Top of the Stack

Keep
Body + Soul
Together-er

Back of the Queue

CS 360 Group Project

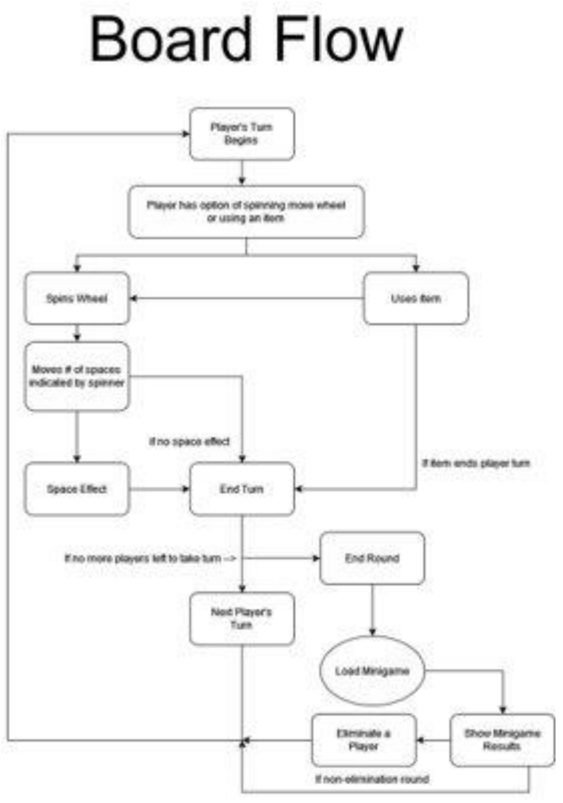
An Educational Adventure

Coding Farm v1.0 Technical Manual

Table of Contents

Game Structure

Our game is a multiplayer party game featuring a board game for players to compete against one another on. There would also be a single player mode where players are able to freely select minigames to play on their own. The board game would be similar to Mario Party in that players would take turns rolling dice to move around the board from space to space, with every few rounds ending with a minigame. Whoever performs the worst in the minigame would be eliminated from the board game section of the game (they will still be allowed to participate in minigames however) until there is only one player still standing.



Board Game flow chart

Menus

The menu is quite simple in terms of how they work, all you need is a button and a simple script that lets the screen change. Just know when you change the screen it destroys the scene you were just on and all the items on it. If you what any of the item to stay loaded, you will need a dontDestoyOnLoad script like saveMe in music. The menu uses a simple script placed on the canvas that changes the screen. For this script it is easy to change for different screens, all you need to change is the String in SceneManager.LoadScene to the screen you what it to be changed to. Just make sure the screen is in the build.

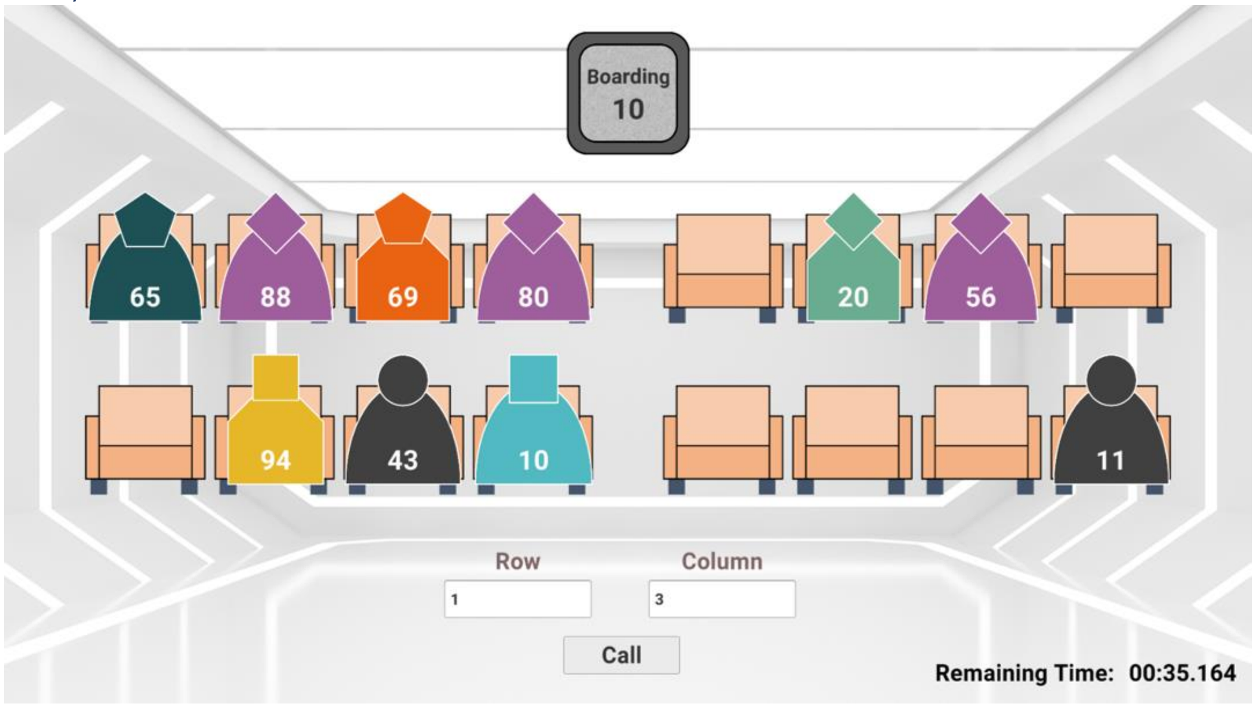
```
public Button OpenOptions;
void Start() {
   OpenOptions.onClick.AddListener(delegate {
   SceneManager.LoadScene("Options Menu"); //The string that needs changed
   });}
```

Minigames

NOTE: When testing any minigame by itself, the minigame will throw three null reference exceptions. These three are a null reference exception thrown by the SwitchAudioOnLoad, a null reference exception thrown by MiniGameData, and a null reference exception thrown by the minigame controller caused by Zombie not being created. These three null reference exceptions are natural and will not be thrown when played in build or from the Main Menu.

Array

Arrays Attack!



Minigame 1 Arrays Attack!

Classes

DOA Controller

This class sets up the minigame by randomly generating the array and validates the player input.

NOTE: This class uses the enum `Errors`. This enum will not be discussed in the documentation.

Variables

- GameWin: a reference to the minigame's `GameWin` object
- GameLoss: a reference to the minigame's `GameLoss` object
- CountDown: a reference to the minigame's `CountDown` object
- Row: a reference to the `TMP_InputField` object that will take the player's row input
- Col: a reference to the `TMP_InputField` object that will take the player's column input
- SubmitButton: a reference to the `Button` object that will validate the player's input
- ErrorMessage: a reference to the `TextMeshProUGUI` object that will display any errors found in validation
- People: an array of `TextMeshProUGUI` objects representing the individual cells in the array and displayed in the minigame
- TVDisplay: a reference to a `Text` object that will display the next correct value
- Numbers: an integer `array` containing the sixteen integers shown on the people
- NumberQueue: a `Queue` that will hold the correct order of values to call
- _numbers: a `HashSet` containing integers; this ensures that no value is repeated in the game.
- Log: A Log that will record the data from the playthrough

Methods

Awake

```
private void Awake() {
    Row.Select();
    Log = new Log() {
       IsPractice = Zombie.IsPractice,
       IsTeamGame = !Zombie.IsSolo,
       ErrorsMade=0,
       TurnsUsed=0,
       TimeTaken=0,
    };
    Numbers = new int[16];
    int i = 0;
    while(_numbers.Count < Numbers.Length) {
       int R = Random.Range(10, 99);
       if (_numbers.Add(R)) {
          Numbers[i] = R;
          i++;
        }
    }
    Numbers.Shuffle();
    for(int j = 0; j < People.Length; j++) {
       People[j].text = Numbers[j].ToString();
       People[j].gameObject.transform.parent.GetComponent<Image>().sprite =
Resources.Load<Sprite>($"DOA/Person{(Numbers[j]%8)+1}");
    }
    Numbers.Shuffle();
    for(int j = 0; j < Numbers.Length; j++) { NumberQueue.Enqueue(Numbers[j]); }
    TVDisplay.text = NumberQueue.Dequeue().ToString();
    ErrorMessage.gameObject.SetActive(false);
    SubmitButton.onClick.AddListener(Submit);
    Row.onEndEdit.AddListener(delegate { Col.Select(); });
    Col.onEndEdit.AddListener(delegate { SubmitButton.Select(); });
}
```
This method randomly generates the array being represented graphically and updates all necessary fields.

Update

```
private void Update() {
   if (GameLoss.gameObject.activeInHierarchy) {
       Log.Win = false;
   } else if (GameWin.gameObject.activeInHierarchy) {
       Log.Win = true;
       Log.TimeTaken = CountDown.TimeElapsed;
   }
}
```
This method checks if the game has resulted in a win or a loss, then updates the game log.

Submit

```
void Submit() {
    if (!string.IsNullOrEmpty(Row.text) && !string.IsNullOrEmpty(Col.text)) {
       string Coord = $"R{int.Parse(Row.text)}C{int.Parse(Col.text)}";
       Debug.Log(Coord);
       Col.text = "";
       Row.text = "";
       Row.Select();
       Log.TurnsUsed++;
       foreach (TextMeshProUGUI person in People) {
           if (person.name == Coord) {
               if (person.text == TVDisplay.text) {
                   person.gameObject.transform.parent.gameObject.SetActive(false);
                   UpdateError(Errors.None);
                   try {
                       TVDisplay.text = NumberQueue.Dequeue().ToString();
                   } catch {
                       GameWin.Show();
                       Debug.Log("Win");
                   }
                   return;
               } else {
                   Debug.Log(Errors.Not_Correct);
                   UpdateError(Errors.Not_Correct);
                   return;
               }
           }
       }
       Log.ErrorsMade++;
```

```
        Debug.Log(Errors.Invalid_Input);
        UpdateError(Errors.Invalid_Input);
     } else {
        Log.ErrorsMade++;
        UpdateError(Errors.No_Input);
        Debug.Log(Errors.No_Input);
    }
    Col.text = "";
    Row.text = "";
    Row.Select();
}
```

This method will check if the correct row and column were selected. If the correct value was selected, the value is then removed, otherwise it checks what error the player did and updates the results.

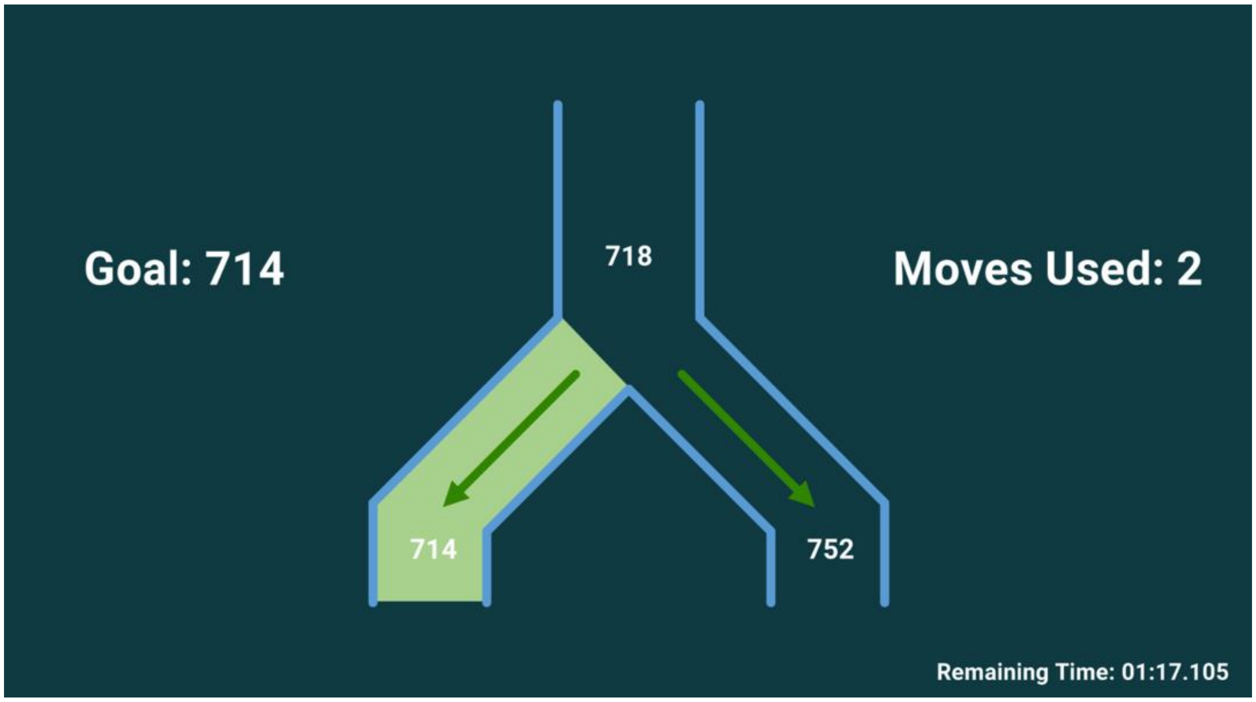
Update Error

```
void UpdateError(Errors error) {
    ErrorMessage.gameObject.SetActive(error != Errors.None);
    ErrorMessage.text = "Error " + error.ToString().Replace('_', ' ');
}
```

This checks if an error has been detected and will display or hide the warning message if necessary.

Binary Search Tree

We Call It Maze



Minigame 2 We Call It Maze

In this minigame a player is navigating a randomly generated binary search tree looking for a specific value displayed as the Goal. The player should complete the search in as little moves as possible and

within the given time span of 90 seconds (though in testing, it should be around 30 seconds).

The player is allowed to go in three directions; up, left, right. Each direction consumes a move, even if the next node is null/does nothing. Up takes the player to the current node's parent; left sets the current node to its left child; right sets the current node to its right child. The player wins if they select the correct node, and the player loses if they fail to select the correct node in the given time span.

Classes

BST Con

Variables

- LeftText: a `TextMeshProUGUI` reference that will display the left child value
- RightText: a `TextMeshProUGUI` reference that will display the right child value
- CurrentText: a `TextMeshProUGUI` reference that will display the current value
- GoalText: a `TextMeshProUGUGI` reference that will display the goal value
- MovesUsed: a `TextMeshProUGUI` reference that will display the number of moves taken
- GameWin: a reference to the scene's `GameWin` object
- Moves: the number of moves used by the player
- Direction: the `Direction` that the current node can go
- BinTree: the `BinaryTree` containing string that will be used by the game
- Goal: the `string` that the player is trying to find
- CurrentNode: a `Node` containing a string representing where the player currently is in the `BinaryTree`

Methods

Start

```
private void Start() {
    Log = new GameLogging.Log() {
        IsPractice = Zombie.IsPractice,
        IsTeamGame = !Zombie.IsSolo,
        ErrorsMade = -1,
        TurnsUsed = 0,
        TimeTaken = 0,
    };
    Moves = 0;
    BinTree = new BinaryTree<string>();
    List<string> Goals = new List<string>();
    SetItems.SetArgs args = SetItems.SetArgs.Empty;
    if (Random.value > .5f) { args = SetItems.SelectSet(); }
    while (BinTree.Count < 25) {
        if(!args.Equals(SetItems.SetArgs.Empty)) {
            string x = args.Set.ElementAt(Random.Range(0, args.Set.Count-1));
            args.Set.Remove(x);
            BinTree.Insert(x);
            Goals.Add(x);
        } else {
            string x = Random.Range(1, 999).ToString();
            BinTree.Insert(x, x.Length);
            Goals.Add(x);
        }
    }
    Goal = Goals[Random.Range(0, Goals.Count-1)].ToString();
    GoalText.text = $"Goal: {Goal}";
    Moves = 0;
    CurrentText.text = BinTree.Root.Data.ToString();
    try { LeftText.text = BinTree.Root.Left.Data.ToString(); } catch { LeftText.text = ""; }
    try{ RightText.text=BinTree.Root.Right.Data.ToString(); } catch { RightText.text=""; }
    CurrentNode = BinTree.Root;
    Debug.Log(BinTree);
}
```

This method will reset all variables and generates a random binary tree with 25 nodes. It will then randomly select a value contained in the tree that the player will need to find. It will then populate the game with the root node and its children's values and print the fully generated tree.

On Enter

```
public void OnEnter() {
    GetComponent<Image>().color = new Color(255, 255, 255, 255);
}
```

This event fires when the mouse enters the bounds of the host object. This script makes the image visible.

On Exit

```
public void OnExit() {
    GetComponent<Image>().color = new Color(255, 255, 255, 0);
}
```

This event fires when the mouse exits the bounds of the host object. This script makes the image invisible.

On Click

```
public void OnClick() {
    Moves++;
    if (Direction == Direction.Right) {
    if(RightText.text == Goal.ToString()) { Debug.Log("Correct Right"); GameWin.Show(); return; }
    if(CurrentNode.Right != null)
    CurrentNode = CurrentNode.Right;
    } else if(Direction == Direction.Left) {
    if (LeftText.text == Goal.ToString()) { Debug.Log("Correct Left"); GameWin.Show(); return; }
    if (CurrentNode.Left != null)
        CurrentNode = CurrentNode.Left;
    } else if(Direction == Direction.Center) {
        if (CurrentText.text == Goal.ToString()) { Debug.Log("Correct Center");
GameWin.Show(); return; }
        if (CurrentNode.Parent != null)
           CurrentNode = CurrentNode.Parent;
    }
    CurrentText.text = CurrentNode.Data.ToString();
    try {
        LeftText.text = CurrentNode.Left.Data.ToString();
    } catch { LeftText.text = ""; }
    try {
        RightText.text = CurrentNode.Right.Data.ToString();
    }catch { RightText.text = ""; }
    Debug.Log("Clicked");
}
```
This will increment the player's move counter and it will check if the selected value is the goal. If the player did not select the right node, the game will then navigate to where the player selected, populating the two child nodes with the selected node's children.

Node

NOTE: This class is generic.

NOTE: This class can only be made of children of the interface, `System.IComparable<T>`, such as int or string.

Variables

- Data: a generic type that will store the value of the node
- OrderingValue: an integer that allows for presorting data that doesn't sort correctly as a string
- Left: a Node reference to the node's left child
- Right: a Node reference to the node's right child
- Parent: a Node reference to the node's parent
- IsLeaf: a bool representing if the node is a leaf node

Methods

Node

```
public Node(T data, int? oV, Node<T> Left, Node<T> Right, Node<T> Parent = null) {
    Data = data;
    this.Left = Left;
    this.Right = Right;
    this.Parent = Parent;
    if(oV != null)
       OrderingValue = (int)oV;
}
```
A constructor taking in all input values and populating the class's variables.

Compare To

```
public int CompareTo(Node<T> other) {
    if(numberLength != other.numberLength)
         return numberLength - other.numberLength;
    else
        return Data.CompareTo(other.Data);
}
```

This method is used to compare the data in two nodes. It will use the orderingValue if available, otherwise, it uses the data's default comparison function.

Binary Tree

NOTE: This class is generic.

NOTE: This class can only be made of children of the interface, `System.IComparable<T>`, such as int or string.

Variables

- Root: a Node reference that will represent the root of the tree
- Count: an integer counting the number of nodes in the tree

Methods

Insert

```
public void Insert(T data, int? oV = null) {
    if(Root == null) {
        Root = new Node<T>(data, oV, null, null);
    } else {
        InsertSort(Root, new Node<T>(data, oV, null, null));
    }
}
```

This method inserts a new node in the tree by calling the *InsertSort* method.

Insert Sort

```
void InsertSort(Node<T> Base, Node<T> InsertValue) {
    InsertValue.Parent = Base;
    if(InsertValue.CompareTo(Base) < 0) {
        if(Base.Left == null) {
            Base.Left = InsertValue;
        } else {
            InsertSort(Base.Left, InsertValue);
        }
    } else {
        if(Base.Right == null) {
            Base.Right = InsertValue;
        } else {
            InsertSort(Base.Right, InsertValue);
        }
    }
}
```

This will recursively insert the value at the correct index position in the binary tree.

Count Nodes

```
int CountNodes(Node<T> Root) {
    if(Root == null) { return 0; }
    return 1 + CountNodes(Root.Left) + CountNodes(Root.Right);
}
```

Recursively counts the number of nodes in the tree following the NLR method of navigation.

To string

```
string Tostring(Node<T> node) {
    if(node == null) { return "T"; }
    return Tostring(node.Left)+" "+node.Data.ToString()+" "+Tostring(node.Right);
}
```

Recursively creates a string following the LNR method of navigation. It then returns the constructed string.

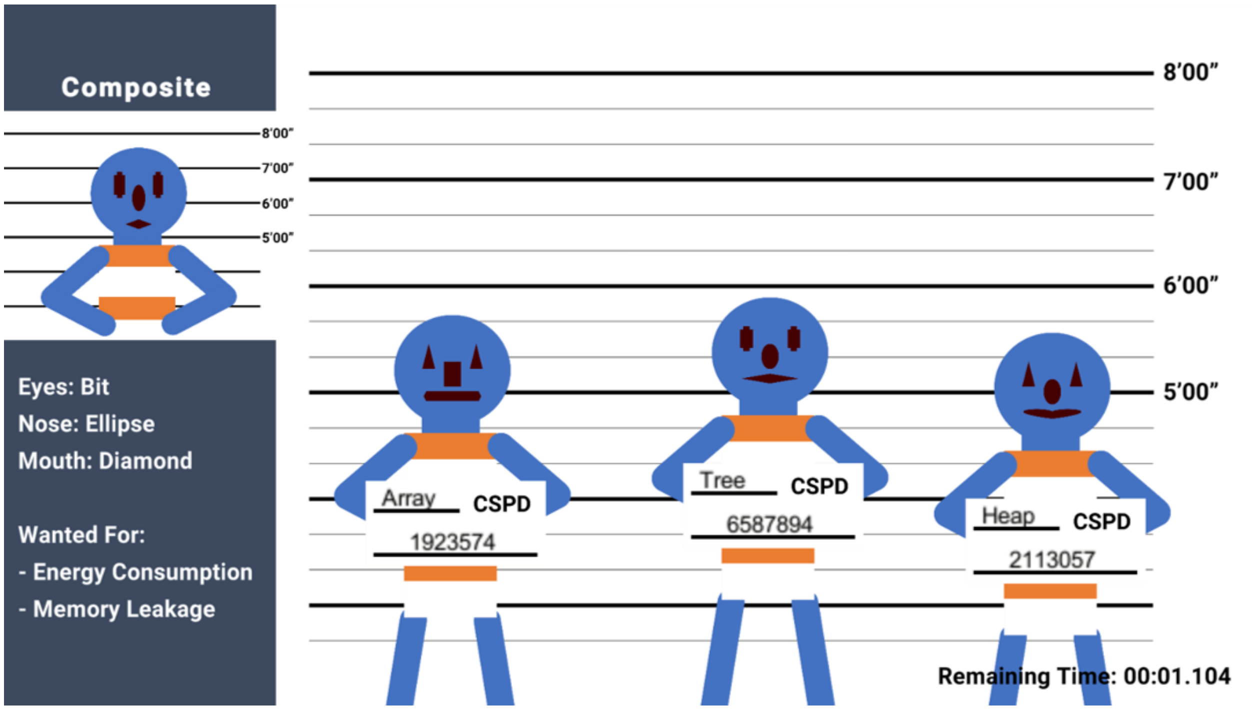
To String

```
public override string ToString() {
    return Tostring(Root);
}
```

Returns the string generated by the recursive to string method.

Pointer

Pointer Panic



Minigame 3 Pointer Panic

In this minigame a player is exploring how pointers are unique memory addresses used by software. The player's goal is to identify as many 'pointers' as they can without making a mistake and in the given time span of 90 seconds.

The player clicks on one of three randomly generated 'pointers' with the hint 'pointer' showing the correct 'pointer' with a few distortions. The game randomly generates a different 'pointer' for every correct answer and ends if the time runs out or the player clicks the incorrect pointer.

Classes

Line Up Controller

This class contains the logic responsible for controlling the minigame, such as choosing the correct person and generating any random shapes.

Variables

- GameWinScreen: a reference to the minigame's `GameWin` object
- GameLoss: a reference to the minigame's `GameLoss` object
- CountDown: a reference to the minigame's `CountDown` object
- Eyes: a reference to the target pointer's eye `sprite`
- Nose: a reference to the target pointer's nose `sprite`
- Mouth: a reference to the target pointer's mouth `sprite`
- Charges: an `array` of references to the suspect pointers
- CorrectFace: an `array` representing the target pointer's face
- Characteristics: the `text` hint displayed to the user
- GameLog: the `log` of this minigame instance.

Methods

Awake

```
void Awake(){
    GameLog = new Log() {
        IsTeamGame = !Zombie.IsSolo,
        IsPractice = Zombie.IsPractice,
        Score = -1,
        TurnsUsed = 1,
    };
    Load();
}
```

This method initializes the `GameLog` for the minigame and then generates the first wave of faces for the player.

Load

```
private void Load() {
    CorrectFace = new Face.Shape[] {
        (Face.Shape)Random.Range(0,7),
        (Face.Shape)Random.Range(0,7),
        (Face.Shape)Random.Range(0,7),
    };
    foreach (Composite charge in Charges) {
        Face.Shape Eye = (Face.Shape)Random.Range(0, 7);
        Face.Shape Nose = (Face.Shape)Random.Range(0, 7);
        Face.Shape Mouth = (Face.Shape)Random.Range(0, 7);
        charge.Populate(Eye, Nose, Mouth, false);
    }
    int CorrectCharge = Random.Range(0, Charges.Length);
    Charges[CorrectCharge].Populate(CorrectFace[0], CorrectFace[1],
        CorrectFace[2], true);
    Eyes.sprite = Face.Eyes[(int)CorrectFace[0]];
    Nose.sprite = Face.Noses[(int)CorrectFace[1]];
    Mouth.sprite = Face.Mouths[(int)CorrectFace[2]];
    Characteristics.text = $"Eyes: {CorrectFace[0]}\nNose: {CorrectFace[1]}\n" +
        $"Mouth: {CorrectFace[2]}\n\nWanted For:\n" +
        "- Energy Consumption\n- Memory Leakage";
}
```
This method randomly generates the correct and incorrect faces that will be displayed to the user. It will then choose a random correct answer and override their face to be correct. It then populates the composite face and characteristic hint with the correct answers.

Choice Made

```
public void ChoiceMade(bool Correct) {
    if (!Correct) { GameLog.ErrorsMade++; Finish(false); return; }
    if (CountDown.Time < .1f) { Finish(Correct); return; }
    Load();
}
```
This method will check if the player selected the correct face and checks if the game should continue. The game ends if the selected face is incorrect or when the time is

Choice Made

```
public void ChoiceMade(bool Correct) {
    if (!Correct) { GameLog.ErrorsMade++; /*Play Wrong Sound;*/ return; }
    //Play Correct Sound
    GameLog.Win = Correct;
    GameLog.TimeTaken = CountDown.TimeElapsed;
    CountDown.StopTimer();
    Zombie.CurrentProfileStats.Stats["Pointer"]["Police Line
        Up"].GameLog.Add(GameLog);
    if (Correct)
        GameWinScreen.Show();
}
```
This method takes in a Boolean, which is passed from the `Composite.OnPointerDown` method, and either registers the game as a win or adds a wrong guess to the log. If the game is a win, this method will add the log to the player and will display the `GameWinScreen`.

Composite

This class contains the logic for the individual pointers that will be displayed on screen.

NOTE: This file also contains the related static class `Face` and enum `Shape`. These will not be discussed as they only store array keys and arrays. For more information on their general structure, see the appendix.

Variables

- Controller: a reference to the `LineUpController` that will control this object
- Eye: the `sprite` reference for the eyes
- Nose: the `sprite` reference for the nose
- Mouth: the `sprite` reference for the mouth
- Name: the `text` reference for the name
- Number: the `text` reference for the number
- IsCorrect: a `bool` determining if this pointer is the correct answer; the default is false
- EyeShape: a `Shape` enum reference that will determine the pointer's eye shape
- NoseShape: a `Shape` enum reference that will determine the pointer's nose shape
- MouthShape: a `Shape` enum reference that will determine the pointer's mouth shape

Methods

Start

```
public void Start() {
    Number.text = $"{Random.Range(0, 9999999)}".PadLeft(7,'0');
    Eye.sprite = Face.Eyes[(int)EyeShape];
    Nose.sprite = Face.Noses[(int)NoseShape];
    Mouth.sprite = Face.Mouths[(int)MouthShape];
}
```

This will randomly generate a 7-digit number, it will then pad the random number if it is less than 7 digits with zeros. It will then populate the pointer with default values.

Populate

```
public void Populate(Face.Shape eyes, Face.Shape nose, Face.Shape mouth, bool
    iscorrect) {
    EyeShape = eyes;
    NoseShape = nose;
    MouthShape = mouth;
    IsCorrect = iscorrect;
    Eye.sprite = Face.Eyes[(int)eyes];
    Nose.sprite = Face.Noses[(int)nose];
    Mouth.sprite = Face.Mouths[(int)mouth];
}
```

Taking in three separate `Shape` enums and a bool, it sets the sprites and references to reflect the inputs.

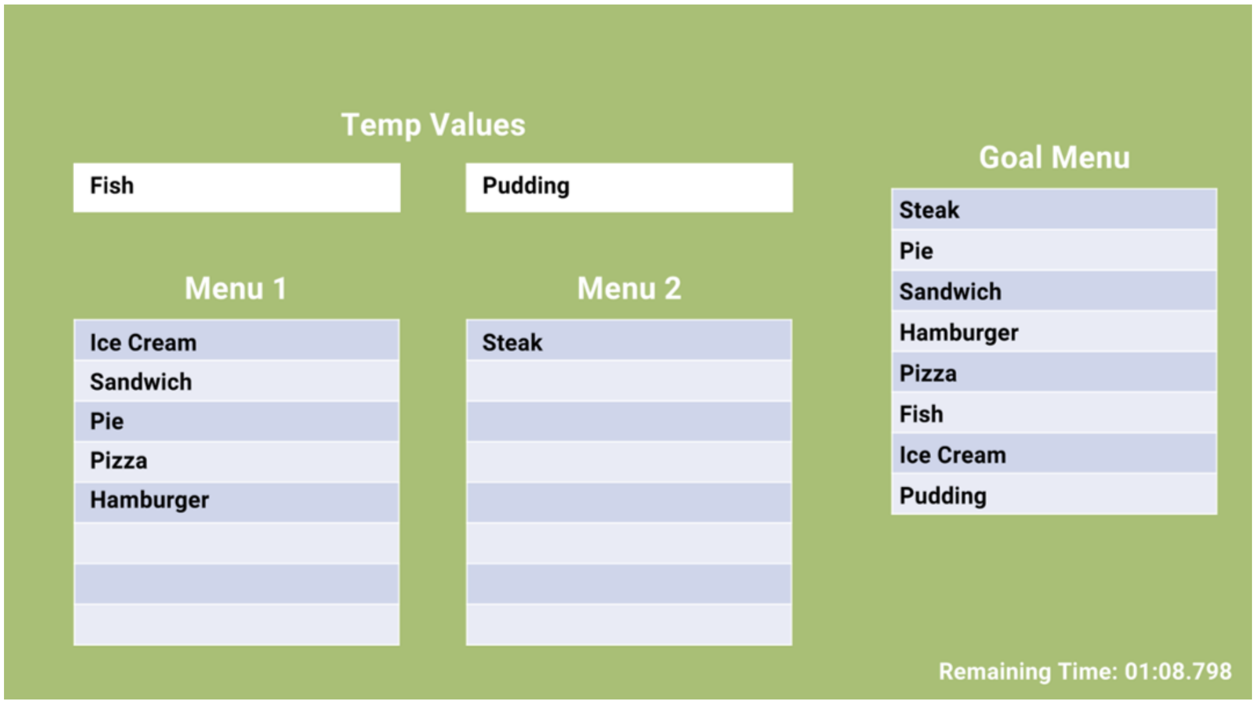
On Pointer Down

```
public void OnPointerDown() {
    if (IsCorrect) {
        Debug.Log("Correct");
        Controller.ChoiceMade(true);
    } else {
        Debug.Log("Incorrect");
        Controller.ChoiceMade(false);
    }
}
```

This will call the Controller's `ChoiceMade` function, discussed earlier, and will print out if the user made the correct choice.

Queue

Manic Menus



Minigame 4 Manic Menus

In this minigame a player is using only Queue behaviors (Dequeue, Enqueue, and FIFO) to sort a menu into the correct way as denoted in the Goal Menu.

The player is allowed to dequeue items into two temporary cells (representing variables) or into a different menu. For the player to win, either Menu 1 or Menu 2 need to match exactly the Goal Menu in the given time span of 90 seconds.

Classes

Queue Host

Variables

- Queue: a list of queue items. This list will be used to validate and coordinate the different items the player can use.
- TurnsTaken: a static integer representing the number of turns taken in this game.
- WinGame: a reference to the minigames GameWin screen
- Correct: a reference to a text label that represents the correct solution to the game.

Methods

Start

```
private void Start() {
    TurnsTaken = 0;
    Correct.text = Food.generateItems(8);
    string[] Foods = Correct.text.Split('\n');
    Foods.Shuffle();
    try {
        int i = 0;
        foreach(QueueItem Item in Queue) {
            Item.GetComponent<TextMeshProUGUI>().text = Foods[i];
            i++;
        }
    } catch { }
}
```

This method sets all values to default values controlling the minigame and sets the queue items to the correct text.

On Click

```
public void OnClick() {
    if (QueueItem.Selected == null) { return; }
    if (Queue.Contains(QueueItem.Selected)) { return; }
    TurnsTaken++;
    Queue.Add(QueueItem.Selected);
    int LocationAdded = 0;
    foreach (QueueItem TP in Queue) {
        LocationAdded++;
    }
    if (QueueItem.Selected.Host != null) {
       QueueItem.Selected.Host.Queue.Remove(QueueItem.Selected);
       QueueItem.Selected.Host.UpdateTree();
    }
    QueueItem.Selected.Host = this;
    try {
       for (int i = 0; i < QueueItem.Selected.Host.Queue.Count; i++) {
           QueueItem.Selected.Host.Queue[i].IsSelectable = i == 0;
       }
    } catch { Debug.Log("Empty Tower"); }
    QueueItem.Selected.LastLegalLocation =
QueueItem.Selected.GetComponent<RectTransform>().anchoredPosition = new Vector3(
    GetComponent<RectTransform>().anchoredPosition.x + 25,
       (10) - 60 * (Queue.Count-1),
     0
       );
    QueueItem.Selected.IsSelectable = (Queue.IndexOf(QueueItem.Selected) == 0);
    QueueItem.Selected = null;
    Check();
}
```

This method takes the selected queue item and tries to add it to the host. It then updates the host and checks if the player has met the win condition.

Update Tree

```
public void UpdateTree() {
    try {
        for (int i = 0; i < Queue.Count; i++) {
            Queue[i].IsSelectable = i == 0;
            Queue[i].LastLegalLocation =
            Queue[i].GetComponent<RectTransform>().anchoredPosition = new Vector3(
                GetComponent<RectTransform>().anchoredPosition.x + 25,
                (10) - 60 * i,
                0
            );
        }
    } catch { Debug.Log("Empty Tower"); }
}
```

This method moves all the objects tied to the host game object to the correct position.

Check

```
void Check() {
    string Answer = "";
    foreach(QueueItem item in Queue) {
        Answer += item.GetComponent<TextMeshProUGUI>().text + "\n";
    }
    if(Answer.Replace("\n",string.Empty) ==
Correct.text.Replace("\n",string.Empty)) { WinGame.Show(); }
}
```

This method checks if the player met the win condition.

Food

Variables

- FoodItems: a string array that represents the food items that can be randomly selected and scrambled for the game.

Methods

Generate Items

```
public static string generateItems(int items) {
    FoodItems.Shuffle();
    string end = "";
    bool start = true;
    foreach (string item in FoodItems) {
        if (start) { end += item; start = false; } else { end += "\n" + item; }
    }
    return end;
}
```

This randomly shuffles the food items and creates a string that will display the goal queue.

Shuffle

```
public static void Shuffle<T>(this T[] list) {
   int n = list.Length;
   while (n > 1) {
        n--;
        int k = Random.Range(0, n + 1);
        T value = list[k];
        list[k] = list[n];
        list[n] = value;
    }
}
```

This will randomly shuffle the given array.

Queue Item

Variables

- Host: a reference to the game object's current QueueHost
- IsSelectable: a bool checking if the player can select the game object
- LastLegalLocation: a vector2 representing the last location that the game object has snapped to.
- Transform: a reference to this game object's RectTransform. This is used to move the game object.
- Selected: a static reference to the current selected Queue Item.
- IsTemp: a bool checking if this game object can be used as a queue item holder.

Methods

Start

```
void Start() { LastLegalLocation = Transform.anchoredPosition; }
```

This sets the default value of LastLegalLocation.

On Click

```
public void OnClick() {
    if (IsSelectable && !IsTemp)
        Selected = this;
}
```

This method checks that the object can be selected and sets the currently selected object to the clicked object.

Temp Capture

Late Update

```
private void LateUpdate() {
    if (Selected == this) {
        StartCoroutine(nameof(Hover));
    } else {
        transform.localPosition = LastLegalLocation;
    }
}
```

This method animates or snaps the host object depending on if it is selected or not.

NOTE: Late update is the last update method that fires. This ensures that any methods firing on update will finish execution before this method fires.

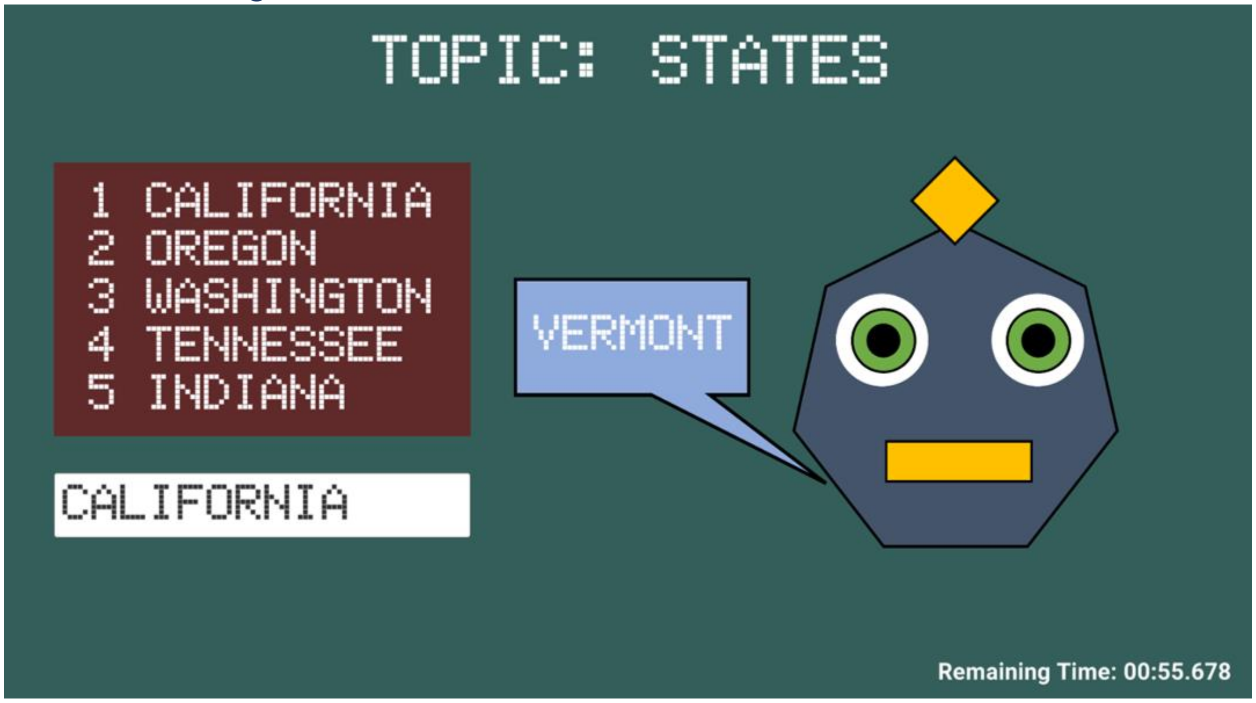
Hover

```
IEnumerator Hover() {
    for (float i = 0f; i <= 1f; i += .1f) {
        transform.localPosition = new Vector3() {
            x = LastLegalLocation.x,
            y = (Mathf.Pow(Mathf.Sin(Time.realtimeSinceStartup * 2), 2) * 10f) +
LastLegalLocation.y,
            z = 0,
        };
        yield return new WaitForSeconds(.015f);
    }
}
```

This method animates the selected object by adjusting the height of the object and having a slight delay between heights.

Set

Robbie's Revenge



Minigame 5 Robbie's Revenge

In this minigame a player is using their memory to understand the set concept of individuality. The player wants to get as many unique values as they can without repeating themselves or the robot, hereafter referred to as Robbie.

The player must input a value that matches the given category and must not have been previously entered by either the player or Robbie. To win, the player must last the full duration of the time span of 90 seconds without repeating a value. The game is possible to end early if $0 < Robbie's\ Difficulty \leq 1$ (with higher values resulting in a higher likelihood of ending prematurely).

Classes

NOTE: These classes use the struct *SetArgs*. Due to its simple nature, it will not be discussed in the documentation.

Set Controller

Variables

- RobbieDifficulty: a float representing the percent chance that Robbie will select a value that is illegal and terminate the game prematurely. It is defaultly set in code to 10% (.1), but testing has placed this value to around 1% (.01)
- PlayerInput: a TMP_InputField reference that will be used to validate the player's selection.
- SetType: a TextMeshProUGUI reference that will be used to display the title text.
- PlayerSet: a TextMeshProUGUI reference that will be used to display the last five entries that the player has said.
- RobbieSpeak: a reference to an animator to animate Robbie's face and voice.
- RobbieTalk: a TextMeshProUGUI reference that represents the text bubble text when Robbie speaks.
- RobbieFinished: a bool representing if Robbie's talk animation has finished playing.
- Robbie_Inactive: a sprite reference that will reset Robbie's talk animation.
- Player: a string hash set that represents all values that the player has said.
- LegalOptions: a string hash set that represents all values that neither the player nor Robbie has said.
- IllegalOptions: a string hash set that represents all values that either the player or Robbie has said.
- Loss: a reference to the game's GameLoss object.
- Win: a reference to the game's GameWin object.
- CountDown: a reference to the game's CountDown object.

Methods

Start

```
void Start() {
    Log = new GameLogging.Log() {
        IsPractice = Zombie.IsPractice,
        IsTeamGame = !Zombie.IsSolo,
        ErrorsMade = 0,
        TurnsUsed = 0,
    };
    RobbieSpeak.enabled = false;
    SetItems.SetArgs GameLoad = SetItems.SelectSet();
    LegalOptions = GameLoad.Set;
    SetType.text = $"Topic: {GameLoad.Name}";
    PlayerSet.text = "";
    PlayerInput.onEndEdit.AddListener(delegate {
        PlayerMove();
    });
    PlayerInput.Select();
}
```
This method sets up the minigame by randomly selecting a valid pre-coded set from the resources and binds all events to their respective game objects.

Update

```
private void Update() {
    if(Input.GetKey(KeyCode.Return) && RobbieFinished) {
        RobbieFinished = false;
        RobbieTalk.transform.parent.gameObject.SetActive(false);
        RobbieTalk.text = "";
        RobbieSpeak.enabled = false;
        RobbieSpeak.gameObject.GetComponent<Image>().sprite = Robbie_Inactive;
        PlayerInput.text = "";
        PlayerInput.enabled = true;
        PlayerInput.Select();
    }
}
```
This method checks if the player has finished entering data and if Robbie has spoken at least one letter. If both of those conditions are met, then it resets the minigame for the next player selection.

Robbie Speaking

```
IEnumerator RobbieSpeaking(string Word) {
   RobbieTalk.text = "";
   PlayerInput.enabled = false;
   RobbieFinished = false;
   RobbieSpeak.enabled = true;
   RobbieTalk.transform.parent.gameObject.SetActive(true);
   foreach (char C in Word.ToCharArray()) {
        RobbieTalk.text += C;
        yield return new WaitForSeconds(.1f);
        RobbieFinished = true;
   }
}
```
This script will animate the text in Robbie's speech bubble by manually adding each character with a slight delay.

Player Move

```
void PlayerMove() {
    if(string.IsNullOrWhiteSpace(PlayerInput.text) ||
string.IsNullOrWhiteSpace(PlayerInput.text)) { PlayerInput.text = "";
PlayerInput.Select(); return; }
    int Location = LegalOptions.ContainsValue(PlayerInput.text);
    if (Location != -1) {
        if (Player.Add(LegalOptions.ElementAt(Location))) {
            LegalOptions.Remove(LegalOptions.ElementAt(Location));
            IllegalOptions.Add(LegalOptions.ElementAt(Location));
            string Last5 = "";
            for (int i = 1; i <= 5; i++) {
                try {
                    Last5 += $"{i} {Player.ElementAt(Player.Count - i)}\n";
                } catch {
                    Last5 += "\n";
                }
            }
            PlayerSet.text = Last5;
            Debug.Log($"Legal {PlayerInput.text}");
            PlayerInput.Select();
        }
    } else {
        Debug.Log($"Illegal {PlayerInput.text}");
        Loss.Show();
        CountDown.StopTimer();
    }
    float RNG = Random.value;
    if (RNG >= RobbieDifficulty) {
        try {
            int RandomIndex = Random.Range(0, LegalOptions.Count - 1);
            StartCoroutine(RobbieSpeaking(LegalOptions.ElementAt(RandomIndex)));
            IllegalOptions.Add(LegalOptions.ElementAt(RandomIndex));
            LegalOptions.Remove(LegalOptions.ElementAt(RandomIndex));
            Debug.Log($"Legal {LegalOptions.ElementAt(RandomIndex)}");
        } catch {
            Win.Show();
            CountDown.StopTimer();
        }
    } else {
        int RandomIndex = Random.Range(0, IllegalOptions.Count-1);
        StartCoroutine(RobbieSpeaking(IllegalOptions.ElementAt(RandomIndex)));
        Debug.Log($"Illegal {IllegalOptions.ElementAt(RandomIndex)}");
        Win.Show();
        CountDown.StopTimer();
    }
}
```
This method checks if the player made a valid choice, if the player made a correct choice, then it displays the last five values that the player has said. Robbie then selects if they will make a correct or incorrect answer. Robbie then selects their answer and displays it. If an incorrect answer is guessed, the game ends.

Set Items

Variables

- Sets: a List of string representing all sets that the game can draw on.
- RegEx: a regex search pattern that will be used to validate the player's input.

Methods

Constructors

```
static SetItems() {
    TextAsset[] Assets = Resources.LoadAll<TextAsset>("Set\\");
    foreach (TextAsset asset in Assets) { Sets.Add(asset.name); }
}
```

Grabs all valid text files that contain sets and populates class variables.

Select Set

```
public static SetArgs SelectSet() {
    string Item = Sets[Random.Range(0, Sets.Count)];
    string[] LegalItems=Resources.Load<TextAsset>($"Set\\{Item}").text.Split(',');
    HashSet<string> Items = new HashSet<string>();
    foreach(string Legal in LegalItems) {
        Items.Add(Legal);
    }
    return new SetArgs(Item, Items);
}
```

Selects a random set from the list generated in the constructor. It then reads the file and generates a new hash set. It then returns a new SetArgs with the selected set and hash set.

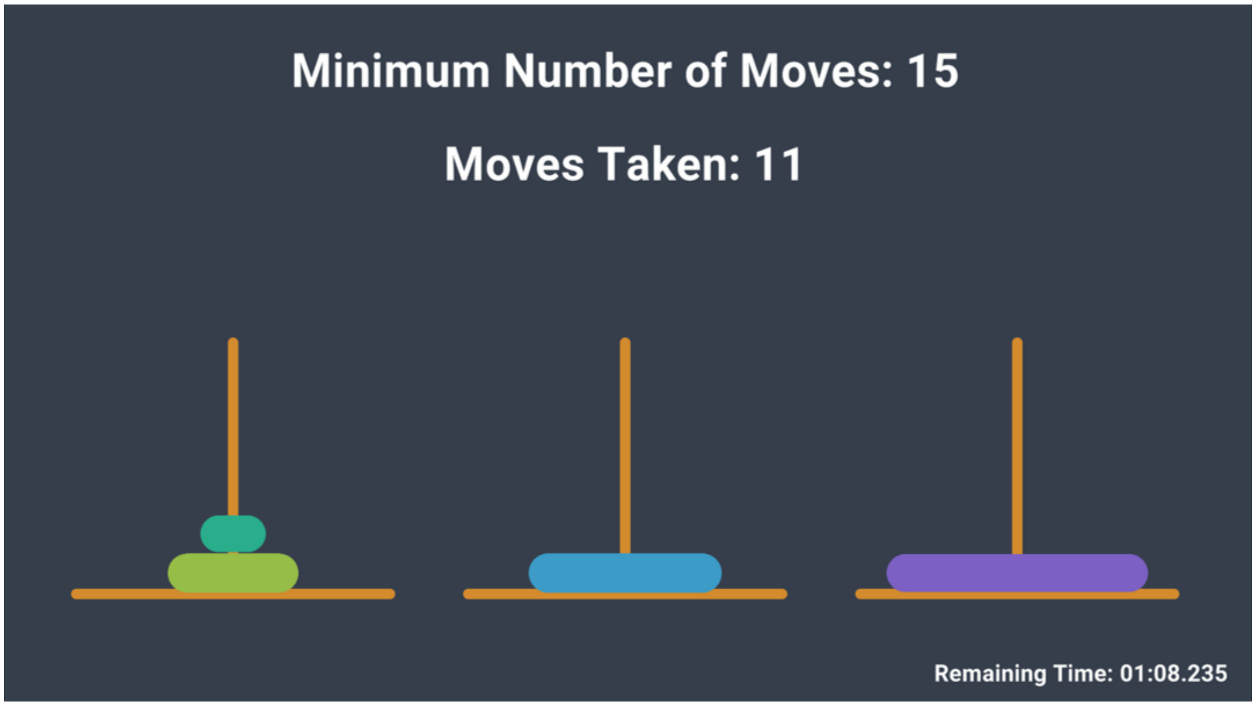
Contains Value

```
public static int ContainsValue(this HashSet<string> Set, string valueCheck) {
    string CleanedCheck =
RegEx.Replace(valueCheck.Normalize(System.Text.NormalizationForm.FormKD),"").ToLowe
r();
    int counter = -1;
    foreach(string Item in Set) {
        counter++;
        string cleaned =
RegEx.Replace(Item.Normalize(System.Text.NormalizationForm.FormKD),"").ToLower();
        if (cleaned == CleanedCheck)
            return counter;
        }
    return -1;
    }
}
```

This is an extension method created for string hash sets. This method checks the input value by stripping any special characters (string.Normalize) and then replaces any characters matching the class's predefined regex pattern. The index that it finds is -1 if the value is not in the set and the index of the value if it is in the set.

Stack

Towers of Hanoi



Minigame 6 Towers of Hanoi

In this minigame a player is using only Stack behaviors (Dequeue, Enqueue, and LIFO) to get the starting tower to the furthest right column in the given time span of 90 seconds (though in testing 30 seconds is all that is required).

The player is allowed to dequeue items from the top of one stack and enqueue the item into one of the other stacks given that the item is smaller than the one below it. If the item is larger than the item below, the move is canceled.

Classes

TOH Tower

Variables

- Tower: a list holding all the tower parts currently on the host.
- IsCorrect: a bool representing if the object is representing if the tower is the correct tower.
- TurnsTaken: a static integer to represent the number of moves taken in the minigame.
- TurnsTakenDisplay: a reference to the minigame's label displaying the number of moves taken.
- WinGame: a reference to the minigame's GameWin object.

Methods

Start

```
public void Start() {
    Log = new GameLogging.Log() {
        IsPractice = Zombie.IsPractice,
        IsTeamGame = !Zombie.IsSolo,
        ErrorsMade = 0,
        TurnsUsed = 0,
    };
    TurnsTaken = 0;
}
```

This method sets all the controlling variables to their default values.

On Click

```
public void OnClick() {
    if(TowerPart.Selected == null) { return; }
    if (Tower.Contains(TowerPart.Selected)) { TowerPart.Selected = null;return; }
    int LocationAdded = 1;
    foreach (TowerPart TP in Tower) {
        LocationAdded++;
        if (TP.Rank < TowerPart.Selected.Rank) { TowerPart.Selected = null;
ShowError(); return; }
    }
    TurnsTaken++;
    TurnsTakenDisplay.text = $"Moves Taken: {TurnsTaken}";
    Tower.Add(TowerPart.Selected);
    TowerPart.Selected.Tower.Tower.Remove(TowerPart.Selected);
    try {
        for (int i = 0; i < TowerPart.Selected.Tower.Tower.Count; i++) {
            TowerPart.Selected.Tower.Tower[i].IsSelectable = i ==
TowerPart.Selected.Tower.Tower.Count - 1;
        }
    } catch { Debug.Log("Empty Tower");}
    TowerPart.Selected.Tower = this;
    TowerPart.Selected.LastLegalLocation =
TowerPart.Selected.GetComponent<RectTransform>().anchoredPosition = new
Vector3(GetComponent<RectTransform>().anchoredPosition.x, (-155) - 60 * (4 -
LocationAdded), 0);
    TowerPart.Selected = null;
    UpdateTree();
    Check();
}
```

This method validates the player's move. If the move is valid, it will update the tower to correctly reflect the move. It then updates the items in the host and checks for the win condition. If the move is not valid, it cancels the move.

Update Tree

```
void UpdateTree() {
    try {
        for (int i = 0; i <Tower.Count; i++) {
            Tower[i].IsSelectable = i == Tower.Count-1;
        }
    } catch { Debug.Log("Empty Tower"); }
}
```
This checks and updates each element in the tower are selectable. Only the top element may be selected.

Check

```
void Check() {
    if (IsCorrect) {
       if(Tower.Count == 4) {
            Debug.Log("Correct");
            WinGame.Show();
        }
    }
}
```
This method checks if the player has met the win condition.

Show Error

```
void ShowError() {
    Debug.Log("Illegal");
}
```
This method logs that the player made an illegal move.

Tower Part

Variables

- Rank: an integer representing the rank of the tower part, allowing Tower of Hanoi functionality.
- Tower: a reference to the current tower that the game object belongs to.
- IsSelectable: a bool representing if the object can be selected.
- LastLegalLocation: a vector2 representing the last location where the object could legally be.
- Transform: a reference to the host's recttransform object.
- TowerPart: a static reference representing the current selected object.

Methods

Start

```
void Start() {
    LastLegalLocation = Transform.anchoredPosition;
}
```
This sets the default value of LastLegalLocation.

On Click

```
public void OnClick() {
    if (IsSelectable)
        Selected = this;
}
```

This method checks that the object can be selected and sets the currently selected object to the clicked object.

Late Update

```
private void LateUpdate() {
    if (Selected == this) {
        StartCoroutine(nameof(Hover));
    } else {
        transform.localPosition = LastLegalLocation;
    }
}
```

This method animates or snaps the host object depending on if it is selected or not.

NOTE: Late update is the last update method that fires. This ensures that any methods firing on update will finish execution before this method fires.

Hover

```
IEnumerator Hover() {
    for(float i = 0f; i<=1f; i += .1f){
        transform.localPosition = new Vector3() {
            x = LastLegalLocation.x,
            y = (Mathf.Pow(Mathf.Sin(Time.realtimeSinceStartup * 2),2) * 10f) +
LastLegalLocation.y,
            z = 0,
        };
        yield return new WaitForSeconds(.015f);
    }
}
```

This method animates the selected object by adjusting the height of the object and having a slight delay between heights.

Board Game

The Board Game is where the multiplayer and minigames would have come together. Unfortunately, due to a lack of time needed to get multiplayer up and working, we were unable to complete the board game segment of the project. That wasn't for lack of trying however, and for future students wishing to finish the project where we left off, we have lots of code that went into the half-complete board game we do have.

Game Space/Graph

First things first; we need a game space class to use as nodes for the graph. We didn't realize this until much later when it would be far too costly to go back and re-code, but the best way to go about it would be to have a game space parent class, with each sub-type being a child of the main class and including its own method for how to handle a player ending their turn on it, along with allowing it to be a physical object in-game so that we wouldn't need to individually mark every space's location on the map.

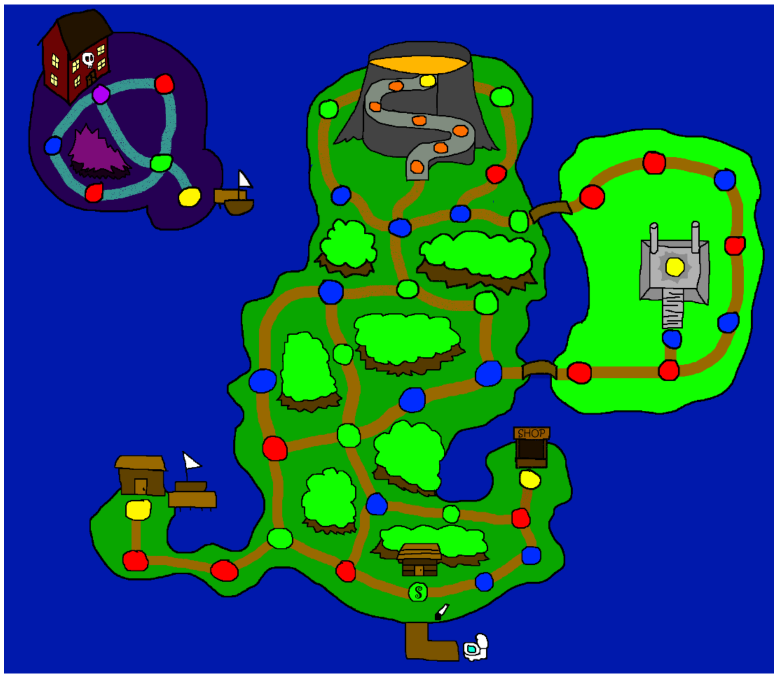
```
public GameSpace(int i, string t, int u, int d, int l, int r, float locX, float
locY) {
    index = i;
    type = t;
    connections[UP] = u;
    connections[DOWN] = d;
    connections[LEFT] = l;
    connections[RIGHT] = r;
    location.Set(locX, locY);
}
```
Constructor for GameSpace class (UP, DOWN, LEFT & RIGHT are pre-defined constant ints)

The main values needed for our version of the GameSpace class were:

- Const ints UP, DOWN, LEFT, RIGHT
- String type – is used to tell what type of space it is
- Int index – makes locating the spaces easier
- Int[] connections – an array of all the GameSpaces that the current one is connected to, found via their index
- Vector2 location – this is necessary in our current system to find the physical location of the space. Note that Vector2 is a structure made up of 2 float values.

As you can see on the constructor, all these values need to be inserted on the GameSpace's creation. All methods for the class are getters, as there is no need to update any values past initialization.

With the GameSpace class defined, we now must plan out the actual layout of the graph, that way when we begin to create the spaces. This will vary heavily depending on your own board's layout, but here is what ours looked like:



The map from our game

After this, creating the graph is as simple as creating all the necessary spaces for the board!

```
public BoardGraph(){
routes[0]  = new GameSpace(0, "Green", -1, -1, 2, 1, 0f, 0f);
routes[1]  = new GameSpace(1, "Blue" , -1, -1, 0, 3, 3.05f, 0.64f);
routes[2]  = new GameSpace(2, "Red"  , 4, -1, 8, 0, -3.28f, 1.17f);
routes[3]  = new GameSpace(3, "Blue" , 5, -1, 1, -1, 5.23f, 1.87f);
...
```

BoardGraph constructor, there's obviously more than 5 spaces, but it's basically the same line of code 50 times

The graph's functions are primarily getters; however, it does have added functions for moving a pointer across the graph:

```
public GameSpace goUp(GameSpace g){
  if(g.getUp() != -1)
  {
    return routes[g.getUp()];
  }
  else{
     return g;
  }
}
```

The go up function, which returns the node's upper connection

Items

Items are built using a parent item class, with each specific item having an overridden use method to decide what happens when the item is used.

```
Public class Item{
{
   Protected String name;
   Protected String itemDescription;
   Protected System.Random numPicker;
   Protected Sprite sprite;
   Protected SpriteRenderer spriteR;
}

Public void use(Contestant c){ }
```

The Item parent class

The values declared by the parent are:

- String name – shows the name of the item
- String itemDescription - a basic description of what the item does
- Random numPicker – This is used for any item that has an element of randomness to them
- Sprite sprite – the image that would appear in the player's inventory
- SpriteRenderer spriteR – what allows the sprite to be shown

It also comes with a use method that takes a Contestant as its arguments, allowing us to freely access the player's data with it. Below is an example of an item making use of these fields:

```
Public RAMItem()
{
   Name = "RAM";
   ItemDescription = "Teleports you to a random point on the island!";
}

Public void use(Contestant c)
{
   GameSpace newLocation = c.getLocation();
   NewLocation = c.island.get(numPicker.Next(50));
}
```

Code for the RAM item

Contestant Class

The contestant class is one of the most important and complex classes of the board game, being the actual player object that each person controls throughout the game.

```
Public class Contestant : Monobehavior
{
   Int bits;
   Public BoardGraph island;
   GameSpace current;
   Private int moveLimit;
   Private Int maxRoll;
   Public bool isTurn;
   Public bool isEliminated;
   Public System.Random die;
   Item[] items;
   Public RigidBody2D body;
}
```

Initializing variables for the Contestant class. Note: Monobehavior means that this is considered a physical object inside of Unity

```
Void Start()
{
   Die = new System.Random();
   maxRoll = 7;
   Bits = 0;
   isEliminated = false;
   isTurn = false;
   items = new Item[6];
   Body = new RigidBody2D();
   Island = new BoardGraph();
}
```

Start class for Contestant, this is called on the first frame of the object's existence and basically acts as a constructor

Values for the Contestant Class:

- ❖ Int bits – keeps track of how many bits a player has collected
- ❖ Int maxRoll – the maximum value the die can roll + 1. (we planned to include 0 as a possibility)
- ❖ Int moveLimit – decided by a die roll, limited how many spaces a player could move per turn
- ❖ GameBoard island – The graph from earlier, with all its connections pre-built
- ❖ GameSpace current – The player's current location on the graph
- ❖ RigidBody2D body - Required for the player to have their position in the scene manipulated
- ❖ Item[] items – An array of up to 6 items the player can carry with them at a time!
- ❖ Bool isTurn – Is true if it's this player's turn
- ❖ Bool isEliminated – Is true if the player has been eliminated!

Below are some important methods for the Contestant

```
Public void movement(){
   if (Input.GetKeyDown(KeyCode.W))
   {
      if(current.getUp() != -1)
      {
         MoveLimit--;
         Current = island.goUp(current);
      }
   }
...
```
The movement method, this is active whenever it's the player's turn and their moveLimit is greater than 0

```
Public void useItem(Item item){
   if (item == null)
   {
      Return;
   }
   If (item == <<Any item that you don't want player to manually use>>)
   {
      Return;
   }
   Item.use(this);
}
```
UseItem method, note that this is specifically for items the player would manually use

```
Public void spaceEffect(){
   Switch(current.getType()){
      Case "Blue": bits += 4;
         Break;
      Case "Red": bits -= 4;
         Break;
...
```

The spaceEffect method which uses a switch statement to find the space's type. Note that this is only necessary due to how we coded the gameSpace class, ideally this would look almost identical to the useItem method

Board Controller

The board controller class was never developed to a fully playable state, however we do have a general idea on how it would have functioned if we had the time to finish everything. First, the necessary variables the class would utilize:

```
LinkedList<GameObject> contestants = new LinkedList<GameObject>();
LinkedList<GameObject> losers = new LinkedList<GameObject>
Int numPlayers;
Int elimCnt;
Int roundCnt;
Bool eliminationRound;
```

Values for board controller class [note – Contestants are converted to an object prefab for the sake of having easily copied versions of the same thing]

Variables for BoardController:

- LinkedList<GameObject> contestants – would be where all the players still in the game are stored
- LinkedList<GameObject> losers – where all eliminated players are stored [this is necessary for them to still access minigames rather than being fully deleted from existence!]
- Int numPlayers – the number of players in the game
- Int elimCnt – how many round between eliminations (if elimCnt = 3, then every 3rd round would be an elimination round)
- Int roundCnt – keeps track of how many round of the game have been played
- Bool eliminationRound – is true when it's an elimination round!

Seeing as we never were able to properly test this class, the following code is very pseudo-code heavy but can provide a framework to build off of, or possibly and idea on how it can be coded.

```
Void update(){
   if (roundCnt % elimCnt == 0)
   {
      EliminationRound = true;
      Zombie.MiniGameList.randomMinigame(); //chooses/loads minigame for round
   }
   LinkedListNode<GameObject> current;
   for(current = contestants.First; current.Next != null; current = current.Next)
   {
      Current.Value.turnStart();
      While(current.Value.isTurn()){
         //This would just buffer until the player's turn ends
   }  }
      If(eliminationRound){
         //Pull up minigame loaded earlier for them to play, and eliminate the
lowest scoring player
```

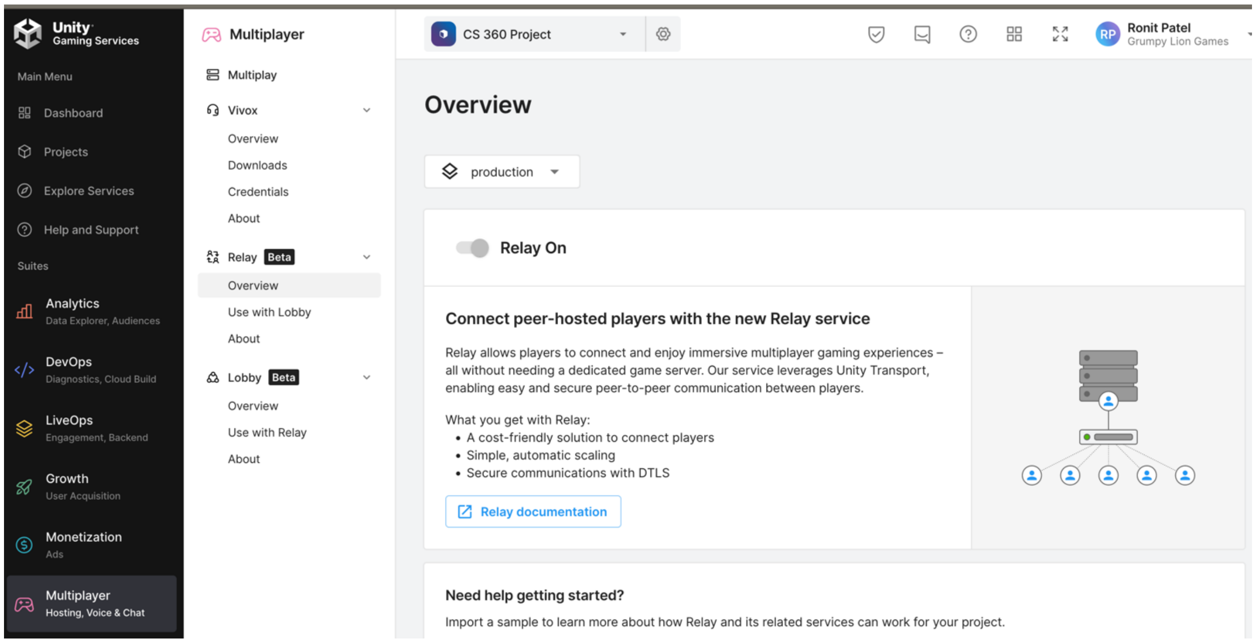
```
        }
    RoundCnt++;
}
```
Pseudo-code for the board controller's update class

Multiplayer

The network connection for the multiplayer, unfortunately, has not been completed due to a time constraint. There is a sample lobby solution created by Unity to see how it works. It is not meant to be a drag-and-drop solution. The sample has been included in our project, for you to use.

For multiplayer to work you must make sure, Lobby and Relay are enabled. There is an option called Vivox which is in-game chat, but this is completely optional for this project. Since this is meant to be played in the classroom.

It should already be enabled by default. If it's not, then go to unity dashboard and go to Multiplayer. Then select either Relay or Lobby.



Overview

Relay On

Connect peer-hosted players with the new Relay service

Relay allows players to connect and enjoy immersive multiplayer gaming experiences – all without needing a dedicated game server. Our service leverages Unity Transport, enabling easy and secure peer-to-peer communication between players.

What you get with Relay:
- A cost-friendly solution to connect players
- Simple, automatic scaling
- Secure communications with DTLS

Relay documentation

Need help getting started?

Import a sample to learn more about how Relay and its related services can work for your project.

Then select About and click Get Started and follow the steps. Follow the same step for Relay and/or Lobby.

The Multiplayer Button is disabled by default, you must make it interactable in the inspector tab.

Also add all multiplayer related scenes in the build settings.

Unity's Documentation on Networking

This link is the Sample Lobby on GitHub which gives a detailed explanation on what each part of the sample does.

How you finish the multiplayer is on you.

Good Luck, trust me you're going to need it.

Supporting Scripts

Music Player

Audio Source

The audio source is a unity object that can be easily used to play mp3 file types. All you need to do is drag in a music clip into it you make the music play on a scene. All the of scripts I had written are for making the audiosource continues to exist after switching scenes and switching the music on a scene load.

The first link from unity on how the audio source works (not my changes)

https://docs.unity3d.com/ScriptReference/AudioSource.html

Save Me

The SaveMe script purpose is to keep the audio source from deleting itself when a new scene is load. It stores a list of audio sources and destroys an audio source there is more than one audio sources. Then makes the item unable to be destroy using the unity built in funtction DontDestoryOnLoad.

```
void Awake() {
    GameObject[] musicObj = GameObject.FindGameObjectsWithTag("GameMusic");

if(musicObj.Length > 1) {
        Destroy(this.gameObject);
    }
        DontDestroyOnLoad(this.gameObject);
}
```

Menu Mixer Controller

The MenuMixerController script purpose is to let the user change the music volume in the options menu using the slider. Unity menu mixers uses logs to set the music volume. The function setVolume simply set the volume to the position of the slider.

```
[SerializeField] private AudioMixer MenuMixer;

public void setVolume(float sliderValue)
{
    MenuMixer.SetFloat("MenuVolume", Mathf.Log10(sliderValue) *20);
}
```

Audio Manager

The AudioManager script purpose is to let the music be able to be change to another. The function ChangeBGM takes in an audio clip and check to see if it is playing and if not changes the music to the audio clip.

```
public AudioSource BGM;

public void ChangeBGM(AudioClip music)
{
    if (BGM.clip.name != music.name) {
        BGM.Stop();
        BGM.clip = music;
        BGM.Play();
        }
}
```

Switch Music On Load

The SwitchMusicOnLoad script purpose is to takes in an audio clip and use the ChangeBGM function to change the music. This script is on the prefab NewMusic and all you need to do I drag the prefab on the new scene and add the music clip.

```
public GameObject auidioMan;
void Start(){
    if(FindObjectOfType<AudioManager>()) {
        return;
    } else {
        Instantiate(auidioMan, transform.position, transform.rotation);
    }
}
```

Audio Manager Check

The AudioManagerCheck script purpose is to look for a AudioManager and if there isn't one, its makes a new one. This script is on the prefab AMCheck and can be added to any new scenes and that it.

```
public GameObject auidioMan;
void Start(){
    if(FindObjectOfType<AudioManager>()) {
        return;
    } else {
        Instantiate(auidioMan, transform.position, transform.rotation);
    }
}
```

Zombie

The Zombie script is a specialized type of game controller script. It primarily focuses on the Discord Rich Presence feature, as well as data relating to the minigames such as *MiniGameLister* and *PlayerStats*. This script is attached to the first scene shown to the player, *MainMenu* to initialize these core features.

NOTE: The attached game object is a DontDestroyOnLoad gameobject. These objects pursist their host scene. So, when the *MainMenu* is unloaded, the host game object will continue to exist and execute the *Zombie* script.

Variables

Public

- isSolo: a check to determine if the user is playing by themselves or against other players. It does not check if the player is online.
- MiniGame: a property that will return the current minigame that is being played. When a new value is set, it will update the *DiscordController's ScreenName* property.
- DiscordController: the current discord controller for the Rich Presence feature.
- PlayerStats: the current player's history. It is used to save and access stats during runtime of the various minigames.
- MiniGameList: provides an access point for the MiniGameList.

Private

- created: a check to determine if the script has been instantiated before. This is used to prevent more than one Zombie running.
- _mg: the private variable used for the property MiniGame (See Properties).

Methods

This class has three different methods, *Awake*, *Update*, and *OnApplicationQuit*. Only *Awake* will be discussed. The other two call the DiscordController's *Update* and *OnApplicationQuit* methods.

Awake

```
private void Awake() {
    if (!created) {
        DontDestroyOnLoad(gameObject);
        created = true;
        CurrentProfileStats = new PlayerStats();
        DiscordController = new DiscordController();
        MiniGameList = new MiniGameLister();
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
}
```

This checks that there are no Zombie scripts running already, and then acts as the constructor. It initializes all the variables not set during runtime or editor and loads the host scene.

Mini Game Data

This class simply allows developers to describe each minigame to the players. It provides information to the MiniGameLister to be cached for use by the Minigame Load Screen.

Mini Game Lister

The MiniGameLister script obtains a list of all scenes in the game for lookup. It quickly loads and unloads each minigame to cache minigame data using the MiniGameData class. This information, including Minigame Name and descriptions, can be accessed by any script that has access to Zombie. The scene data and minigame data are stored in a subclass MGLScene, which is not accessible by outside classes.

Minigames are indexed as “Assets/Scenes/[Type]/([Category]/…/[MinigameName].unity)”, where Type, Category, and MinigameName are cached in the lister for future use. Anything past Type is only cached for minigames.

NOTE: This script is part of the Zombie component, which attached game object is a DontDestroyOnLoad gameobject.

Variables

- scenes: a list of all scenes in the game, including menus, boards, and minigames.
- desiredType: the name of the scene type to be categorized. Should only be changed if the script is being rewritten.
- desiredTypeList: a list of all scenes within the desired type. Therefore, a list of all minigames.
- givenCategories: a list of all subcategories of the desiredType. Therefore, topics for minigames.
- categoryLists: a 2d array of minigames, indexed by their givenCategories.
- CurrentCategory: an optional variable for use in multiplayer if there are enough games in one category to play only within that category.

Methods

This class has eight getters that will not be discussed: GetSceneHowToPlay, GetSceneDataStructureInfo, GetCurrentScenes, GetListofCategories, GetCategoryFromSceneName, GetSceneNamesFromCategories, LookupScene, LookupCategory.

The remaining five methods (CaptureScenes, GenerateCategories, PrepareMinigameData, SetSceneData, randomMinigame) are discussed below. The first three are directly called in the constructor, and the SetSceneData is indirectly called from PrepareMinigameData.

Capture Scenes

```csharp
private void CaptureScenes() {
    if(scenes == null) {
        var sceneNumber = SceneManager.sceneCountInBuildSettings;
        scenes = new MGLScene[sceneNumber];
        Regex r = new Regex(@"Assets\/Scenes\/(?<type>[^\/]+)\/(?<category>[^\/]+)((\/[^\/]+))?\/(?<name>[^\/]+).unity");
        for(int i = 0; i < scenes.Length; ++i) {
            scenes[i] = new MGLScene();
            scenes[i].index = i;
            scenes[i].path = SceneUtility.GetScenePathByBuildIndex(i);
            Match match = r.Match(scenes[i].path);
            if(match.Success) {
                scenes[i].type =    match.Groups["type"].Value;
                scenes[i].category = match.Groups["category"].Value;
                scenes[i].name =    match.Groups["name"].Value;
            }
        }
    }
}
```

This method captures a list of all scenes and stores them in the scenes' variables. For minigames, the type, category and name of each scene is also stored within the corresponding MGLScene.

NOTE: This method uses regex, and should not be edited without understanding of all regex used.

Generate Categories

```
private void GenerateCategories() {
    HashSet<string> categories = new HashSet<string>();
    foreach(MGLScene s in scenes)
        if(s.type.Equals(desiredType))
            categories.Add(s.category);
    givenCategories = new string[categories.Count];
    categories.CopyTo(givenCategories);
    Array.Sort(givenCategories, 0, givenCategories.Length);
    desiredTypeList = new ArrayList();
    categoryLists = new ArrayList[givenCategories.Length];
    for(int i = 0; i < givenCategories.Length; ++i)
        categoryLists[i] = new ArrayList();
    foreach(MGLScene s in scenes)
        if(s.type.Equals(desiredType)) {
            desiredTypeList.Add(s.index);
            for(int i = 0; i < givenCategories.Length; ++i)
               if(s.category.Equals(givenCategories[i])) {
                    categoryLists[i].Add(s.index);
                    break;
               }
        }
}
```

This method generates a set of all Minigame subcategories from the MGLScene of each minigame. This set is converted into an array of category names. After this, the minigame scenes are added to the desiredType list as well as the list for their givenCategory.

Prepare Minigame Data

```
private void PrepareMinigameData(string sceneType) {
    foreach(MGLScene s in scenes)
        if(s.type.Equals(sceneType)) {
            SceneManager.LoadScene(s.name, LoadSceneMode.Additive);
            SceneManager.UnloadSceneAsync(s.name);
        }
}
```

This method loads all Minigame scenes and unloads them all as soon as possible. This runs all awake methods and constructors in each Minigame scene, which allows them to produce their descriptions and other data.

Set Scene Data

```
public void SetSceneData(string sceneName, string sceneHowToPlay, string
sceneDataStructureInfo) {
    int i = LookupScene(sceneName);
    if(i != -1) {
        scenes[i].howToPlay = sceneHowToPlay;
        scenes[i].dataStructureInfo = sceneDataStructureInfo;
    }
}
```

This method is called by each minigame to cache data. Simply, if the corresponding MGLScene is found in the list, update the MGLScene with the data provided by the MiniGameData component.

Random Minigame

```
public string randomMinigame() {
   int minigameIndex = UnityEngine.Random.Range(0, desiredTypeList.Count);
   int sceneIndex = (int)desiredTypeList[minigameIndex];
   return scenes[sceneIndex].name;
}
```

This method selects a random Minigame scene stored in the MiniGameLister. It returns the name of the minigame to be loaded.

Player Stats

Made up of many different classes, this group of objects relate to the storing, management, and access of data points collected during regular gameplay. All classes in this group use the *Serializable* tag, allowing them to be serialized and saved alongside primitive data fields.

- PlayerStats: the entry point for access with the other classes within this group. It is made up exclusively of a constructor and a property.
- Stat: the object that holds all data relating to a particular minigame's collected data points.
- PlayerIO: this class exclusively handles the serialization and deserialization of the game data.
- NoSaveFileException: a custom exception thrown by PlayerIO when an error occurs due to a deserialization failure caused by the save file not existing on a client's computer. It is only used for these classes and will not be discussed in this document.

This set also includes one namespace, called *GameLogging*, containing two additional classes that can handle exclusively gameplay logging.

- GameLogs: a custom list that fires an event anytime that a value is added to the list. It stores only *Log* or *Log* child objects.
- Log: a class containing data points specific to a single playthrough of a minigame.

Player Stats

This class constructs a single access point for the save data relating to the minigame play history.

Variables

- Stats: A dictionary of dictionary, contains a public accessor for the player stats. It is formatted so it goes Minigame type, Minigame name. This will return the *Stat* object for the minigame.
    - PlayerStats.Stats[DATA_STRUCTURE][MINIGAME_NAME]

```
public Dictionary<string, Dictionary<string, Stat>> Stats { get; private set; }
```

Constructor

```
public PlayerStats() {
    try {
        Stats = PlayerIO.LoadData().Stats;
        Debug.Log("Player history detected");
    } catch (NoSaveFileException) {
        MiniGameLister MGL = new MiniGameLister();
        Stats = new Dictionary<string, Dictionary<string, Stat>>();
        foreach(string Category in MGL.GetListofCategories()) {
           Dictionary<string, Stat> DataStruct = new Dictionary<string, Stat>();
           foreach(string MiniGame in MGL.GetSceneNamesFromCategory(Category)) {
              DataStruct.Add(MiniGame, new Stat());
           }
           Stats.Add(Category, DataStruct);
        }
        PlayerIO.SaveData(this);
        Debug.Log("New Player Record Created");
    }
}
```

This will attempt to read a previously created save data file utilizing the *LoadData* method from *PlayerIO*. If there is not a previously created save file, it then creates the *Stats* data field. By creating a *MiniGameLister* object, it allows the code to loop through all data structures and find their related minigames.

Stat

This class holds all data fields related to the minigame as a whole and not individual playthroughs. It also contains a game log containing the history of individual playthroughs and their data points.

NOTE: Not all data fields will be used on all minigames.

Variables

- BestTime: the best time recorded for this minigame.
- BestScore: the highest score recorded for this minigame.
- GamesPlayed: the total number of times that a minigame has been played.
- GamesWon: the number of times that the minigame has been won, discounting practice.

- GamesLost: the number of times that the minigame has been lost, discounting practice.
- TimesPracticed: the total number of times that a game has been practiced.
- TimesPlayed: the total number of times that a game has been played, discounting practice.
- GameLog: a record of previous playthroughs of this minigame containing records relevant only to a single playthrough.

Constructor

```
public Stat() {
    GameLog.ListChanged += (sender, e) => {
        if(e.TimeTaken < BestTime) { BestTime = e.TimeTaken; }
        if(e.Score > BestScore) { BestScore = e.Score; }
        if (e.Win && !e.IsPractice) { GamesWon++; } else { GamesLost++; }
        if (e.IsPractice) { TimesPracticed++; } else { TimesPlayed++; }
        GamesPlayed++;
    };
}
```

This constructor only captures *GameLog*'s *ListChanged* event. In this event, which is assigned via a lambda method, simply checks if the most recent game beats the best score/time and if the most recent game ended in a win or was a practice game.

Player IO

This file IO class is only responsible for storing and retrieving the *PlayerStats* object. It utilizes the *BinaryFormatter* object to serialize and deserialize the data.

Variables

- Path: the file location that will store the data from our *PlayerStats* object.
- Serializer: the *BinaryFormatter* that will serialize and deserialize our object.

TIP: The method `Application.GetPersistentDataPath()` will return a path that will reference the specific file during runtime. Use this to avoid runtime errors caused by different OS' file systems.

Methods

Save Data

```
public static void SaveData(PlayerStats player) {
    using FileStream file = new FileStream(Path, FileMode.Create);
    Serializer.Serialize(file, player);
}
```

This overwrites our previously saved file (or creates a new one) at the specified path and then saves the serialized data in the file.

TIP: The reserved word `using` will automatically close a specified object when the method or block finishes execution. In our case, the `using` statement will automatically close the file that we created when we exit the method without having to specifically call `File.Close()`!

Load Data

```
public static PlayerStats LoadData() {
    if (File.Exists(Path)) {
       using FileStream file = new FileStream(Path, FileMode.Open);
       return Serializer.Deserialize(file) as PlayerStats;
    } else {
        Debug.Log("File was not found " + Path);
        throw new NoSaveFileException();
    }
}
```

Load Data will check if there is a file in our save data location, if there is not a file, it throws the *NoSuchFileException* discussed earlier in this section; otherwise, it will deserialize and return the *PlayerStats* object contained in the file.

TIP: The reserved word `as` is equivalent to casting the specified object. In our case, the `as` statement will cast the value returned by *Deserialize* (an *Object*) and cast it as a *PlayerStats*. Meaning it is equivalent to `(PlayerStats)Serializer.Deserialize(file)`. It is a quick, clean, and handy way to cast objects in C#.

Game Logs

A custom child class of the generic `List<T>` class. This class adds a custom event called *ListChanged* which is fired when an item is added to the List.

Variables

- ListChanged: sends the last log added to the list to the event hook.

Add

```
public new void Add(Log item) {
    if(ListChanged != null && item != null) { ListChanged.Invoke(this, item); }
    base.Add(item);
}
```

This method hides the parent's *Add* method and instead checks if the *ListChanged* event can and should be fired, it then uses the parent's *Add* method to continue normal operation.

TIP: The new reserved word can be used to hide a method that is not virtual (overridable). This can be used with both methods and variables.

Log

This class is used to store all data in reference to a single playthrough of a minigame. It consists of only variables.

NOTE: Not all data fields will be used on all minigames.

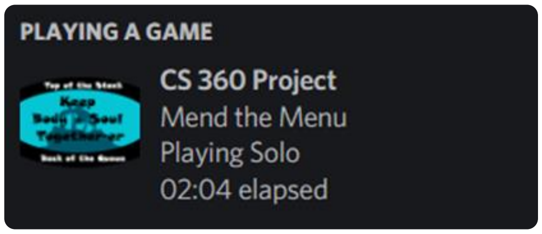
Variables

- TurnsUsed: the number of turns used in this game
- ErrorsMade: the number of wrong moves in this game
- Score: the score obtained in this game
- TimeTaken: the elapsed time between the start and end of the game
- IsTeamGame: a check if this is a solo game or a game with multiple players

- IsPractice: a check if this was a practice game or not
- Win: a check if this game resulted in a player win or not

Discord Controller

During gameplay, our game dynamically updates your Discord Rich Presence (RP) (that is, if you have a Discord account). This status is created using the Discord SDK and controlled by this class .


PLAYING A GAME

CS 360 Project
Mend the Menu
Playing Solo
02:04 elapsed

Figure 1: Discord Controller Output

Variables

Private Variables

- _screenName: a string that stores the current screen that the player is on
- discord: the Discord object from the official Discord SDK
- GameName: the game name that will be displayed as being played. This should not be changed.

Public Variables

- ScreenName: the public accessor for _screenName. If a new value is set for ScreenName, it instead sets _screenName to that value and updates the discord status.

Methods

Constructors

- DiscordController(): tries to load Discord and update the user's rich presence.

Private Methods:

- UpdateDiscord(string Value): Attempts to update the user's rich presence.
- InitializeDiscord() – bool: Returns true if the discord integration loaded properly and false if the discord integration failed.

Public Methods

- OnApplicationQuit(): Removes the user's rich presence and closes discord.
- Update(): Updates the user's rich presence or closes discord on error.
- Unix(DateTime) – long: calculates the time elapsed form the unix epoch (January 1, 1970, 12:00:00). This is used to calculate the time the user has spent in our program.

Code

Constructor

```
public DiscordController(){
    if(InitializeDiscord())
        ScreenName = "Main Menu";
}
```

This script tries to initialize discord, from there, the code then sets the screen name to the Main Menu.

NOTE: The `ScreenName` property will automatically fire the `UpdateDiscord` method whenever it is changed.

Update Discord

```
private void UpdateDiscord(string value) {
   if(discord != null || InitializeDiscord()) {
       Activity Game = new Activity() {
          ApplicationId = 939698372485476403,
          Name = GameName,
          State = value,
          Details = Zombie.isSolo ? "Solo Mode" : "Group Mode",
          Timestamps = { Start = Unix(DateTime.UtcNow), },
          Assets = {
              LargeImage = "logo",
              LargeText = GameName,
           }
       };
       discord.GetActivityManager().UpdateActivity(Game, (result) => {
           if (result == Result.Ok) {
                Debug.Log("Discord Activity Updated!");
           } else {
                Debug.Log("Discord Activity Update Failed: " + result);
           }
       });
   }
}
```

This method first creates a new `Activity` struct called `Game`, which is contained in the provided Discord SDK. `Game` then is populated with our game's name, the current screen (State), if the player is playing Solo or with a Group (Details), resets the timer, and displays our game's logo. `Game` is then pushed to the currently logged in Discord User only if Discord is open on the client computer.

Initialize Discord

```
private bool InitializeDiscord(){
   try {
        discord = new Discord.Discord(939698372485476403,
            (ulong)CreateFlags.NoRequireDiscord);
        Debug.Log("Discord Loaded Successfully");
        return true;
   }catch(ResultException) {
         Debug.Log("Discord load failed. User probably has not opened Discord.");
   }
   return false;
}
```

This method tries to initialize the Discord Rich Presence Controller (RPC) and if it is successfully loaded, it returns true allowing the other methods relying on the RPC to continue. Otherwise, it returns false and forcing the RPC reliant methods to terminate prematurely.

On Application Quit

```
public void OnApplicationQuit() {
    if (discord != null) {
        discord.Dispose();
        discord = null;
        Debug.Log("Discord Controller Closed");
    }
}
```

This event is fired before the application quits (in editor and build) and it then tries to nicely dispose the RPC which will remove our custom RP from the logged in user if the RPC was initialized during gameplay or load.

Update

```
public void Update() {
    if(discord != null){
       try{
           discord.RunCallbacks();
       }catch(ResultException) {
           Debug.Log("Discord Update Failed. Gracefully closing the Discord
               Controller.\nUser may have closed Discord during the game.");
           discord.Dispose();
           discord = null;
       }
    }
}
```

Attempts to run the required `RunCallbacks` method of the RPC manager. If this fails, it then gracefully closes the RPC manager and removes our RP from the logged in Discord User.

Unix

```
public static long Unix(DateTime time) {
    DateTime UnixEpoch = new DateTime(1970, 1, 1);
    return (long) time.Subtract(UnixEpoch).TotalSeconds;
}
```

This method manually calculates the time elapsed since the Unix Epoch (Midnight of January 1st, 1970). It takes in the current time and calculates the time span from the Unix Epoch and gets the total number of seconds elapsed from that date also known as Unix time (hence the name).

Selection Button Script

This class simply provides utility via ButtonGenerator. It updates the button text and adds a listener to the onClick event.

Button Generator

This component gets either a list of categories or a list of minigames in each category from the MiniGameLister. It creates a button for each item in the list and sets up callbacks for the buttons to run when pressed. This is the core of the Single Player menu system, and dynamically generates more buttons as more minigames and categories are added.

This script is attached to the ButtonField prefab, which defines the space the buttons will fill.

This generator can list other items by adding them to the SelectionType enum and making a new selectionType check in the Start method.

Variables

- NextScene: The next menu to be loaded once the player makes a selection.
- SelectionType: whether the script is looking up a list of categories or scenes.
- buttonPrefab: a prefab button game object containing the SelectionButtonScript.
- Callback: the callback to be added to each button generated.
- options: the list of strings given by the MiniGameLister

Methods

This class has only one method, Start, which sets up the whole button field.

Start

```
void Start() {
    if(selectionType == SelectionType.Category) {
        options = Zombie.MiniGameList.GetListofCategories();
        callback = (selection) => {
            Zombie.MiniGameList.CurrentCategory = selection;
            SceneManager.LoadScene(NextScene);
        };
    } else if(selectionType == SelectionType.Scene) {
        options = Zombie.MiniGameList.GetCurrentScenes();
        callback = (selection) => {
            Zombie.MiniGame = selection;
            Zombie.IsSolo = true;
            SceneManager.LoadScene(NextScene);
        };
    }
    float height = GetComponent<RectTransform>().sizeDelta.y;
    float width =  GetComponent<RectTransform>().sizeDelta.x;
    float delta = height / options.Length;
    Vector3 position = new Vector3(0, height / 2, 0);
    foreach(string opt in options) {
        GameObject newButton = Instantiate(buttonPrefab, position, 
Quaternion.identity);
        newButton.transform.SetParent(transform, false);
        newButton.GetComponent<RectTransform>().sizeDelta = new Vector2(width, 
width / 5);
        newButton.GetComponent<SelectionButtonScript>().selectionData = opt;
        newButton.GetComponent<SelectionButtonScript>().callback = callback;
        position = new Vector3(0, position.y - delta, 0);
    }
}
```

First, this method acquires the options list and generates the callback lambda based on the selectionType. It then calculates the size of the ButtonField and spacing required for each button. Finally, it instantiates each button, providing its location, rotation, button text, and callback.

Timer

This is a simple class that will create a dynamically displaying stopwatch. It is used in the UI Prefabs and in many classes in which time is a critical component, such as in Game Win/Game Loss.

Variables

- TimerLabel: a reference to the TextMeshProUGUI label that will display the elapsed time.
- AutoRun: a bool checking if the timer should start immediately.
- Time: a float showing the amount of time elapsed.
- Running: a bool checking if the timer is currently running.

Methods

Start

```
void Start() {
    TimerLabel.text = "00:00.000";
    if (AutoRun) { running = true; }
}
```

This method resets all the timer.

Stop Timer

```
public void StopTimer() { running = false; }
```

This method stops the timer.

Start Timer

```
public void StartTimer() { running = true; }
```

This method starts the timer.

Update

```
protected virtual void Update() {
    if (running) {
        try {
            Time += UnityEngine.Time.deltaTime;
            float minutes = Mathf.FloorToInt(Time / 60);
            float seconds = Mathf.FloorToInt(Time % 60);
            float milliseconds = (Time % 1) * 1000;
            string timedisplay = $"{minutes,2}:{seconds,2}.{milliseconds,3:F0}";
            TimerLabel.text = timedisplay.Replace(" ", "0");
        } catch {
            running = false;
        }
    }
}
```

This method adds the time elapsed and updates the time elapsed and displays the new time.

Count Down

NOTE: CountDown is a child of timer. Only unique variables and methods will be mentioned here.

Variables

- NumberOfSeconds: the number of seconds to count down from.
- GameLoss: a reference to the scene's gameloss object.
- TimeElapsed: the amount of time elapsed since the timer is running.

Methods

Update

NOTE: CountDown's updated function overrides Timer's update function.

```
protected override void Update() {
    if (running) {
        try {
            Time -= UnityEngine.Time.deltaTime;
            TimeElapsed += UnityEngine.Time.deltaTime;
            if(Time <= 0) {
                GameLoss.Show();
                TimeElapsed = NumberOfSeconds - Time;
                TimerLabel.text = "00:00.000";
                StopTimer();
                SaveData();
                return;
            }
            float minutes = Mathf.FloorToInt(Time / 60);
            float seconds = Mathf.FloorToInt(Time % 60);
            float milliseconds = (Time % 1) * 1000;
            string timedisplay = $"{minutes,2}:{seconds,2}.{milliseconds,3:F0}";
            TimerLabel.text = timedisplay.Replace(" ", "0");
        } catch {
            running = false;
        }
    }
}
```

This method updates time elapsed counters and checks if the time has run out. If so, it shows the game over screen and stops the timer. Otherwise, it updates the timer display.

Save Data

NOTE: This method is not implemented and will not be discussed.

Load Screen

This class is responsible for the logic when loading a minigame. It loads an image with the correct clip of gameplay and game controls. It will also read a minigame's `minigamemetadata`.

Variables

- RandomName: a private string that stores the team's name, randomly determined by the struct TName.
- MiniGame: a public string that stores the minigame that is to be loaded next.
- GameName: a UI component that will display the minigame name.
- DataStructure: a UI componenent that will display how the minigame relates to data structures.
- HowTo: a UI componenet that will display how to play the minigame
- TeamName: a UI input component that will display the team's name.
- ReadyButton: a UI input component that will load the minigame or communicate with the server when a user is ready to start.
- ErrorText: a UI componenet that will display an error, such as an illegal team name.
- HowToPlay: an image that will display the controls needed.
- miniGame: an image that will display the snapshot of gameplay.

Methods

Start

```
void Start() {
    if (!string.IsNullOrEmpty(Zombie.MiniGame)) { MiniGame = Zombie.MiniGame; }
    RandomName = TName.SelectRandomName();
    TeamName.text = RandomName;
    HowToPlay.sprite = Resources.Load<Sprite>("How To/" + MiniGame);
    miniGame.sprite = Resources.Load<Sprite>("Minigame Snaps/" + MiniGame);
    ErrorText.gameObject.SetActive(false);
    TeamName.onEndEdit.AddListener(delegate {
    if (TeamName.text.Length > 12 || TeamName.text.Length < 4
        || !TName.CheckLegality(TeamName.text) ||
        string.IsNullOrWhiteSpace(TeamName.text) || (TeamName.text.Length –
            Count(TeamName.text, ' ') < 4)) {
                TeamName.text = RandomName;
                ErrorText.gameObject.SetActive(true);
        } else {
            ErrorText.gameObject.SetActive(false);
        }
    });
    ReadyButton.onClick.AddListener(delegate {
        if (ErrorText.gameObject.activeSelf) { return; }
        SceneManager.LoadScene(MiniGame);
    });
    GameName.text = MiniGame;
    DataStructure.text = Zombie.MiniGameList.GetSceneDataStructureInfo(MiniGame);
    HowTo.text = Zombie.MiniGameList.GetSceneHowToPlay(MiniGame);
}
```

This will check populate the image and text fields with images or text relating to the minigame being loaded. It then adds an event where any time the team's name is altered, it will be checked for legality. And it will add an event for the ready button that will launch the game.

Count

```
int Count(string text, char value) {
    int count = 0;
    foreach(char c in text.ToCharArray()) {
        if(c == value) { count++; }
    }
    return count;
}
```

This will return the number of times that a given string contains a character. It is used to count the number of white spaces in the team's name.

Pause Screen

NOTE: The enum PauseState will not be discussed in the documentation.

Variables

- Timer: a reference to the minigame's timer element.
- Restart: a reference to the button that will restart the minigame.
- Resume: a reference to the button that will resume the minigame.
- MainMenu: a reference to the button that will load the minigame selection screen
- Host: the gameobject that will host this script.
- PauseKey: the keycode that will pause the minigame.
- currentPauseState: the PauseState that the minigame is currently on.

Methods

Awake

```
private void Awake() {
    Restart.onClick.AddListener(delegate {
        Debug.Log("Restart Clicked");
        SceneManager.LoadScene(gameObject.scene.name);
    });
    Resume.onClick.AddListener(delegate {
        Debug.Log("Resume Clicked");
    });
    MainMenu.onClick.AddListener(delegate {
        Debug.Log("Main Menu Clicked");
        SceneManager.LoadScene("SPCategoryMenu");
    });
}
```

This method will bind all events to their respective buttons.

Resume Pressed

```
public void ResumePressed() {
    currentPauseState = PauseState.EndPause;
    UpdatePauseMenu();
}
```

This function exits the pause menu as if the player pressed the PauseKey.

Update

```
void Update() {
    if (Game.activeInHierarchy) { return; }
    if(Input.GetKeyDown(PauseKey) == ((int)currentPauseState % 2 == 0)) {
        currentPauseState = (PauseState)(((int)currentPauseState + 1) %
        (int)PauseState.EPS_StateCount);
            UpdatePauseMenu();
    }
}
```

This method increments the current pause state if conditions are correct. This system allows the pause menu to be activated and deactivated with a single button. If pause starts or ends, UpdatePauseMenu is called to perform logic. Pause functionality is disabled until the initial countdown finishes and disabled again when the game is won or lost.

Update Pause Menu

```
private void UpdatePauseMenu() {
    switch(currentPauseState) {
        case PauseState.StartPause:
            Timer.StopTimer();
            Host.SetActive(true);
            break;
        case PauseState.EndPause:
            Timer.StartTimer();
            Host.SetActive(false);
            break;
        default:
            break;
    }
}
```

This method is called internally to perform the actual logic of the pause menu. When a pause starts, the timer is stopped, and the menu is shown. When a pause ends, the timer continues from where it stopped, and the pause menu is hidden.

Game Win

Variables

- Timer: a reference to the scene’s timer element
- Screen: an image reference that will prevent the user from doing actions in the minigame.
- TimeWin: a text mesh that will display the amount of time elapsed during the minigame.
- Win: a text mesh that will display a message to the user.
- Retry: a button that will reload the scene.
- Next: a button that will load the minigame selection screen.
- DisplayDelay: the amount of time before the screen reaches full opacity.
- DetectWin: a bool representing if the player has won the game.
- group: a canvas group reference that will allow for opacity of the screen.

Methods

Awake

```
private void Awake() {
    Screen.gameObject.SetActive(false);
    TimeWin.gameObject.SetActive(false);
    Win.gameObject.SetActive(false);
    group = GetComponent<CanvasGroup>();
    group.alpha = 0f;
}
```

Hides the screen.

Start

```
private void Start() {
    if (!Zombie.IsSolo) { Retry.enabled = false; Next.transform.position = new Vector3(0f, -250f, 0f); }
    Retry.onClick.AddListener(Retry_OnClick);
    Next.onClick.AddListener(Next_OnClick);
}
```

Binds all events necessary for the screen depending on if the player is in a multiplayer game.

Show

```
public void Show() {
    DetectWin = true;
    Invoke(nameof(ShowGameOver), DisplayDelay);
}
```

Shows the game win screen.

Show Game Over

```
private void ShowGameOver() {
    TimeWin.text = Timer is CountDown ?
        $"Time: {((CountDown)Timer).TimeElapsed}" :
        $"Time: {Timer.TimerLabel.text}";
    Timer.StopTimer();
    float alphacounter = 0f;
    Screen.gameObject.SetActive(true);
    TimeWin.gameObject.SetActive(true);
    Win.gameObject.SetActive(true);
    while (group.alpha < 1f) {
        alphacounter += .001f;
        group.alpha = alphacounter;
    }
}
```

This method gets the number of seconds elapsed in the minigame and displays the game over screen.

Retry On Click

```
private void Retry_OnClick() {
   SceneManager.LoadScene(gameObject.scene.name);
}
```

This method reloads the current scene.

Next On Click

```
private void Next_OnClick() {
    SceneManager.LoadScene("SPCategoryMenu");
    Debug.Log("Next Clicked");
}
```

This method loads the minigame selection screen.

Game Loss

Variables

- Screen: an image reference that will prevent the user from doing actions in the minigame.
- Loss: a reference to the message that will display when the screen is shown
- LossText: a reference to the game over message that will display when the screen is shown.
- Retry: a button that will reset the minigame. Hide this when in multiplayer.
- Next: a button that will progress back to the game board or selection screen.
- DisplayDelay: the amount of time before the screen is shown.
- Message: the message to display when in game over screen is called.
- DetectWin: a bool representing if the player won the minigame.
- group: The canvas group that will make a gradual change in opacity.
- _Message: the message wrapper that contains the message reference.

Methods

Awake

```
private void Awake() {
    Loss.text = Message;
    Screen.gameObject.SetActive(false);
    LossText.gameObject.SetActive(false);
    Loss.gameObject.SetActive(false);
    group = GetComponent<CanvasGroup>();
    group.alpha = 0f;
}
```

Hides the screen on load.

Start

```
private void Start() {
    if (!Zombie.IsSolo) { Retry.enabled = false; Next.transform.position = new
Vector3(0f, -250f, 0f); }
    Retry.onClick.AddListener(Retry_OnClick);
    Next.onClick.AddListener(Next_OnClick);
}
```

This method binds all events necessary depending on if the player is playing multiplayer or not.

Show

```
public void Show() {
    DetectWin = true;
    Invoke(nameof(ShowGameOver), DisplayDelay);
}
```

This method will show the game over screen.

Show Game Over

```
private void ShowGameOver() {
    float alphacounter = 0f;
    Screen.gameObject.SetActive(true);
    LossText.gameObject.SetActive(true);
    Loss.gameObject.SetActive(true);
    while (group.alpha < 1f) {
        alphacounter += .001f;
        group.alpha = alphacounter;
    }
}
```

This method will slowly fade in the game over screen.

Retry On Click

```
private void Retry_OnClick() {
    SceneManager.LoadScene(gameObject.scene.name);
}
```

This method reloads the current scene.

Next On Click

```
private void Next_OnClick() {
    SceneManager.LoadScene("SPCategoryMenu");
    Debug.Log("Next Clicked");
}
```

This method takes the player back to the minigame selection screen.

Game Start

This class prevents a player from doing any activities in a minigame until the minigame has successfully loaded, and in multiplayer situations, starts all players at the same time.

Variables

- Display: an image reference that will hide the background.
- Title: a text label that displays the title text.
- CountDown: a text reference that will display the amount of time before the game starts.
- CountDownDuration: an integer representing the number of seconds before the game starts.
- Timer: a reference to the timer in the minigame.

Methods

Start

```
void Start() {
   CountDown.text = CountDownDuration.ToString();
   StartCoroutine(CountDownFire());
}
```

This method starts the count down.

Count Down Fire

```
IEnumerator CountDownFire() {
    while (CountDownDuration > 0) {
        CountDownDuration--;
        CountDown.text = CountDownDuration.ToString();
        if(CountDownDuration == 0) { CountDown.text = "GO!"; }
        yield return new WaitForSeconds(1f);
    }
    Display.gameObject.SetActive(false);
    Title.gameObject.SetActive(false);
    CountDown.gameObject.SetActive(false);
    Timer.StartTimer();
}
```

Decrement the number of seconds before the game starts and when the countdown reaches zero, starts the minigame after hiding the host.

Important Notes and References

Definitions

Object Types

While C# has a lot of the same code structures as other languages such as classes; C# includes some code structures that are used more frequently. For more information on these types of objects, see the official documentation here.

NOTE: A namespace is the C# equivalent to a package in Java. Both require that the user imports them and can be used to reference contained classes.

Struct

Stucts or Structures are used to encapsulate data and related methods. However, these types will not change their variables values. An example of a struct would be geo coordinates. These coordinates are loaded and are populated and will not be overridden without creating a new struct. For more information see the official documentation.

NOTE: The following restrictions are in place:

- You cannot have a parameter-less constructor
- Instance fields cannot be initialized at declaration
- The constructor must populate all instance fields
- No inheritance (except for interfaces)

Enum
An enum is a specialized type defined by constants. These types must only extend from the `System.Enum` class and cannot be inherited. These types can only contain constants from the base numeric types (byte, short, int, etc.). They are helpful when creating flags and the like.

```
enum Flag {
    None = 0,
    Error = -1,
    Known = 1,
    Success = 2,
    Unknown = 3
}
```

```
public static void Example(Flag flag) {
    Debug.Log(flag);
}
```

In these examples, I declare a new enum called Flags. I then have a method that will print out the given Flag. What is interesting about enums is that when printing out their value, it will print out the name given to them. For instance, `Example(Flag.Known)` will print out `Known`.

NOTE: This will throw a runtime error when a value not in enum is passed in.

NOTE: You can also cast numeric types as an enum (and vice versa). For example, `Example((Flag)Random.Next(-1,3))` will print out a random legal flag. You could also get the value of the enum by casting it to its respective numeric type. For instance, `(int) Flag.Success` will be two.

Static Class
In a static class, all methods and objects contained are static references. Static classes are helpful when you need to extend an object's methods, such as adding a new method to an array, or when you should not need to create a new object to access a reference, such as a game engine. For an example of a static class see the PlayerIO class documentation.

NOTE: A static class and its variables must follow all access guidelines for static objects.

NOTE: Just because a class is static does not mean that it cannot have a constructor. This constructor differs slightly from a typical constructor. Instead of typing `public Class(params object[] parameters) {//Body;}` type `static Class(params object[] parameters) {//Body;}`.

Properties
A property is a type of method in C# that behaves like a variable. These method variables are extremely useful and used frequently in the project. Properties can be any variable and of any visibility.

```
1  public static T Example {get; set;} = “Example 1”
2  public T Example2 {get;} = “Example 2”
3  public T Example3 {get; private set;}
4  private static T Example4 {get;}
```

All the above variables are properties. A keyway to spot a property is by finding the `{get;}`, `{set;}`, `{get; set;}` or any other variation on these. Each of property can specify the visibility of `{get;}` and `{set;}` as seen in line 3.

```
Private T _example;
Public T Example {
    get { return _example;}
    Set { _example = value; Event(); }
}
```

Above is an example of a modified property that will fire a method *Event* anytime that the value is changed. You can do anything that you want inside the bodies of any accessor method.

NOTE: When modifying the body of `{get;}` or `{set;}`, **must** specify the body of any other accessor method (`get`, `set`, `init`, etc.). You **must** also create another variable that will store the value being accessed by the property. If you do not specify this second variable, you will create an infinite loop.

Extension Methods

An extension method is a special user defined method that is treated as a native method for a given object. For example, arrays do not naturally have a method that will randomly shuffle the order of their values; however, we could program a method that does this and have it tied to the array base class. So instead of `Array = Shuffle(Array)` we could call `Array.Shuffle()`.

NOTE: An extension method must be a declared in a static class.

NOTE: An extension method must have the `this` keyword in its method header.

NOTE: `System.Linq` has a helpful collection of extension methods, such as an extension of HashSet that will get the item at a specified index, a function that HashSets typically do not have.

```
public static void PrintAll<T>(this T[] list) {
    foreach(T item in list) {
        Debug.Log(item.ToString());
    }
}
```

The above example will print out every item in a specified list. This method can now be called anywhere in our project by any array.

Export Project
This project is distributed to consumers in three different formats: Windows executable, MacOS executable, and an in-browser HTML document.

Build Settings
Desktop
1. In *Project Settings* navigate to "PC, Mac & Linux Standalone" under *Player*
2. Select *Resolution And Presentation*
3. Set "Fullscreen Mode" to "Exclusive Fullscreen"
4. Enable "Run In Background"

HTML
1. In *Project Settings* navigate to "WebGL" under *Player*
2. Select *Resolution And Presentation*
3. Set "Default Canvas Width" to 960
4. Set "Default Canvas Height" to 540
5. Enable "Run In Background"
6. Select "Default" preset
7. Select *Publishing Settings*
8. Enable "Decompression Fallback"

Windows Executable
1. In *Build Settings* choose "PC, Mac & Linux Standalone"
2. Change *Target Platform* to "Windows"
3. Confirm build settings
4. In *Build Settings* choose "Play and Run"
5. Publish to GitHub under releases. Upload any folders (as a .zip) that have something in them and do not say "DoNotShip"

MacOS Executable
1. In *Build Settings* choose "PC, Mac & Linux Standalone"
2. Change *Target Platform* to "macOS"
3. Proceed as a Windows Executable

HTML
Build
1. In *Build Settings* choose "WebGL"

NOTE: If you cannot see this option, you will need to install the package from Unity Hub.

NOTE: You must select "Switch Platform" to build the project as an HTML page.

NOTE: This will recompile the project. It is highly probable that there will be multiple errors when switching platforms. This will likely be caused by the Discord Integration and any preview packages in use.

2. Change *Code Optimization* to "Size"
3. Confirm build settings

4. In *Project Settings* choose "Build and Run"

Making It Look Nice

NOTE: After every step, it is recommended to save the active document and reload the generated webpage.

NOTE: You will be adding files to the *TemplateData* in the build folder for the changes to work. The following files will need to be manually added: logo.ico, and logo.png. The logo.ico can be created by converting logo.png. Logo.png can be found in resources.

1. The following changes are in the document "index.html" found in the base folder of the build.
    a. Change the name that appears in the web client's tab.
    Change the following line in "index.html"

```
<title>Unity WebGL Player | CS 360 Group Project</title>
```

    To the line below

```
<title>CS 360 Group Project</title>
```

    b. Change the favicon to the game's logo.
    Change the following line in "index.html"

```
<link rel="shortcut icon" href="TemplateData/favicon.ico">
```

    To the line below

```
<link rel="shortcut icon" href="TemplateData/logo.ico">
```

    c. Import the used font.
    Add the following lines to the header section in "index.html"

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Roboto:wght@700&display=swap"
rel="stylesheet">
```

2. The following changes are in the document "style.css" found in the *TemplateData* folder of the build.
    a. Add a background to the website.
    Change the following line in "style.css"

```
body { padding: 0; margin: 0; }
```

    To the line below

```
body { padding: 0; margin: 0; background: url('TotalDramaIsland.png') no-repeat
center; background-size: 140%; }
```

    b. Add a border around the game.
    Change the following line in "style.css"

```
#unity-container.unity-desktop { left: 50%; top: 50%; transform: translate(-50%, -
50%); }
```

    To the line below

```css
#unity-container.unity-desktop { left: 50%; top: 50%; transform: translate(-50%, -50%); border: 10px solid white; background-color: white }
```

c. Change the 'loading' icon from the Unity icon to the game's logo.
   Change the following line in "style.css"

```css
#unity-logo { width: 282px; height: 150px; background: url('unity-logo-dark.png') no-repeat center; }
```

To the line below

```css
#unity-logo { width: 282px; height: 150px; background: url('logo.png') no-repeat center; background-size: contain }
```

d. Change the loading bar size to be center.
   Change the following lines in "style.css"

```css
#unity-progress-bar-empty { width: 141px; height: 18px; margin-top: 10px; background: url('progress-bar-empty-dark.png') no-repeat center; }
#unity-progress-bar-full { width: 0%; height: 18px; margin-top: 10px; background: url('progress-bar-full-dark.png') no-repeat center; }
```

To the lines below

```css
#unity-progress-bar-empty { width: 282px; height: 18px; margin-top: 10px; background: url('progress-bar-empty-dark.png') no-repeat center; background-position: 70.5px; }
#unity-progress-bar-full { width: 0%; height: 18px; margin-top: 10px; background: url('progress-bar-full-dark.png') no-repeat center; background-position: 70.5px; }
```

e. Remove the "WebGL Icon".
   Remove the line containing the text `url('webgl-logo.png')`

f. Reformat the "Game Name Bar"
   Change the following line in "style.css"

```css
#unity-build-title { float: right; margin-right: 10px; line-height: 38px; font-family: Arial; font-size: 18px }
```

To the line below

```css
#unity-build-title { float: left; margin-right: 10px; line-height: 38px; font-family: Roboto; font-size: 18px }
```

Deployment

1. Upload the folder to GitHub.

NOTE: If a build already exists on GitHub, you will need to add the contents of the folder *Build* of the output of the build process to the *Build* folder on GitHub. You will then need to replace *index.html* with the modified output file. This should properly update the hosted build.

2. Wait for GitHub pages to register changes and rebuild or create the website.
3. Enjoy the game.