

Master Thesis
Software Engineering
Thesis no: MSE-2004-20
Jan - Jun 2004



Performance evaluation of the fixed function pipeline and the programmable pipeline

Markus Holmåker, Magnus Woxblom

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:

Authors:

Markus Holmåker
Address: Ejdervägen 11, 372 73 Ronneby
E-mail: markus.holmaker@telia.com

Magnus Woxblom
Address: Krooks väg. 6, 372 30 Ronneby
E-mail: pt00mwo@student.bth.se

University advisor(s):
Lars Lundberg
Department of Software Engineering and Computer Science

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/ipd
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

When developing applications in Direct3D today, developers can choose between using the fixed function pipeline and the programmable pipeline. The programmable pipeline is more flexible than the fixed function pipeline, but what is the price for high flexibility? Is high flexibility desired at any cost? How is the choice of pipeline affecting performance? The purpose of this master thesis is to evaluate the performance of the two pipelines. This will be achieved by developing a benchmark program, which measures performance when various graphical effects are tested. The results of the evaluation will hopefully help developers to decide which pipeline to use, in terms of performance. In the end we will see that the fixed function pipeline is faster than the programmable pipeline in all our tests.

Keywords: Direct3D, “graphics pipeline”, “pixel shader”, “vertex shader”, performance

CONTENTS

1	Introduction	1
	1.1 Background	1
	1.2 Master thesis purpose and goal	1
	1.3 Master thesis method	1
	1.4 Reader's guide to the report	1
2	General 3D introduction	2
	2.1 Vertices	2
	2.2 Polygons	2
	2.3 Pixels	2
3	Direct3D introduction	4
	3.1 The Direct3D Graphics Pipeline	4
	3.2 The fixed function pipeline	4
	3.3 The programmable pipeline	5
	3.3.1 Vertex Shaders	6
	3.3.2 Pixel Shaders	6
4	The benchmark program	7
	4.1 What will be tested	7
	4.2 Why perform these tests?	8
	4.2.1 Vertex processing	8
	4.2.2 Pixel processing	8
	4.3 How will the tests be performed?	9
	4.4 Test setup	10
5	Result presentation	11
	5.1 Result presentation with tables	11
	5.2 Result presentation with graphs	12
	5.3 Result presentation with screenshots	13
	5.3.1 Directional light	13
	5.3.2 Pointlight	13
	5.3.3 Spotlight	13
	5.3.4 Vertex transformations	14
	5.3.5 Multitexture blending	14
	5.3.6 Fog	15
6	Result discussion	16
	6.1 Why are there visual differences?	16

6.2	Why is the fixed function pipeline always faster?	16
6.3	Why do the performance differences increase/decrease with complexity?	16
6.4	Why use the programmable pipeline?	16
7	Conclusion	18
	References	19
	Appendix A – Graphics pipeline	20
1	Coordinate spaces	20
2	The fixed function pipeline	20
2.1	Vertex processing	21
2.1.1	World transformation	21
2.1.2	Vertex blending	21
2.1.3	View transform	21
2.1.4	Vertex fog	21
2.1.5	Lighting and materials	21
2.1.6	Projection transform	21
2.2	Primitive processing	22
2.3	Pixel processing	22
3	Vertex shaders	24
3.1	The vertex shader virtual machine	24
4	Pixel shaders	25
4.1	Pixel shader virtual machine	25
	Appendix B – Effect code	27

1 INTRODUCTION

1.1 Background

Real time rendered 3D is today mostly used in computer- and video games, and virtually every great computer- or video game that is released today uses real time rendered 3D. The two most commonly used graphics standards for real time rendered 3D that exist today are OpenGL and Direct3D. Because of the new Direct3D features, vertex and pixel shaders in HLSL (High Level Shader Language) and the increasing support for it in graphics hardware, this master thesis will focus only on Direct3D.

In Direct3D today, there are two different ways of processing vertices and pixels: by using the fixed function pipeline or the programmable pipeline. In the programmable pipeline, vertex and pixel shaders replace the fixed vertex and pixel processing.

1.2 Master thesis purpose and goal

The purpose of this master thesis is to evaluate differences in performance between applications using the fixed function pipeline versus applications using the programmable pipeline. The goal of the evaluation is to come to a conclusion of what the performance differences are, measured in FPS (frames per second), between the vertex and pixel operations in the two pipelines. The outcome of the conclusion hopefully helps developers to decide which pipeline to use, in terms of performance.

1.3 Master thesis method

To carry out the performance evaluation a benchmark program will be developed. The program will use a static environment displaying a plane. Six different effects will be tested on the plane using three different levels of complexity. All effects will be implemented both with the fixed function pipeline and the programmable pipeline. For every effect, using different complexities and pipelines, the FPS will be measured. By comparing the results of the measurements, performance advantages or disadvantages when using the different pipelines will hopefully be revealed.

1.4 Reader's guide to the report

Chapter 2 gives a general introduction to 3D, explaining vertices, polygons and pixels. Chapter 3 describes the fixed function pipeline and the programmable pipeline in Microsoft Direct3D. A detailed description of the benchmark program is given in Chapter 4. Chapter 5 and 6 presents and discusses the benchmark results and Chapter 7 gives a conclusion of the evaluation.

2 GENERAL 3D INTRODUCTION

This chapter gives an overview of some of the most fundamental concepts of computer-generated 3D: vertices, polygons and pixels.

2.1 Vertices

Every object in the world of 3D, ranging from a simple square to a monster in a 3D game, is built up by something called vertices. A vertex is a point in a 3-Dimensional world [2, p. 4][7, p. 3]. This point is a one-dimensional object, meaning that it has neither height nor width nor depth.

Every point has a position attribute, which locates it in space. The position attribute of a vertex is actually three values, one X-coordinate, one Y-coordinate and one Z-coordinate [2, p. 4].

2.2 Polygons

Every vertex in a 3D object is a part of a polygon [7, p. 3]. “A *polygon is a geometrical figure with at least two sides, although three sides are necessary to represent a surface*” [4, p. 8]. Direct3D polygons have three sides, which means that a Direct3D polygon is a triangle. A triangle is a two-dimensional object, meaning that it has a height and a width, but no depth. A polygon is the most basic *renderable* object in the world of 3D. Hence, all 3D objects are built up from polygons. For example, a plane can be built from two polygons, as can be seen in Figure 1:

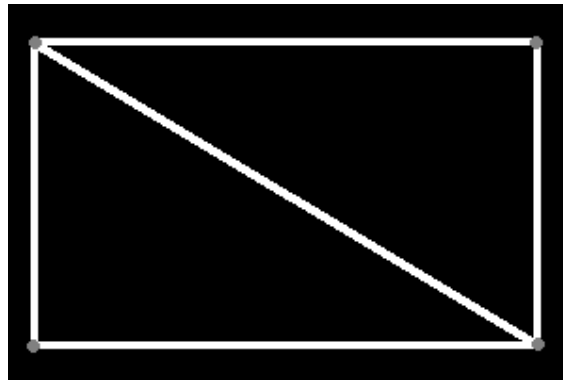


Figure 1. A plane made of two polygons.

In Figure 1, we see that a plane can be made of two polygons. The edges of the polygons in the figure are marked white, and the vertices that make up the polygons are gray. Note that each polygon consists of three vertices. However, a vertex can belong to more than one polygon; the vertex in the upper left corner of the plane in Figure A.1 belongs to both polygons in the plane.

2.3. Pixels

A pixel is a dot on the screen, and everything displayed on the screen is made of pixels. The plane in Figure 1 consists of four vertices, two polygons and a number of pixels. The number of pixels shown on the monitor is dependent on the screen resolution [4, p. 8]. Thus, the number of pixels in the plane depends on the resolution of the screen, but also the size of the plane. The larger the plane, and the higher

resolution, the more pixels are in the plane. Note that this does not affect the number of vertices and polygons. Figure 2 shows the same plane, but this time the pixels are shown instead of the vertices and polygons:

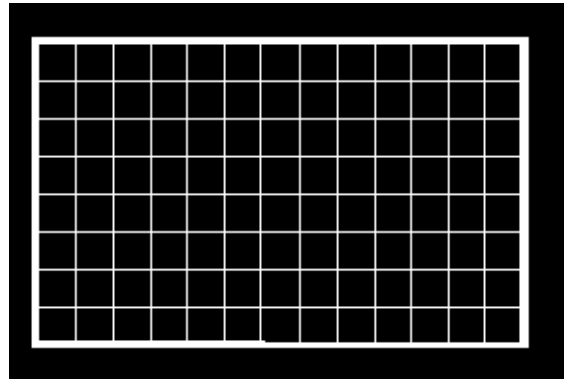


Figure 2. A plane with its pixels.

Figure 2 shows a plane, where every black square is a pixel. This plane is enlarged, because on the screen the pixels are much smaller. The plane is still divided into two polygons, and it is still made up of four vertices. However, vertices and polygons are abstract concepts; when a polygon is to be rendered, primitive processing replaces it with pixels that are visualized on the screen [10].

3 DIRECT 3D INTRODUCTION

This chapter gives an overview of the Direct3D structure as it is today, with the fixed function pipeline and the programmable pipeline. For a more detailed explanation of the pipelines, see Appendix A. If you are already familiar with Direct3D and its structure, you can skip this introduction.

3.1. The Direct3D Graphics pipeline

The Direct3D graphics pipeline consists of three major parts [8]. However, there are two of these parts that are particularly important for this thesis: vertex processing and pixel processing. Vertex processing is a series of vertex manipulating operations, where vertex data can be altered as desired [9]. Pixel processing is, as vertex processing, a series of pixel manipulating operations, where pixel data can be altered [10]. Hence, the vertex processing input is primitive vertex data, and the input to pixel processing is rasterized pixels [10].

Before vertex processing and pixel processing, there is a part called pre-processing, and in between there is also one part called primitive processing. All vertex data, created in the 3D application, has to pass through all this processing before something can be displayed on the screen. After vertex processing, the vertex data cannot go back to pre-processing, after primitive processing the vertex data cannot go back to vertex processing etc. Because of this, all these processing parts together constitute a pipeline; the Direct3D Graphics pipeline. The graphics pipeline can be seen in Figure 3.

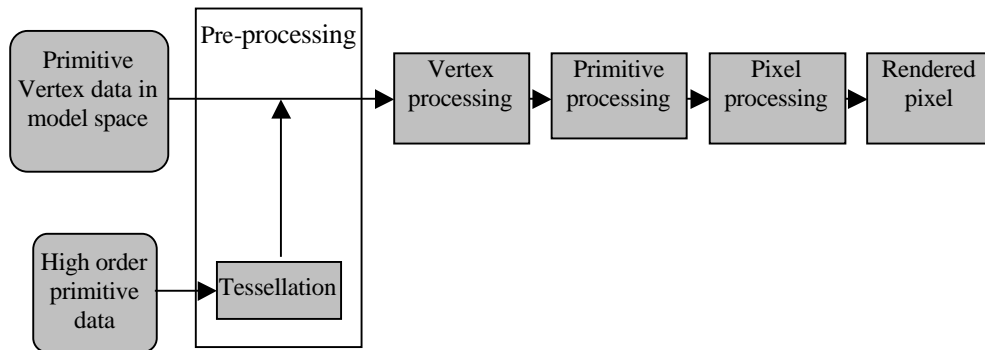


Figure 3. The Direct3D Graphics Pipeline [2, p. 4]

This thesis will focus only on vertex processing and pixel processing, but for completion both vertex processing, primitive processing and pixel processing are further described in Appendix A.

3.2 The Fixed function pipeline

In the early days of Direct3D programming (before DirectX 8), the only way to display 3D graphics on the computer screen was to use the *Fixed function pipeline* [9]. In the fixed function pipeline, vertex- and pixel processing provides a number of functions for the vertex and pixel manipulating operations. These functions cannot be altered, they are *predefined* or *fixed*, hence the name ‘fixed function pipeline’ [1, p. 267].

The available predefined vertex processing operations are shown in Figure 4:

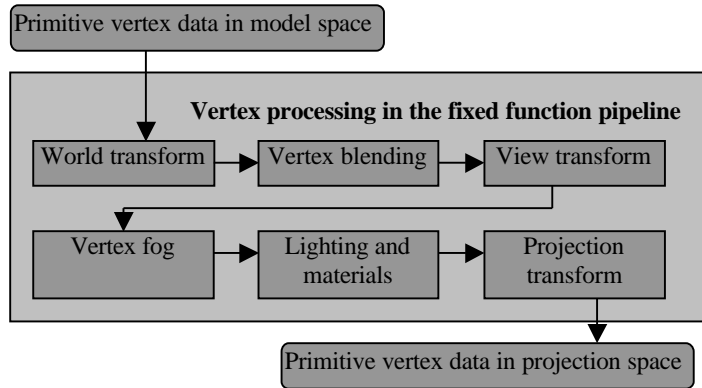


Figure 4. Vertex processing in the fixed function pipeline.

The available operations in Pixel processing (Part 1) can be seen in Figure 5:

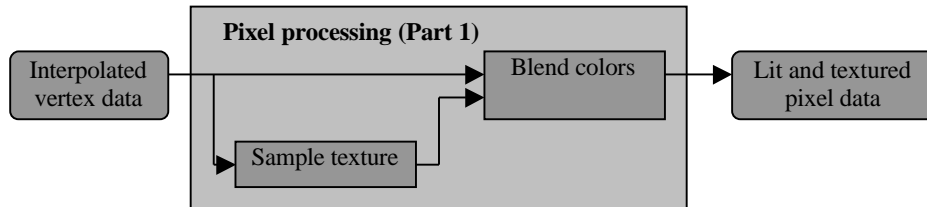


Figure 5. Pixel processing (Part 1) in the fixed function pipeline.

For more information about the fixed function pipeline, see Appendix A.

3.3 The Programmable pipeline

With the release of DirectX 8.0, something very exciting happened in Direct3D; Microsoft introduced the *Programmable pipeline* [3, p. 4]. The idea behind the programmable pipeline is that developers can replace the vertex-manipulating operations and some of the pixel-manipulating operations in the fixed function pipeline with his/her own written programs. This makes the vertex- and pixel processing much more flexible [1, p. 293]. These programs are called shaders, and were first written in a shader specific assembly language. The parts that can be replaced by the programmable pipeline are shown in Figure 6.

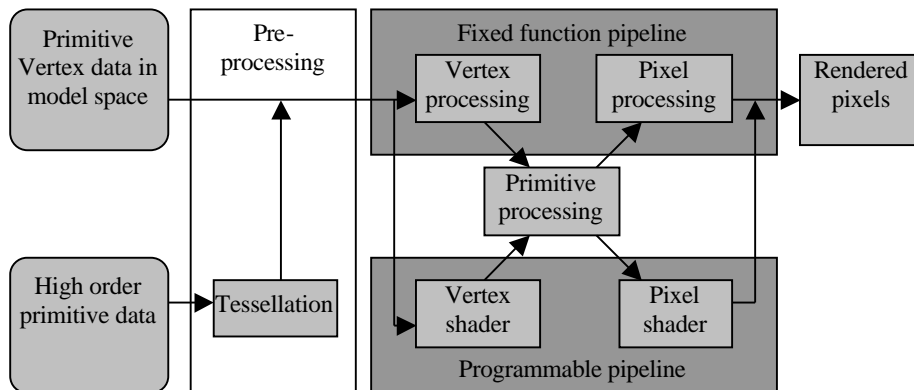


Figure 6. The Direct3D fixed function pipeline.

In December 2002, with the release of DirectX 9.0, Microsoft introduced a simpler shader language called HLSL, or High Level Shader Language [5, p. 17]. HLSL is a C like language, which, compared to shader assembly, is claimed to be easier to write, more readable and easier to maintain [2, p. 109]. Since HLSL code is actually compiled into shader assembly code [2, p. 3], there is no actual difference in performance between writing shaders in HLSL or shader assembly. Therefore the test cases evaluated in this thesis will build solely on HLSL code.

3.3.1 Vertex shaders

The first part of the programmable pipeline consists of vertex shaders. A vertex shader is a relatively small program, written in HLSL or shader assembly, which runs on the GPU (Graphics Processing Unit, the processor on the graphics card) [1, p. 293]. The vertex shader executes once for each vertex that passes through the graphics pipeline [5, p. 1], and the vertex shader input is per vertex data, i.e. position data for the current vertex, texture coordinates etc. A vertex shader has exactly one vertex as input and one vertex as output, it can never delete a vertex nor can several vertices be operated at the same time [5, p. 4]. Vertex shaders replace all vertex-processing operations in the fixed function pipeline, for example vertex transformations, vertex fog, lighting and materials [2, p. 5]. This can also be seen in Figure 4. For detailed information about the vertex processing operations, see Appendix A.

Instead of letting the fixed function pipeline handle the vertex manipulating operations such as transformation, developers can now write their own shaders to do these transformations. This makes the vertex processing a lot more flexible than before. Detailed information about vertex shaders can be found in Appendix A.

3.3.2 Pixel shaders

The rest of the programmable pipeline is called pixel shaders. As with a vertex shader a pixel shader is a program, written in HLSL or shader assembly, which is executed on the GPU. The purpose of a pixel shader is to calculate per-pixel colors, which is then returned to the fixed function pipeline for rendering. A pixel shader is usually much more time consuming than a vertex shader. This is because a pixel shader executes once for each pixel that is affected by rendered vertex data [2, p. 67].

From the vertex processing, per-vertex data is sent to primitive processing. Here, the vertex data is interpolated and passed on to pixel processing [2, p. 68]. The pixel processing is split in two parts; a pixel shader can replace the first part, while part two is still executed in the fixed function pipeline. The first part of pixel processing consists of texture sampling and color blending, as can be seen in Figure 5 [2, p. 69]. For detailed information about primitive processing, pixel processing and pixel shaders, see Appendix A.

By letting developers program their own pixel shaders that replace part one in the pixel processing, great flexibility can be obtained on effects such as texture manipulation.

4 THE BENCHMARK PROGRAM

The third part of the first chapter says: “*To carry out the performance evaluation a benchmark program will be developed...*” This chapter will describe the benchmark program, and provide an answer to these questions:

1. What will be tested?
2. Why perform these tests?
3. How will the tests be performed?

4.1. What will be tested?

In the evaluation of the fixed function pipeline and the programmable pipeline, there are some effects we think are especially interesting, including lighting, vertex transformation, multitexture and fog. We also thought it would be interesting to see if scene complexity affects the performance differences. Low complexity in our benchmark program corresponds to 1250 triangles in the scene, medium to 5000 triangles and high to 20,000 triangles. The 39 test cases are listed in Table 1:

	Complexity	Fixed function pipeline	Vertex shader	Pixel shader
Directional light	Low	Test case 01	Test case 02	-
	Medium	Test case 03	Test case 04	-
	High	Test case 05	Test case 06	-
Point light	Low	Test case 07	Test case 08	-
	Medium	Test case 09	Test case 10	-
	High	Test case 11	Test case 12	-
Spot light	Low	Test case 13	Test case 14	-
	Medium	Test case 15	Test case 16	-
	High	Test case 17	Test case 18	-
Vertex transformations	Low	Test case 19	Test case 20	-
	Medium	Test case 21	Test case 22	-
	High	Test case 23	Test case 24	-
Multitexture blending	Low	Test case 25	-	Test case 26
	Medium	Test case 27	-	Test case 28
	High	Test case 29	-	Test case 30

	Complexity	Fog	Fog Blending
Test case 31	Low	Fixed f. pipeline	Fixed f. pipeline
Test case 32	Medium	Fixed f. pipeline	Fixed f. pipeline
Test case 33	High	Fixed f. pipeline	Fixed f. pipeline
Test case 34	Low	Vertex shader	Fixed f. pipeline
Test case 35	Medium	Vertex shader	Fixed f. pipeline
Test case 36	High	Vertex shader	Fixed f. pipeline
Test case 37	Low	Vertex shader	Pixel shader
Test case 38	Medium	Vertex shader	Pixel shader
Test case 39	High	Vertex shader	Pixel shader

Table 1. Test cases measured in the benchmark program.

4.2. Why perform these tests?

As described in Chapter 3, it is the vertex processing and the pixel processing in the fixed function pipeline that can be replaced by vertex and pixel shaders. The vertex- and pixel processing operations can be seen in Figure 4 and 5. All of these operations can be tested in both the fixed function pipeline and the programmable pipeline, the performance of the two pipelines can be measured and the differences can be evaluated. The operations that will be tested are; lighting, vertex transforms, fog and multitexture blending. Together with vertex blending (which will not be tested) these operations constitutes the entire vertex- and pixel process that can be replaced by the programmable pipeline.

4.2.1 Vertex processing

1. Lighting, test case 1-18

These tests are performed to reveal differences in performance when using different kinds of lights in the two pipelines. Lights that will be tested are directional-, spot- and pointlight. Light is interesting to test because it is a fundamental graphical effect that enhances the realism of environments.

2. Vertex transforms, test case 19-24

Test cases 19-24 compare the performance differences in transforming vertices to screen projection space using the fixed function pipeline or a vertex shader in the programmable pipeline. Transforming vertices to projection space is something that always has to be done when rendering graphics, and because of that it is interesting to measure.

3. Fog, test case 31-39

Fog is a powerful effect in 3D-environments, it consists of two joint operations, fog calculation and fog blending. These two operations can be split up and performed with different techniques. The fog operation can be performed both using the fixed function pipeline and with a vertex shader. Fog blending can be performed with the fixed function pipeline or with a pixel shader. In test case 31 - 39 the different combinations to create fog are tested.

4.2.2 Pixel processing

1. Multitexture blending, test case 25-30

In test cases 25-30 the performance differences between these two different techniques will be measured. Texture mapping and color blending are the main tasks in pixel processing, so blending two textures together is an excellent way of testing those operations.

4.3. How will the tests be performed?

Every test will be measured while rendering the same static scene. The scene consists of one plane with different amount of triangles, depending on the level of complexity. All test operations, such as light, vertex transforming, multitexture blending and fog, will be performed on this plane. The resolution of the test program will be 1152x864. All test cases will be run five times to make sure the results will be valid.

When starting the benchmark program the first test case will automatically begin, the scene will then wait for five seconds to make the FPS stable. After that, the FPS value for each second in five seconds will be measured. An average value of those five values will be calculated and saved. When the two times five seconds have elapsed the program will change to the next test case and repeat the measure process. When all the 39 test cases have been executed and measured the calculated values will be saved to an XML-file. Figure 7 shows one screenshot of every operation that is tested.

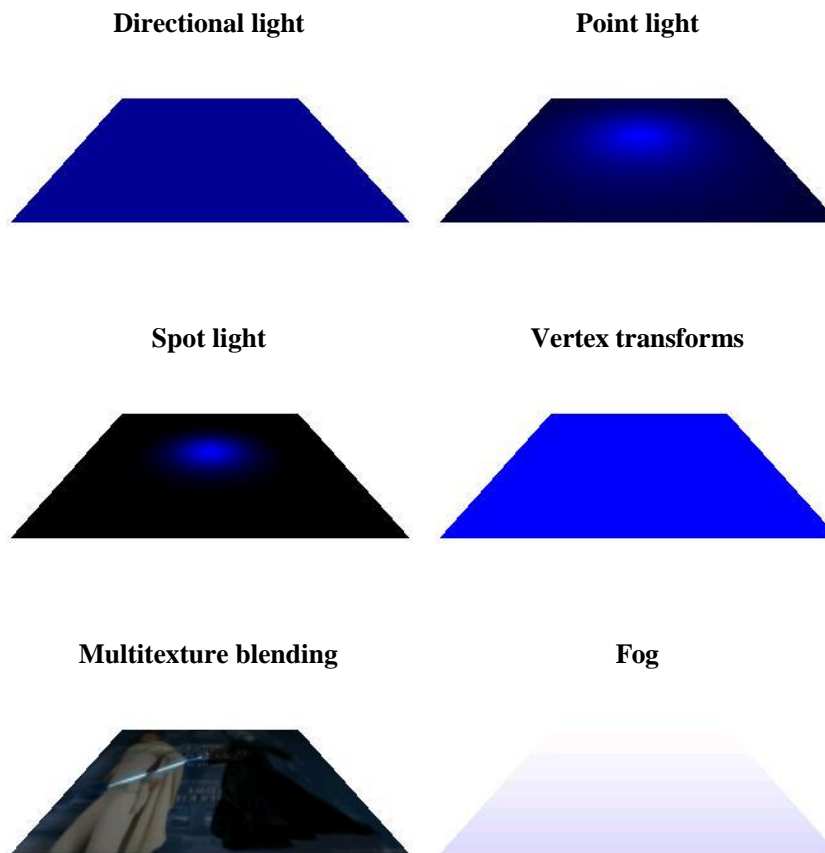


Figure 7. All six operations that are tested in the benchmark program

4.4 Test setup

Test System	
CPU	Intel Pentium4 2.6 GHz
FSB	800MHz
Motherboard	MSI Neo2 865PE
Memory	2x TwinMos PC3200, 512MB
Drivers & Configuration	
Graphics	NVIDIA v53.03
Chipset	Intel Inf. Update v5.1.0.1008
OS	Windows 2000 Prof. No SP
DirectX	DirectX 9.0b
Graphics card	
NVIDIA	GeForce FX 5600 Ultra (Albatron)

Tests were performed on a freshly installed operating system.

5 RESULT PRESENTATION

This section will present the results of the performance tests that were produced with the benchmark program. Section 5.1 shows the results in a table with FPS results and how much faster the fixed function pipeline is compared to the programmable pipeline in percents. Section 5.2 shows the results in a graph presenting how the FPS change during the tests, and Section 5.3 presents the results with screenshots showing visual differences. All test cases were run five times and the results were the same every time.

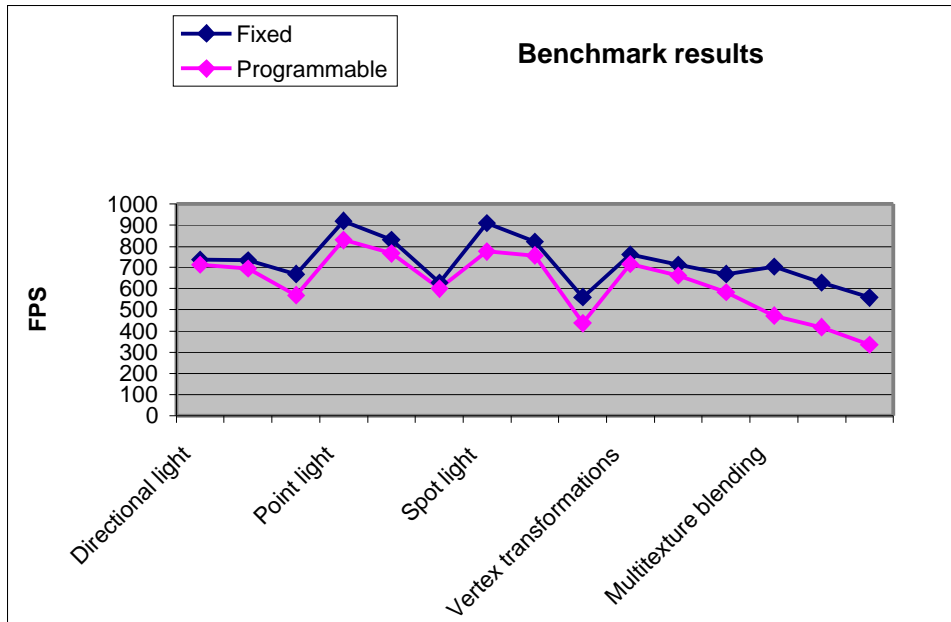
5.1 Result presentation with tables

	Complexity	Fixed f. pipeline	Vertex shader	Pixel shader	Diff. FPS	Diff. %
Directional light	Low	737	714	-	23	3,2%
	Medium	733	695	-	38	5,5%
	High	668	570	-	98	17,2%
Pointlight	Low	920	829	-	91	11,0%
	Medium	829	765	-	64	8,4%
	High	630	601	-	29	4,8%
Spotlight	Low	911	776	-	135	17,4%
	Medium	822	756	-	66	8,7%
	High	561	437	-	124	28,4%
Vertex transformations	Low	762	717	-	45	6,3%
	Medium	713	663	-	50	7,5%
	High	670	584	-	86	14,7%
Multitexture blending	Low	705	-	472	233	49,4%
	Medium	629	-	419	210	50,1%
	High	557	-	336	221	65,8%

FPS	Complexity	Fog	Fog Blending	Diff. FPS	Diff. %
823	Low	Fixed f. pipeline	Fixed f. pipeline	-	-
780	Medium	Fixed f. pipeline	Fixed f. pipeline	-	-
673	High	Fixed f. pipeline	Fixed f. pipeline	-	-
811	Low	Vertex shader	Fixed f. pipeline	12	1,5%
750	Medium	Vertex shader	Fixed f. pipeline	30	4,0%
638	High	Vertex shader	Fixed f. pipeline	35	5,5%
343	Low	Vertex shader	Pixel shader	480	139,9%
294	Medium	Vertex shader	Pixel shader	486	165,3%
240	High	Vertex shader	Pixel shader	433	180,4%

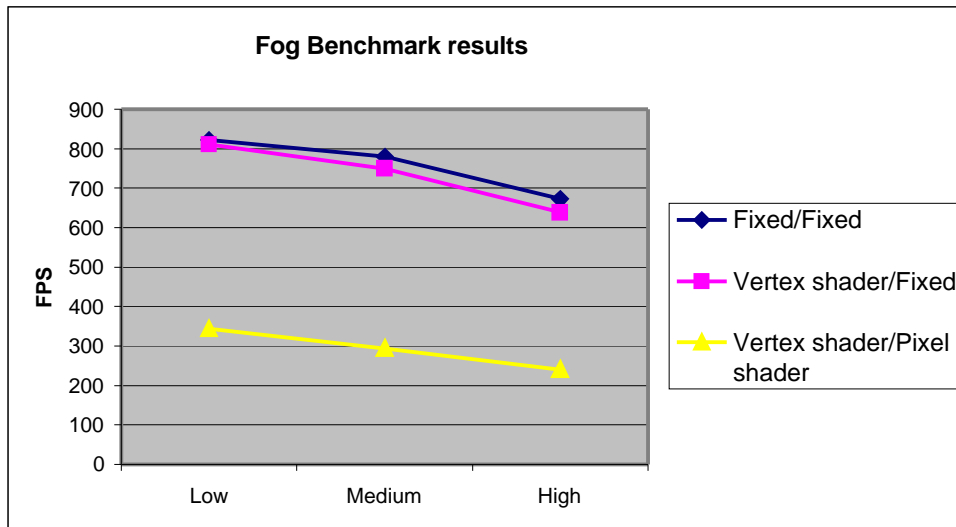
Table 2. Benchmark results.

5.2 Result presentation with graphs



Graph 1. Benchmarks results

Here we can see the results of directional light, pointlight, spotlight, vertex transformation and multitexture blending. The blue line shows the results of the fixed function pipeline and the pink one represents the programmable pipeline. The first three dots, looking from the left, show the FPS of the directional light effect on low, medium and high complexity. The other effects follow in the same pattern. As we can see the fixed function pipeline is faster in all test cases.



Graph 2. Fog benchmark results

In this graph we can see the benchmark results of the fog effect. There are three different lines in the graph, the blue represents fog with both fog calculation and fog color blending implemented in the fixed function pipeline. The pink represents the effect implemented with vertex shader fog calculation and fixed function fog blending and the yellow line shows results only using the programmable pipeline.

5.3 Result presentation with screenshots

5.3.1 Directional light

The first series of tests performed were the fixed function pipeline versus vertex shaders, and the tested effect was directional light. As can be seen in Figure 8 below, there were no visual differences between the fixed function pipeline and the vertex shader.

However, there were differences in performance. When using low complexity, the performance difference was very low (3.2%). The difference in performance increased a bit when using medium complexity (5.5%). However, when using high complexity, the performance difference increased considerably (17.2%).



Figure 8. Directional light with the fixed function pipeline versus a vertex shader

5.3.2 Pointlight

The second test series performed were also the fixed function pipeline versus vertex shaders, but the tested effect was pointlight. Figure 9 below shows that there are differences in the visual appearance between the fixed function pipeline and the vertex shader.

There are also differences in performance. The performance difference with pointlight and low complexity were relatively high (11.0%). When using medium complexity, the differences decreased a bit (8,4%), and even more with high complexity (4,8%).



Figure 9. Pointlight with the fixed function pipeline versus a vertex shader

5.3.3 Spotlight

As with the previous series of tests, the third series of tests performed were the fixed function pipeline versus vertex shaders. Figure 10 shows that there is a slight

difference in visual appearance between the fixed function pipeline and the vertex shader.

The difference in performance when using low complexity was rather significant (17.4%). Switching to medium complexity made the performance difference decrease (8.7%). Surprisingly the performance difference increased radically when using high complexity (28.4%).



Figure 10. Spotlight with the fixed function pipeline versus a vertex shader

5.3.4 Vertex transformations

In the fourth series of tests, vertex transformations were evaluated with the fixed function pipeline and vertex shaders. There were no visual differences, as shown in Figure 11, because this is not a visual effect.

At low complexity the performance difference was 6,3%, at medium it was 7.5% and at high 14.7%.



Figure 11. Vertex transformations with the fixed function pipeline versus a vertex shader

5.3.5 Multitexture blending

In this series of tests, fixed function multitexture blending is compared to pixel shader multitexture blending. This effect showed both visual differences and very large performance differences. As shown in Figure 12, the pixel shader screenshot is smoother than the fixed function pipeline screenshot; the contours melt into each other.

The performance difference at low complexity was very significant (49,4%), but when changing to medium complexity the performance did not change noticeably (50,1%). At high complexity the performance difference increased to 65,8%.



Figure 12. Multitexture blending with the fixed function pipeline versus a pixel shader

5.3.6 Fog

In the last series of tests, when using different combinations of the fixed function pipeline and the programmable pipeline, fog was tested. There were some visual differences, but only when using the fixed function pipeline, as shown in Figure 13.

Compared to fog with a vertex shader, the performance difference when using only the fixed function pipeline was relative small, 1,5% at low complexity, 4.0% at medium and 5.5% at high. But when using both a vertex- and a pixel shader to create the fog effect, the fixed function pipeline was exceptionally faster. At low complexity the fixed function pipeline was 139.9% faster, at medium 165.3% and at high 180.4% faster.

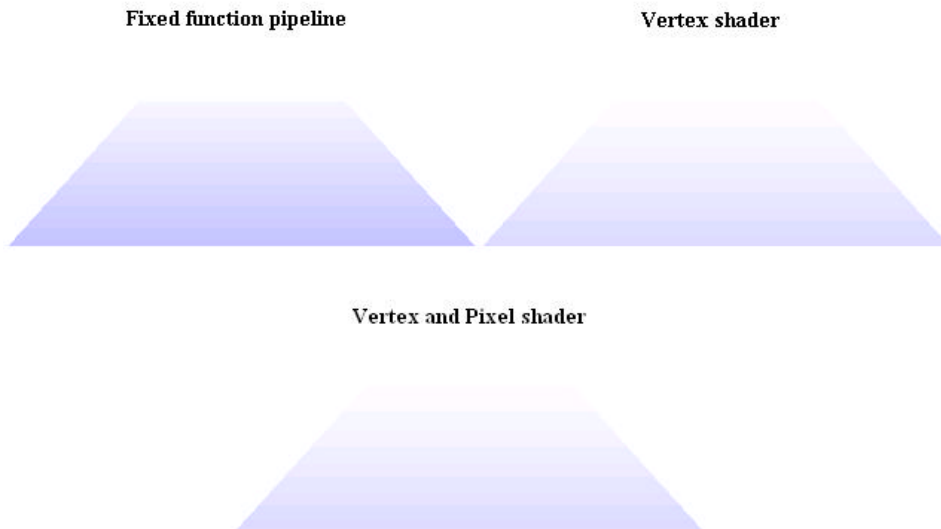


Figure 13. Fog effects with different combinations of the fixed function pipeline and the programmable pipeline.

6 RESULT DISCUSSION

After having analyzed the test results, a few questions arise:

- Why are there visual differences?
- Why is the fixed function pipeline always faster?
- Why do the performance differences increase/decrease with complexity?
- Why use the programmable pipeline when the fixed function pipeline is faster?

This chapter will try to answer these questions.

6.1 Why are there visual differences?

Four of the tests (point light, spot light, multi texture blending and fog) showed differences in visual appearance, this might depend on that the algorithms or the variables in the algorithms of the effects are not exactly the same.

6.2 Why is the fixed function pipeline always faster?

With our test setup, all of the tests demonstrate that the fixed function pipeline is faster than vertex and pixel shaders. We have two theories that can explain this.

The first theory is that since the algorithms in the fixed function pipeline are hard wired on the graphics card, they are always faster than software applications, even though the operations in the software application are supported by the hardware.

The second theory is, as mentioned in Section 6.1, that the algorithms in our shaders are not implemented exactly as the fixed function pipeline. Maybe some of our shaders could have been implemented a bit more effectively, so the differences in performance would not be as large. However, we still believe that the fixed function pipeline will be faster in general.

6.3 Why do the performance differences increase/decrease with complexity?

In all test cases except spotlight and pointlight the FPS difference increases with complexity. Our theory why it is like this is that the fixed function pipeline and the programmable pipeline handles vertices differently, so the complexity change are not linear with the FPS change. Also some overhead, like shader switching, when using shaders could have to do with the non-linear FPS change. Why the spotlight and pointlight test cases have different FPS changes with complexity we think depends on that we have implemented the effect algorithms almost from scratch, while in the other effects we have mostly used already existing intrinsic algorithms.

6.4 Why use the programmable pipeline?

The fixed function pipeline was a bit faster in all our tests. So why even bother using the programmable pipeline? Previously in the report we have mentioned that a great deal of flexibility is achieved when using the programmable pipeline. With flexibility we mean that effects can be visualized in ways that are impossible to achieve with the fixed function pipeline. This is because the developer programs his/her own shaders that control the visualization of the effect. Effects can be created that no one has even thought of yet. If the fixed function pipeline is used instead, there is a fixed amount of functions to apply when creating visual effects, which makes the appearance of effects limited.

When designing a 3D application, developers have to decide whether tailored effects are needed. If not, the fixed function pipeline should be used, with a slight increment in performance to be expected.

However, as can be seen in Figure 6, it is possible to gain a bit of profit from both of the pipelines. A vertex shader can be combined with fixed function pixel processing, gaining vertex processing flexibility but without the performance loss of using a pixel shader. Similarly, fixed function vertex processing can be combined with a pixel shader, gaining performance on the vertex processing and flexibility of the pixel shader. The fixed function pipeline and programmable pipeline combinations can be seen as four paths:

1. Fixed function vertex processing and fixed function pixel processing.
2. Fixed function vertex processing and programmable pixel processing.
3. Programmable vertex processing and fixed function pixel processing.
4. Programmable vertex processing and programmable pixel processing.

Path 1 will give the highest performance, but with rigid effects. Path 2 will give the ability to customize the pixel processing, but with the performance advantage of the fixed function vertex processing. Customizable vertex processing can be obtained with path 3, without the performance loss of programmable pixel processing. At last, path 4 will give the maximum amount of flexibility, but with a substantial loss of performance. Of course, developers can choose to implement all four paths, and in real-time choose which path to use, depending on the hardware of the running platform.

7 CONCLUSION

After having completed this research, we can draw the conclusion that the fixed function pipeline always beats the programmable pipeline in terms of performance, from a tiny bit (1,50% for vertex shader fog and fixed function fog blending with low complexity) to a substantial difference (180,40% for vertex shader fog and pixel shader fog blending with high complexity).

The advantages of using the fixed function pipeline is obvious; it is always faster, and there is no need to implement the vertex- and pixel manipulating operations, since they are already implemented.

On the other hand, the advantage of using the programmable pipeline is that a great amount of flexibility can be achieved when creating graphical effects. In the fixed function pipeline, the vertex- and pixel manipulating operations are already implemented, thus, they look the same for everybody. In the programmable pipeline however, these operations are implemented by the developer, giving him or her possibility to create effects that are impossible to get from the fixed function pipeline.

Depending on the need for flexible effects, different combinations of the fixed function pipeline and the programmable pipeline can be chosen, achieving the largest amount of performance for the demanded flexibility.

Even though these test results, with the FPS difference percentage, presumably only correspond to our exact test setup, we think that our conclusion is correct in general, at least on graphics hardware on the market today.

In the future, tests should be done on other resolutions than the one we choose, also other different graphics cards should be tested on to see if the results will be the same.

REFERENCES

- [1] Frank D. Luna, *Introduction to 3d game programming with DirectX 9.0*, Wordware Publishing Inc, 2003
- [2] Kris Gray, *DirectX9 programmable graphics pipeline*, Microsoft Press, 2003
- [3] Craig Peeper, Jason L. Mitchell, Wolfgang F. Engel, *Shader X²*, Wordware Publishing Inc, 2004
- [4] Lars Ivar Igesund, Mads Henrik Stavang, *Fixed Function Pipeline using vertex programs*, Norwegian University of Science and Technology, November 22, 2002
- [5] Martin Ecker, *Programmable Graphics Pipeline Architectures*, March 24, 2003
- [6] MSDN, *Vertex and Pixel Processing*, http://msdn.microsoft.com/library/en-us/directx9_c/directx/graphics/programmingguide/fixefunction/legacyfvfformats/vertexandpixelprocessing.asp, 2004
- [7] Ian Kasprzak, Adam Ormond, Doug Papenthien, *Senior Design Technology Report*, Team nVDemo, January 11, 2002
- [8] Dom Penfold, *Vertex Shaders and Pixel Shaders*, <http://www.tomshardware.com/graphic/20020116/index.html>, January 2002
- [9] Wolfgang F. Engel, *Introduction to Shader Programming Part I: Fundamentals of Vertex Shaders*, <http://www.gamedev.net/columns/hardcore/dxshader1/>
- [10] Wolfgang F. Engel, *Introduction to Shader Programming Part III: Fundamentals of Pixel Shaders*, <http://www.gamedev.net/columns/hardcore/dxshader3/>

APPENDIX A

GRAPHICS PIPELINE

1. Coordinate spaces

As we saw in Section 2.1, vertices have a position attribute. This position attribute, the x-, y- and z-coordinates, locates the vertex in space. However, when a vertex is created, the position attribute is only a local position; it is only used to locate the vertex in *model space*. Model space is the local space of a 3D object.

Vertices can not stay in model space; they have to be moved to *world space* [2, p. 288]. Every vertex created is transformed from their local coordinate system to the world space coordinate system that is used by all the objects in a scene [6]. Two vertices, belonging to two different objects, can have the same model space coordinates without actually being in the same place. This is not true for world space coordinates. If two vertices, belonging to the same object or not, have the same world space coordinates, they really are in the same place.

When the vertices are transformed into world space, they are transformed again, this time into *view space*, or *camera space*. In view space, world space coordinates are relocated and rotated around the camera's view, turning world space into camera space [6]. The camera is the point where the user looks at the objects, so the vertices in view space are located relative to the user [2, p. 288].

Now when the objects are located relative to the user, they have to be scaled. If an object is far away it should look smaller than if it was close to the camera. This is done with a *projection transformation*, which transforms vertices from view space into *projection space* [2, p. 288].

The final transformation transforms the objects from projection space into *screen space*. The purpose of this transformation is to get a depth in the scene, which means that objects are scaled in relation to their distance of the viewer so that closer objects appear larger than distance objects [6]. Now the vertex data is ready for rendering.

2. The Fixed function pipeline

As mentioned in Chapter 2, 3D graphics in Direct3D can be rendered using the fixed function pipeline. In the fixed function pipeline, several manipulating operations are performed on vertices to get a correct output on the screen, and all of these operations are predefined and fixed.

Let's take a look at the architecture of the Direct3D pipeline. The Direct3D pipeline architecture is shown in Figure A.1:

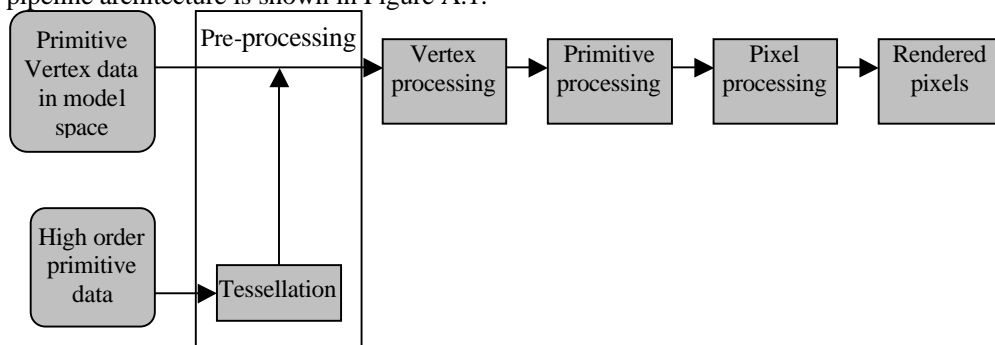
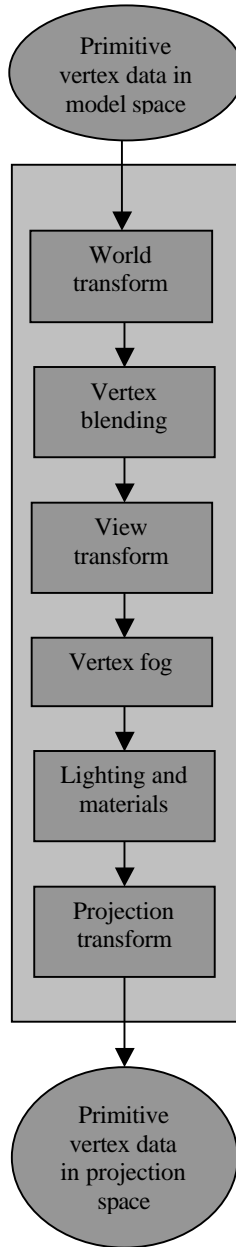


Figure A.1, The Direct3D graphical pipeline [2, p. 67].

The input to the fixed function pipeline is *Primitive Vertex data in model space* and *High order primitive data*. These are the vertices and primitives that are created in the application. First the input data is pre-processed, where the high order primitive data is passed through a tessellation procedure. This thesis will not cover that part.

2.1. Vertex processing



The interesting parts when talking about the fixed function pipeline is the ones after pre-processing. When the input data has been pre-processed, it is time for vertex processing. Vertex processing contains a number of operations, and the vertex processing architecture can be seen in Figure A.2.

2.1.1. World transformation

In the *world transformation* step, vertex position data is transformed from model space into world space, as described in Appendix A, Section 1. The 3D objects are positioned in relation to each other.

2.1.2. Vertex blending

Vertex blending combines one (or more) sets of vertex data to animate vertex data. Vertex blending uses one set of vertex data and one or more transforms [2, p.5].

2.1.3. View transform

View transform transforms vertex position data from world space to view space, as described in Appendix A, Section 1. The 3D objects are positioned in relation to the camera.

2.1.4. Vertex fog

Vertex fog calculates a per vertex fog color. This color is usually blended with other per-vertex colors to generate the final vertex color, to pass on to primitive processing [2, p. 6].

2.1.5. Lighting and materials

The *Lighting and materials* step of vertex processing calculates the vertex color when user supplied lighting and materials are added. It is very difficult to apply natural looking lights to a scene without losing too much performance [2, p. 6].

2.1.6. Projection transform

Projection transform transforms vertex position data from view space into projection space, as described in Appendix A, Section 1. This step is the last operation in vertex processing.

Figure A.2, Vertex processing architecture [2, p. 5]

2.2. Primitive processing

After vertex processing, primitive and pixel processing takes place. They convert per-vertex data into the final pixels that will be rendered [2, p. 68]. A functional diagram of the primitive process is shown in Figure A.3.

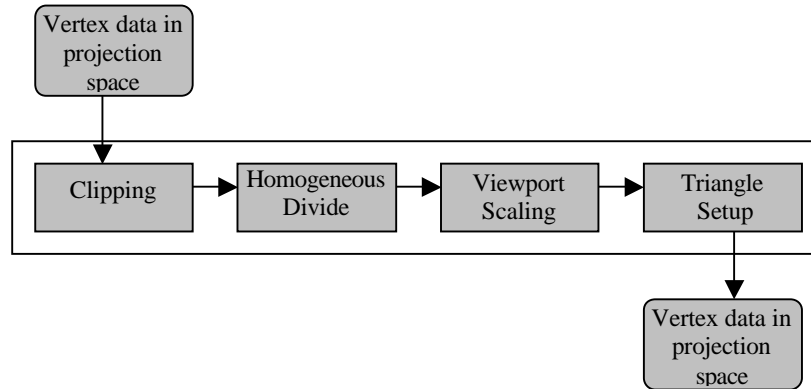


Figure A.3. Primitive processing [2, p. 68].

The primitive processing consists of four steps, shortly described below.

- **Clipping**
Geometry that is outside the viewing frustum is removed for maximum rendering efficiency [2, p. 68].
- **Homogeneous divide**
X, y, z vertex data is converted to non-homogenous space, and then the data is sent to the rasterizer [2, p. 68].
- **Viewport scaling**
2D projection space is scaled to 2D screen space [2, p. 68].
- **Triangle set up**
Triangle-attribute interpolators are prepared and per-vertex attributes are converted into per-pixel attributes [2, p. 68].

The outcome of the primitive processing, and what is sent to pixel processing, is a triangle list of vertex data interpolated per pixel.

2.3 Pixel processing

The task of pixel processing is to blend several per-pixel data types and texture samples [2, p. 68]. The outcome of that task is a pixel color that is rendered to the screen. The pixel process can be divided into two parts, where the first part can be replaced by a pixel shader and the second part that still needs to be run in the fixed function pipeline. The parts of pixel processing are shown in Figure A.4.

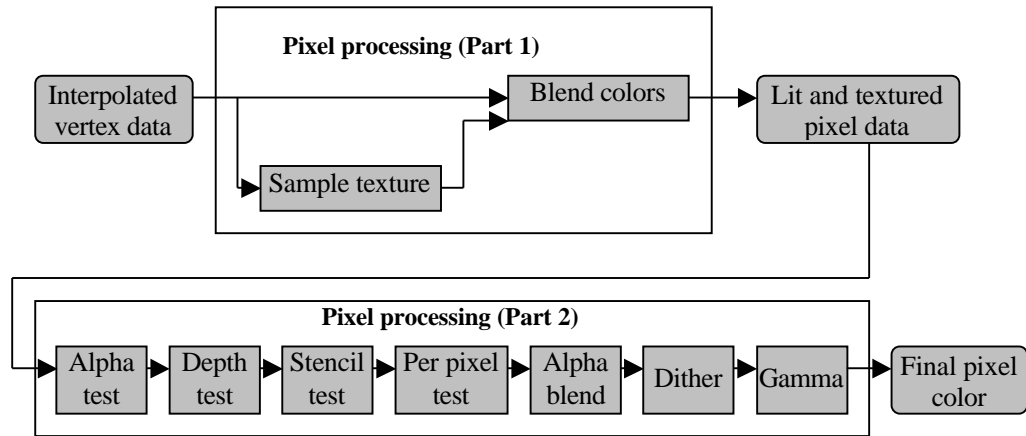


Figure A.4, Pixel processing [2, p. 69]

The steps in pixel processing are shortly described below.

- **Sample texture**
One or more textures are sampled [2, p. 69].
- **Blend**
Per-pixel attributes like; diffuse, specular color and/or texture samples are blended [2, p. 69].
- **Alpha test**
An alpha test is conducted to see if the pixel color affects the final pixel color [2, p. 69].
- **Depth test**
If the pixel is visible the depth buffer is updated with the pixel depth [2, p. 69].
- **Stencil test**
A stencil test is conducted to see if the pixel color affects the final pixel color [2, p. 69].
- **Per-pixel fog**
A fog color is applied to the pixel color [2, p. 69].
- **Alpha blend**
To create a transparent or semi-transparent blend between a source pixel and a frame buffer pixel a pixel alpha is applied [2, p. 69].
- **Dither**
To get a more consistent color range, a dither algorithm is used to blend adjacent pixels [2, p. 69].
- **Gamma**
A gamma correction is applied for the final frame buffer pixel color [2, p. 69].

3. Vertex shaders

3.1. The vertex shader virtual machine

As stated before, vertex shaders in the programmable pipeline replace the vertex processing part of the fixed function pipeline. The vertex processing part of the fixed function pipeline takes pre-processed vertex data as input, and so does the vertex shaders in the programmable pipeline. Vertex shaders communicates with the application and the primitive processing using input and output registers; vertex shaders gets their input from the application via a number of read-only input registers, and sends their output to the primitive processing via a number of write-only output registers [5, p. 5]. Vertex shaders can not send data back to the application, and it can not retrieve any data from primitive processing. The architecture of the vertex shader virtual machine can be seen in Figure A.5:

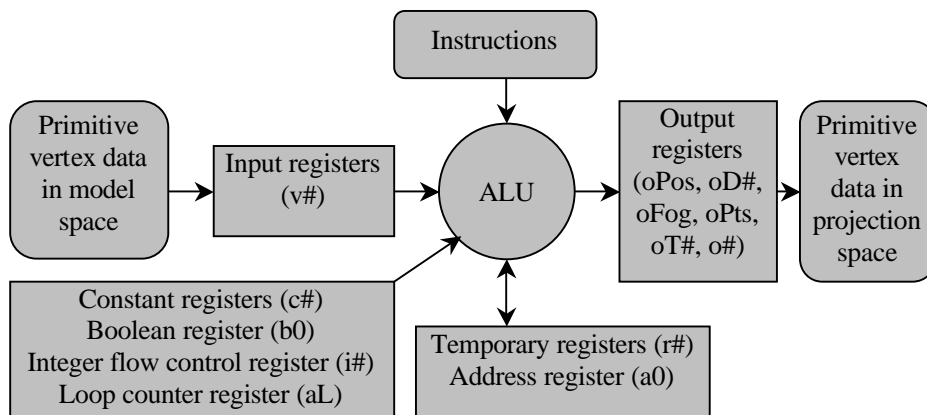


Figure A.5. The vertex shader virtual machine [2, p. 26]

As can be seen in Figure A.5, the center of the vertex shader virtual machine is the arithmetic logic unit, or the ALU. The ALU receives input (primitive vertex data in model space) from the application via input registers $v\#$.

Once this data is received, it performs certain operations on it, which are stated as instructions. These instructions are in fact the vertex shaders implemented by the developer. To perform the instructions, the ALU uses Constant registers $c\#$, Boolean register $b0$, Integer flow control register $i\#$, Loop counter register aL , Temporary registers $r\#$ and Address register $a0$.

When all instructions are executed and all operations performed, the ALU writes the output (primitive vertex data in projection space) to the Output registers $oPos$, $oD\#$, $oFog$, $oPts$, $oT\#$ and $o\#$. The data is now ready for primitive processing.

4. Pixel shaders

Pixel shaders are the second component of the programmable pipeline, and are implemented to replace the pixel processing in the fixed function pipeline. As with fixed function pixel processing, pixel shaders calculate each pixel's final color.

On the way to calculate the final color, a texture sampler is used. A texture sampler consists of texture data, such as a texture file or a rendered target, and up to fifteen samplers. The task of the sampler is to use the texture coordinates to look up a texture color from the texture data [2, p. 71]. The texture color is then returned to the pixel shader, where programmable blending operations are executed, such as multitexture blending and lighting, to calculate the final color to be rendered. The operations that can be used inside the pixel shader to come up with the final pixel color is what makes it so flexible, it can either be inbuilt operations or own programmed operations, only your own creativity is the limit of what can be archived. The interpolated vertex data, which is the input to the pixel shader, can either come from the fixed function pipeline or a vertex shader [2, p. 71].

The frame-buffer blending, which is Part 2 in pixel processing, is always done by the fixed function pipeline regardless if Part 1 is done by a programmable shader or the fixed function pipeline [2, p. 72].

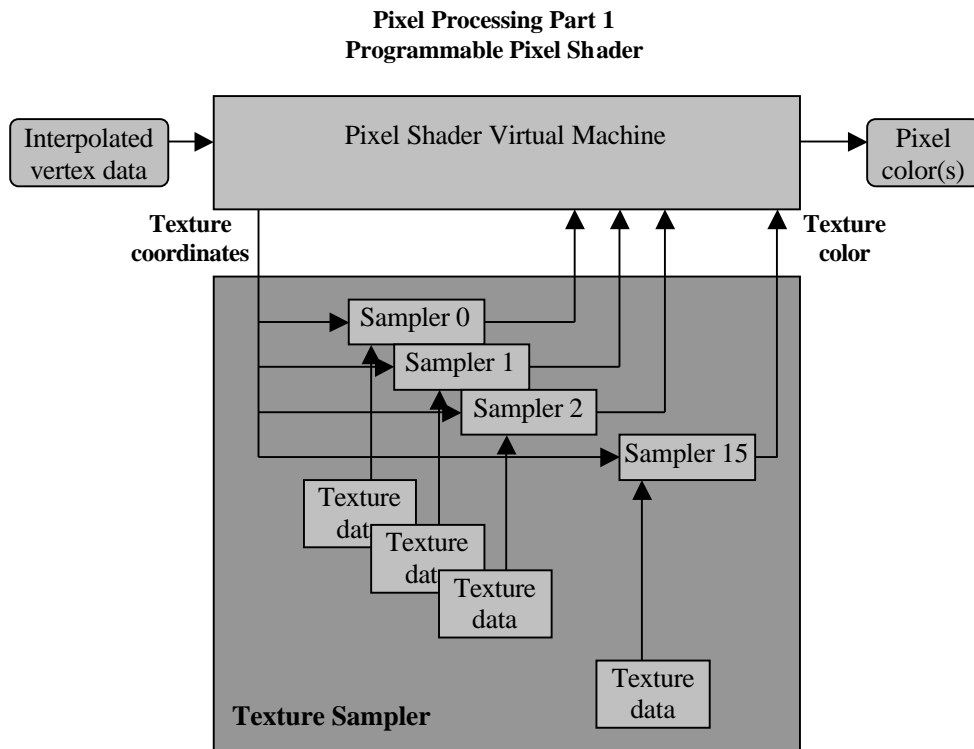


Figure A.6. Pixel processing Part 1 (detailed) [2, p. 71]

4.1 Pixel shader virtual machine

The actual brain in a pixel shader is the ALU (Arithmetic Logic Unit), which is used to calculate mathematical operations on input data to generate a pixel color as

output data. Figure A.7 shows a conceptual diagram of the pixel shader virtual machine, with the ALU as an engine in the middle.

Pixel Shader Virtual Machine

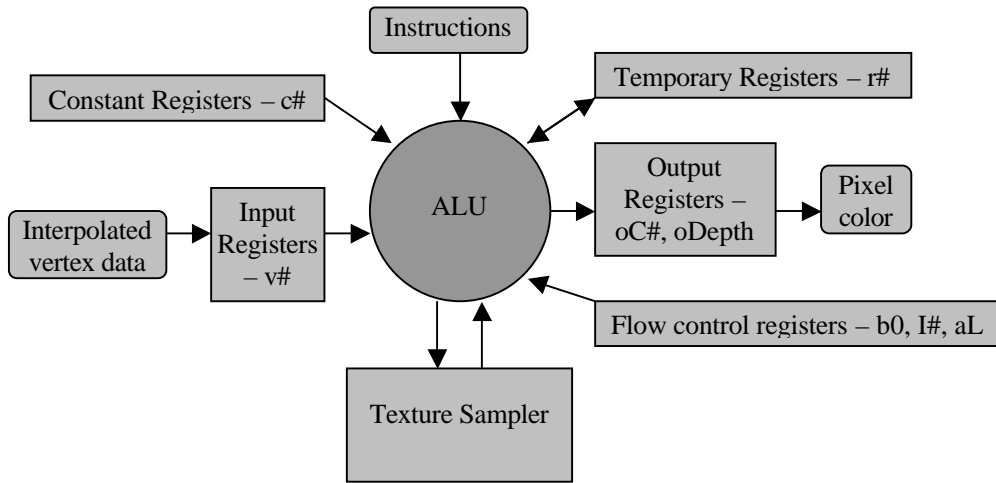


Figure A.7. Pixel Shader Virtual Machine [2, p. 72]

As mentioned before, interpolated vertex data is received as input, then depending on what kind of operations that is to be performed, different registers is used such as; constant registers, temporary registers and flow control registers. The final result, as in a per-pixel color, is then sent to the output register and provided back to the fixed function pipeline for pixel-processing part 2 [2, p. 73].

As a support for calculating the pixel color a set of instructions is provided for the ALU. The instruction set contains lots of different kinds of instructions for pixel processing operations, such as finding a dot product, multiplying by a matrix, or finding min and max values [2, p. 73].

APPENDIX B

EFFECT CODE

Below is the code that generates all the effects. If you are interested in the whole benchmark program, it can be found on www.3dmasters.zapto.org

```
//Global variables
float4x4 World          : WORLD;
float4x4 View           : VIEW;
float4x4 Projection     : PROJECTION;

texture Tex0 <string name = "skull.jpg"; >;
texture Tex1 <string name = "grass.jpg"; >;
float4 white = { 1.0f, 1.0f, 1.0f, 0.0f};

static float3 LightDir < string UIDirectional = "Light Direction"; > = normalize(float3(1.0f,
1.0f,-1.0f));

/*
 * Sampler to use with texture 1
 */
sampler EffectSampler = sampler_state
{
    Texture          = (Tex0);
    MinFilter        = LINEAR;
    MagFilter        = LINEAR;
    MipFilter        = LINEAR;
};

/*
 * Sampler to use with texture 2
 */
sampler EffectSampler2 = sampler_state
{
    Texture          = (Tex1);
    MinFilter        = LINEAR;
    MagFilter        = LINEAR;
    MipFilter        = LINEAR;
};

//-----
//
// Fixed Directional Ligthing
//
//-----

/*
 * Technique for fixed function directional lighting
 */
technique Fixed_DirLighting
{
```

```

pass P0
{
    PixelShader           = NULL;
    VertexShader          = NULL;
    CullMode               = None;
    Lighting              = true;
    NormalizeNormals      = true;
    SpecularEnable        = false;
    FogEnable              = false;

    LightType[0]          = Directional;
    LightAmbient[0]       = { 0.0f, 0.0f, 0.0f, 1.0f };
    LightDiffuse[0]       = { 1.0f, 1.0f, 1.0f, 1.0f };
    LightSpecular[0]      = { 0.0f, 0.0f, 0.0f, 1.0f };
    LightDirection[0]     = { -1.0f, -1.0f, 1.0f, 0.0f };
    LightPosition[0]      = { 0.0f, 0.0f, -10.0f };
    LightFalloff[0]       = 1.0f;
    LightRange[0]         = 5000.0f;
    LightTheta[0]         = 0.5f;
    LightPhi[0]           = 0.7f;
    LightAttenuation0[0]  = 1.0f;
    LightAttenuation1[0]  = 0.0f;
    LightAttenuation2[0]  = 0.0f;

    LightEnable[0]        = true;

    MaterialAmbient       = { 0.0f, 0.0f, 0.0f, 1.0f };
    MaterialDiffuse        = { 0.0f, 0.0f, 0.0f, 1.0f };
    MaterialEmissive       = { 0.0f, 0.0f, 0.0f, 0.0f };
    MaterialPower          = 1.0f;
    MaterialSpecular       = { 1.0f, 1.0f, 1.0f, 1.0f };

    WorldTransform[0]     = (World);
    ViewTransform          = (View);
    ProjectionTransform    = (Projection);

    Sampler[0]            = (EffectSampler);
}
}

//-----
//
// Vertex Shader Directional Lighting
//
//-----

/*
 * input struct for vertex shader directional lighting
 */
struct VS_DirLighting_INPUT
{
    float4 position       : POSITION;
    float4 normal         : NORMAL;
    float4 diffuse        : COLOR;
};

```

```

/*
 * output struct for vertex shader directional lighting
 */
struct VS_DirLighting_OUTPUT
{
    float4 position      : POSITION;
    float4 diffuse      : COLOR;
};

/*
 * vertex shader for directional lighting
 */
VS_DirLighting_OUTPUT VS_DirLighting(VS_DirLighting_INPUT input)
{
    VS_DirLighting_OUTPUT output = (VS_DirLighting_OUTPUT)0;
    float3 L                      = LightDir;
    float3 NormalWorld = normalize(mul(input.normal, (float3x3)World));
    output.diffuse              = input.diffuse * max(0, dot(NormalWorld, L));

    output.position             = mul(input.position, World);
    output.position             = mul(output.position, View);
    output.position             = mul(output.position, Projection);
    return output;
}

/*
 * technique for vertex shader directional lighting
 */
technique VS_DirLighting
{
    pass P0
    {
        CullMode      = None;
        Sampler[0]    = (EffectSampler);

        VertexShader  = compile vs_2_0 VS_DirLighting();
    }
}

//-----
//
// Fixed Exponential Fog
//
//-----

/*
 * technique for fixed function exponential fog
 */
technique Fixed_Fog
{
    pass P0
    {
        PixelShader    = NULL;
        VertexShader   = NULL;
    }
}

```

```

        CullMode                = None;
        Lighting                 = false;
        NormalizeNormals         = true;
        SpecularEnable           = false;

        WorldTransform[0]       = (World);
        ViewTransform            = (View);
        ProjectionTransform      = (Projection);

        Sampler[0]               = (EffectSampler);

        FogVertexMode            = EXP;
        FogEnable                 = true;
        FogColor                  = { 1, 1, 1, 0 };
        FogDensity                = 0.02;
    }
}

//-----
//
// Vertex Shader Exponential Fog
//
//-----

/*
 * input struct for vertex shader exponential fog
 */
struct VS_Fog_INPUT
{
    float4 Position            : POSITION;
    float3 Diffuse              : COLOR0;
};

/*
 * output struct for vertex shader exponential fog
 */
struct VS_Fog_OUTPUT
{
    float4 Position            : POSITION;
    float3 Diffuse              : COLOR;
    float Fog                   : FOG;
};

/*
 * vertex shader for exponential fog
 */
VS_Fog_OUTPUT VS_Fog(VS_Fog_INPUT Input)
{
    float4 clpPos, camPos;
    VS_Fog_OUTPUT Out        = (VS_Fog_OUTPUT) 0;

    // Calculate the clip space position
    clpPos                    = mul(Input.Position, World);
    clpPos                    = mul(clpPos, View);
    clpPos                    = mul(clpPos, Projection);

```

```

        Out.Position          = clpPos;

        // Calculate vertex position in camera space
        camPos                = mul(Input.Position, World);
        camPos                = mul(camPos, View);

        float fDensity        = 0.1;
        float fFogEnd         = 15;
        float fDist           = camPos.z/fFogEnd*4;
        float fog              = exp(-fDist*fDensity);

        Out.Diffuse           = lerp(white, 1*2.0f*Input.Diffuse.xyz, fog);
        return Out;
    }

    /*
    * technique for vertex shader exponential fog
    */
    technique VS_Fog
    {
        pass P0
        {
            CullMode          = None;
            Sampler[0]        = (EffectSampler);

            VertexShader      = compile vs_2_0 VS_Fog();
        }
    }

    //-----
    //
    // Vertex and Pixel Shader Exponential Fog
    //
    //-----

    /*
    * input struct for VS vertex- and pixel shader exponential fog
    */
    struct VSFog_INPUT
    {
        float4 Position      : POSITION;
        float3 Diffuse       : COLOR0;
    };

    /*
    * output struct for VS vertex- and pixel shader exponential fog
    */
    struct VSFog_OUTPUT
    {
        float4 Position      : POSITION;
        float3 Diffuse       : COLOR;
        float3 FogVal        : COLOR1;
    };

    /*

```

```

* vertex shader for vertex- and pixelshader exponential fog
*/
VSFog_OUTPUT VSFog(const VSFog_INPUT Input)
{
    float4 clpPos, camPos;
    VSFog_OUTPUT Out = (VSFog_OUTPUT)0;

    // Calculate the clip space position
    clpPos = mul(Input.Position, World);
    clpPos = mul(clpPos, View);
    clpPos = mul(clpPos, Projection);
    Out.Position = clpPos;
    Out.Diffuse = Input.Diffuse;

    // Calculate vertex position in camera space
    camPos = mul(Input.Position, World);
    camPos = mul(camPos, View);

    float fDensity = 0.1;
    float fFogEnd = 15;
    float fDist = camPos.z/fFogEnd*4;
    float f = exp(-fDist*fDensity);

    // Set the fog value
    Out.FogVal.x = f; // Passed to PixelShader using color register
    return Out;
}

/*
* input struct for PS vertex- and pixel shader exponential fog
*/
struct PSFog_INPUT
{
    float4 Position : POSITION;
    float3 Diffuse : COLOR0;
    float3 FogVal : COLOR1;
};

/*
* output struct for PS vertex- and pixel shader exponential fog
*/
struct PSFog_OUTPUT
{
    float4 Color : COLOR;
};

/*
* pixel shader for vertex- and pixelshader exponential fog
*/
PSFog_OUTPUT PSFog(const PSFog_INPUT Input)
{
    PSFog_OUTPUT Out = (PSFog_OUTPUT) 0;

    // Fog blending
    float f = Input.FogVal.x;

```

```

        Out.Color                = lerp(white, 1*2.0f*Input.Diffuse.xyz, f);
        return Out;
    }

/*
 * technique for vertex- and pixelshader exponential fog
 */
technique VSPS_Fog
{
    pass P0
    {
        CullMode                = None;
        Sampler[0]              = (EffectSampler);

        VertexShader            = compile vs_2_0 VSFog();
        PixelShader              = compile ps_2_0 PSFog();
    }
}

//-----
//
// Pixel Shader MultiTexture
//
//-----

/*
 * input struct for pixel shader multitexture
 */
struct PS_MultiTexture_INPUT
{
    float2 texture1            : TEXCOORD0;
    float2 texture2            : TEXCOORD1;
};

/*
 * output struct for pixel shader multitexture
 */
struct PS_MultiTexture_OUTPUT
{
    vector diffuse              : COLOR0;
};

/*
 * pixel shader for multitexture
 */
PS_MultiTexture_OUTPUT PS_MultiTexture(PS_MultiTexture_INPUT input)
{
    PS_MultiTexture_OUTPUT output = (PS_MultiTexture_OUTPUT)0;

    vector b                    = tex2D(EffectSampler, input.texture1);
    vector s                    = tex2D(EffectSampler2, input.texture2);
    vector c                    = b * s;

    output.diffuse              = c;
}

```

```

        return output;
    }

    /*
    * technique for pixel shader multitexture
    */
    technique PS_MultiTexture
    {
        pass P0
        {
            CullMode                = None;
            Sampler[0]              = (EffectSampler);
            Sampler[1]              = (EffectSampler2);

            WorldTransform[0]       = (World);
            ViewTransform            = (View);
            ProjectionTransform      = (Projection);

            PixelShader              = compile ps_2_0 PS_MultiTexture();
        }
    }

    //-----
    //
    // Pixel Shader MultiTexture
    //
    //-----

    /*
    * technique for fixed function multitexture
    */
    technique Fixed_MultiTexture
    {
        pass p0
        {
            PixelShader              = NULL;
            VertexShader             = NULL;
            CullMode                 = None;
            Lighting                 = false;
            NormalizeNormals         = false;
            SpecularEnable           = false;

            WorldTransform[0]       = (World);
            ViewTransform            = (View);
            ProjectionTransform      = (Projection);

            Texture[0]              = (Tex0);
            ColorOp[0]              = SelectArg1;
            ColorArg1[0]            = Texture;

            Texture[1]              = (Tex1);
            ColorOp[1]              = Modulate;
            ColorArg1[1]            = Texture;
            ColorArg2[1]            = Current;
        }
    }

```

```

        ColorOp[2]          = Disable;
    }
}

//-----
//
// Fixed Vertex Transformation
//
//-----

/*
 * technique for fixed function vertex transformation
 */
technique Fixed_VertexTransform
{
    pass p0
    {
        PixelShader          = NULL;
        VertexShader         = NULL;
        CullMode              = None;
        Lighting              = false;
        NormalizeNormals      = false;
        SpecularEnable        = false;

        WorldTransform[0]    = (World);
        ViewTransform         = (View);
        ProjectionTransform   = (Projection);
    }
}

//-----
//
// Vertex Shader Vertex Transformation
//
//-----

/*
 * input struct for vertex shader vertex transformation
 */
struct VS_VertexTransform_INPUT
{
    float4 position          : POSITION;
    float3 diffuse           : COLOR;
};

/*
 * output struct for vertex shader vertex transformation
 */
struct VS_VertexTransform_OUTPUT
{
    float4 position          : POSITION;
    float3 diffuse           : COLOR;
};

/*

```

```

/* vertex shader for vertex transformation
*/
VS_VertexTransform_OUTPUT VS_VertexTransform(VS_VertexTransform_INPUT input)
{
    VS_VertexTransform_OUTPUT output= (VS_VertexTransform_OUTPUT)0;
    output.diffuse                = input.diffuse;

    output.position                = mul(input.position, World);
    output.position                = mul(output.position, View);
    output.position                = mul(output.position, Projection);
    return output;
}

/*
* technique for vertex shader vertex transformation
*/
technique VS_VertexTransform
{
    pass p0
    {
        PixelShader                = NULL;
        VertexShader                = NULL;
        CullMode                    = None;
        Lighting                    = false;
        NormalizeNormals            = false;
        SpecularEnable              = false;

        VertexShader                = compile vs_2_0 VS_VertexTransform();
    }
}

//-----
//
// Vertex Shader Spotlight
//
//-----

float3 SpotPos = {0, 15, 0};
float3 direction = {0, 1, 0};
float3 SpotDir = {0, 0, 0};
float phi = 1.15;
float falloff = 3.4;

/*
* input struct for vertex shader spotlight
*/
struct VS_SpotLight_INPUT
{
    float4 position                : POSITION;
    float4 normal                  : NORMAL;
    float4 diffuse                  : COLOR0;
};

/*
* output struct for vertex shader spotlight

```

```

*/
struct VS_SpotLight_OUTPUT
{
    float4 position          : POSITION;
    float4 diffuse          : COLOR0;
    float spotFactor        : COLOR1;
};

/*
 * vertex shader for spotlight
 */
VS_SpotLight_OUTPUT VS_SpotLight(VS_SpotLight_INPUT input)
{
    VS_SpotLight_OUTPUT output = (VS_SpotLight_OUTPUT)0;
    float3 L                    = LightDir;
    float3 NormalWorld = normalize(mul(input.normal, (float3x3)World));
    output.diffuse            = input.diffuse * max(0, dot(NormalWorld, L));

    float4 worldPos           = mul(input.position, World);
    output.position          = mul(worldPos, View);
    output.position          = mul(output.position, Projection);

    SpotDir                  = SpotPos - worldPos;
    SpotDir                  = normalize(SpotDir);

    float spotFactor;
    float temp                = dot(SpotDir, direction);
    if (temp >= cos(phi))
    {
        spotFactor            = pow(temp, falloff);
    }
    else
    {
        spotFactor            = 0.0;
    }

    output.diffuse           = input.diffuse*white*spotFactor;
    return output;
}

/*
 * technique for vertex shader spotlight
 */
technique VS_SpotLight
{
    pass P0
    {
        CullMode              = None;
        VertexShader          = compile vs_2_0 VS_SpotLight();
    }
}

//-----
//
// Vertex Shader Pointlight

```

```

//
//-----

/*
 * input struct for vertex shader pointlight
 */
struct VS_PointLight_INPUT
{
    float4 position      : POSITION;
    float4 normal        : NORMAL;
    float4 diffuse       : COLOR0;
};

/*
 * output struct for vertex shader pointlight
 */
struct VS_PointLight_OUTPUT
{
    float4 position      : POSITION;
    float4 diffuse       : COLOR0;
};

/*
 * vertex shader for pointlight
 */
VS_PointLight_OUTPUT VS_PointLight(VS_PointLight_INPUT input)
{
    VS_PointLight_OUTPUT output = (VS_PointLight_OUTPUT)0;

    float4 worldPos          = mul(input.position, World);
    output.position          = mul(worldPos, View);
    output.position          = mul(output.position, Projection);

    float b                  = distance(SpotPos, worldPos);
    float spotFactor;
    float range              = 80;

    spotFactor               = 1 - (b/range);

    output.diffuse           = input.diffuse*white*spotFactor;
    return output;
}

/*
 * technique for vertex shader pointlight
 */
technique VS_PointLight
{
    pass P0
    {
        CullMode             = None;
        VertexShader         = compile vs_2_0 VS_PointLight();
    }
}

```