Mobile Phone 3D Games
with OpenGL ES 2.0

E R I K  O L S S O N

KTH Computer Science
and Communication

Master of Science Thesis
Stockholm, Sweden 2008
Mobile Phone 3D Games
with OpenGL ES 2.0

ERIK OLSSON

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2008
Supervisor at CSC was Henrik Eriksson
Examiner was Lars Kjelldahl

TRITA-CSC-E 2008:025
ISRN-KTH/CSC/E--08/025--SE
ISSN-1653-5715
Abstract

The goal of this master's project was to investigate the possibility of creating a visually advanced 3D game for mobile phones with the use of the OpenGL ES 2.0 library. It also had the intention of evaluating how game and shader development is affected by the performance and design limitations of the platform. A 3D game engine as well as a game prototype was developed on top of a OpenGL ES 2.0 emulator library to act as a test platform.

The conclusion is that while impressive visual effects in mobile 3D games may very well be possible with the arrival of new mobile phone 3D graphics hardware with support for OpenGL ES 2.0, it is too early to predict the complexity possible for the effects, as no such hardware is currently publicly available. However, 3D graphics may very well be successfully used on mobile phones, especially in combination with shaders, not only for visual appeal, but also for flexibility in development. Furthermore, when designing games in general for mobile phones consideration has to be taken to the inherent limitations due to hardware design, such as screen and connectivity limitations.
Spel till mobiltelefon med OpenGL ES 2.0

Sammanfattning


Slutsatsen är att även om det mycket väl kan bli möjligt med imponerande visuella effekter tack vare ny 3D-grafikhårdvara med stöd för OpenGL ES 2.0 i kommande mobiltelefoner, så är det för tidigt att säga hur avancerade effekterna kommer att kunna vara då det ännu inte finns någon dylik hårdvara tillgänglig för allmänheten. 3D-grafik, speciellt kombinerat med shaders, kan dock absolut användas framgångsrikt på mobiltelefoner, inte bara för visuell tilltalan, utan även för flexibilitet i utvecklingen. Vidare är det alltid viktigt att tänka på begränsningarna i hårdvarudesignen när man designar spel till mobiltelefoner, till exempel begränsningarna i skärm och uppkopplingsmöjligheter.
# Contents

1 Introduction.............................................................................................................1
   1.1 Problem Statement..............................................................................................1
   1.2 Cooperation.........................................................................................................1
   1.3 Thesis Outline.....................................................................................................1

2 3D Game Engine Overview....................................................................................3
   2.1 Resource Handling..............................................................................................3
      2.1.1 Models..........................................................................................................3
      2.1.2 Textures.........................................................................................................4
      2.1.3 Shaders...........................................................................................................5
      2.1.4 Materials........................................................................................................6
      2.1.5 Animations.....................................................................................................6
   2.2 Scenegraphs.......................................................................................................6
   2.3 Rendering...........................................................................................................7
      2.3.1 Methods..........................................................................................................8
      2.3.2 View Frustum Culling....................................................................................9
      2.3.3 Occlusion Culling.........................................................................................10
      2.3.4 Spatial Acceleration Structures....................................................................11
      2.3.5 Hardware Specific Optimisations.................................................................12

3 Graphics Libraries................................................................................................13
   3.1 OpenGL..............................................................................................................13
      3.1.1 Versions 1.0 – 1.5.......................................................................................13
      3.1.2 Versions 2.0 – 2.1.......................................................................................15
   3.2 OpenGL ES.........................................................................................................15
      3.2.1 Versions 1.0 – 1.1.......................................................................................16
      3.2.2 Version 2.0....................................................................................................16

4 Graphics Hardware................................................................................................19
   4.1 PC Hardware.......................................................................................................19
   4.2 Mobile Hardware...............................................................................................19

5 Approach...............................................................................................................23
   5.1 Game Engine.......................................................................................................23
   5.2 The Demon.........................................................................................................24
   5.3 3D Kodo.............................................................................................................25

6 Evaluation..............................................................................................................29
   6.1 Why 3D and Shaders?.......................................................................................29
   6.2 Mobile Phone Game Design Limitations..........................................................29
      6.2.1 Multiplayer Limitations...............................................................................29
      6.2.2 Input Limitations.........................................................................................30
      6.2.3 Output Limitations.......................................................................................31
   6.3 Shader Techniques.............................................................................................31

7 Conclusions............................................................................................................35

Bibliography............................................................................................................37

Appendix A: Shader Sample Code..........................................................................41
Chapter 1 - Introduction

1 Introduction

With the market for PC and console games growing rapidly, game developers have been looking for alternative platforms for games for quite some time. One such platform is the mobile phone. So far the only games for mobile phones have been 2D-games and visually simple, most often software-rendered 3D-games. However, 3D graphics hardware manufacturers are starting to explore the potential of the platform, and several 3D graphics chips with shader support for mobile phones are in development by different manufacturers. Supporting the relatively new OpenGL ES 2.0 programming interface, these chips will enable mobile phone games to utilise visual effects that have so far been difficult or impossible to implement effectively on the platform.

The goal of the project was to create a 3D game engine on top of an OpenGL ES 2.0 emulator library currently available with the intention of creating a game prototype and evaluate the capabilities of the platform with regards to effects in the form of shader programs. It will also evaluate the limitations of the platform compared to PC and game consoles, for example input and output limitations.

1.1 Problem Statement

Under the above premises, I formulate my main research question for the thesis:

How can three-dimensional graphics and shaders be successfully used in mobile phone games?

However, since the mobile phone platform has other limitations than graphics, I also formulate a secondary question:

What limitations does the mobile phone platform enforce on game design?

1.2 Cooperation

Since the project was a collaboration between two students, Erik Olsson and Mikael Gustavsson, chapters 2 through 4 are concerted. The remainder of the thesis is individual, and work has been divided with different focus according to:

Mikael focused on the construction of a 3D graphics engine based on the OpenGL ES 2.0 graphics library.

Erik focused on evaluating the limitations of the platform from the experiences gained while creating the game prototype.

1.3 Thesis Outline

The thesis can be divided into two main parts: background and implementation/evaluation. The background part of the thesis, chapters 2 through 4, is aimed at readers with little or no experience of 3D game engines, graphics libraries and graphics hardware. If you have previous knowledge in any or all of these areas you can consider omitting said chapters.
Chapter 1 - Introduction

Background provides some background information about graphics libraries, 3D game engines and 3D graphics hardware in general and is referred to from other parts of the thesis. The background part contains the following chapters:

3D Game Engine Overview
This chapter describes 3D game engines in general, as well as the different parts they are commonly constructed of.

Graphics Libraries
This chapter describes the OpenGL and OpenGL ES graphic libraries, including an overview of the version histories.

Graphics Hardware
A description of the development of graphics hardware, both for mobile phones and PC for comparison.

Implementation/evaluation describes the details of our implementation of the game prototype, as well as an evaluation of the results. The chapters of this part is as follows:

Approach
Implementation details, both of the game prototype and briefly of the 3D graphics engine.

Evaluation
A discussion of the limitations of the hardware as well as of some choice shaders and their usability on the platform.

Conclusions
Conclusions of the work and evaluations.
2 3D Game Engine Overview

A 3D Game Engine is a software component which is intended to perform certain tasks, such as handling resources, scenegraphs and rendering.

2.1 Resource Handling

Resources are the content of the scene, i.e. what eventually is drawn on the screen. Resources may include models, textures, shaders, materials and animations.

2.1.1 Models

Models are the geometries of the scene, which are bound to objects in the scenegraph. Models can be generated through code, but most often they are read from a file, which usually has been exported from a modelling application, such as Autodesk's Maya. The geometry consists of vertex data, as well as a surface description which describes how vertices are connected to create primitives such as polygons, triangle strips, lines or points. Figure 1 below shows an example model.

![Figure 1: A model of the famous Utah Teapot, shown here in wireframe mode](image)

Vertices have different types of attributes, one of which is position. Other common attributes are normals, texture coordinates, colours, tangents, binormals, bone weights and bone indices. A vertex normal defines the surface normal at a specific position and is most often used for lighting calculations. Texture coordinates are used to map a texture image onto the surface. Tangents and binormals, together with normals, form the basis of tangent space, which is sometimes referred to as surface space. Tangent space is used for bump map lighting calculations. Bone weights and bone indices can be used to deform a geometry in a non-rigid way, such as bending an arm of a character. This is called skinning, and is most often used in animation.
2.1.2 Textures

Originally, textures were two-dimensional images which were mapped onto surfaces. Nowadays they are better thought of as general containers of data to be used during rendering. Textures can be one-, two- or three-dimensional. When data is read from a texture it is usually filtered to make the output continuous and reduce aliasing.

Textures can be used for a number of different purposes, for example:

- **Diffuse map** – defines the colour of the surface, i.e. the original use of textures. By far the most common. See figure 2 above.
- **Detail map** – adds more fine-grained details than a diffuse texture and is usually repeated across the surface with a high frequency.
- **Specular map** – defines the reflectiveness of the surface. Usually monochrome.
- **Emissive map** – defines the light emittance of the surface, which enables the surface to glow regardless of external light sources.
- **Ambient occlusion map** – defines the accessibility of the surface and has to be calculated with regards to the surrounding geometry. Points surrounded by a large amount of geometry has low accessibility and becomes dark.
- **Normal map** – stores surface normals, which can be used in bump mapping to give the illusion of a much more detailed geometry than what is actually used.
- **Light map** – stores a pre-calculated lighting of the scene. This technique is on the decline due to the increasing dynamic nature of game worlds.
- **Depth map** – stores the depth of a scene as seen from some view. Often rendered from a light position to be used in shadow calculations.
Chapter 2 - 3D Game Engine Overview

- **Environment map** – stores a view of the environment as seen from an object or the origin of the scene, either in the form of a *sphere map* (for instance a photograph taken with a fish-eye lens) or as a *cube map* (a set of 6 images, one for each direction along the coordinate axes). Environment maps are usually mapped on objects to make them appear reflective.

Generally, hardware limits how many textures that can be used simultaneously when rendering in real time. Textures can have a number of different *channels*, grey-scale textures, also known as luminance textures, only have one channel, while colour textures most often has four; red, green, blue and alpha (RGBA).

### 2.1.3 Shaders

Shaders are short programs that are usually executed on a graphics processing unit (GPU). They combine the power of dedicated hardware with the versatility of a software renderer. Different shader units are arranged in a pipeline, a typical example can be seen in figure 3. Shaders receive data in the form of attributes and uniforms. Attributes vary with every element that is processed and are provided either from the previous shader in the pipeline or by the engine. Common attributes are described in chapter 2.1.1. Uniforms on the other hand vary at most once per draw call. Typical examples of uniforms are object properties such as position, texture bindings and material properties. On modern hardware there are up to three types of shader units available: *vertex shaders*, *geometry shaders* and *fragment shaders*.

**Figure 3: A shader pipeline**

*Vertex shaders* operate on individual vertices, and receive vertex attributes from the engine. The vertex shader can generate or modify any vertex attributes, such as position, colour or texture coordinates. Common usages include making a tree's branches sway in the wind, moving raindrops and skinning a character model.

*Geometry shaders* operate on individual primitives, such as polygons, points or lines and receive input from a vertex shader. The geometry shader can emit zero or more primitives. Common usages include geometry tessellating, generating particle polygons from points or extruding shadow volumes.

*Fragment shaders*, sometimes referred to as *pixel shaders*, operate on individual fragments. The output of the fragment shader is the colour of a pixel written to the *frame buffer*. The fragment shader also has the ability to discard a fragment so that it is not written to the frame buffer. Common usages include per pixel lighting, bump mapping and reflections.

When shader hardware is not available, a *fixed function pipeline* must be used. This pipeline can usually be set up to perform basic calculations such as per vertex lighting, rigid transformations and blending of several textures. Many effects can be accomplished both with and without shaders, but shaders provide a much wider range of effects and better image quality.
2.1.4 Materials

A material is basically a collection of textures, shaders and uniform values. Materials are often exported from modelling applications or from shader studio applications such as NVIDIA's FX Composer or AMD's RenderMonkey. These applications however create and modify what they refer to as effects. Effects are best thought of as material templates; for example, a fur effect might be used in several fur materials with different colours or textures.

2.1.5 Animations

Animations can be divided into three groups, node animations, blend shapes and bone animations, and span a certain number of frames. Animation data is specified either for each frame, or at a number of keyframes. When a keyframed animation is played back, data of two adjacent keyframes are interpolated to produce data for the frames in-between in order to keep the animation fluent.

Node animations modifies the position, rotation or scale of nodes in the scenegraph. Less commonly used in games, where it is primarily adopted for cut scenes and camera movements.

Blend shapes are a sequence of models, which define the positions of the object's vertices. Require much memory but little processing power, and were often used to animate characters in older games.

Bone animations modifies the position, rotation or scale of certain nodes known as bones. Vertices in a model are linked to one or many of these bones in order to follow their movement. This technique is known as skinning. Bones are usually linked in a hierarchical manner to affect each other, mimicking the behaviour of, for example, a skeleton.

2.2 Scenegraphs

A scenegraph is a data structure that arranges the logical and often spatial representation of a graphical scene [1]. A scenegraph contains a number of linked nodes, usually in such a way that a node can have multiple child nodes, but only one parent, thus making it a directed graph, see figure 4 for a simple example scenegraph. In order to be useful, the graph should also be acyclic. Nodes can be divided into two categories, group nodes, which may have children, and leaf nodes, which may not. There can be numerous types of nodes, for instance transform nodes, object/model nodes, light nodes, camera nodes and emitter nodes for particle systems.
Chapter 2 - 3D Game Engine Overview

A transform node is a group node which represents a transform relative to its parent node. This arranges the scene in a hierarchical structure, which is useful for numerous reasons, such as moving a complex object by only moving the parent node.

An object node, or a model node, is a leaf node that represents a graphical object that can be rendered. It has references to a mesh and a material resource.

All leaf nodes, such as those for objects, lights, cameras and emitters receive transforms from a parent transformation node.

Scenegraphs are built and modified as the game world changes, but parts are often loaded from files that have been exported from a modelling application or a custom world editor.

2.3 Rendering

In 1986, Jim Kajiya introduced the rendering equation [3], which is a general integral equation for the lighting of any surface.

\[
L_o(x, \hat{o}) = L_e(x, \hat{o}) + \int_{\Omega} f_r(x, \hat{o}', \hat{n}) L_i(x, \hat{o}') (\hat{o}' \cdot \hat{n}) d\hat{o}'
\]

The equation describes the outgoing light \( L_o \) in any direction from any position on a surface, which is the sum of the emitted light \( L_e \) and the reflected light. The reflected light itself is the sum of the incoming light \( L_i \) from all directions, multiplied by the surface reflection and cosine of the incident angle. All methods of calculating lighting in modern computer graphics can be seen as approximations of this equation.

There are several ways of rendering a scene, such as ray-tracing and radiosity. Such methods allow for advanced lighting effects and global illumination [4]. Global illumination takes the environment into consideration so that effects such as reflections, refractions and light bleeding are possible. However, global illumination is generally
considered too slow to be applied to games. This section will hence focus on using hardware accelerated rasterisation [4], which is the most common method employed by games. Although rasterisation only directly supports local lighting effects, which only considers the actual surface and light sources, modern games include many global effects such as shadows and reflections. This, however, makes the process of rendering a modern game a very complex task.

2.3.1 Methods

Drawing relies on the use of a depth buffer, also called z-buffer, which is a buffer that associates a depth value with each pixel in the frame buffer [1],[4]. This allows the drawing of objects to be performed in any order, since a pixel is written to the frame buffer only if it is closer to the viewpoint than what is currently stored in the depth buffer. This is called hidden surface removal.

The basic way of rendering a scene is as follows:

1. Traverse the scene graph in a depth-first-order, concatenating node transforms with the resulting parent transform.
2. When an object node is reached, draw the object using its associated material and model.

This approach unfortunately has several problems, some of which being that it cannot render dynamic lights, dynamic shadows or transparent objects correctly. To handle dynamic lights, the light node transforms have to be known before any objects are drawn. Dynamic shadows are even more problematic since they require the use of several rendering passes. Due to the nature of the depth buffer on current graphics hardware, transparent objects has to be sorted back-to-front and drawn after the opaque objects. The following method is an example of how to address these problems:

1. Traverse the scene graph and concatenate the node transforms, put the lights and their transforms in a list, and put all objects and their transforms in another list.
2. For each light that casts shadows, render the scene as seen from the light source to one or several depth maps.
3. Sort the list of objects, so that transparent objects are sorted back-to-front and placed after opaque objects.
4. Draw the objects, using information from the list of lights and the light's depth maps for lighting and shadow calculations.

There are several alternatives to this method, mainly related to lighting and shadows. There are two common methods of drawing dynamic shadows in games, shadow mapping and shadow volumes [4]. Shadow mapping uses the depth maps generated in step 2, shadow volumes do not.

While the method listed above calculates lighting per-object-per-light (POPL); the alternative, per-light-per-object (PLPO), is also common. If this method is used, the fourth step in the previous method is replaced with the following two steps:

4. Draw the objects lit by the ambient lighting in the scene.
5. For each light, draw the scene again lit by only this light, additively blending the result into the frame buffer.

This method is compatible with both shadow mapping and shadow volumes, whereas
the previous method only supports shadow mapping. However, it also requires the scene to be rendered once per light. A method that does not have this performance drawback is deferred shading, first suggested in a paper from 1988 by Michael Deering et al. [5], although the term “deferred” is never used in the paper. The method modifies PLPO as such:

4. Draw the objects lit by the ambient lighting in the scene. At the same time, also draw additional information of the fragments such as position, normal, and material information to extra frame buffers, these are collectively called the g-buffer.

5. For each light, draw a light geometry (spheres for point lights, cones for spot lights) of a reasonable size (a reasonable size would be the distance at which the light's contribution becomes negligible) to determine what parts (if any) of the visible geometry in the current rendering that should be affected by this light. These light geometries are drawn with a fragment shader that reads scene information from the g-buffer, calculates the light contribution and additively blends the result into the frame buffer.

Even though this method has been known for quite some time, it is still sparsely used in games since hardware that can support it has only recently become generally available. All of these methods have advantages and disadvantages, POPL only draws the scene once but the shaders become very complex or numerous since both material characteristics and multiple lights have to be handled in a single pass. PLPO is the exact opposite, the shaders are simpler but the scene has to be drawn multiple times. Deferred shading seems to solve this problem since it has simple shaders and only draws the scene once. However, the g-buffer is only possible to implement on the latest hardware and has high memory requirements.

### 2.3.2 View Frustum Culling

Game worlds are often large, potentially containing tens of thousands of objects. Since only a part of the world is normally visible at any time, rendering can be optimised by discarding geometry outside of the view frustum, this is called frustum culling [1]. Such a frustum has the geometrical shape of a square pyramid delimited by a near and far viewing plane, as shown in figure 5. On older hardware, when the number of polygons in scenes were lower and rasterising was slower, culling was often done on a per polygon basis. On modern hardware, where geometry is often stored in dedicated graphics memory, culling is normally done per object.

![Figure 5: No culling (left) and view frustum culling (right) [6]](image)
To speed up the culling of objects, the actual mesh geometry is usually not used but an enclosing bounding volume [7]. The most common object bounding volumes are spheres, boxes aligned to the coordinate system axes (Axis Aligned Bounding Box, or AABB) and boxes aligned to the object (Oriented Bounding Box, or OBB). Frustum culling is also used when rendering depth maps to be used in shadow mapping. A notion related to frustum culling is frustum clipping where polygons that straddle the frustum planes are split so that parts outside the frustum are discarded. This is done by modern hardware in the process of rasterisation and not something that engine programmers normally have to be concerned with.

2.3.3 Occlusion Culling

While view frustum culling potentially greatly reduces the number of non-visible objects that are drawn, it does not hinder the drawing of objects occluded by other objects. A further optimisation would therefore be to cull even such objects (see figure 6). This is not to be confused with the hidden surface removal performed with the depth buffer (although this can be seen as occlusion culling on a per-pixel-basis), as occlusion culling only is an optimisation to discard objects that will not contribute to the resulting image.

There are a number of methods of accomplishing this, worthy of mention are potentially visible set, portal rendering and hardware occlusion queries.

Potentially visible set divides a scene into regions and pre-computes visibility between them. This allows for quick indexing to obtain high quality visibility sets at runtime. However, since it is a pre-computation, changes to the objects in the scene are not possible.

Portal rendering divides a scene into sectors (rooms) and portals (doors), and computes visibility of sectors at runtime by clipping them against portals [7]. This is naturally best suited for small, indoor scenes, and will have little impact on large, outdoor scenes where there are no clear portals.

Hardware occlusion queries are a way of asking the graphics hardware if any pixels were drawn during the rendering of a particular object [8]. That way, it is possible to simulate rendering of the bounding volume of an object to see if the object is currently occluded (i.e. no pixels would be drawn), and if so, that object can safely be skipped.
This method works on dynamic scenes and without any pre-computation, but requires modern hardware and causes some overhead due to the additional draw calls.

2.3.4 Spatial Acceleration Structures

View frustum culling and occlusion culling minimises the number of objects that are drawn by the graphics hardware. However, culling all the objects in a large world against the view frustum can put a significant burden on the CPU. This and other problems can be alleviated by using a spatial acceleration structure, such as a bounding volume hierarchy (BVH) which is easily integrated with a scenegraph. Two popular BVHs are sphere trees [9] and aabb-trees [10]. This is realised by having every node in the scenegraph store a bounding volume, which encloses all objects in the subtree rooted in the corresponding node. This makes many spatial queries, such as frustum culling, much faster since an entire subtree can be tested without having to test every individual object. However, this technique also has some computational overhead since a subtree of the volume hierarchy has to be updated every time a node in the subtree changes.

Bounding volume hierarchies are simple and can handle dynamic updates fast, but the large amounts of static geometry common in games can be difficult to organise in a hierarchy. Spatial partitioning structures are often used to remedy this problem. Such structures are generally computationally expensive to construct and alter, but allow for very fast handling of spatial queries. Common examples are quadtrees (see figure 7 above), octrees, kd-trees and BSP trees [4]. These structures can be kept separate from the scene graph, or be embedded in the scene graph. Some games use both bounding volume hierarchies and spatial partitioning trees, while others store all data in either.

Figure 7: Octree spatial acceleration structure constructed around two spheres [11]
2.3.5 Hardware Specific Optimisations

Achieving high rendering performance with hardware accelerated graphics can be difficult and require good knowledge of hardware and large amounts of testing. Some general principles that can be addressed by a 3D game engine can however be identified: **minimise state changes, minimise draw calls and minimise stalls.**

Minimising state changes can be done by carefully ordering how objects are drawn. This can be done by sorting objects with consideration to their materials, shaders, textures or geometry. State then only need to be changed when necessary as opposed to fully resetting and setting all state for every object that is to be drawn. In less dynamic games with a small number of objects running on simple hardware, this sorting can be done as a pre-computation. Otherwise, sorting can be done every frame on the objects currently in view (after any frustum and occlusion culling). One problem is transparent objects since they need to be drawn after opaque objects and preferably in strict back-to-front order, this makes minimising state changes difficult and is one reason to cut back on the number of transparent objects. Other possible ways of minimising state changes are to use fewer (and perhaps more complex) shaders, fewer textures (possibly packaging several textures into texture atlases) or merging geometry data into fewer and larger buffers.

Minimising draw calls is related to minimising state changes. Since a state change can occur between draw calls only, fewer state changes need fewer draw calls. Minimising draw calls is then done by merging objects that use the same state. This sometimes adds considerable complexity, one example is particle systems. The simplest approach is to draw each particle individually since all particles move independently each frame. In practice, this is much too slow and particles should be batched so that even a particle system consisting of thousands of particles is drawn with at most a few draw calls.

Minimising stalls means that the time that the graphics hardware is idle should be minimised. Drawing commands issued from the CPU to the GPU are queued and executed in the **graphics pipeline** (see figures 9 and 10); for optimal utilisation of hardware resources, this queue should never be empty. To address this, care should be taken to schedule CPU computations to occur when the drawing queue is filled. For optimal performance in complex games, multi-threading will probably have to be used. However, even if the CPU to GPU queue is not left empty, internal hardware stalls can still occur due to the pipeline architecture of the hardware. This can happen for two reasons: if a command needs the results of a yet uncompleted command further down the pipeline or if a command requires state changes which are incompatible with commands currently being processed further down in the pipeline. The first scenario can happen if for instance the drawing of an object needs the content of a texture which is currently being written to. The second scenario happens when some state such as blend settings or the active shader need to be changed, this might not be possible to do without waiting for all previous drawing commands to finish. The problem of internal stalls can be addressed by reordering commands so that a command does not need the result of a recent command or tries to update state which is likely to be currently used, minimising state changes and draw calls also helps.
3 Graphics Libraries

There has been many different programming libraries with the purpose of rasterising images. Naturally, many custom solutions have existed within companies and universities, but since hardware accelerated rasterisation became common graphics programmers generally use the libraries provided by the hardware vendors. The two most common libraries are called OpenGL and Microsoft's Direct3D. OpenGL is used in a wider range of applications and is supported on more platforms. Direct3D is mostly used for games, but is more popular than OpenGL in this area.

3.1 OpenGL

OpenGL (Open Graphics Library) is a standard specification defining a cross-platform application programming interface (API) for rendering 2D and 3D computer graphics [12]. OpenGL was originally developed by Silicon Graphics Inc. (SGI) but has since 1992 been governed by the Architecture Review Board (ARB) which consists of representatives from many independent companies. Since 2006, ARB is part of the Khronos group. OpenGL is widely used in Computer-Aided Design (CAD), scientific visualisation, flight simulators and games.

OpenGL has an extension mechanism, this allows implementers of the library to add extra functionality. Applications can query the availability of specific extensions at runtime, making it possible for programs to adapt to different hardware. Extensions allow programmers access to new hardware abilities without having to wait for the ARB to incorporate it into the OpenGL standard. Furthermore, additions to the standard are tested as extensions first to ensure their usability. This is important since all versions of OpenGL are backward compatible, meaning that once a feature is accepted into the standard it is never removed.

3.1.1 Versions 1.0 – 1.5

Version 1.0 of OpenGL was released in 1992 and it provides features such as per-vertex lighting, texturing, fog and blending. Geometry is specified using the begin/end-paradigm (see code listing 1) which is easy to use. OpenGL commands can be grouped together and stored in display lists, which can then be executed repeatedly. This improves rendering speed and allows calls to be organised in a hierarchical manner.

```c
glBegin(GL_TRIANGLES);
glColor3f(1, 0, 0); glVertex3f( 0, 1, 0);
glColor3f(0, 1, 0); glVertex3f(-1, 0, 0);
glColor3f(0, 0, 1); glVertex3f( 1, 0, 0);
glEnd();
```

*Code listing 1: Drawing a triangle in OpenGL using begin/end-paradigm*

OpenGL supports several drawing primitives: points, lines, line strips, line loops, triangles, triangle strips, triangle fans, quadrilaterals (quads), quad strips and polygons. Supported vertex attributes are positions, colours, normals and texture coordinates. Texture coordinates can also be automatically generated in order to save memory or efficiently animate coordinates. OpenGL is often considered to be a state machine, since it has a large number of global states which affect drawing. Examples of states are lighting settings, material settings, current texture, blending mode and matrix
Chapter 3 - Graphics Libraries

transforms. Figure 8 shows how vertex positions in local object space (object coordinates) are transformed into pixel positions in the framebuffer (window coordinates). To learn more about coordinate transforms, there are several books treating the subject, for example *3D Computer Graphics* by Alan H. Watt [4].

The most important addition in OpenGL 1.1 was vertex arrays. Vertex arrays are an alternative to the begin/end-paradigm and makes it possible to store vertex attributes in arrays and draw directly from these. This greatly reduces the number of OpenGL commands needed to draw geometry. The large number of commands of the begin/end-paradigm began to be problematic as rendering hardware became faster and geometric models became more detailed.

Version 1.2 of OpenGL was released in 1998 and added three-dimensional textures and more blending modes among other features.

OpenGL 1.3 was released in 2001 and added several important features. Compressed textures allow textures to be used while stored in a compressed form, reducing memory consumption. Cube maps enable more detailed environment mapping effects. Multitexturing and texture environment settings allow geometry to be mapped with several textures simultaneously which can be combined in several different ways. This allows for much more advanced surface details and can be seen as a primitive form of shaders. For instance, a special combiner mode allows for bump mapping effects. Also, support for fullscreen antialiasing was added.

In 2002 OpenGL 1.4 was released. It added support for depth maps and shadow rendering with shadow mapping. Texture environment settings were made more powerful and additional blending modes were added.

In the following year, OpenGL 1.5 was released and added two important features: occlusion queries and vertex buffer objects (VBO). Vertex buffer objects allow vertex arrays to be stored in dedicated graphics memory. This allows for significantly faster rendering of complex geometry compared to normal vertex arrays. This problem had already been addressed by display lists, but VBOs are simpler for library implementers to optimise and they are better suited to handle dynamic geometry.
3.1.2 Versions 2.0 – 2.1

OpenGL 2.0 was released in 2004. The reason for the change of major version number was the added support for high-level programmable shaders in the form of vertex and fragment shaders. Despite this, it is still backward compatible with older versions. This version also introduced the OpenGL Shading Language (GLSL) [14], a C-like language used exclusively to write shader programs. An interesting feature is that it is possible to use old OpenGL code with shader-based rendering, due to much of the old OpenGL state being automatically available to shaders through special variables. Another important feature added is Multiple Render Targets (MRT), which allows fragment shaders to write to multiple frame buffer objects. This is important for certain advanced effects such as deferred shading.

Version 2.1, released in 2006, introduced some additions to GLSL, as well as Pixel Buffer Objects which expand on the interface provided by the vertex buffer objects allowing buffer objects to be used with both vertex array and pixel data.

3.2 OpenGL ES

OpenGL ES (henceforth known as GLES) is an adaptation of OpenGL for embedded systems [15]. GLES is a subset of OpenGL, some functionality was removed in order to make the library smaller and simpler. However, fixed-point functionality was added since few embedded systems efficiently handle floating-point calculations. GLES was developed by the Khronos group and has two different profiles, Common and Common-Lite. The Common-Lite profile differs from the Common profile primarily in being targeted at a simpler class of graphics systems not supporting high-performance floating-point calculations. The Common-Lite profile supports only commands taking fixed-point arguments, while the Common profile also includes many equivalent commands taking floating-point arguments.
3.2.1 Versions 1.0 – 1.1

The first version of GLES, 1.0, was released in 2003 and is based on OpenGL 1.3, but with some functionality removed, such as

- Begin/end-paradigm. This was deemed obsolete with preference to vertex arrays.
- Quad, quad-strip and polygon drawing primitives. These were removed since these primitives can be assembled from triangles.
- Automatic texture coordinate generation.
- One- and three-dimensional textures and cube maps, leaving only two-dimensional textures.
- Some of the texture environment settings, leaving only the simpler ones such as replace, blend, add and modulate.
- Display lists, since they are complex to implement.

GLES 1.1 was released in 2004 and is based on OpenGL 1.5. The most important additions compared to GLES 1.0 are vertex buffer objects and the support of all advanced texture environment settings in OpenGL 1.5.

3.2.2 Version 2.0

GLES 2.0, the latest version, was finalised in 2007 [16]. It is based on OpenGL 2.0, and like it introduced high-level shaders. However, unlike OpenGL 2.0, GLES 2.0 completely removes all support for the fixed function pipeline. As a result, GLES 2.0 is not backward compatible with previous versions of GLES. Shaders are specified in a version of GLSL called GLSL ES. GLES 2.0 only supports the Common profile, and the fixed point support has been limited to vertex arrays only.

![Diagram](image.png)

*Figure 10: The programmable pipeline of OpenGL ES 2.0 [13]*
The most noteworthy functionality removed from OpenGL 2.0 is

- Begin/end-paradigm.
- Specific vertex arrays for attributes such as positions, normals and texture coordinates. These functions have been removed in favor of the general function `glVertexAttribPointer`.
- Quad, quad-strip and polygon drawing primitives.
- All functionality that modifies the current matrix transforms. The model-view and projection matrices are removed, programmers choose which transforms are needed and pass these as uniforms to shaders.
- Automatic texture coordinate generation. This can be done in vertex shaders.
- All functionality that handles lighting and material state. Programmers implement a custom lighting solution with shaders and feed data to the shaders with generic uniforms and vertex attributes.
- One- and three-dimensional textures, leaving two-dimensional textures and cube maps.
- Texture environment settings, these are not needed since shaders are a direct substitute and much more powerful.
- Fog settings, handled by shaders instead.
- Display lists.

GLES 2.0 directly supports framebuffer objects, something which is only available as an extension even to OpenGL 2.1. Framebuffer objects allow multiple off-screen framebuffers to be created and rendered to. This allows for fast rendering to textures and also supports rendering to textures larger than the on-screen framebuffer. This is important when rendering depth maps, environment maps or in other effects such as reflections, refractions or post processing effects.

GLSL ES is very similar to GLSL, there are two main differences: in GLSL ES, the special variables that are tied to the fixed function pipeline have been removed and keywords for specifying the precision of variables have been added (`lowp`, `mediump` and `highp`).
Chapter 4 - Graphics Hardware

4 Graphics Hardware

The beginning of computer graphics is often attributed to Ivan Sutherland and his program Sketchpad which ran on a vector graphics display monitor with a light pen input device in 1963. At Xerox Parc, the computer mouse was invented and Graphical User Interfaces (GUIs) were first developed. In the 70's, computer graphics hardware became cheaper and was used in the first Apple personal computers. Arcade game machines such as Pong or Pac-Man became popular. During the 80's, the IBM PC was introduced and workstations from SGI began supporting real-time hardware rasterisation of lines and polygons. During the 90's, SGI workstations that accelerated 3D rasterisation appeared and then were replaced by cheaper IBM PCs fitted with 3D graphics cards. Game consoles with 3D graphics appear and photo-realistic computer graphics effects are introduced in Hollywood films. In the 00's PC 3D hardware continues to improve and hand-held devices with hardware accelerated graphics emerge.

4.1 PC Hardware

Hardware accelerated 3D graphics on the PC became popular with the introduction of the 3dfx Voooodoo chipset in 1996. Graphics cards not only accelerating rasterisation, but also vertex transforms and lighting calculations were introduced with the GeForce 256 in 1999. Simple vertex and fragment shaders were introduced in 2001, the first cards that supported advanced shaders were introduced in 2003. These cards have dedicated hardware for vertex and fragment shaders. The next important evolutionary step happened in 2006 when cards which supports geometry shaders appeared. These cards have a unified shader architecture which means that the cards have a number of more general processing elements which can process fragments, vertices or geometry. Different tasks are automatically assigned to these processing elements, so that processing power is dynamically allocated to fit the processing needs of the application.

Today, the main manufacturers of graphics cards focused on 3D games on the PC are NVIDIA and AMD (previously ATI). Their cards support both Direct3D and OpenGL. For later comparison reasons a NVIDIA GeForce 8600 GT, a modern medium/high range graphics card, supports OpenGL 2.1 plus geometry shaders and can draw more than 4000M pixels/s and 700M vertices/s.

4.2 Mobile Hardware

Mobile hardware is significantly less powerful than a PC because of cost, size and power constraints. While mobile phones capable of hardware accelerated 3D are still rare, Ed Plowman, Product Manager at ARM, predicts in an interview with 3D-Test [19] that mobile 3D graphics acceleration will become more and more common up to the point where a “mass market explosion” will occur somewhere between the middle of 2008 and the middle of 2009. After this point, hardware accelerated 3D graphics will be expected on all new mobile phones. This might seem optimistic, after all – not everyone are interested in games. However, the same hardware that powers 3D graphics will also support better looking and faster 2D graphical interfaces and support decoding of movies, images and sounds. Dedicated graphics hardware generally also consumes less power than if a general purpose processor would handle the same task.
One example is the Nokia N93 which was introduced in 2006 and is one of the few mobile phones that support hardware accelerated OpenGL ES 1.1 [20]. The N93 contains a PowerVR MBX GPU; which, according to the manufacturer Imagination Technologies, can draw 300M pixels/s and 2M polygons/s [21]. Benchmark tests by GLBenchmark [22] indicates a lower actual performance; 870 000 shaded and coloured triangles/s on the N93. The processor is a 32-bit ARM11 330MHz CPU with floating point support. The phone has a 2.4 inch screen with a resolution of 240 x 320 pixels (QVGA).

No OpenGL ES 2.0 capable mobile phones have yet been announced, however, some information is available from press releases and product specifications. Peter Lykke Nielsen, product manager at Nokia, states in a lecture at Game Developers Conference China 2007 [23] that OpenGL ES 2.0 capable phones probably will be available in the second quarter of 2008. A possible graphics chip to be included in those phones is the PowerVR SGX, the successor to the MBX chip. The SGX chip will support OpenGL ES 2.0, and Imagination Technologies states that it will also support geometry shaders, which are not included in ES 2.0 – most likely this feature will be accessible through OpenGL extensions. The SGX chip will be available in many variants and performance is said to lie in the range of 100M – 4000M pixels/s and 2M – 100M polygons/s [24]. The most powerful versions of the chip will most likely not be used in mobile phones, though it seems probable that the theoretical performance in phones will reach 20M polygons/s which would mean an order of magnitude increase compared to the previous generation. In any case, these numbers are very high as they even approach the PC hardware listed in the previous section.

While the most common resolution for mobile screens are 240 x 320 pixels, there are a few exceptions, for instance the Toshiba G900 which has a resolution of 480 x 800 pixels, called WVGA [25]. Such higher resolutions might become more common with the new graphics chips – assuming a resolution of 480 x 800, 60 frames per second and an overdraw factor of 2 a graphics chip would need to handle $480 \times 800 \times 60 \times 2 = 46M$ pixels/s. Taking into account that actual performance will be lower than the theoretical numbers specified by the manufacturer, it still seems very possible that graphics chips such as the SGX will be able to handle a game running in such a resolution.

Besides Imagination Technologies, manufacturers of 3D graphics hardware for mobile phones include NVIDIA, AMD and ARM.

NVIDIA makes the GoForce chips targeted at hand-held devices. The 4800 and 5500 versions of GoForce support OpenGL ES 1.1 and has been included in a number of devices. Nathan Kirsch at Legit Reviews states in a preview of the GoForce 5500 [26] that it is able to render the PC game Quake 3 Arena from 1999 at a 1024 x 768 resolution at 28-35 frames/s. One of the devices the GoForce 5500 is included in is the O2 XDA Flame which, however, does not score very well on GLBenchmark where it only runs the OpenGL ES 1.0 and not the 1.1 tests. Furthermore, it gets much lower scores than the Nokia N93 (225 000 shaded and coloured triangles/s versus 870 000) [22]. There is little information about upcoming chips from NVIDIA except that there is OpenGL ES 2.0 compliant hardware in development.

AMD's hand-held graphics chips are named Imageon. The Imageon 2380 and 2388 were released in 2006 and supports OpenGL ES 1.1 [27]. These chips are not known to be used in any consumer products. AMD is most likely developing OpenGL ES 2.0
Hardware, some indications of this is that AMD has incorporated GLSL ES support into the AMD RenderMonkey shader development application and released a OpenGL ES 2.0 PC emulator library.

ARM is a very successful producer of processors that are used in many mobile phones. ARM also offers two 3D graphics chips: the Mali 55 with OpenGL ES 1.1 support and the Mali 200 with OpenGL ES 2.0 support [28]. In the interview mentioned above, Ed Plowman also states that the Mali 55 can render up to 1M polygons/s [19]. Like the Imageon chips, the Mali chips are not known to be used in any consumer products.
5 Approach

Mikael and I started by studying OpenGL 2.1 and OpenGL ES 2.0 in order to get a better understanding of OpenGL and shaders. Our goals were to examine how to develop mobile games with OpenGL ES 2.0 and how three-dimensional graphics and shaders can be successfully used in mobile phone games. In order to do this we decided to develop a game prototype of a successful two-dimensional game named Kodo and adapt it to three dimensions with shader effects. In this multiplayer game, each player controls a small creature, a kodo, and the objective of the game is to eat the other kodos by moving into them. Figure 11 shows a screenshot of the game.

Since there were no OpenGL ES 2.0 3D game engines available to us, we had to develop one. We investigated if it would be possible to adapt an available PC OpenGL engine but were concerned that they were too complex to be easily adapted to a handheld device. However, we also estimated that we did not have enough time to develop a complete 3D game engine from scratch as part of the project. We decided to handle this problem by trying to keep the engine as small and simple as possible, only supporting the features that we needed to make the prototype. In addition, we had substantial help from the PowerVR OpenGL ES 2.0 SDK examples [30].

5.1 Game Engine

Since our graphics engine would be based on shaders, we needed a way for the game to commune with the shaders directly. This was accomplished by utilising the uniform variables of the shaders. Aside from some standard uniforms with some specific functionalities, we also implemented a way to set custom, user defined uniforms, both globally and per object and material. These can be added or altered dynamically at runtime by the game code, or loaded from files. If a shader were to request a certain uniform, outside the standard range (i.e. a custom uniform), the engine will try to find that uniform, first in the current object itself, then in the object's material, and lastly among the globally defined. However, if the uniform cannot be found at all, the shader cannot run (meaning the object won't be drawn), and thus it is important to set default values for all custom uniforms. Default values are preferably defined as global uniforms, since these will only be used if the uniform is undefined in both the object and material.
Chapter 5 - Approach

For a more detailed description of the implementation of the game engine, refer to Mikael Gustavsson's report *Developing 3D Mobile Games with OpenGL ES 2.0* [31].

From the start of the project we had hopes of acquiring OpenGL ES 2.0 capable hardware. This would allow us to truly test what next-generation mobile phones will be capable of in terms of graphics. However, as it turned out, there were neither any mobile phones with OpenGL ES 2.0 hardware nor publicly available development kits available, and thus we could only test the engine with emulator libraries.

5.2 The Demon

Once we realised we wouldn't receive any OpenGL ES 2.0 capable hardware for the duration of the project, we started to explore other options. We decided to modify the engine to support OpenGL ES 1.1 as well, which would mean we should be able to run the engine on a mobile phone. Being the only phone with hardware accelerated graphics available at the time, the Nokia N93 was an obvious choice for test platform. Since we mainly wanted to test porting an application to a phone, as well as test the hardware capabilities, we decided not to make a playable game, but merely a short demo. Jadestone provided us with an animated model of a demon for this purpose. The demon was animated with bones, which also provided an opportunity to test the matrix palette extension to perform skinning.

Eventually, after modifying the engine and creating the wrapper for Symbian, we managed to get the demo to run on the phone. The demo was however looking rather dull, and in order to make it more visually appealing, as well as test the limits of the hardware, we added a post processing glow effect, which would make the demon appear to emit light from certain parts of its body. This technique utilises an emissive map to define what parts of the demon should glow, and in what colours. The demon is rendered to a glow texture using this map. The scene can then be rendered normally, and by drawing the glow texture blended on top of the rendered scene, slightly stretched outwards from the centre of the texture, a slight glow effect is achieved. By drawing the glow texture repeatedly, each time stretched further, the glow could be increased.

![Figure 12: The demon without glow passes, with five and with nine glow passes](image)

While this technique may provide decent results, it is under no circumstances perfect; the biggest disadvantage is that the light is emitted from the centre of the image, since the glow texture simply is stretched outwards. In figure 12 above, this can be seen at the
lower part of the image, where the light is emitted parallel to the demon's body, instead of perpendicular. A way to achieve better results would be to render the demon several times with the emissive map, blended on top the rendered scene, and each time inflating the model by slightly moving every vertex of it outward along the vertex' normal. Each of these inflated models is called a shell, and the technique is often used in games as a part of creating fur. However, it requires vertex shaders, and thus can't be implemented in OpenGL ES 1.1.

5.3 3D Kodo

For more detailed description of the shaders mentioned in this section, see chapter 6.3.

As mentioned earlier, we decided to develop a game prototype of the game Kodo with 3D graphics and shaders to test our engine. Porting an existing game was preferable as opposed to designing a completely new game, since all the game mechanics and rules were already defined, meaning we could focus on the visual part. Furthermore, the lack of realism in the game gave us more freedom to experiment with the graphics.

As soon as we had our first game scene, containing a game field and four modelled kodos similar to those of the original game, we started developing shaders. One of the first effects we attempted was refraction, to add a more liquid appearance to the kodos. We also added an animated normal mapping to the kodos, to make their surface appear less plain. Furthermore, we used animated normal maps and reflection to create water below the level for added ambience to the scene. Furthermore, we briefly experimented with a shader for animated fire, which in combination with animated attenuation and colour of lights provided a dynamic lighting to the scene. Lastly we added light sources to the kodos, to further increase the dynamic lighting of the scene. Figure 13 below shows the result.

![Figure 13: Early version of 3D Kodo, with refraction on the kodos, normal mapping and reflection on the water, and animated fire](image-url)
With our artistic skills lacking at best, neither the game field nor the kodos were very well designed. Luckily, we received help in this department from three graphical artists at Jadestone. They provided us with concept art, models of a game world and kodos, and animations for said kodos. We now had a clear design goal to strive for, instead of just aimlessly testing shaders with little regard to the end product.

With help from the graphical artists, we decided on a few choice effects that would actually be beneficial to the visual design of the game. We started by adding skinning to the kodos in order for them to be animated when walking, turning and eating. Then we added refraction to the crystals of the scene, but we wanted the edges of the crystals to be less transparent than the middle of the faces, as seen in the concept art (figure 14). This was easily accomplished by using the alpha channel of the texture to specify the transparency of the crystal, this value was then used in the refraction shader to mix between the refracted colour and the texture colour.

Furthermore, we added a glow effect, similar to the one described in chapter 5.2 to certain parts of the scene, namely the crystals and certain parts of the vegetation. The differences in this glow method is that no emissive map is used, instead each object has a special colour variable to define whether the object should glow, and if so what colour it should glow with when rendering to the glow texture. The glow colour is black for objects that shouldn't glow at all. As with the Demon, this glow texture is drawn blended on top of the scene, but unlike it, it is only drawn once, and not stretched but blurred by a fragment shader. This eliminates the problem we had with emission direction in the Demon demo, as well as eliminated the need for an emissive map. We tried applying glow to the kodos as well, but this caused them to look over-saturated.
In addition, we added fog calculations to the fragment shader of objects in the background to dim them out more the further away they are. Lastly, we added light sources to the kodos, crystals and the fruits hanging from trees to add to the ambiance of the night scene. The result can be seen in figure 15 below.

While we were satisfied with the results, we also realised that there wouldn't be much variety in the scene if the entire game would play out during night. We decided to add a sun rotating around the game world, which also gave us a reason to add shadows during the day. However, a number of things should change between night and day, such as ambient lighting colour, fog colour, glow and light sources. All these things need to adjust, preferably by fading seamlessly. Glow and light sources other than the sun and ambient light should disappear completely during the day, while ambient lighting colour and fog colour should simply be changed to different colours. In reality, dusk and dawn also look quite different from day and night, so we effectively had four different states to mix between.

Instead of adding a time variable to determine what time of day it is, we decided to simply use the angle of the sun light source. That way the sun could be animated directly while creating the environment, without having to manually synchronise the movement with the game time. This way it is also possible to change the speed of the sun by simply modifying the level, and not having to recompile the game. The four time states (day, dusk, night and dawn) were thus simply defined as sun angles. The final game prototype can be seen in figure 16.
The main problem we came across while developing shaders was that we had no simple way to change the shader variables at runtime, instead we had to restart the game every time we wanted to try new settings. An option would have been to use a shader development studio such as RenderMonkey or FX Composer, but our shader format was not compatible with those. An option would have been to modify the shaders to be compatible with one of the studios, but at the time we deemed it to be too much work to be worth it. In retrospect, however, this might have been the wisest thing to do, as fine-tuning the shader variables turned out to be more work than expected.

For some select shader variables that we realised we would change often we arranged that their values could be changed by using game keys. This was an acceptable solution, but it did not allow changing more than a few variables at a time, and when changing which variables to modify the game had to be recompiled.
6 Evaluation

The mobile platform has certain effects on game development, as the platform is different from game consoles by not originally being intended for games. Therefore, certain aspects need to be considered when designing and developing a mobile phone game, such as limitations inherent in the design of the hardware. The intention of this chapter is to discuss some of these aspects, and analyse their effect on games.

6.1 Why 3D and Shaders?

There are many reasons to make games in 3D instead of 2D. First of all, it allows for more kinds of games, for instance first-person shooters, which are difficult or impossible to develop in 2D. Furthermore, it adds flexibility to the game design, since it is easy to change things like camera view or zoom level in 3D, but with sprite-based 2D games it often involves remaking some or all of the sprite images. It may also save memory, a large 2D game could have an abundance of images for the sprites, different ones for the same object facing different ways, plus animations, compared to a 3D game which can make do with a few animated models that can often be used for additional purposes by simply changing the texture.

Similarly, there are also reasons to use shaders in a 3D game. Most obviously it is easier to produce advanced visual effects, and more effects can be used simultaneously. Shaders may also help save memory, since occasionally things that normally would be stored in memory instead can be computed in real-time. Furthermore, shaders add flexibility, since changes in shader code most often will not require a recompile of the game engine to take effect.

6.2 Mobile Phone Game Design Limitations

Aside from the obvious limitations in mobile phone hardware, such as CPU and GPU performance and memory size, some other limitations should also be considered when creating mobile games.

6.2.1 Multiplayer Limitations

In the past few years, multiplayer games for PC and game consoles have grown immensely. In some games, henceforth called arena games, only a few players play with or against each other, but there are also Massive Multiplayer Online games, MMOGs, where hundreds, or even thousands of players play simultaneously. There is also an alternative to remote multiplayer games; hot-seat games, where two or more players compete on the same system and using the same screen. All these types of multiplayer games could suit the mobile phone platform well. However, arena games require some sort of phone-to-phone connection, and MMOGs require an internet connection to connect to some central game server.

While there are a few phone-to-phone protocols available in modern phones, such as infrared (IR) and Bluetooth, these are very limited. IR requires a clear visual path between phones, and has a very short range. Bluetooth has a slightly longer range and does not require visible contact, but in order to connect two devices the users would have to establish the connection manually, since the technology does not allow automatic connections. A few modern phones has built-in Wireless Local Area Network
(WLAN) support, which could be used for the purpose of arena games, and there is a possibility that WLAN will become more common in future phones. With WLAN, arena games could be played both locally using ad-hoc connections (i.e. phone-to-phone), or over the internet in case there is a Wi-Fi hotspot (public accessible WLAN with internet connection) within range.

Most modern phones have the ability to connect to the internet through General Packet Radio Service, or GPRS for short. This would in theory allow for the creation of MMOGs, and there are some such games available [32]. However, latency is a big problem, which makes it better suited for turn- or tick-based than real-time MMOGs. Another downside is the fact that phone operators most often charge GPRS communication per byte, which means it is more expensive for users the more they play. With 3G on the rise these problems are however likely to diminish, which hopefully will result in an expansion of the mobile MMOG market in the future. As with arena games, WLAN support could also solve the above problems, but unlike arena games, MMOGs require internet connection, and thus a hotspot must be nearby.

Hot-seat games have frequented both the PC market, mostly prior to the internet becoming popular, and the console market, where they are still the most common multiplayer method. With the above limitations in consideration, mobile phones may currently seem best suited for hot-seat games, but there are problems with this method as well. With the limited size of mobile phones it might be difficult for multiple players to gather around the same device, and a very limited amount of keys disallow advanced control schemes. Turn-based games can utilise this method to a further degree, since players take turns at controlling the game and does not always have to, and in some cases must not, see the screen while another player controls the game. However, this also increases play time since only one player can control the game at a time, and the other players can do nothing but wait for their turn.

6.2.2 Input Limitations

Perhaps the most notable difference between PC/gaming consoles and mobile phones, input limitations are in many cases a considerable obstacle to overcome. Since phones usually only have a few number of buttons and no analogue input whatsoever, new approaches has to be taken unless the game is suited for such limited control scheme. If the game world is 3D, i.e. not simply a 2D game with 3D graphics, this problem is further emphasised due to the addition of another dimension of movement.

Kodo solves this problem, and allows for hot-seat multiplayer sessions, by allowing multiple players to play on the same phone, each using only one button to control their kodo. A small arrow revolves around each kodo, and players move their kodo forward by pressing their button when their arrow is pointing in the desired direction.

With touch screens becoming more common in hand-held devices, this provides game designers with further options for control schemes. The regular touch screens are most likely soon replaced with multi-touch screens which unlike regular touch screens can detect multiple simultaneous touch points. This even makes it possible to utilise them for multiplayer hot-seat games, using Kodo as an example the players could touch their kodo to make it move forward, instead of pressing a button.
6.2.3 Output Limitations

As with most other game systems, the most obvious output source of a mobile phone is the screen, which in most cases is small and has a low resolution. This means it is harder to fit an abundance of objects on the screen, and that objects probably has to be larger to be visible compared to for example a PC. Furthermore, due to the limited space, the GUI would have to be carefully designed to be informative and clearly visible but still small. This is of course important on other systems as well, but with the large screen of for example a PC, people tend to be more tolerant of cluttered or bloated GUIs.

While not really a limitation, sound output might prove to be a problem. Audio feedback is often important in games, and unless the player uses a headset she will most likely turn the sound volume down, if not completely off, when playing a game in public. However, to compensate for this it is possible to use the vibration function built into most phones as haptic feedback to complement the audio.

6.3 Shader Techniques

Since game developers started utilising shaders they have become an important part of any modern game, and as a consequence many different shaders have been developed. Some to add new visual effects, others to produce common effects more efficiently. The two major concerns when developing shaders for mobile phones are the limitations in computational power and screen resolution/size. The latter is important since effects that would make a big difference on a large, high-resolution PC monitor might be too small to be noticeable on a phone screen. In order to determine the usefulness of different shaders with regards to the limitations of hardware in mobile phones, we tried to evaluate some keeping this in mind. Some of these shaders are listed as sample code in Appendix A.

Texture Coordinate Modification – Modification of texture coordinates at runtime. This can be used for different purposes, such as animating fire, lava or water. The texture coordinates of a fragment are simply modified by some value, which could for example be randomised or based on time. As we will see, quite a few of the shaders in this section use some variety of this method.

Refraction – Simulation of light bending while passing through translucent or transparent objects, such as water or glass [14], as seen in figure 17. The whole scene, except for the refracting objects, is rendered to a texture. For each fragment of the object an offset is calculated with a refract function (for example the GLSL built-in function \texttt{refract()}) with regards to the normal of the fragment as well as a refractive index, which is a constant of the medium the light passes through. The texture coordinates of the fragment are modified with this offset and used to read from the generated texture. Rendering just a part of the scene around the refracting object, as well as clipping objects closer to the camera than the refracting object could produce better results. In addition, if refracting objects are to be visible through other refracting objects, a separate texture needs to be rendered for each refracting object. In some cases refraction will add extensive visual appeal to the scene, but if the refracting object is small compared to screen size, simply drawing the object transparent (blended) might be adequate. The fact that refraction adds at least one rendering pass should also be noted, which in combination with other effects might prove too costly for mobile hardware.
Chapter 6 - Evaluation

Reflection – Basically the same procedure as refraction, with the exception of the use of another function to calculate the offset (i.e. the GLSL function reflect()), and that the texture should be rendered from the point of view of the reflecting object, aligned along its surface.

Normal mapping – Used to add complexity to the surface of an object, which can be seen in figure 18 below. The normal of each of the object's fragments is modified by a value defined in a normal map [14]. The normal map is usually produced by modelling the object with great detail, and saving the normals of this model as an image. The object is then simplified to reduce the number of polygons, but when rendered with the normal mapping seems highly detailed. Normal mapping can also be used to create an illusion of waves across water, where a normal map is animated to simulate the waves moving forward. This way the water object can be reduced to a single plane, instead of a complex model. Since the normal map is pre-produced and not created at runtime, this is a computationally relatively cheap method, the exception is if used in conjunction with skinning, as some extra calculations for tangents and bi-normals are required. However, if the objects are small compared to the screen the effect may not be noticeable.

Figure 17: Refraction shader in 3D Kodo

Figure 18: A model of a head with 371 triangles, comparison: with per-triangle normals, with interpolated per-vertex normals and with normal mapping [0]
Chapter 6 - Evaluation

**Vertex Texture Displacement** – While normal mapping can be used to make a simple object appear to have bumps, sometimes it is desirable to actually make the object itself bumpy. This comes in handy to create more realistic water than a normal map alone could produce, since normal mapped water will still look flat if viewed closely or from the side. Vertex texture displacement modifies vertex positions according to a texture [33]. The texture can be moved across the surface to simulate waves moving forward. In reality, using only one texture for this would not look very realistic, instead multiple textures, moving independently, should be combined for added variety in the displacements. This technique requires that the graphics hardware allow vertex programs to read from textures, and while it is probable future phone graphics hardware such as the SGX will do this, it is not yet known for sure.

**Fog effect** – Fog is used to fade objects over distance. This is useful to lessen the annoyance of objects suddenly appearing/disappearing when passing the far clipping plane, or simply if a distance haze effect is desired. Prior to OpenGL ES 2.0 fog could be created with the fixed functionality of OpenGL. The basic idea behind fog is to blend an object's colour with a fog colour depending on the object's distance from the camera. Distance calculations can be performed either per vertex or per fragment, per vertex is often good enough unless polygons span a long distance. Instead of actually calculating the distance to a fragment, it is possible to use its Z-value to save some calculations. This does however cause the fog to start at a plane parallel to the camera, instead of the actual distance from it.

**Cel-shading** – Sometimes called toon shading, this shading technique is intended to make objects look hand-drawn, like a cartoon [34]. First of all, the object's outlines are emphasised by inverting backface culling and drawing the object in wireframe mode multiple times slightly transformed, to produce thick black lines. The object is then drawn normally, textured, and since the back faces are deeper in the scene the only black lines that will be visible are the ones near contours. Lastly, the lighting is calculated quantized, meaning light transitions between light and dark are not smooth, but in a few distinctive steps. See figure 19 to see the results of the different steps. Although cel-shading makes the graphics look simple, the process is rather complex, and objects have to be drawn multiple times to get the outline effect.

![Figure 19: Cel-shading steps: black wireframe with inverted face orientation, basic texturing and quantizing lighting [34]](image)

**Ambient Occlusion** – For a scene to be truly realistic the light has to have a lower effect on parts of the scene that is surrounded by much of geometry. This is called ambient occlusion and is a global illumination effect, since objects are affected by other nearby objects. It is often used in static scenes since it can be pre-computed, but this requires memory to be stored and does not take dynamic objects into consideration. In 2007, Martin Mittring at Crytek presented a new method which he calls Screen-Space Ambient Occlusion (SSAO) [35], an approximation of ambient occlusion that can be performed in real-time. The idea is quite simple; take the z-value of a random point
inside a sphere originating at the fragment. Compare this value with the value stored in the depth buffer for that point, if the difference between them is relatively small and the value in the buffer is lower, the ambient occlusion of the fragment is slightly increased, making it darker. Repeat this procedure with a few random sample points. This method is especially preferable if the depth buffer is already used for other effects and thus already rendered to a texture.

**Shadows** – A scene without shadows often look plain and simple, especially outdoor scenes with sunlight. One way to create shadows is to use a shadow map, which is basically a depth map rendered from the shadow casting light's point of view. If a shadow map is rendered it is easy to compare a fragment's distance from the light with the value stored in the shadow map to determine whether or not the fragment should be lit by this light. However, each shadow casting light would need its own shadow map, and it might not be worth having more than one such light. Furthermore, the shadows created with this method often has rough and aliased edges. Multisampling is one way to remedy this, by combining samples not only from the fragment's position in the shadow map but also from nearby positions into an average the shadow's edges will be slightly softened. Shadows can greatly increase the realism of certain scenes, and may very well be worth the extra shadow texture needed.
7 Conclusions

In order to reach any conclusions, let's first return to the question asked in the introduction of the thesis:

*How can three-dimensional graphics and shaders be successfully used in mobile phone games?*

3D graphics, especially in combination with shaders, can definitely add visual appeal to a game. In theory, games for mobile phones with OpenGL ES 2.0-capable graphics cards can match those of a modern PC in terms of visual effects, but in reality the phone hardware, both CPU, GPU and memory available will probably limit the amount and complexity of effects. Still, 3D graphics may very well be successfully used even for games with 2D mechanics (such as Kodo), since it offers more flexibility than 2D graphics. Shaders further this flexibility by adding the possibility to create effects largely independent of the graphics engine, which means graphics engines can be more generally used since changing effects between games does not necessarily require a change in the graphics engine.

Furthermore, I asked the following question:

*What limitations does the mobile phone platform enforce on game design?*

There are several limitations to the mobile phone platform, such as multiplayer, output and, most importantly, input. The limitations in both ad-hoc and internet communication limits multiplayer to hot-seat games (although this might change with 3G replacing GSM). The limitations of output, primarily screen size and resolution, limits visual appeal since objects and effects might be too small to be noticeable. The lack of analogue input and the number of buttons available limits the complexity of control schemes, especially in hot-seat multiplayer games.

As for a more tangible conclusion, the new graphics cards may very well cause an increase in popularity of 3D games, provided enough mobile phones employ the cards. From a development standpoint, care has to be taken when designing mobile phone games with regards to the limitations of the platform, especially if compared to a PC or game console.
# Bibliography


37
Bibliography


Bibliography


Appendix A: Shader Sample Code

This appendix is intended to supply sample code for some of the shaders described in chapter 6.3. The shaders are coded in GLSL ES.

Refraction vertex shader program

```glsl
attribute vec3 myVertex;
attribute vec3 myNormal;
attribute vec2 myUVMain;
uniform mat4 myMVPMatrix;
uniform mat3 myNormalMatrix;
uniform mat4 myModelViewMatrix;
uniform mat4 myModelMatrix;
uniform mat4 screenTexMatrix;

varying vec3 eyeNormal;
varying vec4 texCoordinates;
varying vec3 eyePos;

void main(void)
{
    // Calculate position and texture coordinates
    gl_Position = myMVPMatrix * vec4(myVertex, 1.0);
    texCoordinates.xy = myUVMain.xy;
    vec4 screenTexCoord = myModelMatrix * vec4(myVertex, 1.0);
    screenTexCoord = screenTexMatrix * screenTexCoord;
    texCoordinates.zw = ((screenTexCoord.xy/screenTexCoord.w)+1.0)*0.5;

    // Calculate normals in eye space
    eyeNormal = normalize(myNormalMatrix * myNormal);
    eyePos = (myModelViewMatrix * vec4(myVertex,1.0)).xyz;
}
```
Appendix A: Shader Sample Code

Refraction fragment shader program

```cpp
uniform sampler2D screenTexture;
uniform sampler2D materialTex;
uniform vec3 matAmb;
uniform vec3 matSpec;
uniform float matShininess;
uniform float refractionMagnification;
uniform float refractionIndex;
uniform vec3 myLightPoss[3];
uniform vec3 myLightColors[3];
uniform float myLightAttens[3];

varying vec3 eyeNormal;
varying vec4 texCoordinates;
varying vec3 eyePos;

vec3 myRefract( vec3 I, vec3 N, float eta) {
  float dotNI = dot(N, I);
  float k = 1.0 - eta * eta * (1.0 - dotNI * dotNI);
  return eta * I - (eta * dotNI + sqrt(k)) * N;
}

void main (void) {
  // Calculate refraction offset
  vec3 eyeVec = normalize(-eyePos);
  float eyeDotNorm = dot(eyeVec, eyeNormal);
  vec2 offset = refractionMagnification *
    -myRefract(eyeVec, eyeNormal, refractionIndex).xy;

  // Calculate lighting
  vec3 diff = matAmb;
  vec3 spec = vec3(0.0);
  for(int i=0; i<3; ++i) {
    vec3 toLight = myLightPoss[i] - eyePos;
    vec3 halfVec = normalize(normalize(toLight) +
      normalize(-eyePos));
    float tmpAtten = myLightAttens[i]/dot(toLight, toLight);
    diff += max(dot(eyeNormal, normalize(toLight)), 0.0) *
      myLightColors[i]*min(tmpAtten,1.0)*2.0;
    spec += myLightColors[i]*(pow(max(dot(eyeNormal, halfVec),
      0.0), matShininess)*tmpAtten);
  }
  diff = min(diff, 2.0);
  spec = min(spec * matSpec, 1.0);

  // Read screen texture to get refract colour
  vec3 refractColour = vec3(texture2D(screenTexture,
    (texCoordinates.zw+offset)));

  // Read material texture and blend with refract colour
  vec4 texColour = texture2D(materialTex, texCoordinates.xy);
  vec3 colour = mix(refractColour, texColour, blendVal);
  gl_FragColor = (vec4(colour, 1.0) + vec4(spec, 0.0));
```

Fog and shadow vertex shader program

attribute vec4 myVertex;
attribute vec3 myNormal;
attribute vec2 myUVMain;

uniform mat4 myMVPMatrix;
uniform mat3 myWorldViewIT;
uniform mat4 myModelView;
uniform mat4 shadowTexMatrix;

varying vec3 eyeNormal;
varying vec3 eyePos;
varying vec4 lightTextureCoord;
varying vec2 texCoordinateMain;

void main(void)
{
    gl_Position = myMVPMatrix * myVertex;
    eyePos = (myModelView * myVertex).xyz;
    lightTextureCoord = (shadowTexMatrix * myVertex);
    eyeNormal = normalize(myWorldViewIT * myNormal);
    texCoordinateMain = myUVMain.st;
}
Appendix A: Shader Sample Code

Fog and shadow fragment shader program

```GLSL
uniform sampler2D materialTexture;
uniform sampler2D shadowTexture;
uniform vec3 matAmb;
uniform vec3 matSpec;
uniform float matShininess;
uniform vec3 fogColour;
uniform vec3 colorMask;
uniform float fogDepthVal;
uniform vec3 myLightPoss[5];
uniform vec3 myLightColors[5];
uniform float myLightAttens[5];
uniform int shadowIndex;

varying vec3 eyeNormal;
varying vec3 eyePos;
varying vec2 texCoordinateMain;
varying vec4 lightTextureCoord;

float readShadow(vec2 tc, float z) {
    vec3 texShadow = texture2D(shadowTexture, tc).rgb;
    // Read shadow depth from a RGB texture and modify it to get a float
    float shadowZ = ((texShadow.r*256.0+texShadow.g)*256.0 + texShadow.b)/65536.0;
    float ret = 0.0;
    if(shadowZ < z) {
        ret = 1.0;
    }
    return ret;
}

void main (void) {
    vec3 DiffuseColor = matAmb*2.0;
    vec3 SpecularIntensity = vec3(0.0);
    float inShadow = 0.0;
    vec3 shadowTexCoord = ((lightTextureCoord.xyz / (lightTextureCoord.w)) * 0.5 + 0.5);
    // Collect multiple samples to soften shadow edges
    const float tDelta = 0.001;
inShadow += readShadow(shadowTexCoord.xy, shadowTexCoord.z)*2.0;
inShadow += readShadow(shadowTexCoord.xy+vec2(tDelta,tDelta), shadowTexCoord.z);
inShadow += readShadow(shadowTexCoord.xy+vec2(-tDelta,tDelta), shadowTexCoord.z);
inShadow += readShadow(shadowTexCoord.xy+vec2(-tDelta,-tDelta), shadowTexCoord.z);
inShadow += readShadow(shadowTexCoord.xy+vec2(-tDelta,-tDelta), shadowTexCoord.z);
inShadow /= 6.0;

    for(int i=0; i<5; ++i) {
        vec3 toLight = myLightPoss[i] - eyePos;
        vec3 halfVec = normalize(toLight - normalize(eyePos));
        float tmpAtten = myLightAttens[i]/dot(toLight, toLight);
        float shade = 1.0;
        float lightValue = dot(eyeNormal, normalize(toLight));
        if(lightValue >= 0.0) {
            // We still use 40% of the shadow casting light, to
            // approximate light “bleeding” from nearby surfaces
            if(i == shadowIndex)
                shade = 1.0 - inShadow * 0.6;
            DiffuseColor += max(lightValue, 0.0) * myLightColors[i]*tmpAtten * shade * 2.0;
            SpecularIntensity += myLightColors[i] * pow(max(dot(eyeNormal, halfVec), 0.0), matShininess) * tmpAtten * shade;
        }
    }
    gl_FragColor = vec4(DiffuseColor + SpecularIntensity, 1.0);
}
```

Appendix A: Shader Sample Code

```cpp
DiffuseColor = min(DiffuseColor, 2.0);
SpecularIntensity = min(SpecularIntensity, 1.0);

gl_FragColor = (texture2D(materialTexture, texCoordinateMain) * 
    vec4(DiffuseColor, 1.0) + vec4(vec3(matSpec*SpecularIntensity), 
    0.0)) * vec4(colorMask, 1.0);

// Fog calculations
vec3 fogPos = eyePos * fogDepthVal;
float dist = length(fogPos);
float fogVal = min(dist, 1.0);
gl_FragColor = mix(gl_FragColor, vec4(fogColor, 1.0), fogVal);
```