Geebelen Mika

Procedural boss generation using graph grammar

Graduation work 2020-2021

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT 3

INTRODUCTION 4

RESEARCH 4

1. Graph grammar 4
   - 1.1. What is graph grammar 4
   - 1.2. Why use graph grammar 4
   - 1.3. Graph grammar implementation 5
2. Boss battles 5
   - 2.1. What are good boss battles 5
   - 2.2. Why generate boss battles 6
   - 2.3. Player choice in 2D top-down roguelike bosses 6
3. How do we go from a graph to a boss battle? 6
   - 3.1. Ruleset 6
   - 3.2. The resulting graph 7
   - 3.3. Converting from graph to a boss battle 7

CASE STUDY 8

1. Introduction 8
2. Graph grammar implementation 8
   - 2.1. Basic graph 8
   - 2.2. Applying rules to the graph 8
   - 2.3. Creating a rule builder 9
   - 2.4. Graph grammar 9
3. Creating a ruleset 10
   - 3.1. Defining node types 10
   - 3.2. Rules 11
   - 3.3. Resulting graph 11
4. Behavior tree implementation 11
   - 4.1. Basic implementation 11
   - 4.2. Behavior tree node types 12
5. Graph to boss converter 13
   - 5.1. Graph to boss 13
   - 5.2. Graph to boss body 13
   - 5.3. Graph to boss behavior 13

6. A boss editor ... 14
6.1. Saving and loading the boss ... 14
6.2. Changing abilities, conditions, and movement behavior ... 14
7. Fighting the boss ... 14
7.1. Gameplay ... 14
7.2. Player character ... 14
7.3. Boss abilities ... 15
7.4. Boss movement behaviors ... 15
7.5. Boss phase conditions ... 15
RESULTS & DISCUSSION ... 16
1. The framework ... 16
2. Procedurally generated bosses versus handcrafted bosses ... 16
2.1. Examples ... 17
3. Limitations ... 18
4. Graph grammar ... 18
FUTURE WORK ... 19
1. A designer tool ... 19
2. More easily expandable framework ... 19
3. Creature generator ... 19
CONCLUSION ... 20
1. The framework ... 20
2. Graph grammar for procedural boss generation ... 20
BIBLIOGRAPHY ... 21
APPENDICES ... 22
Appendix A ... 22
Appendix B ... 24

ABSTRACT

Most procedurally generated bosses in games results in very simple fights. To evaluate the use of graph grammars as a technique to generate procedural boss fights, we will create a framework that will allow us to generate these, with an added level of complexity. We will achieve this by creating this framework in the unity game engine. We will implement graph grammars to generate a representation of what our bosses will do and look like. Graph grammars will also allow us to easily change our generation by adding and removing rules. We will then convert this representation to an actual body for our boss and a behavior. This behavior will be represented by a behavior tree.

We want this framework to allow users to save and edit their bosses. This will allow them to use these generated bosses in their own projects. We want the framework to be a base and allow for as much basic boss behavior as possible, while keeping our framework flexible to allow for easy expansion. This will allow users to expand the framework to allow for their bosses to have more unique events and effects.

Overall, the way I created the framework might not be the most efficient as the structure is rather ridged for expansion. However, it acts as great demo of what is possible. Taking a closer look at the technique itself shows us great potential for the procedural generation of bosses. We can have a lot of control over what and how they get generated.

INTRODUCTION

Graph grammar is a procedural technique used to generate node-based graphs according to predefined rules. Using this technique we will attempt to generate video game boss battles.

The question we will seek an answer for is:
Can we generate bosses using graph grammar that will be similar to handcrafted bosses? We also want as much variation as possible in the generated fights. Each fight should have a learning curve and test the mechanical skill of the player.

Creating a comprehensive ruleset for the graph grammar could give us node graphs that, when converted, give us a similar behavior to handcrafted bosses. This node graph would have to return everything, from the boss' movement behavior to every ability, and the phases the boss can go through. This would then have to be converted to an actual body, and behavior for our boss, that can be used in a game environment.

In this paper, we will investigate if graph grammar gives us sufficient control to achieve this complexity, what bosses we can generate, and how these compare to handcrafted bosses.

RESEARCH

1. GRAPH GRAMMAR

1.1. WHAT IS GRAPH GRAMMAR

Graph grammar is a technique where you continuously apply transformations on a node graph to create larger graphs. A graph grammar is specified by a start graph and a set of production rules. e.g. gl → gr where gl is the original subgraph and gr is the replacement subgraph[1]. With graph grammar, we can apply the same ruleset to the same starting graph and still get diverse outcomes.

Figure 1: rule where gl changes to gr

The graph created by the ruleset is mostly then used as an abstract representation of something else. It is then necessary to convert this graph into something more tangible. E.g. To use this graph you will have to convert it specifically for your application. Some use cases could be the generation of robot bodies[2] or building facades[3]. The applications for this are very diverse.

1.2. WHY USE GRAPH GRAMMAR

Procedural boss generation mostly results in a predictable and linear fight for the player. Using a more complex system like graph grammar allows for more variation in how the player has to defeat these bosses, by generating bosses with a certain learning curve attached. This ensures that players may have to repeat the fight multiple times before they fully understand the boss's mechanics.

Graph grammar allows for a lot of control in how the resulting graph will look. This is required to be able to convert each graph automatically into an AI behavior for each boss. The structure is a graph that also allows for branching which will be used to represent player decisions in each fight.

1.2.1. CONTROL

Graph grammar has many ways for the user to achieve a resulting graph in an expected shape. The first and most obvious are the custom rules the user provides. These rules can steer the graph to an "end" when no more rules are applicable, or you could have no end-node types which would allow for an endless generation. You can also change the likelihood of each rule which allows for more controlled results. The likelihood could also be based on some type of input before the generation starts or change based on how often other rules are applied.

1.2.2. BRANCHING

The node graph can be split up into many different branches. These branches can mean anything but in this papers' context, these will be player decisions. Any branch is a path of the boss fight that the player can take. This allows for different strategies in each fight. Taking a fight again might result in a completely different experience because the player took different decisions.

1.3. GRAPH GRAMMAR IMPLEMENTATION

Let's first talk about the difference between graph grammar and graph rewriting. Graph grammar is the procedural technique that uses rules to generate graphs. Graph rewriting is the technique that concerns itself with detecting and replacing subgraphs. Graph grammar transformations are applied by using graph rewriting.

By using this technique in the Unity game engine, the outcome should be reusable in future game projects. The demo should be playable for testing. Graph rewriting will be coded in C#.

This leaves two possibilities, code graph rewriting from scratch, or using an existing library.

1.3.1. C# LIBRARIES

There are a couple of C# libraries that allow for graph rewriting. Using a library will ensure code optimization but will also bring a lot of extra features that are not necessary for this application. One of these libraries is GrGen.net[4], which transforms intuitive and expressive rewrite rule specifications into highly efficient .NET code.

1.3.2. C# LIBRARY VERSUS STARTING FROM SCRATCH

A fresh implementation will take longer, but the result should be tailored more to this application. Library optimizations are not required since speed is not an important factor. Not using predefined libraries will allow us to fully understand graph grammar without losing any sort of control over the implementation.

2. BOSS BATTLES

2.1. WHAT ARE GOOD BOSS BATTLES

Boss battles are the most challenging parts of a game. They require you to have mastered the mechanics of your game, acquired the necessary gear, and planned ahead on how to fight them. Most of the time they have a story behind them as well.

In the early stages of a game, bosses will test the player on the effective use of specific mechanics. Later on, bosses require more planning and trial and error, to figure out their move sets and their specific transition phases. Sometimes, more gear and consumables will be needed to defeat an advanced boss.

Good examples of this are the Pokemon games. Your team is required to be at a certain level to challenge a gym battle. This is your “gear check”. Gyms have certain types attached to them. Using Pokemon with a type advantage shows your mechanical skill of the game. Additionally, buying potions or catching new Pokemon can be a good strategy to prepare for a gym.

2.2. WHY GENERATE BOSS BATTLES

Generating bosses is a unique challenge. You don’t want it to be a simple fight where the boss’ abilities and stats were simply mapped against the players’ capabilities. Fights would feel very linear and would not make sense, and it would leave out any type of smart decision-making.

The bosses generated with graph grammars will have abilities that fit together, they will be non-linear by adding branching paths for the player based on their decisions.

An example of a branching path could be that there are two boss enemies and killing one results in the second going into a raging state making it more dangerous. This forces the player to decide which boss to kill first.

2.3. PLAYER CHOICE IN 2D TOP-DOWN ROGUELIKE BOSSES

This paper describes how to generate bosses that will fit in a 2d top-down roguelike, similar to The Binding of Isaac, Enter the Gungeon, …. In this section, we will analyze the boss’s abilities that allow for more player choice.

A common way to allow for more player strategy is by using a multi-boss. This boss consists of two or more main boss units that each have a separate health bar. Killing one of them results in a buff, of some kind, for the remaining boss units. Buffs can range from, increasing move and attack speed, to unlocking new abilities. Allowing the player to decide allows for the player to strategize the enrage phases.

Modular bosses, where different parts can be destroyed also allow for additional player choice. When a boss is comprised of a main part and one or more subcomponents, each with their own abilities, the player will have to decide in which order to dismantle the boss.

3. HOW DO WE GO FROM A GRAPH TO A BOSS BATTLE?

3.1. RULESET

To generate the node graph, a custom ruleset will have to be defined. It should allow for as many boss types as possible, while also ensuring that the generated bosses are diversified. Even if they are of a similar type they should never feel the same. A boss-type could be made of a main body and two subcomponents. Many different bosses can be generated but there should be enough variation to keep them apart. For instance, different abilities do a great job at this but if the foundation stays the same, it would feel like a simple reskin of a previous fight.

There are many ways to get these fights to be different. For example, the secondary pieces could be optional on some bosses, the secondary pieces could be linked with some kind of combo move making the destruction of one cripple the other, the secondary pieces could have multiple phases, each piece could have a different amount of phases, …

By using a point system, rules will have a varying cost. This allows for control of boss complexity. Applying costs to rules allows for similar strength bosses.

3.2. THE RESULTING GRAPH

All nodes in the resulting graph will either represent a condition, an action, a boss component, a stat increase, or a behavior. A condition could be, an end-of-phase node that gets triggered when the boss falls below a certain health threshold. Actions could be, boss abilities, or traps in the room. A boss component is an actual piece of the boss. This will represent what the player can damage and where the boss' abilities originate from. A stat increase will just be a multiplier to increase the base stats of the boss. A behavior would be able to represent the movement system the boss will use or special mechanics.

3.3. CONVERTING FROM GRAPH TO A BOSS BATTLE

This representation will be converted to a brain and a body for our boss. The brain will be the behavior tree and the body will mainly be collidable sprites.

3.3.1. A BODY FOR THE BOSS

Each major component, including main, secondary, and room effect pieces needs a representation in the world. The main and secondary pieces will be represented with a sprite and a collider that will detect if the part is being damaged or if they touch the player and need to damage him. The room effect needs to be pre-placed in the arena as well so that they can be used anytime by the player or boss depending on the effect.

3.3.2. A BEHAVIOR TREE

The major nodes here are the conditions, boss components, and behavior nodes. The behavior tree will start with a major sequence that will run the default behavior nodes and have a branch for each boss component to check if it is still alive. A behavior node could be something like movement. In the branches where major boss components are checked, we will be able to follow the graph closely. All nodes under the major component are conditions and actions. These will convert easily to behavior tree nodes.

CASE STUDY

1. INTRODUCTION

To assess the viability of procedural bosses it is required to first write a framework that allows us to visualize, create, and test these bosses. In this case study, all essential implementation requirements for the framework and the setup of the test environment will be explained.

2. GRAPH GRAMMAR IMPLEMENTATION

2.1. BASIC GRAPH

Before we can start applying rules to our graph, we first need to create the graph itself. We create a graph class holding a list of all nodes and connections in the graph. The nodes hold their id and type while the connections hold the two nodes they connect.

The graph class holds functions that allow for the addition of new nodes, the connection of existing nodes, and a function that returns the connections of a node in the graph.

2.1.1. NODES

The node class, as mentioned, holds an id and a type. We will use the type to define what the node means to the conversion (5. graph to boss converter). The type will be used to detect the relevant node during subgraph detection. Meanwhile, the id will be used to keep track of the node in the subgraph detection algorithm.

2.1.2. VISUALISATION

To be able to see how the graph looks, we create a simple visualization class. This class goes over the chosen graph's nodes and connections, and based on the node id gives them a place in the world. (Appendix A)

2.2. APPLYING RULES TO THE GRAPH

To be able to apply rules to our graph, we need to be able to detect and replace a subgraph. Our rules will be defined as a set of two graphs, where one is the subgraph, we have to find which we will call the source graph. The other is the graph we use to replace the source graph which we will name the target graph.

To find our subgraphs we normally need to implement a subgraph detection algorithm. But in our case, all our rules end up being chains of nodes instead of complex webs. This allows us to simplify the algorithm needed for our application.

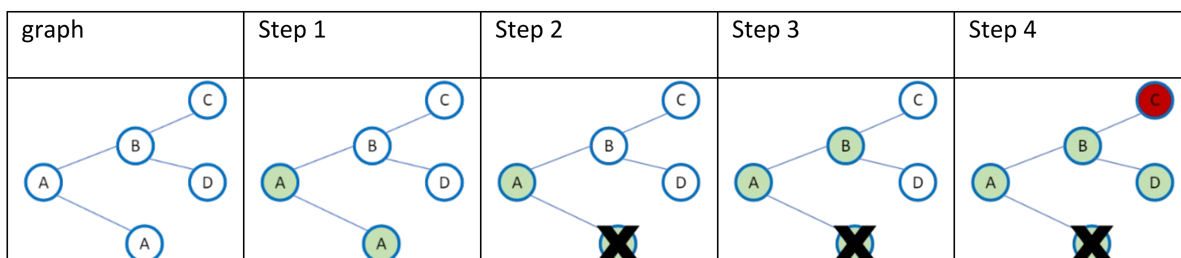

Figure 2: Chain finding algorithm

To explain the node chain finding algorithm we can follow the visuals. Firstly, we will try and look for rule ABD. We will start by finding all possible start nodes. Then we will check if the start node has the required connections. If not we go to the next possible start node. If there is a required connection, we check if that node has it's required connections, we can do this recursively. The moment a node does not have one of the connections it needs we step back to the previous node and check if it has another connection to replace the failed one. If a node has all its required connections it returns success to the previous node. If the start node receives success, we have found our chain.

Now that we found our source subgraph, we can replace it with our target graph. We replace the types of the existing nodes according to the connections we defined in the resulting graph. If a connection is missing, we go over the subgraph nodes to see if the missing connection could be made with any of these nodes. If not, new nodes are added to the graph. The replacement algorithm does not remove any nodes. Therefore, it disallows rules with more start than end-nodes. This is not required for this application.

2.3. CREATING A RULE BUILDER

To allow us to add rules, we need to create a rule builder. To do this, we create a rule builder class that allows for node creation, node connection, setting node types. The rule builder also handles graph swapping between target and resulting graph. Finally, it will save these rules to a JSON file.

To save the rules to a JSON file we need a JSON convertible rule class. This class should hold a list of nodes and connections for the target graph, and a list of nodes and connections for our resulting graph. That means our nodes and connections should also be convertible to JSON. A JSON convertible class in Unity c# requires all members to be serializable. Since the node and connection classes aren't serializable due to the use of private variables, we can create variant classes that hold similar but serializable data. These classes will be used to save the rule to JSON, and to compare the rule graphs to the real graph.

Once we can save the rules to a JSON file, we can easily convert them back using built-in Unity c# commands.

2.4. GRAPH GRAMMAR

We can combine all previous sections to create our first graph grammar. We start by creating a graph grammar class that holds a list of rules and a graph. To kick off the generation we add one node to the graph and make sure at least one rule replaces this node with something else. (Appendix A)

3. CREATING A RULESET

3.1. DEFINING NODE TYPES

A boss has a lot of different elements. We need to define what elements we will generate and in what way. We start by splitting up the boss and defining what nodes belong in each section and what their purpose is.

Core boss fight mechanics	**Node representation**	**Node meaning**
The creature the boss represents	BossCore/ BossCoreDynamic	Defining a main section of the boss in the graph will be used to attach all nodes that define this piece's behavior. When generating a body, we need to know the number of pieces and how they link together.
	SecundaryCore/ SecundaryCoreDynamic	Similarly to the main piece, we can use it to define how we link the body together. The additional nodes that get attached will define this piece's behavior.
The boss's behavior and interaction with its surroundings	Movement	A movement node would rely on its connection to either a main or secondary piece to define that it can move. It doesn't define how it moves.
	Ability	This node would have to rely on being connected to phase nodes to know when the ability is available. It will not define what the ability does.
	PhaseStart	When a condition is triggered, we can unlock the use of the ability nodes that are connected to PhaseStart associated with the condition.
	Death	The connected piece is dead. This will be at the end of the PhaseStart ability chain.
Non-boss related nodes for structure	Start	All Boss core pieces will be attached to this node. This allows the converter to find all core nodes easily.

3.2. RULES

Now that we have defined all nodes we can define our generation rules.

Rule	Result	Goal of each rule
Start	Start – BossCore – PhaseStart – Ability – Death	This rule adds a main core to our graph. The core starts with a simple behavior, with a single ability in its default phase.
BossCore	BossCore Dynamic – Movement	This rule changes the boss from being static to having a movement behavior.
BossCore	BossCore – Secundary Core – PhaseStart – Ability – Death	This rule adds a secondary component to the boss, with its own simple behavior with a single ability in its default phase.
BossCore Dynamic	BossCore Dynamic – Secundary Core – PhaseStart – Ability – Death	This is the same rule as the previous one. We need this rule a second time for when the boss is dynamic.
Secundary Core	Secundary CoreDynamic – Movement	This rule changes the secondary component from being static to having a movement behavior.
Ability – Death	Ability – Ability – Death	This rule increases the number of abilities in the final phase.

Figure 3: Node rules

3.3. RESULTING GRAPH

Now that we can generate our graphs we need to convert them to our bosses. We will do this by stepping through all the nodes of the graph and converting them, either to a piece of the boss, or a node in the behavior tree (5. graph to boss converter). The outcome should already tell us a lot about how the boss will function. The structure of the boss is made clear with the boss core and secondary core nodes. The boss cores act independently from one another, while the secondary cores act based on their connection to a main core. The other nodes: ability, movement, … will represent the behavior of the connected core. (Appendix A)

4. BEHAVIOR TREE IMPLEMENTATION

4.1. BASIC IMPLEMENTATION

We now have a graph that defines our boss, but we need some way to create a behavior for it. We will do this by making a behavior tree that will control our boss. First, we need to implement a behavior tree system.

We start by creating a blackboard class, a behavior tree class, and a node interface. The blackboard is a class that will hold a dictionary with string-type keys and object-type values. Using the object type allows us to put any Unity-defined class in the dictionary. Allowing for gameobject, rigidbodies, int, string, …. We add some functions that allow for easier use of this blackboard including; adding a value, removing a value, changing a value, getting the type of a value, getting the value, and checking the existence of the value.

The behavior tree class holds a list of the node interface class objects. This interface implements one function that is called “ExecuteNode”. This function returns an object containing a node and the state of said node. This state defines if a node succeeded, failed, or is running. The use of these states will become clear in the next section (4.2. behavior tree node types). The behavior tree class has functions that allow for the choosing of a blackboard and to start the execution of the tree. When the tree is “Executing” we go over the list of node interface objects and call the “ExecuteNode” function on all of them. This makes our behavior tree class act as the root node of the tree.

4.2. BEHAVIOR TREE NODE TYPES

All different node classes need to inherit from the node interface. This enforces that they all implement the “ExecuteNode” function. Here you can see the node states being used, and what effect they have on the node that calls them.

Node type	**Function**
Selector	The selector holds a list of nodes. When the selector is called to be executed, it will go over its list and call “execute” on all its nodes. It goes over the entire list until one returns “success” or they all fail. If “success” is returned, this node will also return “success”. If all nodes failed the failed state is returned. If a node returns “running”, the node is saved and it will also return “running”.
Sequence	The sequence holds a list of nodes. When the sequence is called to be executed, it will execute all the nodes in its list until one returns “failed”. If any node failed, this node also fails. If all nodes succeeded, this node returns “success”. If a node returns “running” it is saved and this node also returns “running”.
RootNode	The RootNode holds a list of nodes. When the RootNode is called to be executed, it will go over this list and call “execute” on all the nodes. It will execute every node unless a node returns “running”. In which case the node is saved and the RootNode stops and also returns “running”. If no running state was returned this node always succeeds.
AbilityNode	This node stores an ability and when the node gets executed it tries to use this ability. If the ability is on cooldown, the AbilityNode failed. Otherwise, it will return “running” if the ability takes a while to cast or return “success” when the ability got executed.
MovementNode	This node stores a movement class. When this node gets executed it calls the “move” function of the stored class and always returns “success”.
ConditionNode	This node holds a condition. When this node gets executed, it uses the blackboard information to see if the condition is valid or not. If the condition is valid it returns “success” otherwise it returns “failed”.

The ability, movement, and condition nodes each hold a class based on their respective type. These classes are more specific to the actual fights, so these will be explained in-depth in section (6. gameplay).

5. GRAPH TO BOSS CONVERTER

5.1. GRAPH TO BOSS

We have a way to create our graph and a way to create a behavior, but these systems don't interact with each other yet. There is also no body that moves or uses abilities. Therefore we need to create a converter that changes our graph into two things; a body for our boss, and a behavior for that boss to execute.

We also want the bosses to stay similar in power level. So instead of using random abilities, conditions, room events, and movement, we will use a point system. Each component will cost a certain amount of points and each boss will have a number of points to spend on his abilities.

We also need the boss to have access to the player character, to be able to chase and target it with its abilities. This is the first object we store in the blackboard for our behavior tree.

5.2. GRAPH TO BOSS BODY

To create a body, we will have to traverse the graph and keep track of two node types; the BossCore, and SecondaryCore nodes. These are the nodes that represent the structure of the boss. All main nodes are attached to the start node and all secondary nodes are always attached to their main piece. This way, we can easily find the nodes. We just need to store them properly. We can store them in a list that holds the nodes and their ids. Upon finding the BossCore, we give it an id and store it in this list. Then we find all its secondary pieces and store them in the same list with the same id. Then we go to the next BossCore and give it a different id. This process is then repeated.

Now we have a list of pieces that all keep track of what major node they are attached to. To turn this into a body, we create a couple of prefabs in Unity to represent a boss. We represent the core piece with a health bar and a sprite of a red circle. The secondary piece we represent as a health bar and sprite of a grey circle. They each have a collider for detecting collisions with the player and to detect incoming bullets. While creating these bodies we can also store them in the blackboard for our behavior tree. We will later use them to apply movement or as the origin point for abilities.

In this step, we can also define the amount of health each piece of the boss should have. We will use a random value that uses between 10 and 50% of the point pool that the boss got assigned to give all the pieces health.

5.3. GRAPH TO BOSS BEHAVIOR

Now that we have created a shell for our boss, we need to create a behavior for it. To do this, we will have to step through the entire graph and while we traverse it, translate each node to a node for our behavior tree.

After going through the graph, we are left with a behavior tree that holds the structure of what the boss should do. We still need to define what the abilities are, what the condition in each phase means, and what the movement node's actual movement is. To keep the bosses at a similar power level, we will use the point system here to make the stronger abilities and movement behaviors more expensive compared to the weaker ones. We will select a random ability for each slot and spend the points if they are available. If we run out of points, we will provide very weak abilities with negative cost to make sure there is always something available to select. (Appendix B)

6. A BOSS EDITOR

6.1. SAVING AND LOADING THE BOSS

When we created our graph class we made a save and load system for it. For the bosses themselves, we will do the same. We will save the boss in three separate JSON files; one for the structure, one for the behavior tree, and one that holds all the info about the abilities, conditions, and movement behaviors. When a boss gets generated, all these things get created anyway. The only thing we need to do with them is make them JSON-convertible in c#-unity.

We can achieve this by making classes that just hold similar data to the actual used classes. They have to be fully public and using only basic c# classes. Because the behavior tree holds a list of behavior tree nodes, that on its turn can hold more behavior tree nodes, we can't convert this to a JSON-convertible class. Therefore, we have to make our own converter for this occasion. Storing the structure, abilities, conditions, and movement behaviors, on the other hand, can all be easily done with the build-in converter.

Once we can store everything in JSON, we can easily convert it back to our original data and recreate the saved boss.

This saving and loading system is here to allow the user to create their bosses with the framework. They can then take the saved data into their application to use the bosses there. Only a limited part of the framework will be required to reuse the bosses in other applications.

6.2. CHANGING ABILITIES, CONDITIONS, AND MOVEMENT BEHAVIOR

To make the framework more usable for a designer, we added the ability to edit the resulting boss in a couple of ways; we can change the abilities the boss has, the movement behavior for each piece, and the conditions of each phase.

This is easily achieved because the nodes in the behavior tree hold their respective abilities, conditions, and movement behaviors. We just have to create a small editor window, link it up to the behavior tree node and add a dropdown with the other possibilities for that node. All the possibilities are stored in separate lists in the boss builder.

7. FIGHTING THE BOSS

7.1. GAMEPLAY

To be able to test what we created, we will need to create a small testing arena, a player character, and a batch of abilities for the boss. The player should be able to move and shoot. We can make a testing arena out of static sprite objects. The boss should also be able to move and have multiple sets of different abilities.

7.2. PLAYER CHARACTER

We need to create a player character to be able to fight the boss. We can keep it relatively simple. We need to be able to shoot a projectile at the boss and move. We can achieve this by creating a prefab that holds a sprite with a collider and a "rigidbody". Then we create a player movement class that sets the velocity on our "rigidbody" to a speed value multiplied by the direction we want to head. This direction is based on keyboard input.

We also attach a gun to the player. This gun is also a prefab that holds a sprite and a transform location of where to spawn a bullet and where it has to go. We attach a script to this object that spawns a bullet upon a mouse press.

Setting up a camera that follows the player is quite useful. We can make a script for this or we can just attach the camera to the player in the hierarchy.

7.3. BOSS ABILITIES

Our ability nodes in our behavior tree need to be filled with actual abilities that attack the player or move the boss. To start, we can create a “BaseAbility” class that inherits from “ScriptableObject”. This class will keep track of the ability cooldown and provide functions that the derived classes need to implement. We need a function that triggers the ability and one that creates the ability. Making the base class inherit from “ScriptableObject” allows us to create a lot of variants on the derived abilities.

A simple ability to start with would be a “Shoot” ability. By default, this shoots one bullet to the player. Making the number of bullets a public parameter together with an angle allows us to create a lot of variants out of this one ability. We store them as “ScriptableObject”, each with different parameters, and use these objects in the list of random abilities the boss can use.

Each ability will need a different setup. Some abilities will need access to the player, others will need access to the transform of the boss, and so on. To solve this, we give each ability a function that returns the list of its personal requirements that the converter class will then provide blackboard access to. In the case of “Shoot”, we need to give it an origin and a target. We will then give this ability access to the transform of the boss this ability belongs to, and the player transform.

7.4. BOSS MOVEMENT BEHAVIORS

The movement finds itself in a similar situation as the abilities. We can again make a base class, derive from it, and create different movement systems. Again, we make these as simple as possible with as many parameters as possible to get a lot of use out of a single behavior.

We also use the same technique, where we make the list of requirements for this movement available, so we only give it blackboard access to the things it needs.

7.5. BOSS PHASE CONDITIONS

The conditions need a bit more work than the movements and abilities. Here, the order of when each condition in the chain can get triggered is very important. We don’t want to end with a situation where a condition relies on a piece being dead, and that piece also relies on the other piece being dead. Therefore, never triggering the phases that those conditions block. By stepping through the graph, node by node, we got the order of the conditions. The only thing we need to add there is that they remember which piece the condition belongs to. Doing this makes us able to assign conditions, piece by piece. This then allows us to check, if we want to assign the condition of another piece being dead if the other piece doesn’t already have this check for the current piece.

After getting the order and piece ownership in the mix, we can follow the same logic as the abilities and the movement. Start by making a base class, create a derived class, and make scriptable object versions of it. (Appendix B)

RESULTS & DISCUSSION

1. THE FRAMEWORK

Our framework allows for the creation of procedurally generated bosses based on a basic ruleset. We can save and load these bosses. We can also change their abilities, conditions, and movement behavior using our editor. We can also modify our graph generator to make more complex bosses by providing more or less points to the generator. This is similar to our boss builder in that we can also use a point score that gets used to define max health, movement behaviors, and abilities. Lower point values weaken the resulting bosses while higher point values make the bosses stronger.

The resulting bosses can consist of multiple pieces, each with their own phases and abilities. Some pieces would be able to move, others could be stationary, and secondary pieces can be attached to these.

The abilities the boss can have can be broken down into four categories; attacking abilities, healing abilities, movement abilities, and combo abilities. An attacking ability could spawn projectiles that the player must dodge. If he fails to dodge he will take damage. A healing ability could heal a part of the boss. This gets interesting when the boss has conditions based on heath thresholds. The boss could then heal itself back to a previous phase, if the player isn't careful. A movement ability could dash the boss a certain distance making its movement more unpredictable. A combo ability relies on two different abilities; one to start the combo and one to finish the combo. The start of the combo could spawn an object, for example a tar puddle that slows the player if they walk in it. The combo finisher can then trigger the object, for example throw a fire projectile at the tar puddle to make a fire zone. Giving the player the option to destroy the tar puddle before the boss can light it on fire, gives the player a strategic option. The boss would then not be able to cast its fire ability and instead uses a replacement ability. The replacement ability can be any non-combo ability.

The boss also has conditions for when to go to the next phase. There are basic conditions that just consider the health of the boss piece for which to trigger the next phase. And there is the option for a more complex condition, for example taking into account whether another component is alive or not.

2. PROCEDURALLY GENERATED BOSSES VERSUS HANDCRAFTED BOSSES

If you need multiple similar bosses, our framework could be useful if it is setup with a cohesive ability pool. The bosses would all feel similarly themed to each other while being vastly different in fight structure. You could allow the generator to create these bosses at runtime in your game, or you could keep them in the framework and edit the generated bosses for later use. The framework helps the designer by generating outlandish structures and combinations therefore helping in the creative process.

If you need complex bosses you should create your own bosses. The framework could be of use in helping with being creative in the structure of said boss. You could use the graph generation to give ideas on how to structure the boss and what behavior to give it. But the converter is still limited to abilities, conditions, and movement behaviors. If you want anything more complex you would have to expand the framework to allow for more unique features.

Looking at 2D roguelikes like "The Binding of Isaac" and "Enter the Gungeon", our framework would be able to generate most of the bosses in those games. The early game bosses start out very simple and would be the easiest to recreate with our framework. The limitations for our framework start arising when we want to generate bosses

for the later stages of such game. These end-game bosses have a lot more complexity and unique mechanics that would have to be added to the framework.

2.1. EXAMPLES

Here I'll cover one simple boss from "Enter the Gungeon" and how the framework can generate it. We will also take a look at a more complex boss and the added challenges on how to generate it.

2.1.1. THE BULLET KING

The "Bullet King" [5] is one of the bosses that can be found on the first floor of the dungeon and has access to these abilities:

- Spinning and firing continuous volleys of bullets, each wave alternating in direction. This is followed by a tight-knit circle of bullets. This leaves Bullet King stunned for a moment.
- Firing a circular volley of spiral-shaped, accelerating bullets that curve slightly.
- Firing a spread of three bullets at the player, with a second 3 bullet spread directly behind the first spread.
- Firing a large bullet upwards which bursts into 8 smaller bullets, each of which bursts into 8 more bullets.
- Firing twelve lines of four bullets in all directions.
- Throwing a flask that ignites the ground.

The "Bullet King" will continuously follow the player.

This boss does not transition to any other phases.

If we wanted to create a carbon copy of the bullet king we should just make everything exactly as is without the framework. But if want X amount of bosses that are all unique with similar skill sets to the bullet king then our framework will allow for this kind of variety. We can compact the abilities into base versions that allow for more diversity. For example: we have 3 abilities that shoot volleys of bullets in different shapes with different speed and with different amounts. We can create an ability that has the parameters exposed to allow for all of these abilities. We could reuse this to create more similar, but unique abilities. Looking at movement, we can create a simple chase player with as many parameters exposed as possible. If we want to change speed, we could add add a slight wander to the chase, or we could make the boss pause every x seconds,…. Even though the original bullet king doesn't have any phases, we could simply add a phase shift based on health. Adding all of this will allow for hundreds of similar but diverse fights.

The original bullet king doesn't have secondary pieces. If this is desired, we could remove the rule that generates these pieces. If we do allow for secondary pieces, the amount of variations will increase even more.

2.1.2. THE KILL PILLARS

The "Kill Pillars" [5] is one of the bosses that can be found on the fourth floor of the dungeon and has access to these abilities:

- A spinning volley of bullets that fills the arena. The player must stay close to the Kill Pillars and circle the boss to avoid taking damage. During this attack, the Kill Pillars will jump, sending out rings of bullets, which must be dodge rolled over. Both the rings and spinning arcs are jammed, dealing a full heart of damage.
- Jumping towards the player as a group, sending out bullets in all directions each time they land.
- Staying in the center of the arena while jumping more rapidly in succession, filling the screen with bullets.
- Lining up vertically on the left side of the screen and jumping to the right side, sending out bullets each time they land.

- Final pillar ability: It will continually jump and attempt to crush the player. When hitting the ground, it will release a large ring of bullets before attempting again to crush the player.

“Kill Pillars” all move in sync. If one moves up the others will mirror that move for their quadrant of the room.

Similarly, to the bullet king we can make abilities that are parameterized for more variation on the original abilities. Movement is handled in the same way. The final phase of the last pillar still has to be added. We can achieve this effect by adding a condition that checks if it is the final piece remaining.

The framework will generate a random number of main bosses and they won’t necessarily have a second phase. To have a similar effect to the pillars, we could change the rule that adds main bosses to always start with an extra phase. Spawned bosses would then have that same final condition. Adding extra phases would then become obsolete.

Creating a setup like this will always spawn a random number of bosses where the one you leave for last will have a crazy end phase. We still have a lot of variation for all generated bosses, including how many abilities they each have in their first and second phase. Some may have secondary pieces that will also add a layer to the fight.

Adding the enrage is a lot more complex to setup than the previous mechanics. This can be done by expanding the framework. We need a node to indicate this enrage effect, a rule that adds this node to the generation, and expand our converter to look for this node. When we find it, we also have to add a node to the behavior tree to activate the effect.

Adding things like this to framework once, allows you to keep using this mechanic for all your future bosses.

3. LIMITATIONS

The framework generates a behavior and a structure for the bosses. To use this in an actual game, we would only need an animated character. Where the abilities originate from will have to be redefined, since the framework assumes abilities are spawned in the center.

The framework is also limited to abilities, conditions, and movement behaviors. Nothing happens in a phase transition. The room is only used to keep the boss and player inside. Expanding the converter to accept new node types is possible but rather complex.

4. GRAPH GRAMMAR

Currently, graph grammar is used to create graphs that get converted into bosses. The graph grammar allows us to get a diverse range of similar complexity fights based on the point value given to the generator.

It also allows for easy expansion of the framework by allowing us to create new node types and use those types in new rules.

FUTURE WORK

1. A DESIGNER TOOL

This framework allows for the generation of bosses. We discussed the limitations to create a model with animations after we generated our perfect boss.

Instead of doing it like that, we could look into creating a creature first and then assigning parts of the creature as possible main and side pieces. Then we can generate our abilities and phases over a pre-existing model. When the graph grammar starts generating the main and side piece rules, we can remove those rules after we reached our cap.

In the example on the right we have taken a bird and just assigned 1 main piece (the body and the legs) and 3 secondary pieces (the head and each wing).

If the model has animation made for it we could look into randomly adding those to our abilities. We wouldn't know in the generation if they make sense, but at least we can add them to an editor so the designer can choose the fitting ones for each ability.

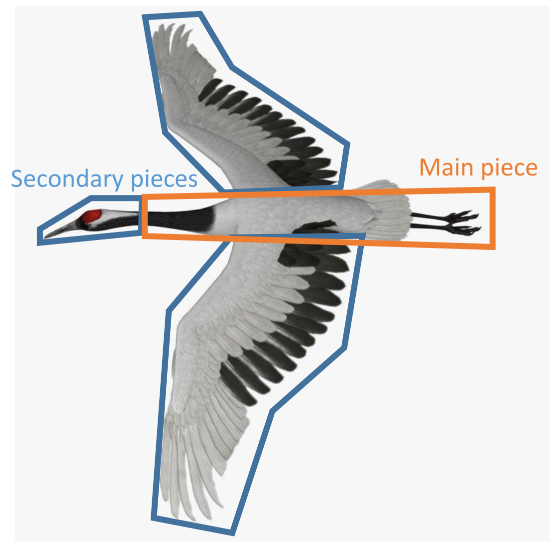

Figure 4: Split up model

Expanding our framework to a tool like this could speed up the workflow for the creation of bosses and enemies in games.

2. MORE EASILY EXPANDABLE FRAMEWORK

Currently, the framework allows for conditions, abilities, and movement behaviors. We can easily add more of each to the framework. But for more unique events and effects, we need to change/add a good amount of code to allow for this. With the acquired knowledge of the creation of this framework. We can look into recreating the framework to be less rigid for expansion.

Instead of having the user go into the framework and manually add a converter for their desired node type. We could let the user base their class on a general converter class and make it possible to just assign this. In general, we want the user to add to the framework and not have to worry about changing things in the framework if it is not necessary. Adding effects and events and letting the converter explore graph branches in different ways should be easy. Similarly, we want the user to be able to define new components for the bosses. It should be easy to add them to the boss body.

If we could upgrade the framework to allow for all these things, we can see it being used as a tool that helps the designer iterate over their boss designs.

3. CREATURE GENERATOR

We can procedurally generate boss fights, but we should also have a look if we can't procedurally generate visuals for our bosses. Creature generators exist, but most of them take a pre-existing structure and add random parts to it. For example: we take a humanoid shape, we choose a random set of legs, a torso, ... till you have a humanoid.

Then we throw a color pallet over it and use the pre-existing humanoid animations and we have our random creature.

Instead, we imagine starting from fully separate components that each have individual animations. We then use graph grammar to generate the structure for them and stitch them together. We could for example start with a torso node and expand our graph with rules from there. Upon receiving our finished graph, we would have to create a converter that takes the best fitting components and sticks them together for a result. We would also have to animate this creature and see if it makes sense.

CONCLUSION

1. THE FRAMEWORK

We created a framework that allows us to generate boss fights. We started by creating a graph using our graph grammar system that we used as a raw representation of the fight. This representation was converted, using our custom converter, into a behavior tree and a body for our boss. We were then able to generate our bosses at runtime during a game, or take the end result and modify it. After we created our boss, we could edit its abilities, conditions, and movement behaviors to our liking using the provided editor.

The resulting boss fights are relatively interesting, even when compared to simple handcrafted ones in other games. The framework loses its ability to compete, once we start comparing it with more complex bosses. Most of these bosses have effects and events that the framework cannot generate without expansion.

The framework can be used in multiple ways: it can be used as a creative tool to help design bosses, and it can be used to generate a lot of similar but still diverse bosses at runtime in game.

Expanding the framework for more complex events and effects is rather complex. This is a drawback of the overall structure of the framework. A redesign would fix this.

2. GRAPH GRAMMAR FOR PROCEDURAL BOSS GENERATION

The graph grammar in our framework gives us the option to keep our boss strength similar over multiple generations. The bosses stay similar in power level because we can manually make the stronger generation rules more expensive than the weaker rules.

We can change a rule and it allows for the system to generate a whole new “species” of bosses. Changing the graph grammar rules to allow bosses to be more or less specified. For example: maybe we always want a second phase to our boss or we can have chance for our boss to not have abilities. The system allowing for this kind of freedom and control is great.

Somethings we didn’t use in our framework, but we could use to expand it, is changing the point cost of rules during the generation. We could increase the cost for a main piece rule every time one gets generated. Or we could decrease the cost of rules that add abilities when more phases get added to the boss.

BIBLIOGRAPHY

[1] H. Fahmy and D. Blostein, 'A survey of graph grammars: theory and applications', in *Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems*, The Hague, Netherlands, 1992, pp. 294–298. doi: 10.1109/ICPR.1992.201776.

[2] A. Zhao *et al.*, 'RoboGrammar: graph grammar for terrain-optimized robot design', *ACM Trans. Graph.*, vol. 39, no. 6, pp. 1–16, Nov. 2020, doi: 10.1145/3414685.3417831.

[3] M. Ilčík, P. Musialski, T. Auzinger, and M. Wimmer, 'Layer-Based Procedural Design of Façades', *Comput. Graph. Forum*, vol. 34, no. 2, pp. 205–216, May 2015, doi: 10.1111/cgf.12553.

[4] E. Jakumeit, S. Buchwald, and M. Kroll, 'GrGen.NET: The expressive, convenient and fast graph rewrite system', *Int. J. Softw. Tools Technol. Transf.*, vol. 12, no. 3–4, pp. 263–271, Jul. 2010, doi: 10.1007/s10009-010-0148-8.

[5] 'Bosses - Official Enter the Gungeon Wiki'. [Online]. Available: https://enterthegungeon.fandom.com/wiki/Bosses

APPENDICES

APPENDIX A

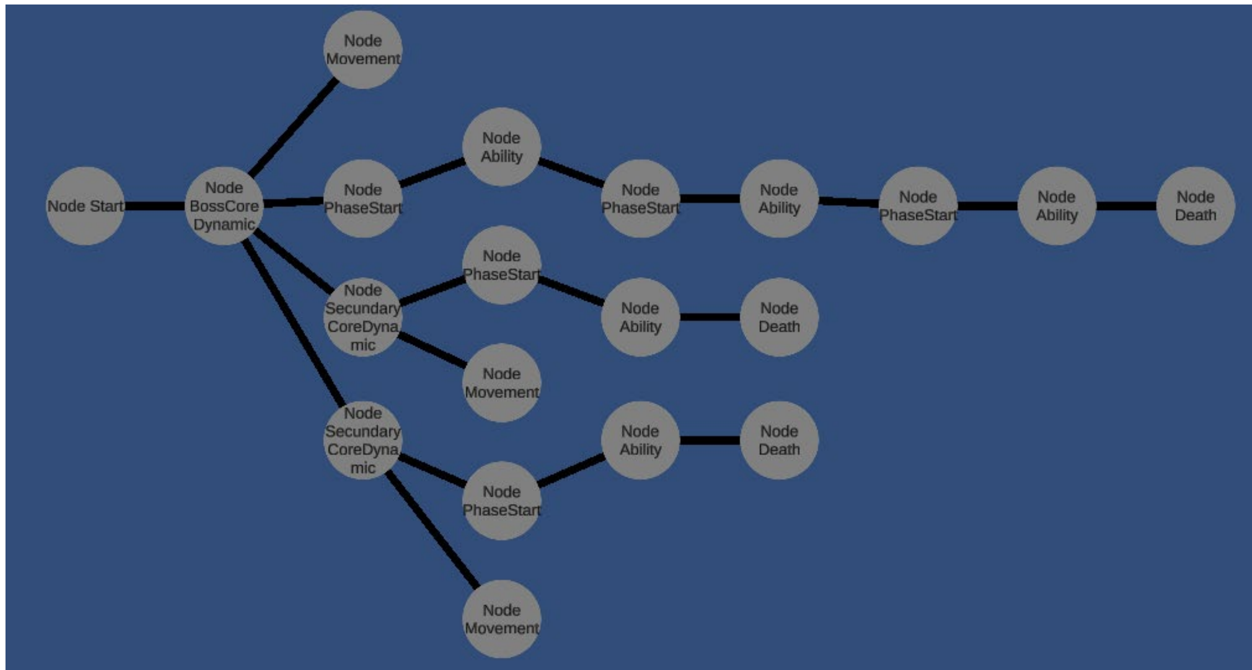

Figure 5: Single boss (500 points)

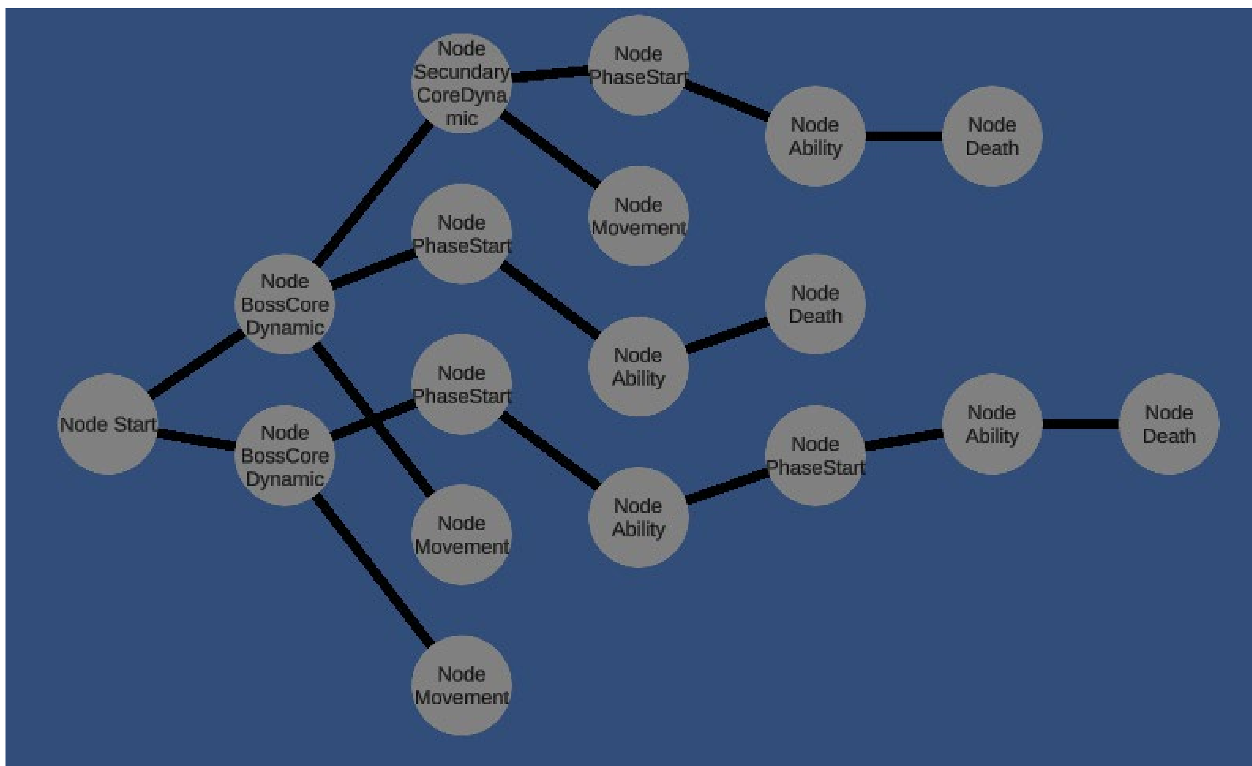

Figure 6: Duo boss (500 points)

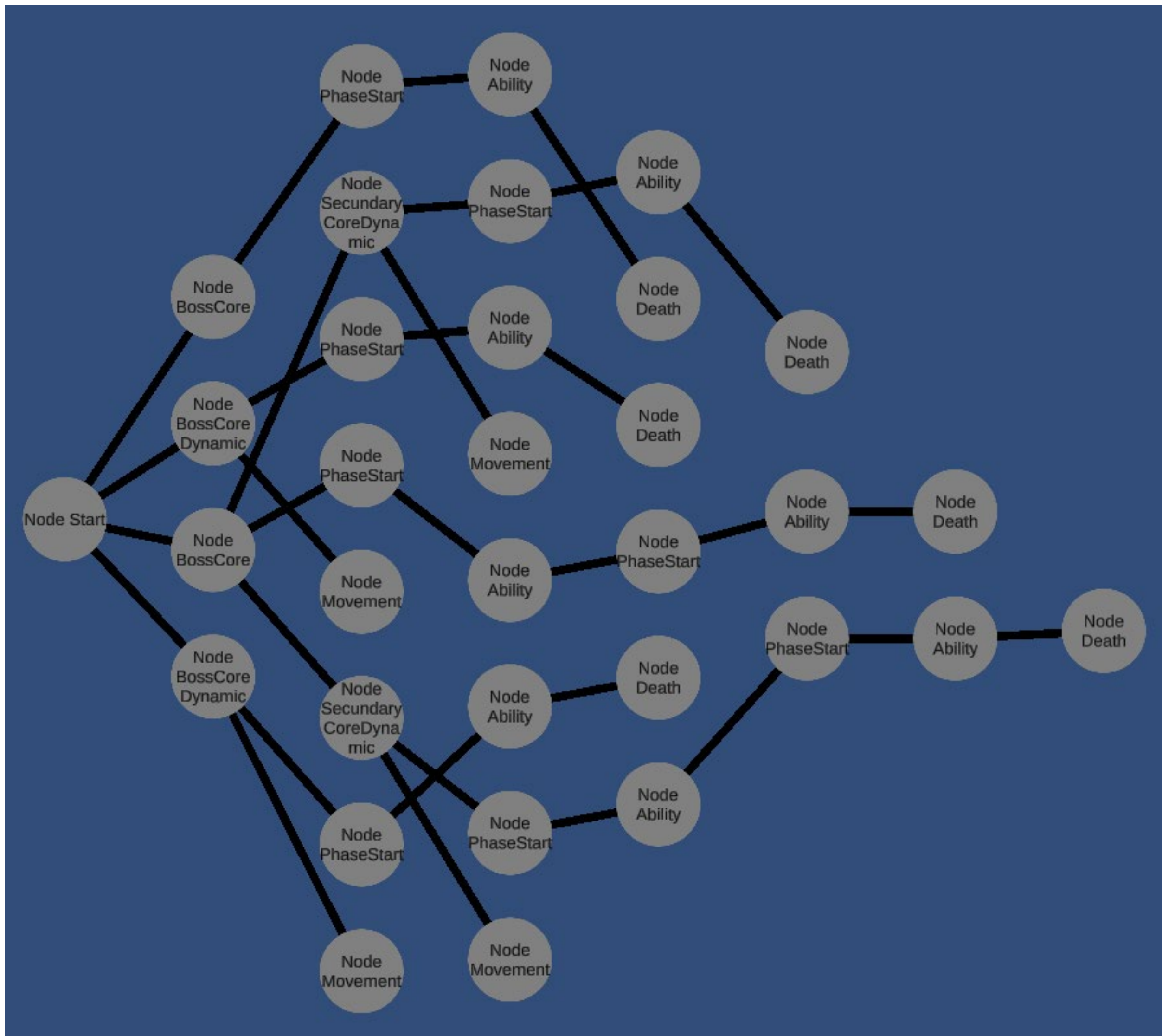

Figure 7: Quad boss (1000 points)

APPENDIX B

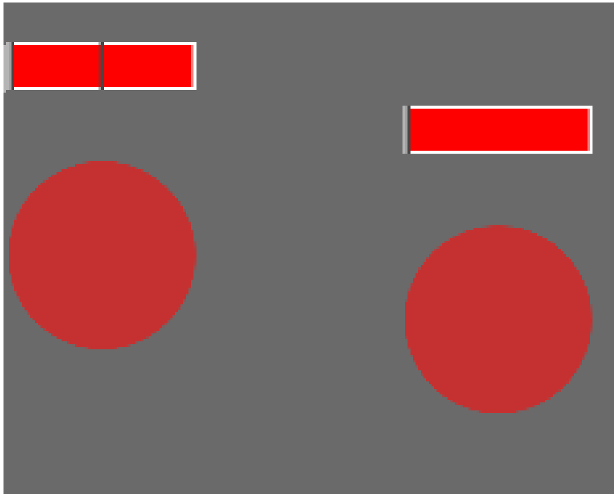

Figure 8: Duo boss structure

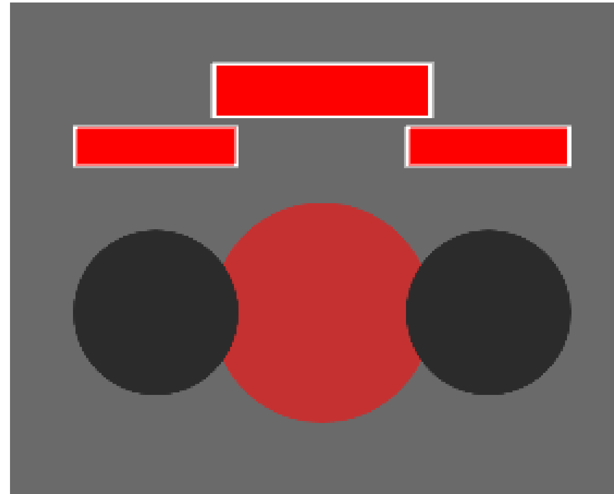

Figure 9: Single boss two side pieces

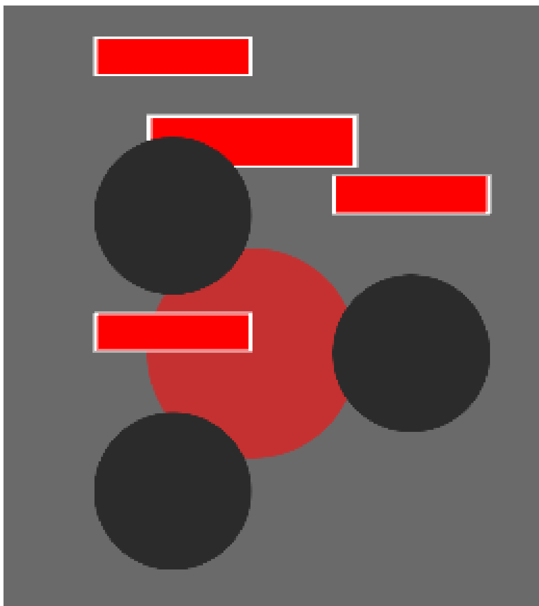

Figure 10: Single boss three side pieces

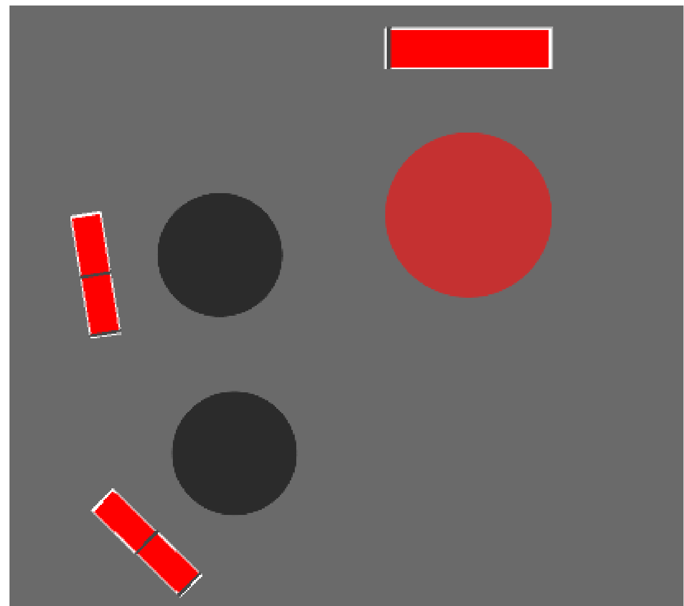

Figure 11: Single boss two orbiting side pieces