Faculty of Science and Technology

BSc (Hons) Games Programming

May 2018

Implementation of a Procedural Level Generation Engine for Developing Rouge-like Dungeons

By

Kane White

DISSERTATION DECLARATION

This Dissertation/Project Report is submitted in partial fulfilment of the requirements for an honours degree at Bournemouth University. I declare that this Dissertation/ Project Report is my own work and that it does not contravene any academic offence as specified in the University's regulations.

Retention

I agree that, should the University wish to retain it for reference purposes, a copy of my Dissertation/Project Report may be held by Bournemouth University normally for a period of 3 academic years. I understand that my Dissertation/Project Report may be destroyed once the retention period has expired. I am also aware that the University does not guarantee to retain this Dissertation/Project Report for any length of time (if at all) and that I have been advised to retain a copy for my future reference.

Confidentiality

I confirm that this Dissertation/Project Report does not contain information of a commercial or confidential nature or include personal information other than that which would normally be in the public domain unless the relevant permissions have been obtained. In particular, any information which identifies a particular individual's religious or political beliefs, information relating to their health, ethnicity, criminal history or personal life has been anonymised unless permission for its publication has been granted from the person to whom it relates.

Copyright

The copyright for this dissertation remains with me.

Requests for Information

I agree that this Dissertation/Project Report may be made available as the result of a request for information under the Freedom of Information Act.

Signed: 

Name: Kane White

Date: 18/05/2018

Programme: BSc GP

Acknowledgements

The successful development of this project would have been impossible if not for the continued support of my loving parents throughout my time spent here at Bournemouth University.

I would also like to thank Simant Prakoonwit and Fred Charles, my primary and secondary supervisors for this project, for their excellent advice and support on the journey from its conception to its completion.

Omar Cornut & Michał Cichoń deserve special mentions for their kind-heartedness in offering their time and energy through the form of their open-source software's that helped form the basis for this project.

Abstract

Procedural content generation techniques have been a popular and fast growing method of automatically creating content for games in the past few years, by utilising these techniques it enables the creation of a mass amount of varied content in a short amount of time.

Procedural generation in relation to the creation of a dungeon level is a construction of two interacting components, a mission and a space, the mission denotes the objectives or actions a player must perform to complete the level, whereas the space describes the topology and geometric arrangement of the level.

The procedural generation technique that has been implemented in this project utilises a form of describing a level through what is known as transformational graph grammars, these grammars consist of a set of nodes and edges that when combined create a graph. The construction of a graph relies on predefined rules, or 'sub-graphs' that are used in combination with random generation to decide how the graph is constructed. These rule sets are specified by the designer and can then be used to generate an infinite number of random levels.

In addition to the graph grammar system an evaluation algorithm is used to determine the size of the level, this allows the designer to specify a minimum and maximum size and distances of traversal of the generated level.

Contents

1.0 Introduction

1.1 Context

The sub-genre that is rouge-like games is derived from role-playing games and consists of a player having to navigate through a labyrinthine scene known as a 'dungeon' whilst confronting enemies and searching for items using turn-based game mechanics. Rouge-like games have been developed since 1975 and the genre has been heavily influenced by table-top role-playing-games (RPGs) such as *Dungeons & Dragons(1974).* The name 'rouge-like' comes from the game 'Rouge'(1980) which came packaged with the UNIX operating system Berkley Software Distribution (BSD) from version 4.2 in 1983 (Brewer 2016)**.**

Early rouge-likes were developed with the BASIC programming language and designed to be run on shared mainframe systems through MS-DOS/UNIX, meaning there was limited memory availability for data storage which lead to developers to generating random procedural levels to surpass the storage issues.

As the genre has developed the game content has evolved to include complex puzzles, these puzzles are usually integrated within the game play. One mechanism of expressing such puzzles is through the form of locks and keys.

1.2 Aims & Objectives

This project aims to implement procedural generation through the use of transformational graph grammars to develop a level generation engine for the creation of rouge-like game levels via rules specified by the designer. The engine will be developed using the Visual Studio IDE and written in C++.

The solution will be evaluated using a top down approach, through visual analysis of the generated levels alongside extracting and investigating empirical variables from the algorithm implementation.

2.0 Literature Review

2.1 Concepts & Terminology

2.1.1 Procedural Content Generation

The term ‘Content’ in the context of procedural content generation for games relates to what is enclosed within the game, this includes items, enemies and their AI, levels, maps, music, non-player characters and interact-able objects. The choice of application is dependent on the designer, for example one designer might choose to generate a mountain range using a height-map and noise functions, whereas another designer may choose to generate a procedural planet complete with vegetation (Vuontisjärvi 2014). Some games have even taken it a step further and have used procedural generation to create entire worlds bringing with them new and uncharted areas, including procedural climates that simulate the existing laws of nature alongside procedural forests and fauna that populate these areas. (Johnson 2017)

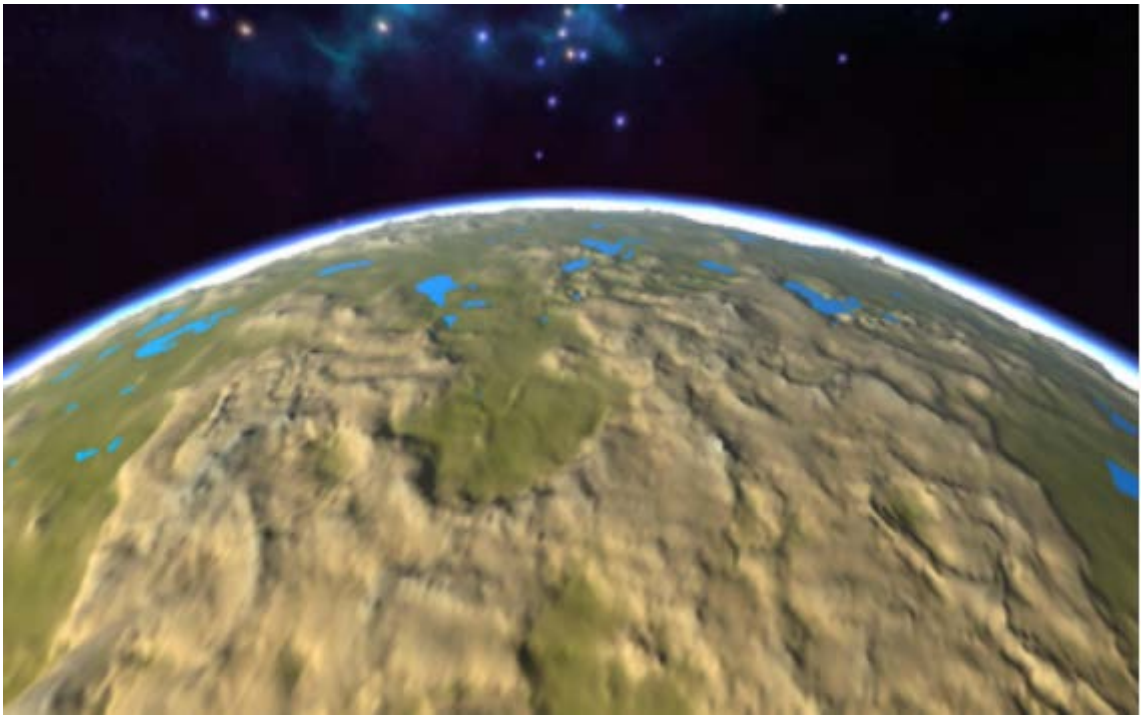


Figure 1 Example of a Generated Planet with Procedural Vegetation in “Planetary Terrain” (Johnson 2017)

The other key terms ‘Procedural’ and ‘Generation’ indicate that there is the involvement of a computer algorithm that creates and outputs content either on its own or with some human assistance, a PCG system can be defined as an application that implements a method of procedural generation, for example a system that implements artificial intelligence assisted game narrative generation. (Julian Togelius 2016c)

2.1.2 Procedural Rules & How they Work

A ‘rule’ is a method of representing information about a transformation that is performed on the generated content, a simple example of this is specifying limiting values for the number of doors a generated room should contain.

To create rules that can be applied to content a formal definition of the quantifiable aspects of the games rules is required. Through the definition of specific features of the game to formalise into rules the interesting aspects of exploration and variation become apparent (Mark J. Nelson 2016). By determining the specific genre of the game one can extract the meaningful elements that create the defined genre, for example a rule encoding for a turn-based game like Rouge (1980) would be able to assume the games timing runs at alternating discrete turns and has objects positioned throughout the game-space, with the game-play being heavily reliant on moving these objects around. Alternatively a rule encoding for a game like Pong (1972) would rely on the fact that game-play is based a continuous time.

2.1.3 Grammars

The concept of generative grammars originates from linguistics, a finite set of recursive rules which define how to create different phrases and sentence structures, used in combination with an alphabet and a set of *‘terminal symbols’* that must appear in the production one can generate any phrase within the language. The example rule seen in Fig 2 takes ‘A’ and replaces it with ‘a’. (Chomsky 1957)

$$A \rightarrow \alpha$$

Figure 2 Generative Grammar Rule Replacement (Chomsky 1957)

2.1.4 Graph Grammars

The technique defined by linguists’ generative grammars is transferrable into other domains, such as graphs, as described in *The Handbook of Graph Grammars* written by Joost Engelfriet (1997) one can model graphs using a set of recursive rules or a set of *‘sub-graphs’* to expand and generate a graph, see figure 3 & 4 for an example graph & sub-graph.

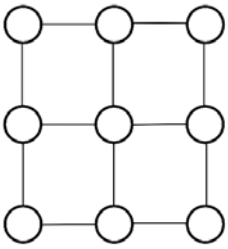


Figure 4 Example Graph (See appendices)

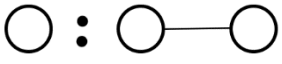


Figure 3 Example Sub-graph (See appendices)

2.1.5 Shape Grammars

The use of generative grammars extends not only to graph grammars but also shape grammars, this implements the painting of shapes defined through a rule set, the right side of a painting rule get painted in the form that is defined by the left side of the rule, an example of this can be seen in figures 5 & 6. This technique was first introduced by George Stiny (1972).

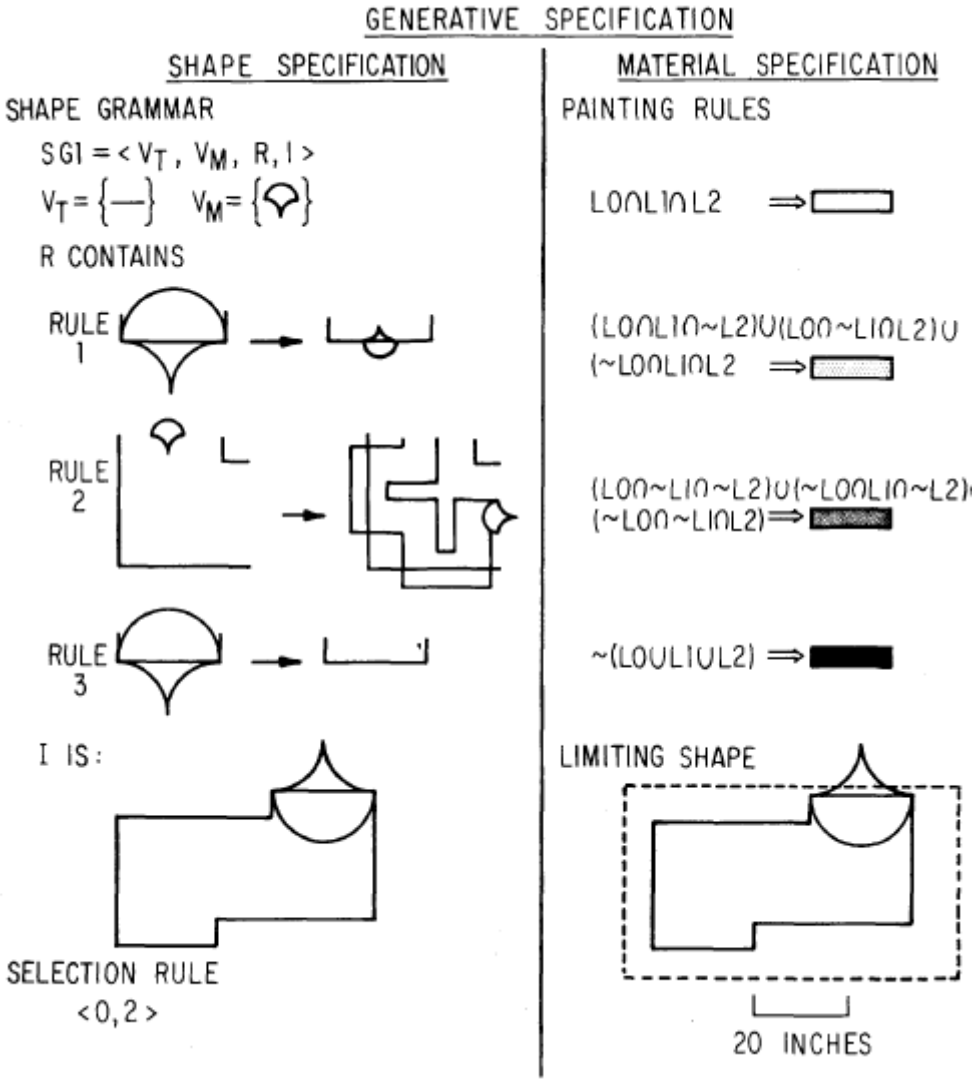


Figure 5 Generative Specification for Shape Rules (Stiny 1972)

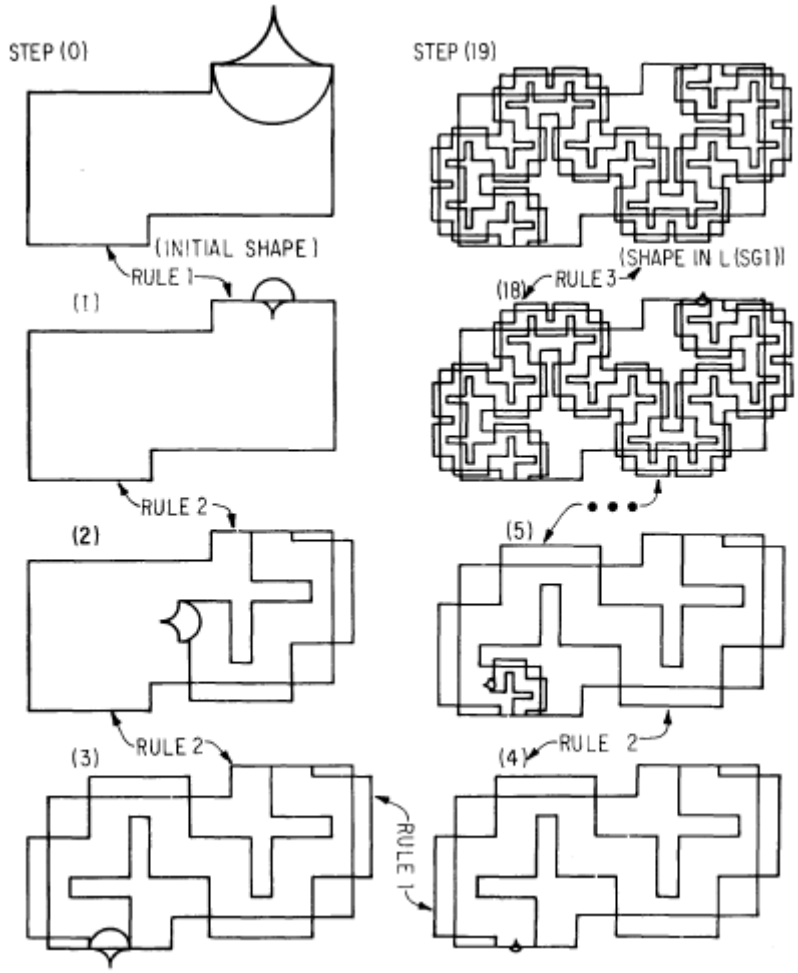


Figure 6 Generation of a Shape Using Gen. Spec. in Fig. 5 (Stiny 1972)

2.1.6 Random Numbers

The generation of procedural content relies heavily on random numbers, usually the goal of PCG is to provide each individual player or group of players with a unique experience, and while varied choices can be generated in a completely deterministic manner, most PCG systems employ an approach that generates a series of random numbers that vary each time the game is played to produce the desired variation. The generation of numbers by such systems is termed 'Pseudo-random' due to the fact that these number sets aren't truly random, rather are a series of numbers that appear to be random. There are many ways to alter the distribution of random number generation, the different methods produce varying spaces of results, see figure 7 & 8 for some examples.

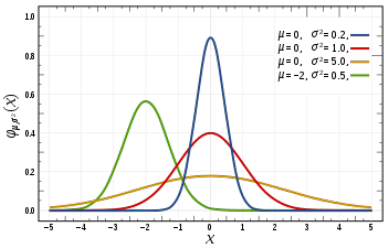


Figure 8 Gaussian Distribution

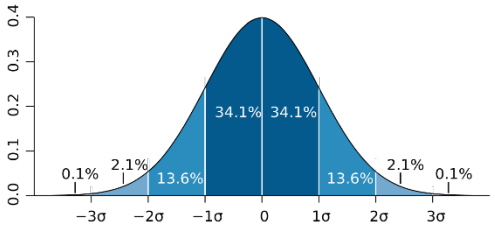


Figure 7 Standard Distribution

2.2 Analysis of Current Procedural Generation Techniques

2.2.1 Pseudo-Random Number Generators (PRNG)

PRNG's are initialised with a starting seed value and upon use the PRNG performs a mathematical operation on the seed value to increment the next value in the series by some amount, this ensure that the next call will be statistically random.

When a PRNG is given the same initial seed value the numbers generated in the series are identical to the previously generated values, this becomes extremely powerful when creating multi-player PCG games as two different players on separate computers can input identical seeds and find themselves in the same unique randomly generated world.

The general seed for PRNG's is a 32-bit integer, usually the current time is used as a seed to provide varied results each time it is run if the desired output isn't intended to be repeated or shared (Kyzrati 2017). Using a regular 32-bit integer such as 5197665346 is not an optimal solution when looking to create sharable seeds, instead a memorable string such as "SuperSeed" can be converted into a 32-bit integer which allows the player to easily remember and supply their seed to others.

To utilise a string as a seed one requires a method of converting a string into a 32-bit integer, fortunately such methods have been widely researched and implemented across all forms of computer-science they are known as 'Hash Functions'. Many different types of hash functions exist and are useful for a variety of applications, some hash functions return more statistically random outputs *(SHA)* which is useful for applications such as cryptography whereas less statistically random output hash functions that run with less memory-overhead are the norm for game related hashing. (Bucklew 2017 P.272-274)

A large issue that comes with PCG is error-reduction, seeds offer the ability to replicate and debug random generation issues that appear as all that is required to return to the generated state is the input seed.

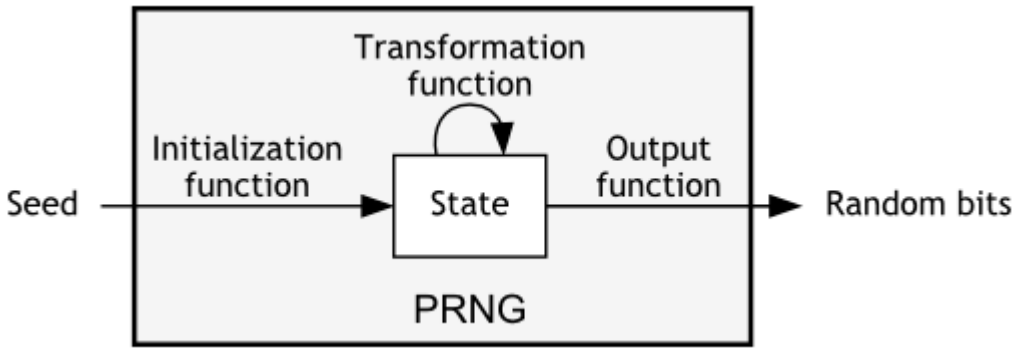


Figure 9 PRNG State Diagram (Vening 2015)

2.2.2 Space partitioning Algorithms

Space-partitioning algorithms implement a technique that subdivides a 2D or 3D space into subsets or *'cells'* that can be combined to represent the entire subject. This type of algorithm works recursively in a hierarchical structure to subdivide each cell further until all spaces are partitioned. (Noor Shaker 2016)

The representation of the output from a space-partitioning algorithm is known as a *'space-partitioning tree'*, this representation means that any point in the space can be queried extremely quickly which results in high performance for computer graphics applications such as rendering, ray-casting/tracing, shadow generation, frustum-culling and collision detection. (Ahmad ca. 2016)

The most prominent space-partitioning method in use today is known as a *'Binary Space Partitioning'* this method divides the space recursively into two sub-sets, with its form of representation being a binary tree. These algorithms implement different techniques to split the hyper-planes of the input based on the space they are applied to, for example a 2D BSP implements a quad-tree that splits two-dimensional space into four quadrants (See Fig. 7), whilst a 3D BSP implements an octree to split three-dimensional space into eight octants (See Fig. 8). (Noor Shaker 2016)

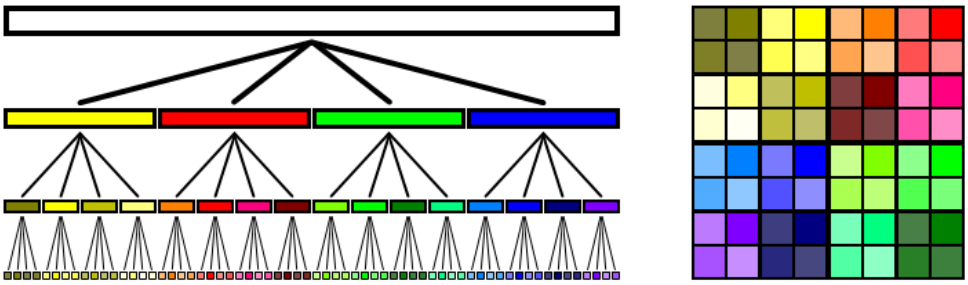


Figure 10 Example of a Quad-tree BSP for a Texture Atlas (Chow 2010)

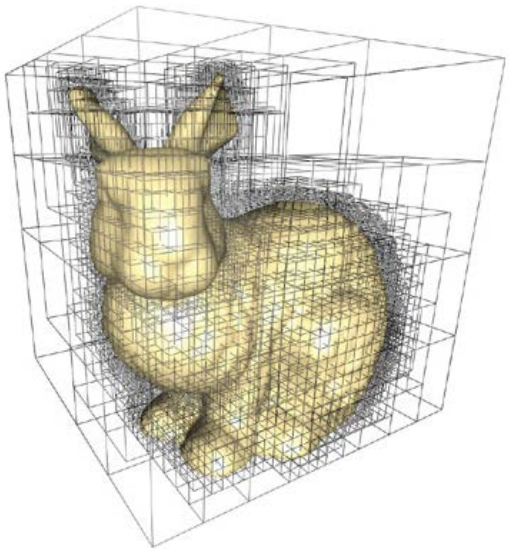


Figure 11 Example of an Octree BSP for a Mesh .Obj file (Chalmers 2013)

2.2.3 Binary Space Partitioning for Dungeon Generation

Implementing a BSP algorithm for dungeon generation ensures evenly spaced room layout and fully connected rooms with no overlapping rooms, it does this by recursively splitting the space until all cells are the approximate size of a room. The basic implementation of this algorithm is as follows:

Choose a random direction : horizontal or vertical splitting

Choose a random position (x for vertical, y for horizontal)

Split the dungeon into two sub-dungeons

Pseudo-Algorithm Source: (Unknown 2017)

Iteration One Result:

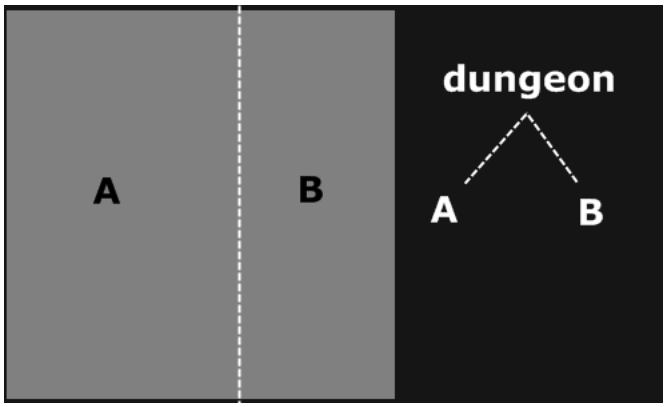


Figure 12 BSP for Dungeon Generation iteration one (Unknown 2017)

Iteration Two Result

(Apply Algorithm to both cells):

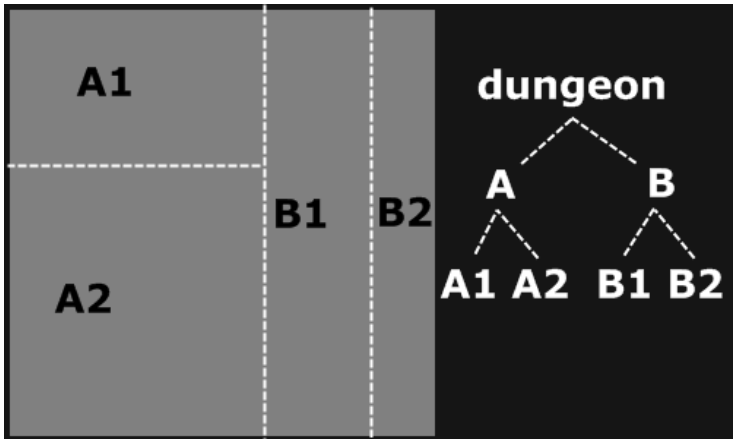


Figure 13 BSP for Dungeon Generation iteration two (Unknown 2017)

The algorithm then continues to divide the sub-cells further until optimum room size is achieved for all cells.

Once applied the divided space can be filled with rooms of a random size, in this example the BSP tree consists of four layers (iterations)

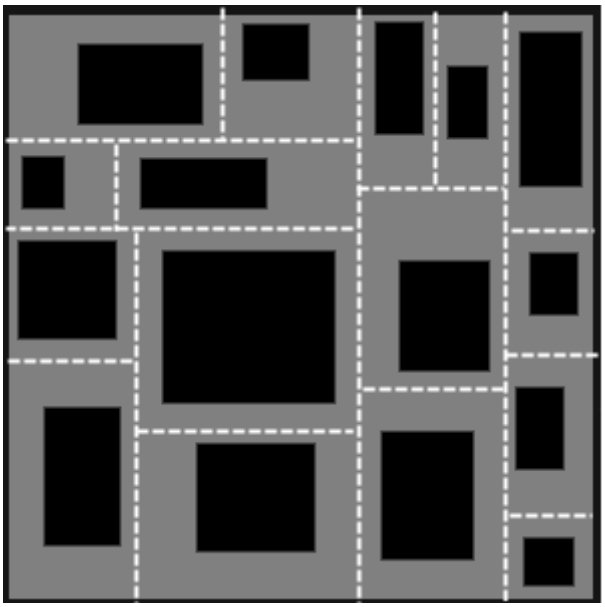


Figure 14 BSP Tree for Dungeon Generation rooms within cells (Unknown 2017)

The generated rooms are then connected in a reverse order starting from the fourth layer.

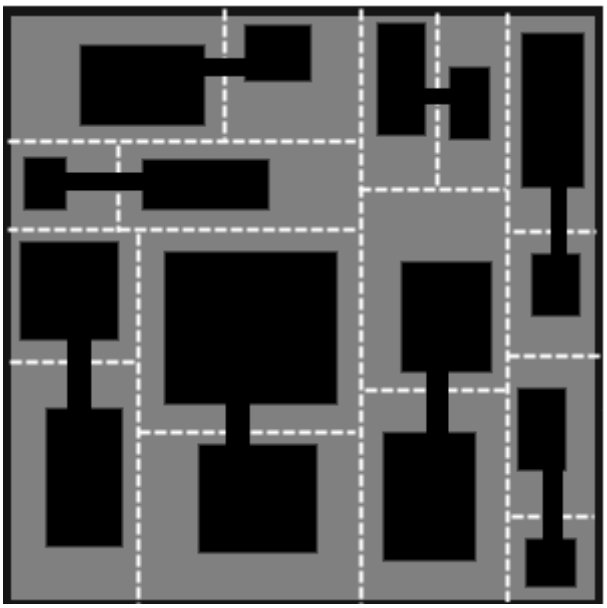


Figure 15 BSP Tree for Dungeon Generation layer four connections (Unknown 2017)

The algorithm can then continue reverse-recursively connecting layer three and two until the first two sub-dungeons (A & B) are connected, as seen in figure 13.

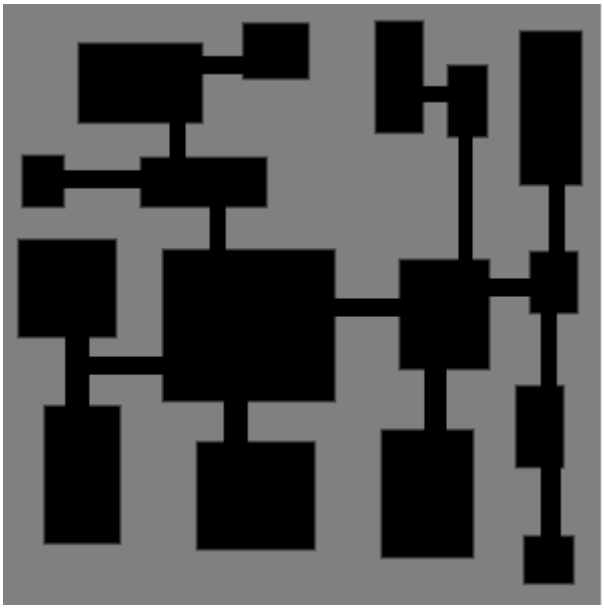


Figure 16 BSP Tree for Dungeon Generation connected layer one (Unknown 2017)

2.2.4 Constraint Solving Techniques

Constraint solving techniques (CSTs) are collation of methods for searching some or all potential solutions to find a match that satisfies all of the defined constraints. CTSs can be applied to finite domains spaces and infinite domain spaces, finite domains consist of a space in which all solutions can be evaluated whereas infinite domains contain variables that have potentially infinite values. Figure 15 lists the expanded tree hierarchy of CSTs.

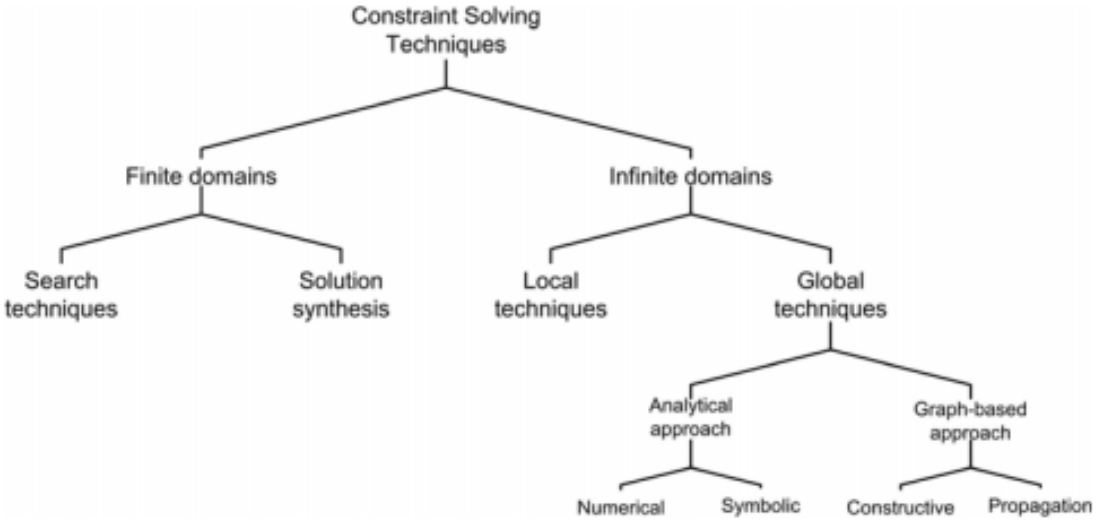


Figure 17 Constraint Solving Schematic (Tim Tutenel 2008 P. 11)

2.2.5 Search-based Procedural Generation

The search-based approach to PCG utilises either an evolutionary algorithm or other stochastic search/optimisation algorithms to find generated content that meet the required parameters specified by the system (Julian Togelius 2016a). The basic methodology is based on the idea that a satisfactory solution exists within the space of possible solutions. The system will adjust certain parameters of a generated solution and iterate over it whilst retaining the adjusted solutions that are closer to matching the required parameters and discarding those that reduce the possibility of reaching an optimum solution until the desired outcome is generated. The required components of a system to implement a search-based approach include a search algorithm, a content representation method, and an evaluation function.

2.2.6 Evaluating Heuristic Search Techniques

The search techniques detailed in this section are comprised of some of the most common approaches taken when generating levels for games, the efficiency of these methods are dependent on their method of application as each method can be described without reference to a specific problem domain, however if implemented properly through the exploitation of domain specific knowledge these algorithms can prove extremely useful.

The generate-and-test algorithm is one of the most basic implementations of a search technique, it simply generates a possible solution and compares it against the target goal state, if a valid solution has been found it quits, otherwise it will iterate indefinitely. The issue with this method comes when searching a large problem space as it is more unlikely that an acceptable solution will be found.

The simple hill climbing algorithm is a variation of the generate-and-test method, it will search the problem space for a solution whilst utilising feedback from the test procedure to adapt the direction in which to move the search. This technique employs a function to estimate parameters of the solution and determine whether the result is closer to the goal state.

An example of a simple hill climbing algorithm is as follows:

1. Evaluate the solution state, if it is a goal state, return the solution and quit. Otherwise continue to use the solution state as the current state.
2. While a solution is not found or there are no remaining parameters to apply:
   A) Select a parameter to test and apply it to the current state
   B) Evaluate the new state
      (i) If it is a goal state return it and quit.
      (ii) If it is nearer to the goal state then the previous state, make it the current state
      (iii) If is worse than the previous state, discard it and repeat the loop.

(Elaine Rich 2009)

Here the use of an evaluation function inserts certain task specific knowledge into the control process which is the defining characteristic of a heuristic search method.

The next algorithm is an expansion of hill climbing, titled *'Steepest-Ascent Hill Climbing'* or gradient search, this method contrasts the simple hill climb by taking all moves from the current state into account, this amplifies the search time of the algorithm as there may be many potential moves, however usually it will require fewer total moves to reach the solution state, these are the key factors that need to be considered when looking to apply these methods to a certain problem space.

Simulated Annealing is another form of heuristic search that adapts the core method of hill climbing however the moves from the initial state are made without regard to the effectiveness of the solution as the idea is to explore the space so much so that the final solution is abstracted from the initial state. This technique lowers the likelihood that the algorithm plateaus or gets trapped in a local maximum.

2.3 Analysis of Graph Grammar Techniques

2.3.1 Lindenmayer-Systems

Lindermayer-systems (L-Systems) are one form of transformational grammars in action. They generate a fractalesque sequence through the utilisation of a seed string and the application of a predefined set of transformational rules. An example of this is adapted from Bucklew (2017, P.283) :

Input string: 'A'

Rule 1: (A -> AB)

Rule 2: (B ->A)

Initial Value: A

(Iteration 1: A becomes AB via rule 1)

Output: AB

(Iteration 2: A becomes AB via rule 1; B becomes A via rule 2.)

Output: ABA

(Iteration 3: A become AB via rule 1; B becomes A via rule 2; A becomes AB via rule 1.)

Output: ABAAB

The results of such an L-system algorithm applied to the procedural generation of fauna can be seen in figure 14.

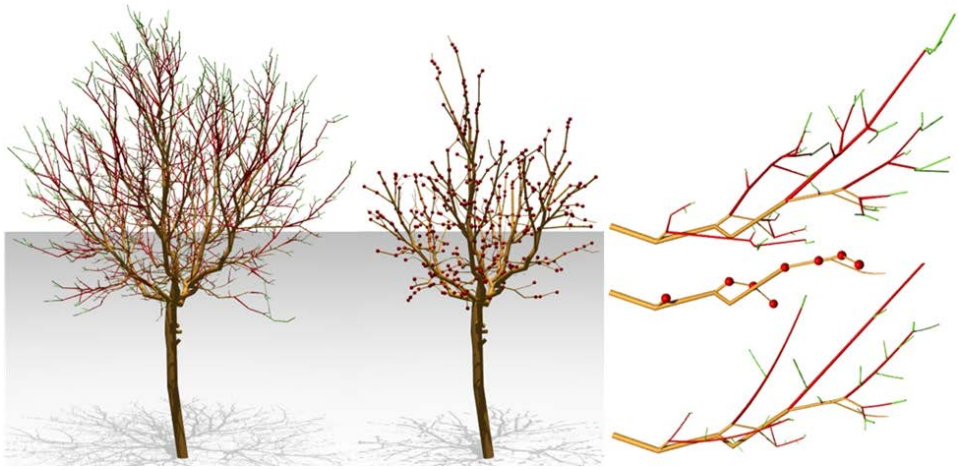


Figure 18 L-System Generated Fauna (Frédéric Boudon 2012, P.12)

2.3.2 Grammars in Games

Generative grammars are an efficient and useful way to represent both the game missions and game spaces that are combined to create game levels. For exploratory RPG and rouge-like games consisting of dungeons that have missions containing lock and key puzzles, treasure objectives and splitting, interconnected paths a graph grammar can easily express the structure, content and variables contained within the level in a very readable and sequential manner (See figure 15). (Julian Togelius 2016b P.80)

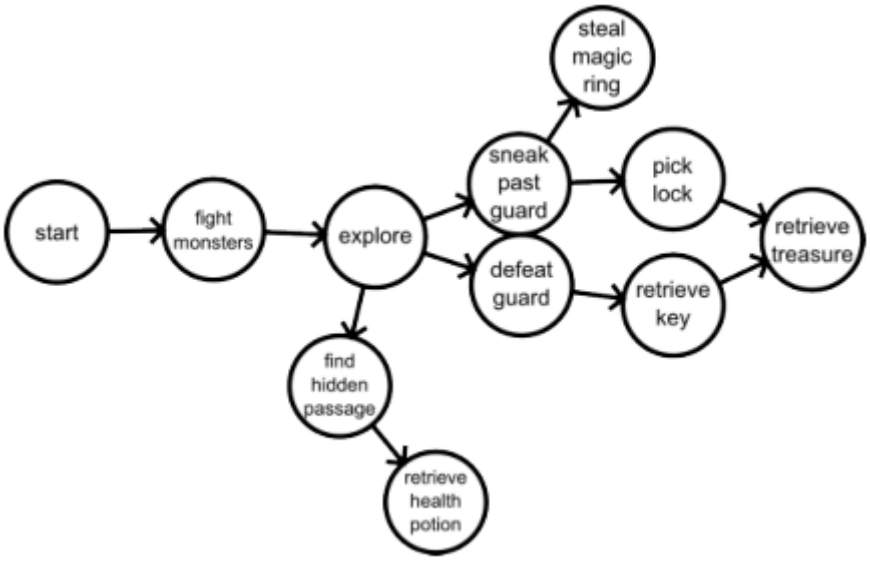
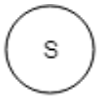


Figure 19 An Example Level Graph Grammar (Julian Togelius 2016b)

Generating a mission with graph grammars first requires the '*alphabet'* or rule set that the grammar is designed to work with. The following is an example of a rule set used to generate a dungeon level:



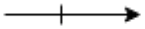
Start Node – The initial node the grammar generates from.

End Node – The goal node that completes the level.

Task Node – Random unspecified tasks node (monsters, treasure)

Lock Node – A node that requires a key to be unlocked.

Key Node – A node in which a key can be found.



Unlock Edge – An edge that specifies corresponding locks & keys

Regular Edge – An edge that defines the connecting nodes.

With this rule set the generation system can now construct rules to generate a mission.

Figure 16 defines an example set of mission rules to be used within a graph grammar level generation system. This rule set applies a *'wildcard'* node denoted with a '*' which indicates a match with any node, a wildcard appearing on the right hand-side of a rule does not change the corresponding node in the graph being transformed. (Julian Togelius 2016b P. 82)

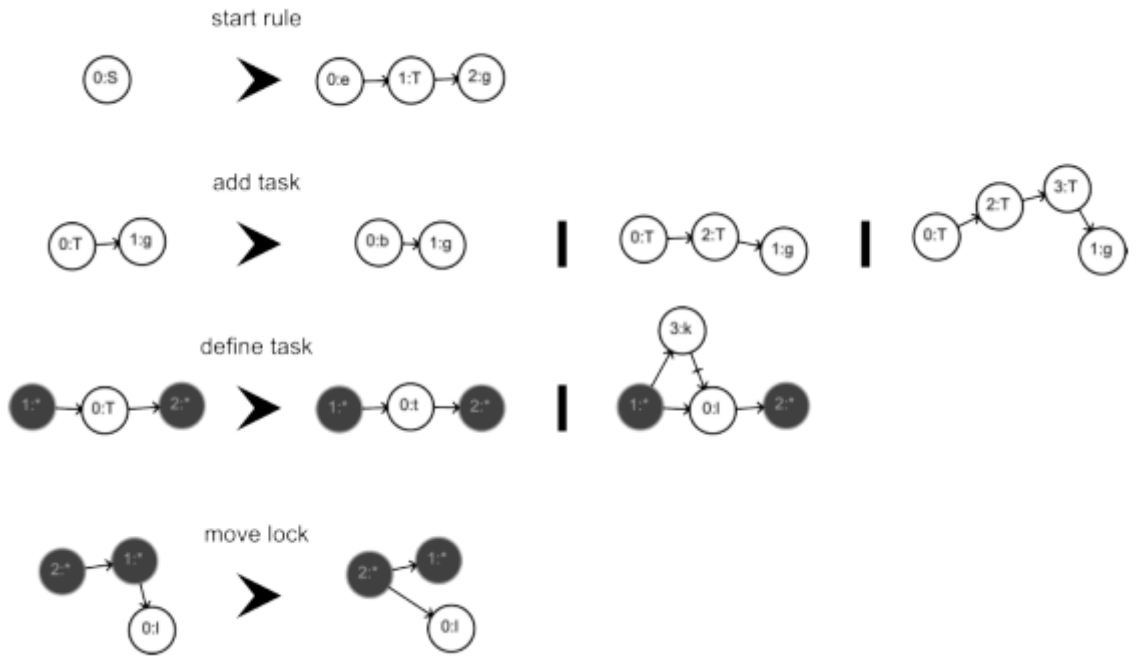


Figure 20 Mission Rules. (Julian Togelius 2016b P.83)

By partitioning the generation into multiple steps the evaluation of generated content becomes a much easier approach, having the ability to tweak a rule and regenerate a level to assess the outcome accelerates the workflow of designing a game, this methodology also falls in line with model-driven engineering practices and when properly implemented delivers a flexible generation process that permits generation of missions from spaces and inversely spaces from missions. (Dormans 2011)

When creating a level generation system having a form to represent the mission itself is only the first step, to create a level from a mission requires it to be transformed into a space that is traversable by the player. There are three main methods of transferring an abstract mission which details the objectives of the game into a tangible space in which the tasks can be performed, the first is a transformation from mission to space, the second is a creating a set of instructions that can be used to build the space, and the third is to build the level geometry to determine a better representation of the space in which to generate the mission. Section 2.3.4-6 details the steps required to perform these methods.

2.3.3 Generating Lock & Key Relationships

Locks and keys are a crucial element of rouge-like games, they do not have to conform to the generalised term of a physical lock and key but rather they can be concealed within the mechanics of the game itself, a good example of the application of integrating said locks and keys into the mechanics of the game is in *Unexplored:*

> "*For example a potion of resist fire can serve as a key to a barrier of lava the player needs to cross.* "
>
> (Dormans 2017 P. 91)

Another application of disguising keys as mechanics is integrating a key into a weapon, this way the key can be used at multiple instances and give the player a sensation of gear progression throughout the randomness.

The stage at which generation and integration of locks and keys is performed varies dependant on the chosen method of transferring from mission to space, if the first method is chosen the locks and keys are integrated within the generation stage, similarly the second method integrates the locks and keys within an instruction, whereas the third method first generates the space then populates the space with mission details.

Figure 17 is an example of lock and key rewrite rules described by Dormans (2011 P. 2) In this rule set the '?' nodes represent wildcards as described previously.

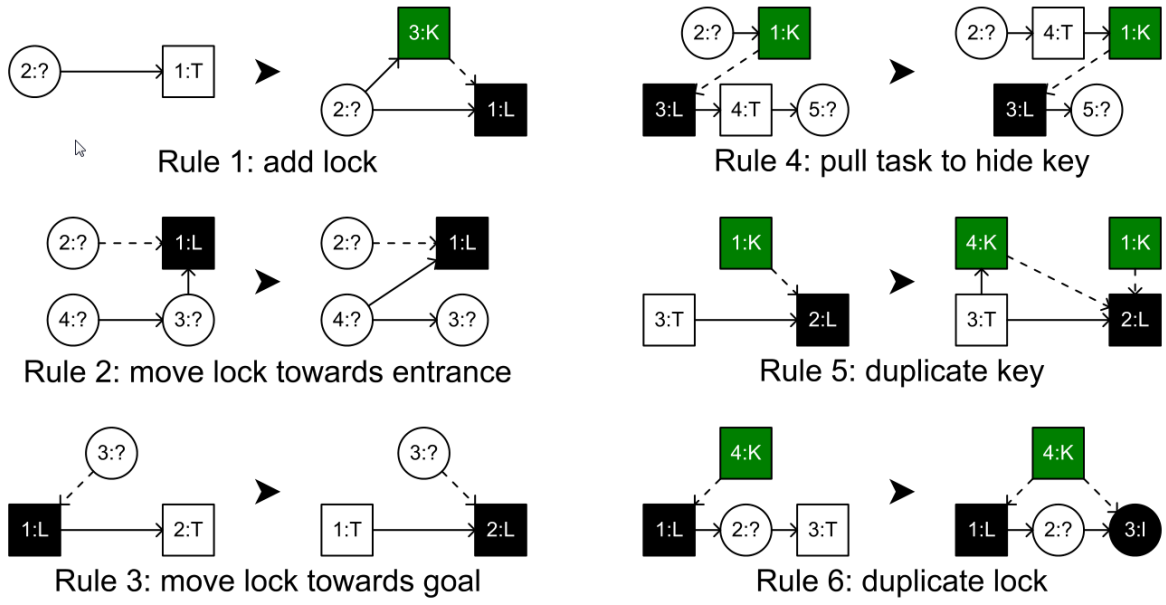


Figure 21 Lock & Key Rewrite Rules (Dormans, 2011)

Rule one can be transcribed as:

If x mission task is preceded by a task node, transform the task into a lock and add a key node linked to the replaced task node.

Rule two and three transform the position of the lock nodes towards the entrance and towards the goal respectively.

Rule four can be transcribed as:

If a task node is preceded by a lock node and a key node, and is followed by another mission task, transform the position of the task node to precede the key that unlocks the lock.

This rule ensures that the player must complete a task before discovering the key.

Rule five creates a duplicated key to allow for alternate solutions to the level.

Rule six creates multiple locks that can be unlocked with the same key, this refers to the earlier example of using a weapon as a key.

Having such a lock and key system allows the game designer to translate from a linear state of level progression into a branching structure, this elevates the level of interest of a mission and produces a higher degree of non-linearity. Figure 18 displays some example graphs generated with this rule set. (Dormans, 2011)

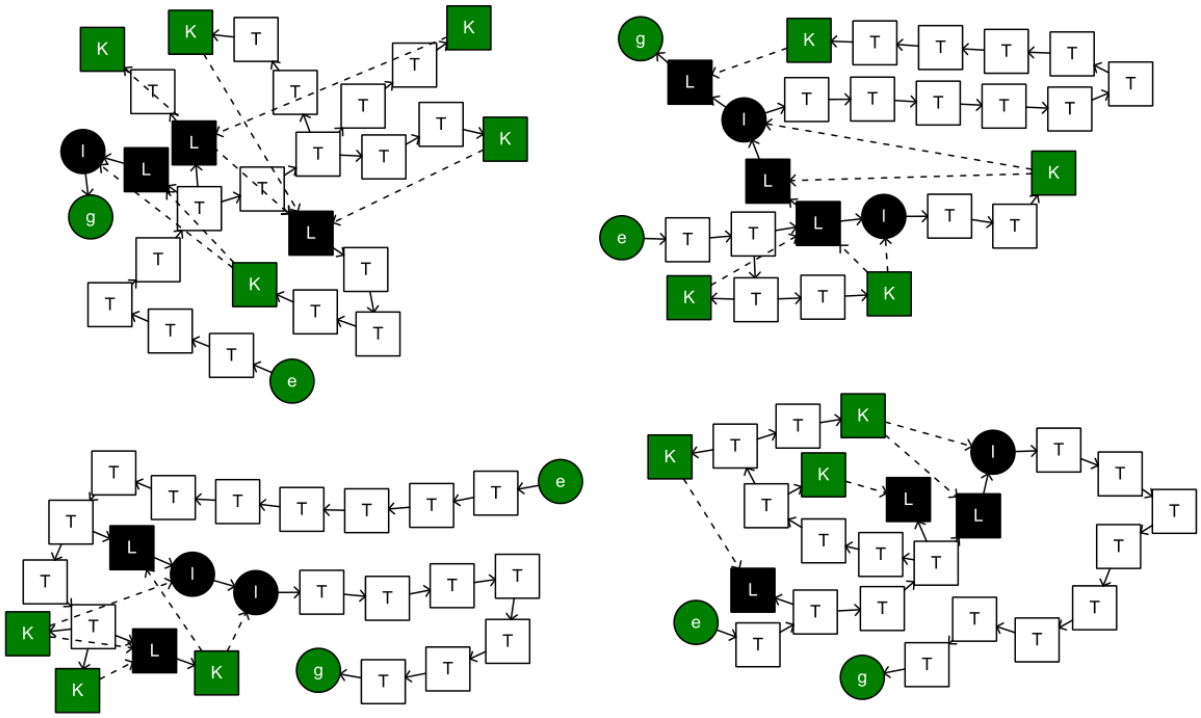


Figure 22 Generated Lock & Key Levels (Dormans, 2011, P.4)

2.3.4 Transferring from Mission to Space

Translating a generated mission graph (Figure 19) into a working level is accomplished by applying lock and key adaptation rules similar to those specified by Dormans (2011) in figure 17, this generates an output graph as seen in figure 20. Once the detail of the space has been derived the level can be further translated through the use of automated graph layout algorithms to position the nodes of the graph, the output can then be sampled into a tile-map and the full level layout can be constructed by applying a shape grammar to the generated tile-map. (Julian Togelius 2016b)

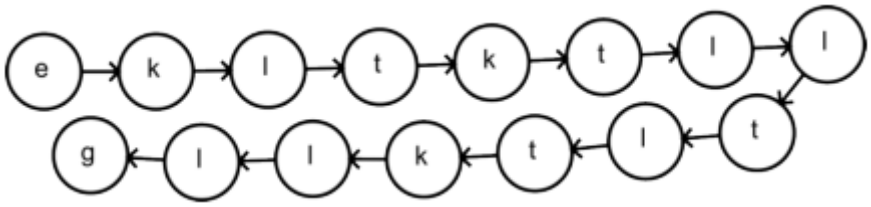


Figure 23 Sample Random Mission Set (Julian Togelius, 2016b P.86)

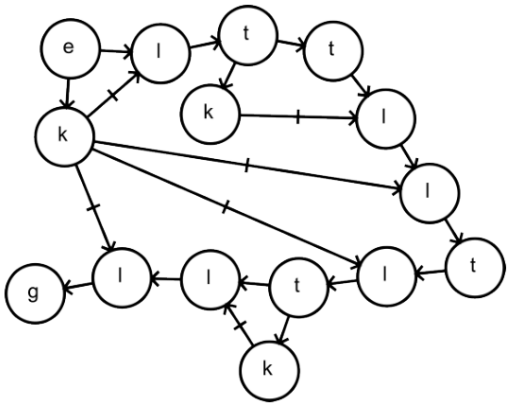


Figure 24 Generate Lock & Key Relationships (Julian Togelius, 2016b P.88)

2.3.5 Creating an Instruction Set to Build a Space

This approach consists of transferring the mission from its graph form into a set of instructions that are then used to build the space that meets the predefined requirements. An example instruction set is as follows:

Start Rule (x1)

Add Task (x12)

Add Boss (x1)

Define Tasks (x12) (Treasure | locks & keys | traps | enemies | etc.)

This method has the benefits that transformation from this form into a tile-map or shape grammar is highly simplified, however the results of this implementation are less likely to have multiple paths leading to the same goal or locations, the applications this method is useful in are linear style games such as platformers and specific types of story-driven games.

2.3.6 Creating Topologies to Generate Missions

By first defining the levels space before the construction of the mission the designer can integrate specific rules that adapt the structure to suit the specification of a theme for individual level types. This is a useful technique if the game requires a consistent type of architecture as using the same input parameters the generator will produce a very similar level layout. The disadvantages of this method are that ultimately the system has control over the mission potential and if not implemented in the correct manner it can lead to insufficient level properties e.g. lack of task space and insufficient amounts of doors to be locked alongside irreverence for core game design layout methodologies.

2.3.7 Shape Grammars

Shape grammars were first introduced in the 1970's by George Stiny (1972), the use of grammars in this form follow similar practices to that of other generative grammars, a set of rewrite rules are used to manipulate and transform existing shapes. Shape grammars for use in procedural modelling computer generated architecture was introduced by Müller (2006) for the use of building high quality visual renders of building shells in great geometric detail. This technique employs context sensitive shape rules to manipulate the interactions of hierarchical shape descriptions and the content that is in generation.

Shape grammars can be utilised within game generation to form the physical translation of a graph grammar into a fully detailed game level representation as seen in figure 21 (Middag 2016).

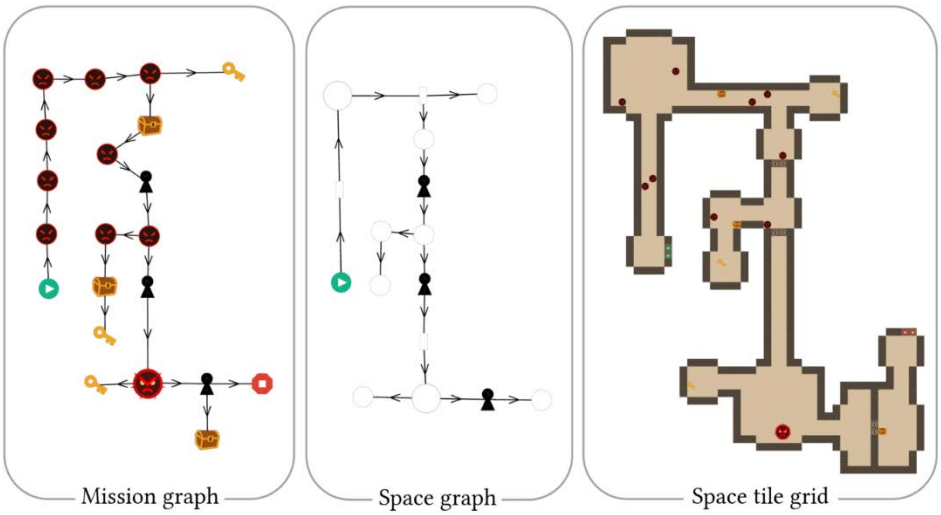


Figure 25 Step by step transformation from mission, to space, to tile grid (Bart Middag, 2016)

2.3.8 Generating Space with Shape Grammars

When generating the space of a mission the system requires a list of all terminal symbols that will be in the mission grammar, these can then be mapped to each rule in the shape grammar and subsequently selected for production based on their designated probability.

Dormans (2010) implements a shape grammar algorithm that encompasses a variance in the difficulty of a generated level through the use of dynamic parameters that can select rules with specific qualities based on the intended degree of difficulty for that segment of the level. This adaptation implements the use of *'registers'* that change the probability of applying rules with increasingly difficult tasks, it can also change from using a register that induces a large amount of traps to be generated to a register that includes a large number of monsters.

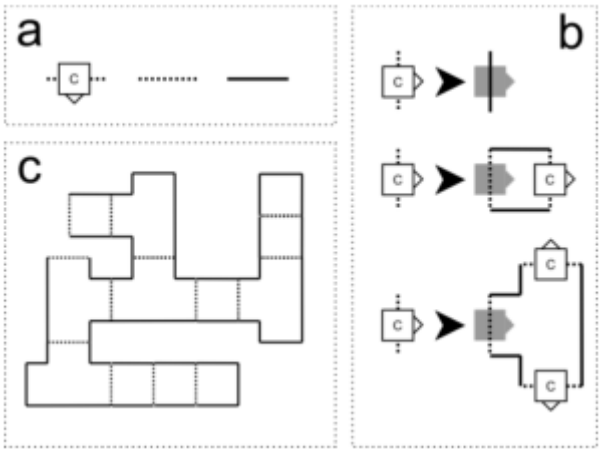


Figure 26 Shape grammar, alphabet, rules and output (Dormans 2010)

2.3.9 Algebraic Theory of Graph Rewriting Systems for the Single Pushout Approach

The algebraic approach to graph grammars as described by Joost Engelfriet (1997) is defined as a double-pushout approach (DPO-approach) as it relies on two pushout constructions whereas direct derivations of a graph grammar are defined as a single-pushout approach (SPO-approach), this approach applies a '*partial graph morphism'*.

As described by H. Ehrig (1997) a partial graph morphism is defined as seen in figure 23 where G is a graph and G_V, G_E denotes the sets of vertices and edges the graph contains,s^G, t^G denote the source and target mappings for each edge and lv^G, le^G denote the labels of the respective vertices and edges.

$$G = \left(G_v, G_E, s^G, t^G lv^G, le^G\right)$$

Figure 27 Algebraic Definition of a Graph (See appendices)

The following algorithm definitions are adapted from H. Ehrig (1997 P. 4-8).

A sub-graph S of G contains $S_V \subseteq G_V,\ S_E \subseteq G_E, s^s = G^s|s_E, t^s = t^G|s_E, lv^s = le^G|\ s_E$ and $le^s = le^G|s_E$

A partial graph morphism (g) from a graph (G) to (H) denotes a full graph morphism from a sub-graph $(dom(g))$ of (G) to(H).

A production that is applied to a graph consists of the left-hand side and right-hand side and a partial morphism between them. $(\ p : L \to R)$.

A graph grammar is defined as such $\acute{G} = (p : r)_{p\subseteq P}, G_0$ where $(p : r)_{p\subseteq P}$ is a set of production rules and G_0 is the graphs axiom node.

The left-hand side of the production defines the vertices and edges that an algorithm searches for in the existing graph and the right hand side of the production contains the sub-graph that will replace the left-hand side of the production.

An example of a production (p) in a formal notation is:

$$L = (\{\circ_1\}, s^L, t^L, lv^L, le^L) \to R(\{\circ_2, \circ_3, \}, s^R, t^R, lv^R, le^R)$$

Figure 28 Example Production (p)

Through the use of a partial morphism as opposed to a total morphism the pushout encompasses both deletion and reconstruction, the existing nodes are removed from the graph whilst their source and target node references are retained to ensure correct reconnection.

A match for production rule $(\ r : L \to R\ )$ in a graph (G) is a total morphism $(\ m :\ L \to G\ )$. The direct derivation of G with r at m is defined as $G \Rightarrow^{r,m} H$ and is the pushout of r and m in $Graph^P$.

A set of sequential direct derivations of graph G in the notation of

$P = G_0 \Rightarrow^{r_1,m_1} \dots \Rightarrow^{r_k,m_k} G_k$ consists of all derivations required to get from $G_0 \Rightarrow G_k$.

The '*graph language'* that is generated by the graph grammar $\acute{G}$ is the collation of all graphs from G_0 to G_k by $r_1, \dots, r_k$ which infers there is a derivation of $G_0 \Rightarrow^* G_k$ using the productions of $\acute{G}$.

The SPO-approach delivers the most natural view of a transformational graph system representing the system state as a whole and productions as a form of corresponding state transformations that are generated as direct graph derivations. This approach also handles one of the large problems that arise with the DPO-approach which is that of dangling edges, it solves this issue simply through the deletion of said edges.

3.0 Methodology & Implementation

3.1 System Design

The developed engine entitled 'DunJenny' has been developed using Visual Studio 15 (2017) and programmed in C++ (2017), with the assistance of third party libraries as detailed in section 3.2.

3.1.1 Graph Grammar System

The graph grammar backend system design consists of the abstract layer of a game, it does not specify anything specific about the mechanics of the world, only the information about the location and type of the nodes and edges that are used to build the graph, alongside the rules that may be utilised to manipulate the generation strategy.

3.1.2 Nodes

The node class design is a simple construct that consists of a unique node identifier, a label to be used as a name, a second label element for defining the node type and the position variables for the location of the node.

3.1.3 Edges

Similarly the edge class was designed with simplicity in mind, it consists of a source and target node and an edge type identifier. The edge type may refer to uni/bi-directional nodes, or be used to define the theme of an edge.

3.1.4 Graph

The graph design is of a similar construct to that discussed in 2.3.8, consisting of a set of nodes and edges, a graph name, a list of rules that have been applied to the current graph, and a set of distance integers to be used in the evaluation function.

3.1.5 Rule

Each rule has been designed to be built with a set of *'components'*, the component object is structured to contain a set of nodes and edges, this way the rule itself can be comprised of left-side components and right-side components, alongside a rule identifier. The functions associated with the rule class design contain getters and setters for all of the components within the rule and an update function that updates its left & right side components when new unique identifiers are generated for the right side of the rule.

3.1.6 Rule Factory

The rule factory has been designed to create new rules in a simple and effective manner, it has its own set of left and right side components, a container for all the created rules and getter and setter functions for all of its variables. The update function of the rule factory takes in two rules, the first is the rule that is being updated and the second has been used in the generation of new unique ids upon production creation, this ensures no overlap in the rule ids when inserting a production in a graph.

3.1.7 Graph Builder

The graph builder is designed to be the entry point for the generation strategy, its content consists of the graph being generated and iterated over, the rule factory that is being employed, a list of graph updates that have been applied, a list of the rules that have been applied and getters and setters for all the individual components of the class.

3.1.8 Generation Strategy

The design of the generation strategy was based on an adaptation of the hill climbing algorithm as seen in pseudo code from Adams (2002 P.29-30). This strategy relies on the testing of certain parameters and adjusting the search space until a valid solution is found, as previously described in section 2.2.7.

3.2 Implementation

3.2.1 ImGui

Omar Cornut's ImGui (2014) was chosen as the graphical user-interface (GUI) after being inspired by Casey Muratori's (2005) video presentation that describes how an immediate mode GUI (IMGUI) has a far more streamlined approach to user interaction and worked in a similar fashion to retained-mode graphics API's. The main benefits of the IMGUI is that there is no need to store widgets or graphics and that it runs in an immediate fashion which allows for easy editing of GUI styling and application.

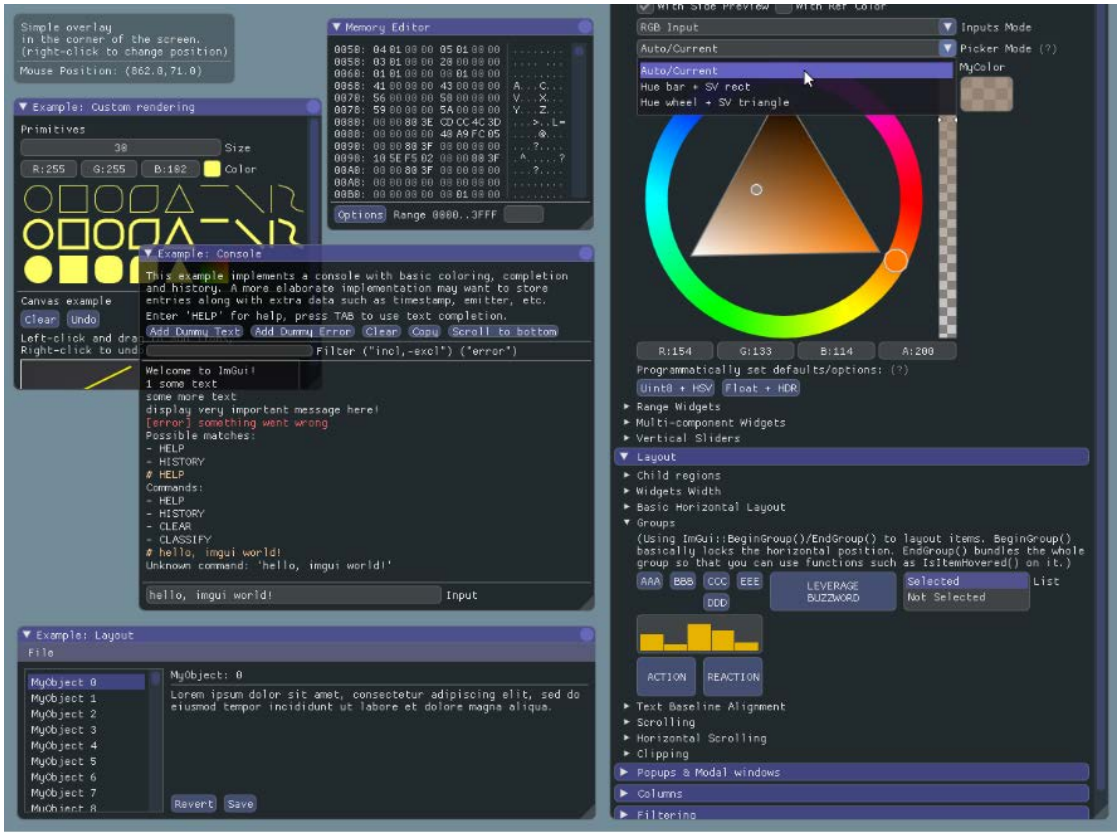


Figure 29 Example of ImGui Environment (Cornut 2014)

3.2.2 Node Editor

The node editor system that has been integrated into this project is an implementation that utilises ImGui (2014) as a GUI and serves as a basis for more complex solutions to be built from, this adaption was created by Michał Cichoń. (2016)

The node editor integrates Kazuho Oku's 'picojson' (2009) to load and save the node editor settings, it is a header-only file and relies only on standard C++ libraries.

This node editor was chosen to be integrated on the basis that it has already predefined the node and edge drawing system, along with the implementation of ImGui (ibid) to allow for extremely fast iteration of the GUI. The application entry point is an adaptation of *'imgui_impl_dx11'* from ImGui (ibid) which implements a WinMain entry point.

3.2.3 Random Generator

Throughout the implementation there have been numerous uses of randomly generated numbers, the 'RandomGenerator' class that has been implemented is inspired by Zadravec (2017) and adapts his method to accommodate both the uniform distribution and Gaussian distribution functions. The Gaussian distribution function was decided to be used for the placement of the level nodes as opposed to uniform distribution due to the clear overlap that is produced (Figure 31). The Gaussian distribution combats this by having a likelihood to generate more numbers around the edge of the distribution curve (Figure 7).

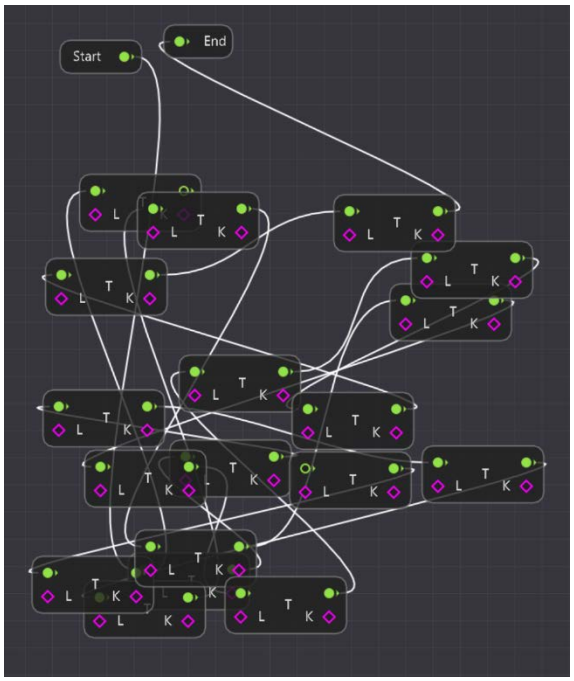


Figure 31 Generated Node Positions (Uniform Distribution)

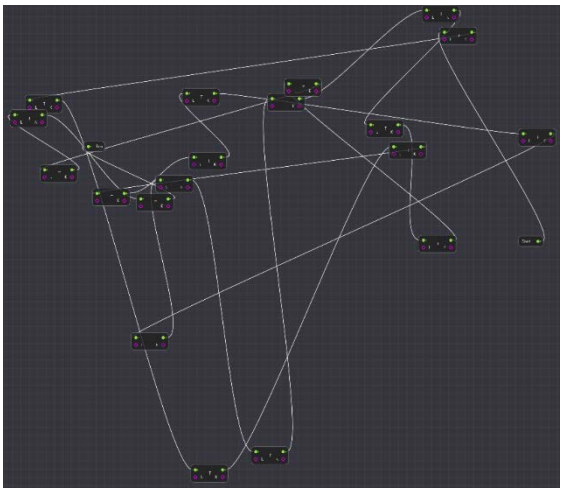


Figure 30 Generated Node Positions (Gaussian Distribution)

3.2.4 Graph Builder

The implementation of the Graph Builder contains the entry point for graph derivation, this occurs within the '*onInit'* function which takes in a ruleset and a graph to perform the generation on. It does this by applying the *'deriveGraph'* function on the instantiated strategy and assigns the successfully derived graph to the input graph then returns the new graph back to the main threads *'GenerateMap'* function within *'DunJunny.cpp'*. Also provided are a few predefined test rules for the user to generate an example graph from, these are created with the *'testRules'* function.

```
//Entry point for graph derivation from DunJenny.cpp
graphSys::Graph GraphBuilder::onInit(std::vector<graphSys::Rule> existingRules, graphSys::Graph G)
{
    //Load test rules
    if (firstLoad == true)
    {
        testRules();
        firstLoad = false;
    }
    //Instantiate generation strategy
    graphSys::GenerationStrategy strat(rf, G, G.getIds());

    if (G.getGraphNodes().size() > 0)
    {
        G = strat.deriveGraph(G);

        //Store graph derivations to see rule progression
        graphUpdates.push_back(G);
        G.completed = true;
        return G;
    }
    else
        return G; //deriveGraph sets graph name to "FAIL" if no solution found
}
```

Figure 32 Code Snippet of GraphBuilder.cpp's onInit function

If the generation algorithm exceeds the number of maximum iterations defined by the user then it will exit with a fail and return the error message seen in figure 33.

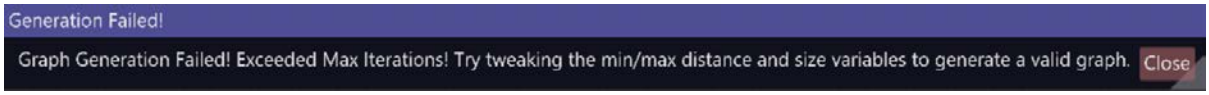


Figure 33 Graph Generation Fail Error

Figure 32 displays a snippet of the deriveGraph's implementation of the hill climbing algorithm, this snippet is from within the while loop that ensure the generator is within the maximum iteration limit. The first if statement contains four attribute checks to test if the generated graph is a possible solution, the first two are checking if the graph size is within the set minimum and maximum and the second two are checking the bounds of the minimum and maximum distances, if these all return true then the main graph is replaced with this solution, otherwise the first else if statement will test if the solution is closer to the desired state and incur a repeat of the loop, otherwise the currently tested rule is removed from the list of potential applicable rules.

```
if (graphCopySize > targetSizeMin + RG.GenerateUniform(0, targetSizeMax - targetSizeMin)
 && graphCopySize < targetSizeMax &&
    currentMaxDist.first < targetXDistMax && currentMaxDist.second < targetYDistMax &&
    currentMaxDist.first > targetXDistMin && currentMaxDist.second > targetYDistMin)
{
    G = G_Copy;
    G.addRuleApplied(rule.getID());
    result = 1;
}
else if (graphCopySize > graphSize)
{
    G = G_Copy;
    G.addRuleApplied(rule.getID());
    rules.erase(rules.begin() + randN);
    rules.push_back(G.getUpdatedRule());
}
else
{
    G_Copy.clearGraph();
    if(rules.size() > 0)
        rules.erase(rules.begin() + randN);
}
G.iteration++;
```

Figure 34 Derive Graph Hill Climb Snippet

3.2.5 Rule Builder

DunJenny creates new rules through the use of the IMGUI, within *'DunJenny.cpp'* the pop-up & modal pop-up system is utilised to allow a user to create new rules.

Instructions are provide on how to build rules and conditions that must be met to generate a rule, figure 36 shows an example of a user failing to meet said conditions.

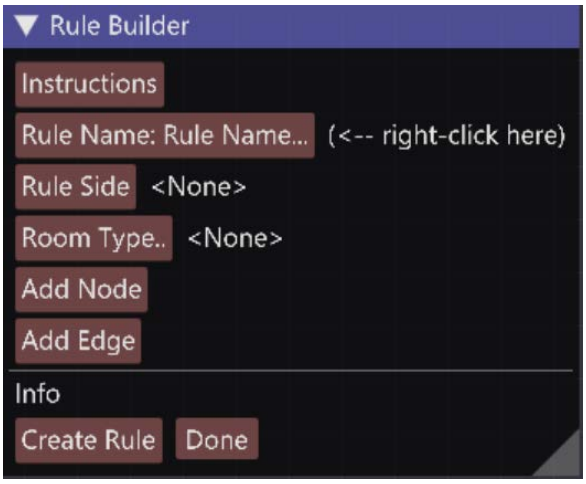


Figure 35 Rule Builder GUI

Figure 36 Error message for invalid rule

The contents of the instructions modal pop-up provided are as seen in figure 37.

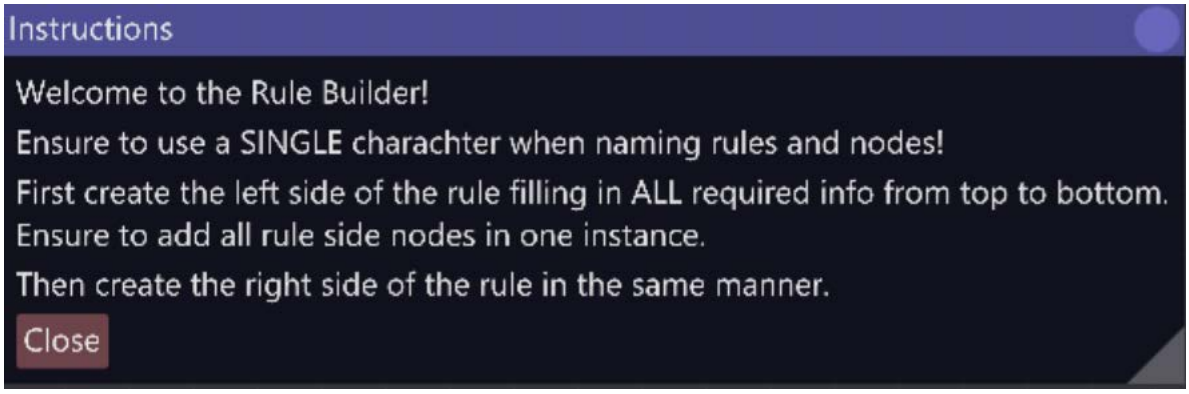


Figure 37 Rule Builder Instructions

Once the name of the rule has been set and the rule side selected the user can add all nodes that reside in that side of the rule using the Node Builder as seen in figure 37.

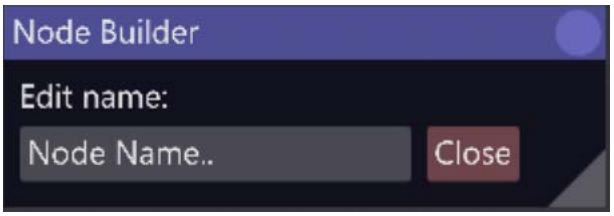


Figure 38 Node Builder

Following all node additions the edges can then be constructed using the Edge Builder by setting the source and target node for each specified edge and clicking the 'A*dd Edge'* button.

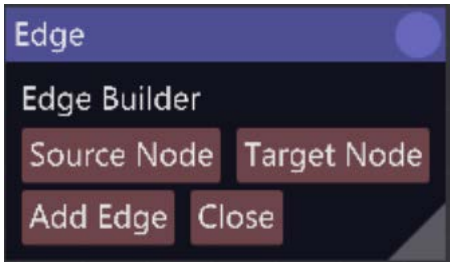


Figure 41 Edge Builder

Figure 40
Source Selector

Figure 39
Target Selector

3.2.6 Constraint Implementation

The *'deriveGraph'* function of the generation strategy class implements hill climbing for graph generation based on size & distance parameters that are able to be set from within the left hand pane of the window (Figure 41). The editable variables are the maximum number of iterations the algorithm will run for before returning a fail, the target graphs minimum and maximum size which is calculated as the number of nodes in the graph and the target graphs minimum and maximum x & y distances which is calculated as the largest difference in space between two nodes.

The distance calculation function resides within *'graph.cpp'* and is entitled *'calcDistances'*. The function iterates through all nodes in the graph and compares the distance between each node, once it has found the maximum distance it returns a pair that contains distance of the node with the furthest x and y distance.

The placement of generated nodes also relies on the Gaussian random number generator, it will generate a position between 0 and the target graphs minimum size multiplied by fifty to accommodate the varying target sizes.

To generate a graph from the start screen the user must first inset a room node into the graph window via the right click mechanic seen in figure 48, this is the axiom node for the algorithm to utilise within the algorithm.

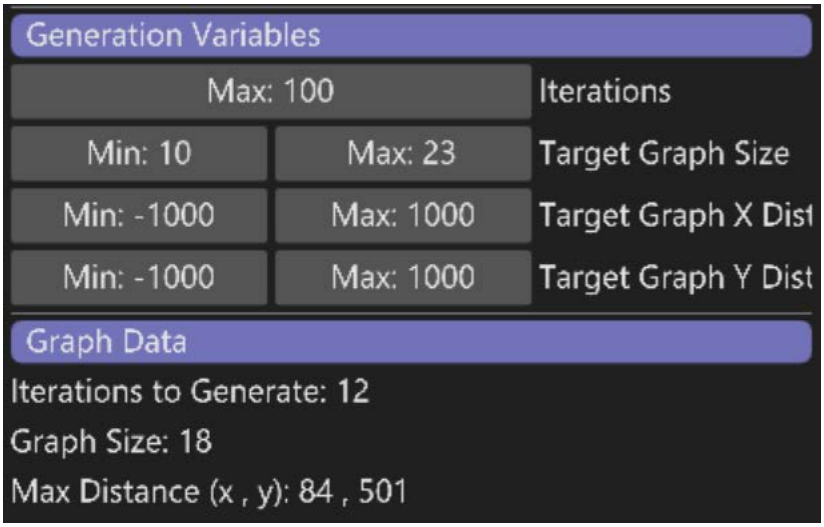


Figure 42 Generation variables & relating graph data

3.2.7 Node and Edge Implementation

The graph task node has been adapted from Cichoń's (2016) *'SpawnLessNode'* to include one room input node, one room output node and a set of lock and key nodes, the SpawnLessNode was also the base for adapting the start and end nodes, where they have simply been modified to a single output and input respectively.

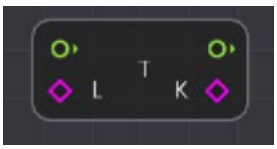


Figure 44 Task node

Figure 43 Start & End nodes

To create new nodes the user can right click anywhere within the graph map and select the type of node to spawn (Figure 44), alternatively to spawn a room that instantly connects to its source node the user can select the node pin and drag it to the desired spawn location of the map (Figure 45). To link two nodes with an edge, the source nodes pin must be selected and dragged to the target nodes pin (Figure 45).

Figure 47 Node Options

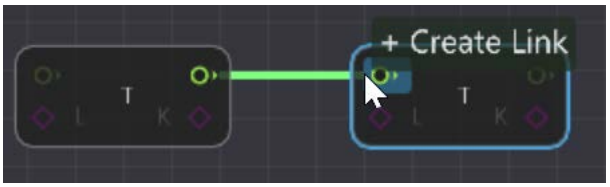


Figure 46 Link Two Nodes

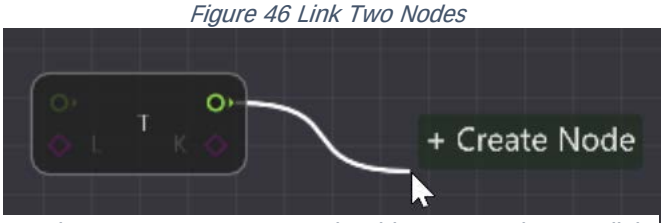


Figure 45 Create a new node with automated source link

3.2.8 Key & Lock Implementation

The lock and key mechanism has been developed into the post-generation stage of the system, it allows the designer to specify the location of as many locks and keys as they wish by simply creating the edge with the source and target pin connections.

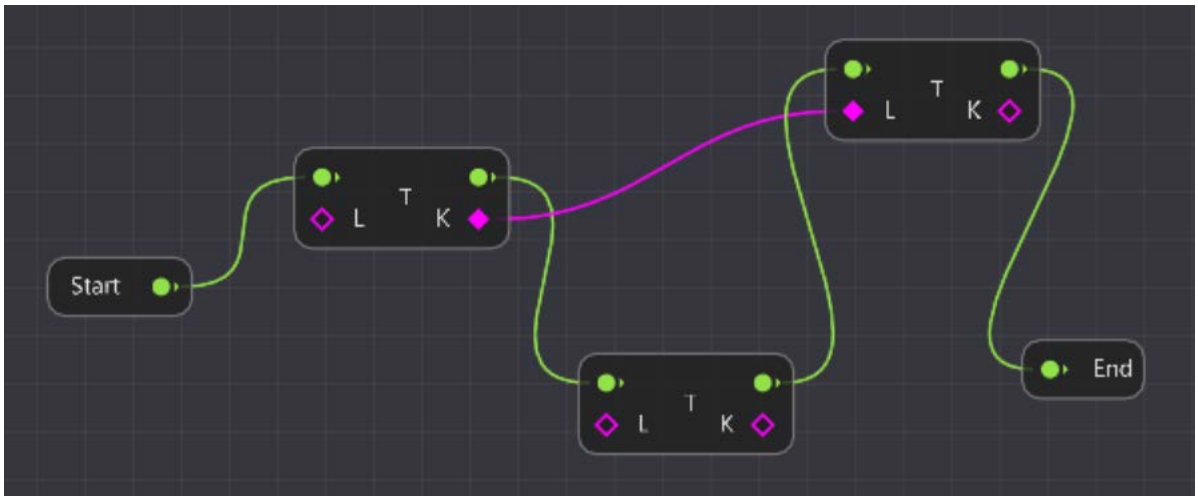


Figure 48 Example Lock and Key Graph

3.2.9 MetaStuff Integration

The json serialization/deserialization library entitled '*MetaStuff* by Daler (2017) was successfully integrated with the intention of loading and saving graphs and rule sets through the Menu, unfortunately a solution was not found to the problem seen in figures 49 & 50, the system was unable to be used to create a member for the custom types which is integral in the successful parsing of json data and in returning an accurate dataset to rebuild the levels/rules from.

'*MetaTest'* fully integrates and parses json from '*Person.h'* as seen through the log example with the *'Show Rules Added'* button. (Figure 51)

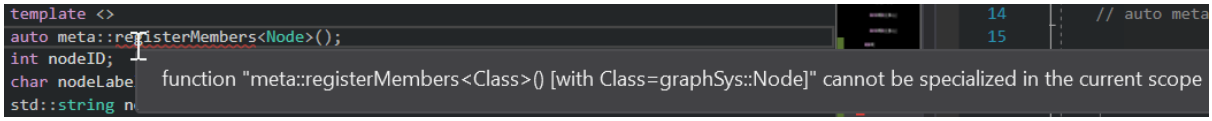


Figure 49 Failing to create custom Node member type

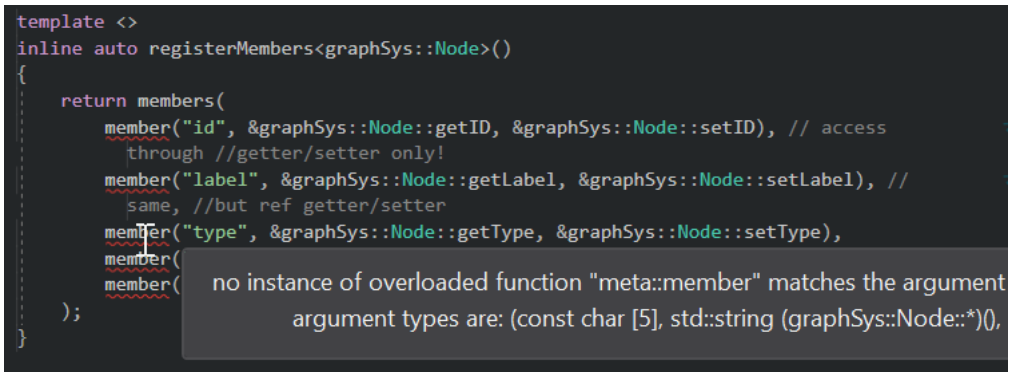


Figure 50 Unable to register members as no member class is registered for Node

Figure 51 Output from the 'Show Rules Added' button in the log

Another aspect that was unable to come to fruition was the in-app rule editor, designed to open when the user clicks the *'Available Rule'* that they wish to edit from the left-hand window pane. Due to the unsuccessful serialization a dummy example of what a rule would look like was implemented in place.

Figure 52 Dummy json File for RuleThree

3.2.10 Implementation Issues

Potential errors arise when creating rules with closed loop rules as the portion of the algorithm responsible for generating an updated version of the right hand side of a given rule will run into an issue when searching for source and target edge replacements.

Errors also arises when using rule and node names that are larger than one char, this arises from ImGui accepting the full name into a character array whereas in the comparison of the old rule name and the new rule name within the *'updateRule'* function of the *'ruleFactory'* class throws an error as it only contains a single char comparison.

Another issue that was encountered was the occasional unlinked single node, as seen in figure 53, throughout the testing process this phenomena would occur once every ~50 graph generations. Another example can be seen in figure 54 where the end node has no edge link.

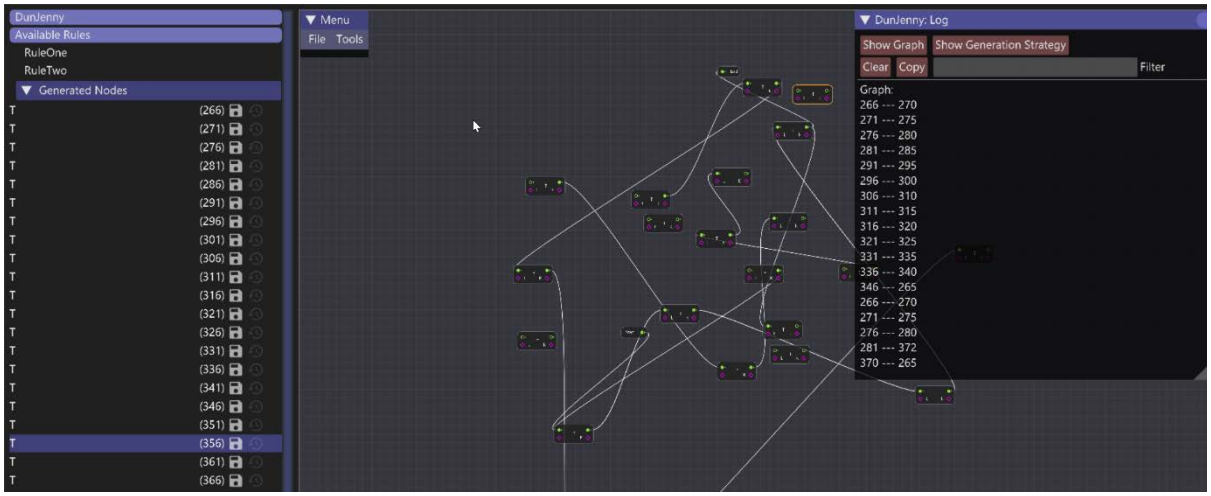


Figure 53 Single Unlinked Node

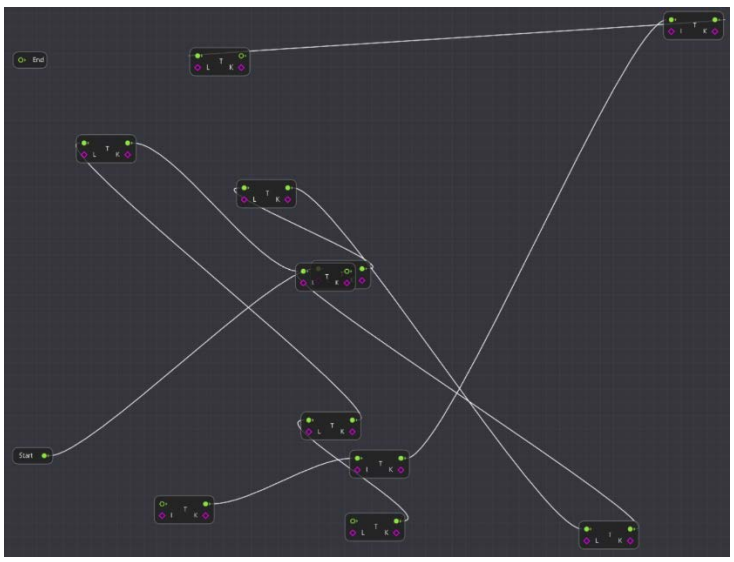


Figure 54 Single Unlinked Node (End)

4.0 Critical Reflection

4.1 Evaluation of Results

Below is an example graph and its matching production variables (Figure 55 & 56).

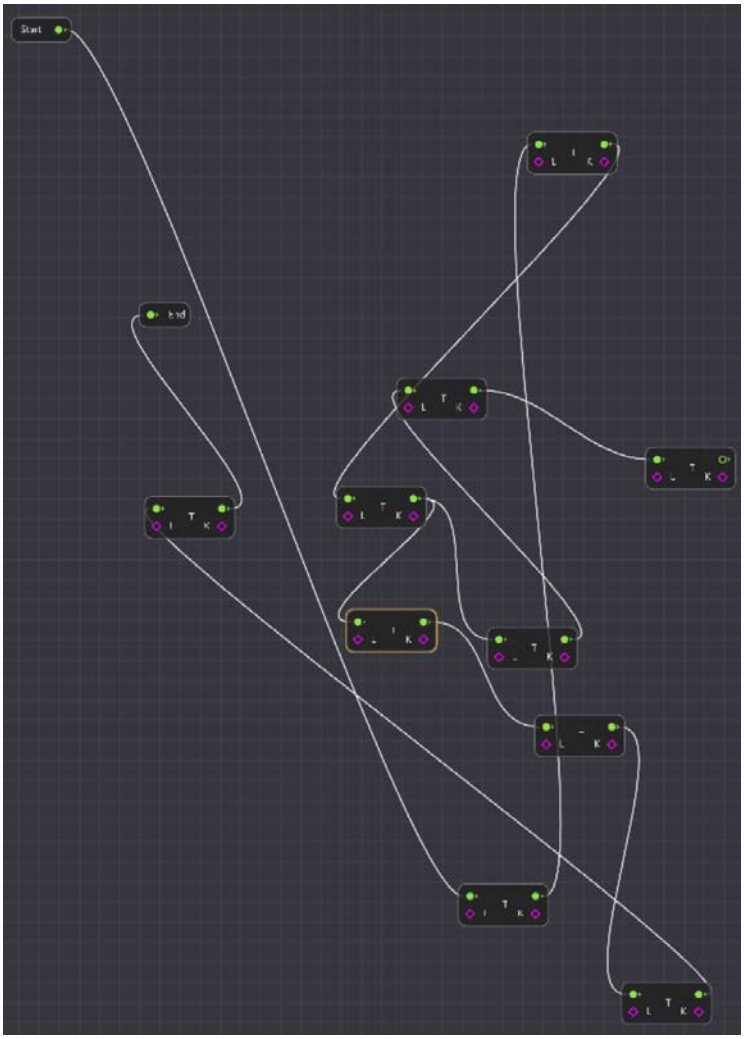


Figure 55 Example Graph

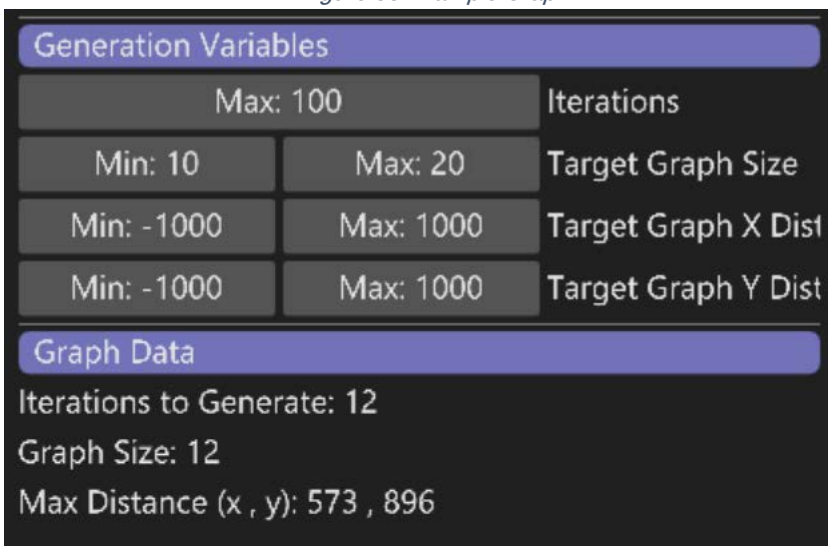


Figure 56 Example Graph's variables

Graph #	Rule Set	Time	Iters	Success	Size	Constraints
1	1,2,3	168	21	True	27	100 | 10/50 |
2	1,2,3	361	34	True	39	100 | 10/50 |
3	1,2,3	168	22	True	30	100 | 10/50 |
4	1,2,3	75	16	True	18	100 | 10/50 |
5	1,2,3	98	16	True	24	100 | 10/50 |
6	1,2,3	227	28	True	27	100 | 10/50 |
7	1,2,3	157	28	True	27	100 | 10/50 |
8	1,2,3	159	22	True	27	100 | 10/50 |
9	1,2,3	126	18	True	25	100 | 10/50 |
10	1,2,3	109	18	True	25	100 | 10/50 |
11	1,2,3	113	18	True	22	100 | 10/50 |
12	1,2,3	47	12	True	19	100 | 10/50 |
13	1,2,3	116	22	True	24	100 | 10/50 |
14	1,2,3	229	30	True	31	100 | 10/50 |
15	1,2,3	101	18	True	28	100 | 10/50 |
16	1,2,3	78	16	True	24	100 | 10/50 |
17	1,2,3	43	16	True	18	100 | 10/50 |
18	1,2,3	40	16	True	18	100 | 10/50 |
19	1,2,3	119	26	True	29	100 | 10/50 |
20	1,2,3	48	14	True	17	100 | 10/50 |
21	1,2,3	63	20	True	29	100 | 10/50 |
22	1,2,3	40	14	True	20	100 | 10/50 |
23	1,2,3	103	26	True	29	100 | 10/50 |
24	1,2,3	70	18	True	22	100 | 10/50 |
25	1,2,3	87	20	True	23	100 | 10/50 |

Figure 56 Test Results Target Size 10-50

The results set displayed in Figure 56 correlates to twenty-five separately generated graphs, alongside the rule set used, time taken to generate in milliseconds, the number of iterations the algorithm required to perform to reach the solution state, a success Boolean that details whether the algorithm managed to find an acceptable solution, the size of the generated graph and the constraints that have been applied in the form of (Number of iterations | Minimum graph size / Maximum graph size).

All twenty-five graphs returned a successful solution for a size range of 10-50, run on 100 iterations. The average time taken was 117.92 milliseconds, taking an average of 20.36 iterations to generate a valid solution and returning an average graph size of 24.88.

Graph #	Rule Set	Time	Iters	Success	Size	Constraints
1	1,2,3	364	38	True	42	100 | 20/50 |
2	1,2,3	303	34	True	36	100 | 20/50 |
3	1,2,3	343	34	True	39	100 | 20/50 |
4	1,2,3	278	26	True	41	100 | 20/50 |
5	1,2,3	4105	0	False	0	100 | 20/50 |
6	1,2,3	205	26	True	33	100 | 20/50 |
7	1,2,3	261	34	True	42	100 | 20/50 |
8	1,2,3	260	38	True	44	100 | 20/50 |
9	1,2,3	3356	0	False	0	100 | 20/50 |
10	1,2,3	104	27	True	27	100 | 20/50 |
11	1,2,3	235	32	True	41	100 | 20/50 |
12	1,2,3	275	42	True	37	100 | 20/50 |
13	1,2,3	46	16	True	27	100 | 20/50 |
14	1,2,3	3613	0	False	0	100 | 20/50 |
15	1,2,3	63	17	True	25	100 | 20/50 |
16	1,2,3	310	38	True	44	100 | 20/50 |
17	1,2,3	302	36	True	46	100 | 20/50 |
18	1,2,3	271	40	True	42	100 | 20/50 |
19	1,2,3	248	38	True	41	100 | 20/50 |
20	1,2,3	169	32	True	29	100 | 20/50 |
21	1,2,3	145	30	True	34	100 | 20/50 |
22	1,2,3	84	22	True	27	100 | 20/50 |
23	1,2,3	372	42	True	46	100 | 20/50 |
24	1,2,3	370	38	True	50	100 | 20/50 |
25	1,2,3	130	24	True	34	100 | 20/50 |

Figure 57 Test Results Target Graph Size 20 - 50

Looking at the same set of variables for another twenty-five graphs with an adaptation to the minimum constrain size to be twenty, there are some interesting results. For the twenty-five graphs that applied these constraints three of them returned as unable to find a solution. The average time taken for all successful solutions to find a suitable graph match is almost double at 205.56 milliseconds, the average number of iterations required to generate usable solutions spiked to 28.88 while the average size increased by 8.2, becoming 33.08.

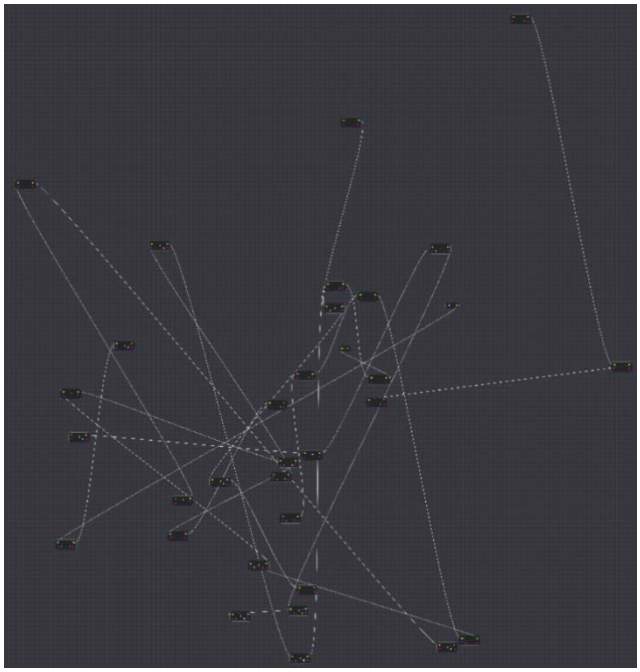


Figure 58 Example Graph from Size Range 20-50

Graph #	Rule Set	Time	Iters	Success	Size	Constraints
1	1,2,3	91	16	True	19	100 | 10/20
2	1,2,3	95	16	True	18	100 | 10/20
3	1,2,3	35	14	True	14	100 | 10/20
4	1,2,3	42	14	True	20	100 | 10/20
5	1,2,3	32	12	True	19	100 | 10/20
6	1,2,3	19	8	True	14	100 | 10/20
7	1,2,3	22	8	True	14	100 | 10/20
8	1,2,3	2896	0	False	0	100 | 10/20
9	1,2,3	7680	0	False	0	100 | 10/20
10	1,2,3	37	18	True	17	100 | 10/20
11	1,2,3	2179	0	False	0	100 | 10/20
12	1,2,3	37	14	True	15	100 | 10/20
13	1,2,3	43	14	True	20	100 | 10/20
14	1,2,3	41	14	True	20	100 | 10/20
15	1,2,3	3017	0	False	0	100 | 10/20
16	1,2,3	3083	0	False	0	100 | 10/20
17	1,2,3	13	6	True	14	100 | 10/20
18	1,2,3	3177	0	False	0	100 | 10/20
19	1,2,3	23	8	True	15	100 | 10/20
20	1,2,3	35	14	True	17	100 | 10/20
21	1,2,3	31	14	True	17	100 | 10/20
22	1,2,3	23	12	True	13	100 | 10/20
23	1,2,3	33	12	True	16	100 | 10/20
24	1,2,3	47	14	True	20	100 | 10/20
25	1,2,3	20	8	True	14	100 | 10/20

Figure 59 Result Set for Target Graph Size 10-20

The third set of data that was collated lowers the maximum graph size while retaining identical variables for all other options. The most obvious deduction from examining these results is that there are six graphs that did not return a suitable production graph, in comparison to the first set of results that is a 25% loss, however the average time taken to generate a graph is 9.44 seconds, better than double the initial set and of course as the maximum range is lower the average size has diminished to 12.64.

What's more interesting when it comes to this data set is the top down approach to evaluating its outcome, looking at Figure 60 this layout is a much more coherent space for the insertion of a mission in comparison to the example given in figure 58.

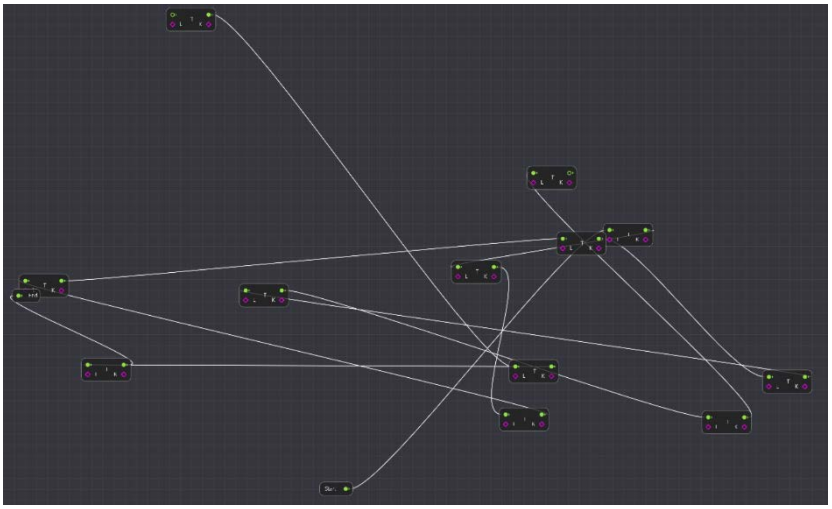


Figure 60 Example Graph from the 10-20 size range

When narrowing the size range there is a certain drop off point where the problem space becomes too narrow and the physical limitations of the space become an issue and overlapping occurs.

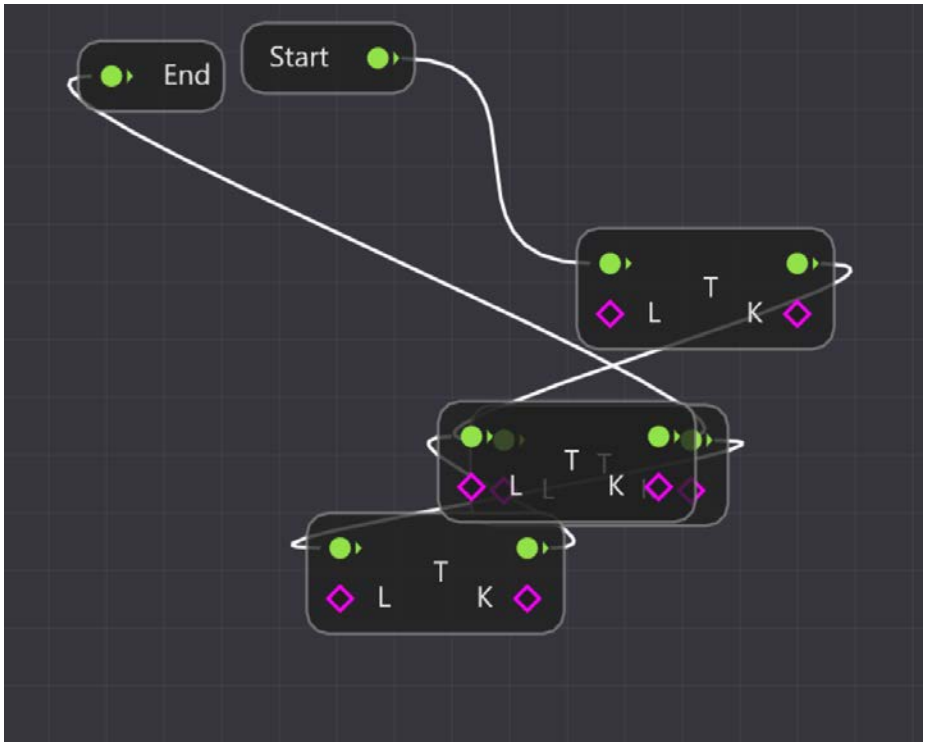


Figure 61 Example Graph size from the 1-10 size range

4.2 Critical Analysis & Reflection

The main challenges that this project presented were in the development of the hill climbing algorithm application within the graph derivation stage, due to the use of random node identifiers a large portion of time was spent tracing the steps of the algorithm on paper to identify why the solution was not producing the expected results. With the use of the IMGUI translating the rule, node and edge from the back-end graph grammar system was a relatively seamless transition from logic to interface. While the solution presented in this project does not generate entirely populated dungeon levels, it qualifies as a base structure for the generation of abstract game levels on which additional objectives can be applied.

Had the integration of locks and keys been enveloped into the back end of the graph grammar systems algorithm design stage then the final implementation would align more with the initial goals of the project.

4.3 Conclusions

The produced product is capable of creating a range of spaces that are suitable for use in creating a dungeon level, or for the conversion and production of a written mission utilising the generalised purpose of the system, and provides a method of doing so through the designer-assisted lock & key system. Through tweaking the supplied variables a user can generate a range of graph styles which can then be further adapted or expanded based on the required needs or theme. Overall the outcome of the project has created a useful tool in the design and creation of game levels, with a good basis for future expansion.

4.4 Further Work

To achieve a fully automated dungeon level generator the solution can be expanded to include the addition of enemies, bosses, and chests populating the mission space through the use of a constraint propagation algorithm alongside a tile-map conversion to transform the level from a space to a dungeon, additionally game themes could be introduced and consist of another node graph editing suite that assimilates a blueprints inspired environment similar to that seen in figure 62.

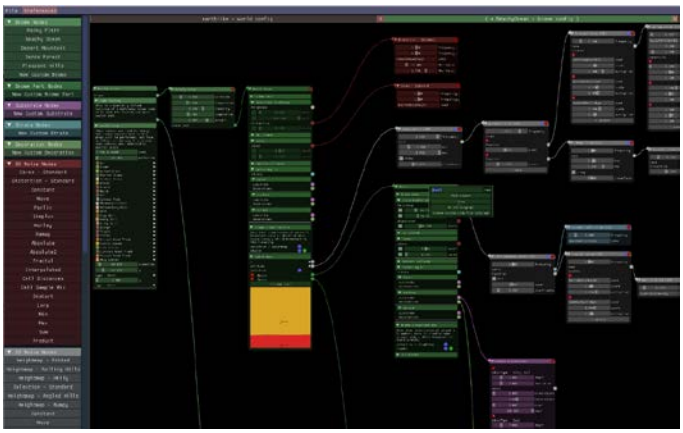


Figure 62 Blueprints World Editor (moohoohoh 2017)

References

Adams, D., 2002. *Automatic Generation of Dungeons for Computer Games.* (Bachelor of Science with Honours in Computer Science). The University of Sheffield.

Ahmad, M. A., ca. 2016. The BSP Tree. 1-4.

Alcorn, A., 1972. *Pong*. Atari.

Brewer, N., 2016. Going Rogue: A Brief History of the Computerized Dungeon Crawl.

Bucklew, B., 2017. *Procedural Generation in Game Design - Algorithms and Approaches*. Bora Raton FL: CRC Press.

Chalmers, A., 2013. *Collision Detection & Intersection Tests* [online]. sites.google.com: Available from: https://sites.google.com/site/mddn442/research-topics/collision-detection [Accessed 13/05/2018].

Chomsky, N., 1957. *Syntactic Structures*. Berlin, Germany: Walter de Gruyter & Co.

Chow, R., 2010. *Texture Atlas* [online]. wordpress.com: Available from: https://slizerboy.wordpress.com/tag/texture-atlas/ [Accessed 12/05/18].

Cichoń, M., 2016. *imgui-node-editor*.

Cornut, O., 2014. *Dear ImGui*. 1.50 [

Daler, E., 2017. *Meta stuff WIP*. 23b670e228050f0146468af55f7afa07131d7fd0 [

Dormans, J., 2010. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. *PCGames* [online].

Dormans, J., 2011. Level Design as Model Transformation: A Strategy for Automated Content Generation. *PCGames* [online].

Dormans, J., 2017. *Procedural Generation in Game Design - Cyclic Generation*. Boca Raton, FL: CRC Press.

Elaine Rich, K. K., Shivashankar B. Nair, 2009. *Artificial Intelligence*. Third edition. 7 West Patel Nagar, New Delhi: Tata McGraw-Hill Publishing Company Limited.

Frédéric Boudon, Christophe Pradal, Thomas Cokelaer, Przemyslaw Prusinkiewicz, Christophe Godin, 2012. L-Py: an L-system simulation framework for modeling plant architecture

development based on a dynamic language. *Frontiers in Plant Science*, 3 (76).

George Stiny, J. G., 1972. Shape Grammars and the Generative Specification of Painting and Sculpture, *Information Processing 71* (pp. 1460-1465). Amsterdam: North-Holland: C V Freiman.

Gygax, G. A., Dave, 1974. Dungeons & Dragons: TSR, Inc.

H. Ehrig, R. H., M. Korff, M. LÕwe, L. Ribeiro, A. Wagner, A. Corrandini, 1997. Single pushout approach and comparison with double pushout approach. *Algebraic Approaches To Graph Transformation Part II.* River Edge, NJ, USA: World Scientific Publishing Co., 247 - 312.

Johnson, D. M. R., 2017. *Procedural Generation in Game Design - Worlds*. Bora Raton FL: CRC Press.

Joost Engelfriet, G. R., 1997. *Handbook Of Graph Grammars And Computing By Graph Transformation*. World Scientific.

Julian Togelius, N. S., 2016a. *The Search-based Approach*. Springer.

Julian Togelius, N. S., Mark J. Nelson, 2016b. *Grammars and L-systems in relation to levels*. Springer.

Julian Togelius, N. S., Mark J. Nelson, 2016c. *Intro to PCG*. Springer.

Kyzrati, 2017. *Working With Seeds* [online]. Available from: http://www.gridsagegames.com/blog/2017/05/working-seeds/ [Accessed 16/03/18].

Mark J. Nelson, J. T., Cameron Browne, Michael Cook, 2016. *Rules and Mechanics*. Springer.

Microsoft, 2017. *Visual Studio 15 2017*.

Middag, B., 2016. *Controllable generative grammars for multifaceted generation of game levels.* (Master of Science in Computer Science Engineering). Ghent University.

moohoohoh, 2017. *Example with imgui from work* [Available from: https://www.reddit.com/r/gamedev/comments/4tz2hd/using_imgui_aka_dear_imgui_with_modern_c_for/ [Accessed 17/05/2018].

Müller, P. W., Peter & Haegler, Simon & Ulmer, Andreas & Van Gool, Luc., 2006. Procedural Modeling of Buildings. *ACM Transactions on Graphics*, 25 (3), 614-623.

Muratori, C., 2005. *Immediate-Mode Graphical User Interfaces - 2005* [Blog Post & Video Presentation]. Available from: https://caseymuratori.com/blog_0001 [Accessed 24/04/2018].

Noor Shaker, A. L., Julian Togelius, Ricardo Lopes, Rafael Bidarra 2016. *Constructive Generation Methods for Duneons and Levels*. Springer.

Oku, K., 2009. *Picojson*.

Stroustup, B., 2017. *Programming Language C++. [Working draft].* . ISO/IEC 14882:2017(E) [Geneva, Switzerland: International Organization for Standardization (ISO). .

Tim Tutenel, R. B., 2008. The Role of Semantics in Games and Simulations. *ACM Computers in Entertainment*, 6 (4), 35.

Unknown, 2017. *Basic BSP Dungeon Generation* [online]. roguebasin. Available from: http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation [Accessed

Vening, E., 2015. *How random is pseudo-random? Testing pseudo-random number generators and measuring randomness* [online]. Available from: http://pit-claudel.fr/clement/blog/how-random-is-pseudo-random-testing-pseudo-random-number-generators-and-measuring-randomness/ [Accessed

versions), A. I. D. N.-u., 1980. *Rouge*. Epyx.

Vuontisjärvi, H., 2014. *PROCEDURAL PLANET GENERATION IN GAME DEVELOPMENT.* (Information Technology, Software Development). Oulu University of Applied Sciences.

Zadravec, D., 2017. *A simple C++ class to generate random numbers* [online]. stackoverflow.com: Available from: https://stackoverflow.com/questions/41875884/how-to-generate-random-numbers-in-c-without-using-time-as-a-seed?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa [Accessed

Bournemouth University

Research Ethics Checklist

Reference Id	18275
Status	Approved
Date Approved	05/02/2018

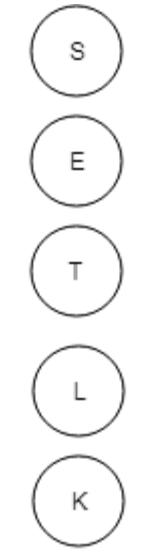
Researcher Details

Name	Kane White
Faculty	Faculty of Science & Technology
Status	Undergraduate (BA, BSc)
Course	BSc Games Programming
Have you received external funding to support this research project?	No

Project Details

Title	Implementation of a Procedural Level Generation Engine for Developing 2D Rouge-like Dungeons
Proposed Start Date of Data Collection	31/10/2017
Proposed End Date of Project	18/05/2018
Supervisor	Simant Prakoonwit
Approver	Simant Prakoonwit

Summary - no more than 500 words (including detail on background methodology, sample, outcomes, etc.)



Rule Set Diagram created with https://www.draw.io

$$G = \left(G_{v,}\ G_{E,}\ s^{G}\ ,\ t^{G} l v^{G}\ ,\ l e^{G} \right)$$

Formulaic definitions created with https://www.fastfig.com/app