PARTICLE SYSTEM RENDERING
THE EFFECT ON RENDERING SPEED WHEN USING GEOMETRY SHADERS

Stefan Petersson
This thesis is submitted to the Department of Interaction and System Design at Blekinge Institute of Technology in partial fulfillment of the requirements for the Bachelor degree in Computer Science. The thesis is equivalent to 10 weeks of full time studies.
Contact Information:
Author: Stefan Petersson
Address: Folkparksvägen 12:10, 372 40 RONNEBY
E-mail: stefan.petersson@bth.se

University advisor: Per Jönsson
Department of Software Engineering and Computer Science
Address: Soft Center, RONNEBY
Phone: +46 457 38 58 15

Department of Interaction and System Design
Blekinge Institute of Technology
SE - 372 25 RONNEBY
Sweden

Internet: http://www.bth.se/tek/ais
Phone: +46 457 38 58 00
Fax: +46 457 271 25
ABSTRACT

It is a great challenge to develop a computer game. Today many games are developed in large game studios where lots of skilled people are working together. Everyone has to know what the final game should look like. Game designers are responsible for how the game should feel and look like. This also means that they decide if a programmer has to develop new techniques or not. Sometimes the game designers require lots of new techniques to be developed. Such a new technique may be rendering particle systems with a lot of particles in it. This is where this report will focus. To render particle systems it is necessary to know about the limitations there are in both hardware and software. Until today particle systems have been updated and calculated using the Central Processing Unit of the computer. With Microsoft Direct3D 10 there are new ways to render particles using Geometry Shaders. Geometry Shaders runs on the graphics card. This thesis focuses on testing rendering performance between using Geometry Shaders and not using Geometry Shaders.

A questionnaire was sent to Swedish game developers to get more information about relevant topics for investigation. A general answer was that Geometry Shaders always increase particle rendering performance. This thesis investigates if and when the statement is true or not. The hypothesis was obtained from the answers to the questionnaire.

Two test applications were used to investigate if the hypothesis was true or false. One test application has particle calculations on the CPU of the computer. The other test application has particle calculations on the GPU of the graphics card. Six different tests were done and the Geometry Shader approach went out to be the fastest in five of the tests. Since not all tests were faster than the CPU approach the hypothesis is not always true.

KEYWORDS

Particle system, DirectX, Direct3D, rendering, vertex shader, geometry shader, pixel shader
# TABLE OF CONTENTS

INTRODUCTION ................................................................................................................................. 7
  1.1 Background .................................................................................................................................. 7
  1.2 Methodology ............................................................................................................................... 8
  1.3 Hypothesis ................................................................................................................................... 8
  1.4 Research Objectives .................................................................................................................. 8
  1.5 Delimitations .............................................................................................................................. 9
  1.6 Acknowledgements ..................................................................................................................... 9

PROGRAMMING IN THREE DIMENSIONS USING DIRECT3D .................................................. 10
  2.1 Primitives ................................................................................................................................... 10
  2.2 Shaders ....................................................................................................................................... 11

PARTICLE SYSTEMS ....................................................................................................................... 15
  3.1 Particle Data ............................................................................................................................... 16
  3.2 Particle Rendering Techniques ................................................................................................. 17

QUESTIONS TO GAME COMPANIES ........................................................................................ 19
  4.1 Answer summary ........................................................................................................................ 20

TESTING, RENDER A PARTICLE SYSTEM ................................................................................... 22
  5.1 Test, Billboarded particles ........................................................................................................ 22
  5.2 Performance Testing .................................................................................................................. 26

DISCUSSION AND CONCLUSIONS .............................................................................................. 37
  6.1 Discussion ................................................................................................................................. 37
  6.2 Conclusion ................................................................................................................................. 38

BIBLIOGRAPHY ............................................................................................................................... 39

WEBSITES ........................................................................................................................................ 39

IMAGES ........................................................................................................................................... 39
APPENDIX A – HLSL SOURCE CODE, VS $\rightarrow$ GS $\rightarrow$ PS................................................................. 40

APPENDIX B – HLSL SOURCE CODE, CPU $\rightarrow$ VS $\rightarrow$ PS................................................................. 44

APPENDIX C – TEST APPLICATION STDAFX.H ........................................................................................... 47
INTRODUCTION

This chapter is an introduction to the thesis and my work. First I present the background information to the topic and this thesis in particular. Research methodology and hypothesis can be found in this chapter.

1.1 BACKGROUND

Particle systems are used in every modern game today. A particle system can be utilized to simulate a wide range of phenomena such as fire, smoke, rain or birds in the sky. You can read more about particle systems in Chapter 3. The use of particle systems will certainly not decrease in the future, since the concept gives the player a more realistic experience while playing the game. To become a good game developer you have to know what particle systems can do for you. It does not matter if you are a programmer or a game designer; everyone has to know what a particle system is and what it can do for the game.

Phenomena such as fire or smoke cannot be represented in a computer the same way they are represented in the real world. Real world fire, or the light energy released from real world fire is almost impossible to render using a computer; today’s computers are not fast enough to achieve this. Here developers have to come up with solutions to make it possible to map continuous fire to discrete computer fire. A good solution to render approximated fire is to use particle systems. Even if it was possible to render real world fire using a computer, it would not be necessary. That is because there is not high resolution enough on a computer screen to put billions of particles on.

When you understand the theory, then you have just scratched the surface of particle systems. The next step is to implement them into a game or a 3D engine. That is where this thesis takes off, to try to understand how particle system rendering works using different techniques. To understand this I want to explore what can be done in the future using Geometry Shaders. See Chapter 3 for more information about Geometry Shaders. To get a clear picture of today’s particle system usage, a questionnaire was sent to game companies. Read more about the questionnaire in Chapter 4.
The received answers are analyzed and presented in this thesis. The thesis contains theory and also some code examples to show how the theory can be mapped to real world examples. The code examples were implemented using the latest shader technology found in Direct3D 10. The results of this thesis can be used in 3D Game Programming courses at Blekinge Institute of Technology.

1.2 METHODOLOGY

The work described in this thesis is done in the form of an exploratory approach [Robson02]. The main focus is to find out how rendering performance varies when rendering particle systems. Tests are performed in Chapter 5 to investigate and find where the major performance differences are.

To get more specific information about particle systems to refer to, a questionnaire was sent to experienced game developers. Information about how and when game companies use particle systems today was important to my work. Hopefully this questionnaire approach may also generate new ideas for future research.

Benchmarking prototypes had to be developed to obtain required information. Without the benchmarked information no conclusions could be drawn. The prototypes had to be valid and as simple as possible to erase unwanted factors like algorithm overhead that may affect the benchmarking results.

1.3 HYPOTHESIS

- Rendering particle systems using Geometry Shaders is always faster than not using Geometry Shaders.

The hypothesis is based on the summarized answers from the questionnaire sent to Swedish developers, see Chapter 4. All developers thought that Geometry Shaders would always increase particle rendering performance.

1.4 RESEARCH OBJECTIVES

This thesis aims at investigating if Direct3D 10 using Geometry Shaders always makes it more efficient to render particle systems, seen from a performance point of view compared to not using Geometry Shaders with Direct3D 10. With performance I mean number of rendered screen images per second.
1.5 DELIMITATIONS

In Chapter 3 some common techniques to render particle systems are covered. There are more techniques available but not all of them are covered in this thesis. Focus stays on simple, but still valid, techniques.

All tests are implemented using the latest rendering software, called Direct3D. Other rendering software such as OpenGL is not covered in this thesis. Information in this thesis applies to Direct3D using the Programmable Pipeline. The Fixed Function Pipeline is mentioned but not covered in any detail, see Chapter 2.

To fully understand this thesis, basic knowledge about 3D programming using Direct3D and linear algebra is required. Knowledge about terms like Z-buffering and vector algebra is required.

1.6 ACKNOWLEDGEMENTS

I would like to send a special thank you to Björn Törnqvist and Niklas Westberg at Massive Entertainment. They helped me whenever I was in need of any help during the implementations and testing. I also want to say thank you to my supervisor, Per Jönsson for his constructive way of thinking, at all times.
This chapter presents an overview of how to program in three dimensions. The intention is not to discuss the topics in detail, but rather to introduce them and make the reader familiar with the concepts.

Since the introduction of the 3Dfx Voodoo card in 1995, the hardware has made significant technological leaps that have led to more advanced games on the market. The Voodoo card made it possible for developers to put more calculations on the graphics card instead of the central processing unit [3dfxWpedia]. Unfortunately the calculations the graphics cards were able to perform were limited.

Today there are two methods of modifying the graphic output. It is possible to use either the Fixed Function Pipeline (abbreviated FFP) or the Programmable Pipeline (also known as shaders). When rendering using Direct3D lots of internal calculations, for example lighting algorithms, are performed at every frame. The algorithms used by Direct3D version 9 and below, are already implemented in the Fixed Function Pipeline. The Fixed Function algorithms are ready to use by developers and cannot be replaced with custom algorithms. When only using the Fixed Function Pipeline no other algorithms may be used. Here the Programmable Pipeline makes it possible to override this limitation. Using the Programmable Pipeline, developers are able to override and write new algorithms instead of using the Fixed Function algorithms. To use the programmable pipeline, small programs called shaders have to be written. A shader is a small program that runs on the graphics processing unit (GPU). In Direct3D 10 there is no Fixed Function Pipeline, only the Programmable Pipeline may be used [GDD3D10]. Because the tests in Chapter 5 use Direct3D 10 only the Programmable Pipeline is used in this thesis.

2.1 PRIMITIVES

All 3D objects are built of vertices. A vertex is a point where two edges of a polygon meet. For example a triangle is built of three vertices [Figure 2.1] and a quad is built of two triangles [Figure 2.2]. Triangle primitives are commonly used when rendering 3D objects on a
computer. More complex 3D objects are used in computer games and may be built of thousands of triangles [Figure 2.3].

![Three vertices](image1)

**Figure 2.1 Three vertices**

![Quad (two triangles)](image2)

**Figure 2.2 Quad (two triangles)**

![Teapot built of many triangles](image3)

**Figure 2.3 Teapot built of many triangles**

### 2.2 SHADERS

Today most graphical calculations can be executed on the graphics card using shaders. A shader is a small program that is executed on the graphics card. Shader programs are often used to get more flexibility compared to the Fixed Function Pipeline. When using shaders and the programmable pipeline developers cannot rely on already implemented Fixed Functions. Learning 3D programming using shaders are harder than learning using the Fixed Function Pipeline. To use the programmable pipeline the graphics hardware has to support the wanted shader type. For example, to use Geometry Shaders [Section 2.2.2], the graphics hardware has to support Direct3D 10. Direct3D 10 hardware has a unified architecture which
means that there are no fixed hardware responsibilities. For example the graphics hardware may perform either Vertex Shader or Pixel Shader operations depending on what is needed [MSDN]. This hardware architecture has effect on the tests in Chapter 5.

The Geometry Shader is a new shader unit that was introduced in Direct3D 10. There are three shader units in total, Vertex Shaders (VS), Geometry Shaders (GS) and Pixel Shaders (PS). The differences between them are explained later in this chapter.

Every vertex in a scene is first passed to a Vertex Shader [Listing 2.1]. In the Vertex Shader a vertex is processed and then sent to the Geometry Shader or sent directly to the Pixel Shader [Figure 2.4]. How every vertex is processed is up to the developer.

```c
for( int vertex = 0; vertex < TotalVertexCount(); ++vertex )
{
    //Pass every single vertex in scene to a vertex shader
    PassVertexToCurrentVertexShader(mVertices[vertex]);
}
```

Listing 2.1 Example: send all vertices to Vertex Shader

Shader programs can be written using High Level Shading Language (HLSL) or in assembly language directly. High Level Shading Language code is compiled into assembly code. Using the High Level Shading Language, developers can think at the algorithm level and does not have to worry that much about hardware details compared to assembly programming. When using shader assembly programming developers have to think more about register allocation, register read-port limits and instruction co-issuing [Peeper04].

![Programmable Pipeline, simplified overview, step 1-3 does not use Geometry Shader but step A-D does.](image)

2.2.1 VERTEX SHADER

In Direct3D Vertex Shaders can be emulated in software. The calculations per vertex are done on the CPU instead of the GPU. It is the only shader unit that can be emulated in software [Luna06]. When using Vertex Shaders parts of the fixed-function-pipeline are replaced. Operations such as transformation and lighting have to be done by the shader programmer. By replacing this fixed process with a Vertex Shader, huge amount of flexibility is achieved because developers are able to write their own GPU programs. Each vertex in a scene is passed to a Vertex Shader when using the programmable pipeline. In Figure 2.5
three vertices are passed to a Vertex Shader and the shader rotates every vertex to the right. If all vertices in a 3D object are rotated equally, the entire object will be rotated.

Input to a Vertex Shader is one vertex. A vertex has to contain position data at least. More information, like color and normal vector may also be present in a vertex and are then also sent to the Vertex Shader. The final vertex is later sent to a Geometry Shader or directly to a Pixel Shader [Figure 2.4].

![Figure 2.5 Vertex Shader example](image)

### 2.2.2 GEOMETRY SHADER

In a Geometry Shader it is possible to append new geometry to a stream of vertices [Figure 2.6]. The stream is then passed on to Pixel Shader stage. The Geometry Shader is the newest shader unit available to developers. Earlier it was not possible to generate new geometry in a shader program. The tests in Chapter 5 are taking advantage of the Geometry Shader by appending new geometry to a stream.

Input to a Geometry Shader is the data sent as output from a Vertex Shader which can be one to six vertices depending on what developers want to do with the Geometry Shader [MSDN]. Output from a Geometry Shader is a set of vertices [GDD3D10].

![Figure 2.6 Geometry Shader example](image)

### 2.2.3 PIXEL SHADER

Pixel Shaders replace the Fixed Function multi-texturing stage, where more than one texture is mapped to a polygon. A polygon is called a “textured polygon” if a texture (image) is mapped to it. Pixel Shaders gives the ability to manipulate individual pixels directly [Luna06]. Due to the high flexibility it is possible to perform advanced algorithms per pixel, such as per pixel lighting.

Alpha blending is applied in the Pixel Shader. Alpha blending is when one pixel is rendered on top of another with transparency [Figure 2.7].
Input to a Pixel Shader are data used to calculate the final pixel color [Luna06]. The Pixel Shader outputs a pixel color [Figure 2.8].
This chapter focuses on general particle system information. A particle system is a collection of particles that are usually small. All particles often behave in a similar yet somewhat random manner. Particle systems can be used to create a more realistic experience when playing a game. In most cases particle systems are used in three dimensional games, but sometimes they are also used in two dimensional games. In Figure 3.1 two particle systems are present. From the volcano fire is emitting and in top of the fire system a smoke particle system is present.

It is important that a particle system is instanciable. Once a particle system is defined, it should be easy to create more than one instance of it. Particle systems are responsible for creating and destroying its child particles when ever needed. Since particle systems often contain many particles, it is important to design them well.

<table>
<thead>
<tr>
<th>Number of particles</th>
<th>Particle size</th>
<th>Sys. Size (~)</th>
<th>Instances</th>
<th>Total memory (~)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>224 bytes</td>
<td>110 kB</td>
<td>4</td>
<td>0.43MB</td>
</tr>
<tr>
<td>5000</td>
<td>224 bytes</td>
<td>1100 kB</td>
<td>4</td>
<td>4.3MB</td>
</tr>
<tr>
<td>50000</td>
<td>224 bytes</td>
<td>11000 kB</td>
<td>4</td>
<td>43.0MB</td>
</tr>
</tbody>
</table>

Table 3.1 Particle system statistics

As seen in Table 3.1, memory usage may become a major issue when designing particle systems. Detailed planned particle systems may increase overall system performance since
they are used often in games. A good rule of thumb is, “make the common case fast” [Bik05]. A common case in particle systems is updating all the particles.

A structure used to represent a particle used in Table 3.1 may look like Listing 3.1.

```c
struct Particle
{
  float Position[3];  //96 bytes
  float Direction[3]  //96 bytes
  float Speed;        //32 bytes
};

Listing 3.1 Example; particle structure
```

Using the structure in Listing 3.1 it is possible to model many types of particles, for example rain or snow. The position data information is used to position the particle in a 3D world. The direction data is used to move the particle in a predefined direction. The speed data is used to move the particle at specified speed. An example of how a snow particle system may look like using the structure is illustrated in Figure 3.2.

![Figure 3.2 Snow particles [DPSnow]](image)

3.1 PARTICLE DATA

A particle is an object that is usually modeled as a point mathematically. Every particle has its own values and properties. In computer graphics a particle is a rendered entity which may have any of the following properties:

- Velocity
- Acceleration
- Mass
- Color
- Texture
- Lifespan
Using these different properties is often enough for many of the effects you can think of. In theory there are no limits on how many properties a particle may have. It is important to understand that many properties take more memory.

### 3.2 PARTICLE RENDERING TECHNIQUES

There are common techniques to render particle systems [Walsh05]. Using an inappropriate approach may lead to visual limitations. The goal is to render beautiful particle systems without having to use many vertices. Below four techniques are listed; untextured primitives, point sprites, textured quads and particle objects.

The techniques listed below are only a few of the techniques available when it comes to particle system rendering. The techniques are listed in ascending order by visual quality. Better quality equals more calculations per frame.

#### 3.2.1 UNTEXTURED PRIMITIVES

When particle systems were used in games for the first time, they were rendered using this technique. There was simply not enough processing power to render textured polygons. Untextured primitives are just colored points, lines or triangles that are rendered. Today this technique may still be used when rendering particle systems, for example rain or snow where each particle is small and no texture is needed. This technique [Section 3.2.1] is identical to the technique “Textured Quads” [Section 3.2.3] except there is no texture mapped to the primitive.

#### 3.2.2 POINT SPRITES

This technique is not available anymore when using Direct3D 10, because it is part of the fixed function pipeline. Point sprites are rasterized as a single pixel but may cover more than one pixel on the screen [Luna06]. Therefore Point Sprites are lacking of flexibility compared to “Textured Quads” [Section 3.2.3] which are full 3D objects. It is possible to resize a point sprite. It is also possible to map a texture to a point sprite. Only one vertex is sent to the graphics card when rendering a point sprite [Luna06].

The reason to mention this technique even if it is not available anymore is because the tests in Chapter 5 use the same theory. The tests simulate point sprites technique by only sending one single vertex to the graphics card. The tests in Chapter 5 are not rasterized as single pixels though. The important thing is to know that only one vertex is sent to the graphics card.

#### 3.2.3 TEXTURED QUADS

Textured quads (planes) are very good when it comes to particle systems. It is easy to transform (translate, scale, rotate) textured quads because they are 3D-objects. It is common to keep the quads always facing the camera. The technique used to render a quad
facing the camera, is called *billboarding* [Walsh05]. The reason why billboarding is commonly used and needed is illustrated in Figure 3.3. If not using billboarding particles will not render correctly because the quad will not face the camera. Think of a quad with a tree texture mapped to it [Figure 3.4]. If the quad does not face the camera at all times then the tree will not be visible at all view angles. A billboarded tree will look the same from every view direction. Billboarding is used in the tests in Chapter 5.

The performance testing uses this rendering technique combined with some theory from point sprites [Section 3.2.2]. More information about the tests is presented in Chapter 5.

3.2.4 PARTICLE OBJECTS

Particle objects consist of entire 3D-objects or meshes. Every single particle may be a complete 3D-object. In the future this technique may be used more frequently, when the processing power allows it. Today this technique is not frequently used compared to the other presented techniques.

A usage example may be a particle system that represents birds in the sky, where every bird is a 3D object.
A questionnaire about particle systems has been e-mailed to twelve programmers in the Swedish game industry. Only programmers in industry, as opposed to academia, were selected to get more accurate answers. Some programmers work in the same game company. The companies I mailed are deliberately not listed here. I have personally contacted every person via e-mail. Unfortunately not everyone answered the questions, seven developers replied with answers and five did not answer at all.

The intention with the questions was to check what different developers think of the use of particle systems in the near future. Direct3D 10 introduced Geometry Shaders. All questions were deliberately targeted to Direct3D and the usage of Geometry Shaders. The first question is extra important. If one programmer answered no to the first question, then the rest of his answers were not taken as seriously as if he had answered yes to the question.

First all questions were sent to Björn Törnquist at Massive Entertainment for a pilot test. Björn is experienced and because he thought that the questions were relevant, they were sent to the rest of the programmers.

The following questions were sent to the programmers:

1. Have you ever implemented a particle system?
2. How important are particle systems in your games?
3. How many resources (CPU, memory etc.) do you allow your particle systems to use?
4. How do **you** think that next generation of particle systems will work (technically), why?
5. What do **you** want to be able to do with next generation of particle systems?
6. Geometry Shaders may be used to create new solutions when it comes to particle systems. If you were assigned to develop a particle system, how would you use Geometry Shaders if you had to?
7. Do you see a long migration period from Direct3D 9 to Direct3D 10, why?
8. Do you think that it is okay to use Direct3D 10 only, when starting a new game project or is it still necessary to support Direct3D 9 as well, why?
4.1 ANSWER SUMMARY

All separate answers are not presented; instead a summary of relevance is presented. Note that the important details are not modified from the original answers! All information is a summary of relevance only.

4.1.1 HAVE YOU EVER IMPLEMENTED A PARTICLE SYSTEM?

Yes! Everyone had implemented a particle system at some time. The answers to question 2-8 were easier to believe when this question was a yes.

4.1.2 HOW IMPORTANT ARE PARTICLE SYSTEMS IN YOUR GAMES?

Particle systems are very important since they are used to increase the level of realism in a game. Explosions and weather effects may all be implemented as particle systems. Good things about particle systems are that they are easy to implement and increase realism a lot compared to not using particle systems at all.

4.1.3 HOW MANY RESOURCES (CPU TIME, MEMORY ETC.) DO YOU ALLOW YOUR PARTICLE SYSTEMS TO USE?

There was no exact answer to this question since a lot of factors may influence the answer. In most cases particle systems were allowed to use a lot of memory. CPU time allowed was in one case about 10% of CPU maximum. Often game players were able to adjust the resource settings used by particle systems.

4.1.4 HOW DO YOU THINK THAT NEXT GENERATION OF PARTICLE SYSTEMS WILL WORK (TECHNICALLY), WHY?

Most important change in the future will be the step from calculations on the CPU to be mapped to the GPU instead. An entire particle system will be able to “live” its entire life in the graphics card using Geometry Shaders.

4.1.5 WHAT DO YOU WANT TO BE ABLE TO DO WITH NEXT GENERATION OF PARTICLE SYSTEMS?

Particle systems in upcoming game titles will probably look better to the human eye. Better looking particle systems will be possible by, for example, using more advanced shader programs. An example of more advanced shaders could be shaders that implement a dynamic level of detail, using Geometry Shaders.

To add advanced lighting algorithms or applied physics to particles will also increase the level of realism. As always developers wants to improve all parts of game rendering, particle systems are good candidates for doing this. Artists also have to get more influence when it
comes to particle systems. By developing better tools that are easy to use, artists will be able to create more attractive particle systems to a game.

4.1.6 **GEOMETRY SHADERS MAY BE USED TO CREATE NEW SOLUTIONS WHEN IT COMES TO PARTICLE SYSTEMS. IF YOU WERE ASSIGNED TO DEVELOP A PARTICLE SYSTEM, HOW WOULD YOU USE GEOMETRY SHADERS IF YOU HAD TO?**

Using Geometry Shaders will increase performance dramatically since less data has to be transferred through the system bus. Earlier the system bus has been the bottleneck when rendering lots of triangles. Geometry Shaders are able to append new geometry to a render stream and it is all done in the graphics card.

*The answers to this question led to the hypothesis [Section 1.3]! Everyone believed that the system bus always were the major bottleneck. The answers to the other questions were important too since they also focus on the same topic.*

4.1.7 **DO YOU SEE A LONG MIGRATION PERIOD FROM DIRECT3D 9 TO DIRECT3D 10, WHY?**

Probably it will take some time to migrate from Direct3D 9 to Direct3D 10 since most cards that support Direct3D 10 are very expensive at the moment and also Direct3D 10 require the Microsoft Windows Vista operating system to operate at all.

All games that support Direct3D 10 today also have full Direct3D 9 support as well. We will probably not see any Direct3D 10 only games in the near future.

4.1.8 **DO YOU THINK THAT IT IS OKAY TO USE DIRECT3D 10 ONLY, WHEN STARTING A NEW GAME PROJECT OR IS IT STILL NECESSARY TO SUPPORT DIRECT3D 9 AS WELL, WHY?**

No! All commercial games still have to support Direct3D 9 as well. There are not enough computers in the world that support Direct3D 10 yet. It may also be too expensive to support both Direct3D 9 and Direct3D 10 in the same game.
Based on the information received from game developers, some questions appeared. A major question is if and how much the Geometry Shader stage may boost particle rendering performance. One other question that appeared was how the system memory was affected by using Geometry Shaders. Unfortunately the answer to the second question cannot be answered because Nvidia has not released their memory tool for the GeForce 8800-series.

This chapter focuses on performance testing. The goal is to investigate, using a simple test, the performance difference between using Geometry Shaders and not using Geometry Shaders.

5.1 TEST, BILLBOARDED PARTICLES

Two test applications are written to render a particle system. One application does the billboard calculations [Section 5.1.2] on the CPU. The second application does the billboard calculation on the GPU using Geometry Shaders. Both applications monitor how many milliseconds it takes to render one screen image (frame) using different render states. The monitored information was saved to file on application exit.

The tests investigate how the rendering speed differs between doing billboard calculation on the CPU versus doing the billboard calculation on GPU. No memory usage could be monitored because the NVPerfHUD with Direct3D 10 support was not released by Nvidia at the time of the test sessions.

All tests render from 0 to 100000 particles during 100 seconds. Every second 1000 new particles are added to the scene. All particles are created at program startup, i.e. no overhead caused by particle creation time. Both versions of the application do the same thing except for the billboard calculation [Section 5.1.2].
The Geometry Shader-test only sends one vertex per particle to the graphics card [Figure 5.1]. The vertex contains the particle center position. In the Geometry Shader stage the vertex is used to calculate four new vertices that represent a particle. See Chapter 5.1.2 for more information about how the calculations are done.

![Figure 5.1 Test application technique using Geometry Shaders. No pre billboard calculation of the input vertices in the GS-test](image)

The CPU test sends six billboard calculated vertices (two triangles) per particle to the graphics card [Figure 5.2]. In the Geometry Shader-test the billboard calculation is done in the Geometry Shader. That is the only difference between the two tests.

![Figure 5.2 Test application technique not using Geometry Shaders. Pre billboard calculated input vertices are sent to the graphics card](image)

### 5.1.2 BILLBOARD CALCULATION USED

Every particle in the test is facing the camera (billboarded). How the calculation of the billboard is done is illustrated below. The position of a particle and the position of the camera is all that is needed to calculate a billboarded quad that may represent a particle in a game application. See [Appendix A] for High Level Shading Language implementation.
5.1.2.1 CALCULATE VECTOR FROM CAMERA TO PARTICLE CENTRE

First a vector from the camera to the particle centre is calculated [Figure 5.3]. Calculating the vector is done by subtracting the world space particle position from the world space camera position. To billboard, the particle should be rendered perpendicular to the CamToParticle vector.

5.1.2.2 CALCULATE PARTICLE RIGHT VECTOR

The right vector should be perpendicular to the vector calculated in the previous step. To get the right vector it is assumed that the world up vector is (0,1,0), since the positive Y is always up. To get the right vector the cross product of the CamToParticle vector with the up vector
is calculated. The result is a vector, RightVec, that is perpendicular to the camera’s look-at vector [Figure 5.4]. The calculated right vector is used to work out the boundaries of the particle quad.

Normalization of RightVec is done to give it a length of 1. After normalization, the vector is multiplied by the particle size divided by two [Listing 5.1]. The particle quad is extended away from the particle position in both directions. That is why half the size is used as scalar value after the vector is normalized.

\[
\text{NormalizedRightVec} = \text{Normalize(RightVec)}; \\
\text{ScaledRightVec} = \text{NormalizedRightVec} \times \text{ParticleSize} \times 0.5;
\]

Listing 5.1 RightVec calculation

### 5.1.2.3 CALCULATE FINAL PARTICLE

The particle is created from four vertices that are used to create two triangles. To find vertex one, follow the negative ScaledRightVec [Listing 5.1] to the left of the particle. Then move up half the particle size [Listing 5.2]. To find the second vertex of the particle, follow the positive ScaledRightVec to the right of the particle. Then move up half the particle size. To get vertex three and four, repeat the first steps and go down half the particle size instead on both sides. In Figure 5.5 all four corners of the particle are calculated.

\[
\text{Point LeftSide} = \text{Particle Centre Position} - \text{ScaledRightVec} \\
\text{Vertex1.x} = \text{LeftSide.x}; \\
\text{Vertex1.y} = \text{LeftSide.y} + \text{ParticleSize} \times 0.5; \\
\text{Vertex1.z} = \text{LeftSide.z};
\]

Listing 5.2 Billboarding calculation, get first vertex
Similar calculations are done for the other three vertices as well to get their correct coordinates.

5.2 PERFORMANCE TESTING

Test computer specification:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core2Duo E6600, 2,4GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>2GB, Corsair Dominator, PC6400</td>
</tr>
<tr>
<td>Graphics card</td>
<td>Gainward GeForce 8800GTS 640MB</td>
</tr>
<tr>
<td>HDD</td>
<td>Western Digital Raptor 37GB 10000RPM 16MB Cache</td>
</tr>
<tr>
<td>Operation system</td>
<td>Microsoft Windows Vista Ultimate</td>
</tr>
<tr>
<td>DirectX</td>
<td>Microsoft DirectX 10 (February 07 SDK)</td>
</tr>
<tr>
<td>Development</td>
<td>Visual Studio 2005 Professional</td>
</tr>
</tbody>
</table>

All tests were run in windowed mode at the resolution 1024x768. The computer was rebooted before every new test session. Each test renders using different values and render states. Render states that are changed are common states that affect Pixel Shader usage and performance. See screenshots where screenshot to the right has Z-test and Z-write enabled. When the Z-states are enabled rendering artifacts appear because of the alpha blended quads [Figure 5.6].

![Figure 5.6 Rendering artifacts, left particles have Z-enabled and right particles have Z-disabled (better result)](image)

From Section 5.2.1 to Section 5.2.6 the tests are presented. Each test contains information about test settings, rendering performance in milliseconds and an estimated workload overview of the test applications. Section 5.2.7 contains conclusions to each test. Information about how the estimated workloads are calculated is found in Section 5.2.7 as well. Section 5.2.8 contains screenshots from the tests.
5.2.1 TEST 1

The first test is simple and does not include any Z-buffer tests and alpha blending is activated. Every particle is written to the screen by the Pixel Shader since no Z-tests are performed. In practice this test setting could represent a rain particle system since the particles are small.

- Settings
  o Number of particles: From 0 to 100000
  o Particle size: From 0.05 to 0.1 units
  o Alpha blending: Yes
  o Texturing: Yes
  o Z-tests: No
  o Z-write: No

![Figure 5.7 Test 1 results](image1)

![Figure 5.8 Test 1 estimated workload](image2)
5.2.2 TEST 2

The second test changes some render states. Theoretically fewer calculations are performed by the Pixel Shader unit since the Z-test and Z-write are enabled. Test 2 is visually identical to Test 1.

- Settings
  - Number of particles: From 0 to 100000
  - Particle size: From 0.05 to 0.1 units
  - Alpha blending: Yes
  - Texturing: Yes
  - Z-tests: Yes
  - Z-write: Yes

---

**Figure 5.9 Test 2 results**

**Figure 5.10 Test 2 estimated workload**
5.2.3 TEST 3

The third test has same settings as Test 1 except the particle size values. More pixels are written by the Pixel Shader because each particle is larger than the particles in Test 1.

- Settings
  - Number of particles: From 0 to 100000
  - Particle size: From 1.0 to 2.0 units
  - Alpha blending: Yes
  - Texturing: Yes
  - Z-tests: No
  - Z-write: No

Figure 5.11 Test 3 results

Figure 5.12 Test 3 estimated workload
5.2.4 TEST 4

The fourth test has same settings as Test 2 except the particle size values. More pixels are written by the Pixel Shader because each particle is larger than the particles in Test 1.

- **Settings**
  - Number of particles: From 0 to 100000
  - Particle size: From 1.0 to 2.0 units
  - Alpha blending: Yes
  - Texturing: Yes
  - Z-tests: Yes
  - Z-write: Yes

![Figure 5.13 Test 4 results](image1)

![Figure 5.14 Test 4 estimated workload](image2)
5.2.5 TEST 5

The fifth test has same settings as Test 1 and Test 3 except the particle size values. More pixels are written by the Pixel Shader because each particle is larger than the particles in Test 1 and Test 3.

- **Settings**
  - Number of particles: From 0 to 100000
  - Particle size: From 5.0 to 10.0 units
  - Alpha blending: Yes
  - Texturing: Yes
  - Z-tests: No
  - Z-write: No

![Test 5 results](image-url)

![Estimated workload](image-url)
The sixth test has same settings as Test 2 and Test 4 except the particle size values. More pixels are written by the Pixel Shader because each particle is larger than the particles in Test 2 and Test 4.

- Settings
  - Number of particles: From 0 to 100000
  - Particle size: From 5.0 to 10.0 units
  - Alpha blending: Yes
  - Texturing: Yes
  - Z-tests: Yes
  - Z-write: Yes

---

**Figure 5.17 Test 6 results**

**Figure 5.18 Test 6 estimated workload**
5.2.7 CONCLUSIONS

This section contains conclusions to each test.

The estimated workloads, seen in for example Figure 5.8, are not measured using the test applications. They are estimated by me myself based on my own knowledge together with the DirectX documentation [MSDN] and are therefore not 100% accurate. The estimates are there to give the reader a clearer picture of how the workload is split among the different hardware units. Because Direct3D hardware uses a unified architecture the computational power available to each shader unit is not always the same. For example in Test 1 the particles are small and do not cover as many pixels as the particles in Test 5. In Test 5 the Pixel Shader unit has more to do than in Test 1. Based on this theory estimated load balance was calculated for each test.

All results will have linear results because of the linear increase of particle count in the scene. All particles are created at application start up and therefore no particles are ever created at run-time.

5.2.7.1 TEST 1 CONCLUSION

Because the particles are so small and only cover one single pixel, the Pixel Shader does not have to draw lots of pixels to the screen. Since the Z-test is disabled, all particles get rendered to the screen. Many particles are written to the same pixel on the screen. The only bottleneck here is the system bus that has to transport more vertices in the CPU test compared to the GPU test. Here the game developers are correct, the Geometry Shader solution is a lot faster here. Results and estimated workload are found in Figure 5.7 and Figure 5.8.

5.2.7.2 TEST 2 CONCLUSION

Because the particle size is very small the enabled Z-states have no major effect here. In practice this test are equal to the first test only because the particle size is small and only a small amount of particles are overlapping. Figure 5.9 and Figure 5.10 are almost identical to Figure 5.7 and Figure 5.8 from Test 1. As in Test 1 the Geometry Shader-test render faster because the major bottleneck is the system bus.

5.2.7.3 TEST 3 CONCLUSION

Because the Z-test and Z-write are disabled all pixels are written to the screen by the Pixel Shader. Still the Geometry Shader solution renders a lot faster than the CPU version. The major bottleneck is still the system bus and therefore the CPU test is still not as fast as the Geometry Shader-test. The Geometry Shader test does render slower in this test compared to Test 2. This is because the workload is now higher on the graphics card because the particles are larger and more calculations have to be done in the Pixel Shader.
The Pixel Shader workload may have an effect to the performance of the Geometry Shader since the graphics card has a unified architecture [Section 2.2]. When more pixels are written to the screen more GPU power are used by the Pixel Shader.

5.2.7.4 TEST 4 CONCLUSION

Because the Z-tests are enabled fewer pixels are written to screen by the Pixel Shader. See Figure 5.6 to see how the artifacts may look like because of that overlapped pixels are not rendered to the screen. Since less pixels are written to the screen it will lead to increased rendering speed, as seen in Figure 5.13. Compared to Test 3 this test does not have the same load on the Pixel Shader.

5.2.7.5 TEST 5 CONCLUSION

Now the particles are really large and the Z-tests are disabled which means that all pixels are written to screen by the Pixel Shader. This is where the Geometry Shader approach is not that powerful anymore since the graphics card now has to do more than before. The Geometry Shader is in the graphics card and is therefore affected by the overloaded Pixel Shader unit.

The stair-case look of the graph in Figure 5.15 is because the test applications do not measure accurate timings at very low rendering speeds. The code in Listing 5.3 is taken from the test applications. Both applications have the same code to measure time. The incorrect timings come from the first if statement where 1.0 seconds may not be enough. Increasing the delay to add particles more seldom may correct the stair-case look. Even if the stair-case look is fixed, the total performance measuring will not be any different; the lines will just straighten out.
5.2.7.6 TEST 6 CONCLUSION

When Z-tests are enabled the Pixel Shader is not as overloaded as in Test 5. Test 6 is the only test where the Geometry Shader approach is slower than the CPU approach. The CPU approach is more balanced in this test because there are more balanced calculations among the CPU and the GPU [Figure 5.18]. The Geometry Shader approach now has the most of its calculations on the GPU which makes the approach slower in this test compare to the earlier tests [Figure 5.18].

This test proves that only by using Geometry Shaders rendering performance will not always be better than not using Geometry Shaders. The test has also proven the hypothesis to be wrong.

Because lots of pixels are written to the screen a high load is put on the graphics card [Figure 5.18]. In practice complex Pixel Shaders may increase the gap between the two curves [Figure 5.17] even more. An important conclusion is that it is very important to know when and how to use Geometry Shaders to get better performance.
5.2.8 SCREENSHOTS FROM EACH TEST

![Figure 5.19 Screenshots from test applications when rendering 100000 particles](image)
DISCUSSION AND CONCLUSIONS

In this chapter discussion and thesis conclusions are presented. During the work new questions became available for future research. More about this is presented in the discussion part of this chapter.

6.1 DISCUSSION

This thesis is focus on particle rendering using Geometry Shaders versus not using Geometry Shaders. The hypothesis was not final until the answers to the questionnaire were analyzed and summarized; see Chapter 4 for more information.

The test results in Chapter 5 were not as expected from the beginning. The hypothesis claims that rendering particle systems using Geometry Shaders will always be faster than rendering without using Geometry Shaders. Rendering particles using Geometry Shaders were not always the fastest way to go. The sixth test in Chapter 5 did not fit into the hypothesis since the Geometry Shader approach was slower than the approach not using Geometry Shaders. In practice the hypothesis is true because games today more look like the third and fourth test seen in Chapter 5. The tests in Chapter 5 are only testing basic particle rendering. By using Geometry Shaders it is possible to generate more advanced particles such as spheres. Generating a sphere using Geometry Shaders instead of a quad [Figure 2.2], may lead to large speed and visual differences compared to the tests in Chapter 5. The Geometry Shader tests in Chapter 5 generate four new vertices from one vertex. A sphere could theoretically be generated from one vertex and a radius [Figure 6.1]. New tests using spheres instead of quads could be performed using the same approach as the tests in Chapter 5. Theoretically the results would be even more dramatic than the results seen in Chapter 5.

The goal was to test if and when the use of Geometry Shaders would render faster than not using Geometry Shaders. The main goal has been reached by the investigation but deeper and further investigations have to be done to get a more final answer to the question.

Further investigations could test the rendering performance using more advanced particle techniques. Volumetric particles generated on the GPU using Geometry Shaders are probably going to increase rendering performance even more than just render a quad for every particle, as in Chapter 5. Today volumetric particles using spheres are used more and more to increase the level of realism and are therefore very interesting to further investigate.
6.2 CONCLUSION

The hypothesis came from the answers to the questionnaire that was sent to twelve Swedish game developers [Chapter 4]. The answers also included information that showed that the migration from Direct3D 9 to Direct3D 10 may take some time. Direct3D is only supported in Windows Vista. It is therefore not possible to develop games only using Direct3D 10 and still make money from it; there are not enough Windows Vista users in the world yet to make it profitable [Section 4.1.8]. Some games may use both Direct3D 9 and Direct3D 10. That may lead to increased development costs because of the additional source code that has to be written to support them both.

Both test applications used in Chapter 5 worked well and were also easy to configure. Using the test applications I have been able to confirm that Geometry Shaders will improve particle rendering performance in many cases. In one test the hypothesis failed, the Geometry Shader test was slower than the non Geometry Shader test. The Geometry Shader test was slower because of the high load on the Pixel Shader unit [Section 5.2.7].

The results are general to simple particle systems using billboarded quads. It is still hard to say that Geometry Shaders always increase billboarded quad rendering performance. More tests have to be done using a variety of computer hardware and particle rendering techniques such as volumetric particles.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peeper04</td>
<td>Peeper C., Mitchell. J. L., Introduction to the DirectX® 9 High Level Shading Language, Microsoft Corporation, ATI Research</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WEBSITES</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDD3D10</td>
</tr>
<tr>
<td>MSDN</td>
</tr>
<tr>
<td>NVVertex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IMAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPSnow</td>
</tr>
</tbody>
</table>
High Level Shader Language source code used in the geometry shader tests. Only render states in the technique were modified between the different test sessions. Entire code were put in a single text file and compiled at application startup. Commented lines in the technique block are used to set different render states used in the tests in Chapter 5.

//--------------------------------------------------------------------------------------
// Particle.fx
// Direct3D 10 Shader Model 4.0 Particle System Demo
// Copyright (c) Stefan Petersson, 2007
//--------------------------------------------------------------------------------------

//--------------------------------------------------------------------------------------
// Input and Output Structures
//--------------------------------------------------------------------------------------
struct VSSceneIn
{
    float3 Pos : POS;
    float  Size : SIZE;
};

struct GSSceneIn
{
    float4 Pos  : POS;
    float  Size : SIZE;
};

struct PSSceneIn
{
    float4 Pos : SV_Position; // SV_Position is a (S)ystem (V)ariable that denotes transformed position
    float3 Norm : TEXCOORD0;  // World transformed normal
    float2 Tex  : TEXCOORD1;
};

//--------------------------------------------------------------------------------------
// Constant Buffers (where we store variables by frequency of update)
//--------------------------------------------------------------------------------------
cbuffer cbEveryFrame
{
    matrix g_mWorldView;
    matrix g_mProj;
    float3  g_CameraPos;
};

//--------------------------------------------------------------------------------------
// Textures and Samplers
//--------------------------------------------------------------------------------------
Texture2D g_txDiffuse;
sampler2D g_samLinear = sampler_state
{
    Texture = g_txDiffuse;
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
DepthStencilState DisableDepth
{
    DepthEnable = FALSE;
    DepthWriteMask = ZERO;
};

DepthStencilState EnableDepthTestOnly
{
    DepthEnable = TRUE;
    DepthWriteMask = ZERO;
};

DepthStencilState EnableDepth
{
    DepthEnable = TRUE;
    DepthWriteMask = ALL;
};

BlendState NoBlending
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = FALSE;
};

BlendState SrcAlphaBlendingAdd
{
    BlendEnable[0] = TRUE;
};

// VertexShader: VSScene
//----------------------------------------------------------------------
GSSceneIn VSScene(VSSceneIn input)
{
    GSSceneIn output;

    // transform the point into view space
    output.Pos = mul( float4(input.Pos,1.0), g_mWorldView );

    output.Size = input.Size;

    return output;
}
GeometryShader: GSBillboardParticle

Create quad that is facing the camera. Quad is created from particle center point. Shader will output 4 new vertices (2 triangles)

```cpp
void GSScene( point GSSceneIn input[1], inout TriangleStream<PSSceneIn> OutputStream )
{
    PSSceneIn output = (PSSceneIn)0;

    float3 particlePos = input[0].Pos;
    float3 camPos = g_CameraPos;
    float3 camToParticle = normalize(particlePos - camPos);
    float3 rightVec = normalize(cross(float3(0,1,0), camToParticle));

    float particleSize = input[0].Size; // 1.0f;
    float3 tmpRight = rightVec * particleSize * 0.5f;
    float3 leftSide = (particlePos - tmpRight);
    float3 rightSide = (particlePos + tmpRight);

    output.Pos.x = leftSide.x;
    output.Pos.y = leftSide.y - particleSize;
    output.Pos.z = leftSide.z;
    output.Pos.w = 1.0f;
    output.Tex = float2(0,1);
    output.Pos = mul( output.Pos, g_mProj );
    OutputStream.Append( output );

    output.Pos.x = leftSide.x;
    output.Pos.y = leftSide.y + particleSize;
    output.Pos.z = leftSide.z;
    output.Pos.w = 1.0f;
    output.Tex = float2(0,0);
    output.Pos = mul( output.Pos, g_mProj );
    output.Norm = -camToParticle;
    OutputStream.Append( output );

    output.Pos.x = rightSide.x;
    output.Pos.y = rightSide.y - particleSize;
    output.Pos.z = rightSide.z;
    output.Pos.w = 1.0f;
    output.Tex = float2(1,1);
    output.Pos = mul( output.Pos, g_mProj );
    output.Norm = -camToParticle;
    OutputStream.Append( output );

    output.Pos.x = rightSide.x;
    output.Pos.y = rightSide.y + particleSize;
    output.Pos.z = rightSide.z;
    output.Pos.w = 1.0f;
    output.Tex = float2(1,0);
    output.Pos = mul( output.Pos, g_mProj );
    output.Norm = -camToParticle;
    OutputStream.Append( output );
}
```
// Restart strip to begin a new strip
OutputStream.RestartStrip();

// PixelShader: PSSceneMain
//------------------------------------------------------------------------------
float4 PSScene(PSSceneIn input) : SV_Target
{
    return tex2D( g_samLinear, input.Tex );
}

//------------------------------------------------------------------------------
// Technique: RenderTextured
//------------------------------------------------------------------------------
technique10 RenderTextured
{
    pass p0
    {
        // Set VS, GS, and PS
        SetVertexShader( CompileShader( vs_4_0, VSScene() ) );
        SetGeometryShader( CompileShader( gs_4_0, GSScene() ) );
        SetPixelShader( CompileShader( ps_4_0, PSScene() ) );

        SetBlendState( SrcAlphaBlendingAdd, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
        // SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );
        // SetRasterizerState( NoCulling );
        // SetDepthStencilState( EnableDepth, 0 );
        SetDepthStencilState( DisableDepth, 0 );
        // SetDepthStencilState( EnableDepthTestOnly, 0 );
    }
}
High Level Shader Language source code used in the non geometry shader tests. Only render states in the technique were modified between the different test sessions. Entire code were put in a single text file and compiled at application startup. Commented lines in the technique block are used to set different render states used in the tests in Chapter 5.

//--------------------------------------------------------------------------------------
// Particle.fx
// Direct3D 10 Shader Model 4.0 Particle System Demo
// Copyright (c) Stefan Petersson, 2007
//--------------------------------------------------------------------------------------

//--------------------------------------------------------------------------------------
// Input and Output Structures
//--------------------------------------------------------------------------------------------
struct VSSceneIn
{
    float3 Pos : POS;
    float2 Tex : TEXCOORD0;
};

struct PSSceneIn
{
    float4 Pos : SV_Position; // SV_Position is a (S)ystem (V)ariable that denotes transformed position
    float2 Tex : TEXCOORD1;
};

//--------------------------------------------------------------------------------------
// Constant Buffers (where we store variables by frequency of update)
//-------------------------------------------------------------------------------
cbuffer cbEveryFrame
{
    matrix g_mWorldViewProjection;
    float3 g_CameraPos;
};

//--------------------------------------------------------------------------------------
// Textures and Samplers
//-------------------------------------------------------------------------------
Texture2D g_txDiffuse;
sampler2D g_samLinear = sampler_state
{
    texture=g_txDiffuse;
    Filter = MIN_MAG_MIP_LINEAR;
    AddressU = WRAP;
    AddressV = WRAP;
};

//--------------------------------------------------------------------------------------
// State Structures
//-------------------------------------------------------------------------------
DepthStencilState DisableDepth
{
    DepthEnable = FALSE;
}
DepthWriteMask = ZERO;

DepthStencilState EnableDepthTestOnly
{
    DepthEnable = TRUE;
    DepthWriteMask = ZERO;
}

DepthStencilState EnableDepth
{
    DepthEnable = TRUE;
    DepthWriteMask = ALL;
}

BlendState NoBlending
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = FALSE;
}

BlendState SrcAlphaBlendingAdd
{
    BlendEnable[0] = TRUE;
}

RasterizerState NoCulling
{
    CullMode = NONE;
}

//-----------------------------------------------------------------------------------------
// VertexShader: VSScene
//-----------------------------------------------------------------------------------------
PSSceneIn VSScene(VSSceneIn input)
{
    PSSceneIn output = (PSSceneIn)0;

    // transform the point into view space
    output.Pos = mul( float4(input.Pos, 1.0), g_mWorldViewProjection );

    output.Tex = input.Tex;

    return output;
}

//-----------------------------------------------------------------------------------------
// PixelShader: PSSceneMain
//-----------------------------------------------------------------------------------------
float4 PSScene(PSSceneIn input) : SV_Target
{
    return tex2D( g_samLinear, input.Tex );
}
technique10 RenderTextured
{
    pass p0
    {
        // Set VS, GS, and PS
        SetVertexShader( CompileShader( vs_4_0, VSScene() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PSScene() ) );

        // Set Blend State
        SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );

        // Set Rasterizer State
        SetRasterizerState( NoCulling );

        // Set Depth Stencil State
        SetDepthStencilState( DisableDepth, 0 );
    }
}
APPENDIX C – TEST APPLICATION STDAFX.H

Both test application included this stdafx.h to make sure that correct tests were performed.

```
#ifndef __STDAFX_H__
#define __STDAFX_H__

#include <windows.h>
#include <D3D10.h>
#include <D3DX10.h>
#include <string>
#include <vector>
#include <GfxStats/GfxStats.h>
#include <D3DUtil/D3DUtil.h>

#define SAFE_RELEASE(x) if (x) { (x)->Release(); (x) = NULL; }
#define SAFE_DELETE(x) if (x) { delete(x); (x) = NULL; }
#define PI (3.14159265358979323846f)
#define PARTICLE_SIZE_LOW 5.0f
#define PARTICLE_SIZE_HIGH 10.0f
#define NR_OF_LOG_DATA 100
#define MAX_PARTICLES_TO_RENDER 100000.0f
#define PARTICLES_TO_ADD_EVERY_NEW_SECOND 1000.0f
#define D3DUTIL D3DUtil::GetInstance()
#define GFXSTATS GfxStats::GetInstance()
#define DRAW_EMITTER_POS 0

#endif
```