On the Use of OpenGL ES 2.0 Shaders for Mobile Phones using Cross-Platform Middle-Ware

MÅNS OLSON

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2010
On the Use of OpenGL ES 2.0 Shaders for Mobile Phones using Cross-Platform Middle-Ware

MÅNS OLSON

Master’s Thesis in Computer Science (30 ECTS credits) at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Henrik Eriksson
Examiner was Lars Kjelldahl

TRITA-CSC-E 2010:067
ISRN-KTH/CSC/E--10/067--SE
ISSN-1653-5715
Abstract

In this report, we explore the use of advanced graphics techniques across multiple platforms using cross-platform middle-ware on OpenGL ES 2.0-enabled devices. In particular, we have evaluated the use of Unity and Airplay SDK for graphically advanced applications on iPhone, iPod Touch and Nexus One (Android) devices. A prototype game, Kodo, was developed as a test platform for these tests. We found that many advanced shader techniques can be successfully used on current high-end mobile devices, though careful consideration is required when applying some of the more costly techniques, such as fragment lighting, normal mapping and full screen post-processing. In a related report, Nyström explores the use of local multi-device multiplayer games, using an iPad as the main playing area and other devices as remote controls for each participant.
Om användning av OpenGL ES 2.0-shaders för mobiltelefoner med hjälp av anpassningsprogram för flerplattformsutveckling

Sammanfattning

I den här rapporten undersöks hur avancerade grafiktekniker kan tillämpas på olika plattformar med hjälp av anpassningsprogram för flerplattformsutveckling. Särskilt fokus lades på användning av Unity och Airplay SDK för utveckling av avancerade grafiska applikationer på iPhone, iPod Touch och Nexus One (Android). En spelprototyp, Kodo, utvecklades som testplattform för dessa tester. Det visade sig att många avancerade shadertekniker framgångsrikt kan användas på moderna mobiltelefoner. Användning av vissa av de mer kostsamma teknikerna, som fragment lighting, normal mapping och full screen post-processing, kräver dock särskild eftertanke. I en relaterad rapport utforskar Nyström hur ett lokalt flerspelarläge kan göras med en iPad som spelyta och andra enheter som fjärrkontroller.
1 Introduction

Making cross-platform applications for mobile phones and devices is not trivial. This is particularly true for graphics-heavy applications such as games. Some of the key issues are related to differences in screen size and resolution, graphics hardware, and processing power. The majority of older mobile devices have no dedicated graphics hardware, though devices supporting the hardware-based Open Graphics Library for Embedded Systems (OpenGL ES) are becoming increasingly common.

In this report, we explore methods for running state-of-the-art graphics applications on mobile devices using cross-platform middle-ware. In this way, one application can be easily ported from one platform to another, saving the developer a considerable amount of work.

1.1 Problem Statement

The work described in this report is primarily an attempt to answer the following question:

*How may cross-platform middle-ware support graphics-intense mobile applications on high-end mobile devices?*

The scope of this question is potentially very large. In order to narrow down the area of investigation, the work has been divided into two parts: Evaluation of cross-platform middleware, and evaluation of a set of graphics techniques for mobile devices. In particular, the graphics part will be focused on the use of vertex and fragment shaders on mobile phones. As part of these graphics evaluations, a prototype game – Kodo – will be developed.

1.2 Notes

Part of the present work – in particular the Kodo prototype – is a collaboration between Måns Olson and Martin Nyström. Thus, the reports by Olson and Nyström include a shared background section and a shared prototype section (chapters 2 and 4).

The remainder of the report is specific to the area of investigation: Måns Olson focuses on using cross-platform middleware to support advanced mobile graphics, while Martin Nyström focuses on new methods for multiplayer interaction on high-end mobile devices.
2 Background

In previous work, Gustavsson [1] and Olsson [2] explored the use of OpenGL ES (GLES), primarily using a PC emulator. In particular, they explored the latest version of the library, GLES 2.0. They discussed, in some detail, how a 3D game engine can be constructed to run well on mobile hardware. As one would expect, many optimizations used for 3D desktop applications can be directly applied to mobile applications as well.

In these projects, Gustavsson focused on the construction of the game engine itself, while Olsson focused on the use of shaders in mobile applications. Both of these aspects are relevant to the present work. Shaders, in particular, is an area that will be explored. Much of the work concerning the engine can be generalized and reused, though some aspects of it (model loading, texture compression and animation logic) can be handled by third-party software. Such code will often be included in cross-platform middleware, such as Unity [3] or Airplay SDK [4]. This will be discussed in more detail in chapter 3.

While the results presented in these previous reports are theoretically sound, it is not clear that one can directly extrapolate conclusions that would hold true for today's mobile market. The GLES 2.0 tests run were made exclusively on emulator software, which makes it difficult to apply any performance results on GLES 2.0 hardware. At the time of writing, hardware supporting GLES 2.0 has become available on high-end mobile devices. It has been our intention to continue the work of Gustavsson and Olsson by performing tests on such hardware, including iPhone 3GS and Android devices such as the Nexus One.

In order to be able to work collaboratively, a prototype game was developed by Gustavsson and Olsson. The game concept, Kodo, is an existing intellectual property (IP) owned by Fabrication Games, that lends itself well to performance tests. Simple controls and gameplay makes Kodo an appropriate test platform, since it is desirable that the core game mechanics are not a development bottleneck. Figure 1 shows Gustavsson's and Olsson's prototype running on an emulator.

Munshi et al. have written a thorough guide to GLES 2.0 in the “OpenGL® ES 2.0 Programming Guide” [24]. The book covers shaders, the OpenGL ES Shading Language, as well as the GLES 2.0 pipeline. In particular, the chapter on GLES 2.0 as applied to iPhone (available from the book's website [27]) is relevant to this work.
2.1 Use of the Kodo Prototype

It has been our intention to make a new prototype based on the Kodo game concept. Although both the old and the new prototypes revolve around the same simple gameplay mechanics, they do not use shared program code; the old prototype was written to work on an emulator for specific hardware, while the new one, written by us, functions on multiple platforms. In this manner, many assets needed for 3D performance tests – in particular, models, animations, and textures – could be reused, which allowed for rapid prototype development, while also providing shared grounding between the two projects. This, in turn, allows a degree of comparison between the emulator tests of previous projects and the device tests that will be performed for this project.

Throughout this text, we will be returning to our version of the Kodo prototype, which we use to illustrate both implementation details and findings about performance. We also use the prototype as a ”real-world example” – that is, an application which can be reasonably compared to both PC games and some of the more advanced games released.
through App Store or Android Market. Since the prototype relies heavily on advanced graphics techniques, it has helped us identify relevant performance bottlenecks (and also boundaries) on current GLES 2.0 devices. In addition, the core gameplay mechanics in Kodo are very well suited to multiplayer conditions, which is a requirement for the work described in Nyström's report [17].

The scene in the older Kodo prototype used by Gustavsson and Olsson is static to some extent; the camera is set to view the scene only from a given angle, and does not move in any direction. Never moving the camera can be a useful technique to maintain a steady frame rate; one can optimize frame rate for a specific scene. If the scene is more dynamic, and number of polygons and textures is unknown, one needs to optimize the frame rate for the worst case scenario.

There are a number of other benefits to having a static camera. First, one can make sure (either when loading a level or when constructing it) that a level consists of only one model, thus requiring fewer draw calls (see section 2.5.1). Second, since the camera is fixed, one knows beforehand exactly which surfaces are hidden. These can be removed with modeling software before a level is loaded, reducing the amount of backface culling in the scene. Third, no view frustum culling [7, pp. 664-667; also see section 2.5.1] needs to be performed since one knows beforehand exactly what is to be rendered in the scene. Fourth, certain visual effects become simpler to make. For example, having a horizon together with a dynamic camera can be expensive if the camera can move away from the horizon, thus increasing the number of polygons between the camera and the horizon. This can be remedied using level of detail techniques. These performance aspects will be discussed in more detail in section 2.5.

The Kodo prototype developed for this project does not have a fixed camera. This increases the complexity of the program (and can potentially create performance issues), but allows for levels larger than the initially seen area. By allowing the camera distance to be changed (using for example pinch-to-zoom), the player can choose between more detailed visuals (camera closer) or better overview (camera further away).

2.2 3D Engine Elements

Common graphics hardware renders a scene by drawing polygons – essentially triangles or sets of triangles in 3d space – which are then rasterized onto a two-dimensional surface. This surface can then be copied to the screen. Upon rasterization, a texture can also be mapped onto the polygon, thus providing additional color information. Additional stages include coloring of each polygon's vertices and lighting. Vertex-based values are interpolated over the polygon's surface to modify the color in each pixel drawn.

Textures

Textures are usually two-dimensional images loaded from the device's storage, though they can also be generated procedurally in an application. Depending on what hardware is used, textures can also be one-dimensional and three-dimensional, though GLES 2.0 supports only two-dimensional textures. By assigning texture coordinates to the vertices of a polygon, one can decide how the mapping of an image onto a polygon behaves.
Models
Higher-level graphics engines often abstract away polygon rendering by allowing developers to use models. A model is essentially a list of polygons and associated information, such as texture coordinates or vertex normals. These can be created with third-party modeling programs such as Autodesk Maya [5] and imported to an application using a model loader. It is also common that models have other associated elements, such as a skeleton and animation data.

Skeletons
Skeletons consist of a set of hierarchically arranged bones. These can be transformed in 3d space while preserving the hierarchy, so that moving one bone (for example an arm) affects the position and rotation of other bones (for example those of a hand). A skeleton can be set to affect a model by associating each vertex in the model with a number of bones and giving these associations individual weights; for example, the weights could be set so that moving your arm affects the position of the skin on your hand, while moving your head will not. The process of associating vertices with bones in this manner is known as skinning. Figure 2.1 shows a skeleton used for bone animation rendered on top of a low-polygon Kodo model. Shown are also two controllers for leg movement.

![Figure 2.1. A skeleton (used for animation) rendered on top of a low-polygon Kodo model.](image)

Animation
One of the most common methods for animating 3D models is bone animation. This method can be implemented using either forward kinematics or inverse kinematics. The former is a technique for directly interpolating bone positions between certain positions, defined in keyframes. For example, one might specify that after one second has elapsed, an arm may have been lifted a certain amount. In forward kinematics, a joint may affect how connected bones move; say, moving an arm could cause the hand to move – but no limitations such as physical laws are taken into account when calculating this.
Inverse kinematics, on the other hand, deals with calculating the position of certain bones given the position of other bones and certain restrictions. For example, a hand always needs to be connected to its arm. By considering a hand in a given position, one could calculate what the position of the arm needs to be for the hand's position to be possible. Often, calculations such as these are performed ahead of time in modeling software and saved in keyframes, so that the application that uses the model can use the cheaper method of forward kinematics. In other cases – such as when doing ragdoll physics (a type of animation often used for character death animations in games [22]) – an application will need to support real-time inverse kinematics.

Another common animation method is *blend shapes*, where multiple versions of a model (each with the same number of vertices) are loaded, and animation performed by interpolating between these variations.

**Shaders**

*Shaders* are programs specifically written to run on graphics hardware (or more recently, on general-purpose processing units). In the GLES 2.0 specification, there are two kinds of shader programs: *vertex shaders* and *fragment shaders*. An additional type of shader, known as *geometry shaders*, is supported by modern desktop graphics chipsets, but is not currently supported on mobile devices.

Vertex shaders can modify vertex data for polygons as they get drawn, for example by altering their positions or normals. Fragment shaders modify fragments – which roughly correspond to pixels – as they get drawn. For example, a fragment shader could change the color of a fragment or discard it, thus preventing it from being drawn. Vertex shaders can pass information onto fragment shaders; when they do, the vertex shader's output gets interpolated between vertices to provide input at each fragment.

### 2.3 Graphics Libraries

A *graphics library* can be either a low-level library that provides an interface against hardware functionality, or a higher level library that provides an interface against a lower level graphics library. In this section, by graphics library we mean the former; a low-level set of functions that let developers use functionality provided by dedicated graphics hardware.

There are two common graphics libraries used for 3D games today: Khronos Group's 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, and is generally more popular than OpenGL in this area on platforms that support it (PCs running Microsoft Windows and Microsoft’s gaming consoles).

### 2.4 OpenGL ES Versions

Three different versions of OpenGL ES have been released: GLES 1.0, GLES 1.1 and GLES 2.0. They are defined relative to the OpenGL 1.3, 1.5 and 2.0 specifications, respectively. This means that the embedded systems versions of OpenGL are based on
desktop versions of OpenGL; most of the features and syntax are identical, although a few features have been stripped in the ES versions. In addition, the ES versions support a few features that are not supported in the desktop versions.

GLES 1.0 does not include direct mode for rendering, which means one cannot draw polygons by sending vertices to OpenGL one vertex at a time [6]. Instead, one must use vertex arrays. It also introduces functions for doing fixed-point math, as embedded system devices often do not have dedicated floating-point math hardware. Many features that can make development easier but are not strictly needed have also been removed, such as pushing and popping the OpenGL state.

GLES 1.1 is similar to GLES 1.0, but improves support for multitexturing as well as makes it a mandatory part of the standard (rather than an extension) [6]. It also adds automatic mip-mapping, Vertex Buffer Objects (VBOs), user-defined clip planes and other features not present in GLES 1.0.

GLES 2.0 is a relatively large step from GLES 1.1. It removes the fixed-function pipeline, which among other things handles multitexturing and lighting, in favor of a more modern shader-based pipeline. Unlike the change from OpenGL 1.5 to OpenGL 2.0 on desktop machines, GLES 2.0 has no legacy support for the fixed-function pipeline, which means that the setup process for developers is made more complex; any developer who wishes to use GLES 2.0 must have knowledge of shaders. However, many devices that support GLES 2.0 also support GLES 1.1. This makes the transition from GLES 1.1 to GLES 2.0 easier for developers.

Shaders in GLES 2.0 are written in the OpenGL ES Shading Language, GLES SL, which is an adoption of GLSL, the shading language used to write shaders for OpenGL 2.0.

2.5 Performance Considerations

Steady frame rate is often a crucial factor in games. Real-time games in particular are often dependent on smooth frame rates for both visual quality and input response time. In order to keep the frame rate steady, the application should keep the time between updates as low as possible. This means keeping the amount of calculations performed before drawing each frame to a minimum.

In general, the most time consuming operation of a three-dimensional game is rendering. This means that unless one is running complex physics simulations, artificial intelligence or other computationally expensive algorithms, graphics will most likely be the primary performance bottleneck.

The process of achieving high rendering performance can be divided into two parts. First, content that can be rendered efficiently has to be created. Second, one should minimize the number of rendered objects and the amount of work needed to render those objects.
2.5.1 Rendering Optimizations

In order to reduce the amount of geometry that is rendered, and to reduce the rendering overhead, one can use any of a number of optimized rendering techniques commonly used in games. A number of useful techniques are described briefly in this section.

**Backface Culling**

The idea of culling is to discard any geometry that cannot be seen. In many scenes, close to half the geometry is facing away from the viewer and is therefore invisible. Thus, one could save a lot of processing power by simply culling all of the polygons that are not facing the camera. To do this, one must either compute the normal of the projected polygon in two-dimensional screen space, or the dot product between the polygon’s normal and a vector from the camera to an arbitrary point on the plane on which the polygon lies. One can then determine whether the polygon is front- or backfacing by looking at the sign of the resulting z-component or dot product.

**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 optimized by discarding geometry outside of the *view frustum*. This is called *view frustum culling* [7, pp. 664-667]. Such a frustum has the geometrical shape of a rectangular pyramid delimited by a near and far viewing plane, as shown in figure 2.2.

To speed up object culling, a pre-computed *bounding volume* [7, pp. 725-726] can be used instead of the actual mesh geometry to calculate occlusion. The most common bounding volumes are spheres, boxes aligned to the coordinate system axes (*Axis Aligned Bounding Box*, or *AABB* [7, pp. 726, 729]) and boxes aligned to the object (*Oriented Bounding Box*, or *OBB* [7, pp. 729-730]). 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. Frustum clipping thus relies on some form of *frustum intersection* [7, pp. 19, 771-778].
Occlusion Culling

Objects that are fully occluded by other objects will not be visible to the viewer and can thus also be discarded. This is illustrated in figure 2.3, where the wireframe objects are culled as they are completely hidden behind the cylinder closest to the viewer. Partially occluded objects can be rendered more efficiently by removing the object's occluded polygons. This differs from depth testing by working on the polygon or object level, while depth testing works on the fragment level.
A number of methods can be used to calculate occlusion culling. Some of the most noteworthy methods are potentially visible set, portal rendering and hardware occlusion queries.

Potentially visible set [18] divides a scene into regions and precomputes visibility between them. This allows for quick indexing, so that one can obtain high quality visibility sets at runtime. However, since visibility is precomputed, moving objects from one visibility region to another may be difficult or even impossible depending on the implementation.

Portal rendering divides a scene into sectors (rooms; also known as cells) and portals (doors), and computes visibility of sectors at runtime by clipping them against portals [7, pp. 667-670]. 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 querying is a way of asking the graphics hardware if any pixels were drawn during the rendering of a particular object [7, pp. 674-677]. 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 the object can safely be skipped. This method works on dynamic scenes (that is, without precomputing occlusion), but requires modern hardware and causes some overhead due to the additional draw calls.

**Bounding Volume Hierarchies**
Culling a large amount of objects with view frustum culling and occlusion culling can be a significant burden on the CPU. This can be alleviated by using a spatial...
acceleration structure, such as a bounding volume hierarchy (BVH). One such hierarchy is shown in figure 2.4 [21]. The idea of a BVH is to organize a scene in a tree-like structure where the root comprises the whole scene and each leaf contains the bounding volume of an individual mesh. This makes many spatial queries – such as frustum culling – much faster, since an entire subtree can be tested without having to test every leaf. If a node gets culled, its entire subtree will get culled as well. However, this technique may incur some computational overhead since a subtree of the volume hierarchy has to be updated every time a node changes.

Several similar methods for spatial acceleration structuring exist. Binary space partitioning (BSP) trees divide space into subspaces and then sorts the geometry into these spaces. This is done recursively for each subspace. The resulting tree can be used similarly to those generated from bounding volume hierarchies, with one notable difference: if the trees are traversed a certain way, the geometrical contents can be sorted front-to-back from any point of view.

**Level of Detail and Mipmapping**

The idea behind level of detail (LOD) is to switch between a set of models depending on how much it contributes to the rendered image. A close-up of a creature would use a detailed version of the creature's model, possibly consisting of thousands of triangles. If the creature was to stand in the horizon, it would use a less detailed model. Since the creature is distant, it would still look approximately correct while having only a fraction of the performance impact it would have in full detail.

Mipmapping (Mip from *multum in parvo*, meaning “many things in a small place”) is a related technique applied to textures instead of models [7, pp. 163-166]. It is intended to increase rendering speed and reduce aliasing artifacts. Each bitmap image of the mipmap set is a version of the main texture, but at a certain reduced level of detail. Although the main texture will still be used when the viewer is sufficiently close, the renderer will switch to a suitable lower-detail mipmap image when the texture is viewed from a distance (or otherwise scaled down to a smaller size). Rendering speed increases since the number of texture pixels (texels) being processed can be much lower using mipmap images. Artifacts are reduced since the mipmap images are effectively already anti-aliased, taking some of the burden off the real-time renderer.
Optimizations Related to Hardware

In-depth knowledge of a device’s hardware can be a great asset when optimizing an application. In order to achieve good performance, one must also carefully consider the algorithms used in the application, as well as test the application thoroughly on device. Two general principles that relate to both the hardware and the algorithm aspects of 3D engines are minimizing state changes and minimizing draw calls.

Minimizing state changes can be done by sorting objects by material, shader, texture or geometry. The state then only needs to be changed when a new set of objects grouped by the above criteria is to be rendered, as opposed to fully switching state for every object drawn. Minimizing state changes in this manner can increase the performance of an application [8].

If a scene is not dynamic, the order of the objects can be precomputed. Otherwise, all objects currently in view may have to be re-sorted every frame. A potential problem is caused by transparent objects, since they need to be rendered after any opaque objects in the scene and preferably in strict back-to-front order. This makes minimizing state changes difficult. For this reason, it is desirable to keep the number of transparent objects to a minimum. In order to obtain good results using sorting, one should strive to use few materials, shaders and textures (for example by packaging several textures into texture atlases).

Using fewer types of geometry buffers – for example only triangle strips – is also useful, because such unified buffers can be merged into larger buffers if they share one and the same OpenGL state. Minimizing the number of draw calls is thus related to minimizing state changes. One example where this technique is useful is for particle systems. The simplest approach to rendering such a system is to draw each particle individually, but in practice this is often much too slow. Instead, particles should be batched into one geometry buffer. In this manner, even a particle system consisting of thousands of particles can be efficiently rendered.

2.5.2 Content Creation

The graphics hardware on mobile devices do not perform as well as their desktop counterparts, which means that one must be particularly careful about such things as polygon count and texture resolution when developing 3D applications for mobile devices. This often means that one has to decide on strict specifications for maximum polygon count, resolution, number of alpha channels and other detail indicators of the content of an application.

Some of the more important variables that one should pay attention to when creating content for 3D applications are:

- Model polygon count
- Textures: resolution, number of alpha channels, number of colors, compression
- Animation type and complexity
All of the above are, of course, highly dependent on variables such as device hardware and the context of use. In addition, one needs to take into account what a typical scene in the application will look like as well as what parts of the scene need to be more detailed than other. These needs must then be compared with the limitations imposed by the hardware so that the engine can be designed appropriately.

The recommended polygon budget for the iPhone using Airplay SDK is 2500-5000 polygons [9], but these numbers can differ greatly depending on whether the polygons are animated, whether they are lit, and whether there are any other computations dependent on the amount of polygons and/or vertices in a scene.
3 Cross-Platform Middle-Ware

Multi-platform development is difficult because each platform targeted by an application has different properties. Cross-platform middle-ware are toolkits that can be thought of as platforms in their own right. Ideally, instead of developing a unique version for each targeted platform, a developer can make a version that uses the middle-ware as a platform and then deploy it to all platforms supported by the middle-ware. In practice, it is nonetheless common that one needs to make adjustments to an application for it to behave well on a given platform. Figure 3.1 illustrates how cross-platform middle-ware can act as a layer between an application and a set of platforms.

Cross-platform development for mobile phones is particularly hard because of fundamental hardware differences between different phone models. Varying screen resolution and size, as well as graphics chipsets and library support, are all significant. Screen resolution in particular is problematic on devices with small screens because user interface components need to be large enough to be clearly visible and legible. A number of common workarounds used for desktop applications, such as making sufficiently large fixed-size components and placing them around the edges of a screen, are not viable on mobile devices. Varying input methods is another common problem that has to be handled by multi-platform mobile applications. Physical keys, full-size keyboards, touch sensors, multitouch sensors, accelerometers and cameras are all input units that are present on some devices, but not on others.

In addition, there is a significantly larger number of common platforms for mobile devices than for desktop machines. iPhone OS (Apple), Android (Google), Windows Mobile (Microsoft), Symbian (Symbian Foundation) and Blackberry OS (Research In Motion) are all platforms that have relatively significant portions of the smartphone market [19], whereas only a handful of platforms are widely used by game consumers on desktop machines (Windows, OS X, and Linux systems).
Middle-ware such as Airplay SDK [4], Unity [3] and PhoneGap [10] support cross-platform development by giving developers a single platform to work against, and providing the means to deploy applications based on this platform to multiple mobile devices. While PhoneGap is used primarily for lightweight applications, Unity and Airplay SDK were both designed with graphics-heavy games in mind: Unity is marketed as a 3D game engine, and Airplay SDK was developed by Ideaworks Labs [11], an offshoot of the game developer Ideaworks 3D.

Middle-ware should ideally expose a wide range of platform functionality. For example, aside from supporting advanced graphics, Unity and Airplay SDK both provide support for playing sounds and interacting with a device's file system.

3.1 Unity

Unity is a cross-platform games engine that targets Windows and OS X desktop machines, browsers (on Windows and OS X systems), Nintendo Wii and iPhone/iPod Touch (and more recently, iPad). As such, it is not solely designed for making mobile applications, though it has been used to produce high-quality games (such as Tumbledrop [12]) for the iPhone.

From a developer perspective, Unity is a high level tool: much of the functionality contained in Unity is available through a graphical user interface, and the inner workings of the engine are typically hidden from the user. Asset loading – including models, animation, sound and images – can be easily handled by Unity, and many common file formats are recognized and accepted by the toolkit. Unity's user interface is shown in figure 3.2. The scene shown in the figure is from an example Unity project provided with Unity.

The ability to release games for desktop machines and browsers can be a considerable distribution advantage, though this study focuses on applications run on mobile devices. Unfortunately, Unity supports only one mobile platform – iPhone OS. For this reason, Unity was discarded in favor of Airplay SDK (see below).
3.2 Airplay SDK

Airplay SDK is a cross-platform middle-ware that can be used to develop both games and other applications for mobile devices. At the time of writing, it supports a number of mobile operating systems – iPhone OS, Android, Symbian, Windows Mobile, Brew and Maemo can all be targeted using Airplay. Since Airplay was originally developed as an inhouse game engine at Ideaworks, it has many properties useful in the process of making games. Model and animation loading and rendering is built into the API, using a custom format for models and animations. The SDK also includes tools to export to these formats from some commonly used modeling programs (Autodesk Maya [5] and Autodesk 3ds Max [13]).

Airplay SDK also contains a software 3D engine and a custom graphics library, IwGx. The latter mimics many of the functions in OpenGL ES, and can be run either on the software engine provided by Ideaworks, on GLES 1.X, or on GLES 2.0. This makes it easy for developers to develop advanced graphics applications without having to worry about whether the targeted platforms have hardware graphics support. Additionally, IwGx lets developers use the full power of shaders in GLES 2.0 while providing a simple means of creating fallbacks to other rendering methods. Because IwGx provides a replica of the fixed-function pipeline in GLES 1.X, one needs not make a difference between GLES 1.X and GLES 2.0 platforms unless one wants to use new functionality in GLES 2.0.
Since both Unity and Airplay SDK abstract away many of the issues surrounding different OpenGL versions on different platforms, both provide higher-level graphics support than OpenGL. Airplay also provides direct access to OpenGL ES. Airplay is considerably more reliant on user programming than Unity; there are a number of graphical tools in Airplay (including a model/animation exporter and viewer), but the scale and scope of these tools are lesser than those in Unity.

3.3 Evaluation

Airplay SDK was used in the development of the Kodo prototype. As the prototype developed, both positive and negative aspects of the SDK were found. In this section, these aspects will be discussed in the context of the Kodo prototype.

On the one hand, while using middle-ware such as Airplay may not lock developers to a certain platform, it does lock them to a given middle-ware. If a critical problem with the middle-ware is discovered on some platform, it is not certain that the developer of the middle-ware is available or has time to immediately fix the issue. If a similarly critical issue was found in an API provided with a given platform, say iPhone OS, a hotfix would likely be pushed out straight away. It is also possible that new features and platform-specific features will be adopted slowly by the middle-ware (as the idea of cross-platform middle-ware is to create a consistent API that can be deployed to multiple platforms).

Middle-ware such as Airplay also commonly have a much smaller user base than the individual platforms they work on. This can mean that the middle-ware is less tested for stability than a platform-specific API would be. Further, it can lead to there being fewer places to turn to if support is needed.

On the other hand, there are many clear benefits to using an SDK such as Airplay. The model and animation loading and playing used in the Kodo prototype was found to be immensely simplified by using Airplay, and IwGx provided an easy means of creating applications that will run on multiple devices with varying levels of hardware support. The Kodo prototype was deployed to five platforms: a second generation, GLES 1.X iPhone, a second-generation iPod Touch (also with GLES 1.X), a third-generation iPod Touch (with GLES 2.0) and a Nexus One, an Android 2.1 phone with GLES 2.0. The application could be deployed with relative ease to all of these platforms, although we did make a few minor changes depending on the platform. For example, when deploying to the iPhone-like platforms, we used PVRTC, which is a texture compression technique that is not available on the Nexus One. The advantage of using texture compression techniques such as PVRTC is that memory use can be kept lower and textures can be loaded and processed faster.
4 The Kodo Prototype

In order to describe the Kodo prototype that we developed for this project, we first define what the contents of a typical scene might be. A scene can have the following objects:

- Kodos – small, round creatures with a large mouth and large eyes.
- Enemy creatures.
- Tiles – these are the “building blocks” of a level in a game of Kodo. These can be for example rocks, grass or volcanic stone.
- Buttons, boxes, spikes, collectible powerups or other objects.
- Water surrounding the tiles.

Kodos, as well as enemy creatures, are animated models. The tiles are static models, rendered in much larger quantities. Objects can be either animated or static depending on the object type. The water is represented by a single large surface with an animated water texture. This texture is repeated across the surface. Naturally, one could also imagine waves with more advanced geometry. Most of the assets included in a level are also lit using the Phong lighting model (see chapter 5.1).

The Kodo prototype also has a number of game modes. The versions considered in this project are a single player puzzle mode on one device, a two-player cooperative puzzle mode on one device, and a multiplayer competitive mode played on multiple devices. This latter mode is the focus of Nyström's work [17]. For this mode, one device acts as the server and main display, showing the position of each Kodo. All connected devices (theoretically, any number of players is supported) act as remote controls for Kodos on the main device. Any powerups collected by a player's Kodo is transferred to the remote device's screen, thus hiding them from other players. These can be activated to give any number of effects beneficial to the player. For example, a Kodo could turn into stone, making inedible, freeze other Kodos, or cause the Kodo to dig down into the ground. In this last case, the display on the corresponding remote device would show a special version of the scene where the underground Kodo is visualized, effectively hiding the Kodo from all players but the user.

A screenshot of our Kodo prototype, played in two-player cooperative mode, is shown in figure 4.1. Figure 4.2 shows the interface of a remote device in the competitive mode; note that the player has collected two powerups: one freeze powerup and one stone powerup. The number 0 (zero) at the top of the screen indicates that the player has not yet scored any points.
4.1 Performance Evaluation

In order to be able to evaluate the performance of the Kodo prototype, all performance tests were based on the rendering speed of a given scene in the prototype. Performance impacts were measured by comparing frame rates to a default graphics settings version of the Kodo prototype, which has a target frame rate of 20 frames per second on a second-generation iPod Touch, and 30 frames per second on a third-generation iPod Touch.
4.2 Performance Considerations

Culling
Airplay SDK automatically performs frustum culling on rendered models. It does this by holding a bounding sphere for each model, and testing whether any part of the sphere falls within the view frustum. Because the tiles in the Kodo prototype are relatively long and thin, a bounding sphere does not approximate them particularly well. This leads to more objects than necessary being drawn outside the left and right edges of the screen. For this reason, we apply stricter manual culling in the x-direction before asking Airplay to render models.

Backface culling is made by default by OpenGL. It can be disabled (since backfaces can be useful, for example in cel shading [7, pp. 508-510; as toon shading], and may also be required if one is to watch models through a mirror), but doing so is not necessary in the Kodo prototype.

Atlasing
In order to reduce the number of materials used in a scene in the Kodo prototype, tile materials are merged into atlases. Each atlas is a large texture that contains the textures for several tiles (say, a few rock tiles and a few grass tiles). Models have individual texture coordinates that define what parts of an atlas is used. In this manner, tiles can use the same material, which helps reduce GLES state changes. Atlasing was implemented for the terrain tiles in the Kodo prototype, as the majority of models and textures in a scene belong to terrain tiles. Each atlas is filled with as many textures as possible within the constraints of a given resolution, and when it has run out of space, a new atlas is created. Figure 4.3 shows part of an atlas used in the Kodo prototype, containing textures for nine terrain tiles and several objects (boxes, crystals and plants).

Texture Size
For the Kodo prototype, an example scene was rendered and run with four different atlas sizes for the tiles: 256*256, 512*512, 1024*1024 and 2048*2048. Each atlas resolution was tried on both a second-generation and a third-generation iPod Touch. On both devices, the frame rate was found to be largely independent of the resolution used for the atlases, which means that the frame rate bottleneck is elsewhere in the application. Using the largest atlas resolution, there was a frame rate drop of less than 5 frames per second. None of the other resolutions led to a decrease in frame rate, which makes the second largest resolution (1024*1024) optimal for quality versus performance.
Polygon Size
Many mobile graphics processors, including the GPU on iPhones, use a technique called tiling [14] to save memory. Essentially, this means that the screen surface is divided into tiles (note that the meaning of tiles here differs from that used in the context of the Kodo prototype). These are rendered and copied to the screen buffer separately from each other. In theory, the developer should not need to be concerned with the details of this, but in practice, this may not be the case. Each tile holds a list of all the polygons intersecting that tile, and all the polygons in a tile's list have to be processed to render that tile. This means that large polygons (and polygons intersecting a tile edge) may be processed multiple times. In turn, this can potentially lead to low-polygon models being more expensive than high-polygon ones, as the smaller polygons of a high-polygon model may be processed fewer times.

To make sure polygon size was not a limiting factor in the Kodo prototype, the models used for the terrain were subdivided to decrease polygon size (thus increasing polygon count). A scene consisting of only rock terrain was rendered with four versions of the rock model with varying polygon counts. See table 4.1 for the results of these tests on a third-generation iPod Touch. Note that the number of polygons in a scene scales a little differently from the number of polygons in a model, since some models in a scene will get partially culled. It was found that decreasing polygon size and increasing polygon count negatively affected frame rate, thus suggesting that tiling is not a bottleneck in our Kodo prototype.
Table 4.1. Effects of varying polygon count on frames per second in a scene from the Kodo prototype run on a third-generation iPod Touch.

<table>
<thead>
<tr>
<th></th>
<th>Low detail</th>
<th>Medium detail</th>
<th>High detail</th>
<th>Very high detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. polygons per model</td>
<td>77</td>
<td>119</td>
<td>221</td>
<td>1480</td>
</tr>
<tr>
<td>No. polygons, entire scene</td>
<td>6500</td>
<td>9000</td>
<td>15000</td>
<td>95000</td>
</tr>
<tr>
<td>Frames per second without shaders</td>
<td>35</td>
<td>30</td>
<td>29</td>
<td>9</td>
</tr>
<tr>
<td>Frames per second with shaders</td>
<td>30</td>
<td>25</td>
<td>23</td>
<td>14</td>
</tr>
</tbody>
</table>

Geometry Batching
A major bottleneck in GLES is the number of draw calls made each frame. The number of draw calls can be reduced by batching geometry. This means that lists of polygons (often triangles) representing several models are merged together into a longer list containing all of these models. If polygon strips are used instead of polygon lists, one may need to add degenerate polygons between each model. A polygon is degenerate if its vertices are aligned so that the polygon will not be visible when rendered.

The majority of the draw calls in the Kodo prototype are made to render terrain tiles. As very few other draw calls are made, batching terrain tiles should be enough to ensure that the number of draw calls is not a performance bottleneck. For the Kodo prototype, we have made two separate batching implementations: real-time scene batching and cached scene batching. Without batching, the maximum number of draw calls per frame in the Kodo application is approximately 130; ideally, Airplay applications should make no more than 70 draw calls per frame [20].

Our real-time batching works by calculating which models are inside the view frustum, and then batching these together. This decreases the number of draw calls to less than five per frame. However, batching geometry in this way incurs a fair amount of additional CPU usage each frame. When the real-time batching method was tested on the Kodo prototype and compared to the non-batched version, there was no increase in frame rate. One possible explanation for this is that gain from minimizing the number of draw calls may have been matched by the cost of additional processing, thus not giving a net increase in frame rate.

In our cached batching implementation, terrain tiles with the same material are grouped into 2 by 2 blocks and batched together when any part of the block is inside the view frustum. Each of these blocks are saved until they fall outside the camera. This leads to significantly less extra CPU cost compared to the real-time method, but also gives a smaller decrease in the number of draw calls. Given that all four tiles of a block have to be rendered if any part of the block is inside the view frustum, the number of models that need to be rendered each frame should be slightly higher. Since only one draw call
needs to be made per four rendered models, the theoretical number of draw calls should be slightly worse than a fourth of the original number. When cached batching was tested on a third-generation iPod Touch, the maximum number of draw calls per frame was approximately 45, which is 35% of the number of draw calls in the non-batched version. The performance gain achieved by using this method varied with the scene rendered, but never exceeded five frames per second.

Content Optimization
The Kodo creature, as well as the tiles, were turned into low-polygon models in order to reduce geometry complexity for our Kodo prototype. The Kodo model was reduced from approximately 800 triangles to around 360 triangles with very little impact on the perceived visuals (see figure 4.4). Any negative effects on the perceived appearance are reduced further by the size at which the Kodos are rendered on mobile devices.

Figure 4.4. High-polygon Kodo model (left) and low-polygon Kodo model (right).
5 Mobile Shader Techniques

Shaders on mobile devices were introduced with the OpenGL ES 2.0 specification, and are only present on modern high-end devices. In the Kodo prototype, there is a fallback renderer to handle devices running GLES 1.X. However, the focus of this work is to explore graphically advanced applications by modern standards. Thus, this chapter only discusses the prototype as rendered by GLES 2.0, and in particular, the shader techniques used to enhance the look of the prototype.

Some of the shaders listed in this chapter – the shaders for the Kodo and tile materials, as well as the shader used to render the water and the powerup shaders – share common code to calculate the effect of fog. This emulates the fog in the GLES 1.X pipeline.

5.1 Lighting Shaders

Most of the materials in the Kodo prototype – including those attached to the world tiles and those attached to the creatures (both Kodos and enemy creatures) – are lit according to the phong lighting model [15, 23]. To describe the model, we need a number of definitions. Each material has the following properties:

- The specular reflection constant $k_s$
- The diffuse reflection constant $k_d$
- The ambient reflection constant $k_a$
- The shininess constant $\alpha$

Let the set of all lights be $L$, the surface normal $N$, the reflection of a ray of light over the normal $R$, and the direction to the viewer $V$. For each light, let $i_s$ and $i_d$ be the intensity of the specular and diffuse components respectively. $i_a$ is defined to be the intensity of the ambient lighting.

The resulting light intensity $I_p$ is then calculated as follows:

$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (L_m \cdot N) i_d + k_s (R_m \cdot V)^\alpha i_s)$$

This lighting is calculated in the shaders for those materials that should be lit. Two approaches were tested – vertex lighting and fragment lighting. In the former, lighting is calculated for the vertices and then interpolated between the fragments, which yields slightly worse visual results – particularly for the specular lighting, as it relies on a power function. In the latter, the light intensity is calculated at each fragment. This gives higher visual quality at the cost of performance. On devices that do not support GLES 2.0, there is a fallback that uses standard GLES 1.X lighting.

To evaluate the performance of the lighting shaders, both methods (vertex lighting and fragment lighting) were run on the same scene from the Kodo prototype on a third generation iPod touch. A version using the non-shaded fallback was run on a second
Fragment lighting was found to have a significant performance cost, with a framerate drop of between 7 and 10 frames per second in the Kodo prototype. Despite being more complex than the non-shaded lighting, the shader using vertex lighting was running approximately 10 frames per second faster than the unshaded version. This is explained partly by the use of shaders, and partly by the third generation iPod's faster CPU. The result of vertex shader lighting, with texturing enabled and disabled, on a scene from the Kodo prototype, is shown in figure 5.1. Both of the screenshots are from a third generation iPod touch.

Figure 5.1. Vertex shader lighting on a scene from the Kodo prototype with textures disabled (top) and enabled (bottom) on a third-generation iPod touch.
To further enhance the appearance of light in the Kodo prototype, *blob shadows* are used for the Kodo creatures. These are simple circular shadow textures blended onto the tiles currently below each Kodo. As such, these are not reliant on shaders.

### 5.2 Normal Mapping

Normal mapping is a technique that is commonly used to add surface detail to models without increasing the number of polygons [7, pp. 187-190]. A *normal map* is a texture containing normal information; most commonly, the x, y and z components of a normal are baked into the r, g and b layers of a texture. In fragment operations, these normals can be used instead of the interpolated normals from the vertex shader, thus giving more precise information at the relatively small cost of one additional texture.

Normal mapping was implemented and combined with Phong lighting in the Kodo prototype. However, normal mapping requires lighting calculations to be performed on the fragment shader level. As fragment lighting is somewhat expensive, normal mapping is not suitable throughout all tiles and creatures. Nevertheless, it might be a useful technique for specific objects and creatures.

### 5.3 Water Shader

The water in the Kodo prototype is rendered using a simple water texture and a shader that modifies the appearance of the water. In addition, a cloud texture is used to modify the appearance of the water. The water and cloud textures used for the shader are shown in figure 5.2.

![Figure 5.2. A water texture (left) and cloud texture (right) from the Kodo prototype.](image)

Two floating point numbers are calculated outside the water shader – the results of a sine function $f(x)$ and a strictly increasing function $g(y)$. In the case of the Kodo prototype, $g(y)$ is a linearly increasing function. These numbers are passed into the shader as uniform parameters; that is, parameters that do not vary over individual vertices or fragments, though they can vary over time. The sine function has two purposes: to move the surface of the water up and down (by adjusting the positions of vertices in the vertex shader) and to offset the water texture along the x axis. The visual
effect resembles that of gentle waves; a slow-moving shift back and forth on the surface of the water.

The cloud texture is offset by $g(y)$, the value of the increasing function, along the x axis, and by a quarter of this value along the y axis. Moving the clouds similarly to the wave motions helps support visual coherency. By shifting the axis of motion subtly, repetition in the surface of the water is made less apparent.

The cloud textures and water textures are combined by first setting the water alpha value to a constant value and then subtracting, at each fragment, the cloud texture from the water texture. In this manner, areas covered by clouds (white on the cloud texture) will be darker on the surface of the water.

To avoid costly lighting calculations on the surface of the water, certain assumptions are made. Clouded areas will have less reflections, thus making them easier to see through. This is accounted for by increasing the transparency of fragments by a fraction of the cloud texture intensity at each fragment.

After any other water calculations, the brightest areas of the water surface – any where the blue component $b$ of the fragment is larger than a threshold $t$ – are made brighter. Let the set of color components of a water fragment be $C$ and the saturation constant be $k$. For each component $i$, the following formula is applied:

$$C_i = b + (b - t)k$$

Saturating bright areas of the water is an inexpensive way of giving the water a reflective appearance without the use of lighting calculations.

5.4 Powerup Effect Shaders

In order to visualize the effects of a Kodo using a powerup, a custom shader was implemented for each powerup. There are two powerups in the prototype: one that briefly freezes your opponents, and one that turns the user into stone, rendering him invulnerable for a few seconds.

The shader used to visualize frozen Kodos has a greatly increased specular component of the lighting, to symbolize a more reflective material. In addition, the color of the Kodo is blended with a bright blue color. Together, these two effects give the Kodo the appearance of being encased in ice.
The stone form, on the other hand, has a decreased specular component, giving the surface a matte look. In addition, the Kodo texture is desaturated. Some noise is then added to the stone surface by using a modulo operation on the Kodo texture's \( r \) component for noise generation, thus making sure the noise does not vary over time without requiring an additional noise texture. The stone effect is shown in figure 5.3.

5.5 Post-Processing

Post-rendering screenspace effects – shaders that alter the final rendering of the image – are commonly referred to as post-processing [7, pp. 467-473]. Common effects include blurring [7, pp. 468-473], color correction [7, pp. 474-475] and tone mapping [7, pp. 475-480]. Two of these, gaussian blur and color correction, were implemented in the Kodo prototype.

To implement these methods, a framebuffer object (FBO) was used to render the scene to a texture. This texture is then rendered as a screen-sized polygon. A fragment shader can then affect the individual pixels in the rendered scene, thus allowing for post-processing effects. Although FBOs in OpenGL are relatively fast, post-processing still carries some overhead as an additional rendering pass is required. Thus, careful consideration is needed when using post-processing methods on mobile devices. In the Kodo prototype, both the blur and color correction effects were discarded as the performance impact of adding a post-processing rendering pass – approximately 10 frames per second – was considered too large in comparison with the visual benefits. However, this cost is considerably smaller than the cost of post-processing without shaders would be, and post-processing effects thus constitute an exciting new possibility for mobile applications, particularly ones where scene rendering is inexpensive.
5.6 Dynamic Lighting

Dynamic lighting is supported by the Phong lighting model, as the model takes into account all lights in a given scene, and the lighting calculations can be performed in a shader in real-time. In the Kodo prototype, we let the Kodos act as light sources to give the world a more dynamic feel. However, calculating the result of multiple light sources is relatively expensive. On a third generation iPod Touch, using multiple lights (one static light and two moving lights) generated a performance hit of around 10 frames per second, which was deemed too costly to be included in a released game. For less complex scenes, however, dynamic lighting remains a viable option that can greatly enhance the visual experience of an application.

5.7 Evaluation

The shader tests, performed on a third-generation iPod Touch, show that the idea that as much work as possible should be offloaded from fragment shaders to vertex shaders is particularly true for mobile devices. Because of the more limited graphics processor, running expensive calculations only once per vertex and then interpolating these results over each polygon often yields a significant performance gain.

It is also worth noting that some seemingly simple shader operations can be deceptively expensive. For example, using a discard statement on the tile materials (as a way of allowing transparency in tile textures) incurred a significant cost – 10 frames per second in the Kodo prototype – even if it is on the side of an if-statement that never gets called. Imagination Technologies, who make the PowerVR graphics hardware present on iPhones, explicitly recommend against using discard statements unless it is absolutely necessary [16]. One should carefully test cross-platform applications on all targeted devices, as pitfalls like the one with the discard keyword often behave differently depending on the drivers and hardware present on each specific device.
6 Discussion and Conclusions

We have found that it is quite possible to make graphically advanced applications using cross-platform middle-ware. However, the middle-ware does impose limitations on what can be done on each given platform; a feature, say bluetooth, may be available on some platforms and not on others. In such cases, the lowest common denominator – in this case, no bluetooth – is often used. On the other hand, the middle-ware can also act as an enabler – the software engine provided with Airplay SDK lets developers make 3D games for devices that do not support OpenGL without having to create a 3D engine from scratch.

Shaders on mobile devices were found to work surprisingly well, supporting a number of techniques previously not viable on mobile devices. Although mobile shaders do not perform as well as their desktop counterparts, offloading CPU work to dedicated graphics hardware can yield significant benefits. Full screen post-processing effects, in particular, are now possible in real-time. Other modern graphics techniques such as normal mapping and fragment lighting can also be successfully used on mobile devices; particularly if they are not used throughout entire scenes, but rather on a subset of models and textures.

Performance tests on computers, whether intended for desktop use or mobile use, is a complex subject. In some sense, the most objective measurement for performance is the number of clock cycles used, though this can be difficult to measure; particularly for techniques that span across both CPU and GPU. More commonly, the time passed from the beginning to end of a particular algorithm is measured instead. This does not mean, however, that per-algorithm time measurement is necessarily the best choice. Interference such as operating system interrupt calls can impact the results, particularly if the time spans measured are very small. Individual algorithms performed many times per second in real-time games are examples of algorithms with such small time spans. Further, as many techniques are combined in a scene, the impact of each technique may change, as algorithm complexity is dependent on the size of the input data. It is also difficult for humans to grasp the effect of per-frame millisecond (or nanosecond) costs, or the number of clock cycles used by a given operation.

As our goal has been to evaluate the feasibility of advanced graphics applications on mobile devices, we chose to use the frame rate metric. The reasoning behind this is simple: low frame rate is immediately perceived by the user of an application. If the frame rate is low, the program will be perceived as performing poorly. On the other hand, if the frame rate is sufficiently high, the user will not notice individual frames.
7 Outlook

There is plenty of room for improvement both in the area of cross-platform middle-ware and in the area of mobile graphics technology. While some of these improvements will no doubt come from hardware vendors (in the form of more advanced graphics hardware) and platform vendors (in the form of more standardized mobile platforms), many interesting software technologies have also yet to become common. Some of these will be discussed briefly in this section.

7.1 Cross-Platform Middle-Ware

Although Airplay SDK and Unity are both exceptionally useful tools for cross-platform development, this is clearly an area where many improvements can still be made. There is an ever-increasing spectrum of platforms to target, as new operating systems and devices are continually released. However, reaching out to more platforms has to be weighed against the benefits of more streamlined toolkits that have been specifically designed to work well for one or a few systems. There is no definite answer for where to draw the line; rather, different middle-ware have different niches. For example, Unity has a very streamlined interface but deploys only to a few platforms, while Airplay SDK is low-level but can deploy to a wide range of platforms. We are looking forward to seeing other options that support graphics-heavy applications, perhaps placing themselves in between these two extremes.

7.2 Graphics

Mobile graphics is another interesting area for further research; in particular, we would like to explore techniques such as parallax mapping [7, pp. 191-193], relief mapping [7, pp. 193-198], full-scene anti-aliasing [7, pp. 126-128], cel shading [7, pp. 508-510; as toon shading] and real-time shadows.

Parallax mapping, while giving better visual results than normal mapping, is also considerably more expensive. Since normal mapping was considered too expensive for the Kodo prototype, parallax mapping was not implemented. For more light-weight applications, however, parallax mapping could prove useful even on mobile devices. Relief mapping is a related technique, which can give even more visually appealing results at higher rendering cost.

Full-scene anti-aliasing (FSAA) is supported by the hardware in many PowerVR chips [25]. While phones such as the iPhone 3GS use such graphics chips, the functionality is not always present; for example, Apple does not expose FSAA functionality on iPhones. Manual supersampling is still possible, but relatively expensive to do as scenes need to be rendered in higher resolution and subsequently downsampled. Though it can be costly, enabling anti-aliasing can give significant visual benefits, which makes it an interesting area for further exploration.

Cel shading (or toon shading) can be used as a way to get distinct and pleasing graphics
even on low-end hardware. Since cel shaded applications often use few textures, as well as low polygon models, the technique is well suited to hardware with shaders that have limited memory or polygon processing power.

Real-time shadows can make many 3d applications look considerably better; by adding shadows, depth is more easily perceived by the observer. In addition, they can help increase the contrast of a scene. Although blob shadows – such as the ones in the Kodo prototype – can achieve this to some extent, shadows shaped by objects in the scene appear more realistic.
8 References


2. Olsson, E. “Mobile Phone 3D Games with OpenGL ES 2.0”. Royal Institute of Technology, School of Computer Science and Communication. 2008.


8. “OpenGL ES Programming Guide for iPhone OS: Performance Guidelines”. iPhone OS Reference Library. Available at: 


13. “3ds Max – 3D Modeling, Animation and Rendering Software – Autodesk”. Available at: 


15. “Phong shading – Wikipedia, the free encyclopedia”. Available at: 

16. “POWERVR SGX – OpenGL ES 2.0 Application Development Recommendations”. Imagination Technologies. Available at: 


18. “Potentially visible set – Wikipedia, the free encyclopedia”. Available at: 

19. “Majority of smart phones now have touch screens (Canalys press release: r2010021)”. Available at: 

20. Personal communication with Ideaworks Labs, authors of the Airplay SDK.

21. “File:Example of bounding volume hierarchy.JPG – Wikipedia, the free encyclopedia”. Available at: 

22. “Ragdoll physics – Wikipedia, the free encyclopedia”. Available at: 


