Table of Contents

Table of content

The lobby system handles many funitionalities such as:

Base system to see other players

The base system to see other players is controlled by the LobbyNetworkingController.cs class. This class can be accessed from the ServiceLocator with the interface ILobbyNetworkManager.cs and is in charge of 3 functionalities: connect to the lobby, send/receive data from the players and disconnect the player.

Lobby connection

The connection to the lobby is done from the ConnectToLobby which needs the lobby index and the connection data and then playfab is in charge of sending who is entering to the azure server with this data. It also receives a **Connection ID** in the **Cloud Script** response, which is used to detect if the current connection is the most recent one.

Sending and receiving data

The sending and receiving of data is done every X seconds in the function TickCheck(). This method is in charge of sending the connection-data of the user and process locally those of the other users. Here we do things like instantiate new players, handle disconnection for not moving for a certain period of time, format the character style, display messages to the user with their respective tags and receive/send challenges from/to other players.

Lobby disconnection

When exiting the application, in an OnApplicationQuit, a call is made to perform the disconnection on the server. This request only needs the lobby index, then the playfab call send this lobby index with the players id so it can disconnect it from the game.

Connection data

The data used in the lobby system are the following:

- **connectionId**: Id of connection to the lobby, this id is generated when the connection is made and is checked to see if the connection has not been opened in another instance. If this is the case, the old instance is closed. ** **displayName**: Nickname of the user. **playerPosition** : x,y,z position of the users in the game. ** **chatMessages**: User's chat message arrangements. ** **shinseiCompanionDna**: Dna of the company shinsei, to show it in the game. ** **characterStyle**: Players character style code. ** **characterState**: Player's state (backpack, combat, lobby, etc). ** **currentMatchId**: If in a match, it is the id of the current match, otherwise is empty. ** **challengedPlayer**: Id that identifies if the player has challenged someone, or cancelled a challenge.

Chat system

The chats system flow has three parts: sending messages, recieving messages and the bad word filter.

Recieving message

Recieving message flow starts when the lobby gets the messages from the player, in the ManageLobby() method inside the LobbyNetworkingController.cs. Basically, it order the messages by their timestamp and then send them in the chat textbox with the ChatTextBox.cs

```
    ...

    List<ChatMessagePayload> chatMesssagesSorted = item.Value.chatMessages?.OrderBy(o =>
o.timeStamp).ToList();
    if (chatMesssagesSorted != null)
        foreach (var chatMessage in chatMesssagesSorted)
            chatTextBox.SendMessage(chatMessage, item.Value.displayName, item.Key.Equals("54BB079356042E83"));

    ...
```

The SendMessage() methods sends the message verifiyng if it's already writter, has bad words or if it's an admin.

```
    public void SendMessage(ChatMessagePayload chatMessage, string displayName, bool isAdmin = false)
    {
        if (alreadyWriteMessages.Contains(chatMessage.id))
            return;

        if (PlayerPrefs.GetInt("BadWordFilterOption", 1) == 1)
            chatMessage.message = CheckForBadWords(chatMessage.message);

        alreadyWriteMessages.Add(chatMessage.id);
        if (isAdmin)
            AddText($"<color=red>[ADMIN] </color>: {chatMessage.message}");
        else
            AddText($"<color=#EFEBCE>[Server] ({displayName}): {chatMessage.message}</color>");
    }
```

Sending a message

The SendLocalMessage method handles the sending of a message in the chat. Checks if has bad words, check if it's directed to someone and then add it to the list of messages of the player in order to send it on the next update of the server.

```
    public void SendLocalMessage()
    {
        if (String.IsNullOrWhiteSpace(chatInput.text))
            return;

        if (PlayerPrefs.GetInt("BadWordFilterOption", 1) == 1)
            chatInput.text = CheckForBadWords(chatInput.text);

        // Whisper verification
        if (!chatInput.text.Contains("/r "))
            AddText($"<color=#DFDBC0>[Server] ({PlayerDataManager.Singleton.localPlayerData.playerName}):
{chatInput.text}</color>");
        else
        {
            string[] trimText = chatInput.text.Split(' ');
            AddText($"<color=#E146CD>[To] ({trimText[1]}): {(chatInput.text).Replace("/r " + trimText[1],"")}
</color>");
            PlayerPrefs.SetString("LastWhisper", trimText[1]);
        }

        //Add to list of player messages in order to send it on the next update of the server.
        PlayerDataManager.Singleton.localPlayerData.currentChatMessages.Add(new ChatMessagePayload() { message
= chatInput.text, timeStamp = DateTime.UtcNow.ToString("o", CultureInfo.InvariantCulture) });

        chatInput.text = "";
        chatInput.Select();
        chatInput.ActivateInputField();
    }
```

The server only conserves the messages for X time and then deletes them from the player in order to not store huge amounts of data.

Bad word filter

The bad world filter takes a dictionary previosly initialize from a json file and takes the message and check if the message contains a word that is a bad word in any of the languages and return the messages with the bad words filtered.

```
public ChatBadWordsDB badWordsDB;

public struct ChatBadWordsDB
{
    public List<List<string>> listOfBadWords;
}

public string CheckForBadWords(string message)
{
    string[] messageWords = message.Split(' ');

    for (int i = 0; i < messageWords.Length; i++)
    {
        string word = messageWords[i];
        foreach (List<string> languageBadWords in badWordsDB.listOfBadWords)
        {
            if (languageBadWords.Contains(word.ToLower()) )
            {
                messageWords[i] = "****";
                break;
            }
        }
    }

    string resultAfterFilter = "";
    foreach (var word in messageWords)
        resultAfterFilter += $"{word} ";

    return resultAfterFilter;
}
```

Challenge other player system

Challenges takes the form of a variable called challengedPlayer. This variable is in the user lobby server data and can have 3 available values:

- "PLAYFABID_OTHERPLAYFABID_RANDOMNUMBER": This value refers to a challenge made from one player to other. This value will also be used as the matchId of the match.
- "CANCELED" : This value refers to a player that canceled a challenge that someone else did to him.
- "": Empty message, player isn't sending challenge to anybody.

In the ChallengePlayer.cs class most of the logic of this functionality is executed. For example, the CheckChallenge() handles both recieving and initiating a combat. Basically, it checks if the player can recieve or challenge someone and if so then it created the matchId and then creates the match sending to the server the petition to create it. Then it waits for the other player to confirm the match. Then once it's confirmed they both enter to the shinsei selection screen.

The game checks the server to see the challenges and responses of other players in the LobbyNetworkingController.cs, more precisely in the PlayerChallengeVerification(). This method is executed in local for every player in order to check if any player has a challenge for the local player or to know if the player has a challenge to other player. Let's see a pseudocode of the method:

```
//WARNING: PSEUDOCODE not real code

        public void PlayerChallengeVerification(KeyValuePair<string, LobbyPlayerBasePayload> item)
        {
            // We check if the challenge variable on server has any value.
            if (!emptyChallenge)
            {
                // If the challenge on the player hasn't change since the last time we check the server data
                // And if the other challeged player has a "CANCEl" in his challenge variable.
                if (isMatchSameAsLastOne && challengedPlayerCanceledMatch)
                {
                    // We proceed to cancel the challenge and erase both players challenge variable in this
player local data.
                    if (isLocalPlayer)
                    {
                        // Erase challenge data on local and send it to the server
                        PlayerDataManager.Singleton.localPlayerData.challengedPlayer = "";
                        TickCheckActivates();
                    }

currentPlayersAvatar[item.Key].challengePlayerController.MatchCanceledByChallenged(isLocalPlayer);

currentPlayersAvatar[playfabIdChallengedPlayer].challengePlayerController.MatchCanceledByChallenged(isLocalPla
yer);
                }
                else if (isCurrentPlayer && canceledAlreadyProcessed)
                {
                    // If it has a "CANCEL" value on local player challenge variable and the canceled was
already processed.
                    // We erase the "CANCEL" to a "" and send it to the server.
                    PlayerDataManager.Singleton.localPlayerData.challengedPlayer = "";
                    TickCheckActivates();
                }
                else if (currentPlayerHasChallenge)
                {
                    // If there is a challenge for me call the challenge player controller
                    // to open a popup with the challenge
                    currentPlayer.challengePlayerController.RecieveChallenge(currentPlayersAvatar[item.Key],
randomMatchNumber);
                }
            }
        }
```

See other player match system

We can see other player battles by clicking on the swords icon that appears above the lobby users if they are on a match. This button will activate the ViewMatch() method in the PlayerUI.cs. This method will show a popup and then if it's confirmed, then we proceed to create a match with the currentMatchId of that player. The match will take a special route on the creation ensuring that the data necesary for the battle to be created is filled, or at least a template while the data arrives from the server. Let's take a look at an example of this in the SendBattle() from GameSceneManage.cs.

```
GameSceneManager.cs

        public void SendBattle(..., bool isViewing)
        {
            ...

            BattleGameMode battleInstance = combatInstance.GetComponent<BattleGameMode>();

            if (getMatchResult != null)
            {
                PlayerDataManager.Singleton.localPlayerData.currentMatchId = getMatchResult.MatchId;
                battleInstance.OnStartMatch(getMatchResult);
            }
            else
            {

// This is the case when it viewing, it initializes the data
                PlayerDataManager.Singleton.localPlayerData.currentMatchId = matchId;
                battleInstance.isViewingMatch = isViewing;
                BattleViewerController battleViewer = combatInstance.GetComponent<BattleViewerController>();
                battleViewer.Initialize(matchId, isViewing);
            }
            ...
        }
```

```
BattleViewerController.cs

        public void Initialize(string matchId, bool isViewing)
        {
            //This is just template  data that will later be filled by the server data of the match.
            battleGameMode.localCombat = new Combat()
            {
                CurrentTurn = 0
            };
            battleGameMode.playerInfo.userIndex = 0;
            battleGameMode.enemyInfo.userIndex = 1;

            battleGameMode.playerInfo.isLocalPlayer = true;
            battleGameMode.enemyInfo.isLocalPlayer = false;

            battleGameMode.localCombat.MatchData.MatchId = matchId;
            battleGameMode.localCombat.MatchData.MatchPlayers = new
System.Collections.Generic.List<CombatPlayer>() { null, null };

            if (isViewing)
                battleGameMode.WaitForOpponentToSelectViewMatch();
            else
                battleGameMode.StartCoroutine(battleGameMode.WaitForOpponentToSelect());
        }
```

Then we continue the normal flow of the game by sending the flow to the WaitForOpponentToSelect() . When this data arrive we override the template with this data and then continue the normal flow of the game.

Back pack

Shinsei Vault System

The data of the shinsei vault it's recieved or created and uploaded by the PlayerDataManager.cs. In the InitPlayerData() we check if we already have data in playfab to download, if we do we download it and set it. If we don't we use the method FillPlayerData() to create new data for the player, including the shinsei vault.

The class ShinseiVaultController.cs it is the manager class of the majority the shinsei vault logic in game. The flow starts when the user request to see the shinsie vault. The vault is drawn an set with the data of the user that is located in the PlayerDataManager.cs. Upon this, we create a callback to the method OnVaultShinseiSelected() where we basically select it and then store it in a temp shinsei, if we select another shinsei then we swapped them and update the player data in the server.

Shinsei Card exchange system

The data of the shinsei's(and also it's cards) it's recieved or created and uploaded by the PlayerDataManager.cs. In the InitPlayerData() we check if we already have data in playfab to download, if we do we download it and set it. If we don't we use the method FillPlayerData() to create new data for the player, including the shinsei.

CardManagementController.cs it's the class that manges all the shinsei card visualization and exchange. The flow starts in the method InitCardManager() where it initalizes all the cards in the inventory and all the cards on the shinseis. Also sets some information like the shinsei type in the background and the images and stats of the shinseis. Then in this same class we use a vairaty of methods like AddCardInDeck, RemoveCardInDeck, AddSelectedCard, etc to basically interchange the cards from the vault .

Ranking System

The class that manages the rank system is NewLeaderBoard.cs //TODO

Character Customization system

In sacred tails you can customize your character in a few of forms The first form is Genre, you can choose between male and female

Male:

Female:

Also you can change the color of clothes and model, in this first implementation only exist one sweat for each genre but the system works.

This system works using a single material that replicates over all parts of character avoiding big number of batches when render

multiple players at time

The input values for this material is injected by script, the textures are injected using a script called MaterialReskin.cs

MaterialReskin.cs

The most important part of MaterialReskin.cs is the Init function, first create a texture and store in a list, first Difuse then normal, metallic and finally Ambien Oclusion after that create a new material and asign the new created textures, initialize the CharacterRecolor and apply that material to all parts of character like a torso, head, legs and arms

```
  ...
  public void Init()
    {
        mergeTextures.Add(new Texture2D(skinList[0].Difuse.width, skinList[0].Difuse.height,
skinList[0].Difuse.format, true));
        mergeTextures[0].SetPixels(skinList[0].Difuse.GetPixels());
        mergeTextures.Add(new Texture2D(skinList[0].Normal.width, skinList[0].Normal.height,
skinList[0].Normal.format, true));
        mergeTextures[1].SetPixels(skinList[0].Normal.GetPixels());
        mergeTextures.Add(new Texture2D(skinList[0].Metallic.width, skinList[0].Metallic.height,
skinList[0].Metallic.format, true));
        mergeTextures[2].SetPixels(skinList[0].Metallic.GetPixels());
        mergeTextures.Add(new Texture2D(skinList[0].AmbientOclusion.width, skinList[0].AmbientOclusion.height,
skinList[0].AmbientOclusion.format, true));
        mergeTextures[3].SetPixels(skinList[0].AmbientOclusion.GetPixels());

        horizontalTextureSize = mergeTextures[0].width / horizontalSplitParts;
        verticalTextureSize = mergeTextures[0].height / verticalSplitParts;

        //Create new material
        Material material = meshRenderers[0].material;
        material.SetTexture(mainTexture, mergeTextures[0]);
        material.SetTexture(normalMap, mergeTextures[1]);
        material.SetTexture(metallicGloss, mergeTextures[2]);
        material.SetTexture(ambientOclussion, mergeTextures[3]);
        material.EnableKeyword("_NORMALMAP");
        targetMaterial = material;
        characterRecolor.Init(targetMaterial);

        //Apply new material to all parts :D
        for (int i = 0; i < meshRenderers.Count; i++)
            meshRenderers[i].material = targetMaterial;
        //Verify if split numbers are even
        if (horizontalSplitParts % 2 != 0)
            horizontalSplitParts += 1;
        if (verticalSplitParts % 2 != 0)
            verticalSplitParts += 1;
        //Add the position of textures in a list
        for (int i = 0; i < horizontalSplitParts; i++)
            for (int a = 0; a < verticalSplitParts; a++)
                texturePosition.Add(new Vector2Int(i * horizontalTextureSize, a * verticalTextureSize));
        isInit = true;
    }
  ...
```

CharacterRecolor.cs

Character recolor is in charge of set the _NewColor Parameters of material in a human conversion, first create a dictionary of convensions from PartOfCharacter to string and will be called using ChangeMaterialColors function, receives a part of character

and desired color

```
public class CharacterRecolor : MonoBehaviour
    {
        private Dictionary<PartsOfCharacter, string> materialColor = new Dictionary<PartsOfCharacter, string>
() {
        {PartsOfCharacter.SKIN,"_NewColor1"},
        {PartsOfCharacter.HAIR,"_NewColor2"},
        {PartsOfCharacter.PRIMARY_COLOR,"_NewColor3"},
        {PartsOfCharacter.SECONDARY_COLOR,"_NewColor4"},
        {PartsOfCharacter.DETAILS,"_NewColor6"},
        {PartsOfCharacter.HANDS,"_NewColor5"},
        {PartsOfCharacter.LEGS,"_NewColor5"}
    };
        private List<Material> newMaterials = new List<Material>();

        public void Init(Material target)
        {
            if (!newMaterials.Contains(target))
                newMaterials.Add(target);
            if (lastColor != null)
                ChangeMaterialColors(lastPart, lastColor);
        }
        private PartsOfCharacter lastPart;
        private Color lastColor;
        public void ChangeMaterialColors(PartsOfCharacter part, Color color)
        {
            lastPart = part;
            lastColor = color;
            foreach (var material in newMaterials)
                material.SetColor(materialColor[part], color);
        }
    }
```

Each part of character have a UI panel designed for change material values by user and replicate that to all players

Skin manipulate the _NewColor1 parameter of material using predesigned values but you can use any value that you need

Color manipulate the other parameters that be asigned to the clothes and details

And finally the charge of change 3D models is

BodyStyle.cs

```
  public class BodyStyle : MonoBehaviour
{
    public List<BodyPartDressable> bodyParts = new List<BodyPartDressable>();

    [System.Serializable]
    public class BodyPartDressable
    {
        [SerializeField] string name;
        [SerializeField] PartsOfCharacter part;
        [SerializeField] List<GameObject> possibleParts = new List<GameObject>();
        [SerializeField] List<GameObject> possiblePartsMale = new List<GameObject>();

        public void SelectObject(int index, bool isLocal = false)
        {
            for (int i = 0; i < possibleParts.Count; i++)
                possibleParts[i].gameObject.SetActive(false);
            possibleParts[index].SetActive(true);
            for (int i = 0; i < possiblePartsMale.Count; i++)
                possiblePartsMale[i].gameObject.SetActive(false);
            possiblePartsMale[index].SetActive(true);
            if (isLocal)
                CharacterStyleController.UpdatePartOfCharacter(part,index);
        }
    }
}
```

This script store the all posible parts of character and allow to change them using the function called SelectObject

The change hair is an example of Body Style

The combination of all systems allow the player change style

Dialogue system

This dialog system has two main actors, Conversations and Dialogables

Dialogable

A dialogable is a component that allow you show conversations in the player screen the most important parts are

Start Conversation This function first hide the player personal UI like a name and icon, later block player movement, change the camera view to this dialoguer camera and start the conversation routine depending of firstDialog variable

```
    ...

    public void StartConversation()
        {
            ServiceLocator.Instance.GetService<ILobbyNetworkManager>().ShowPlayerPersonalUI(false);
            if (dialogUI.IsPlayerDialogate)
                return;
            if (UIGroups.instance != null && !UIGroups.instance.lastActivate.Equals("planner"))
                return;

            isOnDialog = true;
            thirdPersonController.CanBeBlocked = true;
            thirdPersonController.IsMovementBloqued = true;
            thirdPersonController.CanBeBlocked = false;
            dialoguerCamera.SetActive(true);
            canvas.SetActive(false);
            dialogUI.gameObject.SetActive(true);
            if (UIGroups.instance != null)
                UIGroups.instance.ShowOnlyThisGroup("dialogue");
            ServiceLocator.Instance.GetService<ILobbyNetworkManager>().CurrentPlayer.GetComponent<PlayerUI>
().HideNameTag(false);
            if (firstDialog && firstConversation.dialogGraph != null)
            {
                PlayerPrefs.SetInt(playerPrefVar, 0);
                firstDialog = false;
                StartCoroutine(firstConversation.ConversationRoutine(dialogUI, EndConversation, this));
            }
            else
            {
                StartCoroutine(conversation.ConversationRoutine(dialogUI, EndConversation, this));
            }

        }
    ...
```

Conversation

This class keep the logic of conversation, store the dialogs and answers and show them to the player by UI elements The most important function is the coroutine ConversationRoutine

First notify the player is dailogating then clear dialog UI and their components after that check if you use a special code <!Index!> to launch a callback and start to write dialog in the screeen

If find a answer dialog, draw responses in the screen and start wait to the player confirmation using the SendResponse function

At the end of all dialogs and responses, trigger the EndConversationCallback and leave the execution

```
    public IEnumerator ConversationRoutine(DialogUI dialogUI, Action EndConversationCallback = null,
Dialogable targetDialogable = null)
        {
            dialogUI.IsPlayerDialogate = true;
            Init(dialogUI, EndConversationCallback);
            while (true)
            {
                string dialogText = currentNode.dialogText;
                //Get actions inside texts
                if (currentNode.dialogText.Contains("<!"))
                {
                    string[] dialogParts = dialogText.Split(new string[] { "<!" },
StringSplitOptions.RemoveEmptyEntries);
                    string splitedDialog = dialogParts[1];
                    splitedDialog = splitedDialog.Split(new string[] { "!>" },
StringSplitOptions.RemoveEmptyEntries)[0];
                    if (targetDialogable != null)
                        targetDialogable.CallbackEvents[int.Parse(splitedDialog)].Invoke();
                    dialogText = dialogText.Replace($"<!{splitedDialog}!>", "");
                    //TODO fix this later for more cases in this case if you use callback by dialog the dialog
close inmediately
                    EndConversationCallback?.Invoke();
                    EndConversationCallback = null;
                    dialogUI.gameObject.SetActive(false);
                    dialogUI.IsPlayerDialogate = false;
                    break;
                }
                dialogUI.WriteText(dialogText, NotifyPlayerReadText, currentNode.Answers.Count > 0);
                while (true)
                {
                    if (isPlayerReadText)
                    {
                        isPlayerReadText = false;
                        break;
                    }
                    yield return null;
                }
                NodePort port;
                if (currentNode.Answers.Count < 1)                          //Take default node
                    port = currentNode.GetPort("output");
                else                                                        //Take answer node
                {
                    isWaitingResponse = true;
                    while (isWaitingResponse)
                        yield return null;
                    port = currentNode.GetPort("Answers " + responseIndex);
                }
                if (port != null && port.IsConnected)
                    UpdateDialog(port.Connection.node as DialogNode, dialogUI, EndConversationCallback);
                else
                    break;

                yield return null;
            }
            EndConversationCallback?.Invoke();
            dialogUI.gameObject.SetActive(false);
            dialogUI.IsPlayerDialogate = false;
        }
```

All data for conversations is stored in a class called

DialogGraph

Dialog graph is a Custom ScriptableObject made with XNode, you can write conversation visually easy

Each Dialog Graph contains multiple DialogNode

Input and Output node controls de dialog flow, if output is empty and the node has'nt responses the conversation end.

A: Input

B: Output

C: Dialog Text (Here you can put the text of conversation)

D: Responses (Each response has her own output node for conversation flow)

Shinsei interaction system

The pet interaction system contains is a simple script called PetInteraction that allow you to play animations in companion Shinsei.

The user interact with the script using a pie menu and function PlayAnimation(index) when index is index of animation

The script stores multiple animations and setup the enviroment for the correct visualization, stop the player, Hide innecesary elements and show petting UI

Day and night cycle

The day and night cycle is managed by the class The script simulates the passing of time in the game with calculations to make the day last 3 hours. So every 3 hours in the real world 1 day passes in the game.

The script initializes by calling an api(worldtimeapi.org) to get the current time. Once the API call returns, the script parses the response to extract the current hour and minute, and uses that to calculate the current time in the game (from 0 to 24). This logic is only execute once at the beginning of the game to ensure the players have the same hour in all their local clients.

For the rest of the game, the time is updated every 60 seconds. Every 60 seconds we add .125 hours to the current time, in order to preserve the ration of 3hous/day.

Bar

The bar is the lastest update on the game. It functions as a reunion site to drink/eat, watch shinsei fights and check tournaments. Let's see those functionalities in more details

Drinking and eating system

This system is basically a specific NPC interaction for that reason the charge script is called Cooker.cs, this script store references to the NPC animator Cooker and other props like a ber or eat plate, when character interact start to play a coroutine that play secuencially animations, show and hide objects for make the illusión of interaction, using a callback of dialogue turn on a script that show the animation pending of response of character in this case give a beer of serving a plate of food.

For decoration the bar also have a Sit system when you can see other players sitdown or you can sitdown in the chairs of bar, this system works using nearness, the first character that touch the sit can use the other player will be ignored, for reason of delay of server you can see different things when you see from one client or other, the chair always align their forward direction to the character.

Tournament system

The tournament system is a complex system that has to handle multiple cases and states of the users in order to garantee a correct flow of the tournament. We have to check the state of the whole tournament and the state of each bracket. The state of the players if they are ready or not to fight. The disconnection of players. A lot of variables to take in consideration. Here is a general flow of the system:

We start by requesting the tournament list in the bar. We request and display them with the SearchAndShow.cs filtering the ones that hasn't finished yet and then let the user decide one of them. Once the user decides we execute the JoinTournament() in the TournamentSlot.cs. If the joining was a success, we proceed to hide all the things that should be hidden in a tournament such as searching for matches or changing the shinseis position in our vault.

Then we start the timer to wait for the tournament to initiate in the ShowTimerInitTournament() of the TournamentReadyController.cs. This method checks if the tournament has started already, if it doesn't then it proceeds to show the countdown to initiate the tournament. If it has, proceeds to show the ready button for the player to start his current match in the tournament with the StartReadyButton(). This method sets the timer to get ready for the match before a disqualification. If the players clicks on the ready button then the MarkAsReady() method start to execute every X seconds. This method will send a petiton to the azure server to check if the other player also accepted the match. In this method we also check if the player won by default and, if so, then we proceed to show him the time left for the next round to begin with the ShowPendingTimeAfterMatch() . Otherwise, if the two players get ready for the match, the match starts.

Once they finished the match we proceed to check the tournament state every 3 seconds with the method CheckTournamentState() of CheckTournamentState.cs. In this method we send a petion to the server and in it's response we check a variaty of states, ranging from winner of tournament, second place, loser of match and finally winner of match. It's this last case, were we show the player the ShowPendingTimeAfterMatch() method from before, so that he can see the time to end the match. But we keep asking the server if the round has ended before time(if all players have already played their matches). If so then we show again the [StartReadyButton()] and the cycle begins again until there is a winner in the tournament.

Battle system

The system is basically composed of 3 pillar controllers:

- The **GameSceneManager.cs**: Which receives the request to create a match and configures the initial data needed to start the battle.
- The **BattleGameMode.cs**: Which handles general game information such as player information and also handles game status such as the start and end of the game.
- The **TurnsController.cs**: The main controller of the game. It handles the logic of the individual turn and turn flow.

Let's look at each of them in more detail.

GameSceneManager.cs

This class sends a petiton to create a new battle with the SendBattle method. It is necessary to pass it the necessary data such as the matchId of the game, the current shinseis of the players and their stats. After that, it creates an instance of the battle, changes the player status in the database to combat and turns off the unnecesary open windows at the moment. After that it passes the flow to the BattleGameMode.cs with all the necessary data to start the battle.

On the other hand, when the battle is over, this is the controller that returns us to the lobby with the EndBattle method, showing again the lobby UI and deleting the battle instance.

BattleGameMode.cs

In the battle game mode we have the **pre-battle** flow and displaying of the **end of battle**. We also handle here the **general data** of the users in the battle.

We start in the OnStartMatchRoutine() method once the data is initialised, with the initialisation of the pre-battle shinseis selector. These are handled by controllers such as the ArenaShinseiSelectionController.cs and the ArenaShinseiSelectionUserPanelController.cs which handle the entire flow of viewing and choosing the shinseis for the battle.

Once chosen, it **notifies BattleGameMode.cs** with the shinseis that were chosen or with the fact that time is up and no shinseis were chosen. If they were indeed chosen, it proceeds to send the **indices** of the chosen shinseis to the server via a **CloudScript**.

Then the flow proceeds to wait for the other player to choose with the method WaitForOpponentToSelect() and once this is done it **initializes the data of the chosen shinseis, the life and energy bars, the shinsei change system and spawn the shinsei**.

After this it continues with the start-up cameras, which show the shinseis from the front and finally once those cameras finish showing the shinseis, the UI is initialised. From here, the TurnsController.cs will control the rest of the flow of the battle.

Once the match is over the BattleGameMode.cs will return to the BattleGameMode.cs method to display the result screen.

TurnsController.cs

This class handles the flow of the battle once it has started. It initialises along with the BattleGameMode.cs and initialises the Battle Actions dependencies. Then we continue the flow with each turn start in the InitTurnFlow() method. Every time a turn starts we check if the player is sleeping or not in order to run the animation and send a mandatory turn skip to simulate the sleeping state. This is a temporary solution that should be changed to a BattleAction, but for the moment it is like this.

```
Send Turn
```

Then the flow continues when the player decides to send an action with the SendTurnRequest method, either one of his cards, a shinsei change, a skip turn or a surrender. All these actions are sent with an index as follows:

- 0,1,2,3 => Shinsei attacks in that respective order.
- 4,5,6 => Change to the respective shinsei (i.e.: 4 -> first shinsei)
- 7 => Skip Turn
- 8 => Surrender

The index is sent to the server via a **CloudScript**. If the **request fails** it is **reentered by sending the turn** up to a maximum of 7 times. After that **it is considered a disconnection of the player** who could not send the turn. Also the case when for whatever reason the match is over according to the server is taked into account too. In this case BattleGameMode.cs is called to end the match on the client.

In case everything works correctly, we continue to wait for the other player's turn.

Receiving turns from the two players

The method WaitOponentTurn() helps us to wait for the opponent's turn by sending a **CloudScript** every X seconds, where X is the time defined by parameter (i.e: 3seconds). This request performs all the necessary calculations in the **server** for it to process and I have a **snapshot of the game** with those turns processed. Then once this process is finished, it returns the turns of the two players so that the **client processes the turns itself**. This method handles and delegates from end-of-game to the processing of altered states, terrain and BattleActions.

As for **Terrain**, these are stored in the BattleGameMode.cs and can only exist **one at a time**. Each terrain has its own controller which inherits from the BattleTerrainBehavioursBase.

A similar thing happens with **altered states**, only instead of being stored in BattleGameMode.cs per se, they are stored in the specific shinseis that have the altered states. But these can be multiple per shinsei. They are all controlled by classes that inherit from BattleAlteredStateBase.

The **processing of the current player's** altered state and battleActions** is encapsulated in an event or Action that is passed to the InitTurnFlow() method. This method, with the help of the server response, determines who starts the turn first and whether or not the other player can execute the turn due to being dead or asleep. This method handles the timing based on an action time that is defined at the start of the execution of the altered states and Battle Actions. After defining the time for the InitTurnFlow() to wait, these are executed with their vfx and cameras.

Battle Actions

The battle actions are executed with the help of the method CalculateIncomingActions(). This method applies the energy and executes the Battle Actions in the order in which the player has received/sent them. These battle actions vary from an attack, a heal, a buffdebuff, setting up altered states or terrain, etc. Each of these actions has a controller that inherits from BattleActionsBase. This base runs the vfx with its timings so that its inheritance only cares about the logic of the action. As we said before the vfx time is sent to the turns controller so the InitTurnFlow() method can control the waiting time for the skills to finish executing their vfx. When the method has finished executing all the actions, it proceeds to check if the shinseis have died and if they have all died to end the game locally.

End of the game

The end of the game is executed if the server indicates it, either by the detection of a surrender or because the match ended on the server. It is also executed if the client ends the game with its current values, this is a security measure, however all prizes, scores, etc. will be reflected in how the game ends on the server.

Server

Sacred Tails uses Microsoft Azure and PlayFab systems to function, PlayFab to save account data, battle data, and even manage the game lobby, and Azure to allow players to make requests and interact with others, as it is a serverless system that bills based on usage.

Battle Server

The combat system works in several stages, in the first stage it checks if both players have successfully connected, then both clients verify that they are ready, a countdown starts to choose the Shinsei that will be taken to the combat out of the 6 you can take with you all the time, once chosen the main loop of the combat begins which consists of each player sending their turn, using SendTurn.

Send Turn

Here from the client, the button pressed is indicated, existing from 0 to 3 for the current Shinsei cards, from 4 to 6 to change Shinsei, 7 to skip turn, 8 to surrender. This is done to prevent malicious users from manipulating the RAM and using attacks that are not assigned or breaking the game in any other way.

The client is responsible for calling the Azure Function SendTurn.js, this first makes necessary checks for the combat such as checking if there is already a winner, then it checks if the other player is writing their turn to avoid writing at the same time. After the checks are passed, it checks the number that the player sent, if it is greater than 0 and less than 4 it is an attack, then it checks the combat data and looks for the attack that corresponds to the received position, and adds the turn with that action.

```
let indexOfCard = req.body.FunctionArgument.Keys.indexCard;

    let turnToSend
    if (indexOfCard >= 0) {
        if (indexOfCard < 4) {
            await PlayfabHelpers.GetPlayerMatchData(matchId, playFabId).then(async (playerData) => {
                let shinseiCardIndex =
playerData.ShinseiParty[playerData.currentShinsei].ShinseiActionsIndex[indexOfCard];
                turnToSend = { ...cardDatabase[shinseiCardIndex] };
                turnToSend = ParseCardData(turnToSend);

                //Check if it is a random action type
                turnToSend.BattleActions.forEach(element => {
                    if (element.actionType == 8 || element.actionType == 11) {
                        let previousIndex = shinseiCardIndex;
                        if (element.actionType == 8) {
                            shinseiCardIndex = generateRandomBetween(4, cardDatabase.length,
shinseiCardIndex);
                            turnToSend = { ...cardDatabase[shinseiCardIndex] };
                            turnToSend = ParseCardData(turnToSend);
                        }
                        turnToSend.isComingFromCopyIndex = previousIndex - 3;
                    }
                });

                sendTurnHelper.SendTurnWithCard(turnToSend, matchId, currentTurnKey, newTurnIndex, playFabId,
shinseiCardIndex, context);
            });
```

When the players have already sent their corresponding turns, they begin listening to the server to check if the other has already sent their data and the server was able to process the turn, this is done in the Azure Function GetMatchState.

GetMatchState.js And CalculateTurn.js

Get match state is responsible for checking if the data from both players is already uploaded and, in case it is, it calculates them using Calculate Turn.

Calculate turn is the heart of the combat, it is responsible for applying all the modifications to the data to save them in PlayFab. It first verifies that only one of the two players is manipulating the data, then it verifies which of the two Shinsei that are currently in combat has more speed to apply their actions first.

Having decided the order of execution of actions, it verifies a special case that is CopyCat, an attack that allows you to use the same attack as the opponent, then it checks if either of them decided to surrender, and finally, it verifies if the time limit for the combat has not been exceeded. After passing all these filters, the turn is processed using the ExecuteTurn function.

```
function ExecuteTurn(context, matchId, actionsOrdered, turnsOrdered, players, playfabIds, ppCosts, turnsData,
goFirst) {
    if (turnsData.currentTerrain != null) {
        if (turnsData.currentTerrain.turnsLeft > 0)
            terrainsController.ExecuteTerrain(matchId, turnsData.currentTerrain, players[0], players[1]);
        if (turnsData.currentTerrain.turnsLeft <= 0)
            terrainsController.EndTerrain(matchId, turnsData.currentTerrain.type, players[0], players[1]);
    }

    let playerIndex = 0;
    let otherIndex = 1;

    let previousCurrentShinsei = players[otherIndex].currentShinsei;
    CalculateAlteredStates(players, actionsOrdered, playerIndex, otherIndex);
    let gameEnded = actionsController.CalculateActions(context, matchId, actionsOrdered, turnsOrdered,
players, playfabIds, playerIndex, otherIndex, ppCosts, turnsData);
    if (gameEnded)
        return true;

    if (previousCurrentShinsei == players[otherIndex].currentShinsei) {
        CalculateAlteredStates(players, actionsOrdered, otherIndex, playerIndex);
        gameEnded = actionsController.CalculateActions(context, matchId, actionsOrdered, turnsOrdered,
players, playfabIds, otherIndex, playerIndex, ppCosts, turnsData);
        if (gameEnded)
            return true;
    }
    else
        actionsOrdered[otherIndex] = [];

    RegenerateEnergy(actionsOrdered, players, 0);
    RegenerateEnergy(actionsOrdered, players, 1);
}
```

In execute turn, first, the altered states are calculated. These can be Burned, Rooted, Bleeding or Reflecting and their effects are applied. Finally, the effects of the card chosen by the player are applied and the energy of the Shinsei is regenerated at the end of the process. If it turns out that a Shinsei died in the process, it is changed to the next one, or simply if there are no more available, the next time GetMatchState is called, the end of the game will be calculated.

Battle Actions

Each action within the game is represented by a class that is responsible for knowing how to process itself. In the current game, the following actions exist:

- Block
- BuffDebuff
- ChangeShinsei
- CopyCat

- Damage
- Heal
- ReflectDamage
- SkipTurn
- StatSwap
- EndGame

Block

Block allows the Shinsei to evade the damage from the next attack of its opponent. Some cards bring this action as a secondary effect.

```
function BlockActionType(moveData, userData, otherData) {
    //check who gets the movement block
    //write the ban movements into the players data
    let targetPlayerData = moveData.isSelfInflicted ? userData : otherData;
    if (targetPlayerData.forbidenActions == null)
        targetPlayerData.forbidenActions = [];
    if (!(moveData.amount in targetPlayerData.forbidenActions)) {
        targetPlayerData.forbidenActions[moveData.amount]=moveData.turnsDuration
    }
}
```

Buff and Debuff

Buffs increase the statistics of a Shinsei, whether its attack, defense, vigor, stamina or damage, debuffs are the opposite by decreasing the stats.

```
function ExecuteBuffDebuff(buffDebuffData, userData, otherData) {
    if (buffDebuffData.applyEachTurn || (!buffDebuffData.applyEachTurn && buffDebuffData.turnsPassed == 0)) {
        let targetPlayerData = buffDebuffData.isSelfInflicted ? userData : otherData;
        let attackEvaded = buffDebuffData.evadeRoll <
battleStatisticsCalculator.ApplyEvationCritics(targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].
evadeChance);
        if (!attackEvaded) {
            let buffAmount = buffDebuffData.amount;

            if (!buffDebuffData.isBuff)
                buffAmount *= -1;

            let stat = GetStat(buffDebuffData.statToModify);
            let statValue =
targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].ShinseiOriginalStats[stat];

            if (buffDebuffData.isPercertange) {
                let percentageBase = buffAmount / 100;
                let percentage = 1 + percentageBase;
                if (buffAmount > 0) {
                    targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].ShinseiOriginalStats[stat]
= statValue * percentage;
                    buffDebuffData.numberOfTimesBuffApplied++;
                }
                else if (buffAmount < 0) {
                    targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].ShinseiOriginalStats[stat]
= statValue * percentage;
                    buffDebuffData.numberOfTimesBuffApplied++;
                }
            }
            else {
                let finalAmount =
targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].ShinseiOriginalStats[stat] + buffAmount;
                if (finalAmount < 300 && finalAmount > 0) {
                    targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].ShinseiOriginalStats[stat]
= finalAmount;
                    buffDebuffData.numberOfTimesBuffApplied++;
                }
            }
            SetNormalizedStats(targetPlayerData);
        }
        else {
            if (buffDebuffData.evadedTurns == null)
                buffDebuffData.evadedTurns = 1;
            else
                buffDebuffData.evadedTurns++;
        }
    }
}
```

Change Shinsei

This defines the action of changing the current Shinsei.

```
function ExecuteChangeShinsei(changeShinseiData, userData, otherData, actionsOrdered,userIndexes) {
    let targetPlayerData = changeShinseiData.isSelfInflicted ? userData : otherData;
    let targetIndex = changeShinseiData.isSelfInflicted ? userIndexes[0] : userIndexes[1];

    targetPlayerData.currentShinsei = changeShinseiData.amount >= targetPlayerData.ShinseiParty.length ?
targetPlayerData.currentShinsei : changeShinseiData.amount
    actionsOrdered[targetIndex] = [];
}
```

Copy Cat

Copy Cat is a special case where the opponent's ability is used.

```
function ExecuteCopyCat(turnsOrdered, actionsOrdered, userIndex, otherIndex) {
    //Check if its copycat and add actions to the players action list

    for (var index of reverseKeys(actionsOrdered[userIndex])) {
        let action = actionsOrdered[userIndex][index];
        if (action.actionType == 11) {
            let previousIndex = turnsOrdered[userIndex].indexCard;

            actionsOrdered[userIndex].splice(index, 1);
            turnsOrdered[userIndex] = { ...turnsOrdered[otherIndex] };
            turnsOrdered[userIndex].BattleActions.forEach(action => {
                actionsOrdered[userIndex].push({ ...action });
            });

            turnsOrdered[userIndex].isComingFromCopyIndex = previousIndex - 3;
        }
    }
}
```

Damage

This is the simplest type of action because it only processes the damage with the current Shinsei's stats and applies it.

```
function ExecuteDamage(damageData, userData, otherData) {
    let targetPlayerData = damageData.isSelfInflicted ? userData : otherData;
    if (targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].reflectDamageLeft != null &&
targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].reflectDamageLeft > 1) {
        targetPlayerData = damageData.isSelfInflicted ? otherData : userData;
        targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].reflectDamageLeft--;
    }
    let criticsMultiplier = 1;

    //Apply critic
    let criticMultiplier = 1;

    if (damageData.criticsPercentChange > 0 && damageData.criticsRoll <
battleStatisticsCalculator.ApplyEvationCritics(damageData.criticsPercentChange))
        criticMultiplier = 1.5;

    // Apply cardType
    //Shinsei type weakness and strengths
    let actionType = damageData.actionElementType;
    let targetShinseiType = targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].shinseiType;
    let ownerShinseiType = userData.ShinseiParty[userData.currentShinsei].shinseiType;
    let shinseiTypeDifferenceMultiplier = CompareTypesAndGetDamage(actionType, targetShinseiType);
    let stab = actionType == ownerShinseiType ? 2 : 1;

    // Apply evade
    let attackEvaded = damageData.evadeRoll <
battleStatisticsCalculator.ApplyEvationCritics(targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].
evadeChance);

    //Attack
    if (!attackEvaded) {
        let bonusDamage =
battleStatisticsCalculator.GetBonusStat(userData.ShinseiParty[userData.currentShinsei],
getObjKey(constants.ShinseiStatsEnum, damageData.statBonusDamage), damageData.bonusPercent);
        let shinseiDamageStat = userData.ShinseiParty[userData.currentShinsei].ShinseiNormalizedStats.Attack;
        let rawDamage = battleStatisticsCalculator.GetFinalDamage(damageData.amount + bonusDamage,
shinseiDamageStat, stab, shinseiTypeDifferenceMultiplier, criticsMultiplier);
        let finalDamage =
battleStatisticsCalculator.GetDamageReceiveByTarget(targetPlayerData.ShinseiParty[targetPlayerData.currentShin
sei].ShinseiNormalizedStats.Defence, rawDamage);
        let finalDamageInt = Math.floor(finalDamage);

        targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].shinseiHealth =
Math.max(targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].shinseiHealth - finalDamageInt, 0);
    }
}
```

Heal

Heal is the same process as above but increasing the health stat.

```
function ExecuteHeal(healData, userData, otherData) {
    let targetPlayerData = healData.isSelfInflicted ? userData : otherData;
    let bonusHeal =
battleStatisticsCalculator.GetBonusStat(targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei],
getObjKey(constants.ShinseiStatsEnum, healData.statBonusDamage), healData.bonusPercent);
    let currentHealth = targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].shinseiHealth;
    let maxHealth =
targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].ShinseiOriginalStats.Health;
    targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].shinseiHealth = Math.min(maxHealth,
Math.floor((currentHealth + healData.amount + bonusHeal)));
}
```

Reflect Damage

Reflect Damage is a special status in which the damage that would be received is applied to the opponent.

```
function ReflectDamage(reflectDamageData, userData, otherData) {
    //check who gets the movement block
    //write the ban movements into the players data
    let targetPlayerData = reflectDamageData.isSelfInflicted ? userData : otherData;

    if (targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].reflectDamageLeft == null ||
targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].reflectDamageLeft <= 0)
        targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].reflectDamageLeft = 1;
    else
        targetPlayerData.ShinseiParty[targetPlayerData.currentShinsei].reflectDamageLeft += 1;
}
```

Skip Turn

This function is used both to wait for energy to recover and to apply effects such as Rooted, creates an empty turn.

StatSwap

Swaps the value of two stats temporarily, for example, a Shinsei with high defense could convert its defense into damage

EndGameAction

This action is special because it is called when the calculation decides that the game has ended, then it is responsible for delivering the virtual currency prizes, updating data in the tournament status and writing in the respective places who was the winner.

Create Match and Delete Match

These functions serve to create what is known as Share Group in Azure, the fights in sacred tails are Sharegroups where the two contenders are added, it has a key name that identifies it and saves both the turns and the results.

Create Match creates a new match and Delete deletes it.

This is an example of a Share Group of a fight.

Lobby

A large part of the lobby processing is done via server, this is responsible for saving and delivering the positions of all players.

The first step to be part of a lobby is to connect

ConnectToLobby

Connect to lobby first checks that the player who is trying to connect is using the latest version of the game to avoid corrupting data to other players, then it checks if the ID provided is truly a lobby and if it is, it checks if it has space for more players.

When connecting, it checks if the player has ELO among his data, if not, it sets the default ELO value and informs the player that he was able to connect correctly.

Tournaments

Sacred tails allows creating Knock Out tournaments between players in the form of elimination.

Create Tournament

This function creates a sharegroup and initializes the necessary values to add a user to this tournament.

```
module.exports = function (context, req) {
    if (!req.body) {
        context.res = { status: 200, body: { success: false, code: 20, message: 'Please send valid data',
data: null } };
        return context.done();
    }
    playfabController.Init();

    let tournamentId = req.body.FunctionArgument.Keys.tournamentId;
    let request = {
        SharedGroupId: tournamentId
    }

    PlayFabServer.CreateSharedGroup(request, (error, result) => {
        if (error !== null) {
            context.res = { status: 200, body: { success: false, code: 3, message: 'Could not create
tournament', data: null } };
            return context.done();
        }

        request["Data"] = {
            initTimeStage_1: req.body.FunctionArgument.Keys.initTime,
            tournamentName: req.body.FunctionArgument.Keys.tournamentName,
            maxPlayer: req.body.FunctionArgument.Keys.maxPlayer,
            tournamentDuration: req.body.FunctionArgument.Keys.tournamentDuration,
            currentStage: 0
        };

        PlayFabServer.UpdateSharedGroupData(request, (error, result) => {
            if (error !== null) {
                context.res = { status: 200, body: { success: false, code: 2, message: 'Could not create
tournament', data: null } };
                context.done();
            }
            else {
                let displayTournamentData = { ...request.Data }
                delete displayTournamentData.numberOfBrackets;
                delete displayTournamentData.tournamentDuration;
                delete displayTournamentData.currentStage;

                let requestData = {};
                requestData[tournamentId] = JSON.stringify(displayTournamentData);
                let displayRequest = { PlayFabId: constants.availableTournamentPID, Data: requestData }
                PlayFabServer.UpdateUserData(
                    displayRequest,
                    (error, result) => {
                        if (error !== null)
                            context.res = { status: 200, body: { success: false, code: 1, message: 'Could not
set display data tournament', data: null } };
                        else
                            context.res = { status: 200, body: { success: true, code: 0, message: 'Tournament
Created', data: displayRequest } };

                        context.done();
                    }
                );
            }
        });
    });
}
```

Get Tournament List

With this function, users from the game can see which tournaments are active at the moment and request to join them by code. When a tournament is created, a fake user is added that saves the existing tournaments in its keys, this function downloads all these keys, filters them by hour and displays them.

```
module.exports = function (context, req) {
    if (!req.body) {
        context.res = { status: 200, body: { success: false, code: 20, message: 'Please send valid data',
data: null } };
        return context.done();
    }

    playfabController.Init();

    let getTounrnamentDataRequest = {
        PlayFabId: "7F1965D480D991B5",
    }

    try {
    PlayFabServer.GetUserData(getTounrnamentDataRequest, (error, result) => {
        if(error != null){
            //:C
        }
        else{
            let data = [];
            Object.keys(result.data.Data).forEach(key => {
                let tournamentData = JSON.parse(result.data.Data[key].Value);
                tournamentData.tournamentId = key;
                data.push(tournamentData);
            })
            context.res = { status: 200, body: { success: true, code: 0, message: "here is all tournaments",
data} };
            context.done();
        }
        });
    }
    catch (err) {
        context.res = { status: 200, body: { success: false, code: 404, message: err.message, data: err.data }
};
        context.done();
    }
}
```

Check Bracket Data

Like the lobby, tournaments have their own function to check the state of the tournament at certain times and that function is this.

The first thing it does is check if this function was called by both players, the player can call it using the ready button within the game, when the function is called it verifies the maximum time limit to notify that it is ready and if one of the two has not called the function, the one who did wins, and if neither of them called it, the game decides that there will be a default winner in the next bracket.

The first time a request arrives where both are marked as ready, the lobby is notified that a fight is ready and what the match ID of that fight is for the tournament, then both players enter a fight.

After everything is processed like in a normal game, the game notifies that there was a winner and the winning player stays calling the function GetCurrentBracketsData

Get Current Brackets Data

In this function, the server verifies that all games have a winner in order to proceed to update the state of the tournament and

create the next branches.

Varius Server Side Functions

ReportBug

This function allows users to send a bug report through a small form and a button, which includes a screenshot of the user.

```
module.exports = function (context, req) {
    if (!req.body) {
        context.res = { status: 200, body: { success: false, code: 3, message: 'Please send valid data', data:
null } };
        context.done();
        return;
    }

    playfabController.Init();

    let playFabId = req.body.CallerEntityProfile.Lineage.MasterPlayerAccountId;
    let picture = req.body.FunctionArgument.Keys.picture;
    let message = req.body.FunctionArgument.Keys.message;
    let matchId = req.body.FunctionArgument.Keys.matchId;

    let date = new Date();
    try {
        UploadReport(matchId + ":" + date.getTime() + ":" + playFabId, picture, message, context)
    }
    catch (err) {
        CatchError(context, { code: 404, message: "Unexpected error on bug report", data: null });
    }
}


//### Helpers ###
function CatchError(context, err) {
    context.res = { status: 200, body: { success: false, code: err.code, message: err.message, data: err.data
} };
    context.done();
}

function UploadReport(code, picture, message, context) {
    let debugData = {};
    debugData[code] = JSON.stringify({ picture: picture, message: message, timestamp: new Date() });

    let reportBugRequest = {
        SharedGroupId: "Bugs",
        Data: debugData
    }
    PlayFabServer.UpdateSharedGroupData(reportBugRequest, (error, result) => {
        if (error == null) {
            context.res = { status: 200, body: { success: true, code: 0, message: "Bug report succesfully",
data: null } };
            context.done();
        }
        else {
            CatchError(error);
        }
    });
}
```

GetRewards Rank System

This function serves the data stored in a json that configures the rewards.

Shinsei Generator

The objective of this document is to provide a clear and detailed guide of the Shinsei generation process and the main components of the tool so that any programmer can develop the content generation activity for Sacred Tails NFTs.

SetUp:

- Unity version: 2020.3.23.f1
- Repository branch: Feat_NFTVideoGen
- Scene: Character Generator

Structure and components of the scene

The scene is divided into 3 main components: Environment, Lighting, and Core.

Environment:

Contains the props and assets that make up the scene in which the Shinsei generator videos are captured and processed, and it is organized as follows:

The main elements of the stage are separated by category, vegetation, props (buildings and accessories), particle systems, and the sprites that make up the horizon. The Shinsei Gate is outside of the containers because it contains the flag that indicates the type of Shinsei. This flag is the only object in the environment that has a reference within the "ShinseiGenerator.cs" script, which will be detailed later.

Lighting:

The lighting within the scene is handled by different sources to keep in mind in case changes are required. The distribution of lights within the scene is as follows:

The main thing to keep in mind is the directional light controller, which contains the directional light source of the scene, and the light manager, which contains the components necessary to control the time of day and the sky tones.

The other light sources are located according to the elements of the scene that they affect, in Shinsei only lights and Flag Only lights are light sources that only affect the shinsei and the flag respectively. while the support light is only an auxiliary light that affects all elements within the scene.

Core:

Contains the elements responsible for the generation and recording of shinseis. And these are its main elements:

Shinsei Generation Manager is the object that contains the ShinseiGenerator.cs class and is where all the actions of generating shinseis, recording videos and storing JSONs are performed. There are 2 cameras, one responsible for recording the content

(Video Capture) and another that only serves the purpose of showing us the content of the scene in the game view. The ShinseiWardrobe is responsible for changing the parts of the Skinned_Shinsei_Atlas according to the DNA and is referenced within the ShinseiGenerator.cs, while the color swapper is responsible for assigning the correct color palette to the shinsei in question.

Inicialization:

To start, the types of Shinseis that are going to be generated in the pool of types within the CharacterGenerator.cs in the Shinsei generation manager must be established.

In this example, the corresponding families for the second generation of Shinseis are set, so that the Shinseis that will be generated do not have parts of foreign families. Subsequently, the scriptable objects that will contain the generated Shinseis must be created. The name of the scriptable object is "Generated Shinsei Container" which is a database of Shinseis. It is recommended to use this structure for ease of management.

Create a shinsei container per rarity-type of shinsei included in the sale, more or less in the following way:

Shinsei Generation:

To begin generating the shinseis that will be released, you must first drag one of the Generated Shinsei Containers to the InputSO and OutputSO fields in the Shinsei Generator manager, then set the number of shinseis that will be generated (Shinseis to generate) and the index with which the new shinsei count should start (Last Index), this index should be the value of the last shinsei generated in the previous sale or in the last batch of generated shinseis (for example, if the last shinsei in the previous sale has an index of 2999, the Last Index should be assigned as 3000).

Once this is established, you must proceed to enter play mode, and there choose the type and rarity of the Shinseis that will be generated to fill that Generated Shinsei Container. Once the type and rarity are chosen, within playmode you must activate the "Invoke" button within the component.

If the Shinsei type (Desired Type) is not contained within the type pool, then the function will not be executed and a warning log will be displayed. To generate Shinseis of celestial type, only tiers from legendary1 to legendary2 can be chosen.

Repeat this process for all SOs created for the sale.

Note:

As the scriptable objects (Generated shinsei container) are filled in runtime, they are never saved in git changes. It is suggested that after generating a batch, add and remove an element from the scriptable and then save the project to ensure they are included in the changes.

Generation of content for Minting

Generation of videos

To generate videos, first a path must be assigned in the VideoCaptureManager and VideoCapture components within the object in

the scene called VideoCapture. (Here you can also modify the desired resolution and frame rate.)

Once the path is set, you must select the Generated Shinsei Container from which the videos will be generated and drag it to the InputSO and OutputSO fields of the ShinseiGenerator in the shinsei generation manager.

Then, enter play mode and press the record button within the component. The editor will take approximately 30 seconds per shinsei, and will generate the videos in the specified folder. Once finished with a batch, another shinsei container should be assigned and the button pressed again to continue the task until all videos are generated.

Upload to IPFS: Once all videos are generated, they must be uploaded to the timba drive, specifically in the following path: Instinct>SacredTails>Mint>Videos>Folder with the sale number, all while having a backup of the information. Then, from a single PC, all videos must be downloaded and then uploaded to an IPFS folder (IPFS desktop is recommended).

Once there, you must copy the CID of the folder and give it to the client, so that they can import the files to pinata. During this process, the computer that uploaded the files must be kept on and have a stable internet connection. (It is of vital importance to verify that the videos do not require corrections and are correct before the next step). If all files are found and it is certain that no video needs to be corrected, the next step is to copy that CID of the folder and paste it within the ShinseiGenerator.cs script as the value of the string ipfsCID in the following line.

Make sure to save changes in the script.

Generation of Metadata: To generate the metadata JSONs, all the Shinsei containers that were used for video generation must be dragged to the following list in the ShinseiGenerator.cs.

Subsequently, enter playmode and press the Save Json Files button of the component in question.

The editor will appear to freeze for a brief moment, and when it finishes generating the Jsons, the files will be found in the following path of the project: SacredTails(root)>GeneratedShinseis>JSON> Hoardable/OpenSea. It is recommended to compress these folders and send them to the client when the process is finished, they should be responsible for using them for the minting process.