

CO3015 Computer Science Project

Android Game Using an Entity Component System

Dissertation



April 15, 2017

Asamari Egwu

Department of Informatics, University of Leicester

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people’s work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s).

Any part of my own written work, or software coding, which is substantially based upon other people’s work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s).

I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Asamari Egwu

Signed: 

Date: 7/12/2017

Contents

[Abstract 1](#_Toc511555198)

[Introduction 2](#_Toc511555199)

[Aims and Objectives 3](#_Toc511555200)

[Aims 3](#_Toc511555201)

[Objectives 3](#_Toc511555202)

[Background Research 4](#_Toc511555203)

[Entity Component System (ECS) 4](#_Toc511555204)

[Deciding on Artificial Intelligence (AI) 5](#_Toc511555205)

[Pathfinding Algorithms 6](#_Toc511555206)

[Why LibGDX and Artemis-ODB? 7](#_Toc511555207)

[LibGDX 7](#_Toc511555208)

[Artemis-ODB 8](#_Toc511555209)

[Final Software Architecture 9](#_Toc511555210)

[Entity Component System 9](#_Toc511555211)

[Building Game Objects out of Components 9](#_Toc511555212)

[How Systems Interact with Entities 11](#_Toc511555213)

[How Worlds Interact with Systems 12](#_Toc511555214)

[Summary 13](#_Toc511555215)

[Utility AI Design 14](#_Toc511555216)

[Prototype Description 16](#_Toc511555217)

[Gameplay 16](#_Toc511555218)

[Usage of Components and Systems 20](#_Toc511555219)

[Key Components 20](#_Toc511555220)

[Other Components 20](#_Toc511555221)

[Key Systems 21](#_Toc511555222)

[Other Systems 21](#_Toc511555223)

[Expansion 21](#_Toc511555224)

[Planning 22](#_Toc511555225)

[December 22](#_Toc511555226)

[January 22](#_Toc511555227)

[February Onwards 23](#_Toc511555228)

[Bibliography 24](#_Toc511555229)

# Abstract

In game development using traditional object-oriented design can make changes to code difficult as the size of the project increases. Refactoring child functionality into a parent class, may be a solution but the process can become difficult to maintain.

The game design process ever evolves, and it can be difficult to plan every single interaction and change you are going to have when you first start.

In this project I investigated using an Entity Component System (ECS) to create a skeleton of an android game. An Entity Component System is one where all game objects become ‘entities’ and are broken down into different components. For example, a bottle could be made out of a ‘Position’ and ‘Container’ Component. A book could be made out of a ‘Position’ and ‘Readable’ Component. As both objects can be placed, but they have different functionality.

Systems then interact with different entities based on their components. For example, a ‘Movement’ System may be set to only interact with objects with a ‘Position’ Component. Which means the bottle and book could be edited by the Movement System.

The essence of ECS is to find the commonality between game objects so changing an entity from one thing into another thing is a lot easier than with traditional object-oriented inheritance.

The game I finally, settled on making is a turn-based strategy game involving protecting your base and defeating enemies trying to attack it. You travel through different scenarios to eventually confront the final boss protecting the health of your player-controlled units. In which after defeating said boss, the game ends.

//Notes: Is it a good idea in the abstract to mention other technological aspects of my game? Such as the ones shown below? As they weren’t part of the mission statement, but cropped up over time based mostly on my choice of game.

The game also utilizes other concepts such as a score-based decision AI, also known as ‘Utility’ Ai. Pathfinding algorithms, such as A\*. Also, the game is randomly generated upon starting so each run through should be slightly different.

# Introduction

Game development is an area I’ve had a small interest in for quite some time. Due to this when researching how to create my own games I stumbled across a different way to create a game. It was called an Entity Component System (ECS). This architectural pattern is a different way of building and creating game objects. Thief: The Dark Project (1998) was one of the first games known publicly to use this architecture. Over the years it has increased in popularity, some game engines have been built using this idea as it’s core.

The core principles of ECS will be explained further in the dissertation.

The aims of my project were:

* To create a ‘skeleton’ game which utilised ECS to build and create game objects and their interactions with the game word. The use of ‘skeleton’ I did not plan to create a fully fleshed out game as given the time, I knew from the start the use of core principles ECS is what I wanted this project to be about, not the actual game itself.
* Build a game, which uses intuitive controls for the platform I selected. (Mobile). A user should be able to interact with the game without too much confusion.
* Build the game using both LibGDX and Artemis-ODB frameworks. The reason for choosing these frameworks will be explained further

The objectives of my project were to:

* Build a core gameplay loop, which would be the frame I created game objects for. An example of a gameplay loop would be one in Chess, first it is a player’s turn, they move a piece. Then it is the opponents turn, they move a piece. This continue until there is a victor. If the project didn’t have a gameplay idea in place, it would be difficult to build anything for it.
* Finish with a working and functional product. This is an over-arching objective. I wanted the game to be played until a conclusion. To be in a state where it could be ‘completed’. Games are notorious for over-scoping and being able to finish in time, was an objective I wanted to keep.
* The control scheme of the game need to be intuitive to use. This could be done using a tutorial or using commonly known visuals. The game needed to be able to ‘speak’ for itself and the player needed to be able to understand.

I faced several challenges when trying to adhere to both the aims and objectives of the project. The biggest I feel is that, in essence, this was a ‘creative’ project. The canvas may have been too blank, as although I wanted to focus on the principles of ECS. I still needed to also create a game of my own. Perhaps a better alternative, may have been to re-title the project. ‘Tetris: Using ECS’. Create a known game, but use ECS architecture. As the technical aspects of the project would remain the same, and wouldn’t be restricted by my own potentially lacking creativity. I’ll discuss more in the reflective section of my dissertation.

## Requirements

Below is the list of requirements I initially creating when starting my project. In the list the requirements that have changed over the course of the project have been marked with a (\*) and their changes will be written below.

//Note: Here I reference, changes made over the course of the project at this point I haven’t actually //shown the game. Would it be better to just detail all the requirements, and then later reflect what //has changed about them?

### Functional Requirements

#### Basics

* The Player can select different modes of game to play\*
  + The different modes of the game have been changed to instead pick different characters to play through the game with.
* The Player can view their statistics
* The Player can view their character\*
  + Characters. A player controls multiple units instead of just one.
* The Player cannot save their game during combat
* The Player can pause the game
* The Player can exit the game

#### Player Character

* A player can control a single player character\*
  + Multiple player characters.
* The player interacts with enemies by either tapping, swiping or dragging them.
* If the player’s character dies the game is over\*
  + An additional resource has been added,
* A character’s animation responds based on player input (stretch)\*
  + Not included

#### Gameplay (Needs to be tested)

* The game will have a player turn and enemy turn
* During the player turn the player can attack enemies
* During the enemy turn the player can defend themselves
* Enemies will be defeated when they run out of health
* Arenas can hold one or multiple waves of enemies
* A player can recover loot from enemies or chests\*
  + Player rewards are recovered after a battle has taken place.
* A player can activate special moves that do more damage to enemies
* Players both attack and defend themselves using touch controls

#### Map

* The map will display different rooms to players
* The map will display special rooms in different colours to players
* The map will highlight the current room the player is in

### Non-Functional Requirements

* The game utilizes both LibGDX and Artemis ODB.
* The game utilizes both sound and music
* The game utilizes a range of touch controls, (swiping, tapping, dragging)
* The game can be installed on an Android Phone
* The game can be run on minimum Android SDK version of 15 (changeable)
* The game can save and load user data
* Mobile processors should be able to render the game without any significant drops to frames
* The game controls should be intuitive to users, easy to pick up and use
* The game can be hosted on the Play Store without content issues
* The game will have item customization for characters

# A Brief Showcase of The Project

Before moving on to describing the different software and architecture and researched and used to create my project. I first wanted to show and explain different parts of the front-facing side of the game. This is to help with visualization when describing the back-end.

A summary of the game loop I created is this:

* A player first decides which character they want to pick to complete the game.
* A player then must travel through a map, from the left of the screen to the right, until they reach the end.
* A player is faced with different battles that they need to complete. During these battles two resources are threated which determine if a player loses the game or wins.
* These resources are the player’s characters’ health. And their morale. If either of these numbers reaches zero, the game is over. And the player needs to start from the beginning.

Here are some images of the game in action.

This is the Menu Screen before you enter the game proper. You select which characters you want to be in your party You can only select one from each group of characters.



Figure 1: Character Selection Screen

This is the game map. It is randomly generated for each game. In order to traverse to the end of the game you need to tap on one of the bright white nodes and that will trigger an event based on the image. The ones highlighted below will trigger a ‘Battle’ event where player will face off against enemies.

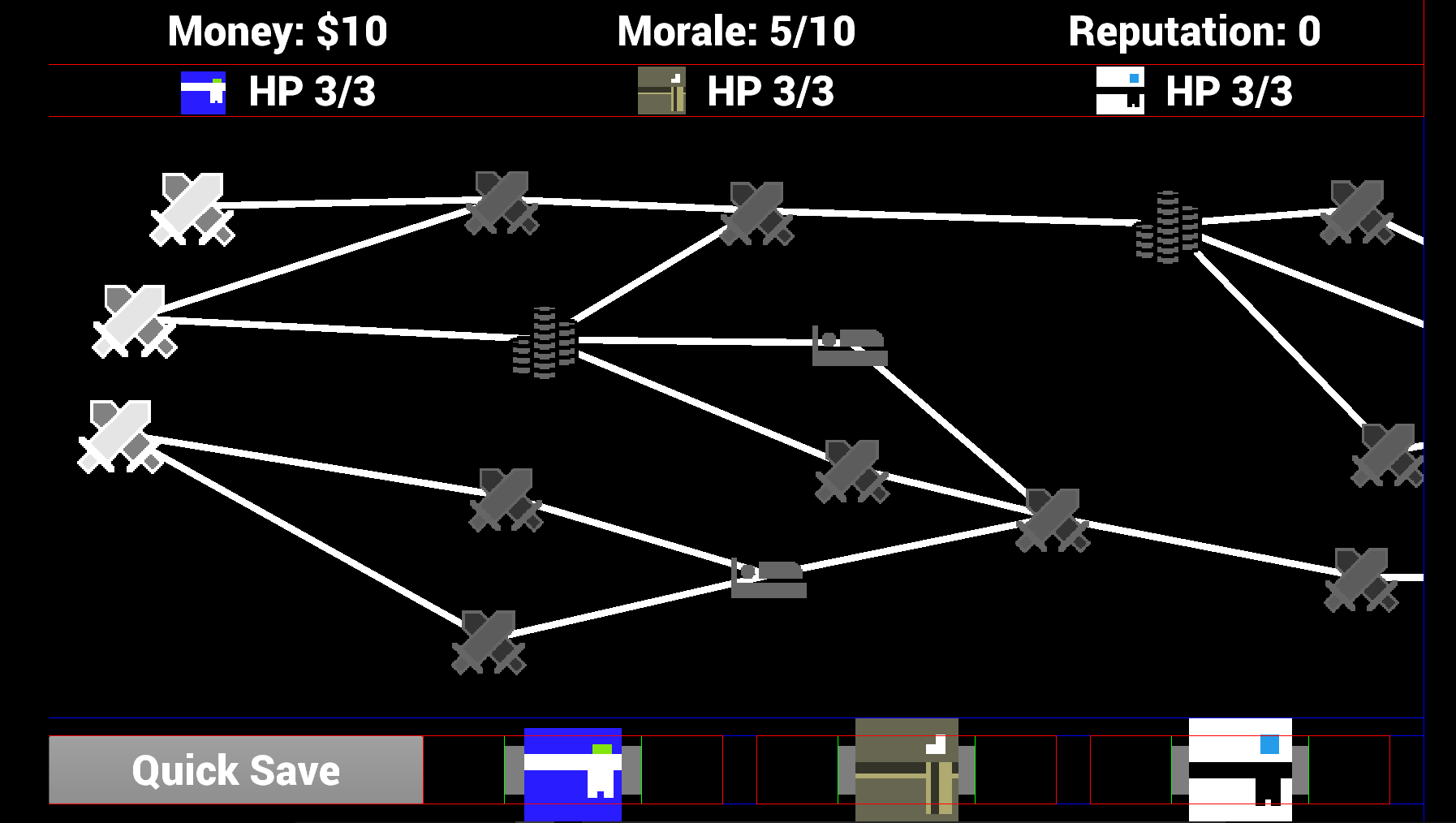


Figure 2: Map Screen

This is a battle. When a battle first begins a players needs to select where to deploy their units, within the blue zones. A battle has objectives (shown on the right) and within a battle you need to protect your bases, while defeating waves of enemies.

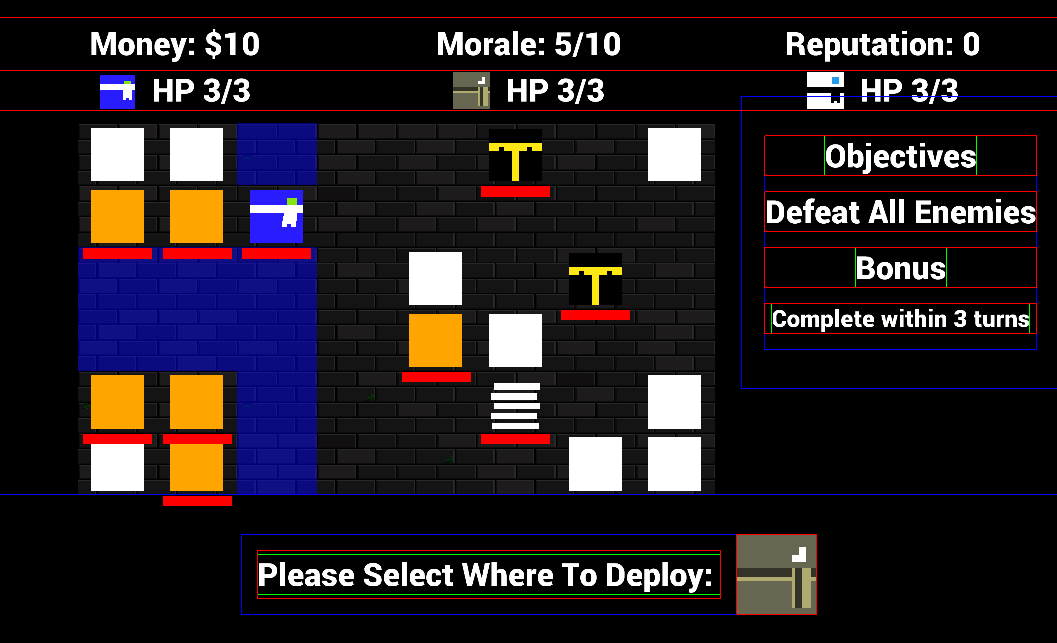


Figure 3: Battle Screen (Deployment)

You can select your characters and their skills. Enemies, move and telegraph where they will attack when a turn is over. It is up to you to push/stun/defeat your enemies so they do not attack either you, or your bases.

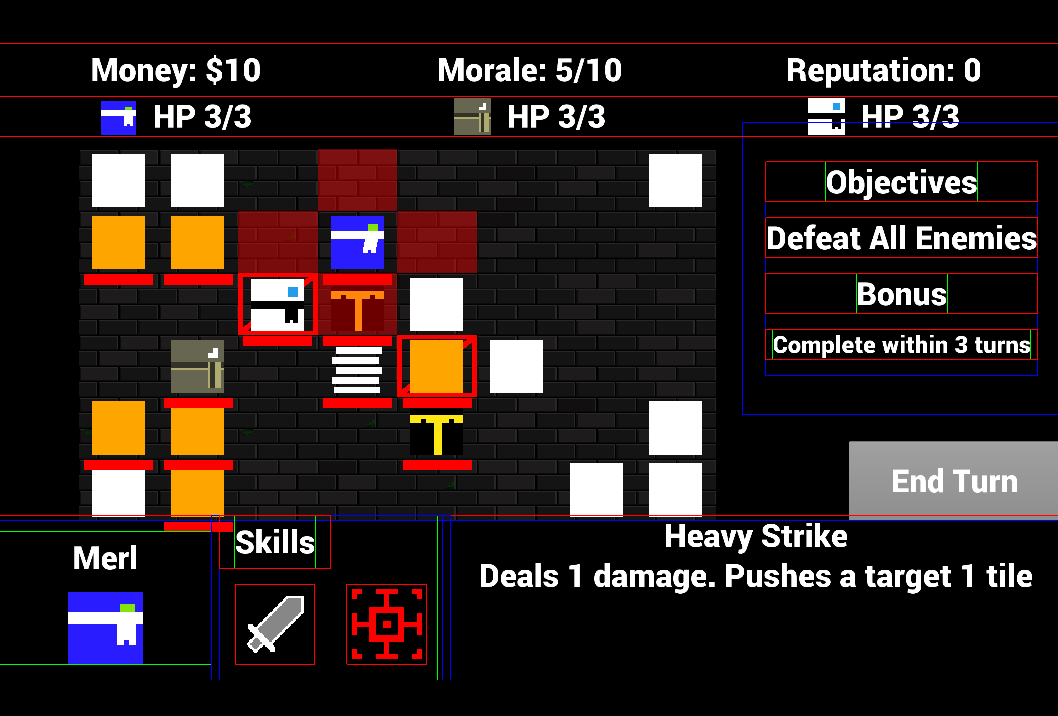


Figure 4: Battle Screen (Player is attacking)

This is what a shop looks like within the game, you can buy and sell items.

# 

Figure 5: Shop selling various skills

Here is a rest site, where a player can decide to either heal their characters or heal their ‘Morale’.

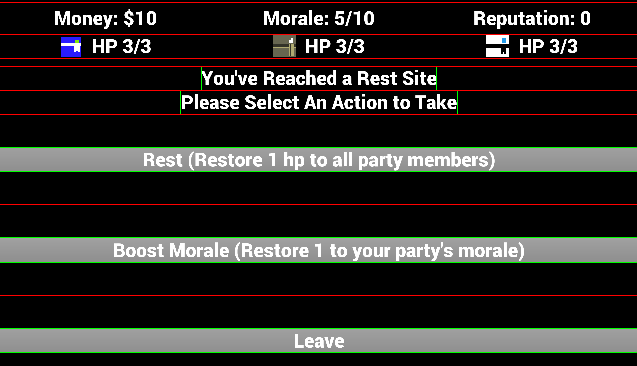


Figure 6: Rest site. Restore Health or Morale?

# Overview of Software, Architecture and Design

Initially, this project was focussed solely on ECS architecture, however as the game developed and I started learning more towards a strategic video game. More was required to see the game to fruition. Design patterns such as Observer and Utility AI. As well as looking into pathing finding algorithms. For each major technically aspect of my project I will detail the research and implementation of it within the project. Starting with the most important, which is the ECS.

# Entity Component System (ECS)

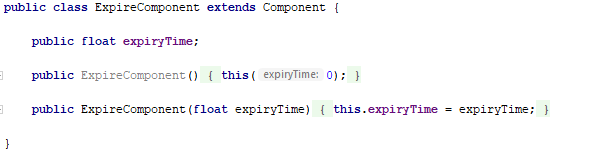
My original reason for doing this project is that I heard about ECS when talking with a friend who was developing their own game. They talked about how they had initially found a bit of trouble inheritance and that they’d started to investigate using ECS. I didn’t know what that was, but was interested in making games at the time and asked him about. I then went on to do further research I figured it was what I wanted to base my project on.

To explain ECS properly you must first break it up into its 3 words. Entity, Component and System.

## What is a Component?

The purpose of a component is to ‘Hold pieces of game data, but not game logic through them’[6].

An example of a component can be seen below.



Very short and very simple. This component only holds data and is unaware of anything to do with the game. Component can be edited by an external force but internally, the best practice is to ensure they are only aware of themselves.

## What is an Entity?

‘Entities are a collection of components’ [6] The soul purpose of an entity is to act like a bag of sorts. It stores as many components as you require it to and those components define what it is in the game world. An entity also does not know what it is. It has no reference to that.

An example of this can be seen here. If you wanted to build a tree in your game it might be as simple as doing this:

Tree

Plant

Texture

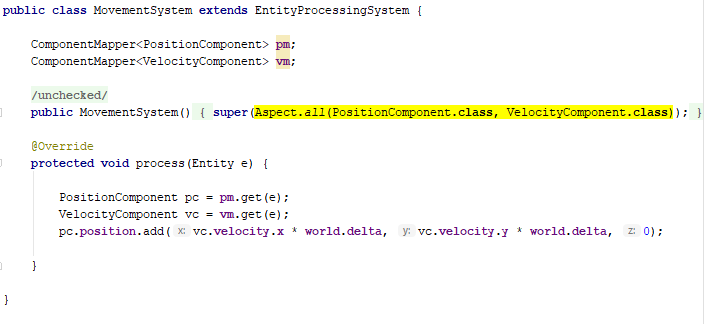
Position

As stated before, this entity wouldn’t know that it’s a tree. But because it hold those three components it is treated like a tree, by the game world and it’s systems.

## What is System?

Systems typically operate on a group of entities that share a specific set of components. It is in systems where the game logic is created. [6]

Systems act like ‘gatekeepers’ to their own functionality. An example of a system can be shown below:

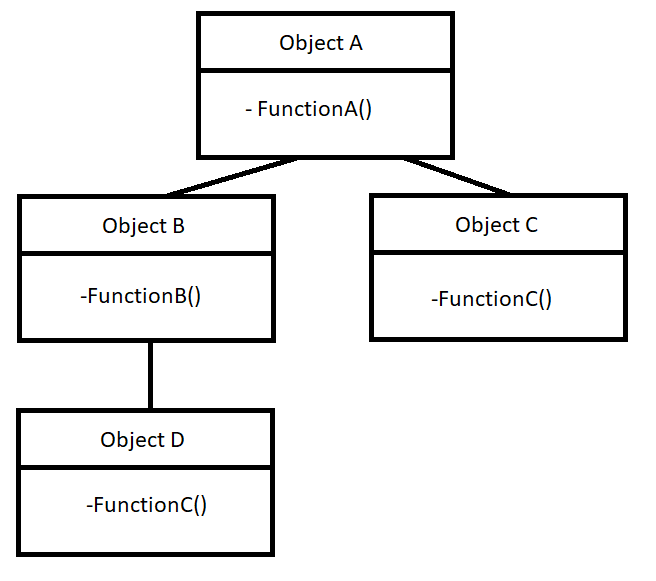


The highlighted portion of the image shows the two components this system looks for in all entities that currently exist within the game word. Only entities that contain the Position Component and Velocity component are operated on by this system.

## So why use an Entity Component System?

‘ECS can circumvent the ‘“impossible” problem of hard-coding all entity relationships at start of project’ [5]. A common problem amongst software is that it can be difficult to plan for unexpected changes in functionality. Object oriented programming tactics such as using inheritance can already help circumvent this problem.

For example, here we have a small problem:



One the key software principles in the D.R.Y principle. Don’t Repeat Yourself. This tree contradicts this as both Object C and D contain FunctionC. Object D requires functionality from B and A and C, so what can you do? In this case a simple refactor is in order, to maybe push FunctionC up in the inheritance tree. There are many solutions to this. However, this is a simple tree. What if the tree had quite a few more levels than just 3?

ECS changes this problem into this:

Entity A – Component A,

Entity B – Component A, Component B,

Entity C – Component A, Component C,

Entity D – Component A, Component B, Component C

The functions are then handled by the systems instead. This increase the flexibility of your design, Entity D can turn into and function like Entity C, simply by removing a component. This can ‘Allow a single entity to span multiple domains without coupling the domains to each other.’[11]

ECS changes the way you view objects within your game. You need to be able to break them down into similar components. And only create new components if you have a functionality that is truly unique to that component. One of the benefits of this system, is that new functionality can be created simply by making a new component and making a new system that only looks for that component.

In my implementation I called components like that ‘identifiers’ as they held no data, but simple by added them to an entity it would act differently in the game world.



This class would be an example of that. Simply by adding this into an entity it is treated differently by the game.

Now, there are downsides to an ECS. As with most software architecture and designs it is only as good as the person who built it. You can end up flooding your game with components and systems that aren’t necessary or create components that aren’t decoupled from other components. Which means it can become harder to maintain.

## My Implementation of an Entity Component System.

In an actual implementation of an ECS your entities are usually composed of a lot more components. Especially ones used the main parts of your game, such as player characters.

The Artemis-ODB framework comes with pre-set classes known as World, System, Entity and Component.

To briefly explain the concept of an Entity Component System, using the introduction from the Artemis-ODB GitHub [6]

Components hold pieces of game data, but no game logic goes through them.

Entities are a collection of components.

Systems typically operate on a group of entities that share a specific set of components. It is in systems where the game logic is created.

# Background Research

## Entity Component System (ECS)

To briefly explain the concept of an Entity Component System, using the introduction from the Artemis-ODB GitHub [6]

Components hold pieces of game data, but no game logic goes through them.

Entities are a collection of components.

Systems typically operate on a group of entities that share a specific set of components. It is in systems where the game logic is created.

An example of a game object being built using components could look something akin to this:

Tree

Plant

Texture

In this case a ‘Tree’ within the game has a position and texture and is a plant. This means if there is ever a system created that only operated on entities with a ‘Plant’ component this tree would be processed by the system.

Later in the software architecture, the idea of components, entities and systems will be broken down and explained in greater detail, but for now these are the basic tenants that go into designing a game using a system like this.

ECS can circumvent the ‘“impossible” problem of hard-coding all entity relationships at start of project’ [5].

If everything is built using components, adding and removing functionality from a game object only requires you to add and remove components from them instead of refactoring a daunting amount of code.

When a game developer utilised this system within a project he worked on, he stated that it allowed ‘unrestrained re-writing of fundamental game features post-launch with very little effort’ [5]. This is what makes this approach stand out from traditional ways of creating games using object oriented programming and why I decided to base my project around using it.

## Deciding on Artificial Intelligence (AI)

The genre I have decided to focus on is turn-based strategy. This opened the question of, how do enemies decide to do things?

Originally, games incorporated the use of finite state machines.

IDLE

Is hit

ATTACK

Nothing to Attack

Is Rested

REST

However, in larger game these machines could balloon to have hundreds of states [1]. The maintainability of this approach is suited more to simpler AI.

Next, behaviour trees allowed for a more readable way to create AI within games. The AI would work its way down decision trees to decide what action to perform. However, some believe that behaviour trees are slowly becoming a thing of the past thanks to the emergence of Utility AI [1].

At a Games Developers Conference (GDC) in 2010 a presentation was shown called ‘Improving AI Decision Modelling Through Utility Theory’ [2]. ‘Utility Theory’ is essentially a way to quantify what may cause a person to pick one action over another action. An example displayed within the conference is:

What would a person do if offered a piece of pizza?

If the person was hungry, they’d take the pizza. What would happen if they were offered another piece? Or another after that? Eventually, a person would decide they no longer wanted the pizza.

Transferring this to AI, based on the situation, certain actions will be programmed to have a greater weight than others. This approach flips the thinking from states and trees to a more human approach to decision making.

As Utility AI is a design-based AI, there is one major flaw: ‘it will still rarely be smarter than the AI developer who designed it.’ [1]

As the developer who is using this approach I’m taking this quote to heart. I vastly prefer this approach to both finite state machines and behaviour trees. I will later explain in greater depth the architecture behind Utility AI.

## Pathfinding Algorithms

As the game shifted to using coordinates as positions, I figured I would need to create a pathfinding algorithm within my program.

Pathing finding algorithms are ways for a computer to find the shortest route between two points.

I already knew about the existence of Dijkstra’s algorithm, but came across the A\* pathing algorithm. A manual tutorial online [3] inspired me to look further into the benefits of using A\* over Dijkstra’s.

In a comparison between A\*, Dijkstra’s and Greed Best-First-Search algorithms [4], when obstacles were introduced, both A\* and Dijkstra’s could find the shortest path. Dijkstra’s took longer as it looked at a larger number of co-ordinates.

This could mean that if your only purpose is to find the single shortest path as fast as possible then A\* is the better option.

However, it might also be beneficial to know the shortest paths to more than just one point, which Dijkstra’s algorithm facilitates.

In brief conclusion, for my current prototype, both differences are negligible. In the comparison, the maps were around the size of (25 x 50). My map is around (5 x 10). The speed concerns between the two algorithms at this current state are so small I could utilize either or both and the gameplay would not be hindered at all.

## Why LibGDX and Artemis-ODB?

There are two precursors I used when I selected the game engine/framework I would use to create my project: Can it use an Entity Component System? Can it create games for the Android platform?

### LibGDX

The reason I picked this framework was because its documentation is quite broad and well written, it is open source, and LibGDX also has two ECS frameworks that can be attached to it.

Additionally, LibGDX is ‘lightweight’ in comparison to game engines, as it is a framework. In LibGDX you are given the tools to create your own game engine easily, but it is not an engine in itself.

Such tools include [10]:

* Cross platform integration (Android included)
* Audio handling (both music and sound)
* File I/O (Saving data)
* Graphics. (The complexities of drawing something to the screen is reduced to a simple draw call).

In a game engine, such as Unity, you are provided with a graphical user interface and the ability to ‘drag and drop’ elements onto the screen. Using Unity, it is possible to create a simpler game without the need to code at all. LibGDX provides no such luxury.

I prefer having the freedom to decide what’s going on within the game, instead of using some pre-built interface and settings. Although, if your goal is to just create a game, picking a higher-level game engine is the better option.

An apt metaphor for the difference between using a game engine and a game framework would be building a house out of Lego.

With a framework, you are given the pieces and then get to build your own house.

With an engine, you’re a given a house frame and then allowed to build the rooms however you want.

Without either you need to make the Lego bricks first. Which would take even longer than building the house.

### Artemis-ODB

ECS frameworks that can be built with LibGDX include the ‘Ashley framework’ and ‘Artemis-ODB’ framework.

The decision was made due to the charts shown in [7]. In comparison to Ashley, Artemis-ODB performs better. In game development, speed is incredibly important. The only difference between the two frameworks is the classes it provides but the general concept is the same.

Both Artemis -ODB and Ashley provide the classes necessary to facilitate the creation of a game using the Entity Component System.

Anything that involves the relationship between components, entities, systems and worlds is abstracted for the developer to use in creating their own version of the ECS model. Without the need to create the backend that allows it all to be possible.

# Final Software Architecture

As both my prototype and final solution will share the same software architecture I felt it would be prudent to first explain the architecture before discussing my prototype.

## Entity Component System

### Building Game Objects out of Components

Components are a way to store to ‘game data’. An example of this can be seen from the ‘PositionComponent’ I use in my prototype.



Figure 7: Constructor of the 'PostionComponent' class

This Component basically holds a Vector position which contains an x and y value. This value is used to determine an entity’s position within the game world.

Below is an example of building game characters using Entity-Component Design. These characters and components are not associated with my game.

Paladin

Priest

Warrior

Melee Component

Entity

Entity

Entity

Heal Component

Heal Component

Rage Component

Melee Component

Range Component

Render Component

Render Component

Render Component

Disclaimer: The images created in this section were inspired by the ones used in [9] but configured to show my own example.

What makes this different from using a more common inheritance-based architecture is that there is no more inheritance when making a game object. Everything falls underneath the ‘Entity’ class. ‘Warrior’ is just what you would call an entity that had a melee, rage and render component.

Entities are essentially just bags that carry different components. The way entities act in the game world are determined by which components they have and the data the components carry.

The complexity of inheritance trees has been lost and all it takes to turn a Warrior into a Paladin or Priest is switching one or two components.

In an actual implementation, entities are composed of a lot more components than simply three. However, the principle is the same. To change the behaviours of an entity within the game you should only have to change its components. The hassle of large inheritance tree refactors are removed.

Below is an entity being built using components from the prototype.

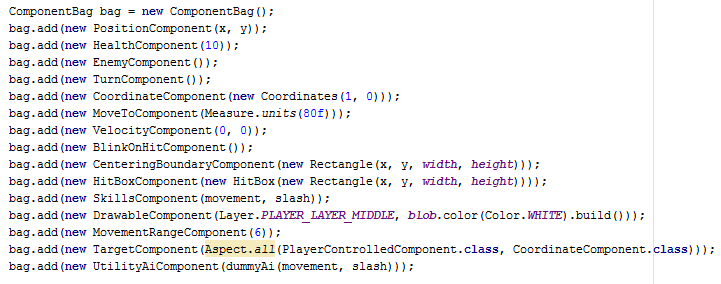


Figure 8: Creation of the 'blob' enemy using components

But if components are just holders of game data, what uses this game data? What determines an entity’s different behaviours when new components are added? Systems. Which I will explain next.

### How Systems Interact with Entities

For an entity to be processed by a system the entity needs to have the correct components. Systems are designed to only accept and process entities that have the components they desire.



Figure 9: Constructor of the 'MovementSystem' class

This snippet from my prototype shows that for an entity to be processed by the ‘MovementSystem’ they must have a ‘PositionComponent’ and a ‘VelocityComponent’.

So, what is processing?

If we look at the larger code of this system, you will see a method called ‘process’.

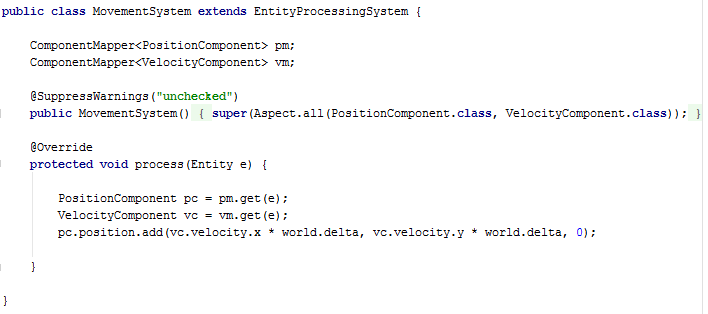


Figure 10: Full ‘MovementSystem’ class

Every game frame this system looks for all entities that have both of its desired components (Position and Velocity) and then manipulates the data stored in those components. In this case this system adds the speed the player is travelling to its current position.

This is the ‘game logic’ I was referring to earlier. Systems enact changes to entities based on their components. Therefore, changing one component can have a large effect on an entity’s behaviour within the game. Removing the ‘VelocityComponent’ from an entity would make it unable to move, as it would never be affected by the Movement System.

Ok, so entities are made up of components, and systems manipulate their components. But what causes a system to be processed? That would be the ‘World’.

### How Worlds Interact with Systems

In any game, before each frame, a myriad of calculations are performed to determine how the world is rendered to the player.

The ‘World’ in Artemis-ODB works similarly. You add several systems to it and every frame you tell the world to ‘process’ itself. The world then processes each system it has sequentially.

Below is the current configuration of the world I use for combat within the prototype. The Systems are ordered from top to bottom and have priority as to when they are run every game frame. When I discuss the prototype some systems here will be explained.



Figure 11: Creation of the 'World' using systems

### Summary

In summary, below is a more abstract depiction of how worlds, systems, entities and components all interact.

World

(Processes Systems)

Movement Component

Rendering Component

Attacking Component

Entity 3

Entity 2

Entity 1

System 3

(Attacking System)

System 2

(Rendering System)

System 1

(Movement System)

Entity 1 holds all three components and is processed by all three systems every frame.

Entity 2 holds only the rendering component is only processed by the Rendering System every frame.

Entity 3 doesn’t have any components and is not processed by any system every frame. But still exists within the world.

Thus concludes, an explanation of the ECS software architecture that will be used within the project.

## Utility AI Design

|  |  |
| --- | --- |
| Decisions | Questions |
| End Turn | Can I do anything? |
| Move Towards Target | Am I in Range to Attack?  Can I use my movement skill? |
| Attack Target | Am I in Range to Attack?  Can I use my attacking skill? |

The Design of utility AI is akin to series of decisions whose values change based on the situation an AI finds itself in. Take the below example, using an enemy within the game.

It has a total of 3 decisions and each decision has a few questions that affect whether the decision can be made.

This can be converted to look something like this:

|  |  |
| --- | --- |
| Actions | Calculations |
| End Turn | Can I do anything?  No = 100 |
| Move Towards Target | Am I in Range to Attack?  Yes = -1000  No = +100  Can I use my movement skill?  Yes = +100  No = -1000 |
| Attack Target | Am I in Range to Attack?  Yes = +100  No = -1000  Can I use my attacking skill?  Yes = +100  No = -1000 |

Every time the AI makes a decision, it calculates each decision it could make and finds which one has the highest score.

Using this table, say I was in range to attack and I could also use my ‘attacking’ skill the score would look like this:

|  |  |
| --- | --- |
| Actions | Total Score |
| End Turn | 0 |
| Move Towards Target | -900 |
| Attack Target | 200 |

In this instance, I would decide to attack my target as it had the highest score.

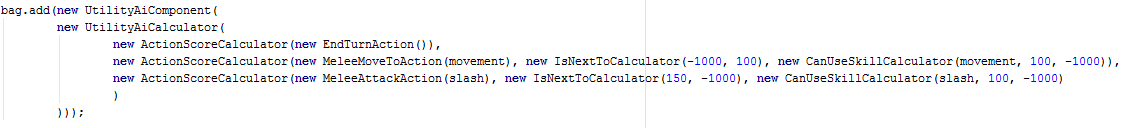
This is what those decisions currently look like in code form:

Figure 12: Building a simple AI using Utility Design

In my current implementation, I’ve restricted myself to using only binary calculations. Yes and No. However, a more advanced implementation, which I aim to do in future, also includes calculating decisions based on linear and exponentials scales when calculating scores.

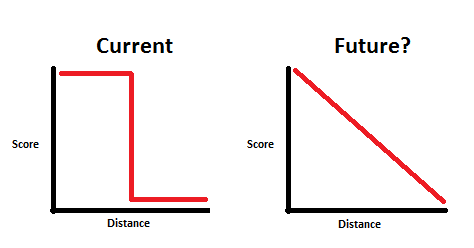


Figure 13: Comparison between a binary and linear calculation of score

# Prototype Description

## Gameplay

The prototype begins with a Menu Screen with a single start button.

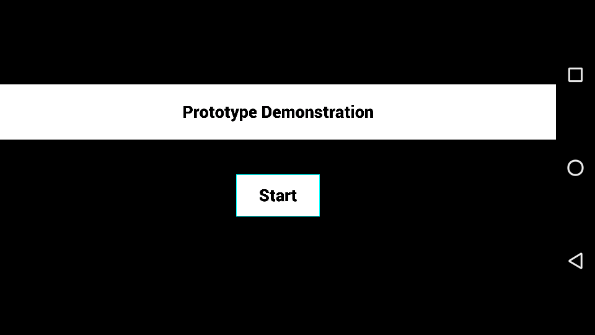


Figure 14: Title screen of the prototype

Upon starting, the player is placed in an arena where there are four enemies and s/he has control of two playable characters (The white squares with eyes).

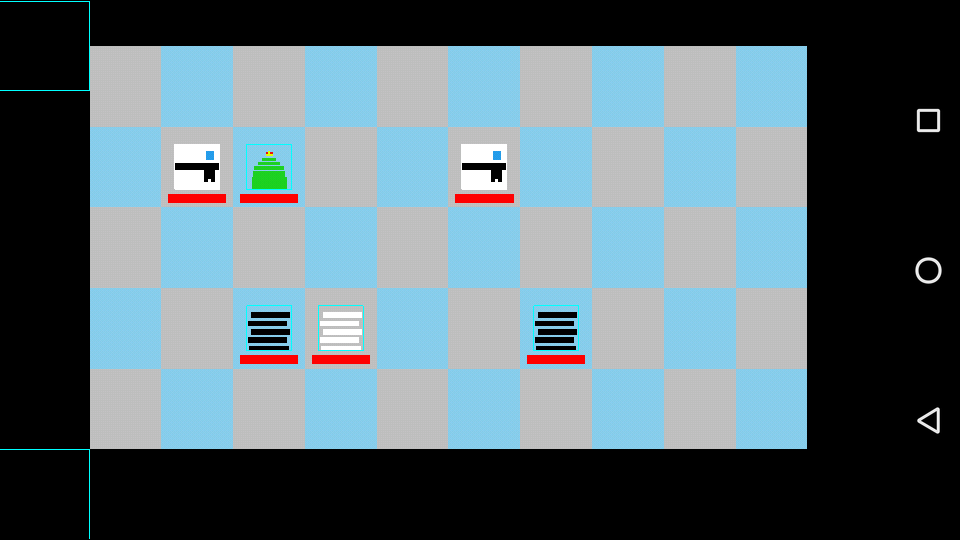


Figure 15: The beginnings of a fierce battle

The game starts on the player’s turn. When the player selects a playable character, that character’s skills appears below the game board. When a character is selected it fades in and out. In future, I will add an additional drawing on top of the character to further highlight which character is selected.

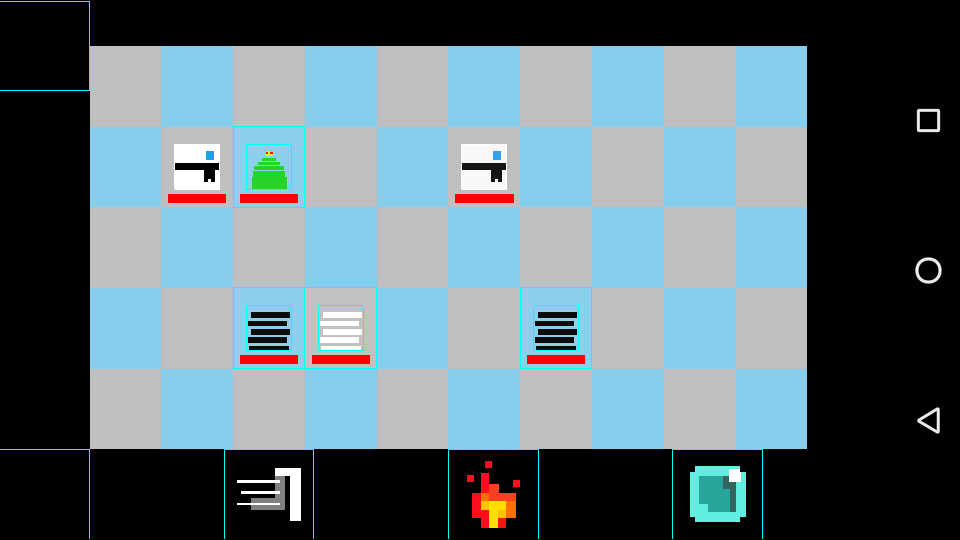


Figure 16: The character on the right has been selected

From left to right you have the skills ‘Movement’, ‘Fireball’ and ‘Frostball’. Movement allows a player to move and the other two fire a ranged attack at an enemy.

When a player selects a skill, targeting is created. When a player clicks on one of the white squares created, the selected skill is cast on that target location.

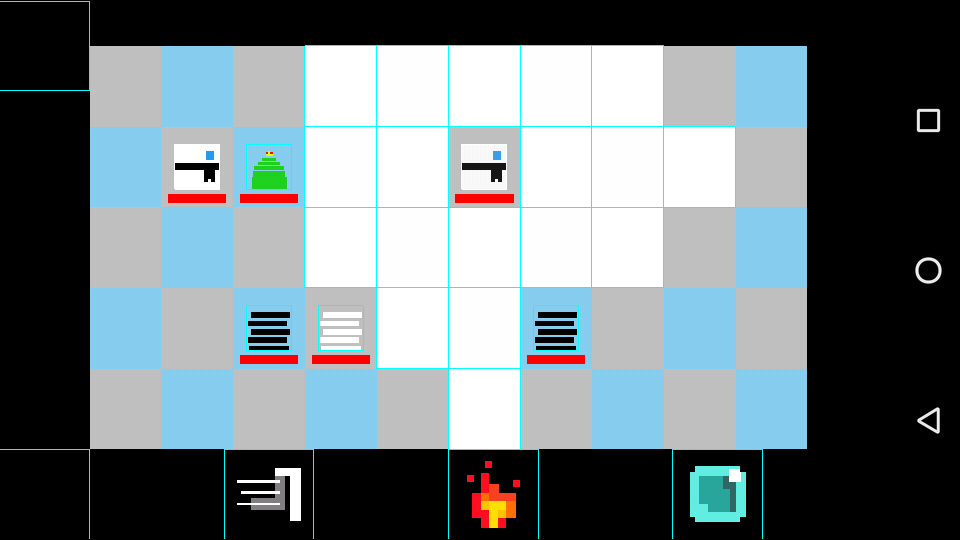


Figure 17 Targeting for the 'Movement' skill

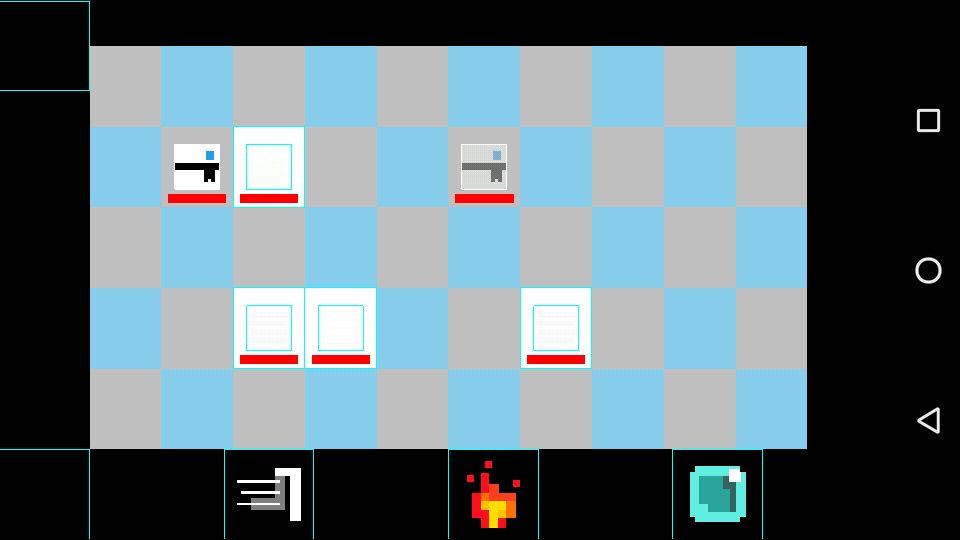


Figure 18: Targeting for the Fireball Skill

Skills work on a per-turn basis (although in future some skills will have a greater cooldown than one turn). Once a skill has been used it is unable to be re-used until the next turn.

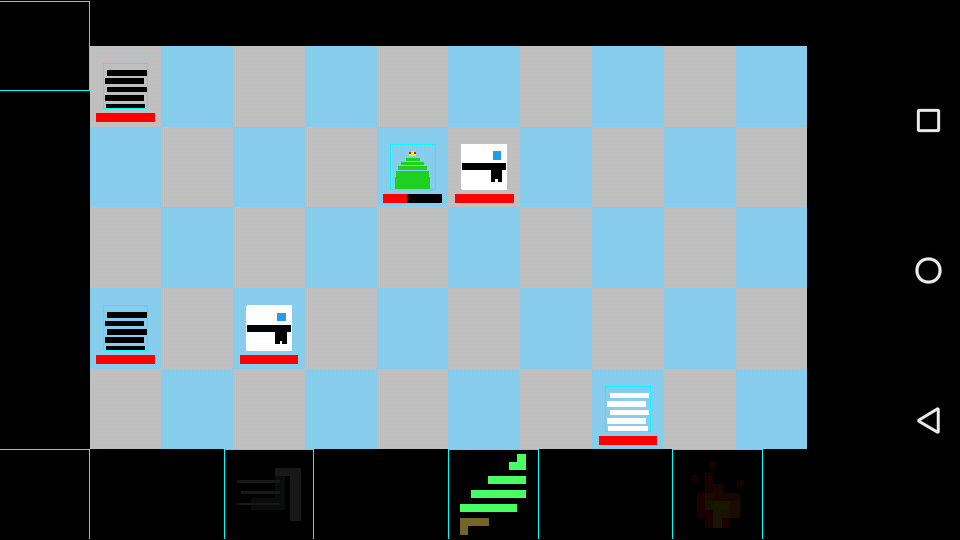


Figure 19: Only one more skill is available this turn for the selected character

There is currently no in-game description as to what the skills can do.

Once the player decides their turn is over they can push the end turn button (on the far bottom-left) to shift the current turn from player to enemy.

When it is the enemy turn, based on where the enemies are, their AI decides if they should either move or attack. Some enemies have greater movement speed than others and some enemies can shoot ranged attacks at other characters.

When all enemy moves have been expended the turn then shifts back to the player’s, whose skills will now be refreshed and available to be used again.

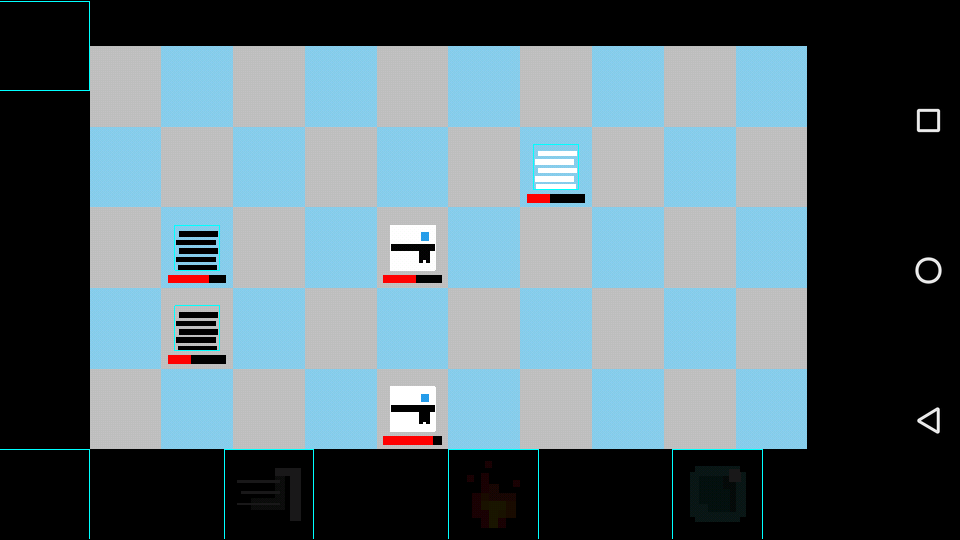


Figure 20 The battle a few turns in both enemies and players have been damaged

Once all enemies have been defeated, or all player characters have been defeated, the game displays a victory or defeat which then leads you back to the main menu.



Figure 21: Victory! All enemies have been defeated

## Usage of Components and Systems

To help showcase the current scale of the prototype I will now outline the key components and systems within the prototype as well as list the less important ones.

### Key Components

These are the components that will most likely be used by most if not all visible entities within the game.

|  |  |
| --- | --- |
| Component | Description |
| PositionComponent | Holds the x and y position of the entity. This component is used by every single entity that is drawn on screen. |
| DrawableComponent | Holds an in-depth description of text/textures that this entity uses to be drawn on screen. This description includes colour, scale, rotation, offset from position etc. |
| CoordinateComponent | Keeps track of where on the game grid the entity is, using a custom ‘Coordinate’ class. |
| PlayerControlledComponent | Identifies if the entity can be interacted with by the player |
| VelocityComponent | Holds an x and y value which corresponds to how fast an entity is moving. |
| HitBox/BoundaryComponent | Details the ‘area’ an entity occupies within the game space. Used to see if an entity contains a position or overlaps another entity. |

### Other Components

ActionOnTapComponent, ConditionalActionsComponent, OnDeathActionComponent, UtilityAiComponent, TargetComponent, SkillsComponent, BulletComponent, ExplosionComponent, HealthComponent, MovementRangeComponent, MoveToComponent, TurnComponent, BlinkOnHitComponent, FadeComponent, UITargetingComponent, ParentComponent, ChildComponent, DeadComponent, EnemyComponent, FriendlyComponent, ExpireComponent.

### Key Systems

These are the systems that are the most used throughout the prototype

|  |  |
| --- | --- |
| System | Description |
| MovementSystem | Affects an entity with both a Position and Velocity Component. Adds the Velocity to the Position to move the entity. |
| RenderingSystem | Handles the drawing of an entity using its textures, text and position. Also orders the drawing sequence of each entity in the system because drawing order affects how textures are displayed. |
| TileSystem | Keeps track of all entities that hold the Coordinate Component. Stores their id in a map to keep track of which spaces are occupied.  This system is used by many other systems that interact with the map |
| TurnSystem | Keeps track of all entities that take part in the turn by turn system. During the enemy turn, tracks whose turn it is and when it is over and when to activate the next enemy’s turn until it is the player’s turn again. |
| SelectedTargetSystem | Based on a player’s input determines which player controlled entity has been ‘selected’ and then creates which skills need to be shown to the player. |
| BoundsDrawingSystem | A system for debugging that draws the hitboxes and boundaries of entities that have those components. |

### Other Systems

ActionOnTapSystem, ConditionalActionSystem, BattleMessageSystem, BlinkOnHitSystem, BulletSystem, DeathSystem, EndBattleSystem, HealthSystem, FadeSystem, MoveToSystem, ParentChildSystem.

### Expansion

The beauty of ECS is that newer systems and components are created through necessity. If there ever comes a puzzle or piece functionality that cannot be handled by the systems of components available all I need to do is create a new component or system that can handle it. An example of this would be when I incorporate animation. I will create a component to hold the data for each animated frame and a system to track when to swap out the animation frames.

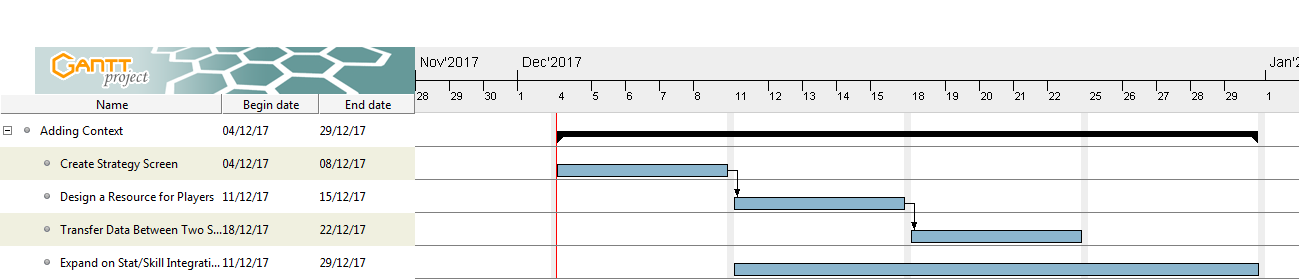
# Planning

The scope of the project has changed. At the time of the project plan I assumed I’d be making a game primarily focussed on just swiping to attack enemies. The game has now moved into a turn-based strategy that is going to involve more complicated AI and decision making. This means the plan needed to be changed.

Once again, these images can be found in a larger format within the repository

## December

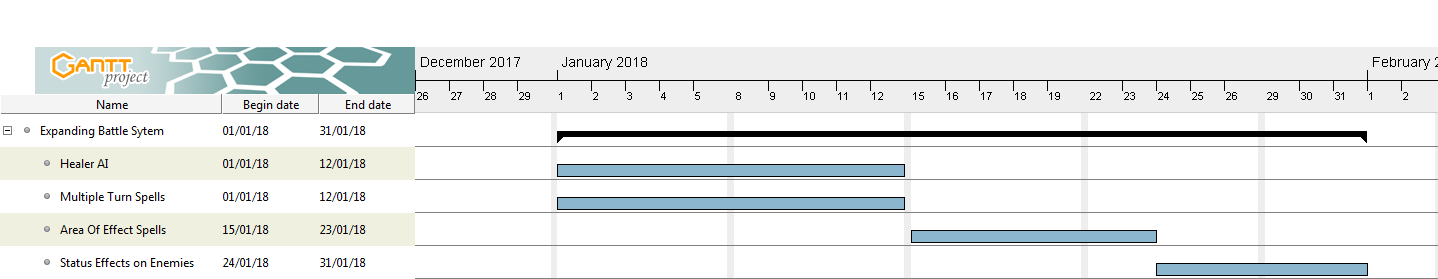
The focus in December will be to add ‘context’ within the current project. The prototype consists of a battle. The question is why? What happens after a battle? What triggers a battle? What happens if you get defeated? I hope to create answers to these questions over December.



This would require me to create new Screens that can pass information about the player between each other.

## January

Once context has been created I want to re-focus on expanding the depth of decisions both a player and AI can make and return to the battle system.



As January ends I would hope to be in a position where the battle system has examples of AI making decisions between attacking and healing, moving away instead of just moving towards and using skills that take a few turns to come off cooldown.

This would also be reflected for players as well. Their decisions should branch out from just using their skills until all enemies are gone; to the placement of their characters, what skills to use that may stop enemies from doing certain actions and when to use a skill that takes multiple turns to re-use.

## February Onwards

As the project continues further into second semester the plan will shift based on how far I’ve been able to both expand the context surrounding the battle system and the battle system itself.

The focus shifts from adding depth to the game to adding breadth. By this I mean expanding and reusing what’s there to increase the length it takes to complete the game. Using Pokémon as an example, the battle system is the same every single battle but the journey to complete the game takes you through numerous battles and takes a fair bit of time. But the same code is used every battle.

Other systems that are currently not important, such as systems that control sound and music would also be added around this stage.

As mentioned in the project plan, this project relies heavily on how players interact with the game. My speculations on future goals are malleable and may change.

# Bibliography

[11] Nystrom, Robert. *Game Programming Patterns*. Genever Benning, 2014.

[1], Rasmussen, Jakob. “Are Behaviour Trees a Thing of the Past?” Gamasutra Article, 4 July 2016, www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are\_Behavior\_Trees\_a\_Thing\_of\_the\_Past.php.

[2], Mark Dave, and Kevin Dill. “Improving AI Decision Modelling Through Utility Theory.” GDC Vault, 2012, [www.gdcvault.com/play/1012410/Improving-AI-Decision-Modeling-Through](http://www.gdcvault.com/play/1012410/Improving-AI-Decision-Modeling-Through).

[3], SolarChronus. “A\* Pathfinding Tutorial.” YouTube, YouTube, 28 July 2012, [www.youtube.com/watch?v=KNXfSOx4eEE](http://www.youtube.com/watch?v=KNXfSOx4eEE).

[4], Patel, Amit. “A\* Comparison.” Introduction to A\*, 2017, theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[5], Martin, Adam. “Entity Systems Are the Future of MMOG Development.” T-Machine.org, 3 Sept. 2007, t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/.

[6], Papari, Adrian. “Introduction to Entity Systems.” GitHub, 7 Feb. 2016, github.com/junkdog/artemis-odb/wiki/Introduction-to-Entity-Systems.

[7], Papari, Adrian. [github.com/junkdog/artemis-odb](https://github.com/junkdog/artemis-odb) GitHub, 2016,

[8], Gaul, Randy. “Spaces: Useful Game Object Containers.” Game Development Envato Tuts+, 14 Jan. 2014, gamedevelopment.tutsplus.com/tutorials/spaces-useful-game-object-containers--gamedev-14091.

[9], Apple Inc. “Entities and Components.” GameplayKit Programming Guide: Entities and Components, 21 Mar. 2016, developer.apple.com/library/content/documentation/General/Conceptual/GameplayKit\_Guide/EntityComponent.html.

[10] “Goals and Features.” Libgdx, libgdx.badlogicgames.com/features.html.