Graphical Assessment of Procedural Dungeons

Student Name: Oliver Baxandall Supervisor Name: Dr. Frederick Li

Submitted as part of the degree of BSc Natural Science to the
Board of Examiners in the Department of Computing Sciences, Durham University

Abstract —

Context/Background - The increasing commercial success of video games is matched only by the difficulty in developing them, therefore it has become fashionable to incorporate Procedural Content Generation (PCG) into game design to meet growing end-user demand for a clean, polished, yet intricate final product, while remaining within production constraints. However, developing game composition programmatically yields the issue of ensuring output appropriately conveys the message and recreational experience intended for the player by a developer, culminating from inability of algorithms too effectively identify desirable content.

Aims - The project aims to combine existing methodologies into an executable game environment that produces level structures based on PCG. The program must accept a developer's/computer's choice from unified criteria that directly inspire the resulting generation, thus having randomised features that embolden characteristics initially specified. An evaluation process will then breakdown level physiognomy to encourage either manual or automatic influence over the current generation or the following one, granting levels a greater visceral meaning and designers a simplified design process.

Method - The implementation will utilise an intelligent model, building upon established grammar centralised methods that inaugurate rules fixating on a set of user provided seeds, followed by controlled randomisation of an enclosed environment against a graphical representation extracted from encapsulated attributes. Automated evaluation follows, providing numerical analytics for a particular set of friendly constraints during level progression or lending the model as inspiration for alternative game structures.

Results - The program distinctively incorporates a unique blend of techniques into a system that produces level content on-demand and internally/externally from a game engine, offering convenience for both a developer and end-user. Parameter specification grants individuals an uncommon precedent, while maintaining evolved, visual and intuitive variety within portrayals.

Conclusions - The combined approach utilising the benefits of random geometric level design and graphical grammars gives a developer valued control over incidental game design, while facilitating scope for diverse content to appear. Offering vast improvements over rudimentary procedural designs, the methodology is, however a relatively expensive process that requires careful application to reap full benefits in incidental level composition and as a developer assistant.

Keywords — Procedural Content Generation, Graphical Grammars, Computational Assessment

INTRODUCTION

Video games in all their forms are swiftly becoming one of the most dominant forms of entertainment in the modern world, explaining the rapid growth in the scale of development capabilities. However design is still restrained by hardware, production costs and development time. PCG techniques are becoming commonplace in development to circumnavigate computational, memory or graphical display issues but frequently applied to only very specific instances. Success stories that more heavily rely on PCG exist but lack mainstream commercial use due to the control given up by designers to suspicious algorithms that are unpredictable by nature

(Van der Linden et al. 2014). The benefits of increased replayability, rapid generation, reduced development load and consumers heightened expectations are altering the demands placed on PCG in modern games. Results can be pleasing but lead to repetitive levels with high burdens or inappropriate products depending on freedom given to randomness. This is especially true when PCG is relied upon for constructing the interactable environment.

A Background - The Nature and Challenge of PCG

PCG is the creation of content algorithmically, allowing game components to be created automatically rather than operating as a burden to developers (Compton & Mateas 2006). Upon initialisation, the system accepts a set of parameters which are processed through defined rules or templates, which output a level. Such a system lends itself to game design by directly assisting human imagination, providing a framework through which developers can manually select desirable instances and subsequently improve upon them (Hendrikx et al. 2013). Alternatively, levels can be produced procedurally in the final application upon meeting specified criteria or once the generation process has completed.

Thompson (2015) consolidates the unknown challenge of dungeon crawlers as providing evidence for why humans continue to find problems interesting. Guaranteeing intriguing level design is difficult, requiring substantial care and experimentation by a developer, with superficially different levels produced that lack meaningful, consistent structure, failing to produce an equal experience to human developed gameplay. Giving an automated game the illusion of a creator and visceral meaning continues to be fundamentally problematic (Cook & Colton 2014).

B The Aims and Objectives of this Project

We centrally advance procedural techniques and provide a solution to PCG's negative side effects by providing algorithms with the capabilities of a human designer, shifting the "labour" involved in development. This will increase simplicity for developers producing games and make it possible for a computer to request characteristics and produce them efficiently. For design, the difficulty in evaluating the usability of generated level techniques suppresses recognition of interesting content (Ashlock et al. 2011). The result will have the capabilities to generically compute level contents, given easily interpretable factors that can be altered computational or by any user of the software. The project therefore has applications in amplifying game development speeds, simplifying development for users unfamiliar with game design, and augmenting the end user experience with continuous fresh, but always relevant playable material.

The proposed game will utilise the Unreal Engine as an encapsulation for development. Game logic will be produced using the visual scripting language Blueprint, to follow the graphical-representation for development and facilitate ease of alteration following evaluation. An executable will be produced that produces levels until appropriate, subsequently offering an assessment of its characteristics for a developer to utilise as they wish, without onerous effort or time investment, within the engine or as a running application. Developers could use our solution to inspire their new content, doing the heavy lifting when requesting a design.

This will provide a player with an interactable environment, while utilising inspiration characteristics for developers to manually alter levels or promote automated level evolution. Production rules will be specified to intelligently create foundation content, followed by fitness analysis to breakdown the overall suitability. Alteration propositions will come as summarised statistics

and level highlighting, then integrating this to the system for subsequent generations, facilitating content production on-demand. The project aims to promote inventive features, so success in this area will be dictated by the variety of relevant content created and their relevance appropriateness. This is maximising the number of levels available from the set created after applying constraints to the total State Space of possible levels. Finally, it is intensive to ensure consistency between design, so a final subjective measure of accomplishment is whether the project comprehensively saves on net development time for a game architects. This will ensure a rewarding experience for a player/developer in a genre/industry where all one maintains upon failure is the experience, punishing for mistakes as opposed to a challenge's unyielding difficulty.

C Achievements

The software works as an external executable/playable application, available for Windows, MacOS and Linux but also as an editable model inside the Unreal Engine. Randomised specification is then followed by generation, separation and filtering in both topological and geometric representations. The in-Engine variation interlays these alternative forms over each other to facilitate staged analysis by a developer. This variation additionally includes a user interface as a closable Heads Up Display (HUD), for design experimentation, exploring versatility and creating desired content, savable within the engine. Assisting in the process is a display of current level characteristics, used both in the initial rendering and those following it. These particulars are easy to interpret and thus allow design alterations to be made computationally or by an individual unfamiliar with the methods of game design.

The showcase in Section D of the solution, is an example of how the model can be adapted to suit different game variations, following the specified ideas of a developer or the mechanics they wish to implement. By fixing parameters and utilising selective room removal followed by experimentation, the solution produces an explorable, organic labyrinth for a player to navigate. Originally, the program aimed to deliver advanced fitness evaluation for automated future progression, however as discussed in the evaluation section, this problem tilted towards reliance on data as opposed to artificial intelligence, since at higher levels, recognition of content becomes difficult without a comparable context or existing data for juxtaposition. Greater emphasis was placed on ensuring outputs matched desired criteria. The size, complexity and possible routes can all be specified, facilitating alteration to desired difficulty and playing style variations. The software thus shows its possible applications during game development as a time saving system for experts and beginners alike, while applying itself to on-demand generation. Although each have their applications, the main objective has been to create a package that alters developer reliance on analytical relationships when maturing development.

The rest of the paper will be organised as follows. After surveying the current state of PCG research in relation to the proposal, the applied algorithms and software methodologies will be addressed. Once it is clear how the model accomplishes automated level design, and with basic instructions for its use, methods for evaluating the effectiveness will be highlighted before giving results from testing measures. This allows for a critical appraisal of model architecture in affiliation to other works and finally an encapsulation of the possible extensions.

RELATED WORK

In this Section, researcher progress in PCG will be epitomized and then applied to the proposed unified program. There has been significant academic focus on producing various PCG methods, in addition to editing and evaluation. This project aims to establish a level structure, break the structure down graphically and then evaluate the generation with data specified against specified conditions, giving the user the opportunity to alter the level as they choose. This acts as a complete model for the development process, so the current state of each art is, and the approaches attempted are summarised separately.

A Roguelike - Level Structure of the Dungeon

Dungeons are a subset level genre, defined by Shaker et al. (2016) as labyrinthic environments, used to guide narrative towards a fixed goal. 'Dungeon Crawlers' or 'Roguelike' [1] require players to guide a character through an unknown environment while using equipment, abilities and intellect to circumnavigate challenges. These are organised into levels, representing a self-contained unit that the player must finish. Algorithms rarely consider the context content is derived under and as such act independently of other structures such as player history (Togelius et al. 2013). It is necessary to establish the desired nature of the dungeon produced, classified by Togelius et al. (2011). Decisions must be made to separate optional features from necessary ones, whether outputs should be stochastic or deterministic [2], and finally the standard with regards to the final production. This factor is the most important, outlining whether the algorithm constructs content until conditions are met or chooses from a sample of generated levels [3]. Construction of dungeons can be characterised into wall adders, carvers and template placement (Foltin 2011). Templates as a technique has the highest randomness management, involving the placement of pre-generated traversable environments into a blank setting, representing the explorable path. This lends the most control over generated content, facilitating intelligent material inclusion and thus is the chosen design technique.

B Graph Theory: Representation and Application

A mathematical graph lends itself to many fields, where an analytical representation of a complex concept is required (Hendrikx et al. 2013). The desire to process random variables through an established process to create a randomised generation is similar to that of Context-free grammars, where logic is represented by regular expressions and strings generate the final output (Foltin 2011), the language depicting all possible level designs. This concept develops onto graphical grammars, capable of representing significantly more complex patterns found in dungeons by generating graphs instead of strings from rule sets. Grammar Outputs are known as a *topology* and can be used as an inference for final level design, specifying the relative position in which locations and objects are met, but nothing of their nature. Graph transformations lend themselves to this process, having greater scope and flexibility for randomness while maintaining given statistics, enforcing the final product to match specified properties. Van der Linden et al.

[1] After the game Rogue created in 1980 by Glenn Wichman and Michael Toy

[2] This also depends on whether generation accepts a random seed from a Pseudo-random Number Generator or a parametrized set of vectors from an alternative source

[3] Known as Search-Based Procedural Content Generation (SBPCG)

(2013) effectively implement such a technique by hard coding their design constraints into production rules, effectively automating the generating process while giving developers full control over the build. Rule specification sacrifices variety, given that such rules cannot be augmented for alternative results automatically without introducing mutation. A substitute technique is generated by Smith et al. (2009) who implement pacing and rhythm techniques to 2D platforming at regular time based intervals. The study helps quantify difficulty and level components as the characteristics of the level are defined by generated rules, however in terms of the quality output similar problems are suffered, with levels lacking meaningfully original content. Such problems fall under the project scope and will be solved by abstracting the graphical generation from the rules that create them.

C Parameter Constraint Control, Evaluation and Inspiration

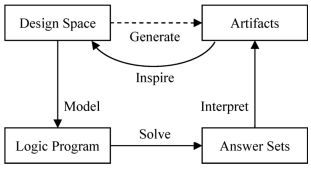


Figure 1: Design Space for enhancement. Provided by Smith & Mateas (2011)

Computational recognition of desirable qualities is a requirement in PCG. Ideal content can be hard-coded into production rules, or fitness functions can be utilised, either in a search based approach or acceptance if a result is above a given parameter. Valtchanov & Brown (2012) incorporate search-based methods, focusing on generating content recursively, instead of just accepting a single pass result compared against other population members, which is subsequently altered in order to maximise the fitness function. Collecting data and representing it meaningfully is the second barrier to PCG analysis (Togelius et al. 2011). Scales & Thompson (2014) use virtual agents that navigate a simulation of the generated environment, replicating the actions of a player, artificially predicting movement, predicting their potential satisfaction. Extracting a Dungeon into virtual space thus facilitates analysis by acquiescently allowing algorithms to recognise meaningful structures, however such abstraction can result in loss of content. (Adams et al. 2017) seek to solve this issue by accepting pure level geometry into the fitness function but simplifying the intelligence of the resulting analysis, balancing detail with potential perception. Where alterations are computationally demanding, a smaller population size must be implemented and thus a weaker spread. (Adams et al. 2017) effectively displays this trade-off by simplifying fitness calculation in favour of more examples of basic dungeons generation that purely focuses on geometry, addressing that more complicated evaluation is necessary in the presence of additional mechanics. Representation of content is the second central barrier to SBPCG (Togelius et al. 2011). While methods such as 2-Dimensional Arrays and vector representation of component frequencies and positions exist, encapsulating all a level's important characteristics is either computational heavy or inaccurate. Vectorisation is a commonly used strategy to represent characteristics that a maze generation exhibits [4] and will be utilised before scalar depiction during indexing. Constructing meaningful design choices is shown in Figure 4, using Answer Set Programming (ASP), which shares many similarities with logic programming in rationale. Bots capable of navigating through an unforeseen environment are created and collect data on the content. This data is fed back into the design space as "inspiration" for further development, evolving levels over iterations by recognition of interesting features. Progressive content improvement through desirable characteristic admission will be an overarching aim for this project.

[4]These include the frequency and density of junctions, loops, crossroads or any predefined design characteristic, complicated as desirable providing it can be numerically described

D Summary

Although each of these components has been extensively researched, few software packages combine each component in a full application, connecting the academic world of research to possible consumer applications. A similar ambition was achieved by Adams et al. (n.d.) and Van der Linden et al. (2014), acting as contrasting attempts to the problem and will therefore be important for disparity and relative performance gauges.

SOLUTION

This Section presents the solution to the discussed problems. The development process and tools are identified, followed by a summary diagram of the application. Algorithms used are then discussed with employment of the model identified, followed by validation methodology.

A Software Specification

The implementation utilises Unreal Engine 4.21. The decision to use an engine for development was influenced by the sped-up ease of developing an executable application by the end of the project in comparison to manual development together with the inbuilt functionality for debugging during a generation. The disadvantages include the learning curve associated with becoming familiar with the workings and potential difficulty following from a unique implementation the engine may not easily facilitate. The Visual Scripting Language Blueprint, offers a node-based interface for creating gameplay elements while offering a markup in C++. The flexibility of this system, together with the focus on a graphical-based implementation lends itself to the project, which as such was the most appropriate for development.

B Life cycle and Development Process

Experimentation was conducted concurrently during an agile development process, with extensive testing required by definition to simulate developer work. As alterations were made to the software, results were viewed and altered appropriately, adapting to new ideas, methodologies or adjustments as they occurred. Overarching goals were established with the production process broken down intso Room Placement, Graphical Representation and Evaluation/Parameter control. These were set as periods for work, with a week meeting cycle, setting targets to be achieved during each run. This allowed expectations to be met and kept prototyping focused towards the end goal while allowing augmentation for better ways to design the software. Experimentation was conducted concurrently, with extensive testing required to simulate a developer working with the system, encouraging an implementation that was easy to interpret and friendly for a foreign user. This allows for adaptability when faced with the unpredictable nature of algorithmic effectiveness as the original design evolves into maturity, avoiding complications to the development process, while providing a constant arch of progression and automatically allocates additional time for evaluation, a functional requirement of the project by objective definition. This was chosen over a waterfall model centrally due to the uncertainty and unfamiliarity of implementing the chosen techniques and using the software discussed in Section C.

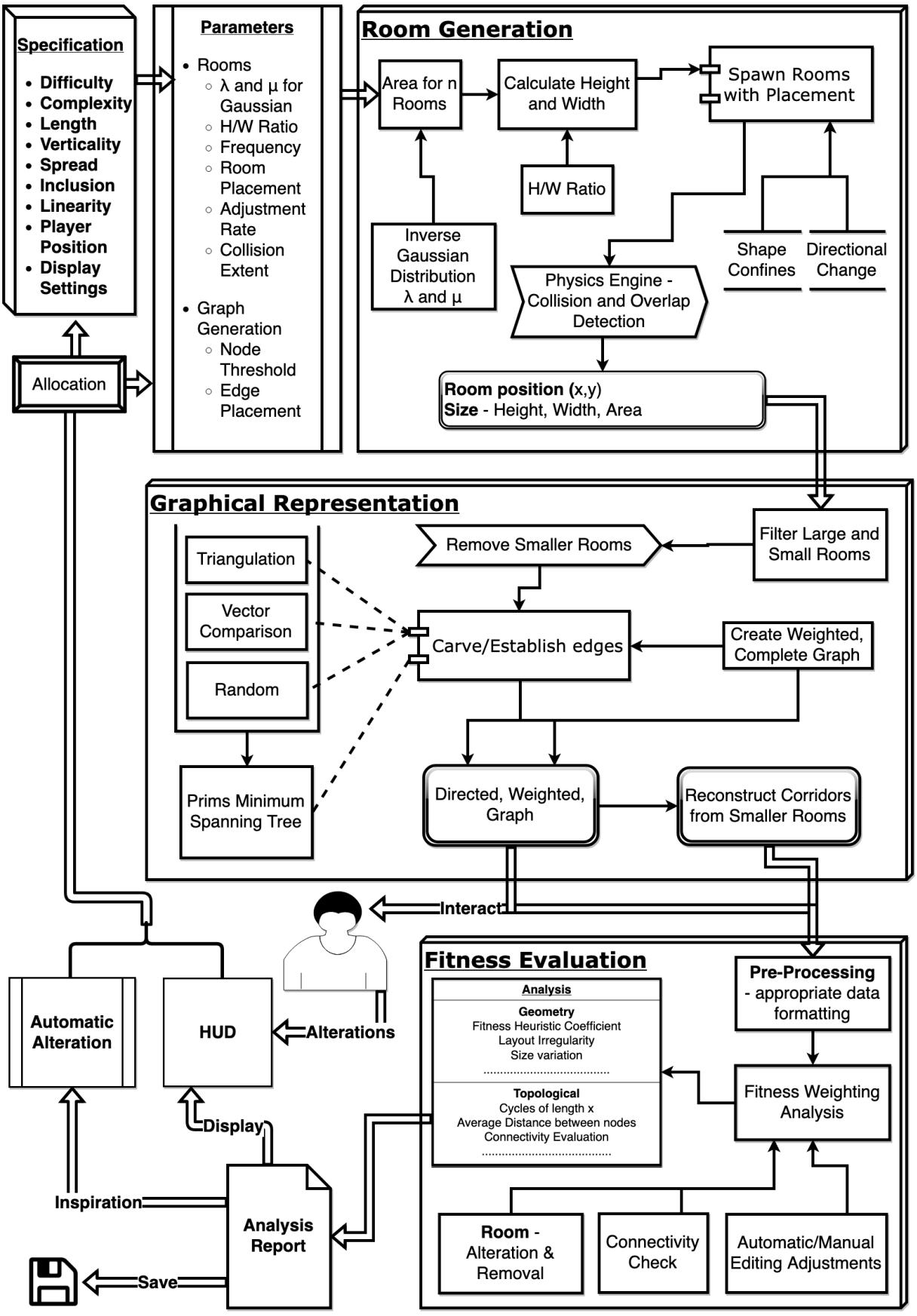


Figure 2: A Design Space for development and enhancement of Level Structure

C Architectural Overview

A geometric description is the physical depiction of the level, whereas the Dungeon Generation System outputs a topological description of a level, a level definition avoiding expose to physical sizes, concerning itself only with the order objects are encountered. The scope for potential designs is endless. PCG methods such as Cellular Automata generate structures that influence certain areas/pixels on a environment, so focus will instead be on positioning explorable areas into the setting. For consistency, these structures will be known as 'Rooms' and represent areas carved out from the surroundings that a player can navigate through or Sections that are blocked off. The choice of level structure lends itself towards graphical level design given that 'Rooms' in turn can be represented by Nodes in graphical space.

Component based architecture is exploited system follows a three step looped process shown below in Figure 2. A set of parameters will be supplied to the system. These will specify details required for the generated level. A set of grammar rules should then use these parameters to estimate a base set of values for the system to function. Room height, width and position are generated so that none overlap with each other. A percentage of the largest rooms are chosen and all other rooms temporarily removed. The layout is then converted to graphical format and edges indicating possible paths through the world are established. Larger rooms are then connected by smaller counterparts so that adjacent nodes represent larger rooms linked by a corridor of smaller ones, although this step is programmatically adaptable. This final output is then broken down combining the graphical and final representation for assessment and offering interpretation for the next attempt.

C.1 Room Generation

Traditionally, PCG operates by converting a topological representation to its geometric counterpart. However, (Dinh 2012) suggests inverting this relationship, leaving the management of randomness to include all level detail and subsequently organise the translation using a simplified representation. This step is utilised to randomly generate the structure of a level. While previous implementations use graphical grammars to generate networks of nodes as a depiction of the final layout, the initial level is used as templating for graphical representation. This ensures node/room placement is controllably random while allowing the size and position of rooms to influence the final fitness calculation instead of leaving this to chance.

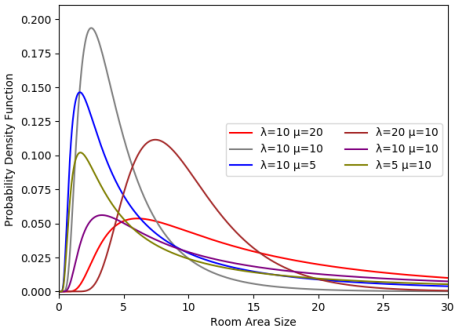
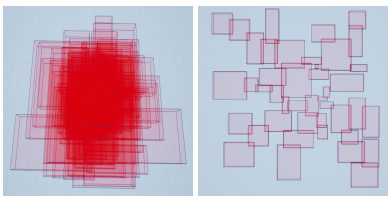


Figure 3: A Wald Probability Density Function for Room Sizes



(a) Collision Detection Deactivated

(b) Rooms Spread Out after Activation

Figure 4: Rooms before and after Spreading Effects

Room Size - An Inverse Gaussian Distribution (Wald Distribution) is used, controlled by the shape parameter λ and mean room size μ as follows:

$sf(x;\lambda,\mu) = [\frac{\lambda}{2\pi x^3}]^{\frac{1}{2}} exp\{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}\}$

The resulting distribution is positively skewed, favoured because it ensures there are many more smaller rooms then larger ones and thus producing many interconnected corridors. The values of λ and μ are thus set relatively low to ensure a controlled number of larger rooms are generated for game

focus. The number of rooms are generated given parameters, each room being allocated an area. The ratio of height to width is randomly generated [5] and then their actual dimensions calculated.

Room Placement - The shape of a sub-environment is initially specified for relative room placement. Each room is allocated a random (x,y) coordinate within this shape from which it is spawned, initially overlapping with each other. Utilising collision and overlap detection within the Unreal Engine, rooms are subsequently separated out to avoid overlapping, relative to their current position. As can be seen in Figure 4, this yields a room distribution, where each room is still connected to at least one other. The initial shape lends itself during collision detection by encouraging rooms to spread out in certain directions. For example, if the generation environment was stretched from an circle to a horizontal ellipse, a horizontal based distribution will result. This gives a developer control over the overarching appearance of the level. For example, if gameplay is vertically focused with a desire for less complicated branching, a squashed rectangular area may be appropriate, ensuring generations are built to suit these needs and as such the player knows they must proceed in a single direction.

C.2 Room Graphical Representation

Room Segregation - To declutter the number of rooms rendered in the final implementation, an extensive filtering process is used so as to include only those necessary for what is to be played. The x largest rooms, proportional to the number of rooms generated (set by default to $\frac{n}{10}$) are extracted into a separate list and rendered darker in Figures. This threshold for separation is also set by classifying all rooms a set size above the mean as large. These rooms are subsequently treated as central components for the level, each representing a node in the graph. Each node is numbered and its position from the centre of the map stored as a vector. Drawing an edge between two nodes establishes a connection between them, used to evaluate a generation and for establishing which smaller rooms are appropriate to include. For example, larger rooms could be connected by small rooms, forming path/corridor structures between them for players to navigate. Alternatively, edges represent carvings into a level where rooms should not be rendered, freeing a player to explore the area independently.

Graphical Creation - Initially, a complete graph is created. The distance between each node is calculated in order to establish weightings for each edge. This is represented as a vector, capable of indicating the position and distance between nodes. Following this calculation, appropriate edges between nodes must be selected. This is completed using two primary methodologies:

- Vector Based Angle Variation - A very rudimentary technique, selecting possible edges between nodes as long as the new edge does not overlap with anything previous placed. The allocation continues randomly until no more edges can be used to join the network.
- Delaunay Triangulation - A methodology credited to Boris Delaunay in 1934, accepting coordinates of points on a Euclidean plane. The algorithm renders an output where each node has a degree greater than or equal to 3, where if circles were to be rendered between adjacent nodes, no other node would be present inside such a circle (Arens 2002). This is very similar to producing Voronoi polygons to partition a plane into regions based of the distance between specified points. The Bowyer-Watson algorithm (shown as Algorithm 1 below) is an incremental function for computing triangulation by adding vertices individual to an existing implementation and altering the till it is valid. The runtime of the algorithm is $\mathcal{O}(n^2)$ as can

[5] This is arbitrarily decided by picking two random numbers between 0 and 10, and dividing one by the other

be seen in 1 but can be reduced too $\mathcal{O}(nlogn)$ if appropriate data structures are used. The algorithm creates an initial triangulation from three nodes and recursively adds a new node. Each triangle is then assessed for overlapping with other triangulations and removed if found. Nodes no longer in a triangulation are then joined to the node most recently added and subsequently new triangulations are formed. This is conducted until all nodes are part of a valid triangulation. The advantages of using triangulation over a randomised implementation are that all nodes are appropriately joined together in optimal fashion as discussed. However, the computation time is significantly slower so the trade-off exist between speed and quality.

Algorithm 1 Bowyer-Watson algorithm for Triangulation

Input: $allNodes$ - maps node to its vector, specifining the position
Output: $finalTriangulation$ - Valid Graph Triangulation

1: *nodesNotTriangulated* ← *Array(allNodes)* ▷ A list of all nodes not yet Triangulated
2: *nodes* ← *get3RandomNode(allNodes)* ▷ Allocate 3 random nodes to temporary list
3: *nodesTriangulated* ← *nodes*
4: *nodesNotTriangulated.remove(nodes)* ▷ Remove added nodes from analysis array
5: *triangulation* ← *nodes* ▷ Adds the triangle represented by the three random nodes to a list of all triangles
6: **while** *nodesNotTriangulated is not empty* **do**
7: *newNode* ← *getRandomNode(nodesNotTriangulated)*
8: *overlappingTriangles* ← *getOverlappingTriangles(newNode, triangulation)* ▷ Identified using Unreals collision detection those triangles containing the new node
9: *triangulation.remove(overlappingTriangles)*
10: *joinNodes(newNode, getNodes(overlappingTriangles))* ▷ Reconnects the new node to the nodes whose triangles were deleted
11: *nodesTriangulated* ← *newNode*
12: **return** *triangulation*

Minimum Spanning Tree - Prims algorithm for finding the minimum spanning tree in a weighted, undirected graph is shown above as Algorithm 2. It is a greedy algorithm that finds the subset of edges of lowest total weight which connect every node, found by adding the smallest un-included edge to the tree after each iterations (Kozlova et al. 2015). The algorithm is detailed below. A node is picked, representing the initial tree. A mapping is then established, assigning a key value to each vertex, representing its distance from the tree. While not all vertices are included, the algorithm will iterate through the mapping and add the vertex to the tree which holds the lowest weighting from it. The mapping is then updated, so that key values represent potentially closer values.

After the minimum spanning tree is found, some edges from the originally graph are included. Currently, this is 35% of the total number of edges randomly picked but is dependent on the complexity condition. After this process, the corridors in the form of smaller rooms are re-rendered into the layout, utilising the collision engine. Directly horizontal or vertical lines are used to join each of the large rooms so that they do not overlap. Any corridors that these lines pass through are rendered. In the rare case that this does not connect two rooms, small rooms are rendered recursively until a connection is made, starting with the room itself, followed by the deletion of unnecessary corridors.

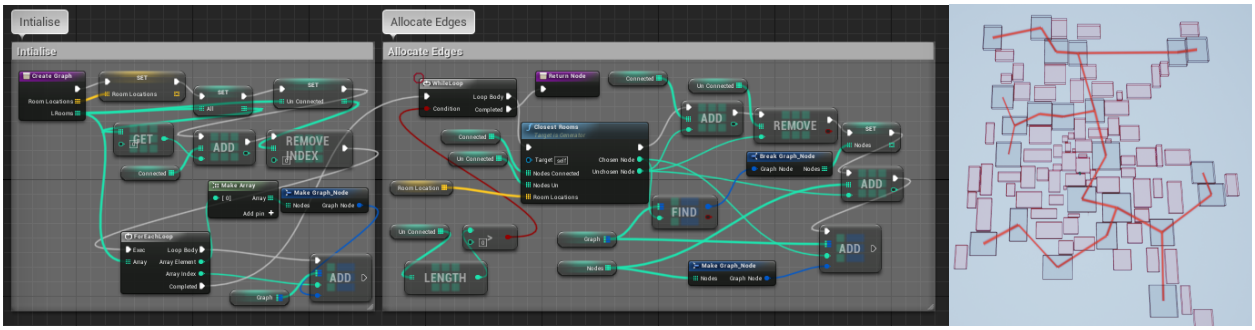
C.3 Derived Characteristic Evaluation, Inspired Analysis and Parametised Management

The project experimented with a number of techniques to either directly or indirectly, appraise or alter, in-game environmental layouts, giving the final application automotive knowledge regarding the quality of its generations.

Algorithm 2 Prims algorithm for Minimum Spanning Trees

Input: $graph$ - adjacency matrix between all nodes, $allNodes$ - list of all nodes
Output: $spanningTree$ - as a list of edges in the spanning tree

$currentNode \leftarrow getRandomNode(allNodes)$
$notIncludedNodes \leftarrow allNodes$
$notIncludedNodes.remove(currentNode)$ ▷ A list of all nodes not yet added to the tree
$keyValues[currentNode] \leftarrow \infty$ ▷ Nodes in the tree have infinity distance to it
$keyValueEdges$ ▷ Initialise edges represented by key values
for each *node in notIncludedNodes* **do** ▷ Loop through all not yet chosen edges
    $keyValues[node] \leftarrow graph[currentNode][node]$
    $keyValueEdges[node] \leftarrow currentNode$ ▷ Identification of Edge for key value
while *notIncludedNodes is not empty* **do**
    $currentNode \leftarrow findLowestKeyValue(keyValues)$ ▷ Node with lowest key value
    $keyValues[currentNode] \leftarrow \infty$ ▷ Nodes in tree have infinity weight, so are never picked
    $notIncludedNodes.remove(currentNode)$
    $spanningTree.append(graph[currentNode][keyValuesEdges[currentNode]])$
    for each *node in nodesNotIncluded* **do** ▷ Update all key values and edges
        if $keyValues[node] > graph[currentNode][node]$ **then**
            $keyValues[node] \leftarrow graph[currentNode][node]$
            $keyValueEdges[node] \leftarrow currentNode$
return $spanningTree$



(a) Code for Prims Algorithm in Blueprint Scripting

(b) The Mimium Spanning Tree for a Generated Level

Figure 5: The use of Minimum Spanning Trees

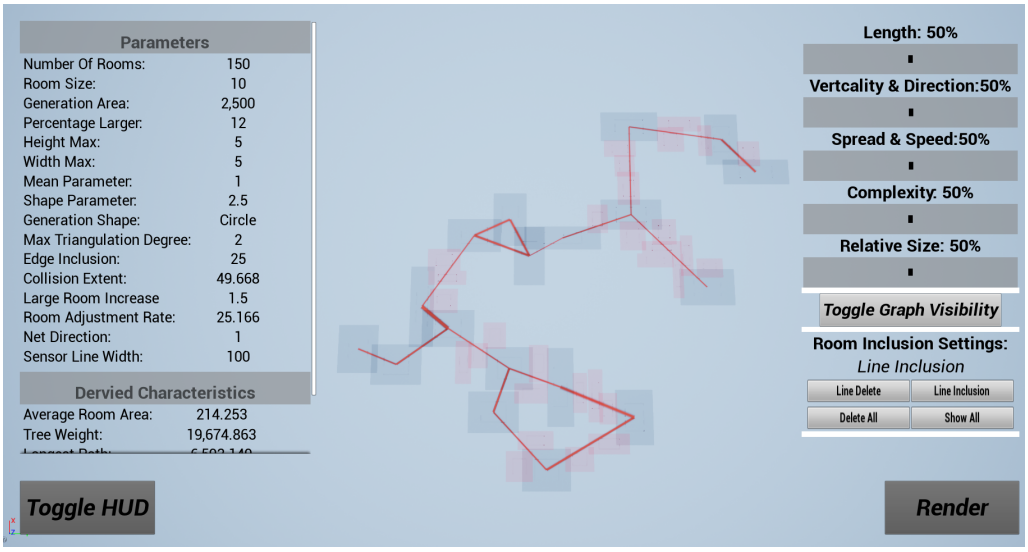


Figure 6: Prototype HUD for a given generation with Corridors rendered between Large Rooms

Derived Characteristic Evaluation - Both geometric (the actual rendering) and topological (simplified graph) in nature. Graphically, the distance between nodes, looping and node valency all contribute towards level complexity/connectivity factor and thus must be considered when evaluating a level. Applying graph theory concepts such as path length, tree weight and cycle lengths; are all applicable to numerically calculating level suitability. This is refereed to as Graphical Routing analysis, with each component being allocated relative importance towards the final output. Realisation of generated content in this way facilitates recognition of certain features. Fitness analysis is presented as a summarised report onto the HUD, shown in Figure 6. Characteristics originally specified to generate the level, the parameter initially altered and a subsequent breakdown of the final product are compiled.

Inspired Analysis - Manual evaluation and alteration is possible through the engine, with guidance from the system, highlighting potential areas for focus. Thus, future generations can improve on the features of its predecessors while preserving its desired character, following a procedure inspired by Answer Set programming implemented by Smith & Mateas (2011). Control for this currently is completely assigned to the user, given them control of all alterations, however in further iterations of the project, amendments could easily be modified through an automated process. The main developer collaborates through the HUD, presenting information and allowing interaction with the application. The right side contains features that allow alterations and the left contains a scrollable analysis panel. Following adjustments, a user can re-render the current request, and in order to navigate and examine the level in more detail, the HUD visibility is toggled. The topological overlay can additionally be adjusted along with small room inclusion. All can be included, deleted or choice render along/near edge traces, thus rendering corridors structures between main locations. As demonstrated in the showcase, choosing to hide large rooms and those located along the edge traces can yield labyrinth structures, ensuring navigational paths through the level via graphical verification.

Parametrised Management - There are five possible condition sets specified, detailing available level controls. These descriptors are available as percentage sliders, indicating desirability of each factor. Intuitively named, each governs initial parameter sets with unified weightings from each condition. *Length* dictates the overall size of the world built, *Verticality/Direction* the shape and room placement pattern, *Speed/Spread* a trade-off indicating how close rooms are rendered, *Complexity* the possible graphical scope and *Relative Size* regulating the large/small room relationship. For example, the collision extent which represents the amount of permitted overlap between rooms to trigger a collision, is affected by the length of level, how fast it should be generated and the direction/verticality measurement of each rooms position. The user is therefore not convoluted or confused by the volume of options when making specific alterations.

D Game Model Application

The model procedurally establishes rectangular structures on a plane with the ability to control their frequency, spread and the amount of overlap. Alterations to these condition sets alter the appearance and are thus useful for in game design. Adaptations to the model can yield a variety of structures within these parameters. In this instance, the underlying graphical structure is utilised to cut a labyrinth into the level, providing guaranteed navigational paths for a player, as shown in Figure 7. The graph can also be further utilised in order to provided guided placement

of interactable player objects, enemies and objective placement [6]. Interpreting the model in a variety of ways can create a powerful way to easily produce different games that follow developer ideas without requiring redesign of complicated underlying procedural networks.

E Testing - Verification and Validation of Implementation Success

Objectively identifying the extent of accomplishment that a user experience driven program achieves is ambiguous, making variation between generations difficult to measure. The evolution of derived characteristics over multiple renderings can be used to numerically observe the amount of change permitted between levels, however statistical analysis on such variables lacks meaning due to the absence of comparability with other models. Combined with this and the alterations to the initial parameter set, the most relevant way to measure success is to visually compare independent generations, thus displaying the level of versatility without making adjustment and showcasing the extent of the impact a user can have. This adds a further evaluative procedure for external authority on dissimilarity.

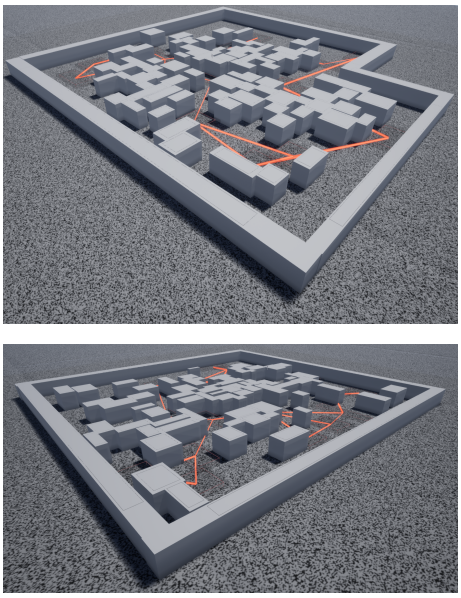


Figure 7: Views of the showcase with line deletion and partial spawning point rendering

There exists a trade-off between rendering speed and quality, which becomes exponentially more important as the size of levels increases. As discussed, room separation is a costly process, made cheaper by increasing both the increment with which rooms move apart and the collision area. However, varying these parameters can yield disjointed levels that cannot be navigated. Thus, directly measuring the time taken to make a level, and how playable the result is, is a central measure of success, particularly given comparisons available with other models.

RESULTS

Table 1: SYSTEM SPECIFICATIONS

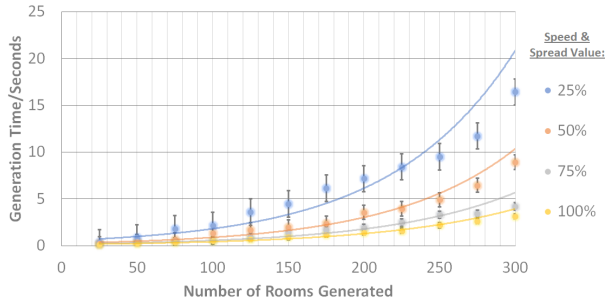
Component	Details
Operation System	Windows 10.1809
CPU Processor	i5-8400 2.80GHz
RAM	16GB, 2400MHz
GPU	Nvidia 1060, 3Gb

In this Section, the levels generated by the program will be presented in a variety of formats, enabling evaluation of the solution to follow. All experiment configurations were conducted and recorded using the specifications specified in Table 1. Additional tests were completed with an executable on a 2015 MacBook Pro running Mojave and Linux Ubuntu 18.10 using the same system specs as in Table 1, verifying the cross platform launch ability while keeping performance results standardised. Unless stated otherwise, all other conditions used for aggregated control over parameter inputs are held constant at "50%" while tested. In addition, the Figures for each experiment are chosen for being appropriate representations of a given setup. While variation in each Figure takes place by definition, the probability of inappropriate generations occurring is discussed where relevant.

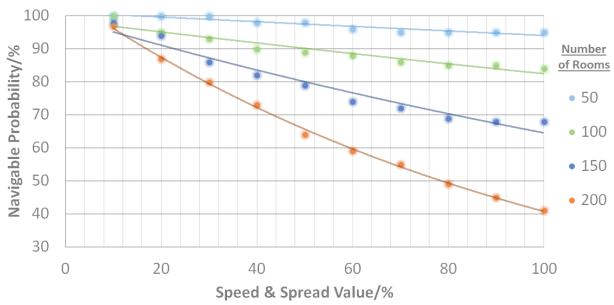
[6] a start and end position for the player

A Computation Time and the Probability of Inappropriateness

The time taken to produce a level is an important factor. Diminishing returns are observed when investing additional time to implement additional control measures and smaller, more precise alternations when compared to the result. Figure 8a shows the time taken to generate a level for different speed criteria and room frequency, sampled individually over 10 cases before averaging and displaying the error bars. As discussed in the Solution, the relationship is exponential, requiring additional time with each new room, given each is placed centrally and has further to travel before reaching the 'map circumference' as it has to be moved past each room before it. This is potentially problematic for more ambitious level sizes, reaching 16.446s for 300 rooms at guaranteed precision, a relatively small number of rooms but as shown, this can be partially reconditioned by sacrificing precision. This penalty nonetheless increases with room frequency.



(a) A Graph to show how time taken to generate a level varies with the Number of Rooms over alternate Speed/Spread Criteria



(b) A Graph to show how the probability a level is completely navigational varies with the Speed/Spread Criteria over alternate numbers of rooms

Figure 8: Graphically plotted data highlighting Generation Times and Suitability

The trade off between quality and time taken is a common issue in PCG but the time recorded that is required to produce a valid level after correcting for immediately detectable impurities. However, due to a limit on computational interpretation this is not always effective. Testing for this, in Figure 8b, shows how the probability that a graph is incomplete /unnavigable during line inclusion varies with the *Speed* it was generated at for different level sizes. The drop off is significant for larger levels and speeds, where the number of instances increased the feasibility of at least one inaccurate adjustment to 41%. Of course, it is possible to assess this factor computationally and adjust, either adding rooms or diagonal structures to fix connectivity. This however reduces the organic appearance and expands production time. There are multiple other factors to consider such as the edge width and collision extent which affect inappropriateness and can thus be altered to improve results, potentially solutions for producing larger levels in the same time. Important to note is this is only relevant for the line inclusion strategy, since other display methods do not rely on level navigation, randomness balances ensures relevant production.

B Content Variation and Parameter Control

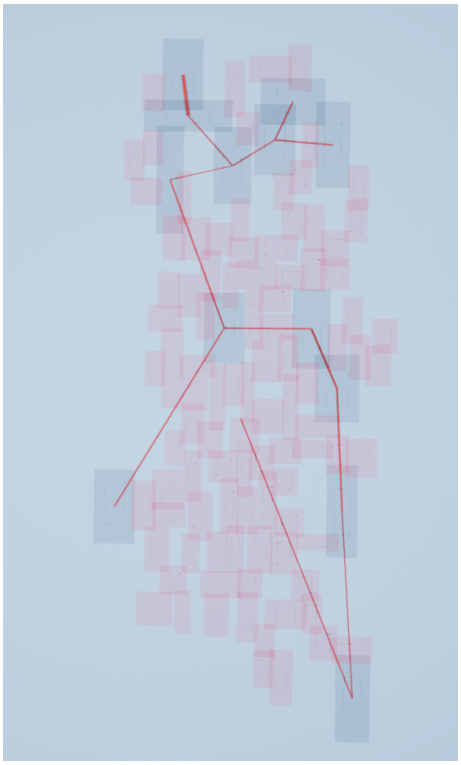
A central requirement was to specify desirable characteristics, influencing creations. The degree to which this takes place is arbitrarily difficult to measure given that user experience is problematic to quantify. Therefore, to establish success, visual variation will be used to identify

the outcomes alterations have including the potential ramifications that result as side effects of diversification. This is discussed below and displayed in Figure 10.

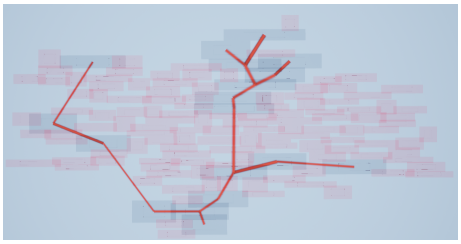
Verticality & Direction - Game mechanics vary in their demand on a player position. The user must knowing intuitively the direction they must proceed and the environment they encounter should be appropriate for matching their ability to face it. For example, if combat focus is vertically grounded, so a character shoots 'up' the screen, such a level must be shaped to appropriate accommodate this. Controlling the distribution and shape of the rooms is therefore a vital alteration as shown in Figure 9a and 9b, which have *Verticality* set to 20% and 80% respectively. A lower score perceivably, flatters out rooms and renders them in horizontally longer spaces, encouraging spread out similarly, with th opposite true during vertical focus.

Complexity - Shows aggregate control of the graphical representation, indicating the edge inclusion ratio, generation method and edge width during collision rendering. A higher value therefore leads to a better connected graph with more edges between nodes as shown in Figures 10b and 10a. This also has an adverse effect on evaluation characteristics, directly enlarging the frequency of cycles and overall tree weight. Alterations therefore to the value therefore make it difficult to directly compare levels based on variables alone. Figures 10e - 10h shows an example level alteration system where each edge is given a width in space and causes hit trigger collisions in the small rooms it passes through. The user can choose to delete or render such rooms depending on the design constraints.

Relative Size - ensures larger rooms have greater significance in the overall level. Large Rooms are capable of acting as focus for activity, enemy locations or open spaces during labyrinth mode. Figure 10c - 10d shows clearly the corresponding increase in large room size, location centrality and frequency.



(a) Verticality = 20%

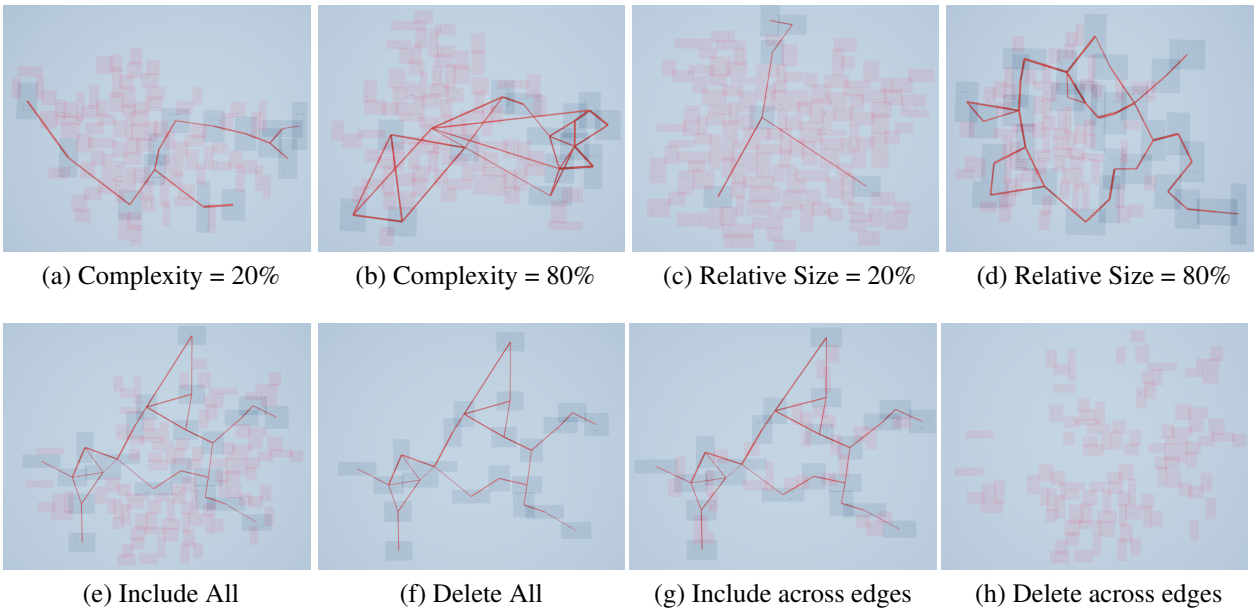


(b) Verticality = 80%

Figure 9: Verticality & Direction alteration affects

EVALUATION

This Section focuses on evaluating the solution against other similar methodologies, using previously presented results. Due to program idiosyncrasy, each design component needs to be assessed individually and as part of the whole program with the effectiveness of their interconnective success. Where little literature exists for comparison, the relative strengths and weakness are discussed against expectations, indicating why performance values are chosen and evaluating conceivable improvements to program trade-off deficiencies. This will be followed by an appraisal of the project's organisation and contents, justifying the decided areas of focus while addressing conceivable advancements to model design.



(a) Complexity = 20% (b) Complexity = 80% (c) Relative Size = 20% (d) Relative Size = 80%

(e) Include All (f) Delete All (g) Include across edges (h) Delete across edges

Figure 10: Showing Content Variation after alteration to Desired Criteria. Unless specified, all Rooms are Rendered

A Comparative Strengths and Limitations of the Solution

The work presents itself as an effective mechanism for game design, with abundant insight for developers to work with and versatile scope adequate for on-demand dungeon production. There is therefore extensive breadth for further applications. Developers wishing to develop a PCG game are not restricted by the considerable time taken to produce randomised algorithms and can instead focus on the games message. Avoiding this repetition is an uncommon feature of PCG, usually designed with an exclusive ambition. Given that dungeon size is all algebraic, computed relative to itself on a consistent scale, multiplying by fixed provided parameters can increase and decrease the size relative to the player, which when combined with the functional condition system, provides a smooth way to prototype a level. The management of randomness in this technique is more lenient than traditional graphical grammar approaches, allowing for significantly more capacity for emerging creative content, without comparative sacrifice of control over contents. Hendrikx et al. (2013) observed that PCG is commonly not recyclable and this implementation has repaired this flaw, at the cost of definitive focus. Also important, is that not all designs can follow the reproducible approach. The program focuses on varying room shape and design in an organic manner, therefore not extending itself towards the other techniques discussed in the survey, limiting productions to dungeon-based designs. As discussed, labyrinths and other structures are however still possible through creative application and this adaptability is thus considered possibly the greatest strength.

Perfect balance between control and feature desirability is fundamentally insurmountable. The implementation does an effective job of curtailing the net loss from this trade-off; nevertheless, implementing heavy adaptions to augment the model may severely damage the quality of the relationship. By specifying more explicit characteristics, there are a reduced finite set of possible outcomes that satisfy the demands, and while other implementations such as Valtchanov

& Brown (2012) maintain greater control of randomness by proceeding with graph grammar analysis, we instead invert this relationship. This is an inevitable consequence of the design.

The use of the Unreal Engine could also be perceived as a limitation. A game engine is an example trade off between ease of implementation and exemption freedom, providing an established environment that has pre-implemented the basics but can become strenuous to alter if a definitive goal is in mind [7]. The engine offers an easy encapsulation for deployment, mechanism manipulation and debugging. However, due to the engine's complexity, potentially unnecessary overhead is realised. This is particularly apparent in extensively large models, where due to finite generations and the exponential run time, normal systems would be unable to render the level in a realistic time period. Also imperative to consider is unification, given that at some point knowledge of internal software workings is required to make explicit alterations. Other models have more alteration options, an easier generation methodology to understand and do not require understanding of the relatively unique Blueprint scripting language; so appropriateness of the model under more complicated categorical examples is degraded. In this way the model analogously acts as a 'jack of all trades, master of none' in its universal development capabilities.

Aforementioned, automated editing is parameter dependent, so programmed flagging requires too many considerations for a general instance and must therefore be made specific to the game. Repeated generations yield data specifying the variation in recorded values and thus allowing computational recognition of extreme results by previous instance distributions. Data analysis is therefore only useful when viewed in relation to previous analysis or when tested manually to generically label particular generations for the level of appropriateness they achieve. Such a method however, is limited to the scope and detail of human experimentation before release to train the model to react appropriately to given instances and recognise symbolic variable changes. This extends the model from artificial intelligence to machine learning or a similar field, providing possible scope for further research.

B Resolved Algorithms and Distributions

Triangulation/Connectivity mechanism - currently a method adapted from Arens (2002) for Delaunay Triangulation is being used for graphical rendering and then a percentage of edges kept for the primary graphical establishment mechanism. Methods such as alternative algorithms that achieve the same effect exist, such as the Divide and Conquer algorithm, generally considered the fastest solution, which involves recursive splitting of node sets. The efficiency depends on the specific nature of the problem; integrating the the calculation into C++ using an established library may be more efficient. Alternatively, different methods for rendering the graph could be used, placing varied emphasis on edges. Exploration in this area could facilitate even greater control over graph production, as well as accelerating computation.

Wald Distribution - The Inverse Gaussian Distribution, used for room size control. As discussed in the Solution, the mean and shape parameter are used to control the function. However, this does not provide complete jurisdiction, limiting the function to the established shape. Constant multiplication for large and small rooms, by a constant artificially alters the final result, however alternative distributions that accept additional parameters may allow for enhanced supervision of Room sizes for example, by adjusting the tail on the larger end so fewer large rooms are generated nevertheless those that are, have a greater size variation. Such a distribution would

[7] A perfect analogy for the use of the final program during assisted development

allow for greater randomness management and also a greater variety of potential levels.

Evaluation and Control - Do not implement a specific algorithm but rather link named criteria to initial parameters, simplifying alteration to understandable statistics. This process however is subjective, with any changes made to criteria having arbitrary influence over parameters. Additional testing could yield more appropriate relationships between criteria and parameters, or alternatively, integrate more conditions together for added imagination. This would be assisted by offering more characteristic analysis from both the graph and level to encapsulate more components and better identify peculiarity. Further, generic fitness analysis could abstract from these derivations, presenting a succinct value, eliminating the need for human identification of better generations. The trade off for this is less compared to the methodology discussed in Adams et al. (n.d.) given that the graph already exists and is not produced purely for appraisal.

C Improvement Opportunities and Extensions

Sharpened Interactivity - While possible within the engine, the ability to add, move or alter rooms the graph by a user will give even better scope and control balance outside of the engine. This will allow a user to act with greater freedom after viewing data advice and also experiment with possibilities. Such an interface could be integrated into the HUD, with manipulation expressed as visualised versions of Room Blueprints, with native Actor modification in Unreal simply extended to the executable. This will enable the full potential of the software and remove the constraints placed by requiring the engine to function.

Advanced Computational Inspiration - Specifying with greater precision the desired features of a certain level within the outlined methodology with the inspiration process, while allowing generations to still vary significantly between iterations while meeting the parameters specified. For example, Pointless Area Analysis involves the identification and alteration of areas that fail to show purpose. This includes ones which do not adequately reward a player for exploring them, resulting in back tracking or otherwise leaving a player dissatisfied. Discovering such areas is possible by replicating graph analysis in subsets, identifying areas that more significantly affect aggregate results. The identification of such areas can result in their automated removal or flagging to a developer. Utilising better parallelisation or more efficient algorithms could offer speed improvements on top of this. Additionally, assessing only components deemed likely to offer efficiency improvements will assist, with relative identifications compared to computation installed as a quality metric. Encapsulating more outcomes and equipping algorithms with improved recognition will enhance performance time, appropriateness and computational guidance.

Advanced Fitness evaluation following Level Population - Games with an established set of player mechanics, designed to exploit and combine such mechanics in interesting ways, is something that cannot be left up to chance and is therefore unsuited to PCG (Liapis et al. 2015). Levels still lack significant empirical made only more complicated by mechanic incorporation. This encourages a player to explore but limits the level or reward they get from doing so. Testing very particular skills that maybe available in the game, such as juggling a specific navigation puzzle made problematic by an enemy cleverly based to prevent completion cannot not always be guaranteed. A 'best' level does technically exist for the specific needs, find this from a set of ones that meet the desired parameters. Fitness characterisation will enable the model to find a level that does not just fit conditions but the best one that fits conditions.

Graph Evolution - involves adding to gameplay possibilities beyond basic maze solving, showing the scope for the PCG evaluation can incorporate additional components. Adapting

from Togelius et al. (2011), who use action graphs to follow the tasks a player must complete, we can explore the possibility of adding closed-off components only accessible if a key has been obtained in a level. Analysis is possible by converting to a directed graph, so routing shows player movement in order to complete the level. Alternatively, the addition of threats to the world will yield a similar complication. This could be incorporated into the evaluation by producing sub-graphs for each component and adding node characteristics. Assessing each graph therefore yields information on object density and relative placement, facilitating assessment. The second methodology suggested by Adams et al. (n.d.), directly alters the graph, combining near nodes into "super-nodes" helping to distil unnaturally rendered content.

CONCLUSIONS

The aim of this project is to establish PCG that meets player expectations, achieved through a sophisticated mechanism converting randomly created rooms into structured designs, followed by intelligent assessment. Anticipated was a partial solution to the uncontrollable randomness of PCG and by this measure it has been a success. The inexorable trade off with the range of material that could be produced and its correctness is enduring and immensely challenging, however the model created has got closer to minimised loss that expected. Superficially with improper balance, different levels can be produced but lack meaningful structure, failing to provide the illusion of a creator. As discussed in the previous Section, the program has its limitations, however many are fundamental bindings to unnecessary possibilities inside PCG. Thus, it is demonstrated that PCG can be replicated for complete game design with minimal scope for content loss under a large outlook. The work provided is of potential interest to those inexperienced with game design or unfamiliar with specific coding protocols, installing an ease to interpret generation system. Equally of use those it is to professionals wishing to experiment with different mechanics or game concepts without extensive production overheads. Finally, as shown during the showcase, the method can be used on the fly to produce a varied but appropriate gaming experience.

Future work would suitably be directed towards each of the possible extensions discussed in the preceding Section, with focus on evaluation and fitness. Computational recognition of interesting content is a direct requirement for on-demand deployment and additionally assists in the developing process by highlighting. The model due to its geometric and topological nature provides a suitable encompassment for experimenting by providing many needed variables as direct consequences of generating, something Summerville et al. (2018) address usually as a difficulty, now presented as a strength. The central issue found was a reliance on data/knowledge on the relative significance of how derived values change over iterations. Investigating better understanding of the data's connotations falls under the scope of machine learning approaches, employable using the model that has been built thus far.

Although the reuseability of some models exceeds others, the PCG focus of producing certain types of content acts a constraint to recycling. However, doing so repeatedly generates an intractable environment, populating it with fresh challenges and relative rewards. Underlying the objective of most video games (outside of economic benefits) is an interest factor, a measure of fun that captures attention. While not essential, some players go to games for alternative challenges not witnessed in usual activities and the intelligence at which content is produced and computationally analysed, plays a significant role in the quality and lasting nature of their experience. PCG improvements and computational involvement, as demonstrated; and will continue to contribute more too and play a larger part in game development.

References

Adams, C., Parekh, H. & Louis, S. J. (2017), Procedural level design using an interactive cellular automata genetic algorithm, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference Companion', ACM, pp. 85–86.

Adams, D. et al. (n.d.), 'Automatic generation of dungeons for computer games', *Bachelor thesis, University of Sheffield, UK.* .

Arens, C. (2002), 'The bowyer-watson algorithm; an efficient implementation in a database environment', *Case study report TU Delft, July 2002, 52 p.* .

Ashlock, D., Lee, C. & McGuinness, C. (2011), 'Search-based procedural generation of maze-like levels', *IEEE Transactions on Computational Intelligence and AI in Games* **3**(3), 260–273.

Compton, K. & Mateas, M. (2006), Procedural level design for platform games., *in* 'AIIDE', pp. 109–111.

Cook, M. & Colton, S. (2014), A rogue dream: Automatically generating meaningful content for games, *in* 'Proceedings of the AIIDE workshop on Experimental AI & Games'.

Dinh, P. (2012), Tinykeep. Available at: http://blog.tinykeep.com/, accessed October 2019.

Foltin, M. (2011), Automated maze generation and human interaction, PhD thesis, Masarykova univerzita, Fakulta informatiky.

Hendrikx, M., Meijer, S., Van Der Velden, J. & Iosup, A. (2013), 'Procedural content generation for games: A survey', *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* **9**(1), 1.

Kozlova, A., Brown, J. A. & Reading, E. (2015), Examination of representational expression in maze generation algorithms, *in* 'Computational Intelligence and Games (CIG), 2015 IEEE Conference on', IEEE, pp. 532–533.

Liapis, A., Holmgård, C., Yannakakis, G. N. & Togelius, J. (2015), Procedural personas as critics for dungeon generation, *in* 'Eu Conference on Applications of Evolutionary Computation', pp. 331–343.

Scales, D. & Thompson, T. (2014), Spelunkbots api-an ai toolset for spelunky, *in* 'Computational Intelligence and Games (CIG), 2014 IEEE Conference on', IEEE, pp. 1–8.

Shaker, N., Liapis, A., Togelius, J., Lopes, R. & Bidarra, R. (2016), Constructive generation methods for dungeons and levels, *in* 'Procedural Content Generation in Games', Springer, pp. 31–55.

Smith, A. M. & Mateas, M. (2011), 'Answer set programming for procedural content generation: A design space approach', *IEEE Transactions on Computational Intelligence and AI in Games* **3**(3), 187–200.

Smith, G., Treanor, M., Whitehead, J. & Mateas, M. (2009), Rhythm-based level generation for 2d platformers, *in* '4th International Conference on Foundations of Digital Games', ACM, pp. 175–182.

Summerville, A., Snodgrass, S., Guzdial, M., Holmgard, C., Hoover, A. K., Isaksen, A., Nealen, A. & Togelius, J. (2018), 'Procedural content generation via machine learning', *IEE Transaction on Game* .

Thompson, T. (2015), " with fate guiding my every move": The challenge of spelunky., *in* 'FDG'.

Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M. & Stanley, K. O. (2013), Procedural content generation: Goals, challenges and actionable steps, *in* 'Dagstuhl Follow-Ups', Vol. 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Togelius, J., Yannakakis, G. N., Stanley, K. O. & Browne, C. (2011), 'Search-based procedural content generation: A taxonomy and survey', *IEEE Transactions on CI and AI in Games* **3**(3), 172–186.

Valtchanov, V. & Brown, J. A. (2012), Evolving dungeon crawler levels with relative placement, *in* 'Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering', ACM, pp. 27–35.

Van der Linden, R., Lopes, R. & Bidarra, R. (2013), Designing procedurally generated levels, *in* 'Proceedings of the the second workshop on Artificial Intelligence in the Game Design Process'.

Van der Linden, R., Lopes, R. & Bidarra, R. (2014), 'Procedural generation of dungeons', *IEEE Transactions on Computational Intelligence and AI in Games* **6**(1), 78–89.