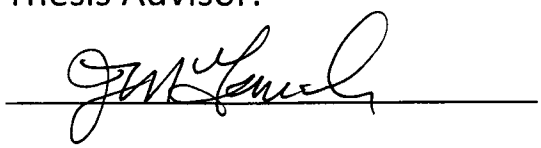
5-2014

The Design and Implementation of a Lightweight Game Engine for the iPhone Platform

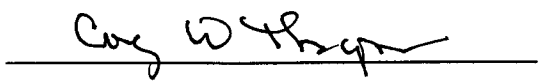
Luke B. Godfrey
University of Arkansas, Fayetteville

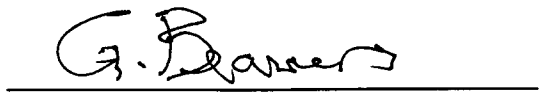
This thesis is approved.

Thesis Advisor:



Thesis Committee:





The Design and Implementation of a Lightweight Game Engine for the iPhone Platform

An Undergraduate Honors Thesis

in the

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

by

Luke B. Godfrey

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Problem

iPhone games have a large market and an even greater market potential, but development for iPhone games is bottlenecked because many projects do not have a good starting point. In terms of development, existing solutions are neither as lightweight nor as flexible as they could be. Furthermore, these solutions do not provide a specific framework for developers to use when building a game on their engines. Although many comprehensive game engines exist for the iPhone platform, the approach to implementing games using these engines is left entirely to the developer. As a result, developers can have a difficult time determining where to begin, and are often required to design their own architecture from scratch for each game, greatly increasing development time.

In earlier years, software distribution required physical disks, which was very slow, and limited the audience that developers — especially independent developers — were able to reach. Today, however, software can be distributed digitally. The distribution of applications designed for the iPhone platform is handled by Apple's App Store. By using a service like the App Store, developers have instant access to millions of potential customers. The "holdup" is no longer in the distribution, but in the development of software. If faster methods for developing games for the App Store were available, developers would be able to better realize this massive opportunity.

Game development for mobile devices comes with a unique set of constraints and challenges. The processors and memory in a smartphone are generally less powerful than their desktop computer counterparts. Battery life is not considered when

developing for a PC or a gaming console, but it is crucial to minimize battery use in programs designed for mobile devices. On the other hand, mobile development also offers a number of benefits. Sensors and touch screens offer new ways to interact with a game. Furthermore, the user base for mobile applications is very large, especially compared to the user base for gaming consoles. In 2013, nearly 56% of American adults owned a smartphone [1].

What is needed, then, is a tool that developers can use to minimize development time. The tool should take into consideration the unique issues faced in mobile application development. A lightweight game engine is just the tool that is needed.

1.2 Objective

The objective of this project is to create a flexible framework for designing two-dimensional games on the iPhone platform and to provide a foundational engine on which these games can be built.

1.3 Approach

To meet the objective, this engine will be designed and evaluated according to three criteria. First, it must be easy to use. If using this engine takes as much time to learn and use as starting from scratch, it has failed to be a good foundation for creating games. Second, the engine must be flexible and modular. Distinct sections of the engine should not be tightly coupled; instead, they should be designed so that the developer can add, remove, or change one section of the engine with little-to-no modifications to other sections. Finally, it must be lightweight, making efficient use of the device's resources such as memory and CPU.

The first objective, ease of use, must be met to make this engine an effective solution. Specifically, it should be easy to build a game on top of the engine's core systems and framework. Because there are already other engines on the iPhone platform, this engine must offer something unique so as not to simply duplicate existing work. Specifically, it should offer a structure that streamlines game development so that developers can produces games more quickly.

In order to ensure the flexibility of the engine's framework, this project makes use of a game design approach called the entity component system (ECS) architecture. The design is inherently modular, so adding, removing, or modifying specific portions of the engine is streamlined. It also keeps the classes small and easy to maintain, rather than having a handful of monolithic classes. The ECS model will be described in more detail in chapter 2.

Although ease of use and flexibility are important, the lightweight aspect of the engine is the most visible part to end users, and is therefore particularly crucial. Inefficient memory usage can cause applications to crash, and inefficient CPU usage can slow down the performance of the game and even of the device itself. Designing the engine to be efficient in these two areas allows games built on it to run more smoothly, consistently, and stably. This "smoothness" can be measured in terms of frames per second, which will be used to measure the efficiency of this engine in chapter 4. In addition to producing a smooth frame rate, an engine that minimizes CPU usage also minimizes battery usage.

As it is designed for the iPhone platform, the engine is coded entirely in Objective-C and makes use of Apple's Cocoa and Cocoa Touch API for data structures

and basic rendering. Designing the engine in the iPhone's native language ensures that it will run smoothly on devices using the iOS platform, but it also eliminates any reasonable way of porting the engine to other platforms. At the end of this thesis will be a discussion of possible future work, including the design of a portable game engine.

1.4 Organization of This Thesis

Key concepts and useful background information is presented in chapter 2. This includes a brief overview of game programming, a description of the ECS model, and some information about the iOS software development kit. The chapter also presents several existing solutions and discusses the pros and cons of each in terms of the design goals of this project.

Chapter 3 describes the engine's architecture. In the first half of the chapter, a high-level design is presented, followed by the design and implementation of a framework that adheres to the high-level design. The second half of the chapter discusses specific implementations of various systems built on the framework and demonstrates the suggested use of the framework.

Chapter 4 evaluates the engine's implementation based on the objective and the criteria set forth in section 1.3. Ease of use and flexibility are demonstrated by the development of a simple game using the engine, and efficiency is tested quantifiably. The methodology is described, followed by the results and an analysis of the results that explores potential improvements.

The final chapter, chapter 5, draws conclusions from the project. A brief summary of the project is presented along with a description of the potential impact this project could have on game developers and on the general public. Finally, future work is

discussed, including improvements to the engine implemented in this project and the possibility of redesigning the engine to be portable to a number of platforms, beyond only iOS devices.

2. BACKGROUND

2.1 Key Concepts

This thesis has been written with the assumption that the reader has a basic knowledge of digital games and programming concepts. This section will present some additional background information that will enable the reader to understand key concepts in game development that specifically pertain to this project.

2.2.1 Early Game Development

When video games first reached popularity in titles such as Pong, Asteroids, Space Invaders, and Pac-Man, every game was meticulously developed from scratch. The software designed for these early games was tightly linked to the hardware for which it was programmed [2]. Software components in these programs were not particularly reusable; at best, the code for one game could only make extremely similar games.

The concept of game engines arose in the 1990's when id Software released the game *Doom* [3]. The *Doom* software was split into two segments: the core systems and the game assets and logic. This careful separation made it easy to reuse the core systems, like rendering and collisions, in other games. As video games grew in popularity, the reusability of software became more important for the rapid development of new games.

Today, almost all game development is concerned with game engines. A number of game engines are widely available — many of them are even available free of charge. Even games that are designed from scratch often make use of third-party libraries of reusable components. Developers that elect not to use existing engines

often design custom engines for their own projects. Specific approaches to the development of game engines will be explored in section 2.1.3.

2.1.2 The Game Loop

At the center of any game program is the game loop. The game loop handles user input, advances gameplay elements, and renders the game world as often as is appropriate. A single update to the gameplay elements is commonly called a tick or a step. Each time the game is rendered is a frame, and the time between two frames is known as the delta time. The basic approach when building a game is to design game objects to update in discrete steps. For example, the player game object may move three pixels along the x-axis each tick.

Simply updating as quickly as possible can cause problems. The delta time for each frame is not always consistent, so if game objects are configured to move in discrete, consistent steps, the game will appear choppy. There are two primary approaches to implementing the game loop in such a way as to fix this issue. First, one may use the delta time to update the game world in partial steps. This is the time-based method. Second, one may set a fixed delta time. By limiting rendering to a specific time interval, game elements can be stepped discretely without the choppiness. This is the frame-based method.

The time-based method has several benefits. Using a variable delta time allows both rendering and physics to update as fast as the device can possibly process it. This method is based on the idea that the time between any two frames is not going to be consistent, so adjustments need to be made so that game time matches time in the real world. Out of the two options, this one has the best potential for making the game feel

smooth, as it can generate more frames on faster devices. The time-based method also ensures that the game runs at the same pace, regardless of the speed of the device on which it is running. A slow device, for example, might need to step 1.5 times per frame compared to normal device; a fast device might step only 0.7 or 0.8 times per frame.

The time-based method also has a few drawbacks. Allowing any value of delta time can result in strange behavior. Consider a fast-moving game object approaching a wall. If the game is running on a slow device, or if rendering hangs up, the delta time will be bigger. If it is big enough, applying delta time can cause the fast-moving object to pass right through the wall. It never actually collided with the wall, it just moved right past it. This issue is known as tunneling, and can be resolved by limiting delta time or implementing a sweeping collision detection algorithm that takes extra steps to determine intermediate game states between two frames [4].

Setting a fixed delta time (the frame-based method) is not prone to the same problems if the physics are carefully planned. Terminal velocity, for example, can prevent an object from flying through a sufficiently large wall. Because every step is guaranteed to be the same size, systems of the game are more stable. Other problems arise, however. Game-time no longer matches real-time, and runs faster or slower depending on the speed of the device on which it is running. Spikes in CPU usage can cause the game to slow down, because steps are tied to frames. This close coupling of steps and frames is the frame-based method.

The iPhone operating system (iOS) has locked screen rendering to 60 times per second, and has a convenient `CADisplayLink` method for hooking the game loop directly into the screen rendering cycle. Because this platform has a fixed configuration,

it is safe to adopt the frame-based approach [5]. Delta time is “close enough” that the difference between frames should be imperceptible, especially considering that this engine is designed for 2D games that should not be complex enough to slow down the processor.

2.1.3 Game Engines

A game engine is a set of classes and program components, used in a game, that do not include game logic or data specific to a particular game [6]. These components are designed to be reusable, so that many games can be built with on a single, underlying framework. The motivation for game engines is that by separating basic tasks performed by the engine and logic specific to any one game, development time for games is drastically reduced.

There are a number of approaches that can be taken when developing a game engine. One common approach, the object-oriented approach, starts with a base game object class from which everything in the game world derives. This base class would include properties and methods common to every object. For example, the characters, items, enemies, and buildings would all derive from the game object class, and would all inherit basic properties such as position on screen and an image to display. This approach often uses a hierarchy to group game objects that require the same properties. Characters and enemies, for example, might have the same inner workings, so they might both inherit from a creature class that contains properties like `velocity` and methods like `move` and `jump`. The creature class in this example would inherit from the base game object class. As the number of game object kinds increases, the

hierarchy becomes more complex. Figure 1 shows a simple object-oriented game object hierarchy.

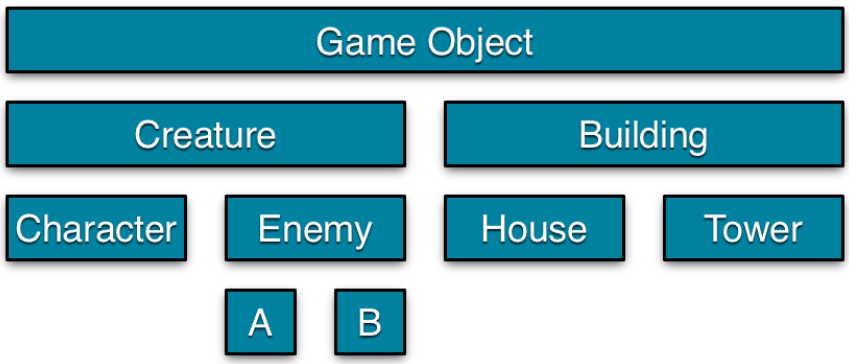


Figure 1: An example of an object-oriented game object hierarchy.

For projects where most of the game objects share most of the properties, this approach to game engine development is sufficient. However, relying on inheritance in this way has two primary drawbacks. First, the hierarchy is not flexible. Consider a character class that has methods for ranged attacks, such as an archer. In order to create a tower class (as a derivative of “building” — see figure 1) that can also have ranged attacks, the code from the character must either be duplicated or moved to the root game object class (so that buildings and creatures can access the shared properties and methods for ranged attacks). Second, because of the inflexibility of the hierarchy, the base game object tends to become monolithic in nature. In other words, the base game object in an object-oriented engine becomes “long and convoluted” by including code that only pertains to certain game objects but not to others [7]. In the example given, the base game object would include properties and methods for ranged attacks, but game objects that do not need ranged attacks would still inherit them as unnecessary properties and methods.

In contrast to the object-oriented approach, the approach used in this project will favor composition over inheritance. Rather than inherit properties from a hierarchy, game objects will be made up of modular units of property groups. The properties needed for ranged attacks, for example, would be grouped together as one such “unit” and added to the game entities that require that functionality. These “units” are called components, and the approach used in this engine is referred to as the entity component system model, which will be examined in the following section.

2.1.4 Entity Component System

The entity component system model (ECS) stands out as having a flexible and extensible architecture that avoids the pitfalls of the common object-oriented approach to game engine development. By adopting a database-like approach, the ECS model effectively decouples logic from data and facilitates modular development. The ECS architecture is made up of three sections: components, entities, and systems.

The first part of the ECS model is the component. Components are collections of related properties. A “physical” component, for example, would contain a number of properties common to all physical objects, such as velocity, mass, elasticity, friction, and so on.

Some component-based approaches package methods and game logic with each component [7]. In these approaches, each component has its own turn to update in the game loop. However, this requires a component to be able to update other components. The physical component, for example, would update the position component based on the velocity property. This produces tightly-coupled components; in the example, tightly-coupled means that every time an entity has a physical

component, it must also have a position component. This approach to component-based game engines results in an architecture that is not as flexible as it could be and leaves it to the developer to remember which components depend on which. Therefore, in order to retain complete modularity, components must not contain any methods or logic. It should be possible to add a component to an entity — or remove a component from an entity — without affecting other components.

The second part of the ECS model is the entity. An entity is a game object with no properties or logic. Instead, it is an aggregate made up of components. After dividing up groups of related data into components, entities can be constructed by picking and choosing relevant components like building blocks. In example, a player entity could be made up of a physical component, a movement component, a ranged attack component, and an input component. A tower entity could be made up of a physical component and a ranged attack component. This demonstrates the advantage of the ECS model; unlike in the object-oriented approach, where every entity would have been forced to inherit an unnecessary ranged attack component, the ECS approach allows each entity to be made up of only the components it needs.

Some implementations of the ECS model allow a few global properties in the entity. This is more of a hybrid approach that assumes that every entity will share at least some global properties and methods. Conversely, a pure ECS approach has all of an entity's properties contained in components. One possible implementation of an ECS-based game engine has no entity class, but assigns an ID to instances of components and represents entities as the list of all components with matching IDs.

The final part of the ECS model is the system. Because neither entities nor components contain logic, this third aspect of the engine is required. A system is a collection of methods that act on one specific kind of entity. As an example, a physical system might be responsible for moving entities with a physical component and a position component. Systems select entities based on the types of components they have been assigned. Each system is updated during the game loop.

The ECS architecture was chosen for this project because, by design, it meets the flexibility requirement for this project. After establishing a basic framework for adding entities and systems to the game, specific systems can be added, removed, or modified without breaking the functionality of other systems. Entities are also modularly constructed, as components can be added to a single entity without adding unnecessary properties to every entity.

2.1.5 iPhone SDK

iOS is the operating system developed by Apple, Inc., and used by iPhones, iPads, and iPod touches. Apple has provided the iOS software development kit (SDK) for developers to create applications to run on the iOS platform. Development is most easily done in Xcode, Apple's IDE.

Objective-C is a strict superset of C, so it can be used at a very low level. The Cocoa framework, included with the iOS SDK, adds lots of high-level functionality. Using Objective-C with the Cocoa framework gives the developer lots of control and flexibility, with the ease of high-level functionality with the power of a low-level language. This thesis uses "Cocoa," "Apple's framework," and "the iPhone SDK" as interchangeable terms.

The Foundation framework, included with Cocoa, provides a number of useful classes, like the `NSString` class for string literals, and several convenient data structures. The primary Cocoa-provided classes that will be used in this project are `UIView`, `NSMutableArray`, `NSMutableDictionary`, `CGPoint`, `CGSize`, and `CGRect`. The `UIView` class defines a rectangular area to be drawn on the screen, and can be used to capture touch screen events. The engine will use this class to handle rendering and player input. An `NSMutableArray` is a resizable array that facilitates the storage of an indeterminate number of objects, and (in the context of this project) can be used to store pointers to entities in each system. An `NSMutableDictionary` is like a hash table that maps keys to objects. In this project, dictionaries will be used to store components in an entity by type, so that systems can access a component based on component type. A `CGPoint` is a C struct of two `double` values to define x and y coordinates (or some other x and y pair). A `CGSize` is also a C struct of two `double` values, but it defines width and height instead of x and y coordinates. Finally, a `CGRect` is a struct that defines a rectangle as a `CGPoint` origin and a `CGSize` size.

The Core Animation framework, also included with Cocoa, contains the `CADisplayLink` class. This class allows a program to hook an update method into the device's screen refresh rate. In terms of a game engine, this class provides an entry point for the game loop. By adding a display link that targets the base run loop for the engine, the run loop is called approximately 60 times per second.

2.2 Related Work

A number of other game engines already exist for the mobile platforms. In this section, a few of these solutions will be presented and discussed. Each of these

alternatives will be compared to the objectives of this project and contrasted with the engine developed for this project.

2.2.1 Unity

Unity is a comprehensive game development engine originally designed for 3D games. It is a powerful tool that supports iPhone and Android among seven other distinct platforms as of the writing of this thesis. In November 2013, Unity 4.3 was released with a new toolkit for developing 2D games [8]. There is a free version of Unity for companies or individuals that make less than $100,000 annually.

Unity is a good choice for developing games. It is well documented and fully featured. However, it is not open source, so it is not possible to modify or add features to the engine itself. Therefore, although it is a good solution, it is not as flexible as some alternatives.

2.2.2 Cocos2d for iPhone

Cocos2d is an open-source solution. It is a framework originally developed in the Python programming language, with several branches to work with other platforms, including an Objective-C branch for the iPhone platform. The basic game object in Cocos2d is the sprite, and much of the framework is designed for the use of sprites, sprite sheets, and animations. Cocos2d supports the Box2d physics engine that can be used for physics and collisions.

Cocos2d provides a solid framework for displaying images and transitioning between game scenes, but leaves the implementation of game objects and various systems to the developer. It does not provide a specific structure or model for adding

new functionality. Theoretically, an ECS model could be built on top of the Cocos2d engine.

2.2.3 Android Engine by Jon Hammer

The Android game engine developed by Jon Hammer was also designed to be easy to use, modular, and efficient [9]. Hammer's approach was similar to the approach used in this project, although it used a different architecture. Specifically, his engine used the notion of engine "cores" that hook into the game loop.

Hammer's engine is not a true alternative, as it is designed for another platform. However, his work on the subject of mobile game development is related and worth comparing to the engine designed in this thesis. The primary contrast to be made between the two engines is that Hammer's engine follows a more traditional approach that makes extensive use of object-oriented programming and inheritance, while the engine designed in this thesis favors composition over inheritance.

3. ARCHITECTURE

3.1 High Level Design

The game engine designed for this project adheres to the ECS architectural model and includes additional functionality commonly required for games. The key element in this engine that is not defined in the ECS model is the scene, which represents a game state. A game is assumed to be composed of a number of scenes, with exactly one scene active at any moment. Structurally, the engine core is at the root of the engine's hierarchy. The engine core class contains a scene. The scene class contains an array of systems and a global array of entities. The system class contains a "local" array of entities (a subset of the active scene's entity array). Finally, the entity class contains an array of components. See figure 2 for a visual representation of the engine's hierarchical structure.

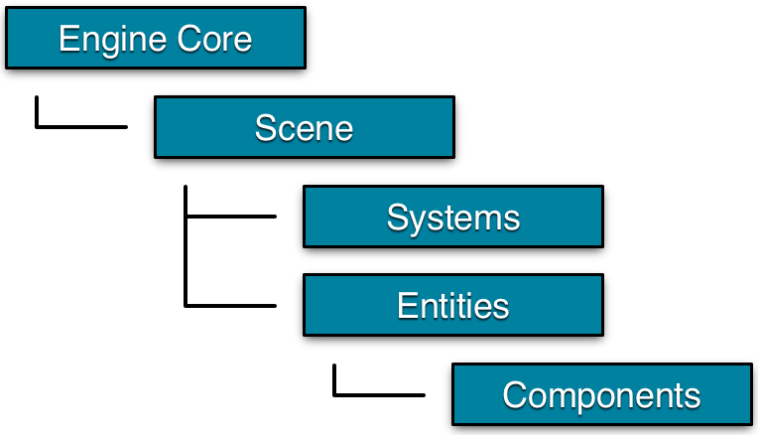


Figure 2: The engine hierarchy used in this project.

To initialize the engine, an instance of the engine core will be created. A subclass of the scene class will be created and set as the engine core's active scene. In the scene subclass's constructor, the scene will instantiate systems add them to the its

systems array. Finally, the scene will also create entities (with the appropriate components) and add them to its entities array.

In a tick of the game loop, the engine core will propagate the loop to the active scene, and the scene will propagate the loop to its systems. The scene will also distribute entities to the appropriate systems. Each system has its own update loop, which will contain logic for updating its subset of the scene's entity list.

The engine is divided into two parts: the framework and the core systems and components required for a simple game. The framework provides an entry point for using the ECS architecture, and includes the engine core, an entity class, and abstractions for scenes, systems, and components. The rest of the engine includes specific implementations of various systems and their associated components that are generally required in most games, such as rendering, physics, and collisions.

3.2 Framework

In this section, the framework portion of the engine will be examined more closely. The framework can be broken down into five key parts: the engine core, scenes, systems, entities, and components. The following subsections will present the design and implementations of each of these parts.

3.2.1 Engine Core

The engine core is the starting point for all game functions. It is primarily concerned with keeping track of the game's active scene and running the game loop. No game logic is included in the core; instead, it continues to run the game loop until it receives instructions to change the active scene.

There are three properties in the engine core class. The first property is a pointer to the active scene. The second property is a `UIView` (which gets the UI prefix because it is from Apple's UIKit framework). This will be the empty canvas for rendering the game. Finally, the engine core also contains a reference to the device's display link, to which we will hook the game loop.

The engine core's initialization method accepts a `UIView` as a parameter and stores it as the game canvas, then automatically populates the display link. The only other public method in the engine core is the `gotoScene:` method, which deactivates the currently active scene, releasing it from memory, and activates a new scene. Finally, it starts the game loop if the loop is not already running. The only action taken in the game loop at the engine core's level is to call the update method of the active scene.

3.2.2 Scenes

Scenes are not a standard piece of the ECS architecture, but there is appropriate motivation for including them in a game engine. Every game is made up of game states. For example, consider a game that starts with a menu state. When a user provides input to start the game by tapping the start button, the game transitions to the playing state. The playing state remains active until the game ends because the player lost, beat the game, or chose to quit from a pause menu. At this point, the game transitions back to the initial menu state. The motivation for scenes, then, is that because games can be broken down into states, the framework behind a game should facilitate that modularity.

The scene is the first part of the engine that can be subclassed, and is where a developer can start implementing game logic. A subclassed scene will override the

initialization method to register applicable systems and add entities to itself. Conditions for scene transitions are defined in the scene, and, when such conditions are met, the scene notifies the engine core that a transition is needed and passes a new instance of the next scene to the engine core's `gotoScene:` method.

The interface for scenes is simple. It is a subclass of the iPhone SDK's `UIViewController`, which streamlines the use of events triggered by user input, for example, a tap, swipe, or other gesture. Because a scene is made up of systems and entities, the only other properties included in the basic scene class are two `NSMutableArray` objects to store these groups of objects.

There are only four methods in the scene interface. The initialization method is first, which accepts a pointer to the engine core and adds its `UIView` as a subview to the game canvas. Second, there is a method for adding systems to the scene. This method is careful to order systems appropriately in the array so that during the game loop, they are executed in the appropriate order. Third, there is a method for adding entities to the scene. In this method, the scene stores pointers to each added entity and distributes them to the appropriate systems. It is for this reason that entities must be added to the scene after the systems have been added. Finally, the scene has an update method, which does nothing more than propagate the game loop to each of its systems.

In order to utilize the scene interface, the developer should subclass it and override an initialization method, most likely the `viewDidAppear:` method inherited from the `UIViewController` class. The initialization should be made up of two phases: a phase for adding systems and a phase for adding entities. Once the scene

has been activated or entities have been added to it, no additional systems may be added. Entities, however, may be added or removed as needed throughout the scene's lifespan.

3.2.3 Systems

In the ECS architecture, systems are where all of the game logic is implemented. Each system contains a list of entities in which it is interested, methods for hooking into the scene, and logic that controls a specific part of the game. These entity lists are not necessarily disjoint; multiple systems may be responsible for a single entity. By putting all of the logic into systems rather than directly inside of entities or components, systems facilitate data-driven programming by completely separating data from functionality. This eliminates the tendency to have tightly coupled components, and encourages efficiency by having a single update loop for an entire group of entities, rather than an update loop for each entity.

Like scenes, systems are abstractions to be subclassed. The system interface includes three properties: a pointer to the scene, an array of entities, and an integer value that represents its order in the update loop. When a system is added to the scene, the scene inserts it into the update list based on this last property, putting systems with low values closer to the beginning of the update loop and systems with high values closer to the end. This ensures that systems update in an appropriate order, so that, for example, rendering occurs after collisions have been resolved.

There are several methods in the system interface, most of which are event-driven. The first method accepts an entity and returns a Boolean value indicating whether the system is interested in that entity. This is the only method that is not event

driven. The scene uses this method to distribute entities to the correct systems as they are added to the scene. A rendering system, for example, only returns true when the entity passed to it contains a renderable component.

The system interface also includes an initialization method. The `initWithScene:` method accepts a pointer to the scene and stores it, then allocates memory for the entities array and sets the default update loop priority. Finally, it calls an overridable `initialize` method that provides a convenient hook for initializing system properties prior to the first update loop.

Also included in the system abstraction are two empty methods, `touchDown` and `touchUp`. These methods are triggered by the scene class when the player taps the device's screen. The scene simply passes the event along to the systems, when these two methods are implemented.

The final method of the system interface is the update method. All or most of the game logic in a system is hooked into this method, which is called by the scene with each iteration of the game loop. The basic usage of the update method is to loop through each stored entity and perform some action on it. The rendering system, for example, should loop through each entity and render it to the screen. Other systems might not go through each object one at a time, and still other systems may contain only one or two entities, as will be demonstrated in section 3.3.

3.2.4 Entities

Everything in the game world is an entity. The terrain, characters, and other interactive game elements are entities. Even non-diegetic elements, such as menus, heads-up displays, music, or a virtual camera, are considered entities. In this engine,

entities contain neither game logic nor properties. Instead, an entity is defined as an aggregate of components, with its properties distributed to components and game logic delegated to systems.

The entity class contains an indexed list of components, so that querying an entity to determine if it has a specific type of component — and obtaining a reference to that component — is relatively inexpensive. In order to accomplish this, the component list is stored as an `NSMutableDictionary`. The dictionary works like a hash table, mapping keys to component objects; by inserting components at a key that depends on the component type, lookup of components based on type is trivialized. By inserting arrays of components for these keys, the entity class can have multiple components of the same type.

Three methods are provided in the entity class. The first method adds a component to the components dictionary, and creates a new array at the component's key if one does not already exist. The next method turns the dictionary entry for a key, that is, an array of components given a component type. The last method accepts a list of component types and returns a Boolean value indicating whether the entity contains components of all of those types. As entities are added to the scene, systems use this last method to determine whether they will add a particular entity to its local list.

3.2.5 Components

Components are lists of related properties that belong to an entity. They should be designed as "plain old data" classes that contain no logic. The only methods in a component should be getters, setters, or convenience methods for getting and setting property values.

Storing an entity's attributes in modular components has many benefits. Distributing data into groups of related properties circumvents the possibility of creating monolithic entity classes; a specific entity need not contain any irrelevant or unused properties. Using components also gives entities implicit properties; along with the data explicitly defined in the component, the very fact that an entity has a specific type of component is useful. For example, no explicit property is necessary to determine whether an entity can collide with the environment — this information can be inferred by checking whether an entity has transform and collider components.

The basic component class, designed to be subclassed, has one class method and one instance method. The class method returns a string that indicates its type — this is the key used in the entity's component dictionary, and defaults to the class name. The instance method is an empty initialization method that can be overridden.

Most component subclasses will only contain a list of properties with their data types, overriding the initialization method to set property defaults. Subclassed components can be subclassed again to provide special cases of components — a collider component, for example, might be subclassed for different collision shapes. When a specific component is subclassed, it may be beneficial to override the class method to return the root component type rather than the specific type.

3.3 Core Systems and Components

In addition to the ECS framework described above, the engine has a few core systems that provide the basic functionality required for most games, as well as the components required for each of those systems. This section will present the designs of

systems implemented as part of the engine, as well as details on how they are implemented.

3.3.1 Rendering System

Rendering is one of the most fundamental portions of a game engine. The rendering system has the responsibility of drawing all drawable entities to the game canvas. As rendering is one of the most computationally expensive pieces of the engine, it is imperative that the system responsible for it is designed and implemented as efficiently as possible. In this engine, the rendering system has the task of adding renderable entities to the game canvas and updating each one's placement each cycle of the game loop. It is important that the rendering system is updated as the very last system in the game loop, so that the player does not see an intermediate game state — for example, a state in which the player has moved inside a wall and collisions have not yet been resolved. Therefore, the rendering system should have the highest update order so that it goes last in the update loop.

The rendering system for this engine uses Apple's `UIKit` framework. This framework is built on the iPhone SDK's Core Graphics which uses Quartz (rather than Open GL) as its basic drawing technology. `UIKit` was chosen instead of another technology to reduce the engine's development time, as it provides much of the common functionality required for rendering, such as redrawing. Unfortunately, using `UIKit` makes the engine less portable and less efficient than a custom implementation using Open GL. However, it meets the requirements of this project.

There are actually two rendering systems included in this engine. The first is the basic renderer, in charge of everything that can be drawn to the screen. The second is a

subclass of the basic renderer, the sprite renderer, which is responsible for drawing textures and animating sprites.

3.3.1.1 Basic Rendering System

In order for an entity to be considered renderable, it must have two components. First, it must have a `Transform` component. This component has a single property: a `CGPoint` that indicates the Cartesian coordinates of the entity. The transform component is not unique to the rendering system, and will be used by most other systems as well. Second, the entity must have a `Render` component. This component is only used in the rendering system. The most important property is a `UIView`; this is what the system will add as a subview to the game canvas. The render component also includes a `CGPoint` offset value that determines where the view is placed relative to the entity's transform position. Finally, the render component contains a z-order integer. This property determines which elements should be rendered on top and quick elements should be rendered in the back. For example, the background has a low z-order so that it stays behind everything and the player has a high z-order so that it stays on top of most things. In figure 3, the background has a z-order of 1, the tree has a z-order of 2, and the player has a z-order of 3.

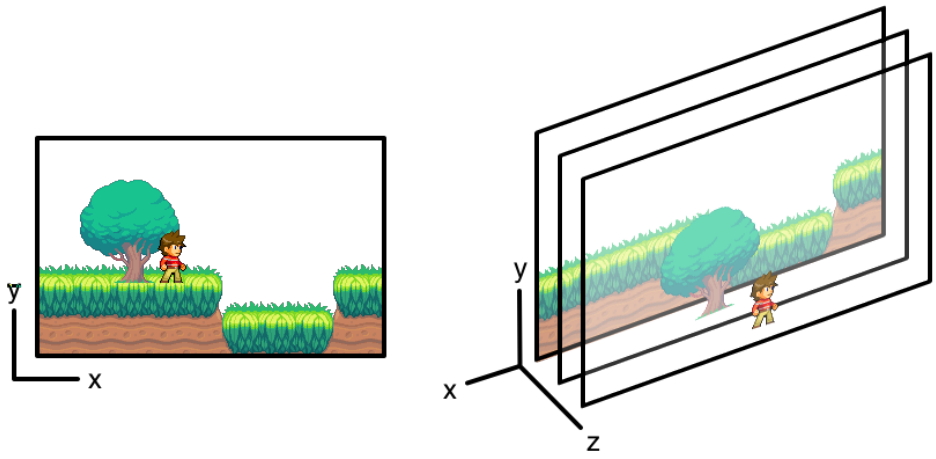


Figure 3: Demonstration of the z-order property [10].

By using `UIView` objects, the implementation of the rendering system was greatly simplified. Each time an entity is added, the rendering system adds the entity's render component's view as a subview to the active scene's view. The system's update loop has a nested loop that goes through each entity and adjusts its render's view frame, setting the x and y coordinates based on the transform's position and the render's offset.

3.3.1.2 Sprite Rendering System

The sprite rendering system is a subclass of the basic rendering system and makes use of the parent class's update loop. The sprite system introduces a new kind of render component, a subclass of render called the `Sprite` component. The sprite component is used for drawing textures, that is, images. The sprite class also has a resource management system to cache textures for reuse, so that sprites that use the same image resource share the texture and thus minimize memory usage.

Sprites have several properties. First, it has a `UIImage` sprite sheet, that is, an image loaded from a file composed of several sub-images. Each sub-image is a frame

of the sprite sheet, and for any sprite, exactly one frame is displayed at a time. In many implementations of sprites, an atlas is used to map frame keys to specific frames in the sprite sheet. In this implementation, however, the sprite sheet must have frames of equal size, so that frames are easily indexed by an integer using modular arithmetic. Figure 4 shows part of a character sprite sheet that make up a “run” animation.

Figure 4: The run animation in a character sprite sheet [11]

Sprites have three `UIView` objects. The first is the root view from the parent render class. The second is the sprite view. The sprite view clips its contents to its bounds, and its size is exactly the size of a single frame of the sprite sheet. The final view is the image view that loads in the sprite sheet texture and is offset negatively based on the sprite’s current frame, so that the origin of the sprite has the correct frame displayed.

Sprites also have states, which are stored in a dictionary. States are simply pairs of start and end frames with a Boolean value indicating whether the state is loopable. During the update method of the sprite rendering system, the sprite will go to the next frame in the sprite’s current state, so that a sprite can be animated by switching between frames each time the update loop is processed. By storing states in a dictionary, it is easy to reference a specific state — the player’s sprite, for example, would probably have “walk,” “fall,” and “idle” states.

The last four properties of the sprite class are primitive types. There is an integer representing the current frame of the sprite, two integers used as an animation counter

and delay value to determine how often the sprite frame should change, and a Boolean value indicating whether a non-loopable state has completed its animation. This Boolean value can be used to prevent the sprite from changing in the middle of a specific animation. The animation counter is incremented each tick of the update loop, but the sprite frame is only changed once the animation counter reaches the animation delay value, at which point the animation counter is reset.

The sprite interface has two class methods and four instance methods. The first class method returns an `NSMutableDictionary` of cached textures, stored as `UIImage` objects. By storing this as a class method, cached textures are shared across every sprite in the game. The other class method is a factory method that returns a copy of a sprite.

The four instance methods are convenience setters to be called from the sprite class. The first method increments the animation counter and resets it when it reaches the animation delay value. Second is a method that makes the sprite display the next frame of the sprite sheet; if the next frame is outside the current state's frame range, a loopable sprite goes back to the first frame and a non-loopable sprite stays at the last frame. The third method is used for adding a sprite state. The fourth and final method is used for setting the sprite sheet name, which is the file name of the sprite sheet to be loaded; this method checks for a cached texture with the same name, returning it if it exists and creating and caching it if it does not exist.

In the sprite system's update method is a for loop that iterates through each entity with a sprite and a transform component. Each sprite is cast as a render component and processed through the basic rendering system's update, then, if the

sprite is visible and the sprite state contains more than one frame, its animation counter is incremented and, if the counter was reset because the counter reached the animation delay value, the sprite's next position method is called to display the next frame.

3.3.2 Physical System

The physical system introduces physics to the game, and is another fundamental portion of any game engine. It is responsible for the movement of and the application of forces to entities. Environment variables, such as gravity and terminal velocity, are contained in the physical system. The physical system is the primary means of entity movement, and is therefore the primary cause of collisions, so it will update first in the game loop.

Entities distributed to the physical system must have a `Transform` and a `Physics` component. The physics component adds three physical properties to an entity. The main property is a `CGPoint` that stores an entity's velocity along the `x` and `y` axes. This component also stores a double value for the entity's mass and a Boolean value indicating whether the entity responds to gravity. The last property is required for entities that have velocity but do not respond to external forces; for example, a moving platform should ignore gravity completely.

The implemented physical system is extremely basic. In the initialization method, default values are set for gravity and terminal velocity. The update method then loops through each entity. The system applies gravity to each entity that responds to gravity by updating its velocity. Next, each entity's velocity is limited to the value of terminal velocity. Finally, the system adjusts the entity's transform by the amount of the entity's velocity.

It is important to note that the physics system is concerned with the movement of individual entities, rather than with the interactions of multiple entities. Every entity is moved in the physical system's update method, regardless of whether an entity's movement causes overlap between one or more other entities. Overlap between two entities is commonly called a collision, and the detection and resolution of collisions is handled by a separate system — the collision system — which will be described in section 3.3.5.

3.3.3 Camera System

In a simple game, the entire game world may be able to fit on the device's screen at one time. However, many games — especially platformers — have worlds that are much larger than can be displayed. The camera system provides a means to select what portion of the game should be displayed. The camera system should be updated only after all entities (which includes a camera entity) have been moved to valid positions, so it should be included in the game loop just before the rendering system.

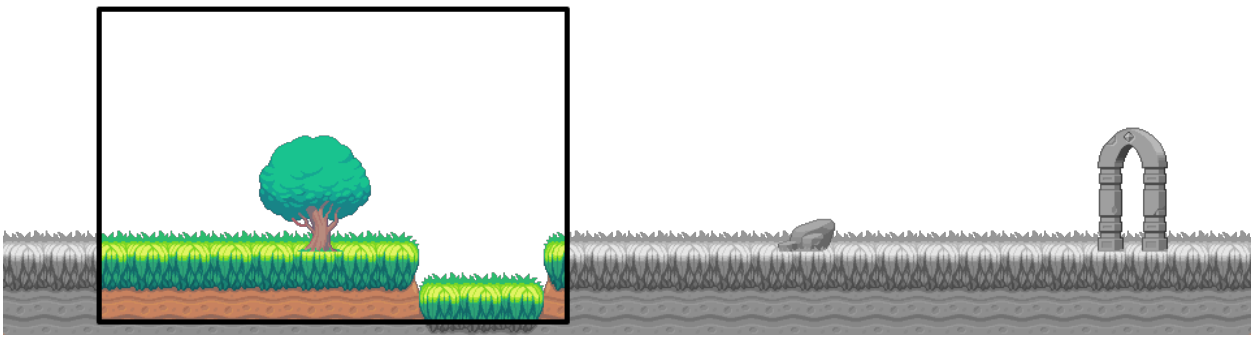


Figure 5: Using a "camera" to display only part of the game world [12]

Unlike the rendering and physical systems, the camera system modifies the entire scene rather than individual entities. Only one entity is added to the camera

system: the camera entity. The camera entity has two components: transform and a camera component. The camera component has a `CGPoint` offset, the position of the camera's origin relative to the entity's position. It also has a `CGRect` that describes the camera's boundaries, which is usually the game world's boundaries. In figure 5, the colorized portion of the game world in the camera rectangle is the part of the game world displayed on the device, and the gray portion outside the rectangle is the part of the game world that the camera can display if it moves.

The update method of the camera system simply adjusts the frame of the scene's view. The camera entity is inside the game world, but the camera's origin — its transform's position adjusted by the offset — should be at the top-left corner of the device's screen when rendering. To accomplish this, the scene's view origin is set to the negative of the camera's origin. Finally, to ensure that the camera is in a valid position, the scene's view origin is forced inside the camera's boundary rectangle using `MIN` and `MAX`.

The suggested use of the camera system is to attach a camera component to the player entity. As the player moves through the game world, then, the camera will follow along. Alternatively, the camera component could be attached to a dedicated camera entity that moves on its own, perhaps at a constant rate.

3.3.4 Tile System

A tile system is commonly utilized in two-dimensional games. The tile system splits the world into a grid. Each cell of the grid, called a tile, is an image that represents a small part of the game world. These tiles are designed to be reused and repeated, so it is natural to use a sprite sheet, as described in section 3.3.1.2, to store the graphics

for all of the tiles. Each tile, then, is a sprite that uses a “master” sprite sheet that is shared between all tiles. A two-dimensional array of tiles makes up a tile layer, and an array of tile layers makes up a tile map. Figure 6 shows a game world as a grid of tiles.

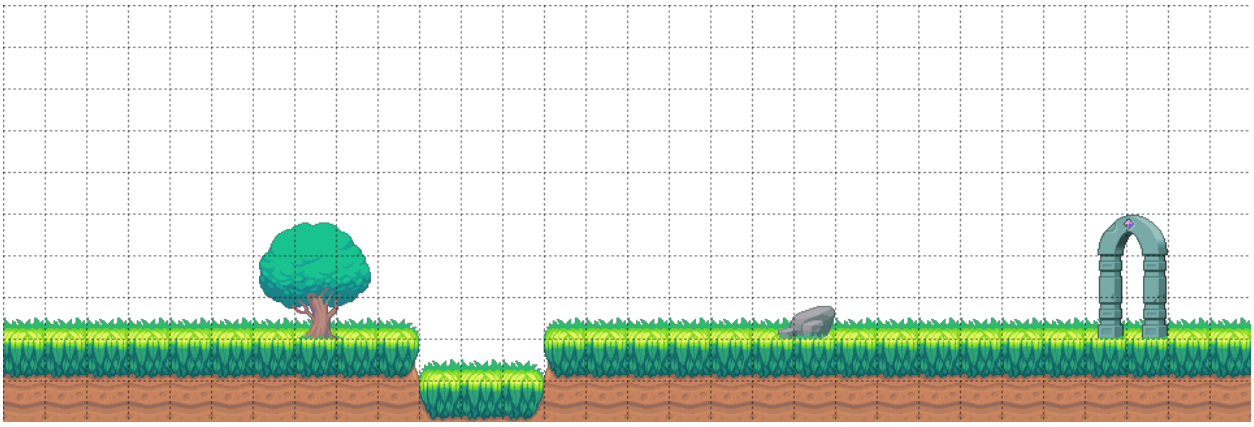


Figure 6: A game world split into a grid of tiles [13]

Although a tile system is not required for every game engine, many two-dimensional games make use of one. Breaking the game world down into a grid of tiles facilitates level design and storage. It makes rendering a small portion of the game world a trivial process, and reduces the number of calculations that must be run to determine what portion of the game world should be rendered. The tile system also acts as a spatial partitioning scheme for some of the collisions, which will be discussed in section 3.3.5. The tile system depends on the camera entity, so it should be updated about the same time as the camera system, that is, just before the rendering system.

Instead of accepting tile entities as they are added to the scene, the tile system produces entities to add to the scene for rendering. In order to determine what portion of the game world should be displayed, the tile system only accepts one entity — the camera entity — so that it knows what portion of the game is currently on the device screen.

The tile system has several properties. A tile map may be composed of several layers of two-dimensional arrays, so the first property is an `NSMutableArray` for storing layers. The second property is a reference to a `Sprite` component that contains the textures to be drawn for each tile. Finally, the tile system has integer properties for the size of the tile map (in tiles) and the maximum number of tile rows and columns that can be displayed on the device screen at one time, the visible x and visible y properties. These last properties determine how many tile entities are produced and will be less than the total number of tiles in the tile map.

The tile system has a helper class, the tile layer class. The tile layer class has an array of sprite states the size of the entire tile map layer. It also has a mutable array of entities, each of which stores the graphics for a tile. The visible x and y properties from the tile system determine how many tile entities are needed at one time. As the camera scrolls through the level, and rows and columns of tiles scroll off the screen, they can be shifted to new locations and reused for new rows and columns. The tile layer class has methods to facilitate the process of shifting rows and columns.

There is only one public method in the tile system class, and it is used to generate tile map layers. It accepts a two dimensional array of sprite states, the z-order of the layer (corresponding to z-order in the rendering system), a Boolean value indicating whether the layer is visible, and another Boolean value indicating whether the layer is a collision layer. If the layer should be visible, the tile layer generation method creates a two dimensional array of tile entities — entities with a sprite and a transform — to be used as the graphics for tiles, and each of these entities are added to the scene. Next, a tile layer object is created from the passed array of sprite states and the

created array of tile entities (or `nil`, for non-visible layers). The layer is added to the array of layers. Finally, if the collision Boolean is set to true, a new entity is created with a transform and a tile collider component (which will be explained in section 3.3.5) and this tile collider entity is added to the scene.

The update method of the tile system has one task: shift the tiles, if necessary. Using while loops, the system checks if the border rows or columns are outside the camera boundary. If a border row or column is out of bounds, it is shifted to the opposite side. For example, if the left-most column is outside the camera boundary, it is "shifted," and each tile entity in that column is moved so that the column becomes the right-most column. The current implementation actually shifts the representation of the sprites in memory, but a more efficient approach might simply be to keep track of which column in the array is considered the left-most column, and so on for each border edge.

3.3.5 Collision System

One of the basic rules of a game is that there are forbidden placements of entities in the game world. This may depend on the world — entities should not move outside of the world — or it may depend on other entities, that is, a pair of entities should not occupy the same space simultaneously. The collision system governs the interactions of entities with other entities and ensures that, by the time the rendering system displays the game, every entity is in a valid position. Collisions must be resolved after movement, so it should be updated second — immediately after the physical system updates.

3.3.5.1 Basic Collision Detection and Resolution

Resolving collisions is one of the most complex parts of any game engine. A collision system that accurately simulates the real world without slowing down the processor with a massive number of calculations is practically impossible. The collision system for this engine is designed to handle a small subset of the ways that entities can collide by limiting collision shapes and types. Specifically, this collision system will only handle collisions using axis-aligned bounding boxes (AABBs) that cannot be deformed (that is, stretched or squished). Furthermore, elasticity and friction, two physical properties that affect the way collisions change an entity's velocity, are excluded from consideration in order to simplify the system.

The collision system will only accept entities with a `Transform` and a `Collider` component. The collider component has three properties. First, the collider has a type property that determines whether the entity is "static" or "dynamic," that is, whether a collision can be resolved by moving that entity. The player, for example, is dynamic; if the player collides with something, the player can be moved to a valid position. The floor, conversely, is static; if something collides with the floor, the floor cannot be moved to a valid position — the other entity must be moved. Static objects are not necessarily immobile, however. A platform may move, but should not be moved due to a collision. The two other properties of the collider component, size and offset, are analogous to the size and offset properties of the render component.

The update method of the collision system involves collision detection (determining whether two entities collide, and therefore have invalid positions) and collision resolution (moving the colliding entities to valid positions). The brute force

approach to collision detection requires that each entity be compared to all entities. This means that, when there are n entities, the collision detection method must be called n^2 times each frame. In order to reduce the cost of collision detection, we can break down collision detection into two phases: broad phase and narrow phase [14]. In the broad phase, the system determines which pairs of entities are *likely* to collide, excluding pairs that are clearly not colliding. The narrow phase continues as usual, using more precise calculations to determine whether a pair of entities is colliding, but only on the pairs of entities not excluded in the broad phase.

The broad phase of collision detection can be implemented using a spatial data structure [15]. In this engine, a hash grid is used for broad phase detection. The system splits the game world into a uniform grid and assigns entities to specific cells in the grid. An entity can only collide with entities in grid cells it is touching; all pairs of entities that are not in the same or nearby cells are excluded from narrow phase collision detection. Using dense arrays to represent the grid is unreasonable [16], so an `NSMutableDictionary` is used, keyed using a hash that represents the grid location.

Narrow phase collision detection also includes collision resolution when a collision is detected. The basic outline of the narrow phase is to 1) detect overlap, 2) calculate a resolution vector, and 3) divide the resolution vector between the colliding entities. This is where different shapes could be implemented — the resolution vector would be different between two circles than between two rectangles, for example, but the application of the resolution vector would be the same. This collision system's implementation, however, is limited to axis-aligned bounding boxes for both broad and narrow phase collision detection.

Using the separating axis theorem, the system can determine whether two entities are overlapping, and, if so, can calculate the exact overlap amount for each axis [17]. If no overlap is found on either axis, then no collision has occurred. On the other hand, if overlap exists for both axes, then a collision has occurred. The axis with the smallest overlap is selected to minimize the distance the entities have to be moved, and the resolution vector is calculated as the vector pointing from the second entity to the first entity along the selected axis.

This collision system also includes a special type of collider for tile layers. The tile collider component is a subclass of the basic collider component, and has two properties: a tile layer object and a `CGSize`, the size of each tile. The tile layer is treated as a bit mask, where a tile position value of 0 corresponds to an empty (non-colliding) tile, and any other position value corresponds to a solid (colliding) tile. Because tiles will always be perfectly aligned to a grid, checking for a collision with a tile layer uses the tile grid as a spatial partitioning scheme — only the tiles touching the other entity need to be checked for collisions. If a collision is detected, the resolution vector is calculated by moving the colliding entity back to its original position, then moving it along each axis, one at a time, and placing it next to the colliding tile.

Distribution of the resolution vector depends on whether the colliding entities are static or dynamic. Tile colliders are always static. If both entities are static, no resolution is possible — pairs of static objects, therefore, should never be tested for collisions. If one of the entities is static, then all of the resolution vector must be applied to the dynamic entity. Distribution of the resolution vector between two dynamic entities is based on physics. If the physics system included properties like elasticity and friction,

they would be applied here, and the velocities of the colliding entities — not just the positions — would be updated. For this implementation, however, only mass is taken into account, as this engine is primarily designed for simple games rather than for accurate physics simulations. The velocities of colliding entities are simply set to zero along the colliding axis. A physics component is not required for an entity to collide with another entity, so if an entity with no physics component collides with another entity, it is assumed to have a mass of zero. If neither entity has a physics component, both entities are assumed to have a mass of 1, that is, equal mass. It may eventually be determined that a physics component is required for two entities to collide.

To calculate the ratio of the resolution vector distribution, the system takes the mass of the second entity and divides it by the combined mass of the colliding entities. This calculated value is stored as the scalar coefficient of the resolution vector for the first entity. Next, the compliment of this coefficient is calculated (that is, 1 minus the first entity's coefficient) and stored as the scalar coefficient for the second entity. Finally, the resolution vector is applied to each entity, scaled by their respective coefficients. After the resolution vector has been applied, both entities will be in valid positions relative to one another.

3.3.5.2 Collision Chaining

Unfortunately, in a scene with many entities, resolving one collision can often cause another. This problem is addressed by implementing some kind of shock propagation algorithm [18]. Shock propagation in this engine is accomplished by making recursive calls to the collision detection and resolution methods. This method will hereafter be referred to as collision chaining.

When a collision is chained, the two colliding entities should be temporarily “merged” and treated as a single unit until the collision chain has been completed and propagation has stopped. This ensures that any collisions that occurred as a result of the original collision's resolution do not reintroduce the original collision — both of the original entities are resolved as one. The two original entities should also be treated as a single unit in terms of mass so that the distribution of the new resolution vector is physically appropriate. If one of the original colliding entities is static, the unit should be treated as static for every propagated collision. In figure 7, two dynamic entities (the circles) collide with a static entity (the rectangle). The collision between the two circles is chained, and the two circles are treated as a unit that collides with the rectangle.

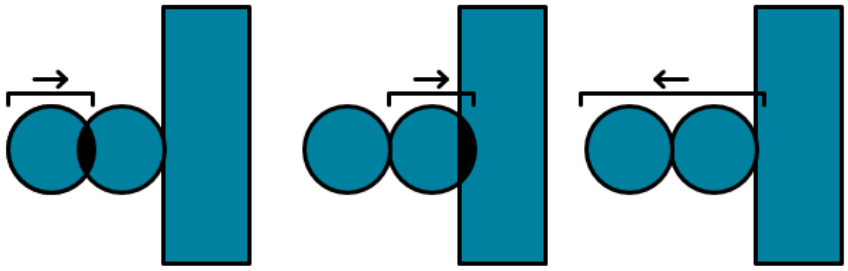


Figure 7: An example of collision chaining [19]

In this implementation, collision chaining is achieved by adding two parameters to the collision resolution method. The first parameter is a double value that is the sum of the mass of all the entities in the collision chain. For example, if entities A and B collide and their resolution causes a collision between entities B and C, the recursive call would set the chained mass to the sum of the masses of entities A and B. If resolving B and C caused a collision between entities C and D, the chained mass would be the sum of the masses of entities A, B, and C. The second parameter is a Boolean value that indicates whether any of the chained entities are static — if this is set to true,

the entity attached to the chain is treated as static, regardless of whether its collider is actually static or dynamic. In figure 7, the chained resolution of the two circles would have the static flag set to true; any chained collisions involving those two circles along that axis would be treated as static, because the circles are against a static object.

When checking for collisions that need to be chained, it is difficult to differentiate collisions that were already there and collisions that were caused by the resolution of the original collision. The original implementation did not take into account the possibility that the detected collisions could have existed before and ended up chaining collisions that should not have been chained. For example, collisions with the player always treated the player as static, because it was always chained with the resolution between the player and the ground. In order to overcome this issue, another parameter was added to the collision method that limits chaining to a specific axis. This solution would not work with circle colliders, but is valid for axis-aligned bounding boxes. Given additional time, a better solution could be implemented.

4. RESULTS AND ANALYSIS

4.1 Methodology

The initial criteria presented for evaluating this game engine were that the engine must be easy to use, flexible, and efficient. This section will discuss the methodology used for testing how well the engine meets those requirements. Efficiency is the most straightforward aspect to test, because CPU and memory usage can be directly calculated. Ease of use and flexibility are harder to quantify, but methods for determining relatively how well these requirements have been met will be discussed in this section.

4.1.1 Ease of Use

To test ease of use, a simple game will be implemented. The game will use each of the systems developed as core systems. “Ease” will be measured in terms of how complicated the code is and the volume of code required to make a working game.

4.1.2 Flexibility

Flexibility and modularity are important parts of the engine’s design. It should be easy to add a new system that adds functionality to the core engine. To test how well the engine meets that criterion, an additional system meant for a specific game will be implemented; in particular, the player input system for a platforming game. This new system will be used with the simple game created to test the ease of use requirement.

4.1.3 Efficiency

As is mentioned above, efficiency is the most quantifiable of the three criteria for evaluating how well the engine meets the objective for this project. Tests will be run to determine how well the engine utilizes CPU and memory, as well as how that usage

translates to visible performance in terms of frame rate. The game developed to demonstrate the usability and flexibility of the engine will be used to test the engine's efficiency as well.

4.2 Results

The results are described here using the metrics given in 4.1.

4.2.1 Ease of Use

To test ease of use, a simple game was implemented on top of the engine. The platformer genre was selected as the type of game to implement, as it is a common 2D gaming genre. The starting point was to include the engine as a static library, then to subclass the Scene class. By including the engine as a static library, the code for the game was completely separate from the engine, and the engine was treated as a "black box."

A few resources were created for use with the game. A sprite sheet was made for the player, with frames and animations for various states. Another sprite sheet was made for use with the tile system, with frames of the sprite sheet dedicated to various textures for the game environment. Finally, a property list XML file was created, containing an array of dictionaries that represent tile layers. Each dictionary contains a comma separated list of sprite frames, with newlines separating each row of the tile array. Two optional properties may be included in the dictionary. The first is a Boolean flag that determines whether the layer is visible, and the second is another Boolean flag that determines whether the layer is a collision layer.

To read the XML file, a `TileMapParser` class was created. The purpose of this class was to read in the .plist file and create tile layer objects to add to the tile system,

as explained in section 3.3.4. The parser contains a single static class method to accomplish this, and is meant to be called at the end of the game initialization method.

To create the entities necessary for the game, an entity factory class was created. In general, an entity factory class has a number of static class methods that piece together entities by creating a basic entity and attaching and initializing specific components. For this game, the entity factory had only one method, which was used for creating the player entity. The player entity method creates an entity and adds the components sprite, collider, camera, physics, and transform. When the player's sprite component is added, the entity factory sets up various sprite states for idling, walking, jumping, and falling, assigning specific frames from a player sprite sheet for each.

A new system was created for player input, as is described in section 4.2.2. This system introduced a new component, the player component, to identify the entity that responds to user input. This player component is also added in the entity factory's player entity method.

At this point, all of the pieces for the game were in place. A subclass of `Scene` was created, and the `viewDidAppear:` method of the `UIViewController` class (a superclass of `Scene`) was overridden and used to add the systems and entities required for the game. The sprite rendering, camera, player input, physical, collision, and tile systems were all added to the scene. Next, the player entity was created using the entity factory and added to the scene. Finally, the tile map parser loads the level property list into the tile system.

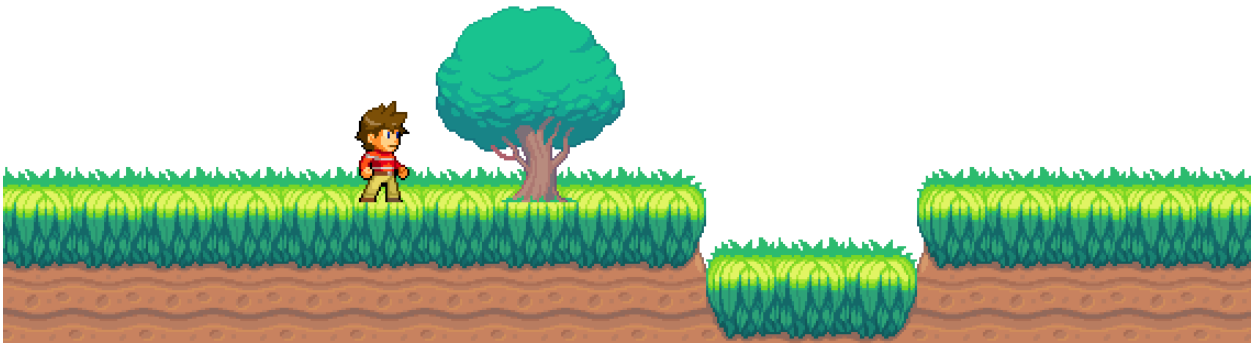


Figure 8: A screenshot of the simple game implemented using the engine [20]

The game worked as expected. The character entity and the tile layers appeared on the screen, as in figure 8. Gravity pulled the character to the ground, and the collision system prevented it from falling through the collision tile layer. Tapping on the screen allowed the character to move, and the camera followed the character as the character moved. As the character moved or jumped, the character's sprite changed to reflect what the character was doing, as in figure 9. As the character walked past tiles that were set to a lower z-order, like the tree in figure 10, the rendering system accurately placed the tiles behind the character.

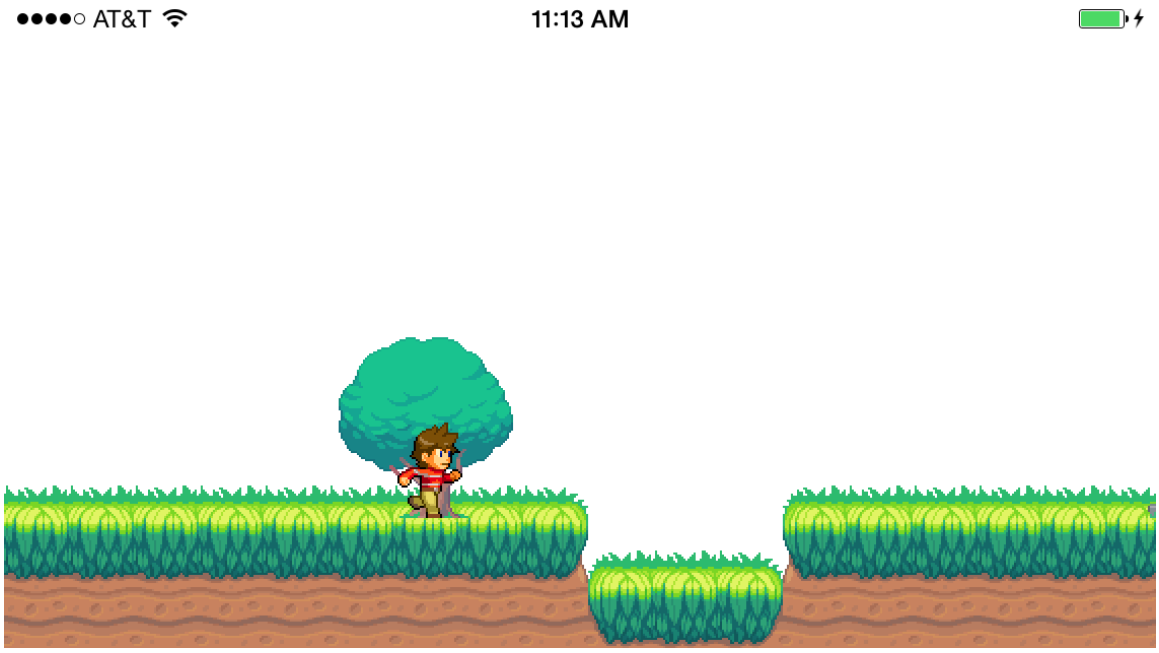


Figure 9: The character in the "run" state [21]

Figure 10: The character in the "jump" state [22]

With little effort in terms of programming, a simple game was produced using the engine. The game used approximately 200 lines of code, which included the tile map

parser. Only 27 lines of code were needed in the `viewDidAppear` method of the main scene. This does not prove that the engine is easy to use, but it certainly supports that assertion. Figure 11 shows the entire initialization method of the main scene.

```
1  [super viewDidAppear:animated];
2
3  LGEntity *player = [EntityFactory player];
4  [self addEntity:player];
5
6  [self addSystem:[[LGCameraSystem alloc] initWithScene:self]];
7  [self addSystem:[[LGPhysicalSystem alloc] initWithScene:self]];
8  [self addSystem:[[LGCollisionSystem alloc] initWithScene:self]];
9  [self addSystem:[[LGPlayerInputSystem alloc] initWithScene:self]];
10 [self addSystem:[[LGSpriteRenderingSystem alloc] initWithScene:self]];
11
12 LGTileSystem *tileSystem = [[LGTileSystem alloc] initWithScene:self];
13 [self addSystem:tileSystem];
14
15 LGTMXParser *parser = [[LGTMXParser alloc] init];
16 [parser setCompletionHandler:^(LGTileMap *map)
17 {
18     LGSprite *sprite = [[LGSprite alloc] init];
19     [sprite setSpriteSheetName:[map imageName]];
20     [sprite setSize:CGSizeMake([map tileWidth], [map tileHeight])];
21
22     [tileSystem setSprite:sprite];
23     [tileSystem setMap:map];
24
25     [self ready];
26 }];
27 [parser parseFile:@"level"];
```

Figure 11: The initialization method of the simple game's main scene.

4.2.2 Flexibility

To test the flexibility of the engine, an input system will be implemented for the platforming genre of games. All of the logic and functionality for accepting player input and making the game respond to it will be implemented in this input system. For an application to be considered a game, there has to be some kind of user interaction; in this case, the interaction is between the user and the device's touch screen. How the game responds to that input depends largely on the type of game, which is why this

system was not included in the core engine, but must be created specifically for each game. This player input system is designed so that user input is set to update a specific entity, the player entity. The input system is set up to work with platformer games using a landscape device orientation. As in section 4.2.1, the engine itself was included as a "black box" static library, so that the development of the input system was completely separate from the engine code.

In a more sophisticated implementation, the player input system would likely be the subclass of a generic input system that maps some kind of input to specific actions on a particular entity. For example, there could be an enemy AI input system that uses the same interface as the player input system. In this implementation, however, the player input system is a direct subclass of the base system class.

The system only accepts a single entity, the entity designated as the player. The player entity must have a physics component, a sprite component, a transform component, a collider component, and a player component. The player component contains a speed property to determine how fast the player can move, horizontally, and a jump speed property to determine the initial velocity of a jump.

The player input system serves two primary functions. First, as was mentioned, it responds to user input. This is accomplished by overriding the `touchDown` and `touchUp` methods of the system class. The input scheme used in this engine assumes a landscape orientation and divides the screen into two halves, a left half and a right half, using each half as a "button." When the player taps and holds on the left side of the screen, the character should move left. When the player taps and holds on the right side of the screen, the character should move right. Finally, if the player taps with two fingers

anywhere on the screen (or taps with one finger while holding the other finger to move), the character should jump.

The second primary function of the player input system is to update the player sprite's state. The player graphics can be modeled as a state machine. The update loop of the player input system updates the sprite state based on the various components attached to the player entity. A Boolean value is saved that determines whether the player can jump, based on the velocity's y component and whether the collider encountered a collision on the bottom side of the player's bounding box. If the player cannot jump, then the player is not standing on the ground, so he should go to a "fall" state or a "jump" state — this is determined based on whether the player velocity's y component is positive or negative, respectively. If the player can jump, that is, the player is standing on the ground, then the player sprite is set to "walk" if the speed is non-zero or "idle" otherwise. Finally, the sprite is mirrored horizontally, if needed, to ensure that the player sprite is facing the correct direction, the direction of movement.

The input system was added to the game engine's built-in systems. Integrating it into the game loop only required one additional line of code in the scene subclass. The input system worked as expected, which demonstrates that adding new systems on top of the engine is very easy, even when treating the engine itself as a black box. The engine, therefore, is flexible and modular.

The input system was designed with direct access to the source code. Because this engine is open sourced, any developer can use the built-in systems as a reference for how to build custom systems. Ideally, however, robust documentation as to how to extend the engine should be provided with the engine's source code.

4.2.3 Efficiency

To test the efficiency of the engine, the Mac application Instruments will be used to measure the CPU and memory usage of the engine when running various scenes. The game implemented in section 4.2.1 will be used as the test case for the engine's efficiency. The tests will attempt to find the limits of the current implementation of the engine.

Two tests will be run to measure efficiency. In each test, increasing numbers of entities will be added to the active scene until the CPU usage reaches (or approaches) 100%. The first test will measure the efficiency of the engine's rendering system, and the second test will measure the efficiency of the engine's collision system.

First, the collision system will be removed from the engine and gravity will be disabled (so that objects do not simply fall through the floor). Next, various numbers of entities (blocks with a transform, a sprite, and physics) will be added to the scene. For each number of blocks n, the CPU usage, memory usage, and frame rate will be measured and presented in a table.

When running the game engine without the collision system, the increase in CPU usage was approximately linear, and 2000 entities seemed to be the maximum number of entities before completely using up the CPU allocated for the application on the device. As previously stated, the iPhone OS limits the frame rate to 60 frames per second. The engine did not cut into that frame rate until 1500 or more entities were added, as shown in figure 12. In terms of CPU usage, adding more than 500 entities used up more than half of the available CPU.

n	CPU (%)	Memory (MB)	FPS
1	18	2.2	60
50	22	2.3	60
100	24	2.4	60
200	28	2.6	60
300	32	2.8	60
400	36	2.9	60
500	41	3.1	60
1000	63	3.9	60
1500	84	4.8	57
2000	98	5.7	45
3000	98	7.3	31

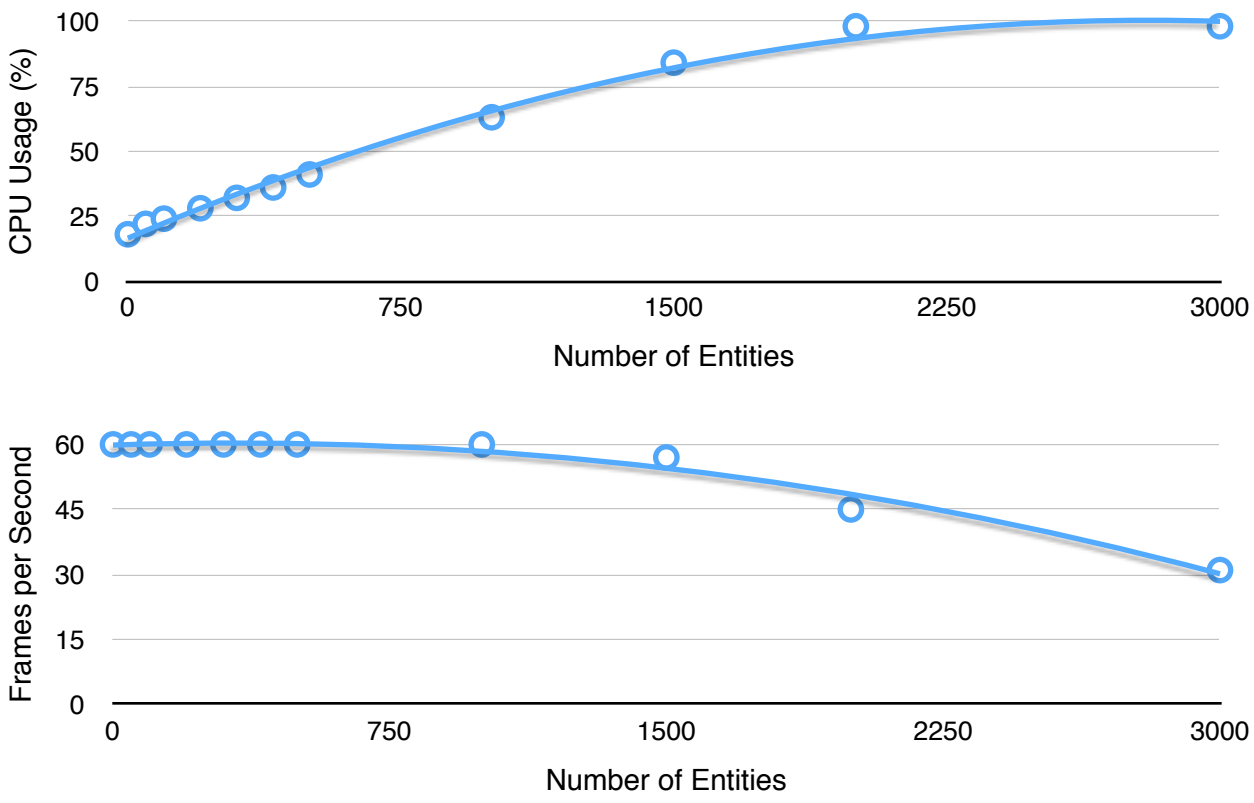


Figure 12: Engine performance without collisions

The second test was devised to test the limits of the collision system. When running the game engine with the collision system, performance degraded much more quickly than in the first test in terms of CPU usage. CPU usage growth was approximately quadratic, which makes sense considering that the basic collision method takes $O(n^2)$ time. Although this should be cut down by spatial partitioning techniques, quadratic growth is still expected. The grid-based spatial partitioning scheme in this collision system has a fixed size of grid cell, and because the only the number of entities was varied (and not the size of the game world), the number of grid cells in the test was fixed. Interestingly, the frame rate did not slow down significantly until 45 or more entities were added to the scene.

The test shows that the collision system is not particularly efficient, cutting down the number of entities that can be used simultaneously from 2000 to 40 before the CPU is completely used up. In order for the game to run smoothly, the recommended number of entities in this case is about 35 (as the frame rate did not begin to fall until 40 or more entities were added to the scene). The CPU usage reached over half of the available CPU at about 25 entities, as can be seen in figure 13.

n	CPU (%)	Memory (MB)	FPS
1	26	2.3	60
5	31	2.3	60
10	37	2.3	59
15	42	2.3	59
20	47	2.3	59
25	59	2.3	59
30	72	2.4	59
35	85	2.3	59
40	97	2.4	57
45	99	2.5	57
50	99	2.4	52
60	100	2.4	38

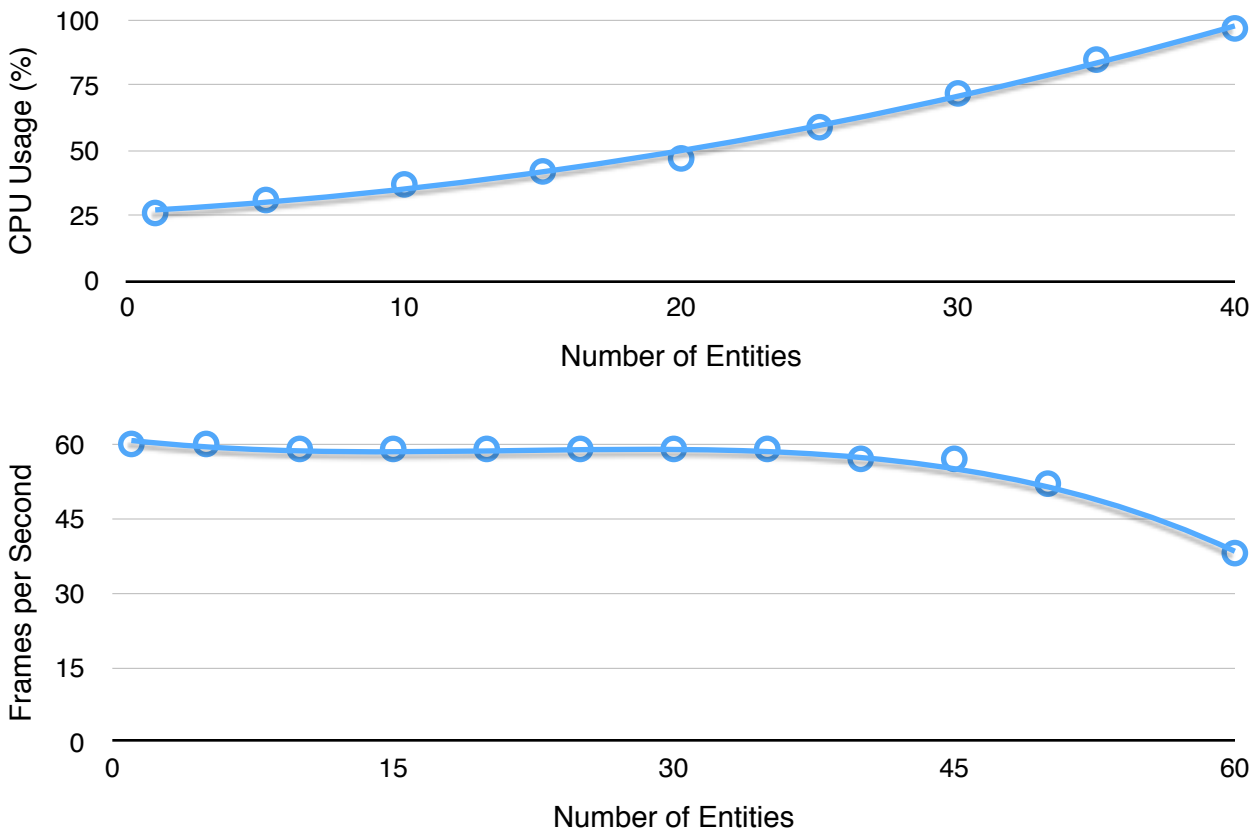


Figure 13: Engine performance with collisions

4.3 Analysis

As is clear from the tests described in section 4.2 and the test results, the implemented engine has mixed results as to how well it met the criteria. The ease of use requirement was clearly met. A simple game was built using very few lines of code, and developers not previously acquainted with the engine were quickly able to learn how to use it to perform simple tasks.

The flexibility requirement was also clearly met. This is not surprising, as the ECS model is inherently modular. Adding new functionality in the form of systems only required a few extra lines in the scene subclass compared to using only the core systems. Creating more complex systems with new types of components is a simple process.

The final requirement, efficiency, is the weakest part of the engine. Being able to add up to 1500 entities to a scene at once without perceptible lag is an acceptable result, and this was the case without the collision system. Not being able to add more than 45 entities at once without perceptible lag, which was the case with the collision system, reveals an area of inefficiency. Although not ideal, this number should be high enough for simple 2D games, as there is not enough room on the screen for many more than that. A more efficient implementation of a collision system should be considered, especially for more complex games. Because of the modularity of the ECS model, switching the collision system out for a more efficient one is not a difficult task. A developer using this engine can use the provided collision system, or, if necessary, implement a better collision system.

5. CONCLUSIONS

5.1 Summary

In this thesis, the design for a two dimensional game engine for the iPhone using the ECS model was presented as a solution to enable developers to streamline game development. Some background concepts were explained, including the ECS model and a brief overview of part of the iOS SDK used in this project. Related work was explored. An implementation of the design was then described in detail, presented as two parts: a framework and a core set of systems. Finally, the implemented engine was evaluated according to three criteria: ease of use, flexibility, and efficiency. Data was collected as the engine was tested, and the results were collected and presented. The conclusion was that the implementation strongly meets the ease of use and flexibility requirements, but has significant room for improvement in efficiency. This solution is a decent option for developers to use to easily and quickly create simple, two-dimensional games on the iPhone platform.

5.2 Potential Impact

This project has the potential to impact game developers. It will likely be most useful to individual developers, as it is a free resource and is primarily intended for relatively small-scale games. The implementation is flexible enough that it can support many different kinds of two-dimensional games and easy enough to use that developers do not have to spend a lot of time learning it.

The potential impact of this project extends to the audience that game developers might be able to reach by creating games using this engine. Games built on this engine will not necessarily be primarily concerned with entertainment. One

emerging genre of games (“serious games”) centers around the potential educational impact of games by transferring knowledge through immersive storytelling [23]. Fitness-centered games, like Nintendo’s Wii Fit, also have a wide-open opportunity in the mobile market. By cutting down on development time, the engine can enable developers to spend more time designing engaging games.

Finally, this project can be used as an educational resource for programmers. The engine has been released as an open source project under the MIT License. As an open source project, the engine will be continuously improved upon. It can serve as a tool for new programmers to learn how game engines work and for experienced programmers to explore a new approach to the design of a game engine, namely the ECS model. The source code for this project (approximately 3200 lines of code) is available on GitHub [24].

5.3 Future Work

One of the fundamental limiting factors in the design of this engine is its lack of portability. By using Objective-C and the Cocoa framework throughout, the engine cannot be easily ported to platforms that do not use these technologies. This excludes every platform except OS X and iOS. Future work might include a similarly designed engine using more portable technologies, such as the C++ language and Open GL ES for rendering [25].

Other future work includes improvements to the existing engine. The collision system can be significantly improved in the area of efficiency, as is indicated by the results in chapter 4. Better shock propagation techniques can be implemented, and functionality for additional collision shapes and rotation can be added.

The parsing of level data files is currently left to the developer. The engine could include a parser for universal data formats, such as the .tmx XML format for tile-based games [26]. This would eliminate some of the work a developer has to do to create a game and minimize development time even further.

Also currently left to the developer are methods for scene transitions, automated sequences (such as dialog boxes), inventory systems, and many other common game components. Many of these systems are used widely enough in games that adding them to the game engine would be beneficial.

REFERENCES

[1] Smith, A. (2013). Smartphone ownership 2013. Pew Research Center. http://www.pewinternet.org/2013/06/05/smartphone-ownership-2013/

[2] Shantz, M. (1998). *Designing a PC game engine.*

[3] Gregory, J. (2009). Game engine architecture. CRC Press.

[4] Blow, J. (2004). Game development: harder than you think. *Queue*, *1*(10), 28.

[5] Valente, L., Conci, A., & Feijó, B. (2005). Real time game loop models for single-player computer games. *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment 89*, 99.

[6] Lewis, M., & Jacobson, J. (2002). Game engines. *Communications of the ACM*, *45*(1), 27.

[7] Wenderlich, R. (2013). Introduction to component based architecture in games. http://www.raywenderlich.com/24878/

[8] Goldstone, Will. (2013). Unity 4.3: 2D game development overview. http://blogs.unity3d.com/2013/11/12/unity-4-3-2d-game-development-overview/

[9] Hammer, J. (2012). *The design and implementation of a mobile game engine for the Android platform.*

[10] Godfrey, L. (2014). Screenshot of a simple game to demonstrate z-order.

[11] Godfrey, L. (2014). Run animation from a sprite sheet.

[12] Godfrey, L. (2014). Screenshot of a simple game to demonstrate camera system.

[13] Godfrey, L. (2014). Screenshot of a simple game to demonstrate tile system.

[14] Hubbard, P. M. (1995). Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics, 1*(3), 218–230.

[15] Samet, H. (1990). The design and analysis of spatial data structures. Addison-Wesley, Reading, MA.

[16] Schornbaum, F. (2009). Hierarchical hash grids for coarse collision detection. University of Erlangen-Nuremberg.

[17] Gottschalk, S., Lin, M., & Manocha, D. (1996). OBB-tree: a hierarchical structure for rapid interference detection. ACM SIGGRAPH.

[18] Erleben, K. (2005). Stable, robust, and versatile multibody dynamics animation. University of Copenhagen, Denmark.

[19] Godfrey, L. (2014). Collision chaining: a collision system solution. http://devblog.lukesterwebdesign.com/collision-chaining-a-collision-system-solution/

[20] Godfrey, L. (2014). Screenshot of a simple game.

[21] Godfrey, L. (2014). Screenshot of the character running in a simple game.

[22] Godfrey, L. (2014). Screenshot of the character jumping in a simple game.

[23] Yust, T. (2012). *A framework for constructing serious games.*

[24] Godfrey, L. (2014). Open source. http://devblog.lukesterwebdesign.com/open-source/

[25] Rideout, P. (2010). iPhone 3D programming: developing graphical applications with OpenGL ES. O'Reilly Media, Inc.

[26] Itterheim, S., & Löw, A. (2012). Working with tilemaps. *Learn cocos2D 2* (pp. 265-290). Apress.