Contents

[Introduction 3](#_Toc417215936)

[Overview 3](#_Toc417215937)

[Aims & Objectives 3](#_Toc417215938)

[Possible Extensions 3](#_Toc417215939)

[Research 4](#_Toc417215940)

[Tools 4](#_Toc417215941)

[Unreal Engine 4 4](#_Toc417215942)

[Cry Engine 5](#_Toc417215943)

[Implementation Techniques 5](#_Toc417215944)

[Methodology 7](#_Toc417215945)

[Implementation 9](#_Toc417215946)

[BSP Map Generation 9](#_Toc417215947)

[Square-ception Map Generation 10](#_Toc417215948)

[Findings & Discussion 11](#_Toc417215949)

[Conclusion 12](#_Toc417215950)

[References 13](#_Toc417215951)

* INTRODUCTION
  + Aims & Objectives
  + Context
  + Structure of write-up
  + Conventions / Exclusions etc
* LIT REVIEW / RESEARCH
  + Tools (2 pages?)
    - Unity vs Unreal/Cry Engine
  + Implementation Techniques (2 pages?)
    - BSP, Prefab sections, other
  + Existing projects (2 pages?)
* METHODOLOGY / IMPLEMENTATION
  + Context paradigm
  + Constraints
  + Design Strategy
  + Map generation (3-4 pages?)
    - BSP vs Overlapping Squares
    - UML
  + Puzzle implementation (3 pages?)
    - UML
* FINDINGS / DISCUSSION
* CONCLUSION
* REFERENCES / APPENDICES

# Introduction

* Aims & Objectives
* Context
* Structure of write-up
* Conventions / Exclusions etc

## Overview

Within this project I aim to create a dynamically generated puzzle level within an industry standard game engine. Upon each playthrough the layout of the level and the placement of puzzles will be different with potentially limitless possibilities.

This differs from my initial proposal which focused more on the usability aspect of providing a dynamicly generated environment without the use of traditional combative gameplay elements. Deciding how to guide players through the level and convey objectives without explict instructions. An obvious non-combative gameplay element are puzzles which I chose to incorporate as the core gameplay concept within the game environment.

Upon further investigation I decided to focus on the procedural aspect of generating an environment and the puzzle that will populate the level rather than player usability.

## Aims & Objectives

* Create a self-contained open world level with a definite start/end condition
  + Use simple player controls for movement and interaction
  + Procedural environment layout
  + Relevant puzzle elements with clear reasoning and resolution
* Dynamic map generation
  + Investigate most suitable methods for procedural map
  + Implement and compare different methods
* Dynamic puzzle generation
  + Procedural layout of puzzle elements (locked doors, switches, keys etc.)

## Possible Extensions

* Implementation of weather or day/night cycle
* Basic Non-playable characters (NPC) implemented within the world
* Controller support for easier user input

# Research

* Tools (2 pages?)
  + Unity vs Unreal/Cry Engine
* Implementation Techniques (2 pages?)
  + BSP, Prefab sections, other
* Existing projects (2 pages?)

## Tools

To determine which industry standard game engine would be the most suitable to use as a development platform for this project I compared three popular engines; Unreal, Cry Engine and Unity. With my previous knowledge of these engines being limited I took into account factors such as breadth/depth of documentation as well as the features of the engines themselves.

Each game engines bares stark similarities in their “What you see is what you Play” (WYSIWYP) development environment, which aided the familirization of each program. While their user interfaces may present tools differently they each possess very similar features, including node based scripts and use of prefabricated assets being two key aspects that I will utilize.

As a basis for my comparison I created a simple game environment within each engine to understand the capabilities of each engine and its suitability for the intended project. The levels created may be basic but allowed for a greater understanding of each engine’s toolset which subsequently led to the definitive choice of engine. For each mentioned engine I used only what was available free to use for educational or non-commercial purposes.

### Unreal Engine 4

Unreal Engine 4 (UE4) caters towards high visual fidelity primarily used for AAA titles but recently Unreal have strived to make their software more accessible to new developers, particularly for mobile deployment. This is presumably due to Unity’s success in this area but allows students and enthusiasts to begin developing using Unreal engine for free without previously costly requirements.

While upon first impression the program is rather overwhelming with a vast collection of tools and steep learning curve it still allows for quick rewarding environments using the pre-packaged assets. It is clear that Unreal have produced a powerful engine capable of large highly-detailed scenes. Actor components are initialized within the game environment as the most common gameplay class used within UE4. These Actors are C++ classes which can be extended or customized through either Unreal’s Blueprint scripting or any generic C++ IDE.

Although my experience with C++ is not as extensive it’s clear that UE4 requires the greater low level control particularly in highly detailed and complex projects. Despite this the Blueprint scripting interface provides access to most fundamental features without needing to delve too deep into the underlying C++ code. Blueprint script components present functions as visual nodes which although useful can become a bit convoluted within more complex classes. I found these scripts were easier to write within C++ and use the Blueprint script view as a visual representation of the program.

ADD MORE ONCE MADE UNREAL ENGINE LEVEL AND KNOW MORE SHIZ

### Cry Engine

Much like Unreal Engine 4 it seems Cry Engine is very much geared towards graphical fidelity particularly producing beautiful outdoor environments with highly detailed natural elements such as water and foliage. Though while the latest UE4 and Unity released with a free to use non commercial version the latest Cry Engine is offered on a monthly subscription basis or as a full costly license. Fortunately Crytek released a free CryEngine SDK of an older unsupported version which is the software used within this comparison.

The initial interface of Cry Engine is similar to the layout of the other engines but it becomes immeditely clear the extent of visual and rendering tools available. The engine is very focused on graphical fidelity to often produce near-photorealistic results using the depth of real-time lighting and rendering options. Along with bundled realistic vegetation and water systems it allows for very impressive scenes relatively easily.

Despite the impressive results I decided against using Cry Engine as my project does not require such depth of graphical options that the engine clearly emphasizes. Cry Engine proves itself as a powerful engine with much potential but the core features I require are included within the other engines. Another deciding factor was restrictive usage options of the latest versions which is not an issue with either Unreal Engine or Unity.

WRITE MORE MAYBE WHEN I DO CRY ENGINE TINGS

### Unity

As the most accessible engine with ever growing popularity within industry Unity was impossible to ignore. While in the past it was more associated with mobile game development and less complex projects, compared to the other engines, it hasn’t only grown in popularity but also in its wealth of features. Each itteration of the engine has grown the toolset to match that of it’s AAA engine competitors, all while being highly portable to a variety of platforms. The free non-commercial version known as the “Personal Edition” includes the engine with all features albeit some specialised analytics only required within professional context. This already made Unity a tempting option compared to the somewhat restricted versions offered by it’s competitiors.

The WYSIWYG interface is very much similar to the other engines which helped familiarization and the large amount of concise documentation made for an enjoyable learning curve. Unity uses predominently C# scripts as a basis for gameplay logic which is contrast to the comparison engines use of C++. Elements within the game environment are added as GameObjects which contain a set of components to control their properties, these work much in the same way as Actors within UE4. GameObjects can then be saved as Perfab assest which allows the reuse of these elements easily throughout the project. Prefabs can act as a template or base asset whereby each instance will inherit the assigned components however individual instances can be modified seperately.

The level of abstraction provided by prefabs in Unity is very well suited for my intended project since procedural generation is primarily through intelligently reused elements. Unity quickly distingushed itself as a solid platform focusing on function rather than form allowing for relatively quick results but with enough depth to expand in interesting ways. As I discovered more features in Unity I was constantly seeing the valid applicaibility to my project.

NEED MOAR COMPARISOOOON?

## Implementation Techniques

In terms of implementing a dynamically generated level with dynamic puzzles the structure of how to handle these procedural elements becomes incredibly important. Clear rules and context have to be applied to prevent unexpected results. For this reason the first task is to define a clear procedural environment generator which will be extended to handle dynamic puzzle inputs.

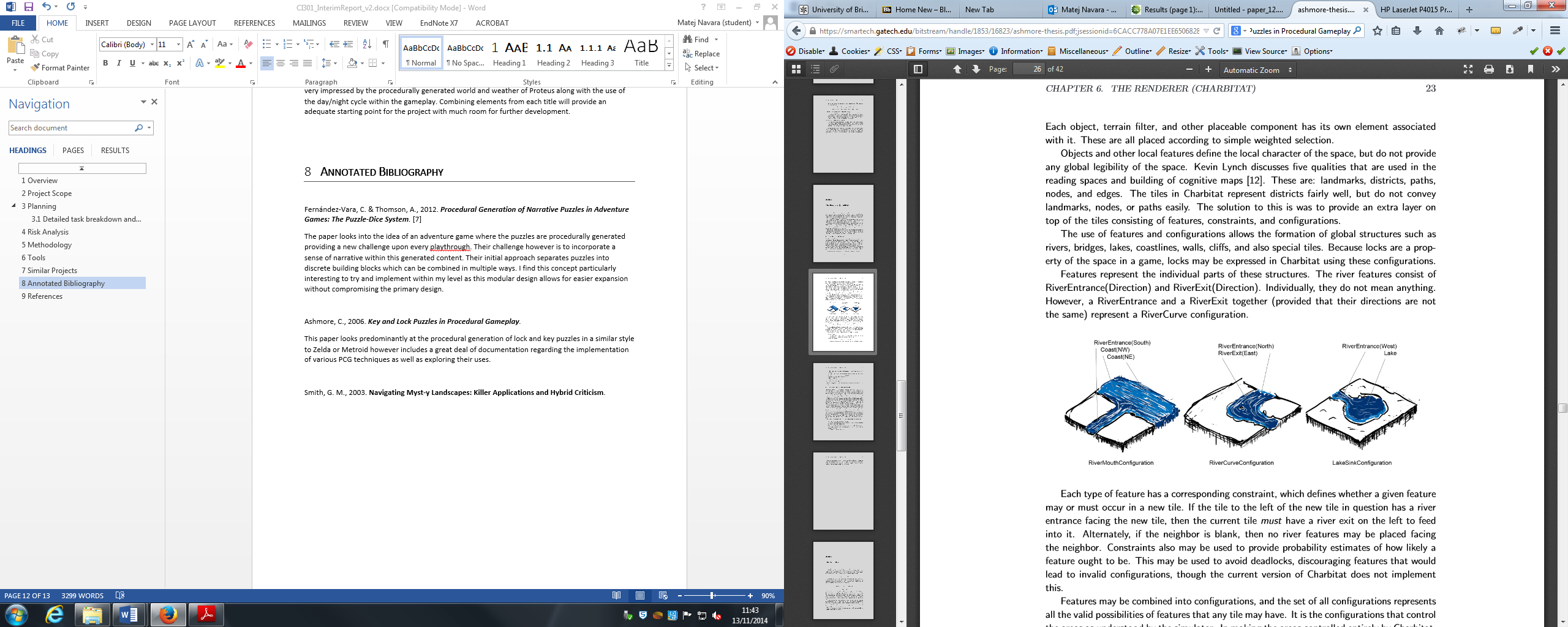
BSP tree, Prefab tiles,

### Prefab Tiles

The use of pre-fabricated tiles with defined connection points.

Ashmore, C., 2006. ***Key and Lock Puzzles in Procedural Gameplay****.* [8]

This paper looks predominantly at the procedural generation of lock and key puzzles in a similar style to Zelda or Metroid however includes a great deal of documentation regarding the implementation of various PCG techniques as well as exploring their uses. An interesting topic is the concept of zoning for procedural environment generation (see below) which uses constrained configurations of features such as rivers, cliffs and other zones to create dynamic environment formations. This is another element to explore within my project.



### BSP Tree

The use of binary space partitioning to divide up parent nodes into sibling nodes.

# Methodology

* Context paradigm
* Constraints
* Design Strategy

The implementation of my project comprises of two major components, the level generation and puzzle generation mechanics. Because these two tasks are intrinsically linked they will often be developed in tandem and change throughout the project.

This suited the principles of Rapid Application Development (RAD) to break up the project into smaller segments and quickly construct prototypes will help to identify important issues early on and cater to changing requirements based upon these findings.

Rapid Application Development emphasizes flexibility and adaptability whereby the knowledge gained throughout the project leads to changing requirements which are fed back into the next iteration. This differs from more linear methodologies such as the popular waterfall model which focuses heavily on specification and planning before any implementation takes place.

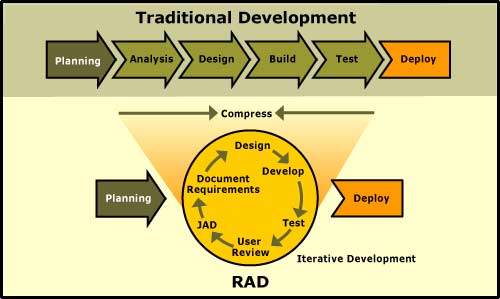


Figure 1: RAD vs Traditional development - http://www.clientsolv.com/images/DevModels.jpg

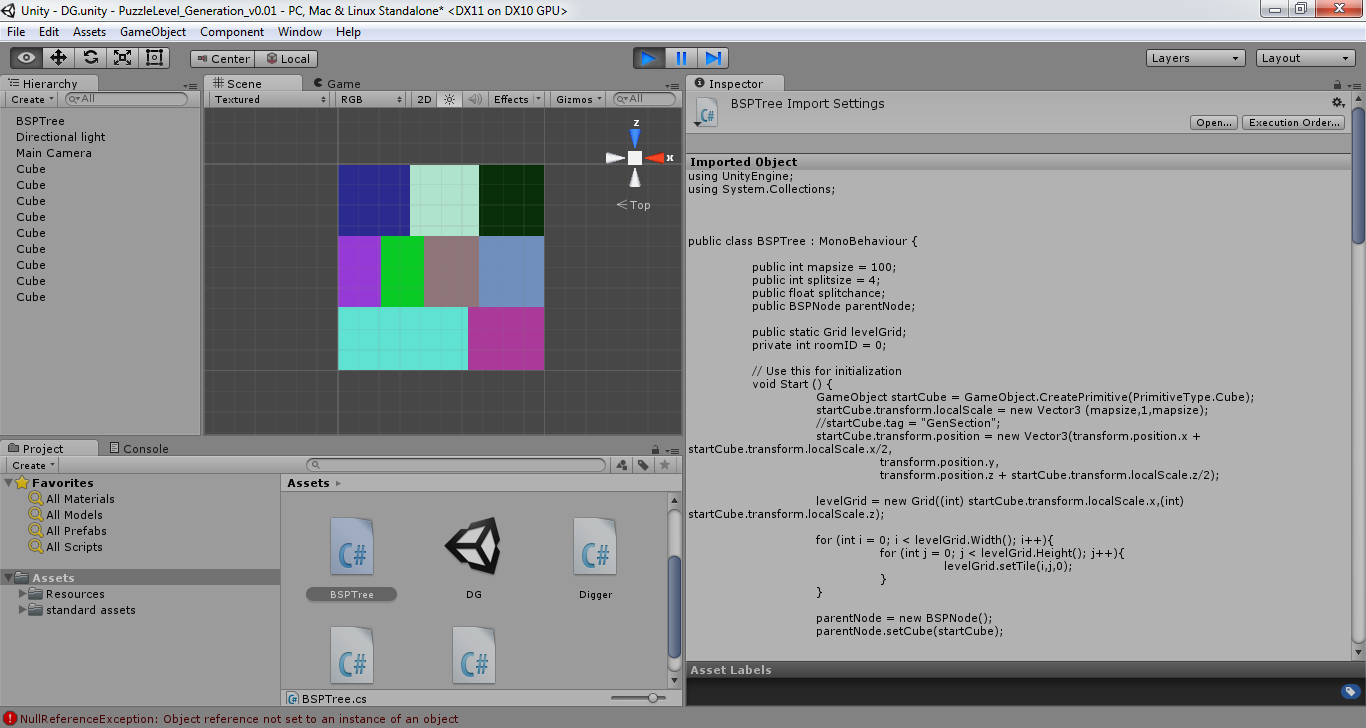
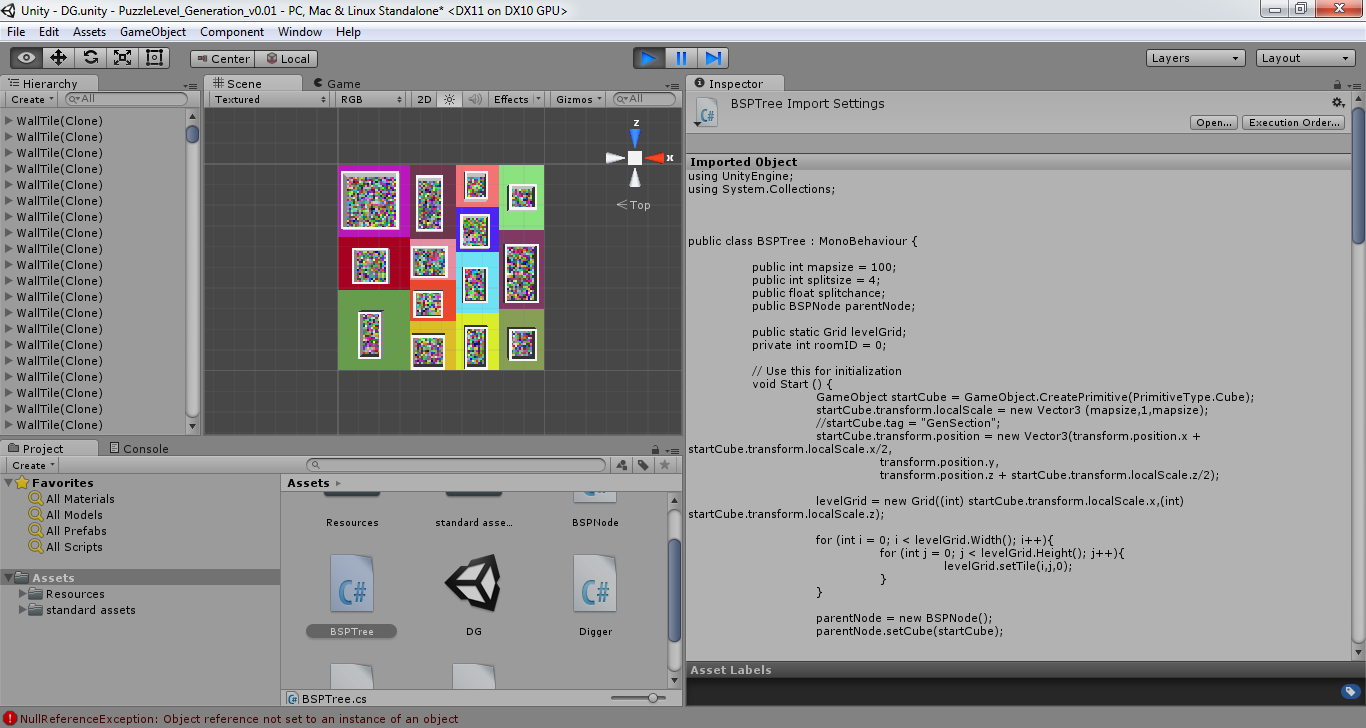
The advantage of using a constant cycle of Design, Develop, Test and Analyse is that it allows working prototypes early in the project. The primary benefit being to greatly reduce the risk associated with implementing mechanics by identifying the feasibility of a design without expending too many resources. Problems can be discovered early in the project and alternatives can be implemented within further iterations. By having many usable prototypes it is much more valuable when gathering user feedback which is a primary requirement within my project. These prototypes provide the basis of the final product, beginning as a minimum functional build, growing incrementally more complex and finally into the completed system. Throughout this process the RAD methodology provides an indicator of the final results as well as having the security of functional builds early on.

An obvious disadvantage with Rapid Application Development is that due to the iterative process of adapting the project it is common to stray from the original purpose. Because of the expanding nature of iterative prototyping this can lead to less control and bad design as a result of rushed planning before implementing further features. The project can continue to grow leading to scope creep whereby the ever changing requirements are never really met. These risks are more present in large scale systems so for my project I will persistently check the project’s direction to avoid this.

# Implementation

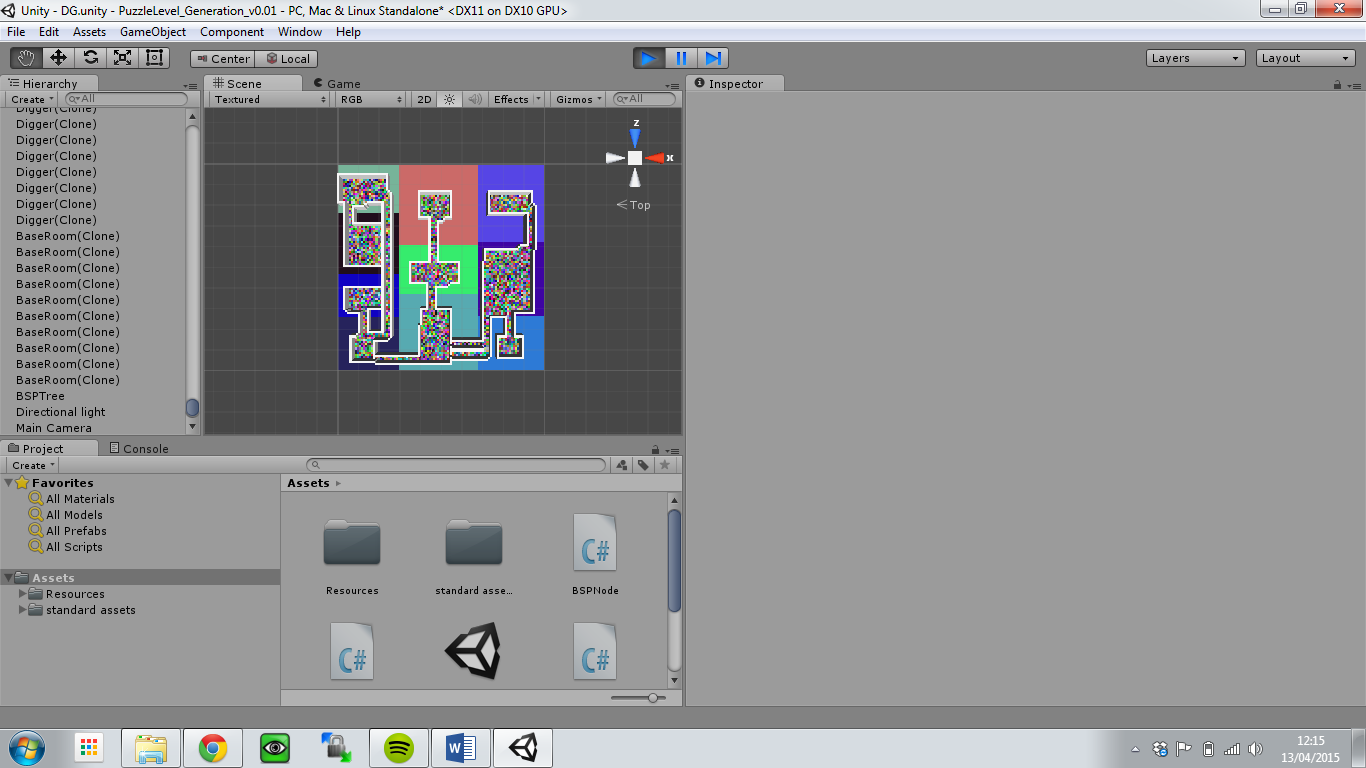
## BSP Map Generation

With the first implementation of the procedural dungeon generation I used binary space partitioning of a grid, where a plane is divided into smaller sections. The initial area is divided either vertically or horizontally and split into two sibling nodes, these nodes are subsequently split up to the defined split size. The completed binary space partition produces a hierarchy of nodes with varying sizes. These separate node sections are depicted in my model by randomized colours.

Within each of the partitioned spaces I randomly select a room area which are assigned within the grid as floor and wall tiles. The connecting of the rooms achieved by checking for its sibling node if available and finding the most suitable door locations which are then connected by the digger class. This digger is initilised with its target position which it strives for and updates the grid references for wall and floor tiles.

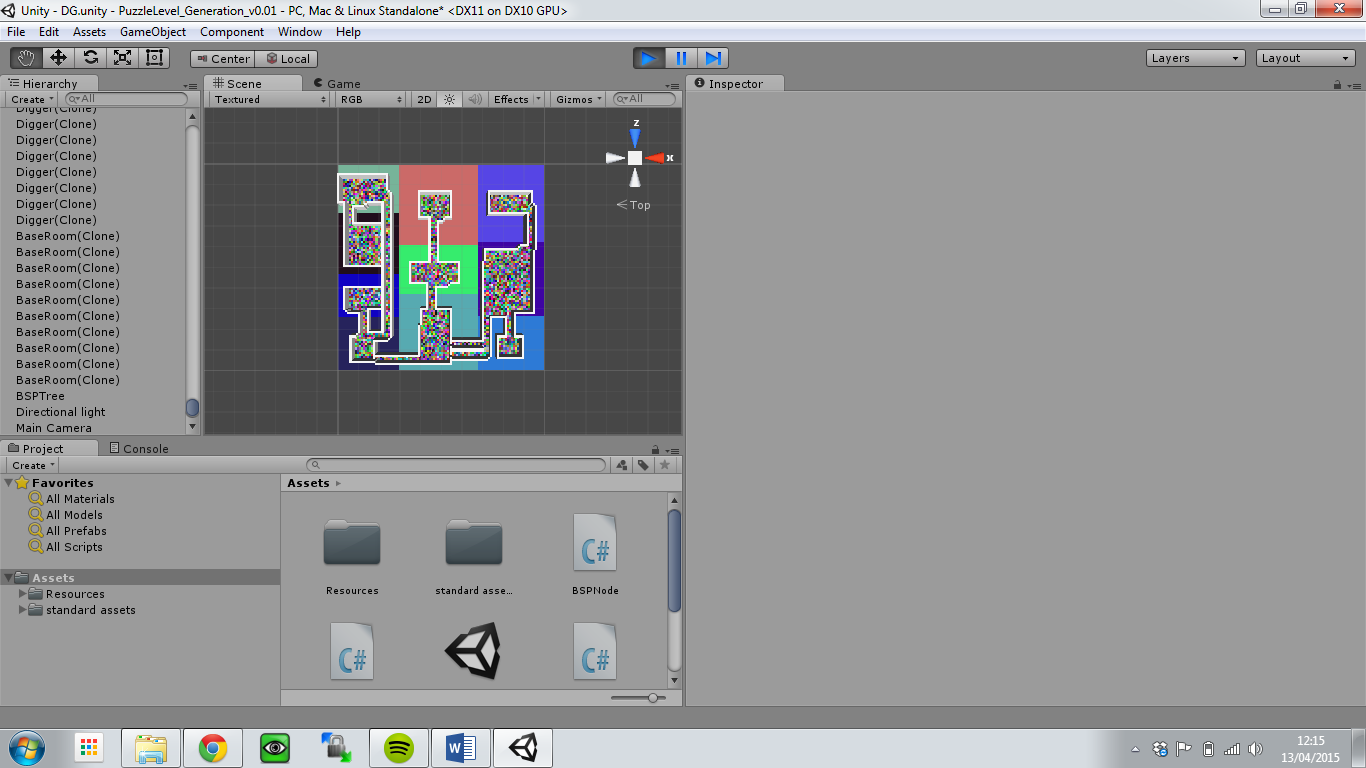
Finally the finalised grid is instantised with prefabricated assets for wall and floor tiles. Each floor tile has a randomized colour to differentiate it from the node colour.



The resulting map produced a well connected series of rooms of varying sizes. This good connectedness is due to explicit checking for unconnected rooms within the connection method. The connecting corridors do add good variety of connectivity with multiple corridors both long and short and turning correctly when needed. A key success criteria is obviously that the player must be able to navigate the whole map without error.

While this is a generally successful initial implementation there are a number of issues. Most notably this method results in rather predictable spacing of rooms due to the nature of binary space partitioning. Also the consistently rectangular room shapes become rather repetitive. While these issues are undesirable they are not “game-breaking”. However there was one recurring error which affected the core success criteria whereby the corridor opening was not dug out properly resulting in blocked paths for players.

The issue with testing these procedural techniques is that these errors do not occur consistently and finding the cause is often much more difficult. Below is the testing for this particular map generation and how often these critical errors occur.



This issue could be resolved through further implemented systems such as cellular automata checking to see if any areas are blocked off to the player. Regardless the linear nature of binary space partitioning made this implementation rather undesirable. It was following this first working model that I decided to revise the core method of map generation to produce more varied and interesting environments.

## Square-ception Map Generation

Following the findings from the initial map generation model using binary space partitioning I opted to design a more varied map generation technique using overlapping squares. The method of map generation is based upon a grid as a two-dimensional array much like the BSP implementation. Initially the values of the array are set to 0 and then numerous sized squares populate the array with 1s. Each cluster of overlapping squares is then numbered sequentially within the array to give a set of uniquely identifiable rooms. This produces rooms with much more varied shape and structure as opposed to the linear rectangular rooms of the BSP implementation.

To connect these rooms I check the bounds of each room along two axis to determine its nearest neighbour. Since there is no hierarchical relationship between the rooms as there was in the BSP implementation this is a rather crude search for possible connections but works rather effectively. These corridors do not bend since due to the abstract shapes of the rooms there is often a simple straight connection to another room.

This method of map generation does have its own limitations primarily the unpredictable nature of the room selection. Because the randomized squares are not limited to a set area as they were in the BSP model they can potentially result in one very large room where all the squares have overlapped. Although this can produce a rather interesting room it is not suitable for the context of this study. To prevent this I simply redraw the two dimensional array if the resulting number of rooms is below a specified amount.

With the size of the grid to number of squares created there is often a sweet spot which produces the best results. This required testing numerous generations at varying grid sizes and squares created to determine which combination produced the best results. Below are results of these tests.

## Puzzle Generation

With the procedurally generated level the task now became to create a strict puzzle framework to adapt to work with any given scenario. This meant the player would not get stuck or be unable to complete the puzzles generated. Due to the rather unpredictable nature of the room development I first ensured that the Room classes contained all the necessary information required for the generation of relevant puzzle.

ROOM CLASS BREAKDOWN

The rooms hold the blueprint to what puzzles will be developed.

The process of puzzle generation involves several sequential steps; lock selection, puzzle assignment, puzzle selection and element generation.

### Lock Selection

The first step is to determine which rooms will be locked to provide the core of the level’s gameplay. These locked rooms will hold an element required for the end condition. Due to the multi-connected nature of the map generation I use the connections of a room to determine their suitability as a locked room. A locked room has doors positioned at each connecting entrance to avoid any attempt to bypass the locked door through alternative paths.

This is why a single entrance room with only one connection is the starting point of lock selection with every single connection rooms being selected as locked rooms. The reason for this being that dead ends make the ideal locked paths, they avoid any possible complications with other connected rooms. From a playability perspective the player is also rewarded for exploring to find these dead ends.

Should there be no single connection rooms or there are less than three locked rooms the process will examine rooms with two connections. These often act as thoroughfares and also make a good locked door so long as the player is able to access a mechanism to bypass the lock. For all two connection rooms the chance to be a locked room is 50% if there is less than 3 locked rooms.

While evaluating one and two connection rooms will generally produce at least three locked rooms there is a potential to evaluate rooms with three connections. These must meet the same criteria established for the two connection rooms. Should this fail and be unable to find three suitable rooms to lock, the application will reload itself with new map data.

### Puzzle Assignment

For the assignment of the puzzles the locked rooms will have their connections evaluated for the suitability of a puzzle element. This is where potential conflicts may arise and assignment must be carefully considered. For each locked room their list of connections must fulfil a specific criteria to have a chance at being the assigned puzzle room. The potential puzzle room cannot be locked or already assigned as a puzzle room to another locked room.

Upon successful assignment the locked room’s lockAssigned Boolean variable returns True and the puzzle room’s puzzleTo variable becomes the Id of the locked room. This partnership allows for only one puzzle room to each locked room, avoiding potential issues with multiple conflicting puzzles. However this could be expanded to allow multiple puzzles to be assigned to a locked room adding more complex gameplay.

In many instances a locked room cannot find a suitable connection to assign as a puzzle room, in this case the application will check each locked room for it’s lockAssigned Boolean. If after puzzle assignment the room is unassigned it will be provided with a key puzzle. This baseline puzzle will spawn a key pickup dynamically within other freely accessible rooms which will open the previously unassigned locked rooms. While I originally implemented this as a safety mechanic to resolve issues with unassigned rooms it quickly became a very desirable puzzle mechanic.

### Puzzle Selection

The selection of puzzles for the assigned puzzle rooms randomly selects any of the puzzles passed in for selection. It is here where the application could easily expand with any number of varied puzzles. When a puzzle is randomly selected from a switch statement the puzzle room area is passed into the puzzle object along with any additional parameter information.

### Element Generation

Puzzle elements are instantialized during the puzzle selection, followed by the instantializing of all locked doors. Most puzzle elements are loaded within their puzzle logic class selected within the puzzle selection. Each puzzle functions in very similar ways, firstly identifying which positions within the room it can spawn elements and then dynamically choosing a spawn location.

Logic is held primarily within puzzle gameObjects, initialised upon a specified trigger. This means puzzles will not need to continuously check their state within the Update() method unnecessarily. This is to optimise the execution of the application during runtime particularly when scaling to larger map sizes or weaker hardware.

# Findings & Discussion

# Conclusion

# References

