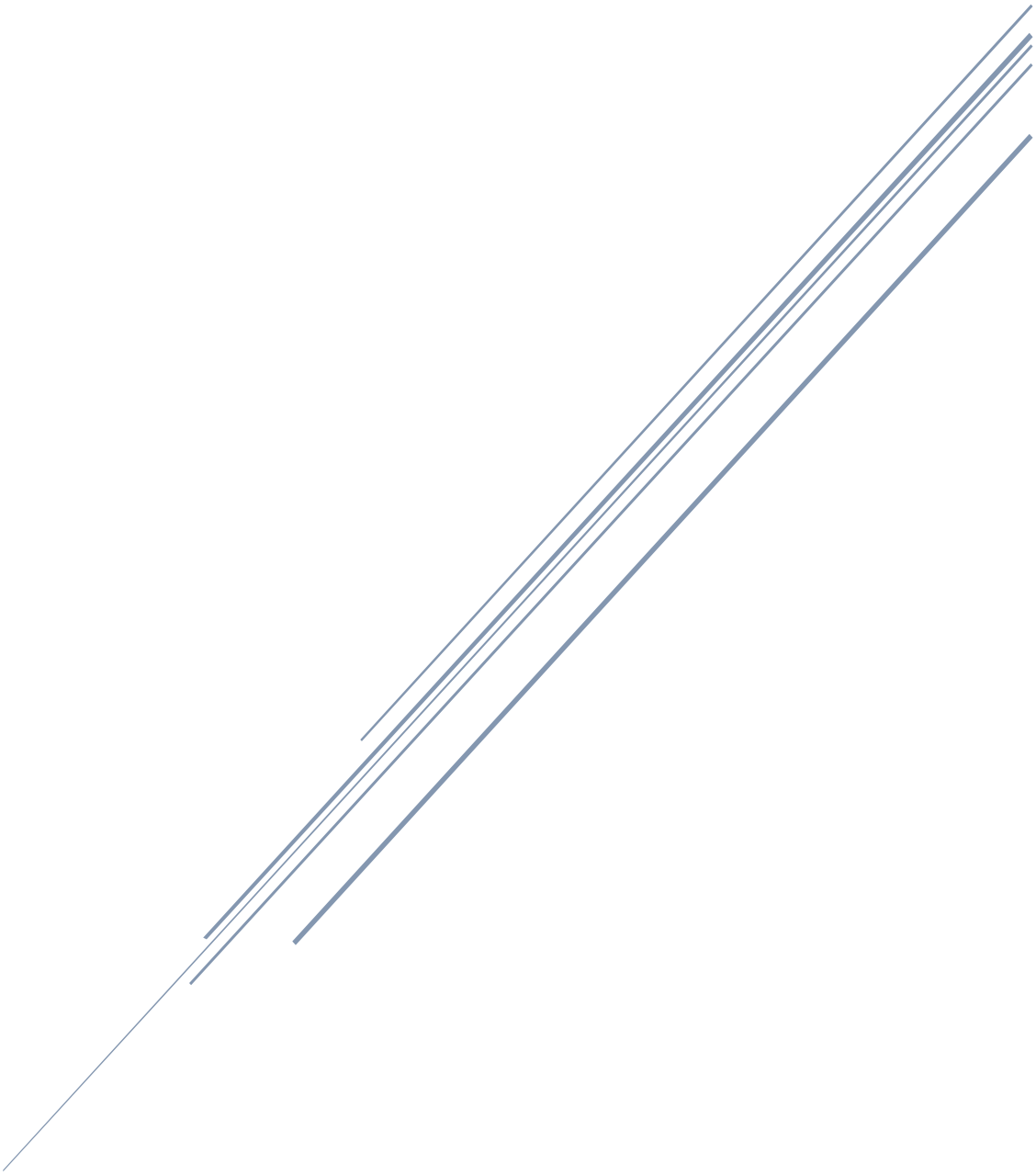
GRAPH REWRITING

Will Bennett, 20014262k, B014262k@student.staffs.ac.uk,
and Shaun Reeves



DTA
Masters of Negotiated Study

Table of Contents

Figure Table

Glossary

PCG – Procedural Content Generation

Key Words

PCG, Cyclic, User, Graph Grammars, Unity, Level Generation, Developer

1. Introduction

In an industry that is rapidly adapting, it is increasingly important to create tools to ease development of game. This is especially true with procedural content generation, as this generative algorithm has the potential to speed up the development of games when used correctly. An example of a tool to do this is Ludoscope (Dormans, N.D) which was created by Dormans and used to create Unexplored's levels. While this tool is able to achieve its purpose, the lack of public access and documentation makes it hard to use. This project will explore the procedural content generation technique of graph rewriting in order to create a graph that represents a roguelike dungeon. In doing so it will aim to produce a generative grammar tool that is able to be used in a similar way of Ludoscope was used in "An exploration of cyclic procedural content generation" (Bennett, 2023).

1.1 Aim

An existing prototype game system will be assessed and developed further to identify how it can be improved in terms of user experience.

1.2 Objectives

- What makes a good roguelike game?
- What type of users would be using the tool?
- Consider improvements to the framework for the developer.
- Break down what cyclic PCG could offer a developer framework wise.
- What type of PCG will the framework be using?

2. Project Methodology

The methodology chosen is Agile (Wrike, N.D), which is an iterative approach to project management. Agile has multiple ways of being applied, this project is using a Kanban board (Rehkopf, n.d.). The main benefit of this method is allowing for constant reflection and the project progress being visually presented. Both create an easier environment to manage the current and future progress of the project. By using a Kanban board, the larger tasks are broken down into smaller sections. These cards are then sorted into columns to indicate their progress. These cards can have extra details, but the main component is strict deadlines, this assures that the project is being complete.

3. Background

This project is a follow up to a previous project completed at undergraduate (Bennett, 2023). That project used Ludoscope to create grammar rules which was then exported to Unity which created mission space in a pseudo environment to gather rating of the dungeon layout. The grammar rules were replicating Dormans' theory of Cyclic grammar (2017). The downside of this is to adjust the grammar rules, the developer would have to access the rules through Ludoscope (Dormans, N.D). Additionally, Ludoscope has no documentation due to it being private software made by Dormans. This made the project awkward to use and learn, while external developers could not edit grammar rules. The aim of this project is to make the experience easier by removing the need for Ludoscope by recreating part of the software in Unity. This will not only allow for external usage, but documentation can be created to allow the framework to be easy to use.

4. Project Plan

4.1 Scope

The aim of this project is to expand on an existing game system, this system was using an external tool to create grammar rules which were then exported into Unity (Unity Technologies, 2023). To scope the development of this project, the artifact will be created to allow for the creation and application of grammar rules in Unity. This is limited to a numerical application which is not the easiest way to use such a framework, but it will be able to be expanded upon in later iteration of the project.

5. Literature Review

5.1 Perspective from a user

5.1.1 What makes a good roguelike game?

Roguelikes have been a popular genre for indie developers to create games for ever since Rogue (Epynx, Inc., 1985) was released. Many of these games are labelled as shining examples of the genre due to their mechanics, map design, or just pushing the standard for roguelikes forward. This section will breakdown what makes a roguelike good.

Design rules

Roguelikes balance difficulty and punishing the player through permadeaths throughout the playthrough. To aid the player, the inclusion of design rules can introduce "reasonable play", which is a state of neutral danger (Harris, 2011). Harris further details some rules that facilitate a successful roguelike. Most of the rules relate directly towards combat and items, however the "No beheading rule" can relate to the design of the dungeon. This rule means that the player should not instantly die in a single attack. Harris refers to Nethack (1978) which had a Medusa enemy that killed the player on sight. This was changed to Medusa being in a certain room that the player can learn about and be prepared to avoid the one shot.

Gameplay elements

Roguelikes also have common gameplay elements between all games, below is some theorised by Viana (2021) and Linden (2014) in their research. All of them can be used combined with generality (Breno Viana, 2021) to create a cohesive dungeon for the user.

Gameplay element	**Description**
Item/skill	which is an obtainable object that grants
Barrier	a feature that is a temporary block of progression
Reward	an encouragement bonus for the player which is not mandatory
Challenge	obstacle that hinders players progression
Puzzle	an obstacle that requires intelligence to progress past it.

Randomness control

The roguelike genre highly relies on controlled randomness. Too much randomness can cause the game to be unfun. Therefore, to give the player an advantage with playing the player can learn mechanics and try to use them to their advantage. The design rules listed above can aide in this as the player can learn these and how they relate to the game they are playing and then reduce the randomness by having a greater understanding of the game.

5.2 Perspective from a developer

5.2.1 Generative Grammars

Generative grammars take an input of a sequence of symbols and then generate sequences over or into the original sequence (Noor Shaker, 2016). An example of this shows a sequence of flowers and the grammar rules related to them.

- $0 \longrightarrow P$
- $P \longrightarrow Fl$
- $Fl \longrightarrow Fr$
- $Fr \longrightarrow Fl$
- $Fr, Fl \longrightarrow P$
- $Fl, P \longrightarrow 0$

Figure 1: Grammar rule for flowers (Franco, 2023)

1. 0,
2. 0, P
3. 0, P, Fl
4. 0, P, Fl, Fr
5. 0, P, Fl, Fr, Fl
6. 0, P, Fl, Fr, Fl, P
7. 0, P, Fl, Fr, Fl, P, 0

Figure 2: Step-by-step application of this grammar (Franco, 2023)

5.2.2 Graph Grammar

Graph grammars (GG) allows for the manipulation of graphs instead of strings that other methods offer, this offers greater control over randomness in the procedures and the end results (Adams, 2002). Graphs can be broken down to nodes and edges. Edges can be directional or unidirectional, and nodes can have variables attached thus allowing them to store more information (The Shaggy Dev, 2022).

Most approaches of using GG can be broken down to left-hand and right-hand, both contain graphs. In context-free graphs grammars, the left hand must be one node only, while context-sensitive grammars allow for full graphs on both hands (Adams, 2002). A redex happens within graph transformations, this is where through repeatedly applying productions, a match from left-hand is found in the origin graph. The matching nodes then get replaced with the right-hand production, and then reconnected to the origin graph with nodes. This production application can be applied in two different ways, algebraic and algorithmic.

Algorithmic node replacement system

Using this method, when a match is found all connected edges to the match are deleted and then the new part is reconnected via an embedding mechanism which is specified by connection instructions (Rozenberg, 1997). Node Label Controlled (NLC) mechanism is one of these instructions, this method relies on node labels. To set up for its use the left-hand production must be a single mother node or as context free graph grammar. Then when the production occurs each connection instruction is an ordered pair which all connections will form a connection relation. However, a downside of this is all productions use the same connection instruction. This means the rest of generated nodes will be connected in the same way.

Algebraic Node Replacement Systems

Algebraic offers a similar method for node replacement as any matching nodes in the host graph and are not present in the right-hand of production are deleted and if there are nodes that are on the left side but are missing from the right, they get added back to the host graph (Rozenberg, 1997). As this system is context sensitive, an identifier needs to be added to the graph for the system to choose the correct nodes for transformation. This is solved through morphism, which is a relation between nodes on the left-hand and right-hand of the production.

As these are graph grammars this can be done via numbers, allowing matching nodes to share numbers. Generally, to increase readability the same number nodes share the same colour and shape. Where this differs from algorithmics' replacement is through its reattachment process. There are two steps for reattaching the new graph to the host graph, with both using context elements. Context elements stay the same on both hands of the process which allows a bridge from the host graph to the new nodes as the context elements stay connected in the same way. To attach the new nodes a method called pushout is used, instead of embedding mechanisms. Rozenberg (1997) separates the pushout process into two methods, double-pushout (DPO) and single-pushout (SPO).

Both methods aide in cleaning up dangling edges, which is when a connecting edge from the host graph loses the other node from the transformation process. The SPO method automatically deletes this dangling edge. Meanwhile DPO method allows for rules to be defined for the process of dealing with deletion of edges. If a dangling edge occurs and deletion is not specified, then the rewrite process would not occur. SPO allows for a simple approach to creating productions, while DPO is more complex but avoids accidental deletion of hanging edges which can leave nodes unconnected. To set up DPO an interface graph must be set up to display the context elements.

Negative Application Conditions

The previous two sections were dedicated to positive application additions, where if a condition is met then the production is done. Negative Application Conditions (NACs) can be defined to set conditions for when the right-hand should not be applied (Annegret Habel, 1996). These can be used to give more control on the expected outcome of the graph and to stop transformations from affecting certain nodes.

5.2.3 Level design

Key considerations

Level design is an important part of creating a dungeon, especially procedurally since key concepts must be prevalent in all iterations. Below are some concepts that when considered help boost level design quality. Saltzman (2000), Bleszinki (2000), and Ryan (1999) (1999) all explore these concepts with key ones being level architecture, pacing, difficultly, resource balance, and nonlinear. These concepts can be considered for the user's expectations for the framework to accommodate.

Missions into spaces

Dormans (2010) explored the idea of generating mission and spaces while exploring generating a dungeon like the Zelda series (Nintendo EAD, N.D). In this he explored the idea of generating missions first then generate spaces to accommodate the missions using GG to suit each need. Dormans narrows down Zelda dungeons to spaces that can be represented by abstract nodes and edges. While missions can be shown via directed graphs that show the order of completion of tasks. The separation of these two can be seen in Figure 3.

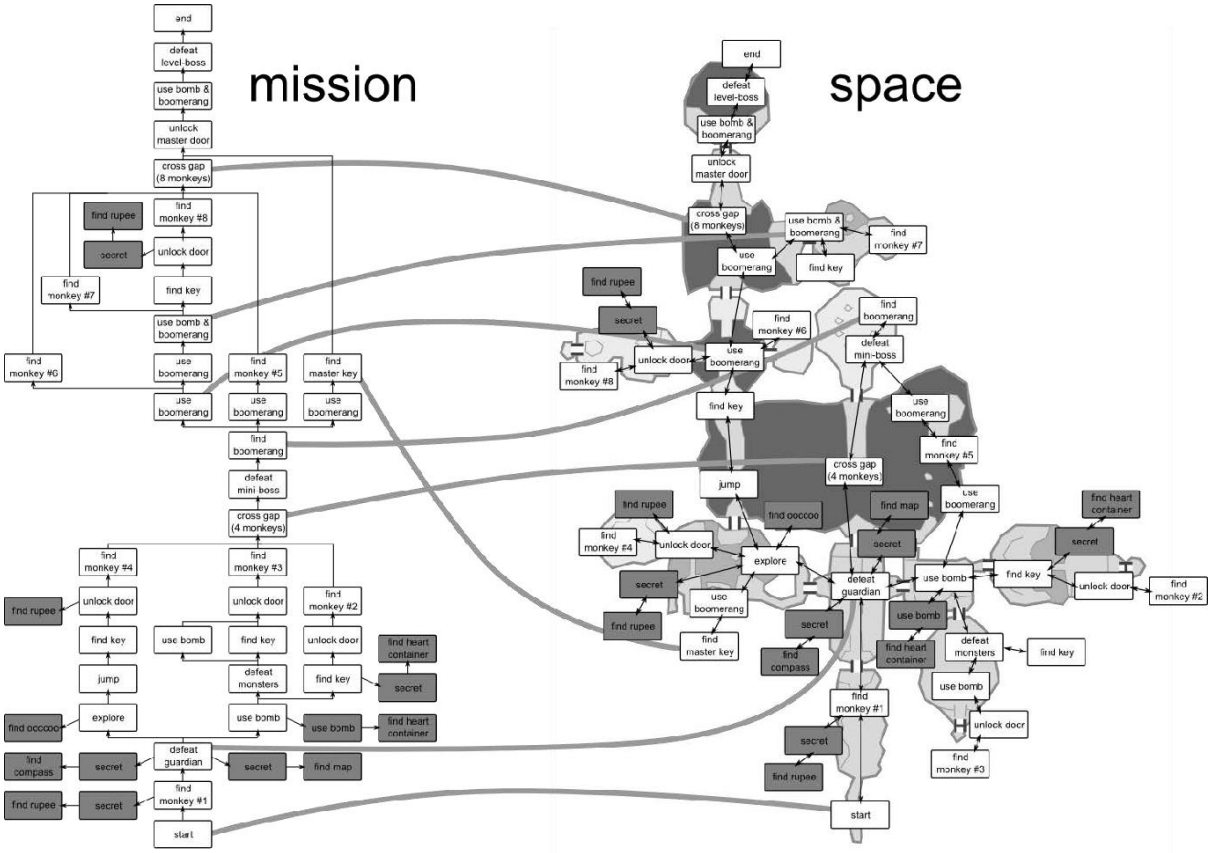


Figure 3: Mission and space in the Forest Temple of The Legend of Zelda: The Twilight Princess (Dormans, 2010)

Dormans (2010) declares that graph grammars are well suited to generate mission spaces due to the ease of expressing them as non-linear graphs. This would need an alphabet to depict this graph and its tasks and rewards. Dormans then details how shape grammar is best suited to generate spaces, this is due to them working like generative grammars, with the alphabet relating to physical space. This can then be used to replace connections through walls to create junctions or corridors. Shape grammars can be done recursively which allows for more variation and have a natural feeling.

Continuing this idea, Dormans talks about generating space from missions. To do these adjustments are made to the shape grammar, this is to change rules for shape grammar to associate with the terminal symbols from the mission grammar. These terminal symbols act as building instructions for the shape grammar. A way of prototyping this is to allow the shape grammar to find the next mission symbol and choses a rule randomly based on relative weight. This rule is then applied at a location that is picked randomly based on its relative fitness. The replaced symbol is then saved as a reference to allow for coupling. In this instance coupling would stop keys and locks from being placed randomly (Dormans, 2010). Dynamic parameters can be introduced to allow for level design features to affect the level such as difficulty changes.

After the mission grammar is accounted for, the shape grammar will return to normal operations, such as iterating over all non-terminal symbols. This will be led by a set of rules to finalise the space.

5.1 Framework

As this project aims to create a framework for developers to use, it is important to know what a framework is. A framework is a structure that provides a base for development process, this can help other developers from creating projects from scratch. This can be used as a template to help meet project requirements and speed up development processes (aprekshamathur, N.D) . Frameworks should have specific uses and to be easy to use (Principle To Practice, N.D). This is to help reduce redundancy in code and promote good coding practices (aprekshamathur, N.D). Overall, this can help reduce time and cost associated with the frameworks use case in the project.

In terms of this framework, its focus is providing a base framework for creating a level design using graph based PCG. One key feature of the framework is to be able to represent mission data in a graphical way. This should be done in a way to allow a developer to create rules that will transform an empty graph into a coherent representation of missions. As Dormans' (2010) research states these mission spaces can then be transformed into a playable environment that can be used in a game. This can help reduce the amount of time spent on developing a replayable environment by the developer creating rules.

Using this framework different PCG approaches can be used to improve the flow of missions and spaces into each other. An example of this is cyclic PCG (Dormans, 2017), Dormans devised this concept and then later applied to his game Unexplored (Ludomotion, 2017). This combined with the level design concept of mission and spaces allowed for reduction of branching in dungeon level design which leads to a more cohesive level layout. This combined with the framework using algebraic node replacement, a wide range of rules can be made to consolidate a range of outputs.

6. Design

6.1 Unity

Unity is a software that is developed by Unity Technologies. It is a cross-platform game engine which supports both 2D and 3D video games and simulations. Within this it provides dimensional manipulation and simulations (Unity Technologies, 2023).

Unity has two purposes for this project, to run grammar rule scripts and to visualise the graph's data which will be done through gizmos. Unity does not have an advantage over other game engines such as Unreal or Godot as all three are popular within the game development environment. Unreal is unable to be run due to specification issues, as on launch it throws an unexpected error as seen in Figure 4. Unity was chosen due to previous experience with the software will allow for the focus to be on the framework instead of being split on learning a new engine too.

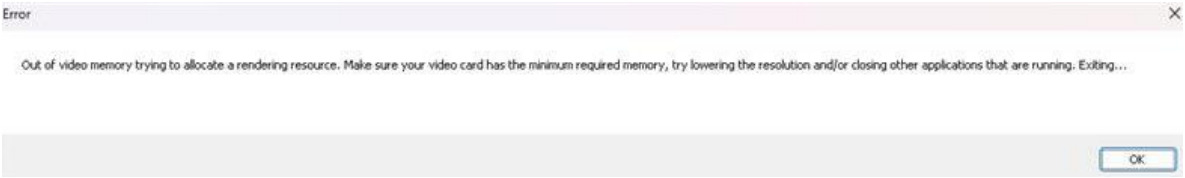


Figure 4: Image of video memory error from Unreal Engine

6.2 Requirements

The graph grammar system should be its own separate entity, this will allow it to be coupled with other systems to generate missions which can turn into physical game space. While the graph should only consist of nodes and edges, the nodes can store data about the mission for later usage. The system should have features that enable it to generate these missions.

Requirement	Purpose of requirement
Ability to store nodes within nodes	this will allow mission data to be associated with a node.
Ability to apply rule randomly if no pre-set has been specified	random will act as a default.
Ability to link rules together	this will create a recipe which rules will be applied in order.

6.3 Graph Grammar Approach

Graphs can come free formed, but in this project the graph will be linked to a grid. This will make applying rules easier as edges cannot overlap each other. There are two approaches that can be taken with graph grammar, algorithmic and algebraic. Due to algorithmic being limiting on how new nodes are connected to the graph, algebraic will be used. As the graph will be limited to a grid, implementing SPO is the better solution for dangling edges. While nodes will store data about themselves and the mission edges will not, this is because edges will only indicate which nodes can be access from another. Edges will have additional data which if its directional.

Grammar rules will use a context sensitive approach. This means more than one node can be on the left-hand side. This will allow for more expressive rule design which is needed to replicate a cyclic PCG approach.

7. Implementation

7.1 Data storage

Mission data will be represented in a graph. A graph in C# is a data structure that is made of nodes and edges. In this project this graph will be stored as a dictionary, this will allow a key and a value to represent the mission data. The key will be the unique index of the node, and the value will be the node data.

An issue with this implementation is Unity has no way to serialize the data to the editor, to overcome this the dictionary must be broken down into a list. This list will store a data linker class, which stores the index (key) and the struct for the node data (value).

As there will only be one graph, other classes will be able to access this data through a static class called GraphInfo. This allows the user to input data into a component class called GraphComponent which handles the communication to and from the GraphInfo class. This component class inherits from Unity's MonoBehaviour allowing it to be attached to game objects within the scene. When the framework is run, it will send initialise the graph from the serialized variables called rows and columns.

7.1.1 Representing the data

To allow the user to see rules being run, the graph data is represented using Unity's Gizmos (Unity Technologies, N.D.), this is done within GraphComponent. The gizmos draw edges which can be directional and nodes. Nodes are represented by a sphere; this sphere is assigned a colour based on its symbol which will be explained later. Edges are a line unless they are directional which makes them an arrow. This is determined by taking its origin node index from its to node index, and the direction is determined form this. A blank graph can be seen in Figure 5.

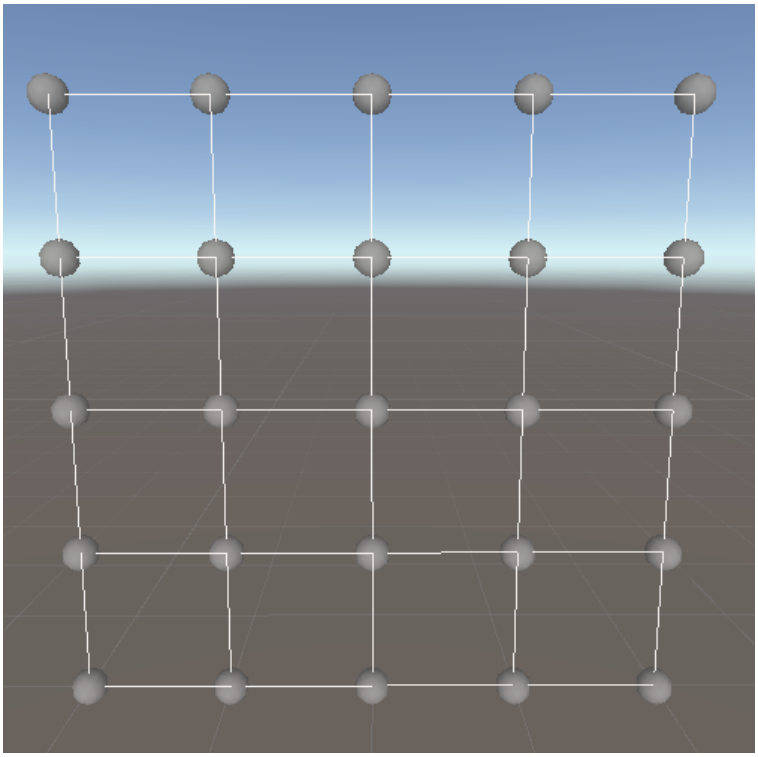


Figure 5: Example image of a blank graph made with gizmos

7.2 Grammar

Due to scaling the project back to not over scope, grammar rules have been limited to numerical values. This means to reference a node the user will have to use grid positions. To make creating rules easier, they have been made into scriptable objects, this is a data container that allows the saving of data independently to class instances (Unity Technologies, N.D). Using scriptable objects allow for creation of new rules to be easily implemented into Unity's editor under "Assets" as seen in Figure 6. This creates a blank rule as seen in Figure 7. The rule is broken up into three sections, run info, left hand, and right hand. Run info dictates how many times you want the rule to be ran, this allows for the rule to be run multiples times until the rule cannot be implemented, and a probability slider, this dictates the probability of the rule being ran. Left hand shows nodes and edges the rule should find. While the right hand is what these rules will be replaced with.

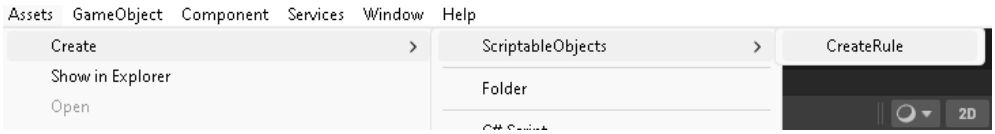


Figure 6: Screenshot of where the creation of new rules is in editor

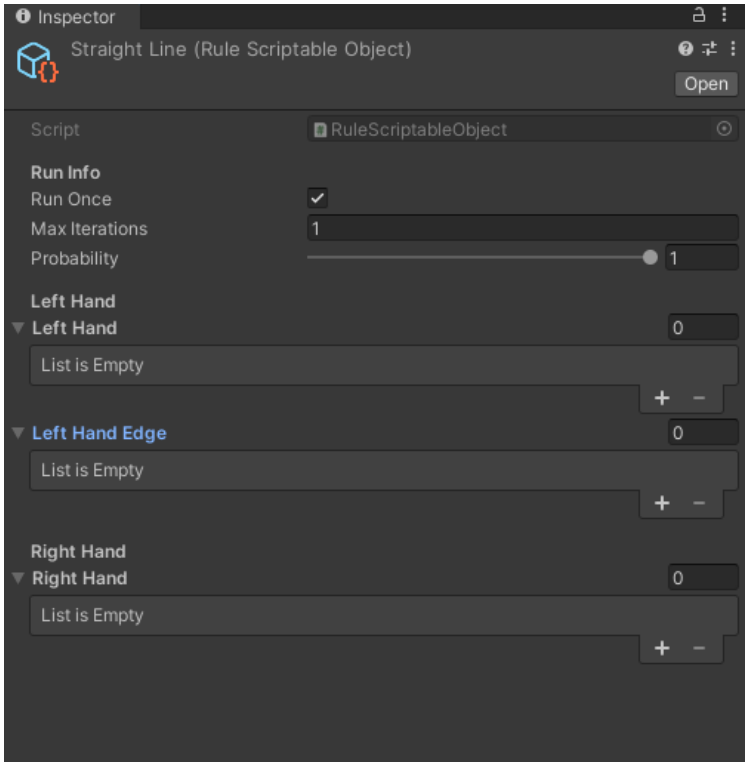


Figure 7: A screenshot of a blank rule.

7.2.1 Left Hand Algorithm

The left hand represents the nodes the rule will find, to determine this the user will have to put in the symbol they want to look for and its position. The find node's position will determine the starting point of the rule, to allow the user to find a random starting position, using (-1, -1) as seen in Figure 8. The other elements dictate the direction from the last node using their position data. In this example the default symbol is 'n'.

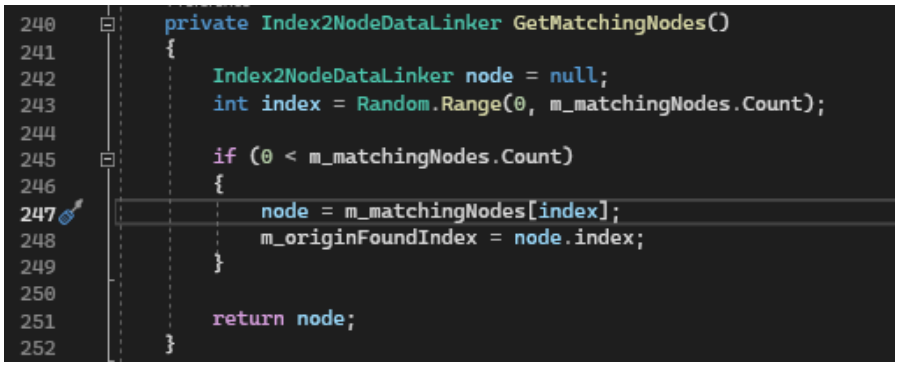
Figure 8: Screenshot of an example of left-hand data for a rule

The algorithm for finding the nodes is PopulateMatchingNodes() is seen in Figure 9, it gets passed reference s of rule, nodes, and index, which are the current rule being applied, the dictionary of nodes, and the index of the node left-hand list. This then loops through the graph to find nodes with matching symbols and adds them to a separate list called m_matchingNodes. As seen on line 215, if the left-hand position is (-1, -1) and the symbol matches any position will get added.

```
209     private void PopulateMatchingNodes(RuleScriptableObject rule, List<Index2NodeDataLinker> nodes, int index)
210     {
211         foreach (Index2NodeDataLinker node in nodes)
212         {
213             if (node.nodeData.symbol == rule.m_leftHand[index].m_symbol
214                 && (node.nodeData.position == new Vector3(rule.m_leftHand[index].m_nodePosition.x, rule.m_leftHand[index].m_nodePosition.y)
215                 || rule.m_leftHand[index].m_nodePosition == new Vector2(-1, -1)))
216             {
217                 m_matchingNodes.Add(node);
218             }
219         }
220
221     }
```

Figure 9: Screenshot of function PopulateMatchingNodes()

m_matching nodes is used to get a node to apply the rule to, this is used in GetMatchingNodes() which can be seen in Figure 10. If no specific node position was searched for the index will be randomly selected from list, otherwise it will be the only node added.



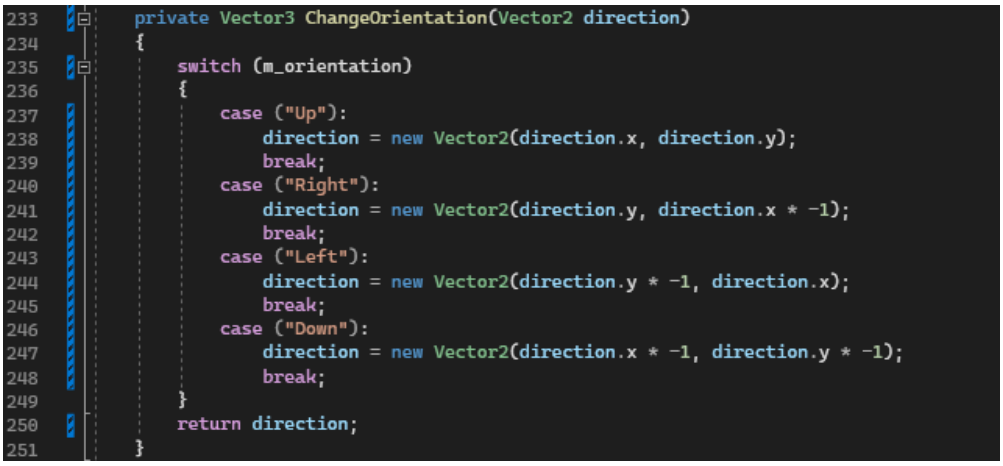
```
240     private Index2NodeDataLinker GetMatchingNodes()
241     {
242         Index2NodeDataLinker node = null;
243         int index = Random.Range(0, m_matchingNodes.Count);
244
245         if (0 < m_matchingNodes.Count)
246         {
247             node = m_matchingNodes[index];
248             m_originFoundIndex = node.index;
249         }
250
251         return node;
252     }
```

Figure 10: Screenshot of function GetMatchingNodes()

After the first node is found, a different function is used to find the node based off the direction which is called GetNeighbouringNodes() as seen in Figure 11. This function is more complex than GetMatchingNodes(), due orientation. Being able to rotate the direction of the nodes allows for the rule to have more variation. This orientation change must persist throughout the rule to keep the developer's intent for the rule intact. The implementation of this can be seen in Figure 12, where is returns the node's new position based on its direction.

```
215     private Index2NodeDataLinker GetNeighbouringNodes(RuleScriptableObject rule, int index)
216     {
217         Index2NodeDataLinker node = null;
218         Index2NodeDataLinker lastNode = m_nodeGraph[m_originFoundIndex];
219
220         Vector3 directionToCheck = lastNode.nodeData.position + ChangeOrientation(rule.m_leftHand[index].m_nodePosition);
221
222         node = CheckNode(rule, m_nodeGraph, index, directionToCheck);
223         if (node != null)
224         {
225             m_originFoundIndex = node.index;
226         }
227         else
228         {
229             Debug.Log("node doesnt match orientation");
230         }
231         return node;
232     }
```

Figure 11: Screenshot of GetNeighbouringNodes().



```
233     private Vector3 ChangeOrientation(Vector2 direction)
234     {
235         switch (m_orientation)
236         {
237             case ("Up"):
238                 direction = new Vector2(direction.x, direction.y);
239                 break;
240             case ("Right"):
241                 direction = new Vector2(direction.y, direction.x * -1);
242                 break;
243             case ("Left"):
244                 direction = new Vector2(direction.y * -1, direction.x);
245                 break;
246             case ("Down"):
247                 direction = new Vector2(direction.x * -1, direction.y * -1);
248                 break;
249         }
250         return direction;
251     }
```

Figure 12: Screenshot of ChangeImplementation().

After the direction of the node is applied, it must be checked. As seen in Figure 12, the graph is scraped to find a node that matches the directional position. This node is then checked to see if it matches the rule's symbol. If it does, then the node is returned which GetNeighbouringNodes() uses to set m_originFoundIndex and return to apply the right-hand rule data to.

```
252     private Index2NodeDataLinker CheckNode(RuleScriptableObject rule, List<Index2NodeDataLinker> graph, int index, Vector3 direction)
253     {
254         Index2NodeDataLinker node = null;
255         Index2NodeDataLinker nodeToCheck = null;
256 
257         foreach (Index2NodeDataLinker vertex in graph)
258         {
259             if (vertex.nodeData.position == direction)
260             {
261                 nodeToCheck = vertex;
262             }
263         }
264         if (nodeToCheck != null && nodeToCheck.nodeData.symbol == rule.m_leftHand[index].m_symbol)
265         {
266             node = nodeToCheck;
267         }
268 
269         return node;
270     }
```

Figure 13: Screenshot of CheckNode().

The returned node is then passed by reference along with the righthand, and the index of the left-hand rule as seen in Figure 14. This adds the node to m_nodesToChange which will be explained later, and changes the symbol, as well as changing the colour based on the new symbol by cross-referencing the alphabet. This is then updated visually automatically.

```
            2 references
270         private void SetNodeData(RightHand rightHand,int i, Index2NodeDataLinker matchingNode)
271         {
272             m_nodesToChange.Add(matchingNode);
273             matchingNode.nodeData.symbol = rightHand.m_nodeDataList[i].symbol;
274             foreach (AlphabetLinker data in m_alphabet.m_alphabet)
275             {
276                 if (matchingNode.nodeData.symbol == data.m_symbol)
277                     matchingNode.nodeData.colour = data.m_colour;
278             }
279         }
```

Figure 14: Screenshot of SetNodeData().

Changes to approach

The approach for finding initial nodes to apply the rule was changed halfway through the project, the current approach allows for orientation differences as well as keeping the structure of the rule consistent. The old approach originally found a single node and chose a random direction for the next. This led to rules that only cared for symbol changes and held no form. Due to this change, orientation had to be consistent for the node's application direction. The approach used for this can be seen in Figure 12.

7.2.2 Resetting

When all nodes have tried to be applied, they must be checked to assure the rule has been completely applied. This is done via checking m_nodesToChange's count against the right-hand's count as seen in Figure 15 If the check fails then ResetRule is called, this function resets the nodes in m_nodesToChange back to their symbol and data before the rule was applied as seen in Figure 16. If the check passes, then edges and stored nodes are applied to the graph which can be seen in Figure 17.

```
302     private bool ApplyNodeChanges(RuleScriptableObject rule, RightHand rightHand)
303     {
304         bool applied = false;
305         if (m_nodesToChange.Count != rightHand.m_nodeDataList.Count)
306         {
307             ResetRule(rule);
308         }
309         else
310         {
311             Debug.Log("node changes applied");
312             applied = true;
313         }
314         return applied;
315     }
```

Figure 15: Screenshot of ApplyNodeChanges().

```
2 references
346     private void ResetRule(RuleScriptableObject rule)
347     {
348         Debug.LogWarning("rule cant be applied");
349         for (int i = 0; i < m_nodesToChange.Count; i++)
350         {
351             m_nodesToChange[i].nodeData.symbol = rule.m_leftHand[i].m_symbol;
352             foreach (AlphabetLinker data in m_alphabet.m_alphabet)
353             {
354                 if (m_nodesToChange[i].nodeData.symbol == data.m_symbol)
355                     m_nodesToChange[i].nodeData.colour = data.m_colour;
356             }
357         }
358     }
```

Figure 16: Screenshot of ResetRule().

```
146             if (ApplyNodeChanges(rule, rule.m_rightHand[rightHandIndex]))
147             {
148                 ApplyEdgeChanges(rule, rule.m_rightHand[rightHandIndex]);
149
150                 //sort stored nodes
151                 ChangeStoredNodeData(rule, rightHandIndex);
152                 break;
153             }
154             m_nodesToChange.Clear();
155         }
156         return true;
```

Figure 17: Screenshot of applying edges and stored nodes.

7.2.3 Edges and Stored Nodes

Edges are created in two steps, setting the to and from nodes. On first pass, when the first node is found, the from node is set by passing SetFromNode the matching node's index. After this, when the next node in the rule is found the from node is set before the new matching node is set, which after the node changes have been done the to node is set via the SetEdge function as seen in Figure 18 and Figure 19. This is then repeated for all other edges with each loop updating m_lastNodeIndex which is used for looping the last nodes together.

```
108             m_edgeCount = 0;
109             for (int i = 0; i < rule.m_leftHand.Count; i++)
110             {
111                 Debug.Log("This node = " + i);
112                 if (1 <= i)
113                 {
114                     if (matchingNode != null)
115                         SetFromNode(matchingNode.index);
116                     matchingNode = GetNeighbouringNodes(rule, i);
117                     if (matchingNode == null)
118                     {
119                         break;
120                     }
121                     SetNodeData(rule.m_rightHand[rightHandIndex], i, matchingNode);
122                     SetToNode(matchingNode.index);
123 
124                     m_lastNodeIndex = m_toNodeIndex;
125                     SetEdge(rule);
126 
127                 }
128                 else if (1 < rule.m_leftHand.Count)
129                 {
130                     matchingNode = GetMatchingNodes();
131                     if (matchingNode == null)
132                     {
133                         break;
134                     }
135                     SetNodeData(rule.m_rightHand[rightHandIndex], i, matchingNode);
136                     SetFromNode(matchingNode.index);
137                     m_firstNodeIndex = m_fromNodeIndex;
138                     SetOrientation();
139                 }
140             }
```

Figure 18:Screenshot of a for loop setting edge and node data.

```
374     3 references
        private void SetFromNode(int fromNode)
375     {
376         m_fromNodeIndex = fromNode; ;
377     }
        2 references
378     private void SetToNode(int toNode)
379     {
380         m_toNodeIndex = toNode;
381     }
        2 references
382     private void SetEdge(RuleScriptableObject rule)
383     {
384         rule.m_leftHandEdge[m_edgeCount] = new LeftHandEdge(
385             rule.m_leftHandEdge[m_edgeCount].m_symbol,
386             m_fromNodeIndex,
387             m_toNodeIndex,
388             rule.m_leftHandEdge[m_edgeCount].m_directional
389             );
390 
391         m_toNodeIndex = m_fromNodeIndex = -1;
392         m_edgeCount++;
393     }
394 
```

Figure 19:Screenshot of setting edge data functions.

Looping the last and first edges together is done if the user has specified in the rule, this can be seen in Figure 20, and Figure 21 to see the option in editor. This is similar to setting normal edges but uses previously assigned variables to set the from node and to node from the first and last node that was found while setting previous edges.

```
280     private void LoopEdge(RuleScriptableObject rule)
281     {
282         Debug.Log("from node = " + m_firstNodeIndex);
283         Debug.Log("to node = " + m_lastNodeIndex);
284         foreach (var edge in m_edgeGraph)
285         {
286             if (edge.edgeData.graphToNode == m_firstNodeIndex || edge.edgeData.graphToNode == m_lastNodeIndex)
287             {
288                 if (edge.edgeData.graphFromNode == m_lastNodeIndex || edge.edgeData.graphFromNode == m_firstNodeIndex)
289                 {
290                     if (edge.edgeData.symbol == rule.m_leftHandEdge[rule.m_leftHandEdge.Count - 1].m_symbol)
291                     {
292                         Debug.Log("setting last link edge");
293                         SetFromNode(m_lastNodeIndex);
294                         SetToNode(m_firstNodeIndex);
295                         SetEdge(rule);
296                         break;
297                     }
298                 }
299             }
300         }
301     }
```

Figure 20: Screenshot of LoopEdge().

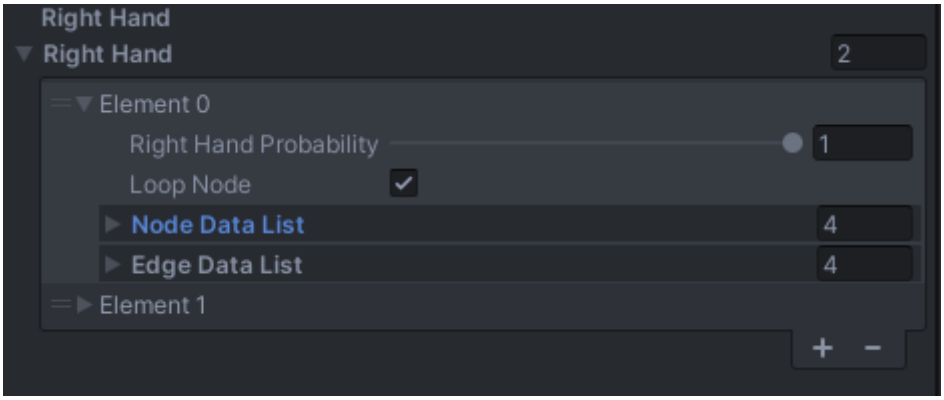


Figure 21: Screenshot of loop node option.

When ChangeStoredNodeData is called it created new stored nodes based on the right-hand rule. When the node is being created it is assigned a random location within a range of the node to avoid overlapping. This node then gets its data set and is added to m_storedNodeGraph as seen in Figure 22. A similar process is done with edges, but instead the edge will be positioned along the z direction towards the graph, these edges are added to m_edgeGraph as seen in Figure 23.

```
for (int i = 0; i < m_nodesToChange.Count; i++)
{
    foreach (StoredNodeData storedNode in rule.m_rightHand[rightHandIndex].m_nodeDataList[i].storedNodes)
    {
        GraphInfo.graphInfo.nodeIndexCounter++;
        Index2StoredNodeDataLinker newStoredNode = new Index2StoredNodeDataLinker(GraphInfo.graphInfo.nodeIndexCounter, storedNode);
        newStoredNode.storedNodeData.parentIndex = m_nodesToChange[i].index;
        float randomPosMod = Random.Range(-0.25f, 0.25f);
        newStoredNode.storedNodeData.position = new Vector3(m_nodesToChange[i].nodeData.position.x + randomPosMod, m_nodesToChange[i].nodeData.position.y + randomPosMod, 1f);
        foreach (AlphabetLinker data in m_alphabet.m_alphabet)
        {
            if (newStoredNode.storedNodeData.symbol == data.m_symbol)
            {
                newStoredNode.storedNodeData.colour = data.m_colour;
            }
        }
        m_storedNodesGraph.Add(newStoredNode);
```

Figure 22: Adding new node for contained nodes within ChangeStoredNodeData().

```
EdgeData edgeData = new EdgeData();
edgeData.symbol = 'b';
foreach (AlphabetLinker data in m_alphabet.m_alphabet)
{
    if (edgeData.symbol == data.m_symbol)
    {
        edgeData.colour = data.m_colour;
    }
}
edgeData.directional = true;
edgeData.fromNode = GraphInfo.graphInfo.nodeIndexCounter;
edgeData.toNode = m_nodesToChange[i].index;
edgeData.graphFromNode = GraphInfo.graphInfo.nodeIndexCounter;
edgeData.graphToNode = m_nodesToChange[i].index;
edgeData.position = new Vector3(newStoredNode.storedNodeData.position.x, newStoredNode.storedNodeData.position.y, newStoredNode.storedNodeData.position.z);
Index2EdgeDataLinker newEdge = new Index2EdgeDataLinker(GraphInfo.graphInfo.edges.Count, edgeData);

m_edgeGraph.Add(newEdge);
```

Figure 23: Adding new edge for contained nodes within ChangeStoredNodeData().

Both edges and stored nodes use a new type of edge, directional edges. These are stored via a bool in edgeData and are visualised via an arrowing pointing to the to node which can be seen in Figure 24.

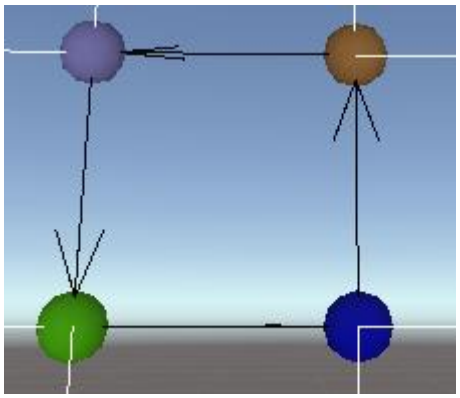


Figure 24: Screenshot of directional edges.

7.2.4 Alphabet

An alphabet has been refenced across the implementation process, in this project's terms an alphabet acts a centralised location to associate a symbol for a node and an edge along with its colour. This aides in creating consistency for the user while following Rozenberg's (1997) idea of applying graph grammar via an algebraic approach.

8. Testing

To allow for the analyse of this project a test plan will be used. This will define sections of the software that will be tested. A test plan is a base document for conducting testing activities (GeeksForGeeks, 2023), its benefits are providing a guide for large projects testing process, it allows for solutions to be identified, and enables scope management. This test plan aims to test the wider scope of the project. This is important as through development small errors might occur between systems. Additionally, it will outline and test smaller scopes of the project to assure the systems are working as expected. The approach being used for this project is to create small scope tests to evaluate the project. These can link to the objectives and finding from research. Another test will be done at the end of the project to outline the success of the framework.

8.1 Test Plan

Table 1: Test plan

Test NO.	**Test description**	**Excepted result**	**Actual result**	**Required Actions**
1	Graph can be drawn.	When program is run the graph is correctly visualised.	Graph is partly correct, there are extra edges on top of the graph.	Set parameters to meet the desired size from user inputted width and length.
2	Being able to create rule.	A blank rule can be created via unity interface.	A blank rule was able to be created.	No actions are required.
3	Specified left-hand nodes can be found.	Left hand nodes can be found.	Left-hand nodes were able to be found.	No actions are required.
4	Consistent application of right-hand side of rule.	The same rule can be applied and give the same results.	Nodes appear to be in a random order which is not desired.	Data is passed correctly but a directional application is needed.
5	Random application locations.	The same rule will keep desired form while being in a random position.	The same rule keeps desired form while being in the same position. It is always in the same orientation.	Implement a way to rotate rule's application to introduce more randomness.
6	Random orientation of rules.	Rules will be in a random orientation in one of the cardinal directions when applied.	Rule is applied in random cardinal orientations.	No actions are required.
7	Multiple right-hand rules can be run in sequence.	A list of rules is applied one after each other.	List of rules are applied one after each other.	No actions are required.
8	Multiple right-hand options.	Each right-hand has a random chance to be applied but can be.	All right-hand rules were applied eventually.	No action needed.
9	Probability of a rule being applied.	Probability of the whole rule being applied as well as each right-hand rule.	Rule and right-hand gets applied according to the probability	No action needed.

10	Alphabet can store data and be applied.	When the rule is run, the applied node's colour matches the alphabet database.	Colour and symbol in data base match the applied node's data.	No action needed.
11	Outputted graph text can make a level in Cyclic project	Outputted graph text can be used in Cyclic project (Bennett, 2023) to create a level.	Level is cramped due to positions being closer for default dungeon project. But works as expected.	Few tweaks to format dungeon to not be visually cluttered.

To be able to test to see if the algorithm can create a level in a similar way that Ludoscope does, the program had to be able to output the graph as text which can be seen in Figure 25: Outputted graph generated to match Cyclic project. The output was based of the graph seen in Figure 26. After adjusting the alphabet and rules to match the cyclic project (Bennett, 2023) and then ported over it successfully recreated the level. This can be seen in Figure 27, the key difference is the level has been rotated 90 anti-clockwise direction as well as some edges being missing due to them not being directional.

GRAPH u(x=0, y=0, tileX=0, tileY=0) Path(x=0, y=1, tileX=0, tileY=1) Path(x=0, y=2, tileX=0, tileY=2) Path(x=0, y=3, tileX=0, tileY=3) Path(x=0, y=4, tileX=0, tileY=4) Path(x=0, y=5, tileX=0, tileY=5) Path(x=0, y=6, tileX=0, tileY=6) u(x=0, y=7, tileX=0, tileY=7) Path(x=0, y=8, tileX=0, tileY=8) Path(x=0, y=9, tileX=0, tileY=9) u(x=1, y=0, tileX=1, tileY=0) Path(x=1, y=1, tileX=1, tileY=1) Path(x=1, y=2, tileX=1, tileY=2) Path(x=1, y=3, tileX=1, tileY=3) Path(x=1, y=4, tileX=1, tileY=4) Path(x=1, y=5, tileX=1, tileY=5) Path(x=1, y=6, tileX=1, tileY=6) u(x=1, y=7, tileX=1, tileY=7) Path(x=1, y=8, tileX=1, tileY=8) Path(x=1, y=9, tileX=1, tileY=9) u(x=2, y=0, tileX=2, tileY=0) Path(x=2, y=1, tileX=2, tileY=1) Path(x=2, y=2, tileX=2, tileY=2) Plain(x=2, y=3, tileX=2, tileY=3) Water(x=2, y=4, tileX=2, tileY=4) Path(x=2, y=5, tileX=2, tileY=5) Path(x=2, y=6, tileX=2, tileY=6) Path(x=2, y=7, tileX=2, tileY=7) u(x=2, y=8, tileX=2, tileY=8) u(x=2, y=9, tileX=2, tileY=9) u(x=3, y=0, tileX=3, tileY=0) Path(x=3, y=1, tileX=3, tileY=1) Path(x=3, y=2, tileX=3, tileY=2) Wood(x=3, y=3, tileX=3, tileY=3) Path(x=3, y=4, tileX=3, tileY=4) Path(x=3, y=5, tileX=3, tileY=5) Path(x=3, y=6, tileX=3, tileY=6) Path(x=3, y=7, tileX=3, tileY=7) Plain(x=3, y=8, tileX=3, tileY=8) u(x=3, y=9, tileX=3, tileY=9) u(x=4, y=0, tileX=4, tileY=0) u(x=4, y=1, tileX=4, tileY=1) u(x=4, y=2, tileX=4, tileY=2) Path(x=4, y=3, tileX=4, tileY=3) u(x=4, y=4, tileX=4, tileY=4) Water(x=4, y=5, tileX=4, tileY=5) Path(x=4, y=6, tileX=4, tileY=6) Path(x=4, y=7, tileX=4, tileY=7) Water(x=4, y=8, tileX=4, tileY=8) u(x=4, y=9, tileX=4, tileY=9) Path(x=5, y=0, tileX=5, tileY=0) Path(x=5, y=1, tileX=5, tileY=1) u(x=5, y=2, tileX=5, tileY=2) Path(x=5, y=3, tileX=5, tileY=3) Plain(x=5, y=4, tileX=5, tileY=4) Plain(x=5, y=5, tileX=5, tileY=5) Path(x=5, y=6, tileX=5, tileY=6) Water(x=5, y=7, tileX=5, tileY=7) Path(x=5, y=8, tileX=5, tileY=8) u(x=5, y=9, tileX=5, tileY=9) Path(x=6, y=0, tileX=6, tileY=0) Path(x=6, y=1, tileX=6, tileY=1) u(x=6, y=2, tileX=6, tileY=2) Path(x=6, y=3, tileX=6, tileY=3) Water(x=6, y=4, tileX=6, tileY=4) u(x=6, y=5, tileX=6, tileY=5) Path(x=6, y=6, tileX=6, tileY=6) Plain(x=6, y=7, tileX=6, tileY=7) Path(x=6, y=8, tileX=6, tileY=8) Path(x=6, y=9, tileX=6, tileY=9) u(x=7, y=0, tileX=7, tileY=0) u(x=7, y=1, tileX=7, tileY=1) u(x=7, y=2, tileX=7, tileY=2) u(x=7, y=3, tileX=7, tileY=3) Path(x=7, y=4, tileX=7, tileY=4) u(x=7, y=5, tileX=7, tileY=5) Plain(x=7, y=6, tileX=7, tileY=6) Water(x=7, y=7, tileX=7, tileY=7) Path(x=7, y=8, tileX=7, tileY=8) u(x=7, y=9, tileX=7, tileY=9) u(x=8, y=0, tileX=8, tileY=0) Path(x=8, y=1, tileX=8, tileY=1) Path(x=8, y=2, tileX=8, tileY=2) Path(x=8, y=3, tileX=8, tileY=3) Path(x=8, y=4, tileX=8, tileY=4) Path(x=8, y=5, tileX=8, tileY=5) Path(x=8, y=6, tileX=8, tileY=6) Path(x=8, y=7, tileX=8, tileY=7) Path(x=8, y=8, tileX=8, tileY=8) Plain(x=8, y=9, tileX=8, tileY=9) u(x=9, y=0, tileX=9, tileY=0) Path(x=9, y=1, tileX=9, tileY=1) Path(x=9, y=2, tileX=9, tileY=2) Water(x=9, y=3, tileX=9, tileY=3) Plain(x=9, y=4, tileX=9, tileY=4) u(x=9, y=5, tileX=9, tileY=5) u(x=9, y=6, tileX=9, tileY=6) u(x=9, y=7, tileX=9, tileY=7) Path(x=9, y=8, tileX=9, tileY=8) Water(x=9, y=9, tileX=9, tileY=9) edge(0, 1, d=false) edge(0, 10, d=false) e(1, 2, d=false) e(11, 1) edge(2, 3, d=false) e(2, 12) e(3, 4) e(13, 3, d=false) edge(4, 5, d=false) edge(4, 14, d=false) e(5, 6) e(15, 5, d=false) edge(6, 7, d=false) edge(6, 16, d=false) edge(7, 8, d=false) edge(7, 17, d=false) e(8, 9, d=false) e(18, 8) e(9, 19) edge(10, 11, d=false) edge(10, 20, d=false) edge(11, 12, d=false) edge(11, 21, d=false) edge(12, 13, d=false) edge(12, 22, d=false) e(14, 13) edge(13, 23, d=false) edge(14, 15, d=false) edge(14, 24, d=false) e(16, 15) edge(15, 25, d=false) edge(16, 17, d=false) edge(16, 26, d=false) edge(17, 18, d=false) edge(17, 27, d=false) edge(18, 19, d=false) edge(18, 28, d=false) edge(19, 29, d=false) edge(20, 21, d=false) edge(20, 30, d=false) e(21, 22, d=false) e(31, 21) edge(22, 23, d=false) e(22, 32) e(23, 24) e(33, 23) edge(24, 25, d=false) e(24, 34) e(25, 26) edge(25, 35, d=false) edge(26, 27, d=false) e(26, 36, d=false) edge(27, 28, d=false) e(37, 27) edge(28, 29, d=false) edge(28, 38, d=false) edge(29, 39, d=false) edge(30, 31, d=false) edge(30, 40, d=false) edge(31, 32, d=false) edge(31, 41, d=false) edge(32, 33, d=false) edge(32, 42, d=false) e(34, 33) edge(33, 43, d=false) edge(34, 35, d=false) edge(34, 44, d=false) e(36, 35) edge(35, 45, d=false) edge(36, 37, d=false) edge(36, 46, d=false) e(37, 38) e(47, 37) edge(38, 39, d=false) e(38, 48) edge(39, 49, d=false) edge(40, 41, d=false) edge(40, 50, d=false) edge(41, 42, d=false) edge(41, 51, d=false) edge(42, 43, d=false) edge(42, 52, d=false) edge(43, 44, d=false) e(53, 43) edge(44, 45, d=false) edge(44, 54, d=false) e(45, 46) e(55, 45) edge(46, 47, d=false) e(46, 56) e(48, 47) edge(47, 57, d=false) edge(48, 49, d=false) edge(48, 58, d=false) edge(49, 59, d=false) e(50, 51) e(60, 50, d=false) edge(51, 52, d=false) edge(51, 61, d=false) edge(52, 53, d=false) edge(52, 62, d=false) e(53, 54) e(63, 53) edge(54, 55, d=false) e(54, 64) e(56, 55) edge(55, 65, d=false) edge(56, 57, d=false) e(56, 66) e(57, 58) e(67, 57) edge(58, 59, d=false) e(58, 68) edge(59, 69, d=false) e(61, 60) edge(60, 70, d=false) edge(61, 62, d=false) edge(61, 71, d=false) edge(62, 63, d=false) edge(62, 72, d=false) e(64, 63) edge(63, 73, d=false) edge(64, 65, d=false) edge(64, 74, d=false) edge(65, 66, d=false) edge(65, 75, d=false) edge(66, 67, d=false) edge(66, 76, d=false) e(68, 67) edge(67, 77, d=false) e(68, 69) edge(68, 78, d=false) edge(69, 79, d=false) edge(70, 71, d=false) edge(70, 80, d=false) edge(71, 72, d=false) edge(71, 81, d=false) edge(72, 73, d=false) edge(72, 82, d=false) edge(73, 74, d=false) edge(73, 83, d=false) edge(74, 75, d=false) e(84, 74) edge(75, 76, d=false) edge(75, 85, d=false) e(76, 77) e(86, 76) edge(77, 78, d=false) e(77, 87) edge(78, 79, d=false) e(88, 78) edge(79, 89, d=false) edge(80, 81, d=false) edge(80, 90, d=false) edge(81, 82, d=false) e(91, 81) edge(82, 83, d=false) e(82, 92) e(83, 84) e(93, 83) edge(84, 85, d=false) e(84, 94) e(86, 85) edge(85, 95, d=false) e(87, 86) edge(86, 96, d=false) edge(87, 88, d=false) edge(87, 97, d=false) e(88, 89) e(98, 88) e(89, 99) edge(90, 91, d=false) e(92, 91, d=false) edge(92, 93, d=false) e(94, 93) edge(94, 95, d=false) edge(95, 96, d=false) edge(96, 97, d=false) edge(97, 98, d=false) e(99, 98)

Figure 25: Outputted graph generated to match Cyclic project (Bennett, 2023).

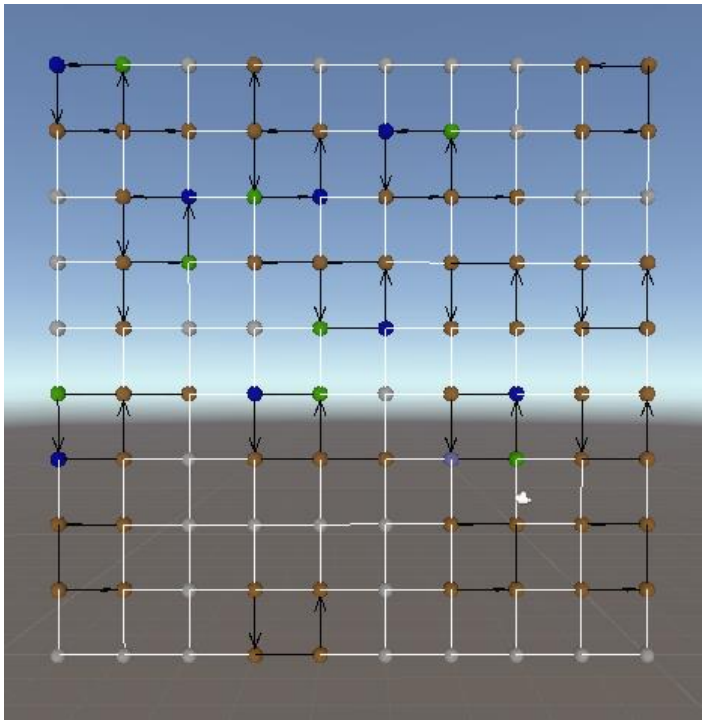


Figure 26: Screenshot of graph created by graph grammar algorithm, visual representation of Figure 25.

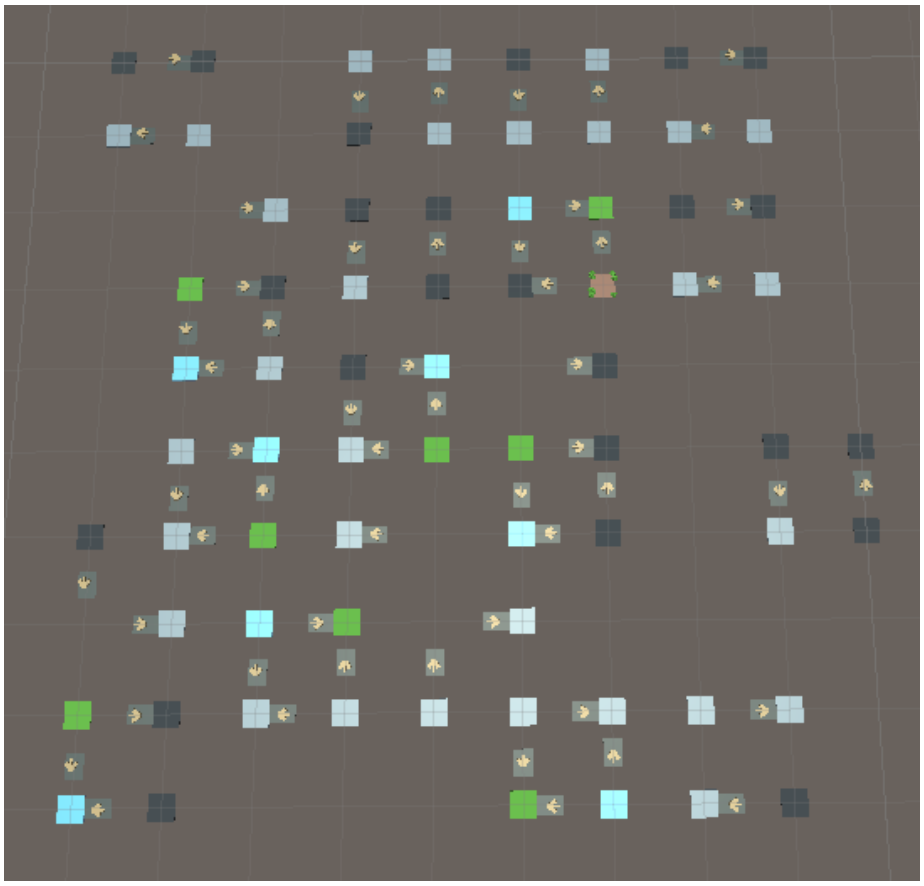


Figure 27: Generated level layout using Figure 27 text in Cyclic project (Bennett, 2023).

9. Results

Testing for this artifact was split into two sections, in development testing and a test to assess the success of the framework. The results from in development testing show that key aspects of the program work, while also outlining key changes that needed to be made. An example of this is test 4 in Table 1 which tested for consistent rule form. Due to this test, the algorithm was able to be adjusted to better suit the user's rule. This was an unexpected change but in hindsight it made sense as it allowed for the user's rule to keep its desired form allowing for more control over the randomness factor of PCG.

Test 6,8, and 9 all show that the framework is able to introduce randomness within the rules. This is an important feature as if the rule was to be applied the same way every time, then the levels it outputted would be stale. While test 9 does introduce randomness it also allows for more controlled randomness. This is due to the user being able to choose the probability of each rule to be applied. This can increase the variance of the final output.

As stated in the Background, the Cyclic project (Bennett, 2023) used Ludoscope to create the graph and used Unity to visual the output. Test 11's purpose was to see if the framework could output a similar graph that is able to be used in the Cyclic project. As seen in Figure 27, the output from the framework was successfully put into the Cyclic project's visualiser. This shows the success of the artefact as it was able to be exported across projects with only a few adjustments to be made. These adjustments were to be expected due to the specialised circumstances that the Cyclic project used. An example of this is the scale used to represent each graph, with this artifact using gizmos, the space needed to represent these clearly was small. Meanwhile the Cyclic project had to use large game objects due to limitations from Ludoscope's scale. This meant that adjusting the nodes scale on the Cyclic project side was needed for the output to be cohesive. Additionally, an unexpected issue to fix was the order that the graph that was drawn in the Cyclic project. Ludoscope's output and this artifact's output are slightly different, this caused the graph to be drawn in a 90-degree difference in rotation as mentioned earlier.

Overall, this shows that this artifact and Ludoscope can produce similar outputs. This indicates that this project was a success as it enables users to create graph grammar in one software thus making the user experience more streamlined.

10. Critical Evaluation/Conclusion

This project was inspired from a previous project mentioned called “An exploration of Cyclic procedural content generation” (Bennett, 2023), this project explored a PCG technique for graph rewriting using Ludoscope (Dormans, N.D) which was created by Dormans. This software was not user friendly due to its lack of documentation and caused restrictions on that project, thus inspiring this project’s aim of creating a graph rewriting framework within Unity.

This aim was able to be reached as seen in results as the output from the framework was able to be used in the Cyclic project. This was achieved through various steps seen in this project. Through research into the PCG technique of graph grammar, a high-level understanding of the logic was gained. To be able to successfully implement this, testing would be key. A test plan was created to analyse the project’s success. In doing so errors in code were able to be efficiently found before becoming bigger issues. This worked well with an agile methodology due to each test being able to be put onto the Kanban board.

At the end of the project, the graph grammar rewriting system was working as expected, with the rule that the user created being applied to the graph. The testing process outlined a few issues that were corrected such as the lack of randomness. These were corrected through adding orientation and probability which created more variance in levels as rules could be applied in different ways. As mentioned in results, the rewriting algorithm had to change its search and application functionality due to it being too random. These issues were caused by lack of expression from the grammar rule of direction, this meant that the algorithm picked a random cardinal direction. This resulted from creating smaller systems without evaluating the large scope of the project. To overcome this the grammar rule outlines the direction of the nodes according to each other. Combined with random orientation the user is able to describe a rule and the shape is kept while being applied in random directions and positions.

As stated, the project was able to reach its original goal. While it met its goal, Ludoscope offers a wide range of PCG techniques for the user to create a level while giving a graphical interaction. This outlines the next steps for this project. Due to this project using text to outline rules, it is not the most user friendly. This could be overcome by allowing the user to interact with a graph to create a rule, thus keeping development a consistent process. Furthermore, this framework can be expanded by giving the user more customisability options such as negative application conditions or generating a physical level.

Bibliography

Adams, D., 2002. Automatic Generation of Dungeons for Computer Games. pp. 9-13.

Annegret Habel, R. H. G. T., 1996. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae.*

aprekshamathur, N.D. *What is a Framework?.* [Online]
Available at: https://www.geeksforgeeks.org/what-is-a-framework/
[Accessed November 2023].

Bennett, W., 2023. *An exploration of Cyclic procedural content generation,* s.l.: s.n.

Bleszinki, C., 2000. The Art and Science of Level Design. *Game Developers Conference Proceedings.*

Breno Viana, S. R. d. S., 2021. Procedural Dungeon Generation: A Survey. *Journal on Interactive Systems,* pp. 83-101.

Dormans, J., 2010. *Adventures in level design: generating missions and spaces for action adventure games,* New York: Association for Computing Machinery.

Dormans, J., 2017. Cyclic Generation. In: T. X. Short, ed. *Procedural Content Generation in games.* s.l.:CRC Press, pp. 83-95.

Dormans, J., N.D. *Ludoscope.* s.l.:s.n.

Epynx, Inc., 1985. *Rogue.* [Online]
Available at: https://store.steampowered.com/app/1443430/Rogue/
[Accessed Novemeber 2023].

Franco, A. d. O. d. R., 2023. Harnessing generative grammars and genetic algorithms for immersive 2D maps. *Entertainment Computing,* Volume 47.

GeeksForGeeks, 2023. *Test plan – Software Testing.* [Online]
Available at: https://www.geeksforgeeks.org/test-plan-software-testing/
[Accessed January 2024].

Harris, J., 2011. *Analysis: The Eight Rules Of Roguelike Design.* [Online]
Available at: https://www.gamedeveloper.com/pc/analysis-the-eight-rules-of-roguelike-design
[Accessed October 2023].

Linden, v. d., 2014. Procedural generations of dungeons . In: *IEEE Transactions on Computational Intelligence and AI in Games.* s.l.:IEEE Transactions on Games, pp. 78-89.

Ludomotion, 2017. *Unexplored.* [Online]
Available at: https://store.steampowered.com/app/506870/Unexplored/
[Accessed December 2023].

NetHack DevTeam, 1978. *NetHack.* s.l.:gurr, Bartlet Software, Gandreas Software.

Nintendo EAD, N.D. *The Legend of Zelda.* s.l.:Nintendo.

Noor Shaker, J. T. M. J. N., 2016. Constructive generation methods for dungeons and levels. In: *Procedural Content Generation in Games.* s.l.:Springer, pp. 31-55.

Principle To Practice, N.D. *What is a Framework?.* [Online]
Available at: https://www.principletopractice.org/from-principle-to-practice/what-is-a-framework/
[Accessed November 2023].

Rehkopf, M., n.d.. *What is a kanban board.* [Online]
Available at: https://www.atlassian.com/agile/kanban/boards
[Accessed December 2023].

Rozenberg, G., 1997. Handbook of Graph Grammars and Computing by Graph Transformation. *Foundations. World Scientific Publishing,* Volume Vol 1.

Ryan, T., 1999. *Beginning Level Design Part 2: Rules to Design By and Parting Advice.* [Online]
Available at: https://www.gamedeveloper.com/design/beginning-level-design-part-2-rules-to-design-by-and-parting-advice
[Accessed November 2023].

Ryan, T., 1999. *Beginning Level Design, Part 1.* [Online]
Available at: https://www.gamedeveloper.com/design/beginning-level-design-part-1
[Accessed Novemeber 2023].

Saltzman, M., 2000. In: *Game design: Secrets of the sages.* s.l.:Brady/Macmillan.

The Shaggy Dev, 2022. *An introduction to graph rewriting for procedural content generation.* [Online]
Available at: https://www.youtube.com/watch?v=MxeJh2Asigg
[Accessed October 2023].

Unity Technologies, 2023. *Unity User Manual 2022.3.* [Online]
Available at: https://docs.unity3d.com/Manual/index.html
[Accessed Decemeber 2023].

Unity Technologies, N.D.. *Gizmos.* [Online]
Available at: https://docs.unity3d.com/ScriptReference/Gizmos.html
[Accessed December 2023].

Unity Technologies, N.D. *ScriptableObject.* [Online]
Available at: https://docs.unity3d.com/Manual/class-ScriptableObject.html
[Accessed December 2023].

Wrike, N.D. *What Is Agile Methodology in Project Management.* [Online]
Available at: https://www.wrike.com/project-management-guide/faq/what-is-agile-methodology-in-project-management/
[Accessed December 2023].