Milestone 3 – Game Menus and Collectables

This is an *Individual* assignment.

Late Policy

See syllabus for details.

Use Unity version: <SEE SYLLABUS FOR REQUIRED VERSION NUMBER>

Description

In this assignment, you will be modifying your M1/M2 project to include a game menu with post-processed scene imagery. Additionally, you add a ball collectable that SomeDude_RootMotion will pick up. The ball collectable will be implemented by creating a GameObject with a collider that is a trigger. Lastly, you will make a game object that animated is some fashion when the player gets close enough. For instance, this could be an alien plant that retracts if anything gets too close and then pops back out when the player leaves.

For extra credit, you can make SomeDude_RootMotion throw the collected ball

Regarding the game menu, you will leverage the "new" UI system introduced in Unity 4.6 / Unity 5. You will **not** be using the legacy GUI Skin (aka IMGUI) for this milestone. This is mentioned because students that haphazardly search for online resources may find very old tutorials using the old system.

If you aren't happy with your M1/M2, you can start from the original git project.

Walkthrough

Game Menu System

First, remove/disable the pause script from M2.

You will be creating a very simple game menu. The game menu you will be making is very minimal and boring but should get you familiar with some of the basics. The menu you and your team make for the final project should be way cooler!

Set up an interesting backdrop for your menu

In order to make a start menu, you will actually make a dedicated Unity scene with an overlay menu and interesting visual features. In this case, you will use your current level with some modifications to serve as a backdrop. You will be going for an effect similar to that used in many games such as Half-Life 2: https://www.youtube.com/watch?v=pveer0jDCmk

In order to support a game start menu, you should first make a copy of your existing M1/M2 scene.

Execute "save as" and name the scene *GameMenuScene*. Next up, modify your Main Camera to disable the Third Person Camera script (or remove it). Make the camera look down on some interesting part of your scene.

In the future, you can add a script to the camera to make it do something interesting to make the menu more compelling.

Post Processing

You will need to go to Window->Package Manager and install the Post Processing package. Documentation is here:
https://docs.unity3d.com/Packages/com.unity.postprocessing@3.1/manual/index.html

The quickest approach is to add a Post Process Layer and Post Process Volume to your Camera. Set the Post Process Layer's Volume Blending:Trigger to the Camera's transform. Change the layer of your Camera to user-defined layer PostProcessing or another user defined layer of your choosing. Set the Post Process Layer's Volume Blending:Layer setting to the same layer (e.g. PostProcessing). In the Post Process Volume component, create a profile and make sure it is assigned to the same component. Lastly, in the new profile enable your desired effects. Depth of Field can be used for a blurry effect. Grain might also work (and is good for confirming that post processing is working), possibly stacked with other effects. You will probably need to enable 'Is Global' in the Post Processing Volume to get the effect to work.

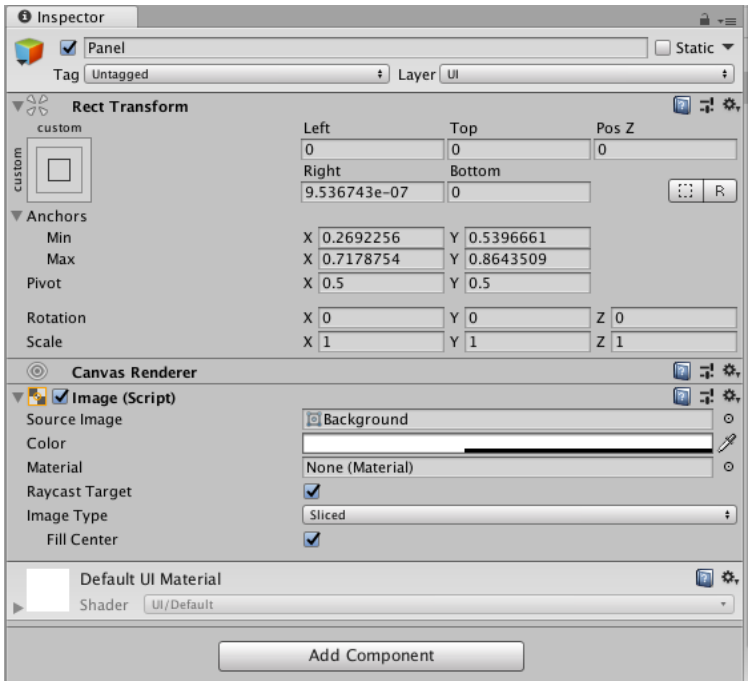
On OSX, if you get errors from shaders then you might enable Metal Support for rendering. Go to Edit->Project Settings->Player: Other Settings->Metal Editor Support and enable.

GUI Creation

Next up you will be adding GUI elements that overlay the screen. These elements will include a Canvas to draw on and a panel with buttons on it to execute commands such as "start new game."

A Canvas is already in the starter project and is named UICanvas in the Hierarchy Window. In future Projects, you can add a GameObject->UI->Canvas to your scene (confirm Render Mode is Screen Space Overlay)

Next add a GameObject->UI->Panel to your Canvas. In the Scene window, switch to 2D mode so you can see it in the correct perspective.



Note that if your Panel's Rect Transform Anchors are at the maximum extents of the Canvas (default) then your Panel sizing will scale with Canvas resolution and aspect. If you drag the circular blue control points you will manipulate pixel offsets (Left, Right, Top, Bottom) from the Rect Transform Anchors. This will create a sort of hybrid situation where the panel will partially scale with Canvas resolution but there are absolute pixel count offsets from the Anchors. Often, this is not what you want. Typically, a designer wants to scale with resolution or have the same pixel size always. If you want to scale with resolution, The Rect Transform's Left, Right, Top, Bottom should stay at 0 offset. Then set The Rect Transform's Min and Max Anchors to the proportional offset within the Canvas. If you want your Panel to always be the same size, shrink your Anchors down to the same X,Y position (say the center of the canvas or a corner) then your panel will maintain the configured pixel size and position offset from the Anchors.

Just be aware of a couple issues. If you try to support extreme differences in resolution, absolute pixel offsets could result in impossible dimensions. In those cases, Unity shows a red X over the UI widget while in the editor mode. Also, some things like fonts won't scale automatically even if you have Rect Transform Anchors set to scale proportionally. Luckily, there is the CanvasScaler that can be configured to help with this:
https://docs.unity3d.com/Manual/script-CanvasScaler.html

This Rect Transform and the mix of relative and absolute positioning is an important concept in Unity UI. To aid with GUI design, try switching the Game view from "Free Aspect" to a common full screen aspect ratio. This will help you visualize positioning.

At this point, you should decide how you want to anchor your panel and adjust accordingly. Next size your panel so that it only fills a central area of the screen, say quarter-sized. In other

words, make the panel small enough that you can see the background scene to the sides of the panel.

Now add a GameObject->UI->Button to your Panel. Note Unity's default Rect Transform for the button versus the Panel. Change the default text to "Start Game".

Now create a script called GameStarter and add public method StartGame(). This method should simply call SceneManager.LoadScene ("SCENE_NAME") where SCENE_NAME is the name of your original interactive scene. This command simply loads the scene during runtime. Add this script to your Button. (Note: For better software engineering, in the future you may wish to place game management functionality separately from the UI elements.)

Next up, view the Button in the Inspector. Under the Button script settings, you should see a OnClick() event. Assign the GameObject callee slot to the same Button that the script belongs to. Now use the dropdown to find GameStarter->StartGame and pick it. Now when you click the "Start Game" button at runtime the level will load!

Make another button called "Exit". Make use of the existing script called GameQuitter, which is very similar to GameStarter but the command executed is Application.Quit() and is contained within the QuitGame() method of the GameQuitter script. Wire up QuitGame() to the "Exit" button. Note that the existing GameQuitter immediately calles QuitGame() if "x" is pressed.

Go ahead and test/debug your menu buttons. Try testing with a build a well. Note that your build must include both the GameMenuScene and your actual gameplay scene to work correctly.

Lastly, be aware that your characters are still enabled in your scene. If you hit keys on the keyboard, you'll probably see the selected character walk around. This is ok for this assignment, but something you'll want to avoid in your final game. If you like, you can remove the characters or just disable their keyboard controls.

Setting Up the In-Game Menu

Another important type of menu is an in-game menu. This type of menu typically provides features like restarting the level, exiting the software, and also serves to pause the game. Let's make a simple in-game menu with these features.

Save your GameMenuScene and reopen your original scene with the controllable characters. Leveraging what you learned previously, add a screen space overlay Canvas with a centered panel that is smaller than the overall screen resolution. On the panel, create buttons with text "Restart Level" and "Quit Game". You can reuse the GameStarter and GameQuitter scripts from above to wire up your "Restart Level" and "Quit Game" buttons.

Go ahead and test/debug your menu buttons. The menu will be in the way of controlling your characters but you will deal with that next.

Next up, let's add the ability for the menu to be opened/closed when hitting the Escape key on the keyboard. First, add a Canvas Group component to your Canvas. The Canvas Group allows you to control all child UI components and their descendants in a coordinated fashion. In the Inspector view of the Canvas Group component, set Alpha to 0, Interactable to false, and Block Raycasts to false. This will completely hide and disable your menu. Now, you'll make a script that responds to the Escape key to toggle those settings on/off.

Create a new script called PauseMenuToggle. This script should be attached to the Canvas. The script will be acting on the CanvasGroup so add a component requirement to the class: [RequireComponent(typeof(CanvasGroup))]

Use GetComponent<CanvasGroup>() to grab a reference in Awake() and store in a private member variable named canvasGroup. You should print a Debug.LogError() if GetComponent() doesn't find the component you are looking for.

In Update() of the PauseMenuToggle script, add the following:

```
if (Input.GetKeyUp (KeyCode.Escape)) {

   if (canvasGroup.interactable) {

      canvasGroup.interactable = false;
      canvasGroup.blocksRaycasts = false;
      canvasGroup.alpha = 0f;

    } else {

      canvasGroup.interactable = true;
      canvasGroup.blocksRaycasts = true;
      canvasGroup.alpha = 1f;
    }

}
```

Note that Input.GetKeyUp() should eventually be replaced with Input.GetButtonUp() with a virtual button created in the InputManager settings. This will allow multiple game controllers to map to common input events (e.g. simultaneous keyboard, and handheld game controller support).

Try running your game and use Escape to demonstrate toggling your menu. One problem you may notice is that your game is still running in the background of the menu. That may be ok for a multi-player game, but for single player you should pause your game.

Go back to PauseMenuToggle and add Time.timeScale = 0f for when the in-game menu is visible and Time.timeScale = 1f for when the menu is off. Setting Time.timeScale is a simple way to pause your game. It's not always the complete solution for pausing, but it works for most simple games.

Be aware that Time.timeScale is preserved after loading a new level. So, if you paused in one scene then call SceneManager.LoadScene(), you will find the newly loaded scene to still be paused. To avoid this, you should set the timescale back to 1f immediately following the call to LoadScene().

That covers some basic menu concepts, but you can make some much cooler menus with custom graphical elements, animated transitions, etc. You can even place menu elements in 3D space rather than overlays. Definitely explore these more advanced concepts when making menus for your team project.

Trigger-Based Collectables

Collectable Items are another important concept for designing compelling games. Unity's Trigger is a simple way to implement a collectable. The colliders that you are already familiar

with double as trigger zones. You will create a simple one that SomeDude_RootMotion can pick up.

First, make a GameObject that is a sphere. Place it just above the ground and away from anything moving. Give it a pink color. Change the SphereCollider component to set IsTrigger to true (checked). This change will make the collider not cause collisions with rigidbodies, but will generate OnTriggerXXX() callbacks.

Note that it is common to make the size of the collider much bigger than the graphical object. For trigger-based collectables, that means the player doesn't need to get as close to initiate the pickup. Also, trigger colliders don't need a graphical representation at all. Invisible trigger zones are useful for detecting that a player has fallen to their death, etc.

You will now implement a script called CollectableBall that will implement OnTriggerEnter(). In this script, delete Start() and Update() and add "void OnTriggerEnter(Collider c) {}". In the body of this method, call "EventManager.TriggerEvent<BombBounceEvent, Vector3>(c.transform.position)" and "Destroy(this.gameObject)". This will generate an event that results in a sound played and will also delete the ball GameObject. (Note that the use of "BombBounceEvent" is just a placeholder. Ideally, you would create a new event for collection and add a new sound to the AudioEventManager.)

Try out the collectable by playing the game. You should be able to control a character and bump into the ball creating a sound and making the ball disappear from the scene. This is close to being a collectable, but we never updated the character to know that the ball was collected. Also, any collider that enters the trigger will initiate deleting of the ball.

In order to support collection, you will make a script that can be attached to characters that the ball collectable will act upon. Create a new script, BallCollector. Add a public member variable Boolean hasBall (initially false). Also add "public void ReceiveBall() {hasBall =true; }". Now attach the script to SomeDude_RootMotion.

Next up, modify the CollectableBall script to only respond to triggering GameObjects that contain the BallCollector script. In OnTriggerEnter(), first check if c.attachedRigidbody is **not** null. If that is the case, attempt to grab a reference to a BallCollector component like so: BallCollector bc = c.attachedRigidbody.gameObject.GetComponent<BallCollector>();

Only if bc is **not** null should you execute the TriggerEvent() and Destroy() that you added previously.

This code simply makes sure that whatever caused the trigger has a rigidbody and the parent GameObject has a BallCollector attached. One other thing you need to do is call bc.ReceiveBall() if bc was **not** null.

In summary, you are checking if what entered the trigger zone was a BallCollector. If so, tell the BallCollector that the ball is collected and also play audio and destroy the collected ball.

Test everything out, confirming that only SomeDude_RootMotion can collect the ball and that the hasBall bool switches to true. Any character without a BallCollector component should **not** be able to collect the ball.

There's nothing to stop you from attaching triggers to moving objects, including rigidbodies. So you could make the collectable ball roll around, etc.

<u>Trigger-Based Animated Object</u>

Your last task is to create a trigger-based prefab game object that plays a Mecanim animation if the player gets close enough. You should select a game object concept that you think will be useful in completing your team project. (Please note that you can coordinate with your team on what concept is appropriate for each team member, but the work should be done individually.) Refer to "Trigger-Based Collectables" section above for key concepts that you will be leveraging. Mecanim should be used for the animation support. You should place the animatable geometry under a root empty game object. This will allow you to animate coordinates relative to where the prefab is placed.

Specific requirements:

- Object is prefabbed
- Object animates via Mecanim in a compelling way when player gets close enough
- Object resets when player is far enough away (if obj in triggered state)
- Animation onset and reset should occur smoothly regardless of when player moves away.

NOTE: You can use *documented* (in your readme) 3rd party graphical assets, but you must create the mecanim animation state machine yourself

Once developed, place at least three (3) clones of your prefab in the scene.

Checklist

- Make sure M2 pause script is not active in your scenes
- Working Game Start Menu with dedicated scene **(25 points)**
    - Overlay UI Panel (Start Menu)
        - Start Game button
        - Exit button
    - Camera background
        - Post-processing effect on the camera
- Working In-Game Menu in original gameplay scene **(25 points)**
    - Overlay UI Panel
        - Restart Game button
        - Exit button
    - Panel responds toggles on/off with Escape key
    - Game pauses when menu enabled
- Collectable ball that only SomeDude_RootMotion can collect **(20 points)**
- Trigger-based animated prefab object placed in three (3) locations in scene **(30 points)**
    - Object is prefabbed
    - Object animates via Mecanim in a compelling way when player gets close enough
    - Object resets when player is far enough away (if obj in triggered state)
    - Transitions are smooth
- Consider doing the extra credit below
- Auditor and build test!

    Build: Make sure when you create a build that both your main scene and your GameMenuScene are included! Test it out to be sure! Make sure that the game build starts from the start menu scene and that the grader can access both scenes!

Extra Credit (+5)

For extra credit, you can implement SomeDude_RootMotion throwing a ball after collecting one. He must be able to throw at any time, including walking/running. **If you do complete the extra credit, make sure your readme clearly indicates that fact!**

You will need to do many, many, many steps (and associated sub-steps). But the end result is pretty cool. Also, if you master this technique you can make skeletal animated mesh characters do all kinds of things with different weapons and items.

1.) Add a "Throwing" animation layer to SomeDude_RootMotion's SimpleAnimatorController. This will ultimately allow blending of a throwing animation on top of any other animations such as running/walking.
2.) In the same Animator, create a mecanim animator parameter, "Throw", for the throwing action (Boolean). This will allow you to map user inputs to the mecanim animation system.
3.) In the new animator layer, create a "Nothing" animation state that is the "Layer Default State". Also, leave it with an unset animation (e.g. "None"). When not throwing, the new layer will rest in this state so as not to influence the locomotion animations.
4.) Create a non-looping throwing animation state in the new layer (associate with the existing Throwing animation in the Assets).
5.) Make a transition from the Nothing animation state to Throwing state with condition of animator parameter Throw=true
6.) Make a transition back from the throwing state to the Nothing state. This should occur at exit time from the throwing animation. You can optionally trim the start and exit time down for a tighter animation sequence. You might want to revisit this step after you complete all the other steps.
7.) Configure this new layer with a humanoid avatar mask suitable for throwing. This should allow throwing while standing/walking/running/etc. SomeDude is left handed and the avatar graphic is facing you (so left arm is on the right). It's recommended to enable (set green) the left arm, left hand IK corrections, torso, and head. Everything else disabled (set red).
8.) Configure the new layer to have a Weight of 1.0. (Where the avatar mask allows, the throwing animation will have 100% influence on the skeleton.)
9.) Assign SomeDude_RootMotion to Layer "player". It's at the top of the Inspector view. (You will want to keep the character's capsuleCollider from interfering with the ball while it is held or thrown and will leverage physics layers to do this.)
10.)Create a new Layer (in Tags and Layers) called "projectile". This will be used for a throwable ball that has rigidbody+collider.
11.)Configure this projectile layer to not collide with the character that is holding it by disabling interaction between "projectile" and "player" (Physics Layer Collision Matrix)
12.)Next you need to create a sphere GameObject in your scene. This will ultimately become a prefab of your throwable Ball once you finish configuring it.
13.)Give it a GameObject name of "ThrowableBall"
14.)Size the sphere so it's about softball size relative SomeDude's hand. 0.1 uniform scale probably works
15.)Assign the ThrowableBall to the "projectile" layer
16.)Give the sphere a color other than gray (maybe pink to match the collectable)
17.)Give the sphere a rigidbody and mark it kinematic. This is because the ball will be attached to SomeDude's throwing hand and we want it to follow along with the hand until it is released with a throw. Since it's small, make collision detection Continuous to deal with tunneling potential.
18.)Now drag the ball GameObject you just made down into Project:Assets\Prefabs. That action just created a prefab

19.)Disable the original ball in the scene. (If the prefab needs tweaking, you can re-enable it, make adjustments, use the prefab options in the Inspector view to commit changes to the prefab in Assets, then disable the one in the scene again. Or you can edit the prefab directly with the new-ish prefab editor view)
20.) Now in the Hierarchy view of SomeDude_RootMotion, dig down the skeleton starting with MixamoRig:Hips all the way down to the left hand. Add an empty GameObject to the hierarchy here named BallHoldSpot
21.) Adjust BallHoldSpot's' position so that it is offset from the palm/fingers a little bit. This will be used for placing the ThrowableBall prefab in a reasonable location relative to the hand.
22.) Now you modify your BallCollector script to handle the throwing. This isn't the best software engineering, but it will keep things simple for this exercise
23.) To the script, add a "Transform handHold" member variable and implement Awake() callback. Assign the handHold transform to the BallHoldSpot using the this.transform.Find() function. Refer to BasicControlScript's code that grabs leftFoot and rightFoot to get an idea of the syntax. However, there are lots of ways to get a reference to a descendant GameObject, so feel free to use a different method if you prefer.
24.) Add a "public Rigidbody ballPrefab" member variable and error check that it is assigned a value in Awake(). It will be assigned manually in the Inspector view.
25.) Add a component reference to SomeDude's Animator. It will be the same as in BasicControlScript (e.g. use "RequireComponent" and "GetComponent<Animator>()")
26.) Add "Rigidbody currBall" member variable. This will be a reference to a held ball, until it is thrown/released by the character.
27.) Save and confirm the script compiles correctly
28.) Drag from Assets your ThrowableBall prefab to Inspector field ballPrefab of component BallCollector of SomeDude
29.) Go back to the BallCollector script and modify ReceiveBall() to Instantiate<>() the ballPrefab as a child of the handHold and set it to currBall. Also, set currBall's localPosition to Vector3.zero and set isKinematic to true. This will make SomeDude hold the ball when he picks up the collectable.
30.) Play the game and confirm that upon collecting the ball that a ThrowableBall appears in the left hand.
31.) In the script, implement a public ThrowBall() method.
32.) In ThrowBall(), set currBall's parent to null (releases the ball), set kinematic to false (makes ball under force control), and set its velocity and angularVelocity to zero (this velocity zeroing is necessary due to a Unity bug. Otherwise the ball is sometimes under several seconds worth of gravity's acceleration and will want to go straight down at high speed upon release)
33.) Also, apply a force to the ball using SomeDude's "this.transform.forward" multiplied by some appropriate scalar with ForceMode.VelocityChange. This is the actual "throw"
34.) Lastly, set currBall to null. This allows for another ball to be picked up later
35.) In the BallCollector script, implement Update()
36.) In Update(), conditionally set the animator's Boolean "throw" parameter to true if currBall is not null and Input.GetButtonDown("Fire1") is true. Else, set the same Boolean animator param to false. This ties user input to the throwing action, but only if SomeDude has a ball. (Note that you should disable the IK button pressing part of script from the previous assignment that uses same button.)
37.) If you run the game and pick up a ball, you should see the throw arm motion but the ball doesn't release. The last step is to configure the animation to execute an animation callback and release the ball.
38.) Go to the Inspector view of "Throw Import Settings" object in Assets (named "Throw" with blue cube icon and not the gray play icon with same name, which is the animation)

39.) In this Inspector view, go ahead and bake "Root Transform Rotation" into pose so that throwing won't cause rotations of the chase camera. Though, if your Avatar Mask is configured properly this shouldn't happen.
40.) Expand Events in the same Inspector view. Scrub through the animation preview at the bottom of the Inspector view and find a good release point for the ball. Maybe around frame 25 is good.
41.) Add a new animation event at your selected release point
42.) Set the animation event to use Object "BallCollector" and set Function to ThrowBall
43.) Click Apply on the Throw Import Settings (bottom of Inspector)
44.) Congratulations, you made it to the end! SomeDude should now be able to throw a ball! Maybe make your collectable into a prefab a replicate it several times in your level so you have some ammo to play with.

Submission:

You will submit a Zip/7Zip of your project via Canvas. If the file is too big for Canvas, then submit a link to a private cloud hosting (such as GT's Box license). **Please clean the project directory to remove unused assets, intermediate build files, etc., to minimize the file size and make it easier for the TA to understand. Refer to Assignment Packaging and Submission on the Canvas Syllabus for further details.**

The submissions should follow these guidelines:

a) Your name should appear on the HUD of your game when it is running.
b) Follow the *Assignment Packaging and Submission* steps including:
    i. ZIP file: <*lastName_firstInitial*>_m<*milestone number*>.*zip*
    ii. Complete Unity Project
    iii. Builds
    iv. Readme file should be in the top-level directory: <*lastName_firstInitial*>_m<*milestone number*>_*readme.txt* and should follow base requirements from *Assignment Packaging and Submission*
        i. What post processing effect are we looking for?
        ii. Where are your collectables?
        iii. Details of trigger-based animation (where to go, expected behavior, etc.)
    v. Size reduction

Submission total: (**up to 20 points deducted** by grader if submission doesn't meet submission format requirements)

Be sure to save a copy of the Unity project in the state that you submitted, in case we have any problems with grading (such as forgetting to submit a file we need). Do not alter or remove your submission from cloud hosting until your grade has been returned.

Resources:

Example GUI with background scene from Half-Life 2:
https://www.youtube.com/watch?v=pveer0jDCmk

http://blogs.unity3d.com/2014/05/28/overview-of-the-new-ui-system/ - overview of the "new" UI system

http://docs.unity3d.com/Manual/HOWTO-UIScreenTransition.html - how to integrate UI with animation controller to control screen states and transitions.

http://openfontlibrary.org/ - has a collection of fonts many of them on very permissive license agreements, please note where you got them from in your writeup.

Custom button makers:
https://dabuttonfactory.com
http://buttonoptimizer.com