Final Year Project Interim Report

Full Unit - Interim Report

Building a Game

Submitted By Sulaimaan Bajwa

Evaluation of my progress so far in the final project critical to the completion of the 3rd year of

Msci (Hons) Computer Science with Software Engineering

Supervised by Hugh Shanahan
EPMS School, Department of Computer Sciences
Royal Holloway, University of London

1 Abstract

During the course of this project I will be attempting to design, create and deploy a video game. I hope to learn more about Unity as a game engine and how to develop and test with it, with the objective that it will be a strong foundation for my software engineering portfolio. Unity is a game engine rooted in the very foundation of the industry, helping developers create games for nearly two decades, and already contains many of the elements of important software design patterns I intend to use. I have chosen to create a Roguelike game due their current popularity and approachable- but challenging- nature. Roguelikes often have a repetitive cycle consisting of frequently reused modules, perfect for tackling with software design patterns in mind.

1.1 Aims and Objectives

2 Background

2.1 Video Games

Perhaps one of the main forms of software mass produced by teams and solo developers globally, video games frequently feature the newest techniques and sometimes require the frontier of consumer technologies to run. As a result, efficiency within the code of a video game is paramount. In this project, I seek to create a video game using software engineering techniques, creating an efficient and fun game. This requires intimate understanding of both software engineering and the genre or genres I plan to develop within.

There are many pieces of software that enable me to do this, one of which is the game engine Unity, which efficiently handles core elements of a video game, such as the game loop and physics interactions.

One possible solution to tackling the power required to run some video games has been the exporting of processing to external servers, sending only the frame back to the players system. This concept is the marriage of cloud computing with video games, and while it can be a great solution, many players can feel like they do not actually own the game they are playing, and at any time the cloud service could pull support for the game in question.

2.2 Roguelikes

As a genre, Roguelikes[1] get their name from a game released about 1980 as free shareware. The game was called ‘Rogue’ and was an ASCII art visual game about navigating several floors of a dungeon to find a relic. Many elements of Rogue are still used to this day within new roguelike games. These features include permadeath, items and a room-by-room iterative playstyle. Back then, items were noted as ? symbols, the player character was an @ symbol, and enemies were capital letters, all 26, each of which specified a different creature.

The current iteration of the genre sees several of Rogues features still in use, namely a variety of enemies, traversing and descending a multi-layered dungeon, collecting items that enable certain mechanics, and permadeath. All of these mechanics are important elements that define the genre, but some have been altered since their first appearance in Rogue. Permadeath in particular is a simple mechanic that has been altered sometimes on a game-by-game basis. While the definition of permadeath remains the same - the death of a character ends the run, the character cannot be revived to continue or restart the run - there have been more engaging implementations of it that allow the user to maintain a sense of direction and progression that keeps them playing the game, run after run. If the player dies more frequently then they progress, the player generally loses interest.

3 Technology Theory

3.1 Unity and Game Engines

To facilitate production of my game, I opted to use a game engine, ‘a software framework designed for the creation and development of video games’ [2]. These pieces of software form a toolkit that allows for the largely repetitive low-level programming that renders graphics, calculates physics, handles and integrates inputs and scripts, and runs the game loop to be reused incredibly quickly. While parameters within these modules can often be tweaked, video games can be created with these fundamentals untouched. Live-support game engines are frequently updated to bring optimisations for games running on them, as well as features for developers to utilise.

Building on top of an engine means that a developer now only has to provide assets and scripts to successfully create a game. These scripts are largely the focus of my project, as I can use software engineering techniques, such as the use of design patterns, to solve common issues, like reducing the technical resource requirements for my game.

3.2 Quaternions

While my game will be 2D, there are two primary variants of 2D games: Top-Down and Side-Scrolling. Most forms of 2D rotation in Unity are designed to support side-scrolling games, while my game is Top-Down. As a result, I have to utilise 3D rotation methods, which include the other axis I need. One such form of rotation is Quaternions, they solve a very specific problem that I encountered during the creation of my game: gimbal lock.

3.2.1 Gimbal lock

Consider the pitch, roll and yaw of an object. These are descriptors used to identify the rotation of an object in 3D space, and while generally they do a good job in doing so, there are very specific cases in which this gimbal setup can lock itself out of proper rotation, physically. In the case that 2 of the 3 rotational axes line up, a gimbal lock occurs. In which one of the rotational axes can no longer be added to the rotational matrix and have an effect. This is evident within the mathematical depiction of a rotation using matrices. In the below example I will use images of matrices taken from an external source[] for the purposes of explanation.

$$\begin{pmatrix} C_\alpha C_\beta & -C_\beta S_\alpha & S_\beta \\ C_\gamma S_\alpha + C_\alpha S_\beta S_\gamma & C_\alpha C_\gamma - S_\alpha S_\beta S_\gamma & -C_\beta S_\gamma \\ -C_\alpha C_\gamma S_\beta + S_\alpha S_\gamma & C_\gamma S_\alpha S_\beta + C_\alpha S_\gamma & C_\beta C_\gamma \end{pmatrix}$$

A 3x3 z-y-x rotation matrix, where C_n and S_m are Cos(n) and Sin(m), using angles α, β, and γ

If I were to substitute in π/2 as β, where π/2 is a 90degree rotation in radians, Cos(π/2) = 0 and Sin(π/2) = 1, resulting in the following matrix.

$$\begin{pmatrix} C_{\frac{\pi}{2}}\, C_\alpha & -C_{\frac{\pi}{2}}\, S_\alpha & S_{\frac{\pi}{2}} \\ C_\gamma\, S_\alpha + S_{\frac{\pi}{2}}\, C_\alpha\, S_\gamma & C_\alpha\, C_\gamma - S_{\frac{\pi}{2}}\, S_\alpha\, S_\gamma & -C_{\frac{\pi}{2}}\, S_\gamma \\ -S_{\frac{\pi}{2}}\, C_\alpha\, C_\gamma + S_\alpha\, S_\gamma & S_{\frac{\pi}{2}}\, C_\gamma\, S_\alpha + C_\alpha\, S_\gamma & C_{\frac{\pi}{2}}\, C_\gamma \end{pmatrix}$$

The substitution before simplification

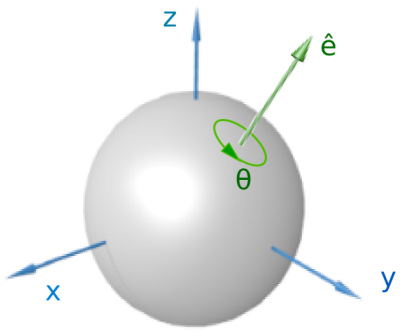
$$\begin{pmatrix} 0 & 0 & 1 \\ C_\gamma\, S_\alpha + C_\alpha\, S_\gamma & C_\alpha\, C_\gamma - S_\alpha\, S_\gamma & 0 \\ -C_\alpha\, C_\gamma + S_\alpha\, S_\gamma & C_\gamma\, S_\alpha + C_\alpha\, S_\gamma & 0 \end{pmatrix}$$

Simplified substitution

The result of this substitution is an inability to substitute in any more β values, ergo the inability to rotate one of the axes at all. This is a gimbal lock mathematically, and without external force physically, this cannot be corrected for. Quaternions do not suffer from this issue, and so are a better option for me.

3.2.2 Quaternions For Rotations

The use of quaternions hinges on Euler's Rotation Theorem, where the rotation of an object in 3-dimensional space can be defined by a single rotation θ about an axis defined by a vector from a fixed point. This is directly why a quaternion's notation is q = (w,r), where w is a scalar value representing the angle or rotation, and r is a vector, representing the line around which the rotation occurs. This is evident in the following image, where θ is the rotation scalar, and ê is the Euler Axis Vector.



3.3 Pathing and Navigation

4 Product Research

In order to effectively create a game, I must break down other games in the genre to see exactly what the players find appealing, where the selling points lie, and what's missing. From this, I should be able to get some understanding of what I can do to place myself in a competitive market, in a position that sets me apart from the rest.

4.1 Cult of the Lamb

4.1.1 Similar Features

Almost top down/2.5D graphics
Simple attack gameplay
Random generated dungeons
Boss fights
Generated rooms, with unique features

4.1.2 Distinguishing Features

Cartoony, Captivating Graphics
Integrates colonySim genre gameplay elements
Cult-Religion themes
Cartoon violence and light gore
Very detailed, well-liked soundtrack that lines up with key actions made in game

4.1.3 Key Notes

Blends in elements of another genre, in order to cut the repetition of runs and death.
Complex dungeon generation, rooms vary sizes and shapes, with connecting safe zones.
Captivating artstyle and interactivity really makes the game shine.

4.2 Hades

4.2.1 Similar Features

Randomly generated map
Boss fights
Large variety of enemies
Levelled progression, multiple floors of dungeon
Character ability building in the form of boons

4.2.2 Distinguishing Features

Traps/other forms of damage outside of combat
Greek mythology theme
Incredibly detailed comic-esc artstyle
Full voice acting for distinct characters
Permanent upgrades that persist between runs
Story forward
Huge variety of types of weaponry and damage forms
Isometric instead of grid 2/2.5D
Incredibly large, complex dungeon room generation

4.2.3 Key Notes

Much higher graphical quality than I can achieve in the time I have, far out of scope.
Much more complex generation than I can achieve in the time I have, far out of scope.
Boom system similar to the item system I have planned.

4.3 Enter the Gungeon

4.3.1 Similar Features

Has a wide variety of weaponry, also works in place of items, generally
2D Pixel art graphics style
Has a variety of unique, but similarly themed enemies
Random generated dungeons, descending floor system

4.3.2 Distinguishing Features

Weapons take on the role of essentially all itemisation
Several starting characters to choose from
Uniquely sized rooms, with connecting safe zones
Expansive shop system
Additional resources to provide safety, and dodge system
Story driven
Falls under a sub-genre of roguelikes, called bullet hells, designed to provide a seemingly overloading experience of learnable patterns.

4.3.4 Key Notes

Rather similar to what I have planned, but with a much more granular scope, incredibly refined compared to my intended final product.
Truly unique enemy abilities makes for an incredibly unique run each time.
Shop system would be interesting to introduce, and doesn't seem complex to do so, may add that to scope if I finish intended systems early

4.4 Binding of Isaac

4.4.1 Similar Features

Simple item system, with active, passive and culminating items
Boss fights
Templated rooms, and enemies
Procedurally generated box rooms, forming a level of a dungeon
Player descends to progress toward the final boss

4.4.2 Distinguishing Features

Entirely unique enemies, often with effects the player can obtain through items
Unique bosses
Story driven
Roughly set final bosses

4.4.3 Key Notes

The major inspiration for my own game, this contains very similar aspects to my intended scope
I hope to improve the cohesion of items to the theme of the game, over what was done in The Binding of Isaac

5 Software Engineering

5.1 Design Patterns in Video Games

As one of many forms of good practice, modularity, and therefore decoupling, of code is quintessential to the ongoing work of a software developer. Being able to re-utilise elements of your portfolio again makes you a valuable developer, bringing with you the correct tools for the job at hand. My game is no different, and if I were to develop games in the future, or even allow access for another individual to see my source code, the game would be designed in such a way that elements could be reused. Design patterns also seek to create a readable, maintainable and efficient codebase, such that another person could easily get to grips with the code if need be. Below are the efforts I have made to incorporate design patterns into my work, by breaking down how and why I use each one.

5.1.1 Visitor

Visitors exist to decouple a repeated function from every participant of some structure, allowing a single traversing object to assert the function on each object instead. In the case of my game, projectiles act as visitors, in a much less obviously connected structure of enemies to one another. Each enemy exists as a child of the enemy controller, also acting as the FlyWeight. Any projectile sent by a player would effectively traverse this structure.

Projectiles, such as the bullets shot by both enemies and players, operate as visitors, even though they lack the exact visit/accept functions. Unity handles collision between any two objects as, essentially, visiting. When an object collides with a second object, both gain the ability to refer to each other. This allows the projectile to hold all the functions it needs to afflict anything onto any given object it collides with, and potentially multiple objects, if the bullet has the ability to pierce entities. This decouples the objects from each other, utilising visiting projectiles to interact with any given object, so long as it has a health value. Once the purpose of the bullet is served, it can be destroyed to preserve resources.

5.1.2 Factory

The purpose of a factory is to separate the need to mechanism by which you instantiate a set of objects. These objects could be in large numbers, or just rather convoluted to make individually within a script. As a result, a script is written to specifically instantiate a type of object, with minimum required arguments going forward.

All prefabs are generated as children of a class that has a designated factory, spawning in objects, and then passing on all required specifics for that class to operate. A good example of a factory in my code will likely be the enemy factory, which spawns in enemies, and assigns them their Object Types. This enables easy creation of enemies from prefabs, and allows me to quickly summon enemies using debugging tools if necessary.

5.1.3 FlyWeight

In order to significantly streamline all entities spawned into the scene from prefabs, I will need to remove all the shared components and implement them within some parent spawning class, working as a means to have just a single instance of unchanging data, rather than spawn every entity with its own hard copy of identical data.This is called a FlyWeight. It follows from the idea that duplicate code is wasteful, by essentially decoupling unnecessary elements from the object they should relate to, and making each object just refer to the single instance. An example of which within my game will likely be enemies, of which there are a few types, but they all will share common variables, such as damage values, the location of the target, and their movement speed.

5.1.4 Type Object

In order to specialise objects of the same type, such as enemies, or projectiles, I can use the Type Object design pattern, providing an additional script to each object of a given type, importing all the specific functions designed for that subtype of object. There are currently 3 types of enemies planned for my game, a melee chaser, a shooter, and a stationary shooter. Each of these types will be specified by their own script.

5.1.5 Game Loop

Game loops are one of the most critical pieces of game design patterns to have been formed. It seeks to solve the issue of the broad variety of hardware that game developers often need to accommodate for. Most hardware is designed for many purposes, and as a result, to be able to deal with the differences in processor clock-speeds, the game executes instruction on a unified ‘Tick’ generally occurring a certain number of times a second. This way, the game doesn’t actually increase/decrease in pace based on the speed of the processor running the game. The game loop system is provided by the game engine itself, and I can tie instructions to it using the update methods available to me.

5.1.6 Update Method

In order to tie scripts to game ticks, I must use the update functions. There are a few available to me that allow me to control the order in which scripts are executed. The first, most prioritised function is the awake function, which happens as soon as the object is instantiated, before its initial start call. The start call then runs, frequently setting up initial script relationships with the controller, and any other required objects. The update and fixedUpdate functions are then called every tick; while update is tied to the graphical frame rate of the game, which can change due to processing limitations, fixedUpdate is always executed every tick. Every class within my game operates alongside the Game Loop, and so each class must implement at least one of these Game Loops updater.

5.2 Testing a Video Game

5.2.1 Unit Testing

Unit testing seeks to take a small, manageable element of code and provide it data to process, asserting that the result from that processed data is what was expected. In a typical environment, Unit testing is automated, to ensure that a developer does not need to manually test individual aspects of their code every time a change is made to a pathway that uses that section of code. This is a highly effective and efficient way of testing code, and frequently squashes bugs resulting from incorrect inputs, incorrect outputs, or incorrect processing.

5.2.2 Identified Issues

In order to conduct unit testing, test inputs have to be created in the manner that data is expected to be received by the function. Writing an identical function that takes in a differently formatted data could be seen as an approach to combat having to write incredibly complex data, however, you are then testing a unique piece of code, as it has to handle the test data differently. As a result, this is a flawed approach. If constructing valid test data in an identical format is an issue, it becomes impossible to unit test. That is unfortunately largely the case with my game thus far.

5.2.3 Current Approach

To effectively test functionality along a similar idea to unit testing, I can create a series of testing maps/environments within the game, allowing me to engage each element of a script in an orderly way. I must avoid attempting to test several features at once, as I may struggle to narrow down the culprit lines within my scripts. For every piece of functionality I write, I will need to write tests for all interactions between that function and other game objects or functions. This will be particularly important, I foresee, for the intended item system.

6 Progress Evaluation

6.1 Project Plan Course

The project plan was both over-optimistic and under-considered in many ways, there are many elements I now have to create, and many I could not create that were initially planned. Such elements include research, code, and submission writing.

6.1.1 Materials (Section 3 in Project Plan)

Of the reports, I have created:

Initially Planned

- Design Document - Partial
- Major Element of a Roguelike Game - Partial
- Current Major titles - Partial
- List of required Algorithms and Structures - Partial

Unplanned

- Design Patterns
- Testing
- Quaternions
- Refactor Plan

Of the proof of concepts, I have created:

- Player Motion
- Enemy and Damage System

Of the Testing Tools, I have created none as I am currently testing unaided by custom tools, but intend to still produce each testing tool in the near future.

6.2 Encountering Risks

6.2.1 Scope Creep

I failed to accommodate for refactoring my code, actually implementing design patterns correctly required refactoring of produced code at times, as I attempted initially to pass tests, and then create more elegant, modular code. As a result of my initial plan being very intensive, I have had to push back the item system into the second term, and work on refactoring my code, so that I can demonstrate my use of design patterns. In this sense, incorporating design patterns is the additional scope I had not considered, because I assumed my code would be well designed from the start, without actually producing an initial software design document or diagram.

To combat this, I will produce a software design document and diagram in order to provide myself a basis on which to write code, in order to reduce the amount of time I need to spend refactoring, and I will additionally insert code analysis and refactoring time into my term 2 plan.

Additionally, I have realised that I have no GUI planned for the game, which is a quintessential element of any video game. I will have to implement a GUI in order for the player to start the game, a GUI for the player to exit the game and a GUI to act as a loading screen, while the next level of the dungeon generates. This should be planned for between term 1 and 2, to make up for time I may not have in term 2.

6.2.2 Lack of Slack - Unaccounted For

I frequently suffered from burn-out while trying to complete the tasks I had set for term 1, and as a result of this, I have not been able to do tasks effectively. I failed to accommodate other activities outside of the project, consuming large amounts of time.

Considering the 3 major elements of a project, scope, quality and time, increasing one of these elements will lead to a decrease in at least one of the other. In order to provide myself with much needed time between tasks, I will create a much less condensed term 2 schedule, by incorporating time over the winter period between terms 1 and 2. This will strain me less during term time, and will also provide extra time I initially did not consider to be used within the project, essentially creating time without adjusting scope or quality. Most projects do not have this luxury as it is not linked to academic terms, however I will utilise any time I can to provide my intended scope, as I believe the planned scope contains all major elements of a roguelike game, and to produce less than the intended scope would mean to produce a low-quality roguelike game.

6.3 Revised Schedule

(Academic) Week	Milestones	Specific Notes
Winter Break		
12/12/2022	Construct GUI diagrams	Part of missing required elements
	Start creating UI elements within Unity	
19/12/2022	Link GUI to the beginning and escape menu of the game	Likely need to sever update() functions
26/12/2022	Build item proof of concept	
02/01/2023	Start Research on random generation	
Term 2		
Week 17 & 18 09/01/2023 & 16/01/2023	Create algorithm that randomly generates map	Major complexity of project, see report on PCG
Week 19 23/01/2023	Create template rooms to insert into randomised maps	Bulk of final programming, allowing me to tunnel-vision and complete the code, mitigating a lack of report detail. Large majority of this section will fall under the crucial design patterns section of my final report, important to constantly reflect in my diary.
Week 20 30/01/2023	Finish templates, add them to generation algorithm	
Week 21 06/02/2023	Work on character visual modification algorithm	
Week 22 13/02/2023	Finish character visual	
	Work on enemy set templates	
Week 23 20/02/2023	Integrate the enemy randomisation into algorithm, using both regular and boss enemies	
Week 24 27/02/2023	Draft up Final Project Report	Important for feedback on major elements of the final report
Week 25 06/03/2023	Start finalising and testing programming side of project for submission	Squash major bugs, ensure code runs as expected, on target systems.
Week 26 13/03/2023	Finish Final Report draft and start work on feedback	Crucial for an optimal Final Report
Week 27 20/03/2023	Finish report and project for submission	March 24th Final submission for all project documents

7 Bibliography

7.1 Background

[1] https://web.archive.org/web/20150217024917/http://www.wichman.org/roguehistory.html

7.2 Unity and Game Engines

[2] Technologies and Innovation: Second International Conference, CITI 2016, Guayaquil, Ecuador, November 23-25, 2016, Proceedings. Germany, Springer International Publishing, 2016. Page 146.

F. Messaoudi, G. Simon and A. Ksentini, "Dissecting games engines: The case of Unity3D," 2015 International Workshop on Network and Systems Support for Games (NetGames), 2015, pp. 1-6, doi: 10.1109/NetGames.2015.7382990.

7.3 Quaternions

https://www.allaboutcircuits.com/technical-articles/dont-get-lost-in-deep-space-understanding-quaternions/

7.4 Design Patterns

Ampatzoglou, A., & Chatzigeorgiou, A. (2007). Evaluation of object-oriented design patterns in game development. *Information and Software Technology*, *49*(5), 445-454

https://www.habrador.com/tutorials/programming-patterns/

http://gameprogrammingpatterns.com/game-loop.html

Sharif, M., Zafar, A., & Muhammad, U. (2017). Design patterns and general video game level generation. *International Journal of Advanced Computer Science and Applications*, *8*(9).

Gamma, Erich. Design Patterns Elements of Reusable Object-Oriented Software. 37th printing. ed., Addison-Wesley, 2009.