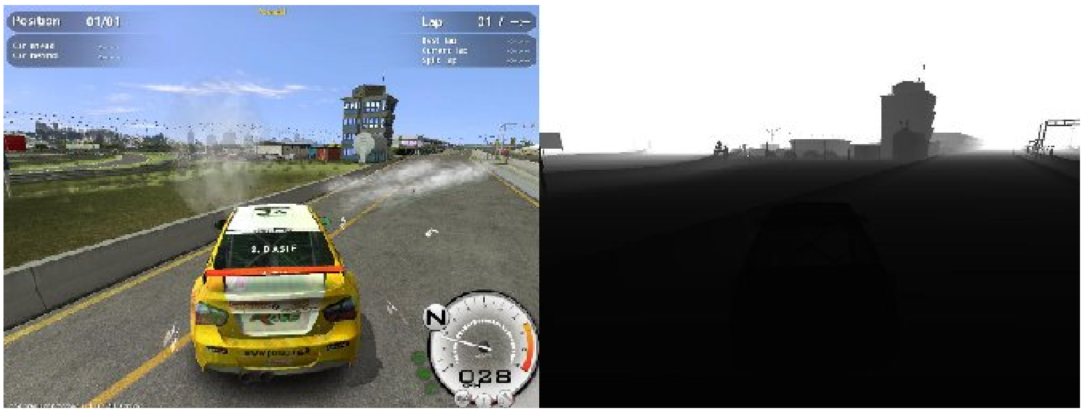


Image 30: The image on the right shows the rendered linear depth of the scene. The image on the left shows the same scene rendered with soft particles.

Not much difference can be seen in the game after the implementation of soft particles. But this was expected since this addition should be treated as a tool for game designers and artist. They should be able to improve the quality of the smoke in the game, because they can now allow intersections and increased alpha. Without the risk of visual artifacts. The soft particles implementation isn't restricted to smoke, but can also enhance other particle effects, like for example water and dirt splashes.

As can be seen in Table 1, the measured fps drop with soft particles instead of normal hard particles were only 5-10%. One reason the difference is so small, is that particles are rarely on the scene. The bottle-head in the pipeline could not have been in the pixel shader stage. The game is most likely CPU bound on the test setup. Even if the performance drop is slim, two optimization methods were tested.

The first one utilizes the prepass which renders the scene depth. This pass also writes to the z-buffer (must do so) and this can later be utilized when rendering the scene normally. Since the correct depth is already written to the z-buffer, the second pass can use the EQUAL z-comparator, and skip writing to the z-buffer. This could speed up the rendering if there is a lot of over-draw in the scene and the geometry uses complex shaders. But this optimization strategy didn't improve performance accordingly to measurements.

The second optimization strategy was to render the particles to an off screen texture with size ¼ of the full screen. If the pipeline were fillrate-limited, this would improve performance a lot. Unfortunately, the implementation only slowed down rendering, so the application was not fill-rate limited.

Method	**Setup**	**Min FPS**	**Max FPS**	**Average FPS**
Hard Particles	Fullscreen, 1280x1024, Max quality	102	416	212
Soft Particles, no optimizations	Fullscreen, 1280x1024, Max quality	104	384	194
Soft Particles, utilize prepass z-buffer	Fullscreen, 1280x1024, Max quality	105	376	190
Soft Particles, render particles to ¼ of fullscreen sized texture	Fullscreen, 1280x1024, Max quality	104	264	166

Table 1: This table shows the four methods of rendering particles that were tested and measured in the CUBE renderer.

3.5 Tool usage

This work could not have been done without the tools used. Especially RenderMonkey provided a good environment for fast prototyping of the shaders. Unfortunately, the tool is full of of bugs which slowed down development. There is so far no best option of IDE for shader development. Most probably since it's a new field of software engineering. Developing a high-quality commercial IDE for shader development might be a good business opportunity for a company.

One important feature lacking in RenderMonkey is that when using different cameras, they share view and projection matrix. So only one camera matrix can be used at a time. When doing for example shadow mapping, both current camera matrix and light camera matrix is needed in the same shader. This isn't possible without a workaround in RenderMonkey. To solve this, an additional pass was added that rendered the camera matrix to a 32-bit float texture. When this matrix had to be used, it was sampled from this texture. The solution is of course slow, but made it possible to develop without these restrictions of one camera only.

RenderMonkey is also very restricted in what data can be sent to the shader. No instancing is possible, and no user defined values per vertex.

Some bugs in RenderMonkey worth to mention are the following:

– View direction is wrong, it's not the same as:

  - normalize(viewPosition)

- The FX Exporter exports render states with mistyped names
- Undo doesn't always work

But even if it misses the mentioned features and have the bugs above, it's a very good IDE for shader development and prototyping.

The Microsoft DirectX10 SDK served as a good tutorial of how DirectX9 and DirectX10 should be best used. Some code from the SDK was reused in the test application in this decreased development time.

For implementation in the CUBE renderer, PIX was an invaluable tool that helped debugging most issues. The simplicity to debug single pixels, shaders and render targets was very useful. Without the tool, the implementation would have been much slower, since a debugging environment would have too be developed. The only thing PIX couldn't debug was timing issues between CPU and GPU. The reason is that PIX is rendering slower than the stand alone application, so a timing issue might disappear. During development of the Simbin implementation, one bug was only visible in the application, but not when run through PIX.

4 Conclusion

Three different methods to render smoke were evaluated. Soft particles proved to be a reliable, visually appealing method to render smoke. Mega particles is both hard for artist to control and suffers from a shower door effect. The smoke with volumetric lighting method wasn't developed to completion. Therefore, it was unstable and had visual artifacts. If more work were put in this method, it could be a good candidate for real use. The results of the three methods tested motivated the choice to use soft particles in the CUBE renderer.

The implementation of soft particles were quite easy. Although a flexible way of rendering to a texture were missing and had to be implemented as well. The result were a good looking smoke without any visible artifacts. Now, artist and game designers can use any alpha or sprites they want, without need to bother about intersections.

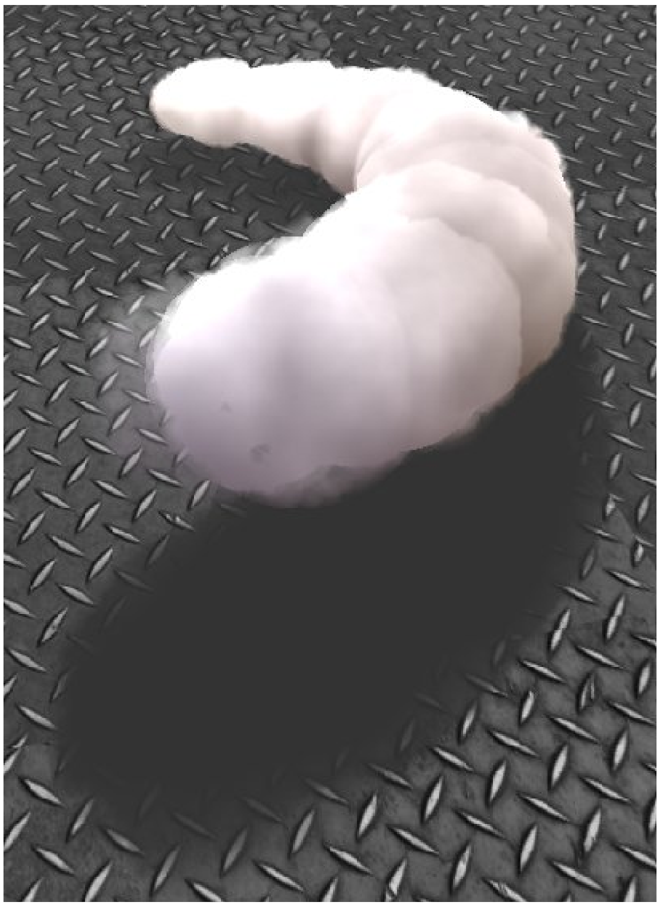


Image 31: The result of the smoke with volumetric lighting.

4.1 Future work

The smoke with volumetric lighting can probably be extended a lot with future work. The most stable solution might be a combination with soft particles. But this has to be tested and evaluated.

When taking samples inside the volume. Not much time have been spent on understanding the impact of the number of samples or where they are taken. It might be enough to take much fewer samples, if they are carefully positioned.

With the current limitations of smoke with volumetric lighting, it cannot be used in racing games. But maybe in other games or visualizations, for example volumetric clouds.

5 References

[1] T. Lorach, *“Soft Particles”*, NVIDIA, 17 January 2007, http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf, December 2009

[2] T. Umenhoffer, L. Szirmay-Kalos, G. Szijártó, *"Sperical Billboards and their Application to Rendering Explosions"*, Department of Control Engineering and Information Technology, http://www.iit.bme.hu/~szirmay/firesmoke.pdf , December 2009

[3] M. Krazanowski, *"A More Accurate Volumetric Particle Rendering Method Using the Pixel Shader"*, Gamasutra article, http://www.gamasutra.com/view/feature/3680/a_more_accurate_volumetric_.php, December 2009

[4] Microsoft, *"SoftParticles Sample"*, DirectX10 SDK (August 2009), http://msdn.microsoft.com/en-us/library/bb172449(VS.85).aspx, December 2009

[5] H. Bahnassi, W. Bahnassi, *“Volumetric Clouds and Mega Particles"*, http://www.inframez.com/events_volclouds_slide01.htm, December 2009

[6] M. Deering, S. Winner, B. Schediwy, C. Duffy, N. Hunt, *“The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics”*, ACM, New York, NY, USA, 1988

[7] W. F. Engel et. al, *"Shader X2 Introductions & Tutorials with DirectX 9"*, Wordware Publishing inc, 2004

[8] M. Pharr, et. al, *“Deferred Shading in S.T.A.L.K.E.R”* in GPU Gems 2, 2005

[9] M. Pettineo, *“Reconstructing Position From Depth, Continued”,* http://mynameismjp.wordpress.com/2009/05/05/reconstructing-position-from-depth-continued/, December 2009

[10] M. Mittring, *“Finding next gen: CryEngine 2”*, Crytek GmbH, http://ati.amd.com/developer/SIGGRAPH07/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf, December 2009

[11] “Improve Batching Using Texture Atlases”, NVIDIA SDK White Paper, July 2004, http://http.download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf, December 2009

[12] I. Ivanov, “Practical Texture Atlases”, Gamasutra article, http://www.gamasutra.com/features/20060126/ivanov_01.shtml, December 2009

[13] T. Akenine-Möller, E. Haines, N. Hoffman,*“Real-Time Rendering, Second Edition”*, 2008

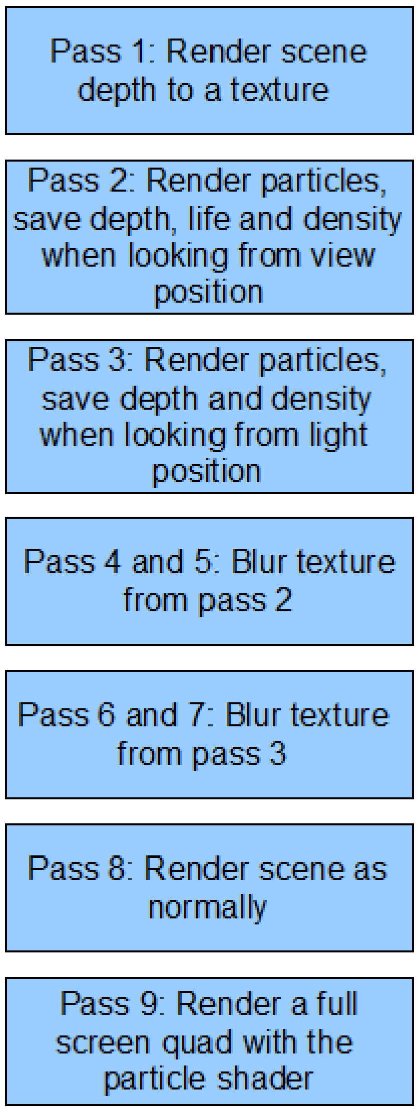
[14] A. R. Fernandes, *“Billboarding Tutorial”*, http://www.lighthouse3d.com/opengl/billboarding/billboardingtut.pdf, December 2009

[15] S. Premoze et al. “*Practical Rendering of Multiple Scattering Effects in Participating Media*”, Columbia University and Stony Brook University, Eurographics Symposium on Rendering (2004)

[16] A. Wiley, T. Scheuermann, “The Art and Technology of Whiteout”, SIGGGRAPH 2007 Tech Talk

6 Appendix

6.1 Passes when rendering smoke with volumetric lighting


Pass 1: Render scene depth to a texture
Pass 2: Render particles, save depth, life and density when looking from view position
Pass 3: Render particles, save depth and density when looking from light position
Pass 4 and 5: Blur texture from pass 2
Pass 6 and 7: Blur texture from pass 3
Pass 8: Render scene as normally
Pass 9: Render a full screen quad with the particle shader

6.2 Soft self-shadowing

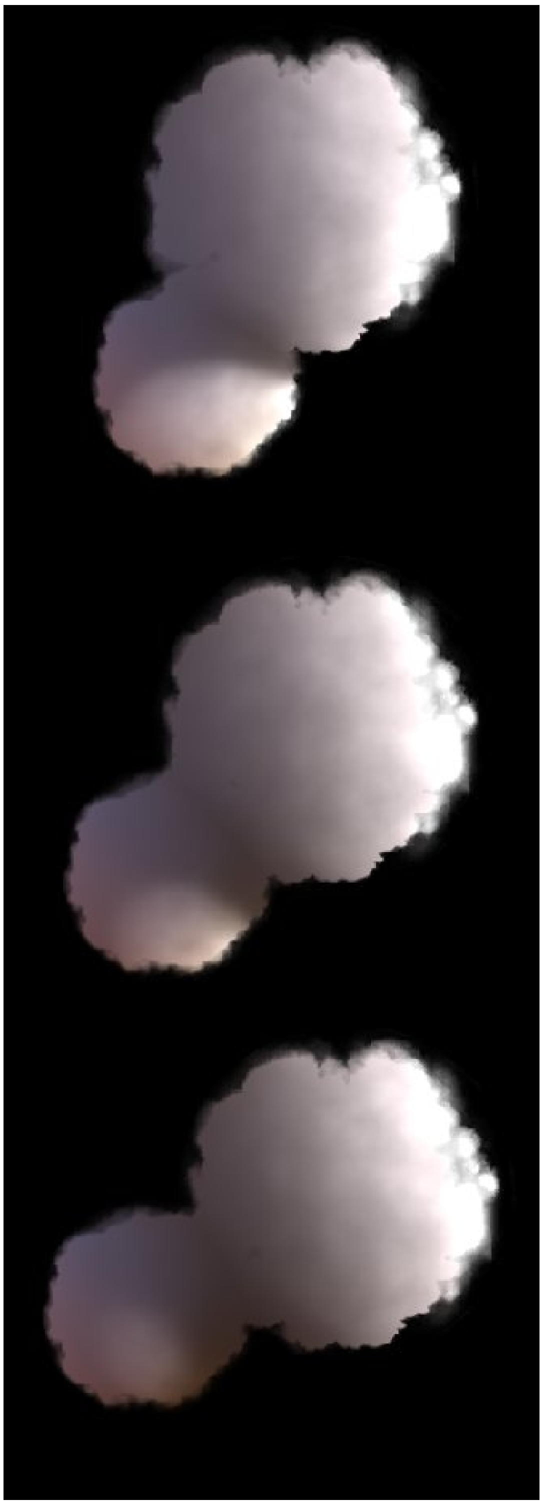


Image 32: These three images shows how the large ball realistically shadows the small one when it moves.

6.3 VPOS

Starting with DirectX Pixel Shader Model 3.0 there exist a new input type called VPOS. It's the current pixels position on the screen and it's automatically generated. This can be useful when sampling from a previously rendered texture when rendering an arbitrarily shaped mesh to the screen. To do this, we need uv-coordinates that represents where to sample on the texture. These coordinates can be gained by simply dividing VPOS with the screen dimensions.

When working with older hardware, that doesn't support shader model 3.0, there is a need to manually create the VPOS in the vertex shader and pass it to the fragment shader as a TEXCOORD. This is the way to do so (including the scaling to uv-range which manually has to be done for VPOS if you're using it).

Vertex Shader:

```
float4x4 matWorldViewProjection;
float2 fInverseViewportDimensions;
struct VS_INPUT
{
  float4 Position : POSITION0;
};
struct VS_OUTPUT
{
  float4 Position : POSITION0;
  float4 calculatedVPos : TEXCOORD0;
};
float4 ConvertToVPos( float4 p )
{
  return float4( 0.5*( float2(p.x + p.w, p.w - p.y) +
p.w*fInverseViewportDimensions.xy), p.zw);
}

VS_OUTPUT vs_main( VS_INPUT Input )
{
  VS_OUTPUT Output;
  Output.Position = mul( Input.Position, matWorldViewProjection );
  Output.calculatedVPos = ConvertToVPos(Output.Position);
  return( Output );
}
```

Pixel Shader:

```
float4 ps_main(VS_OUTPUT  Input) : COLOR0
```

```
{
  Input.calculatedVPos /= Input.calculatedVPos.w;
  return float4(Input.calculatedVPos.xy,0,1); // test render it to the screen
}
```

Image 33 shows an elephant model rendered with the shader above. As can be seen, the color (red and green channels) correctly represents the uv-coordinates for a fullscreen quad. Since 0,0,0 = black, 1,0,0 = red, 0,1,0 = green, 1, 1,0 = yellow.

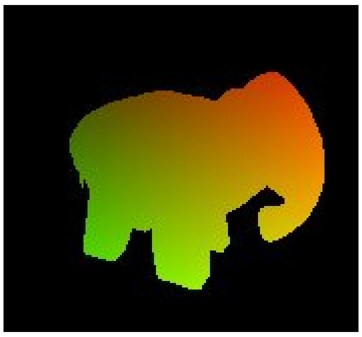


Image 33: This image shows an elephant model rendered using the manual vPos shader. As can be seen, the color (red and green channels) correctly represents the uv-coordinates for a fullscreen quad. Since 0,0,0 = black, 1,0,0 = red, 0,1,0 = green, 1, 1,0 = yellow.

6.4 Reference images

This is a part of the collection of reference images of thick white smoke collected before starting the work. All these images are creative common licensed, and allowed to be used commercially.

http://www.flickr.com/photos/racecarphotos/2870172056/

http://www.flickr.com/photos/ariander/3642486348/

http://www.flickr.com/photos/ericcastro/1573080019/

http://www.flickr.com/photos/iraxmas/1425419766/

6.5 Work in progress images

A collection of images taken during development, showing the progress.

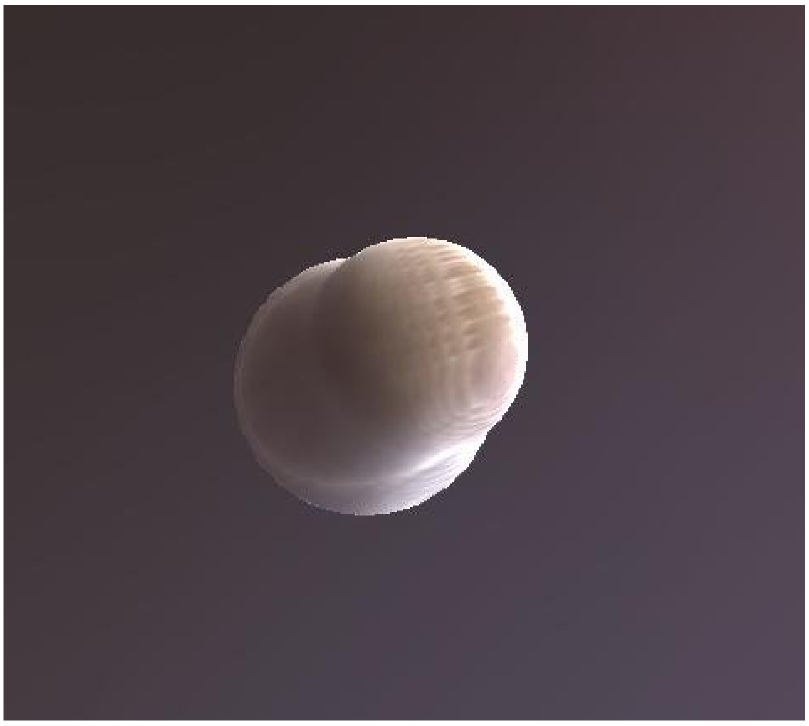


Image 34: One of the first images of the rendering of smoke with volumetric lighting. This image shows how the lighting is approximated by the cheating version of multiple scattering. This is before any noise or texturing is added.

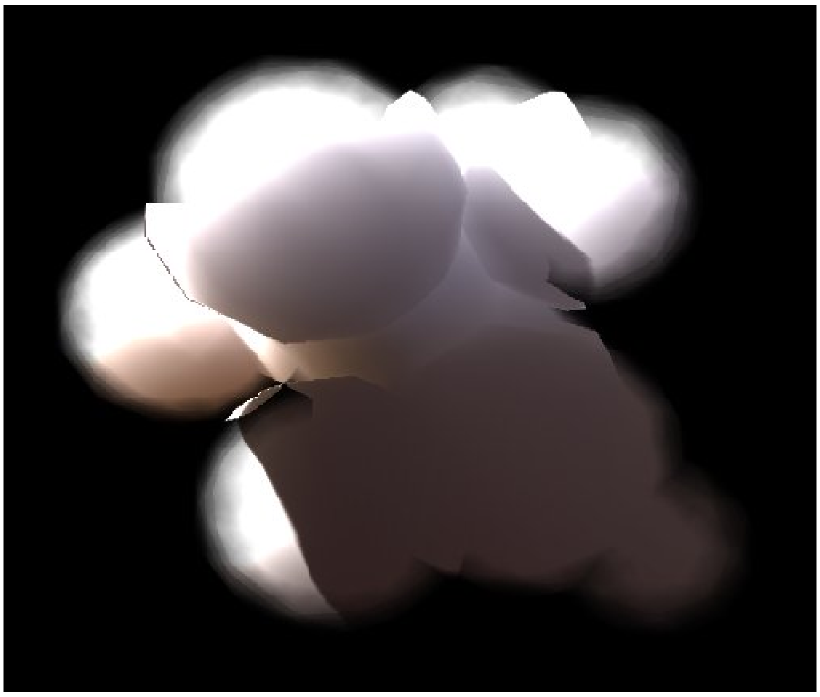


Image 35: This image demonstrate the first tests with multiple particles. As seen, there are still many artifacts that later in the development process were removed.

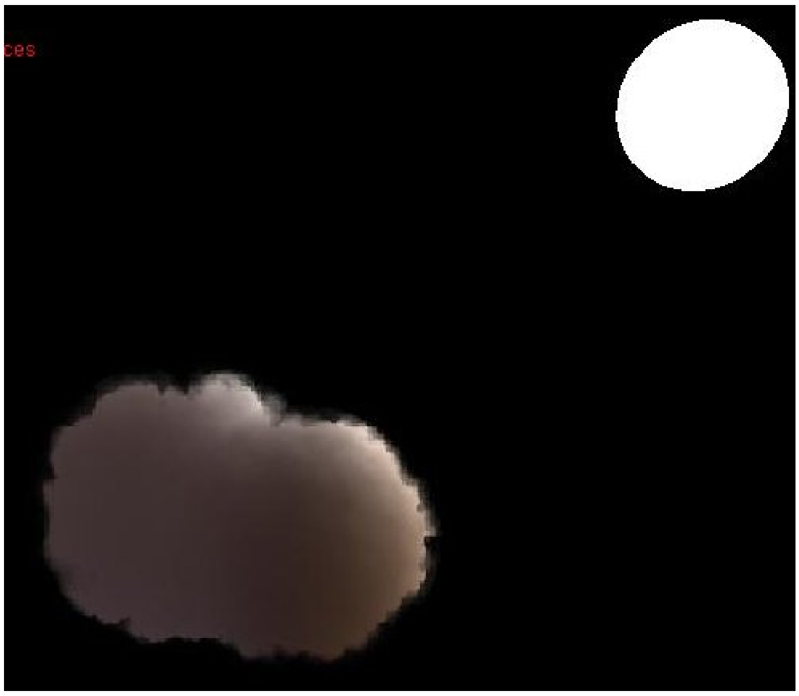


Image 36: This image shows the lighting of the smoke, when combining with density texture and noise.

Image 37: One of the first tests with objects intersecting the smoke. Note that there is a small vase inside the smoke.

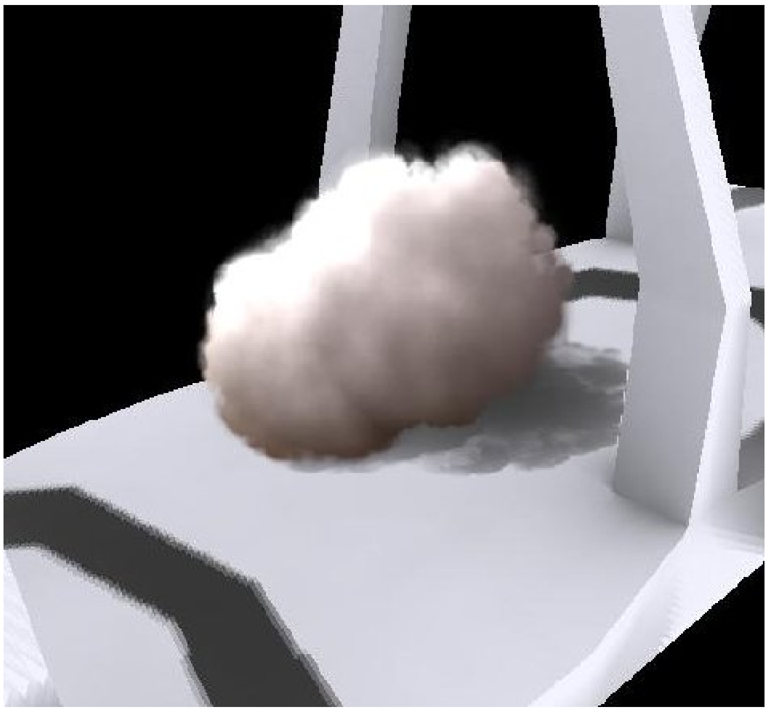


Image 38: Here shadows are tested. Both the scene and the smoke casts shadows. The smoke shadows are simply projected while the scene is using real exponential shadow mapping.

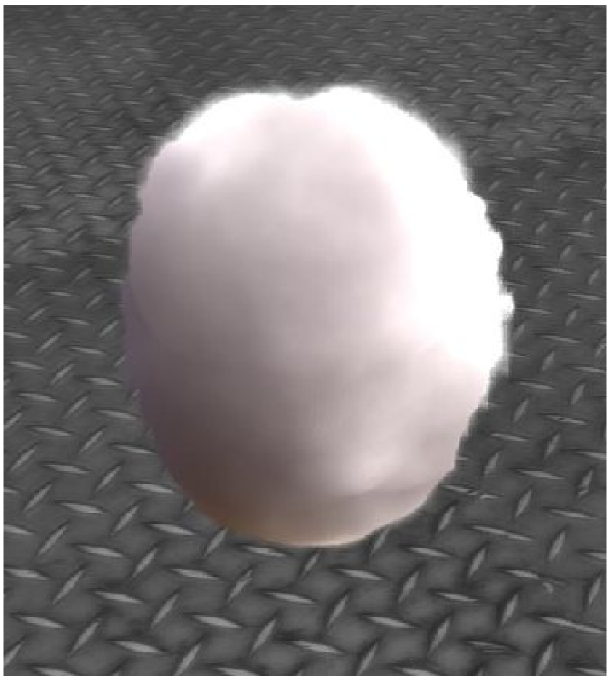


Image 39: First test of the smoke in the test application. Here, it's slowly rising as very thick white smoke.

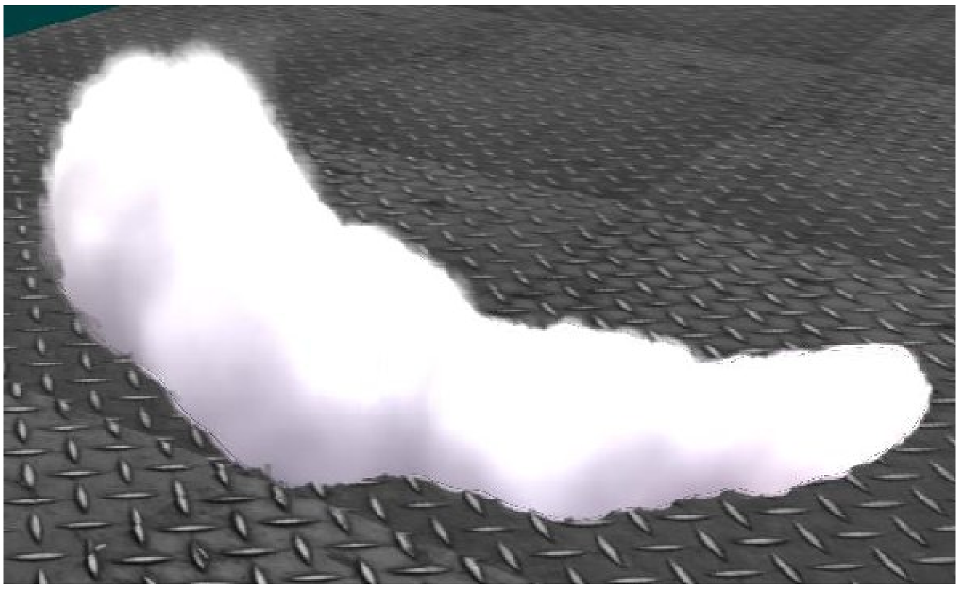


Image 40: This image shows that the smoke can be simulated realistically. Here it's spawned from a moving emitter that moves around in a circle.

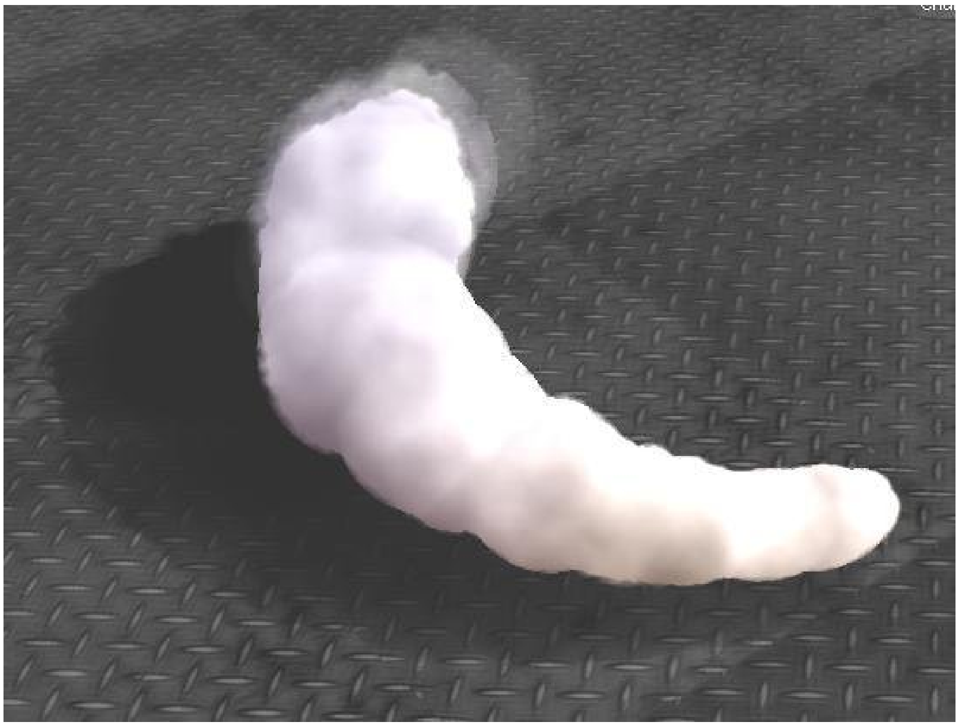


Image 41: Final render of the smoke, this time with the projected shadows from the smoke onto the ground.

6.6 Documentation of CUBE Renderer

The following is the documented behavior of the renderer, consisting of information gained from testing. It was used as a reference when implementing the final solution. There are three important phases in a game. The init sequence and the game loop which consist of an update sequence and a rendering sequence.

Init sequence

Game::Init(..) - *might multithread the game, otherwise just runs InitProc*

Game::InitProc(..) - *initializes all components of the game, including the gSpecialFX component*

SpecialFX::init() - *reads all the reactions from file (as settings), reads some reactions from terrain file, loads the reactions*

Reaction::Load(..) - *initializes the reaction and creates a particle system by calling a factory method*

cubeScene::CreateParticleSystem(..) - *creates a new particle system, and adds it's particles to the correct array*

cubeParticleSystem::Initialize(..) - *calls either initInternalParticle() or*

InitGMTParticle()

cubeParticleSystem::InitInternalParticle(data) - *inits the particle systems*

Render sequence

Game::Dyn()

Render::Dyn() - *handles dev. shortcut keys , render the scene*

cubeScene::Render(..) - *updates the lights, perform animations, renders the scene with the different viewports (cubeViews)*

cubeView::Render(...) - *render shadow maps, calls render objects*

cubeDirectX::RenderObjects(...) - *sets cam. parameters as (view, fov, clipplanes), sorts in buckets (called phases by cube) and renders them all*

Update sequence

Game::Dyn()

SpecialFX:Dyn() - *updates the special effects, including the reactions*

Reaction::Continue(float* currentVehicleCollisionAmount) – *called for each reaction, uses a switch to selected correct behaviour*

Reaction::Continue...(...) - *specific for each type of reaction (hardcoded in specialfx.cpp), updates position of the particles*