Inter-Device Multiplayer and Performance Optimization in Games for Modern Mobile Devices

MARTIN NYSTRÖM

Master of Science Thesis
Stockholm, Sweden 2010
Inter-Device Multiplayer and Performance Optimization in Games for Modern Mobile Devices

MARTIN NYSTRÖM

Master’s Thesis in Computer Science (30 ECTS credits) at the School of Engineering Physics Royal Institute of Technology year 2010
Supervisor at CSC was Henrik Eriksson
Examiner was Lars Kjell Dahl
TRITA-CSC-E 2010:066
ISRN-KTH/CSC/E-10/066--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.kth.se/csc
Abstract

Inter-Device Multiplayer and Performance Optimization in Games for Modern Mobile Devices

In this thesis we explore the use of a large-screen mobile device, the iPad, in multiplayer games across multiple modern mobile devices. Specifically, the Wi-Fi capability on these devices is used to implement and evaluate multiplayer where the iPad acts as a game surface while smaller devices such as the iPhone and iPod touch are used as wireless controllers. We also evaluate and optimize performance in advanced 3D games on the iPad, iPhone and iPod touch. A game prototype, Kodo, was developed so that tests could be made in a real-world scenario. The end result is a successful implementation of the prototype with inter-device multiplayer and good 3D performance, which shows that current mobile devices can run advanced 3D games with highly responsive networked multiplayer. It was also found that in order to achieve this, special consideration of hardware restrictions and limitations is required.

Sammanfattning

Flerspelarläge över flera enheter och prestandaoptimering av spel för moderna mobiltelefoner

I denna rapport undersöks användning av en mobil apparat med stor skärm – i detta fall Apples iPad – för ett flerspelarläge där flera mobila enheter samverkar. Trådlösa nätverk (Wi-Fi) utnyttjas för att implementera och utvärdera ett flerspelarläge där iPad agerar spelplan. Mindre apparater, såsom iPhone och iPod touch, används som trådlösa handkontroller. Dessutom undersöks prestandan hos och optimering av avancerade 3D-spel på iPad, iPhone och iPod. För att kunna göra tester i ett verkligt scenario utvecklades en spelprototyp, Kodo. Resultatet är en lyckad implementation av prototypen med flerspelarläge och goda 3D-prestanda, vilket visar att dagens mobila apparater kan köra avancerade 3D-spel med responsivt flerspelarläge över trådlösa nätverk. Det visade sig emellertid att speciell hänsyn till begränsningar i hårdvaran måste tas för att åstadkomma detta.
### Table of Contents

1. Introduction .................................................................................................................. 1
   1.1 Problem Statement ................................................................................................. 1
   1.2 Delimitations ........................................................................................................... 2
2. Background ..................................................................................................................... 3
   2.1 Use of the Kodo Prototype ...................................................................................... 4
   2.2 3D Engine Elements ............................................................................................... 5
   2.3 Graphics Libraries ................................................................................................. 7
   2.4 OpenGL ES Versions ............................................................................................... 8
   2.5 Performance Considerations ................................................................................... 8
      2.5.1 Rendering Optimizations .................................................................................. 9
      2.5.2 Content Creation ............................................................................................ 13
   2.6 Middleware ............................................................................................................. 14
3. The Prototype .................................................................................................................. 15
   3.1 Performance Evaluation ......................................................................................... 16
   3.2 Performance Considerations ................................................................................... 16
4. Multiplayer ....................................................................................................................... 23
   4.1 Technology .............................................................................................................. 23
      4.1.1 Wi-Fi ............................................................................................................... 23
      4.1.2 Bluetooth ...................................................................................................... 23
      4.1.3 Cellular Network ........................................................................................... 24
   4.3 Compensation Techniques ...................................................................................... 25
      4.3.1 Aspects of Compensation .............................................................................. 26
      4.3.2 Protocol Optimization .................................................................................... 26
      4.3.3 Dead Reckoning ............................................................................................. 27
      4.3.4 Local Perception Filters ................................................................................. 28
      4.3.5 Synchronized Simulation .............................................................................. 29
      4.3.6 Area-of-Interest Filtering ............................................................................... 29
   4.4 Implementation ........................................................................................................ 29
      4.4.1 Sockets ............................................................................................................ 30
      4.4.2 Bluetooth ...................................................................................................... 31
      4.4.3 Implemented features ................................................................................... 31
   4.5 Evaluation ................................................................................................................ 33
5. Conclusions and Future Work ....................................................................................... 35

Bibliography .................................................................................................................... 37
1. Introduction

Software and games for mobile devices is growing into a huge market. While mobile phones have been around for a long time and almost everybody has one, it is only now that a majority of the more advanced phones come equipped with the hardware required for more complex games. Possibly even more important than hardware is that there are now ways for developers to sell and distribute their applications and games through online stores directly accessible from the devices. When consumers today buy high end phones they expect plenty of great games available for purchase, and with the amount of competition out there developers need to make their game stand out of the crowd. This can be achieved in many ways such as making good use of shaders to create impressive visual effects or by bringing something unique to the gameplay. Most games for mobile devices such as the iPhone/iPod touch do not incorporate multiplayer modes, and when they do it is often simple and restricted to playing on the same device.

Newer devices often come with Wi-Fi and/or Bluetooth connectivity along with the cellular 3G network for phones, which gives many opportunities for interconnectivity between devices. Developing multiplayer games is no trivial task however, as games have very small tolerance for lag, insecurity, and synchronization failures.

This thesis aims to evaluate the options for inter-device multiplayer on current mobile devices, specifically iPhone, iPod touch and the iPad. The goal is to provide a playable 3D game with a fun multiplayer mode and impressive visuals that runs smoothly on current hardware. This thesis will focus on multiplayer while my co-worker Måns evaluates shaders and middleware.

1.1 Problem Statement

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

What are the specific considerations relating to networking and 3D performance that apply when developing multiplayer games for mobile devices?

This thesis is specifically focused on the development and evaluation of networked multiplayer involving multiple mobile devices, and does not cover other areas such as sound or input. Since games in general, and even more so games for mobile devices, are performance-critical applications, this thesis will also cover 3D performance optimization for games on mobile devices.
CHAPTER 1. Introduction

1.2 Delimitations

Parts of the work was done in collaboration between students Martin Nyström and Måns Olson. As such, most of the content in chapters 2 and 3 are common to both reports.

The remainder of the report will be individual and specific to the area of investigation: Måns focuses on using cross-platform middleware to support advanced mobile graphics, while Martin focuses on multiplayer interaction on high-end mobile devices.
2. Background

In previous thesis work, Gustavsson [1] and Olsson [2] explored the use of OpenGL ES (GLES), primarily using an emulator running on a PC. In particular, they studied the latest version of the library, GLES 2.0 and 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 theses, Gustavsson focused on the construction of the game engine itself, while Olsson focused on 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 [4] or Airplay SDK [5]. Middleware is evaluated in Måns’ report “OpenGL ES 2.0 and shaders for mobile phones using cross-platform middle-ware” [9].

While the results presented in these previous theses are theoretically sound, it is not clear that one can directly extrapolate conclusions that hold true for today's mobile market. The tests were made exclusively on emulator software, which makes it difficult to apply any performance results on GLES 2.0 hardware. Today, hardware supporting GLES 2.0 has become available on high-end mobile devices. It is our intention to continue the work of Gustavsson and Olsson by performing tests on such hardware, including iPhone 3G, iPhone 3GS, iPad and 3rd generation iPod touch.

In order to be able to work collaboratively, a prototype game was developed by Gustavsson and Olsson. The game concept, Kodo, is an existing IP (intellectual property) 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”. The book covers shaders, the OpenGL ES Shading Language, as well as the GLES 2.0 pipeline.
2.1 Use of the Kodo Prototype

It is 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 will not use shared program code; the old prototype was written to work on specific hardware, while the new one will function on multiple platforms. In this manner, many assets needed for 3D performance tests – in particular, models, animations, and textures – can be reused, which should allow for rapid prototype development, while also providing shared grounding between the two sets of thesis work. This allows for some degree of comparison between the emulator tests of previous theses and the device tests that will be performed for this thesis.

Throughout this text, we will be returning to our version of the Kodo prototype, which we will use to illustrate both implementation details and findings about performance. We will 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 will rely heavily on advanced...
graphics techniques, we expect it will help us identify the relevant performance bottlenecks (and also boundaries) on today's GLES 2.0 devices. In addition, the core gameplay mechanics in Kodo is very well suited to multiplayer conditions, which is a requirement for the work described in this thesis.

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 the number of polygons and textures is unknown, the frame rate needs to be optimized 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). 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 [8, 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 thesis will 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 a 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, 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.
CHAPTER 2. Background

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 [6] 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 rotations of another bone (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 shows a skeleton used for bone animation rendered on top of a low-polygon Kodo model.

![Figure 2: A skeleton (used for animation) rendered on top of a low-polygon Kodo model.](image)
CHAPTER 2. Background

Animation
One of the most common methods for animating 3D models is *bone animation*. This method can be performed 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, the position of every bone that has moved needs to be saved in every keyframe. In other words, if an arm is moved, one needs to explicitly state that the bones of the hand have moved as well.

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 using ragdoll physics – an application will need to support 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 are drawn, for example by altering their positions or normals. Fragment shaders modify fragments – which roughly correspond to pixels – as they are 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.
CHAPTER 2. Background

There are two common such 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 the desktop versions of OpenGL; most of features and syntax are identical, although a few features have been removed 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 [7]. 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) [7]. 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 handle 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 3D 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, we
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 [8,
pp. 664-667]. Such a frustum has the geometrical shape of a square pyramid delimited by
a near and far viewing plane, as shown in Figure 3.
To speed up object culling, a pre-computed bounding volume [8, pp. 725-726] is 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 [8, pp. 726, 729]) and boxes aligned to the object (Oriented Bounding Box, or OBB [8, 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 [8, pp. 19, 771-778].

Figure 3: Scene with no culling (top) and view frustum culling (bottom).
CHAPTER 2. Background

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 4, where the wireframe objects will get 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.

![Figure 4: Occlusion culling. The wireframe objects are culled because they are completely hidden by the cylinder closest to the viewer.](image)

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 [12] divides a scene into regions and precomputes visibility between them. This allows for quick indexing to 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 [8, 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 [8, 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 5. 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.

![Figure 5: A bounding volume hierarchy showing the bounding volumes (left) and the resulting tree (right).](image)

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”) [8, pp. 163-166] is a related technique applied to textures instead of models. 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 both to the hardware and 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 [10].

If the 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 for sorting to yield good results, 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 some variables.
CHAPTER 2. Background

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 game 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 [11], 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.

2.6 Middleware

Multi-platform development is difficult because each platform targeted by an application has different properties. Cross-platform middleware are toolkits that can be thought of as platforms in their own right – instead of developing a unique version for each targeted platform, a developer can make a version that uses the middleware as a platform and then deploy it to all platforms supported by the middleware.

The Kodo prototype used in this thesis work was developed using Airplay SDK [4] which supports 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. Airplay SDK was designed with graphics-heavy games in mind and was developed by Ideaworks Labs [14] – an offshoot of the game developer Ideaworks 3D.

Airplay SDK supports a number of mobile operating systems – iPhone OS, Android, Symbian, Windows Mobile, Brew and Maemo. Model and animation loading and rendering is built into the API (Application Programming Interface), using a custom text format for models and animations. The SDK also includes tools to export to this format from commonly used modeling programs. Airplay SDK also contains a software 3D engine and a custom graphics library, IwGx. The latter mimics many of the functions found in OpenGL ES, and can be run either on the software engine provided by Ideaworks, on GLES 1.X, or on GLES 2.0.
3. The Prototype

To understand the Kodo prototype we have in mind, we will 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 lava.
- Buttons, boxes, spikes, collectible powerups and other objects.
- Water surrounding the tiles.

Kodos, as well as enemy creatures, will need to be animated models. The tiles will be static models, rendered in much larger quantities. Objects could be either animated or static depending on the object type. The water could be represented, for example, by a single large surface with a water texture. Naturally, one could also imagine waves with more advanced geometry and texture animation. All the assets included for a Kodo level also need to be lit using some appropriate lighting model (such as Phong shading; see chapter 5.1).

The Kodo prototype has a number of game modes. The versions considered in this thesis work 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. 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 screen of the remote device, 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 it harder to eat, freeze other Kodos, or cause the Kodo to dig down into the ground. In this last case, the display on the 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 6.
3.1 Performance Evaluation

In order to enable evaluation of the performance of the Kodo prototype, all performance tests will be based on the rendering speed of a given scene in the prototype. Performance impacts will be 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.

3.2 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, a stricter manual culling is applied in the x-direction before asking Airplay to render models.

Backface culling is made by default in OpenGL. It can be disabled (since backfaces can be useful, for example in cel shading [8, 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.
CHAPTER 3. The 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 7 shows part of an atlas used in the Kodo prototype, containing textures for nine terrain tiles and several objects (boxes, crystals and plants).

![Figure 7: Part of a texture atlas from the Kodo prototype.](image)

Texture Size
For the Kodo prototype, an example scene was rendered, using 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** [17] 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
CHAPTER 3. The Prototype

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 3.1 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 implying that tiling is not a bottleneck in the Kodo prototype.

\textit{Table 3.1. Varying polygon count on a scene in the Kodo prototype on a third-generation iPod Touch.}

<table>
<thead>
<tr>
<th>No. polygons per model</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, entire scene</td>
<td>77</td>
<td>119</td>
<td>221</td>
<td>1480</td>
</tr>
<tr>
<td>Frames per second without shaders</td>
<td>6500</td>
<td>9000</td>
<td>15000</td>
<td>95000</td>
</tr>
<tr>
<td>Frames per second with shaders</td>
<td>35</td>
<td>30</td>
<td>29</td>
<td>9</td>
</tr>
</tbody>
</table>

Geometry Batching

A major bottleneck in GLES is the number of draw calls made for 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 [23].

18
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 and the tiles were turned into low-polygon models in order to reduce geometry complexity for the 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 8). Any negative effects on the perceived appearance are reduced further by the size at which the Kodos are rendered on mobile devices.

![Figure 8: High-polygon Kodo model (left) and low-polygon Kodo model (right).](image)
iPad Specific Considerations
The Kodo prototype was also adapted to run in native resolution on the iPad. It was found that the iPad suffered from performance issues due to the increased resolution; so called **fill rate** issues. By replacing the water texture rendered across the screen with a black background sprinkled with stars, the frame rate was significantly improved (from about 10 frames per second to 25 frames per second). Two screenshots of the space version of Kodo running on an iPad are shown in Figure 9 and Figure 10.

![Image of Kodo prototype running on iPad](image_url)

**Figure 9:** The Kodo prototype running on an iPad. Note that the water rendering has been replaced by a space background in order to increase performance.
CHAPTER 3. The Prototype

Figure 10: The Kodo prototype running on an iPad. The purple Kodo is about to eat the blue Kodo.
4. Multiplayer

Games have a very small tolerance for lag, insecurity, and synchronization failures. Because of this, one has to design the network code to fit the game's specific requirements.

In this chapter we will discuss the technology making inter-device multiplayer possible, basic considerations that have to be made depending on type of game, common methods and algorithms to deal with problems faced when making networked multiplayer games, describe the implementation in the Kodo prototype and discuss the results gathered from making this implementation. Another big issue for multiplayer games in general is cheating and how to prevent it, but that is beyond the scope of this thesis.

4.1 Technology

There are numerous ways for electronic devices to communicate with each other today, each with different strengths and weaknesses. High-end mobile phones and other devices often have access to three such technologies: Wi-Fi networks, Bluetooth and Cellular networks (mostly phones).

4.1.1 Wi-Fi

Wi-Fi, which stands for wireless fidelity, refers to any system that uses the 802.11 standard that was developed by the Institute of Electrical and Electronics Engineers (IEEE) [28]. Wi-Fi technology uses radio for communication, typically operating in the 2.4 GHz band. Electronics that are "Wi-Fi Certified" are guaranteed to interoperate with each other regardless of brand.

Wi-Fi networks are commonly available in people’s homes, workplaces and at public locations such as railroad stations or restaurants. Often these networks provide a connection to the internet and as such one could potentially use them to play games online in addition to being able to play on the local network.

Wi-Fi networks have limited range. A typical wireless router using 802.11b or 802.11g with a stock antenna might have a range of approximately 30 meters indoors and 100 meters outdoors. Common network speeds varies from 11Mbit/s (802.11b) up to 300Mbit/s (802.11n).

4.1.2 Bluetooth

Bluetooth is a proprietary open wireless technology standard for exchanging data over short distances using short length radio waves from fixed and mobile devices, creating personal area networks (PANs) with high levels of security.
Bluetooth uses a radio technology called *frequency-hopping spread spectrum* [19], which divides the data being sent into smaller fragments and transmits it on up to 79 bands of 1 MHz width in the range 2402-2480 MHz. This is the same radio frequency band which Wi-Fi uses and this can lead to interference.

Bluetooth versions 2.0 and 2.1, which are used on most Bluetooth capable devices on the market today, can achieve data rates of up to 3Mbit/s while version 3.0 can achieve rates of up to 24Mbit/s. The range varies greatly depending on the device and its purpose, but for mobile phones it is generally around 10 meters. This number is greatly influenced by physical obstacles between the communicating devices.

There can be up to eight Bluetooth devices talking to each other in what is called a *piconet*. Among these devices, there can be only one master device, all the rest are slave devices. A device can belong to two piconets simultaneously, serving as slaves in both piconets or a master in one and slave in another. This is called a bridging device. Bridging devices connect piconets together to form a *scatternet*, see Figure 11 (c).

![Figure 11: Single-slave piconet (a), multiple-slave piconet and scatternet (c) [19]]

### 4.1.3 Cellular Network

A cellular network is a radio network made up of a number of cells, each served by at least one fixed-location transceiver known as a cell site or base station. When joined together these cells provide radio coverage over a wide geographic area. This enables a large number of portable transceivers (mobile phones, pagers, etc) to communicate with each other and with fixed transceivers and telephones anywhere in the network (via base stations) even if some of the transceivers are moving through more than one cell during transmission.

In a cellular system, as the distributed mobile transceivers move from cell to cell during an ongoing continuous communication, switching from one cell frequency to a different cell frequency is done electronically without interruption and without a base station operator or manual switching. This process is called the handover or handoff. Typically, a
new channel is automatically selected for the mobile unit on the new base station which will serve it. The mobile unit then automatically switches from the current channel to the new channel and communication continues.

The range of a cellular network is determined by coverage in the area of the device using it, but only areas with very low population density remains without coverage in industrialized countries. One drawback of having such a huge range is limited bandwidth, ranging from 56-114kbit/s (GPRS / 2.5G) [22] up to 14Mbit/s downstream and 5.8Mbit/s upstream (3G) [20]. Future cellular networks (4G) [21] will offer even higher bandwidth.

### 4.3 Compensation Techniques

Networked applications face three resource limitations: bandwidth, latency and the nodes processing power for handling the network traffic. Because of this, one has to find ways to utilize the available resources effectively. The amount of resources required in a networked application is directly related to how much information has to be sent and received by each participating node and how quickly it has to be delivered by the network. [3]

In this section a few different compensation methods which try to reduce resource requirements are presented. The basic idea is to replace communication with computation: Network traffic can be reduced at the cost of processing power. Before reviewing compensation methods, we will briefly look at what aspects affect the choice of method.
CHAPTER 4. Multiplayer

4.3.1 Aspects of Compensation

The two biggest concerns when creating a networked game are how to achieve a good balance between consistency and responsiveness, and how to be able to scale well as the amount of players increases.

Consistency refers to the similarity of states on the different nodes. Absolute consistency means that each node has uniform information, but can only be guaranteed if all nodes wait for everyone to receive the information updates before proceeding.

Responsiveness refers to the delay that happens before an update event is registered by the nodes. In order to achieve high responsiveness everything has to proceed before everyone has received an information update which can lead to inconsistencies between the nodes. This means that one cannot have both high consistency and high responsiveness at the same time and the choice of which techniques to use in an application is essentially a trade-off between these two attributes.

Scalability is the ability to adapt to change in available resources or resources needed, which for a networked game translates to how well it handles an increased amount of players or changed capacity limits in the connected network. Scalability is affected by the choice of deployment architectures such as peer-to-peer or client-server and how these are implemented. This thesis does not aim to evaluate scalability beyond the scope of being able to support a handful of players in a single game.

Different types of games put different emphasis on these aspects. A massively multiplayer online game where thousands of player share the same game world will have to scale incredibly well, a strategy game needs to have very high consistency, while a first person shooter will probably favor responsiveness over the other two.

4.3.2 Protocol Optimization

Every message sent on the network uses network resources and processing power, and as such usage of available resources can be improved by reducing either the size of each message or the number of messages sent.

The networked application can save bandwidth at the cost of computational power by compressing messages. Compression reduces the number of bits needed to represent some information and as such provides an intuitive approach to minimize network traffic. More advanced ways of using compression in networking involves referencing previously transmitted data in order to reduce redundancy in messages; however, this requires a reliable transmission protocol (e.g. TCP).

In order to reduce the number of sent messages message aggregation can be used. Message aggregation reduces transmission frequency by merging information from multiple messages and sending it as one. Doing this will in all cases negatively affect responsiveness as the method either waits a fixed amount of time to collect messages before sending them, or waits for a fixed amount of data or number of messages to build up before sending the data.
4.3.3 Dead Reckoning

Dead reckoning [3, pp. 191-195] is the process of estimating current position based upon a previously determined position and advancing that position based upon known or estimated speeds and course over time. In networked games, it is used to reduce the number of update messages needed by sending them less frequently and estimating state information between the updates. In addition to extrapolating the current state directly from past states, the state update can include additional rate of change information (such as acceleration) useful when predicting how state will change in the future. When the next update message arrives, the predicted value can differ from the actual value, which can cause disruptive visual effects or even errors in assumed follow-up events based on the predicted value unless handled properly. To preserve visual coherence when a state is corrected the difference can be corrected by converging closer to the true value over time.

Prediction of state between state updates is of course dependent on what type of entities the state updates applies to. If the entity is a moving object in a game world its movement is commonly predicted using first or second order derivate polynomials meaning velocity and possibly acceleration. These are either sent together with the update messages or calculated using earlier positional updates, and the values are used to help predict that object’s movement until next update message is received.

When a node (meaning an instance of the game, running in the network) using a dead reckoning technique receives an update message, the predicted state of an entity is likely to differ from the state contained in the update. When this happens, the entity has to be updated to this new state using a convergence technique. The simplest technique is zero-order convergence (or snap) [3, pp. 194], where the entity is changed immediately to a new state without any sort of interpolation in between. This method can cause jerky movement or impossible changes such as an object appearing to move through walls.

A good convergence technique corrects errors quickly while looking as natural as possible. To do that a convergence period within which the error is corrected must be selected. If the state represents a position of an entity, a convergence point along the prediction can be chosen and when the entity reaches the convergence point it continues along the new predicted path. The path in which the entity moves from its current position to the convergence point can be calculated either as just a straight line or with more sophisticated curve-fitting techniques. Situations can occur where dead reckoning causes positional problems that cannot be solved in other ways than letting visual disruptions occur, such as an entity’s predicted position ending up on the wrong side of a solid object in a game world as shown in Figure 13.
4.3.4 Local Perception Filters

Local perception filters (LPFs) [3, pp. 196-198] hide communication delays by exploiting the human perceptual limitations. Instead of predicting state, LPF allows temporal distortions in the view and entities are rendered at out-of-date locations based on how big the underlying communication delay is.

In order to enable the use of this technique, entities of a game world are divided into two classes:

- **Players** which are indeterministic entities, whose behavior cannot be predicted.
- **Passive entities** which are deterministic, and whose behavior follows some set of rules, such as the laws of physics.

LPF address the problem of delays by discerning the actual situation from the rendered (and perceived) situation. Local players whose actions and movements are delivered with no delay to the local game state are rendered using up-to-date state information while remote players with a delay of $x$ milliseconds will be rendered with the known $x$ milliseconds old information from the last received update. When interaction between two players takes place, for example when one player fires a projectile at another player, the players will perceive the event differently but with the same outcome. This is because of the temporal distortion of the projectile changing for both players so that when the projectile are closer to themselves, it is rendered closer to its current position for that player while the other player will see it where it actually was moments ago. This can be seen as a way to keep consistency between what the different players think the end result of an event should be: if temporal distortion was not used one player could see their projectile clearly hitting the other player while the other player perceived it as him or her dodging the projectile without problems.

One important limitation of LPFs is that the closer players are to each other, the more noticeable the temporal distortion becomes. Another weakness of LPFs is the assumption
that we know the communication delays between players and if these fluctuate too much
the passive entities will begin to bounce back and forth in time instead of having smooth
temporal transitions.

4.3.5 Synchronized Simulation

Synchronized simulation [3, pp. 205] works with a replicated architecture with absolute
consistency where all players have their own copy of a complete state that is identical
across all participants. This can be done because all events in a game world can be
divided into deterministic events which are generated by the simulation and
indeterministic ones which are commands issued by a player. Because each simulation
should be able to generate the same deterministic events (by using a common seed for
pseudo random numbers and such), players only need to send their commands to each
other and let the outcome of these commands be determined on all participants.

A real-world implementation would have to include consistency checks and recovery
mechanisms in case some input gets lost. Also this method is only applicable to games
where input delay will not be a huge problem, such as strategy games, since the order of
events needs to be the same on all nodes in order for the simulation to end up with the
same results, and therefore the local player cannot process its own input immediately but
will have to wait for confirmation from a server first.

4.3.6 Area-of-Interest Filtering

Often in multiplayer games, a node has no use for state updates for entities that does not
currently affect it, such as another player on the opposite side of a map. Therefore we can
save resources by only sending update packets to those who need them. This can be done
by dividing the game world into zones where nodes only receive updates for entities
within the zone they are in or some subset of zones related to it. There are many different
ways to create these zones or incorporate things such as where a player is heading into the
process of determining which entities are of interest to this player. [3]

4.4 Implementation

For this project, we explore ways to use Apple’s new product iPad in conjunction with
smaller devices such as the iPhone and iPod touch for multiplayer games. The concept is
to use the iPad as a common play area for all players in the game, and letting each player
use a smaller device to control something in the common play area. In addition, one could
use the smaller device to display some personal information – such as hidden cards,
powerups or something else the player wants to keep hidden from his or her competitors.

In this thesis, we focus on creating a playable and fun prototype of this concept. As such,
extensive evaluation of advanced algorithms and methods for connecting devices other
than Wi-Fi networks had to take a back seat.
4.4.1 Sockets

In this text, the term sockets refers to an API for the TCP/IP protocol stack, which is usually provided by the operating system. In this case, the s3eSocket API provided in Airplay SDK is used. Sockets constitute a mechanism for delivering incoming data packets to the appropriate application process or thread, based on a combination of local and remote IP addresses and port numbers. For most purposes a socket will send and receive on one of two transport protocols: TCP (Transmission Control Protocol) [25] or UDP (User Datagram Protocol) [26].

Multiplayer in the Kodo prototype was implemented using TCP sockets, but UDP was also tested when measuring response times. The network part is implemented in three static classes using a server/client architecture; one class for common code, and the other two for server and client specific code respectively. The game can then choose to initialize whichever class it needs and then have access to a set of functions for finding servers or clients, getting received messages, sending messages to the joined server or one or all clients, joining or leaving a server and starting the game. Any class including these classes also has access to some additional information, such as response times.

When initiated, the classes will keep track of servers on the network and connected clients, measure response times, and will automatically flag or remove servers and clients as necessary if they time out.

A user will typically use the initiated client class like this: First get a list of active servers on the network by calling findServers(), then join one by calling joinServer() with an index referring to a server in the serverlist. After joining a server the user should call waitForGameToStart() which will return true if the game has started, and after this the user can get all the update messages from the server by calling getMessages().

A user who wants to act server will call functions in this manner: First let clients find and join the server by repeatedly calling findClients(), and then start the game by calling startGame(). After this the server user has access to a list of joined clients and can use sendMessageToClientID() or sendMessageToAllClients() to send update information, and getMessagesForClient() with an index corresponding to a client to get messages from that client.

One limitation when using Airplay SDK is the inability to create threads. The implementation of the classes described above could have been made cleaner with the use of threads, for example by letting the user register callback functions that are fired whenever a new message is received instead of having to repeatedly call getMessages() or getMessagesForClient().

During initial tests it was found that devices responded to messages over Wi-Fi a lot slower than a PC (the PC would respond to messages within 20-40 ms while the devices took 300-400 ms, and often higher than that). Troubleshooting this difference in response time proved difficult. Several potential reasons for this issue were explored:

- Network hardware on the devices. Maybe it is just much slower than PC network hardware.
CHAPTER 4. Multiplayer

- Poor distribution of available CPU resources, which could mean that when the device is under load, it will not spend as much time getting and/or sending packets on network sockets. This was tested by turning off rendering and other updates to reduce CPU load, and by using an API call that yields to the device to let it perform any needed background work.

- Airplay. Either poor compatibility with Apples APIs or some other more general performance issue in their network libraries. This was never fully tested but could have been done by making a small test application with the iPhone SDK instead of Airplay.

- Network encryption. The devices could be poor at handling encryption, but this proved unlikely as problems persisted on unprotected networks.

- Nagle’s algorithm [24] causing holdups when sending data. However, using the UDP protocol yielded the same results and no change was seen when trying to send TCP packets immediately, ignoring Nagle’s algorithm.

- Power saving functions on devices.

It was noticed that measuring response time to a device that had just joined the Wi-Fi network resulted in a very quick response for the first few seconds. The idea was formed that network hardware on the device would enter a low power state when network traffic was low, and therefore take extra time when leaving this state, or simply being slower at responding while in this state. This was tested by repeatedly sending small packets filled with junk data in intervals of ~200 ms. This method proved very effective at reducing response times.

The final measured round trip times from a client when using sockets over TCP during a game with two players ranged from 17 ms up to 96 ms with an average of 34 ms. Using UDP, results were slightly better ranging from 15 to 95 ms with an average of 32 ms.

4.4.2 Bluetooth

The intention was to implement Bluetooth functionality into the Kodo prototype. Unfortunately, it was found that the implementation of Apple’s *iPhone Game Kit* API [27] in Airplay SDK was not functional. This API is needed to connect devices via Bluetooth, and therefore time was spent on the Wi-Fi implementation instead. At the end of the period of time in which this thesis work was done a new version of Airplay SDK was released that fixed this issue; however, there was not enough time left to do a full implementation.

Instead, some simple tests of response times were run in order to get an idea about how Bluetooth compared to Wi-Fi when it comes to gaming. Results were that the response time between two connected devices was ranging from 21 to 117 milliseconds with an average of 47 milliseconds.

4.4.3 Implemented features

The main goal was to implement a multiplayer mode where players would all look at the same screen (the iPad) while controlling their character with a remote. This makes for some fun competitive play where each player can have some hidden information on their
remote that they can use to their advantage. In this implementation the hidden information consist of powerups that can be gathered from white crystal-looking objects in the game world. When a player picks up a crystal they will get a random powerup from a set of predefined powerups.

There are two different types of multiplayer implemented, which can be played in networked or local multiplayer: Competitive and Cooperative. The goal of competitive multiplayer is to eat each other. When eaten, the victim will have to wait for some time before respawning, and the player eating will gain a point. In cooperative mode, players help each other solving puzzles and eating enemies.

When playing networked multiplayer, players will use an iPad as the game surface and control their characters with an iPhone or iPod touch. In local multiplayer all players share the same device and control their character by touching their designated area of the screen to jump forward.

The networked multiplayer is implemented with a *Synchronized Simulation* technique described in section 4.3.5. The complete duplicated gamestate needed to realize things such as networked multiplayer not using a shared surface was not implemented, since it was not needed for the concept described at the beginning of this chapter. Therefore the only parts that need to be synchronized between server and client acting as a remote control are score, powerups picked up and powerups being used.

The remote made for iPhone and iPad is shown in Figure 14 and consists of three buttons and some other UI elements. The triangular button to the right is pressed to make the character move, and two buttons to the left each activates a powerup. The buttons to the left change appearance depending on the powerups held, and if touched, triggers the appropriate powerup. Near the center of the screen is the score display, showing how many enemy kodos the player has eaten. At the top of the remote, on both the left and
right hand side, are two bars that display the remaining duration of powerups used by, or against, the player.

Two different powerups were implemented for the prototype: a freeze powerup and a stone powerup. The former will freeze all your opponents for three seconds, during which time they cannot move. Stone will turn the kodo that activates it into stone, and all attempts to eat it will cause the attacker to die from horrible toothache. The freeze powerup is demonstrated in Figure 15.

![Image of freeze powerup in action](image)

*Figure 15: The purple player (left) has used the freeze powerup and the red player (right) cannot move*

**4.5 Evaluation**

The multiplayer components of the game were added iteratively. To begin with, a single player version was developed as a test platform and proof of concept. When this was mature enough, a cooperative multiplayer mode was added, followed by the network multiplayer mode.

Adding the ability to eat other kodos, introducing the inherent delay of networks and involving the effects of powerups meant that assumptions, originally made for the single and cooperative multiplayer modes, about who could move where and when no longer held true. Special rules had to be implemented that increased complexity drastically. One such rule is preventing a player from moving at the last few hundred milliseconds when being eaten, and had to be implemented because it would otherwise be too easy (if you can jump while being eaten) or too hard (if you were considered dead as soon as another player tried to move into the tile you were currently in) to dodge another player trying to eat you. These rules came with follow-up rules when powerups came into play, such as removing the aforementioned movement disable even if a player was about to be eaten – if the player trying to eat was frozen, or if the player being eaten used a stone powerup and became immortal, the lock would have to be removed again. In short, adding
CHAPTER 4. Multiplayer

multiplayer is not done by simply implementing the ability for two clients to communicate, but also adds complexity to the game engine itself.

Some particularly interesting mobile Wi-Fi hardware specific issues were discovered, as described in section 4.4. Extensive testing was necessary in order to deduce the source of the problem. The issue was resolved by continuously transmitting data, thus keeping the Wi-Fi hardware from entering what is assumed to be a power saving mode.

Networked multiplayer over Wi-Fi in the Kodo prototype has been successfully tested with up to 5 players on different devices, with an iPad running as server and game surface. A video of the game session [29] was recorded and can be seen through YouTube.

The tests done using Bluetooth indicate that response times are on par with those of Wi-Fi networks. Range was not an issue; connection between two devices was sustained for distances of up to 15 meters in an office environment. This indicates that a Bluetooth implementation could work just as well as its Wi-Fi counterpart. Of course, choosing Bluetooth over Wi-Fi could create other problems such as increasing the amount of background work needed to be done by the operating system, bandwidth issues or poor connection stability. Note, however, that none of these issues were present in the response time tests.
5. Conclusions and Future Work

Developing a multiplayer game for mobile devices is in many ways similar to developing for PC or other platforms with network capabilities. When using Wi-Fi one can write and use the same network code for several target platforms, changing only how the code interacts with the Sockets API available on that specific platform.

The mobile devices tested perform just as good as a modern PC with regards to networking for games if data is sent continuously in order to prevent the network hardware from slowing down. Future work could include testing outside of the Kodo prototype in order to establish how well the devices handle very high network traffic when limitations such as network hardware speed, computational power or memory read/write performance could turn out to be a bottleneck.

Bluetooth was not implemented into the game, but the tests that were run indicate that it could work just as well for multiplayer games as Wi-Fi does. Bluetooth alleviates the need to have an accessible Wi-Fi network in close vicinity to where you want to play, but instead further limits the range between players. When using a shared game surface range is not an issue since a natural range limit is imposed by having to actually see the screen on which you are playing. As such, Bluetooth could be an even better alternative than Wi-Fi for this type of game – if the technology and device hardware are as stable and can handle traffic as well as the Wi-Fi counterpart. Future work would involve using Bluetooth in a multiplayer game, and even an online mode using cellular networks, to explore the usability of these technologies and compare them to a Wi-Fi implementation.

Using the iPad as a game surface with smaller devices acting as remote controllers turned out to be both feasible from a technical perspective, and also resulted in a fun and intense gaming experience. Using a shared game surface helps create a feeling that everyone is playing together, as opposed to feeling like you are playing alone when everyone is looking only at their own device. Future work should include implementing synchronization of more variables than done in this prototype, in order to be able to use things such as the burrow powerup briefly described.

The game prototype runs at acceptable framerates and can render a number of polygons exceeding recommended budgets by quite a margin – probably thanks to the use of some optimization techniques described in this thesis. We have as such shown that current mobile devices are capable of delivering advanced 3D graphics, but not without spending time on optimization of both code and assets such as 3D models and textures. One notable find is that the biggest bottleneck on the iPad seems to be related to fill rate performance. This can probably be explained by the fact that its graphics hardware is similar to that of the iPhone 3GS and 3rd generation iPods, but the screen resolution is much larger (over 5 times as many pixels). We get the framerate back up to acceptable levels despite this issue. This is done by simply clearing the screen to black between frames, and then not rendering things onto a large portion of the screen except for some
small white triangles resembling stars. This creates a “space look” that is fully acceptable in our case. Other techniques on how to alleviate this problem should be tested in future work, for example rendering onto a lower resolution surface and scaling up to fit the screen.
Bibliography


[22] ”EDGE (Enhanced Data Rates for GSM Evolution)” Available at: http://en.kioskea.net/contents/telephonie-mobile/edge.php3 (fetched May 21, 2010)

[23] Personal communication with Ideaworks Labs, authors of the Airplay SDK.


[29] “YouTube – Kodo iPad Multiplayer” Available at: http://www.youtube.com/watch?v=mOBGj8zzvVs (fetched May 26, 2010)
