Chair for Open Source Software

Master of Science in International Information systems

Master Thesis of

Christian F r i e d r i c h

The Lmits of Application Programming Interfaces

Summer Term 2013

Supervisor: Prof. Dr. Dirk Riehle

Nürnberg, 22 September 2013

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Christian Friedrich

Abstract

This paper deals with the approach of examining standard specifications. Standard specifications make it possible to exchange implementations without clients noticing this change. Using the OpenGL specification as an example, a testing strategy has been developed that is designed to test software specifications from the perspective of an external tester. Three OpenGL implementations are compared for their compliance with the standard. The selected implementations are covering a broad spectrum from low-budget onboard graphics card to a high-end professional card. These graphics cards support different versions of the specification. This paper discusses a way to select a suitable function subset and shows how the specification specifies wanted and unwanted variation. This thesis does not analyze all possible errors of OpenGL but delivers basic approaches how specifications have to be tested, what opportunities graphics card manufacturers have and where the testing of specifications fails.

Content

List of Figures

List of Tables

1. Introduction

1.1. Specification in computer technology

In recent years computers gained more and more in importance. Their performance has strongly been improved and is still subject of constant progression. There is a wide range of manufacturers of computer hardware and software on the market and every manufacturer has own techniques to develop and implement them. Because of this manifoldness there has to be an interface between hardware and software components of different manufactures. These interfaces are defined by specifications and without them the progression would be more complicated because it is almost impossible that software can communicate with various hardware components. A specification is a description of a product to define features like limiting values or communication interfaces (PAT 05).

One important function of a computer for example is the way to display the graphic content. For this purpose a graphics card is used to reach more detailed resolutions with evermore complex, visual contents. During the software execution, the processor compiles the data into machine code. The graphics card calculates the output by the given machine code and an output device like a monitor or a projector displays the rendered context. There are two kinds of specifications for graphics cards. On the one hand for the construction and on the other hand for the software to control the graphics card (KOM 13).

The specification for the hardware is mostly written by interest groups including manufacturers for controllers, graphics cards and graphics chips. The software specifications are working in a similar way. In order that many software vendors can produce software for all graphics cards, a graphics interface is needed like Direct3D or OpenGL (see Figure 1). Without using a graphics interface, every program has to be written for a specific driver of a graphics card manufacturer. Consequently the programming effort would be many times higher and software would much more expensive. In this case software vendors have to concentrate for a small amount of graphics cards (SAL 13).

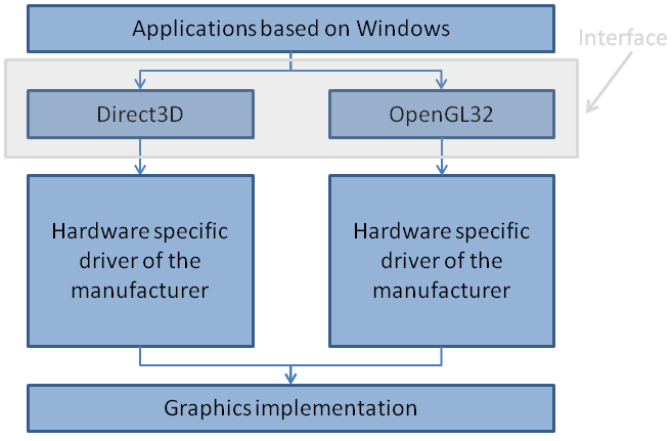


Figure 1 APIs

Generally graphics interfaces support that as many developers as possible can develop software for as many different performing graphics cards as possible. Direct3D and OpenGL are used for application software or application programs. Examples for application programs are CAD - programs, simulations or games. While Direct3D is only working on Microsoft Windows systems, OpenGL is portable and works on almost every system like Windows, UNIX and portable devices. In comparison to Direct3D, OpenGL is a free available application programming interface (API) without costs through the use of developers. Graphics interfaces support that as many developers as possible can produce software for as many different performing graphics cards as possible.

APIs and other forms of standard specifications exist to make it possible to exchange implementations of these APIs without clients noticing this change. Such interchangeability has the purpose of allowing for competing implementations. With these come lower costs, choice of quality-of-service, and faster innovation. Using the OpenGL API as an example, this thesis analyses three OpenGL implementations for their compliance with the standard. It shows how the API specifies wanted and unwanted variation. It makes suggestions how to make the desired variation clearer and how to avoid unwanted specification.

1.2. The OpenGL graphics system

Open Graphics Library (OpenGL) is a specification for a platform and programming language independent interface and consists of several hundred commands to describe more complex 3D scenes in real time (OPE 13).

OpenGL shoved all responsibilities for hardware drivers to the hardware manufacturers and transferred video display functions to the operating system. That is why the specification does not contain information about a context menu and does not support other hardware components like audio, windowing or input devices. Developers have to use other mechanisms to combine user inputs and OpenGL (WRI 11).

1.3. History

In the 1980s software developers had to challenge with a wide range of graphics hardware. Silicon Graphics (SGI) was trying to reduce these challenges by developing the first graphics API so-called IRIS GL. IRIS GL became the industrial standard API, because this API was easier to use then everything before. SGI had competitors like IBM, Sun Microsystems or Hewlett-Packard who also tried to bring up graphics API's. Increasing the market share of SGI, IRIS GL was changed to an open standard and released as OpenGL 1.0 in January 1992. Innovations for this standard were realized by OpenGL Architecture Review Board (OpenGL ARB) between 1992 and 2006. Companies like AMD/ATI, Apple, 3DLaps, IBM, NVIDIA, Intel and many more were members of the ARB.

Until 2001 OpenGL was leader on the graphics API market but the main competitor Microsoft set a milestone with the introduction of DirectX 8. For the first time Microsoft's API was more than a copy of SGI and brought some real innovations such as support for vertex and pixel shaders. Along the way the principal source of income for SGI was to sell expensive 3D-workstations, therefore SGI missed out the trend of graphics cards for gamers. ATI and NVIDIA launched huge amounts of low budget graphics cards on the market and SGI could not keep up with the competitors (ABI 08). Furthermore OpenGL's development was also restrained by disputes among the supporters. The OpenGL ARB consists of many different and competing companies which followed their own agenda and new features that should have been added to the API were not implemented. In 2004 OpenGL 2.0 was published and the introduction of the OpenGL Shading Language (GLSL) was the major innovation. Since OpenGl 2.0 could not compete with DirectX, the Khronos Group adopted the further development of the OpenGL specification in 2006. The Khronos Group is an industrial consortium that has more than 100 members like Google, Intel, NVIDIA or AMD/ATI. OpenGL 3.0 was launched by this consortium in August 2006. In this release certain functions were marked as deprecated and can be completely replaced with more modern functions. Up to the deprecation of certain functions the specification was divided into the core profile and the compatibility profile. In March 2010 OpenGL 4.0 was released with new features like Tessellation and OpenCL. OpenGL is currently available in version 4.3 (LUT 12, WOG 13).

1.4. Implementation of new functions

New functions are decided by the Khronos Group including graphics cards designers, operating system designers and general technology companies. To accelerate the development of functions for OpenGL, graphics cards manufacturers can add functions in the form of extensions. With these extensions the manufacturers have the chance to publish new functions and constants and to remove restrictions of existing OpenGL functions. Extensions have no need to be validated by other members of the Khronos Group. They get a short identifier based on the name of the company for example "NV" for NVIDIA. If more members decide to implement the same function, a shared extension will be used with the identifier "EXT". After the whole Khronos Group decided to use the new extension, the extension gets the identifier "ARB". Only features with the extensions "EXT" or "ARB" can be added to the core profile of new OpenGL releases (WOG 13).

2. Implementations

2.1. Introduction

There are many different implementations of graphics cards as use cases. The most favorite manufacturers are NVIDIA, ATI, Intel and S3 Graphics. All of them offer different kinds of cards divided in low-end or onboard solutions, business solutions, high-end solutions, professional solutions and software-renderer.

2.2. Onboard-solutions

The functionality of the graphics card is integrated within the chips of the main board or within the processor (e.g. Intel i5). This is generally called Integrated Graphics Processor solutions (IGP). IGPs provide all 2D functions but they are limited with 3D functionality. Mostly the 3D functionality is slow or outdated. Therefore IGPs are commonly used in areas with lower graphics requirements. Due to the lower power consumption they are often used in notebooks. In extremely compact or cheap solutions there is no need for a dedicated graphics memory. In these cases the main memory of the computer is used for graphics applications, which has a negative impact on performance (KOM 13).

2.3. Low-End - / Business solutions

Low-End solutions are full-fledged graphics cards with little attention for 3D features. The features are designed for a sharp and high-contrast image. They have less processing units, lower chip and memory clock rates and slower memory access compared to High-End or professional solutions (KOM 13).

2.4. High-End solutions

High-End solutions are representing the technically feasible features in the area of 3D-programming. Graphics cards with full hardware support for OpenGL can calculate almost all 3D functions in real-time. These cards are mostly useful for gamers, because the system is optimized for anti-aliasing, tessellation and high density of detail (KOM 13, WGK 13).

2.5. Professional solutions

Professional solutions are especially suitable for CAD and GIS applications. These cards offer special functions, that can only be emulated on the most other graphics cards or the calculation is much slower. Only AMD/ATI and NVIDIA are offering solutions for the OpenGL workstation segment. Both companies are using derivations of their High-End cards with modified ROMs and drivers. The modifications are more optimized for 2D-rendering of OpenGL than 3D-rendering of DirectX and OpenGL (WGK 13).

2.6. Software Rendering

Software rendering refers to a method of graphics calculation without specialized graphics hardware. All calculations are done by CPU. The graphics card forward the data to the output medium without any calculations, therefore cheap graphics cards without dedicated memory are enough for this task. The performance of software rendering is much slower in comparison to real graphics cards, because CPU is not optimized for graphics calculations (WSR 13).

2.7. Selection of Implementations

This thesis compares three different implementations of graphics cards. Since modern onboard and Low-End solutions mostly have the same functionality, taking one graphics card from the low budget segment of cards is sufficient. Since High-End cards and professional cards use almost identical hardware with one main difference within the OpenGL optimized driver of professional solutions, a graphics card from the field of professional solutions is taken for examinations (SAL 13). A software renderer is suiting as a third implementation. Although the rendering of a software renderer will be slower, there are other advantages. There are situations in which a conscious renunciation of the use of graphics cards can be useful.

For example during the past few years two substantial transformations in the IT world are virtualizations and cloud computing. With the next step virtual machines will be brought into the cloud. CPU, RAM, disk and other computing resources can be selected within the cloud. The selected hardware is provided on servers and a High-End computer which is running remotely on such a server can be controlled on a low performance workstation. This offers many cost advantages because multiple workstations can share the hardware. In case of a multi-core CPU the server can simply split the multi-core to single-cores to provide them for single virtualizations. But the server cannot split the graphics card; therefore the server would need a single graphics card for every virtualization or uses software rendering. Furthermore software renderers are often used as a fallback to simulate missing functions of graphics cards.

Covering a broad range of possible implementations, various manufacturers have to be considered. As additional condition only graphics cards that passed the conformance tests of the Khronos Group will be chosen, confirming that the graphics cards are completely supported by OpenGL. For this reason the following graphics cards were selected. The onboard graphics chip Intel HD Graphics 3000 will be used as card from the low budget segment (WHD 13). The NVIDIA Quadro FX 5800 will be used as card from the high budget segment. The open source renderer Mesa3D is chosen as software renderer. Mesa3D offers drivers for graphics cards for UNIX distributions and uses a software renderer as fallback(MES 13). But the software renderer can be used as standalone solution also

working on Windows systems. Information to the underlying system, operating system and driver version can be found in Appendix Systems.

3. OpenGL

3.1. Selection of the Specification

Graphics cards are designed for different tasks. Onboard cards often only support 2D graphics or slow or outdated 3D graphics. As a conclusion not every graphics card which is tested will support the latest OpenGL version. The maximal supported version of OpenGL is anchored in the driver. The program GLView shows the supported extensions of OpenGL of a graphics card and the program also submits the supported and the unsupported functions.

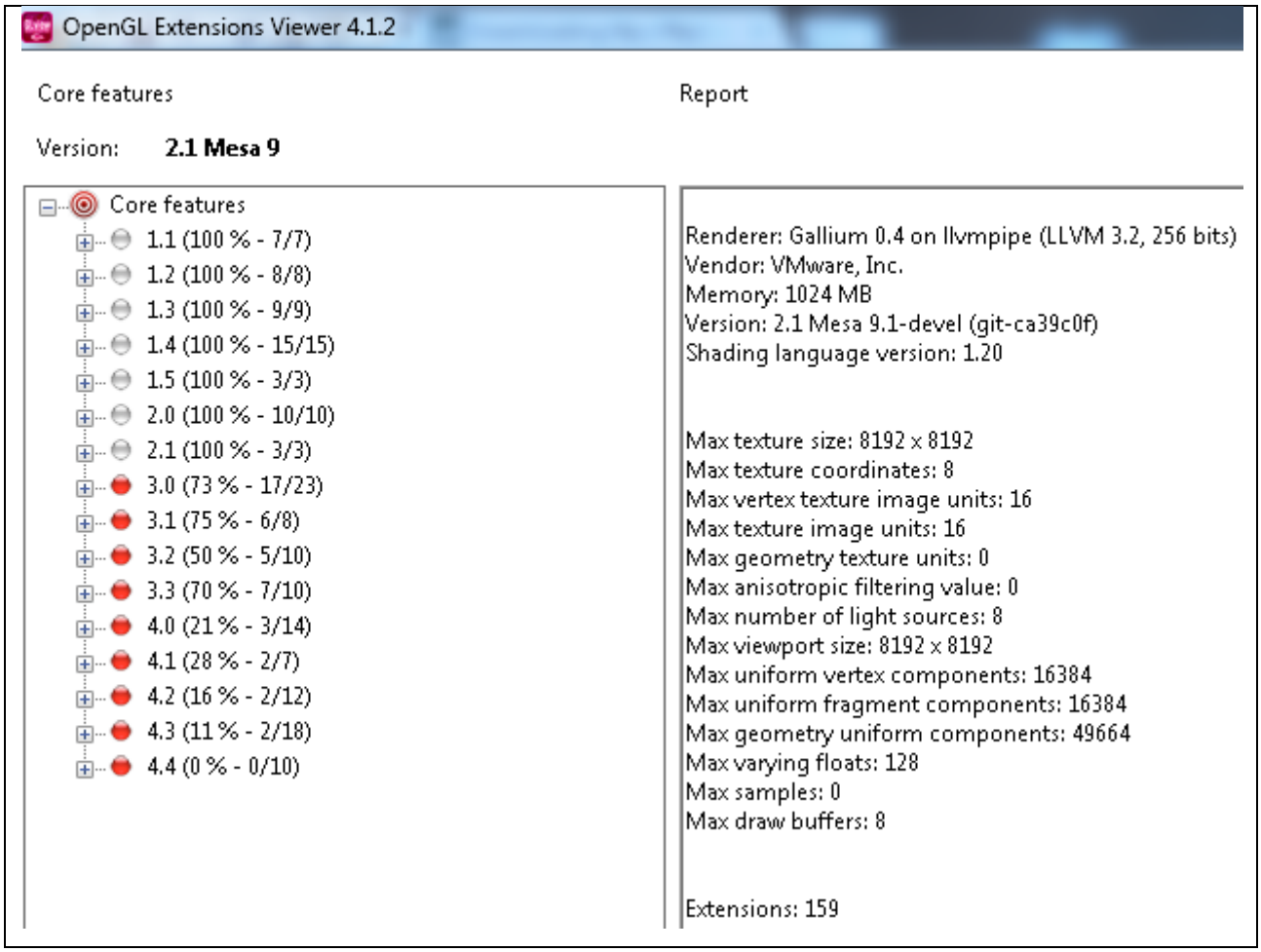


Figure 2 GLView

Figure 2 shows that Mesa3D 3D supports OpenGL 2.1. In comparison the NVIDIA card supports the functions of OpenGL 3.3 and the Intel HD Graphics 3000 card supports version 3.1. Since OpenGL is backward compatible, this thesis will focus on the functions of the maximal supported OpenGL version of the weakest graphics card. All implementations will use their latest OpenGL drivers with the limitation to the functions of OpenGL 2.1. The latest drivers of the implementations will be used because bugs have been fixed and limits of input values have changed in comparison to older versions.

3.2. Subset of OpenGL

There are two main differences between OpenGL 2.1 and the newer versions OpenGL 3.1 and 3.3. On the one hand deprecated functions were reduced by the new versions and the code base was cleaned up, that a modern programming is possible with OpenGL. Functions like glBegin, glEnd were removed and also the fixed-function-pipeline, the direct-drawing-mode and the color-index-mode were abandoned from the code base (SEG 06, SEG 09, SEG 10). These functions are only available with optional “compatibility extension” packs. On the other hand the new OpenGL releases support a higher shader language than OpenGL 2.1. The term shader language will be explained in chapter 3.3. Consequently only a subset of functions of OpenGL 2.1 will be analyzed in this thesis. Doing an intersection of the functions of OpenGL 2.1, OpenGL 3.1 and OpenGL 3.3 is one possibility to select the subset, as shown in Figure 3.

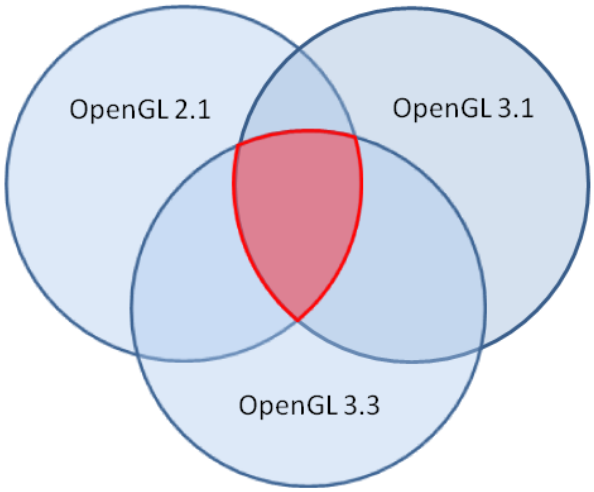


Figure 3 Intersection of three OpenGL spezifications

Since this subset still contains too many functions, a different approach was adopted to select a better subset. OpenGL was written platform independent and was used for personal computers originally. Besides the largest three operating systems Windows, Linux and Mac also smaller systems like Solaris and FreeBSD are supported by OpenGL. Due to the rapid development of embedded systems like mobile devices and tablets or gaming consoles, the used hardware is becoming more and more powerful and better 3D applications will be written for these hardware. These 3D applications are also using OpenGL. Since the graphics performance of mobile devices does not reach a standard personal computer, the Khronos Group has adopted a stripped-down specification of OpenGL with the name OpenGL ES (OPE 13).

Similar to OpenGL 3.1 redundant functions, leading to the same results, have been removed from OpenGL. OpenGL ES 2.0 bases on OpenGL 2.0 but uses the concept of modern programming like OpenGL 3.1. In order to get a better performance on embedded systems, some data types have been abolished or reduced. For example the data type double does not exist in OpenGL ES 2.0 (MUN 11).

All functions with the input parameter double are changed to data type float. If this restriction will be repealed, OpenGL ES 2.0 offers a manageable subset and enables a modern programming in OpenGL shown in Figure 4. Appendix Functions lists an overview of all functions for the selected subset (MUN 10).

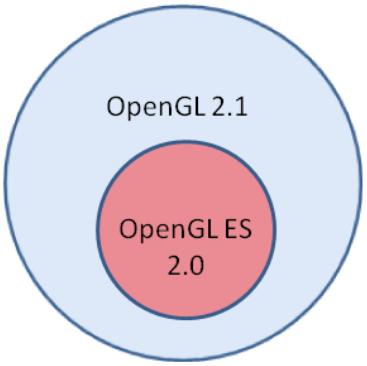


Figure 4 OpenGL function subset

3.3. Subset of GLSL

Shaders are hardware or software modules that implement a particular effect within the 3D computer graphics. The standardized programming language to get access on the GPU with OpenGL is called OpenGL Shading Language (GLSL). Thereby individual vertices and fragments are changed within the graphics pipeline. Some essential tasks are for example texturing and lighting. At the beginning of OpenGL only the fixed-function pipeline existed, where individual calculation steps of the shader are immutable and only single parameters can be set through the application. Graphics cards used defined algorithms of the manufactures. Since OpenGL 2.0 a programmable-function pipeline allows to freely define programs. Thus, for example, a specific illumination model or texture effect can be implemented (ROS 10).

The initial GLSL version offered only a vertex and fragment shader (KES 04). For modern graphics cards that support OpenGL 4.0 or higher, there are four different types of GLSL shaders: vertex, tessellation, geometry and fragment shaders (SEG 10, SEL 10). Since this thesis deals with OpenGL 2.1 tessellation and geometry shaders are only mentioned for completeness.

The further development of the fixed-function pipeline has been discontinued with the introduction of OpenGL 3.0 and is no longer used in modern OpenGL. The fixed-function pipeline is also no longer part of the specification. In OpenGL ES2.0 the fixed-function pipeline is not supported at all. Since OpenGL ES2.0 was selected as a subset of the examination and OpenGL 2.1 does not support tessellation and geometry shaders, this thesis is limited to the programmable-function pipeline with support for vertex and fragment shaders.

Subset of GLSL

GLSL is a programming language similar to C that is specially adapted to the needs of shaders. There are built-in types for vectors and matrices and a variety of math and graphics functions. In contrast to C, there is no pointer. Each shader type has characteristic input and output parameters. The shader source code and any additional variables and constants are passed at run time of the 3D application to the graphics card driver. The driver compiles and links the shaders to a shader program. For low-cost graphics chips, the shader units are often omitted, that the shader is using the CPU for calculations, which is much slower than the GPU (ROS 10).

3.3.1. Programming Function Pipeline

Figure 5 shows the functionality of the programmable-function pipeline also called graphics pipeline (LEA 13).

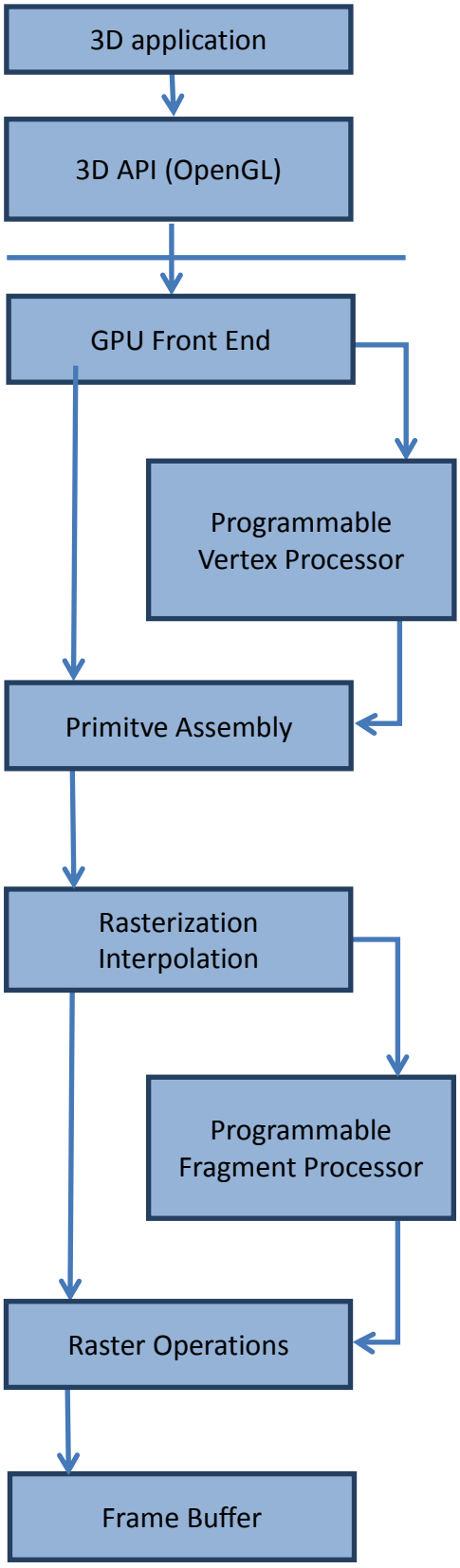


Figure 5 Programmable-function pipeline

Driver and GPU Front End

Each primitive, which is drawn by the application, is passed through the API to the GPU Front End. Thereby the Front End receives commands and data from the API.

Programmable Vertex Processor

The vertex shader is executed for each vertex once. The shader achieves only access to the actual vertex including its texture coordinates, normals and other data but has no information of the neighboring vertices, the topology or the like. The vertex shader performs geometric calculations and dynamic changes of objects.

Primitive Assembly

This step connects the manipulated vertices and constructs a triangle. Graphics cards perform best with triangles because triangles are the smallest area within a plane. The graphics card has not to handle curvatures.

Rasterizing and Interpolation

The coordinates of the triangles will be transformed to pixel coordinates and all pixels between vertices will be interpolated with the information of the manipulated vertices from the programmable vertex processor.

Programmable Fragment Processor

The fragment shader is executed for each pixel once and determines the resulting pixel color.

Raster Operations

The raster operations are the final operations before the picture can be shown on the display. There the triangles will be combined and linked with the background.

Frame Buffer

The results are stored in the frame buffer and the graphics card displays the data directly on the screen.

3.4. Drawing in OpenGL

OpenGL was designed as a state machine; therefore a state is set until the state is changed. The advantage is that a state which is set once, can be used for many function calls without any changes. As an example in pseudo-code:

```
setColor (0, 0, 1);    // set the color to blue
DrawTriangle (1);   // draw a triangle at position 1
DrawTriangle (2);   // draw a triangle at position 2
setColor (1, 0, 0);   // set the color to red
DrawTriangle (3);  // draw a triangle at position 3
```

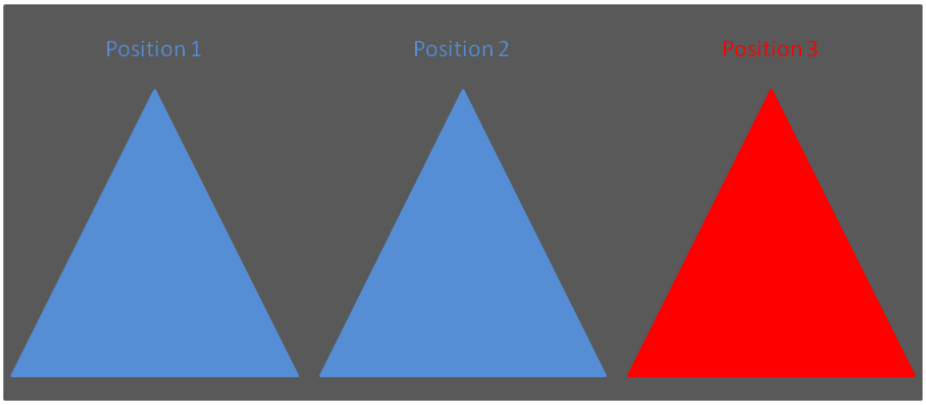


Figure 6 Three triangles

In Figure 6 the principle of the state machine can be seen. Because the color is set to blue by the command setColor(0,0,1) the following will be drawn in this color until the color is changed. Therefore the two triangles at position 1 and position 2 are drawn blue. After the color has been changed with the command setColor (1,0,0) to red, the triangle at position 3 is drawn red.

Many parameters can affect the appearance of rendered objects, for example objects can be textured, lighted, stretched, shifted, transparent or opaque or they can have a rough or smooth surface.

The reason for this design is that almost any change of the drawing mode needs extensive reorganization of the graphics pipeline. Rewriting dozens of parameters again and again would be exhausting for the programmer. Often many thousands of vertices can be processed before a status has to be changed, and some states never change. For example, light sources mostly remain at the point for all objects in a scene(MUN 11).

The implementation of the OpenGL API is usually carried by system libraries of the operating system (SHR 10, SHR 06). While all OpenGL applications are written in C++ for this thesis, OpenGL supports also many other languages. Thus, there are large communities for Java or C with very detailed tutorials.

3.4.1. Create an OpenGL context

Creating an OpenGL context is the process of initialization to use an OpenGL implementation. A context is localized as a process in any OpenGL application on all operating systems. The creation of an OpenGL context is not given by the OpenGL specification, because using OpenGL means to develop portable applications. But creating a context menu is done differently on every platform. To abstract this process using libraries is the most common way. Libraries abstract this process and in this manner the same codebase can be used on all platforms. This thesis is only covering Windows-platforms. The following pseudo-code shows the abstract structure of an OpenGL application:

```
#include <libraries>
int main()
{
	initOpenGLContext(settings, title, width, height);
	while ( contextExist )
	{
		while (!newEvent())
		{
			doEvent( event );
		}
		drawGraphics()
		presentGraphics();
		updateDisplay();
	}
	return 0;
}
```

A context contains all OpenGL properties and states of the existing instance. By closing the application, everything will be cleaned up and the OpenGL context will be destroyed. The properties which are stored within the context are for example the title, the size and settings like the anti-aliasing level. After the context is initialized the application has to implement features like event-handling for example mouse clicks or keyboard entries and features to handle the window for updating the rendering state and the drawing (OVE 12).

3.4.2. Draw simple primitive

Producing the final output image means that input data have to pass all steps of the graphics pipeline. Figure 7 is a strongly abstracted view of the graphics pipeline/ programmable-function-pipeline. These steps are mandatory (OVE 12).

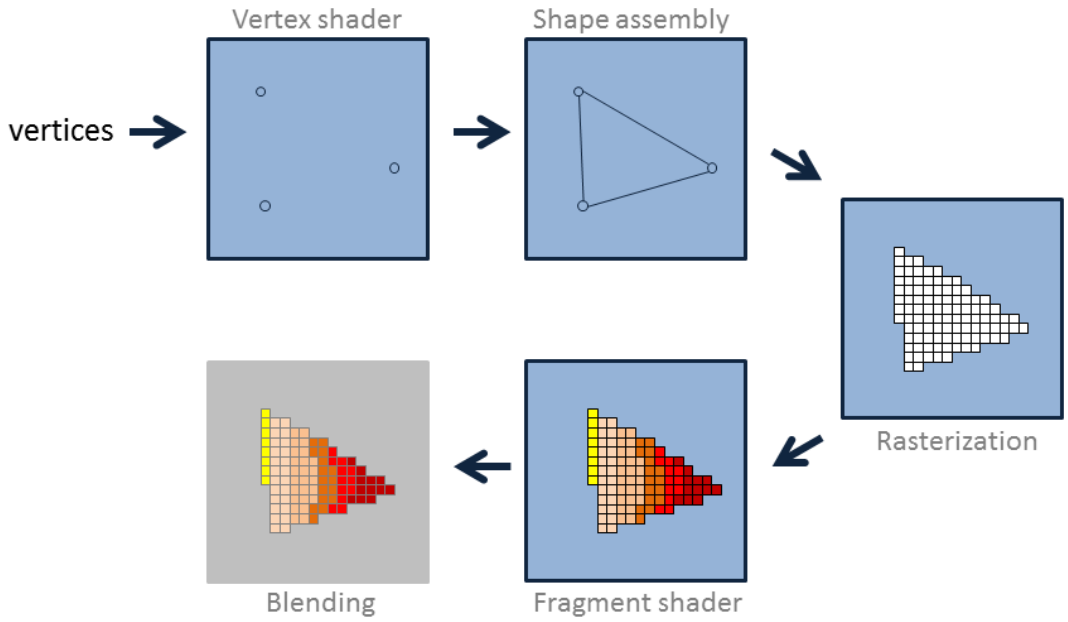


Figure 7 Abstract of drawing a simple primitive

The input data consists of vertices. These vertices are points constructing shapes like triangles and containing attributes. Commonly the attributes are texture coordinates, colors or normals. The developer decides what kind of attributes is stored.

Within the programmable-function pipeline the vertex shader is a small program where the perspective transformation takes place. Also colors, texture coordinates and other important attributes will be passed further down the pipeline through the vertex shader. After the input data have passed the vertex shader, the graphics card will model primitives. Primitives are shapes out of the vertices like points, lines or triangles and form the basis of more complex figures. A triangle is the simplest plane in a 3D space. There are some other primitives and drawing modes but triangles are sufficiently complex for all drawings.

During the rasterization process the whole list of shapes will be converted into pixel-sized fragments. These fragments are the input of the fragment shader, whereby the vertex shader already interpolates the attributes like the colors of the shapes. That is why three vertices are enough to create a whole triangle.

The fragment shader produces the final color of each fragment, which will be presented in the output image. There are several different operations used by fragment shaders like texturing,

coloring, lighting, shadowing or other special effects. Equal to the vertex shader, it is a small program within the programmable-function-pipeline produced by the developer.

3.4.3. Vertex input

Vertices have to be passed in device coordinates. The coordinate system in OpenGL is a right-handed Cartesian coordinate system (Figure 8, SAL 13).

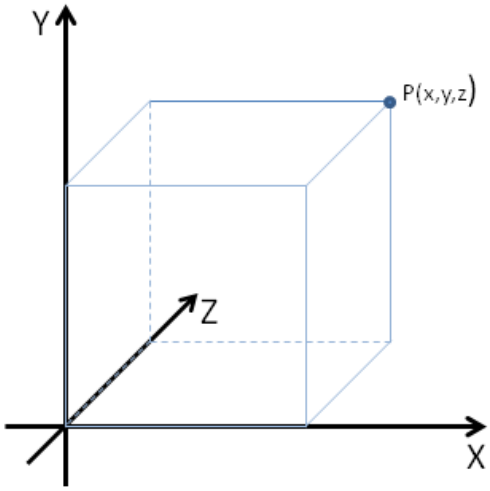


Figure 8 Right handed coordinate system

To enable a resolution-independent programming, the device coordinates are between -1 and 1. Compared to standard graphics programs like GIMP, where the coordinate origin is in the upper left corner, the origin is at the image center in OpenGL. This coordinate system offers many advantages for calculations. For an easier understanding, Figure 9 shows only the x-and y-axis at that point (OVE 12).

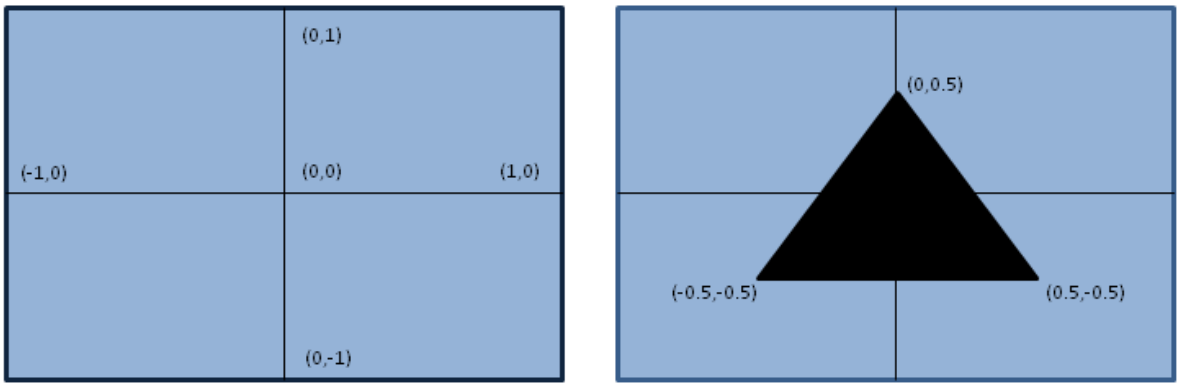


Figure 9 XY-coordinate system

3.4.4. Texturing

Texturing describes a method, where the area of three-dimensional models is filled with two-dimensional images, the so called textures, and is endured with surface properties. So the appearance of computer generated images can be displayed more detailed and more realistic without refining the underlying model. In order to do texturing, each vertex gets a texture coordinate. In this manner the polygon is notified how the texture has to be imaged on the surface. Texture coordinates consist of UVW-parameters.

The most typical applications use two-dimensional textures, therefore only the UV-coordinates are required to determine which part of the image is mapped on the polygon. Mathematical or volumetric textures often use the W-coordinate, but this is not part of this thesis. Figure 10 illustrates the UV-mapping (WUV 13).

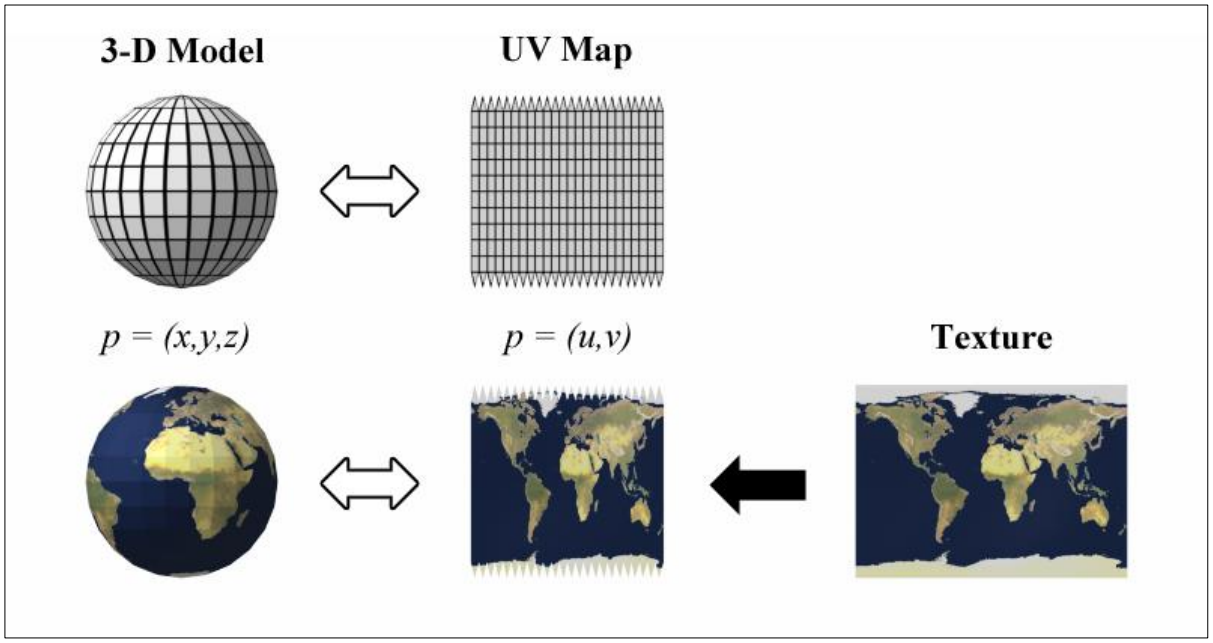


Figure 10 UV-Mapping

3.4.5. Wrapping

The UV-coordinate (0, 0) corresponds to the lower left corner of the texture and the UV-coordinate (1, 1) to the upper right corner. To achieve overlapping effects multiple textures can also be put on each other. UV-coordinates can also be greater than 1 or less than 0 to generate edge repetition effects of the texture. These edge repetition effects are called wrapping. OpenGL offers 4 ways to handle wrapping. Figure 11 gives an overview of the 4 options (OVE 12).

- GL_REPEAT: Only the decimal places of the coordinates are used, thereby a repeating pattern is formed.
- GL_MIRRORED_REPEAT: The decimal places of the coordinates are used to a repeating pattern; if the integer part is odd then the pattern will be mirrored.
- GL_CLAMP_TO_EDGE: The texture will be stretched between the borders.
- GL_CLAMP_TO_BORDER: All coordinates outside the given range will be set to a specified color.

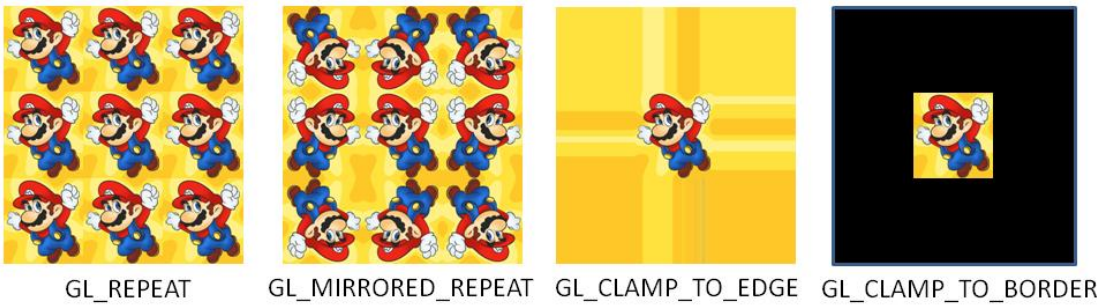


Figure 11 Wrapping

3.4.6. Filtering

Since drawing with OpenGL is resolution independent, also texturing has to be independent. One consequence is that the pixels of the primitives will not always match to the pixels of the texture, because images have to be stretched or sized down. Therefore OpenGL offers operations for filtering.

Usually OpenGL uses Mip-Mapping. Mip-Mapping is an anti-aliasing technique for textures and is used in modern 3D graphics chips with the intention to increase rendering speed and reduce aliasing artifacts. Aliasing effects especially occur through scaling of images as disturbing artifacts. These artifacts are getting worse at patterns with oblique or rounded lines as a distortion or flickering.

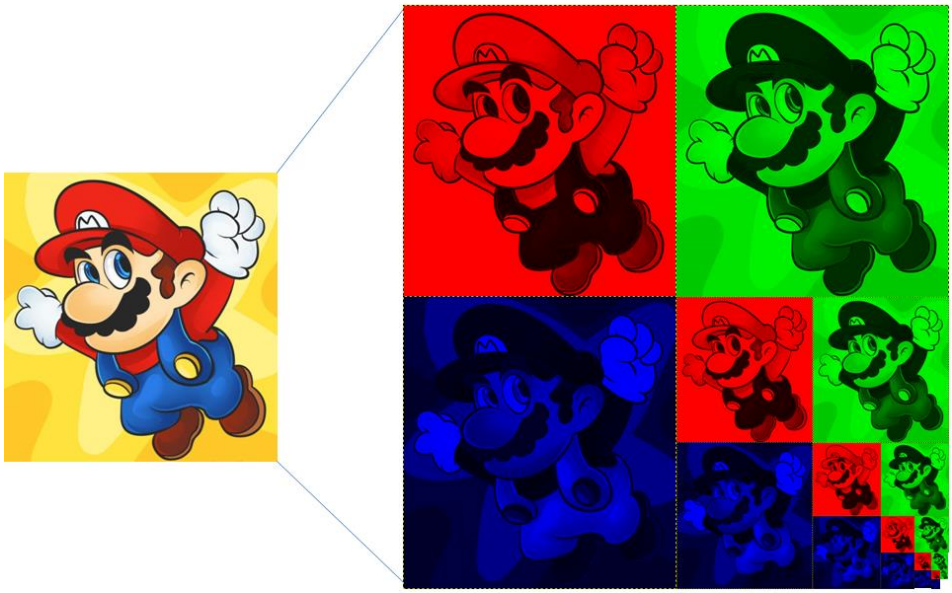


Figure 12 Mip-mapping

Thereby a set of smaller textures with a reduced level of detail is generated out of the main texture (Figure 12). Since Mip-Mapping would require a lot of performance to render in real time, these textures are pre-compiled. Mip-Maps have up to 1/3 higher memory consumption than the largest texture alone (textures are saved in RGB, therefore 3 pictures are stored). The performance increases through the use of smaller textures, because the real-time renderer adapts the smaller textures faster to the corresponding polygons. Furthermore artifacts are reduced because Mip-Maps have been effectively filtered already.

The two disadvantages of Mip-Maps are the larger memory consumption and that textures can only be reduced by the size. If only small textures are available or graphics cards have little memory then other filter-modes should be used such as linear-filtering or nearest-filtering. Figure 13 gives an overview of the two filtering modes (OVE 12).

- GL_NEAREST: Returns the value of the texture element that is nearest to the center of the pixel being textured.
- GL_LINEAR: Returns the weighted average of the four texture elements that are next to the center of the pixel being textured.

Figure 13 Filtering modes

3.4.7. Matrices

Essentially a matrix is nothing more than a multi-dimensional array in programming. Matrices are important to understand OpenGL. They describe translations, rotations and dilations. Different results are produced by changing the order of the commands to manipulate the matrix. Each manipulation transforms a vector into a new coordinate system. Transforming a vector is going step by step therefore all 3D graphics applications follow a process similar to the shown process in Figure 14.

Figure 14 Transforming process

The final transformation is the cross product of the model, view and projection matrices.

$$v' = M_{proj} \cdot M_{view} \cdot M_{model} \cdot v$$

In older Versions of OpenGL the programmer was forced to use model-view and projection transformations through the fixed-function pipeline. The model and view transformations have been combined into one matrix. Modern OpenGL allows modifying the view matrix independent from the model matrix. For the following explanation the y-axis will be excluded for simplification.

3.4.7.1. World Matix

At the beginning of a matrix calculation every primitive or object is within the object space. The object space is the basic state without any transformations. The objects are committed to the world matrix or also called world space to transform them into objects which reflect the reality (VPM 10).

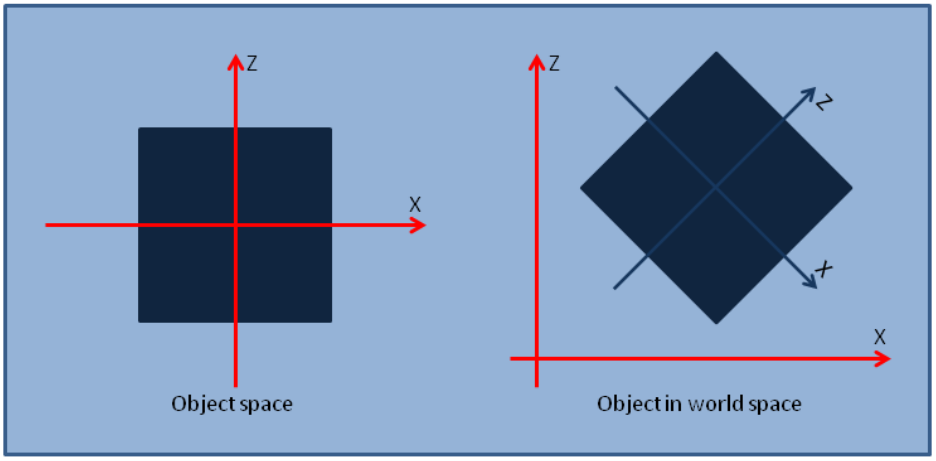


Figure 15 World matrix

3.4.7.2. View Matrix

The view matrix is also known as the camera matrix. The entire world space will be transformed into eye coordinates. In real life the camera has to be moved to get an alternative view of a specific scene. In OpenGL the camera cannot be changed. The world is moved and rotated around the camera. For example instead of moving the camera up, in OpenGL the world has to move down. Figure 16 shows the transformation from the world space into view space(VPM 10).

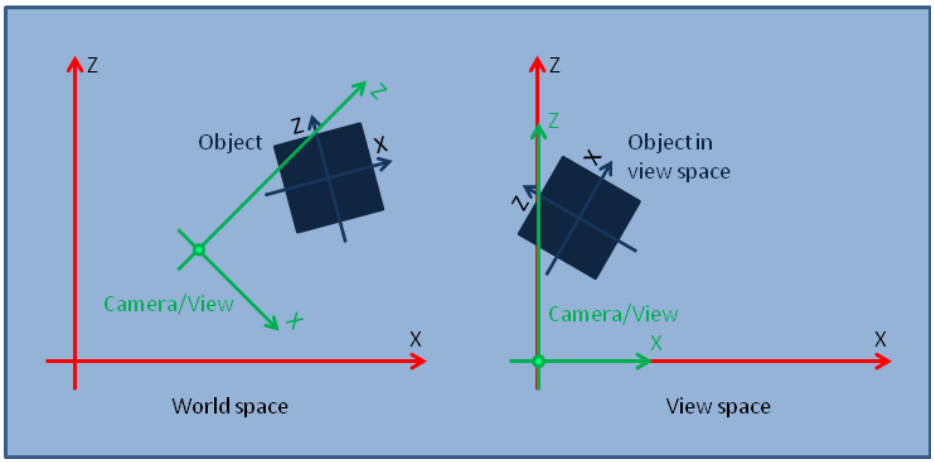


Figure 16 View matrix

3.4.7.3. Projection Matrix

The projection matrix transforms the eye coordinates in clip coordinates. This matrix handles the near and far view distance, the screen resolution settings and the angle of view. Without using the projection matrix the picture has no perspective, that means all projected objects have the same size and they are parallel. With the projection matrix the picture can be clipped. Clipping means to cut off basic objects at the edge of a desired window. Because of the clipping the picture gets a perspective like shown in Figure 17 (VPM 10).

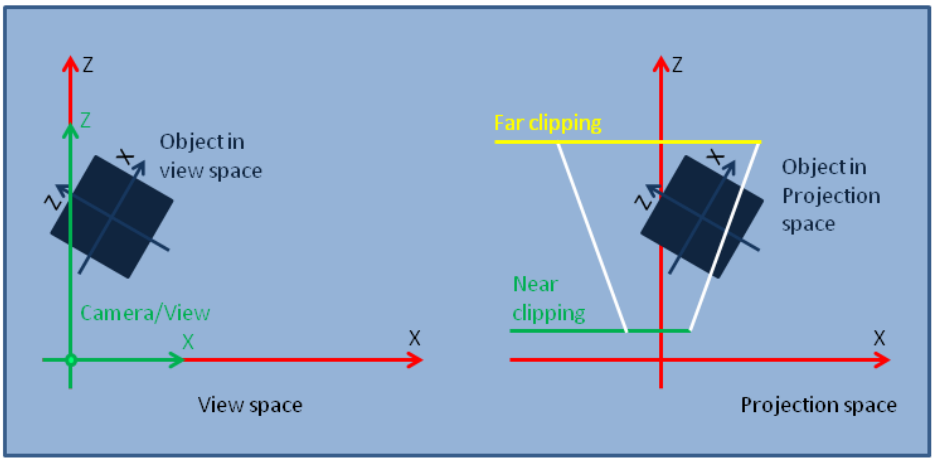


Figure 17 Projection matrix

3.4.8. Modeling

The computer graphics wants to generate images out of the description of artificial scenes. Since nowadays scenes are composed from millions of images, a programmer is not able to create complex scenes in OpenGL through the manually input of primitives. Therefore objects are created as models by external tools and loaded with special loaders in OpenGL. The main concern is to model objects as realistic as possible in high-detail. Models are stored in many different containers. The containers have information about positions, textures, light positions and other properties. There are many applications which can generate such containers like Photoshop, 3ds Max, Blender or Maya. In this thesis the container format OBJ is used. The OBJ format was designed by the company Wavefront Technologies and is adopted by almost all animation programs. The OBJ format is one of the easier formats because OBJ-files generally relate to static models. The vertices and other properties to the model are written down in this container. The models pass through the whole drawing process as mentioned from chapter 3.4.1 to chapter 3.4.7. For example the coordinates have to be converted to device coordinates. Then the coordinates are given to the shader and the model is textured there. These textures are usually stored in UV-maps. Subsequently the matrices are set to move and to rotate the model on the screen. Figure 18 shows a very simplified loading of an OBJ-file.

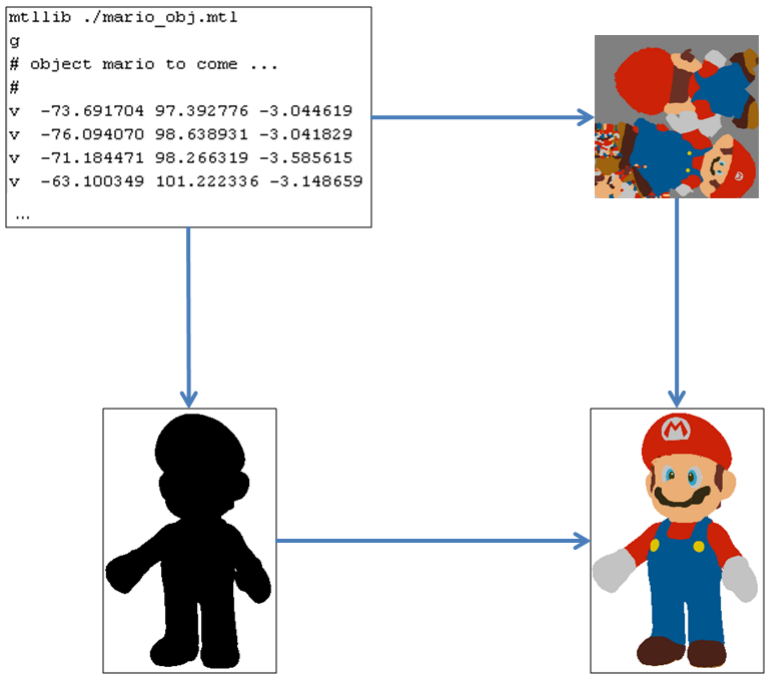


Figure 18 Modeling

4. Software Testing

4.1. Introduction

To verify the implementations of graphics card manufacturers against the OpenGL specification, OpenGL cannot be seen as a programming language but as software application. Therefore this thesis is concerned with software testing of OpenGL implementations. *Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item* (WIL 06). Software testing is an activity that should be done throughout the whole development process(BER 01). Since the technical capabilities of graphics cards are constantly improving, OpenGL also needs to be improved and tested cyclically.

Only those products are officially supporting OpenGL that pass the conformance tests of the Khronos Group. Theoretically, every implementation should generate the same results. To confirm this statement, every OpenGL command has to be tested, taking account of all possible permutations and input combinations. Only in this way all pieces of OpenGL could be verified. This type of testing is called exhaustive testing. Exhaustive testing is usually only done for very small projects. Finding all possible permutations and combinations is almost impossible for larger projects.

In addition, very high costs and an enormous amount of time are needed with exhaustive testing. For this reason, exhaustive testing has rather a theoretical relevance. Therefore software testing will be analyzed in more detail.

4.2. Verification and Validation

Software testing is part of the Software Validation & Verification Plan (SVVP) defined in IEEE 1012. SVVP describes a minimal standard of software practices how a validation and verification have to be documented.

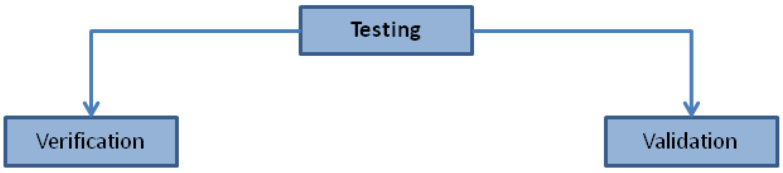


Figure 19 Verification and Validation

As shown in Figure 19, testing is divided into two main classes within this thesis. Verification is a process of software development which ensures that the software conforms to a specification. On the one hand a formal verification includes a formal specification and on the other hand a formal understanding of the meaning of „conforms". Conformity often means that the program only needs to meet a finite subset of possible inputs. Therefore a method requires an adequacy criterion that determines the minimum of the subset to test the test case properly. The second class is the validation. Validation is a process during or at the end of the development, which checks that the

requirements are fulfilled in practice. This is usually performed by programmers, external testers and customers in order to check that the software keep the conditions of the requirements. For example a customer orders a chess game and the programmer implements the game checkers, because he has misunderstood the requirements of the customer. The verification has been successfully conducted by the programmer, but the validation fails because the customer had expected something completely different. Generally two questions have to be answered, first of all whether the software has been built correctly (verification) and secondly whether the correct software has been built (validation) (IVV 98).

4.3. Economics of Software Testing

In software development, testing is always a cost factor. Testing takes time and that is why intensive testing is connected with high expenses. Therefore SVVPs have to be created to keep the costs as low as possible. Test cases have to be created and systematically executed. The target is to detect as many errors early and to eliminate them. Often errors that were found cannot be fixed, since they were discovered too late and repairs are too expensive or cannot be done until the release start.

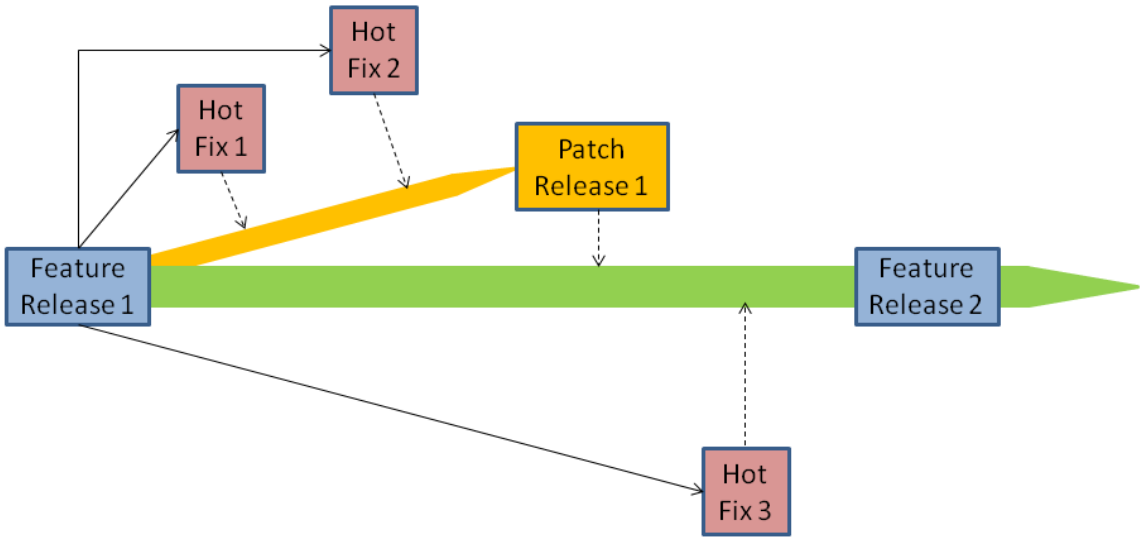


Figure 20 Release Cycle

Figure 20 shows a possible course how errors are compensated after the release start. In such a case, hot fixes are often restocked to the customer. If too many hot fixes are required, several hot fixes are grouped into patches and provided as major repairs (SER 13). Usually hot fixes and patches do not contain new features, instead they only correct errors. In order to keep these costs low, errors have to be found as soon as possible. A tester has to understand the economics of testing and to find as many bugs as possible in as few test cases as required (BMC 04).

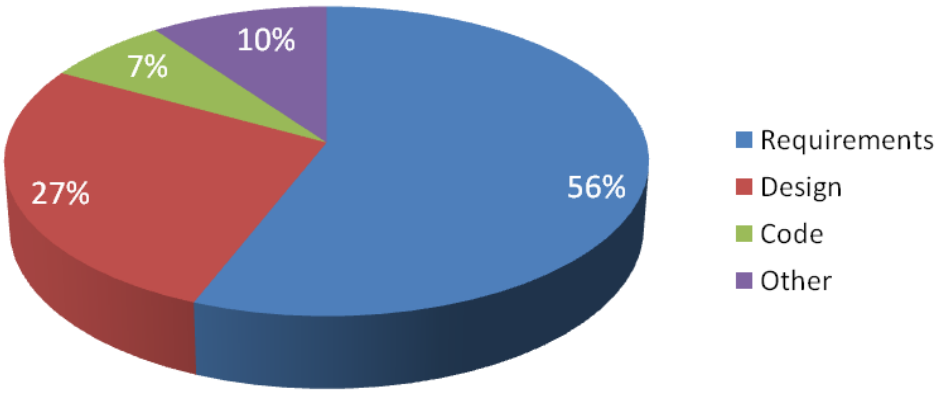


Figure 21 Distribution of Defects

Figure 21 shows at which point most of the errors can be found during software development(MCN 12). The information is based on long experience of many software testers. 56% of all errors are found in the requirements. Often requirements are not formulated sufficiently or they are not communicated properly between customers and developers.

Cost of Fix		Time Detected				
		Requirements	Design	Building	Testing	Post-Release
Time Introduced	Requirements	1	3	5-10	10	10-100
	Design	-	1	10	15	25-100
	Building	-	-	1	10	10-25

Table 1 Costs of fixing bugs

The costs increase exponentially with each project phase by eliminating a software error. Table 1 shows that an error costs one unit if the error is founded in the requirements during the definition phase of the requirements. A unit can be either an amount of time or a value of money(KAN 02). If an error in the requirements will be found during the system test, then the elimination of the error already costs 10 times more. Occurring during the delivery phase, an error costs 100 times more than a detected error in the definition phase.

Bringing together the statements of Figure 21 and Table 1 show that many errors are in the specification and software testing has to accompany throughout the whole development process, otherwise the costs of eliminating the errors are too high. Since this thesis deals with the implementation of the OpenGL specification, special attention is put on the requirements.

To cover as many test scenarios as possible for this thesis, some test principles have to be established first:

1. *Testing is the process of examining or executing a program with the intention of finding errors, and not to somehow deduce (or prove) that the software is error-free.*
2. *It is impossible to completely (or exhaustively) test any nontrivial module or system.*
3. *Testing takes creativity and hard work.*
4. *Test results should be recorded, for comparison with results that are later obtained during ``retesting,'' after changes have been made - in order to look for any unexpected or undesirable ``side effects'' of changes, as well as to see whether the changes helped.*
5. *Testing is best done by several *independent testers, and not (entirely) by the developers who designed and coded the system.(*CPS 97*)*

A general method for testing can be derived by these test principles. The first test principle mentions that knowing the exact functionality of the software is necessary to find errors in this software. Since this thesis considers OpenGL as software, the OpenGL specification corresponds as the requirements. Therefore the requirements have to be considered first of all. Often requirements are insufficiently specified and their behavior is unexpected. Therefore the specification of OpenGL has to be checked on inaccuracies, errors and other linguistic anomalies that suggest any errors in advance.

Since a test cannot check every possible behavior of the system (test principle 2), strategies for writing fewer test cases and still finding as many faults as possible have to be discussed in more detail.

On these strategies concrete test cases have to be developed and tested on the implementations (test principle 3). Ensuring of test principle 4, all source codes of the tests are supplied on the provided DVD. Test principle 5 is omitted in this thesis and was only mentioned for completeness.

4.4. Test-Case-Strategies

Generally there are two subspecies of the two main classes (verification and validation). On the one hand there is black-box testing and on the other hand there is white-box testing. Black-box testing is also called functional testing and ignores the internal structure of the system or of the function. Black-box testing focuses exclusively on the output that is generated in response to selected inputs. The tester knows the output corresponding to the input, based on requirements. The term white-box testing is a method of software testing whereby the tests are developed with knowledge of the inner workings of the system. In contrast to black-box testing, white-box testing needs a look into the source code. Black-box testing and white-box testing are available for verification as well as for validation (Figure 22).

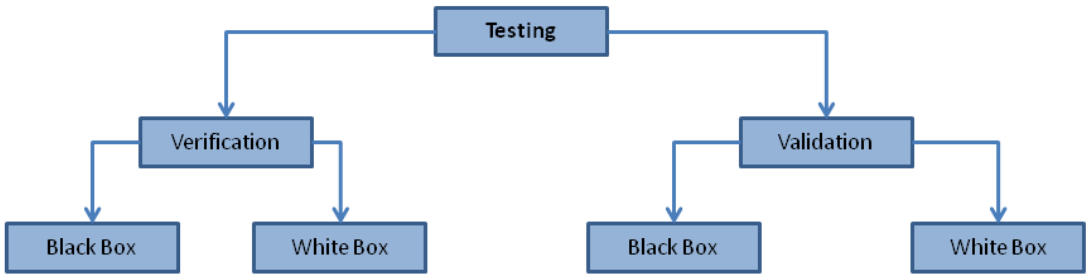


Figure 22 Black- and white-box testing

In the case of SVVP black-box testing is mostly used for validation and white-box testing for verification. Since OpenGL is considered as software, the implementation of the graphics card manufacturer (graphics card driver) corresponds to a black box. That means the result is output at the output medium like a monitor when the input parameters of individual OpenGL functions were changed. The internal mechanisms of the commands within the graphics card driver are only known by the graphics card manufacturer(CPS 97). Almost all source codes of a graphics card are not publicly available. For this reason only black-box testing is discussed in the thesis, since white-box testing is not possible (Figure 23).

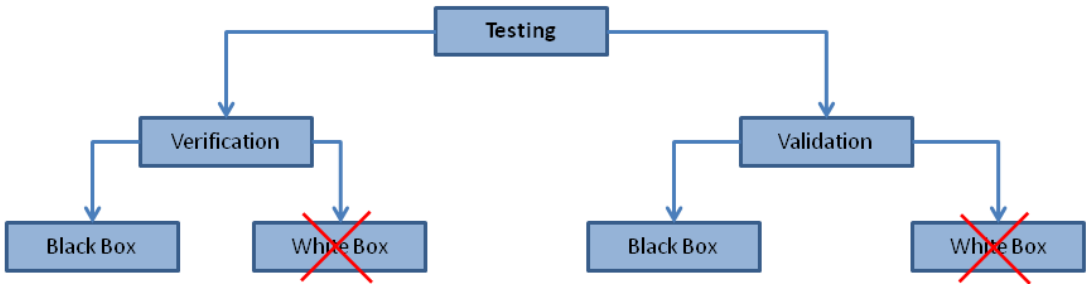


Figure 23 Black-box testing

4.5. Validation

Black-box testing of the main class validation can be divided in functional testing and non-functional testing as shown in Figure 24.

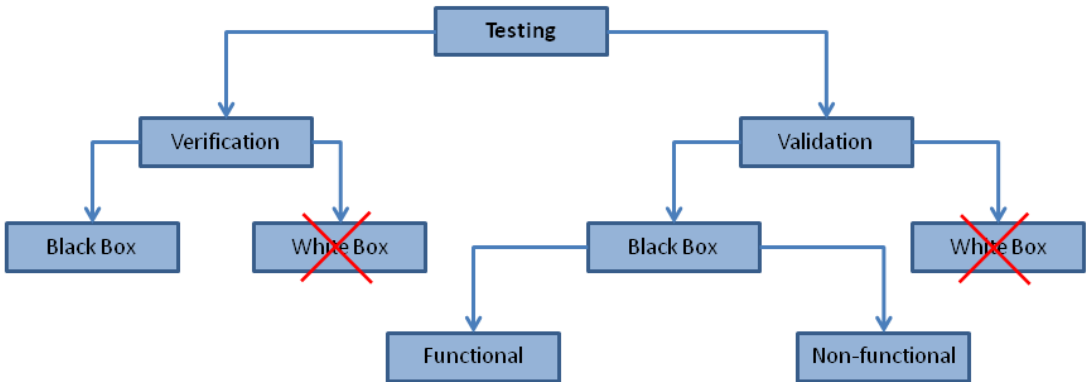


Figure 24 Functional and non-functional testing

The term non-functional testing describes the characteristics of software where the results may vary but the test still counts as passed. Non-functional testing can be performed after completion of the functional tests and provides information about the quality of the product. Non-functional testing means that software attributes are tested which are not influenced by user actions such as performances, security, reliability and so on. The response time for performance testing is an example of a non-functional test. A high quality of the product supports a high customer satisfaction, therefore non-functional testing is very important. Functional testing checks the software against the business requirements. Functions are checked by entering input parameters and evaluated by the outputs. The internal structure is irrelevant. Functional testing checks "what" the software does (WIL 06). All stages of development are considered by functional testing. Normally, functional testing processes 5 steps.

1. *Preparation of test data based on the specifications of functions*
2. *Business requirements are the inputs to functional testing*
3. *Based on functional specifications find out of output of the functions*
4. *The execution of test cases*
5. *Observe the actual and expected outputs(*FNT 12*)*

4.5.1. Functional Testing

Functional testing is part of the entire process of software development that is why functional testing has to be further limited because the software has already been developed and is available on the market. There are several types of tests and the most important ones are listed and briefly explained below.

4.5.1.1. Unit Testing

Unit testing tests functional parts of computer software. The term unit testing is seen as an early test stage where the inner and most detailed components of the software are tested. As the complexity of the functions is limited, they can be completely tested with a relative small amount of test cases. Many sources assume that unit testing is part of black-box testing, but most of the tested combinations are only derived by the source code or by a very detailed technical knowledge of the software. Therefore unit testing belongs to the class of white-box-testing and is executed by the developers.

4.5.1.2. Integration Testing

Integration testing combines interdependent modules in order to test the interaction of more complex systems. Individual modules had to overcome a module test and to work flawlessly. With an increasing number of modules, integration testing is getting too complex for the overall system and that is why the system will be divided in subsystems. Subsystems that passed the integration tests are often combined into one new module. In small projects, integration tests often takes place during programming by the developer. After completion of a module, the interaction with the whole

system is tested through an integration test. For larger projects, the amount of integration tests increases so much that the tests cannot be done manual. Therefore integration tests are often done by test automation. Generally integration testing can be classified as white-box-testing for small projects. Larger projects with automated test sequences belong more often to the class black-box-testing.

4.5.1.3. Functional and system testing

Functional and system testing can be considered together since they are concerned with controlling a product or system against the functional requirements. This type of testing takes place after integration testing and is often passed as the last step of development. The entire system is tested against the requirements (requirement specification). Usually this type of testing uses a test environment that is very similar to the environment of the customer. Furthermore knowledge about the internal structure of the software is not required therefore functional and system testing can be classified to the class of black-box testing. These tests should be executed by independent testers because undesirable behavior of the software can occur by a different understanding of requirements between developers and customers.

4.5.1.4. Acceptance testing

Acceptance testing checks that all requirements of the software are fulfilled. This type of test is the final test before the software is delivered to the customer. Acceptance testing is ranged to the class of black-box testing looking at the overall system. The test does not concentrate on the code but only on the behavior at specified situations. Acceptance testing is by definition roughly the same as system testing with the main difference that it is performed by the customer.

4.5.1.5. Beta testing

Beta testing is more or less an optional variant of testing. During the development phase, a beta version of the software can be carried out to the end user who is testing the software and looking for errors. A beta version is an unfinished version of the expected software where generally all essential functions are implemented but the functions have not been fully tested by anyone. Therefore the program certainly has many errors and is not intended for routine business. In the software industry beta testing has become a common testing method to check stability and accuracy at minimal cost in the recent years. Beta testing belongs to the class of black-box-testing and is mostly used by external testers.

4.5.1.6. Regression testing

Regression testing is a repetition of all or individual test cases to detect effects of changes in previously tested parts of the software. Such changes of test case results regularly occur at changes, extensions or corrections of the software. Regression testing is a procedure which is parallel to all other testing methods. Therefore regression testing belongs to both classes white-box-testing and black-box-testing.

4.5.1.7. Evaluation Functional Testing with regard to this thesis

Unit testing is often seen as a black-box testing method but knowing of the source code is assumed by the tester. That is why this method is omitted since white-box testing is not possible. Integration testing is conducted by the programmers because knowing the source code is beneficial, therefore integration testing is not pursued further. Beta testing is left out as well since there is no beta software. In addition acceptance testing is removed from further consideration since this test method is included in the system testing with the only difference in the role of the tester. Regression testing is not possible at this point, since there is no way to improve errors of the OpenGL driver or to update the driver and to repeat tests. System and functional tests fulfill all requirements to create suitable test cases.

Testing Type	Specification	General Scope	Opacity	Who generally does it?
Unit	Low-Level Design Actual Code Structure	Small unit of code no larger than a class	White Box	Programmer who wrote the code
Integration	Low-Level Design High-Level Design	Multiple classes	White Box Black Box	Programmers who wrote code
Functional	High-Level Design	Whole product	Black Box	Independent tester
System	Requirements Analysis	Whole product in representative environments	Black Box	Independent tester
Acceptance	Requirements Analysis	Whole product in customer's environment	Black Box	Customer
Beta	Ad hoc	Whole product in customer's environment	Black Box	Customer
Regression	Changed Documentation High-Level Design	Any of the above	White Box Black Box	Programmer(s) or independent testers

Table 2 Types of testing

In Table 2 all types of test are listed once again (WIL 06). This list is not complete, since there are also modified types of tests such as smoking testing or sanity testing. Figure 25 shows an updated of the used model.

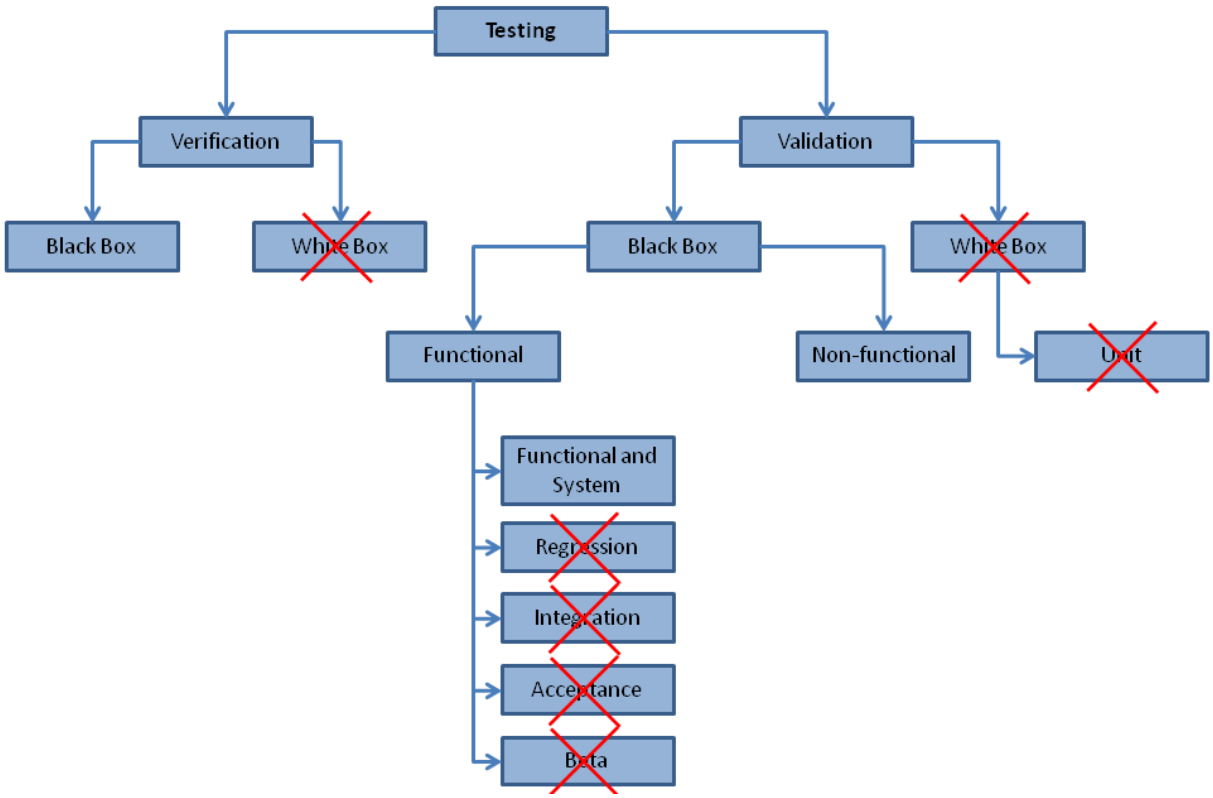


Figure 25 Types of functional testing

4.5.2. Non-functional testing

Non-functional requirements will be tested by non-functional testing methods. A non-functional requirement describes criteria that can evaluate the working of a system. There are a lot of non-functional testing methods like security testing, stress testing, usability testing, performance and quality testing. Because non-functional tests do not have a specific behavior, they can be used interchangeably by many methods for example performance testing could also be a part of reliability testing or stress testing.

Non-functional testing is often seen as the quality testing of a system. Therefore the tester has to know what is important for the customer (AJI 04). As OpenGL is an application programming interface non-functional testing is complex. Finding good criteria is very difficult. For example reliability testing checks that the software is working without breaks and without errors. But testing each function of OpenGL for reliability does not lead to any results because functions are dependent on other functions. Only programs which are created through the combination of individual OpenGL commands are usable and testable for the customer. That is why in case of non-functional testing, OpenGL has to be seen less than software but as a programming language. There are infinitely many different programs that can be written by using a programming language. Furthermore OpenGL is

designed for a lot of different hardware. Thus many test results are dependent on the underlying hardware and the results cannot be easily evaluated in terms of passed, failed or good and bad. As part of this thesis simple test cases were written that process and review a small portion of non-functional testing. The programs are limited to performance testing, quality testing and security testing as shown in Figure 26.

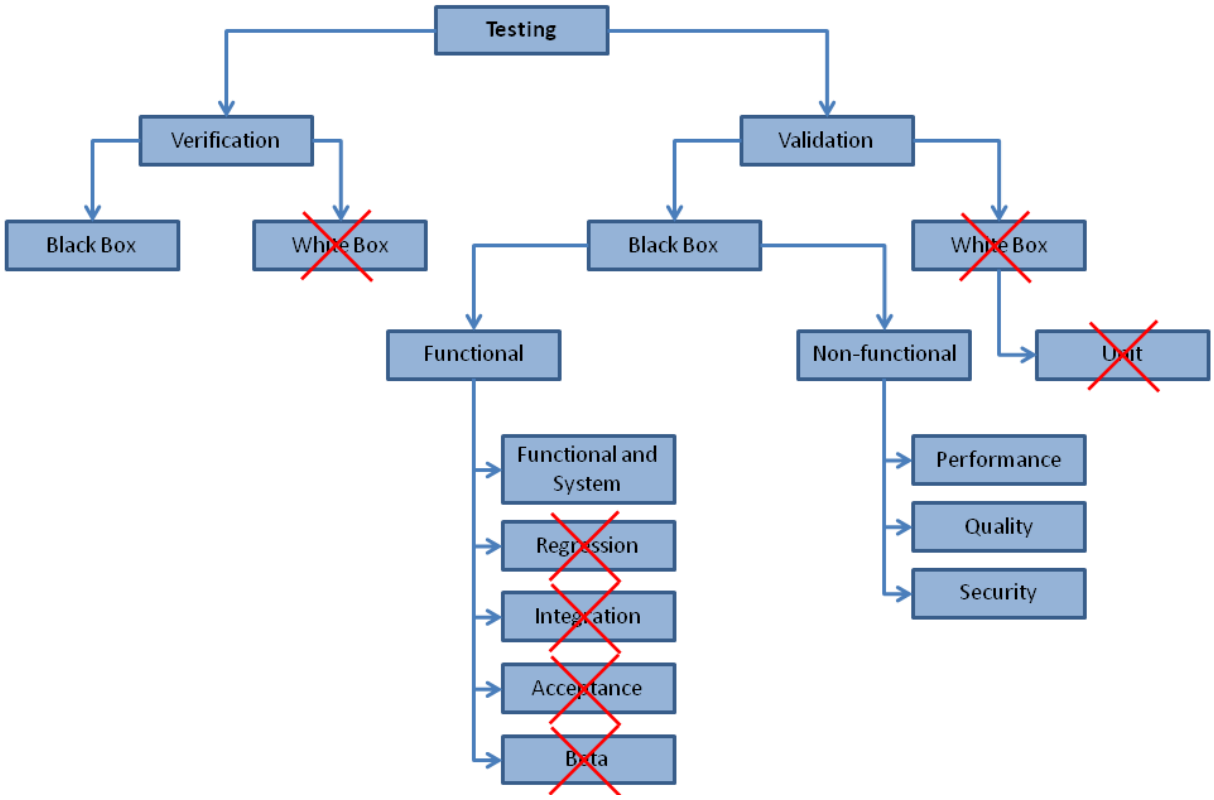


Figure 26 Types of non-functional testing

4.6. Methods of testing

Black-box testing attempts to find errors caused by external commands. Test cases may be divided into different categories such as incorrect or missing functionality, interface errors, performance errors, initialization and termination errors and so on. Finding good test cases is crucial, but there are a few general statements that are intended to help in testing, such as write simple test cases or avoid writing redundant test cases. These two general statements alone do not help to write good test cases. To reduce the amount of test cases and cover as many tests as possible, the following methods are applied: error guessing, exploratory testing, boundary value analysis and equivalence class partitioning (AJI 04).

4.6.1. Error Guessing

During error guessing, the tester creates test cases through the past experience of the tester. Also the experience of other testers can be used like researches from the internet. Since these test cases only base on experience, error guessing can only support other testing strategies. Inexperienced testers can use others experiences to prove that errors from a previous software version are

eliminated by a newer version or the error still exists. Simple errors which are often found with the help of error guessing are for example dividing by zero, null pointers or invalid parameters. There are no explicit rules for error guessing and the test cases depend on the current situation.

4.6.2. Exploratory Testing

Exploratory testing is a creative testing technique where the tester considers how the software should behave in easy and difficult situations and how it can lead to problems. Testers with a good technical knowledge of the software are able to use this technique. The quality of this technique depends on the skills of inventing test cases of the tester. Exploratory Testing is particularly suitable when the requirements or specifications are incomplete or when there is not enough time to apply more extensive testing methods. This method is not really suitable for inexperienced testers, because inexperienced testers need a lot of time and the test cases do not produce the desired results.

4.6.3. Boundary Value Analysis

In most systems errors occur at the turning points of input. Testing the limits of the functions range is important. In the boundary value analysis functions are running with the limits of their validity. These limits include the maximum, minimum, values outside of the range, error values and typical values for the function.

Example: A value between 1 and 100 has to be entered in a function. Checking all 100 numbers is not suitable. Instead the values 1 and 100 are better solutions as the minimum and maximum and 0,2,99 and 101 are good choices to check the behavior at the end of range. A value in the middle of 1 and 100 can also be chosen as a typical value of the function. The limit of boundary value analysis is that only variables of fixed values can be used but it is not suitable for strings or other kinds of inputs.

4.6.4. Equivalence Class Partitioning

Equivalence class partitioning splits the input into partitions of equivalent data. The test cases access these partitions. This guarantees that every partition of the function is covered at least once. An equivalence class partition is a subset of data, forming a larger class. This technique tries to find classes of errors and thereby to reduce the total number of test cases.

Example: A numeric input can accept values between -200 and +200. A possible partitioning should always use an equivalence relation for example "same sign". This would make a partitioning:

- 200 to -1 (negative sign)
0 (no sign)
1 to 200 (positive sign)

Since error classes have to be discovered, a possible test case for this set could be {-8, 0, +8} where -8 is negative sign, 0 is no sign and 8 is positive sign.

4.6.5. Conclusion Methods of Testing

There are many more examples for test strategies like use case testing, fuzz testing, decision table testing, etc. Since testing is a time and cost factor, it is not possible to use all available test strategies. Depending on the specific application, the suitable programming experience, the available time and the available budget, it is important to select the most fitting test strategies. The strategies error guessing, exploratory testing, boundary value analysis, equivalence class partitioning which were explained more detailed, fit best for this thesis to examine the OpenGL specification (Figure 27).

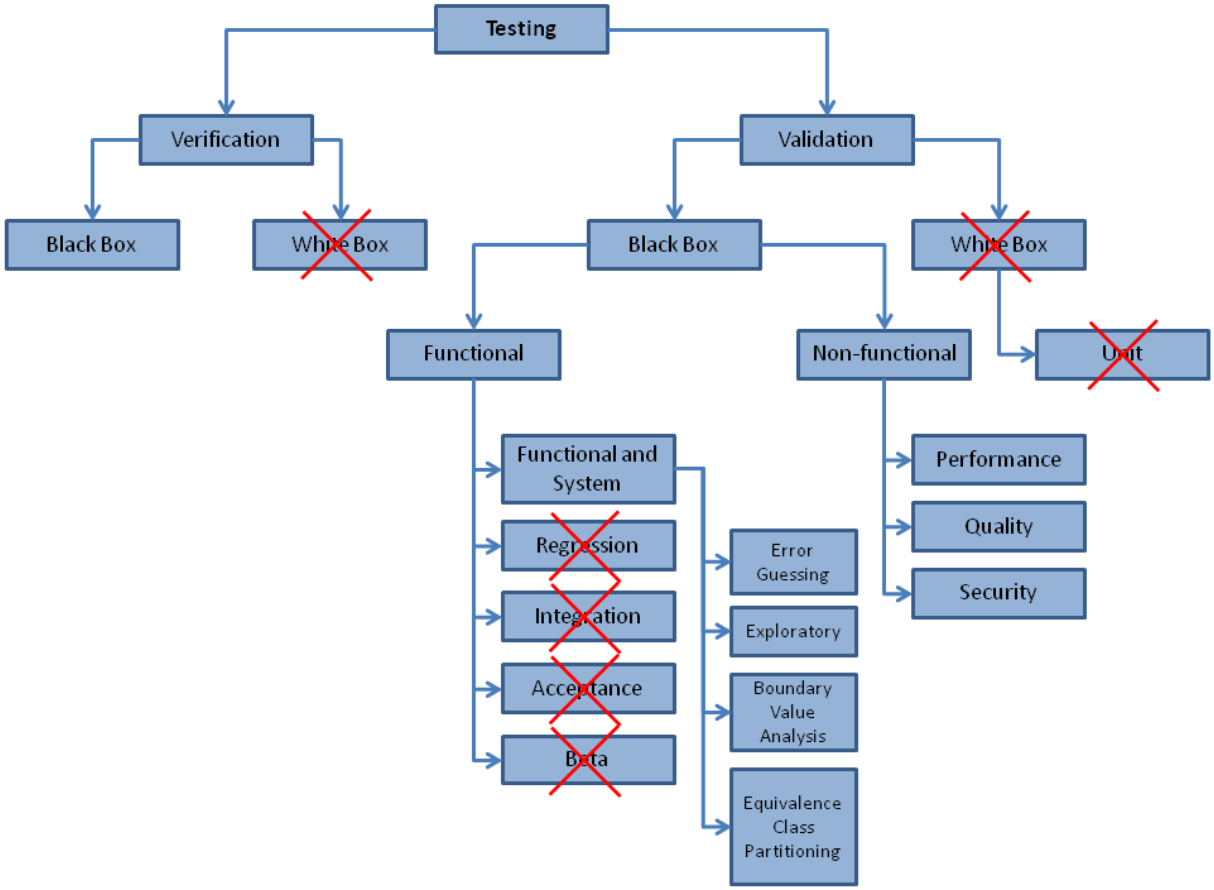


Figure 27 Methods of testing

4.7. Verification

The verification deals with the guarantee that the software meets the specification. An important part is the analysis of the specification because the specification can already contain inaccuracies, errors and linguistic anomalies. Therefore a review of the specification has to be made. A method for a review is the inspection.

The inspection is a test method of the verification, where the tester empathizes with the user's role and examines courses of actions. In this thesis the usability of the specification is tested by the perspective of a developer of a graphics card driver (AJI 04).

Figure 28 shows the point where the inspection can be included into the whole study.

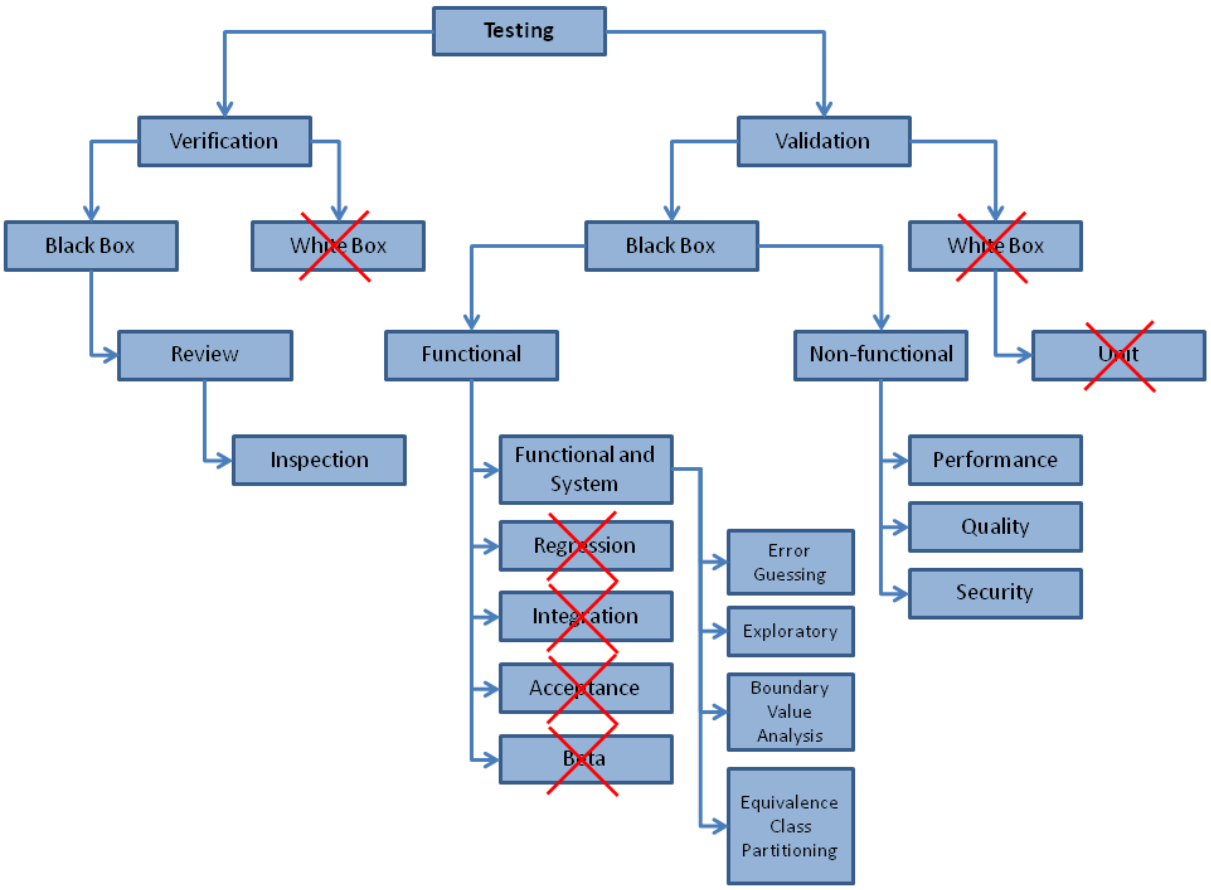


Figure 28 Complete testing strategie

4.8. Presentation of the results

The results of the performed tests have to be evaluated in a way that all OpenGL implementations can be compared with each other. Thereby the actual value is compared with the desired value and a decision has to be made about the test result. This result must be classified into passed and failed. If the test is not passed, an adequate description of the fault must be specified. In case that the test passed, the test is done.

It is important that all tests are well documented and the results are archived in some way. The strategy of test evaluation must be taken into account during the planning phase and before executing the tests. Test cases have to be grouped that the results can be converted into a professionally qualified structure, e.g. in security tests and performance tests or in arithmetic operators and assignment operators. The groupings always depend on the application. The more test cases are written, the more groups should be there.

All tests must be clearly assigned therefore each test requires a unique identifier. In order to improve the readability of the test, all test evaluations need a similar structure. The test results should be in a tabular form. Table 3 shows the principal headline for such a table.

Unique Identifier	Real Result	Expected Result	Overall Result	Test Description	Error Description

Table 3 Header of Output

Important fields are the unique identifier, real result, expected result, overall result, test description and error description. The test description has to be clearly formulated to ensure the repeatability of the test. For black-box testing the input is set and the output is read out, therefore the input should be specified in the test description. The error description should provide information about the nature of the error and where the error has occurred. All information must be stored in an archive. The data can be stored and archived in databases, XML-, JSON-files or similar containers. Often reference tables are used that the tests can be evaluated automatically. In these reference tables the desired values are recorded in order to compare them with the test results and to evaluate whether the test passed or failed.

Test cases should have as few dependencies between test cases as possible. A fixed sequence should be observed during the test run. For example test case B can be made successful only if test case A was successful. In this case test case B does not need to be performed if test case A was not successful. In addition a new context should be generated for each test case because the loaded content within the memory can be invalid. If the previous test case fails and causes an error in the memory, the next test case cannot work fine for sure.

Since tests are performed by different groups, test results should always be visualized in an appropriate way. For example the test results of this thesis are stored in JSON files, because JSON files are easier to handle than a markup language like XML in matters of simple test cases. Afterwards HTML pages are generated from the JSON files which can be output via any web browser.

5. Review of the Specifications

5.1. OpenGL

The analysis of the specification is part of the software verification since the specification reflects the requirements of the software. Often requirements are insufficiently specified and can lead to errors. The requirements of OpenGL can be found in the specifications for each version. Three different versions of OpenGL have been chosen and these versions have to be compared with each other. Only the functions that have been determined by the selected subset are considered for the comparison. The goal of this analysis is not to find all errors and weaknesses but to check that there are weaknesses in the specification. Therefore the three versions OpenGL 2.1, OpenGL 3.1 and OpenGL 3.3 are checked for inaccuracies, errors and other linguistic anomalies.

The specification is not technically written in wide areas. In particular the fault description is very deficient, if any is available. OpenGL is programmed as a state machine hence errors are only set with error flags. The C-command *GLenum glGetError(void)* reads the last set error flag. Possible errors of the specifications are described at the beginning of the document. These errors are generally valid for all subsequent commands. However, there are few error codes. In addition each function can possess other errors if they are described in the function description.

> *Currently, when an error flag is set, results of GL operation are undefined only if OUT_OF_MEMORY has occurred. In other cases, the command generating the error is ignored so that it has no effect on GL state or framebuffer contents. If the generating command returns a value, it returns zero.*
> *[…]*
> *Otherwise, errors are generated only for conditions that are explicitly described in this specification.(* SEG 06*,* SEG 09*,* SEG 10*)*

Furthermore the individual functions do not have a uniform structure. The basic structure of the functional description tries to follow a simple structure:

1. Function Name
2. Description of Operation
3. Possible Input Parameters
4. Error Evaluation

This structure is not obvious for most functions. The points 2-4 are frequently mixed that the correct functioning has to be often interpreted into the description. That is why the programmer has a lot of room for wrong interpretations which can result in errors within the driver.

The descriptions of the functions are also poor. Partly possible input values are not specified and there are no references to tables, lists or the like. For Example, generating a shader needs the C-command *uint glCreateShader(enum shader)*.

> *To create a shader object, use the command*
> *uint CreateShader(enum type);*
> *The shader object is empty when it is created. The type argument specifies the type of shader object to be created. For vertex shaders, type must be VERTEX_SHADER. A non-zero name that can be used to reference the shader object is returned. If an error occurs, zero will be returned. (* SEG 06*,* SEG 09*,* SEG 10*)*

In the description the possible input values are missing for the variable *type*. There is only one known example of a vertex shader. Furthermore the section of the error description already specifies that the return value of this function returns 0 in case of an error. Thus this statement was made twice. But the error which occurs entering an invalid enumeration, is not mentioned again. This was also set in the error description at the beginning of the specification.

> *If a command that requires an enumerated value is passed a symbolic constant that is not one of those specified as allowable for that command, the error INVALID_ENUM is generated.(* SEG 06*,* SEG 09*,* SEG 10*)*

This behavior occurs very frequently in the specification. Some errors that have already been defined at the beginning are called again in the description of the individual functions. Other errors were also defined at the beginning and are not called again in the description. There is not a consistency which type of errors are called a second time.

Furthermore the developer is offered too many freedoms that should not be included in a technical specification. The specification uses terms such as "recommended" as shown in the following excerpt which recommends to set a specific error flag in case of failure by using the OpenGL command *void glDrawArrays (enum mode, int first, Sizeï count).*

> *Specifying first < 0 results in undefined behavior. Generating the error INVALID VALUE is recommended in this case.(* SEG 06*,* SEG 09*,* SEG 10*)*

A technical specification requires a clear indication what behavior of a function has to assume among which conditions. Since OpenGL is designed for many different platforms and the results should be the same everywhere, such inaccurate information may not be included in the specification.

OpenGL also leaves some decisions completely to the developer. For example the description of the function *void glDrawElements (enum mode, sizei count, enum type,void *indices)* describes in case of an error that the developer has to choose a behavior in this situation.

> *It is an error for indices to lie outside the range […] Such indices will cause implementation dependent behavior. (* SEG 06*,* SEG 09*,* SEG 10*)*

Such information is not acceptable for a technical specification.Furthermore some descriptions are written at the beginning of a sub-chapter where the affiliaton is not clear. Chapter "2.15.1 Shader Objects" of the OpenGL 2.1 specification explains at the beginning of the chapter:

> *Commands that accept shader or program object names will generate the error INVALID VALUE if the provided name is not the name of either a shader or program object and INVALID OPERATION if the provided name identifies an object that is not the expected type. (*SEG 06*)*

This description could be valid only for the chapter 2.15.1 or for all subsequent chapters. The shader and program objects are needed as well in the chapters "2.15.2 Program Objects" and "2.15.3 Shader Variables". However there are no descriptions in case of errors. But in chapter "6.1.14 Shader and Program Queries" the same error description was supplied again. At this point the developer can only derive which description is valid for each function. This behavior and the same example are also found in the OpenGL specifications 3.1 and 3.3.

As none of these versions corresponds to the latest OpenGL version, the current version OpenGL 4.3 is additionally taken into account for this thesis. The specification of the current version has been completely redesigned. The new specification tries to implement a better structure and to separate the description and error behavior. Descriptions which were at the beginning of a chapter and valid for several chapters, were added individually to each function. In addition input parameters are listed in tables. The following excerpt deals with the function *uint glCreateShader (enum shader)* which was used above as an example for the older specifications.

To create a shader object, use the command
uint CreateShader(enum type);
The shader object is empty when it is created. The type argument specifies the type of shader object to be created and must be one of the values in table 7.1 indicating

type	Shader Stage
VERTEX_SHADER	Vertex shader
TESS_CONTROL_SHADER	Tessellation control shader
TESS_EVALUATION_SHADER	Tessellation evaluation shader
GEOMETRY_SHADER	Geometry shader
FRAGMENT_SHADER	Fragment shader
COMPUTE_SHADER	Compute shader

Table 7.1: CreateShader type values and the corresponding shader stages

the corresponding shader stage. A non-zero name that can be used to reference the shader object is returned.

Errors

An INVALID_ENUM error is generated and zero is returned if type is not one of the values in table 7.1

(SEG 12)

There is a clear structure in the description recognizable. The error behavior has been highlighted and the possible input parameters were listed in a table. The structure ensures that the developer exactly knows what is required and that there is no room for misinterpretation. Also in the new version of the specification are functions where input parameters or errors are not written in tables or lists but in the middle of the function description.

On the other side some function descriptions have been left out. An example is the function *void glTexImage3D (enum target, int level, int internalformat, Sizeï width, height Sizeï, Sizeï depth, int border, enum format, enum type, void * data)*. The function call still exists in versions 2.1, 3.1 and 3.3

however it is not included in the current version. The function is still part of the specification, but the function call can only be taken from the old specifications, since the new version makes no indication of the required input parameters.

With the development of new graphics cards, OpenGL was also developed further. Thereby new functions have been added to the OpenGL specification, other functions have been removed and functions were also developed further. By structuring the document, the developers can keep track. For example a function should always refer to the same function group, also after the specification is updated. In case of OpenGL there are functions that belong to another group after each update of the specification for example the function *void glEnable (enum target)*. In the OpenGL specification 2.1 this function is described in the group "2.11.3 Normal Transformation", in the specifications 3.1 and 3.3 in the group "2.8.1 Transferring array element" and in the current specification 4.3 in the group "10.3.5 Primitive restart". Since graphics card drivers are not reprogrammed with each update of the OpenGL specification but updated, the developer has problems to keep track with the changes of the OpenGL specification. Errors can occur if the developer does not find certain functions at the usual place.

5.2. GLSL

Another part of the OpenGL specification is the specification for the shader language GLSL. The used GLSL version has to be entered in the header. Thus only the functions of the corresponding version of the shader are available. Since OpenGL 2.1 supports GLSL 1.20 as latest version, the examination of this version is enough.

Basically the structure of the GLSL specification is completely different to the OpenGL specification. While OpenGL provides its own commands and can be implemented with different programming languages, GLSL is a C / C++ similar high-level programming language. The big difference to C / C++ is that GLSL does not support pointers.

Although OpenGL and GLSL belong together, they are very different in programming. The specification for GLSL has a simpler structure. Many examples help to understand the specification. In GLSL is clearly defined which value is returned by each function with corresponding input. However GLSL has a huge disadvantage compared to OpenGL, since there is no error description. The following excerpt shows that the compiler will issue warning and error messages when a program is incorrectly programmed.

> *Compilers, in general, accept programs that are ill-formed, due to the impossibility of detecting all ill-formed programs. Portability is only ensured for well-formed programs, which this specification describes. Compilers are encouraged to detect ill-formed programs and issue diagnostic messages, but are not required to do so for all cases. Compilers are required to return messages regarding lexically, grammatically, or semantically incorrect shaders. (*KES 06*)*

The developer hast to decide how the program should behave when an error occurs. The following excerpt shows that even a division by zero is not an error and the developer has to come up with an own solution.

> *Dividing by zero does not cause an exception but does result in an unspecified value.(*KES 06*)*

Another problem is that shaders can be very difficult debugged. Finding errors is very difficult when they are encountered during the run-time. The specification only applies the statement that the program may not be interrupted under any circumstances.

> *Similarly treatment of conditions such as divide by 0 may lead to an unspecified result, but in no case should such a condition lead to the interruption or termination of processing.(*KES 06*)*

Since almost no error handling is set, finding errors is very difficult. In this case the developer of the GLSL driver has to interpret a lot of information. Therefore the implementations can only be checked, if the functions are programmed against the specification.

6. Tests

6.1. Introduction

After the test review has shown that the specification leaves much room for errors, test cases can be generated. The execution of test cases belongs to the area of validation. The test cases are divided into 7 sections. Thereby the areas are divided into two groups, functional testing and non-functional testing. Table 4 gives an overview what tests are carried out in which area.

Functional Testing	
Piglit	Piglit is an OpenGL test suite, that tests OpenGL against the specifications.
Shader Tests	This test validate shaders against the specification. Shaders will be checked whether they can be compiled and linked. But the shaders will not be proved whether they show the desired results.
Error Tests	A subset of functions of the OpenGL specification is tested, whether the functions work in accordance with the specification by entering wrong values and validating the error codes.
Minimum and Maximum Value Tests	This test checks the minimum and the maximum values from the chapter "State Tables" of the OpenGL specification.
Non-Functional Testing	
Benchmark Tests	An Obj-file is loaded and rotated and the frame rate is measured.
Quality Tests	Multiple Obj-files are loaded simultaneously and the result is compared with a reference image.
Security Tests	This test checks whether the texture memory and the vertex buffer object memory is set to a constant value during initialization and do not contain information of the old memory contents.

Table 4 Functional and non-functional tests

The discussed methods error guessing, exploratory testing, boundary value analysis and equivalence class partitioning of the chapter 4.4 are used to determine the individual tests. These methods interact partly with each other.

6.2. Piglit

First of all the method error guessing is used. Therefore an Internet research was done to integrate the knowledge of other testers. This research shows that there were two open source approaches to test OpenGL against the specification. The two open source solutions are called "Piglit" and "Glean", whereby the tests of Glean are integrated in the Piglit suite. The suite provides tests to improve the quality of open source OpenGL drivers (PIG 13).

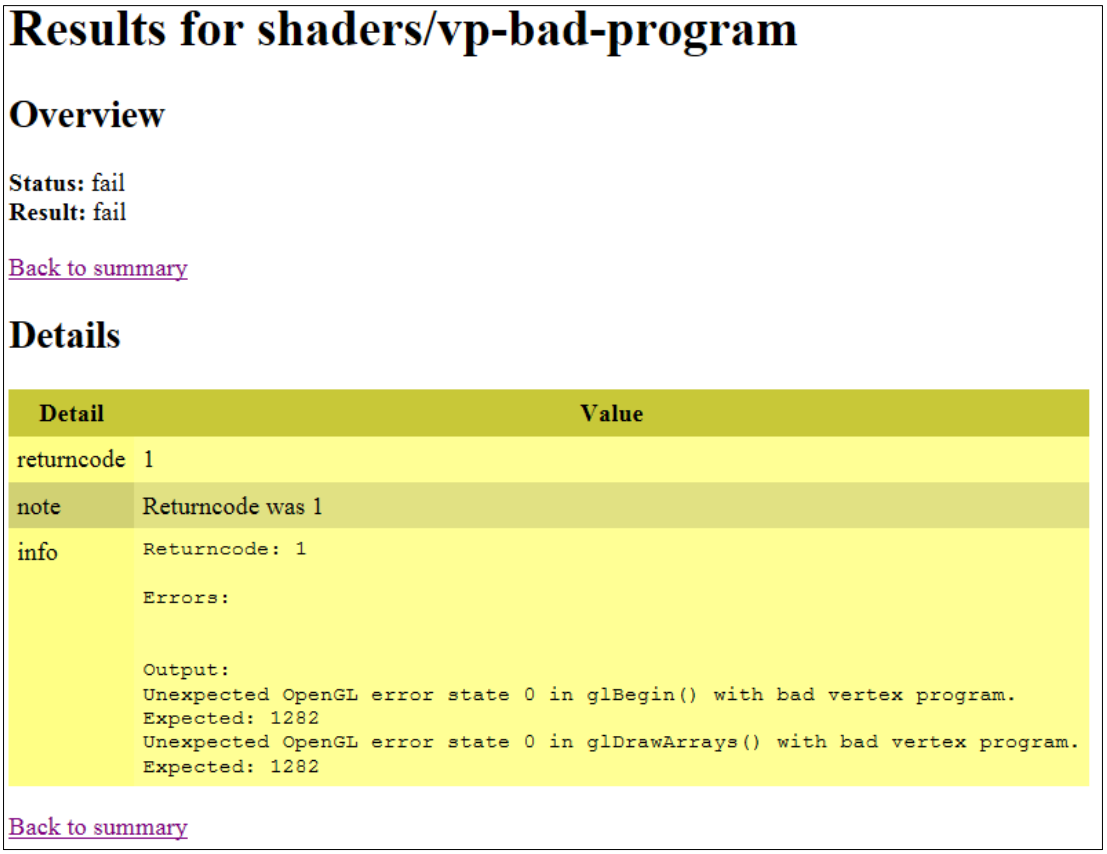
Piglit collects tests provided by many users. As a result many tests are included from various sources. Piglit is a large free collection of automated tests for OpenGL Implementations [piglit]. Running all tests was not possible without producing a system crash because the framework is not running stable on Windows systems hence the amount of tests has to be restricted.

Determining the amount of test cases is difficult because new tests are added every day and there are already much more than 1000 tests. I have finally decided against the Piglit suite since there are some disadvantages. The biggest disadvantage can be found in the evaluation. The basic structure of a test and a test evaluation are already discussed during this thesis. Piglit is not using all needs of a good evaluation. Basically the evaluation of Piglit has some positive points, for example the results are stored in JSON-files and thereby archiving results is possible. HTML documents are also used that is why the test results can be displayed in any browser on any operating system. In addition the tests were grouped and each test has a unique name which is used as a unique identifier. The tests are listed in a table as shown in Figure 29 Piglit structureFigure 29.

	main (info)
All	**31/50**
bugs	**3/3**
fdo10370	pass
fdo14575	pass
fdo9833	pass
shaders	**0/6**
fp-fragment-position	fail
fp-incomplete-tex	fail
fp-kil	fail
fp-lit-mask	fail
trinity-fp1	fail
vp-bad-program	fail
texturing	**3/3**
fdo10370	pass
fdo14575	pass
fdo9833	pass

Figure 29 Piglit structure

The evaluation is displayed graphically whether the test passed or failed. Each test is linked with a detail view. For example the test "vp-bad-program" has failed and is displayed red. Opening the detail view shows the greatest disadvantage of Piglit (Figure 30).The test description is missing. Without a clear description of each test, the test has no use. It is not possible to determine what was tested. Therefore the source code has to be found and analyzed of each test. Since Piglit is using its own framework, this is a costly operation which is not the purpose of a test. Furthermore looking at the source code shows that even modern tests are using old functions that are not available in the core profile of the OpenGL specification anymore (since version 3.0). Because this thesis refers to a modern OpenGL, this behavior is seen as a disadvantage.



Results for shaders/vp-bad-program

Overview

Status: fail
Result: fail

Back to summary

Details

Detail	Value
returncode	1
note	Returncode was 1
info	Returncode: 1 Errors: Output: Unexpected OpenGL error state 0 in glBegin() with bad vertex program. Expected: 1282 Unexpected OpenGL error state 0 in glDrawArrays() with bad vertex program. Expected: 1282

Back to summary

Figure 30 Piglit detail

Piglit provides tests for all versions of OpenGL from version 1.0 to version 4.3. This thesis considers only a subset of all functions of the OpenGL 2.1 specification that is why most tests of Piglit are not usable. The reasons are on the one hand by using old functions that are no longer used in modern OpenGL and on the other hand that functions are used from newer specifications. The benefits of Piglit for this thesis are to provide approaches for test cases. Furthermore the GLSL shaders can be used for the verification of the GLSL specification since they cannot use an unknown Framework and the source code is very short and clear.

6.3. Shader Tests

6.3.1. Implementation

In modern OpenGL programming the essential graphical calculations are executed by shaders. Functions that were fixed programmed in previous versions of the graphics driver can be independently programmed using the shader language. These fixed functions have no more importance in the OpenGL specification. The essential component of the pure OpenGL code prepares the data for the shader. Since shaders have become very important for OpenGL, the behavior of implementations against the specification must be considered more detailed. The GLSL specification only describes the fault-free behavior of the functions. This specification provides many opportunities for undefined behavior or behavior where the implementations can differ. The OpenGL program provides entry points for the shader. These entry points are needed for the shader

therefore the shader cannot be used as a stand-alone application. Shaders can already be implemented in OpenGL code or integrated with external files. Furthermore shaders are using their own compilers which are always supplied by the driver of graphics card manufacturer. For this reason debugging shader programs is difficult. Testing the OpenGL code and the GLSL code together is not done by actual programming tools. In most cases this can be done only by external programs. In practice another way is often performed and illustrated by the following pseudo-code.

```
Void main()
{
Color color1 = red;
If(something wrong)
        color1 =blue;
output  = color1;
}
```

The output is not directly set with a color but with a variable. If an error occurs during the run-time, the value of the variable is changed and the output assigned a different color. The programmer knows that there is an error. This approach has some disadvantages. The programmer has to know how the results of individual calculations have to look like to check the IF-conditions on correct behavior. The programming effort is significantly higher, since each critical function needs an additional IF-condition. The programmer also has to know what color code corresponds to the respective code position. Searching for errors during the run-time is almost impossible. Such errors are randomly discovered through larger software projects. Finding errors during compiling or linking of GLSL shaders is easier. The shader is processed by the external GLSL-compiler and GLSL-linker of the graphics card manufacturer but OpenGL provides commands which load the compiler and linker information into the OpenGL context. The tested GLSL shaders were taken from the Piglit suite. In comparison to the evaluation of Piglit, the test cases get an evaluation with a test description to compensate the biggest disadvantage of the Piglit test suite. The shader tests from the Piglit suite examine whether the implementations keep the conditions of the GLSL specification. Therefore vertex and fragment shaders are created, compiled and linked. Further tests are performed to analyze the syntax. GLSL also supports a number of built-in functions to support the programmer. These functions have to be checked that they provide correct results. While the first two test types are checked by the compiler and linker, the third test type needs to be checked for correct results through a more difficult way.

Getting the correct results needs a similar way as testing during the run-time. By debugging at run-time, the color values are changed if a checked value does not match the expected value. Instead of the changed color value, a compilation error should be generated. An illegal array will be generated in case of Piglit.

> *It is also illegal to index an array with a negative constant expression. […] Undefined behavior results from indexing an array with a non-constant expression that's greater than or equal to the array's size or less than 0.(* KES 06*)*

An array cannot be defined with a negative size. If this happens, the behavior is illegal, as shown in the above quote. There is no definition how the shader behaves in such a case but in reality the compiler of any implementation returns a compiler error. The following pseudo-code illustrates the structure of the GLSL-tests of the Piglit suite:

```
Void main()
{
If(something wrong)
int error1[-1];
        if(something other wrong)
                int error2[-1];
}
```

In case of an error the line with the error is not returned, but the line with the illegal array. The true error is not mentioned in the compiler information, but the error of the illegal array. That is why a clear test description is very important to analysis a test case. With a clear description the error can be deduced without analyzing the whole source code.

6.3.2. Evaluation

For this test several hundred shader files were processed by the compiler and linker. Due to the large number of tests, only on a rough evaluation is done with some significant differences. The shader files have been grouped into several subfolders. Figure 31 shows the basic structure of the analysis.

assignment-operator

back

Display Renderer: Intel(R) HD Graphics Family

Display Vendor: Intel

Display Version: 3.1.0 - Build 8.15.10.2372

Name	compiled	linked	compiler working	linker working	expected result	result
assign-array-allowed.frag.html	pass	pass	pass	pass	pass	pass
assign-array-allowed.vert.html	pass	pass	pass	pass	pass	pass
assign-builtin-array-allowed.vert	pass	pass	pass	pass	pass	pass
modulus-assign-00.frag.html	fail	-	pass	pass	fail	pass

Figure 31 Structure of shader tests

For the traceability of the tests the graphics card and the version of OpenGL are printed on each page. In this case the Intel graphics card is used with OpenGL version 3.1. Below the tests are listed in

a table structure. The test name serves as a unique identifier. Each test contains a detailed view with the description of the test case. If the test fails, there is also an error description. The shader test checks whether the "expected result" matches with the OR-operation of the fields "compiled" and "linked". Test cases can also be programmed that they expect to fail. If the OR-operation is true, “pass” appears in the field “result” otherwise “fail”. The fields "compiler working" and "linker working" check that neither compiler nor linker crashed. If one of the two crashs because of a faulty shader file, the test is evaluated as "fail".

For example the detail view to the test case "modulus-assign-00.frag.html" (Figure 32) shows the description of the test case and the error that occurred. This is the desirable representation of test cases.

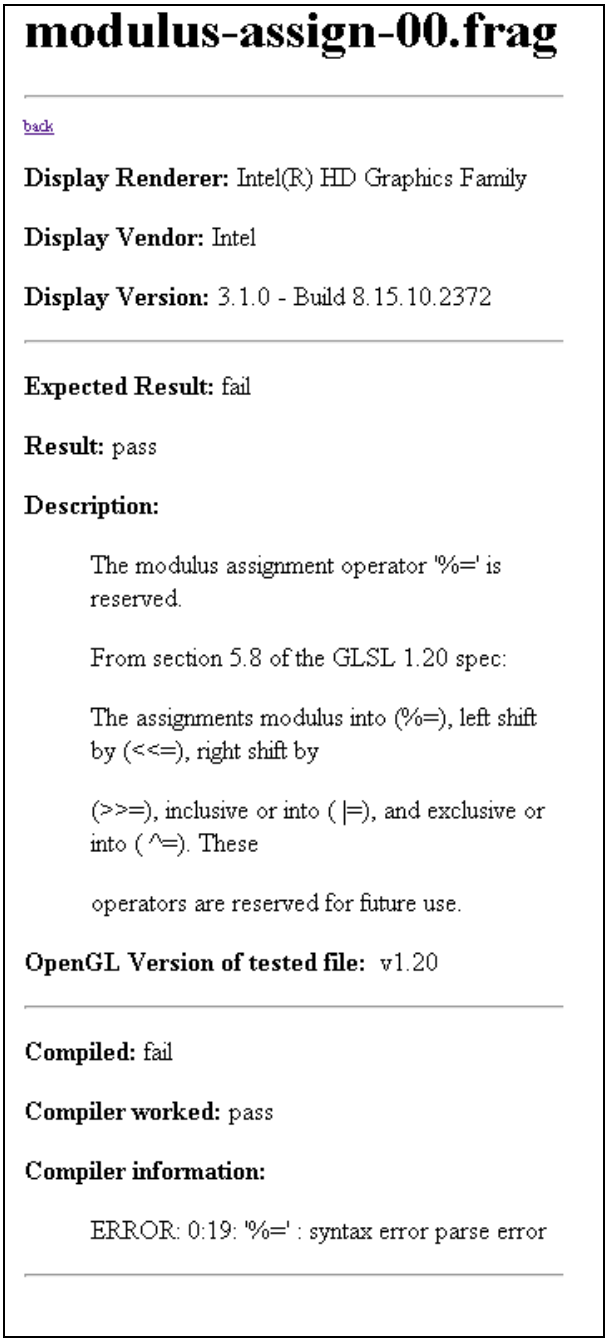
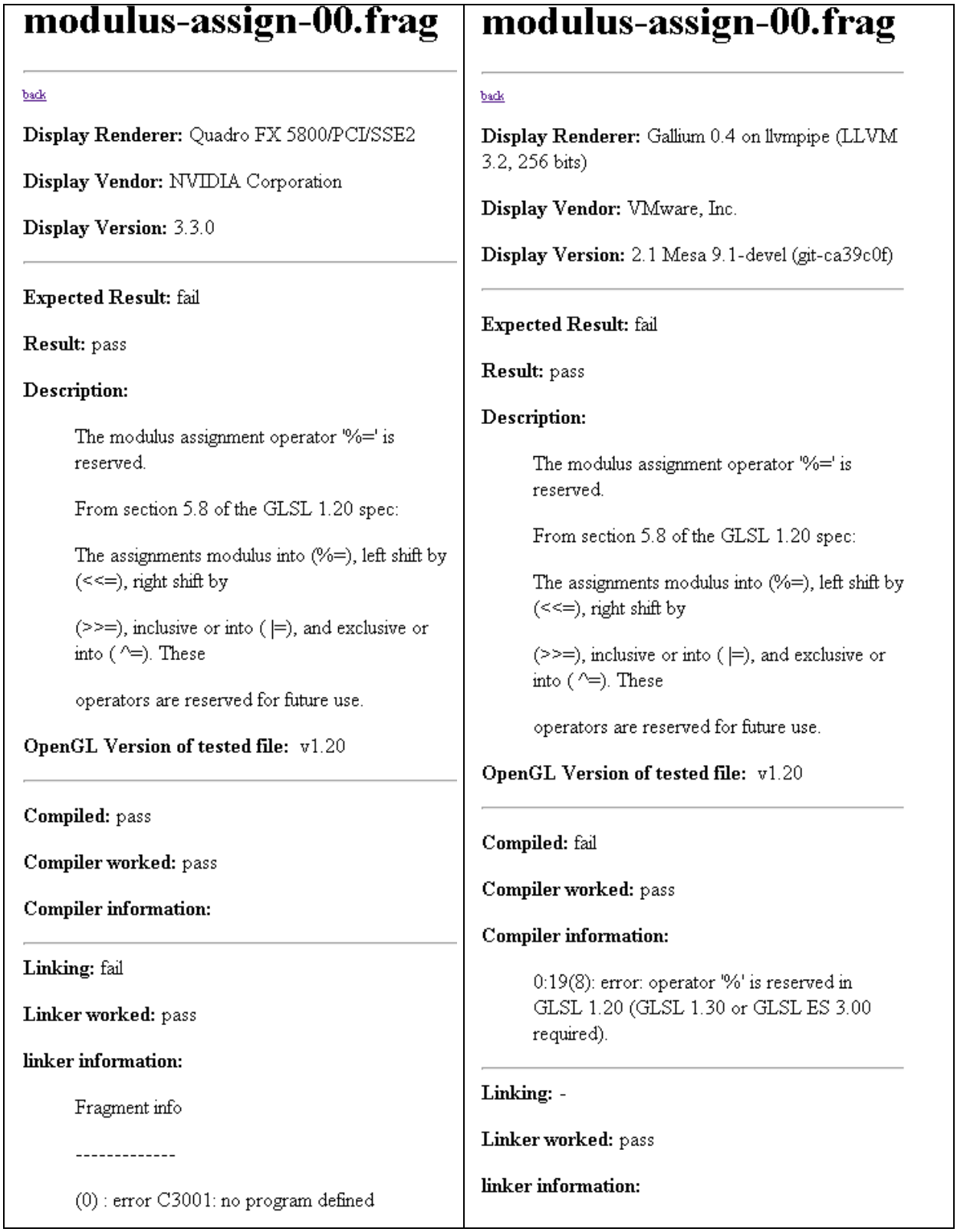


Figure 32 Modulus Operator Intel

The test is passed because the specification determines that certain operators are reserved and may not be used in the GLSL version 1.20. Therefore an error was thrown by the compiler. Depending on the quality of the programming of the shader files, all evaluations should look similar. Since the shader files mostly come from the Piglit suite, the quality of the tests varies in a wide range. Differences are obvious by considering the same test on the other two implementations.



modulus-assign-00.frag

back

Display Renderer: Quadro FX 5800/PCI/SSE2

Display Vendor: NVIDIA Corporation

Display Version: 3.3.0

Expected Result: fail

Result: pass

Description:

The modulus assignment operator '%=' is reserved.

From section 5.8 of the GLSL 1.20 spec:

The assignments modulus into (%=), left shift by (<<=), right shift by

(>>=), inclusive or into (|=), and exclusive or into (^=). These

operators are reserved for future use.

OpenGL Version of tested file: v1.20

Compiled: pass

Compiler worked: pass

Compiler information:

Linking: fail

Linker worked: pass

linker information:

Fragment info

---

(0) : error C3001: no program defined

modulus-assign-00.frag

back

Display Renderer: Gallium 0.4 on llvmpipe (LLVM 3.2, 256 bits)

Display Vendor: VMware, Inc.

Display Version: 2.1 Mesa 9.1-devel (git-ca39c0f)

Expected Result: fail

Result: pass

Description:

The modulus assignment operator '%=' is reserved.

From section 5.8 of the GLSL 1.20 spec:

The assignments modulus into (%=), left shift by (<<=), right shift by

(>>=), inclusive or into (|=), and exclusive or into (^=). These

operators are reserved for future use.

OpenGL Version of tested file: v1.20

Compiled: fail

Compiler worked: pass

Compiler information:

0:19(8): error: operator '%' is reserved in GLSL 1.20 (GLSL 1.30 or GLSL ES 3.00 required).

Linking: -

Linker worked: pass

linker information:

Figure 33 Modulus operator NVIDIA & Mesa3D

Figure 33 shows the evaluation of the NVIDIA graphics card and the evaluation for the Mesa3D software renderer. Mesa3D behaves similar to the Intel graphics card. The only difference is in the wording of the compiler information. The NVIDIA graphics card deviates from the specification and does not reserve the operator for future versions. That means NVIDIA has already been assigned a task to the operator without knowing whether this operator should get this task in a future version of OpenGL. There is another aberration at this place. The test case expects that the test fails. Although the NVIDIA graphics card may compile the shader file, the allover result of test passed. ´While Intel and Mesa3D are aborted by compilation, the shader file was not trying to link. NVIDIA passed the compilation and was trying to link the shader file. The linking has failed that the overall test failed and the result of the OR-operator corresponds to the expected result. The error code which is thrown by NVIDIA means that there is no main function in the shader file. A shader without a main function cannot work because there is no entry point for the shader. As a conclusion, the test case was insufficient programmed.

An example of the built-in functions is the function *genType abs (genType x)*. *Abs()* computes the absolute value of a number. Since the amount of possible input values is too much, equivalence class partitioning has been operated. The areas are divided into negative numbers, positive numbers and the third area is zero. The test description in Figure 34 shows that the values 0.75 and 1.5 are used for the positive range and that -0.75 and -1.5 are used for the negative range. Furthermore the figure shows that the test case passed in case of the Intel graphics card.

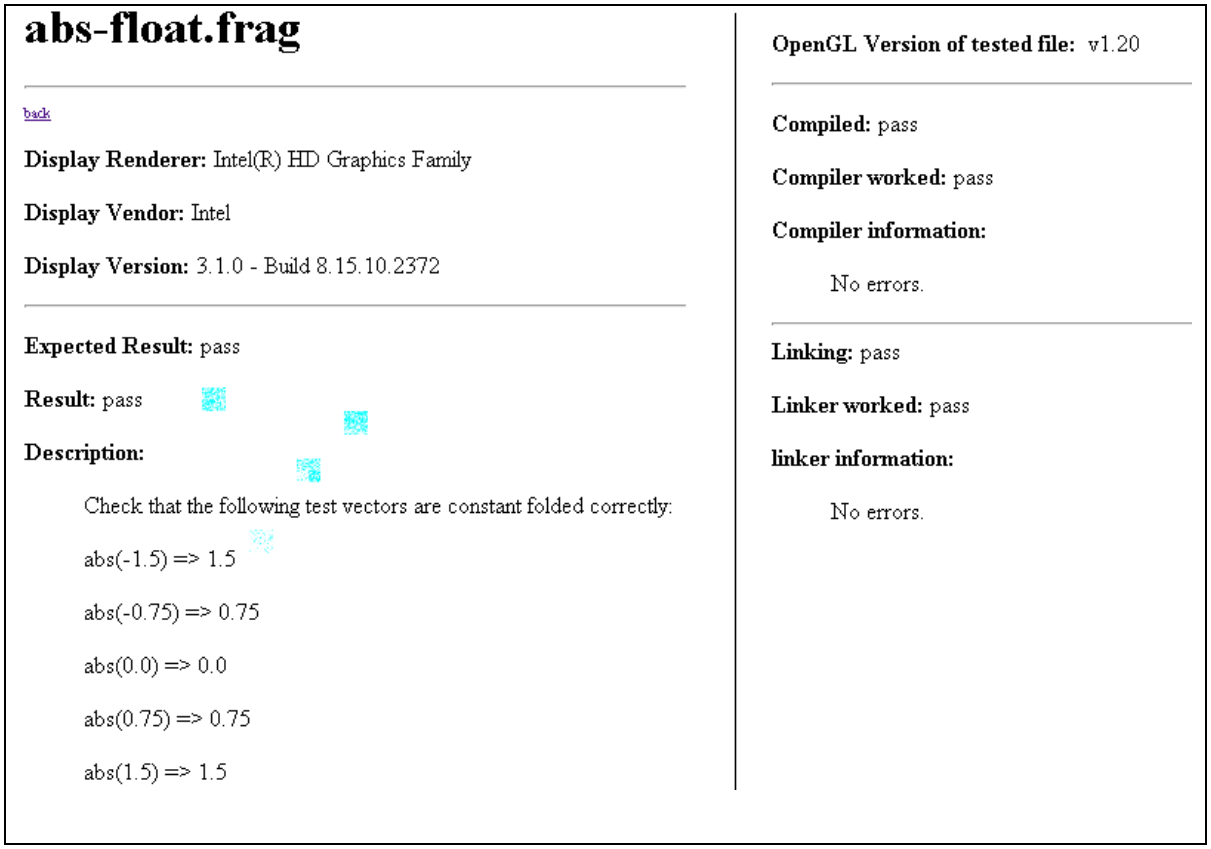
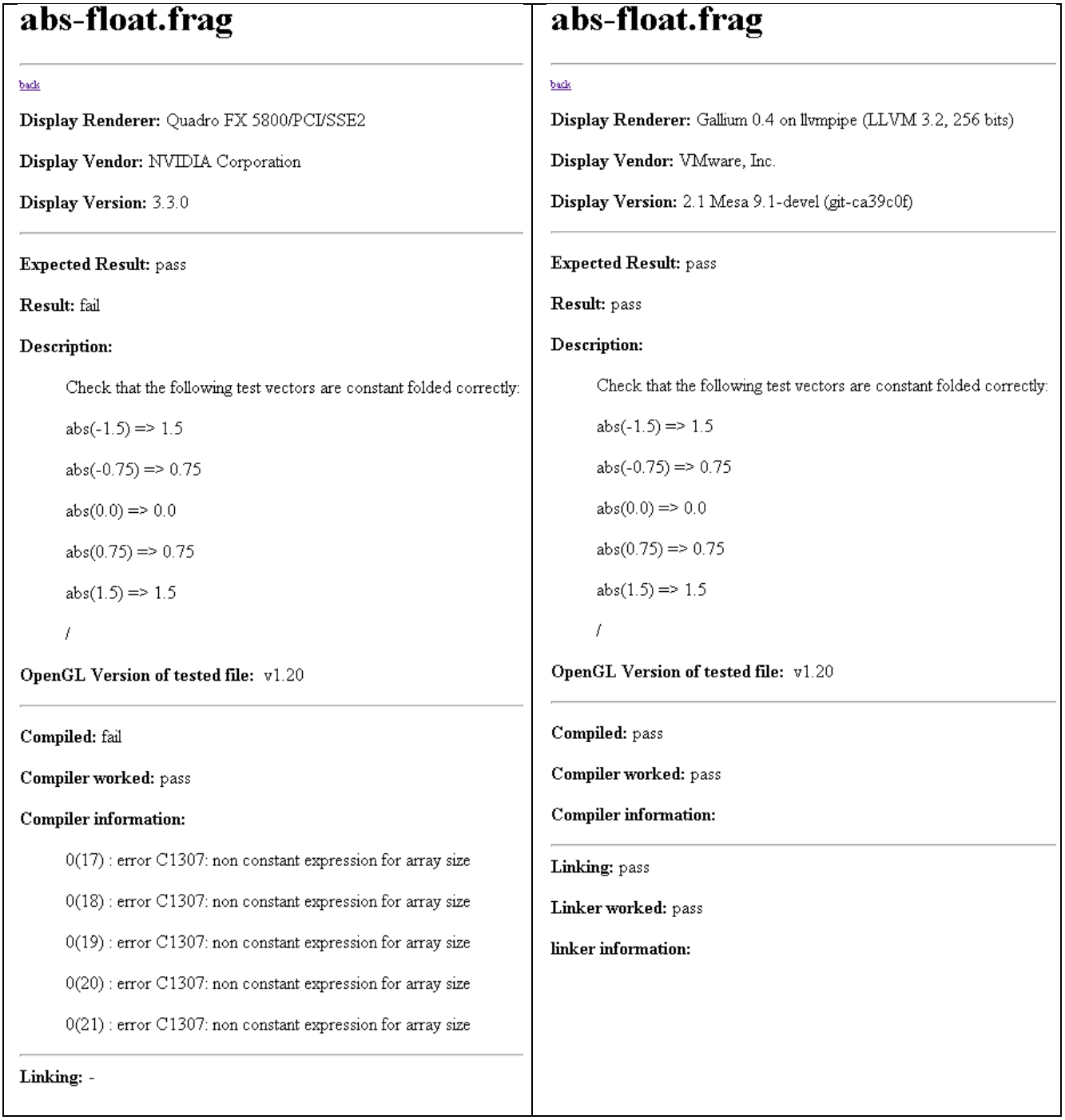


Figure 34 Built-in function Intel

The result of the test has to be known to decide whether the test passed or failed. The known value has to be compared with real result of the test. For example the known value of the absolute value 1.5 is 1.5. If the real result is also 1.5 the test passed otherwise the test failed. Figure 35 shows the results of the NVIDIA graphics card and the Mesa3D software renderer.



abs-float.frag

back

Display Renderer: Quadro FX 5800/PCI/SSE2

Display Vendor: NVIDIA Corporation

Display Version: 3.3.0

Expected Result: pass

Result: fail

Description:

Check that the following test vectors are constant folded correctly:

abs(-1.5) => 1.5

abs(-0.75) => 0.75

abs(0.0) => 0.0

abs(0.75) => 0.75

abs(1.5) => 1.5

/

OpenGL Version of tested file: v1.20

Compiled: fail

Compiler worked: pass

Compiler information:

0(17) : error C1307: non constant expression for array size

0(18) : error C1307: non constant expression for array size

0(19) : error C1307: non constant expression for array size

0(20) : error C1307: non constant expression for array size

0(21) : error C1307: non constant expression for array size

Linking: -

abs-float.frag

back

Display Renderer: Gallium 0.4 on llvmpipe (LLVM 3.2, 256 bits)

Display Vendor: VMware, Inc.

Display Version: 2.1 Mesa 9.1-devel (git-ca39c0f)

Expected Result: pass

Result: pass

Description:

Check that the following test vectors are constant folded correctly:

abs(-1.5) => 1.5

abs(-0.75) => 0.75

abs(0.0) => 0.0

abs(0.75) => 0.75

abs(1.5) => 1.5

/

OpenGL Version of tested file: v1.20

Compiled: pass

Compiler worked: pass

Compiler information:

Linking: pass

Linker worked: pass

linker information:

Figure 35 Built-in function NVIDIA & Mesa3D

Mesa3D has passed as well but NVIDIA has failed the test. The compiler information does not show the real error but the information shows that an array was created with the wrong size. This corresponds to the above-mentioned approach of Piglit tests (chapter 6.3.1). 5 values were tested and 5 error messages appeared that means all 5 values have failed the test. Since the built-in functions are standard functions of GLSL, these functions should work in every implementation or fail

only sporadically. Looking at the source code (quote …) for this function shows that there exists a certain dependence of various functions. Additionally to the function *abs()*,the function *float distance (genType p0, genType p1)* is used and also an operator for condition jumps.

```
float[distance(abs(-1.5), 1.5) <= 1.5e-05 ? 1 : -1] array0;
float[distance(abs(-0.75), 0.75) <= 7.4999998e-06 ? 1 : -1] array1;
float[distance(abs(0.0), 0.0) <= 0.0 ? 1 : -1] array2;
float[distance(abs(0.75), 0.75) <= 7.4999998e-06 ? 1 : -1] array3;
float[distance(abs(1.5), 1.5) <= 1.5e-05 ? 1 : -1] array4;
```

There are many possibilities why the test crashed. Maybe one of the two functions is not working or the NVIDIA card does not support the operator for conditional jumps. Therefore tests need to have fewer dependencies. This example shows on the one hand that the quality of the Piglit tests is not very good and on the other hand that debugging of shaders is not very easy.

Several other differences of the implementations can be found on the provided DVD. The shown tests are only a short extract of all test cases but they already prove that all examined implementations have deviations from the official GLSL 1.20 specification. All test results can be found on the provided DVD.

6.4. Error Tests

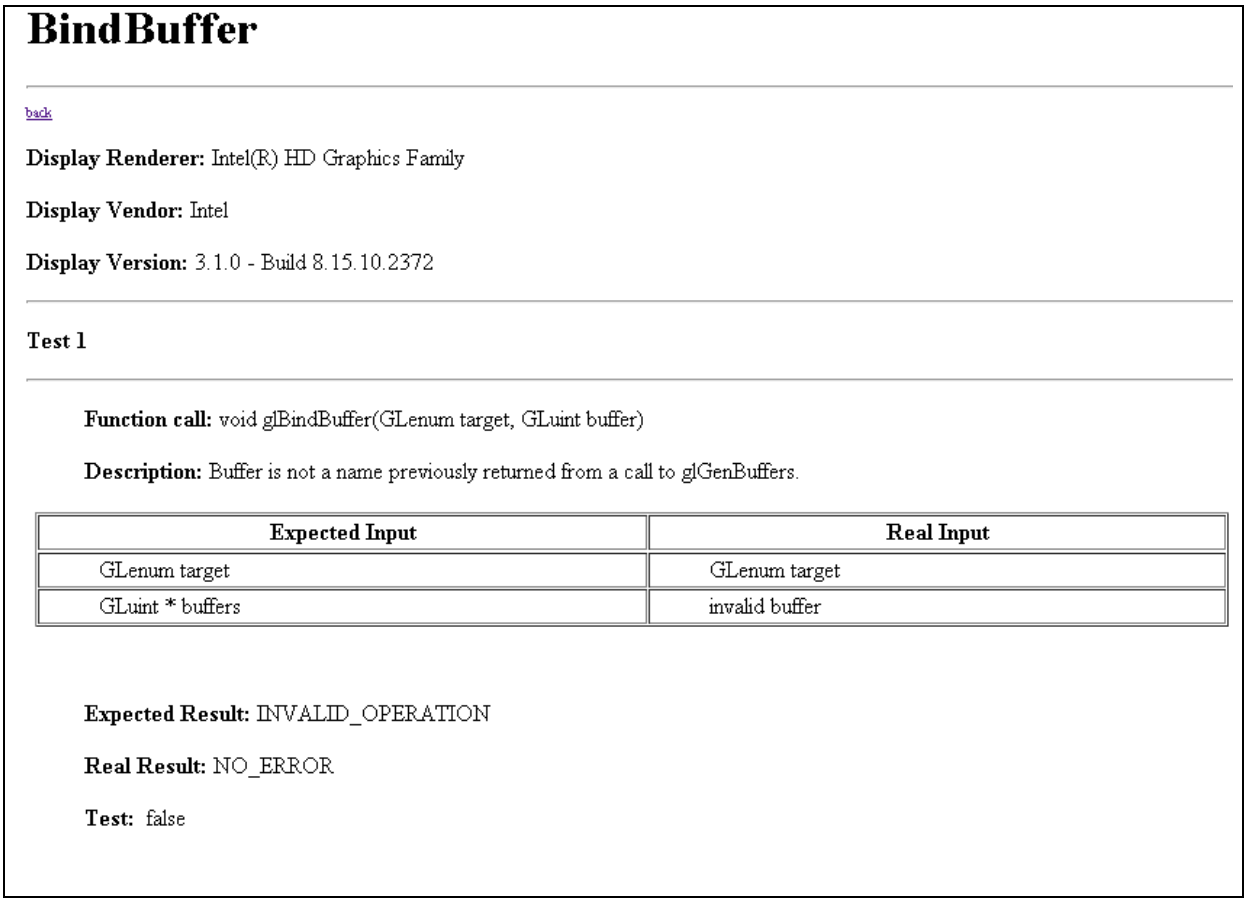
6.4.1. Examination

The review of the OpenGL specification has shown that the treatment of the functions in case of an error is inaccurate. For this reason the functions need to examine with illegal input values to produce errors on purpose. The methods exploratory testing and boundary value analysis were used to create test cases. Each function has different requirements and needs its own test scenario. This test can only examine functions which have input parameters. All functions with input parameters are used from the subset. Since the specification changes with every new version as well as error descriptions are changed in each specification, the implementations always have to be tested with their own specification. In this case writing a reference table can be useful. The test cases only need to be written once and the implementations use the reference table to compare with the specification.

All possibilities cannot be tested for the input parameters of the functions. Therefore a limited amount of test cases has to be created. The values for the input parameters differ from test case to test case. For example the values for enumerations are one valid and one invalid value. The same istrue for objects and pointers. Numeric values are examined at their limits and with a valid value. Because functions can also have multiple input parameters, various combinations are considered depending on the application. 53 functions are tested with 198 test cases.

6.4.2. Evaluation

The structure of the evaluation of the tests contains also the necessary information like the unique identifier, the test description, the expected result and the actual result. In addition information of the implementation is included again. Figure 36 shows the structure of a test case for the function *void BindBuffer(enum target, uint buffer)* for the Intel graphics card.



BindBuffer

back

Display Renderer: Intel(R) HD Graphics Family

Display Vendor: Intel

Display Version: 3.1.0 - Build 8.15.10.2372

Test 1

Function call: void glBindBuffer(GLenum target, GLuint buffer)

Description: Buffer is not a name previously returned from a call to glGenBuffers.

Expected Input	Real Input
GLenum target	GLenum target
GLuint * buffers	invalid buffer

Expected Result: INVALID_OPERATION

Real Result: NO_ERROR

Test: false

Figure 36 BindBuffer Intel

This test case shows that an invalid buffer was passed to the function. According to the OpenGL specification 3.1, the error INVALID_OPERATION should be returned but the real result is NO_ERROR. Therefore the test was rated as failed. Considering the same test case for NVIDIA and Mesa3D the results are similar what is shown in Figure 37. The OpenGL specification 2.1 does not define this error for *glBindBuffer()*. The reference table returns NOT_DEFINED for Mesa3D. The graphics card vendor can decide if the implementation returns an error. A technical specification has to define such cases otherwise OpenGL cannot work on different hardware. With the expected result NOT_DEFINED the programmer has to decide if the test case passed or failed. For this thesis all tests with the expected result NOT_DEFINED are failing the test because the inaccuracy of the OpenGL specifications is counted as an error, although the graphics card vendors is not responsible for this error. NVIDIA does not pass the test either since NO_ERROR is returned although the OpenGL specification 3.3 specifies that the error INVALID_OPERATION has to be returned.

All other tests in this category have the same structure. The results can be found in the appendices on the DVD.

BindBuffer

back

Display Renderer: Quadro FX 5800/PCI/SSE2

Display Vendor: NVIDIA Corporation

Display Version: 3.3.0

Test 1

Function call: void glBindBuffer(GLenum target, GLuint buffer)

Description: Buffer is not a name previously returned from a call to glGenBuffers.

Expected Input	Real Input
GLenum target	GLenum target
GLuint * buffers	invalid buffer

Expected Result: INVALID_OPERATION

Real Result: NO_ERROR

Test: false

BindBuffer

back

Display Renderer: Gallium 0.4 on llvmpipe (LLVM 3.2, 256 bits)

Display Vendor: VMware, Inc.

Display Version: 2.1 Mesa 9.1-devel (git-ca39c0f)

Test 1

Function call: void glBindBuffer(GLenum target, GLuint buffer)

Description: Buffer is not a name previously returned from a call to glGenBuffers.

Expected Input	Real Input
GLenum target	GLenum target
GLuint * buffers	invalid buffer

Expected Result: INVALID_OPERATION

Real Result: NO_ERROR

Test: false

Figure 37 BindBuffer NVIDIA & Mesa3D

6.5. Minimum and maximum value tests

6.5.1. Examination

Since OpenGL supports shaders, there are a lot of details to be known for developers to create software of the implementation independently. These are details like the number of shaders that can be used or the count of variables that can be passed to the shader as well as the textures that need to be supported by OpenGL simultaneously. Trying to standardize OpenGL on all implementations, minimum and maximum limits exist for a variety of state variables. The implementations have to be checked whether the minimum and maximum limits comply with the requirements of the specification. With each new version of OpenGL the graphics cards have also improved, therefore the requirements of the minimum and maximum values of the state variables are also getting higher. Additionally new state variables are added and old state variables are omitted. For this reason each implementation has to be checked against its corresponding specification. In this test all the state variables are tested, which have known minimum or maximum value within in specifications. Other state variables are not considered because they have no comparative value.

By using the functions *void glGet*(enum value, * data)*, OpenGL can check these values. Depending on the return value, the get() function can query integer, float and double values. The analysis also follows the discussed structure. Compared with the other tests no detailed analysis is needed at this point. The main fields are the unique identifier, the function description, the expected result, the real result, the allover result and the information about the used hardware.

6.5.2. Evaluation

Considering the results, there are a few specialties. Mesa3D is tested against the OpenGL 2.1 specification and passes all tests. The test results are better than the required values of the specification. Although the results of the different specifications cannot be compared with each other, there is the fact that only Mesa3D has passed without errors. All tested state variables for the specifications OpenGL 2.1, OpenGL 3.1 and OpenGL 3.3 can be found in Appendix StateVariables.

Intel passes 32 of the 39 tested state variables. The test cases "GL_MAX_PROGRAM_TEXEL_OFFSET" and "GL_MIN_PROGRAM_TEXEL_OFFSET" fail because of returning the error code INVALID_ENUM. This error means that OpenGL does not know these state variables. The implementation does not conform to the OpenGL 3.1 specification.

Furthermore the test cases "GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS" and "GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS" return the value 1, although the values have to be in the range of 45000 to 55000. Figure 38 shows the section of the OpenGL specification for this specific case.

Table 6.40. Implementation Dependent Values (cont.)
(1) The minimum value for each stage is MAX_stage_UNIFORM_BLOCKS × MAX_stage_UNIFORM_BLOCK_SIZE + MAX_stage_UNIFORM_COMPONENTS

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VERTEX_UNIFORM_COMPONENTS	Z^+	GetIntegerv	1	Number of words for vertex shader uniform variables in default uniform block	2.11.4
MAX_FRAGMENT_UNIFORM_COMPONENTS	Z^+	GetIntegerv	1	Number of words for fragment shader uniform variables in default uniform block	2.11.4
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	Z^+	GetIntegerv	1	Number of words for vertex shader uniform variables in all uniform blocks (including default)	2.11.4
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS	Z^+	GetIntegerv	1	Number of words for fragment shader uniform variables in all uniform blocks (including default)	2.11.4

Figure 38 Section of OpenGL 3.1

The "minimum value" fields of both state variables write 1 but the values refer to the footnote of this excerpt The footnote describes the calculation of the two values. The Intel graphics card is taking the wrong values at this point. The programming of complex shaders is limited, because this card supports far too few vertex- and fragment-uniform components. The values of the remaining state variables which have not passed the test, are too low. These errors can have two causes on the one hand there is a programming error of the driver or on the other hand the hardware does not provide the necessary support.

In comparison only one test case failed of the NVIDIA implementation. The value of this state variable is to low. The same two possibilities apply as for the Intel graphics card because an inaccuracy cannot be detected in the specification.

6.6. Benchmarks

6.6.1. Execution

A benchmark test is the comparison of results with a fixed reference value. In this test the performance of individual implementations is compared with each other. This test does not return pass or fail because an implementation can only fail this test, if the test cannot be performed.

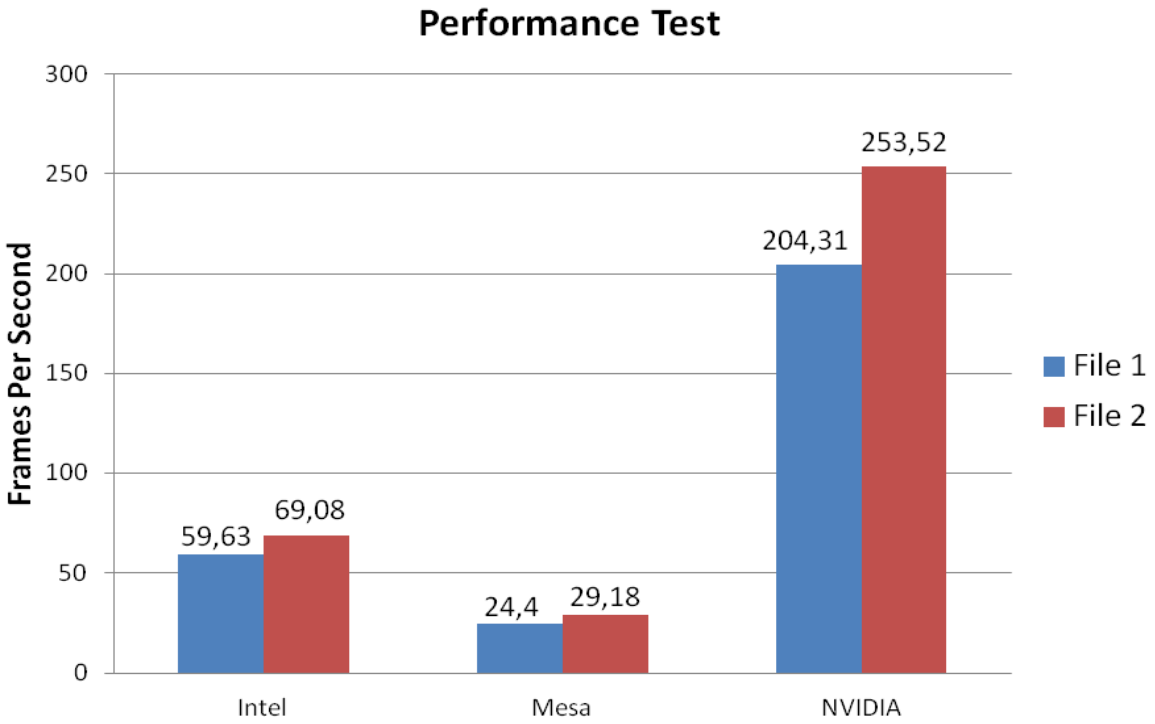
The implementations are divided into a high-performance graphics card, an onboard graphics card and a software renderer. The results of the high-performance graphics card should be significantly higher than the results of the other two implementations.

A benchmark test can prove such a hypothesis. On the Internet many benchmark tests are available, but mostly there is no information about how and what is tested. Furthermore the source code of external benchmark tests is unknown. This work considers only a subset of functions of OpenGL 2.1 and only these functions are used for the benchmark test.

Therefore a small benchmark test was created for this thesis and the source code is located on the provided DVD. The benchmark reads an OBJ-file and loads the information into memory. Afterwards the model is rotated 360°. Meanwhile the average frames per second are measured for the whole rotation. An evaluation of the performance can be done by using different OBJ files with different numbers of vertices.

6.6.2. Evaluation

For the test two OBJ files have been loaded with every implementation. Table 5 shows a summary of the amount of vertices, the resulting triangles and the average frames per second rate.



	File 1	File 2
Vertices	77796	145536
Triangles	63845	121912

Table 5 Benchmark results

The results reflect the hypothesis. The powerful NVIDIA graphics card has a significant performance advantage against the two competitors. The software renderer is significantly weaker than the Intel onboard graphics card. Since Mesa3D is a software renderer and uses the CPU to calculate the result, the result can be improved by a more powerful CPU. Test file 1 has over 14,000 vertices more than test file 2. By raising the vertices, the performance

deteriorates because the graphics card is charged more. More information about the behavior of the performance is not provided at this point, as this is not subject of this thesis. The performance test shows the difference between the capabilities of the implementations.

6.7. Quality Test

6.7.1. Execution

Just as the performance test, the quality test compares the results of the implementations. The OpenGL specification was written that programs run on any platform and deliver the same results but until now the tests have shown that the implementations differ in many aspects from the specification. For this reason this test examine whether the results of an OpenGL program have optical differences. A simple program was created, which loads any number of OBJ files simultaneously in the frame buffer. Afterwards the frame buffer is read out and stored in a bitmap. Differential images are created out of these pictures to compare the implementations.

6.7.2. Evaluation

For the graphical analysis two OBJ files have been loaded into the frame buffer. Figure 39 to Figure 41 show the content of the frame buffer for the software renderer Mesa3D and the graphics cards Intel and NVIDIA.

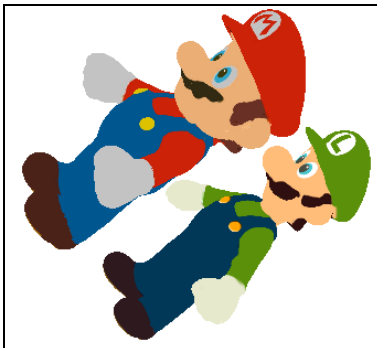


Figure 39 Quality test Mesa3D

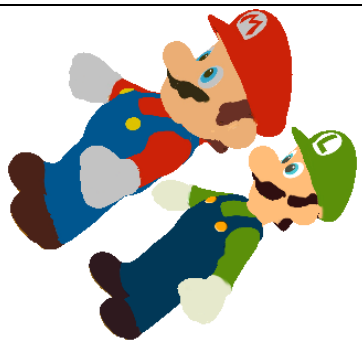


Figure 40 Quality test Intel

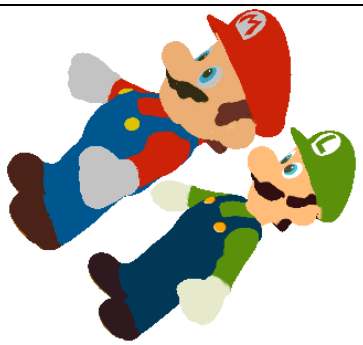


Figure 41 Quality test NVIDIA

At a first glance all three images appear to be identical. Since no reference image exists, the differential images are generated always between two pictures. Figure 42, Figure 43 and Figure 44 show the differential images.

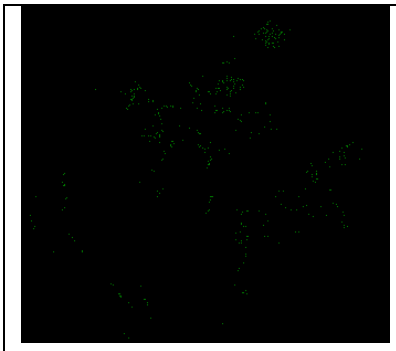


Figure 42 Diff image NVIDIA-Intel

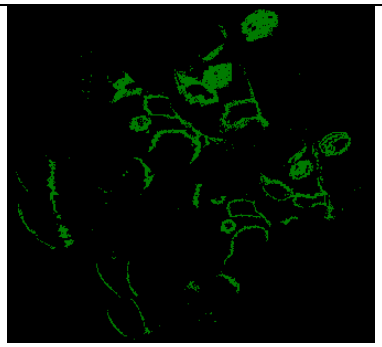


Figure 43 Diff image Intel-Mesa3D

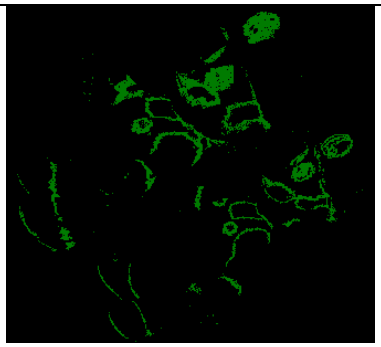


Figure 44 Diff image NVIDIA-Mesa3D

If two images are identical, the differential image would be completely black. Differences between two images are shown with green dots. All differential images show significant differences. The white background of the original images must not be considered for the analysis because there is nothing drawn at these positions and that is why all pixels have to be determined that are not white. This amount of pixels has to put in ratio with the amount of green pixels of the differential images. The percentage is calculated to evaluate the differences. Histograms are generated to determine the number of pixels.

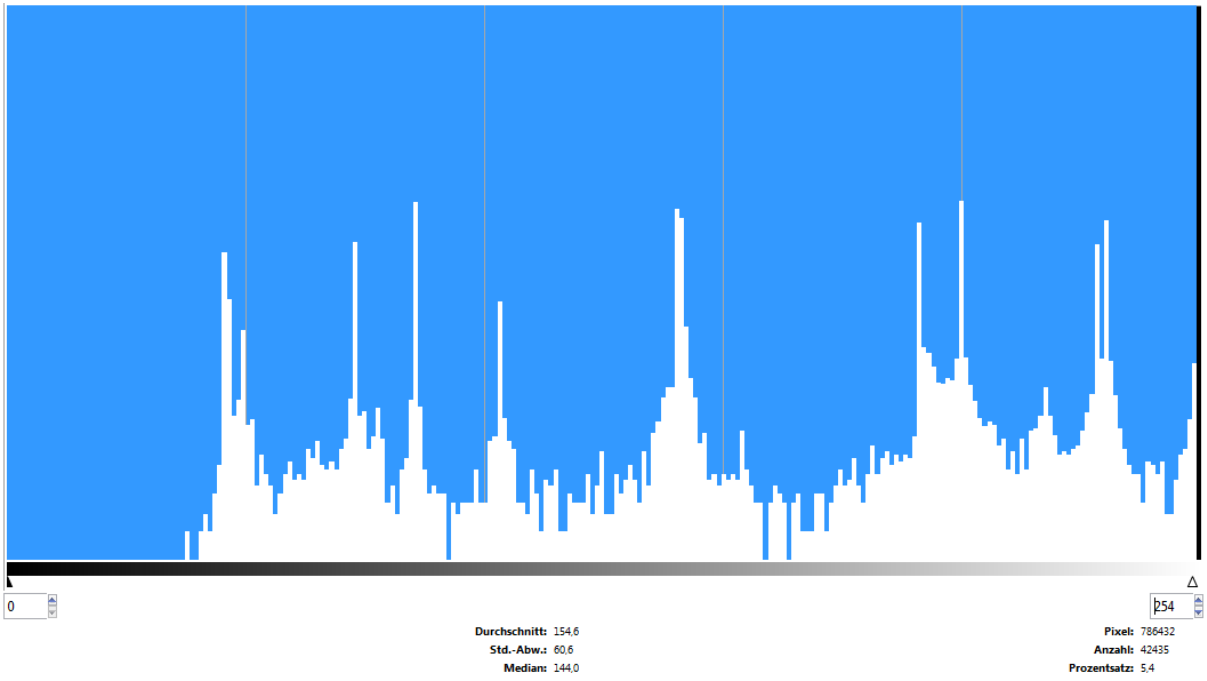


Figure 45 Histogram Intel

Figure 45 shows the histogram of the Intel graphics card. The color range was split into 256 values. The amount of pixels in the range from 0 to 254 is considered, since the value 255 corresponds to white, which is not relevant as mentioned before. The histogram of the differential image between the implementations of Intel and Mesa3D is shown in Figure 46 Histogram Differential Intel-Mesa3D.

The middle portion corresponding to the color green is of interest. As the differential image refers to two pictures, the percentage deviation is calculated to both original pictures.

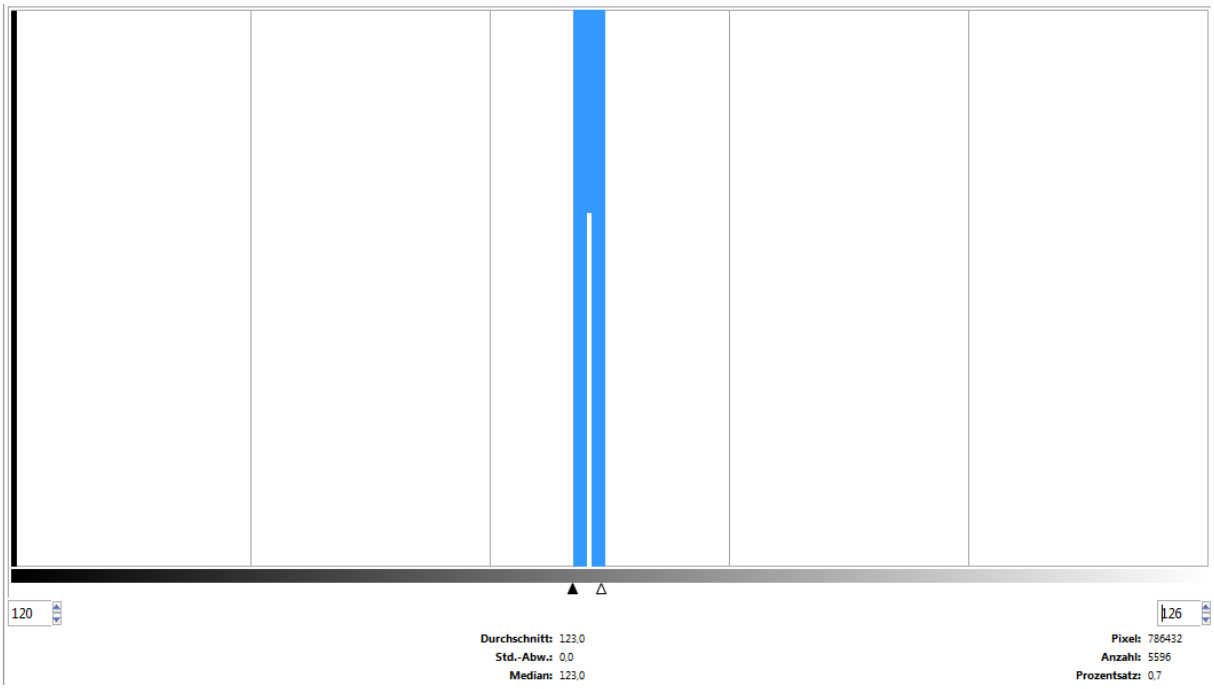


Figure 46 Histogram Differential Intel-Mesa3D

Table 6 shows the amount of considered pixels of the original pictures and the amount of considered pixels of the differential images for all combinations. The percentage deviation is shown in this table. That means for example that the differential image of Intel and NVIDIA has a deviation of 10.64 % referred to the original picture of NVIDIA and also a deviation of 10.64 % referred to the original picture of Intel.

Implementation	NVIDIA	Intel	Mesa
Pixel of original image	42435	42435	42638

Differential image	NVIDIA - Intel	NVIDIA - Mesa	Mesa – Intel
Pixel of differential image	397	5549	5596
Deviation to NVIDIA in %	0.94	13.08	-
Deviation to Intel in %	0.94	-	13.19
Deviation to Mesa in %	-	13.01	13.12

Table 6 Diff pixel

The average difference is about 9.04 %. The calculation precision of the implementations is one reason for this deviation. The inaccuracies between the implementations and the specifications are another reason. These values above do not represent a universal behavior of an implementation to another and relate only to the specific record. They demonstrate that already simple graphics differ

from each other. If the implementations run at their limits, the deviation could be significantly higher.

Such differences are often not important because the human eye cannot detect these small differences. But for more complex graphical applications like required in medical technologies, this behavior is a poor quality characteristic.

6.8. Security Test

6.8.1. Execution

Another important quality criterion is safety. Safety tests have a different question than most of the other tests because they are able to prove that the software does not contain features which should not be there. Important properties of security are confidentiality, availability and integrity. Checking these properties, many tests have to be done. For this thesis a test case for confidentially was created which is an example how such a test case can look like. Confidentiality is the property that information is only provided to a limited circle of recipients. For example when textures or vertex buffer objects are created, memory space is reserved for the objects. This test verifies that old data is not reloaded from the memory by proving that the memory is initialized with a constant color or a constant value. Therefore the memory is allocated and the content is checked before new content will be assigned to the memory. The content of the memory is compared byte by byte.

The test fails and indicates that the memory contains old data, when the same value is not assigned to each byte.

6.8.2. Evaluation

All considered implementations have passed the test case. Both the texture memory and the vertex object buffer memory are initialized to a constant value. This is only one example of a safety test. In general the formulation of a safety test is problematic because defining the patterns of vulnerability cannot be done precise enough. The general task is to find security-critical vulnerabilities in programs. It is important that the test extend over the whole program, because a single error would be sufficient to compromise the entire program. The outcome of this is that security testing can never prove that software is one hundred percent safe.

7. Evaluation of OpenGL

The results proof that significant differences between the implementations exist. These differences can be divided into 3 categories, differences because of inaccuracies in the specification, differences because of variations of implementations to the specification and quality of the tests. This thesis does not analyze all possible errors of OpenGL, but delivers basic approaches how specifications have to be tested, what opportunities graphics card manufacturers have and where the testing of specifications fails. The examined test scenarios reflect my interpretation of the OpenGL specification.

7.1. Evaluation of inaccuracies in the specification

The specification is the basis of OpenGL. A technical specification may not leave room for interpretations, but the specification of OpenGL is imprecise in many cases. The developers of the drivers have much room for interpretations.

The OpenGL specification was first published in 1992. Since then over 20 years have passed but the specification still does not have a clear structure. The structure of the specification has been completely revised with the latest release, OpenGL 4.3.

Many errors and deviations have occurred in the tests because the specification is imprecise or there is no unique assignment. With OpenGL 4.3 the structure has been improved but there are still plenty of structural problems. One of the biggest weaknesses is the lack of error analysis. Old applications which were created with previous OpenGL versions often used only a few commands. Furthermore the meaning of graphics cards increased in the last years. In earlier years when graphics cards did not have heavy tasks, an error analysis was not very important but today graphics cards calculate geometry data and manage shaders. OpenGL is used for complex calculations in different fields of technology for example medical technologies. Therefore a good error analysis is very important.

The shader language GLSL was launched with the publication of OpenGL 2.0 in 2004 and with it the programmable-function pipeline was introduced. From OpenGL 2.0 to OpenGL 3.1 the fixed-function pipeline and the programmable-function pipeline existed parallel (SEG 04, SEG 09). The publication was a necessary step because the graphics cards have evolved very quickly. Thus OpenGL 2.1 also offered many different ways to develop software with an identically visual result.

OpenGL did not separate the fixed-function pipeline and the programmable-function pipeline which results in a potential mix up of both types of programming. With the exception of the shader files, the Piglit test suite was not used in this thesis because old functions from the fixed-function pipeline were used together with functions of the programmable-function pipeline. OpenGL 3.0 started to

clean up the specification. OpenGL followed the idea of deprecation. Functions of the specification which have been marked as deprecated are removed with the next release of OpenGL (SEG 08).

Due to an unclear structure many inaccuracies slipped into the OpenGL specification. Since the release of OpenGL 3.1, the specification developed much faster and a lot of improvements are achieved up to the current version, OpenGL 4.3. The shader language GLSL got a better structure from the beginning. Functions are clearly separated and clarified. GLSL has the advantage to OpenGL that the language is similar to the high level language C and C + +. Therefore many ideas are taken from the C and C++ specifications and some errors or inaccuracies could be prevented. However the error analysis is also lacking in GLSL. The error handling is left to the compiler and linker.

7.2. Evaluation of variations of implementations to the specification

The various implementations also differ much from each other. The examination shows that Intel is the weakest. The Intel graphics card has failed most the functional test cases, followed by the NVIDIA graphics card. The software renderer Mesa3D passed most functional test cases.

The Intel and NVIDIA graphics cards have great differences in terms of shader programming. Often operators that are reserved for future versions of GLSL are already occupied by the two graphics card manufactures. Mesa3D has proved to be very precisely to the specification.

All implementations show weaknesses in the error analysis. The specification is not accurate in many points but the implementations also differ in some cases where the behavior is clearly defined. The error analysis is insufficiently considered by all implementations. In many practical cases of the graphics programming, error analyses helps to improve the quality of the code especially on the technical limitations of the implementations.

Examining the state variables, Intel performs worst and Mesa3D is best again. But this test depends on the considered specification because different state variables are used by different versions of the specification.

The graphics card manufacturers have difference experience with the development of OpenGL drivers. Mesa3D is open source software for Linux and supports a series of graphics cards manufacturers. The programming of Mesa3D has started in 1993 with the main focus on OpenGL. Mesa3D has a large community that push the programming forward. Another advantage of Mesa3D is located in the distribution of the software. Mesa3D is provided in a free repository, where every developer has always access to the current version of the source code. Therefore changes can be easily integrated and errors can be fast corrected without launching a new release. In addition Mesa3D is compiled by a cross compiler. This has the advantage that there is not a compiled driver

and each user always generates the driver for the corresponding hardware. Thus the software renderer can also be compiled under the operating system Windows (MES 13).

The programming of the software renderer has started in 2007. Therefore a completely new approach of programming was chosen, since the underlying hardware is the computer's CPU and no graphics card. At this time the developers had already enough experience in terms of OpenGL to avoid previous errors.

NVIDIA on the other hand has introduced its first graphics card, NVIDIA Riva 128, with OpenGL functionality in 1997 (NVI 13). Thus NVIDIA has a long experience in driver programming for OpenGL. NVIDIA does not provide the source code of the driver as open source. Therefore only available drivers of NVIDIA can be used and the driver cannot be compiled manually. Minor errors can be corrected only by updates from NVIDIA. Since NVIDIA does not provide an update to fix every little mistake, this takes time.

Compared to Mesa3D and NVIDIA, Intel published its first onboard graphics card, GMA 900 graphics chip, which supports OpenGL 1.4 in 2004 (WGM 12). At this time OpenGL 2.0 was already the current version. Mesa3D as open source software does not follow commercial goals, so a slower development is not critical. Intel pursues commercial goals therefore Intel is forced to compete with the market, but the graphics cards manufacturer had a backlog in terms of hardware and software. Intel, whose core business was not in developing graphics card, had been forced to close this gap quickly. The fast development maybe led to many errors in the software.

Generally the graphics card driver is not developed from scratch with every new version of OpenGL. As already discussed errors which are found very late are very expensive to compensate. Some of these errors with a low priority for the developers are not compensated and remain in the driver.

Additionally OpenGL has stagnated in its development between versions 2.0 in 2004 to version 3.1 in 2009 and only changed a little. The direct competitor Direct3D had a significantly faster development at this time. Probably the graphics card manufacturers Intel and NVIDIA used the experience of Direct3D and already occupied operators and functions in OpenGL which are reserved for future versions of OpenGL. The software renderer Mesa3D cannot fall back on this experience because only drivers for OpenGL are developed.

7.3. Evaluation of the quality of the test cases

Some test cases failed because the quality of the test cases is not sufficient. Testing software needs high quality test cases. Software testing is a wide topic and good testing reduces the amount of errors and increases the confidence in the software. In the case of the OpenGL specification, there is

the open source test suite Piglit. As already discussed, Piglit is a good approach, but there are many weaknesses.

For example there are several dependencies in a test case whereby the test criterion is not checked under certain circumstances. Furthermore test descriptions are missing and test results cannot be evaluated.

Besides the test suite Piglit, there is no other test suite to examine functional testing of OpenGL. In the field of non-functional testing, there are a few test suites that measure the performance and image quality and compare the measured results with results of other testers. These test suites are not used because there is no information about the tested features. The non-functional tests, listed in the thesis, serve as examples to compare the three implementations in performance and qualitative differences.

8. Perspective

This thesis has examined three different implementations against the OpenGL specification. Therefore the roles and functions of OpenGL were clarified first. Afterwards the used implementations were selected. Covering a broad spectrum, the Intel HD Graphics 3000 is used as an onboard graphics card, the NVIDIA Quadro FX 5800 is used as a high-end professional graphics card and Mesa3D is used as a software renderer. These implementations use different versions of OpenGL, therefore the selected subset of OpenGL functions is discussed in detail to compare the OpenGL specifications 2.1, 3.1 and 3.3. Then the operation of OpenGL has been considered in more detail and shaders are explained and how they are linked with OpenGL.

An appropriate test strategy is selected out of this information. OpenGL is considered as software, what makes software testing the most suitable kind of testing. Then a proper software testing strategy has been developed that is designed for testing software specifications from the perspective of an external tester.

Theoretically testing the OpenGL specification or another software specification needs black-box testing and white-box testing. In this case white-box testing is not possible, because this thesis has no access to the source code of the drivers implementations. This thesis explains how errors and undefined behavior can be determined only using black-box testing. The designed strategy has been applied based on the three selected implementations and the OpenGL specifications. The results are evaluated afterwards and possible reasons for the differences in the results are discussed.

The thesis does not put emphasize for completeness. There are many more errors in the implementations and the specifications that are not mentioned in this thesis. In the case of OpenGL the behavior of matrix calculation could deliver approaches for further analysis. Matrices can be computed through shader programs by the graphics card or in the OpenGL program by the CPU. An approach can check the differences in performance of calculations between CPU and graphics card.

Furthermore this thesis makes the point that the correction of small errors has the consequence of high costs, if they are found lately. A further approach could check whether errors are inherited to newer implementations for cost reasons.

The development of OpenGL was passed to the Khronos Group in 2006, because the development began to stagnate at this time. The Khronos Group is a consortium of several companies, such as hardware manufacturers and software vendors. After some time as the consortium has overcome the first difficulties, the development of new OpenGL releases has been rapidly progressed and OpenGL has caught up on the advantage of the main competitor Direct3D. The development of

OpenGL significantly helps for the development of new techniques. This thesis shows that the development is driven faster by a common interface.

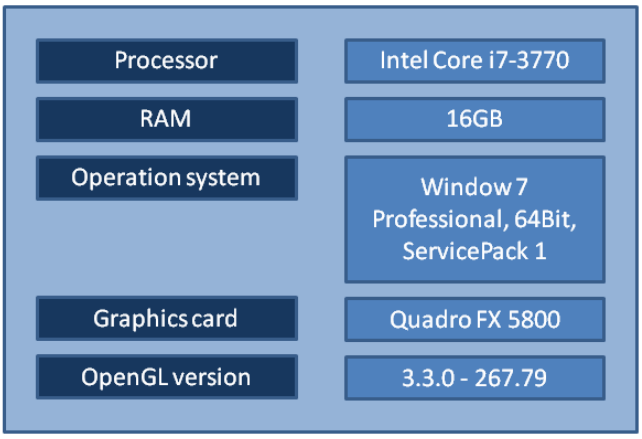
Many errors have been uncovered in the specification of OpenGL and in the various implementations. With the OpenGL 4.3 release, the OpenGL specification has been completely revised. Nevertheless, this is not enough and there are still detected errors. Especially for an API, the specification has to be defined very precisely and cannot leave room for interpretations, as proven by this thesis.

In addition implementations have to abide more closely by the specification. Software testing is very important to achieve these improvements. During this thesis the quality of the test cases of the test suite Piglit has been questioned. For evaluating future versions of OpenGL, the quality of test cases have to be improved.

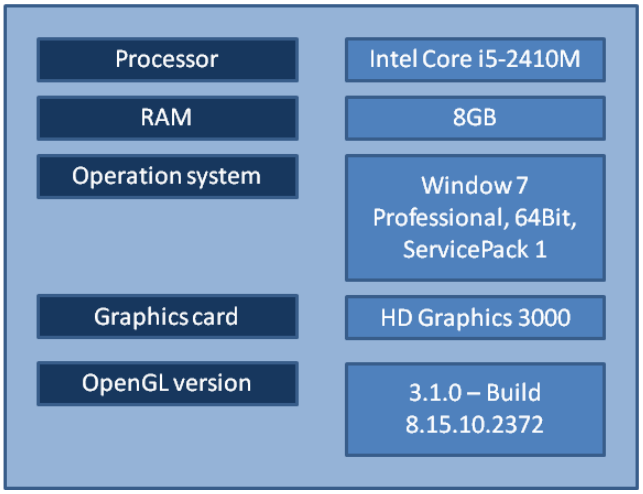
Appendix

Systems

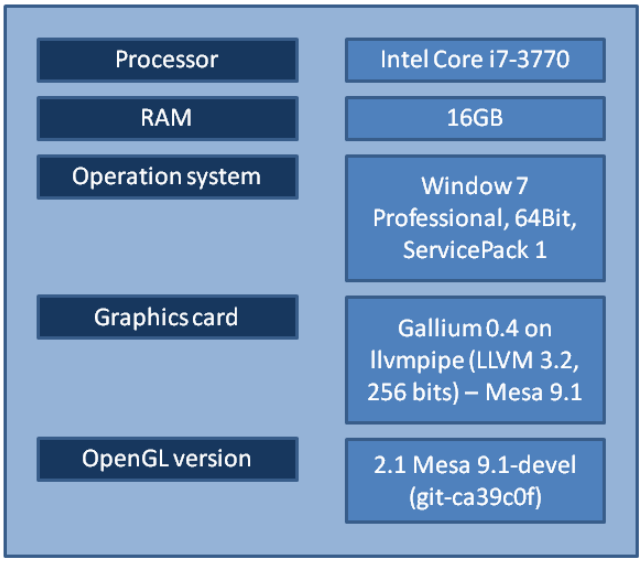
NVIDIA



Intel



Mesa3D



Functions

Function name	Function name
GLenum glGetError(void)	void glBindAttribLocation(GLuint program, GLuint idex, const GLchar *name)
void glEnable(GLenum cap)	void glGetAttribLocation(GLuint program, const GLchar *name)
void glDisable(GLenum cap)	void glGenBuffers(GLsizei n, GLuint *buffers)
GLuint glCreateShader(GLenum type)	void glBindBuffer(GLenum target, GLuint buffer)
void glShaderSource(GLuint shader)	void glBufferData(GLenum target, GLsizeiptr size, const void *data, GLenum usage)
void glCompileShader(GLuint shader)	void glBufferSubData(GLenum mode, GLint first, GLsizei count)
void glGetShaderiv(GLuint shader, GLenum pname, GLint *params)	void glDeleteBuffers(GLsizei n, const GLuint *buffers)
void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei *length, GLchar *infolog)	void glLineWidth(GLfloat width)
void glDeleteShader(GLuint shader)	void glDrawArrays(GLenum mode, GLint first, GLsizei count)
Uint glCreateProgram(void)	void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid *indices)
void glAttachShader(GLuint program, GLuint shader)	Void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)
void glDetachShader(GLuint program, GLuint shader)	void glDepthRange(GLclampf n, GLclampf f)
void glLinkProgram(GLint program)	void glGenTextures(GLsizei n, GLuint *textures)
void glGetProgramiv(GLuint program, GLenum pname, GLuint *params)	void glDeleteTextures(GLsizei n, GLuint textures)
void glGetProgramInfoLog(GLuint program, GLsizei maxLength, GLsizei *length, GLchar *infoLog)	void glBindTextures(GLenum target, GLuint texture)
void glValidateProgram(GLuint program)	void glTexImage2D(GLenum target, GLint level, GLenum internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, Glenum type, const void *pixels)
void glDeleteProgram(GLuint program)	void glPixelStorei(GLenum pname, GLint param)
void glUseProgram(GLuint program)	void glTexParameteri(GLenum target, GLenum pname, GLint param)
void glGetActiveUniform(GLuint program, GLuint index, GLsizei bufSize, GLsizei *length, GLuint *size, GLenum *type, GLchar *name)	void glGenerateMipmap(GLenum target)
GLint glGetUniformLocation(GLuint program, const char *name)	void glActiveTexture(GLenum texture)

void glUniform1f(GLint location, GLfloat x)	void glCompressedTexImage2D(GLenum target, GLint level, GLenum internatFormat, GLsizei width, GLsizei height, GLint border, GLsizei imageSize, const void *data)
void glUniform2f(GLint location, GLfloat x, GLfloat y)	void glTexSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLsizei width, GLsizei height, GLenum format, GLenum type, const void *pixels)
void glUniform3f(GLint location, GLfloat x, GLfloat y, GLfloat z)	void glCompressedTexSubImage2D(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLsizei width, GLsizei height, GLenum format, GLenum imageSize, const void *pixels)
void glUniform4f(GLint location, GLfloat x, GLfloat y, GLfloat z, GLfloat w)	void glCopyTexImage2D(GLenum target, GLint level, GLenum internalFormat, GLint x, GLint y, GLsizei width, GLsizei height, GLint border)
void glVertexAttrib1f(GLuint index, GLfloat x)	void glCopyTexSubImage2D(GLenum target, GLint level, GLint level, GLint xoffset, GLint yoffset, GLint x, GLint y, GLsizei width, GLizei height)
void glVertexAttrib2f(GLuint index, GLfloat x, GLfloat y)	void glClear(GLbitfield mask)
void glVertexAttrib3f(GLuint index, GLfloat x, GLfloat y, GLfloat z)	void glClearDepth(GLclampf depth)
void glVertexAttrib4f(GLuint index, GLfloat x, GLfloat y, GLfloat z, GLfloat w)	void glClearStencil(GLint s)
void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *ptr)	void glDepthMask(GLboolean depth)
void glEnanbleVertexAttribArray(GLuint index)	void glStencilMask(GLuint mask)
void glDisableVertexAttribArray(GLuint index)	void glStencilMaskSeparate(GLenum face, GLuint mask)
void glGetActiveAttrib(GLuint program, GLuint index, GLsizei busSize, GLsizei *length, GLint *size, GLenum *type, GLchar *name)	void glScissor(GLint x, GLint y, GLsizei width, GLsizei height)
void glBlendFunc(GLenum sfactor, GLenum dfactor)	void glStencilFuncSeparate(GLenum face, GLenum func, GLint ref, GLuint mask)
void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum dstAlpha)	void glStencilOp(GLenum sfail, GLenum zfail, GLenum zpass)
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, void *pixels)	void glDepthFunc(GLenum func)

StateVariables

OpenGL 2.1

GL_MAX_LIGHTS
GL_MAX_CLIP_PLANES
GL_MAX_MODELVIEW_STACK_DEPTH
GL_MAX_PROJECTION_STACK_DEPTH
GL_MAX_TEXTURE_STACK_DEPTH
GL_SUBPIXEL_BITS
GL_MAX_D_TEXTURE_SIZE
GL_MAX_TEXTURE_SIZE
GL_MAX_TEXTURE_LOD_BIASGLfloat.
GL_MAX_CUBE_MAP_TEXTURE_SIZE
GL_MAX_PIXEL_MAP_TABLE
GL_MAX_NAME_STACK_DEPTH
GL_MAX_LIST_NESTING
GL_MAX_EVAL_ORDER
GL_MAX_ATTRIB_STACK_DEPTH
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH
GL_AUX_BUFFERS
GL_SAMPLE_BUFFERS
GL_SAMPLES
GL_MAX_TEXTURE_UNITS
GL_MAX_VERTEX_ATTRIBS
GL_MAX_VERTEX_UNIFORM_COMPONENTS
GL_MAX_VARYING_COMPONENTS
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
GL_MAX_TEXTURE_IMAGE_UNITS
GL_MAX_TEXTURE_COORDS
GL_MAX_FRAGMENT_UNIFORM_COMPONENTS
GL_MAX_DRAW_BUFFERS
GL_ALIASED_POINT_SIZE_RANGE
GL_SMOOTH_POINT_SIZE_RANGE
GL_ALIASED_LINE_WIDTH_RANGE
GL_SMOOTH_LINE_WIDTH_RANGE

OpenGL 3.1

GL_MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS
GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS
GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS
GL_MAX_CLIP_DISTANCES
GL_SUBPIXEL_BITS
GL_MAX_D_TEXTURE_SIZE
GL_MAX_TEXTURE_SIZE
GL_MAX_ARRAY_TEXTURE_LAYERS
GL_MAX_TEXTURE_LOD_BIAS
GL_MAX_CUBE_MAP_TEXTURE_SIZE
GL_MAX_RENDERBUFFER_SIZE
GL_NUM_COMPRESSED_TEXTURE_FORMATS
GL_MAX_TEXTURE_BUFFER_SIZE
GL_MAX_RECTANGLE_TEXTURE_SIZE

GL_MAX_VERTEX_ATTRIBS
GL_MAX_VERTEX_UNIFORM_COMPONENTS
GL_MAX_VARYING_COMPONENTS
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
GL_MAX_TEXTURE_IMAGE_UNITS
GL_MAX_FRAGMENT_UNIFORM_COMPONENTS
GL_MIN_PROGRAM_TEXEL_OFFSET
GL_MAX_PROGRAM_TEXEL_OFFSET
GL_MAX_VERTEX_UNIFORM_BLOCKS
GL_MAX_FRAGMENT_UNIFORM_BLOCKS
GL_MAX_COMBINED_UNIFORM_BLOCKS
GL_MAX_UNIFORM_BUFFER_BINDINGS
GL_MAX_UNIFORM_BLOCK_SIZE
GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT
GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS
GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS
GL_MAX_DRAW_BUFFERS
GL_SAMPLE_BUFFERS
GL_SAMPLES
GL_MAX_COLOR_ATTACHMENTS
GL_MAX_SAMPLES
GL_POINT_SIZE_RANGE
GL_ALIASED_LINE_WIDTH_RANGE
GL_SMOOTH_LINE_WIDTH_RANGE

OpenGL 3.3

GL_MAX_CLIP_DISTANCES
GL_SUBPIXEL_BITS
GL_MAX_D_TEXTURE_SIZE
GL_MAX_TEXTURE_SIZE
GL_MAX_ARRAY_TEXTURE_LAYERS
GL_MAX_TEXTURE_LOD_BIAS.f
GL_MAX_CUBE_MAP_TEXTURE_SIZE
GL_MAX_RENDERBUFFER_SIZE
GL_NUM_COMPRESSED_TEXTURE_FORMATS
GL_MAX_TEXTURE_BUFFER_SIZE
GL_MAX_RECTANGLE_TEXTURE_SIZE
GL_MAX_VERTEX_ATTRIBS
GL_MAX_VERTEX_UNIFORM_COMPONENTS
GL_MAX_VERTEX_UNIFORM_BLOCKS
GL_MAX_VARYING_COMPONENTS
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS
GL_MAX_GEOMETRY_UNIFORM_BLOCKS
GL_MAX_GEOMETRY_INPUT_COMPONENTS
GL_MAX_GEOMETRY_OUTPUT_COMPONENTS
GL_MAX_GEOMETRY_OUTPUT_VERTICES
GL_MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS
GL_MAX_GEOMETRY_TEXTURE_IMAGE_UNITS
GL_MAX_FRAGMENT_UNIFORM_COMPONENTS
GL_MAX_FRAGMENT_UNIFORM_BLOCKS
GL_MAX_FRAGMENT_INPUT_COMPONENTS
GL_MAX_TEXTURE_IMAGE_UNITS

GL_MIN_PROGRAM_TEXEL_OFFSET-
GL_MAX_PROGRAM_TEXEL_OFFSET
GL_MAX_UNIFORM_BUFFER_BINDINGS
GL_MAX_UNIFORM_BLOCK_SIZE
GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT
GL_MAX_COMBINED_UNIFORM_BLOCKS
GL_MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS
GL_MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS
GL_MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS
GL_MAX_VARYING_COMPONENTS
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS
GL_MAX_SAMPLE_MASK_WORDS
GL_MAX_COLOR_TEXTURE_SAMPLES
GL_MAX_DEPTH_TEXTURE_SAMPLES
GL_MAX_INTEGER_SAMPLES
GL_MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS
GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS
GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS
GL_MAX_DRAW_BUFFERS
GL_SAMPLE_BUFFERS
GL_SAMPLES
GL_MAX_COLOR_ATTACHMENTS
GL_MAX_SAMPLES
GL_POINT_SIZE_RANGE
GL_ALIASED_LINE_WIDTH_RANGE
GL_SMOOTH_LINE_WIDTH_RANGE

DVD – Content

The provided DVD contains:

- 01_MasterThesis
- 02_Mesa3D-SoftwareRenderer-64Bit (OpenGL driver and compilation description)
- 03_OpenGL-Tester (test program and description)
- 04_OpenGL-TesterSource (source code of test program)
- 05_TestResults (test results and archives)

Bibliography

IVV 98 o. V.: *IEEE Standard for Software Verification and Validation* Internet 2012-08-11 <https://cours.etsmtl.ca/mgl800/private/Normes/ieee/1012standard.pdf>

VPM 10 o. V.: *World, View and Projection Matrix Unveiled* Internet 2013-08-10 <http://robertokoci.com/world-view-projection-matrix-unveiled/>

FNT 12 o. V.: *Software Testing Class: Functional Testing Vs Non-Functional Testing* Internet 2013-08-10 <http://www.softwaretestingclass.com/functional-testing-vs-non-functional-testing/>

KOM 13 o. V.: *Elektronik Kompendium - Grafikkarte* Internet <http://www.elektronik-kompendium.de/sites/com/0506191.htm>

ABI 08 Abi-Chahla, Fedy: *DirectX 11 und OpenGL 3: Der 3D-API-Krieg ist beendet* Internet 2013-08-25 <http://www.tomshardware.de/DirectX-OpenGL,testberichte-240159.html>

AJI 04 Ajitha, Amrish; Shah, Ashna; Datye, Bharathy J.; Deepa M G, James: *Software Testing Guide Book* Internet 2004-07-20 <http://de.scribd.com/doc/2175351/Software-Testing-Guide-Book-Part-1>

BER 01 Bertolino, Antonia: *Chapter 5 Software Testing* Internet 2013-08-03 <http://www.chaseplace.com/UWClass-2/files/SWEbok/SWEBOK-Test.pdf>

BMC 04 bmc: *The Economics of Software* Internet <https://blogs.oracle.com/bmc/entry/the_economics_of_software>

CPS 97 CPSC333: *Introduction to Testing* Internet <http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC333/Lectures/Testing/intro.html>

KAN 02 Kaner, Cem u. a.: *Lessons learned in software testing: A context-driven approach.* New York: Wiley, 2002

KES 06 Kessenich, John: *The OpenGL Shading Language: Language Version 1.20 Document Revision 8 07-Sept-2006* Internet 2013-09-01 <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf>

KES 04 Kessenich, John; Baldwin, Dave; Rost, Randi: *The OpenGL Shading Language: Language Version 1.10 Document Revision 59 30-April-2004* Internet 2013-09-01 <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.10.59.pdf>

LEA 13 Leach, Geoff: *Graphics pipeline and animation* Internet 2013-09-21 <http://goanna.cs.rmit.edu.au/~gl/teaching/Interactive3D/2013/lecture2.html>

LUT 12 Luten, Eddy: *OpenGLBook: What is OpenGL?* Internet 2013-09-21 <http://openglbook.com/the-book/preface-what-is-opengl/>

MCC 09 McCaffrey, James D.: *Software Testing:* Samir Riad, 2009

MCN 12 McNickle, Gary: *Sensible Software Developement* Internet 2013-07-12 <http://mcnickle.org/category/sensible-software-development/>

MES 13 Mesa3D: *The Mesa3D 3D Graphics Library* Internet 2013-06-13 <http://www.Mesa3D.org/intro.html>

MUN 11 Munshi, Aaftab u. a.: *OpenGL ES 2.0 programming guide.* Open GL. 4. pr. Upper Saddle River, NJ: Addison-Wesley, 2011

MUN 10 Munshi, Aaftab; Leech, Jon: *OpenGL ES Common Profile Specification: Version 2.0.25 (Full Specification) (November 2, 2010)* Internet 2013-08-06 <http://www.opengl.org/registry/doc/glspec33.core.20100311.withchanges.pdf>

NVI 13 NVIDIA: *Die Geschichte von NVDIA* Internet 2013-09-13 <http://www.NVIDIA.de/object/corporate-timeline-de.html>

OPE 13 OpenGL: *The Industry's Foundation for High Performance Graphics* Internet <http://www.opengl.org/>

OVE 12 Overvoorde, Alexander: *OpenGL* Internet <http://open.gl/>

PAT 05 Patterson, David A.; Hennessy, John L.: *Computer organization and design: The hardware/software interface.* Princeton, N.J: Recording for the Blind & Dyslexic, 2005

PIG 13 piglit Internet 2013-07-17 <http://cgit.freedesktop.org/piglit/tree/>

ROS 10 Rost, Randi J.; Licea-Kane, Bill: *OpenGL shading language.* OpenGL series. 3rd ed. Upper Saddle River, NJ: Addison Wesley, 2010

SAL 13 Salvator, Dave: *ExtremeTech 3D Pipeline Tutorial* Internet <http://www.pcmag.com/article2/0,2817,9722,00.asp>

SEG 08 Segal, Mark; Akeley, Kurt; Leech, Jon: *The OpenGL Graphics System: A Specification: Version 3.0 - August 11, 2008* Internet 2013-08-01 <http://www.opengl.org/registry/doc/glspec30.20080811.pdf>

SEG 09 Segal, Mark; Akeley, Kurt; Leech, Jon: *The OpenGL Graphics System: A Specification: Version 3.1 - March 24, 2009* Internet 2013-08-01 <http://www.opengl.org/registry/doc/glspec31.20090324.pdf>

SEL 10 Segal, Mark; Akeley, Kurt; Leech, Jon: *The OpenGL Graphics System: A Specification: Version 4.0 (Core Profile) - March 11, 2010* Internet 2013-08-04 <http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf>

SEG 10 Segal, Mark; Akeley, Kurt; Leech, Jon: *The OpenGL Graphics System: A Specification: Version 3.3 (Core Profile) - March 11, 2010* Internet 2013-08-04 <http://www.opengl.org/registry/doc/glspec33.core.20100311.withchanges.pdf>

SEG 12 Segal, Mark; Akeley, Kurt; Leech, Jon: *The OpenGL Graphics System: A Specification: Version 4.3 (Core Profile) - August 6, 2012* Internet 2013-08-06 <http://www.opengl.org/registry/doc/glspec33.core.20100311.withchanges.pdf>

SEG 04 Segal, Mark; Akeley, Kurt; Leech, Jon; Brown, Pat: *The OpenGL Graphics System: A Specification: Version 2.0 - October 22, 2004* Internet 2013-08-01 <http://www.opengl.org/registry/doc/glspec20.20041022.pdf>

SEG 06 Segal, Mark; Akeley, Kurt; Leech, Jon; Brown, Pat: *The OpenGL Graphics System: A Specification: Version 2.1 - July 30, 2006* Internet 2013-08-01 <http://www.opengl.org/documentation/specs/version2.1/glspec21.pdf>

SER 13 ServiceNow: *ServiceNow System Administrating,* 2013

SHR 10 Shreiner, Dave: *OpenGL programming guide: The official guide to learning OpenGL, versions 3.0 and 3.1.* OpenGL series. 7th ed. Upper Saddle River, NJ: Addison-Wesley, 2010

SHR 06 Shreiner, Dave; Woo, Mason: *OpenGL programming guide: The official guide to learning OpenGL, version 2.* 5th edition. Upper Saddle River (N.J.), Boston, Indianapolis [etc.]: Addison-Wesley, op. 2006

WGM 12 Wikipedia: *Intel GMA* Internet 2013-09-10 <http://de.wikipedia.org/wiki/Intel_GMA>

WSR 13 Wikipedia: *Software rendering* Internet 2013-08-14 <http://en.wikipedia.org/wiki/Software_rendering>

WUV 13 Wikipedia: *UV-Koordinaten* Internet 2013-08-10 <http://de.wikipedia.org/wiki/UV-Koordinaten>

WHD 13 Wikipedia: *Intel HD Graphics* Internet 2013-09-10 <http://de.wikipedia.org/wiki/Intel_HD_Graphics>

WGK 13 Wikipedia: *Grafikkarte* Internet 2013-09-15 <http://de.wikipedia.org/wiki/Grafikkarte>

WOG 13 Wikipedia: *OpenGL* Internet 2013-09-20 <http://en.wikipedia.org/wiki/OpenGL>

WIL 06 Williams, Lauri: *Testing Overview and Black-Box Testing Techniques* Internet 2013-07-20 <http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf>

WRI 11 Wright, Richard S.: *OpenGL superbible: Comprehensive tutorial and reference.* 5th ed. Upper Saddle River, NJ: Addison-Wesley, 2011