iOS Game Programming

Ville Forsman

Thesis

HAAGA-HELIA
ammattikorkeakoulu

Tiivistelmä

10.2.2013

Tietojenkäsittelyn koulutusohjelma

Tekijä tai tekijät Ville Forsman	**Ryhmätunnus tai aloitusvuosi** HETI09
Raportin nimi iOS peliohjelmointi	**Sivu- ja liitesivumäärä** 42 + 9
Opettajat tai ohjaajat Sirpa Marttila	

Työ keskittyy pelien ohjelmointiin Applen mobiililaitteille. Esiteltyjä asioita on käytetty työn ohella tehdyssä peliprojektissa.

Tavoitteena oli tutkia peliohjelmoinnin keskeisiä periaatteita sekä toteuttaa nämä käytännössä. Laitealustana on toiminut sekä iPhone että iPad ja täten myös iOS-ohjelmointiin on perehdytty.

Ohjelmointiympäristönä on toiminut Applen Xcode ja siihen liittyvät oheisohjelmistot. Pelin sisällön tuottamiseen on käytetty ilmaisia ja kaupallisia sovelluksia.
Näistä keskeisimpiä käydään läpi erityisesti pelien kannalta.

Asiasanat
Cocoa , Objective-C 2.0, Cocoa Touch, OpenGL ES, iPhone, iPad,
Game Programming

HAAGA-HELIA
University of Applied Sciences

Abstract

10. February 2013

Degree Programme in Information Technology

Authors Ville Forsman	**Group or year of entry** HETI09
The title of thesis iOS game programming	**Number of pages and appendices** 42 + 9
Supervisor(s) Sirpa Marttila	

The thesis focuses on programming games for Apples mobile devices. The subjects covered were used in a related game development project.

The goal was to study the general principles of game programming and applying these in practice. The devices used for games were the iPhone and the iPad which created the necessity for studying iOS-programming as well.

The programming environment consisted of Apples Xcode and related applications. The game's content was produced with both free and commercial applications. The most essential ones of these were also covered from the viewpoint of the games.

Key words
Cocoa , Objective-C 2.0, Cocoa Touch, OpenGL ES, iPhone, iPad,
Game Programming

Contents

1 Introduction

Apples mobile devices have become very popular and so has the software development for these devices. The AppStore is a great way for companies and especially individuals to publish and/or sell their programs. Apple provides a very good framework and development tools for creating programs. However, it is a very large and confusing world at first and especially when developing games. Game development usually means applying many different technologies in the product.

The iOS platform has some specific requirements concerning application development. These are covered extensively by Apple.
Game programming also has its own twists regardless of the platform. Combining these two creates the concept of iOS game programming.

Games are pretty much like magic tricks: what you see is very different from what is actually going on. The purpose of games is generally to create an illusion and to fool the player. Also, like in magic, the players are willing to be fooled. The example project is named Moggy and it's a simple 2-d shooter game that I've been developing.

The purpose of this thesis is to play a supporting role for anyone interested in developing games for the iPhone or iPad. Unfortunately it is impossible to cover the whole subject extensively but hopefully the general ideas will become clear.
The focus will be on how to build the game world in a programming environment, how the game actually works and how to present the game world.

The programming environment and technologies are developed and updated constantly but the general guidelines probably will remain. Many of the sources were difficult to place since a lot of the research material contained the same information and many things can be considered general knowledge since they have to be done exactly as described.
The documents structure is fairly modular since the concepts are also about individual entities working together. Understanding of objective-oriented programming is expected from the reader.

2 Developing for iOS

Games use many types of technologies to create the finest possible illusion. In addition to normal programming games typically implement graphics and sound. They might also handle user input in creative ways. Many games also update constantly by using a game loop.

The entities used are mostly done with general programming principles. The objects are placeholders for data and contain methods for modifying this data and the state of the object.

Every visible object needs to use coordinates associated with the game world so that it can be drawn in the correct position and also for detecting collisions with other objects. The object must also hold information on what the system needs to draw when presenting the object and these are the coordinates for the texture atlas which holds all the graphics for the game.

The graphics are drawn by sending the data in interleaved vertex arrays to the graphics processing unit and the drawing commands are given with Open Graphics Language.

The game constantly updates itself making the game "alive". This is the game loop where all the objects states are updated, the collisions are detected and appropriate measures are taken depending on what is going on.

For example if a bullet hits an enemy the sound manager is ordered to play the corresponding sound at the collisions coordinates, the enemy's health is reduced by the amount the bullet does damage, the bullet is removed from the game objects and the particle emitter is told to draw the corresponding particles in the spot where the collision took place. This is a very simplified example.

2.1 Game Programming

Games are a fairly special extension to general programming. Everything happens inside the game world. This consists of the entities in the game, such as the environment, enemies and the player. These can be achieved from the programming perspective through objects.
In the real world it is possible to define ones coordinates globally or in another environment. The same principle is used for the objects in the game.
The world also needs to react to user input or otherwise it would be fairly boring. Depending on the type of the game, this logic can vary a whole lot.
The game also constantly checks what is going on and updates the world accordingly. Enemies die, bullets are fired and so on. All of this affects the game world which is basically just data.
The data then needs to be displayed to the user and this is done with graphics.

2.2 Cocoa-programming

The general programming for Apples devices is called Cocoa-programming. When programming for the iOS devices (such as the iPhone, iPod and iPad) the Cocoa Touch –libraries are used.
The programming language in Cocoa programming is Objective-C 2.0, which is an extension for C. This means that a lot of the basic code is written as C. It is also possible to use C++ when programming. This makes transporting the code to other platforms easier but the platform specific things still need to be written in Objective-C, such as communicating with the operating system and using classes from Cocoa libraries.

The two most important programs in the development environment are Xcode and the Interface Builder. In the latest versions the Interface Builder is integrated in to the Xcode.
The Xcode is very similar to other development tools, such as Eclipse which is mainly used for Java-programming.

The Interface Builder is mostly used for creating and modifying the views and linking them to the code. The files created have been called nib-files or xib-files. These can be found in the Xcodes project management.

The components for the user interface can be created by coding, although it is best done using the Interface Builder in order to reduce your personal workload and to minimize the risk of bugs. Objects can also be created from the nib-files. These are created and linked during run-time when the nib-file is opened.
(Apple Inc. 2011. View Controller Programming Guide for iOS. 68)

Communication in Objective-C is done through messages.

Objective-C uses messaging in many different ways. The messages related to the retain count are one example of how messages are used (2.5). Methods are also called upon using messages. Objects might also send messages to each other.
(Clair, R. 2011. 93)

Protocols are used so that objects know how to communicate with each other. These are especially useful when using inheritance extensively, although it can be done in many different ways.
When a class is defined so that it implements the protocol it must respond to the messages defined in the protocol.
A protocol can contain optional and required messages. It is up to the programmer to decide how the objects should respond to the messages defined in the protocol.
(Clair, R. 2011. 249)

Delegation messages are also an important part of Cocoa programming. It allows objects to receive notifications about other objects automatically.
(Steinberg, D. 2010. 146)

One of the most useful things in the development environment is the Instruments application which can be used to find memory leaks, track memory and CPU usage and to optimize code for the GPU.

The environment is constantly developed and updated by Apple so it is advised to regularly check for new updates.

2.3 The Programming Language

Knowledge of Objective-C is a natural demand when writing the code for an iOS game. However, there is also a strong need for at least understanding basic C. Objective-C is an objective extension to C so when programming in Objective-C most of the code is in C.
The trickier parts are something like writing callback functions that might have to be in C. Usually things like this occur when working at the low level system operations. The OpenGL ES and OpenAL are also basic C APIs which basically means doing objective-oriented work with a non-objective language, which tends to make the code very confusing.
Working with C does have its advantages, such as increased performance and multiplatform compatibility.

In game programming some things are done in basic C (such as OpenGL ES), some things you can decide to do in either C or Objective-C (Audio) and the main part of the code can be done in C, Objective-C or C++.
In Moggy the code is written in Objective-C whenever possible and in C when required. This ensured ease of writing the code since there were no plans in multiplatform usage.

One important thing is variable types. When defining variables that are used by the drawing code it's wise to think ahead.
It would be best to define them as the types that OpenGL uses (GLfloat, GLbyte…).
This might require importing the OpenGL headers to the class in order to do so.
This way the variables don't need to be converted by the system and performance is increased.
If there are performance issues it might help to allocate these types of variables straight into video memory (which was not done in Moggy).

There are also some other useful variables defined in existing libraries, such as CGPoint, which is found in Core Graphics.

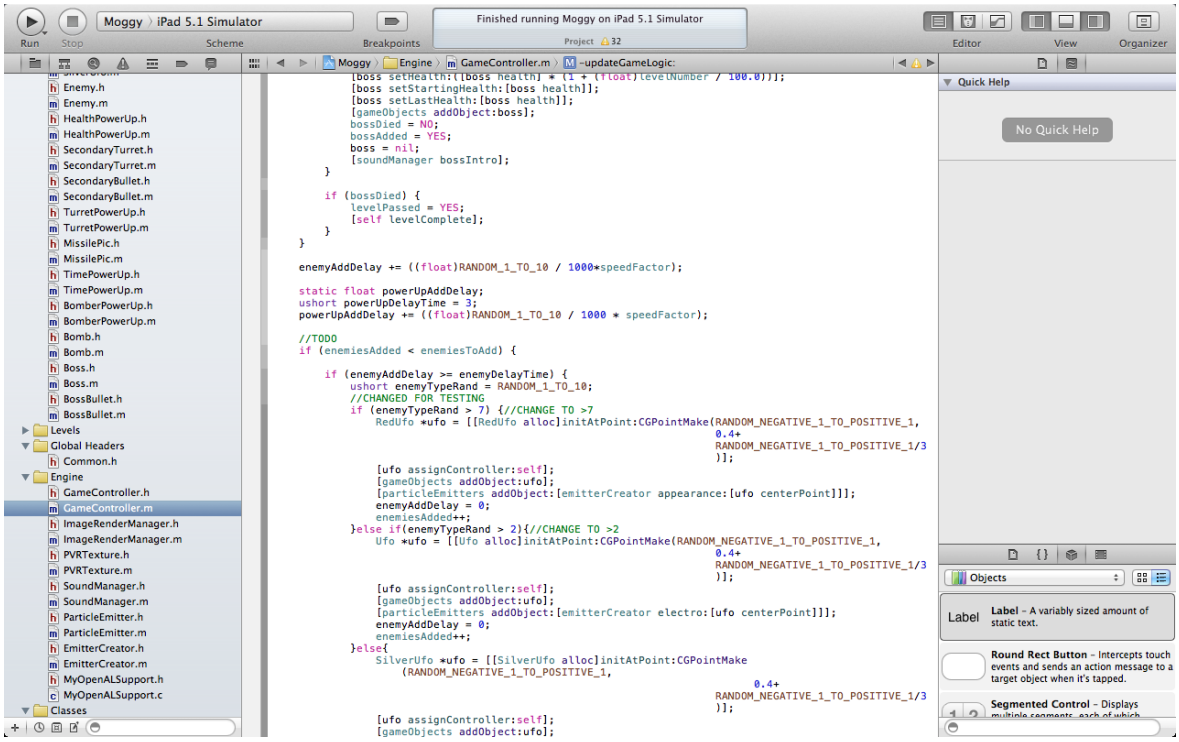


Figure 1. Xcode IDE

2.4 iOS -programming

iPhone programming has many special aspects. The device itself has a web-connection, a gps, motion detectors, camera(s) and a multi-touch screen. These generate new possibilities when designing applications.
The iPad runs the same operating system (iOS) but the device may have different capabilities. Also the libraries have some differences between the iPhone and iPad. If the application is designed for both devices then there must be separate resources for the iPhone and the iPad for the view-part of the mvc-principle.
The iPod also uses iOS so the application could also work on some iPod models.

Mobile devices do have their own problems as well. There are many different models for each device and they may or may not run different versions of the operating system.
There are also some aspects that require special attention such as memory usage, optimizing battery-life and running multiple applications simultaneously.
(Aachen University 2010.)

The iOS also uses the special versions of OpenGL, called OpenGL ES (chapter 2.8). The actual programming is fairly similar to Mac (Cocoa) programming, there are just more things to consider.

With iOS devices there is also the application delegate which receives messages from the application class. Each iOS program has an application class and it is not meant to be modified. Instead you implement the custom code in to the application delegate. This way you can respond to messages from the operation system during run-time. These include starting the application, ending it, memory warnings and so on.
This is one example of how delegation allows further developing a class without subclassing it.
(Goldstein, N. & Bove, T. 2010.)

2.5 Memory management

Memory management is one of the most important things in iOS programming. There is no automatic garbage collection as found on most computer programs. This will reduce the need of processing power from the hardware. Instead iOS uses a retain-count –method.
Whenever an object needs to hold on to another object it retains it. This causes the retain count to rise by one for the object that is being retained.
When the object is no longer needed by the other object it releases it which causes the retain count to go down by one.
Some automation can be done with the usage of the property- and synthesize –features, which are more or less glorified getters and setters.

When an objects retain count is zero then it is "dead". The run-environment periodically checks for the objects that have a retain count of zero and removes them from memory. When this happens the object will be deallocated and its dealloc-method is called. Here the object should release all the objects it has hold on to and finally call the superclasses dealloc method.
This principle is sometimes referred as a dog and its leashes are the retainers. When there are no leashes on the dog's collar it will be removed from memory.
(Clair, R. 2011. 268)

These issues require close attention when designing the architecture of the program. One of the most common downfalls is caused by objects retaining each other which causes the retain count to never reach zero.
There are many scenarios where the retain counts can cause a headache. Fortunately the program can be analyzed in the Xcode, which will show the memory issues that it finds but not necessarily all of them. Proper planning here can save a lot of time and effort.

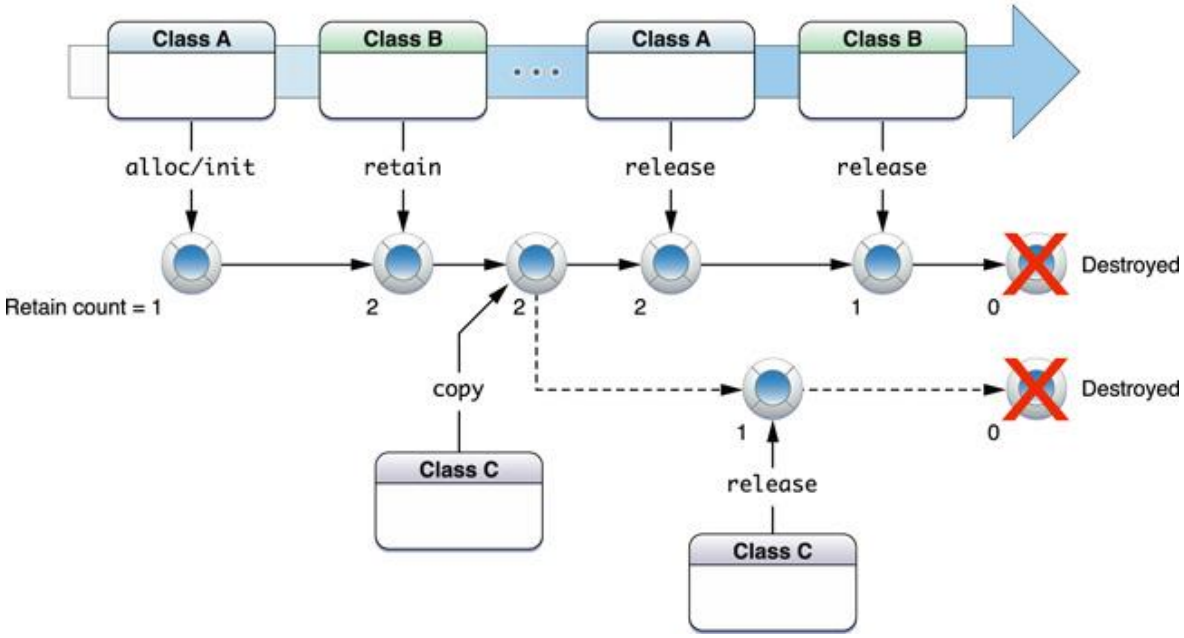


Figure 2. iOS Basic Structure (Apple Inc. 2012. Memory Management.)

2.6 Autorelease-pool

The autorelease pool is there to tackle another problem with memory management. Let's say there's object 1 and object 2 (Appendix 5.6). If you want to create object 3 from object 2 and pass it onto object 1 then the basic retain count won't work.
The object 2 has to return the object 3 at the end of its method. If it would release the object before returning it there is a chance that the object would be removed from memory before object 1 has a chance to retain it.
The autorelease is designed to send the release message some time later on either automatically or at the very least at the end of the program.
This way object 2 can pass the object 3 to object 1 and add a release message to the autorelease pool. The common syntax would be: `return [object3 autorelease];` (Clair, R. 2011. 270)

Many classes have so called convenience constructors, which add themselves to the autorelease pool. These can be found in the class documentations and are also simple to implement in to custom classes.
These methods usually contain the word "with" in their initializers, such as "initWithString". This is due to the Apples naming policy and it is advisable to follow these guidelines.

These constructors are in fact the class-objects methods (even classes are objects in Objective-C). (Clair, R. 2011. 143)

2.7 Views and Controllers

The visible part of the user interface consists of multiple components. Each application only has one window. This is like an invisible frame and you can change the contents with views. Another way to look at it is that it contains the general information and functionality which is always necessary and would therefore be a waste of resources to implement into every view.
The views are the essential parts of the UI and they are extremely flexible. They may contain a hierarchy (parent-child, siblings), although they appear like a single view for the user.

Views also have controllers. These are for controlling the input and output of the interface. There are many ways you can attach controllers and views to each other but it is usually best for a controller to control a logical set of vies.
The controller is also the first class to receive user input. If there is a hierarchical collection of views then the messages are sent up the parent-chain until a controller responds to the input or none respond and the message is discarded.
(Aachen University 2010.)

The contents of a view usually consist of pictures, buttons and such. They are easy to implement using the Interface Builder and it is up to the programmer to implement these for their applications.
(Apple Inc. 2011. iOS Application Programming Guide. 10)

Views can be presented also through the OpenGL ES (Open Graphics Language Embedded Systems) and in the example project all of these approaches were implemented. (Stanford University. 2010.)

There are controllers for basic functions already in the libraries. From these only two can be modified (subclassed) which are the UITableViewController and the UIView-Controller. The most common way is to use the UIViewController to build the user interface. This way the controller owns the views and controls the views components. (Apple Inc. 2011. View Controller Programming Guide for iOS. 27)

When a controller operates one logical view it is natural to have multiple controllers if there are multiple views. There are many ways to go about it but the basic principle is to ease the management of the views by handling the views and controllers as individual packages.
(Steinberg, D. 2010 217).

This also has a positive impact on memory management since the view associated with the controller is created only when needed and unnecessary views can be removed from memory.
(Stanford University. 2010.)

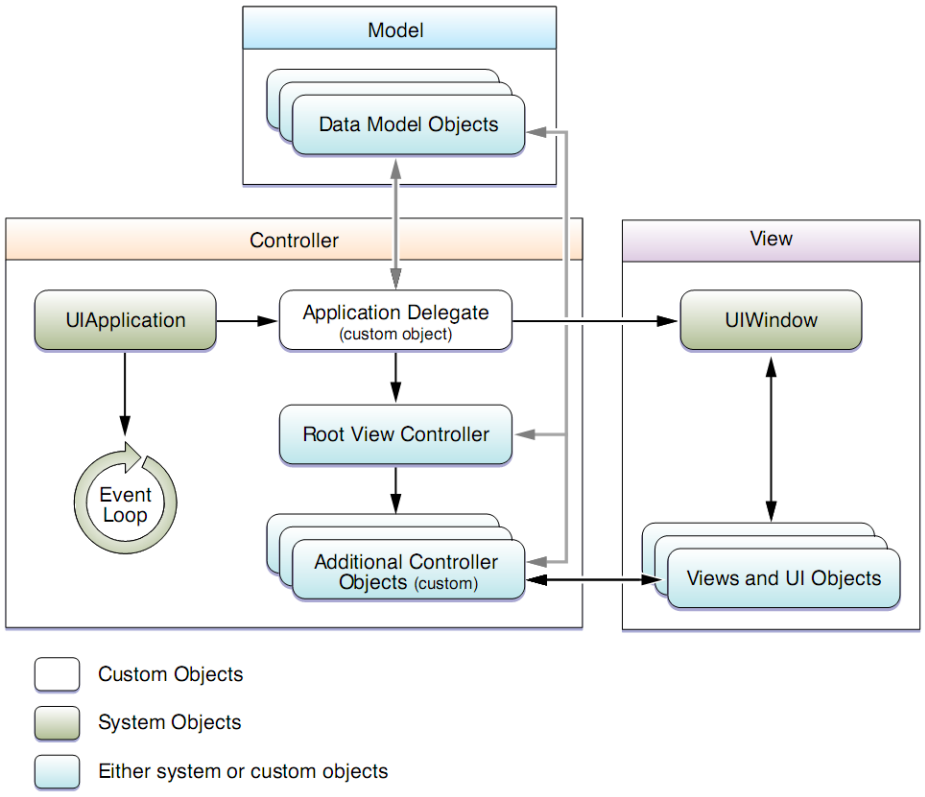


Figure 3. iOS Basic Structure (Apple Inc. 2011. iOS Application Programming Guide.)

2.8 OpenGL ES

The OpenGL ES (Appendix 5.4) is a version of OpenGL for mobile devices. There are two versions: 1.1 and 2.0. These are not just different versions but they are completely different from each other. The latest iOS devices support both versions but the older ones only support 1.1. There is also the iOS versions requirement which is 3.1 for 2.0.
The OpenGL ES is a C-based API to pass drawing commands for the GPU using a client-server –principle.

The 1.1 versions functions as a "fixed-function-pipeline"-principle. This means that the parameters are configured and then the draw commands are executed. These commands cannot be modified. This version is recommended for basic graphics and the coding requires slightly less work. This is the version used in the example project. The 2.0 –version uses shader programs and there is greater freedom to modify and control the drawing.

Using OpenGL ES also means that the program will be easier to transport to other manufacturers devices. If this is the long-term goal then the application should use minimal amount of system specific resources, such as libraries and Objective-C. Instead the coding should mainly be in C and C++.

The OpenGL ES is wrapped in a special class provided by Apple. The class is an EAGLView –class (EA = Embedded Apple) which provides the layer where the rendering is done along with the functionality of a view. The class by default also creates the corresponding renderer which handles the actual rendering on to the layer.
In the example project the rendering was implemented in a different class using interleaved vertex arrays and the renderer only handled setting up and displaying the buffer.

OpenGL ES 1.1 provides a standard fixed-function pipeline that provides good baseline behavior for a 3D application, from transforming and lighting vertices to blending fragments into the framebuffer.

If the programs needs are simple, OpenGL ES 1.1 requires less coding to add OpenGL ES support to your application. The application should target OpenGL ES 1.1 if it needs to support all iOS devices.
(OpenGL ES Programming Guide)

Although each object type in OpenGL ES has its own functions to manipulate it, all objects share a similar programming model:

1. Generate an object identifier.

An identifier is a plain integer used to identify a specific object instance. Whenever a new object is needed, call OpenGL ES to create a new identifier. Creating the object identifier does not actually allocate an object, it simply allocates a reference to it.

2. Bind the object to the OpenGL ES context.

Most OpenGL ES functions act implicitly on an object, rather than requiring to explicitly identify the object in every function call. The object is set to be configured by binding it to the context. Each object type uses different functions to bind it to the context. The first time an object identifier is bound, OpenGL ES allocates and initializes the object.

3. Modify the state of the object.

The application makes one or more function calls to configure the object. For example, after binding a texture object, you typically would configure how the texture is filtered and then load image data into the texture object.

Changing an object can potentially be expensive, as it may require new data to be sent to the graphics hardware. Where reasonable, create and configure objects once, and avoid changing them afterwards for the duration of the application.

4. Use the object for rendering.

After creating and configuring all the objects needed to render a scene, bind the objects needed by the pipeline and execute one or more drawing functions. OpenGL ES uses the data stored in the objects to render the primitives. The results are sent to the bound framebuffer object.

5. Delete the object.

When done with an object, the application should delete it. When an object is deleted, its contents are destroyed and the object identifier is recycled.
(OpenGL ES Programming Guide)

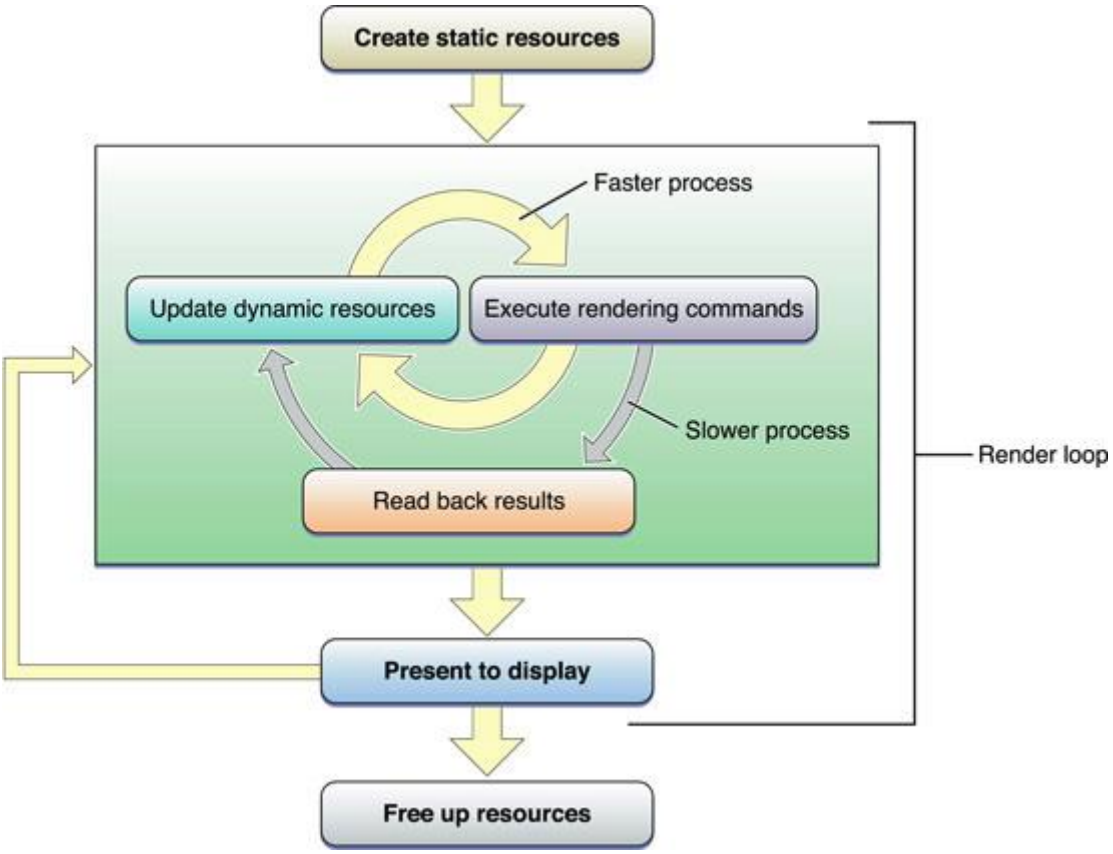


Figure 4. Application model for managing resources (OpenGL ES Programming Guide.)

2.9 Interleaved Vertex Arrays

The interleaved vertex array is a fairly confusing concept at first but very simple and effective after understanding it. The name itself is a bit misleading since the drawing is actually done with two arrays. The first array is the actual iva (interleaved vertex array) which is an array of structures. These structures hold information of the images vertex positions, texture positions and colors.
To be specific the iva is an array of structures of structures of structures. This is illustrated below.
All the attributes are defined for every vector so a quad will have four vectors containing these attributes. In Moggy the header Common.h contains the structures required for drawing.
To chop it down to smaller pieces:
Each vertex has a geometry coordinate (x, y). This is used to define the position on the screen (or "world").
Each vertex has a texture coordinate (u, v. Sometimes referred as s, t). This is used to define the point in the texture atlas for applying an image for the complete rectangle (or quad).
Each vertex has a color attribute (r, g, b, a). The colors in Moggy consist of red, green, blue and alpha. For practical reasons every vertex in the quad have the same color values. Daley covers the subject extensively in his book.

So there are the basics for a vertex. The vertex basically is a point in the rectangle (or quad).
As said before the image is a quad, so it has four vertices, as in bottom left, bottom right, top left and top right.
As also stated the OpenGL ES can draw only triangles. Therefor the quad is chopped down to two triangles. The order in which these are drawn are very important.
The interleaved vertex array holds the information on these four points. The second array mentioned before holds information of which of these points will be drawn next (usually referred as "indices").
The basic idea is that you can save resources when you don't have to pass multiple instances of the same information to the graphics hardware.

Instead you pass each information once and then use the indices to inform which vertice to draw.
To further simplify the logic: the first array holds the information and the second tells which one to draw from the first array.

This can be confusing at first but after working with the OpenGL for some time it starts to become clear. One reason for this is that the OpenGL syntax is pretty cryptic at first since it performs object oriented tasks with pure C.

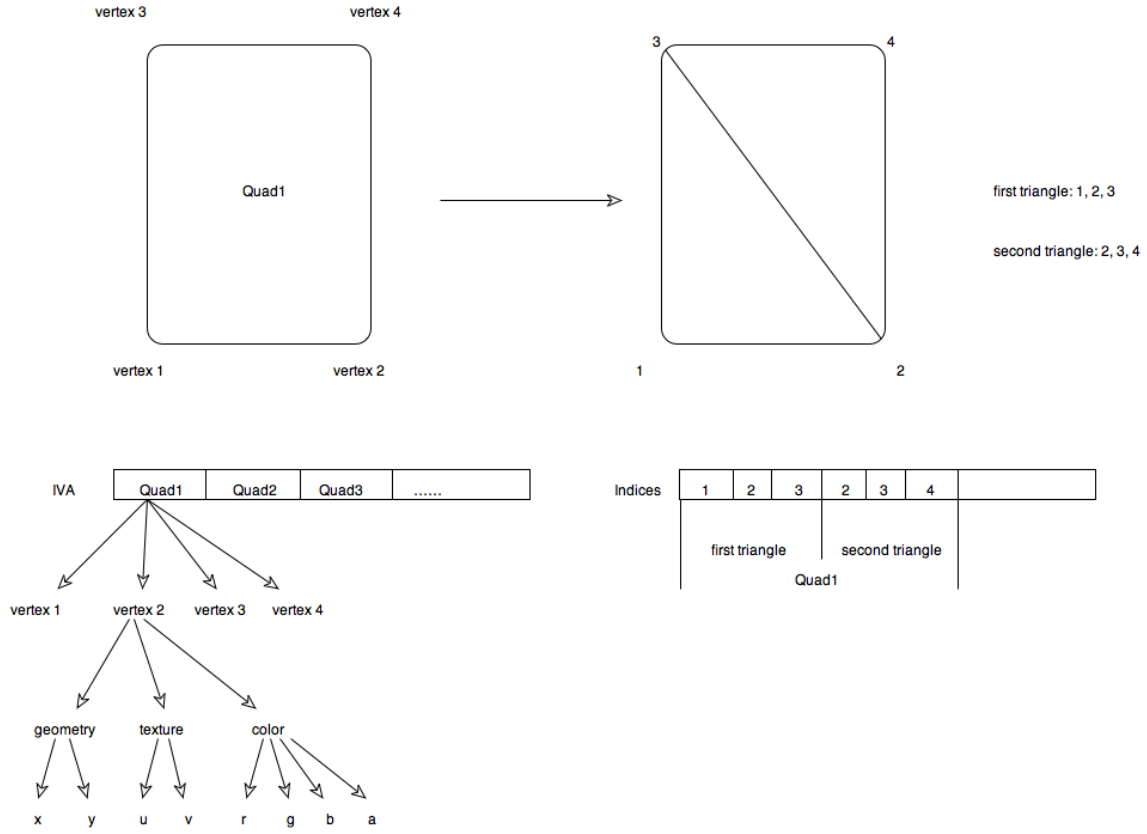


Figure 5. Drawing With OpenGL ES

When working with arrays the starting point is at index 0, so the first quad would be drawn from indices 0,1,2,1,2,3.
The method that creates the IVA takes the basic info from the game object and transforms it in the required form. Here is one of the most useful implementations of inheritance, since these structures exist in the GameObject class from which all drawable objects inherit from.

The OpenGL commands have a “gl” prefix. Since the IVA is an array of structures the OpenGL is given the appropriate pointer to each attribute. They include the geometry, texture and color -pointers.
After the pointers are set the indices –array is used for the drawing.

The pointers take the following parameters:
number of attributes, type of the attribute, the distance to the next parameter in bytes (also known as stride), and finally where to read the attributes.

The glDrawElements takes the following parameters:
type, number of indices, type, where to read the attributes.

Below is a graphical presentation of the stride.

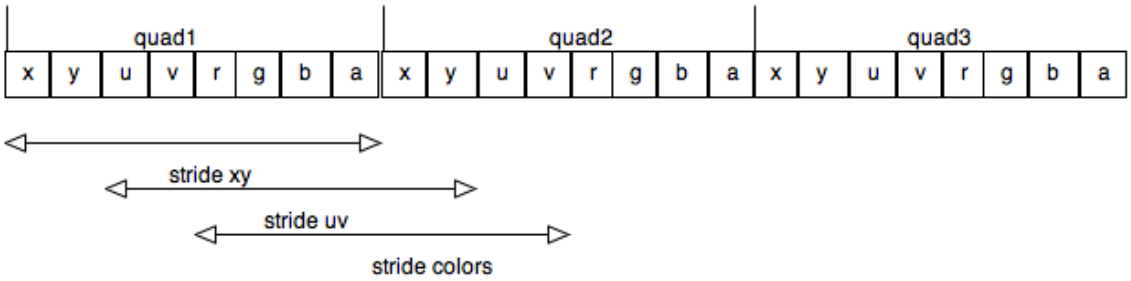


Figure 6. Stride

The compiling of the indices array requires a bit of calculating since the next quad doesn’t just flow naturally afterwards.
If the first quads triangles indices are 0, 1, 2 and the second 1, 2, 3.
The second quads indices are for first triangle 4, 5, 6 and second 5, 6, 7.
This is why the method for creating the array might look a bit odd.

It can’t be done 0, 1, 2 | 1, 2, 3 | 2, 3, 4 | 3, 4, 5, since the second quad would use the first ones parameters.

The geometry uses OpenGL coordinates which are from -1, -1 to 1, 1. Center (0, 0) in the middle of the screen.
The textures use UV coordinates which are from 0, 0 (top left) to 1, 1 (bottom right).

With the coordinates the screen ratio which is 3:2 has to be taken in to account. The coordinates are still from -1, -1 to 1, 1.
This can cause unwanted transformations if neglected.
Another thing is that the mathematical calculations use a standard x,y coordinate system. On top of that the user input has a coordinate system of 0, 0 (top left) to 320, 480 (bottom right).
So basically there's a need to use 4 different types of coordinate systems and get them to work together. The GameObject class handles much of this and is a crucial component. The coordinate systems are illustrated on appendix 5.5.

2.10 Texture Atlas

Creating and loading textures is an expensive operation. For best results, avoid creating new textures while your application is running. Instead, create and load the texture data during initialization.
(OpenGL ES Programming Guide)
Binding to a texture takes time for OpenGL ES to process. Applications that reduce the number of changes they make to OpenGL ES state perform better. For textures, one way to avoid binding to new textures is to combine multiple smaller textures into a single large texture, known as a texture atlas. A texture atlas allows the application to bind a single texture and then make multiple drawing calls that use that texture, the texture coordinates provided in your vertex data are modified to select the smaller portion of the texture from within the atlas.
(OpenGL ES Programming Guide)

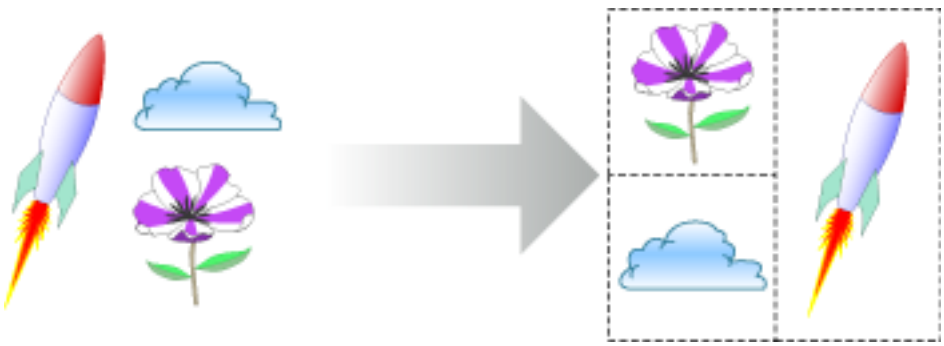


Figure 7. iOS Basic Structure (OpenGL ES Programming Guide.)

3 Moggy

The game that is used as an example is a simple 2d game that I've made. There is more information at www.villeforsman.com.
Below are three images with the left one being the earliest.
I suppose these could be called from left to right: alpha-, beta- and release-versions.

Figure 8. Moggy Screenshots

It has been a learning experience and I am fairly pleased with the release version. The game has changed a lot and I have put a lot of hours in to it. It was actually shocking how much time and effort it takes even to make a simple game from scratch.

3.1 The Project

The idea was to start out small and to get everything working and then start adding features. The first things to get working are the game world, graphics and user input. Without previous experience in game programming these will present new challenges.

The basic principles are fairly common but the technologies used will need some special attention. Also the target environment(s) will have their own possibilities and demands.

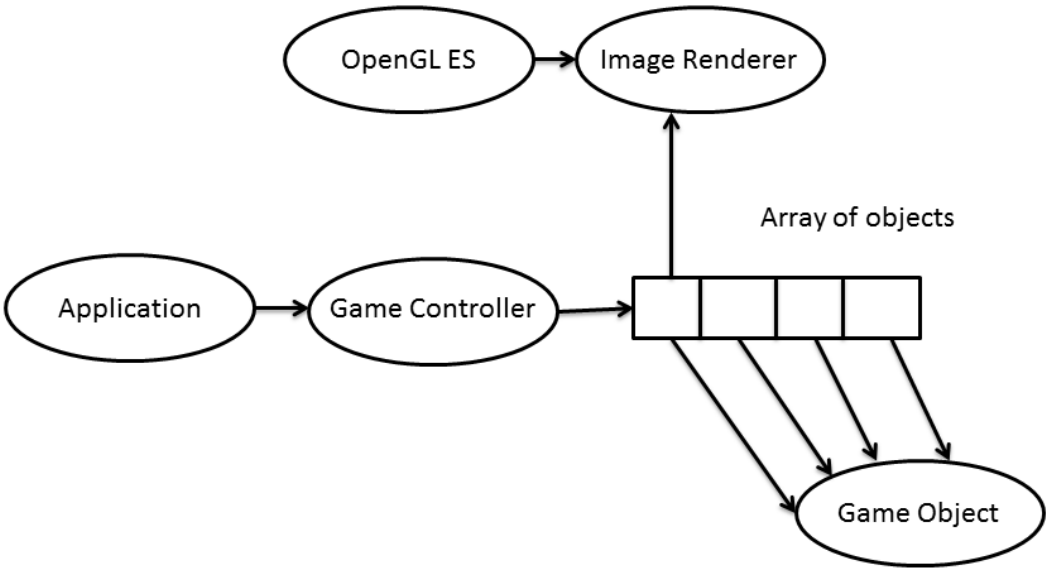


Figure 9. Simplified Game Architecture

The structure is fairly simple. The Game Controller is the core of the game. It controls the flow of the game as well as responds to notifications from the application and reacts to user input. The controller runs the game loop (Appendix 5.2) and updates the objects. The array is then passed to the Image Renderer class which does the rendering.

The game loop, rendering, audio, input and other necessary modules make the game engine. There are commercial and free game engines already which do not require (at least as much) coding or deeper understanding of what is going on.
In the example project everything is created from scratch. The engine can then be refined and expanded for later usage and it won't be dependent on other developers and/or copyrights.

3.2 Graphics

The graphics used in Moggy are 3-d models which have been rendered to a picture or pictures, also known as animation frames and then combined in to the texture atlas. There are a lot positive outcomes in this kind of an approach and for someone like myself who has limited talent at drawing this might be the only one.
The positives are that the pictures can be modified easily, the animation frames are easy to get accurate and the 3-d objects are reusable.

You can also get some nice looks for your graphics by using bump maps, lights and other cool features available in 3-d modeling.

Graphics made this way can really fool the user since everything you see in a 2d screen is 2d. Even if you had actual 3-d objects in the game you would see them in 2d in the end. Some big commercial games have used this kind of a "cheating" approach, such as StarCraft 2 made by Blizzard.
The graphics in Moggy are made to look nice but they're not even trying to be cutting edge. This brings us to the downside of 3-d modeling: it takes forever.
It does get faster when you develop a routine for it and learn your way around the programs used for it. The programs I used in this project were Blender, Cheetah 3D and Abobe Photoshop Elements 8. The graphics were created using either Blender or Cheetah 3D and combined into the texture atlas by using Photoshop.
Cheetah 3D is really simple compared to other similar programs, but it does cost about 100 euros and is not as powerful.
Personally I prefer Blender on a PC and Cheetah 3D on the Mac, besides it only runs on Mac.

The texture atlas would then be compressed into PVRTC format using a command line tool in the terminal. The PVRTC format is a very fast way for the iOS devices to read a picture and it also saves a lot of space on the hard-drive and in the memory while running the game. (OpenGL ES Programming Guide)
The atlas in the example project was 2048x2048 pixels, which was the current maximum size for iOS systems over 3.1, with a transparent background and the final size was approximately 2,5 megabytes.

It is also possible to use a picture such as a .png for the atlas and it might be a good idea to start from there. This way it is easier to locate errors because the PVRTC conversion and usage present new possibilities for failure.
This is a very hard thing to get working because if something is done incorrectly you might not see anything screen and finding the bugs can be extremely difficult.

3.3 Graphics Designin

When designing the graphics there might be very limited space for them, depending on the texture atlas used. Animating the objects heavily will cause the animation frames to take additional space. The animations in Moggy consist from three to six frames.
It is also possible to use objects or multiple atlases. However these approaches are more advanced.
The abilities of views can also be used here to add additional layers for the interface without using OpenGL.
In Moggy the graphics consisted of five layers. First the background using interleaved vertex arrays, then the background particle emitters are drawn, then game objects using interleaved vertex arrays, then the particle emitters which all draw themselves and finally the user interface layer which was a UIView with its own view controller.

This approach was primarily used so that the background could be animated. If it wasn't it might have been possible to use an UIView with an image for the background. Whatever the solution, putting the OpenGL graphics on the texture atlas translates into faster rendering.
I've heard advices to both for and against using UIKit and OpenGL graphics together but it worked out just fine. The application should be tested constantly and the frame rate monitored among other things.

One thing that I also implemented for the re-usage of images was the "nighttime"-method in the GameObject-class. This just changes the color values of each vertex from 1 to 0,5. This makes the image darker. Other effects can also be achieved this way.
For some reason I had trouble assigning new color structures to the object from another class, so the way to go was to do all the transformations inside the GameObject class and to just call a function (such as [gameObject nighttime]) without parameters.

3.4 Modeling

As for the modeling itself the resolutions used was a full HD camera resolution (1920x1080) and the final pictures scaled on the atlas. The resolution can be lowered so that the rendering becomes faster, although this might cause difficulties in clipping the pictures on to the atlas.
Most of the time I started with a cube and began extruding it.
There are plenty of modifiers that can be attached to an object and turned on/off as necessary.
One of the most common is the subdivision modifier which smooth's the object.
The catmull-clark subdivision works best on cube-like objects and simple on rounder objects.
The shift-E –command can be used to apply sharpness to lines or vertices. This is useful if some edges should be sharper and if used fully it will make the edge razor-sharp.

When creating animated objects it's easier to have all the moving parts as their own objects. This way you can modify (and see) their actual coordinates in the object tab.
If an animation has 5 frames and an object moves from say, 1.0 X to 1.4 X, the coordinates for the objects for the frames would be 1.0, 1.1, 1.2, 1.3 and 1.4.
When moving the objects it might help to change the coordinate system from global to local. The animation can either go back and forth the frame (1, 2, 3, 4, 5, 4, 3, 2, 1…) or loop them (1, 2, 3, 4, 5, 1, 2…). If you choose to loop them then the last frame needs to transition smoothly to the first one or there will be some twitching in the animation.

When working with objects where the left and right side are mirror images of each other, such as the human face, there are a few options on how to proceed. When not making a hugely complex model you could start extruding from the middle onto the both sides of the center and work on the vertices of both sides.
Sometimes however it might be impossible or very hard to modify the corresponding vertices on both sides at the same time and this result in some displacement in the final model.

However the picture will probably be a lot smaller when the user sees it and it doesn't have to be perfect.

Another approach would be to make the right or left side completely and then use the mirror tool to get the other side as an exact mirror image. This is one of those things where you have to practice and keep practicing.

Modeling of an object is usually done in the edit-mode where you modify the selected object. The object mode is for transforming the object as a whole. After having selected an object the edit-mode can be toggled with the TAB key.
When modifying an object the first thing to do is to choose what to modify. Here are three choices: points, lines and vertices.
Points are points. Lines are lines between two points. Vertices are the planes which consist of the two above and in the end are the visible things. You need at least three points so that you can have a vertice. This would be a triangle. With four points you'll have a square. Vertices can have as many points as you like but then it comes down to how it is split. Even a square consists of two triangles.

You can use the materials provided in the program or create your own ones in both Blender and Cheetah 3D. In Cheetah the metallic materials reflect a lot and if there's a black background the metals might reflect the black and therefore will not show at all. In both programs materials nodes can also be used to create materials.
The rest of this sub-chapter will be about Blender.

There are so many things that can be done with materials but again, one doesn't have to use everything. Basic materials, textures and bump-maps will go a long way. The core material will have its unique properties, as how light will reflect from it, color and transparency.
When using transparency (how much one can see through the object), the receive transparent –selection must be selected on all the materials that will be behind the transparent object.

The textures that are applied to the core material are kind of layers on top of the material. There are many types of material, which can be clouds, marble, image and so on. Then you choose the color (or even an image) for the material. The textures preview window will update according to the choices made. There are a lot of choices on how to use the textures and which values they will change. There can be multiple textures on one material.
Some other things can be easily applied for a powerful effect.

For example the grass in Moggys startup picture (which by the way only shows for a fraction of a second) is a cube with multiple subdivisions. Then it's been given some vertical differences by moving vertices up and down so it's not just a flat surface. Afterwards it's been subdivided with catmull-clark so that it's smooth again.

The material is pretty basic. Its reflectiveness has been reduced to 0, because grass does not reflect. It only uses a single texture which is a photograph of a lawn. This was used for the textures color and bump map. This makes the grass look more real.

The bump map is a way of making the surface look un-even. The bump map can be used by selecting the "Normal" in the textures properties. In Cheetah 3D the bump map has its own selection in the material.
The bump map uses the color values of the image provided for it to figure out which parts are lower and which higher.

Some metals used in Moggy were given a scale-like look by using a picture of the skin of a fish as a bump map.

When the model and its materials are in order there should be some final set-ups for the picture. In the cameras settings sliding the percentage to 100% will cause the rendering to use the full resolution defined. This is at 50% by default which means that the resolution is only 50% of the values given. This is just an easy way to make the rendering faster when in the modeling process. Also selecting the maximum values for the antialiasing will make the edges look less pixelated.

From the scenes settings the Indirect lighting -selection gives the picture a bit more reality. After this is turned on there is a selection between raytrace and approximate. Raytrace should be more realistic but also slower to render.
The background you use affects how the objects will appear especially when using materials that have high reflectiveness. Here the future can be made a bit easier by having the objects stand out from the background. When making the texture atlas the object must be cut from the rest of the picture and the intelligence in these programs can be a pain if the contrast is low between the modeled objects and the background.

In Cheetah 3D the look of the picture can be changed a lot by selecting the radiosity tag for the camera and the hdr tag. Then a picture can be used as either visible (will show as background) or non-visible background. This will really change how the objects will look like, since the rendering will make it look like the object is in the selected image.
The pictures can be in actual HDR, but a high-end camera is needed to snap them. These can also be purchased online (check Cheetah 3D's manual for more information).
I used normal .jpg or .png pictures myself and it was good enough for me.
HDR stands for High Dynamic Range and when using the pictures this way the idea is to get the light values from the picture to create realistic 3-d images.

The look of the picture can also be changed a lot by assigning multiple light sources. Light is very important for the final look and its easy experiment different outcomes. There are a lot of different types of lights in both programs and they can also be given color values.

3.5 Texture Atlas

The atlas used for Moggy was 2048x2048 pixels. The atlas can be smaller but it does have to be a square and in the power of two, such as 1024x1024. The atlas was compressed from png to pvrtc using a command line in the terminal. The actual program is in the developer tools. Also, a pvrtc loader class will be needed in the program but fortunately Apple provides a class for this (downloadable from the developer resources).

The atlas was created in Photoshop by giving it the dimensions and selecting a transparent background. The grids can be toggled on and off which makes positioning the individual pictures on the canvas a lot easier.
The texture atlas is used by the OpenGL in the game and it has coordinates from 0,0 (top left) to 1,1 (bottom right right) no matter what the size is in pixels. These coordinates are referred usually as UV or ST.
I tried to make life yet again a little easier by using the grids provided. In Photoshop a 2048x2048 image has approximately 36,2 grids. Therefore one grid would be (1/36,2) in the coordinate system. The indexes then are from 0 to 35.
I created methods so that the GameObject could use the indexes of the grids for position and size so that I wouldn't need to give every actual coordinate myself.

When adding a new picture to the atlas it is opened in a new window and using the "magical" selection tool to the outline of the object can be determined. You can either add to or subtract from the selection and change the brush size used for selecting.
Here the contrast between the object and the background is crucial.
Once the object is outlined the edges of the selection can be refined, after which the selection should be copied onto the atlas.
Here the image can be resized and repositioned to fit the grids.
When creating animation frames it's easier to copy-paste all the pictures needed for the animation and only then resize them.

The easiest way however is to render the background transparent. This way all that is left is to copy and paste the image. Also it would help to render the image as a square, such as 640x640, if the image is a square in the game.

Animation frames can be a real pain and you might have to make the atlas multiple times just to get the coordinates right. If they're off then the animation will twitch.
If all the right coordinates but everything is off then you need to change the grid value used for calculation (in this case 36,2).
This will take some tweaking.
Below are the alpha and final versions of the texture atlas.

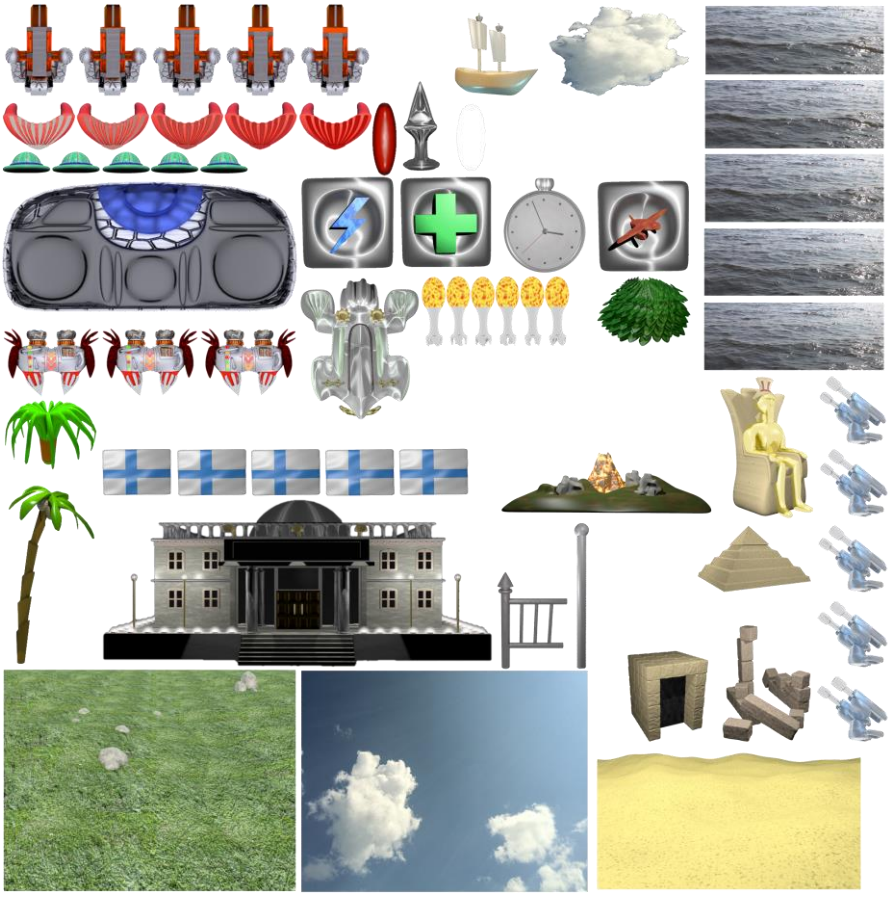


Figure 10. Moggy Alpha Texture Atlas

Figure 11. Moggy Final Texture Atlas

3.6 AI

The AI in the game is pretty basic but considering the game it should be more than adequate.
In general an AI can be something as simple as taking a random number and then deciding a single action based on it. (Appendix 5.3)

Complex AI -systems can learn, take other game entities in consideration or even predict different outcomes.

The AI in Moggy uses "mind states" to determine the entities own behavior. The entities that are referenced here are enemies so might as well call them that. The enemies don't care about their comrades or the player. What they do care about is if they have taken damage recently and their overall health status. Based on these they might change their "mind state" to another one or commit a specific action.
There are six different mind states in which the enemy can be in.

There is a diagram of the states and how moving from one to another is possible.
In the actual code there are different probabilities in each scenario on how the enemy will react.
For example an evasive enemy is more likely to panic if it has taken damage recently.
This creates a logical "human-like" aspect.

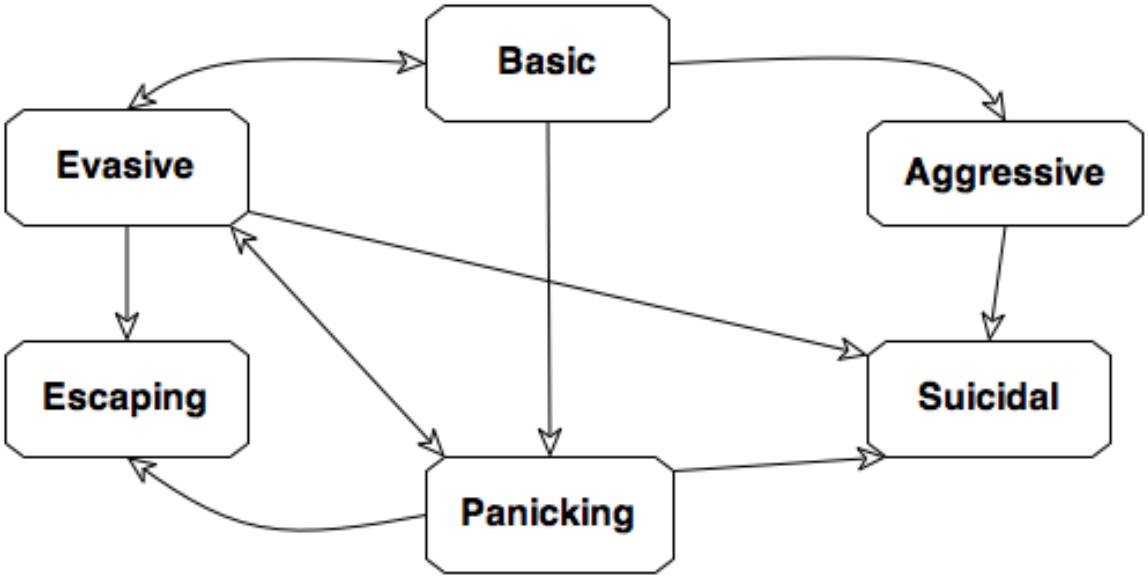


Figure 12. Mindstates.

It is achieved by having a massive switch statement with the mind states as the individual statements.
At the AI loop it runs the factors based on the current AI state. If the AI is to change its mind state it takes a random number and then checks which mind state to move to.

Example: You take a random number from 1-10.
If the AI is at the "Panicking" state it can only move to "Escaping" or "Suicidal".
If the health state is "Massive Damage" and it has taken damage recently then under these checks would be an if statement
Simplified: IF(random < 8)

{[self changeMindstate:escaping];}
ELSE{[self changeMindstate:suicidal]};

The actual code is a bit more complex since it might also stay in the current mind state and the AI contains a ton of these checks.
There are also timers so that the AI doesn't change mind state tens of times every second.

The mind states are typedeffed as whole numbers for convenience.
The enemies in Moggy inherit from the Enemy-class. This way most of the code is in the Enemy-class and the subclasses themselves specify what they are and what they can do. Also some enemy-specific implementations might be found in the subclasses themselves.
The Enemy-class has a lot of variables since it handles AI, moving, firing and pretty much everything.

The method that is invoked by the GameController is
updateState:(CGFloat)deltaTime

In this method are four sub-methods that are invoked.
First the Enemy-class calls its superclass to update with the given delta.

Secondly the AI values are updated with the delta.
These values consist of timers and other values needed by the AI, such as the current health condition.

Third step is to update the current mind state.
In this phase there's a check in the beginning of the current states method if the mind state can be changed. If it can, there are different probabilities to change the mind state or to stay at the current one, resulting from the current health condition, if damage has been taken recently and what the current mind state is.

Based on these factors the mind state might change. If it does the changeMindstate-method will be invoked.
If the mind state remains the same then the method might invoke specific actions, such as escaping, flipping or crashing.

The fourth step is to execute the actual movement based on the current values.
The values used can be divided into three categories:
1. The base-values for the current enemy-type (defined in the subclass)
2. The AI values that result from the current mind state
3. The buffer that takes care of the transformation from the old values to the new values.
Based on these the movement executes a randomly selected operation such as moving right or left and applies the values above to this action. In here are also timers so that the enemy can't change direction ten times a second.

The value buffer handles the transformation from the old values to the new ones. This occurs when the enemy either changes the vertical or horizontal direction and also when a new speed is implemented by changing the AIs mind state.
Without this kind of a buffer the enemy would change its movement instantly. The buffer gives it a smooth transition.
For example if the enemy is moving right and decides to switch direction it will keep moving right and slow down until the speed is minimal, after which it will change direction and start speeding up until it reaches its normal speed.

Another important function for the buffer is to store the new values and implement them as soon as possible. This means that the values will be locked when the enemy is performing a specific action such as flipping. This way the flip happens smoothly since the values remain the same for the duration of the action.
As soon as the action is complete the values will be unlocked and the buffer starts the transition to the new values.

The Enemy-class is fairly big and complex compared to other classes but this enables easier handling of the subclasses. You can select the subclassed enemy's abilities by simply changing a boolean variable to YES or NO in the initialization.
As an example the Boss-class has the variables canCommitSuicide, canPanic and canEscape set to NO.
That is all that is needed to handle the specific behavior.

3.7 Coordinates

The coordinates can be a real pain at first but you need to struggle with them only once. The Window coordinates are used for taking user input. If your program supports the iPhone and the iPad then you need to figure out the device at startup and then make the calculations based on that.

```
if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad)
{
    iPad = YES;
}
```

The OpenGL uses the normalized device coordinates and it's easiest if your game objects live in these coordinates also. The OpenGL also uses the texture mapping (or UV mapping) coordinates. (Appendix 5.5)
The calculations that you apply to the game objects yourself will naturally be done using normal mathematical coordinates.

3.8 Game Objects

The GameObject class handles much of the graphical transformations since all of these are visible. You could put a lot of this in a separate class just for the graphical aspects and sometimes it's hard to decide what kind of classes to create. I didn't want to explode everything but tried to keep things in some kind of order so these features are inside the GameObject class. One reason for this is that the class is subclassed to AnimatedGameObject and the subclass is also subclassed.

The class AnimatedGameObject is a subclass of the GameObject. The class provides the basic handling for animation. The class is also often subclassed and in the subclass the handling of the animation is overridden.
There were two approaches used in Moggy when handling animation. The first is to use the frames provided and loop through them.
The second approach is to move back and forth from the first frame to the last one.

When using the first approach the animation frames need to transition seamlessly from the last frame to the first. In the second approach there is more wiggle room.

At its core the AnimatedGameObject has a collection of texture atlas coordinates and the animation consists of switching the current coordinates to another from the collection. When the image is rendered it just renders the image based on the objects current values. The actual animation is just switching coordinates.

The animation needs a few variables to work with. There has to be variables for the time that has passed since the last frame (the last time the coordinates were switched), the delay (the amount of time that has to pass between frames) and the time that has passed since the last update, which is provided in the form of delta (13.3). The delta is added to the time passed and if this becomes larger than the required delay then the next frame is switched and the time passed is set to zero.

The Common.h is a header file that contains definitions that many classes use. It contains structures, macros and constants. This way only the header can be imported and there's no need to define the same things multiple times.
The structures are used mainly for storing the variables that the graphics use.

3.9 Game Loop

The game loop is often described as the heartbeat of the game. (Appendix 5.2)
Usually it "beats" 30-60 times a second. In the example project the rendering takes places each time and the update interval is set to 60 times per second. This way you can say that the frame rate is 60 frames per second, or 60fps. This is called a frame-based animation.
The rendering is considered the most resource-demanding process so sometimes the frame rate will be lowered for up to 30. If the frame rate drops lower the human eye will start to see lag in the animation. There is no use to update more than 60 times a second since it would only be a waste of resources or at least the battery if nothing else.

The loop is called using the CADisplaylink (Core Animation Displaylink). This is to ensure that the call to update the games state and animation will be done after the screen has finished drawing. This is generally called "vertical sync" in games.

You could use a timer with a value of 1/60 seconds to update the game but the new animation might be drawn before the previous rendering is complete. This will cause tearing in the animation.
It is also possible to update the game more frequently than it is animated. The reason to do this is to ensure stability and precision in the games "world". One of the most important things in a game is collision detection. If the games state is not updated frequently enough some of the collisions might go unnoticed, for example a bullet might go through an enemy but it won't be noticed.

When testing Moggy it was fairly clear that the frame rate won't drop under 30fps and the updating could be done in tandem with rendering.

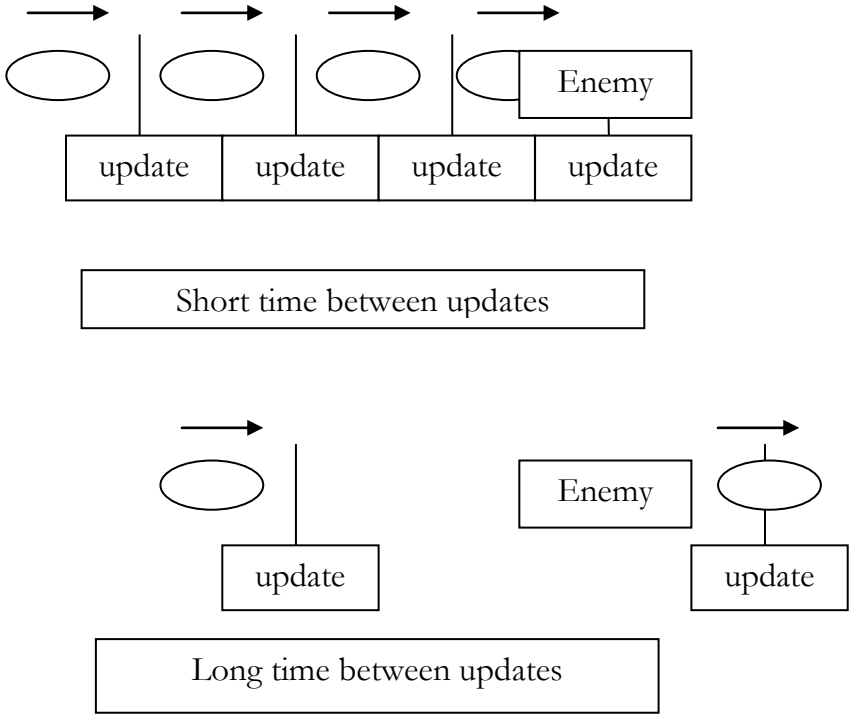


Figure 13. Difference Between Large and Small Delta Values

3.10 The Delta

The delta, often named deltaTime, is the time between the heartbeats. It can be calculated simply like:
delta = currentTime – lastFrame;
There are some things to consider though. When you start the game you will need to initialize the delta one way or another. Some functions can also cause some disharmony, such as a pause function which will cause the delta to become really big.

After the delta has been calculated it is then used to update the game objects. The root class for game objects has a function that takes the delta value as a parameter:
-updateState:deltaTime;
Inside this method the objects state will be manipulated using the delta value.
Often the changes will be an outcome of many variables, for example if the object is meant to move left the amount to move could be calculated:
float movementAmount = delta * speed * baseMovementAmount;
Then the value can be used to actually move the object:
[self moveVertical : -movementAmount];
The reason why it moves a negative amount is because it's supposed to move left. It's also possible to chain methods to make life easier so that there's a method "moveLeft : amount" which first does a transition "amount = (-1)*amount" and then calls the method "moveVertical : amount".

It might be easiest to work with variables that directly apply to the OpenGL coordinates. This way there's no need to transform them before rendering. This means that the x and y –axis range from -1 to 1. One thing to take notice is that the screen ration is different. This is covered in chapter Coordinates.
Therefore the values are usually somewhere between -1 and 1. This means the variables you work with are relatively small. If you take the example before:
float movementAmount = delta * speed * baseMovementAmount;
you need to think what the outcome will be. If we apply the formula with some fictional values:
float movementAmount = 0.15 * 1.1 * 0.1;

The calculation would produce a value of 0.0165. This would be the amount to move. Taking in to account that this method is called 60 times a second meaning that the object would move from the other side of the screen (-1) to the other (1) in approximately two seconds: (1-(-1))/0.0165/60.

The baseMovementAmount is a value that should be defined in the objects initialization and is used in these types of calculations. There should be own values for both axis such as xBaseMovementAmount and yBaseMovementAmount respectively.

3.11 User Input

The input is received by the corresponding view, which in this case is the EAGLView. The view passes the received actions to the game controller.
The user input then triggers functions in the game controller. These functions modify the state of the game, such as firing the turret or pausing the game. The input is in window coordinates and needs to be transferred to the OpenGL coordinates since the games "world" is in these coordinates.

Also after transitioning the project to work on the iPad as well it has its own coordinate system. To figure out if the current device is an iPhone
or iPad the GameController uses the following code in its initialization:

```
if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad)
{
    iPad = YES;
}
```

When transforming the user input the methods used will take in consideration the iPad boolean.
However, there might be many different types of devices in the future with different resolutions and coordinates. Because of this it would be smart to do the calculations with values asked from the operating system instead of hard-coded values. The games view resides in the applications window and the window can be asked for its frame. The frames values can then be used to calculate the correct OpenGL coordinates.

There are many methods that exist already for handling the input, such as
`-(void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event`

The programmer needs to implement these methods and decide what to do with the input. It might be easier to create the classes used to handle input as subclasses of ViewController instead of NSObject. You can then take advantage of the parent-child hierarchy which causes the input go up the chain until one of the classes implements these methods.

When programming a game there might be a need to handle multiple different types of inputs, such as double taps or continuing taps where the player holds their fingers on the screen. There are plenty of other types also and these can be easily implemented if necessary.

3.12 Events In The Game

The events need to be discovered inside the game loop. The updateGameLogic – method handles a lot of checks to determine what is going on in the game and then calls for the necessary functions. This is easily the largest method in the game and should be programmed one thing at the time in order to find possible bugs.

It's fairly easy to keep track on events by using boolean variables. One of the most common checks is to see if an object is alive (in-game) and remove from the game objects it if necessary. The object itself might be an enemy which decreases its health when hit by a projectile and sets its boolean variable alive to NO when its health is zero or less. The object might also be a particle emitter that has been alive for the required time and then sets its boolean alive to NO. When the objects handle their own state you only need to interact with them with the logical methods. For example an enemy has a method that decreases its health by the amount given (this can be acquired from the projectiles).

The collision detection is another very important section in almost any game. The logic is to figure out if an object collides with another object. In Moggy it is only necessary to determine if a bullet hits an enemy or if an enemy bullet hits the player. All of these objects are in the same array so that they can be easily updated with the delta and also passed on to rendering.
When detecting the collision it comes down to iterating over the array and detect if one of the objects collides with another. Here the game takes advantage of the Core Graphics library in order to make a rectangle (CGRect) from the current object and to detect if it intersects with another rectangle, made from the next object. The GameObject –class has convenience methods for creating these, although they are very simple but it reduces the amount of code in the game loop which is already a big chunk.

When making the rectangle the CGRect object has constructors which are given the vertex coordinates of the game object. Because the game world is in OpenGL coordinates there is no need for any transformations (which is nice).
The CGRect also has the intersectsWithRect-method which is given another CGRect as a parameter so everything is already done.
The one thing left is to get the boolean if the rectangles intersect and then figure out what type of objects are colliding by checking their booleans (isEnemy, isBullet, isPlayer) and then take the necessary action.

3.13 Image Renderer

The graphics are created using OpenGL ES 1.1. All game objects and the background objects are drawn as squares consisting of two triangles. This is because the embedded system OpenGL doesn't know how to draw squares. The ES1Renderer class is responsible for setting up the context for drawing. This includes setting various parameters and creating the buffers on to which rendering is done.

Afterwards the drawing can (and will) be done from another class. Since the game uses only one context all the drawing is done to that context no matter which class performs the drawing.

At the end of each game loop iteration the ES1Renderer presents the buffer. In other words it shows the graphics on the screen.
The game takes advantage of interleaved vertex arrays (chapter 2.9) which are the recommended way to create powerful graphics. The basic idea is to reduce the calls to the graphics chip by handling transformations in the program itself and passing the graphics data as few times as possible.

The actual drawing and game logic still use the OpenGL coordinates so those did not require any modifications.

For more information on the device coordinates some good reading is Apples View Programming Guide iOS, (View Geometry and Coordinate Systems).

4 Conclusion

Developing games can be a lot of fun and rewarding but it will take a lot of time and effort. If you do everything yourself then you need to understand a lot of different concepts and technologies. Some experience with computers and programming is pretty much a necessity. There are a lot of problems you need to overcome so you should have a genuine interest and determination, or at least stubbornness.

Starting out small and having a realistic view on things is the way to go. The bigger games are developed by hundreds of professionals for years, so it's impossible to compete with that. However mobile devices do give some of the advantage back to individual developers and many people can make something extra by developing games if not a living.

There are plenty of free game development tools that you can use without having to understand what is happening under the hood. Here you will have to decide if the destination is more important than the journey.
Gaming industry also employs all kinds of professions, from programming to marketing so there are a lot of different kinds of opportunities.

For myself it was a struggle at first and it took a long time to get anything done. I did the mistake of trying to understand everything before actually making anything. After actually starting the project I started to get somewhere. Even then it took several weeks before actually seeing something other than error messages.
This was all new to me so the next projects should go a lot smoother. The current project is now in the release process.

I would not recommend this kind of do-it-yourself approach to anyone who is not interested in programming. Even a simple game like this uses hundreds of pages of code. It does include many features that could not be covered in this thesis.
Usually the programming takes most of the time and the actual game is secondary. However once you have done the programming then making the game is fairly straight forward.
Games can be developed in many different ways using different technologies. The targeted environment also has a big impact on the development. Apples frameworks and libraries are very convenient but sometimes confusing. They do provide a lot of material in their developer pages which also include practical examples.
After the initial shock and awe you will start to find out the ideas behind the cryptic chunks of code and start to appreciate the work behind them. The environment is built to be very flexible and powerful.

The iOS platform is very effective for any kind of an application. Starting out with simpler programs is a good idea just to get the lay of the land. Even games can be developed without the confusing rendering and complex methods. iPhone and iPad Game Development for Dummies is a great book for anyone starting iOS game programming and I would recommend it as a first reading. Books like Learning iOS Game Programming will go heavily on the more complex subjects and are really confusing at first.

Anyone with a Mac can start programming for the iOS devices. Apples policy has been (and might change) that you can register for free as a developer and start making programs. You can then run them on the iOS simulator.

Paying the required fee will allow the developer to run their programs on the actual devices and also publish them to the AppStore.

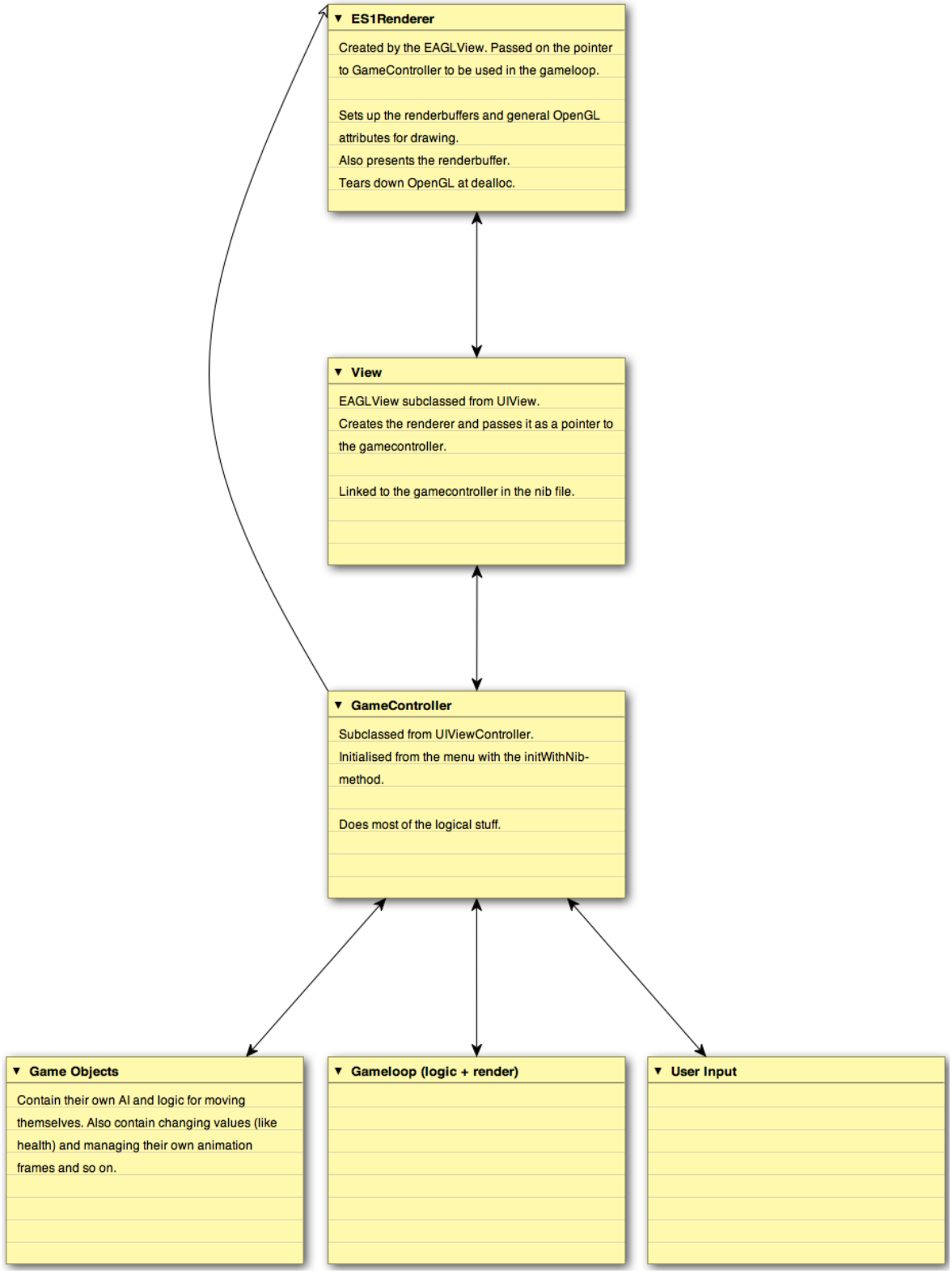
Applications made for Mac are also developed in a similar way. The differences come in the libraries and technologies. The programming environment and the language(s) stay the same. It might also be an easy way to make applications, with a special praise to the Interface Builder, which makes user interface creation fast and easy.

It will be interesting to see if anyone even buys Moggy from the AppStore. In any case I am already designing a new game and maybe someday I might be lucky enough to do this professionally.

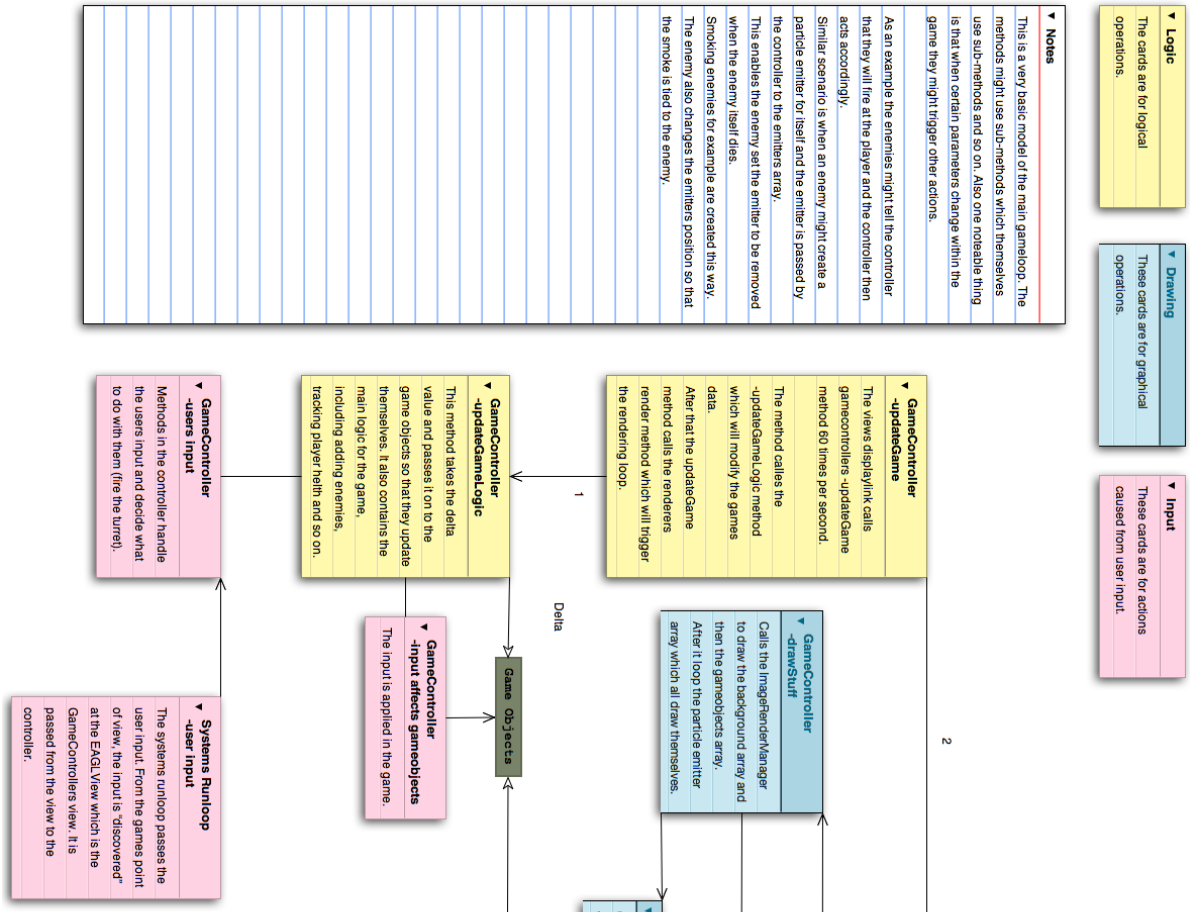
During this project I learned a lot about games, programming, 3d-modeling, audio, iOS programming and software development. I am certain that this is only the beginning and there will always be more to learn but I wouldn't have it any other way. Programming has interested me for years and developing games takes everything even further. The Wikipedia article on programming includes a very good definition: "There is an ongoing debate on the extent to which the writing of programs is an art form, a craft or an engineering discipline.". This is a very good definition and it describes what makes programming so fascinating.

5 Appendix

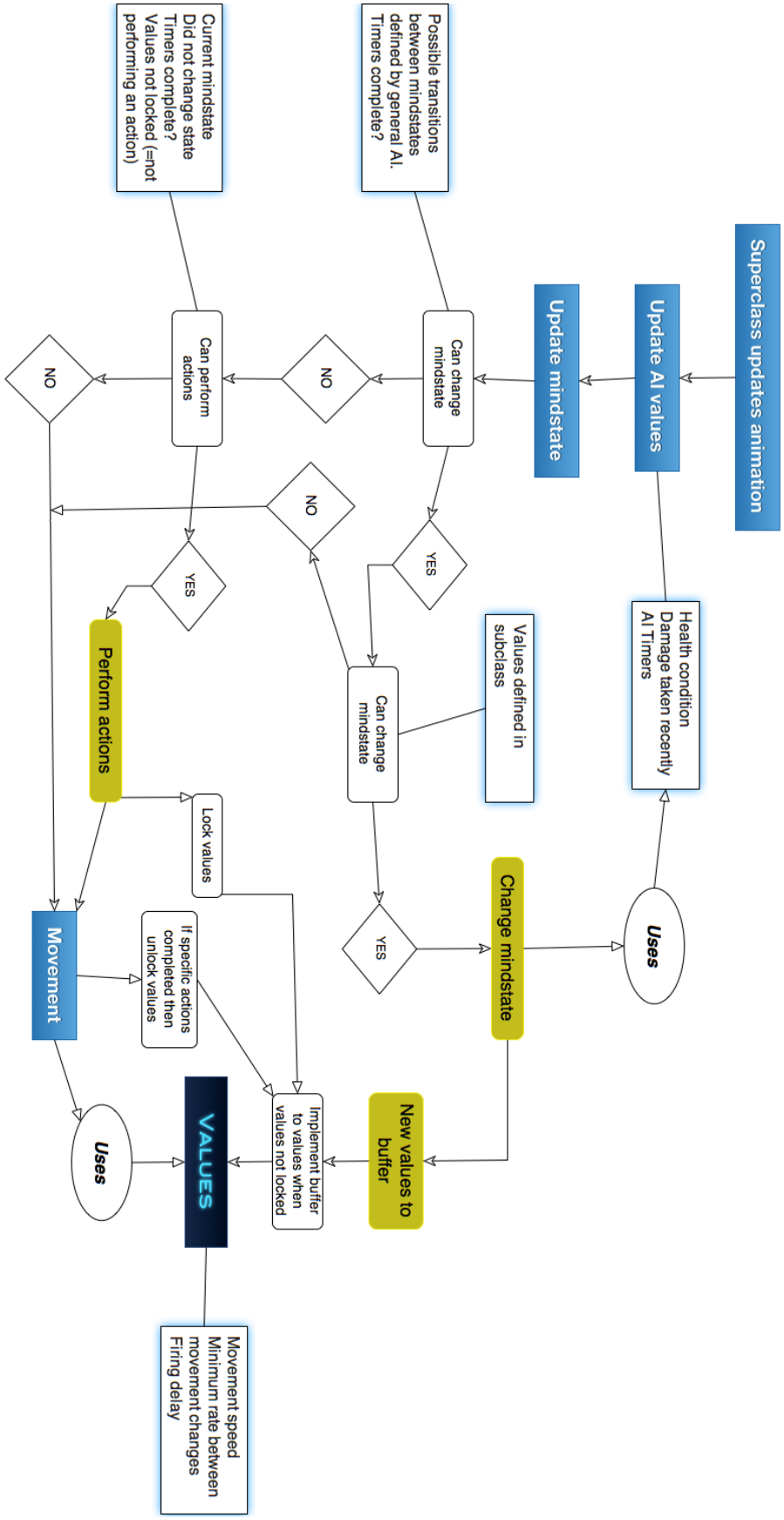
5.1 Architecture



5.2 Game Loop


Logic
The cards are for logical operations.
Drawing
These cards are for graphical operations.
Input
These cards are for actions caused from user input.
Notes
This is a very basic model of the main gameloop. The methods might use sub-methods which themselves use sub-methods and so on. Also one noteable thing is that when certain parameters change within the game they might trigger other actions.
As an example the enemies might tell the controller that they will fire at the player and the controller then acts accordingly.
Similar scenario is when an enemy might create a particle emitter for itself and the emitter is passed by the controller to the emitters array.
This enables the enemy set the emitter to be removed when the enemy itself dies.
Smoking enemies for example are created this way. The enemy also changes the emitters position so that the smoke is tied to the enemy.
GameController -updateGame
The views displaylink calls gamecontrollers -updateGame method 60 times per second.
The method calles the -updateGameLogic method which will modify the games data.
After that the updateGame method calls the renderers render method which will trigger the rendering loop.
1
GameController -updateGameLogic
This method takes the delta value and passes it on to the game objects so that they update themselves. It also contains the main logic for the game, including adding enemies, tracking player helth and so on.
GameController -users input
Methods in the controller handle the users input and decide what to do with them (fire the turret).
Delta
Game Objects
GameController -drawStuff
Calls the ImageRenderManager to draw the background array and then the gameobjects array.
After it loop the particle emitter array which all draw themselves.
GameController -input affects gameobjects
The input is applied in the game.
ParticleEmittersArray
Contains all the particle emitters which draw themselves.
2
3
ImageRenderManager
Draws the background and then the game objects.
Every drawable object derives from GameObject class (including background). This way the manager can easily access the objects graphical data and draw them using interleaved vertex arrays.
Renderer -render
Calls the renderers render method which has two stages.
The method asks the controller to draw the game and afterwards it presents the renderbuffer (displays the current frame on-screen).
4
Displays the buffer
Presents the final picture on screen.
Systems Runloop -user input
The systems runloop passes the user input. From the games point of view, the input is "discovered" at the EAGLView which is the GameControllers view. It is passed from the view to the controller.

5.3 AI


Superclass updates animation
Update AI values
Update mindstate
Health condition
Damage taken recently
AI Timers
Uses
Possible transitions
between mindstates
defined by general AI.
Timers complete?
Can change
mindstate
NO
YES
Current mindstate
Did not change state
Timers complete?
Values not locked (=not
performing an action)
Can perform
actions
NO
YES
Perform actions
Lock values
Movement
If specific actions
completed then
unlock values
Uses
Values defined in
subclass
Can change
mindstate
YES
Change mindstate
New values to
buffer
Implement buffer
to values when
values not locked
VALUES
Movement speed
Minimum rate between
movement changes
Firing delay

5.4 OpenGL ES

The most important OpenGL ES object types:

A **texture** is an image that can be sampled by the graphics pipeline. This is typically used to map a color image onto primitives but can also be used to map other data, such as a normal map or pre-calculated lighting information. The chapter "Best Practices for Working with Texture Data" discusses critical topics for using textures on iOS.

A **buffer** object is a block of memory owned by OpenGL ES used to store data for your application. Buffers are used to precisely control the process of copying data between your application and OpenGL ES. For example, if you provide a vertex array to OpenGL ES, it must copy the data every time you submit a drawing call. In contrast, if your application stores its data in a vertex buffer object, the data is copied only when your application sends commands to modify the contents of the vertex buffer object. Using buffers to manage your vertex data can significantly boost the performance of your application.

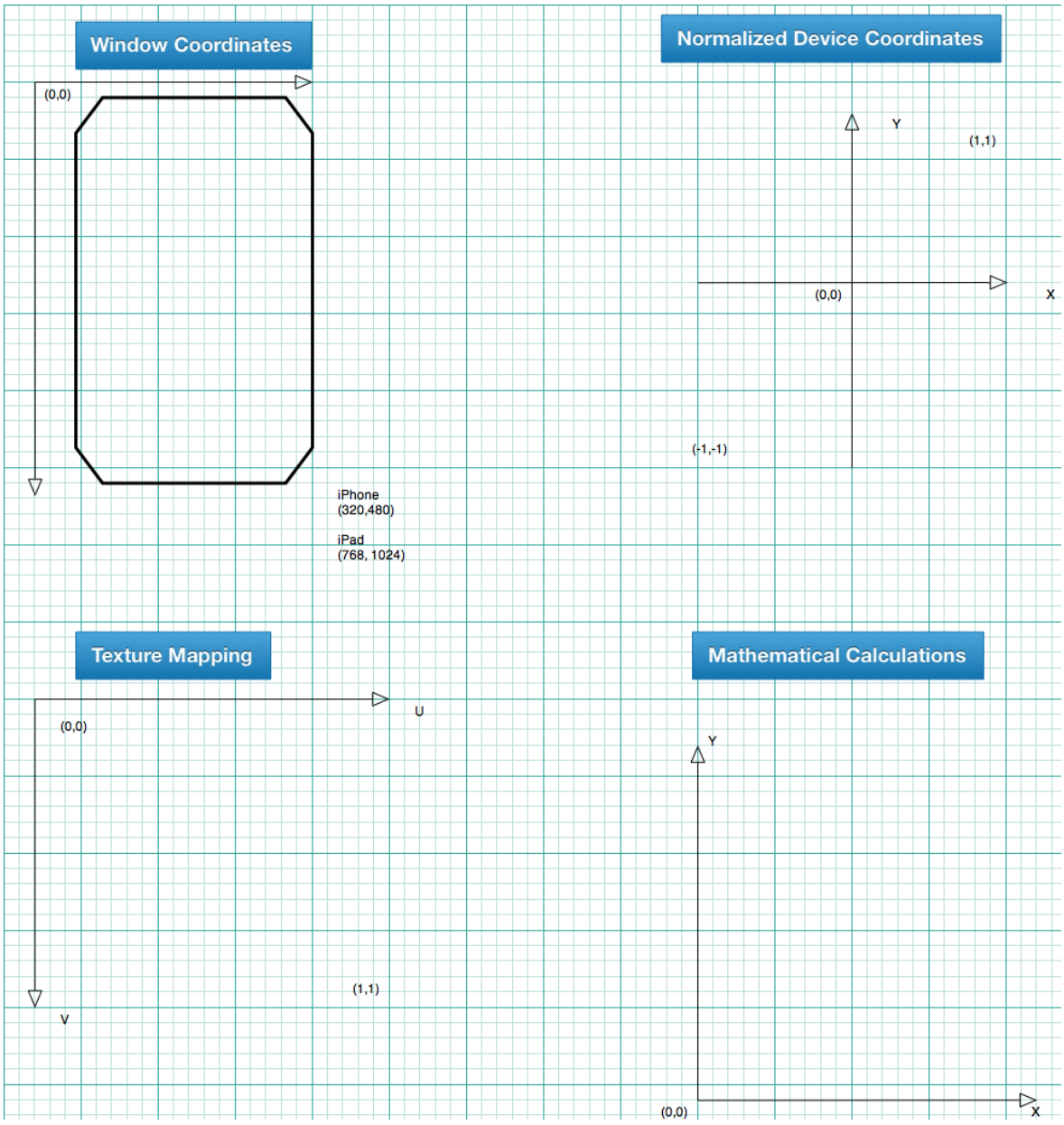
A **vertex array** object holds a configuration for the vertex attributes that are to be read by the graphics pipeline. Many applications require different pipeline configurations for each entity it intends to render. By storing a configuration in a vertex array, you avoid the cost of reconfiguring the pipeline and may allow the implementation to optimize its handling of that particular vertex configuration.

Shader programs, also known as shaders, are also objects. An OpenGL ES 2.0 application creates vertex and fragment shaders to specify the calculations that are to be performed on each vertex or fragment, respectively.

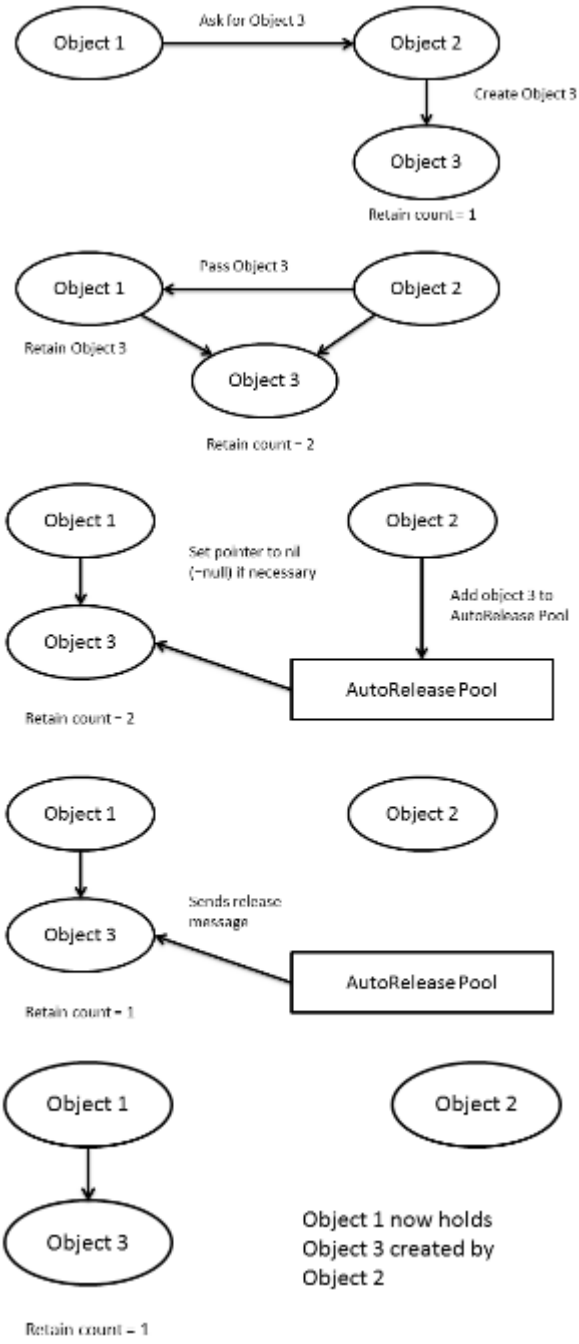
A **renderbuffer** is a simple 2D graphics image in a specified format. This format usually is defined as color, depth or stencil data. Renderbuffers are not usually used in isolation, but are instead used as attachments to a framebuffer.

Framebuffer objects are the ultimate destination of the graphics pipeline. A framebuffer object is really just a container that attaches textures and renderbuffers to itself to create a complete configuration needed by the renderer.
(OpenGL ES Programming Guide)

5.5 Coordinates



5.6 AutoRelease



Sources

Apple Inc. 2011. View Programming Guide for iOS.
Luettavissa:
http://developer.apple.com/library/ios/#DOCUMENTATION/WindowsViews/Conceptual/ViewPG_iPhoneOS/Introduction/Introduction.html

Apple Inc. 2011. View Controller Programming Guide for iOS.
Luettavissa:
http://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007457

Apple Inc. 2011. iOS Application Programming Guide.
Luettavissa:
http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgramming-
Guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007072

Apple Inc. 2011. OpenGL ES Programming Guide.
Luettavissa:
http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html

Apple Inc. 2012. Memory Management.
Luettavissa:
http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html

Steinberg, D. 2010. Cocoa programming. The Pragmatic Bookshelf. Dallas, Texas.

Goldstein, N. & Bove, T. 2010. iPhone Application Development All-In-One For Dummies. Wiley Publishing, Inc.. Indianapolis, Indiana.

Clair, R. 2011. Learning Objective-C 2.0. Pearson Education, Inc.

Aachen University 2010. iPhone Application Programming.
Ladattavissa: iTunes.

Stanford University. 2010. iPhone Application Development.
Ladattavissa: iTunes.

Daley, M. 2010. Learning iOS Game Programming. Pearson Education, Inc.

Goldstein, Manning & Buttfield-Addison. 2011. iPhone & iPad Game Development for Dummies. Wiley Publishing, Inc.