**1.0 Introduction**

The creation of content for games is continually increasing in time and monetary cost as games become larger and more detailed. The reliance on hand crafted content cannot continue indefinitely within the industry. This project aims to resolve by developing a procedural content generation system for game levels. An overview of the problem is described in Section 2. The proposed game level generation system is outlined in Section 3.

**2.0 Background and Existing Systems**

**2.1 Project Background**

The procedural generation of content (PCG) in games has around since the 1980's with early examples like Rogue and Diablo (Blizzard Entertainment) and though the topic has been an area of much research it still has yet to be fully adopted by the industry.

There are numerous reasons to apply procedural techniques to at least some aspects of game development, the most readily apparent being monetary. The cost of game development is ever increasing, particularly for AAA games, as consumers have come to expect more detailed and expansive games ever year. Being able to generated a near infinite number of different and interesting assets for just the initial investment in the generation tool and in a fraction of the time an artist would normally take is obviously an appealing prospect for development studios.

There are however other reasons for the integration of PCG beyond time and money saving. It can deepen immersion and enable emergent narratives through the interaction of relatively simple generation systems. For example, Crusader Kings 2 generates personality traits that effect the decision making of the AI. The dynamic interaction between these NPCs guides the player to write their own narratives for each session with no involvement from the designer.

PCG can also foster exploration and greatly extend the player retention of a game by creating new and interesting content for players to interact with. Playing the same hand crafted level repeatedly can often become dull and any perceived imbalance will likely be treated far more harshly by players than with a generated one.

Finally PCG can be used as a creative tool for designers. Algorithms do not suffer from limited imagination and can produce incredibly novel content that can inspire or be refined by a human designer.

**2.2 Existing Systems**

While there are now many games that make use of PCG the majority are written for that specific game and thus are very difficult to reuse for others. While this is partly due the type of content being generated and the target genre, i.e. a 2D Mario level generator is of little use for a developer of a first person shooter, even within the same genre there are only a few generic tools to aid in the generation of content.

The two most well known commercially availability content generation tools are Interactive Data Visualization, Inc.'s SpeedTree modelling software and NaturalMotion's Euphoria animation engine.

SpeedTree is a specialised modelling tool for the creation of foliage in computer graphics. The tool can procedurally generate tree for the artists brush strokes, adjusts to simulated forces (e.g. increasing gravity will dynamically twist the tree) and even supports 'growing' the tree around or within arbitrary meshes.

There are numerous clear advantages to using SpeedTree beyond the features described above; It's a mature product (initially released in 2002) that is integrated in both Unreal 4 and Unity 5 engines, has already seen use in hundreds of games, automatically creates efficient LODs and texture atlases, provides access to the SpeedTree library with a wide range of pre-made models.

However the system has the clear disadvantage of being a specialised product that only generates foliage offline (i.e. not during gameplay).

Euphoria is an animation engine for games that procedurally generates a model's animations by simulating the bone and muscular structure.

This allows for the characters in game to move and react to the environment naturally without having to create numerous hand animated or motion captured sequences. For examples of its use Euphoria has been integrated in Rockstar's GTA IV and V.

Despite the advantages to generating animations the technology has not been widely used within the industry. This is largely due to the fact that NaturalMotion do not directly license out Euphoria; instead they evaluate prospective clients and directly integrate the engine for a very limited number of development studios. While NaturalMotion to not list prices it is widely reported to be incredibly expensive for the few studios given licenses.

While both the systems described above offer impressive solutions they are both designed to generated one specific asset for a game. That combined with Euphoria's exclusivity and likely exorbitant cost show they are both obviously not suitable for the proposed system.

Similarly while there are a number of games that generate their levels they are all tied to their specific game and cannot be used across multiple games. Thus the need for the system proposed below.

**3.0 Proposed System**

**3.1 System Overview**

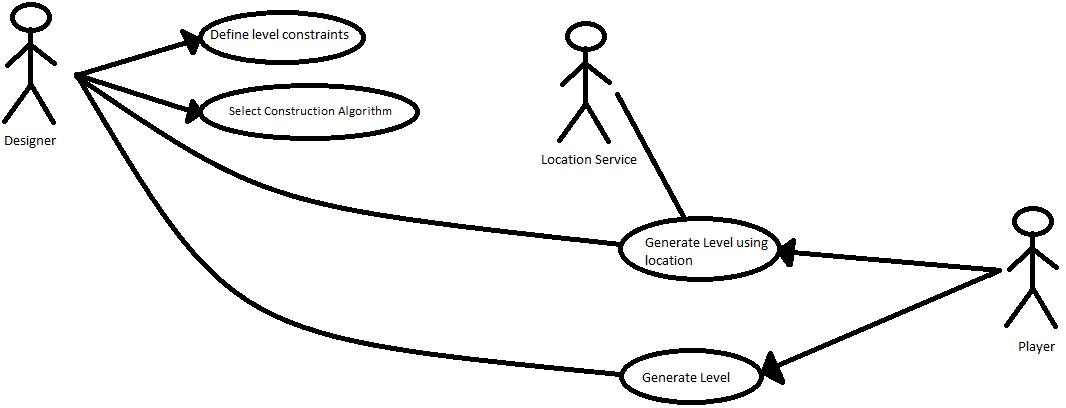
The proposed system is a library for the Runtime procedural generation of game levels. This library is targeted towards mobile games developers and so will be written for the Android platform using Java.

The core functionality of the library will be broken into two main algorithms; construction and population. The construction algorithm will generate the base terrain and layout of the level in the form of a heightmap. This heightmap will then be passed to the population algorithm which, using user defined constraints, will optimally populate the terrain with game entities (weapons, enemies, chests etc.) as well as objectives using a genetic algorithm.

The population constraints will be written in a designer friendly format such as XML and specify features of the level such as the number and density of game entities, start to end distance and pathing.

While the user constraints should be flexible enough to handle the majority of the user's requirements, in an effort to support portability between game genre, the library will allow users to override and extend the construction and population algorithms for their own specialised logic.

**3.2 Use Cases**

Figure 1

1. **Define Level Constraints**: The designer either programmatically or with XML builds a list of requirements for the content generator to aim for. The target fitness and maximum number of generations for the genetic algorithm are also defined here. Any number and combination of the constraints can be defined. Unspecified constraints use a configurable default value.
2. **Select Construction Algorithm**: The designer specifies which base level terrain generation algorithm the system should use. They do this by constructing and passing an object that implements the 'ConstructionGenerator' interface (see below) to the level generator. If no 'ConstructionGenerator' is supplied the system will use the default Perlin Noise generator.
3. **Generate Level**: The player requests a new game level. The game will then call through to the level generation system supplying the constraints and 'ConstructionGenerator'. The level generator then returns a heightmap and positions of the game objects.
4. **Generate Level Using Location**: Similar to the Generate Level use case but additionally calls to the Location Service and passes the location data to the level generator as well as the constraints and 'ConstructionGenerator'. The location data will be ignored if a) The constraints disable location usage. b) The implementation of 'ConstructionGenerator' does not support it. c) The Location Service fails to obtain it.

*Designer: The game designer/programmer using the library.*

*Player: The end user of the Designer's game.*

*Location Service: The mobile device's location system.*

**3.3 Interfaces**

ConstructionGenerator

Methods:

* buildMap(width:int, hight:int, location:LocationService[0..1]) :double[][]

Classes that implement this are responsible for constructing the level's heightmap.

PopulationGenerator

Methods:

* populateMap(heightmap:double[][], constraints:Constraint) : List<Entity>

Classes that implement this are responsible for interpreting the constraints and heightmap to define the number and location of the entities in the level.

Entity

Methods:

* setPosition(x:int, y:int)
* getPosition(): int[2]

The root interface that all game objects the generator should populate the level with must implement.