12-2012

The Design and Implementation of a Mobile Game Engine for the Android Platform

Jon Hammer
University of Arkansas, Fayetteville

THE DESIGN AND IMPLEMENTATION OF A MOBILE GAME ENGINE FOR THE ANDROID PLATFORM

THE DESIGN AND IMPLEMENTATION OF A MOBILE GAME ENGINE FOR THE ANDROID PLATFORM

A Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR
December, 2012

by

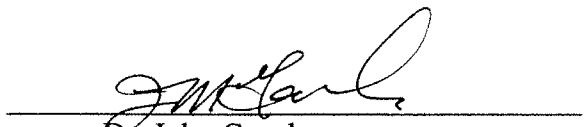
Jon C. Hammer

ABSTRACT

In this thesis, a two-dimensional game engine is proposed for the Android mobile platform that facilitates rapid development of those games by individual developers or hobbyists. The essential elements of game design are presented so as to introduce the reader to the concepts that are crucial for comprehension of the paper. A brief overview of the Android Operating System is also included for those unfamiliar with it. Three primary design goals are identified, and a prototype solution is described in detail. The prototype is then evaluated against those design goals to see how well it accomplishes each task. The results are collected and presented at the end of the thesis and demonstrate that though there is always room for expansion, the prototype does indeed satisfy the requirements given.

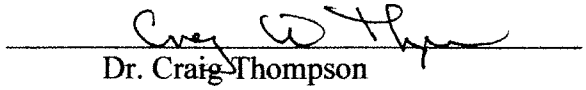
This thesis is approved.

Thesis Director:

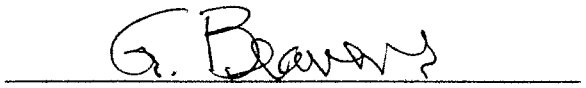


Dr. John Gauch

Thesis Committee:



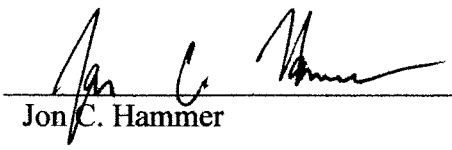
Dr. Craig Thompson



Dr. Gordon Beavers

THESIS DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed 
Jon C. Hammer

Refused _____

ACKNOWLEDGEMENTS

This project has been a monumental undertaking and would not have been possible without the support of a large number of people. I thank Dr. Russell Deaton and Dr. Nilanjan Banerjee for providing me with the original foundation of the project. Dr. Deaton encouraged the idea of a developing a gaming engine and was kind enough to be my Thesis Director during the preliminary stages of the engine's development, while Dr. Banerjee provided the knowledge and tools necessary to implement it on the Android mobile platform.

I also thank Dr. John Gauch and Dr. Craig Thompson. With the departure of Dr. Banerjee and Dr. Deaton, Dr. Gauch graciously took over the task of being my Thesis Director in the last semester while I finished the details of the project and has been a source of encouragement along the way and Dr. Thompson took over the responsibility of coordinating the students in the Honors program, myself included, and served on my thesis committee. I also thank Dr. Gordon Beavers for serving on my committee, and the rest of the Computer Science faculty at the University of Arkansas. I consider my education to be my most valuable asset.

My family and friends have been particularly important throughout this process. My mother and father have always pushed be to do my best and to strive for the highest level excellence possible. Their encouragement has pushed me through high school and college and will continue to push me for the rest of my life. My friends have always been available when I need them, either for advice or a distraction, and for that, I wish to thank them.

Finally, to all who have helped me along the way, I sincerely thank you. It has been a pleasure working with you.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Problem

In recent years, the number of "smart phones" has increased significantly. Smart phones are cellular phones capable of performing significantly more complicated tasks than their earlier counterparts through the use of specialized applications. They currently represent 47% of the cellular market in the United States [9]. Many new kinds of applications have emerged due to the unique nature of the cell phone, specifically its status as a device that is mobile, because it is carried around constantly throughout the owner's day. Because people tend to have many short periods of inactivity throughout a given day (such as those incurred when waiting at a bus stop or when taking a break from work), games have become increasingly common on mobile platforms, as they can provide a momentary "escape" from the hustle and bustle of reality [10].

Games currently represent a large percentage of the mobile application library, and more are routinely being developed [11]. Game development has therefore become a non-trivial part of the various mobile application markets. Compared to other gaming markets, however, those designed for cell phones are unique in that the market is not completely dominated by professional studios (as is the case in the home console sector, for example). The relative ease of working on cellular phones combined with a reduction of many of the fundamental barriers involved with console development greatly encourage individual development. Development kits are either freely available or available with a minimum number of restrictions when working with mobile devices. A wealth of documentation and examples are also available for the most popular platforms, like Android and iOS. As such, individual developers and hobbyists are free to pursue their own projects, and their output is placed alongside that of the larger game development companies [12]. The notion is attractive for many.

As development time is a scarce resource for this particular audience, at least compared to larger studios, reducing it is very important because it reduces costs and allows for a faster return on the investment. Addressing this issue requires examining "bottlenecks" in the development process, just as similar bottlenecks must be examined when attempting to optimize a piece of software.

One of the most commonly faced problems is that each project tends to start from scratch, which wastes a considerable amount of development time. A more efficient solution, then, is to develop a common starting point for game designers--a core set of basic functionalities that allow developers to express their intentions at a high level, without having to worry about the intricacies of the hosting platform or most of the mundane details involved with displaying an image on the screen. The more efficient solution is to build a mobile gaming engine, one that takes care of these issues, abstracting them away so the designer can focus on game content and creation.

1.2 Objective

The objective of this project was to develop game development middleware between the Android operating system and the game developer, enabling the developer to focus on game play and design, rather than the mechanics of interacting with the hosting hardware and other "boilerplate" activities. Specifically, the engine absorbs the tasks of file system interaction, user input, the "game loop," which will be explained in detail in Section 2.1.2, and rendering objects to the screen.

1.3 Approach

The game development engine was not intended for use by studios or even professionals. It was designed to simplify the task of an individual developer, a hobbyist, or a student who wishes to learn about game design itself. It was not designed to be all-encompassing ("everything and the kitchen sink") , as is common for larger engines, but rather to be small, fast, and simple. As such, the primary goals of the project were for the engine to be easy to use, modular, and efficient. Ease of use refers to the level of difficulty involved with learning how to properly use the engine (which should ideally be minimal), while modularity allows the engine to be extensible, so new behavior can be added by another developer without modifying the engine as a whole. Efficiency refers to both computation speed and the size of the storage needed to properly implement the algorithms that need it.

To actually make any progress in the development of a game using the engine versus starting from scratch, it has to be "easier" in some form. In this particular context, it means development time should be minimized. An obscure or inconsistent API would only serve to lessen the benefits created by using an engine in the first place. If more time is spent learning how to use the engine than is saved by using it at all, the designer's time has been wasted. This ease-of-use requirement is met by using a clear internal structure and a consistent format, as well as providing lucid documentation.

Power is another issue to consider. It is important to remember that the actual devices in context are not "normal" computers in the sense that they are designed to be mobile. They are not constantly tethered to a wall, providing, from an application's point of view, an essentially unlimited source of energy. The designer of a mobile application must be mindful of power usage, primarily through the use of input devices, sensors, and CPU clock cycles. Each of these

hardware components will require a certain amount of the phone's battery usage to operate properly.

Of the sources previously listed, the CPU clock cycles are the element that developers have the most control over. Reducing the number of cycles used generally corresponds to using a more computationally efficient algorithm for the same task, since less efficient solutions tend to require more clock cycles. Therefore, if the proposed engine is to use less power and fewer clock cycles, it must be designed to be more efficient, again in terms of computation speed. Another benchmark, the "frames per second" will be introduced in section 2.1.2 and will be used to gauge the performance of the engine in Chapter 4.

Another issue to consider is that of limited storage space. The internal memory (Random Access Memory, or RAM) and the backing storage (typically some sort of flash-based memory) is usually more limited on a cell phone, even a "smart phone", than on the average home computer, though there are exceptions. Some versions of the Samsung Galaxy S III, for instance, contain up to 2 GB of RAM with 64 GB flash storage. [13] Although special considerations like these typically do not need to be made for applications written on platforms such as Android, it is something to keep in mind when developing.

The engine, as proposed, should try to meet these goals as well as possible. To actually construct it, though, several specific details had to be considered. First, since the engine is a program (technically part of one), it must be written in a programming language. As the Android Standard Development Kit (SDK), as well as the Android runtime itself, are both written in the Java language, programs written for Android devices tend to also be written in Java [14]. Writing the engine in that language avoids an arguably unnecessary layer of complication that would likely arise if another language was used.

Another caveat relates to the integration with the hosting Android operating system (or OS). The decision had to be made of how much of that host to "abstract away", in the sense that the developer does not directly call routines provided by the host SDK, though they are likely called by the engine itself. The notion of abstraction has the immediate benefit of making developer code portable across other mobile platforms, provided the engine itself has been ported to that platform, but unique functionality provided by one platform may not be available as a result. In accordance with the modularity objective mentioned above, OS-specific functionality has been put in its own "core" (see Chapter 3), thus alleviating those drawbacks.

1.4 Organization of this Thesis

Chapter 2 on Background is split into two primary sections. The first is an overview of the background material needed for comprehension, and includes a discussion on the fundamentals of graphics processing, the game loop necessary for the engine to run, an overview of animation itself, and a brief section on the Android operating system. The second section describes several related projects that encompass the spectrum of gaming engines on the Android platform, including AndEngine, LibGDX, and Corona.

In Chapter 3 provides a detailed description of the game development engine’s architecture. A high level design of the framework is followed by the designs of each of the primary components. The implementation is discussed in detail afterwards, again starting from the core framework and working down to that of each of the primary components. Two of the basic engine extensions are also covered, sprites and particle systems, that demonstrate proper usage of the API to create new constructs.

The engine is benchmarked by several methods in Chapter 4, according to how well it fulfilled the various goals set forth in Section 1.3. Example code is given to show ease of use, explanations are provided for extending the engine beyond its current feature set, and three test applications are proposed to stress the engine in different ways, allowing the engine's performance to be gauged appropriately. The various results are collected and presented at the end of that chapter.

Chapter 5 presents a short summary of the project, as well as the various conclusions derived. This chapter also demonstrates the significance of the engine, not only to its intended audience, but also to the software development community as a whole. The final section outlines various ways the engine could be improved to further aid game designer making use of it.

2. BACKGROUND

2.1 Key Concepts

In order to fully comprehend this thesis, the reader should be familiar with some of the essential mechanics involved with general 2D game design and with the Android operating system. The three fundamental topics in game design include graphics programming, the "game loop", and the essential mechanics behind animation. An in-depth familiarity with the Android platform is not required, but an overview covering the important information is included in Section 2.1.3. It is, however, assumed that the reader understands basic programming constructs, such as arrays, loops, simple data structures, and object-oriented design (including inheritance and the idea of an interface).

2.1.1 Digital Images

Colors are the basis for all images, digital or otherwise. Without them, there would be no coherent way to tell different images from one another, and the discussion of graphics programming would be a moot point. As such, it is vital to understand how colors are typically represented in a computer so that we may work with them.

Though there are many such representations, the one most commonly used splits a given color into three core components, red, green, and blue. These three colors form a basis for all other components, in that all other colors, black and white included, can be generated using different amounts of each. For example, equal amounts of red, green, and blue will typically yield a shade of grey, while equal amounts of just red and blue will typically produce a shade of purple.

Actually quantifying these colors requires a somewhat arbitrary choice to be made. Typically, a range of some sort is selected for each component (though usually that range is the same for each, for simplicity's sake). For example, it could be decided that each component would be a whole number between 0 and 9, inclusive, meaning there would be 10 separate values for each color component, giving 10 * 10 * 10 = 1,000 possible different colors in this scheme. It could also be decided that each component would be a floating point number between 0.0 and 1.0, inclusive, in which case the number of colors is only limited to the precision that those floating point numbers are stored.

Because of limitations placed by screen manufacturers, the range has traditionally been set to a whole number between 0 and 255, inclusive, which provides 256 unique values for each color component (giving 256 * 256 * 256 = 16,777,216 possible colors). By convention, white is defined to be the set (255, 255, 255), while black is defined to be the set (0, 0, 0). A nice side-effect (and actually a contributing factor to the decision in the first place) is that each color component can be stored on a computer in a single byte, with no wasted or unnecessary space. If each component takes up one byte, the whole color then, can be stored using three.

Unfortunately, on most computers, bytes are not addressed (able to be retrieved) one at a time. For performance reasons, they are typically packed into groups of four or eight (called a "word") and sent off as a single unit. If a color is stored using three bytes, then one byte per color is not being used at all (assuming a 32-bit word, or four bytes), which results in a waste of storage space. It is for this reason that the fourth byte is usually given some sort of " auxiliary " status. That is, it is available for use to serve some other purpose than providing color information. A common usage is to include an "alpha channel" that stores transparency

information in the fourth byte. A value of 255 means the color is completely opaque, while a value of 0 means the color is completely transparent, for example.

When referring specifically to digital images, each color is displayed on the screen using "pixels." A pixel is the smallest unit of information a monitor or screen is capable of displaying, and one pixel stores one color, usually represented using the four-byte approach described above. The "resolution" of a screen represents how many pixels it can display and is usually given in two parts (for example, 1280 x 800). The first number represents the number of pixels that can be displayed in the horizontal direction, while the second represents the number of pixels that can be displayed in the vertical direction, assuming the "wider" of the two sides is horizontal. The resolution of a screen is important because it is fundamentally limits what can be shown on that screen. That is, anything we wish to render must be either scaled (magnified or minified) or cropped (part of the image removed because the image is too large) to fit in the space allotted [2].

To display digital images, then, is to map the pixels of some input image to a certain region of the screen. The source of the image can be a digital picture, an artist's design, or can be generated programmatically, but the mapping is the key factor. At its most basic level, the screen is a blank canvas of pixels whose size is limited by the resolution. Though the undertaking is almost always abstracted away in some form, it is essentially the task of the programmer to fill the canvas--to decide the values of each and every pixel on screen at any given point in time.

2.1.2 The Game Loop

Although static images can be pleasant to the eye, filling the screen with data just once does not usually constitute a "game." It more closely resembles a picture or a painting. A game is defined by the interaction between the player and the system designed by the game developer. Specifically, a game must be dynamic, representing a changing environment in some form or another. For this to happen, a static image is not enough. The pictures must move.

This new requirement poses important challenges for the developer. Not only is he tasked with filling every pixel on the screen, but he must now do it repeatedly--and quickly. Several new questions are raised, though. "When should we fill the screen? How often? Do we only need to fill the screen when something changes?" The answers will require more explanation. This process is commonly known as "rendering" and is but a single step in the pipeline that is the “game loop.”

The change from rendering static images to dynamic ones introduces at least one new element that must be considered in our pipeline. The images must now be updated before they are rendered. These two operations form a cycle that is to repeat as long as the game is active. Images are updated, then they are rendered. Then the process repeats itself until the game is exited. This process is known as the "game loop" and is the backbone of most games and game engines.

During the update step, each object on screen is updated according to its own set of rules. For example, objects might be affected by gravity and fall toward the bottom of the screen, or they might be moving and must be moved left or right depending on its current velocity. The "trick" is to achieve the motion smoothly, which requires handling movement over a series of iterations of the game loop, each time only moving the object slightly.

During the rendering step, each object is drawn to the screen in some order. This is done by mapping the source image's pixels to a predefined location on the screen, then moving on to the next object. The order in which objects are rendered is important--items drawn first tend to be partially overwritten by objects drawn later on. This is often exploited in 2D games to create the illusion that one object is "in front" of another, with respect to the player's viewpoint. Backgrounds, for instance, are usually drawn first, followed by characters and level objects, then a "heads-up display", or HUD that provides information to the player during the game. This ensures important details are always visible to the player.

Unfortunately, there is still one significant piece missing from our current game loop. The player has no way to actually interact with the game. To borrow a piloting metaphor, the game is on "autopilot." The addition of the update mechanic turned a static scene into a dynamic one, turning out painting into a movie. Similarly, the addition of a user input mechanic will turn the movie into a proper game.

To accomplish this, the hosting operating system should inform the game when the user has provided some input, by means of a mouse movement or click, a key press, or as in the case of mobile phones, touching the screen. This "event" (see Section 2.1.3 below) is recorded by the game and saved until it can be processed. Processing is deferred until the next update step in order to give a predictable flow of events. Each iteration of the game loop, input events are handled appropriately, objects are updated, then everything is drawn to the screen as described previously. This is the final "flow" of the game loop [1].

In order for the game loop to be effective, though, it should operate within certain parameters. Most importantly, it must run (complete a single iteration) very quickly. For the human eye to perceive motion, the "frame," a shot of the screen at any given time, must change

at least 25 times per second. There are a number of caveats with this figure, as the human eye technically can perceive motion at lower numbers under certain conditions, but 25 is a reasonable value to target. A number that is higher will result in smoother motion (up to a point) while numbers that are considerably smaller will cause the motion to appear jagged or choppy. As a result, it should be a goal of the game designer to maintain at least 25 fps (or frames per second) on average while the game is running, or else risk losing the intended audience because the game is perceived as being "slow."

It is important to note, however, that different iterations of the game loop might take different amounts of time. A scene with a multitude of objects, for example, would require much more time to update than one with only one object. Because the fps, the number of game loop iterations per second, is variable, the game must also be flexible enough to show movement at the same pace, regardless of whether the fps is very high or very low (ideally). This also has the side effect of making the game playable on a huge variety of computers, such as those found in the cell phone market. If the animation remains constant regardless of the fps, a scene will display at the same speed regardless of whether it is being played on a fast computer or a slower one, rather than displaying very slowly on a slower computer and incredibly quickly on a faster one.

A common solution to this problem is to include a *dt* variable as a parameter to the update routine [15]. This variable represents the time that has elapsed since the last frame was rendered and can be included in velocity calculations to ensure fps independence of animation. Small values imply a higher fps, so the animation should be slowed, while higher values imply a lower fps, meaning the animation should be sped up. A pseudocode example of the completed game loop is shown in Figure 1 below.

```
1  void gameLoop()
2  {
3     //store the previous time and the current time
4     def prevTime  = null;
5     def beginTime = currentTime();
6
7     while (active)
8     {
9         //calculate the value of dt
10        prevTime  = beginTime;
11        beginTime = currentTime();
12        def dt    = beginTime - prevTime;
13
14        //perform the game loop operations in order
15        processInput();
16        update(dt);
17        render();
18    }
19 }
```

Figure 1: Pseudocode for Game Loop Routine

2.1.3 An Introduction To Animation

At this point, we are prepared to discuss how animation itself works. As mentioned earlier, the frames per second (fps) required for the human eye to perceive motion is roughly 25. In other words, the screen must be redrawn once every 1/25 of a second at a minimum for an object's movement to be smooth. If the same image is used every frame but the coordinates change, the object will appear to have a velocity that moves it in some direction, though the object itself will not move. On the other hand, if different but similar images are used every frame but the coordinates remain stationary, the object will appear to move in place. Changing both the images used and the coordinates drawn will cause the object to both move and have velocity in some direction.

As a concrete example, assume we are trying to draw an image of a walking man to the screen. His feet are placed at the bottom of the screen and will walk in the direction he is facing until he hits a wall, at which point he turns around and begins to walk in the opposite direction. To start off, we will attempt to make the man "walk in place", that is, the x or y coordinates (assuming a typical coordinate system, where x is the horizontal axis and y is in the vertical axis) will not change between frames. Because walking is inherently a continuous operation, and drawn frames are a discrete construct, we will need to "sample" the man's stride. We will need to take a set of *n* static images over the period of one stride and switch between them every frame at a speed of at least 1/25 of a second so the user perceives the man as actually walking [1].

If the images taken were evenly spaced over the course of the man's stride, we can make use of the inherently periodic nature of walking and the entire sequence can be "looped". The images are played in the order they were taken, and after the last frame has been displayed, the sequence starts over again. In this way, the animation can continue indefinitely. As an effect, however, a sequence containing an animation can only be placed once, in which case a non-periodic animation results. If the man were to kick a soccer ball, for example, he would probably only kick once, not repeatedly.

If at this point, the animation sequence is played continuously, the man would appear to be walking in place, though he would not actually be moving anywhere. To remedy this, the man should be given a velocity in each of the horizontal and vertical directions. The vertical velocity would likely be 0 in this case, as the man is not supposed to be flying. At each frame then, the position where the image is drawn, the coordinates of the image, would change according to the velocity in each direction. A positive horizontal velocity might move the man

to the right, while a negative one would move him to the left, for example. A value of 0 in both directions results in a stationary object [1].

This basic process is the foundation upon which all animation lies, though there are a multitude of complications that may be added to deal with different circumstances. Multiple animation sequences can be used to animate different actions, such as walking right versus walking left, and collision detection can be implemented to detect when one object has come in contact with another, which usually spurs one of these changes of sequence.

2.1.4 Android, An Overview

Android is an "open source" Linux-based operating system designed for use in higher end cell phones, sometimes called "smart phones". Open source refers to the fact that the source code for the entire system is freely available for anyone who has a desire to inspect it. Android is currently maintained by Google, Inc., who supervises its progress and development, and currently holds the majority of the worldwide smart phone market share [8].

To develop for the platform, a programmer need only install the Java Development Kit (JDK) and the Android Development Kit (ADK) on their development machine. The ADK includes tools for testing applications in both an Android emulator as well as on an actual mobile phone that has Android installed. The actual applications are typically written using the Java programming language, as that is also the language the Android runtime is written in. Publishing applications can be done by registering as an Android developer, which gives the developer the ability to submit their work to the Android marketplace, from which that application can be downloaded directly onto a user's device [14].

Android, like many other operating systems, uses an "Event-driven" model [16]. When something "important" happens, such as touching or redrawing the screen, an "event" is raised, which is then sent off to anyone who has registered with the event handler that they are interested in the news. To respond to a given event then, one must simply tell the hosting operating system that they wish to receive those updates.

Touch events are the most important to a game designer on the Android platform because they provide the main source of user input to the game, though Android does provide access to a multitude of other sensors that an individual mobile phone might possess. These might include an accelerometer, GPS, proximity sensor, and light sensor, depending on the specific phone model. Typical Android phones also have access to various radios, including cellular, wifi, and 3G/4G which allow communication between other cellular devices and other networked devices, like computers.

2.2 Related Work

The use of game engines has become more prevalent in recent years. It is becoming increasingly common for major studios to use them as a starting point for high-profile games, rather than completely starting from scratch at the beginning of each project. As such, many more options are being created, both for professional use by large companies and for individual or small team development by hobbyists. Engines that are designed to work specifically with the Android platform and that are catered to 2D games are less common, but are in existence. Three such are described here to show the wide variety of engines of this type and are presented in order of escalating complexity.

2.2.1 AndEngine

AndEngine is an open source solution that tends to fall in the "Hobbyist" end of the spectrum described above. It has been created and is currently maintained by a single developer, and is available free of charge (including the source code) to anyone with a desire to design a game using it. AndEngine is designed specifically for the Android platform (hence the name), and so is not designed to be cross-platform at all. This is one of the primary differences between it and the other engines discussed later.

AndEngine is discussed because it bares the most similarity to the proposed system. Its primary focus is essentially the same, but the implementation is very different. Specifically, the internal structure is significantly more complicated, but no less structured than the design presented for this engine. AndEngine also absorbs some tasks that are not part of the engine design given. It includes a wrapper for Box2D, a physics engine, for example [3].

2.2.2 LibGDX

LibGDX is similar to AndEngine in that it is also maintained by a single developer and is open source, but has a more expanded set of use cases [5]. For example, it is designed not only to operate on the Android platform, but also as desktop applications and HTML 5 web applications using Java. It also includes a set of APIs to work with advanced math and physics in an effort to more completely encapsulate the game development process [4]. With the expanded set of features comes several drawbacks, however. The engine is necessarily much more complicated to accommodate the other features, and rendering requires knowledge of OpenGL, which can contribute to the time needed to learn the API considerably. Though it is more powerful, it is more difficult to use.

2.2.3 Corona

Corona Labs produces a commercial gaming engine called Corona that is considerably more opulent than the other engines described so far. Corona is not an open source solution. It uses a licensing system in which each developer pays to have access to the development kit for a full year. Corona Labs does not collect any royalties from applications built using its software, however. Any of these applications can be built for both Android and iOS devices, but requires that the user write in the Lua scripting language, rather than the more familiar Java [6].

Users of Corona have access to a massive set of libraries to handle nearly any situation the game developer might come across, making it one of the "everything and the kitchen sink" solutions described above. Besides basic rendering, Corona provides access to mathematics routines, physics engines, networking functionalities, video playback, and a complete set of native user interface elements like buttons and text boxes, among many other things. One of the goals of the Corona project is to eliminate any interaction whatsoever with the hosting platform, so it must necessarily be much more extensive than either of the other two projects mentioned above [6].

Corona is included in this discussion because it contrasts so starkly with the other engines presented, as well as with the engine that will be presented in the next section. It serves to demonstrate just how completely an engine can encompass a project. This is one of the solutions that would likely be used by a large studio, rather than a single developer, so the design goals are very different.

3. ARCHITECTURE

3.1 High Level Design

The engine is organized in a hierarchical fashion. It is composed of a set of cores, each of which contains specific functionality necessary for operation. Each of these cores is composed of one or more components that accomplish specific goals. The components "report" to the enclosing core, which then communicates with the engine itself. There are five primary cores, the File Core, the Graphics Core, the Input Core, the System Core, and the Update Core. A high level diagram of the engine is shown in Figure 2 below:

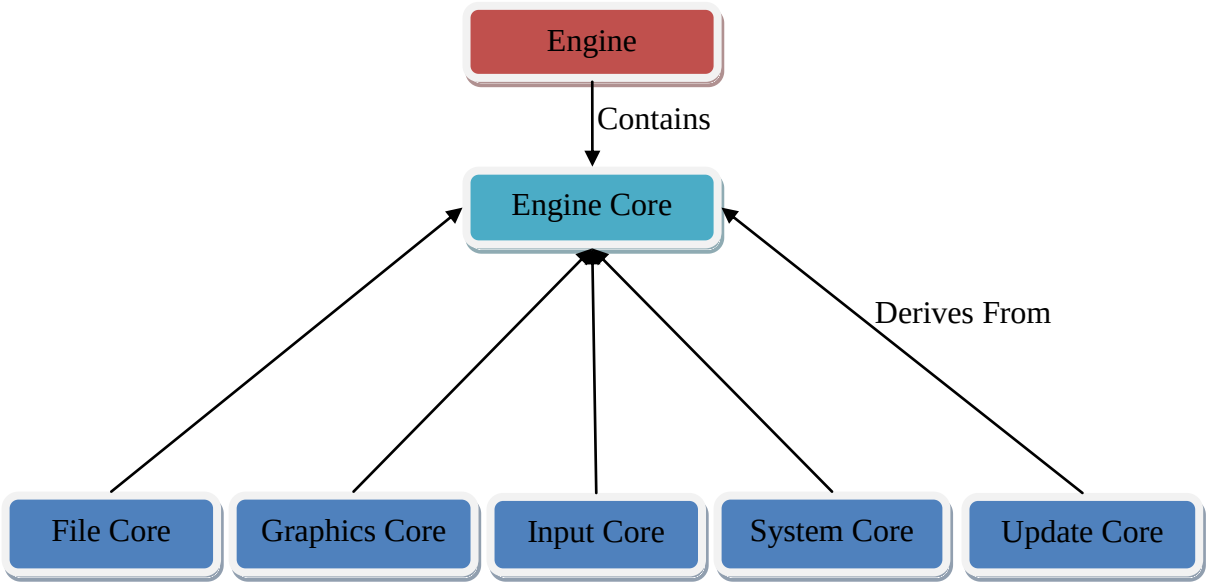


Figure 2: High Level View of the Engine

The engine is built on top of the Android Standard Development Kit (SDK) and performs almost all communication with the host operating system. The user communicates primarily through the engine to perform important operations, though it can be bypassed in favor of the Android SDK for particularly performance-intensive operations or when using the engine would

be inappropriate. That being said, it is recommended for the user to make use of the engine whenever possible for the sake of portability.

3.2 Design

In the sections that follow, the basic purpose of each core is defined, omitting specific Architectural decisions. Those will be explained in detail in Section 3.3 with specific implementation issues and their solutions. The information presented here represents a high level discussion, independent of those issues.

3.2.1 System Core

The system core is responsible for functionality that is unique to the Android operating system. If the engine were to be ported to another platform, such as iOS or Windows Mobile Phone, capabilities that are unique to those systems would be placed here. In the current version, for Android, the methods are primarily designed to aid with the input of resources such as images and level maps and to aid with "bootstrapping", giving the developer an entry point from which to start. The process is described in more detail in Section 3.3.1 below.

3.2.2 Graphics Core

The graphics core is by far the most sophisticated of the engine cores. In order to optimize the rendering of objects, each object that needs to be drawn on the screen must register itself with an object called the Renderer. It is the Renderer's job to draw each object that has been registered every frame.

To be drawn by the Renderer, an object must either be, or contain, a Drawable. A Drawable contains information on how exactly to draw itself, delegating the task from the Renderer. The Renderer only has to tell each Drawable to "draw" itself on the screen. Unfortunately, there are many different types of objects that can be drawn to the screen. Images read from files, strings of text, and shapes (like squares, circles, and the like) can all be drawn, so each needs its own "Drawable" implementation. There are also several extra Drawables to help with specific tasks. Sprite sheets and tile sheets for maps are two examples. Using these extra Drawables over their fundamental counterparts can potentially impact performance drastically.

To actually create the content that is displayed on the screen, a set of managers are used. Each manager provides a way to create a certain resource. After creation, a unique identifier is returned to the creator, which is used to locate the resource when it is needed. The actual resource is stored by the manager, along with all other resources of the same type. This common storage allows pooling of resources so multiple copies of each resource are unnecessary. Indeed, if an attempt is made to create a resource that already exists, the unique identifier is simply returned, skipping the creation step (as the resource already exists).

The Renderer, the Drawables, and the various managers work together to facilitate drawing of the object, while the graphics core supervises the operation. It is important that this operation be efficiently implemented, as the drawing time is non-trivial, especially for larger projects, where the fps can drastically increase or decrease simply by using one method of drawing over another. This is demonstrated in the benchmarks run on the engine in Chapter 4.

3.2.3 Input Core

This core provides the ability to process input events, specifically touch events. Its primary component is the Touch Propagator, whose purpose is to respond to touch events sent by the operating system. It will record the event and propagate it to all objects that have requested the information during the update phase of the game loop.

Those objects are "tagged" by implementing an interface called Touchable. Any object that implements this interface can register itself with the Touch Propagator to receive touch events. The primary method of interest is called `onTouch()` and is called whenever one of these events is received.

3.2.4 Update Core

The update core operates very similarly to the input core. The Updater, the primary component, is in charge of updating every object that moves on the screen. Any object that is to be updated, then, must register itself with the Updater and implement the Updatable interface, which defines the action to be taken by the Updater. The update core informs the Updater when to actually perform the update operation.

3.2.5 File Core

The classes and methods in the file core allow the user to access the underlying Android file system to read and write standard text files. Methods are also included to delete a given file (regardless of type) and to determine if a given file actually exists. Also included is the ability to read in XML files using a standard parser. XML files are returned in an *n*-ary tree data structure called an XML Tree, in which the hierarchical structure of the original file is preserved. The

contents can be accessed by traversing the tree in some fashion. The tree is composed of nodes, each of which has an element name, a set of attributes, and a list of child nodes, which hold the connections in the document.

3.3 Implementation

A more detailed description of the inner workings of the engine created for this project is given here. The singular class at the top of the engine hierarchy is named, conveniently enough, Engine. Engine maintains a collection of cores, each of which inherits from the abstract base class Engine Core. It also is responsible for sending appropriate commands to each of the cores when needed. For example, it tells the graphics core when to render everything and it tells the update core when to update each item on the screen.

Each Engine Core in turn maintains a collection of Engine Core Components. Generally each core has several components that accomplish various goals. Each core will be described in more detail below. This collection of components can be added to or removed from at any point, even at run time. As a result, so long as a given class inherits from Engine Core Component, it can be included into the engine as seamlessly as any of the predefined components. This structure allows the engine framework itself to be very extensible and modular.

3.3.1 System Core

The system core deals most directly with the hosting operating system. As such, most of its job deals more with "bootstrapping" than actual engine operation, though the system core does expose some functionality that can be used during the run time of a game, such as the ability to display dialog boxes or to vibrate the phone, for example.

Every program must have a starting point of some sort, typically in the form of a `main()` method where normally a window would be created and the game loop would begin. In the Android environment, however, the `main()` method is hidden from the programmer, buried in a series of initialization routines that handle most of the "grunt work". Where does the programmer begin, then? He must first create an object that inherits (directly or indirectly) from a class called "Activity". An activity is similar to a window in a more traditional operating system. It is a single screen, an empty canvas, that can be filled by the programmer in whatever manner he desires [17].

What goes on the canvas then? A typical Android application is filled with buttons, text boxes, labels, and other accoutrements common in graphical user interface design. In our case, what we want really is an empty canvas. That is, we want an object to put on the screen that covers the entire screen that allows us to modify individual pixel values. Using the Android terminology, what we need is a Surface View, an object that accomplishes this goal. A view represents any user interface element and is the ultimate super class of anything that is shown on the screen.

This Surface View must be added to the Activity so that it may be used. Also, an instance of the engine must be created and initialized before any other task can be accomplished. To simplify the task of the application programmer, an Activity that performs these duties, Game Activity, has already been created and is part of the system core.

As part of its initialization, Game Activity creates a Game View, an extension of Android's Surface View. The Game View starts a new thread that is responsible for handling the game loop. The thread starts an infinite loop during which the input processing, updating, and rendering actions in the engine are called in turn, as described in Section 2.1.2. Though the

game loop starts immediately, the game does not actually begin until the engine is formally started. This allows the game developer the opportunity to perform any game-specific initialization, like creating objects and levels, before starting the engine himself.

Although not strictly necessary, the abstract base class Game has been provided to allow the developer to separate the actual game initialization routines from the hosting activity, which aids in producing a game that is easily portable to other platforms. One could certainly initialize the gaming constructs without it, but the abstraction might prove to be helpful, especially in larger projects where object creation and initialization can incur a significant amount of the development time. It is recommended that the user create a custom game class that inherits from Game that is instantiated in their starting Activity.

3.3.2 Graphics Core

There are four primary components to the graphics core, plus a number of auxiliary classes that accomplish different goals. These primary components include the Renderer and three content managers: the Texture Manager, the Font Manager, and the Shape Manager. The majority of the auxiliary classes are various implementations of the Drawable interface for different purposes and will be described in detail in Section 3.3.2.4. The diagram below shows the breakdown of the Graphics Core.

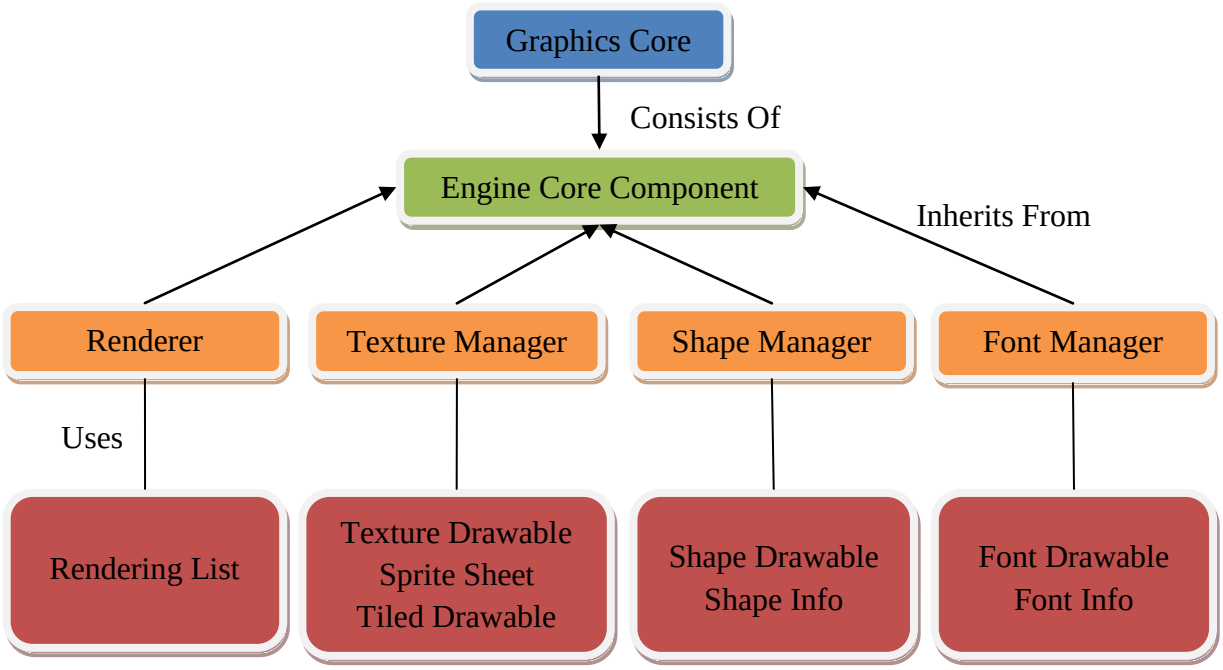


Figure 3: Breakdown of the Graphics Core

3.3.2.1 Renderer

The purpose of the renderer is to actually display every item on the screen when requested to do so by the graphics core. Because of the wide variety of objects that can be displayed on the screen (images, text, and shapes are the primary candidates), the renderer must be flexible enough to draw any object that meets certain conditions (namely the ability to be drawn somehow). This is accomplished by requiring any object that is drawn to the screen to inherit from a "Drawable" abstract base class, the main method of which is called `draw()`. Drawable is essentially an interface, but was implemented as an abstract base class so all Drawables can inherit certain properties and methods common to that type, such as the ability to have a "color filter" placed upon them that changes the hue of the object being drawn.

The renderer's duty, then, is to hold a collection of objects (each of which is a subclass of Drawable) and to call the `draw()` method on each when told to do so by the graphics core (which is itself told to do so by the engine). The renderer must also supply mechanisms for inserting Drawables into this collection and removing them later. Otherwise, there would never be anything to draw, as the collection would perpetually be empty.

Unfortunately, this particular approach becomes too constricting when dealing with 2D games because there are frequently specific objects that must be drawn on top of others. The background, for instance, is usually drawn below any other objects on the screen, so the developer would be forced to always add the background object to the rendering list before any other objects. To provide more flexibility, the screen, as seen by the player, is split into three regions, or panes, along the axis into the screen. Each pane is essentially independent of the others and contains its own rendering list. The three panes generally correspond to background objects, middleground objects, and foreground objects, and allow the developer to focus on one section of the screen at a time.

Regarding implementation, each pane is simply its own rendering list. So rather than having one that is used for all drawing, three are used. All of the items in the background list are rendered first, followed by all items in the middleground list, and finally all items in the foreground list. This causes foreground objects to always be displayed, regardless of anything in the background or middle ground panes. Similarly, items in the middle ground are always shown, provided there is not an item in the foreground at the same location. The background simply takes up what space was unfilled by either of the other two panes. All items within a single pane are parallel to one another, and so there is no preference as to which objects are rendered first. The user simply informs the engine which list the image should be put in.

3.3.2.2 Resource Managers

Apart from the renderer, there are three managers that are included as part of the graphics core. Each manager is responsible for allocating, storing, and retrieving the raw resources that are needed to draw an object on the screen, and there is one for each type of primary resource: fonts, shapes, and textures. Fonts are used to display written text on the screen, and shapes are important for primitive rendering objects like circles, rectangles, and points.

Textures are a slightly more abstract notion. A texture itself represents some image that will be displayed on the screen and is typically a bitmap. Textures can also, however, represent a *part* of an image that is to be displayed, and are then referred to as a "texture section". A common usage of texture sections is a typical sprite sheet, wherein frames of animation of an object are all stored in one picture. At any given point, only one animation frame should be displayed, so only a small piece of the source image should be displayed. The concept of animation in general was discussed previously in Section 2.1.3, and sprite sheets are discussed in more detail in Section 3.3.2.4.

There are many benefits to the management system described here. One of the most fundamental is that each resource is created only once, even if it is used multiple times when rendering. For example, a bitmap might represent the image of a single character, and multiple characters might be drawn to the screen at the same time. Rather than storing a separate copy of the bitmap for each character, one version is stored with the appropriate manager, and that version is used when drawing each character. This pooling scheme saves resources, as the item to be drawn only has to store a reference to the resource it needs, rather than the actual resource itself (which can be a substantial improvement when the resource is very large, like a bitmap image).

Another advantage is that of resource encapsulation. Since each manager holds all resources of a given type and the managers have been specifically designed to *not* provide access to the resources themselves, they must be responsible for understanding the mechanics behind actually rendering the item. In other words, the renderer knows only that a given Drawable can be rendered, and a Drawable knows only that one of the managers knows how to actually copy the resource pixels to the screen properly. The task is delegated entirely to the managers, as they are the only objects with direct access to the resources. It is not important how exactly that task is accomplished; only that the appropriate manager understands how to do it. Because of this, the user of the engine (and even most of the engine itself), does not have to bother with cumbersome details involved with properly displaying a given resource.

The architecture of each individual manager tends to follow the same design pattern. First and foremost, a manager has to envelop a collection of whichever resource it is charged with managing. It must also include a way of adding and removing items from this collection. Adding to the collection is usually done via a creation method, which takes in the information necessary to construct a single resource, performs the creation (if necessary), inserts the resource into the collection, and returns an index of some sort that uniquely identifies it so the resource can be used later. As mentioned before, if a creation operation is attempted when the resource already exists, the actual creation is skipped so multiple copies of that resource are not made.

Since resources cannot be accessed directly from outside the manager, deletion is performed by simply telling the manager which resource to remove via the index that was given when the object was created. In this manner, the encapsulation principle is not violated. A manager must also have the capability to draw resources of the type it handles. This is done with

some sort of draw method, that is given the identifier of the resource to be used and coordinates at which the object should be drawn.

The creation step is unique for each of the three managers. For the Texture Manager, all that is needed to create a given bitmap is a way of identifying that bitmap somehow, like a file name. In Android, images that are included as part of a project are assigned a unique resource identifier instead, so that identifier is what is passed into the creation method. To create a texture section rather than a complete texture, the texture identifier is needed (the value returned by the texture creation step) and a set of coordinates that represents which part of the source image will be drawn. Usually this is done using a rectangle structure, something that defines a rectangular region to be "cut out", as it were.

Fonts and shapes are created using a slightly different process. For either, the user must first fill out a structure containing information about that font or shape and pass that structure to the creation method. This is necessary because both have a large number of parameters that can be modified and it would be impractical to pass them all into a single creation method. The Font Info structure holds the font color, size, and alignment (left, center, or right). There is also a flag signaling that this font is to be bolded.

The Shape Info structure is slightly more complicated. There is a set of properties common to each shape, and there are several properties unique to certain shapes. Common properties include the color, style, and the stroke width. The style represents whether the shape is filled or not (that is, whether only an outline is drawn). The stroke width is used to define how "wide" the outline should be when only outlines are drawn. Also included is a dither flag, which will enable dithering on the shape, essentially smoothing the edges slightly.

The types of shapes currently available are circles, ovals, points (individual pixels), lines, rectangles, and rounded rectangles. Many have unique parameters, so to facilitate that, each has a specialized structure that inherits from the general Shape Info structure. Points and lines use no other parameters, but circles require a radius, ovals require a "bounding rectangle" that completely encloses the oval, rectangles require coordinates at which to be drawn, and rounded rectangles (which themselves inherit from standard rectangles) need individual radii in both the x and y directions to facilitate the rounded edges.

3.3.2.3 Core Drawables

At this point, we are ready to more closely examine the various Drawables that have already been created. To begin, we must decide what all Drawables have in common. This core functionality is placed in the abstract base class Drawable. As previously mentioned, each instance needs at a minimum a `draw()` method that is called by the renderer. Since each Drawable might do this in different ways, it should be defined as `abstract`, meaning the body of the function will be defined by some concrete subclass of Drawable. In addition, each Drawable needs to know where it should be drawn on the screen somehow. This is done by letting each hold its own coordinates, as well as its width and height. Finally, a given object on the screen might not necessarily need to be displayed all of the time. If a Drawable should be temporarily hidden, rather than removed from the rendering list completely, a "visible" property is needed. Only objects that claim to be visible will be rendered. Those who are not visible will be skipped completely in the rendering step.

Various Drawables have been created to serve different purposes in the engine. The three most crucial are directly related to the three resource managers described above (the Font Manager, the Shape Manager, and the Texture Manager), and are so called the Font Drawable,

the Shape Drawable, and the Texture Drawable. The Font Drawable is used to hold information about strings of text drawn to the screen, the Shape Drawable is used to hold information about shapes drawn to the screen, and the Texture Drawable holds information about images that are to be rendered.

Each of the three follows a similar architecture. They all inherit from the Drawable super class and implement the `draw()` method by asking the appropriate manager to handle the task. The Font Drawable asks the Font Manager to render strings, the Shape Drawable asks the Shape Manager to render a given shape, and the Texture Drawable asks the Texture Manager to render a certain image. Each tells the corresponding manager which resource is to be shown and at what location. Therefore each of the three primary Drawable sub classes must also contain the unique identifier of a resource that is to be shown. This identifier is returned by one of the managers when the resource is initially created.

3.3.2.4 Additional Drawables

These three primary sub classes can be used to render any combination of strings of text, shapes, and images, but they are inappropriate in certain circumstances. In the cases where using one of the fundamental Drawables would prove to be unnecessarily cumbersome to deal with, several other implementations have been written. Also, because each of the additional implementations derives from the same base class, Drawable, new functionality can be added by the user for a specific problem simply by writing another implementation that inherits from Drawable. It will inherently fit into the rendering pipeline just as easily as one of the predefined implementations. This contributes greatly to the modularity of the engine.

Certain situations are very common in 2D games and have custom Drawable implementations designed for their particular purpose. For example, the Drawable Group represents a collection of other Drawable objects. Although similar to the rendering lists used by the renderer, the Drawable Group is unique in that only *one* of the items in the list is displayed at a given time. One of the items in the collection must be "selected", and the selected item is the one that is displayed at runtime. This is most appropriate for situations when a cycle or marquee effect is desired in a game. Several Drawables are created and stored in the Drawable Group. After a certain amount of time, the selected item would be changed, and the next item in the cycle would be displayed.

Another situation that is very common deals with the representation of animations in a file. As discussed in the discussion on animation in Section 2.1.3, animations in 2D games are usually discrete samples of a fluid animation, rather than a continuous motion. To create the illusion of a walking character, for example, a series of static images must be produced that represent the position of the character at different points in time. When the images are displayed one after the other at a certain minimum rate of change, the human eye perceives motion.

In practice, storing a separate image for each frame of the animation tends to be a large waste of resources and adds significant complexity to the developer working with those images, especially if there are many of them. A single game might have thousands of these frames in a non-trivial project. A common solution is to combine the various frames (or at least some subset of them) into a single file by forming a grid, placing one frame in each cell. The author of the file is responsible for determining how many rows and columns should be in the file. This single image is then accessed by the program and one frame is rendered at a time, usually based on a

current row and column index, similar to the Drawable Group described above. Colloquially, this larger image is often referred to as a "sprite sheet".

To handle objects like this, the Sprite Sheet class was developed. It is very similar to the other Drawables discussed so far, but requires a slightly more complicated initialization. In order to draw a piece of a source image, a texture section is needed, as mentioned in the discussion of the Texture Manager above. To be able to index each frame in the sprite sheet, a texture section is needed for each of those frames. If they don't already exist, they need to be created when the Sprite Sheet is created. If the number of columns in the source file are known, as well as the width of the source file in pixels, the width of one cell can be found by dividing the total width by the number of columns. Similarly, the cell height can be found by dividing the total height by the number of rows. It is assumed that the grid in the input file is regular and that each cell is the same size.

Once the cell width and height are found, the texture sections can be created by iterating over each cell, assigning source boundaries based on the current index of iteration and the cell size. These source boundaries simply represent a rectangle (whose size is smaller than the original image) that tells the engine where to look for this particular texture section in the original image. The process is analogous to cutting a rectangular cake into smaller pieces, each of which has the same size. Each time a texture section is created, the identifier that is returned is saved in an array to be used later. By preprocessing the image in this way, changing the frame that is currently active is as simple as changing either the current row or the current column index. When the drawing operation is called, the texture section that has been requested is looked up in the array, using the current row and column as keys, and the Texture Manager performs the rendering using that identifier.

Apart from animation, the Sprite Sheet can also be used for other images that are stored in a grid-like format. For example, the collection of background objects shown on the screen can also be stored in a single file. Storing level elements like ladders, platforms, trees, and blocks that are all the same size in one file has the same benefits provided as when storing all the frames of an animation in one file described above. As a result, sprite sheets that store similar objects are also very common, though one could certainly store each object in its own file, which would be necessary if the various images were all different sizes.

This idea has proven to be the basis of what is called a "Tiling System". If all of the level information can be broken down into a set of core "parts" or building blocks, as is common with certain 2D game genres like platformers, it makes logical sense to divide the screen itself into a rectangular grid, placing a certain image in each cell. One can see that a partnership with a Sprite Sheet would be a very natural extension. The Sprite Sheet would represent a single image, divided into rows and columns, with a frame or single object in each cell. The tiling system could ask the Sprite Sheet for a given image at one of the cells on the screen that has been divided into a grid of its own. The Sprite Sheet does so, and the tiling system proceeds to repeat the operation for the next cell, likely with a different image. In this manner, a complete level can be constructed in a very uniform way.

To facilitate this particular operation, the Tiled Drawable exists. Though some of the details are slightly different, the core idea is the same. The Tiled Drawable divides its own area (which could be equal to the screen size or smaller) into a rectangular grid of a given size. The cells are then filled with identifiers that represent texture sections of the original image and the entire grid is submitted to be drawn when requested. Since the Tiled Drawable is itself a

Drawable, it can be stretched, moved, or hidden completely, which could prove useful to a game developer.

For the purpose of storage efficiency, however, each cell is not a Drawable of its own--just an identifier representing a unique texture section (though extending the design to do so would be very straightforward). Each cell can, however, be hidden individually. The need is actually a performance one. The average level map (a common use of this class) is equitable to a sparsely populated matrix or database. Usually, every cell does not contain an image that is important to the game play. The cell is not completely void because of the design, but it is considered logically empty. It would prove to be a waste of time to go through the rendering process for a cell that isn't even visible to the player. As such, each cell can declare itself as "invisible" so that it will be ignored by the renderer when the time comes to draw the object. Though the final result is unaffected, the performance can be impacted drastically, as seen in Section 4.2.

3.3.3 Input Core

Though significantly less intricate than the graphics core, the input core is no less important to the operation of the engine as a whole. As mentioned in its introduction in Section 3.2.3, the primary function of this core is to provide the engine with user input, which, since the target platform generally has a touch screen, is usually a touch event of some sort. Though not included as part of this design, support for other hardware devices on that platform like accelerometers and gyroscopes could easily be added by creating new classes that extend the Engine Core Component class and registering those new classes with the input core.

Of paramount interest in this topic is the singular input core component, the Touch Propagator. During the engine's initialization routine, it registers itself with the element of the underlying operating system that detects when the user has physically touched the screen of the device. For Android, the `onTouchEvent()` method of the View class is overwritten. The operating system creates an event and sends it to all objects that have requested that information, so the engine is aware of touch events almost as soon as they occur. The engine itself does not care about this event. It simply forwards it on to the input core, which then sends it to the Touch Propagator.

The Touch Propagator's first task is to synchronize the event. This is a very crucial task, as the underlying operating system sends the initial event asynchronously. The processing of touch events must be folded into the event loop, which runs in a single thread, to ensure consistent processing and to avoid synchronization issues that would be very difficult to track down for a developer. To accomplish this, a flag is set internally that says the screen was just touched. The location of the touch is also recorded, along with the "type", such as a down-press, movement, or up-press. During the next iteration of the game loop, the engine asks the Touch Propagator to process the touch event, and the flag is unset. This ensures each touch event is cached until it can be processed at the appropriate time, at which point it can safely be discarded.

The second task of the Touch Propagator is to maintain a collection of objects that desire to be informed when a touch event has, in fact, occurred. Objects can be added to and removed from this list at any time and identify themselves by implementing a "Touchable" interface. There are two methods in this interface. One of these is `getTouchBounds()`. This method allows the Touch Propagator to query the object for its current location on the screen. This is necessary because otherwise there would be no way to tell which object (or objects) are to be the

recipients of the touch message. The second method, `onTouch()`, is called when an object has been successfully identified as that recipient, and so must respond in some way.

Once these objects have registered themselves with the Touch Propagator, the propagator must wait until a touch event has been received. At that point, after the event has been synchronized, each item in the collection must be examined to see if the coordinates of the touch lie within the boundaries of that item. If so, the `onTouch()` method is called and the propagator continues its search. In this architecture, multiple objects can be touched at once. Regarding performance, since touch events are relatively rare (that is the number of frames processed that contain touch events is very small compared to the total number of frames), and the number of items in this collection on average is moderately small, a linear search like this contributes very little to the average run time.

3.3.4 Update Core

The update core is extremely similar to the input core in design. It consists of a singular component, the Updater whose function is analogous to that of the Touch Propagator in the input core. The Updater maintains a collection of objects that implement the "Updatable" interface, allowing them to be updated. This interface includes one primary method and one implicit property.

The method is of primary importance and is called `update(long dt)`. The parameter passed into the function, *dt*, represents the amount of time that has passed since the last frame was rendered, as described in Section 2.1.2, and should be included in any calculations that involve movement by the object being updated to ensure frame rate independence. The implicit property is a variable named `active` whose usage is exactly the same as that of

`visible` in the Drawable class. When this flag is set, the object will be updated. When it is not, the object will be skipped by the Updater during the update cycle. This allows a given object to ignore update requests, which is useful under certain circumstances, such as the pausing of a game.

Once per frame, the Updater checks the status of each object. If that object is active, meaning it is ready to be updated, its `update()` method is called and the Updater moves on to the next item in the collection. Any action at all can be taken during the update phase by a particular object, but usually game play is advanced in some form. An object will move in a certain direction according to its velocity or an animation will advance to the next frame. Collision detection and resolution schemes would be applied here as well. Generally speaking, most of the logic of the game is implemented in the `update()` method, which means the input core, simple as it is, is indeed a vital part of the engine's existence.

3.3.5 File Core

For all but the most trivial of projects, external files will be needed in some form or another. They hold high scores, objectives completed, and save states, among any number of other pieces of information. At some point in a project, their existence will be needed. Interfacing with the native operating system's file system is a task that can cleanly be abstracted away using the engine as an intermediary. The piece of the engine that accomplishes this task is the file core. The file core consists of two primary components, the File IO module and the XML Reader.

The File IO module primarily concerns itself with the reading and writing of simple text files, though it also includes methods to determine whether a certain file exists in the file system or not and to delete files. For the sake of convenience, each method in the class can either be called locally using an instance of the engine or statically by providing an Android activity to use as a gateway. Typically, the local versions of each method will be used because an instance of the engine is already available.

The other primary component of the file core is an XML Reader, provided for reading standard XML files. The XML Reader will create an instance of a parser that reads the file and extracts the important information. A simple tree data structure, XML Tree, is used to return the results to the user. This tree consists of a set of nodes, each of which has references to one or more child nodes. A single node contains an element name that corresponds to the tag name and a set of key-value pairs that correspond to the various XML attributes and their values.

Storing important information, such as levels or character information, as XML files not only provides a uniform way to access and modify key game elements, but also has the benefit of being portable across multiple platforms. For example, a level editor on a Windows machine could create or modify the XML files, which could then be read by the game during the initialization of that level.

3.4 Extensions to the Engine

Now that the core building blocks of the engine have been realized, those pieces can be used to build more sophisticated constructs than were previously possible. Two such assemblies are the "sprite" and the "particle system", both of which have their foundations in the various pieces of the engine described up to this point. They formally lie outside the domain of the engine in that they make use of its services, but are not categorized under any single core of

operation. Rather, they make use of the functionality provided by multiple cores working in harmony to achieve a greater goal than is possible by only exploiting a single part of the engine.

3.4.1 Sprite

Although the various cores can be used directly, properly initializing an object requires several steps. To display an item on the screen requires creating a Drawable appropriate to the task at hand, obtaining a reference to the Renderer, and adding that Drawable to the renderer's list. To set up an item for updating, that item must first implement the Updatable interface, a reference to the Updater must be obtained, and the item must be added to the Updater's list. A nearly identical process must be followed to allow an item to be touched. In games especially, at least two of these effects are usually desired, and generally all three will be used. For example, a character in a game must be drawn to the screen, move around (which requires updating), and could jump when touched. A button could be formed by drawing a static image to the screen and processing events when it is touched, but would not require updating.

As common as scenarios such as these are in game development, it would make sense to create a common starting point that handles the "bookkeeping" involved with displaying an object, updating it, and allowing it to process touch events. The "Sprite" class accomplishes this. Since a Drawable, Updatable, and Touchable object all must interact, the Sprite class inherits directly from Drawable and implements the interfaces given in Updatable and Touchable. Because of this relationship, the Sprite (or its internal Drawable) can be inserted into any part of the engine. The Drawable will be rendered, while the Sprite itself will be updated and can respond to touch events.

In order for the Sprite class to be functionally useful, it must be very flexible. Exploiting the object-oriented features of the host language, Java, simplifies this particular task greatly. For each of the three tasks, rendering, updating, and receiving touch events, the Sprite class has a default handling function. Users of the class need only "rewrite" those methods that are needed using the principles of inheritance. These three methods are `createDrawable()`, `update()`, and `onTouch()`, and the process for changing each is moderately simple.

While the Sprite class can manage the insertion of its Drawable into the rendering list, it must first know what to draw. This can vary for each Sprite, so it makes sense to let the user define the action taken. The `createDrawable()` function's task is to create a concrete Drawable object for the Sprite, thus taking care of that definition . The Drawable that is returned may be any of those mentioned in Sections 3.3.2.3 or 3.3.2.4, or it could be a custom implementation.

Each, however, likely needs its own set of construction parameters. For example, the creation of a Texture Drawable requires the resource identifier of the image to be used, while the creation of a Tiled Drawable requires a resource identifier *and* the number of rows and columns in the source image. As there is no way to account for all of the various possibilities involved with each construction, the `createDrawable()` method takes as a parameter an array of Objects (the ultimate super class of every class in the Java language). This array allows any number (and any type) of parameters to be given to the creation method. Usage, then, depends on which type of Drawable is being used. In the default implementation, a Texture Drawable is returned, so a single resource identifier must be passed into `createDrawable()` in the Object array.

This leads to an important side note about the Sprite class. In order to maintain the maximum flexibility, the object creation step has been split into two parts, creation and initialization. In the creation phase, the Sprite is allocated its memory, as is common with the creation of any other object. It is in the initialization phase, however, where internal properties are set, where the Drawable is created (via the `createDrawable()` method), and the Sprite is registered with the Renderer, the Updater, and the Touch Propagator. The primary advantage of this two stage method is that the sprite can be effectively reset by calling the initialization function again after a Sprite has already been initialized once.

The most apparent results of this are that each Sprite must call `init()` at some point after its creation or have it called for them. Because it might be inconvenient to force the developer to always include this initialization call after the Sprite has been created (especially if they have no use of the two-stage system), there exists a separate constructor that can be used to create and initialize the object at once. In this way, the developer can choose which approach they prefer.

The `init()` function needs four pieces of information to work properly. With only three exceptions, the width and height are a necessary part of each sprite. Circles, points, and text are drawn at the coordinates of the sprite, and so have no need of the two values. Any number can be given in these circumstances. Also necessary is the rendering list that this Sprite should be added to. As explained in section 3.3.2.1, there are three options, the foreground, middleground, or background. A Sprite can be set in any of the three, so the engine needs to know which. The final piece of information needed is the set of parameters for the `createDrawable()` function, (stored in an Object array). The `init()` method will call that function during its execution, so it will simply pass along the information given to it. In effect,

the user can either pass parameters to the `createDrawable()` function directly through the `init()` function or indirectly through the secondary constructor (that calls the `init()` function implicitly).

The two other methods of note are `update()` and `onTouch()`. The former allows the Sprite to change every frame, while the latter provides the user with a way to respond when this Sprite has been touched. Both are fairly straightforward. The default implementation of `update()` ensures the Drawable and the touch boundaries are kept in sync. As a result, overloaded versions of the method should take care to ensure the default implementation is called at some point before the method returns. Otherwise the sprite would never move. The default implementation of `onTouch()`, however, is completely empty, so there is no such requirement.

Enhancing the Sprite class for special needs, then, tends to be a very simple process. One simply overloads one or more of the base methods to change the functionality. A Text Sprite is included as an example. The `createDrawable()` method is updated so that it creates a Font Drawable using a given color, text alignment, and size. A fourth parameter, if provided, bolds the text. A pair of getters and setters are also included, which provide a simple way to modify and retrieve the text of the Text Sprite.

Special care has been made to explain how the Sprite class works because it is often the only class that the developer deals with directly. It is the backbone for the majority of projects, both large and small, and as such, understanding its operation is vital to the effective use of the engine as a whole. Furthermore, not only does it simplify many common tasks in game development, it is also a good example of how the various pieces of the engine can be used together to accomplish a common goal.

3.4.2 Particle System

Another engine extension has also been included--the particle system. The particle system is a set of classes built on the engine in the same way the Sprite class was, but its purpose is significantly different than that of the Sprite. A particle system is essentially a collection of shapes, but is unique in that the collection as a whole is the point of interest, rather than the individual particles themselves. Particle systems are frequently used to produce special effects, ranging from fire and smoke to streams like water, and as such tend to rely on a large number of construction parameters. The parameters can depend on the particle, like its color for instance, or they can be a property of the environment, like gravity. It is the combined effect of all of these that produce the net result seen on screen [18]. An example particle system representing a fire with smoke is shown below:

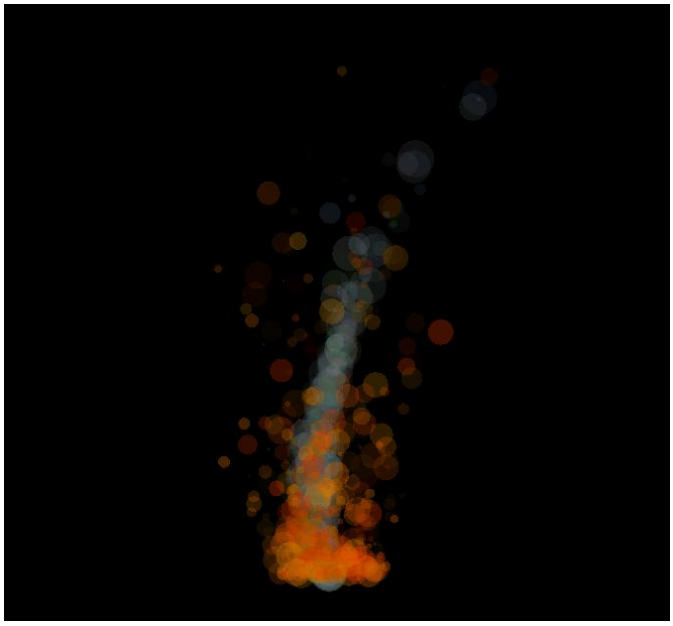


Figure 4: An Example Particle System

A particle system satisfies at least two of the same conditions that a Sprite does. It must be drawn to the screen and must be "updatable" (since individual particles usually move). As such, it is implemented in a similar fashion, inheriting from the abstract base class Drawable and implementing the methods found in the Updatable interface. (The sprite actually *contained* a Drawable, though, whereas a particle system *is* one. The difference is a subtle one.) Depending on the circumstances, a given particle system might or might not need the ability to be touched, so the standard implementation forgoes that particular functionality.

The class Particle System encompasses the functionality described so far. It also includes a collection of "particle emitters". An emitter is responsible for maintaining its own collection of particles--creating new ones when necessary, removing those that grown too old, moving those that are still active, and drawing the active ones to the screen every frame. Different effects can be achieved simply by modifying the particle creation method. Particles can originate from a single point, a line, or a shape by changing the particle's initial location. They can move in a focused direction or be scattered randomly by changing the particle's initial velocities. The possibilities are quite expansive.

A singular particle is defined by a number of factors. Each particle has its own location and velocity, color, fade rate, shape, size, lifetime, and age. The location gives the current position of the particle, while the velocity tells it which direction to move, both in the horizontal and vertical directions. The color of a particle can be static (one color is used throughout its lifetime) or dynamic (a start and an end color are given, and the particle interpolates between the two over the course of its lifetime). The fade rate determines how fast (if at all) a particle becomes transparent over the course of its lifetime and is independent of whether the color

chosen is static or dynamic. The shape of a particle can either be a circle or a square, where the size represents the radius of a circle (in pixels) or the length of one side for a square.

The final two parameters require more detailed explanation. Typically, a particle is only active for a certain amount of time. After this period of time has elapsed, the particle will "die", and will no longer be visible on the screen. The lifetime represents how long (in frames) the particle should be allowed to live at a maximum, while the age represents how many frames it has already been active. Other conditions can affect the lifespan of a single particle, though. If a particle moves off the screen, there is no point in maintaining it, so it will usually be killed. Similarly, if a particle becomes too transparent to actually see, it will be terminated. Particles that have completed their course are often reset with new parameters, essentially recycled, so they may be used again.

Because the phenomena that are usually modeled using particle systems are inherently very nondeterministic, values for particle parameters tend to be established using a "fuzzy" range system. The velocity would be a random number between 0.0 and some maximum value, for example. The color also might have a random amount of each of red, green, and blue. Although a minor point, efficient generation of these random numbers is important because it can contribute nontrivially to the average run time. The implementation given relies on the Java language's standard mechanisms for doing so.

4. RESULTS AND ANALYSIS

Three objectives were defined for this project initially. The engine developed had to be easy to use, modular, and as computationally efficient as possible. The first two criteria, however, tend to be very subjective and difficult to quantify. Instead of formally testing them, an argument will be presented as to why the engine as presented accomplishes those goals. The computational efficiency, though, can be quantified and will be demonstrated using a set of three test applications designed to "push" the engine in different ways. The primary measure for these tests will be the frames per second (fps) that is achieved on average under certain circumstances.

4.1 Methodology

A different set of criteria apply for each of the three primary goals listed above. Those criteria are explained in detail first, followed by the actual arguments that show how the engine meets those various conditions. A brief analysis is provided thereafter.

4.1.1 Ease of Use

In order to demonstrate ease of use, it must be shown that the engine can be created and initialized with a minimum amount of effort from the game developer. It also must be shown that simple tasks, like displaying an image on the screen, can be done in a few lines of code. (Brevity of code does not directly correspond with ease of use, but it is generally a good indicator. Shorter code generally takes less time to comprehend or process than longer items.) Finally, it must be exhibited that more complicated tasks, those that the engine does not support directly or natively, can be built using the fundamental building blocks in a relatively

straightforward manner--without unnecessary complications arising from the use of the engine itself.

4.1.2 Modularity

Modularity is established by the ability of another developer to add functionality to the engine without modifying the core components themselves, and applies to the engine itself (the ability to add new cores), an individual core (the ability to add new core components), and to applicable core components (the ability to process different types of objects than were initially accounted for). Demonstrating the engine's modularity, then, involves showing how each of these subtasks can be accomplished without changing any of the code that is responsible for handling the related task.

4.1.3 Efficiency

As mentioned in Section 2.1.2, a game should ideally run at least 25 frames per second for it to appear smooth. Larger values result in smoother game play, while smaller values will make the game appear jagged and unresponsive. Keeping this in mind, three tests were constructed to stress the engine in different ways. The first creates a certain number of random circles, giving each a random position, velocity and a size based on that velocity. Faster velocities result in smaller circles. The circles each continue until they hit the window boundaries, at which point they reverse direction. To gauge the performance of the engine, the number of circles is increased until the average frame rate drops below 25.

The second test is similar, but uses a particle system rather than a set of Sprites. A maximum cap is set on the number of particles shown on screen at any one time, and that cap is divided between two emitters, one that represents a fire and another that represents smoke. The

particle cap is raised until the frames rate drops below 25. Though the rendering times are roughly the same for the two experiments, the particle system requires more time updating each particle, since the conditions are more complicated.

The final test demonstrates shows a very common usage scenario involving the drawing of two separate level maps using a Tiled Drawable, described in detail in Section 3.3.2.4. The test shows what happens if the user neglects to inform the Drawable which indices in the map are invisible. If the Drawable is indeed told which indices are safe to skip, the frame rate will increase dramatically.

These three tests serve to show how different design decisions can impact the engine's performance, as well as how well the engine can withstand the kind of stress that will undoubtedly come as a part of normal game play.

4.2 Results

4.2.1 Ease of Use

Demonstrating the ease of use of the engine requires demonstrating that each of the goals mentioned in Section 4.1.1 can be achieved in a logical fashion. The first of these was that creation and initialization of the required a minimum amount of effort from the game developer. Provided the user follows the advice given in Section 3.3.1, in which the primary activity inherits either directly or indirectly from Game Activity (provided in the System Core), this requirement is already satisfied, as Game Activity creates an instance of the engine and initializes it as part of its own creation phase. As the developer has actually write no code to create or initialize the engine, then, the condition is satisfied.

The second condition required it to be possible to accomplish a simple task, such as displaying an image on the screen with only a few lines of code. This too can be accomplished with comparative ease, as seen in the following example:

```
1  public void onCreate(Bundle savedState)
2  {
3      Engine engine = getEngine();
4
5      int width         = 100;
6      int height        = 100;
7      int renderingList = Renderer.RENDERER_FOREGROUND;
8      int id            = R.drawable.example;
9
10     Sprite example =
11        new Sprite(engine, width,
12             height, renderingList, id);
13
14     engine.start();
15 }
```

Figure 5: Source Code Needed to Draw an Image to the Screen

The above is the complete contents of an Activity that simply draws an object to the screen. Note that the "work" is essentially one line of code, as the variables were declared solely for clarity and line 10 extends over the next two lines due to formatting. The sprite created will have a width and height of 100 pixels and will be drawn in the foreground (though the rendering list is irrelevant for a demonstration such as this). Line 8 saves the resource identifier of an image in the project folder called example.png, and shows how Android typically handles images. The sprite will be placed at location (0, 0) by default (the top-left corner of the screen). Another position can be set by adding the following line after lines 10 - 12:

```
13  example.x = 200;
```

This will change the sprite such that it is drawn at (200, 0), rather than (0, 0). More complicated position changes can be included by overriding the `update()` method of the Sprite class. As this example shows, simple tasks are indeed relatively easy to implement.

The final condition asks that it be possible to implement functionality that is not native to the engine in a straightforward manner, without a multitude of engine-specific complications. An example of this is the entire particle system, as described in Section 3.4.2. The system lies outside the engine, but exists because it is able to be inserted into the workflow. It can be rendered because it inherits from Drawable and so may be inserted into the Renderer's rendering list. It is able to be updated because it implements the Updatable interface, thus allowing it to be inserted into the Updater's list. The drawing just requires the implementation of a custom `draw()` method, while the updating requires an `update()` method. The overhead involved with interfacing with the engine is very small compared to the work actually done by the particle engine, and although the position is arguable, implementing new functionality in the engine is indeed very straightforward.

4.2.2 Modularity

As with the "Ease of Use" requirement, there were three tasks that were deemed necessary to show the engine to be modular. The first of these was that it was possible to add another engine core without actually modifying the Engine class. The following example shows how that might be done with a mythical "Sound Core" class, assumed to inherit from Engine Core, just as all other engine cores do.

```
1  public class ExpandedEngine extends Engine
2  {
3      private SoundCore sound;
4
5      public ExpandedEngine(GameActivity parent, int
6          screenW, int screenH)
7      {
8          super(parent, screenW, screenH);
9      }
10
11     public void init()
12     {
13         super.init();
14
15         sound = new SoundCore(this);
16         sound.init();
17     }
18
19     public void destroy()
20     {
21         super.destroy();
22
23         if (sound != null) sound.destroy();
24         sound = null;
25     }
26
27     public SoundCore getSoundCore() {return sound;}
28 }
```

Figure 6: Engine Modularity Example

As seen in the above example, adding a new core to the engine can be done by creating a subclass that overrides two of the essential methods of the Engine class and adding a getter to retrieve the new sound core. Although this particular method is not as elegant a solution as used in the latter two cases, it certainly does accomplish the initial goal--that the engine itself can be modified without changing the core Engine class. (It is for the sake of efficiency that this method is taken. Under normal circumstances, the engine framework itself would not be modified very often.)

While the ability to add a single core is important, more so is the ability to add core components to one of those cores. This usage scenario is much more common and can actually be achieved without requiring the developer to create a subclass of the affected core if desired. If a new component has already been created (done by inheriting from the Engine Core Component class), a new core component can be added by simply calling the `addComponent()` method of the affected core with two parameters. The first is a unique integer constant identifying that component, and the second is an instance of the new component itself. This can be done at any point during run time, allowing the developer to modify the engine while it is running if the need arises.

The final condition for modularity involves the integration of custom components into engine core components that are applicable. The Renderer, for example, is one of the most important components of the graphics core. The Renderer is not very selective about objects admitted to one of its rendering lists--the object must only declare itself as Drawable. Though many standard Drawable implementations are provided, new ones can be created by the user at any time. Inserting them into the appropriate rendering list requires exactly the same process as inserting any of the predefined Drawables into a rendering list.

In a similar fashion, any item that declares itself to be Updatable can be inserted into the Updater's list, and any item that declares itself to be Touchable can be inserted into the Touch Propagator's list. The most important components are very flexible in their operation and contribute greatly to the overall modularity of the engine as a whole.

4.2.3 Efficiency

To gauge the efficiency of the engine, we must first find a baseline value against which others may be gauged. In order for this number to make any sense, however, it must have units associated with it. As mentioned previously, a common metric for games is the frames per second (or fps). The frames per second is often used because it generally corresponds very well to the amount of work being done by the processor at any given point. When the processor has a lot to do, the fps will typically drop, and vice versa. Higher values for the fps are better in terms of quality because a smoother image results.

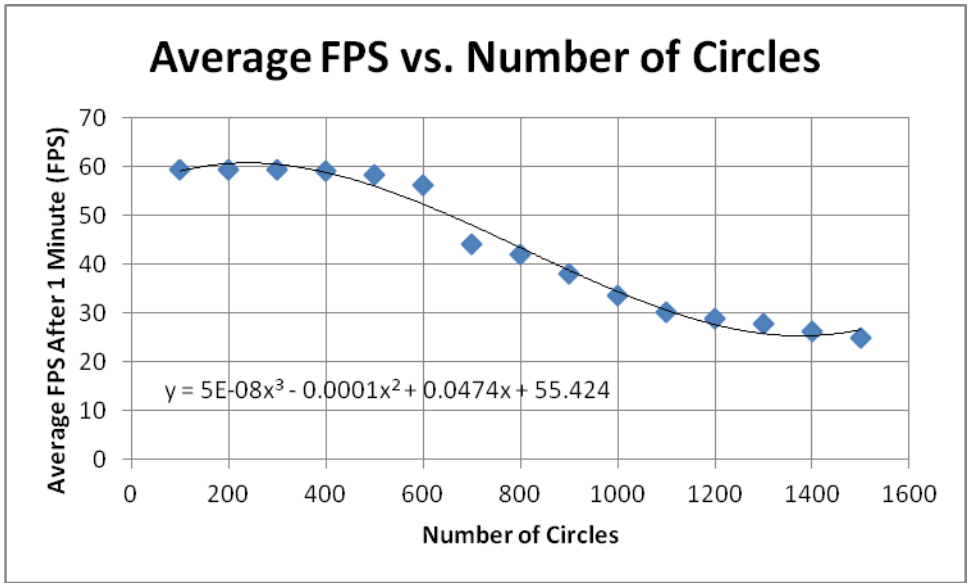
To obtain the baseline fps value, the engine must be run in an isolated environment (a control group). To do so, all experiments are performed on the same Samsung Galaxy Nexus device, containing a 1.2 GHz dual-core processor and 1 GB of RAM [7]. An extremely simple "pass though" application was designed that did nothing but initialize the engine and start it. No sprites were created, and nothing was added to the Renderer, the Updater, or the Touch Propagator. The application was allowed to start and sit until the fps stabilized. Doing so yielded the baseline value of **59.6 fps**. As this value is considerably more than the 25 fps required to perceive motion, the engine does well initially.

The first test was run a number of times with different numbers of circles. After one minute, the average fps was recorded. The results were placed in Figure 7 below. A graph representation follows thereafter. As apparent by the data, the engine, as currently implemented is capable of rendering nearly 1500 simultaneous objects before slowdown becomes drastic enough to warrant calling the game "unplayable" by our current definition. An interesting side note is that 600 such objects can rendered simultaneously with only a 5% slowdown from the baseline state. Another feature to notice is the drastic performance drop between 600 and 700

objects on the screen. While this could be random error, it more likely corresponds to a point where the operating system had to do more work than normal, such as where the RAM had filled and had to be swapped into the permanent memory, for example. Regardless of the reason, it gives the resulting graph a distinctly "cubic" look.

Figure 7: Test 1 Results

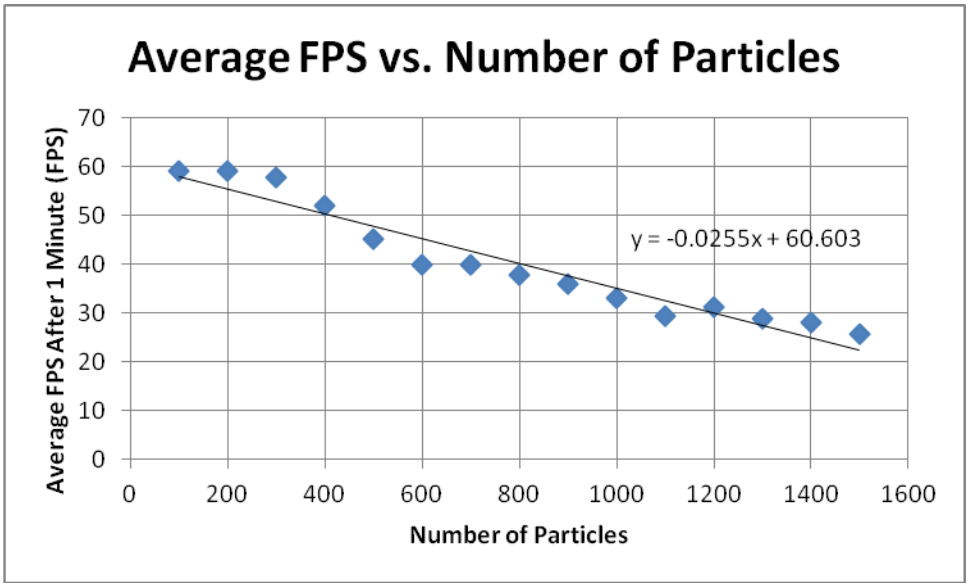
Number of Circles	Average FPS (After 1 Minute)
100	59.5
200	59.5
300	59.4
400	59.2
500	58.4
600	56.3
700	44.2
800	42.0
900	38.1
1000	33.5
1100	30.1
1200	28.9
1300	27.7
1400	26.2
1500	24.9



Test 2 was designed to test a similar situation, with the exception that the circles were replaced with a particle system representing fire and smoke. Individual particles in the particle system had a more costly update routine than the simple circles, so they are probably a better approximation of objects that would be generated by a real game. The results, however, are very similar to the first test. The frame rate only dropped to unplayable levels after about 1500 simultaneous particles were rendered on screen at any one point. The graph has a distinctly more "stair step" shape to it, but the correlation is essentially linear in nature.

Figure 8: Test 2 Results

Number of Particles	Average FPS (After 1 Minute)
100	59.2
200	59.1
300	57.7
400	52.1
500	45.3
600	39.9
700	40.0
800	37.7
900	36.0
1000	33.1
1100	28.9
1200	31.1
1300	28.9
1400	28.1
1500	25.8



The final test was constructed to show how expensive displaying two typical tiled level maps were under two sets of circumstances. Typically, drawing a level map is a moderately expensive operation by itself because the drawing time is proportional to the number of rows times the number of columns in the result--for a 20 x 12 map, 240 separate drawing calls have to be made in the worst case, compared to just one for a single sprite. Minimizing this number, then, is very important. Looking at a typical level map, like that shown in Figure 9 below, we can observe that most of the map is empty space. The grid contains an invisible cell in most locations. (In this particular example, only 26 of the 240 cells actually have a visible image in them.)

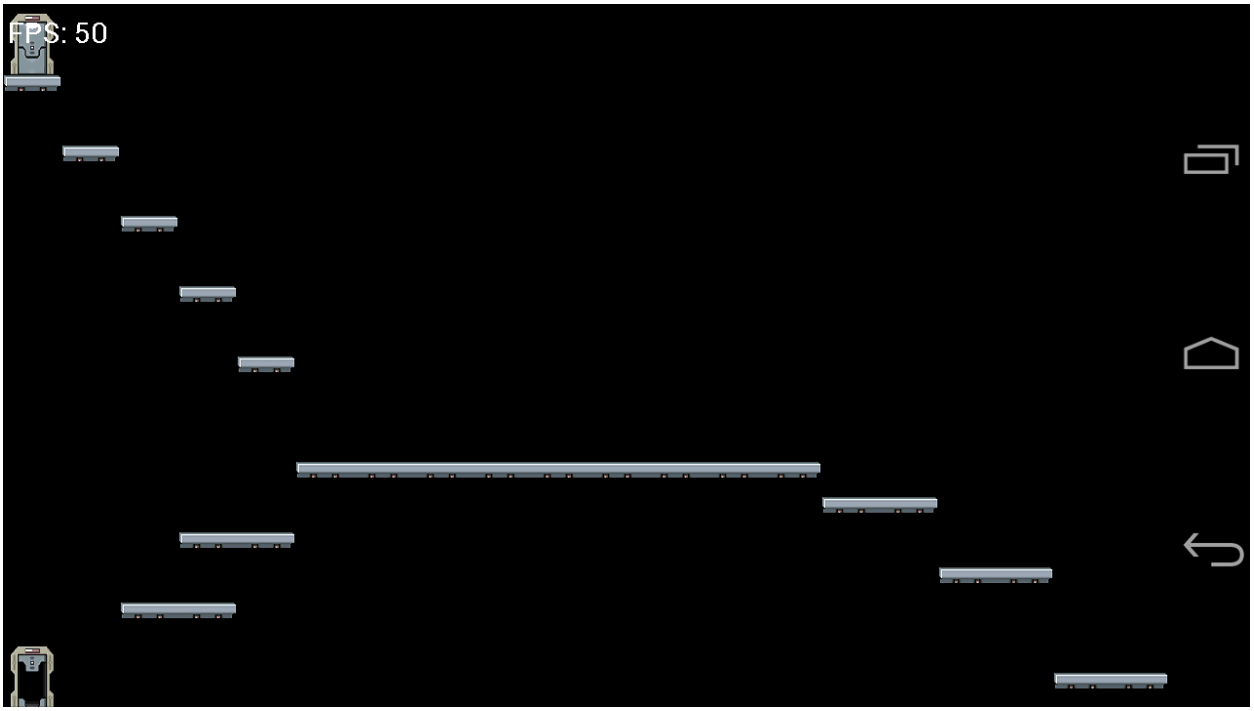


Figure 9: A Typical Level Map

We can save a substantial amount of time by simply ignoring those cells that aren't going to be displayed. There are two options to accomplish this. The engine could preprocess each image that comes along, looking for any cell where the alpha channel is 0 for each pixel in that cell, but that would also waste a bit of time. A more effective solution would be to simply ask the user which cell values are supposed to be invisible. Since the user has to tell the engine which image is in each cell anyway, they likely already know which cells should be ignored. This also has the benefit of letting the user "hide" cells selectively (even if they are not empty), which could be a useful game play mechanic.

Giving the user the option to "inform" the engine can have serious performance repercussions, however. In test 3, two level maps were displayed in the traditional manner and the fps was measured for each after one minute, as in the previous two tests. The maps differed in that the second was more populated than the first. There were fewer empty cells that could be

skipped. In the first case, the invisible indices were set--the engine was "informed" about which parts of the map to skip. In the second case, the line that accomplishes this was removed. The fps was measured for both cases and is shown in Figure 10 below.

Figure 10: Test 3 Results

Map	Invisible Indices Set	Invisible Indices Not Set	Percent Drop
1	59.0	30.7	47.96%
2	45.7	30.0	34.35%

The results are quite drastic. Omitting the one line that informed the engine about the invisible indices resulted in nearly a 50% performance drop in the case of map 1. Map 2, being less sparse was not affected as much, but the drop was still about 35%. In the case where the invisible indices were not set, the worst case scenario always results. Each of the cells in the image will always be drawn, which can drag the performance down miserably. Including those invisible indices provides such a performance increase it would be foolhardy to ignore it.

4.3 Analysis

The arguments presented above show the engine to be both easy to use (at least in some fashion) and modular, which would benefit greatly if the project were to be expanded. The API is not perfect, however, and could be further updated to enhance both of those features. The performance data shows the engine to be capable of running very fluidly under normal circumstances, only slowing under 50 fps for 400-500 objects on what would probably be considered a "standard" Android smart phone. Many more could likely be used, but care would have to be taken to ensure that major slowdowns would not occur on smaller (or slower) devices. More testing would be needed to be sure.

5. CONCLUSIONS

5.1 Summary

In this thesis, a two-dimensional game development engine was proposed for the Android mobile platform that facilitates rapid development of those games by individual developers or hobbyists. The essential elements of game design were presented in order to introduce the reader to concepts that were crucial for comprehension of the paper. Three primary design goals were identified: ease of use, modularity, and computational efficiency, and were chosen to aid the user of the system. A prototype solution was then described in detail that included a hierarchical structure composed of engine cores and engine core components . The prototype was then evaluated against those design goals presented to see how well it accomplished each task. The results were collected and presented in the last chapter and showed that the proposed design was indeed able to meet the objectives provided. It would be a viable option for a single game developer wishing to quickly and easily construct games on the Android platform.

5.2 Potential Impact

This engine would likely be of use only to its intended audience. Large corporations or game design studios with many more resources to devote to a single project are not likely to benefit as much from this design as the individual developer who has few resources. Time and knowledge, especially, are serious bottlenecks for a solo developer, as a large task that would normally be broken up into several smaller pieces can only be done by a single person.

Especially for students who are new to the field of game design, time can be roughly split into two halves--the time taken to learn how to accomplish a certain goal, and the time required to actually complete it. Those with more experience can eventually shorten the learning time,

but development time can only be shortened so much before a fundamental barrier is reached. Passing that barrier requires a new approach--that a shortcut of some sort be taken. The engine accomplishes this task.

The engine is capable of reducing both the learning time and the development time, if used properly. As such, its potential impact is very widespread. It could be used to teach the fundamental concepts of game design to a beginner or to allow construction of a rapid prototype for a developer. Most importantly, however, it provides a blank canvas. One whose contents are bound only by the imagination of the designer. To provide a means for the expression of those ideas is the broadest impact of this engine.

5.3 Future Work

The engine as presented aims to simplify the task of 2D game development for an individual developer, hobbyist, or student wishing to learn about game design. As such, there are several areas in which the engine could be expanded further. Integrating a sound core, allowing more hardware integration through the use of other sensors on the device, and expanded graphical capabilities are all ways the project could be improved.

The primary focus of the engine was to simplify the rendering of objects to the screen. It was not specifically designed to be a complete game development solution like many commercial engines, so a sound module was left out in the design, though the inherent modularity of the engine would provide a straightforward method of integrating one, as referenced by Figure 6 in Section 4.2.2. Another engine core could be developed to facilitate the playback of sound effects and music and would indeed be a logical advancement of the current technology, as most commercial games do make use of it in some form or another.

Another area for improvement would expand the capabilities of the current input core by providing access to the phone's accelerometer, compass, gyroscope, GPS, light sensor, or camera, where available. Using these sensors as an addition to the current touch-based input would serve to give the developer a richer set of options to make use of when creating games and could spur a form of creativity that is currently unavailable through the engine alone. The accelerometer and gyroscope, in particular, could prove to be viable input options.

There are also a host of options that could be added to enhance the engine's capabilities graphically. For example, while the engine does support scaling and translation of items shown on the screen, rotations are not currently available. Color filters can be applied to objects, changing their overall hue, but more advanced lighting methods could also be useful. More complicated Drawables could also be created for specific tasks, like a scrolling tile system for platformer-style games.

Another idea would be to provide some options for device-independent rendering--that is, an image on one device would display in a similar manner on another without any changes to the application logic. Currently, that task falls to the game developer, because there are multiple ways of implementing it. Not all Android devices have the same aspect ratio, unfortunately. Depending on the individual game, it might be appropriate to assume all devices have some standard dimensions and scale everything drawn appropriately, but this approach could have the unintended side effect of stretching images that should not be stretched. Another approach would be to scale everything so the correct aspect ratio is always maintained, introducing black bars around the sides when necessary. This, however, might not be ideal for smaller devices, where screen real estate is already at a premium. Giving the developer choices like these within the engine itself could be a beneficial addition.

REFERENCES

[1] J. Harbour, Beginning Game Programming, Third Edition, Course Technology PTR, Boston, M.A., 2010.

[2] R. Gonzalez, R. Woods, Digital Image Processing, Third Edition, Pearson Prentice Hall, Upper Saddle River, N.J., 2008.

[3] N. Gramlich, "AndEngine Home", http://www.andengine.org/blog/, referenced: November 2, 2012.

[4] "LibDGX Goals and Features", http://libgdx.badlogicgames.com/features.html, referenced: November 3, 2012.

[5] "LibDGX About", http://www.badlogicgames.com/wordpress/?page_id=2, referenced: November 3, 2012.

[6] "Corona SDK", http://www.coronalabs.com/products/corona-sdk/, referenced: November 3, 2012.

[7] M. Joire, "Galaxy Nexus HSPA+ Review", http://www.engadget.com/2011/11/24/galaxy-nexus-hspa-review/, referenced: November 12, 2012.

[8] J. Pepitone, "Android Races Past Apple in Smartphone Market Share", http://money.cnn.com/2012/08/08/technology/smartphone-market-share/index.html, referenced October 5, 2012.

[9] I. Lunden, "ComScore: US Smartphone Penetration 47% in Q2; Android Remains Most Popular, But Apple's Growing Faster", http:// http://techcrunch.com/2012/08/01/comscore-us-smartphone-penetration-47-in-q2-android-remains-most-popular-but-apples-growing-faster/, referenced October 5, 2012.

[10] "ComScore Reports April 2012 U.S. Mobile Subscriber Market Share", http://www.comscore.com/Insights/Press_Releases/2012/6/comScore_Reports_April_2012_U.S._Mobile_Subscriber_Market_Share, referenced October 5, 2012.

[11] L. Whitney, "Games Reign as Most Popular Mobile Apps", http://news.cnet.com/8301-10797_3-20077213-235/games-reign-as-most-popular-mobile-apps/, referenced October 5, 2012.

[12] J. DiPane, "Who Says You Can't Make Money on Android Development?", http://www.androidcentral.com/who-says-you-cant-make-money-android-development, referenced October 6, 2012.

[13] "Samsung 19300 Galaxy S III", http://www.gsmarena.com/samsung_i9300_galaxy_s_iii-4238.php, referenced October 14, 2012.

[14] "Android, the World's Most Popular Mobile Platform", http://developer.android.com/about/index.html, referenced October 15, 2012.

[15] J. Gregory, "Updating Game Objects in Real Time", http://www.gamasutra.com/view/feature/132587/book_excerpt_game_engine_.php?print=1, referenced October 7, 2012.

[16] "Android: Event Driven Programming", http://independentlyemployed.co.uk/2010/12/03/android-event-driven-programming/, referenced October 2, 2012.

[17] "Activity Lifecycle", http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle, referenced November 5, 2012.

[18] J. van der Burg, "Building an Advanced Particle System", http://www.gamasutra.com/view/feature/3157/building_an_advanced_particle_.php, November 1, 2012.