3D Game Engine Design for Mobile Phones with OpenGL ES 2.0

M I K A E L   G U S T A V S S O N

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2008

3D Game Engine Design for Mobile Phones with OpenGL ES 2.0

M I K A E L   G U S T A V S S O N

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 2008:024
ISRN-KTH/CSC/E--08/024--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se

3D Game Engine Design for Mobile Phones with OpenGL ES 2.0

Abstract

This master's project investigated the capabilities of mobile phones to support 3D graphics for games and how to develop for these devices using the OpenGL ES graphics library. A simple 3D game engine was developed that runs on a PC using a OpenGL ES 2.0 emulator library. Additionally, a game prototype was developed using this engine. The report investigates the differences between PC and mobile games, and how the mobile platform affects the design of a 3D game engine. Furthermore, the differences between OpenGL ES 1.1 and 2.0 are described, covering the implications of developing game graphics with shader programs. In conclusion, mobile phones supporting OpenGL ES 2.0 will be available in 2008 and they will probably support 3D graphics approaching the quality of recent PC games. Developing games for these devices would be very similar to developing PC games. The largest differences relating to graphics are the screen size and memory constraints.

Utformning av 3D-spelmotorer för mobiltelefoner med OpenGL ES 2.0

Sammanfattning

Det här examensarbetet hade som mål att utreda möjligheterna för mobiltelefoner att stödja 3D-grafik för spel och hur utveckling för denna plattform kan ske med hjälp av grafik-biblioteket OpenGL ES. En enkel 3D-spelmotor utvecklades genom att använda ett emulatorbibliotek för OpenGL ES 2.0 på PC. Med hjälp av denna motor utvecklades en spelprototyp. Denna rapport undersöker skillnaderna mellan mobil- och PC-spel, samt hur den mobila plattformen påverkar utformningen av en 3D-spelmotor. Dessutom så beskrivs skillnaderna mellan OpenGL ES 1.1 och 2.0, och hur utvecklandet av spelgrafik påverkas av shader-program. Slutsatsen är att mobiltelefoner som stöder OpenGL ES 2.0 kommer att finnas tillgängliga under 2008 och att de troligtvis kommer att stödja 3D-grafik vars kvalitet närmar sig moderna PC-spel. Att utveckla spel för sådan hårdvara kommer i stora drag vara likvärdigt med att utveckla PC-spel. De största skillnaderna ur ett grafikperspektiv är skärmstorlek och minnesbegränsningar.

Table of Contents

1 Introduction 1
1.1 Problem Statement 1
1.2 Delimitations 1
1.3 Thesis Outline 2

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 18
4.1 PC Hardware 18
4.2 Mobile Hardware 18

5 Approach 21
5.1 3D Game Engine 21
5.1.1 The Effect Files 21
5.1.2 The Model Files 23
5.1.3 Scene Representation 24
5.1.4 Scene Management 25
5.1.5 Animations 25
5.1.6 Built-in and Custom Uniforms 26
5.1.7 Scene Rendering 28
5.2 The Demon Demo 29
5.2.1 OpenGL ES 1.1 Adaptation 30
5.2.2 Symbian OS Adaptation 31
5.3 3D Kodo Game Prototype 31

6 Evaluation 34

6.1 3D Game Engine....34

6.1.1 Resource Handling....34

6.1.2 Data Files....35

6.1.3 Scene Management....36

6.1.4 Rendering....37

6.2 Demon Demo....37

6.3 3D Kodo Game Prototype....38

7 Conclusions and Further Work 40

1 Introduction

Mobile phones constitute an interesting hardware platform for game developers since so many people always carry a phone around with them. However, mobile phones are generally not specifically designed to support gaming, which poses problems for game developers. One of these problems has been that mobile phones traditionally have provided comparatively simple graphics. This thesis aims to evaluate the graphics capabilities of current and upcoming mobile phones, specifically focused on 3D graphics using the OpenGL ES graphics programming interface. OpenGL ES is an adaptation of the OpenGL industry standard aimed at embedded systems, and is available in two main versions: 1.1 and 2.0. This thesis is mainly focused on the later version which supports more advanced graphical effects through the use of shader programs.

The goals for the project were to examine how to develop mobile games with OpenGL ES 2.0 and how three-dimensional graphics and shader effects can successfully be used in mobile games. This was accomplished by developing a 3D game engine on top of a OpenGL ES 2.0 emulator library currently available and using this engine to create a game prototype. The differences between PCs and mobile phones, between 2D and 3D games, and between 3D graphics with and without shaders were evaluated.

1.1 Problem Statement

My main research question for the thesis was:

What are the specific technical considerations relating to graphics that apply when developing 3D games for mobile phones?

This thesis is specifically focused on graphics, and does not cover other areas such as game design, sound or input. Furthermore, since most games are built on top of a game engine and game engines handle the technical details of the platform, this thesis puts much focus on the design of game engines for mobile phones.

1.2 Delimitations

Since the project was a collaboration between two students, Erik Olsson and Mikael Gustavsson, the background information in chapters 2 – 4 are common to both reports. The remainder of the theses are individual. We made the following delimitations regarding the individual parts of the theses:

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

Erik will focus 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.

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 engine and game prototype, as well as an evaluation of the results. The chapters of this part is as follows:

Approach
Implementation details, focusing on the 3D graphics engine.

Evaluation
An evaluation of our results as well as some more general discussion about 3D graphics and games for mobile phones and how the limitations of the hardware affect the design of game engines for mobile platforms.

Conclusions and Further Work

Conclusions of our work and suggestions for further study.

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 below shows an example model.

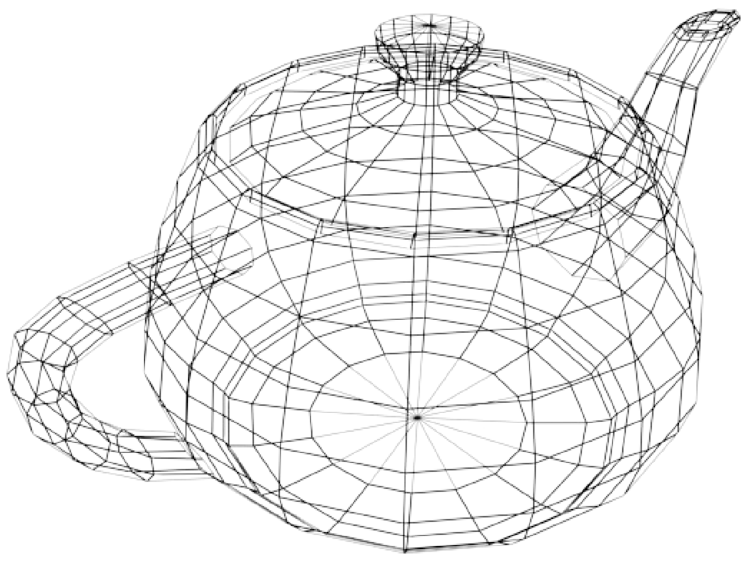

Figure 1: A model of the famous Utah Teapot, shown here in wireframe mode

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.

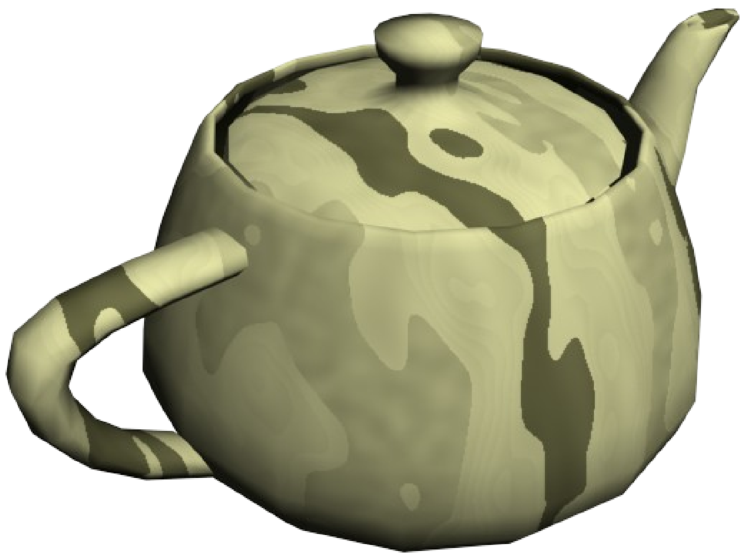

Figure 2: A textured Utah Teapot

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.

- *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*.

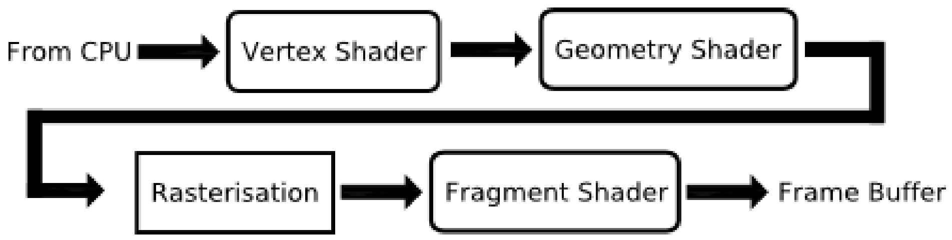

Figure 3: A shader pipeline

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 ouput 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.

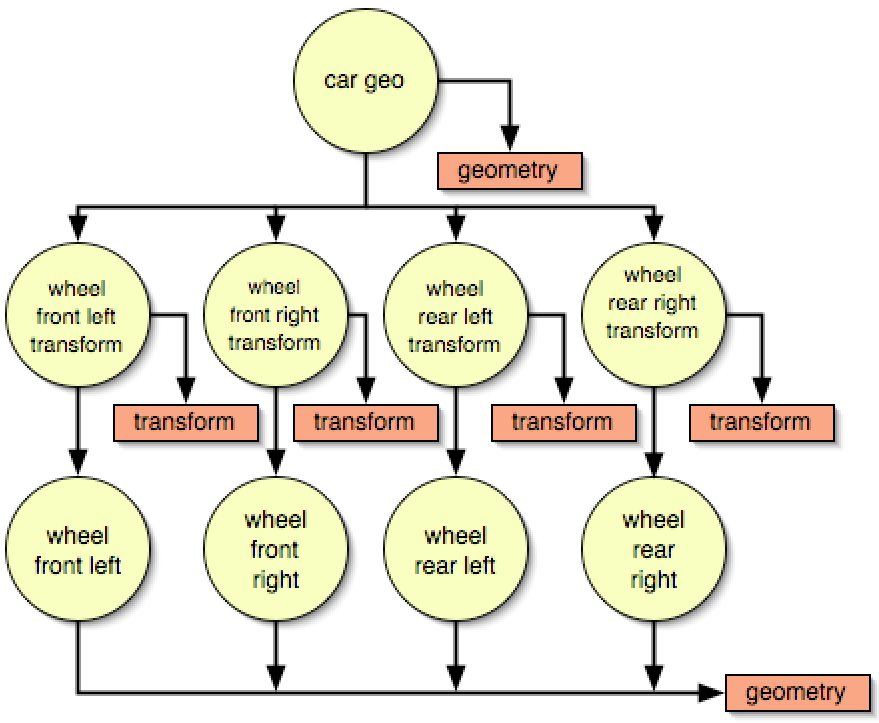

Figure 4: A simple scenegraph of a car [2]

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 transform 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.

$$L_o(x,\vec{\omega})=L_e(x,\vec{\omega})+\int_{\Omega} f_r(x,\vec{\omega}',\vec{\omega})\,L_i(x,\vec{\omega}')(\vec{\omega}'\cdot\vec{n})\,d\vec{\omega}'$$

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.

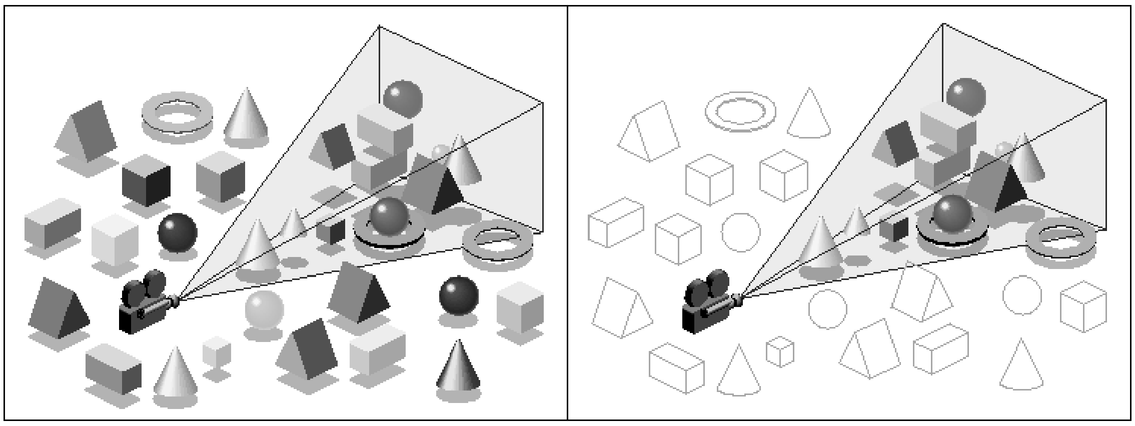

Figure 5: No culling (left) and view frustum culling (right) [6]

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.

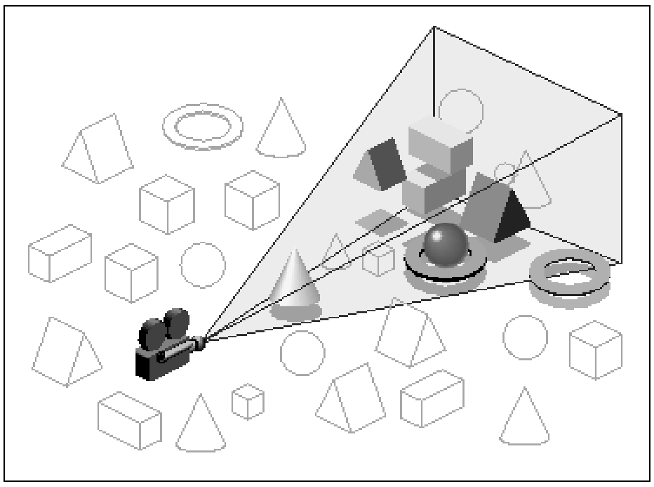

Figure 6: View frustum culling and occlusion culling combined [6]

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.

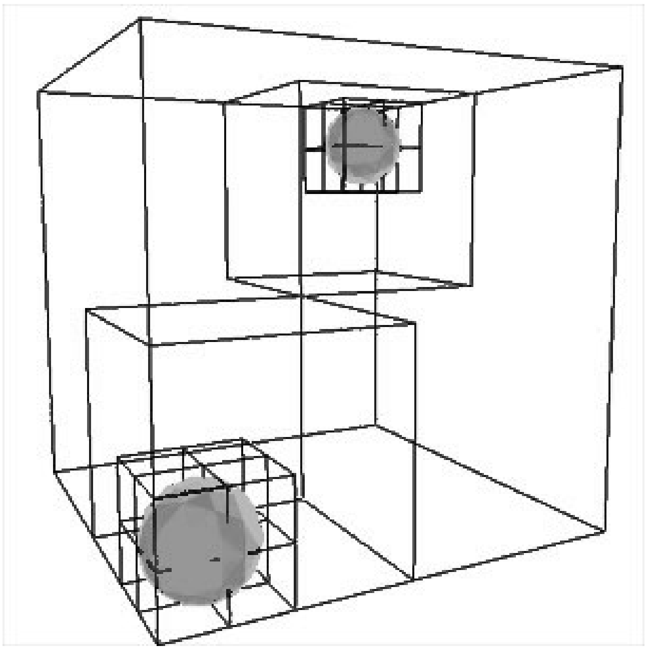

Figure 7: Octree spatial acceleration structure constructed around two spheres [11]

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. S*patial 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*, *octrees* (see figure 7 above), *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.

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.

```
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 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].

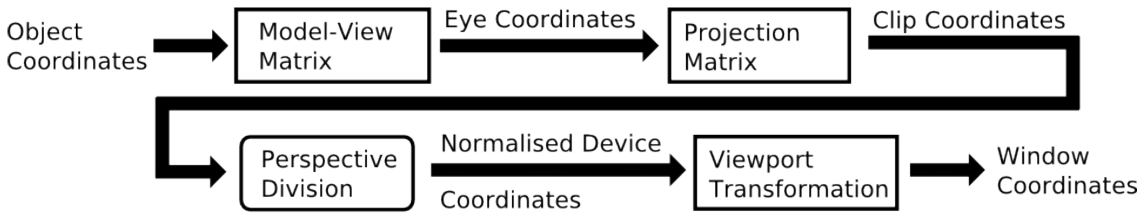

Figure 8: Vertex transformation sequence in OpenGL

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.

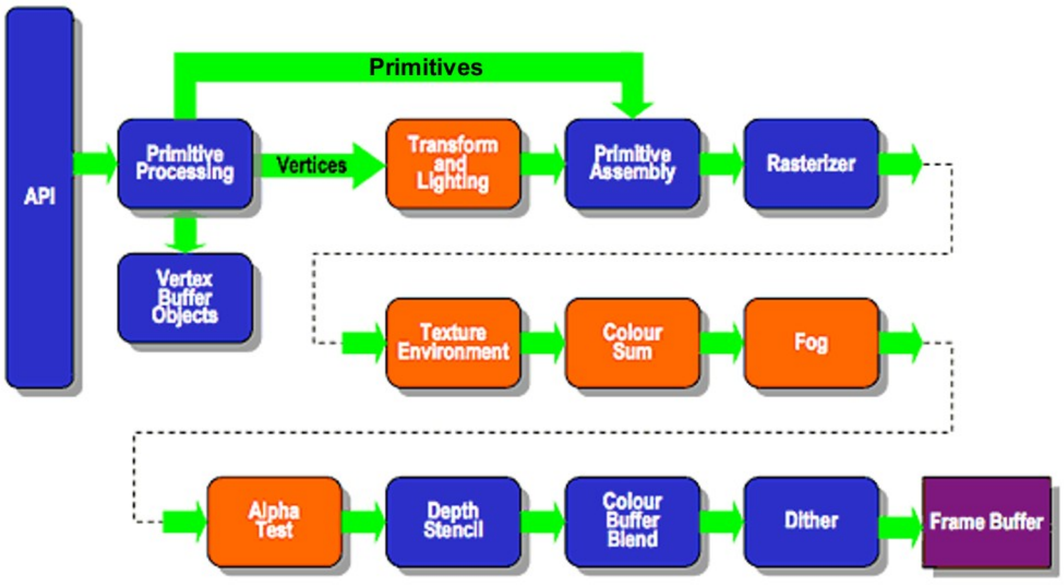

Figure 9: The fixed function pipeline of OpenGL 1.5 and OpenGL ES 1.1 [13]

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.

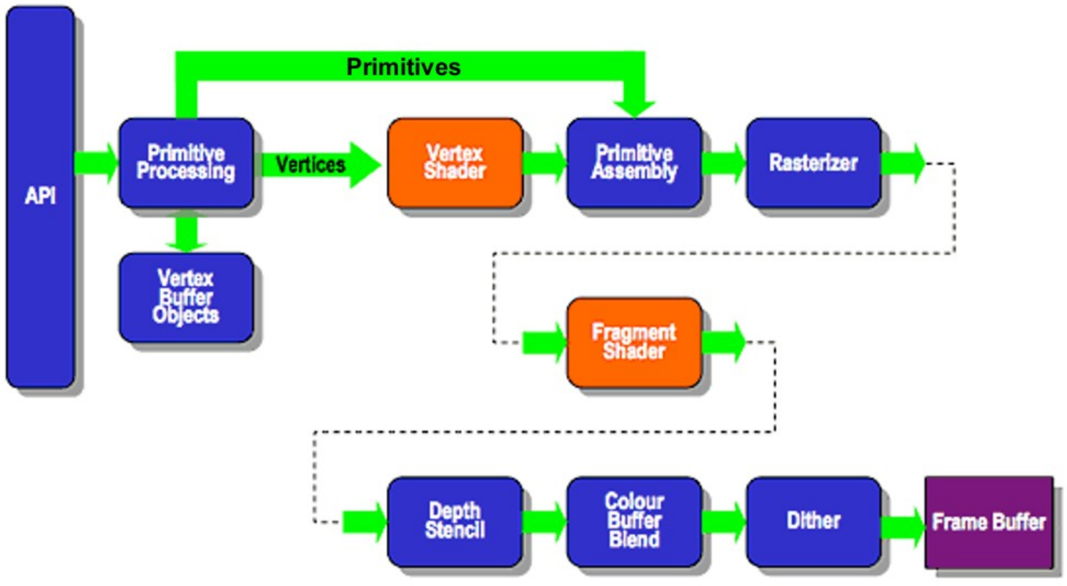

Figure 10: The programmable pipeline of OpenGL ES 2.0 [13]

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*).

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 were introduced in Hollywood films. In the 00's PC 3D hardware continues to improve and hand-held devices with hardware accelerated graphics emerge [17].

4.1 PC Hardware

Hardware accelerated 3D graphics on the PC became popular with the introduction of the 3dfx Voodoo 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 [18].

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 $480*800*60*2=46\text{M}$ 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 an 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

We 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 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. Since there were no OpenGL ES 2.0 3D game engines available to us, we needed 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 and large to be easily adapted to a hand-held device. However, we also realised 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. Also, the PowerVR OpenGL ES 2.0 SDK was of great help to us.

5.1 3D Game Engine

Loading and handling resources is a complex part of a 3D engine. A number of decisions has to be made about which file formats to support, how the files should be created and manipulated, the organisation of data in these files and the organisation, use and manipulation of this data when it has been loaded into the application memory. The PowerVR SDK contains a number of file format definitions and tools which work with these formats. We decided to make use of the PowerVR file formats since they are simple and designed to support OpenGL ES and mobile hardware. This worked well, but caused some problems since some of these file formats aren't very general and we couldn't easily extend them.

The PowerVR file formats that we use are texture files (.pvr), effect files (.pfx) and model files (.pod). The texture files are created from standard image files with a PowerVR application which we found very useful. PowerVR effect files are similar to the effect files used in NVIDIA's FX Composer and AMD's RenderMonkey, but much simpler. An application called PowerVR Shaman can be used to view and modify pfx-files. However, we decided not to use this application since it wasn't very versatile, we instead created and edited pfx-files with a text editor. The model files (.pod) are exported from the modelling program 3ds max with a custom exporter plug-in. The pod-files contain scenegraphs, models, simple materials and animations. The code to load pvr-files, pfx-files and pod-files was incorporated into our application and provided the basis for the rest of the engine.

5.1.1 The Effect Files

The pfx effect files represent material effects. Each file contains a number of texture specifications, vertex shaders, texture shaders and effect definitions. The effect definition specifies which shaders the effect uses, which uniforms and attributes it uses and also maps actual textures to any texture uniforms. Other uniforms and attributes are mapped to specific names defined by the application. These names correspond to different state that the application holds, such as transformation matrices, lighting settings or material settings. A simple example can be seen in code listing 2.

```
[TEXTURES]
      FILE wall   wall.pvr    LINEAR-LINEAR-LINEAR
[/TEXTURES]

[VERTEXSHADER]
      NAME        SimpleVertShader
      [GLSL_CODE]
      attribute   vec4  myVertex;
      attribute   vec2  myUVMain;
      uniform     mat4  myMVPMatrix;
      varying     vec2  texCoordinateMain;

      void main(void)
      {
            texCoordinateMain = myUVMain;
            gl_Position = myMVPMatrix * myVertex;
      }
      [/GLSL_CODE]
[/VERTEXSHADER]

[FRAGMENTSHADER]
      NAME        SimpleFragmentShader
      [GLSL_CODE]
      uniform sampler2D tex;
      varying vec2      texCoordinateMain;

      void main (void)
      {
            gl_FragColor = texture2D(tex, texCoordinateMain);
      }
      [/GLSL_CODE]
[/FRAGMENTSHADER]

[EFFECT]
      NAME        simpleTex
      UNIFORM     myMVPMatrix MODELVIEWPROJECTIONMATRIX
      UNIFORM     tex         TEXTURE0
      ATTRIBUTE   myVertex    VERTEX
      ATTRIBUTE   myUVMain    TEXCOORD0
      VERTEXSHADER      SimpleVertShader
      FRAGMENTSHADER    SimpleFragmentShader
      TEXTURE 0 wall
[/EFFECT]
```

Code listing 2: An example pfx file

These effect files proved to be powerful enough for our purposes except for the handling of textures. As can be seen in the example in code listing 2, texture files are specified directly in the effect file and bound in the effect definitions. This is very limiting since it is often desirable to make general effects which can be used for many different objects and materials. For example, the effect in listing 2 would be a general shader which could draw any unlit and textured geometry if it wasn't bound to be used with the *wall.pvr* texture. To solve this problem, we introduced a number of image file names with special meanings: *materialTexture* (diffuse map of the current material)*, normalTexture* (normal map of the current material)*, screenTexture* (the current screen contents rendered to a texture)*, objScreenTexture* (like screenTexture but centred around the current object) and *shadowMapTexture* (a depth map to be used in shadow mapping). This allows an effect to request textures specific to the current material or object, as well as dynamically generated textures not loaded from files. The primary reason for choosing this solution was that it did not require any changes to the file format or the pfx parsing and loading code.

Other possible extensions to the effect files would be to support default, minimum and maximum values for uniforms.

5.1.2 The Model Files

The pod model file definition, the 3ds max exporter and the pod loading code represent a significant amount of work and is the primary reason that we chose to use the PowerVR SDK code. Implementing this functionality by ourselves would have taken too much time and choosing some other model format was considered problematic since loading code might be hard to find or not suited to be used on mobile hardware due to portability problems or size and processing requirements. A pod file contains a scenegraph consisting of transform nodes arranged in a hierarchy. These nodes can be of any of these types: mesh nodes, light nodes, camera nodes or simple transform nodes.

The mesh nodes reference materials and mesh data. Materials support one diffuse texture; ambient, diffuse and specular colour; opacity and shininess values as well as a reference to an effect from an effect file. Mesh data is represented in one of these formats: triangles, indexed triangles, triangle strips or indexed triangle strips. Supported vertex attributes are positions, normals, tangents, binormals, multiple number of texture coordinate sets, colours, bone indices and bone weights. These attributes can be placed in separate arrays or in one interleaved array. Attributes can be stored as different data types such as one- two- or four-byte integers or two-byte floating point values to save memory. Light nodes are either point lights or directional lights, they have a colour and a target node (for directing the light). Camera nodes have a field-of-view value, far and near view frustum plane depth values as well as a target node. Simple transform nodes are used as target nodes or bones. All nodes can have animation data for position, rotation and scale.

Almost all of this data can be exported from 3ds max. The material effect file and effect name can not be exported since there is no “effect file” and “effect name” settings in the material options in 3ds max. There is probably a way to make a custom material plug-in to 3ds max but we settled for a simpler solution: materials are named in a special way to specify the effect for the material. For example, naming a material “simple_simpleTex_sky” in 3ds max, creates a material named “sky” in the engine that uses the effect “simpleTex” defined in the file “simple.pfx”.

Several other limitations with the model files were encountered, for example, there is no support for spotlights and the corresponding cut-off angle. We didn't solve this problem since we never needed to use spotlights. Furthermore, there is no support for uniform values in the materials (except for the pre-defined material values). This was handled by specifying uniforms in text files, which will be explained in 5.1.6. These uniform files are also used to specify light attenuation factors which are not supported by the pod files. Also, there is no support for several textures per material like normal maps or multiple diffuse maps. We handled this for normal maps with the special texture file names for effect files described in the previous section. If an effect uses the textures *materialTexture* and *normalMapTexture,* all materials that use this effect will automatically use a normal map that is named like the material diffuse texture. For example, if a material uses “rock.pvr” as a diffuse map and uses such an effect, the file “rock_normal.pvr” will automatically be used as the normal map for that material.

5.1.3 Scene Representation

The code that handles loading of textures, effects and models was provided by the PowerVR SDK. We started by combining these features so that after loading a model, all effects that the model uses are loaded, after which all textures that the materials or the effects use are loaded. All the data is simply stored by the loading code in a couple of arrays, one for each type of data: nodes, cameras, lights, meshes, materials and textures. This means that the scenegraph is not stored as an explicit tree, the tree structure is instead implicitly defined by storing parent node indices in the node data structures. This is the simplest solution possible, very memory efficient and accessing or modifying objects is simple and fast. However, adding and deleting objects becomes extremely costly. For example, when removing or adding a node, the node array will have to be rebuilt. This might cause node indices to be remapped, and to handle this, all node indices in the program will have to be checked and possibly modified. Since games are by nature quite dynamic (several objects might be added or deleted every frame) this is not an ideal scene representation. However, we decided to keep it because the SDK code relied on this and we wanted to use as much of it as possible. Also, if objects are not added or deleted during the actual gameplay, this representation is highly optimised. For example, since all nodes are stored in a single array, all this data is stored in one block of memory and this works very well with the processor's cache memory. In a more dynamic linked structure, each node normally has its own memory block and this might result in bad performance while traversing the scenegraph.

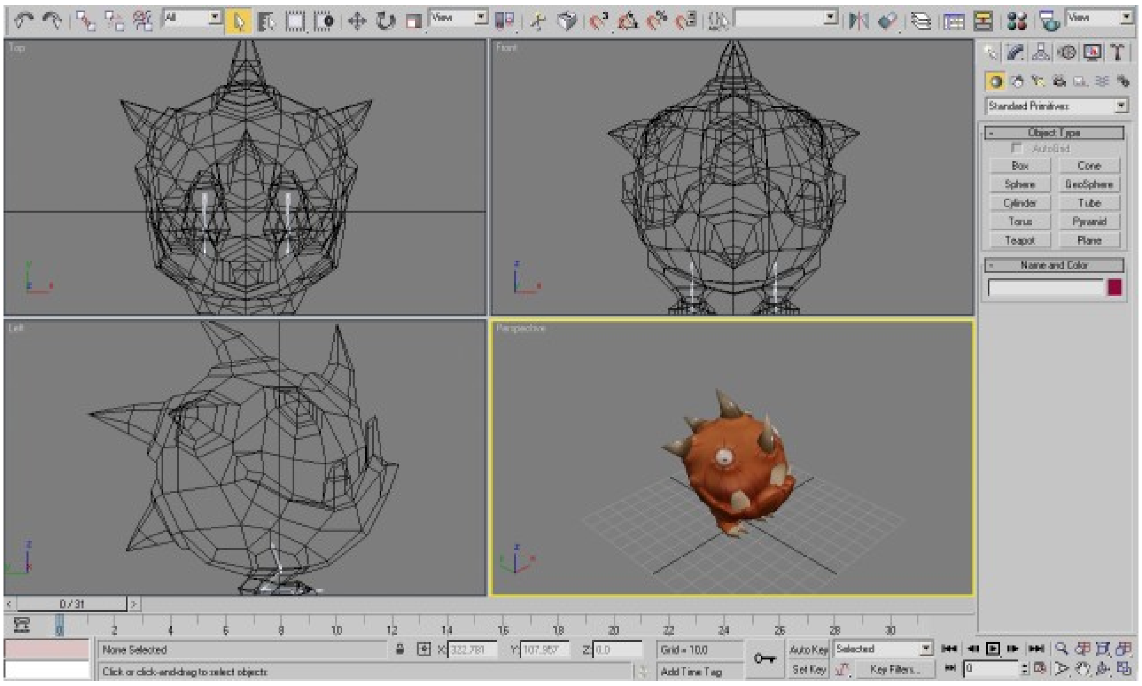

Figure 11: Kodo character in 3ds max, the mesh, texturing and bones are visible

To use this scene representation in an actual game, the following procedure is followed: first, all resources that are needed are loaded and merged into one scene; secondly, a number of empty mesh nodes are added, these empty nodes are then dynamically bound to material and mesh resources depending on what game object they should represent. When a new level should be loaded, the scene is cleared and all resources are reloaded. This procedure avoids the expensive adding and removal of objects during gameplay.

This works well for simple level-based games, but adds some complexity for the game programmer that is required to know how many objects will be needed beforehand and handle the mapping between game objects and scene nodes.

To allow this usage, we implemented support for merging scenes and adding empty mesh nodes. The engine does not support removal of nodes. When merging scenes and adding nodes, remapping of node indices has to be performed which makes these operations fairly costly. Also, when loading a new scene to be merged with an existing scene, some resource management must be performed so that the new scene reuses the effects and textures of the old scene instead of loading duplicates.

5.1.4 Scene Management

Ideally, the game engine should represent a simple but powerful interface to the application code so that it can handle the game logic without having to be concerned with details such as resource management or rendering implementation. We keep the scene representation hidden from the application code, and provide helper functions for modifying the scene. The application code refers to different objects by integer handles and thus is insulated from the actual data structures used. These handles are simply indices into the arrays in the scene representation.

The scene management functions allow for example the reading and modification of node transforms; material, mesh and animation node bindings; animation triggering, setting node parents, setting custom uniform values, modifying light parameters such as colour or attenuation, adding effects to the scene or specific objects or assigning shadow maps to specific lights.

5.1.5 Animations

The engine supports node animations and bone animations but not blend shapes. Animation data is exported from 3ds max to pod files and the animation data for position, rotation and scale is stored in each node that is animated. Bone animations are handled by treating bones as ordinary nodes that are animated just like other nodes. The difference is that in a bone animation, there is a mesh that references these nodes and use them as bones, the mesh also contains the skinning information in the form of bone weights and indices for every vertex. A single mesh can be split up into a number of *bone batches.* This is automatically done by the 3ds max exporter (by setting a maximum number of bones per batch) and each bone batch will be drawn by an individual draw call. The actual skinning is performed by a vertex shader, so the models need to be bound to an effect that has such a shader to enable the bone animation to be played.

Each model file has a certain animation length, and can contain many node or bone animations. This allows entire film-like scene animations to be created in 3ds max, exported to one pod file, and played back in the engine. While useful, this does not suit our needs very well since we use this model format as the actual scene representation for the interactive game world.

A typical game places the following demands on the game engine regarding bone animations:

1. A certain set of bones, a *skeleton,* can be used by several animations and meshes.
2. If a mesh is compatible with a skeleton, it should be able to use all animations for that skeleton.
3. Meshes and animations can be loaded as individual resources and be used by multiple game objects simultaneously.

As described in section 5.1.3, when pod files containing game levels or meshes are loaded, they are merged into the same scene. Game objects are handled by adding empty model nodes and binding these to mesh and material resources. This allows meshes to be handled as individual resources. Animations can be created by exporting a set of nodes and animation data to an individual pod file, excluding any meshes or materials. However, when loading these animations, they are not added to the master scene but instead stored as individual scenes in a list of animations. This is done to allow animations to be handled as bindable resources, and to avoid the complexities of remapping bone indices and having multiple points in time active simultaneously in one scene.

Consequently, support for loading a pod file as an animation was added to the engine, along with support for binding animations to objects. When triggering the start of an animation, a time point is specified that specifies when the animation should start. This allows simple scheduling of animations and also makes it possible to jump to a specific point in the middle of an animation by specifying a triggering point preceding the current time. It is also possible to set a looping property for individual animations.

The engine also supports automatic position interpolation for objects as this simplifies the application code and also results in smoother movement. Position interpolation is activated for an object by specifying start and end position, along with start and end time points.

5.1.6 Built-in and Custom Uniforms

During the project we added many uniforms built into the engine. These uniforms are requested by shaders through uniform definitions in the effect files. All built-in uniforms represent general state that is known or can be computed by the engine. However, many shaders also need specific uniforms that have a more specialised meaning. These custom uniforms are not built into the engine code, but defined in text files. This allows users of the engine to add uniforms without recompiling the application.

The built-in uniforms are those that are general enough to be usable by a large number of shaders. We added these as we needed them to avoid adding features to the engine that we would never use. The 4x4 model, viewing and projection matrices are basic uniforms needed by the vertex shader. For optimisation purposes and ease of use, concatenations of these matrices are also available as the model-view, view-projection and model-view-projection matrices. Making all these matrices available gives the shader writer the freedom to design shaders in multiple ways. Also, a 3x3 normal matrix calculated from the model-view matrix is available to transform normals from object to eye space; a corresponding 3x3 world normal matrix provides a normal transform from world to eye space.

Material ambient, diffuse and specular colours are provided, as well as shininess and opacity values. Light positions are available in either eye or world space, they can also be requested as light directions in eye or world space. Lights have an associated colour and attenuation value. The number of active lights can be queried, and another integer uniform describes which light that has a shadow map enabled (we do not support multiple simultaneous shadow maps). A shadow map texture matrix is used for converting world positions to shadow map texture coordinates.

Bone animations are supported by a bone matrix uniform and an integer uniform naming the number of bone indices per vertex. Finally, a time uniform is available for making animated shaders, a colour mask for tinting or changing the contrast of objects and a screen texture matrix for mapping screen textures onto objects.

The pfx effect files support adding an integer value to the end of a uniform name. This integer value is used by the engine in a number of ways. Most uniforms just ignore it since these uniforms only have a single value, while the texture uniform uses the value for choosing a texture, so that TEXTURE0 refers to the first texture, TEXTURE1 to the next one and so on. Many of the built-in uniforms are actually uniform arrays though, and the integer value is then interpreted as one less than the length of the uniform array. For example, LIGHTPOSITION0 request one light position, LIGHTPOSITION1 requests two and so on.

For the custom uniforms we identified three distinct uses; global uniforms, material uniforms and object uniforms. An example of a global uniform is the ambient colour of the scene, a material uniform might be a reflectiveness value, and an object uniform might be a glow colour. When the value of a custom uniform is requested by a shader, the current object is first searched, if the uniform is not set for this object, the current material is searched and then the global uniforms are searched. We found this system to be very powerful and simple to use. We added support to modify these uniform values from the code, so that it was possible to animate them or change them according to gameplay events. We actually started to use the uniform files for other kind of variables that were not shader uniforms such as setting visibility, disabling shadow casting or depth writes for specific objects.

5.1.7 Scene Rendering

An outline of the rendering method looks like this:

1. Update object positions due to position interpolations.
2. Calculate view transformation and camera frustum.
3. Calculate light positions.
4. For each rendering pass:
    - 4.1. Set up rendering pass.
    - 4.2. For each object sorted in drawing order:
        - 4.2.1. Calculate and set up uniforms.
        - 4.2.2. Draw mesh
    - 4.3. Finish rendering pass.
5. For each post processing effect:
    - 5.1. Draw effect.

Updating object position interpolations is done together with rendering so that it is only done when needed, and can take advantage of the current time to place objects exactly where they should be. The view transformation and camera frustum are calculated from the camera node model matrix and the camera settings. Light positions are extracted from the scene, converted to eye space (by the inverse of the view matrix) and placed in an array for later use.

Hierarchical transforms and node animations are handled automatically by the scene implementation; when a node is asked for its position or model matrix, it calculates its local matrix from its animation data (if it has any) and also recursively asks its parent (if it has one) for a model matrix to multiply with the local node matrix to get the resulting model matrix for the node. This is sped up by caching the last calculated model matrix in the node and returning this if the scene time hasn't changed since.

Additional rendering passes are needed for every light that use shadow mapping, for certain post processing effects such as glow, and for effects such as refraction and reflection. The set up of rendering passes might involve clearing frame buffers, setting OpenGL state and modifying viewing or perspective matrices. Finishing render passes might involve copying frame buffer contents to textures and resetting OpenGL state.

The objects are sorted in a certain drawing order. This sort is done once after loading and adding all objects. Since objects can't easily be moved around, the sort instead builds an array of object indices which indicates the drawing order. This sort is done to make sure that any transparent objects are drawn last and to minimise state changes. Transparent objects are handled by giving transparent materials a high *priority.* Sorting makes sure that objects that have materials with high priority are placed last in the drawing array. Objects that have materials with equal priority are sorted according to the material index, this makes sure that objects with equal materials are drawn directly after each other to enable minimising of state changes.

Bounding volume hierarchies or any spatial acceleration structures are not employed since we only used game worlds with a relatively small number of objects. Frustum culling was however implemented to speed up rendering to textures around refractive objects. The frustum culling is done with bounding spheres per object every frame. The bounding spheres are constructed from the mesh geometry at load time. Spheres were chosen since they are fast to cull against a frustum and the object's rotation does not have to be taken into account.

Much of the core engine functionality is implemented in the code that sets up the uniforms. Uniforms are uploaded to OpenGL as they are requested by the effect used by the object. Many uniform values are only calculated if they are requested by the effect. For optimisation purposes, some caching is done to prevent the same uniform data to be calculated repeatedly. While the uniform set up is mostly trivial, the handling of uniforms regarding light sources is a bit more complex. The drawing is done per object per light (POPL), and shaders can realistically only support a small number of lights at once for performance reasons. For example, it's typical for fragment shaders to support four to eight lights simultaneously – but it's often unsatisfactory to be limited to this amount of lights in the whole scene. For this reason, the engine calculates which lights are the most important for the active object and uploads light uniforms according to this. So if, for example, a shader requests four light positions, the positions of the four most important lights for the current object are uploaded. In the calculation of importance, the distance between the object and the light positions, the light attenuation factors, and the light colours are taken into account. Since this calculation is done per object and not per fragment, it is only an approximation. When the set of lights that are used for an object changes, due to the movement of the object or lights for example, visually disturbing sudden changes in the shading of the object can occur. To minimise this, use fewer lights, smaller objects or more active lights in the shaders.

Vertex buffer objects are used to draw meshes that are exported as interleaved vertex data, other meshes are drawn with vertex arrays. This is a simplification but there are no fundamental difficulties with modifying the engine to use VBO:s for all meshes.

5.2 The Demon Demo

When we realised that we wouldn't be able to get a hold of any mobile OpenGL ES 2.0 capable hardware during the time frame of the project, it was decided that we should test the engine on an OpenGL ES 1.1 capable mobile phone. The hardware that we used was a Nokia N93 mobile phone which is described in section 4.2. We decided to do a simple graphical demo, which should look as good as possible while not taking too much time to implement. The demo features an animated model of a demon, provided to us by Jadestone. We put the model in front of a background image, and added a post-processing glow effect in order to make the demo more visually interesting.

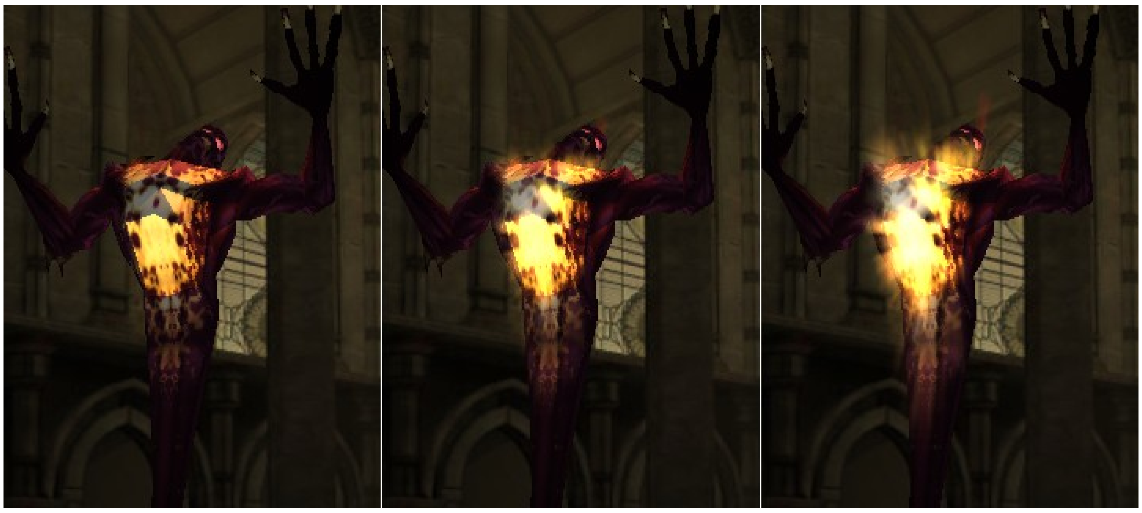

Figure 12: The demon demo with zero, five and nine glow passes

The glow effect is achieved by rendering the model with a special emissive map to a texture, this texture is then additively blended multiple times on top of the scene as a post processing effect. When this is done the texture is stretched by different amounts to create the illusion of glowing rays coming out of the demon. However, since this is a screen space effect, the rays are actually simply stretched away from the centre of the screen.

5.2.1 OpenGL ES 1.1 Adaptation

Most of the engine could be kept intact while adapting it to support GLES 1.1. The main differences are that effect files are not used, that OpenGL state variables are set instead of loading uniforms and that skinning is performed differently. The GLES 1.1 engine does not support any kind of advanced materials, only simply vertex lighting, standard material settings and texturing with a single diffuse map. A new kind of effect file could be designed to be used with this engine, where more advanced materials could be specified, these materials would have to be implemented with texture environment settings. Transformation matrices, texture bindings, lighting and material settings are uploaded to the built in GLES 1.1 state variables as opposed to shader uniforms. Since the PowerVR chip in the N93 supports an extension to GLES 1.1 that is specifically designed to do hardware skinning, we used that. The extension is called *matrix palette* and allows several model matrices to be loaded, along with vertex arrays for bone indices and bone weights. One problem with this extension is that it isn't supported by modern PC hardware, this resulted in the demon demo running slowly with the PC emulation library (presumably the skinning is emulated on the CPU).

5.2.2 Symbian OS Adaptation

Our applications that use the GLES PC emulation library are normal Microsoft Windows programs, however, care was taken to try to isolate the OS specific code. This is the code that handles window initialisation and destruction, window handling such as shutdown events and keyboard input. Naturally, we had to re-implement this part of our program and adapt it for the Symbian operating system. Since the input and window handling required by our application is very simple and example programs using OpenGL ES 1.1 were available in the Symbian SDK this was fairly unproblematic. The reasons that the adaptation was comparatively simple was that we did not use any external libraries or non-standard C++. Furthermore, if we had not used an OS-independent graphics library like OpenGL ES this would have been much more difficult to do.

To develop a Symbian application for the N93, the Nokia S60 SDK was used, S60 is a version of the Symbian OS used by several Nokia phones. Besides making sure that the code compiled for the S60 PC emulator and for the actual N93 hardware, the code project and Symbian installation files had to be created and configured.

5.3 3D Kodo Game Prototype

Making a 3D version of the game Kodo had several advantages. Kodo was designed for mobile phones so it suited us well, furthermore, since Kodo is such a simple game, it was possible for us to actually have time to implement a playable game prototype as opposed to just a technical demonstration. Kodo is a multiplayer game where each player controls a Kodo with one button on the keypad of a mobile phone. The kodos automatically rotate, and walk forward when the button is pressed. The goal of the game is to eat the other kodos. The basic goals were to adapt Kodo to 3D while keeping the gameplay intact and trying to make it look as good as possible while using some interesting graphical shaders.

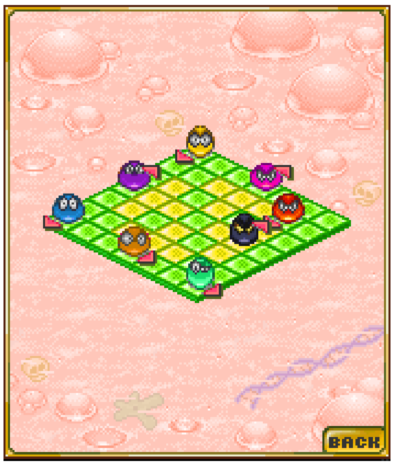

Figure 13: The original version of Kodo

Much of the engine capabilities described in section 5.1 were implemented as the need for them arose while making 3D Kodo. We started by implementing the basic gameplay, adding models, animations and shader effects. At first we didn't have any art direction and actually did the art ourselves. The result of that can be seen in figure 14.

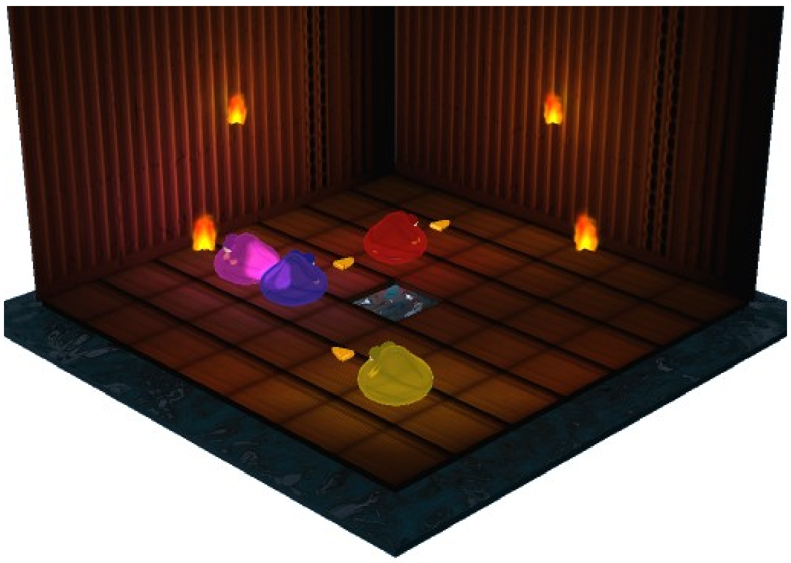

Figure 14: Early version of 3D Kodo

This early version featured effects such as refractive kodos, reflective water and animated torch flames. Our approach was basically to use as many advanced shading effects as possible. This approach was however not very successful in providing a good gameplay experience. Fortunately, we received help from artists at Jadestone. The artists produced a concept image, which can be seen in figure 15, models and textures for the game world and Kodos, as well as animations for the Kodos.

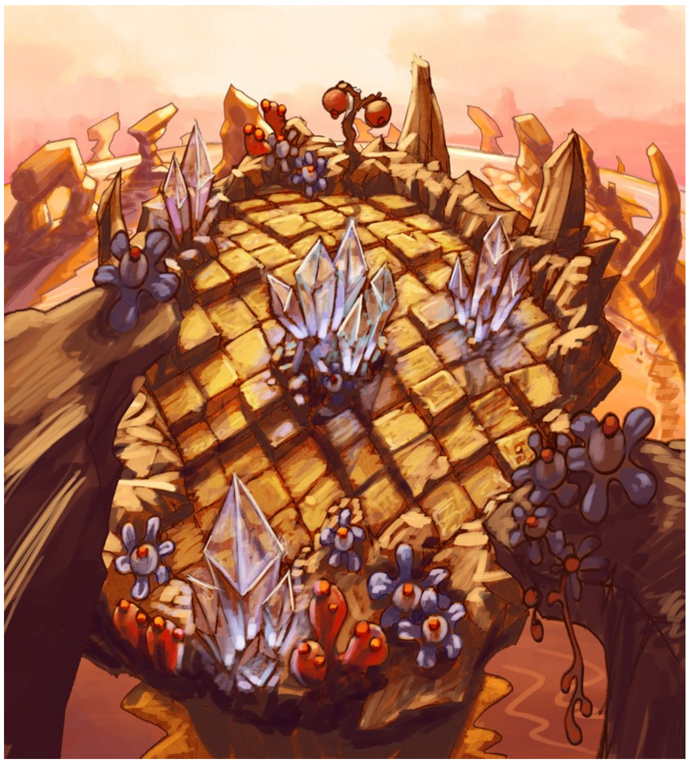

Figure 15: Concept art for 3D Kodo

By working with the artists, implementing the effects they wanted and adjusting different parameter values with them, we managed to achieve a result we were all pleased with. The game is placed in a rocky environment with lizard-like Kodos. Crystals were placed in the middle of the level to allow for a refraction effect to be shown. The game also features a day cycle that seamlessly morphs the world between day, dusk, night and dawn.

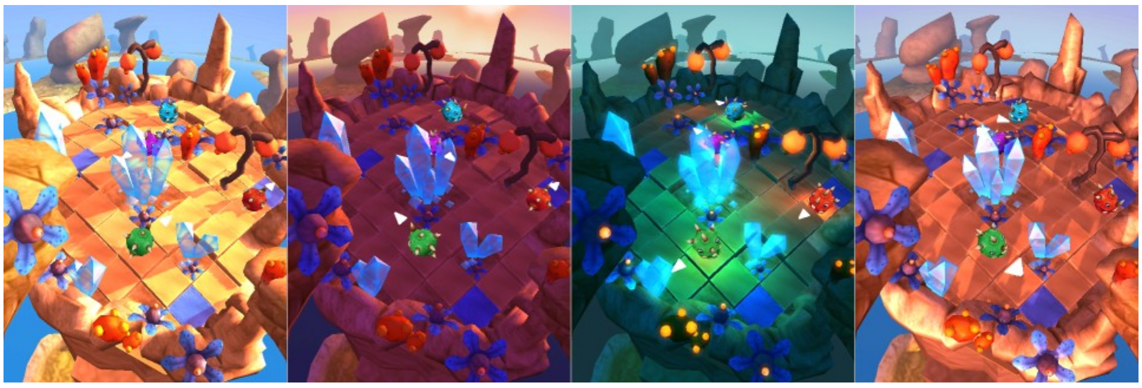
Figure 16: The Kodo day cycle: day, dusk, night and dawn

During the day, the sun casts dynamic shadows that are realised with shadow mapping. During the night, lights in the fruits, kodos and crystals are activated to light the scene, a post processing glow effect is also used on the fruits and crystals. The sun light colour and the ambient colour of the scene are animated to give different lighting settings depending on the time of day. Fogging is used on the rocks in the background and textures are used for the sky and planet sea. There are different textures for the different times of day, and blending between these enables smooth transitions. The fog colour is also animated according to the time of day. We noticed that we needed to use what is sometimes referred to as “modulate 2x” mode for the lighting. This means that the result of the diffuse lighting is multiplied by two so that it can brighten or tint a surface beyond the colours contained in the surface's diffuse map.

6 Evaluation

This chapter aims to evaluate what we did, identify what was most interesting and discuss alternatives and improvements. Recalling the research question of the thesis, "*What are the specific technical considerations relating to graphics that apply when developing 3D games for mobile phones?",* this chapter describes and evaluates the considerations made in the engine and prototype applications that were developed. Particularly, resource handling, scene management and rendering are described.

6.1 3D Game Engine

The 3D engine worked well since we were able to build working demonstrations with it in a short time. However, the engine in itself is not particularly interesting to discuss further so this section will mainly focus on some more general concepts.

6.1.1 Resource Handling

The resource handling in the engine is pretty basic as it requires all resources to be reloaded when changing levels in a game. Due to hardware constraints, mobile games is dependant on good resource handling. Ideally, a mobile game is small in order to facilitate distribution, also the game should start up quickly and load screens in-game should be minimised. The game can obviously be made smaller by compressing the data files; this can be done automatically using algorithms such as zip, as well as intelligently by using fewer resources, omitting unused data, choosing texture compression individually per texture, using smaller data types or generating data procedurally instead of loading it from file. However, these methods might increase load times, especially since mobile processors are relatively slow.

The simplest approach to resource handling is to load all resources at start up, and keep them loaded. This makes resource handling trivial and shortens or eliminates load screens in-game but increases the start up time of the program. This approach might very well be the best one for simple games. However, the limited amount of RAM and the lack of demand paging of memory on mobile platforms might make this approach infeasible and require games to intelligently keep only the currently needed resources in memory. The approach our engine takes minimises RAM usage but might result in very long load times between levels. A variation of this approach is to use a couple of environments, so that multiple levels use the same environment. Reloading of resources should then only need to happen when switching between levels with different environments. A more advanced approach would be to generally reuse resources across level switching and only unload resources that are unused in the current level. This makes the engine more complex, but might be needed since long in-game load screens are frustrating for players.

Multithreading can be used to load resources while drawing to the screen. This can possibly reduce apparent load times by, for example, loading resources while showing start up logos or while the player traverses the main menu. The most advanced form of resource handling is often called *streaming* and generally means that very large levels are supported and that resources are automatically loaded and unloaded during gameplay. This naturally relies on multithreading and can be very complex since, for instance, resources should ideally start to load just *before* they are needed and unloading should be done for resources that will not be needed in the near future. If implemented well, this approach allows a game to consist of a large world, abandoning levels altogether. One problem with streaming that becomes extra problematic on mobile hardware is *memory fragmentation,* which can result from continuously allocating and freeing memory without ever starting over from scratch. This might result in memory depletion if the game is run for long periods of time. Freelists[29] is an example of a method that can help minimise memory fragmentation.

6.1.2 Data Files

In the project we used a number of data files: effect files, model files (also used for scenegraphs and animations), texture files and uniform value files. These were generally good, but not general enough to directly support everything that we wanted to do. The ideal data file format would be compact, fast to load and write, versatile, extendible, writeable and readable by many applications and easy to read and edit with a text editor. Of course, satisfying all these needs at once is impossible; however, two main classes of needs can be identified: the needs during development and the needs of the distributed application. Therefore, the best solution might very well be to have a versatile data format for use during development, and a compact one for the final application distributed to players.

As for effects, the main problem was that we couldn't easily modify uniform values interactively while observing the results. There are a number of ways of accomplishing this, it could be done in the modelling application, in a shader studio or in a custom application. Having the support in the modelling program is good because it is very quick and easy for the artists to edit the values. However, it might be hard or even impossible to integrate all functionality (such as rendering of general shaders, modification of general uniforms etc.) into the modelling application. Shader studio applications are designed to ease rapid development of shaders and provide easy modification of uniforms. However, it might be difficult to design shaders to be compatible both with the shader studio and with the game engine. A custom tool requires much work to implement, but has the potential of being the most powerful since it can be built on the actual game engine. This will enable actual game resources to be loaded and see the same results while modifying values as will be seen in the actual game. Also, other values which are specific to the game or game engine can be modified from the same tool.

The model files are quite powerful as they contain geometry, scenegraphs, material definitions and animations. Since they are binary and compact, they work well as distribution files. However, they weren't quite as powerful as we would have wanted, and modifying them was pretty much impossible since we did not have access to the source of the exporter tools. Having a custom and compact format is good, since this allows you to support everything you want while at the same time having small distributed files. However, implementing loaders and exporters for this format can be costly. Modifying such files can either be done by implementing exporters or importers for all applications that use the format. If many tools are used, this becomes infeasible, a better approach would then be to write a converter tool which can convert a more common format to the custom one. One example of a common format that many newer 3D applications support is the Khronos group's Collada file format.

The texture file format worked really well, since the loading code is compact, many texture formats are supported and the tool that generates the files worked well. The uniform text files also worked well, but support for modifying values interactively is needed and the tool that support this would probably need to be able to output files with similar content to the uniform files.

6.1.3 Scene Management

The scene representation used in the engine is too rigid to be easily used in an advanced game. If the scenegraph were represented as an explicit tree, insertion and deletion of nodes as well as traversal of the scenegraph would be simpler and faster. By using manual handling of memory, for example freelists[29], memory allocation speed and cache coherency can be improved. To support large game worlds with many objects, bounding volume hierarchies or spatial partitioning structures would need to be used, possibly simultaneously. Simple partitioning structures such as octrees[30],[31] and quadtrees can easily be built and modified while the game runs, more advanced structures such as bsp-trees or potential visibility sets would most probably have to be pre-calculated on a PC and stored as data files. These methods might however consume too much memory to be feasible.

The simplest method for referencing scene nodes is to use ordinary pointers to the nodes, this however causes problems in C/C++ based games. The main problem is that game objects often need to refer to each other while objects are constantly being added and removed. For instance, a rocket might contain a reference to the shooter, and when the rocket hits someone, the game tries to add points for the kill to the shooter. But what happens if the shooter object has been removed? Maybe it was a networked game and he has left the server? If the rocket has a raw pointer, it might now point to a deleted object or to a totally different object. Following this pointer might cause a crash or cause some other unexpected result. Many similar problems might arise, and therefore there are many advantages of not exposing node references as raw pointers from a C/C++ based game engine to the application.

For this reason, some game engines use some kind of integer handles[32],[33] to refer to scene nodes and resources. Reference counting together with raw pointers can also be used, reference counting allows an object to be owned by several entities at the same time since any entity that holds a reference to an object is able to keep the object alive. This might be desired for resources such as textures, but probably not for scene nodes since a scene node often has a clear owner (possibly itself, a rocket for example might destroy itself when it hits something or has travelled a maximum distance). If integer handles are used, the engine can use a binary search tree or hash map to map the handles to the actual objects instead of a simple array where handles are used as indices as it is in our engine. Using handles causes some overhead, but it should not be significant since it is often safe for the internal functions and structures in the engine to use simple pointers to speed up processing of nodes. Using integer handles can also simplify the game logic tremendously; for example, if using simple pointers, all pointers to an object have to be found and removed if some object is to be removed. Keeping track of this information can be a very difficult task in a complex game. By using integer handles this becomes trivial - simply don't take care of it, whenever something tries to use this handle later, it should be notified by the engine that this handle is no longer valid and appropriate action, such as clean-up, can be performed.

6.1.4 Rendering

The rendering in the engine generally worked well, but could be done more general and be more optimised. For example, the code for handling several render passes is somewhat complicated and only a little effort has been done to minimise state changes between rendering of objects. If many objects are drawn, the uploading of uniforms would most probably be a performance problem. To achieve better performance, uploading of the same information redundantly for several objects would have to be avoided.

6.2 Demon Demo

Implementing the demon demo was interesting for two main reasons, getting experience developing for Symbian OS, and evaluating the hardware capabilities of the N93 mobile phone. We found that the developer tools for Symbian OS are not as mature as the equivalent PC software, and the Symbian C++ API:s and code conventions are somewhat non-standard and takes some time to understand and get used to. However, we found that getting standard C or C++ code to run on Symbian hardware was a relatively straightforward process.

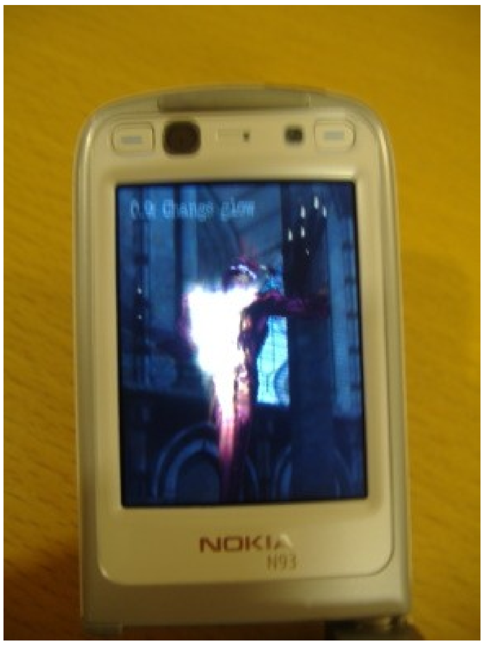

Figure 17: The demon demo running on a Nokia N93 phone

We did not have time to do a thorough evaluation of the performance of the N93's hardware; however, some things can be noted. The demon demo makes use of the CPU mainly for calculating the bone matrices and running the GLES driver which handles all GLES function calls. The graphics hardware's vertex processing was heavily used since the demo utilises hardware skinning and the demon is drawn twice every frame due to the glow pass. The graphics hardware's pixel fill rate was also heavily used since we allowed up to 9 glow passes to be used. The frame rate was not noticeably affected by the number of glow passes, which indicates that the bottleneck lies elsewhere. The conclusions that can be drawn is that the fill rate is impressive and that vertex processing and CPU performance seem to be satisfactory for producing good looking

3D games on the N93 or similar hardware. The main problem for 3D games on this hardware may very well be the amount of RAM, only about 20 MB are free for applications to use on the N93. This is limiting for a 3D game, since frame buffers, geometry, animations, textures and sound effects can easily take up hundreds of megabytes in a typical PC game. However, it should by no means be impossible to create good looking 3D games on such hardware since, for example, the Playstation 2 game console has 32 MB ram and an additional 4 MB video memory and that is evidently enough to create large good looking games.

6.3 3D Kodo Game Prototype

The Kodo game prototype provided us with a number of insights about how well 3D graphics work for mobile games, how shaders can be used in games and how to work together with artists to realise a graphical concept.

The key difference between mobile and stationary hardware regarding graphics is screen size. Smaller screens with lower resolutions directly affects the design of a game since much less information can be shown simultaneously on a smaller screen. Certain game types, such as real-time strategy games, might not be suitable for mobile hardware at all for this reason alone. Thoughtful design of user interfaces should be employed to minimise these problems. Small screen size also affects which kind of graphical effects that are meaningful to use. Kodo was successfully ported to three dimensions and there seems to be no fundamental problems with using 3D graphics in games on small screens. 3D graphics can actually be used successfully to incorporate more information on a small screen without making the presentation confusing. For example, the playing field in 3D Kodo was slightly spherical; this looks more interesting, allows more tiles to fit in the view and would be hard to accomplish in 2D (see figure 18).

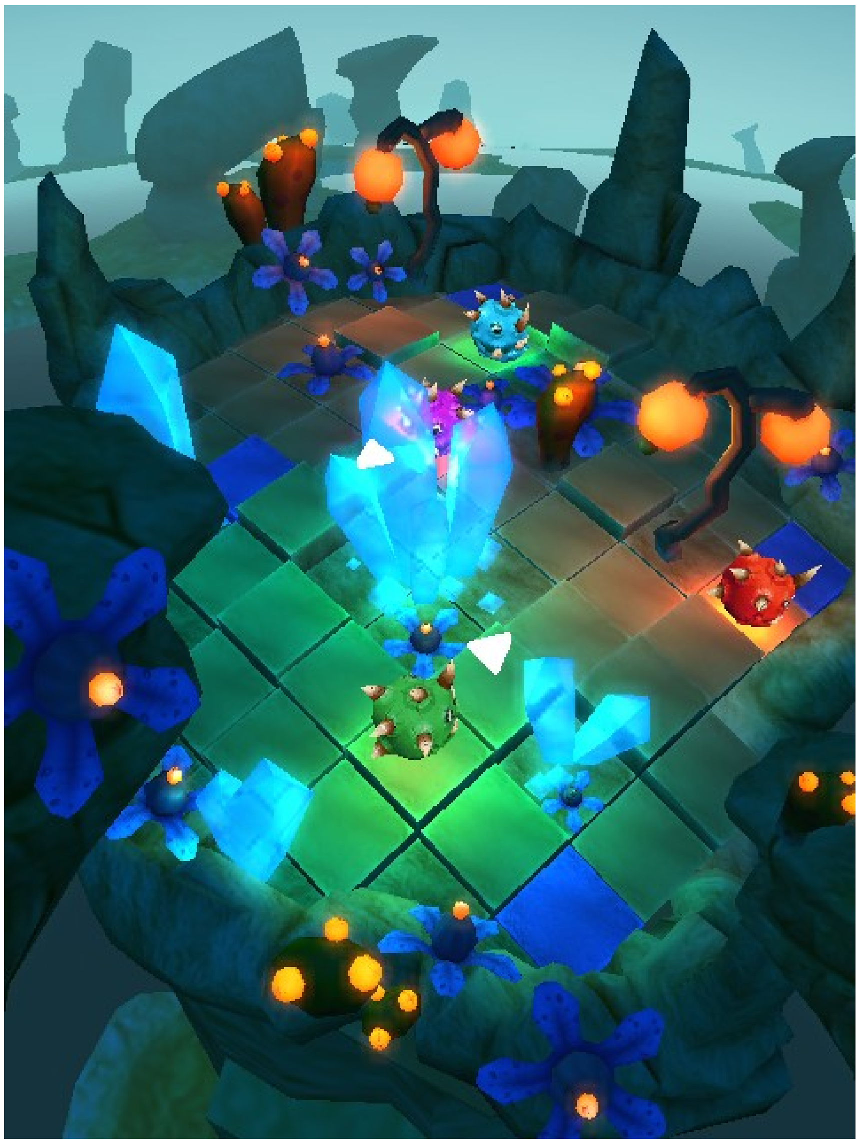

Figure 18: Night in 3D Kodo

We did not have time to incorporate many shader effects into the prototype, the only advanced effect used was refraction, and honestly it did not add much to the game; the crystals could have been drawn with blending instead and the difference would not have been significant for the casual observer. However, the main advantage of shaders is perhaps not that they enable more advanced effects but instead the freedom and precision that they allow. Shaders give the freedom to implement virtually any kind of appearance that can be thought of and the precision to make it work exactly as envisioned. This might involve coming up with a totally new and odd surface effect, but might just as well simply be used to slightly adjust a well known effect to fit a certain situation. For example, in 3D Kodo we used the “modulate 2x” lighting calculation and we blended between different background textures depending on the time of day. Such effects are not necessarily advanced, but might be difficult or even impossible to achieve without shaders. As a result, shaders allow games to look more interesting and stay closer to the artist's vision.

7 Conclusions and Further Work

In most aspects, a 3D game engine for mobile hardware is very similar to a PC game engine. The biggest difference is that a mobile game engine must be small and able to minimise memory consumption. Although the actual engine must be smaller and simpler than contemporary PC game engines, the accompanying tools can be just as advanced since they are used on a PC. Even though enabling advanced effects and providing optimised rendering are important aspects of a 3D game engine, perhaps it is even more important for it to allow game development to be as simple and fast as possible. This is something that became even more clear to us while we worked on 3D Kodo. The engine should provide a simple interface to the application programmer so that is easy to implement game features, this enables rapid prototyping of ideas and speeds the whole process of game development up. It should also be simple for artists to import art assets into the engine, and preferably it should be possible for artists and programmers to modify and test different art assets and settings in real-time in the engine. These features must be addressed by both the engine and accompanying tools.

Some current mobile hardware, such as the Nokia N93, seems perfectly capable of handling advanced 3D graphics without shaders. If phones with equal or better performance soon become more common, 3D games on mobile phones might very well become commonplace.

Since we didn't get a hold of any mobile OpenGL ES 2.0 hardware, many questions regarding the capabilities of future phones are left unanswered. Based on what is currently known, phones with OpenGL ES 2.0 support will probably be capable of fairly advanced graphics, almost rivalling the image quality of recent PC games (albeit at a lower resolution). Such phones will most probably be made available sometime during 2008, although if and when they become commonplace is hard to say. Other interesting questions that remain unanswered on mobile OpenGL ES 2.0 hardware are how the performance of shadow volumes would be relative to shadow mapping and if deferred shading would be a viable option.

Bibliography

[1] James M. Van Verth, Lars M. Bishop. 2004. *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*. Morgan Kaufmann. ISBN 155860863X.

[2] Scenegraph Image: http://opensg.vrsource.org/trac/wiki/Tutorial/OpenSG1/FirstApplication. Last accessed Nov. 2007.

[3] James T. Kajiya. 1986. The Rendering Equation. *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pg. 143–150.

[4] Alan H. Watt. 1999. *3D Computer Graphics*. Addison Wesley. ISBN 0201398559.

[5] Michael F. Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, Neil Hunt. 1988. The Triangle Processor and Normal Vector Shader: a VLSI System for High Performance Graphics. *ACM SIGGRAPH Computer Graphics, 22(4)*, pg. 21–30.

[6] Culling Images: http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/0650/bks/SGI_Developer/books/Optimizer_PG/sgi_html/ch05.html. Last accessed Nov. 2007.

[7] David H. Eberly. 2004. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic*. Morgan Kaufmann. ISBN 012229064X.

[8] Michael Wimmer, Jiři Bittner. 2005. Hardware Occlusion Queries Made Useful. *GPU Gems 2*, pg. 91–108. Addison Wesley. ISBN 031335597.

[9] John W. Ratcliff. 2001. Sphere Trees for Fast Visibility Culling, Ray Tracing,and Range Searching. *Game Programming gems 2*, pg. 384–387. Charles River Media. ISBN 1584500549.

[10] Miguel Gomez. 2001. Compressed Axis-Aligned Bounding Box Trees. *Game Programming gems 2*, pg. 388–393. Charles River Media. ISBN 1584500549.

[11] Octree Image: http://dev.gameres.com/Program/Visual/3D/OCTREETutorial6.jpg. Last accessed Nov. 2007.

[12] Mark Segal, Kurt Akeley. 2006. *The OpenGL Graphics System: A Specification*.

[13] OpenGL Pipeline Images: http://www.khronos.org/opengles/2_X/. Last accessed Nov. 2007.

[14] Randi J. Rost. 2006. *OpenGL Shading Language*. Addison-Wesley. ISBN 0321334892.

[15] Khronos Group. 2007. *OpenGL ES Common/Common-Lite Profile Specification Version 1.1.10 (Full specification)*.

[16] Khronos Group. 2007. *OpenGL ES Common Profile Specification 2.0*.

[17] William Shoaff. 2000. *A Short History of Computer Graphics*. http://cs.fit.edu/~wds/classes/graphics/History/history/history.html, Last accessed Nov. 2007.

[18] Wikipedia. 2007. *Comparison of NVIDIA Graphics Processing Units*. http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units, Last accessed Nov. 2007.

[19] 3D-Test. 2007. *Q&A Ed Plowman, ARM Mali Product Manager.* http://www.3d-test.com/interviews/arm_2.htm, Last accessed Nov. 2007.

[20] Wikipedia. 2007. *Nokia N93*. http://en.wikipedia.org/wiki/Nokia_N93, Last accessed Nov. 2007.

[21] Imagination Technologies. 2007. *PowerVR MBX - Overview.* http://www.imgtec.com/PowerVR/Products/Graphics/MBX/index.asp, Last accessed Nov. 2007.

[22] Laszlo Kishonti. 2007. *GLBenchmark*. http://www.glbenchmark.com, Last accessed Nov. 2007.

[23] Mathew Kumar. 2007. *GDC China: Developing For The N-Gage And 3D Graphics In A 3G World*. http://www.gamesondeck.com/feature/1677, Last accessed Nov. 2007.

[24] Imagination Technologies. 2007. *PowerVR SGX - Overview.* http://www.imgtec.com/PowerVR/products/Graphics/SGX/index.asp, Last accessed Nov. 2007.

[25] Ryan Block. 2007. *Toshiba's G900 and E01 with Windows Mobile 6*. http://www.engadget.com/2007/02/12/toshibas-g900-and-e01-with-windows-mobile-6/, Last accessed Nov. 2007.

[26] Nathan Kirsch. 2006. *The Future Of Handheld Gaming: GoForce 5500*. http://www.legitreviews.com/article/327/1/, Last accessed Nov. 2007.

[27] AMD. 2007. *Imageon 2380/2388 Overview.* http://ati.amd.com/products/imageon238x/index.html, Last accessed Nov. 2007.

[28] ARM. 2007. *Mali Graphics Solution - Products Overview.* http://www.arm.com/products/esd/multimediagraphics_malioverview.html, Last accessed Nov. 2007.

[29] Paul Glinker. 2004. Fight Memory Fragmentation with Templated Freelists. *Game Programming Gems 4*, pg. 43-50. Charles River Media. ISBN 1584502959.

[30] Dan Ginsburg. 2000. Octree Construction. *Game Programming Gems*, pg. 439-443. Charles River Media. ISBN 1584500492.

[31] THATCHER ULRICH. 2000. Loose Octrees. *Game Programming Gems*, pg. 444-453. Charles River Media. ISBN 1584500492.

[32] SCOTT BILAS. 2000. A generic Handle-Based Resource Manager. *Game Programming Gems*, pg. 68-79. Charles River Media. ISBN 1584500492.

[33] BRIAN HAWKINS. 2002. Handle-Based Smart Pointers. *Game Programming Gems 3*, pg. 44-48. Charles River Media. ISBN 1584502339.

TRITA-CSC-E 2008:024
ISRN-KTH/CSC/E--08/024--SE
ISSN-1653-5715