# Abstract

Object-oriented programming is a programming paradigm that can be difficult for students to conceptualise. This project attempts to create an engaging teaching aide that offers itself as a resource to those relatively new to programming, while framing itself within the use of object-oriented concepts. This is delivered in the form of a turn-based fighter in which the player and opponent characters are inheritances of the same superclass, while actions taken by the player character come in the form of user generated ‘methods’ pieced together in the form of literal blocks of code seen in interfaces such as Scratch or Blockly. The user achieves a win state by fighting through ten increasingly difficult levels in which they unlock more blocks with which to program new and inventive strategies after each level’s completion. My project was created with the Unity Engine and written in C#.

Contents

[Abstract 2](#_Toc100516534)

[Introduction 5](#_Toc100516535)

[Block Programmers and Teaching Efficacy 5](#_Toc100516536)

[Aims and Objectives of the project 5](#_Toc100516537)

[Literature Review 5](#_Toc100516538)

[Design and Implementation 5](#_Toc100516539)

[Unity System Basics 5](#_Toc100516540)

[The MonoBehaviour Lifecycle 5](#_Toc100516541)

[Prefabs 7](#_Toc100516542)

[System Architecture 8](#_Toc100516543)

[Block Programmer 9](#_Toc100516544)

[Block Anatomy 9](#_Toc100516545)

[Selection Blocks 10](#_Toc100516546)

[Drag and Drop 10](#_Toc100516547)

[Running user generated code 11](#_Toc100516548)

[Unlocking New Blocks 12](#_Toc100516549)

[Terminal 13](#_Toc100516550)

[The Terminal Shell 13](#_Toc100516551)

[Battle System 15](#_Toc100516552)

[The Character Class 15](#_Toc100516553)

[Game Actions 16](#_Toc100516554)

[Interactions 16](#_Toc100516555)

[The main Script 17](#_Toc100516556)

[Update and PlayLoop 17](#_Toc100516557)

[Playing each Interaction 17](#_Toc100516558)

[Handling Graphical Transitions 18](#_Toc100516559)

[Critical Analysis 18](#_Toc100516560)

[Fulfillment of Criteria 18](#_Toc100516561)

[Block Programmer 18](#_Toc100516562)

[Turn-Based Logic 18](#_Toc100516563)

[Interface 18](#_Toc100516564)

[What went badly 18](#_Toc100516565)

[Graphical Polish 18](#_Toc100516566)

[Planning 18](#_Toc100516567)

[Further Development 18](#_Toc100516568)

[More Object-Oriented Concepts 18](#_Toc100516569)

[More Blocks and Deeper Game Design 18](#_Toc100516570)

[Expanding Upon the Terminal 19](#_Toc100516571)

[Computer Player Dynamic Method Generation 19](#_Toc100516572)

[Smoother Transitions and Other Graphical Improvement 19](#_Toc100516573)

[Conclusion 19](#_Toc100516574)

[References and Citations 19](#_Toc100516575)

[Appendix 19](#_Toc100516576)

[Class Diagram 19](#_Toc100516577)

[Testing Results 19](#_Toc100516578)

# Introduction

## Aims and Objectives of the project

The aim of my project is to create a Unity build for a turn based action game that introduces general programming concepts in a block based format as well as some object-oriented computing concepts. To achieve these aims, these criteria must be fulfilled:

* Create a block programming interface
  + Create a series of rendered blocks as Unity game objects.
  + Create functions for each block and a system through which to interpret and call these functions from a user generated sequence of blocks.
  + Create a system for the user to drag and drop blocks in which to instantiate and alter the hierarchy of blocks.
* Create the logic for a turn based action game
  + Create a character class
  + Create an action system with various moves a character can use to affect one another

# Literature Review

block programmers and what makes them good

turn based systems

# Design and Implementation

## Unity System Basics

### The MonoBehaviour Lifecycle

The Unity Engine requires every individual element of a game be a game object. A game object can have (within certain parameters) multiple components attached to it. By using game objects and assigning different components to them, I could craft the visual scenes of my game and create the logic behind them. Unity’s components are written in C# and extend Unity’s own MonoBehaviour class in order to actually interact with the game objects they are assigned to. Each game object follows a lifecycle that the extending the MonoBehaviour class in my custom written components allowed me to manipulate. For the purposes of my 2D only, GUI based game with no physics nor complex animations, the order in which each point in the lifecycle starts can be reduced to only the following critical points:

Awake

OnEnable

Start

OnMouse events

Update

Coroutine Events

OnGUI

OnDisable

OnDestroy

Starting Up

Running

Closing Down

Implementing the MonoBevhiour class into scripts tied to game objects allowed me to write methods named after a specific stage in the lifecycle that will be called when the game object reaches that stage in the lifecycle. The Update part of the lifecycle is called once for every frame rendered and re-rendered onto the user’s screen and used for code that should be called continuously. Given the constant looping nature of the update method, it is the best place to write code anticipating changes of state in my system as I can repeatedly check the state of boolean flags representing events such as user input in another part of my system architecture or the game itself ending. OnGUI is called several times per frame and the designated section to place any code related to GUI rendering or handling different GUI based events. At several places within my code, OnGUI methods (or those closely linked to it) are written to accommodate changing positions of game objects with components to have a graphical representation. Coroutines allow me to incorporate concurrency into my program, meaning that I can impart the illusion of parallelism and add a timed element to certain functions.

### Prefabs

Prefabs (short for prefabricated or prefabrication) are game objects that I can load from the Resources folder in my Unity project and instantiate when needed.

## System Architecture

My system is largely divided into three main parts with a main script that they all talk to while also handling the cyclical flow of the entire product. These parts are the battle system which is responsible for character information, the block programmer and the in-game terminal that allows the user to type commands as a means of user input. The main script houses the gameplay loop of my project’s battle portion and handles showing and hiding the appropriate objects when the user moves away from the programming section of the game to the battle section.

The interaction between the main script and the three larger systems are diagramatically shown below:

Main

Block Programmer

Battle

Terminal

Used when user is creating their methods

Used when user is playing

The dashed arrows represent a small part of cross-communication between each subsystem in error checks from the block programmer to the terminal and sending the user selected method from the terminal to the battle section of the project.

## Block Programmer

The block programmer is the feature that the user interacts the most meaningfully with and is what primarily makes my project an educational resource. The block programmer is a means for the user to create methods that they can name and call by that name when they move onto the battle phase of the current level. To create methods the user first drags a method starter block into the programming area, this now enables the user to attach other blocks to it and create their method. The blocks available to the user to create methods with come in several categories: Action (which is sub-divided into three categories[hyperlink to explaination]), control, maths, logic and information. Action and control blocks are the only types that are literally ‘stacked’ like blocks one after the other, while the other blocks serve as inputs for blocks that require them.

In order to create my block programmer, I researched implementations of block programming interfaces to better formulate my list of criteria and understand the paradigms involved in dealing with user generated code. Blocks Engine 2 (BE2) is an implementation of a block programming engine available on the Unity Engine Assets Store. BE2’s applies user generated code to specific game objects using a system architecture serving a purpose so far removed from my own that all I could glean from it was the idea of of separating out blocks into several hierarchical game objects and what Unity native components and scriptable operations on those components that would help me create stretchable images to simulate block headers and bodies expanding and shrinking to accommodate and fit around blocks that are nested within them. The images and components

### Choice of Blocks to Include

The blocks from BE2 include a handful of basic blocks representing operations integral to a block programmer or any other method of programming including if statements, while/for loops, logical operations and some basic math functions.

### Block Anatomy

Blocks are split up into several different game objects and their assigned components. This allows for independent control over each game object down a block’s hierarchy, helping me with such things as changing the dimensions of independent parts of each block.

#### Sections

Blocks are split into sections to accommodate blocks with multiple distinct groupings of code nested within them. While most blocks only have the one section, this allows blocks such as If/Else statements to exist, as two groups of blocks can exist within the If and Else section atomically and be treated as such when its boolean input is evaluated and only one of these sections are executed. Each section will have a header and optionally a body if it takes blocks as children.

#### Headers and Input Fields

The anatomy of a block changes with its functionality. All blocks have headers that are used to display their name and house their input field if they have one. Input fields are spaces where input blocks can be dragged and dropped into as parameters to be computed by the function associated with the input field’s block if and when it is triggered at runtime. Many input blocks themselves, such as the set of blocks that act as logic gates, have inputs of their own and function as a way to change or run comparisons against values stemming from player and opponent attributes.

#### Bodies

Block body game objects serve as a parent to all blocks that exists lower than that block in its heiarchy. All blocks that contain code blocks within them will store them as children of their body object. As examples, all blocks within a method are children of that method starter block’s body game object – also, if statements or looping blocks will contain all blocks nested within them as children of their body object within the appropriate section.

##### Block Spaces

Each body game object stores a dynamic number of block space objects. These non-visual objects store a position value and a reference to their index within the string of blocks in the body object they belong to to in order to work in tandem with the drag and drop system, allowing it to look for block spaces at positions within a certain distance and insert a block at the desired index stored within the block space that has the nearest position.

### Selection Blocks

Selection blocks are dummy blocks that exist for the player to drag a copy of the selected block from and onto the programming area and live adjacent to it within a scrollable sidebar. They have the exact same anatomy but lack the components that exist within real blocks that allow them to perform operations and communicate with the larger block programming system. Selection blocks contain a component that does nothing except store the Unity prefab of their ‘real’ counterpart block that gets instantiated by the drag and drop system when the user drags their mouse away from their chosen selection block, as if to ‘pick up’ a block and place it later.

### Drag and Drop

Dragging and dropping blocks is a big part of what makes the concept of block programming feel so intuitive, as users pick up and place physically the code that they wish to be executed at a given point. By implementing Unity interfaces I can run code at the point a user sends a mouse event within my project. This includes the point at which the player initiates a click, when the player moves the mouse pointer and updates its position while this click still persists and the point of when the click is released. To accommodate this, dragging and dropping a block can be conceptualised as the three events where a user initiates a click on a chosen block, drags it to a chosen space and the releases the mouse, thus placing the block. The methods called on mouse events capture the position of the mouse in a PointerEventData object. Using this, I could perform operations based on the location of the pointer, but I needed to create a custom raycaster component to identify any potentially interactable game object underneath the user’s mouse.

Raycasters work in tandem with an Unity.EventSystem object to produce rays of specified or infinite length over a Unity canvas game object in an attempt to hit one or more game objects marked as interactable and children to the given canvas. Using this, I could send a ray out at the point of the player’s click and return the block game object under their pointer. There are many different cases within the process for a click, drag and drop motion, and my drag and drop component has to accommodate all of them.

Upon the player clicking their mouse down, their mouse pointer could either be on top of a selection block, an already instantiated block within the programming area or over nothing. If the mouse is clicked over a selection block, that selection block is searched for the block prefab it represents - that prefab is instantiated and defined as the currently dragged object. If the mouse is clicked over a block that is already instantiated and inside the programming area, it is defined as the currently defined object. Upon a mouse drag input, the following logic only applies if the currently dragged object exists and isn’t null. As the currently dragged block is dragged around the programming area, the nearest block space within a hardcoded radius is continually searched for. If this nearest block space exists, a ghost block (which is a regular header block but greyed out) is inserted into the body that the game object belongs to in order to indicate where the currently dragged block would be inserted if the user were to release the mouse at this moment. The ghost block’s inclusion within the method pushes all blocks underneath it in the heirarchy lower, changing the shape of the method indentically to if the currently dragged block were to be added at that moment. If the currently dragged block is an input block, and not to be stacked like action and control blocks, then in lieu of block spaces input fields are searched for instead. If there is an input field that is open and near enough to the input block being dragged, it will glow to signify that it will take the input block on the player’s mouse release.

When the mouse is released, if the currently dragged object exists and has found a space for itself, it will insert itself into that space. In the context of stackable non-input blocks, that block is inserted at the chosen point in the method.

### Running user generated code

Every time a user created method is called in play, the blocks stacked one after the other belonging to that method are run in sequence by the block stack manager when called by the main script until the either player has run out of game actions to perform in a single method, there is no more code in the method left to run or it is detected that the user has tried to create an infinite or near-infinite loop with a repeating block in which there are no actions to decrease the player’s stamina or end to the method and thus no exit condition.

#### Block Functions & The Block Stack Manager

The block stack manager is responsible for executing each block in a method and returning game actions to the main script so the player character can use them. The block stack manager first creates the block stack itself by linearising the blocks in the method. Each block from the chosen method is added recursively to the block stack while adding special block components without an actual block game object called ‘End of Section’ or EOS blocks at the end of each section. These EOS blocks keep note of which block the section began with, meaning backtracking within the block stack until the EOS block’s ‘start block’ is found allows me to loop over specific sections of code.

The block stack manager uses an integer pointer to determine the current block whose block function should be called and then chooses what to do next based on that block’s type with a switch case statement. Given that the only blocks that are stacked linearly to make up the body of a method are attack and control blocks, the three possible types the chosen block can be are action, control and end of section blocks.

If the current block is an action block, the block stack manager calls its function which returns the action that block represents. If the block is a control block, the block’s function is called which returns a possibly repositioned pointer value if blocks of code need to be skipped immediately, such as an if/else block evaluating false and logging the original value of the pointer for later use if code must be skipped later, such as an if/else block evaluating true. If the block is an EOS block, the EOS block’s start block is quieried. If this start block is a control block, it must have a logged pointer variable inside its class. The control block’s function class’ ‘onRepeat’ method is now called by the block stack manager. This function returns a pointer value based upon what code should be run next after the end of the current section.

Every block besides the method starter block has a component tied to them that contains their associated function. These functions can be called by the BlockStack manager to control the flow of user generated code or by blocks themselves if using other blocks as inputs. Since all block functions are tied to their respective blocks as components, they need to be separate classes. Because of this, each block function’s class inherits from a superclass based upon their block’s category (game action blocks inherit from the ActionFunction superclass, information blocks from the InfoFunction class etc.). Each function class being a subclass and each superclass having the same return type meant that I could generalise via upcasting any block function classes, be they input blocks whose functions are called by other blocks, or those read by the block stack manager.

As each non-action block’s associated function is called, it outputs text to the terminal so the user can see the decisions being made for each block as the method progresses – this includes input blocks called from other blocks and the inputs that those input blocks may take etc. in a depth-first manner. For example, when an if statement block that has input blocks slotted into it with the equivalent of !PlayerHealth<50&&opponentDefence<20 is executed, the values for playerHealth and opponentDefense, their comparisons and the playerHealth comparison’s inversion will be input before that of the AND operation, which then allows the if statement block to output whether its condition has been fulfilled and its nested blocks will be executed or not.

#### The Computer Player

The computer player includes the code running functionality of the block stack manager while performing the job of populating its block stack itself to simulate a computer player creating methods to invoke agains the user. This is done by loading and instantiating required blocks before adding them to the block stack in the order they will be run. As these blocks have to be instantiated and actually exist within the game to access the block component within them, I decided to simply ascribing them as children of a game object outside of the main heirarchy of the project, effectively hiding them from view before the computer’s turn is over and they are all destroyed to make way for the next set of blocks.

The computer player has access to four methods each level and has an equally random chance to select any of those methods when propted to. I have hardcoded methods with and increasing number of available blocks that prove more of a threat to the user as they progress through the game. These more advanced methods are slowly woven into the different selections of possible methods for the computer player as if the computer is unlocking blocks itself. To further this illusion, in the blocks used by the computer player, I have tried to make it seem as if the computer is trying to evenly distribute its possible actions over several different strategies such as strengthening specific attributes of itself to make blocks used later in the same method more threatening and other offensive actions along with and in contrast to healing and defence based methods.

### Unlocking New Blocks

The player starts off with a basic collection of action and non-action blocks that will expand by three with each level cleared. The blocks given to the player allow them enough freedom to grasp the concept of my project while leaving room for more complex thinking to arise with the tools that they will be semi-randomly given. The player starts off with four game action blocks (two attacks, one status effect and one heal), an If statement, while statement and several other logic and number input blocks as well as character blocks that give them the freedom to target a character not reccomended by a given action block (action blocks come with character input blocks already). With each round cleared, the player is given one game action block and two other random non-action blocks. This ensures that the player is continually given action blocks to form strategies around and the non-action blocks with which to fulfil those strategies.

## Terminal

My in-game terminal is based upon the Command Terminal Unity asset by Stillwwater in which I have retrofitted several components that enable the functionality of quality of life features such as autocomplete and an array for a history of previously entered strings, and written my own code for visuals and I/O. The terminal provides all user interaction outside of stacking methods together and is most notably used for concluding the programming stage of the current level and calling those user created methods in battle. The terminal class handles the visual representation of the terminal as well as the subsystems involved such as the terminal history and autocorrect which allow a user to scroll through past inputs and tab to correct their current input respectively.

### The Terminal Shell

Internally, the terminal shell class behaves differently given what kind of command the user enters. The shell differentiates between calls for built-in commands and user generated methods. The shell’s built in-commands return nothing back to the terminal when called and are tabulated in detail below:

|  |  |
| --- | --- |
| Command | Description |
| HELP | Displays help information about each bulit-in command in the terminal. Takes name of a built-in command as a parameter and only displays help for that command if found. |
| CLEAR | Clears the terminal of text |
| FINISH | Performs checks on user generated methods. If all clear, stops signals the main script to stop programming and start battle. |
| PLAYERSTATS | Displays player stats (including distinctions between base stats and base stats + current status effects) in the terminal |
| OPPONENTSTATS | Same as PLAYERSTATS but for opponent information |
| CHANGENAME | Takes one string parameter and changes player’s character instance’s name to that string |
| QUIT | Quits back to the main menu |

The commands stored in the shell are done so using dictionaries to map the string containing their name to the C# methodInfo objects pointing to that method they represent. This is functionally the same for user generated methods, as instead of a methodInfo object, the dictionary maps from the method’s name to the method starter block that the method comes from. Given that I can execute a user generated method from its method starter block, this achieves the same result.

As the user can only enter commands into the shell, each string given to the terminal is validated against the regular expression to enforce that strings entered start with a letter followed by a string of alphanumeric characters, ending with open and closed parentheses that optionally contain another command name without the parentheses. The shell is equipped to handle parameters, but only the help command is actually defined to accept any. User generated methods also do not take any parameters.

## Battle System

### The Character Class

At the heart of the battle system is the character class which includes attributes for all the stats involved in play and all status effects applied to a given character. Each stat pertains to a numerical representation towards different aspects of character’s ability, the higher the better.

The following table details each character stat and how they affect character interactions:

|  |  |
| --- | --- |
| Stat | Description |
| Max Health | The maximum health value of the character. The character’s health cannot exceed this value through the use of healing actions. |
| Health | The current health value of the character. If the opponent’s health reaches zero, the player has won and vice versa. If the character’s maximum health changes, their health is affected proprtionally. |
| Attack | Affects the damage dealt by attack based game actions. |
| Speed | Dictates the order in which both characters will take their turn and affects the chance for individual actions to hit or miss their target. |
| Defence | Dictates how much damage a character takes from opposing attack based game actions. |
| Stamina | The number of game actions a character is able to perform in one method before their turn ends. |

The types of game actions a character can use during battle includes status effects, as mentioned later. When they succeed in affecting a target, status effects positively or negatively affect a given status by a certain amount and are represented in the character class with a data structure called an attribute modifier. Attribute modifiers as a structure contain the status that they affect, the multiplier by which it is affected and the amount of turns it will remain active for. The character class implements these attribute modifiers by storing each active attribute modifier in several lists, categorised by status type. Every time a character stat value is needed (e.g. to calculate health to display or speed to judge the chance to dodge an attack), the active status effect multipliers pertaning to that stat are totalled and multiplied by the base value for that stat before being returned. With each turn taken in battle, the remaining turn counter with each active attribute modifier is decremented by one, and removed from their respective lists if this counter has become less than 0. In the interest of balancing the use of status effects, when all relavent status effects pertaining to a given attribute are totalled, their multipliers are reduced by a third each time. This ensures that players cannot directly manipulate character attributes to an extent that would break and consequently remove the challenge from my design such as subtracting heavily from a target’s maximum health or making attacks deal magnitudes more damage.

### Game Actions

Game actions refer to the blocks available to the player and opponent character outside of controlling the flow of code, and deal with affecting the characters mid-battle. These game actions can be categorised as attacks, status effects and heals. Game action objects primary function is storing the hardcoded values that make them unique and all have names and a target. Attack game actions have a damage and speed values calculated from character attack and speed with the addition of hardcoded multipliers to differentiate them. Status effect game actions contain information for what character attribute they alter, a multiplier dictating whether they raise or lower that attribute and by how much as well as the amount of turns that status effect should last for, give that it succeeds in being applied to the target. Heal game action objects store the hardcoded amount of health they return to the target and the chance to heal the target. The ActionCollection class is a static class containing methods that return a game action object. This lets me instantiate the game actions I have designed by calling ActionCollection.gameActionName(). To balance actions, if their multiplier dictating how strong that action is is high, the multiplier dictating its chance to succeed or hit its target is lower. This adds a small extra layer to the decision making involved in the user creating their character’s methods.

### Interactions

When a game action from either the player or the opponent is due to happen, it is first turned into an interaction by the interaction handler encompassed within the game script. Interactions on instantiation take an action and determine the outcome of that action, storing it in the attributes of the interaction object itself for later reference. The interaction handler takes a game action of one of the three types and returns an interaction object of the appropriate sub-class dependent on the type of game action.

Attack interactions calculate the prospective damage of an attack on its target with the formula: . If , then the attack has hit and conversly the attack misses if it is not. Status effect interactions create an attribute modifier object based to be applied to the target character and the . Heal interactions work similarly to status effects, calculating the chance to hit randomly based off of the speed of the character who instigated the heal and envoloping the game action within itself for later reference.

## The main Script

The main script is a component belonging to the ‘Main’ game object which acts as a parent game object to all other game objects that I wish to be visible to the user. The main script itself acts to unify all systems under it to manage the flow of the different logical and graphical transitions between each level and the win and loss states.

### Update and PlayLoop

When the user has shifted from the programming stage of a given level, the game object containing the main script as a component is enabled, and the game loop housed within the script’s MonoBehaviour.Update method becomes active.

The update method is called once every frame and conducts several boolean checks to determine where in the course of the current level the user is. First, the update method checks that both characters are loaded, the game is not over and that the player and opponent are not currently using methods against each other. If all these conditions are met, all coroutines are stopped to avoid conflict between any new coroutines and existing ones. It is now that the opponent’s method of choice for the upcoming clash is chosen and the player allowed to enter theirs (or any other command of their choice). After the player’s method has been chosen, the terminal is set back to a readonly state and the player’s block stack is initialised for the blocks in their method to be read by in playInteraction.

playRound decides the order in which both characters will use their chosen methods and clears both the player and computer’s block stacks before ending the turn and flipping ongoingInteraction, resetting the state of the game back to when neither character has made their choice of method. playRound is a coroutine to facilitate the delay between a character’s successive game actions and invokes those coroutines in such a way that playRound is called concurrent with the rest of the system’s ongoing actions, yet each character’s interactions are played successively by using the yield and return keywords before calling StartCoroutine. If the value for each character’s speed attribute is equal, the system leaves the prioroty to a 50/50 chance of swaying either way.

### Playing each Interaction

playerInteraction executes each block within the player’s chosen method and if the current block is an action block, an interaction is created from the action is returns to then play. With each time this happens the player’s stamina is decremented by 1, ending their turn when they have used their maximum amount of game actions. The playerInteraction method also checks for methods with no exit condition by incrementing a counter if logic is being applied by the blocks in a player’s method, just without any result. When this happens over a thousand times, the system assumes the player has constructed an infinite or equally useless loop, and they relinquish their turn. The opponent’s version of this method is identical save for it having no infinite loop detection, as the current computer opponent’s methods are all hardcoded with no such case.

playInteractionAfterDelay is a coroutine that runs the playInteraction method with the same interaction object and character parameters but after the specified float type delay value. Using Unity’s Time.deltaTime value, I can compare the time between playInteractionAfterDelay being called and when the specified time has been exceeded. This methodology for timing functions is not perfect but has an accuracy of milliseconds on the machines I have tested it on, making it servicable.

playInteraction takes an interaction object, the character that instigated the interaction and applies the numerical results of that interaction while displaying text information about it, dependent on its result, through the terminal.

### Handling Graphical Transitions

The main script handles the graphical transitions involed with winning a level or wining/losing the game. There are game objects containing graphical elements for the win and loss states for the game that the main script activates while making sure all other game objects are hidden by calling the MonoBehaviour.SetActive method, which accepts a boolean and sets the activity of the chosen game object to that state. An inactive game object is no longer rendered and all components lower than it in its hierarchy are also set as inactive, effectively ‘switching’ off all affected game objects. As an example, when the player wins a level, the main script shows the next level game object responsible for showing the amount of remaining levels, which then handles the user input to move onto showing the player’s unlocked blocks by itself. After the user clicks to acknowledge the blocks they have unlocked, the next level game object calls the main script’s method to perform the neccesary tasks for moving onto the next level in the game, which includes hiding the next level game object.

# Critical Analysis

## Fulfillment of Criteria

### Block Programmer

The block programming system satisfied my criteria. The interface works like many other block programming interfaces and facilitates a familiar satisfaction in how blocks slot into place and expand and shrink around their child blocks or lack thereof. The system for block functions is highly modularised, making for easier code maintenance and creation of new blocks. Blocks that move the position of the pointer in my linearisations of user generated code are generalised as they are interpretted, meaning that implementing new control blocks is as easy as creating any other type of block, requiring only that new control blocks contain a function and onRepeat method in which to possibly reassign the pointer.

### Turn-Based Battle System

The turn-based battle system performs its purpose well,

### Interface

The terminal interface serves as a novel way of interacting with the larger scope of the game while leaning into the theme of programming.

## What went badly

### Graphical Polish

The lack of smooth visual transitions between different game objects being actively rendered makes my project look less professional in some areas. Game objects tied to different states within the game pop in and out rather than

### Planning

# Further Development

## More Object-Oriented Concepts

In order to expand upon the theme of object-oriented programming, users could unlock ‘inheritances’ that come with premade methods the character can use, inherited from a base class that they could equip like a class from a traditional role-playing game. Another object-oriented concept to explore would be defining and accessing/updating custom attributes other than those integral to the character class. This would expand upon the degree of player expression and be an even more hands-on way of conveying how a singular object’s attributes change than that which I have done with the attributes that are the player’s stats.

## More Built-In Terminal Commands

The shell is written to support built-in commands of one or more parameters,

## More Blocks and Deeper Game Design

Mention parameters for methods, making each action its own sub class of their current type so they have completely individual behaviour such as changing function as target character’s attributes satisfy a math function for e.g which will deepen strategy.

A feature that would be easier to implement, given that changes to the mainloop are made to accommodate it, is actions each being separate subclasses with more complex logic to them instead of instances of more general subclasses of actions as they are now. As well as enveloping the interaction system, this would allow each action to respond to specific criteria in more unique ways that the player would have to respond to with more strategy.

## Computer Player Dynamic Method Generation

The computer player’s methods are made up entirely out of hardcoded sequences of action blocks in order to form the illusion of them making their own decisions like the player - employing artificial intelligence within the construction of the computer player’s methods in order to make it no longer an illusion would be the most complex addition to my project. This could involve different weightings asribed to more defensive or offensive actions (for example) which would change based upon the computer player’s current stats and health, as well as those of the player. To incorporate the blocks available to the computer player, every action block could have offensive/defensive values of their own that allow them to be included in a method provided that the sum of offensive and defensive values for each block does not exceed the multiplication of their respective weighting and the computer player’s stamina stat.

## Smoother Transitions and Other Graphical Improvement

With greater understanding of Unity’s tools I could create less abupt transitions between my project’s different visual states. Given further development time, I would have liked to implement more elements moving on and off screen rather than them popping in and out of existence in an effort to give my project more of a visual professionalism.

# Conclusion

# References and Citations

**There are no sources in the current document.**

# Appendix

# Class Diagram

# Testing Results