Examensarbete

3D Graphics Technologies for Web Applications
An Evaluation from the Perspective of a Real World Application

Master thesis performed in information coding
by

Klara Waernér

LiTH-ISY-EX--12/4562--SE
Linköping 2012-06-19
3D Graphics Technologies for Web Applications
An Evaluation from the Perspective of a Real World Application

Master thesis in information coding
at Linköping Institute of Technology
by

Klara Waernér

LiTH-ISY-EX--12/4562--SE

Supervisors: Fredrik Bennet
SICK IVP AB

Jens Ogniewski
ISY, Linköping University

Examiner: Ingemar Ragnemalm
ISY, Linköping University

Linköping 2012-06-19
Abstract

Web applications are becoming increasingly sophisticated and functionality that was once exclusive to regular desktop applications can now be found in web applications as well. One of the more recent advances in this field is the ability for web applications to render 3D graphics. Coupled with the growing number of devices with graphics processors and the ability of web applications to run on many different platforms using a single code base, this presents an exciting new possibility for developers of 3D graphics applications.

This thesis aims to explore and evaluate the technologies for 3D graphics that can be used in web applications, with the final goal of using one of them in a prototype application. This prototype will serve as a foundation for an application to be included in a commercial product. The evaluation is performed using general criteria so as to be useful for other applications as well, with one part presenting the available technologies and another part evaluating the three most promising technologies more in-depth using test programs.

The results show that, although some technologies are not production-ready, there are a few which can be used in commercial software, including the three chosen for further evaluation; WebGL, the Java library JOGL and Stage 3D for Flash. Among these, there is no clear winner and it is up to the application requirements to decide which to use. The thesis demonstrates an application built with WebGL and shows that fairly demanding 3D graphics web applications can be built. Also included are the lessons learned during the development and thoughts on the future of 3D graphics in web applications.

Keywords
3D graphics, web applications, WebGL, Stage 3D, JOGL, mobile, HTML5, JavaScript
Abstract

Web applications are becoming increasingly sophisticated and functionality that was once exclusive to regular desktop applications can now be found in web applications as well. One of the more recent advances in this field is the ability for web applications to render 3D graphics. Coupled with the growing number of devices with graphics processors and the ability of web applications to run on many different platforms using a single code base, this represents an exciting new possibility for developers of 3D graphics applications.

This thesis aims to explore and evaluate the technologies for 3D graphics that can be used in web applications, with the final goal of using one of them in a prototype application. This prototype will serve as a foundation for an application to be included in a commercial product. The evaluation is performed using general criteria so as to be useful for other applications as well, with one part presenting the available technologies and another part evaluating the three most promising technologies more in-depth using test programs.

The results show that, although some technologies are not production-ready, there are a few which can be used in commercial software, including the three chosen for further evaluation; WebGL, the Java library JOGL and Stage 3D for Flash. Among these, there is no clear winner and it is up to the application requirements to decide which to use. The thesis demonstrates an application built with WebGL and shows that fairly demanding 3D graphics web applications can be built. Also included are the lessons learned during the development and thoughts on the future of 3D graphics in web applications.
## Contents

### Abstract

1. Introduction
   1.1 Problem Description
   1.2 Scope
   1.3 Methodology
   1.4 Report Outline

2. Required Theory
   2.1 3D Graphics
     - Creating 3D Geometry
     - Drawing in 3D
     - 3D Graphics Programming
     - Height Fields
   2.2 Markup Languages
   2.3 The Web Today
   2.4 3D Graphics on the Web

3. Current Technologies
   3.1 Main Focus
   3.2 Evaluation Criteria
   3.3 JavaScript
     - WebGL
   3.4 Plug-ins
     - Shockwave
     - Silverlight
   3.5 Java
     - Java 3D
     - JOGL
     - LWJGL
   3.6 Flash

---

Abstract

1. Introduction

2. Required Theory

3. Current Technologies
CONTENTS

List of Tables 78
Listings 79
Acknowledgements

Writing this thesis was no easy task, but having a good supervisor made it much easier. I would like to thank Fredrik Bennet for the insightful discussions that improved this thesis and the web application, for showing up at my desk whenever I had a question, providing detailed and useful comments on every chapter in this thesis (even though I delivered them in bulk), allowing me to (at least try to) help him track down a few hard to find bugs within another application, and much more that cannot fit in this small space.

My examiner, Ingemar Ragnemalm and my university supervisor, Jens Ogniewski, deserve thanks for answering all my questions and providing constructive criticism on both the thesis and the overall methodology.

I would also like thank everyone at SICK IVP for showing an interest in my thesis work and the resulting web application, for providing feedback and showing me how this thesis work fits into a bigger picture, offering general advice in terms of JavaScript programming and user interface design, and for giving me the opportunity to work at a great company.
Chapter 1

Introduction

Web applications are rapidly entering domains where desktop applications have previously been the only viable alternative. From a developers perspective it is easy to see why the interest in these web browser-based applications has been rising. The resulting applications can run on any combination of hardware and operating system which supports a modern web browser, including devices such as mobile phones and tablets.

Furthermore, the end user does not have to install the software on their device to use it, this is done automatically by the web browser and any associated plug-ins. Overall, web applications offers developers the opportunity to reach a large audience without much of the difficulties of supporting multiple platforms that has previously been the case with desktop applications.

Interactive 3D graphics is one of the largest areas where web applications have not seen much success until recently. Part of this has been because this type of interactivity demands much of the underlying device and software, such as the virtual machines for scripting languages so often involved in these types of applications. Another reason is the lack of a 3D graphics application programming interface (API) for most web application technologies.

Both of these areas have seen improvements, the former with techniques such as just-in-time (JIT) compilation and the latter with the introduction of technologies such as Stage 3D for Flash[1] and WebGL[2]. But there are also older alternatives, such as Java, available for developers who wish to create these types of applications. The choice of technology for 3D graphics does therefore not have an obvious answer when it comes to web development.

This thesis aims to evaluate current technologies that enable interactive 3D graphics in web applications and use one of these in the development of an application. This demo application will serve as a foundation for a more advanced application which will be released onto the market as a component of a larger product. Because of this, the thesis is more practical than theoretical and will hopefully be useful for similar web application projects.
1.1 Problem Description

A new 3D camera developed by SICK is at the centre of this thesis. This 3D camera captures not only an intensity image of the objects in its view, but also measures the depth at the corresponding position for each pixel in this image. The camera captures an image using a process called laser triangulation. By using an application running on a standard PC, it can be programmed to locate and inspect objects in these images. The camera also includes a web server which provides a web-based interface for configuring certain aspects of the camera.

SICK has previously developed 2D cameras which included not only a web-based configuration interface, but also a web-based live view of the images that the camera produced. This makes it easy for an operator to check that the camera is operating correctly and is properly configured. A similar web-based live view for the 3D camera displaying a 3D model of the combined depth and intensity images from the camera is needed and the development of such a view is one of the objectives for this thesis. The 3D model will be continuously updated as new images are requested and received from the 3D camera.

The first step in developing the web application for displaying the 3D model is to know how the 3D graphics part of it can be implemented using technologies that are readily available today. There have been a few previous studies of this issue, evaluating 3D graphics technologies for web applications. Some studies have evaluated different technologies, identified their shortcomings and presented their own alternative (see for example [3] and [4]).

Others have focused on further developing one of the alternatives available at the time (such as [5]) and done such comparisons as a part of their study. However, few studies so far have focused on the evaluation from the perspective of an application that will be used in the real world. The field also changes quickly and evaluations that were applicable at the time of their writing may no longer be up-to-date.

1.2 Scope

There are, as described above, two parts to the thesis. The first is the evaluation of current technologies for 3D graphics in web applications and will be carried out from the perspective of an application running in a web browser. This means that even if there are other ways to run an application built using a specific technology, say for example a Java applet versus a Java application, only the parts of the technology used when running the application in a web browser (in this case the Java applet) will be considered.

To be able to efficiently compare the technologies, they will each be evaluated against a few fundamental criteria (which will be presented in the next section). These criteria have been selected with the application that will be developed
1.3 Methodology

Each technology presented in the chapters to come will be evaluated using a few well-defined criteria, briefly presented below:

- **Compatibility** with web browsers, operating systems and hardware
- **Installation** of software required on the target computer
- **Support** given by the development environment, tools and documentation
- **Integration and communication** with the rest of the web page
- **Performance** in terms of frames per second (FPS) and image quality
- **Future** of the technology

A more in-depth description of each criterion can be found in chapter 3.

The evaluation itself is split into two parts. First, a comprehensive list of current technologies will be presented together with a brief description of each. Each of these technologies will be graded using a simple scale and the evaluation criteria. At the end of this presentation, three of them are selected to compete in the next round. In the second part of the evaluation, these three technologies are evaluated further by implementing a small test application in each. This will
CHAPTER 1. INTRODUCTION

then lead to one being selected and used in the development of the application itself.

By following this approach, a large number of technologies can be evaluated and those which have significant drawbacks can be removed from the evaluation early. Through the more practical evaluation of the remaining three with the use of test applications, implementation issues can be discovered before development of the real application starts and they can be taken into consideration in the evaluation.

Many of the sources used in this report are web-based sources, for example the official web sites of the technologies evaluated. This is to be expected for a thesis about web technologies, with specifications such as the WebGL specification not being available in any other format. For a short discussion concerning the choice of sources, see chapter 1.

1.4 Report Outline

The report begins by evaluating a large number of technologies and successively narrows down the best possible technology. Each chapter and its contents are presented below:

- **Chapter 2** presents the relevant background theory for the thesis to readers unfamiliar with web technologies or 3D graphics

- **Chapter 3** gives an overview of the currently available technologies for 3D graphics in web applications, evaluates each of them briefly and selects potential candidates for further evaluation

- **Chapter 4** provides a more detailed discussion of the strengths and weaknesses of the potential candidate technologies using the results from implementing a small test program in each. At the end of the chapter, one technology will be selected for use in the demo application.

- **Chapter 5** details the implementation of the demo application and the lessons learned during the implementation

- **Chapter 6** discusses the evaluation and the demo application. It also presents the conclusions drawn from these. Possible future directions are also discussed, both technologies and future work.

Also included are the small example programs used in chapter 4 as a part of evaluating the development environment of each technology. The test programs used in the same chapter to evaluate other aspects, such as performance, are not included in the report because of their large size.
Chapter 2

Required Theory

Before introducing all of the available technologies which will be evaluated, a presentation of the concepts and terms used in the coming chapters is in order. This chapter will present the absolute fundamentals of 3D graphics and web technologies and will provide the reader with the knowledge that is required for an understanding of the coming discussion. Readers who are already familiar with these two topics may choose to skip this chapter entirely.

2.1 3D Graphics

Computer graphics (and 3D graphics in particular) is a very broad topic. This section will only describe the concepts and ideas that are necessary for an understanding of the rest of this thesis. For more information, please refer to one of the many books or articles on the subject.

A simplified description of how 3D graphics works would be that points in 3D space are transformed, projected onto the screen and data about how these points are connected is used to draw 2D surfaces between these points.

2.1.1 Creating 3D Geometry

Starting from the beginning, a 3D model is a collection of 3D points and data about how these points are connected to each other. Such a point is called a vertex and each of these points has several properties associated with it. What kind of properties that are associated with a vertex depends on the application, but every vertex has at least one property, namely its position in 3D space. All of these positions are specified relative to some origin on the 3D model.

All of the 3D model’s vertices are stored in a list called a vertex buffer. The simplest (and one of the most commonly used) method of connecting vertices together to form the surfaces of the 3D model, is to connect them in pairs of three vertices and let every vertex be a corner in a triangle. It is completely up
CHAPTER 2. REQUIRED THEORY

to the creator of the 3D model to decide which vertices are going to be connected to other vertices.

To specify how vertices are connected to each other, a list of indices are used, called an index buffer. Each index refers to a position in the vertex buffer and hence one vertex. For example, an index buffer consisting of the indices 0, 1, 2 will, when drawn as triangles, create a triangle with the first, second and third vertices in the vertex buffer as the corners of the triangle (assuming the first index is zero). The index buffer 0, 1, 2, 3, 4, 5 would create the same triangle and another triangle with the following three vertices in the vertex buffer as corners.

An alternative to drawing triangles is to draw a triangle strip, where the first three indices produce a triangle and for every following index, a triangle is created from the current index and the two previous indices. For example, using the index buffer 0, 1, 2, 3 and drawing triangle strips would produce two triangles; triangle one is formed from 0, 1, 2 and triangle two from 1, 2, 3. Triangle strips should be used whenever possible since they require smaller buffers which in turn requires less memory.

2.1.2 Drawing in 3D

Many 3D graphics applications require that multiple 3D models should be displayed on the screen at the same time. A collection of 3D models is commonly called a 3D scene. To place 3D models at different locations and to rotate them, a transformation matrix is set each time a model is drawn. The process of producing an image from a 3D model or the whole 3D scene is called rendering. Rendering a scene involves drawing all the 3D models within it from a specified viewpoint, often called a camera.

A typical 3D graphics pipeline is shown in figure 2.1. The actual layout can be different depending on what library or API is used and what functionality is available. Rendering is commonly performed on a dedicated hardware unit, called the Graphics Processing Unit or GPU. The functionality provided by the GPU will also affect the layout. Since the text below is written from an application developer’s perspective, the term 3D graphics library will refer to all components involved in rendering, including any potential GPUs. Please also remember that the process is simplified and as such will only cover drawing filled triangles, not lines or any other type of primitives (there are only slight differences in the pipeline for these cases, mainly in the rasterisation step).

The process starts with a function call known as a draw call. The draw call tells the 3D graphics library to draw a part of or the whole 3D model using the specified vertex and index buffers. A draw call often also takes a range of indices that should be drawn and how to interpret these indices (as triangles, a triangle strip or any type of supported primitives).

To render anything, the library needs to know what vertices will be referenced
while drawing. The primitive processing step goes through specified range of indices from the index buffer and at each index, sends the referenced vertex off to the vertex shader.

In this simple graphics pipeline, there are two steps that can be programmed by the application programmer. These are the vertex shader and the fragment shader steps. A shader is a program that will run on the GPU if the rendering is hardware-accelerated. Newer GPUs allows for more programmability than this, such as creating new vertices while rendering. In the case of the vertex shader, this program can transform the position of the vertex and calculate input data for the fragment shader. The vertex shader works on one vertex at a time, but many vertices can be processed in parallel by a GPU.

The vertex that goes into the vertex shader has, as previously mentioned, a position that is relative to some origin on the model. The next step in the rendering process, the clipping step, requires that the position of each vertex is relative to the volume of 3D space that can be seen from the camera. A common way to perform this transformation is to first transform the vertex position to coordinate system common to all 3D models in the scene (world coordinates) and from there transform this position to a position that is relative to the camera (view coordinates). A projection (often perspective projection) is then performed to produce a new position (clip coordinates) that will later be up-scaled to a 2D position on the screen. This transformation chain, and applying the current transformation matrix, is a common task for the vertex shader.

Vertices that lie outside the viewing volume (called the frustum) of the camera should not be processed further since they do not affect the image on screen. It is the responsibility of the clipping step to remove any such vertices. Special care
CHAPTER 2. REQUIRED THEORY

has to be taken for vertices that are part of a triangle (or similar polygon) which in any way intersects the camera’s frustum.

So far, vertices have been processed separately, but drawing the polygons on the screen requires information about how the vertices are connected. The primitive assembly step produces triangles using the index buffer, independent of whether the vertices were originally connected in triangles, triangle strips or any other type of polygons. Triangles are used because they are the simplest possible 2D shape. The position of each vertex as the corner of a triangle has also been up-scaled to coordinates on the screen after this step.

Determining the pixels on the screen that lay within, or on the boundaries of, each of these new triangles is the task of the rasterisation step. These pixels are called fragments to distinguish them from the pixels in the final image, because they may or may not become pixels in the resulting image on the screen. They could be overwritten by other pixels from the same or later draw calls. A fragment could also be discarded by the fragment shader and will not be part of the final image if that happens.

The fragment shader (pixel shader is another name) is an application program that will be invoked for each of these pixels and just like the vertex shader, a modern GPU can run many fragment shaders on different fragments at the same time. As previously mentioned, data can be sent from the vertex shader to the fragment shader. Each fragment will get an interpolated value of the data depending on its position relative to the three vertices that make up the rasterised triangle.

Common uses of the fragment shader involve producing realistic lighting effects (using a particular lighting model) and putting a texture on the model (a texture is an image that is overlaid on top of the model). Texturing requires that each vertex has a pair of texture coordinates which specify where in the image the colour for the current fragment should be fetched. Texture coordinates are interpolated as specified above from the corners of the current triangle.

The last step before a fragment is written to the image on screen is for it to pass a couple of tests and other processing in the per-fragment operations step. This step among other things include depth testing, which prevents fragments that are part of triangles further away from the camera to overwrite fragments that are part of triangles closer to the camera. To perform this test, the depth value from the clip space coordinates are saved together with each pixel that is written to the image.

Producing the illusion of moving or changing 3D models in the scene requires the screen to be cleared, models or their transformations updated and drawing the 3D scene on the screen again. This process is analogous to a film camera, which captures an image of the world at specified rate and produces the illusion of motion when played back. The number of times the 3D scene is drawn and
displayed on the screen per second is called *frame rate* or *frames per second* (FPS). The scene can either be drawn a specified number of times per second (fixed frame rate) or as fast as possible. This is a useful performance measure if the latter method is used.

### 2.1.3 3D Graphics Programming

Developing 3D graphics applications requires a programming library which provides an API for 3D graphics. These libraries can be classified into two categories. The first group of libraries provide what is commonly called *immediate mode rendering* where the developer tells the library what 3D models should be drawn, and how each of them should be transformed, each time the scene is to be drawn. The other category provides *retained mode rendering*, a type of rendering where the developer constructs a scene using abstract data types provided by the library and then the library traverses the scene and renders each 3D model on the screen. The main differences between these two categories are who is in charge of the rendering process and who owns the properties (such as transformation) of each 3D model within the scene.

The rendering process can be performed on the CPU and is then called *software rendering*. Until recently, the most common types of CPUs could only run one process at a time. Because rendering 3D graphics is a process that is both computationally intensive and easy to parallelise, a GPU is commonly used to perform the task of rendering instead of the CPU. When rendering is done on a GPU, it is called *hardware-accelerated rendering*. Hardware-accelerated rendering is generally much faster than software rendering since it offloads the CPU to run the program at hand and the GPU is built to process many vertices and fragments in parallel.

Two of the most popular libraries for 3D graphics programming are OpenGL and Direct3D. OpenGL is an open standard managed by the Khronos Group and is available on many devices, from desktop computers and workstations to game consoles and mobile phones. Direct3D is a library created by Microsoft as one of the APIs that make up the DirectX suite of multimedia programming libraries available exclusively on Microsoft’s own platforms, such as the Windows operating system and the Xbox gaming consoles. Both of these libraries are immediate mode libraries and utilise hardware-acceleration. OpenGL can run in software mode if a requested feature is not supported by the GPU.

### 2.1.4 Height Fields

The 3D camera produces a type of image called a height field and it is the task of the demo application to produce a 3D model from this image and display that model. An example of a height field is shown in the first image in figure 2.2. It is
an image in which each pixel represents a height value. Mapping these values to the Y (vertical) components of vertices in a plane, such as the one shown in the second image of figure 2.2, results in a new 3D model with some vertices raised above others. The final result is shown in the last image of figure 2.2.

Figure 2.2: (a) The height field (b) A plane (c) Resulting 3D model

Height fields are often used when rendering landscapes as 3D models. If all vertices are equidistant, the plane can be generated at runtime and only the height field has to be permanently stored. This results in much less data that needs to be stored. However, the height fields have a few drawbacks. If a part of the resulting 3D model has to have a higher resolution, the resolution of the whole height field has to be increased. They are also incapable of displacing vertices along any other axis than the Y axis, which prevents features such as arcs to be created. A displacement map can solve this problem by storing the displacement along all three axes in the each of the red, green and blue components of the height field.

As some features may be occluded by other objects in the 3D camera’s field of view, it will produce height fields in which a certain value means that there is no data for that pixel. Because of this, the height field that is displayed within the application will have holes in it, which is something normal height fields do not have.

The camera uses a process called laser triangulation to create the height map. Describing this process is somewhat complicated and outside the scope of this thesis. Readers who are unfamiliar with laser triangulation are encouraged to study the subject briefly to gain a closer understanding of how the camera works and in what context it is used.

2.2 Markup Languages

A markup language defines the syntax for a text document in which some passages of text (or data) have been surrounded by elements called tags. The tags tells the application that reads the document how to interpret the data between the start
tag and end tag. Markup languages are common in web technologies, with the *HyperText Markup Language* (HTML) being the markup language used on web pages. XML is another popular markup language commonly used for exchanging data between applications in a common format. XHTML is a variant of HTML based on XML that is also used to describe web pages. A markup language is a type of declarative language, meaning that it only carries data and does not describe how to compute values like regular programming languages.

An example of how a document written in a markup language can look is shown in listing 2.1. This specific example shows a web page written in HTML. In HTML, start tags are enclosed within "<" and "">", while end tags are enclosed within "</" and ">".

```xml
<!doctype html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <p>This is an example of <b>formatting</b></p>
  </body>
</html>
```

Listing 2.1: Example of a markup language: a HTML document

Opening this document in a web browser puts "Hello World!" in the title bar of the web browser and "This is an example of *formatting*" is visible where the web browser displays the web page. Notice that the text within the "b" (bold) tags is emphasised.

## 2.3 The Web Today

A special tag in HTML allows scripts to be included in HTML documents. These scripts can traverse HTML, XML and XHTML documents and also manipulate them through an interface called the *Document Object Model* or DOM. The HTML standard itself does not mandate a specific scripting language, it is left up to each web browser developer whether they want include support for a certain programming language or not. A programming language called *JavaScript* has become the de-facto language to use for scripting HTML documents and is supported in almost all web browsers. The distinction between a web page (a HTML document) and a web application is that the web application is, through the use of scripting, structured like and works much like an ordinary desktop application.

*HTML 5* is the next generation of the HTML standard currently under development, but many parts are already supported in modern web browsers. Together with a couple of related standards, it adds many new features to both HTML
documents and the scripting APIs available in web browsers. Many of the new features are helpful for large scale web applications that, for example, need access to persistent storage, parallel processing in threads and more generic types of network communication. These APIs generally provide a higher level of abstraction than similar APIs built in to the underlying operating system and are also portable across platforms.

Measuring the current market shares of different web browsers is difficult. Web sites are targeted toward and attract different groups of people. A web browser with the highest market share in one of these groups may not be the largest in another. However, the statistics that are available, show that Google Chrome has 32.56% of the market, followed by Microsoft Internet Explorer with 31.61%, Mozilla Firefox at 25.53%, Apple Safari at 7.08%, Opera with 1.75% and other web browsers at 1.47\%[6]. Internet Explorer has in previous years had a majority share of the market.

Even though the DOM and JavaScript provide much functionality in modern web browsers, some pages need features which are not available. Such features can be added to most browsers by installing a small program, called a web browser plug-in.

2.4 3D Graphics on the Web

Including 3D graphics on web pages is not a new trend. In 1994, a way to present 3D scenes in the web browser through a markup language called VRML was standardised. VRML allows 3D scenes to be specified in a language similar to HTML. While there are (or have been) niche applications that use VRML, there are few popular sites today that include VRML documents. It seems to have become a standard that never really grew popular enough to see wide-scale usage for a number of reasons.

Chief among these reasons is likely that the processing power available, first and foremost in reasonably priced computers, back in 1994 was not good enough to provide convincing 3D graphics. This has changed recently with dedicated graphics processors finding their way into more and more types of devices. Thanks to this, there has been a renewed interest in technologies that provide 3D graphics on web pages and in web applications and the next chapter will look more closely at these.
Chapter 3

Current Technologies

It will soon be evident that there are many approaches to delivering 3D graphics in a web application and as such, it is necessary to put some constraints on what technologies should be included in the evaluation. This chapter will do precisely that and once the constraints have been presented, technologies available today and matching these requirements will be discussed one by one. The goal is to quickly evaluate a large number of technologies and find those that are the most suitable for the demo application.

The discussion is based on the theory presented in the previous chapter and on a number of criteria which will be presented at the beginning of this chapter. At the end of the chapter, three technologies will be chosen as the best alternatives given the evaluation criteria. In the next chapter these three will be compared in a more practical manner by implementing a short test program in each of the technologies.

3.1 Main Focus

This evaluation will focus on what the application that will be developed as part of this thesis needs, which is primarily a small lightweight library or language for low-level graphics programming. This means that higher level libraries, such as 3D engines, are not included if they are built on a technology that could be used directly by the application instead. Part of the reasoning behind this is that a 3D engine or similar is not very likely to perform any better than the technology on which it is built and performance is an important criteria in the evaluation. It also provides a kind of filter removing many libraries that provide more or less the same functionality, while keeping the alternatives that actually differ to a greater amount.

A 3D engine might also not be very useful for many types of applications. For example, most 3D engines include a sound system, but the type of application that this thesis aims to produce has no need for such functionality. Furthermore,
if the engine is provided as a library to be included with the application the extra functionality will most likely mean a bigger library, which can be a problem for some applications.

However, if a technology uses more than one of the other technologies as a means to display 3D data, then it can be included in the evaluation. This should generally make the solution capable of running on more platforms or support more browsers than any of the technologies it was built upon and thus becomes more interesting to evaluate from a compatibility perspective.

At the end of this chapter, a number of products will be presented that did not satisfy these criteria, but are otherwise notable. They are not included in the evaluation or later testing.

### 3.2 Evaluation Criteria

To simplify the evaluation and to be able to more easily compare one technology against another, the criteria outlined below will be used to judge each technology. As previously noted, the criteria are first and foremost based on the needs of the demo application, but thought has gone into trying to make them more general and relevant to other applications as well. The amount of weight put into each criterion is likely to depend on what type of application that will be developed.

**Compatibility** In what web browsers can the technology be used and on what operating systems? What are the requirements on the hardware? Does it work on mobile devices such as tablets and mobile phones?

**Installation** Does the end-user have to install any extra software (such as a web browser plug-in) on their device to use an application built with the technology? Are there any libraries that have to be bundled together with the application?

**Support** What tools (for example a debugger) are available when working with the technology? How easy or complex is it to develop applications using it? Is there a standard development environment? How much support is available from the documentation and other sources and what is the quality of each?

**Integration and communication** What possibilities are there to communicate data from the 3D environment onto the web page and vice versa? As an example, is it possible to get the position of a 3D object and display it as text on the web page?

**Performance** How many frames per second (FPS) can be attained when drawing a static model? How big is the difference when drawing an animated
model? In terms of image quality, are there any visible artefacts in the resulting image?

**Future** Is it likely that the technology will be maintained and developed further in the (near) future? Could it become available on future platforms?

In this chapter, some of the above criteria will only be applied briefly to keep the evaluation short. These are first and foremost the *performance* and *integration and communication* criteria. Testing the performance generally requires implementing a benchmark program for each alternative and then measuring the results, but this would take too much time. There could be already finished benchmark programs for a technology which can be used, but this might not always be the case and would force a benchmark program to be implemented as part of this thesis. As previously mentioned, this is not feasible to do. The same reasoning is applied to the *integration and communication* criterion.

After each technology has been presented, a short table follows with a summary of the strengths and weaknesses in terms of the above criteria for the particular technology. For each criterion, a grade of *Good*, *Average* or *Weak* is given. These grades are based upon objective measurements. For example, the platform criterion is graded *Good* if it supports most desktop platforms and mobile devices. *Average* if it only supports most desktop platforms, but no mobile platforms. The grade *Weak* is given to technologies that only supports some desktop platforms. It should be fairly evident why a technology got a particular grade by reading the section concerning that technology.

These grades will hopefully aid the reader by providing an brief summary when skimming through this chapter. All of them will then be compiled into a larger table at the very end of this chapter to allow for easy comparison of all of the presented technologies.

### 3.3 JavaScript

As mentioned in section 2.3, JavaScript is the de facto scripting language for web pages and web applications. In this first section, technologies which provide an API through JavaScript to display 3D graphics are presented.

#### 3.3.1 WebGL

WebGL is a relatively new standard for low-level 3D graphics programming (revision 1.0 was published in February 2011)\(^2\). It is based upon OpenGL ES 2.0, a standard for 3D graphics implemented in mobile devices such as the iPhone, iPad and many Android-based devices. OpenGL ES in turn is a subset of the 3D graphics API OpenGL (presented in section 2.1.3), which is widely used on desktop computers. The Khronos Group is the organization behind all these
standards. As a standard, web browsers are free to implement WebGL as they see fit, this means that the underlying graphics API that the browser use does not have to be OpenGL-based, but most implementations are hardware accelerated.

Support is present in most major browsers. Firefox, Chrome and Opera support WebGL by default on Windows, Mac OS X and Linux. Safari also supports WebGL on Mac OS X, but it has to be enabled manually. Internet Explorer does not support WebGL without the use of a plug-in. Both Firefox and Opera provides support for WebGL on Android devices, but there is no support on iPhone or iPad. This means that, for all these browsers, no plug-in has to be installed to run web applications using WebGL. However, relatively new graphics hardware and drivers are required on desktop computers.

A WebGL application can be developed without leaving the familiar web development environment of HTML and JavaScript, all calls to the graphics API are made in JavaScript. This also gives perfect integration with the rest of the web page. WebGL only provides functions to draw 3D objects, the application itself controls positioning and other properties. There is no official development environment, any JavaScript development environment and debugger can be used.

The specification merely states the differences between OpenGL ES 2.0 and WebGL, the rest of the API is identical to OpenGL ES 2.0 and so the documentation for the latter is applicable to WebGL as well. WebGL (and OpenGL in general) has a fairly high learning curve, but there is much help to be found in both books and on the web, so developing a WebGL application does not seem too difficult. Internet Explorer may never support WebGL and this might impact the usage of the standard, but with all other major web browser developers behind it (and the perhaps familiar API), this is likely to become the standard 3D graphics API for web applications if there is ever going to be one.

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
</tr>
</tbody>
</table>

3.4 Plug-ins

The functionality of many web browsers can be extended by the use of plug-ins (discussed in section 2.3) and a web browser can be made to display 3D graphics using this approach. Below are some technologies that do precisely that.

3.4.1 Shockwave

Adobe Shockwave is a platform for creating multimedia applications. It was originally developed for distribution on physical media, but is now also available in web browsers. To view Shockwave content, a user must first install the
Shockwave Player which can be added as a plug-in for Internet Explorer, Firefox and Chrome on Windows and Safari on Mac OS X. The player can use a wide range of Direct3D versions (5.2, 7 or 9), use OpenGL or a software renderer to display the 3D graphics. The Shockwave Player does not support any mobile devices.

Shockwave applications are developed using Adobe Director, a visual development environment that mostly involves setting up animations, sprites and sound, but there is also the possibility to use either JavaScript or a language called *Lingo* to program applications\[11\]. The surrounding web page and 3D objects can be manipulated from the scripting language and a debugger is also included. Director’s visual interface should make it easy to develop an application, but it might also be a limiting factor to what the final application can do. The extensive documentation provided by Adobe, guides available on the web and the many books written about Director should also help in development.

The latest major version of Director was released 2008 and has been followed by minor updates, with the last minor update in 2009\[11\]. Adobe also develops Flash, which is comparable to Shockwave in terms of 3D graphics and web capabilities (Flash is discussed in greater detail in section 3.6). Shockwave has a considerable lower market penetration than Flash (41% of users have Shockwave Player installed versus 99% for Flash Player, \[12\]) and many features that were once exclusive to Shockwave are finding their way into Flash (such as 3D graphics). This seems to suggest that Adobe is focusing resources on Flash instead of Shockwave and that there might not be much of a future for Shockwave.

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Weak</td>
</tr>
</tbody>
</table>

### 3.4.2 Silverlight

Microsoft created Silverlight as a web browser plug-in for running applications written using the .NET framework\[13\]. With the release of version 5, it possible to use Microsoft’s own XNA framework in Silverlight applications. XNA is used in developing hardware accelerated games and multimedia applications on a number of platforms and the 3D graphics part of the framework is almost identical to Direct3D.

Silverlight supports Internet Explorer, Firefox and Chrome on Windows and Safari on Mac OS X. There is no official support for mobile devices except Windows Mobile 7. For desktop applications, XNA is only supported on Windows. Whether this applies to Silverlight applications as well is unclear, but if it is, the 3D graphics parts of Silverlight cannot be used on Mac OS X or any other platform than Windows.
XNA shares many similarities with OpenGL and, just like OpenGL, it only provides a relatively low-level interface to the graphics hardware. Silverlight also includes an API for communication with the surrounding web page and can call JavaScript functions defined on the web page.

Silverlight applications are developed using Visual Studio together with the Silverlight Tools. Developing user interfaces, graphics, and animations is simplified by using a markup language called XAML, for which a graphical editor is included. A debugger is also included with the Silverlight tools and Microsoft publishes an extensive documentation through the Microsoft Developer Network (MSDN).

Beyond the official documentation, there are many tutorials and other sources of information available. With the broad support for different languages that Silverlight has and the developer tools available, it seems to be easy to build applications with Silverlight.

Silverlight is the only framework for developing applications for Windows Phone 7 and will therefore likely be used and supported for a foreseeable future. New minor versions are published at about 5 times a year and major versions are published about once a year. Microsoft, like many other companies, have lately begun focusing on HTML5 and integrating it into products such as Windows 8. Whether Silverlight will remain an important part of Microsoft’s platform therefore remain to be seen.

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Average</td>
</tr>
</tbody>
</table>

### 3.5 Java

Java is a programming language that is compiled into byte-code and can be executed by a virtual machine on a large number of platforms. This virtual machine is included in the Java Runtime Environment (JRE), which supports Windows, Mac OS X, Linux and Solaris. However, an application can be limited to only some of these platforms if it uses a Java library which is not implemented on all of them. Although some mobile devices have previously supported a subset of Java (Java ME), current devices (such as those with the iOS or Android operating system) do not support Java and thus cannot run standard Java applications.

When an end-user installs the JRE, a web browser plug-in is installed as well and through this plug-in Java can be used on a web page. A Java application embedded into a HTML page is called a Java applet. JavaScript can be used to call functions in the Java applet and the applet can call JavaScript functions on
the web page. Supported web browsers are Internet Explorer, Firefox, Chrome, Opera and Safari. Applets also have the ability to use external libraries and these can be downloaded automatically together with the applet.

There is no official development environment for Java, but popular ones include Eclipse and NetBeans. Likewise, there is no debugger included, a debugger interface is built-in to the virtual machine instead and the debugger’s user interface is the responsibility of the development environment.

Java does not have a 3D graphics library built-in, despite an otherwise impressive class library. The major 3D graphics libraries available in Java are presented below and if anything will differ when using a library, such as what platforms are supported, it will be noted.

3.5.1 Java 3D

As a high-level scene graph library, Java 3D allows the programmer to specify how the 3D scene is structured rather than providing functions for drawing 3D graphics directly\[18\]. On Windows, Java3D can use Direct3D or OpenGL and on the other supported platforms, being Mac OS X, Linux and Solaris, OpenGL is used. Java3D supports more advanced graphics concepts such as shaders, which can be written in several shader languages. Beyond 3D graphics, the library also supports playing sounds. However, the size of the library can be prohibitive for some applications, since it weighs in at 18 MB.

Except for the extensive documentation published for the library and a few tutorials, there is not much information available. Judging from the tutorial, the API itself is fairly straightforward and simple, but there is quite a lot of theory surrounding the scene graph to be understood to be able to fully utilise the API. The API also allows applications to access the data in the scene graph, even though the library itself manages the objects in the scene. Java3D is not actively developed, there has been no new releases since 2008 and the developers are currently focusing on a 3D API for JavaFX instead (JavaFX is discussed briefly below)\[19\].

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>Average</td>
<td>Average</td>
<td>Average</td>
<td>Weak</td>
</tr>
</tbody>
</table>

3.5.2 JOGL

JOGL is a Java binding for OpenGL\[20\] currently maintained by the JogAmp organisation. Being an OpenGL binding means that any calls made through JOGL will simply be translated into calls to the OpenGL library on the platform that the application is run on. When bundled with the applet, JOGL will add
about 5 MB to the size of the application and an application that uses JOGL can run on Windows, Mac OS X and Linux.

There is a thorough documentation available and there are also a number of tutorials, although the documentation has not been updated in some areas\[21\]. One of the primary sources of documentation when using JOGL is the official OpenGL documentation, since the API is exposed directly. With the OpenGL API, there is quite a bit of theory that must be learned before it can be used to its full potential, but other than that, it is usually straightforward.

Development is active and JOGL currently supports all OpenGL versions except the latest (version 4.2).

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Good</td>
</tr>
</tbody>
</table>

3.5.3 LWJGL

Just like JOGL, the Lightweight Java Game Library provides direct access to OpenGL from Java\[22\]. It also includes access to a couple of libraries for playing sound and general platform utilities which are useful when writing multimedia applications. Despite having game in its name, the library only provides a simple low-level API for each of these libraries and is thus useful for many types of multimedia applications. LWJGL is an open source project maintained by the LWJGL.org organisation.

Applications written using LWJGL can run on Windows, Mac OS X and Linux, but the library only supports hardware accelerated graphics. The LWJGL libraries adds about 4 MB to the size of the application and is therefore only slightly smaller than JOGL. The documentation is well-written, complete and is comparable in quality to JOGL\[23\]. Additional information is provided through tutorials and judging from them, once a developer is familiar with the OpenGL API, using LWJGL should be easy and straightforward. The OpenGL documentation is of course also a great help when using LWJGL for 3D graphics.

The library is actively developed and supports all OpenGL versions up to and including version 4.2, the latest as of this writing.

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Good</td>
</tr>
</tbody>
</table>
3.6 Flash

Adobe Flash is similar in many respects to Shockwave. Just like Shockwave, applications can be developed using a visual development environment, but in the case of Flash a more traditional programming approach can also be used. Flash applications can run in the web browser using a plug-in called the Flash Player which is available for Internet Explorer, Firefox, Chrome, Opera and Safari on Windows, Mac OS X, Linux and Solaris. A few Blackberry devices are also supported. Adobe has previously provided a version of the Flash Player for Android, but has since discontinued its development, focusing instead on HTML5.

Flash applications are written in a language called ActionScript, which has its roots in JavaScript. Adobe provides both Flash (a development environment that is more visually oriented, like Director) and Flex (a development environment built upon programming using a high level framework). ActionScript applications can call JavaScript embedded in the web page when running in a web browser. A debugger for ActionScript is included with the development environment and there is an extensive documentation library and many resources available on the Internet and in books.

As mentioned, Adobe has cancelled the Flash Player for web browsers on mobile devices such as those running Android, with the intent on focusing on developing HTML5 and native applications instead. On the desktop computer side, a large portion of web browser users have Flash Player installed and it therefore seems as if Flash Player will be present there for a foreseeable future. However, with HTML5 having large parts of the same functionality as the Flash Player (at least for simpler applications) and desktop computers being at least as capable of running HTML5 applications as mobile devices, this strategy could in the future also be implemented on the desktop.

Versions prior to Flash Player 11 did not support hardware accelerated 3D graphics. There are a few libraries for presenting 3D graphics running on these older versions of the Flash Player, but they all used software rendering and thus were slow. Examples of such libraries includes Papervision3D and Sandy 3D, but neither of them is actively maintained any more.

3.6.1 Stage 3D

With the introduction of Flash Player version 11, Adobe added support for hardware accelerated 3D graphics, falling back to software rendering if no 3D graphics hardware is present. The resulting technology is called Stage 3D. Although the Flash Player supports a large number of platforms, hardware accelerated 3D graphics is only present on Windows and Mac OS X and there is no support for 3D graphics in web browsers on mobile devices at this time. Although the
technology is fairly young, many 3D engines for Flash have already migrated to it, including Away3D, Alternativa3D, Flare3D and Yoghurt3D.

The API is similar to that of OpenGL and Direct3D in the sense that it is a low-level graphics API where the application developer specifies what to draw, where and how\[27\]. With the many tutorials and the large documentation provided by Adobe and help from other sources, developing an application using Stage 3D in Flash seems to be fairly easy.

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Good</td>
</tr>
</tbody>
</table>

### 3.7 Markup-based

A slightly less common method of creating 3D graphics applications is through the use of markup languages (as discussed in section 2.2). In general, 3D objects and data are described using a markup language which is then interpreted and displayed by a viewer or a web browser.

#### 3.7.1 3DMLW

3D Markup Language for the Web is a markup language based on XHTML for describing 3D scenes and objects created by 3D Technologies R&D\[28\]. A web browser plug-in developed by the same company must be installed to view the scenes. Within the markup language there is not only support for 2D and 3D graphics, but also for loading 3D models, playing sounds and simulating physics. The plug-in uses OpenGL to deliver hardware accelerated 3D graphics where it is available. Most of the major web browsers are supported (specifically Mozilla, Firefox, Internet Explorer, Safari, Chrome and Opera) on Windows, Mac OS X and Linux, but there is no support for mobile devices.

The documentation is mostly incomplete and those parts with documentation are sparsely documented. It is unclear whether the language can be integrated into a HTML page at all, but documents can be read and programmatically modified using the scripting language Lua. As a markup language it should be editable in almost any text or HTML editor, there is no official development environment. There are no tutorials or examples, which makes it difficult to judge how hard it is to develop an application using 3DMLW.

Overall, the website containing the documentation does not seem to be actively maintained and the latest release of the plug-in was in 2010.
3.7. MARKUP-BASED

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>Average</td>
<td>Weak</td>
<td>Weak</td>
<td>Weak</td>
</tr>
</tbody>
</table>

3.7.2 X3D and X3DOM

Like its predecessor VRML (introduced in section 2.4), X3D is a standard for real-time interactive visualizations based on a markup language. It has been created by the Web3D Consortium, the same organization responsible for the development of VRML. While there are multiple X3D browsers (applications for viewing 3D data written in X3D), there are only a few plug-ins that allow web browsers to display X3D data. As the focus of this comparison is on 3D graphics in web browsers, other types of browsers will not be considered.

Out of the many possibilities to display X3D content, there is only one which match the focus criteria given at the beginning of this chapter, namely X3DOM. X3DOM allows X3D documents to be easily embedded directly into web applications by integrating X3D into the DOM. This allows the web application to read and modify X3D documents using JavaScript. WebGL (or Flash 11 if WebGL is not available) is used to display the 3D graphics, this allows any browser that can support either to display X3D documents and this includes the majority of browsers on almost all platforms. For more information on which platforms and browsers WebGL and Flash 11 support, please see the respective section above. This means that in many cases, no installation of plug-in software is required. An extra JavaScript library has to be delivered together with the application, but it is relatively small, under a megabyte.

There are many third party applications available for editing X3D data and any JavaScript development environment can be used for X3DOM, but there is no official development environment for either. The specification for X3D provides much information about the format itself and there is some information to be found on the web, but not much. X3DOM only provides the most basic documentation for the library on the official website, which makes it hard to assess the difficulty of developing an application using the library.

While X3D is a mature standard, X3DOM overall is not very mature yet as it has not implemented the X3D specification fully and is, according to the developer, an experimental library under active development. The founders of X3DOM have moved on to form the Declarative 3D Community Group to standardise the integration of declarative 3D documents into the DOM, which is what X3DOM currently provides. As such, the future of X3DOM seems uncertain.

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>Good</td>
<td>Weak</td>
<td>Good</td>
<td>Average</td>
</tr>
</tbody>
</table>
3.8 Other Notable Technologies

These technologies are included here only for the purpose of giving the reader a more complete overview of the field of 3D graphics for web applications, they will not be evaluated. Some of these are considered too big for the application judging by the criteria presented at the beginning of this chapter, while others are built upon technologies presented above.

3.8.1 JavaFX

As the name implies, JavaFX is based upon Java and allows the developer to build web applications which are similar to desktop applications in terms of user interface and capabilities\footnote{32}. Version 1.0 of JavaFX used a custom language called JavaFX Script in which graphical user interfaces could be written in a declarative manner (not too unlike a markup language, which is also declarative), but starting with version 2.0, JavaFX is instead distributed as a library for Java. The user only has to have a Java runtime environment installed to run JavaFX applications and the JavaFX library can be bundled with the application.

There is only basic support for 3D graphics in JavaFX, simple 2D shapes such as rectangles can be transformed in 3D space and rendered. More advanced rendering requires an external library, such as any of those previously presented for Java, but support for this type of 3D graphics is planned for a later version of JavaFX.

3.8.2 O3D

Google created O3D as an open source project for providing 3D graphics to applications running in the web browser\footnote{33}. The original project was centred around a plug-in which users had to install if they wanted to use applications built with the library. It has since been replaced by a JavaScript library with the same API, but which is built upon WebGL instead of using a plug-in. The new library therefore requires a web browser that supports WebGL.

3.8.3 Unity

Unity is a game development tool consisting of an editor which is used to create the game and related content, integration with other applications for content creation and a runtime engine which will run the final game or simulation\footnote{34}. There is also a plug-in for web browsers available called Unity Web Player through which the final product can run.

Unity has become a major force on the market for 3D engines in recent years and is thus included here, but since the application this thesis is based upon is
neither a game nor a full-fledged simulation, Unity is not suitable for inclusion in the evaluation.

### 3.8.4 XML3D

Another alternative to the markup languages presented above is XML3D, a standard based on XML[3]. The small amount of documentation that exist about the project states that it can run using any WebGL compatible browser or a custom XML3D browser. The specification, which is at version 0.4 and is currently a working draft, declares that JavaScript can be used for manipulating the data specified in XML3D.

No major project seems to be using it, development appears to have ceased and mailing lists are almost empty. It is therefore included here, instead of in the evaluation above. The creators have instead joined together with the Web3D consortium and others to form the *Declarative 3D Community Group*, which makes the future for XML3D uncertain.

### 3.9 Selecting the Best Technologies

The previous sections of this chapter have provided some basic knowledge about each technology. In table 3.1 all grades with regards to the evaluation criteria have been compiled into one table.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Compat.</th>
<th>Installation</th>
<th>Support</th>
<th>Integration</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DMLW</td>
<td>Average</td>
<td>Average</td>
<td>Weak</td>
<td>Weak</td>
<td>Weak</td>
</tr>
<tr>
<td>Java 3D</td>
<td>Average</td>
<td>Average</td>
<td>Average</td>
<td>Average</td>
<td>Weak</td>
</tr>
<tr>
<td>JOGL</td>
<td>Average</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Good</td>
</tr>
<tr>
<td>LWJGL</td>
<td>Average</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Good</td>
</tr>
<tr>
<td>Shockwave</td>
<td>Weak</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Weak</td>
</tr>
<tr>
<td>Silverlight</td>
<td>Weak</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Average</td>
</tr>
<tr>
<td>Stage 3D</td>
<td>Average</td>
<td>Average</td>
<td>Good</td>
<td>Average</td>
<td>Good</td>
</tr>
<tr>
<td>WebGL</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>X3DOM</td>
<td>Good</td>
<td>Good</td>
<td>Weak</td>
<td>Good</td>
<td>Average</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of strengths and weaknesses of each technology

This table forms the basis for the selection of three technologies to investigate closer. The table clearly shows that both WebGL and X3DOM are good
choices for further evaluation, with JOGL, LWJGL and Stage 3D not far behind. However, X3DOM is particularly weak in terms of development environment and support. JOGL, LWJGL and Stage 3D all have very good developer support and scores only slightly below X3DOM in the other categories (and higher in terms of future) without any major weaknesses compared to X3DOM. They will therefore be considered instead of X3DOM.

Both JOGL and LWJGL serve approximately the same purpose as far as 3D graphics is concerned, namely providing an OpenGL API for Java applications. To get a more interesting and varied evaluation, it would be better to choose one of these and then Stage 3D as the third technology (with WebGL being the first). Since JOGL and LWJGL scored equally, they should be equally good choices for 3D graphics in a Java application. For the purpose of this evaluation, JOGL will be evaluated. The sole reason for this is that JOGL is already used in other projects at SICK and existing knowledge can then be used to develop the demo application further. For any other project, the choice between JOGL and LWJGL would be based upon other reasons and may not give the same result.

The conclusion of this chapter is that the three most suitable technologies for developing web applications with 3D graphics are WebGL, JOGL and Stage 3D. These are the technologies upon which the following chapter will focus using a more practical rather than theoretical evaluation. LWJGL is also a good choice for 3D graphics, but it will not be evaluated further in this thesis.
Chapter 4

Practical Evaluation

A theoretical evaluation, like the one in the previous chapter, serves as a good foundation for choosing a 3D graphics technology to use. However, it is very likely that there are issues which will only present themselves when actually using a technology. The purpose of this chapter is to evaluate those technologies which were selected in the previous chapter (WebGL, JOGL and Stage 3D) by implementing an actual test program in each and try to discover any issues as early as possible.

The test programs will be based upon the features of the demo application and the functionality they are going to test is outlined in the beginning of this chapter. After this overview, each of the three technologies will be evaluated in depth by the criteria put forth in section 3.2. This evaluation will be complemented by the issues encountered and the results from the test programs. The chapter ends with one of the three technologies being chosen and then later used in the development of the demo application, which chapter 5 will cover in detail.

4.1 Chapter Outline

This chapter is divided into a number of sections, one for each evaluation criterion. The aim is to present each technology in depth and highlight the differences between the three technologies with regards to the current criterion within each section. One technology will be chosen at the end of the chapter for use in the development of the demo application. A number of requirements for the application will serve as a basis for this selection.

The primary purpose of the test programs is to test the performance criterion of the evaluation. The integration and communication criteria will also be tested in each test program. Developing the test programs will also provide more detailed information about how the development environment works for each technology, thereby testing the support criterion.

To keep the test programs simple, they will be implementing a subset of the
features of the demo application. They will also implement a number of other features to make the evaluation apply to a broader spectrum of 3D graphics applications. An example of a test application can be seen in figure 4.1. In the test application, the user can rotate and move the 3D model as well as zooming in on details using the three buttons at the bottom of the web page.

![Figure 4.1: One of the three test programs](image)

The demo application will load an image from the web server on the camera and produce a 3D model in the form of a height field (presented in section 2.1.4). At a periodical interval the image will be reloaded from the web server and the height field will be updated to match the new image. This height field will then be drawn on the screen and the user can rotate and move it to get a better look at the details in certain areas.

Each test program will have the functionality outlined below.
4.2. SUPPORTED PLATFORMS

**Drawing detailed objects** An image at the resolution 512x512 pixels will be loaded into the application and drawn as a height field, resulting in a 3D model with 522,242 triangles. The height field will be constantly redrawn on the screen to allow for measuring performance.

**Updating objects** The image from which the height field is generated can be changed by the user. This allows for testing the performance of loading a new image and updating the vertex buffer for the height field.

**Lighting and textures** A texture will be placed on the height field to test performance. The height field will also be lit using a light source.

**Interaction** The user can navigate the height field and rotate the camera by using the mouse.

The test programs will be implemented as one test program for each technology instead of multiple smaller ones, which each would have tested a certain aspect (such as lighting and textures). By doing this, it is possible to get a better idea of the overall performance of a technology in a real application where all these parts are used at the same time.

### 4.2 Supported Platforms

Table 4.1 summarises on what platforms each technology can be used. The data in the table is based on the latest versions of the respective web browsers as of the writing of this thesis. This is especially important in the case of WebGL, which has not been supported in most previous versions of the browsers.

Microsoft has decided to not support WebGL in Internet Explorer because of security concerns. Among the issues is the possibility of a denial of service caused by shaders that take a long time to execute or long draw calls. There are a couple of alternatives for using WebGL in Internet Explorer, see section below.

To use WebGL in Safari on Mac OS X, the user has to enable it manually by first activating a developer console and through that console activate WebGL. This can be difficult to do for regular end users. Safari for Windows does not support WebGL at all. On iOS, Apple’s operating system for the iPhone and iPad, WebGL is only supported in advertisements so far and not in the web browser. On the competing mobile platform Android, WebGL is supported through Opera Mobile and Firefox on newer mobile devices.

Very few mobile platforms have support for running Java applications and neither Android nor iOS can run Java applications through the web browser. Likewise, Flash applications using Stage 3D cannot be run in the web browser of a mobile device running Android or iOS. Flash applications can run on those
# Supported operating systems and web browsers

<table>
<thead>
<tr>
<th>Platform</th>
<th>WebGL</th>
<th>JOGL</th>
<th>Stage 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Firefox</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Chrome</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IE¹</td>
<td>No²</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Opera</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Safari</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mac OS X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Firefox</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Chrome</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Opera</td>
<td>Yes³</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Safari</td>
<td>Yes³</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Linux</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Firefox</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes⁴</td>
</tr>
<tr>
<td>Chrome</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes⁴</td>
</tr>
<tr>
<td>Opera</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes⁴</td>
</tr>
<tr>
<td>Android</td>
<td>Default⁵</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Firefox</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Opera</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>iOS</td>
<td>Safari</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

¹ Internet Explorer  
² Supported via plug-in or Java wrapper library  
³ Requires the user to manually enable WebGL once  
⁴ Only runs in software rendering mode  
⁵ The built-in web browser on Android devices

Table 4.1: Supported operating systems and web browsers
operating systems using the Adobe Integrated Runtime (AIR), but only as standalone applications and not directly through the web browser.

Whether an application written using WebGL, JOGL or Stage 3D can run on a computer depends not only on the combination of operating system and web browser, but also on the hardware of the computer. It is common among the web browser vendors who develop web browsers that support WebGL to disable WebGL when running certain hardware or using certain drivers because of known bugs. In the case of JOGL, the application will only work if the hardware supports the OpenGL version that the application has been written against. Stage 3D has a slight advantage here, since it can also run in software mode independent of hardware, but this will likely be slow.

4.3 Installation

Statistics show that 49.5% of Internet users have a web browser that supports WebGL\[37\]. Flash applications can be used by 95.7% of these users, but only 74.2% have Flash Player 11 installed, which is the only version so far which supports 3D graphics\[38\]. For Java, the equivalent number is 63.4% of Internet users. With the exception of Stage 3D, which can be run in software mode and thus should work on almost any hardware configuration, not all users that can run these types of applications are able to run 3D graphics applications because of unsupported drivers or hardware that is not powerful enough.

The statistics therefore form an upper bound on what percentage of users can run 3D graphics applications built upon one of these technologies. However, the statistics themselves have been collected from a general population of Internet users and may not apply to the users of a certain type of application, such as an expert system.

In a web browser that supports WebGL, no extra software has to be installed for WebGL applications to be able to run. For Internet Explorer, which does not support WebGL, there are a number of alternatives. One alternative is the IEWebGL plug-in which simply provides the WebGL API through a plug-in which the user has to install. The user can also install Google Chrome Frame which runs Google Chrome within Internet Explorer and provides the WebGL functionality to the application.

A developer of a WebGL application can also choose to use a JavaScript library called JebGL\[39\], which provides the same API as WebGL. JebGL translates all WebGL calls to JOGL through a Java applet if the web browser does not support WebGL directly. This should in theory be able to provide support for WebGL in both Internet Explorer and Safari on Windows, which are the only major browsers on desktop computers that do not support WebGL. The disadvantage of this setup is that a Java Runtime Environment has to be installed on
4.4 Development Environment

To keep things as simple as possible, Eclipse was used as the development environment for both the WebGL and JOGL test applications. The Eclipse plug-in JavaScript Development Tools (JSDT) was used for the WebGL application. The JSDT supports syntax highlighting, syntax checking and also some support for refactoring, to the extent possible for a dynamic programming language such as JavaScript. There is no debugger included with the JSDT, instead a debugger built-in to the web browser is generally used and in this case Firebug for Firefox was used. Firebug also includes other tools than just a debugger and makes it easy to debug the application’s integration with the rest of the web page. The Eclipse JSDT development environment is shown in figure 4.2.

Both JOGL and WebGL uses the OpenGL Shading Language (GLSL) as the programming language in which the developer writes shaders. The GLSL is a high-level language influenced mainly by C as is shown by the example in listing 4.1. The example below is a vertex shader used in the WebGL and JOGL applications (with minor differences between the two). It calculates vertex texture
coordinates depending on where the vertex is positioned in the height field, sets lighting and material properties and finally transforms both the vertex normal and position.

```glsl
attribute vec3 position;
attribute vec3 normal;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform float width;
uniform float height;

varying vec2 textureCoord;
varying vec3 fragmentNormal;
varying vec3 light;
varying vec4 diffuseColor;

void main(void) {
    // Since only translations and rotations are used, the normal matrix becomes the upper part of the model-view matrix
    mat3 normalMatrix = mat3(modelViewMatrix);
    light = vec3(0.0, 0.0, 1.0); // points towards the light
    diffuseColor = vec4(vec3(1.0), 1.0);

    fragmentNormal = normalize(normalMatrix * normal);
    textureCoord = vec2(position.x/width + 0.5, 0.5 - position.y/height);
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Listing 4.1: Example of a vertex shader in GLSL

This shader is compiled when the program starts and before anything is rendered to the screen. The shader compiler is part of the WebGL/OpenGL runtime and generally provides good feedback in the form of warnings and error messages, but this is managed by the graphics driver and may thus be different across several hardware configurations. An important limitation in WebGL, which is not found in JOGL, is that vertex buffers can at most hold 65,536 vertices. This is because index buffers can only contain 16-bit indices and can cause problems if large 3D models are used.

Eclipse has very good support for Java development, including code completion, syntax highlighting and advanced refactoring support. The Java debugger runs as a standalone application (the applet viewer) and not in the web browser, although it is possible to connect to a running instance of a Java applet in the web browser. The Java development environment for Java is shown in figure 4.3.

OpenGL 2.0 was used in the JOGL test application, mainly because it is the
CHAPTER 4. PRACTICAL EVALUATION

Figure 4.3: Debugging an application in Eclipse IDE for Java Developers

first OpenGL version with similar functionality from a developer perspective as WebGL and Stage 3D. A later version of OpenGL might have had slightly better performance, but it would not have run on older hardware.

Contrary to Eclipse, which is provided as open source software free of charge, Adobe sells their official Flash development environment as commercial software. Adobe’s development environment for Flash is called Flash Builder and is based upon Eclipse. There are alternative development environments (one example is FlashDevelop, which is open source and free of charge) and support for Flash can also be added to an Eclipse installation. This thesis will only evaluate Flash Builder, the official development environment, shown in figure 4.4.

Flash Builder has syntax highlighting, syntax checking and basic support for refactoring. Its debugger requires a separate runtime to be installed and runs in the web browser. This, once again, makes it easy to debug the application together with the rest of the web page and see how they interact. However, the debugger had a tendency to crash when used in Firefox during the development of the Stage 3D test application.

For shaders, Adobe only provides a runtime environment in the form of a byte-code format. The language that shaders are written in is called AGAL (Adobe Graphics Assembly Language) and is an assembly language, as the name
4.4. DEVELOPMENT ENVIRONMENT

Figure 4.4: Debugging an application in Adobe Flash Builder

suggest. There are libraries such as EasyAGAL and the experimental high-level language PixelBender3D (created by Adobe) that makes developing shaders easier. An example program written using the AGALMiniAssembler as the compiler is displayed in listing 4.2. It performs the same operations as the GLSL shader in listing 4.1.

```plaintext
// Vertex shader constants:
// vc0-3 = modelViewProjectionMatrix
// vc4-7 = normalMatrix
// vc8 = (width, height, 0, 0)
// vc9 = (0.5, 0.5, 0, 0)

mov vt0, vc9 // vt0.zw = 0
div vt0.xy, va0.xy, vc8.xy // vt0.xy = position.xy / size.xy
add vt0.xy, vt0.xy, vc9.xy // vt0.xy = vt0.xy + 0.5
mov v0, vt0 // textureCoordinate = vt0

mov vt1, vc9 // vt1.w = 0
mov vt1.xyz, va1.xyz // vt1.xyz = normal.xyz
m44 vt2, vt1, vc4 // vt2 = normalMatrix * vt1
nrm vt2.xyz, vt2.xyz // vt2.xyz = normalize(vt2.xyz)
mov v1, vt2 // fragment normal = vt2
```
Listing 4.2: Example of a vertex shader in AGAL

Program constants has to be sent in as vertex constants, they cannot be specified directly in the program. An error message is not always displayed if there is an error in the shader code, but rather results in a blank screen being drawn. This often leads to trial and error when trying to find errors. Other compilers might produce better error messages. This, coupled with the very low-level language (especially compared to GLSL), makes it a bit difficult to develop Stage 3D applications.

There are limits on how large vertex buffers, index buffers and textures can grow and there is also a limit to the amount of total memory that can be allocated to each type. For example, 16-bit indices force a maximum size of vertex buffers to be 65,536 and there is also a limit of the amount of indices that can be stored in an index buffer. This makes it difficult to draw larger models, but in return the resulting application will likely be able to run on any computer that can run Flash 11 because of the low requirements on the graphics hardware. This is in sharp contrast to OpenGL, where buffer sizes depend on what the hardware supports, although there are lower limits specified.

For a complete minimal example program written in each technology, please refer to appendices A, B and C.

4.5 Integration and Communication

All three technologies are low-level graphics libraries and the developer directly specifies what 3D models to draw and how to draw them, the latter being done by using shaders (although shaders do not have to be used when using an older version of OpenGL within JOGL). Any information about the 3D models, such as position, is up to the application itself to keep track of. When the model should be drawn, the application uses the position to setup the transformation matrices using the library.

As stated in chapter 3, WebGL applications do not go through a plug-in to integrate with the web page, but rather do so directly since they are written in JavaScript. In the case of Java and Flash, each has its own interface for web page integration.

A Java applet can access the surrounding web page by using the JSObject class. This class allows the applet to call JavaScript functions on the web page. An example, from the JOGL test application, can be seen in listing 4.3.
try {
    counter.onFrameEnded(); // update the counter
    float fps = counter.getSampledFPS();

    // Tell the web page to update the text showing the FPS
    String updateFunction = "updateFPSCounter";
    JSObject window = JSObject.getWindow(JOGLApplet.this);

    // Avoid calling undefined methods until the page has loaded
    if (window != null) {
        if (window.getMember(updateFunction) != null) {
            window.call(updateFunction, new Object[] { fps });
        }
    }
} catch (JSException e) {
}

Listing 4.3: Calling JavaScript from Java

Communication in the other direction, from JavaScript to Java, is much simpler. Any function that is declared public in the Java applet’s main class is accessible as member functions on the DOM object for the Java applet.

The approach taken in ActionScript is similar to the one in Java. Here too, the Flash application can call JavaScript functions on the web page by using the ExternalInterface library. An example is shown in listing 4.4 and has once again been taken from a test application, in this case the Stage 3D test application.

// Update the FPS counter when running on a web page
if (ExternalInterface.available) {
    ExternalInterface.call("updateFPSCounter", this.getSampledFPS());
}

Listing 4.4: Calling JavaScript from ActionScript

Calling functions in the Flash application requires first registering what functions that are available to the web page by using the same ExternalInterface library as before. An example of this is shown in listing 4.5, again from the Stage 3D test program.

// Registers callbacks when running on a web page
if (ExternalInterface.available) {
    ExternalInterface.addCallback("setCurrentTool", this.setCurrentTool);
    ExternalInterface.addCallback("nextImage", this.nextImage);
    ExternalInterface.addCallback("previousImage", this.previousImage);
}
Listing 4.5: Registering JavaScript callbacks in ActionScript

These functions, such as setCurrentTool, can then be called by using the DOM object for the Flash Player, in the same way as with Java. The functions are available as member functions of the object. However, there are some security features of the Flash Player which can block the application from calling JavaScript and vice versa, depending on the origin of each (from which web server they came from).

4.6 Performance

One of the major reasons for doing a practical evaluation is to be able to measure the performance of a program written using one of the three technologies. The purpose of the tests were primarily to see if any of the three performed significantly worse on some combination of software and web browser or if it did not run at all. The results from the performance tests are presented in this section, but the sample size is too small to be considered representative of the technologies in general.

Three computers were used in the performance tests, each representing a different hardware range in terms of performance. The high-end computer was a HP Workstation wx4400 with an NVIDIA Quadro FX 580 graphics processor running Windows 7 Enterprise. An ASUS U46SV with the Intel GMA HD Graphics 3000 graphics processor running Windows 7 Home represented the medium range. For the low-end, a Dell Latitude D420 with the graphics processor Intel GMA 950 and the operating system Windows XP Professional was used.

On the medium range computer, Internet Explorer 9 was used and on the others Internet Explorer 8 was used. This difference is due to updates applied to Windows and what Windows version is available on the computer. For all other browsers, the same versions were used on all three computers; Firefox 11, Chrome 18 and Opera 12 Alpha. Performance was measured in terms of number of frames per second that the application could draw, the memory usage of the application and the quality of the resulting image.

4.6.1 Rendering Performance

The average number of frames per second which was achieved with the respective test application is shown in table 4.2.

As can be seen in the table, WebGL had to be manually enabled in some web browsers because the driver or graphics hardware had been blocked by the web browser developers. With newer graphics drivers and hardware, an end-user will probably not have to do this. Opera had some problems running the WebGL
4.6. PERFORMANCE

<table>
<thead>
<tr>
<th>Technology</th>
<th>Browser</th>
<th>High-end</th>
<th>Medium</th>
<th>Low-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebGL</td>
<td>Firefox</td>
<td>72</td>
<td>55</td>
<td>4(^1)</td>
</tr>
<tr>
<td></td>
<td>Chrome</td>
<td>47</td>
<td>48</td>
<td>5(^1)</td>
</tr>
<tr>
<td></td>
<td>Opera</td>
<td>30(^2)</td>
<td>_3</td>
<td>_3</td>
</tr>
<tr>
<td>JOGL</td>
<td>IE(^4)</td>
<td>60</td>
<td>59</td>
<td>_</td>
</tr>
<tr>
<td></td>
<td>Firefox</td>
<td>47</td>
<td>59</td>
<td>_</td>
</tr>
<tr>
<td></td>
<td>Chrome</td>
<td>60</td>
<td>59</td>
<td>_</td>
</tr>
<tr>
<td></td>
<td>Opera</td>
<td>29</td>
<td>44(^5)</td>
<td>_</td>
</tr>
<tr>
<td>Stage 3D</td>
<td>IE(^4)</td>
<td>41</td>
<td>59</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Firefox</td>
<td>39</td>
<td>59</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Chrome</td>
<td>60</td>
<td>59</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Opera</td>
<td>28</td>
<td>43(^5)</td>
<td>3</td>
</tr>
</tbody>
</table>

1 WebGL had to be enabled manually  
2 The browser did not update the image this often  
3 WebGL could not be enabled in the browser  
4 Internet Explorer  
5 Large FPS variations

Table 4.2: Frames per second

Test program, the number of frames reported was high, but the image on screen was not updated this often and the application felt slow and unresponsive when interacting with it. This, together with the large FPS variations seen in the other test programs might be expected from an alpha version of the browser.

Even though the performance on the low-end system was not impressive, it still managed to run both WebGL and Stage 3D, but was unable to run JOGL because it cannot run OpenGL 2.0 applications. This is notable, because both the WebGL and Stage 3D applications could run on the same computer and all three of the technologies has similar capabilities and functionality. They should therefore also require about as much from the hardware itself. It should be possible to be able to run the JOGL test application on the low-end computer by re-implementing the application using an older version of OpenGL.

A quick comparison of which technology could load an image and update the vertex and index buffers fastest was also carried out. The results were that Flash
was slowest at doing this and WebGL and JOGL were on par. These results are important for the demo application since it is one of the main functions of it.

4.6.2 Memory Usage

Another performance measure is the amount of memory the application uses. This is especially important if a developer wishes to target hardware with limited memory, such as tablets and mobile phones. The maximum amount of memory that each technology used is displayed in table 4.3. These numbers varied quite a bit during execution of each test program and thus may not represent the total maximum amount of memory used by the applications. One of the causes behind this could be that the three languages used (JavaScript, Java and ActionScript) all use garbage-collection as the memory management technique.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Browser</th>
<th>High-end</th>
<th>Medium</th>
<th>Low-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebGL</td>
<td>Firefox</td>
<td>123</td>
<td>169</td>
<td>102(^1)</td>
</tr>
<tr>
<td></td>
<td>Chrome</td>
<td>92</td>
<td>116</td>
<td>135(^1)</td>
</tr>
<tr>
<td></td>
<td>Opera</td>
<td>78</td>
<td>(-^2)</td>
<td>(-^2)</td>
</tr>
<tr>
<td>JOGL</td>
<td>IE(^3)</td>
<td>80</td>
<td>134</td>
<td>(-)</td>
</tr>
<tr>
<td></td>
<td>Firefox</td>
<td>177</td>
<td>233</td>
<td>(-)</td>
</tr>
<tr>
<td></td>
<td>Chrome</td>
<td>129</td>
<td>151</td>
<td>(-)</td>
</tr>
<tr>
<td></td>
<td>Opera</td>
<td>148</td>
<td>134</td>
<td>(-)</td>
</tr>
<tr>
<td>Stage 3D</td>
<td>IE(^3)</td>
<td>87</td>
<td>80</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Firefox</td>
<td>146</td>
<td>171</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>Chrome</td>
<td>84</td>
<td>67</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Opera</td>
<td>143</td>
<td>86</td>
<td>95</td>
</tr>
</tbody>
</table>

\(^1\) WebGL had to be enabled manually  
\(^2\) WebGL could not be enabled in the browser  
\(^3\) Internet Explorer  

Table 4.3: Memory usage (in megabytes)

The high memory usage numbers are to be expected, since a fairly complex 3D model was used. Once again, the web browser used seems to primarily determine how much memory will be used when running the application. Stage 3D had the lowest memory consumption of the technologies, followed by WebGL and then JOGL. The fact that WebGL is built into the web browser should give it an advantage here, since no extra plug-in application has to run beside the web browser, but Stage 3D still utilised less memory.

Both these performance tests show that Adobe has put effort into making sure Stage 3D can run almost everywhere. WebGL and JOGL had multiple instances where they either failed to run or had to be enabled manually, but Stage 3D could run on all computers.
4.6.3 Image Quality

A comparison of the quality of the resulting image is displayed in figure 4.5. The top-most image is of an application implemented using regular OpenGL in C++ developed to serve as a reference image for comparing the image quality of the technologies. Anti-aliasing had to be turned of in the WebGL test application to get a fair comparison, WebGL automatically uses anti-aliasing if it is available.

As can be seen in the figure, both Stage 3D and JOGL are similar in quality to the reference application. The lighting effects implemented in the fragment shader causes visible artefacts in the WebGL application in the form of black lines along the edges of the 3D model. This effect is not visible in any of the other application. Although JOGL and WebGL uses the same shading language (GLSL) and are both related to OpenGL, WebGL is not necessarily implemented using OpenGL and this could explain the image artefacts.

4.7 Future

WebGL is a relatively new standard and its future is somewhat uncertain. Most of the larger web browsers have support for it, but Internet Explorer lacks support and will maybe never support WebGL. In Safari it is deactivated by default, but it might become supported in the future. WebGL is also supported in advertisements on iOS, so it would be very likely that it could become supported in the web browser for iOS some time in the future. WebGL is the only technology out of the three to support web browsers on mobile devices.

As of the writing of this thesis, there are few applications (apart from small demo applications) which use WebGL. One of the few big applications is MapsGL, a version of Google Maps that uses WebGL for extra effects and performance when displaying a map. If more web applications start to use WebGL, it will likely continue to be implemented in future web browsers. The fact that WebGL is an open standard could give it an advantage over any of the other technologies when it comes to built-in support in web browsers.

Contrary to WebGL, there are widely used applications that use JOGL, NASA World Wind is an example of such an application. JOGL is actively developed and will probably continue to be used in applications within a foreseeable future. However, Java applets using JOGL cannot run on popular mobile devices such as iOS and Android. This could limit the audience which can use JOGL-based applications and seeing as these devices are becoming more and more popular, it could become a problem in the future for developers who are looking at targeting the largest possible audience. On the desktop side, the usage of Java-applets has decreased in favour of applications written in HTML and JavaScript.

Just like WebGL, Stage 3D is a relatively new technology with some applications starting to use it, but few large applications are currently using it. Flash
will probably be present on future platforms in the form of Adobe Integrated Runtime (AIR), but it will probably not be possible to run these applications directly in the web browser on these platforms, just as the current situation on Android and iOS. According to Adobe, HTML 5 replaces Flash in the web browser on mobile devices such as Android and iOS. However, the same could be said for web browsers on desktop computers and it would therefore not be impossible that Flash might eventually disappear from the desktop as well.

### 4.8 Application Requirements

As can be seen above, there is no clear winner in all categories. Each technology has its advantages as well as disadvantages. The ultimate choice of which 3D graphics technology to use in a web application will therefore likely depend on the type of application to be developed. The technology which might be considered the best for the application which will be developed as part of this thesis might not be the optimal one for another type of application.

It is therefore important to understand the requirements of this application before choosing technology. Below are those requirements which were decided to be the most important for this type of application, and this list is followed by a description of each requirement.

- Support for mobile devices
- Users should not have install anything
- Can run on a wide range of hardware
- Must run without an Internet connection
- Small library footprint
- Support for Internet Explorer

Recent trends in the targeted user group shows that mobile devices are popular for monitoring systems that use the 3D camera. If the demo application could run on mobile devices, it would be of greater use for the user and possibly also lower the potential cost of purchasing hardware that can run the application, seeing as a desktop computer with sufficient performance to run a 3D application is more expensive than a mobile device with 3D capabilities. The finished product will have a 2D live view that can be used on computers where the 3D capabilities are not available.

The following two items are for the convenience of the user, they should preferably not have to install any extra software or upgrade their hardware to
be able to use the application. This includes installing runtime environments for Java or Flash.

Another important detail is that the application will most likely not be run in an environment where Internet access is present. This means that any libraries which are required to run the application must be included with it. However, the only technology that requires extra libraries is JOGL and these are approximately 5 MB in size. With the limited available space on the 3D camera, this is a drawback for JOGL.

If JebGL is used to provide WebGL support in browsers which do not support WebGL directly, both JOGL and JebGL must be included with the application. Regular WebGL does not require any extra libraries, the needed functionality is built in to the web browser. JebGL requires less than 200 kB of extra libraries itself, but JOGL also has to be included and this will increase the size of the bundled libraries to about 5 MB.

Most customers will probably run Internet Explorer and it is therefore important that this web browser is supported. This requirement is not as strict as the others, a user could easily install another web browser if they wanted to use the application. Installing a new web browser could be seen as equal to installing a Java Runtime Environment or the Flash Player.

4.9 Choice of Technology

The requirement of being able to run the application on mobile devices was considered to be very important. The only technology that is able to run in the web browser of mobile devices is WebGL (Stage 3D can run on mobile devices via AIR, but for this application, the target was the web browser). WebGL also fulfills the requirement of the user not having to install any extra software and no libraries have to be bundled with the application.

The decision was also made to not use JebGL, since the size of the JOGL libraries are prohibitive considering the limited amount of space available on the 3D camera. As previously mentioned, users could also change web browsers if they want to run the application. Another insight was that users who want to run this slightly experimental application would probably be willing to go through the hassle to change to a web browser that supports WebGL.

In the next chapter, the demo application will be presented in more detail as a working application that was built using WebGL.
Figure 4.5: Top to bottom: OpenGL (reference), WebGL, JOGL and Stage 3D
Chapter 5

Demo Application

The evaluation is now over and WebGL has been chosen as the technology to use for the demo application. In this relatively short chapter, the demo application will be presented in more detail than previously, together with a list of the application’s features and a few images of the working application. Also, a few tips for others who wish to develop web applications with 3D graphics components will be presented at the end of the chapter. As previously noted, this chapter will be kept short because some of the functionality and implementation details are confidential.

5.1 Application Features

Upon navigating to the address on which the application is hosted, the user will see the application as it is depicted in figure [5.1]. The application displays the 3D model with the intensity data (black-and-white image, shows the text on the object for example) from the camera as a texture placed upon the model. It is similar in many aspects to the test applications in terms of the user interface, but it has a couple of extra features and was written separately from the test applications. Some of these differences are visible in the user interface and some only affect how the application interacts with the 3D camera.

Images are loaded into the application differently from how they were loaded in the test applications. The demo application will request a new image at regular intervals from the 3D camera. This request is performed asynchronously and when the images has been received, the 3D model will be updated. This allows the user to interact with the model without any interruptions while the image is being received.

Other differences from the test applications of course include more thorough error handling and performance tuning. For example, the demo application reuses index buffers where possible and draws triangle strips instead of triangles and this reduces the amount of memory required to run the application as well as speeds
CHAPTER 5. DEMO APPLICATION

Figure 5.1: The demo application

up rendering.

This allows the application to run at reasonable speed even on the low-end computer used for performance tests in chapter 4, without the camera controls feeling too slow. The performance boost also helps tablets and mobile phones which generally have fairly weak 3D graphics processors compared to desktop computers.

A list of the main features of the application follows below. These are numbered and have been highlighted in figure 5.2 to give a better understanding of how the application works.
1. Height fields of any size can be loaded and drawn. Large height fields will impact performance however.

2. A marker for the three axes in the Cartesian coordinate system shows how the object is rotated compared to the rest of the world.

3. The gradient background does not interfere with the model and gives a clear separation between background and object.

4. By using the pause button, the user can stop the application from requesting another image and proceed to inspect the current model more closely. Once
paused, the button shows a play symbol and pressing it will resume the image request process.

5. The move button allows the user to move the model in the XY-plane with the mouse

6. The user can zoom in and out of the model by going into zoom mode using the zoom button

7. Rotating the model is done by selecting the rotate button and using the mouse to rotate around the X and Z axes

8. Pressing the auto zoom button places the model at the centre of the screen and zooms out just enough so that the whole model is visible on screen

5.2 Lessons learned

The evaluation in the previous chapters is the main objective of this thesis and will be discussed in chapter 6. However, a short discussion of the demo application and its development can be beneficial to others who wish to develop web applications that use WebGL. A few important lessons learned during the development of the application will be presented below.

Loading images and files from other sources

One suggestion was to have the application stored as a web page locally on a computer instead of on the camera itself, because of the limited space available on the camera. WebGL disallows loading images and other resources that does not originate from the same domain as the web page itself. The idea was therefore abandoned.

WebGL utility libraries

Although setting up a web page and creating a WebGL context is relatively simple (especially compared to the same process in a desktop application), a utility library such as webgl-utils.js can make the experience even simpler. It also provides helpful error messages if the user’s browser or hardware does not support WebGL, including links to download a browser with WebGL support. The JavaScript library webgl-debug.js is also a great help when testing and debugging the application. These libraries are both available on the Khronos website for WebGL.

Uploading new vertex and index data

Uploading new data to existing vertex and index buffers is faster than re-creating the same buffers. This may seem evident, but in the JOGL test application, the speed difference between updating and re-creating the
buffers generated from the height map was negligible. In WebGL however, it took a second or more to re-create the buffer compared to uploading new data into an existing buffer on the fastest of the three computers used in the performance tests.

**Luminance textures**

The intensity texture which covers the 3D model was originally uploaded as a luminance texture to the graphics processor. On some computers, the surface of the 3D model became completely white instead of showing the texture. The same thing happened when using Opera as the web browser, independent of the hardware used. The solution was to use RGB textures instead.
Chapter 6

Discussion and Conclusions

In this final chapter, the evaluation in chapters 3 and 4 will be discussed at a greater length than previously. The aim is to identify possible shortcomings of the evaluation and cover alternative approaches to the evaluation. After this discussion, the most important conclusions to take away from the thesis will be presented. The chapter (and the thesis) ends with an outlook on what the future of 3D technologies for web applications might be and a few suggestions for future work in this area.

6.1 Comments on the Evaluation

Possible criticism of the evaluation is presented and responded to item-by-item in the list below.

The evaluation criteria
Finding good criteria for the evaluation is difficult and finding one that will cover every possible use is almost impossible. The criteria laid out in chapter 4 has proven to be good enough at least for the demo application presented in chapter 5. Other applications may need different criteria, but as pointed out earlier, effort has gone into making all of the criteria as general as possible without making them lose their usefulness.

The future criterion is weak
The future criterion is the hardest to measure of all the criteria, it is only possible to make an educated guess of how a technology will stand the test of time. This guess could be very wrong and still the criteria is given quite a lot of weight in the evaluation. The idea behind this criterion is to identify technologies that absolutely do not have a future, rather than guessing exactly what will happen to it in the future. Without it, a technology which obviously does not have a future might end up as one of the chosen technologies or even the chosen technology. If this happened, the result of
this thesis might become almost useless and the demo application would not have much of a future either.

**The scale “good”, “average” and “weak” is lacking in resolution**

The scale has been designed to be as simple as possible. It could be likened to a scale of “yes”, “no” and “neither”. If two extra levels (or more) were introduced it would likely make the scale more subjective since it would be harder to decide whether, for example, the platform support of WebGL should be considered “good” or “very good” even with the same measurements as now that are meant to be objective. The current scale also made it easier to select the three best technologies at the end of chapter 3, with more levels it would be more difficult to judge which technologies to select for further evaluation.

**No operating systems other than Windows in the tests**

Only computers running the Windows operating system was included in the performance tests. Even though this may be the most widely used operating system, the performance of the applications is important when running on other operating systems and on mobile devices also. This is mainly because of time and resource (in the form of additional computers running other operating systems) constraints and although the performance criterion is not among the most important criteria, this is definitely a part of the thesis that could have been improved.

**Some sources used in this thesis lack academic rigour**

It is true that this thesis builds upon much information from web sites and that these cannot be held to the same high standard as academic literature. However, information has been taken from sources such as academic articles whenever possible. The most important thing about the evaluation itself is that it has to be up-to-date, otherwise it is useless. An evaluation that uses decade old books will probably not come to the same conclusions as this thesis, not because it uses a more accepted form of sources, but rather that the information is outdated and developments in this field are happening very rapidly. This also makes it hard to find books and similar sources that have current information.

### 6.2 Conclusions

All of the three technologies that were investigated in the more in-depth evaluation were similar not only in their performance, but also in the resulting image quality, how their API was designed and other aspects. This is a good thing, it means that all three are competent technologies and the choice of which one to use will instead be decided by the requirements of the application that will
be built and the current knowledge in terms of tools and technologies of the de-
velopers behind it. With some kind of background in or knowledge of computer
graphics, it should be fairly easy to start using any of the three technologies, they
all map onto common computer graphics concepts.

It is possible today, on modern hardware, to write graphics-intensive applica-
tions completely in a scripting language, without having to extract performance
sensitive code into modules that run directly on the hardware. The performance
optimizations performed by the Java virtual machine and the JavaScript inter-
preters probably make a big difference here, but Flash was also able to run the
test application with a good frame rate.

Developing an application as a web application can allow it to run on wider
array of platforms than what would have been possible if it had been written as
a normal desktop application. The demo application in this thesis is an example
of this. When the thesis started out, there were no real plans on using the ap-
plication on tablets or mobile phones, but when the opportunity arose it became
an important part of the application. However, with the requirements on the
graphics hardware that applications of this type have, they might be limited on
which platforms and hardware that they can run compared to other types of web
applications.

Finally, although there are many alternatives for realising 3D graphics in web
applications, not all of them are production-ready (meaning that software to
be delivered to a customer can be written using them). This area seems to be
a popular one for people to work on in the form of hobby and research projects
and hardly any of these will ever reach a level of quality where they can be used
in end-user software. Examples shown in this thesis include the library X3DOM
and the markup-language XML3D.

6.3 Future of 3D Graphics on the Web

Since 3D graphics has not been available in web applications up until very re-
cently, the web has mostly attracted applications which are not dependent on 3D
graphics to work. It is therefore not so much of a question of what current ap-
plications might benefit from these new technologies, but rather what new kinds
of applications will find their way onto the web using them.

Not all applications can benefit from using 3D graphics. One of the main
questions that will likely determine whether this usage of 3D graphics will be-
come successful is whether there are enough applications that can benefit. With
a number of other services, such as audio, becoming available to JavaScript de-
velopers in the near future, the web might become a more attractive platform for
more and more developers, especially those working on multimedia applications.

Although many browsers already support WebGL, there is still a need for more
CHAPTER 6. DISCUSSION AND CONCLUSIONS

WebGL applications if the standard is going to survive. Since WebGL is based upon OpenGL, which has had a long time both to mature and prove itself, the likelihood of another standard replacing WebGL is currently very low unless the new API has some revolutionary features. Microsoft may reverse their position on WebGL if more applications start to use it or they may try to create their own API as a competitor (akin to DirectX or XNA). They no longer have the web browser market share they once had however and may not be very successful if they create a new API, since they need the support of both web developers and the other web browser developers. There is also the possibility that Internet Explorer may never support WebGL, but if the number of WebGL applications continues to grow, this might hurt their browser market share.

Java (and JOGL in particular) will probably continue to exist on desktop computers, but will not make onto future platforms and especially not future mobile devices. There is currently no Java Runtime Environment in wide use on neither Android nor iOS and these platforms currently seem to be among the fastest growing mobile platforms in terms of users. Stage 3D on the other hand, will likely become a widely used technology on both mobile devices and desktop computers through the use of AIR. Chances are that it will not be as successful embedded into web pages through the Flash Player plug-in, because of Adobe’s stance on HTML 5.

Markup languages can provide an easy to use programming environment for simpler 3D applications, especially those which only provide a 3D scene for the user to move around in and perform basic interactions. As discussed in chapter 3, a couple of independent groups have come together to form a new standard for a 3D graphics markup language and it may eventually be integrated into the web browser in the same way WebGL has been integrated, assuming the new standard gains traction. WebGL could then be provided for applications using more advanced 3D graphics and the markup language would provide 3D scenes for the simpler ones.

Specialised web browser plug-ins are likely still going to be used in the future, but just like today, almost none of them will have a large user base. They will provide the functionality not offered by others (such as WebGL). An example of such a plug-in could be one that provided 3D graphics, sound and input from multiple device types in one high-level package. However, fewer of these will probably be developed in the future compared to the present considering that there is now a standardised API for 3D graphics in web applications.

6.4 Future Work

VRML was standardised almost two decades ago (1994) and many technologies have come and gone since then. Thanks to today’s large market penetration of
6.4. FUTURE WORK

high-performance graphics processors in devices ranging from desktop computers to mobile phones, it is now more viable to introduce 3D graphics to the web than ever before. In terms of applications, this area is still in its infancy and there are many new exciting possibilities. Below are some suggestions for future work within this area.

- How does WebGL interact with upcoming multimedia standards such as audio? Can any type of complex multimedia application be ported to web using these?
- Would WebGL also be a good choice for more interactive applications, such as games? Is the slow-down due to garbage collection a big issue?
- What types of applications can be built on top of 3D scene markup languages and what benefits do these offer?
Bibliography


[37] Alexis Deveria. When can I use... support tables for HTML5, CSS3, etc. \url{http://caniuse.com/}, 2012. Accessed 25-Apr-2012. [cited at p. 33]


Appendices
<doctype html>
<html>
<head>
<meta http-equiv="Content-Type"
content="text/html;charset=UTF-8">
<title>WebGL Example</title>
<!-- This example program uses jQuery and webgl-utils -->
<script type="text/javascript" src="jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="webgl-utils.js"></script>
<script type="x-shader/x-vertex" id="vertex-shader">
attribute vec3 position;

void main(void) {
    gl_Position = vec4(position, 1.0);
}
</script>
<script type="x-shader/x-fragment" id="fragment-shader">
precision mediump float;

void main(void) {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
</script>
<script type="text/javascript">
// Variables
var gl = null;

// Creates a WebGL program from the shaders given above
function createProgram() {
    var vs = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vs, ($('#vertex-shader').text()));
    gl.compileShader(vs);

    var fs = gl.createShader(gl.FRAGMENT_SHADER);
APPENDIX A. WEBGL EXAMPLE PROGRAM

```javascript
// Creates and uploads data to the vertex buffer
function createBuffer() {
    var vertices = new Float32Array([ -0.5, -0.5, 0, 0.5, -0.5, 0, 0, 0.5, 0 ]);  
    var vb = gl.createBuffer(gl.ARRAY_BUFFER);
    gl.bindBuffer(gl.ARRAY_BUFFER, vb);
    gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
    gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 0, 0);
}

$(document).ready(function () {
    // Initialize WebGL
    var canvas = $('#webgl-canvas')[0];  
    // gl = canvas.getContext("webgl");
    gl = WebGLUtils.setupWebGL(canvas);
    if (!gl) return;

    gl.viewport(0, 0, canvas.width, canvas.height);
    gl.enable(gl.DEPTH_TEST);
    gl.depthFunc(gl.LEQUAL);
    gl.clearColor(0, 0, 0, 1.0);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

    // Load shaders and create the 3D model
    createProgram();
    createBuffer();

    // Will be regularly called to redraw the scene
    function drawCallback() {
        gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
        gl.drawArrays(gl.TRIANGLES, 0, 3);
        requestAnimFrame(drawCallback); // draw again
    }

    requestAnimFrame(drawCallback); // start drawing
```

});
</script>
</head>
<body>
<canvas id="webgl-canvas" width="512" height="512"></canvas>
</body>
</html>
Appendix B

JOGL Example Program

```java
import java.applet.Applet;
import java.awt.BorderLayout;
import java.nio.FloatBuffer;
import java.nio.IntBuffer;
import java.media.opengl.GL;
import java.media.opengl.GL2;
import java.media.opengl.GLAutoDrawable;
import java.media.opengl.GLCapabilities;
import java.media.opengl.GLEventListener;
import java.media.opengl.GLProfile;
import java.media.opengl.awt.GLCanvas;
import com.jogamp.opengl.util.Animator;
import com.jogamp.opengl.util.GLBuffers;
import com.jogang.opengl.util.glsl.ShaderUtil;

public class JOGLExample extends Applet {

    private static final long serialVersionUID = 1L;

    // OpenGL objects
    private int vertexBuffer = -1;
    private int vertexShader = -1;
    private int fragmentShader = -1;
    private int program = -1;

    // Interfaces the rendering with JOGL
    private class Renderer implements GLEventListener {

        @Override
        public void reshape(GLAutoDrawable d, int x, int y, int w, int h) {
            // glViewport has already been called
        }
    }
}
```
// Creates all OpenGL objects and the 3D model
@Override
public void init(GLAutoDrawable drawable) {
    GL2 gl = getGL(drawable);

    gl.glEnable(GL.GL_DEPTH_TEST);
    gl.glDepthFunc(GL.GL_LEQUAL);
    gl.glClearColor(0, 0, 0, 1.0f);

    createProgram(gl);
    createBuffer(gl);
}

// Called when OpenGL objects should be deleted
@Override
public void dispose(GLAutoDrawable drawable) {
    GL2 gl = getGL(drawable);

    // Delete buffers
    int[] buffers = { vertexBuffer };  // Add other buffers if needed
    gl.glDeleteBuffers(1, buffers, 0);

    // Delete programs and shaders
    gl.glDetachShader(program, vertexShader);
    gl.glDetachShader(program, fragmentShader);
    gl.glDeleteShader(vertexShader);
    gl.glDeleteShader(fragmentShader);
    gl.glDeleteProgram(program);
}

// Draws the 3D model on screen
@Override
public void display(GLAutoDrawable drawable) {
    GL2 gl = getGL(drawable);

    gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
    gl.glDrawArrays(GL.GL_TRIANGLES, 0, 3);
}

// Called on applet start up, initializes JOGL
@Override
public void init() {
    GLProfile profile = GLProfile.getDefault();
    GLCapabilities capabilities = new GLCapabilities(profile);
    GLCanvas canvas = new GLCanvas(capabilities);
    canvas.addGLEventListener(new Renderer());

    this.setLayout(new BorderLayout());
    this.add(canvas, BorderLayout.CENTER);

    Animator animator = new Animator();
`animator.add(canvas);
animator.setRunAsFastAsPossible(true);
animator.setPrintExceptions(true);
animator.start();`  

// Returns an OpenGL 2 context from the given drawable  
`private GL2 getGL(GLAutoDrawable drawable) {  
  return drawable.getGL().getGL2();`  

// Create the GLSL program used when rendering the model  
`private void createProgram(GL2 gl) {  
  // Create the vertex shader  
  IntBuffer vertexShader = GLBuffers.newDirectIntBuffer(1);  
  String[][] vertexSource = {  
    new StringBuilder()  
      .append("#version 120\n")  
      .append("attribute vec3 position;\n")  
      .append("void main(void) {\n")  
      .append("  gl_Position = vec4(position, 1.0);\n")  
      .append("}\n").toString()  
  };  
  ShaderUtil.createAndCompileShader(gl, vertexShader,  
    GL2.GL_VERTEX_SHADER, vertexSource, System.err);  
  this.vertexShader = vertexShader.get(0);  
  // Create the fragment shader  
  IntBuffer fragmentShader = GLBuffers.newDirectIntBuffer(1);  
  String[][] fragmentSource = {  
    new StringBuilder()  
      .append("#version 120\n")  
      .append("precision highp float;\n")  
      .append("void main(void) {\n")  
      .append("  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n")  
      .append("}\n").toString()  
  };  
  ShaderUtil.createAndCompileShader(gl, fragmentShader,  
    GL2.GL_FRAGMENT_SHADER, fragmentSource, System.err);  
  this.fragmentShader = fragmentShader.get(0);  
  // Link them to form one program  
  this.program = gl.glCreateProgram();  
  gl.glAttachShader(this.program, vertexShader.get(0));  
  gl.glAttachShader(this.program, fragmentShader.get(0));  
  gl.glBindAttribLocation(this.program, 0, "position");  
  gl.glLinkProgram(this.program);  
  System.err.println(ShaderUtil.getProgramInfoLog(gl, this.program));  
  gl.glUseProgram(this.program);  
  gl.glEnableVertexAttribArray(0);`
private void createBuffer(GL2 gl) {
    float vertices[] = {
        -0.5f, -0.5f, 0,
        0.5f, -0.5f, 0,
        0, 0.5f, 0
    };

    long size = vertices.length * GLBuffers.SIZEOF_FLOAT;
    FloatBuffer vb = GLBuffers.newDirectFloatBuffer(vertices);
    vb.rewind();

    int buffers[] = {0};
    gl.glGenBuffers(1, buffers, 0);
    gl.glBindBuffer(GL.GL_ARRAY_BUFFER, buffers[0]);
    gl.glBufferData(GL.GL_ARRAY_BUFFER, size, vb, GL.GL_STATIC_DRAW);
    gl.glVertexAttribPointer(0, 3, GL.GL_FLOAT, false, 0, 0);
    this.vertexBuffer = buffers[0];
}
}
Appendix C

Stage 3D Example Program

// This program uses AGALMiniAssembler.as, available from Adobe at:
package
{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.display.Loader;
  import flash.display.LoaderInfo;
  import flash.display.Sprite;
  import flash.display.Stage;
  import flash.display.Stage3D;
  import flash.display3D.Context3D;
  import flash.display3D.Context3DCompareMode;
  import flash.display3D.Context3DProgramType;
  import flash.display3D.Context3DTextureFormat;
  import flash.display3D.IndexBuffer3D;
  import flash.display3D.Program3D;
  import flash.display3D.VertexBuffer3D;
  import flash.display3D.textures.Texture;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.external.ExternalInterface;
  import flash.geom.Matrix;
  import flash.geom.Matrix3D;
  import flash.geom.Vector3D;
  import flash.net.URLRequest;
  import flash.utils.getTimer;

  [SWF(width="512", height="512", frameRate="60",
      backgroundColor="#FFFFFF")]
public class Stage3DExample extends Sprite
{
  // Graphics objects
  private var vertexBuffer: VertexBuffer3D = null;
  private var indexBuffer: IndexBuffer3D = null;
private var program: Program3D = null;

// Constructor, called upon startup
public function Stage3DExample(): void {
    this.stage.addChild(this);

    // Create the 3D context
    var stage3D: Stage3D = this.getStage3D();
    stage3D.addEventListener(Event.CONTEXT3D_CREATE, this.onContextCreate);
    stage3D.requestContext3D();
}

// Returns the Stage3D object
private function getStage3D(): Stage3D {
    return this.stage.stage3Ds[0];
}

// Called when a 3D context has been created
private function onContextCreate(e: Event): void {
    // Configure the 3D context
    var context: Context3D = this.getContext();
    const ANTIALIAS_LEVEL: int = 0; // no anti-aliasing
    context.configureBackBuffer(stage.stageWidth, stage.stageHeight, ANTIALIAS_LEVEL);
    context.setDepthTest(true, Context3DCompareMode.LESS_EQUAL);
    context.enableErrorChecking = false;

    // Clean up from previous context
    this.indexBuffer = null;
    this.vertexBuffer = null;

    // Create programs and the 3D model
    this.program = this.createProgram();
    this.createBuffers();

    this.addEventListener(Event.ENTER_FRAME, this.onRender);
}

// Returns the 3D context used in all 3D operations
private function getContext(): Context3D {
    return this.getStage3D().context3D;
}

// Creates the AGAL program used when rendering the model
private function createProgram(): Program3D {

```javascript
const SHADER_DEBUGGING: Boolean = false;

// Create the vertex shader
var vertexSource: String = "mov op, va0";
var vertexAssembler: AGALMiniAssembler =
    new AGALMiniAssembler(SHADER_DEBUGGING);
vertexAssembler.assemble(Context3DProgramType.VERTEX,
    vertexSource);

// Create the fragment shader
var fragmentSource: String = "mov oc, fc0";
var fragmentAssembler: AGALMiniAssembler =
    new AGALMiniAssembler(SHADER_DEBUGGING);
fragmentAssembler.assemble(Context3DProgramType.FRAGMENT,
    fragmentSource);

// Link them together to form one program
var program: Program3D = this.getContext().createProgram();
program.upload(vertexAssembler.agalcode,
    fragmentAssembler.agalcode);
return program;

// Called when the height map has been loaded, updates buffers
private function createBuffers(): void {
const VERTEX_SIZE: int = 3;
var vertices: Vector.<Number> = new <Number>[
    -0.5, -0.5, 0, 0.5, -0.5, 0, 0, 0.5, 0
];

var numVertices: int = vertices.length / VERTEX_SIZE;
this.vertexBuffer = this.getContext().createVertexBuffer{
    numVertices, VERTEX_SIZE};
this.vertexBuffer.uploadFromVector(vertices, 0, numVertices);

var indices: Vector.<uint> = new <uint>[0, 1, 2];
this.indexBuffer = this.getContext().createIndexBuffer{
    indices.length};
this.indexBuffer.uploadFromVector(indices, 0,
    indices.length);
}

// Called when a new frame should be rendered
private function onRender(e: Event): void {
    var context: Context3D = this.getContext();
    if (!context)
```

APPENDIX C. STAGE 3D EXAMPLE PROGRAM

// Context has been lost, request a new one
// and reload resources
this.getStage3D().requestContext3D();
return;
}

try
{
  this.renderFrame(context);
}
catch (e: Error)
{
  // Recreate the context when it has been lost
  this.getStage3D().requestContext3D();
}
}

// Renders a frame
private function renderFrame(context: Context3D): void
{
  context.clear(0, 0, 0, 1);

  // Set drawing parameters
  const WHITE: Vector.<Number> = new <Number>[1, 1, 1, 1];
  const FS: String = Context3DProgramType.FRAGMENT;

  context.setProgram(this.program);
  context.setProgramConstantsFromVector(FS, 0, WHITE); // fc0

  // Draw vertices from all buffers
  context.setVertexBufferAt(0, this.vertexBuffer, 0,
    Context3DVertexBufferFormat.FLOAT_3);
  context.drawTriangles(this.indexBuffer, 0, 1);

  context.present();
}
List of Figures

2.1 The 3D graphics pipeline ................................. 9
2.2 (a) The height field (b) A plane (c) Resulting 3D model 12

4.1 One of the three test programs .......................... 30
4.2 Developing an application in Eclipse JSDT ............... 34
4.3 Debugging an application in Eclipse IDE for Java Developers 36
4.4 Debugging an application in Adobe Flash Builder .......... 37
4.5 Top to bottom: OpenGL (reference), WebGL, JOGL and Stage 3D 46

5.1 The demo application ....................................... 48
5.2 Demo application with features highlighted .............. 49
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary of strengths and weaknesses of each technology</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Supported operating systems and web browsers</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Frames per second</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>Memory usage (in megabytes)</td>
<td>42</td>
</tr>
</tbody>
</table>
Listings

2.1 Example of a markup language: a HTML document . . . . . . . . . . . 13
3.1 Example of a vertex shader in GLSL . . . . . . . . . . . . . . . . . . . 35
4.2 Example of a vertex shader in AGAL . . . . . . . . . . . . . . . . . . . 37
4.3 Calling JavaScript from Java . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.4 Calling JavaScript from ActionScript . . . . . . . . . . . . . . . . . . . . 39
4.5 Registering JavaScript callbacks in ActionScript . . . . . . . . . . . . . 39