Björn Nordström, Vincent Thuning

Adwall

Visualization of data in OpenGL

Computer Science
C-level thesis

Date/Term: 2012-06-05
Supervisor: Thijs Jan Holleboom
Examiner: Donald F. Ross
Serial Number: C2012:07
Björn Nordström, Vincent Thuning

Adwall
Visualization of data stream in OpenGL
Adwall
Visualization of data stream in OpenGL

Björn Nordström, Vincent Thuning

© 2012 The author and Karlstad University
This thesis is submitted in partial fulfillment of the requirements for the Bachelors degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Björn Nordström, Vincent Thunin

Approved, 2012-06-05

Opponent: Anton Danielsson, Niklas Kling

Advisor: Thijs Jan Holleboom

Examiner: Donald F. Ross
Abstract

This thesis describes the problems and solutions of visualizing data in a real time environment. The implemented system consists of three parts, a view, controller and fetcher. These parts are loosely connected by the Model-View-Controller pattern. The performance aspect of the view will be addressed, as of the real time requirement of the visualization. The topics that the performance aspect leads to are; OpenGL optimizations and the feasibility of using the interpreted programming language Python for real time rendering. The OpenGL optimization topic examines different rendering techniques; how they are implemented and how they perform. The role of the controller is to synchronize and organize the data over a number of views. This thesis will discuss how this was achieved with the publisher-subscribe pattern. The data source in the implementation of the project is the advertisement website Blocket.se. Blocket.se are also the customer of the system.
Acknowledgements

This project has been a great experience. We have received invaluable help from our supervisor Thijs Holleboom who we would like to thank. We would also like to thank Blocket.se and Daniel Melani especially for the opportunity.
Contents

1 Introduction 1
  1.1 The task ................................................. 1
    1.1.1 Background ........................................ 1
    1.1.2 Purpose ............................................. 2
  1.2 Outline ................................................. 2

2 Background 5
  2.1 Background ............................................ 5
  2.2 Previous system ....................................... 6
    2.2.1 Fetcher ............................................. 6
    2.2.2 Controller ......................................... 6
    2.2.3 View ................................................ 7
  2.3 Summary ................................................ 9

3 Tools 11
  3.1 Python ................................................. 11
  3.2 Blocket API ........................................... 12
  3.3 ØMQ library ............................................. 12
  3.4 OpenGL ................................................ 14
    3.4.1 2D versus 3D ....................................... 15

4 OpenGL and 3D 17
  4.1 OpenGL mesh rendering ............................... 17
    4.1.1 Rendering techniques .............................. 18
6  Results and evaluation

6.1  OpenGL tests

   6.1.1  Test with single object
   6.1.2  Test with multiple objects
   6.1.3  Test of scalability

6.2  Summary

7  Conclusions

7.1  Implementation

7.2  Performance problem

   7.2.1  Tests of rendering techniques

7.3  Final product

   7.3.1  Animation system & animations
   7.3.2  Integration
   7.3.3  Summary of the implementation

7.4  The project

   7.4.1  The beginning of the project
   7.4.2  Middle of the project
   7.4.3  End of the project

7.5  Suggestions for improvement

7.6  Summary of the project

References

A  Test application
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>New ad animation</td>
</tr>
<tr>
<td>2.1</td>
<td>Picture of the Adwall system</td>
</tr>
<tr>
<td>4.1</td>
<td>Quad</td>
</tr>
<tr>
<td>4.2</td>
<td>Image of frustum, [13]</td>
</tr>
<tr>
<td>5.1</td>
<td>Objective-C View</td>
</tr>
<tr>
<td>5.2</td>
<td>Replacing middle ad with shaking animation</td>
</tr>
<tr>
<td>5.3</td>
<td>Diagram over the movement of the falling ad animation</td>
</tr>
<tr>
<td>5.4</td>
<td>Filling grid with falling ads</td>
</tr>
<tr>
<td>5.5</td>
<td>Camera</td>
</tr>
<tr>
<td>5.6</td>
<td>HSL [14]</td>
</tr>
<tr>
<td>5.7</td>
<td>Quad</td>
</tr>
<tr>
<td>6.1</td>
<td>Graph of test results rendering a single object</td>
</tr>
<tr>
<td>6.2</td>
<td>Graph of test results rendering multiple objects</td>
</tr>
<tr>
<td>6.3</td>
<td>OpenGL test: single object</td>
</tr>
<tr>
<td>6.4</td>
<td>OpenGL test: multiple objects</td>
</tr>
</tbody>
</table>
List of Tables

6.1 Table of test results rendering a single object. ........................................... 53
6.2 Table of test results rendering multiple objects. ................................. 53
Listings

3.1 server.py ................................................................. 13
3.2 client.py ................................................................. 14
5.1 Quadratic shape ......................................................... 28
5.2 RGB convert ............................................................. 39
5.3 Measure time ............................................................ 42
5.4 glBegin ................................................................. 43
5.5 display list ............................................................... 44
5.6 VBO ................................................................. 45
5.7 Vertex Array ............................................................. 46
5.8 Render array ............................................................. 47
A.1 prestandatest.py .......................................................... 69
A.2 tester.py ................................................................. 69
Chapter 1

Introduction

This chapter presents an introduction to the project. It consists of a short background on the subject, the task and an outline of the thesis.

1.1 The task

1.1.1 Background

As computers become more powerful the field of use expands. The computers have not only become more powerful in the aspect of the CPU (Central Processing Unit), but also in the aspect of graphics calculations in the GPU (Graphics Processing Unit). The GPU has also lately been discovered to be able to compute calculations in areas other than graphics, for example physics and medical science [5]. The use of computers to animate and visualize increases [1].

Computers can handle and process large amount of data. Presentation of data can be done in several ways: text, graphs, numbers etc.

The data source may vary, it may be static data that wont change or varying data which may change. Depending on the type of data and source some presentation methods may be discarded. This is due to the fact that a human is unable to process data in certain rates and forms.
1.1.2 Purpose

The goal of the project is to develop an application which receives data from blocket.se. Blocket.se is a company which hosts advertisements for both private commercial products. Users can advertise their products for a fee. An advertisement may include pictures. The application should receive images from the blocket.se database and render them on the screen.

The advertisements images will be shown as they are added to the database. The new image will replace an old image with the same color. To highlight images which are new an animation will affect the new image and the surrounding images. Shown in figure 1.1.

A delay is added if the rate of images is unmanageable. The delays are both for the application and the observer. The application can only handle certain amount of animations.

1.2 Outline

Chapter 2: Background The background of the project, existing implementation and further development. How the existing implementation works and what changes that has to be done will be thoroughly explained.

Chapter 3: Tools A review of the tools used in the project. A motivation for each tool
and technique will be discussed. Also small examples that will explain how to use the tools.

**Chapter 4: OpenGL and 3D** In depth analysis of OpenGL and 3D. This chapter will introduce the basics of OpenGL and 3D to the reader. Code snippets which show different techniques will be explained. The chapter also explain the math which the project rely on, such as the equations to place object at correct positions in a 3D space.

**Chapter 5: Experiment and implementation** The implementation of both the system and rendering techniques that was used with OpenGL, as well as tests of the rendering techniques.

**Chapter 6: Results and evaluation** Presentation and evaluation of the results from the tests of rendering techniques.

**Chapter 7: Conclusions** The conclusions drawn from the tests, the implementation and the project as whole.
Chapter 2

Background

In this chapter we will provide an overview of the background of the project. The task of the project and different approaches will be discussed.

2.1 Background

The purpose of this project is to develop a system to visualize advertisements hosted on Blocket.se. Blocket.se is a website that hosts advertisements, many of the advertisements that are hosted on Blocket.se have an image which makes it ideal for such a product. The result will be a system that can show and animate the images from the advertisements. The product will make it easier for Blocket.se to show visitors of the office, how many advertisements that Blocket.se hosts. The final product will show, in total around 6000 to 15000 images in a grid layout and it will be located in the Blocket.se office in Stockholm.

The purpose of the animations is to highlight new advertisements. We will create three categories of animations; one that will fill the screens, one that will animate when the screens are filled and a new advertisement is replacing an old, and also we will create an animation to remove all the ads and clear the screen. We will start to create the animation that replaces advertisements since Blocket.se has expressed the largest interest towards it.


2.2 Previous system

The system running at the Blocket.se headquarter is a simple visualization of incoming advertisements. It shows the ads in a grid layout and whenever a new ad has been received, it replaces an already existing ad if the location is not free. The most basic functionality is already implemented and running at the headquarter of Blocket.se, their wish however is for this system to be able to handle more complex animations.

The system is composed of three parts, a fetcher, a controller and several views, see figure 2.1. We will use the fetcher and redo the controller and the view. The hardware that the system uses are 3 Apple-computers with 6 TV screens. Each computer will control two screens. The communication between the parts are through network. The network communication will be implemented with a library called ZeroMQ.

2.2.1 Fetcher

The fetcher is responsible for downloading data from Blocket.se and forward the data to the controller. It uses the blocket Python API 3.2, which can be used by developers, registered on the Blocket.se API program. The fetcher is configured to ask the Blocket.se database for advertisements that recently have been hosted on Blocket.se. This part we will not change since its task is simple and does not need to be extended.

2.2.2 Controller

The controller is the part that will control how the ads are visualized. The controller will receive images from the fetcher and send images to the view. The images sent to the view will be sorted by their color and then filtered. The filter will remove images which have a too bright border color. This is to prevent rendering images that have a white background and is implemented for aesthetic reasons. The communication between the controller and the view will be done via a publisher/subscribe mechanism [15].

The controller will first packetize the image data with relevant data that the view needs to place it in the right location. The packet will also contain information about which monitor that is is supposed to handle the image. When a packet is ready it will
publish it to all subscribers and through the information within it, the subscribers will know how to handle it.

2.2.3 View

The view will be responsible for rendering the images of the advertisements. This part will be the most time consuming as we are going to implement it in OpenGL. To lessen the time to get a working prototype, we will write the view in Python. This way we can, without much effort, rewrite the view in Objective-C. To change platform might be necessary if the performance of the Python prototype is to low.

The current view is implemented in Objective-C and runs on a Mac-mini. The function
of the view is to receive images and animate them on two screens. To render the images, the view is using OpenGL. However, the view is still quite simple since it has implemented a 2d-space instead of a 3d-space, this allows the view to position the images relative to a simple coordinate system with pixels. This is used together with a grid system that maps an ad from the controller to render-able image in OpenGL. Whenever an ad has been published to a monitor on which it is supposed to be showed on, the view interpolates the distance from a starting position not visible above the upper edge, to the position in the grid where the ad will be placed. This interpolation is done over a time that is the same for every ad, regardless of the distance. Images that have a longer distance will thus have a higher speed.

Overview

Our system will be composed of three parts; a fetcher that will receive images from the latest ads added to Blocket.se and send it to the controller, a controller that receives images from the fetcher and sends them to the view, a view that receives images from the controller and renders them on the screen.

Performance

There will be two kinds of optimizations, one is optimization of calculating the physics and showing the pictures, the other optimizations will be the communication between the server and the clients. Both optimizations are important, the optimization for the physics and the graphics will determine the total amount of advertisements that simultaneously can be shown in the ad-wall and still look fluid. The first prototype of the product will be developed in Python. The prototype might prove to be to unoptimized and in that case the work will be moved to an existing prototype implemented in Objective-C. Since the product will be using OpenGL to render the graphic, different rendering methods will be tested and analyzed. The optimization between the server and the clients will also determine the total number of images from the advertisements that the system can handle.
2.3 Summary

The project will be composed of three parts; the fetcher, the controller and the view. All of these three parts are already implemented, our biggest initial task will thus be to understand the existing system, the environment and what we need to focus to improve on. We will also try to get a good understanding of what affects the performance of, especially the view. Any lag in the view will be very noticeable so whatever affects the rendering time in the view will need to be scrutinized.
Chapter 3

Tools

In this chapter the tools and techniques used in the project and the existing system and how they are relevant to the project, will be discussed. The tools that will be presented are the Python programming language, the Objective-C programming language, the ZeroMQ network library, the Blocket.se API and the OpenGL computer graphics library. A more detailed explanation of the relevant parts of the OpenGL library for this project will be discussed in chapter 4.

3.1 Python

Python is a high level programming language [12]. Python was designed with focus on code readability. The Python software foundation claims that:

Python combines remarkable power with very clear syntax

However, the performance might be relatively good for an interpreted scripting programming language, it is considerably slower than a low-level programming language, for example C.

When calculating simple mathematical problems, the Python programming language can be up to 1543% slower than an application implemented in the C++ programming language [17]. One remarkable feature in Python is that instead of delimit code by using { and } as is usual in imperative programming languages, python delimits code by indentation levels. This enforces that the indentation levels are correct, further increasing code readability.
The fetcher and the controller in the existing system are implemented in Python. Unless the Python programming language is deemed to slow to be used in the controller or fetcher, the existing Python prototypes can be used as a base.

3.2 Blocket API

An API(Application Programming Interface) will be used to communicate with the Blocket.se database [3]. The API that will be used is implemented in Python. A key is required to use the API, to obtain a key the user has to register as a developer on blocket.se. Once a key is acquired the user has access to parts of Blocket.se database. The API-key the project will use has extended access to the database, such as access to all the categories.

3.3 ØMQ library

ZeroMQ is a library that simplifies network sockets. It is asynchronous and aimed for high-performance concurrent applications. Even though our system is not that dependent on a high-performance network implementation, we still benefit from the simple implementation and not having to redo an already implemented network component, since the ZeroMQ library is already used in all the parts of the existing system.

The code example 3.1 and 3.2 implements a server and client respectively using the ZeroMQ library. They are implemented in Python and uses the publisher/subscribe network model.
Listing 3.1: server.py

```python
import zmq
import time

def create_message():
    return ['Hi there', 'this is a message', 'to my', 'clients', 'EOL']

context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://127.0.0.1:54321")
clients = ['client1', 'client2']

message = create_message()

while True:
    msg = message.pop(0)
    for client in clients:
        socket.send(client + ': ' + msg)
    if len(message) < 1:
        message = create_message()
    time.sleep(1)
```
import zmq

def create_socket(num):
    context = zmq.Context()
    socket = context.socket(zmq.SUB)
    socket.connect("tcp://127.0.0.1:54321")
    socket.setsockopt(zmq.SUBSCRIBE, "client" + str(num))
    return socket

def change_cli_num(num):
    if num == 1:
        return 2
    else:
        return 1

cli = change_cli_num(1)
socket = create_socket(cli)
while True:
    msg = socket.recv()
    print(msg)
    if msg.split().pop() == 'EOL':
        cli = change_cli_num(cli)
        socket = create_socket(cli)

3.4 OpenGL

Open Graphics Library is an API for computer graphics. The library handles both 2D and 3D space. The purpose of OpenGL is to provide an interface to hardware that renders graphics. Manufacturers can also provide new functionality through OpenGL that can later
be accessible for every user with the sufficient hardware. OpenGL is now managed by a non-
profit consortium called Khronos Group. OpenGL accepts primitives which later through
a pipeline is converted to pixels that can be shown on the screen. The pipeline is called
the OpenGL state machine, the OpenGL state machine is a highly complex architecture
that cannot be covered in this thesis.

In summary the OpenGL pipeline works as a state machine, each state is represented
by various variables. Changing these variables changes the state, which in OpenGL is
done with function calls. Most of the errors that occur is due to not understanding the
OpenGL states. The views will run on Mac OS X, which is not supported by the alternative
Direct3D.

3.4.1 2D versus 3D

When using OpenGL it is possible to choose to either only use a 2d-space or a 3d-space.
Since we will only use images which only has two dimensions, it is possible for us to only
use the 2d part of OpenGL. There are some advantages of using a 2d-space instead of a
3d-space, the most relevant to our problem is that it becomes easier to map the images
that only have two dimensions, onto the screen which also only have two dimensions. To
bring our images into a 3d-space is costly in both development time and performance. This
is at least true in theory, but there are some things that speaks against using a 2d-space,
even if there are only two-dimensional components that are being rendered:

- Contemporary hardware is often built to handle 3d-spaces instead of 2d, thus it
gives a lot more functionality that is usable for optimization (an example of this is
instantiation where the graphics card can use a single object to render several variants
of that object with some transformation applied to it). Even though many of the
techniques and optimizations is not applicable to our situation, the added flexibility
could in the end be beneficial.

- The biggest use of OpenGL is on 3d-spaces, this means that it is easier to find help
and techniques that are not available for 2d-spaces.
• The use of shaders is not available for 2d-spaces. A shader is some calculation that changes the color of an object that is rendered on graphics hardware. Shaders are primarily used in computer games to manipulate the material of a 3d-surface. Shaders might however prove to be useful in our situation, and thus needs to be taken into consideration.

• Changing camera angles or positions is easier using a 3d-space. This will add more options for animations.

Many of the advantages of a 3d-space are not always relevant to our situation as we are only handling 2d-components. However not using a 3d-space will limit the system in terms of animation capabilities, performance and future extensions. Based on these limitations it is more reasonable to invest more time for a 3d-space system.
Chapter 4

OpenGL and 3D

This chapter will give a more detailed and technical explanation to the OpenGL library as well as some general 3d-theory. The sections are not necessarily specific for the project, but rather important for most of the OpenGL applications and sometimes 3d-applications in general. Four common rendering techniques will be discussed, as well as how they work.

4.1 OpenGL mesh rendering

OpenGL provides several alternatives to render objects to the screen. In general, there are two techniques to render a surface, either to define the surface procedurally (patch modeling) which is the 3d-equivalent to vector graphics. The other technique is to define the surface (mesh) with a number of geometric points (vertices). Only the latter is supported by OpenGL but patch modeling can also be faked in OpenGL, by using vertices. We are only using vertices to define our mesh since we will only be using quadratic shapes and it is the only technique discussed in this chapter. When programming in OpenGL there are two relevant parts, the host and the client. The host is the Graphics Processing Unit (GPU) where all the OpenGL commands are run. The client is the processor with the internal memory [6], this difference is useful to know later on when we discuss the different rendering techniques.
4.1.1 Rendering techniques

The term rendering technique means how the application and OpenGL together is integrated to render the mesh. The four rendering techniques presented are not all the rendering techniques available for OpenGL, but a pick of new and old techniques that are quite common to use. The four rendering techniques are:

**glBegin/glEnd** This rendering technique directly use OpenGL operations. The operations are ordered in a sequence to render a coherent mesh [9].

**Display List** Display lists works like `glbegin/glEnd` but instead of having the list in the application, it is sent to the host in the initialization of the application, all the operations in this list can then be called by using one call [7].

**Vertex arrays** The vertex array rendering technique is to store the vertex data on the client in an array. In each draw call, the data is sent to the host [2].

**Vertex Buffer Object** This rendering technique is a combination of display lists and vertex arrays. The data is stored in an array, but is at the initialization of the application, sent to the host [11].

One advantage when storing the data on arrays, instead of drawing the vertices from operations, is optimization. Especially when using objects with many vertices, OpenGL can then on the host, perform optimizations on the mesh [10].

The rendering techniques can be divided into two classifications; immediate mode and retained mode. The definition of these classifications are informal and can differ depending in different contexts. In this section, the usage of the terms; retained mode and immediate mode, is also the one most used in the context when discussing OpenGL rendering techniques.
4.1.2 Immediate mode

Immediate includes the rendering techniques of both glBegin/glEnd and vertex arrays. The problem with using immediate mode is that the program is locked until the entire sequence of operations between glBegin and glEnd function calls are finished or, in the case of vertex arrays, until the GPU has fetched the data in client memory. This lock is to prevent the data in the memory of the client to be altered or freed while the data is transferred to the host, or in the case of glBegin/glEnd to assure the atomicity of the operations. This side effect along with the transfer cost of sending data from the client memory to the GPU has a severe implication in the final performance. This is especially true if the rest of the program requires a lot processor computation. Example of drawing a quadratic shape with glBegin/glEnd in OpenGL:

```c
glBegin (GL_QUADS) ;

glVertex3f (1,1,0);
glVertex3f (0,1,0);
glVertex3f (0,0,0);
glVertex3f (1,0,0);

glEnd () ;
```

This will draw a quadratic shape with the coordinates: (1,1,0), (0,1,0), (0,0,0), (1,0,0), shown in figure 4.1.
4.1.3 Retained mode

To accommodate for the drawbacks of immediate mode, retained mode was introduced in OpenGL from the beginning in the form of display lists. Since display lists sends a list of operations in the initialization, an application can immediately resume, while still guaranteeing the atomicity of the sequence of operations. Along with Display lists, another rendering technique that uses retained mode is Vertex Buffer Object (VBO). Vertex buffer object combines the efficiency of storing the data on an array, with the efficiency of also storing the data on the host. The Graphics Processing Unit can also do optimization operations on the data when using VBO. Because of the overhead with VBO it can be
inefficient to use if there are a lot of small objects. The drawback of using retained mode is that, since the data is stored on the host, it is more costly to change the data. If the scene would involve a lot of dynamic geometry and minimal unchanged geometry, there would be a gain in using immediate mode.
4.1.4 Conclusion

So far four different rendering techniques have been mentioned that each can be assigned into two different kind of categories; how the data is called or how the data is stored. In the category how the data is stored we have immediate mode against retained mode. In retained mode we have display lists and vertex buffer object. In immediate mode we have vertex arrays and glBegin/glEnd. The other category is how the data is called; with operations or from an array of data. glBegin and display lists are called with operations and VBO and vertex arrays are drawn from an array. This gives us four different techniques with each a unique solution that makes more or less suited for certain situations:

**glBegin** Immediate mode/Operations:
- most suited for small amounts of dynamic data

**Vertex Array** Immediate mode/Array:
- most suited for large amounts of dynamic data

**Display list** Retained mode/Operations:
- most suited for small amounts of static data

**Vertex Buffer Object** Retained mode/Array:
- most suited for large amounts of static data.

The difference between static and dynamic data will be discussed in [4.2](#). Note that this is a general assessment of the four rendering techniques on a per object scale. Specific implementation and performance testing of the different rendering methods will be explained in [5.6](#).
4.2 Static vs. Dynamic properties

An important aspect in the context of 3d-rendering, is the composition of the objects. The composition of the objects determine in which levels the objects have either static or dynamic properties. Imagine a particle system that have a lot of moving objects. If all the particles in the system were composed into a single object, it would make much more costly to change for some rendering techniques. The vertex buffer object and display lists are the rendering techniques that have the highest cost for changing the properties inside an object. This is because they store the data on the host, which means the data needs to be resent. Vertex buffer object are also not optimized for having to many small objects instead of a few larger objects. This is because the benefits from using vertex buffer objects diminishes as the objects gets smaller and the application have to do more draw calls.

4.3 3D space in OpenGL

It is important to understand how OpenGL and other 3d-rendering libraries handles the 3d-space. The space that we occupy is composed of three dimensions and this is what OpenGL is trying to simulate. In the same way the real space works, without any subject, the OpenGL space works independently from the screen or viewer. The mapping from the 3d-space to a viewable 2d-frame is called rendering.

The result of the rendering is determined by many factors; scene content, camera position, camera properties and shaders are amongst some. The OpenGL library does not actually supply functions to change the camera position directly but is instead always looking at the position (0,0,0). It is also not possible to change the properties of the camera from the OpenGL library. This can be done however with the \texttt{gluPerspective()} function that is located in the OpenGL utility library (GLU) \cite{8}. This function can change

- field of view
- aspect-ratio
- the closest clipping plane
- the farthest clipping plane.
To give the illusion of moving in the 3d-space, transformation matrices are applied to the matrix stack in an inverse manner, effectively moving the entire scene around (0,0,0). Even if there is no function to change the position of the camera in the standard OpenGL library, the GLU library supplies such a function: \texttt{gluLookAt()}. With this function it is possible to specify the camera position in three ways:

- What it is looking at
- Where it is positioned
- How it is rotated.

![Figure 4.2: Image of frustum](image)

### 4.3.1 Camera properties

The viewable region generated by a camera, can be seen as a frustum, specifically a square frusta, see Figure 4.2.
The smaller plane can be seen as the beginning of the 3d render-able region, the location
of this plane is determined by the closest clipping plane. The larger plane is the end of
render-able region and is determined by the farthest clipping plane. The ratio between
these planes are determined by the field of view (FOV). The aspect ratio determines how
wide the planes are.

Field Of View

The field of view (FOV) value determines the viewing angle. In OpenGL the field of view
value is the angle on the Y axis A low field of view will give the effect of zooming in, since
the camera can see less but still fills the screen. A high field of view will thus give the
effect of zooming out, since it is now possible to see more. The field of view angle for a
human is said to be around 95% in the outward direction on the X axis and 60% in the
inward direction. A field of view angle of 155% would be to wide to have in a computer
graphics application, since a screen or a computer modeled camera is not as intricate as a
human eye and does not example have a periphery.

Aspect-ratio

The aspect-ratio is the ratio between the height and the width. This value should be the
same as the screen that is being used, otherwise objects will not look as intended, but
instead longish or broad. The human eye aspect-ratio can be calculated from the FOV angle

\[
\begin{align*}
outangle &= 95 \\
inangle &= 60 \\
totalXangle &= 95 + 60 = 155 \\
upangle &= 60 \\
downangle &= 75 \\
totalYangle &= 75 + 60 = 135 \\
aspect - ratio &= 155/135 \approx 1.15
\end{align*}
\]
This aspect ratio is for one eye only. In reality since we have two eyes the aspect ratio is higher. This is why screens have an aspect-ratio above 1.

**Clipping planes**

The clipping planes $z_{\text{Far}}$ and $z_{\text{Near}}$ determines of how far away and how close to the camera, objects will be rendered. It is most desired to have the ratio between the farthest clipping plane and the closest clipping plane as small as possible. This is because this ratio determines the resolution OpenGL can use to calculate the depth buffer. This buffer is used to determine which object is in front of the other. The precision in bits that are lost is roughly

$$b \approx \log_2\left(\frac{z_{\text{Far}}}{z_{\text{Near}}}\right)$$

In this equation, the precision in bits that is lost will reach infinity when $z_{\text{Near}}$ reaches zero. Having a very small $z_{\text{Near}}$ value will severely limit the depth buffer precision. In addition, the precision is biased to the advantage of the closer clipping plane. When the ratio increases, this bias will also increase.

**4.3.2 Camera translation and rotation**

Having the camera pointed in the correct position is imperative in an OpenGL application. With three different kind of parameters, it is possible to control how the camera should be pointed. These three parameters are; The position of the camera, the position the camera is looking at and also a direction which tells what side is up. Since the camera exists in a three dimensional space, the parameters should be in format of a three dimensional vector; x, y and z.
Chapter 5

Experiment and implementation

This chapter will discuss the implementation of the project, the implementation is based on the tools and techniques discussed in chapters 3 and 4. The beginning of the chapter will discuss the first prototype in python as well as the final version implemented in Objective-C. The end of the chapter will further investigate different OpenGL implementations.

5.1 Overview of the system

The system consists of three parts; the view which renders the images of the advertisements, the controller which receives data and sends it to the view and the fetcher which sends data to the controller. This is a variant on the Model-View-Controller (MVC) pattern, but since the model part is already implemented by Blocket.se in their database, a fetcher which pulls data from this database and pulls it to the controller is more suitable. Both the fetcher and the controller is implemented in the Python programming language, the first prototype of the view was also implemented in Python but was later replaced by an existing Objective-C prototype.

5.2 Python view

The first prototype was made in Python, using the PyGame library with OpenGL for Python to draw the scene. With this prototype, a MVC pattern was used, that is, to sep-
arate the data, the scene rendering and the animation system into 3 independent modules. Using the MVC pattern, made it easier to make a reliable model and controller part and at the same time experiment with OpenGL in the View part.

The first implementation rendered the objects using `glBegin()` and `glEnd()`, we also learned how to initialize an OpenGL window. An example of how to initialize an OpenGL window and draw a plane using the `glBegin()` and `glEnd()`. Example 5.1 will render a quadratic shaped object.

Listing 5.1: Quadratic shape

```c
 glBegin(GL_QUADS)  
    glVertex3f(0.0, 0.0, 0.0)  
    glVertex3f(1.0, 0.0, 0.0)  
    glVertex3f(1.0, 1.0, 0.0)  
    glVertex3f(0.0, 1.0, 0.0)  
 glEnd()  
```

Using `glBegin()` and `glEnd()` is the simplest way of rendering in OpenGL, but the performance can be severely limited by using it.

In an effort to increase the performance, the rendering technique was changed to display lists. Display lists are sets of compiled OpenGL operations that can be stored on the host and can then be executed, using only one operation. Using Display lists improved the performance but it was still not sufficient, to further increase the performance, Vertex Arrays and Vertex Buffer Objects was implemented. Both Vertex arrays and Vertex Buffer Objects are contemporary rendering techniques, where Vertex Buffer Objects are almost exclusively used in larger projects. Since these techniques are newer and more advanced they should have increased the rendering performance, instead only a minor improvement at best was experienced, and a slightly worse performance at worst. How vertex arrays, vertex buffer objects and the other rendering technique works and why they did not perform as believed, will be discussed in chapter 6.
5.3 Objective-C view

Since the desired performance was not achieved with Python, the view was replaced by an existing prototype of the view, implemented in Objective-C. This view was then modified for our problem.

The prototype used the Cocoa API that is available when developing for Mac OS X. To easily render the images, the prototype maps the images to a 2d coordinate system. The render system was the first thing that was changed. The existing prototype only had one animation implemented, an animation that simulated an advertisement falling down to a position in the grid.

Whenever the controller sent an Ad, the existing prototype placed it above the visible area of the screen. To make it look like it fell down, the prototype calculated the trajectory based on its current location, the destination and a duration. The structure of the animation system was kept but how the animations were created and used was changed.

5.3.1 Overview of the view

The view is responsible for rendering the images it receives from the controller. The view is also responsible for animating the images. To achieve this, the view consists of three parts that control the application: a render part responsible for rendering the images, a grid part which is responsible for implementing the images into ads, along with handling the events from the controller, there is also an animator part which is responsible for handling the animations on the ads, see figure 5.1. There are also four other parts, a fetcher part which is responsible for handling the network communication, a model part for an ad, a model part for a message and a model part for an animation.

Ad model

The ad model in the system represents an ad. The ad holds information of which column and which row the ad has in the grid, as well as which image that the ad should render. The ad also implements a function to render the ad, this function is directly invoked from the grid.
Message model

The message model is instantiated each time a new message is received in the fetcher. The message contains data about a new ad. The data consists of the image, the row and column information, the event information as well as the information about which monitor that should render the ad.

The render part

The render part contains an instance of both the grid and the fetcher. It consists of two parts; the first part is initialization and the second part is the main loop of the application. In the initialization part, the fetcher and the grid is instantiated, the proper OpenGL functions are also called for setting up the OpenGL window. In the second part, the render part invokes two methods in the grid, The first is to iterate all the ads and call their render method. he second method invoked in the grid is an update method. The render part also delegates messages from the fetcher to the grid.

The grid

The grid stores all the ads in the system. It has an instance of the animator part. It has two responsibilities, to handle the messages received from the render part and to invoke an
update method in the animator. When a message is handled to the grid, it first determines the event of the message. If the event is either replace or fill, the grid will create a new ad from the data in the message and replace it with the ad that has the same index. Also depending on the event in the message, the grid will invoke methods in the animator to create new animations. For example, if the event is replace it will invoke a method in the animator to create the replace animation. This particular animation requires three parameters; the new ad, a list of adjacent ads and a list of the distance values from all the adjacent ads to the new ad. The grid then also need to calculate this information before invoking the method in the animator.

The animator

The animator has two responsibilities; the first is to create new animations from the grid and the second is to iterate each animation and invoke its animate method. The animator will invoke the animate method for each animation each time the update method gets invoked from the grid. When the update method in the animator is invoked, the current time is stored. The time difference can then be calculated and sent to the animate method in each animation, from this method the animation can move the ad a distance depending on the time since the last movement. The animate method in the animation will return true if the animation is completed. The animator will then check if the animation have a next animation, if so the animator will add the next animation to the list of animations and remove the first animation.

5.3.2 Creating animations in the animation system

The work flow for creating an animation is to first instantiate an animation class and adding it to the list of animations. The animation can also reference to a next animation that will start after the animation has finished, this is to ease synchronization of animations and remove the dependency of knowing the times on beforehand. This is an example of how to create an animation A with a next animation B:

1. Create an animation A.

2. Put the animation A in the list of animations.
3. Set speed and/or end positions on animation A.

4. Create animation B that will succeed animation A.

5. Put animation B in the next list of animation A.

6. Set speed and/or end positions of animation B.
The simplest method for creating a movement with the animation is to set an end position. The animator will calculate direction vector for the speed based on the end position, each time the animation is added to the list of animations:

\[
\begin{align*}
  dX &= x_1 - x_2 \\
  dY &= y_1 - y_2 \\
  dZ &= z_1 - z_2 \\
  sp &= \sqrt{(X^2 + Y^2 + Z^2)} \\
  vX &= \frac{X}{sp} \ast gV \\
  vY &= \frac{Y}{sp} \ast gV \\
  vZ &= \frac{Z}{sp} \ast gV
\end{align*}
\]

Where:
- \(dX\) is the difference between the current position and the new position in the x axis.
- \(dY\) is the difference between the current position and the new position in the y axis.
- \(dZ\) is the difference between the current position and the new position in the z axis.
- \(sp\) is a normal vector.
- \(gV\) is a variable to change the speed.
- \(vX\) is the speed in the x axis.
- \(vY\) is the speed in the y axis.
- \(vZ\) is the speed in the z axis.
The new coordinates for the ad in each axis can now with the speeds and a delta time, be calculated:

\[
\begin{align*}
x &= cX + vX \cdot dt \\
y &= cY + vY \cdot dt \\
z &= cZ + vZ \cdot dt
\end{align*}
\]

Where:
- \(x\) is the new position in x axis.
- \(y\) is the new position in y axis.
- \(z\) is the new position in z axis.
- \(cX\) is the current position in x axis.
- \(cY\) is the current position in y axis.
- \(cZ\) is the current position in z axis.
- \(dt\) is the time since the last calculation.

It is possible to give an animation an arbitrary speed, but it is not recommended. The speeds will be set to zero when the new coordinates will move through the end positions in this iteration, this is checked before the coordinates are updated. And as the animation will exit when speed in the x,y and z directions are zero, this opens the possibility of an animation that never exits, if for example the velocity is not in the same direction as the end position of any axis.
5.3.3 Implemented animations

There are three animations implemented in the system; two animations that move the ads and one animation that simply turns the images invisible. The two animations that move the images are used in two different states, when the grid is full and when the grid is filling up.

![Replacing middle ad with shaking animation](image)

**Figure 5.2: Replacing middle ad with shaking animation**

**Replacing animation**

The animation when the grid is full, replaces the ad with a new ad and gives the new ad, and the ads close to it, an oscillating movement. The oscillating movement is one part in the x-axis and one part in the y-axis. Each oscillation reduces the speed and the amplitude of the oscillation. The speed and the amplitude of the oscillating movement is also affected by the distance of the ad from the new ad, the new ad in the center will have the highest amplitude with the highest speed, and the ads furthest away from the new ad in the center
will have the lowest amplitude and speed. There is also a delay of when the oscillating movement begin for the ads, based on the distance of the new ad. The end result of the animation gives an illusion of the new ad causing the ads surrounding it, to shake. See figure 5.2.

To have all the surrounding ads to move with the new ad, the application calculates which ads are adjacent to the new ad. The algorithm to get the adjacent ads works by getting the row and column from the index value of the ad. With the row and column value, the indexes for the ads, one column left, one column right, one row up and one row down can be obtained and put in a list of adjacent ads. To get the surrounding ads, further away then one step, the algorithm is iterated on all the ads in the adjacent list and the new adjacent ads is put in this list. If iterated n-times, the list will be filled with ads n-step away from the center. With the complete list of adjacent ads, the distance is calculated using Pythagoras theorem.

![Diagram over the movement of the falling ad animation](image)

**Figure 5.3: Diagram over the movement of the falling ad animation**

**Fill animation**

The fill animation is used to fill the grid of ads. The end position of the ad is determined by the controller, the start position is in the same column as the end position and on a row that is not inside the viewing area. The ad is given a negative velocity in the y-axis, this causes the ad to move to its end position. When the ad reaches its end position it will oscillate in the y-axis with the end position as the center point. The ads 3 steps under the new ad will also start to oscillate. The speed and the amplitude is lower the further away the ad is from the new ad. The animations are also slightly delayed so that the new ad starts to oscillate first and the ad furthest down oscillates last. The end result of this animation is that the new ad falls down on the ads below it, causing the ads to vibrate from the impact. See figure 5.4.
5.3.4 Fetcher

The fetcher is the part that connects the view with the controller. The connection is publish/subscribe-based, meaning that the controller will publish its data, and the view will subscribe for it. The fetcher runs in a separate thread. Whenever the controller sends a message, the fetcher will interrupt the program. Based on the payload in the message, the fetcher will invoke different methods in the main thread. This introduces an easy method to change the settings in the different views.

5.3.5 Camera settings

In OpenGL it is possible to change the perspective with the function call `gluLookAt`. The parameters are in groups of 3:

- Where the camera is.
- At what direction the camera is looking at.
- What direction is up.

Since the camera needs to look directly at the objects in such a way that they perfectly fits the screen, see figure 5.5 the exact coordinates needed to be calculated. The result of
the calculations puts the camera in the X direction, one half as many steps as how many columns we had, and in the Y direction; one half as many steps as how many rows we had. To get the offset in Z-axis we needed some trigonometry:

\[ z = \frac{x}{\tan(A)} \]

Where:
- \( z \) is the calculated distance
- \( x \) is half the number of rows
- \( A \) is the field of view angle in the Y direction.

In the picture:
- \( z \) is the distance from the camera to the objects.
- \( x \) is half the height of all the ads.
- How cameras work in a 3d-view is explained in more detail in chapter 4.3.
5.4 Controller implementation

The controller which is written in Python acts as a broker for the images and is responsible of delegation of the images. The controller is also responsible for synchronizing events for all the views. When an advertisement is added to the blocket.se database and the fetcher have forwarded the image of the advertisement, the controller decides to which view and on which position the image should be rendered. The decision of which view and position, is determined by the hue value of the image. This creates a grid where the images are sorted by color.

5.4.1 Color sorting

Each pixel in a digital image consists of color data. In this system RGBA is used (Red, Green, Blue and Alpha) \[10\]. This information can be transformed to a HSL (Hue, Saturation and Lightness) color system. See Figure 5.6 for explanation. The HSL color system is used in the controller to sort the images based on their color. Since the images needs to be sorted on a one-dimensional list, and a color is defined by three values, the best compromise is to sort the images based on their hue value from the HSL system. The hue value would be the most relevant part of the HSL system to use as it best at representing what normally would be considered colors, even though a color is actually all the three components in totality. The problem arises when images with either very high or very low saturation and lightness values enters the system. These images need only to have a trace of the same hue value, to be processed the same by the system. This means for example a very dark green image will be treated the same as a very light green image. Generally the end result still gives the illusion of sorted colors.

Code example 5.2 shows how the hue, saturation and lightness is calculated from the red, green and blue values from the pixels in the images.

Listing 5.2: RGB convert

#Converts the image from RGB system to HSL system
pixels = list(image.getdata())
avg\_r = sum(zip(*pixellist)[0])/float(len(pixellist))
avg\_g = sum(zip(*pixellist)[1])/float(len(pixellist))
avg\_b = \text{sum}(\text{zip}(*\text{pixellist})[2]) / \text{float}(\text{len}(\text{pixellist}))

H, L, S = \text{colorsys.rgbtohls}(\text{avg}\_r, \text{avg}\_g, \text{avg}\_b)

One of the requests from Blocket.se was that images with a very bright border color should be filtered. This is easily achieved with an algorithm similar to the one that calculates the HSL values. Instead of using the entire image, only the pixels at the border is converted. The average lightness value for all the border pixels are calculated and if the average value is too high, the image will be filtered.

5.4.2 Events

Several events have been implemented to synchronize the views; initialize, replace and clear. When an event is sent to the view, the view is then responsible for the task of the event. This creates an asynchronous system which is robust but not always perfectly timed.

The controller only transmits what type of event and the end position of the image. The view is then responsible to handle the event and render the image on the correct end position.

The controller have a state for each event-type. There are four states; filling up the grid with ads, replacing ads in a full grid, clearing the grid and suspend the view in a sleeping state. These states are in the system called; init, replace, clear and sleep respectively. The
states will change in a rotation, first the controller will start in the sleep state to make sure each view is ready for the init state. Each change of states will have a sleep state in between, for this reason. After the init state and the sleep state in between, the next state in the rotation is the replace state. And finally the last state is clear state to restart the rotation. The time for each state and the order of the states is configurable from a file. This enables the customer to easily change the total rotation time and individual states. To limit the strain on the view, each state is configured to only send events at a certain rate.

5.5 Performance testing

Since the implementation of the project involves real time graphics rendering, the implementation needs to achieve acceptable performance. This means that a short time between frames is needed, it also means that the time between frames cannot have large fluctuations. Both aspects are important to support the experience of a fluid animation. Another focus in this chapter will thus be to explore how different implementations affect the performance. The overall performance, measured in frames per second, is determined not only by the Central Processing Unit and the Memory, but also by the GPU and the interaction between the two. Because of the complex relationship between implementations and performance, a complete mapping of how different implementations affect the performance will not be possible. The goal will thus be to find the most optimal solution for the problem.

5.5.1 Measure performance

The easiest way of determining if the performance is good enough, is to measure the frames per second. This rate is affected by how many times the program can complete a rendering cycle. The higher this rate is, the more fluid and responsive the experience will be. This rate can be measured in several ways. To minimize the impact of the measuring method, it is possible to measure the rate over an interval and then calculate the mean value. The problem with measuring the frames per second in this way, is that fluctuations are hard to notice. To reveal any fluctuations that would seem annoying to the viewer, the measuring method can be complemented with actually observing how fluent the application seems to
run. This method is hardly the best one, but is easy to implement and works relatively well.

This is one way to calculate the frames per second rate:

\[ fps = \frac{n}{t} \]

Where:
- fps is frames per second.
- n is the total number of frames rendered.
- t is the total running time.

### 5.5.2 Measuring time

Example of starting a clock and measuring time.

Listing 5.3: Measure time

```python
def start_clock(self):
    self.time = time.clock()

def clock(self):
    return time.clock() - self.time
```

### 5.5.3 Python versus Objective-C

The first prototype in Python was compromised of several performance issues, mostly due to the limitations in the Python programming language. The largest limitation was that looping in Python is much slower compared to a more low-level language. This limitation proved to be very cumbersome since the program had to loop up to 4000 objects in each frame. The advantage of Python is its readability rather than pure performance, even if it is possible to achieve decent performance.

Objective-C on the other hand is a thin layer on top of the C programming language. Any non-object oriented syntax are based on the C syntax. All the object oriented syntax comes
5.6. IMPLEMENTATION AND TESTING OF RENDERING TECHNIQUES

from the Smalltalk programming language. In the Mac OS X programming environment, Objective-C also includes the Cocoa framework. Unlike Python, Objective-C could easily loop through 4000 objects.

5.6 Implementation and testing of rendering techniques

This section will explain the specific implementation steps of the four different rendering techniques. There will also be an explanation of the tests of conducted on these rendering techniques. The results and evaluation of the tests will be discussed more thoroughly in 6.

5.6.1 glBegin and glEnd

This code will draw a plane in immediate mode directly when called from the program.

```
  Listing 5.4: glbegin

  glBegin (GL_QUADS);
  glVertex3f (1.0f, 1.0f, 0.0f);
  glVertex3f (0.0f, 1.0f, 0.0f);
  glVertex3f (1.0f, 0.0f, 0.0f);
  glVertex3f (0.0f, 0.0f, 0.0f);
```

Figure 5.7: Quad
glVertex3f(0.0f, 0.0f, 0.0f);
glEnd();

This code consists of two parts; the first part is `glBegin()` with `glEnd()`. When calling `glBegin()` it is needed to supplement a parameter to tell OpenGL what it will create with the vertices drawed after, since we want a plane we use the parameter GL_QUAD. The other part is the four `glVertex3f()` function calls that actually draws the vertices in the correct to position to create a plane.

The number 3 and letter f in `glVertex3f()` is to tell OpenGL that it should handle the parameter as a float number and create a vertex with three dimensions. There are other variants on `glVertex`, for example it is possible to use 4 dimensional vertices or use double numbers.

### 5.6.2 Display lists

The OpenGL operations can be compiled to a list using: `glNewList()` and `glEndList()`. This is only necessary to do once.

Listing 5.5: display list

```c
quad = 1
glNewList(quad, GL_COMPILE)

glBegin(GL_QUADS)

glVertex3f(1,1,0)
glVertex3f(0,1,0)
glVertex3f(0,0,0)
glVertex3f(1,0,0)

glEnd()

glEndList()
```

To call the list of operations the identifier `quad` is used:

```c
glCallList(quad)
```
5.6. IMPLEMENTATION AND TESTING OF RENDERING TECHNIQUES

It is not only possible to store `glVertex3f()` or `glBegin() / glEnd()` in a list, rather any OpenGL command can be stored.

5.6.3 Vertex Buffer Objects

Vertex Buffer Object (VBO) is an object consisting of vertices, texture coordinates and other data. It is used to render meshes in OpenGL. There are three steps to implement a VBO:

**Generate buffer** First we generate the buffer object with `glGenBuffers(n, buffers)`. Here, `n` is the number of buffers to be generated, the identifiers of the buffers will be put the `buffers` array.

**Bind buffer** After we have generated the vertex buffer objects, we need to bind them using `glBindBuffer(target, buffer)`. Here `target` is a hint to OpenGL what kind of data that will be put in the array, `buffer` is the identifier of the buffer that you want to use.

**Send data to buffer** In the final step we send the data to the vertex buffer object we have created and binded from a local array in our program: `glBufferData(target, size, data, usage)`. Here `target` is also a hint to OpenGL of what kind of data that is being sent. `Size` is the size in bytes of the new buffer, `data` is an identifier to the data that will be sent, `usage` tells OpenGL how the vertex buffer object will mostly be used.

The `glBufferData()` function does not take an identifier as a parameter, this is because `glBind()` is used to specify which vertex buffer object will receive data.

```
Listing 5.6: VBO
vbo = vbo.VBO(array(  
    [ 0.0, 0.0, 0.0 ],  
    [ 1.0, 0.0, 0.0 ],  
    [ 1.0, 1.0, 0.0 ],  
)
```
5.6.4 Vertex arrays

A similar way of storing vertices is an array on the computers memory. In this example every vertex is stored in an array.

Listing 5.7: Vertex Array

```python
def create_arr(n):
    a = numpy.zeros((n*4,3), 'f')
    step = 0
    for i in range(n):
        a[step,0] = i/10.0
        a[step,1] = 0.0
        a[step,2] = (i/20.0)

        a[step+1,0] = (i/10.0)+0.1  #1
        a[step+1,1] = 0.0
        a[step+1,2] = (i/20.0)

        a[step+2,0] = (i/10.0)+0.1  # 1
        a[step+2,1] = 0.4
        a[step+2,2] = (i/20.0)

        a[step+3,0] = i/10.0
        a[step+3,1] = 0.4
        a[step+3,2] = (i/20.0)

    step+=4
    return a
```
The main reason for using a large array for all the vertices is that it is only needed to make one call to render.

First the array to be used is specified, a function to render the content in the array is invoked.

Listing 5.8: Render array

```c
glVertexPointer(3, GL_FLOAT, 0, arr)
glDrawArrays(GL_QUADS, 0, n*4)
```

Using one call will speed up the rendering but it will not allow us to change texture or color for the planes we draw. This method should be used when there is several surfaces with the same color or texture to be drawn.
5.6.5 Tests of rendering techniques

All these rendering techniques were tested in a Python application. The application simply rendered a number of quads with the different techniques and measured the time it took to complete each call to the host to render the object. The application tests the performance for each rendering technique for two cases; the first is few large objects and the other case is a lot of small objects.

The conclusion that can be drawn from the results of the tests are that, when using vertex buffer objects or vertex arrays, it is most efficient to use as large objects as possible, since having larger arrays in the GPU have a negligible effect on the performance, up to a certain limit. Instead the greatest performance loss is ascribed when using a lot of objects, since there is some overhead when using arrays.

5.7 Summary

In this chapter, the implementation of the project have been described. At first, a Python prototype was implemented from nothing. As the complexity of the program increased together with the unique circumstance of having a lot of small objects, the Python prototype became more and more unsustainable. To further investigate how the performance changed with how OpenGL was used, we conducted a series of tests. The tests were designed to measure how different rendering techniques perform in different situations. The conclusion from these tests were that it would not be possible to achieve acceptable performance with the Python prototype by changing rendering techniques itself. Even though the performance improved by using the most optimal rendering technique, the system still had poor performance. The problem instead was rooted in Python.

Since we had been given a prototype implemented in Objective-C, the work that had been made was moved to that prototype. The performance on the Objective-C prototype was significantly better. It was, an estimation of 3-4 times faster than the Python prototype. After the alterations in the view in the Objective-C prototype, the performance was adequate for focusing on improving the animation system.

The initial animation system was to simple for our problem. The animation system was improved by adding flexibility, it is now able to do more complex animations that also
are synchronized. The different animations created with the new animation system have also been described, they include the falling animation and the replacing animation.
Chapter 6

Results and evaluation

This chapter contains the results of the tests of different rendering techniques, discussed in chapter 5. The test results for the different rendering techniques will be analyzed. The rendering techniques will be analyzed, not only compared to each other, but also how they perform in different contexts.

6.1 OpenGL tests

The tests of the performance for the different rendering techniques were conducted by rendering a number of quads (GL_QUADS) with the different techniques. Two types of tests will be done; One where the objects are composed as one object. The other test will be done with objects that are composed as many objects. The difference is explained in section 4.2.

The test program was written in the Python programming language. Since Python is not as fast in performing basic operations such as; for-loops, if-cases etc, the test results will be exaggerated, the exaggeration will explained in detail for each test. How the different rendering techniques might be affected by being implemented in Python etc, will be discussed.
6.1.1 Test with single object

The first test shows the total rendering time for the different rendering techniques:

![Graph of test results rendering a single object](image)

Figure 6.1: Graph of test results rendering a single object

Table of results:

As can be seen from the graph 6.1 and the table 6.1 vertex buffer objects and vertex arrays are a lot faster than using glBegin/glEnd or display lists. As has been mentioned, the fact that the test application is written in Python exaggerates the results. The skewed results comes from that neither vertex buffer object or vertex arrays needs to be looped to be rendered, instead all the data is loaded initially. This is actually a unique circumstance since the test are rendering identical objects and is able to group them together into one object.
6.1. OPENGL TESTS

<table>
<thead>
<tr>
<th>Rendering technique</th>
<th>time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>GlBegin</td>
<td>0.021</td>
</tr>
<tr>
<td>Display List</td>
<td>0.0070</td>
</tr>
<tr>
<td>Array</td>
<td>0.00055</td>
</tr>
<tr>
<td>VBO</td>
<td>0.00018</td>
</tr>
</tbody>
</table>

Table 6.1: Table of test results rendering a single object.

6.1.2 Test with multiple objects

To investigate this effect, a test case where the quads are not grouped together as one object, was introduced. This forces vertex buffer objects and vertex arrays to loop through all the objects. This is because in OpenGL, the data in the vertex buffer object will only be rendered when they are called from the client. These tests shows the total rendering time when the quads are distributed over multiple objects. Table of results:

<table>
<thead>
<tr>
<th>Rendering technique</th>
<th>time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>GlBegin</td>
<td>0.020</td>
</tr>
<tr>
<td>Display List</td>
<td>0.0074</td>
</tr>
<tr>
<td>Array</td>
<td>0.029</td>
</tr>
<tr>
<td>VBO</td>
<td>0.010</td>
</tr>
</tbody>
</table>

Table 6.2: Table of test results rendering multiple objects.

The results from the graph 6.2 and the table 6.2 gives a hint of how the rendering techniques perform. vertex buffer object and vertex arrays, no longer have a far superior rendering time compared to display lists or using glBegin/glEnd, but instead show more modest results. Using vertex buffer object still have an acceptable performance but is outperformed by rendering with display lists. This is remarkable since in the previous test, rendering with display lists produced considerably worse performance results, compared to vertex buffer objects and vertex arrays.

Even though using the Python programming language gives skewed results, having a lot of objects that each need its own draw call, will severely affect the performance of rendering with vertex buffer objects and vertex arrays, regardless of which platform that is used.
Figure 6.2: Graph of test results rendering multiple objects

It is also worth noting that even when using an extreme test case, designed to make the vertex buffer object perform at its minimum potential, it still produced acceptable performance values. The previous test case, where vertex buffer objects performed at its best is at the other extreme. In that case, the vertex buffer object can, with a single draw call, draw very big meshes, with almost negligible performance differences. Both of these extreme test cases are almost only interesting from a testing perspective. Most of the OpenGL applications would use medium amount of both large and small objects. In these circumstances, using vertex buffer objects would give the best performance.
6.1.3 Test of scalability

Another relevant quality is how the rendering techniques scale. To test this, two tests were conducted, with multiple object composition and single object composition. The test measures the rendering time for the different rendering techniques when the number of quads increases.

As can be seen from the graph 6.3, vertex arrays and vertex buffer objects are barely affected by the increasing number of quads rendered. Rendering with \texttt{glBegin/glEnd} or display lists on the other hand, makes the rendering time to be linear with the increased number of quads.

The graph 6.4 shows how the rendering time increases dramatically for the vertex
buffer objects and vertex array when the objects are dynamic. display lists now have the best performance.

## 6.2 Summary

In this chapter, the results from the tests have been presented. The tests were designed to test the performance of different rendering techniques in different circumstances. The rendering techniques that has been tested are:

- Calling the operations `glBegin/glEnd` directly in the application.
- Storing the operations in a display list on the video RAM and then calling the
operations

- Storing the vertex data on an **vertex array** and then drawing from this array.

- Storing the vertex data in an **vertex buffer object** on the video RAM and then drawing from this **vertex buffer object**

The circumstances that we tested were two extreme cases. The first case had all the vertex data in one object, the other case had the vertex data divided into many smaller objects. The results showed that **vertex buffer objects** and **vertex arrays** performed very well, compared to using **glBegin/glEnd** or **display lists**. In the other test case, **vertex buffer objects** and **vertex arrays** performed more in line with how **glBegin/glEnd** and **display lists** performed.

The test-application can be downloaded from [http://atlantis.cse.kau.se/~di8norl/opengl-test/index.html](http://atlantis.cse.kau.se/~di8norl/opengl-test/index.html)
Chapter 7

Conclusions

This chapter will discuss the conclusions of the entire project. An important issue that will be discussed, was how the predicted work changed because of inadequate performance. This chapter will also include discussions about other problems that occurred had as well as a conclusion about the project overall.

7.1 Implementation

This section explains the problems related to the implementation of the system, from the initial prototype in Python to the integration of the system at Blocket.se.

7.2 Performance problem

The largest assumption we made were that the prototype of the view part of the system could achieve good enough performance when implemented in Python. This assumption caused the project to change platform for the view, well into the project time. The switch of platform did cause a delay of the implementation of the project.

Another assumption was that good enough performance could be obtained by optimizing the use of OpenGL, a lot of optimizations were tried in the prototype. The optimizations that was tried were different rendering techniques. When the view part of the system was switched, the optimizations could be transferred to the new view.
7.2.1 Tests of rendering techniques

Many of the optimizations did not work as expected, for example the Vertex Buffer Object gave us in the Python prototype, worse performance than Display lists. This is remarkable because using Vertex Buffer Objects is a newer and much more advanced rendering techniques compared to using Display lists, which was introduced from the first OpenGL version. To understand how using Vertex Buffer Objects could result in worse performance, all the rendering techniques were tested.

An assumption of why Vertex Buffer Object did not perform as expected was that the view part handled a lot of small unique objects. The tests were then designed to show the difference in performance between the rendering techniques when used with a lot of small unique objects. The opposite case was also tested; when rendering one large object with all the data. The tests revealed that the assumption was correct. When the application could load all the data into one Vertex Buffer Object from the start, a lot of performance could be obtained, compared to iterating over a larger set of smaller Vertex Buffer Objects.

7.3 Final product

When the Python prototype of the view did not have the required performance, the work was moved to the new platform. The platform was Objective-C with Cocoa. Cocoa is the framework that is used when developing OS X Mac applications. Since a prototype already existed, the initial work was for understanding the existing prototype implemented in Objective-C. The prototype in Objective-C used the simplest rendering technique, because of our tests, it could easily be replaced by a more suitable rendering technique; Display lists.

7.3.1 Animation system & animations

When the OpenGL part of the view in the Objective-C prototype had been adjusted, the focus was shifted to the animation system. A lot of the work on the animation system was to simplify the process of creating animations and to increase the functionality of
the animation system. Instead of having to synchronize and coordinate the animations manually, the animations could now be created by specifying a lot less information.

Even if adjusting the system to a more general approach, there is a limit to how much that is gained by generalizing the animation system. The development cost of generalizing the system should not be too large, compared to the complexity of the task the system should handle. The animation system was thus not too general, but rather general enough so that creating and modifying animation did not consume to much time.

An attempt was made later, to introduce collision detection with basic physics to the animation system. An implemented physics system would have greatly simplified the process of creating animations. The attempt was canceled because the development cost of implementing a working, stable physics system, was far more than creating and modifying the animations in the system that already existed.

There were at first one animation that was preferred; the animation that occurs on a full grid of images. Before the integration another animation was created, even if it was not specified, we were encouraged to come up with our own ideas for animations. The most time consuming aspect of completing an animation, is not to create the animation itself. It is rather the process of adjusting the details; speed, coordinates etc that takes most of the time. Even if the adjustments are not needed, the animations will have a very crude appearance without them.

7.3.2 Integration

The integration of the system took place in Stockholm at the Blocket.se headquarter. The process of integration involved installing the view application on the three Mac-Mini computers that are connected to the screens, and to install the controller application on one of the Mac-Mini computers. It is not important on what computer that will run the wall application, as long as the other computers are configured correctly.

The three Mac-Mini computers were not physically available, as they were installed in a sealed compartment, rather they were instead available through SSH and an application called "Screen sharing". With SSH it was possible to start the wall application but for the view application, screen sharing was needed.

When the system had been installed, a memory leak was detected after the system
had been running for a time. The memory leak caused a considerable performance loss in the system, until it eventually crashed. The memory leak was discovered to reside in the Objective-C wrapper for the network library ZeroMQ. When the memory leak was discovered, there was little time to fix it. The quick fix currently in place is to restart the application at the end of each day.

### 7.3.3 Summary of the implementation

The implementation started with a Python prototype, this introduced the first and the major problem. Because of the limitations of applications developed with the Python language; iterating over a large set of objects takes a considerably longer time than a more low level language, the view part of the project had to be moved to a new environment. Since there already existed a prototype implemented in Objective-C, the work was moved to understand and improve this prototype. When the Objective-C prototype had been modified to function as the Python prototype, it was estimated that the Objective-C prototype performed 4-5 times faster than the Python prototype.

During the implementation of the Python prototype, several rendering techniques in OpenGL was tried. The rendering techniques was tested in order to try increasing the performance and not having to move the work to the Objective-C environment. Despite implementing and testing the newest and most advanced rendering techniques, the Python prototype still did not perform as required.

Since a more radical increase in performance was expected, a series of tests were conducted to analyze why the Python prototype still did not perform as required, despite using one of the most efficient rendering technique available for OpenGL; Vertex Buffer Objects.

The results of the tests were that the specific problem that the view part of the project had; a lot of small unique objects, were especially ill suited in combination with the Python programming language and also the Vertex Buffer Object rendering technique.

The results also revealed that the most efficient rendering technique for the problem; a lot of small unique objects, is to use Display lists. This rendering technique is also the one that is being used in the current version.

Once the 3d rendering part of the Objective-C implementation had been modified, work
could be started on improving the animation system. Since no one in the project had done anything similar prior to the project, the development could be described as exploratory at best. The major change of the animation system was the simplification of creating and modifying animations. The simplification involved implementing functions that calculated coordinates, speeds and other data relevant to an animation, instead of supplementing them as parameters.

When the animation system had been improved, work began on creating the animations. Two animations were created; the first animation occurs when the grid of images is filling, the other animation occurs when the grid is full.

7.4 The project

7.4.1 The beginning of the project

The project started with an experimental phase of exploring OpenGL. Since the project involved a large OpenGL part, a decent understanding of the OpenGL library was needed. The first experimental OpenGL implementations were conducted on a Python prototype of the view in the system. Since the prototype in Python was built with a model-view-controller pattern, it was relatively straightforward to experiment with different OpenGL implementations.

The OpenGL experimentation phase gave the members of the project an adequate understanding of OpenGL, which enabled them to implement a simple 3d-view. This phase replaced the initial studying of related tools that usually precedes any implementation. The idea for replacing the initial study phase was that learning how to create a simple 3d-view with OpenGL would be easier and faster, if it was learned by doing.

7.4.2 Middle of the project

Since the implementation of the project began very early, and the integration was made quite late, the implementation phase was quite long. The implementation phase was prolonged as the project members had to learn both OpenGL and Objective-C without any prior knowledge. Even if the basic knowledge was obtained quite early, some problems
arose that was not difficult, but took a long time to fix, due to not having a complete understanding of the problem.

7.4.3 End of the project

The integration of the system at Blocket.se occurred in a quite late stage of the project. This was due to the prolonged implementation phase. Even as the implementation of the system had some problems, the minimal requirement from Blocket.se was met, the replacing animation. The filling animation which was not required, but desired, was also implemented. Despite the memory leak problem, Blocket.se was very content with the end result.

Following the integration, a lot more time on the thesis could be allocated. When the integration was finished, the thesis was about one third completed, which meant that there were less time than usual to complete the thesis.

7.5 Suggestions for improvement

Several improvements on the system could be introduced. The most important improvement is to fix the memory leak. Other possible functionalities that could be added to the system are:

- Add new animations.
- Glossy effects on the images.
- Input to the views
- Particle effects

The existing animations could also be polished.

7.6 Summary of the project

The project has been very implementation heavy. The system as it stands, is not much more than what was required. It was important to the group of the project that at
least a minimal system that satisfied Blocket.se could be deployed. This lead to a longer implementation phase, which was already long due to the complete lack of earlier experience in both Objective-C and OpenGL. Still the system is integrated at Blocket.se and they are pleased with the visual component of the system. That the group members also had to learn OpenGL and Objective-C has also been positive, especially OpenGL as it normally can be difficult to learn on their own.
References


Appendix A

Test application

Listing A.1: prestandatest.py

```python
import pygame
import argparse
from tester import *

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Test different rendering algorithms.')
    parser.add_argument('-q', metavar='q', type=int, nargs='+',
                        help='number of textures', default=[1,100,500,1000])
    parser.add_argument('-i', metavar='i', type=int, nargs=1,
                        help='number of iterations.', default=10)

    screen = pygame.display.set_mode((640,480), HWSURFACE | OPENGL | DOUBLEBUF)
    args = parser.parse_args()
    mytest = OpenGLTest(args.i[0], args.q)
    mytest.run()
```

Listing A.2: tester.py

```python
from numpy import *
```
import numpy
import OpenGL.GL
from OpenGL.arrays import numpymodule
from OpenGL.arrays import *
from OpenGL.arrays import vbo
import pygame
import time
from OpenGL.GL import *
from OpenGL.GLU import *
from pygame.locals import *
import time

WHITE = 1,1,1
GREEN = 0,1,0
BLUE = 0,0,1
RED = 1,0,0
YELLOW = 1,1,0
COLORLIST = [WHITE, GREEN, BLUE, RED, YELLOW]

from matplotlib import pyplot
from pylab import *
def bar_output(num_textures, tests, directory, suffix):
    fig = pyplot.figure()
    for i in range(len(num_textures)):
        names = [x for x,y in tests.items()]
        times = [y for x,y in tests.items()]
        names = []
        t = []
```python
pos = arange(4) + 0.5

for k, v in tests.items():
    names.append(k)
    t.append(v[i])
print(pos, array(t))
bar(pos, array(t), align='center')
xticks(pos, array(names))
ylabel('Time (s)')
title("Quads:" + str(numTextures[i]))
pyplot.show()

def plot_output(numTextures, tests, directory, suffix):
    fig = pyplot.figure()
    for name, times in tests.items():
        subplot = fig.add_subplot(111)
        subplot.plot(numTextures, times)

        pyplot.xlabel('Quads')
        pyplot.ylabel('Time (s)')
        pyplot.legend([x for x, y in tests.items()])
        pyplot.show()

class OpenGLTest:
    def __init__(self, iterations, textures, Looping = False):
        self.iterations = iterations
```
self.textures = textures
self.plots = {}
self.loop = 1

def run(self):
    for tex in self.textures:
        self.run_tests(tex)
plot_output(self.textures, self.plots, 'noloop', '')
bar_output(self.textures, self.plots, 'noloop', '')

    self.plots = {}  # name, times[]

for tex in self.textures:
    self.run_tests(tex, True)
plot_output(self.textures, self.plots, 'loop', '')
bar_output(self.textures, self.plots, 'loop', '')

def run_tests(self, textures, loop = False):
    print "\n\n#######\n\nRUNNINGS TEST WITH", textures, " TEXTURES"

loop_iterations = 1
self.tests = []
if loop:
    loop_iterations = textures
    self.tests.append(ArrayTest(1))
    self.tests.append(VBOTest(1))
else:
    self.tests.append(ArrayTest(textures))
    self.tests.append(VBOTest(textures))
self.tests.append(ImmediateTest(textures))
self.tests.append(RetainTest(textures))

if len(self.plots) == 0:
    for test in self.tests:
        self.plots[test.getName()] = []

k = 0
p=0
for i in range(self.iterations):
    color = 0
    k = k+1
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()  # i
    gluPerspective(70,1,1.01,1000)
    gluLookAt(0,20,10 ,0,0,0 , 0,1,0)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    glClearColor(0.0,0.0,0.0,1.0)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    for test in self.tests:
        r,g,b = COLORLIST[color]
        glColor3f(r,g,b)
        glTranslatef(-2,0,0)
        test.render(loop_iterations)
        color += 1
        if color > len(COLORLIST):
            color = 0
        pygame.display.flip()
for test in self.tests:
    print test.get_name() + "  \t: " + str(test.get_time() / float(self.iterations))
    self.plots[test.get_name()].append(test.get_time() / float(self.iterations))

class TestClass:
    def __init__(self, name = "TEST"):  
        self.time = 0 
        self.start_time = 0 
        self.timer_running = False 
        self.name = name 
    def render(self):
        print "not implemented yet" 
    def start_timer(self):
        if not self.timer_running:
            glFinish() 
            self.start_time = time.time() 
            self.timer_running = True

else:
    print "timer already started" 
    def stop_timer(self):
        if self.timer_running:
            glFinish() 
            self.time += time.time() - self.start_time 
            self.timer_running = False 
        else:
            print "no timer started" 
    def get_time(self):
return self.time
def get_name(self):
    return self.name
def get_result(self):
    return TestResult(self.name, self.time, self.textures, 0)

class ArrayTest(TestClass):
    def __init__(self, textures):
        TestClass.__init__(self, "Array")
        self.textures = textures
        self.a = numpy.zeros((textures*4, 3), 'f')
        self.createArray()

def render(self, loop = 1):
    self.start_timer()
    for i in range(loop):
        glPushMatrix()
        if loop != 1:
            glTranslatef(0,0,(i/10.0))
        glEnable(GL_VERTEX_ARRAY)
        glVertexPointer(3, GL_FLOAT, 0, self.a)
        glDrawArrays(GL_QUADS, 0, self.textures*4)
        glDisable(GL_VERTEX_ARRAY)
        glPopMatrix()
    self.stop_timer()

def createArray(self):
step = 0
for i in range(self.textures):
    self.a[step,0] = 1.0
    self.a[step,1] = 0.0
    self.a[step,2] = 0.0+(i/10.0)

    self.a[step+1,0] = 1.0
    self.a[step+1,1] = 1.0
    self.a[step+1,2] = 0.0+(i/10.0)

    self.a[step+2,0] = 0.0
    self.a[step+2,1] = 1.0
    self.a[step+2,2] = 0.0+(i/10.0)

    self.a[step+3,0] = 0.0
    self.a[step+3,1] = 0.0
    self.a[step+3,2] = 0.0+(i/10.0)
step+=4

class VBOTest(TestClass):
    def __init__(self, textures):
       TestClass.__init__(self,"VBO")
       self.textures = textures

        """self.vbo = vbo.VBO(array(
            [ [0.0, 0.0, 0.0 ],
              [1.0, 0.0, 0.0 ],
              [1.0, 1.0, 0.0 ],
              [0.0, 1.0, 0.0 ]
            ], 'f' )))"""
def createArray(self):
    step = 0
    for i in range(self.textures):
        self.a[step, 0] = 1.0
        self.a[step, 1] = 0.0
        self.a[step, 2] = 0.0 + (i / 10.0)
        self.a[step + 1, 0] = 1.0
        self.a[step + 1, 1] = 1.0
        self.a[step + 1, 2] = 0.0 + (i / 10.0)
        self.a[step + 2, 0] = 0.0
        self.a[step + 2, 1] = 1.0
        self.a[step + 2, 2] = 0.0 + (i / 10.0)
        self.a[step + 3, 0] = 0.0
        self.a[step + 3, 1] = 0.0
        self.a[step + 3, 2] = 0.0 + (i / 10.0)
        step += 4

def render(self, loop=1):
    if loop:
        self.start_timer()
        for i in range(loop):
            glEnableClientState(GL_VERTEX_ARRAY)
            glPushMatrix()
            if loop != 1:
```python
glTranslatef(0,0,(i/10.0))

glDrawArrays(GL_QUADS,0,self.textures*4)
glPopMatrix()
glDisableClientState(GL_VERTEX_ARRAY)
self.stop_timer()
else:
    self.start_timer()
glPushMatrix()
glEnableClientState(GL_VERTEX_ARRAY)
self.vbo.bind()
glDrawArrays(GL_QUADS,0,self.textures)
glPopMatrix()

self.vbo.unbind()
glDisableClientState(GL_VERTEX_ARRAY)
self.stop_timer()

class RetainTest(TestClass):
    def __init__(self, textures):
        TestClass.__init__(self, "Display List")
        self.textures = textures
        self.create_stored_list()

    def render(self, loop = 0):
        self.start_timer()
        for j in range(self.textures):
            glPushMatrix()
            glTranslatef(0,0,(j/10.0))
glCallList(1)
```
glPopMatrix()
self.stop_timer()

def create_stored_list(self):
  glNewList(1, GL_COMPILE)
  glBegin(GL_QUADS)
  glVertex3f(0.0, 0.0, 0.0)
  glVertex3f(1.0, 0.0, 0.0)
  glVertex3f(1.0, 1.0, 0.0)
  glVertex3f(0.0, 1.0, 0.0)
  glEnd()
  glEndList()

class ImmediateTest(TestClass):
  def __init__(self, textures):
    TestClass.__init__(self, " glBegin/glEnd")
    self.textures = textures

  def render(self, loop = 0):
    self.start_timer()
    for j in range(self.textures):
      glPushMatrix()
      glTranslatef(0, 0, (j/10.0))
      glBegin(GL_QUADS)
      glTexCoord2f(0.0, 0.0)
      glTexCoord2f(1.0, 0.0)
      glTexCoord2f(1.0, 1.0)
      glTexCoord2f(0.0, 1.0)
      glEnd()
glVertex3f(0.0, 0.0, 0.0)
glVertex3f(1.0, 0.0, 0.0)
glVertex3f(1.0, 1.0, 0.0)
glVertex3f(0.0, 1.0, 0.0)
glEnd()

glPopMatrix()
self.stop_timer()