GUIDE

CONTENTS

STEP 1: SET UP YOUR DEVELOPMENT ENVIRONMENT

1. Visit the official Unity website (https://unity.com) and download the Unity Hub.
2. Install the Unity Hub on your computer and launch it.
3. In the Unity Hub, navigate to the "Projects" tab and click on the "New" button to create a new project.
4. Specify a project name and location on your computer.
5. Choose a Unity version and template for your project (e.g., 3D, 2D).
6. Click on the "Create" button to create the project.

STEP 2: UNDERSTAND THE UNITY EDITOR

1. Familiarize yourself with the various windows in the Unity Editor, including the Scene View, Game View, Hierarchy, Inspector, Project, and Console.
2. Learn how to navigate within the Scene View using the mouse and keyboard shortcuts.
3. Experiment with selecting and manipulating objects in the Scene View, such as moving, rotating, and scaling.
4. Understand the purpose and functionality of each window and how they relate to the game development process.

STEP 3: CREATE A NEW SCENE

1. In the Unity Editor, navigate to the "File" menu and select "New Scene" to create a new empty scene.
2. Save the scene by clicking on "File" -> "Save Scene" or using the shortcut Ctrl+S (Cmd+S on macOS).
3. Specify a name and location for the scene file and click on the "Save" button.

STEP 4: IMPORT ASSETS

1. Collect or create the assets you need for your game, such as 3D models, textures, audio files, and animations.
2. In the Unity Editor, navigate to the "Project" window and create new folders to organize your assets.
3. Locate the assets on your computer, and drag and drop them into the appropriate folders within the "Project" window.
4. Unity will automatically import and process the assets, making them available for use in your game.

STEP 5: BUILD THE GAME WORLD

1. Begin by creating a terrain if your game requires one. In the Unity Editor, navigate to the "GameObject" menu and select "3D Object" -> "Terrain."

2. Sculpt the terrain using Unity's terrain editing tools. Adjust the brush size, strength, and height to shape the landscape.

3. Texture the terrain by applying textures to different terrain layers. Use the "Paint Texture" tool to paint textures onto the terrain surface.

4. Place and arrange 3D models and objects in the scene to create the game world. Use the Transform tool to position, rotate, and scale objects as desired.

5. Adjust the scene's lighting by placing light sources in the scene and modifying their properties in the Inspector window. Experiment with different lighting techniques to achieve the desired atmosphere.

STEP 6: SET UP LIGHTING

1. In the Unity Editor, navigate to the "Window" menu and select "Rendering" -> "Lighting" to open the Lighting window.

2. Choose the appropriate lighting mode for your game (e.g., Realtime, Baked, Mixed).

3. Place light sources in the scene by creating new GameObjects and adding light components to them. Adjust the properties of the light sources, such as color, intensity, range, and shadows, to achieve the desired lighting effect.

4. Use the lighting window to adjust other lighting settings, such as ambient light, lightmapping, and light probes, to enhance the scene's visuals.

STEP 7: IMPLEMENT GAMEPLAY MECHANICS

1. Create scripts to implement gameplay mechanics using C# or Unity's visual scripting systems, such as Bolt or Playmaker.

2. In the Unity Editor, navigate to the "Assets" window and select "Create" -> "C# Script" to create a new script. Name it according to the functionality it will implement.

3. Double-click on the script file to open it in your preferred code editor.

4. Write the necessary code to implement the desired gameplay mechanics. For example, if you want to implement player movement, write code to handle input, translate the player's position, and respond to collisions.

5. Attach the script to the appropriate game objects in the scene by dragging and dropping it onto the object in the Hierarchy or Inspector window.

STEP 8: ADD AUDIO

1. Import or create audio files for sound effects, background music, and voice-overs.

2. In the Unity Editor, navigate to the "Project" window and locate the audio files.

3. Select the audio files and drag them into the desired game objects or create an empty GameObject to serve as an audio source.

4. Adjust the audio source component's properties in the Inspector window, such as volume, pitch, and spatial blend, to achieve the desired audio effect.

5. Use scripting to control when and how the audio is played, stopped, or manipulated based on game events.

STEP 9: CREATE USER INTERFACE (UI)

1. Design the UI elements for your game using Unity's UI system. In the Unity Editor, navigate to the "GameObject" menu and select "UI" to access UI components like buttons, panels, and text.

2. Drag and drop the UI components into the Scene View or Hierarchy window to create the desired UI layout.

3. Customize the appearance of the UI elements using the Inspector window, adjusting properties such as size, position, color, and text content.

4. Write scripts to handle user interactions with the UI, such as button clicks or menu navigation. Attach these scripts to the appropriate UI elements.

5. Use scripting to update the UI dynamically based on game events, such as displaying player scores or health bars.

STEP 10: TEST AND DEBUG

1. Regularly playtest your game to identify and fix bugs or gameplay issues.

2. Use Unity's debugging tools, such as the Console window, to track and resolve errors or unexpected behavior.

3. Set breakpoints in your scripts to pause the game execution and inspect variable values at specific points.

4. Utilize logging statements to output information to the console for debugging purposes.

5. Involve others in playtesting your game to gather feedback and insights on gameplay, difficulty, and overall user experience.

STEP 11: OPTIMIZE PERFORMANCE

1. Profile your game to identify performance bottlenecks using Unity's built-in profiling tools or external profiling tools like Unity Profiler or Visual Studio Profiler.

2. Optimize assets by reducing polygon counts, adjusting texture sizes or compression settings, and removing unnecessary components or scripts.

3. Use object pooling techniques to efficiently manage frequently created and destroyed objects, reducing performance overhead.

4. Implement batching to reduce draw calls by combining multiple objects into a single draw call where possible.

5. Apply performance optimizations specific to your game's requirements, such as level of detail (LOD) systems or occlusion culling, to improve rendering performance.

STEP 12: BUILD AND PUBLISH

1. Determine the target platform for your game (e.g., PC, Mac, iOS, Android).
2. In the Unity Editor, navigate to the "File" menu and select "Build Settings."
3. In the Build Settings window, choose the target platform and click on the "Switch Platform" button.
4. Configure additional settings for the target platform, such as resolution, quality settings, or platform-specific optimizations.
5. Click on the "Build" button to build the game for the selected platform. Specify a location to save the built game files.

Once the build process is complete, you can distribute the game through app stores, game platforms, or your own website according to the guidelines of each platform.

STEP 13: UNDERSTAND THE SCRIPTS

DEATHROLL

```
using UnityEngine;

using UnityEngine.SceneManagement;
using UnityEngine.UI;
using System.Collections;

public class deathRoll : MonoBehaviour
{
    public GameObject playerObject;
    public RawImage blackoutImage;
    public Text deathText;
    public float detectionRange = 1f;
    public float deathDuration = 5f;
    public AudioSource deathAudioSource;
    public AudioClip deathAudioClip;
```

This script handles the player's death in the game. Here's a simpler breakdown:

- The script keeps track of important game objects like the player, blackout image, and death text. It also stores values for detection range and death duration.
- In the **Update** function, it checks if the player object is missing and tries to find it using the "Player" tag. Then, it calculates the distance between the current object and the player. If the distance is within the detection range and the player is not dead and hasn't died before, it triggers the **PlayerDeath** function.
- The **PlayerDeath** function marks the player as dead and prevents multiple deaths. It starts a coroutine called **DeathSequence**.
- The **DeathSequence** coroutine handles the death sequence. It activates the blackout image and death text, waits for a specific duration, and then loads a specific scene (number 6 in the build index).

PLAYERCASTING

```
using UnityEngine;


public class PlayerCasting : MonoBehaviour {

    public static float DistanceFromTarget;
    public float ToTarget;


    void Update () {
        RaycastHit Hit;
        if (Physics.Raycast (transform.position, transform.TransformDirection
```

This script is called "PlayerCasting" and it handles raycasting from the player's position to detect the distance between the player and the target object

- The script defines two variables: **DistanceFromTarget** (a static variable accessible from other scripts) and **ToTarget** (a variable specific to this script).
- In the **Update** function, it uses raycasting to detect if there is an object in front of the player. It shoots a ray from the player's position in the forward direction and checks if it hits something.
- If a collision occurs (the ray hits an object), the distance from the player to the hit object is stored in the **ToTarget** variable.
- The value of **ToTarget** is then assigned to the **DistanceFromTarget** variable, allowing other scripts to access and use the distance value.

STALKERAI

```
using System.Collections;

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class StalkerAI : MonoBehaviour
{
    public GameObject stalkerDest;
    NavMeshAgent stalkerAgent;
    public GameObject stalkerEnemy;
    public static bool isStalking;
    public AudioSource stalkerSound;
```

This script is called "StalkerAI" and it controls the behavior of a stalker enemy character in the game.

- The script begins by declaring some variables:
    - **stalkerDest**: a reference to the destination object the stalker should move towards.
    - **stalkerAgent**: a reference to the NavMeshAgent component attached to the stalker character, used for navigation.
    - **stalkerEnemy**: a reference to the stalker enemy object itself.
    - **isStalking**: a static boolean variable that determines whether the stalker is actively stalking the player or not.
    - **stalkerSound**: an AudioSource component for playing stalker-related sounds.
- In the **Start** function, the script initializes the **stalkerAgent** variable by getting the NavMeshAgent component attached to the same game object as this script. It also activates the **stalkerEnemy** object.
- The **Update** function is called every frame. If the **isStalking** variable is **false**, the stalker enemy plays an animation called "Sad Idle" using its Animator component. This means the stalker is not actively stalking the player.
- If **isStalking** is **true**, the stalker enemy plays an animation called "Swagger Walk" using its Animator component. The **stalkerAgent** is instructed to move towards the position of the **stalkerDest** object using the **SetDestination** method.
- The **PlayStalkerSound** function checks if the **stalkerSound** is not currently playing and plays the sound if it's not already playing.

STALKERTRIGGER

```
using UnityEngine;


public class StalkerTrigger : MonoBehaviour
{
    public GameObject runText;
    private bool isPlayerInside = false;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            isPlayerInside = true;
```

This script is called "StalkerTrigger" and it handles triggering events when the player enters or exits a trigger zone.

- The script begins by declaring some variables:
    - **runText**: a reference to the GameObject that represents the text to be displayed when the player enters the trigger zone.
    - **isPlayerInside**: a boolean variable that keeps track of whether the player is inside the trigger zone or not.
- The **OnTriggerEnter** function is called when a collider enters the trigger zone. If the collider has a tag "Player", the **isPlayerInside** variable is set to **true**. The **runText** object is then activated, displaying the "runText" to the player. The **Invoke** function is used to schedule the execution of the **DisableRunText** function after a delay of 5 seconds.
- The **OnTriggerExit** function is called when a collider exits the trigger zone. If the collider has a tag "Player", the **isPlayerInside** variable is set to **false**. The **runText** object is deactivated, hiding the "runText" from the player.
- The **DisableRunText** function is called after the specified delay in the **Invoke** function. It checks if the player is still inside the trigger zone (**isPlayerInside** is **true**), and if so, deactivates the **runText** object

DOORINTERACTION

```
using System.Collections;

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class DoorInteraction : MonoBehaviour
{
    public float TheDistance;
    public GameObject ActionDisplay;
    public GameObject ActionText;
```

- The script begins by declaring variables:
    - **TheDistance**: a float variable that represents the distance from the player to the door.
    - **ActionDisplay**, **ActionText**, **ExtraCross**: GameObjects representing UI elements for displaying the interaction action and crosshair.
    - **lockedDoor**: an AudioSource component for playing a sound when the door is locked.
- The **Update** function is used to update the value of **TheDistance** by retrieving it from the **PlayerCasting.DistanceFromTarget** variable.
- The **OnMouseOver** function is called when the mouse pointer is over the door object. It checks if **TheDistance** is less than or equal to 7 units. If true, it activates the **ExtraCross** object, sets the text of the **ActionText** component to "Open door", and activates the **ActionDisplay** and **ActionText** objects.
- Inside the **OnMouseOver** function, there is an additional check for the "Action" button press (**Input.GetButtonDown("Action")**). If **TheDistance** is less than or equal to 4 units, it disables the **BoxCollider** component attached to the door, hides the **ActionDisplay**, **ActionText**, and **ExtraCross** objects, and starts the **DoorReset** coroutine.
- The **OnMouseExit** function is called when the mouse pointer exits the door object. It deactivates the **ActionDisplay**, **ActionText**, and **ExtraCross** objects.
- The **DoorReset** coroutine is responsible for resetting the door state after a delay. If **GlobalInventory.firstDoorKey** is **false** or **true**, it plays the **lockedDoor** sound, waits for 1 second, and then enables the **BoxCollider** component attached to the door.

ENDGAME

```
using System.Collections;

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class EndGame : MonoBehaviour
{
    public float TheDistance;
    public GameObject ActionDisplay;
    public GameObject ActionText;
    public GameObject ExtraCross;
```

- The script begins by declaring variables:
    - **TheDistance**: a float variable that represents the distance from the player to the door.
    - **ActionDisplay**, **ActionText**, **ExtraCross**: GameObjects representing UI elements for displaying the interaction action and crosshair.
    - **lockedDoor**: an AudioSource component for playing a sound when the door is locked.
- The **Update** function is used to update the value of **TheDistance** by retrieving it from the **PlayerCasting.DistanceFromTarget** variable.
- The **OnMouseOver** function is called when the mouse pointer is over the door object. It checks if **TheDistance** is less than or equal to 7 units. If true, it activates the **ExtraCross** object, sets the text of the **ActionText** component to "Open door", and activates the **ActionDisplay** and **ActionText** objects.
- Inside the **OnMouseOver** function, there is an additional check for the "Action" button press (**Input.GetButtonDown("Action")**). If **TheDistance** is less than or equal to 4 units, it disables the **BoxCollider** component attached to the door, hides the **ActionDisplay**, **ActionText**, and **ExtraCross** objects, and starts the **DoorReset** coroutine.
- The **OnMouseExit** function is called when the mouse pointer exits the door object. It deactivates the **ActionDisplay**, **ActionText**, and **ExtraCross** objects.
- The **DoorReset** coroutine is responsible for resetting the door state after a delay. If **GlobalInventory.firstDoorKey** is **false** or **true**, it plays the **lockedDoor** sound, waits for 1 second, and then enables the **BoxCollider** component attached to the door.

EYEPICKUP

```
using System.Collections;

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class EyePickup : MonoBehaviour
{
    public float TheDistance;
    public GameObject ActionDisplay;
    public GameObject ActionText;
    public GameObject ExtraCross;
    public GameObject theKey;
```

- The script begins by declaring variables:
    - **TheDistance**: a float variable that represents the distance from the player to the eye object.
    - **ActionDisplay**, **ActionText**, **ExtraCross**: GameObjects representing UI elements for displaying the interaction action and crosshair.
    - **theKey**: the GameObject representing the eye object to be picked up.
- The **Update** function is used to update the value of **TheDistance** by retrieving it from the **PlayerCasting.DistanceFromTarget** variable.
- The **OnMouseOver** function is called when the mouse pointer is over the eye object. It checks if **TheDistance** is less than or equal to 5 units. If true, it activates the **ExtraCross** object, sets the text of the **ActionText** component to "Pick Up Key", and activates the **ActionDisplay** and **ActionText** objects.
- Inside the **OnMouseOver** function, there is an additional check for the "Action" button press (**Input.GetButtonDown("Action")**). If **TheDistance** is less than or equal to 5 units, it disables the **BoxCollider** component attached to the eye object, hides the **ActionDisplay**, **ActionText**, and **ExtraCross** objects, deactivates the **theKey** object (eye), and sets **GlobalInventory.theEye** to **true** indicating that the eye has been picked up.
- The **OnMouseExit** function is called when the mouse pointer exits the eye object. It deactivates the **ActionDisplay**, **ActionText**, and **ExtraCross** objects.

FLASHLIGHT

```
using UnityEngine;

using System.Collections;

public class Flashlight : MonoBehaviour
{
public Light flashlightLightSource;
public bool lightOn = true;
public float lightDrain = 0.1f;
private static float batteryLife = 0.0f;
public float maxBatteryLife = 100.0f;
```

- The script begins by declaring variables and assigning references to various components and assets in the game.
- The **Start** function initializes the **batteryLife** to the maximum battery life value and retrieves the **Light** component attached to the flashlight object.
- The **Update** function is called every frame. It updates the battery life based on the light drain rate and adjusts the intensity of the flashlight's light source based on the remaining battery life.
- The **OnGUI** function is responsible for displaying the battery life as a progress bar on the screen.
- The **toggleFlashlight** function enables or disables the **Light** component of the flashlight based on the current **lightOn** state.
- The **toggleFlashlightSFX** function sets the appropriate audio clip for the flashlight turning on or off and plays the audio.

LOCKEDDOOR

```
using System.Collections;

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class LockedDoor : MonoBehaviour
{
    public float TheDistance;
    public GameObject ActionDisplay;
    public GameObject ActionText;
```

- The script starts by declaring variables and assigning references to various components and game objects.
- The **Update** function updates the **TheDistance** variable based on the player's distance from the locked door.
- The **OnMouseOver** function is called when the player's mouse hovers over the door. It checks if the player is within range and displays the action prompt to open the door.
- If the player presses the designated action button (**Action**) while in range, the **OnMouseOver** function checks if the player has the necessary key (**GlobalInventory.firstDoorKey**). If the key is not available, the locked door sound is played and the door remains locked for a brief period before becoming interactable again. If the key is available, an animation is played on the door, the door creak sound is played, and the door becomes permanently open.
- The **OnMouseExit** function is called when the player's mouse is no longer hovering over the door. It hides the action prompt and the crosshair.
- The **DoorReset** coroutine is responsible for handling the behavior of the locked door. If the player doesn't have the key, it plays the locked door sound and waits for a short duration before making the door interactable again. If the player has the key, it plays an opening animation on the door, plays the door creak sound, and waits for a slightly longer duration before making the door non-interactable.

MENUFUNCTION

```
using System.Collections;

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class MenuFunction : MonoBehaviour
{
    public GameObject fadeOut;
    public GameObject loadText;
    public AudioSource buttonClick;
```

- The script contains various public variables representing game objects and audio sources that are assigned in the Unity Editor.
- The **PlayButton** function is called when the "Play" button is clicked. It starts the **PlayStart** coroutine.
- The **PlayStart** coroutine handles the transition from the main menu to the gameplay scene. It activates a fade-out effect by enabling the **fadeOut** game object and plays a button click sound. After a 3-second delay, it activates the **loadText** game object to display a loading message and loads scene number 2 (presumably the gameplay scene).
- The **LoadSite** function is called when the "Load Site" button is clicked. It opens a specified URL (https://sirsaucey.github.io/GIP/site/index.html) in the default web browser using the **Application.OpenURL** method.
- The **QuitButton** function is called when the "Quit" button is clicked. It checks if the game is running in the Unity Editor (**UNITY_EDITOR**) and sets the **UnityEditor.EditorApplication.isPlaying** flag to **false** to stop the game. If the game is running in a standalone build, it calls **Application.Quit()** to quit the application.
- The **LoadScene6** function is called when a specific event occurs (not mentioned in the script). It loads scene number 8.

STEP 14: IMPORT THE SOURCE CODE

Click on the following link to download the source code: Download Source Code.

In the Unity Hub, navigate to the "Projects" tab and select the project you created in Step 1.

Click on the "Open" button to open the project in the Unity Editor.

In the Unity Editor, navigate to the "File" menu and select "Open Project."

Browse to the location where you extracted the downloaded source code files and select the project folder.

Click on the "Open" button to import the source code into the Unity Editor.