Unity Certified Programmer Exam Prep

Participant E-book

Unity Certified Programmer Participant Guide

Preface: Your Learning Journey with Unity

How to use this ebook

This ebook is to be used as a follow-up guide to the workshop. It contains descriptions of sample solutions to the programming challenges you were asked to complete in the workshop, along with links to completed project files for each workshop topic. We've also provided examples of how we solved programming challenges in our project files, but our solutions are among many possible solutions — you may have completed the task differently, but just as effectively. We suggest using this ebook and the completed project files and associated programming code to help you review topics discussed in the course and prepare for the certification exam.

Mob Programming

In this workshop you are going to participate in "Mob" programming. What this means is that one member of your group will be entering code and making changes that the GROUP has agreed upon. Nothing will go into the Visual Studio or be edited in Unity that the group hasn't mandated. This gives you all the ability to solve the problems presented together

Further resources

Your resources from Unity are not static! If you want further support, want to ask fellow Unity creators for advice, or you have any questions about this workshop, please reach out on Unity Connect. Unity Education has a dedicated Unity Learners group where Master Trainers and other members of the Learn team are happy to help! Unity also has official documentation that is comprehensive and serves as an excellent reference point if you wish to explore additional aspects of a specific subject. Finally, the Unity Learn site is a huge repository of official Unity tutorials and excellent partnered content. Tutorials are constantly being added, so check back regularly to see what's new.

Unity Learners Connect Group
Unity Official Documentation
Unity Learn

Assets

The course asset files may be found on the dedicated Unity Learn page, here.

Chapter 1: Core Interaction Programming

Activity 1: Player Movement and Firing

Challenge Prompt

A team of artists has created a very basic model of the spaceship and turret, and you have added the ship and projectile Prefab to a new Unity project. You've set the ship and the camera in the appropriate starting location and are ready to implement movement and firing.

Tasks to Complete

The PlayerShip

- Set the screen resolution to be fixed at 1920x1080.
- Implement movement of the spaceship with WASD keyboard controls. Because this game will eventually also be built for mobile platforms, you should use the CrossPlatformInputManager class from Unity's Standard Assets — included in the starter project — to turn this input into an Input Axis.

Turret Movement and Firing

- Let the player shoot bullets by clicking the mouse 0 button (by default, the left mouse button). A bullet object should be instantiated from the ship and move toward the mouse location.
- Do not let bullets collide with the player's ship or with each other. Manage physics Layers to control collisions.
- To keep the Hierarchy window tidy, make all bullets the children of an otherwise empty GameObject named BulletAnchor.
- Bullets should destroy themselves after two seconds.

Tips

- *Think in terms of keeping your classes relatively small and making code that could work for multiple objects — such as the screen wrapping behavior — into a single component script that can be attached to the different objects.*
- *The UnityEngine Attributes (such as [RequireComponent()], [Header()], [ToolTip()], etc.) are extremely helpful. Find them at the Unity Scripting Reference by looking in the table of contents at the left under UnityEngine > Attributes.*

Solution

The PlayerShip

For this challenge, we made the PlayerShip move using the WASD keys. Keeping in mind that the game will be built out for mobile platforms, we opted to use the CrossPlatformInputManager to get the vertical and horizontal axes of the PlayerShip. We then created a new Vector3 based off the input axes, which we will use to set the velocity of the PlayerShip's Rigidbody. We then chose to normalize the Vector3 if its magnitude is greater than 1 to prevent the ship from flying at twice the speed when moving diagonally. For the PlayerShip's velocity, we simply multiplied the Vector3 by a variable called Speed.

Firing Bullets

To fire bullets from the turret, we returned to the PlayerShip script and created a new Fire() function. We set the function to run when the Fire button `Input.GetMouseButtonDown(0);` is pressed. The function receives the position of the mouse, instantiates a bullet, and points the bullet at the mouse position using the Bullet script.

The Bullet script controls the bullet itself. To make sure all bullets are children of a GameObject named BulletAnchor, we set the bullet object's parent to the BulletAnchor at Start().

To make sure the bullets self-destruct after two seconds, we invoked the DestroyMe() method with a lifetime variable set to 2. We set the velocity of the bullet using its Transform.forward and a bulletSpeed variable, since we set the direction when the bullet was instantiated.

Activity 2: Turret Pointing and Screen Wrapping

Challenge Prompt

Now that we have bullets firing, we need to be able to control the direction they fire by aiming our turret with the mouse pointer. We also need to implement screen wrapping so objects don't fly off the screen into the inky void of space.

Tasks to Complete

Turret Movement

- Let the player aim the ship's turret by moving the mouse so that the turret always rotates to the mouse location.

Screen Wrap

- Both bullets and the PlayerShip should screen wrap — that is, when they exit one side of the screen, they should reappear at the opposite side. Handle this with a single Script component that works for both the bullet and the PlayerShip.
- Make sure the screen wrap happens only when the GameObject has completely exited the screen. This is a lot more challenging than wrapping when the pivot of the GameObject has left the screen.

Tips

- *Remember that the screen resolution should be fixed to 1920x1080.*
- *Think in terms of keeping your classes relatively small and making code that could work for multiple objects — such as the screen wrapping behavior — into a single component script that can be attached to the different objects.*
- *The UnityEngine Attributes (such as [RequireComponent()], [Header()], [ToolTip()], etc.) are extremely helpful. Find them at the Unity Scripting Reference by looking in the table of contents at the left under UnityEngine > Attributes.*
- *For screen wrapping, consider using a trigger Collider to represent the on-screen area and wrapping when a GameObject exits that trigger.*

Solution

Turret Aiming

We created a new script called 'TurretRotator' and attached it to the TurretRotator gameObject which is a child of the PlayerShip Prefab. We declared a new Vector3 called CursorPosition that stores the position of the mouse. To convert it to worldspace we use `Camera.ScreenToWorldPoint(CursorPosition);` Once this is set we simply use `Transform.Lookat()` and pass in the world space mouse position. We made sure to offset the Z value of what we pass by the distance between the camera and the PlayerShip (in this case about 10).

Screen Wrap

To make both the PlayerShip and bullets wrap around the screen, we made a GameObject called ScreenBounds that's a child of the Main Camera and contains a Box Collider and the ScreenBounds scripts. We made sure the resolution was fixed at 1920x1080 and the Camera was set to Orthographic. On Start(), we set the Box Collider component on the object to the Box Collider variable, and we made sure the size of the Collider was set to Vector3.One; otherwise, later calculations might be wrong. After that, we set the size of the Box Collider to be the size of the playable area for our game, and we created a method to determine if the PlayerShip is out of bounds using the Collider we scaled to the screen size.

For the asteroids and bullets, we added a new script for the screen wrap. To determine if an object has left the boundary of the ScreenBounds trigger, we call the OnTriggerExit() method. When the PlayerShip or a bullet leaves the bounds, that object's Transform location is set to the inverse location on the other side of the screen.

Activity 3: Spawning and Destroying Asteroids

Challenge Prompt

With the PlayerShip and projectiles implemented, it's now time to work on asteroids and their behaviors so the player has something to shoot at. Each asteroid is constructed using a different combination and position of the three asteroid models provided by the artists. The art department needs to be able to replace asteroid art when necessary. Consult the Requirements Document to learn about asteroid size, speed, rotation, and how child asteroids are spawned.

Tasks to Complete

Spawning Asteroids

- Store the base asteroid models in a ScriptableObject so the art department can replace them when necessary.
- Make each asteroid spawn as a cluster: a parent asteroid should spawn two children, and each of those children should spawn two children (progressing from an initial size of 3 to a size of 1).
- Make the size of an asteroid affect its initial speed and rotation velocity (per the Requirements Document).

Collisions with Bullets

- When a bullet collides with an asteroid, destroy both the asteroid and the bullet.
- Make the asteroids collide with the PlayerShip and bullets, but not with each other.
- When a parent asteroid is destroyed, promote each of the children to top-level asteroids and give them their own velocity. (OffScreenWrapper will also need to be modified to keep these now-top-level asteroids on screen if they were off screen when promoted — see below for more info.)
- Make child asteroids smaller in size and give them a higher maximum velocity.
- Make asteroids screen wrap like everything else in the game.

Bonus Challenge

If an asteroid's child is outside the ScreenBounds when their parent is destroyed, it can cause all sorts of issues depending on your implementation of OffScreenWrapper. Fix all of the issues caused when an asteroid's parent is destroyed while the child is off-screen. Because the child asteroid is outside of the ScreenBounds trigger, its OffScreenWrapper script will never receive an OnTriggerExit() call, and it will continue drifting away from the screen.

Solution

Remember your code might be different than ours.

Spawning Asteroids

We started this challenge by creating our Asteroid ScriptableObject, which controls variables such as minimum and maximum velocity, initial size, the number of asteroids at each level, and the different asteroid Prefabs. Be sure the Asteroid ScriptableObject has a Rigidbody and an OffScreenWrapper attached.

To instantiate our asteroids and keep track of the asteroids we've spawned, we attached an AsteraX script to the Main Camera, which maintains the reference to the ScriptableObject. On Awake(), this script will set itself to be the Singleton, then on Start(), it will clear out the list of asteroids. After that, it will spawn three parent asteroids. Upon instantiation, each parent asteroid checks to see if it's too close to the PlayerShip; if so, the asteroid finds a new place to spawn.

To split the asteroid into smaller asteroids upon collision, we ensured that all asteroids have a Collider. The Asteroid script determines whether the asteroid is a parent or a child. If it's the parent, it will know to split into multiple children upon collision. Each child will then become a parent, enabling its OffScreenWrapper and destroying the parent asteroid as well as the bullet. A counter keeps track of the child's level to make sure the child instantiates smaller than its parent. That way we control the size and speed.

Collisions with Bullets

In this part of the challenge, we need to be sure the bullets only collide with the asteroids, and that asteroids do not collide with each other. For this, we looked at the physics settings. Under the Layer Collision Matrix, we set the boxes to either accept or ignore the collision settings. Be aware, AsteraX doesn't require many physics checks.

To make sure the bullet doesn't destroy a child asteroid still attached to a parent, we made the bullet send information about the hit through the child to the parent. Lastly, within the Asteroid script, we used Tags to determine whether the asteroid has hit a bullet, another asteroid, or the PlayerShip. We set a different functionality for each interaction; for example, the PlayerShip can spawn an explosion particle when an asteroid hits it, whereas the bullet would destroy the asteroid as well as itself.

Bonus Challenge

Our Bonus Challenge was to fix the issue with the OffScreenWrapper, where if a parent asteroid was destroyed while partially screen-wrapped, one of the child asteroids would go flying out of bounds without wrapping around the screen. To fix this issue, we simply made a child GameObject of ScreenBounds called ExtraBounds. ExtraBounds is scaled 1.5 on all axes so that it's larger than ScreenBounds on all sides. In the OffScreenWrapper script, OnTriggerExit() checks to see if the asteroid is leaving the playable area. If not, then it checks to see if it's leaving the ExtraBounds, and if so, it returns the child to the playable area.

Activity 4: Implementing Points, Jumps, and UI

Challenge Prompt

The next step is to keep score, implement a way for the player to lose a life when hit by an asteroid, and keep track of the number of lives the player has left. You'll show this information to the player in a simple HUD (Heads-Up Display) User Interface. Refer to the Requirements Document for how the HUD should work once implemented.

Use professional scripting practices, such as descriptive method names and comments, to make sure your code is easy to read by other programmers.

Tasks to Complete

PlayerShip Collisions and Jumps

- When the PlayerShip collides with an asteroid, make the PlayerShip jump as it teleports away from danger. It will disappear when hit, and then respawn in a safe section of the screen after a short time. A jump decreases the number of remaining lives by one.
- Destroy the asteroid that hit the PlayerShip and promote each of the children to top-level asteroids (as in the previous challenge).

Heads-Up Display

- Implement a simple HUD that shows the current score and the number of jumps remaining. Be sure to implement the 9-slice boxes around each text field.

Points and Scoring

- Let the player earn points by shooting asteroids. Smaller asteroids should be worth more points.

Game Over

- If the PlayerShip is hit and no jumps are remaining, destroy the PlayerShip, end the game, and reload the Scene after a short delay.

Solution

PlayerShip Collisions and Jumps

To get our PlayerShip to jump after colliding with an asteroid, we first added variables for the startingJumps and respawnDelay, and then we initialized the fields LAST_COLLISION, COLLISION_DELAY, and JUMPS. We then made sure that OnAwake(), we set the JUMPS to the number of startingJumps; from here on, the number of jumps is controlled by JUMPS.

OnCollisionEnter() checks for collisions. When the PlayerShip collides with an object, OnCollisionEnter() finds out if it's an asteroid by checking the Asteroid component. If it's an asteroid, we make sure not to hit it more than once by making sure Time.time is less than LAST_COLLISION plus COLLISION_DELAY. If it is, then we return; otherwise we set LAST_COLLISION equal to Time.time. In doing this, we check the time we first hit the asteroid; if we hit it again within a specified delay length, it will no longer register the second hit. This prevents any unwanted bugs we might have otherwise run into.

To find out what happens to the PlayerShip next, we check to see if the JUMPS are greater than 0. If so, then we call the Respawn() method; otherwise we call the AsteraX.GameOver() method and deactivate the PlayerShip GameObject, causing it to disappear.

To respawn our ship, we created a Respawn() function that first starts the AsteraX.FindRespawnPointCoroutine coroutine to determine a good spot for our ship to teleport to. We then disabled the OffScreenWrapper and moved our ship offscreen and out of the Scene. That way, the object doesn't do anything weird and remains offscreen until we're ready to bring it back on screen again, at which point we re-enable the OffscreenWrapper. When we call our coroutine, we pass it in the Transform.position of our ship as well as a RespawnCallback(), which lets us know that the coroutine has finished. Within the callback is where we set the new Transform.position as well as re-enable the OffscreenWrapper.

Back in the AsteraX script, we put the FindRespawnPointCoroutine that we may reuse for other purposes later. Within the FindRespawnPointCoroutine, we set up the variable RESPAWN_DIVISIONS to split the screen into eight segments horizontally and vertically, numbered sequentially, 0, 1, 2, 3, 4, 5, 6, 7. Then using a RESPAWN_AVOID_EDGES variable, we told it to not use row 0, 1, 6, and 7, leaving us with a nice 4x4 grid to spawn in. The coroutine starts by instantiating the particle for the Jump (we haven't created it yet). We then set up the grid of points using a nested For loop to get the horizontal and vertical position, and then store the distance from the ship to the closest Asteroid. We call a yield that waits for 80 percent of the player's spawn time, after which we check the location of the closest asteroid and iterate over all the respawn points to determine which is the farthest from any asteroid and also not the same point where we started. Once we determine the best point, we instantiate the Jump particle to indicate where the ship is spawning. When we return to the method within the PlayerShip script, that respawns the ship.

Heads-Up Display

We created our GameOver panel using a UI panel containing three children: the Game Over text background image, the Game Over text, and a panel that has the final level and final score. For the final

level and score panel, we used the 9-slice image as per the challenge. On the GameOver panel, we placed a GameOverPanel script. We made a UI object for the remaining jumps as well as the score by making two image objects using our 9-slice image. On top of them, we created text objects for the jumps and score, accordingly. To display jumps, we added a JumpsGT script to the Jumps UI object, in which we updated the text object on the Jumps UI object to check if we have any jumps remaining on the PlayerShip.JUMPS. If so, then we set the text to that number, and if not, then it sets Jumps to be blank.

Points and Scoring

To get our point system working, we added an array of points to the Asteroid ScriptableObject. That way, we can get the number of points based on the size of the asteroid. We then set up an AddScore() method to the AsteraX script. To call this method, we used an OnTriggerEnter() to check when the Asteroid first collides with another object. If that object is a bullet, we then check the size of the asteroid and add the points from the points array, depending on the size. To add up the score, we called the AddScore() method, which checks whether we have a Score GameObject set up; if so, points are added to the SCORE int, which holds the game's definitive score. We set the text within the Score UI GameObject to be the SCORE int. By using SCORE.ToString("N0"), the score string is automatically formatted to use commas (or decimals, depending on your computer's settings) in numbers over 1000.

Game Over

With the GameOverPanel script, we called ActiveOnlyDuringSomeGameStates by first registering with the GAME_STATE_CHANGE_DELEGATE on AsteraX, then calling the DetermineActive() method. The ActiveOnlyDuringSomeGameStates script uses a System.Flags enum, which has individual bits to represent each state, to determine whether an object should be active within the Scene. By doing this, we can have all states either active or inactive at the same time. Within ActiveOnlyDuringSomeGameStates, we used the DetermineActive() function to determine whether or not the object should be active by checking the activeStates plus the AsteraX.GAME_STATE against the AsteraX.GAME_STATE.

Next, within the GameOverPanel script, we set up an enum for the eGameOverPanelState, which will help to better control the GameOver text's fadeIn and fadeOut and any text animation. We also added references to the text's Rect Transform and the image, and we set an initial fade time. All of these are initialized and assigned to their appropriate references upon Awake(). We used the DetermineActive() to set the state to eGameOverPanelState.fadeIn once the game is over, which starts the fading in of the Game Over panel. By setting up our fade over multiple states (fadeIn1, fadeIn2, fadeIn3, display) we were able to more smoothly control our fade and the animations that will accompany it later down the road.

Chapter 2: Application Systems Programming

Activity 1: Implementing Particles and Explosions

Challenge Prompt

Create particle effects for **AsteraX**. For the PlayerShip, create an exhaust trail and effects for the ship's jumps. Also create bullet trails and two different asteroid explosions.

To allow easy modification and replacement of the particle effects, each particle effect should be on its own GameObject and not a component of an existing GameObject. None of our solutions used a Trail Renderer component, though some did use the trails that are part of the Particle System component.

The Textures and Materials that you need to recreate the particle effects are in the Textures & Materials folder in the downloadable Unity starter project for this challenge.

Tasks to Complete

Ship Trail and Jump Particles

- Give the ship an exhaust trail using a Particle System. The trail should emit only when the ship is moving and never appear in front of the ship.
- When the ship collides with an asteroid and jumps away, add an effect for the collision and another effect when the ship reappears.

Bullet Trail

- Give bullets an exhaust trail without using a Trail Renderer.

Asteroid Explosions

- Create two different explosion effects that appear when asteroids are destroyed.
    - Display one of the two effects randomly each time an asteroid is destroyed.
    - Scale each effect to the size of the asteroid.
    - Use AsteroidRubble1_Mat and AsteroidRubble2_Mat for the two effects.
    - Create a separate GameObject that is instantiated by the asteroid when it's destroyed.
    - Add a list or array holding the two GameObject Prefabs to the AsteroidsScriptableObject.

Bonus Challenge

Whenever the PlayerShip or a bullet wraps around the screen, you will see a particle trail jump in a straight line across the screen. To fix this problem, find a way to disable the Emission module when the PlayerShip or a bullet wraps the screen.

Write one small script that can be attached to both the Ship Exhaust GameObject and the Bullet Trail GameObject and solve the bonus challenge for both of them.

Solution

Remember that your particles may look completely different than ours.

Ship Trail and Jump Particles

We created a GameObject and attached a Particle System and a new script to it. We gave this script a Transform variable that we set as a reference to the ship. Then using a Vector3 as an offset, we set the GameObject to always be slightly off from the Transform variable's position. This ensures our ship's particle trail always emits from a certain offset point and not in the middle of our ship.

To make sure the Particle System doesn't emit while the ship isn't moving, we used the Emissions tab under the Particle System and set Rate over Time to 0 and Rate over Distance to 50. To make the particles look as if they're flying out from the exhaust, we set the Inherit Velocity to a negative number, allowing the particles to fly off in the opposite direction of the ship, making it look as if it's pushing out a exhaust trail.

To implement the Jump particle, we gave the PlayerShip script a reference to our Jump particle Prefab. Then, within the AsteraX script, we initiated the Respawn coroutine and set it to call the Jump particle right before the ship disappears. After the ship has found a new location to spawn and has loaded roughly 80 percent, it will instantiate a new Jump particle in the new location. This gives the effect of warping in and out of space.

Bullet Trail

We created a Particle System for our bullet trail by giving our Bullet script a reference to the Bullet Trail Particle Prefab. In the Bullet script Start() function, we instantiate the Bullet Trail Particle Prefab, set the particle Prefab's parent to be the bullet, make sure its position is set back to 0. As for the particle itself, we created it in much the same way as the ship trail, using the Inherit Velocity and Emission Rate over Distance, except we also included Size over Lifetime. By setting a curve in Size over Lifetime, we were able to make the trail appear smaller over time, leaving us with a cleaner looking effect.

Asteroid Explosions

To ensure we used both asteroid explosion materials, we created two separate explosion Prefabs, applying the AsteroidRubble1_Mat to one and the AsteroidRubble2_Mat to the other. As the challenge states, we need to be sure that when the Asteroid spawns, it picks a random Prefab from our two options. Our particles are held within an array or list. To randomize the effect, we set up an array within the Asteroid ScriptableObject, and we placed our asteroid explosion Prefabs within that array. We set up a function to randomly select an asteroid explosion, and we called the function that will instantiate the chosen particle. Using the level variable on the Asteroid Script, we were able to set the scale of the explosion particle.

Bonus Challenge

We made a script for both the ship and the bullets that has a reference to the Emissions tab within the Particle System on the object. The script checks whether it's within the ScreenBounds; if not, then the emission is disabled until the object is back within the game boundaries.

Tips: *Each particle effect should be attached to its own GameObject. This allows easier editing and replacement of the Particle System without the danger of accidentally modifying something else on the GameObjects you already have.*

If you only make the Ship Exhaust particle effect a child of the PlayerShip, then the tilt of the PlayerShip will cause it to jump around when you change directions, and the exhaust won't emit directly behind the ship. Come up with a way to avoid this issue.

Activity 2: Creating Multiple Levels and Pause

Challenge Prompt

Create 10 levels of difficulty. As specified in the Requirements Document, a level's difficulty is defined by the number of initial asteroids and the number of children of each asteroid at each size. Create a title screen and interstitial screens between levels.

Implement a pause for the game, and make the game pause when levels transition and whenever the player hits the pause button (see the Requirements Document for pause button position and color). Any new images you need are in the project's Textures & Materials folder.

Tasks to Complete

Multiple Levels and Screens

- Create the 10 levels according to the Requirements Document. Configure each level to the specified number of initial asteroids and the number of children of each asteroid at each size.
- Create a title screen GUI called TitleScreenPanel that includes a start button.
- Create a level interstitial GUI called LevelAdvancePanel that will display between each level (see Requirements Document). To make it different from the Game Over panel, implement a LevelAdvancePanel script that is not a subclass of ActiveOnlyDuringSomeGameStates.
- Modify GameOverPanel to show the final score and final level that the player achieved.
- Modify GameOverPanel such that after 4 seconds, the game will return to the title screen.

Pause

- Implement a pause for the game that will happen when levels transition and whenever the player hits the pause button. See the Requirements Document for pause button position and color.

Bonus Challenge

Add animations that play on the interstitial screen, as shown in the challenge video.

Tips: *Callbacks and delegates will both be important for this challenge. Even when a GameObject has active=false or a MonoBehaviour has enabled=false, scripts can still receive callbacks. Think about the different steps (e.g., world events) that you need to track to cover all of these Achievements.*

Solution

Multiple Levels and Screens

To create multiple levels, we set up a ParseLevelProgression() method within the AsteraX script. This method pulls in the level progression string and parses it to generate the different levels in our game. By making many calls to the Split() method within Strings, we are able to take the level progression string and split it into an array of strings. We first split it on the comma, which splits it into individual levels, then on the colon, which splits out the level number from the Asteroid amount. Then we split it on the slash, which gives us the numbers of asteroids and children.

After processing the level progression string, we send that data to the LevelInfo struct in the AsteraX script. We also pass the data to the LevelInfo constructor to set up information such as level name, number, number of initial asteroids, and number of children asteroids.

We set up our title screen within the GUI in a Screen Space - Overlay canvas. Then we set up a Start Game button. Using the button's OnClick() event, we call the TitleScreenPanel script that holds the StartGame() method, which in turn calls the AsteraX.StartGame() method. Within the AsteraX StartGame script, the StartGame() method sets the GAME_LEVEL to 0, then calls the EndLevel() method, which is located on the Singleton instance of AsteraX. By calling EndLevel() immediately after setting the level to 0, the game will then automatically increment itself to GAME_LEVEL 1, allowing it to go through the same process as if we were to end a regular level without needing a special case at the beginning of the game. We then make sure GAME_STATE is not equal to none, which ensures that we don't start a new level right as the game is stopping. We used the EndLevel() method to first pause the game and then increment up the GAME_LEVEL, which then sets the GAME_STATE to the postLevel state. We passed the AdvanceLevel() method within the LevelAdvancePanel script, two separate callbacks that are used as a displayCallback and an idleCallback on the LevelAdvancePanel Singleton, which then sets the state of the eLevelAdvanceState to fadeIn to bring up the interstitial screens.

After the fadeIn and displaySetup states, the level advances. This way, the levelAdvancePanel can fade in smoothly, and while the player is looking at a static screen, we can do the processing and take the performance hit without the player ever knowing.

At this point, we called the StartLevel() method, where we passed in our level number. We first made sure the LEVEL_LIST was set and that there were levels within the list itself. It changes the GAME_STATE to preLevel and the GAME_LEVEL to (level — 1) to ensure we get the first element in the list. We then made sure to clear the game of all asteroids and bullets before setting up and spawning the next wave of asteroids. In the AsteraX script, we set up a list of all asteroids and bullets to clear out the levels after each transition.

Pause

To add the pause functionality, we first created a PlayPauseToggle object under the GUI in the Screen Space - Overlay object in the Hierarchy, making sure that PlayPauseToggle has a UI.Toggle component

attached to it. The toggle's OnValueChange setting is set to toggle the PauseGameToggle() method on the Main Camera's AsteraX script. Within the AsteraX script, the PauseGameToggle() method calls the PauseGame() method, which returns the opposite of the current timeScale. While Unity's timeScale is set to 0, most calls are paused, except Unity will still send updates and late updates.

We used a delegate, the DetermineActive() method, from our previously created ActiveOnlyDuringSomeGameStates. By using a new enum ePauseEffect, we set different states for the object: ignorePause, activeWhenPaused, and activeWhenNotPaused. This allowed our DetermineActive script to set the Active state of objects according to whether the game is paused. Remember, every time you add a method to a delegate, you also need to remove that method from the delegate later.

Bonus Challenge

We set up Animations for the LevelAdvancePanel that transition between levels in the game. We implemented it into our eLevelAdvanceState, the enum that controls the transition between levels and the fadeIn and fadeOut of the LevelAdvancePanel. Within the fadeIn state, we set the level name and number on our LevelAdvancePanel. By having multiple fadeIn and fadeOut states, we can more accurately control what happens during the fade duration. During the fadeIn2, we pass the y local scale to the LevelTextYScaleEffect() method, which takes a 0 to 1 value, multiplies it by Pi, and returns the sine of that. The sine of Pi creates a curve that goes up and down from 0 to Pi, which will be our 0 to 1. By adding our U element, which is a 0 to 1 diagonal, we get a nice little bounce effect.

Activity 3: Achievements

Challenge Prompt

Implement an Achievements system to reward players for reaching different milestones in the game. According to the Requirements Document, you need to implement several Achievements that will eventually be used to unlock various ship and turret parts so the player can customize the look of the ship. You should expect that the designers will add additional Achievements in the future, so you need to make a flexible system that will allow more Achievements with as little additional code as possible.

Achievements

- Refer to the Requirements Document for the Achievement descriptions and rewards.
- Implement a drop-down Achievements notification, AchievementPopUp, to notify players when they've earned an achievement. The Achievements notification should include both the name of the Achievement and its description.
- If a new Achievement is earned while an existing notification is being shown, the new notification should be shown after the existing notification disappears.

Solution

Achievements

For the Achievements, we created an AchievementManager script, and within that script, created a class called Achievement. Achievement keeps track of different variables, such as levelUp, bulletFired, hitAsteroid, luckyShot, and scoreAttained. Each Achievement is set up with its own name, description, stepType, stepCount, an image of the unlockable part attained for completing the achievement, and a private bool _complete that has a public getter but an internal setter so that only classes within this script can set it. To check if the Achievement is complete, we set up a CheckCompletion() method within the Achievement class that allows us to pass in a number and a stepType, which is used to check if the Achievement is complete; if so, then it returns true and returns _complete as true, otherwise it returns false.

To keep track of how many bullets are fired and how many asteroids are hit, we set up another class within the AchievementManager script called StepRecord. This class takes a stepType, if that stepType is cumulative or not, and a number that's the total steps of that stepType. We then have a Progress() method that checks to see if the stepType is cumulative; if so, it adds the passed-in number to the step number; otherwise, it makes the passed-in number the new step number.

Within the AchievementManager class, we set up the AchievementStep() method, which takes a stepType as well as a number. This is where we tell the Achievement script that we have made some progress or taken a step. We call this method every time we meet one of the step requirements, i.e., every time we shoot a bullet or switch levels.

To indicate whether we have received an Achievement, we made a pop-up UI within our GUI in the Screen Space - Overlay canvas. The pop-up has two children: one for the title of the Achievement and one for the description of the Achievement. Within the AchievementPopup script, we called a PopUp() method, which takes in an achievementName and achievementDescription and sets the text within our PopUp UI display. PopUp gets called from the AchievementManager by the TriggerPopUp() method, which is initially called from the AchievementStep after determining if it's newly completed or not.

Activity 4: Saving Information Locally

Challenge Prompt

Implement secure storage of application and user data by creating a SaveGameManager script that will manage saving, loading, and deleting save-game data. Use the saved data to improve the user experience by implementing a high score feature.

Tasks to Complete

Save and Load Game

- Create a SaveGameManager script that will manage saving, loading, and deleting save-game data.
- Try implementing the SaveGameManager as a static C# class. Note that a static class can still have a static constructor to initialize static variables.
- The data that should be saved includes StepRecords, Achievement completion, and High Score.
- Save the file to: Application.persistentDataPath + "/AsteraX.save"

Delete Save

- Add a Delete Save button to the Title Screen. This button should only appear (i.e., be active) if Application.isEditor == true.
- If the player pressed the Delete Save button, the save-game file should be deleted, all StepRecord_nums should reset to 0, and all Achievements should be reset.

High Score

- When the player achieves a high score, announce "High Score! You've achieved a new high score." using the AchievementPopUp. You may have to add a static method to AchievementPopUp to enable this.
- If the player achieves a high score during the game, instead of displaying "Game Over" when the game ends, display "High Score!"

Resources

For when implementing the SaveGameManager as a static C# class:
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

Solution

Save and Load Game

We began this challenge by making a SaveGameManager class, within which we set up a reference to the saveFile script and a string that will become our filePath. We then set up a LOCK bool that will prevent us from saving the file if it's locked. This lets us do things while loading a file that might normally cause a save to happen, which preserves the integrity of the save file. We set our filePath, which becomes our Application.persistentDataPath per the challenge requirement. We chose to go with the persistentDataPath over PlayerPrefs because it's a safer option to guard against hacking. We then set a constructor for our SaveGameManager. This constructor sets our LOCK to false, sets the filePath, and then calls a new instance of the SaveFile class we created within the SaveGameManager. The SaveFile class is where we store the StepRecord and Achievement arrays and the highScore. To actually save the game, we called the static method Save() within our SaveGameManager script. This method grabs our StepRecords and Achievements from the AchievementManager script and passes them to our saveFile instance, which then parses that saveFile instance using JSONUtility.ToJSON(), saving our file as a JSON text string. We wrote that string to the filePath using the File.WriteAllText() method. Note: Be sure you're using the System.IO namespace within your script in order to access the File class and write to files. Be aware that we set up our game to save only after receiving a new Achievement or when the game ends.

To load the game, we set up a static Load() method within our SaveGameManager script. This method checks to ensure there is a saved game within our filePath, and if that file exists, we get a reference to that JSON text file as a string. Using that string, we converted the JSON text file into an instance of our saveFile() class, which then tries to assign to the saveFile instance within the SaveGameManager. By setting a catch in case anything fails, we can easily debug if the saveFile is in some way corrupted. After the catch, if everything succeeded, we set LOCK to true to ensure that we don't accidentally save over any of our load game data. We called LoadDataFromSaveFile() within the AchievementManager. Using the passed through saveFile, we first loop through all StepRecords, figure out the type of Step, and then search through STEP_REC_DICT. This finds the proper setRecord and then sets its num to the number within the saveFile. This method runs a For loop through all of our Achievements within the saveFile, checking their complete status, then uses that status to set whether or not we have completed the Achievement. To be sure the game will load after each start, we placed the original call to the SaveGameManager.Load() method within the Start() method of the AsteraX script.

Delete Save

We made our delete save button and placed an instance of the ActiveOnlyDuringSomeGameStates script on it, within which we set up an activeInEditor bool to allow objects to display only in the Editor and not during play. Remember the delete save button is for debugging purposes.

For the save button, we check to make sure a saveFile exists. If it does, we then use the File.Delete() to delete the filePath and then set the saveFile to a new instance of our SaveFile class. The ClearStepsAndAchievments() method we created in the AchievementManager loops through all

StepRecords, setting each step's num to 0 and looping through the completed bool of each Achievement to set them all to false. We've now deleted our save.

High Score

To add the High Score feature, we set up an AddScore() method within the AsteraX script, which adds points to the score depending on the size of the asteroid destroyed. After that, it checks against the SaveGameManager to see if the current score is higher than the high score; if so, it will call the High Score Achievement pop-up via our AchievementPopUp script. Once the level or game is over, it will set the player's new high score.

Activity 5: Player Ship Customization and UI

Challenge Prompt

Create a Customization Panel that will allow the player to select any of the four Body and four Turret Ship Parts. Create a script to replace the ship parts on the PlayerShip with the parts selected by the player. Refer to the Requirements Document for information about the Customization Panel.

Tasks to Complete

User Interface

- Allow the player to select any of the four Body and four Turret ShipParts from a graphical interface. The selected parts replace the parts on the PlayerShip.

Ship Customization

- Make the Achievements system unlock the various ShipParts as outlined in the Requirements Document.
- Create a ShipCustomization script and attach it to PlayerShip.
    - This script finds the ShipPart components attached to the children of PlayerShip that can be customized/replaced.
    - This script replaces the ShipParts originally on the PlayerShip with any of those from the correct ShipPartsScriptableObject.
- Whenever SaveGameManager.DeleteSave() is called, relock ship parts.
- Make the SaveGameManager save every time a Ship Part is selected.

***Tips**: It's very important to LOCK the SaveGameManager when loading selected ShipParts, because if you have implemented it the same way we did, selecting a ShipPart will trigger a SaveGameManager.Save() in the middle of the load if SaveGameManager is not locked.*

To make the ShipCustomization UI look nice and work well, we added two more cameras to the scene:

1. UI Camera: A camera that has a Screen Space – Camera Canvas as a child, which then contains the ShipCustomizationPanel.

2. CloseUpCamera: This is the camera that shows the close-up of the PlayerShip and the space background for the ShipCustomizationPanel. It's actually a child of the PlayerShip in the Hierarchy, and it contains its own SpaceBackground, which is on the CloseUp. This camera is set to show only that SpaceBackground and the PlayerShip.

Solution

User Interface

We created a new empty GameObject and named it “GUI in Camera Space.” As a child of that GameObject, we attached our UICamera, which will display our UI elements. Inside that camera is a Canvas with its Render Mode set to Screen Space Camera. Within that Canvas, we set up our ship customization panel with four toggles for the different ship bodies and four toggles for the different ship turrets. We also set up an additional camera on the PlayerShip that’s displayed on the ShipCustomizationPanel to give a close-up view of the part we’re changing. By setting the camera’s depth to two, we can ensure our close-up camera will always display on top of our ShipCustomizationPanel. Next we set up the functionality behind the toggles.

Ship Customization

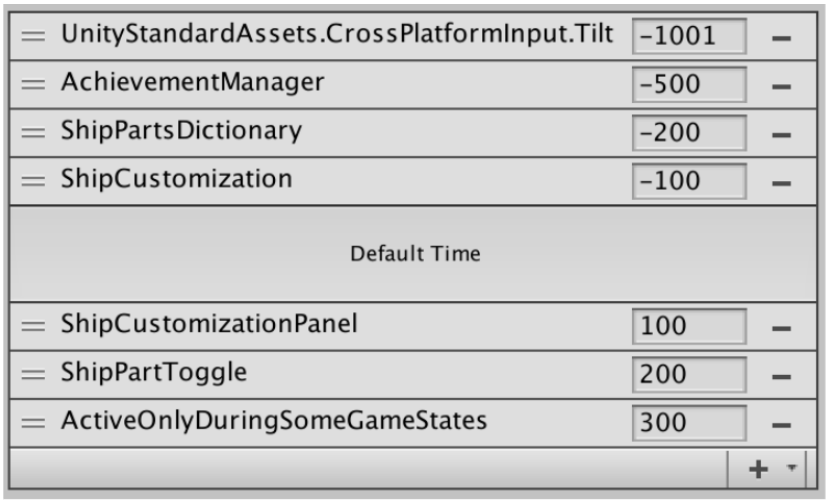
We created a ShipPartToggle script that specifies the ShipPart type and its part number. This ShipPartToggle script is attached to each of the UI toggles within our Ship Part Customization Panel. When the toggle value is changed, ShipPartToggle calls the SetPart() function from within the ShipCustomizationPanel script, swapping out the part. The ShipPartToggle also uses the background image we previously set up in the ship customization UI panel, which allows us to set each part’s background color to either white or red, depending on whether the ShipPart has been unlocked.

Within our ShipCustomizationPanel script, we used the ShipPartToggle script we placed on each toggle to fill our TOGGLE_DICT dictionary. The TOGGLE_DICT checks whether each ShipPartToggle is a body or a turret, and then places it into its respective array. This ensures all parts line up to their appropriate places within the Ship Part Customization Panel. We used the ShipPart number we declared within the ShipPartToggle. By matching the number to the array slot, we made sure the part fills in the correct toggle spot. Within the ShipCustomizationPanel script, we set up the LockPart() and UnlockPart() methods, which allows us to lock and unlock the ShipParts so we can properly save the game while switching out parts later.

To check which body part or turret part is currently selected, we set up a `GetSelectedPart()` function that returns the number of the selected part so that the SaveGameManager knows which parts we’ve selected. Similarly, we created the `SelectPart()` function that takes in a part number and returns a part when the game is loaded.

To save the parts within the `SaveGameManager` script, we added two new integers to the SaveFile class, one for the body part number and one for the turret part number. This will allow us to define which two parts we need to save onto our PlayerShip when we call `SaveGameManager.Save()`.

We found that script execution order really mattered. Here is an example of how we ordered them:



Activity 6: Unity Analytics and Remote Settings

Challenge Prompt

Create the following Standard Event calls to Unity Analytics:

Event Name	Event Data
LevelStart	Level number — Date & Time as "time"
AchievementUnlocked	Achievement name (ID) — Date & Time as "time"
GameOver	Date & Time as "time" — Final score as "score" — Final level as "level" — Was this a high score for this player (true or false) as "gotHighScore"

Create the following Custom Event call to Unity Analytics:

Event Name	Event Data
ShipPartChoice	Date & Time as "time" — Dictionary entry for ship parts (see below)

For each type of ship part in ShipPart.eShipPartType, include a dictionary entry where the key is the name of the part type, and the value is the number of that type that's selected in the ShipCustomizationPanel at the end of the game.

Add RemoteSettings to the game. Create a levelProgression Remote Settings key on the online Analytics dashboard and use it to update the levelProgression field on AsteraX (thereby changing the difficulty of the levels).

Solution

Analytics Standard Event

We created multiple Standard Analytic Events to keep track of when a player has unlocked an Achievement, received a GameOver, or started a level. We also set up a Custom Analytic Event in order to keep track of which ship parts the player has chosen.

We first ensured that Unity Analytics was set up within our game by clicking on the cloud icon on the top right of the Unity Editor window. We also made sure our project ID and organization were set up.

After that, we clicked on the Analytics tab and turned the slider to on. Then we started our script to call our events.

To call our Standard Analytic Events, we made a new static script called CustomAnalytics, which will house all our Analytic events. We were sure to use the namespace UnityEngine.Analytics as well as the System namespace. (Keep in mind the Standard Analytic Events are events predefined within the Analytics namespace.) We then set up our three different methods AnalyticsEvent.AchievementUnlocked(), AnalyticsEvent.SendLevelStart(), and AnalyticsEvent.GameOver(). Each of these events will be passed an ID, name of the Achievement or level number, and a dictionary that has a string key and an object value. That way, we can set our string to “time” and our object to DateTime.Now, allowing us to get the real time of the event call.

We handled the GameOver event a little differently. Instead of passing it a name or ID, we passed it null since the value is optional for this event. In addition to having the time in our dictionary, we added score, level, and gotHighScore bool, which gives us a record of those stats within our analytics. These functions are called within their respective scripts, for example AchievementUnlocked() is called within the AchievementManager script whenever an Achievement is earned; likewise SendLevelStart() is called whenever we start a new level, and GameOver() when we get a Game Over.

Analytics Custom Event

To make the Custom Event, we made a new function within the CustomAnalytics script that creates a new dictionary containing the time and time now. We then went through each type of ShipPart with a foreach loop of the ShipPartTypes in the array of types that we got from Enum.GetValues. By doing it this way, we are able to accommodate for future ShipParts that might be added to our game. We then set up a break statement that will limit the size of the dictionary to 10; this is necessary because AnalyticsEvent.Custom has a limit of 10 parameters. We then set an integer variable, which is what the ShipCustomizationPanel tells us is the selected part of that type. We added that to the dictionary and continued the foreach loop.

Once the dictionary was completed, we passed it off to Analytics using the AnalyticsEvent.Custom with the name of the custom type event ShipPartChoice and the dictionary we just made. We called this Custom Analytics Event function from within our GameOver() method within AsteraX, which ensures that the Analytics event runs only when the player gets a Game Over.

Remote Settings

We opened up the Unity Dashboard website, opening the Remote Settings tab under Analytics. We then set up a key value pair called levelProgression, containing the values from our normal levelProgression from the AsteraX script. We modified the levelProgression value on the Dashboard and changed the number of asteroids or child asteroids as desired. To make our Remote Settings work, on Start() in the AsteraX script, we called RemoteSettings.Updated and added to it our newly created RemoteSettingsUpdated() function. By using the RemoteSettings.GetString() function within the RemoteSettingsUpdated() function, we can switch the levelProgression with the levelProgression Remote Setting. We now have the freedom to change our game's levelProgression through the Unity Dashboard without having to rebuild the game every time we want to change it. Remember, after adjusting Remote Settings on the Dashboard, we need to hit the Sync button and enter a clear description of the changes made.

Chapter 3: 3D Interactions, Cameras, and Navigation

Activity 1: Enemy Navigation

Challenge Prompt

In this challenge you will build out **Stealth**, a third-person 3D game with stealth mechanics. You will implement the navigation of the enemy, a robot sentry, and set up the patrol system for the enemy robot (the ACS-17 robot by Vladimir Tim, available in the Unity Asset Store but included here for free in the project starter package). This will involve setting up a NavMesh in Unity and implementing the robot as a NavMeshAgent. There are some additional subtleties of movement that will require you to handle rotation at Waypoints. You'll also be adding animation for the Enemy ACS-17 robot.

In the starter project, you will find the Navigation Test Scene, which contains everything you'll need to get started. This includes three Waypoints that have already been positioned and rotated.

Tasks to Complete

NavMesh

- Bake a NavMesh from the hallways.
- Add a NavMeshAgent to the enemy.

Waypoints

- Write a Waypoint script that makes it easy for other scripts to access the position and orientation of the Waypoint and includes a public field for the number of seconds that the enemy should pause at that Waypoint before moving on.
- The EnemyBot should move sequentially between Waypoints in an order defined by a list in the Inspector.
- Implement the following movement sequence for the EnemyBot:
    - The EnemyBot should rotate to face the next Waypoint (without changing position).
    - The EnemyBot should walk to the next Waypoint.
    - Upon reaching the next Waypoint, the EnemyBot should rotate to match the orientation of the Waypoint (without changing position).
    - The EnemyBot should wait at the Waypoint for the specified duration.
    - Repeat the sequence.

You will be using the ACS-17 robot designed by Vladimir Tim.

Solution

Keep in mind your solution to the challenge is likely to be different than ours.

NavMesh

We first made sure our Hallway was set to Navigation Static, meaning that our Mesh would not be moving around relative to the navigation. Navigation Static is required to bake our Hallway into a NavMesh. We then set up the NavAgent component on our EnemyBot. Within the NavAgent, we adjusted the Obstacle Avoidance Radius to 0.75 to fit that of our EnemyBot; otherwise, it would be too small and our robot could end up clipping objects or walls. We then baked our NavMesh using the Bake button at the top of the Navigation window. Be sure to bake with the Height Mesh to ensure the NavMeshAgents stay on the ground and do not float above it.

Waypoints and Enemy Navigation

With our NavMesh baked, we moved into our Waypoint script, starting with creating Waypoint objects within our scene. These objects have a location and facing as well as a wait time that we defined within the objects' Waypoint script. Within the Waypoint script, we first set an array that gets all renderers within the object and all their children renderers, and we set them to false, ultimately making Waypoint invisible during play. We then set up multiple property getters to get the deeper properties of the Waypoint, such as position, forward direction, the look at position, and rotation. By doing this, we gained easier access to these properties, which we can take advantage of in our EnemyNav script, which is what controls our EnemyBot's movement.

We set up an enum to define the EnemyBot mode: whether it be idle, wait, preMoveRot, move, or postMoveRot. This allows us to control each part of the movement. For this script, we split our variables into two sets of fields, Inscribed and Dynamic. The Inscribed variables, such as the list of Waypoints, a float for speed, and a float for angularSpeed, are the settings that do not change for our EnemyBot. Our Dynamic variables are the variables that change throughout play, such as the robot's mode, the Waypoint number, the pathTime, and how long to wait at each point. By breaking them up in the Inspector using the Header attribute, we can more clearly see what's going on within our EnemyBot.

On Start(), we give our EnemyNav script a reference to our NavMesh, initialize the Waypoint where we're starting, and call the MoveToWayPoint() function. We also set a stoppingDistance variable of 0.01f to ensure we stop when we're close enough to the Waypoint; generally, stopping on zero is hard to do, so this will stop us when we're just about there. Within our MoveToWaypoint script, we set the Waypoint by its number in the list then check to see if we are stopped. If so, then we switch our eMode to our preMoveRot. Within this preMoveRot, we can determine which Waypoint is next on our list, then rotate our EnemyBot in that direction before setting the eMode to eMode.Move. We also set up a function that allows us to manually control the EnemyBot's rotation rather than relying on the NavAgent's rotate settings. Be sure you've followed the Waypoint sequence specified in the challenge so your EnemyBot moves in the correct order. After our EnemyBot moves to the next Waypoint, it performs the postMoveRot, again checks where the next Waypoint is, and rotates accordingly. This method gives it a more robotic feel and alleviates any errors we might have with the EnemyBot rotating while walking.

Lastly we entered the wait eMode, where the EnemyBot waits for a certain amount of time before continuing its path. FixedUpdate() is where we call our switch statement and initialize the movement speed and angular speed. We chose FixedUpdate() because it tends to be more reliable for movement with its 50 calls per second. By checking whether our nav speed and speed variable match Mathf.Approximately, we check our current speed but only if it differs from the correct initial speed. We also set up a few helper methods to determine the next Waypoint on the list and the Waypoint we're at.

Activity 2: Character Animation

Challenge Prompt

In this challenge you will add animations to the player and the enemy. The player already has animations that play when she runs around the scene. You will add animations that have her sneak around when hugging a while using the Animator. You will add animations to the enemy using the Animator.CrossFade() method.

Tasks to Complete

Player Animation

The Player already has an animation state machine in the Animator that was pulled from Unity Standard Assets. To add the animations, the Animator needs to be updated. Code is already in place to implement the animation Parameters correctly. Modify it in the following ways::

- Disable the transitions to Airborne and Crouching.
- Add an InCover state that is entered when the parameter inCover == true.
- Make sure the state returns to Grounded when InCover==false.
- Inside of the InCover state create a Blend Node that blends appropriately between the Left Cover Sneak, Cover Idle, and Right Cover Sneak animations based off the value of the property Creep. These animations can be found in Project > Adventure Player by Unity >Cover Animations from Mixamo.

Enemy Animation

The Animator for the EnemyBot (ACS-17) has all states in it, but we want to implement this animation without using the transitions in the Animator. Instead, please use the Animator.CrossFade() method. Note: The Animator for the enemy is on the ACS-17 child GameObject, not the root-level Enemy GameObject.

- Make changes to the ACS17AnimationManager script that is already attached to the ACS-17 GameObject.
- Read public mode and turnDir information from EnemyNav on the parent GameObject to know which Animator state to switch to:
    - When mode == (idle || wait), show the ACS_Idle state.
    - When mode == move, show the ACS_Forward state.
    - When mode == (preMoveRot || postMoveRot), show ACS_TurnLeft or ACS_TurnRight based on turnDir.
- Adjust the speed of each ACS animation to match the movement of the enemy.
- Try using the anim.CrossFade() method with a transition time of 0. It will make things easier.

Bonus Challenge

Try changing the transition time of the CrossFade() method to something like 0.25f. This will require you to call CrossFade() once only for each transition to a new state (otherwise, the cross fade will continually restart and appear to freeze the animation).

Resources

You will be using the ACS-17 robot designed by Vladimir Tim.

Solution

Keep in mind your solution to the challenge is likely to be different than ours.

Enemy Animation

For this challenge, we need to implement the transitions for our ACS-17 robot without using the transitions within the Animator. We started by placing the ACS17AnimatorManager script on our ACS-17 robot. Within this script, we have access to the Animator and the EnemyNav placed on the parent, allowing us to access the enemy nav mode information and the turn direction information. In the Start() method, we get a reference to our Animator component, and we also reference the parent with our EnemyNav component. If EnemyNav component is null, we don't set this as initiated. If it's not initiated, it will return without trying to set the states, preventing new errors. In the Update() method, we created a switch state that's based off our Nav mode. Within this statement, we set our anim.CrossFade to idle if either idle or wait. We also set our CrossFade to rotate if it's either preMoveRot or postMoveRot, allowing it to turn left if that direction is negative or turn right if the turn direction is positive. Lastly, if we are moving, we set our CrossFade to walk, allowing our ACS-17 to move appropriately.

Bonus Challenge

For this bonus challenge, we were asked to set up an animation rotation time for our ACS-17 that wasn't 0 so the transition would not instantaneously cause our animation to freeze. To solve this, we simply added a new CrossFade() method to our ACS17AnimationManager. The new CrossFade() method takes in the new state as a string: the name of the state we are transitioning to within the animation state machine. If the new state is not the current state, we then call anim.CrossFade() and set the current anim state to be this new state. However, if it's already cross fading to that state, we set it to not call the anim.CrossFade() again, preventing any weird issues that may occur with it constantly resetting the transition. Within our state machine under the Update()method, we set it to call the CrossFade() method we just created instead of the anim.CrossFade(). By doing it this way, we ensure our animation won't repeatedly be called or freeze during transition.

Activity 3: Camera Control

Challenge Prompt

With player animation and movement established, tweak the camera to have the standard move-to-look-down-halls behavior that's seen in other stealth games, like *Metal Gear Solid*. Using information from the ThirdPersonWalkCover.GetCoverInfo() method, modify the StealthPlayerCamera script to move to look down hallways when the Player is in cover and near the edge of a hallway. You will be successful if the camera zooms in on the player when she's in cover and moves close to a corner, as in the Challenge video.

Tasks to Complete

Camera

- The default is a camMode of eCamMode.far.
- Use coverInfo.zoomL and coverInfo.zoomR to determine whether to move into eCamMode.nearL or nearR.
- In the near camModes, the camera should move closer to the height of the Player's head and look down the hallway that she's next to.
- Use the coverInfo.inCover value to determine which direction the camera should look in the near modes.

Bonus Challenge

When the camera is in a near mode and looking in the -Z direction (inCover=2), the left and right arrows should move the Player in the opposite direction than usual. However, it's more subtle than that. If the camera is in far mode, and the player is inCover==2 and holding left, when the camera switches directions, the controls should not switch direction until the Player releases the key.

Solution

Camera

This challenge calls for us to add to our StealthPlayerCamera on the Main Camera. In the Update(), we first check to see if we are in cover by checking the camMode. By adding an else clause to the states when we're taking cover, we can check our zoomL and zoomR bools against one another. If zoomL is true and zoomR is false, we know we must set the camMode switch statement to eCamMode.nearR; alternatively if zoomR is true and zoomL is false, we know to set the camMode to eCamMode.nearL. Otherwise the camMode will default to eCamMode.far. By setting the eCamMode to nearL or nearR, the Camera then knows we are looking around a corner to the left or right and can zoom in accordingly, based off the relative position of our player.

To get a smooth transition, we set the camera to interpolate our Vector3 variable between the desired origin point and desired rotation, and the near point of our character. This ensures that the camera is set to the height of the Player's head as the challenge requests. By defaulting our Vector3 to Vector3.zero, it sets the Camera's desired position to the origin by default, which easily allows us to see if there's a bug within our code. The Camera should snap to the Player and not be at the origin, so if it's there, we clearly know something is wrong.

Bonus Challenge

We started with our GetCreepValue() method within the ThirdPersonWalkCover script. In case 2, we look for the camera mode we want, either near or far, and if it's not the same as the last camera mode (so that our far and near cameras are swapped in this frame, and the value of our horizontal direction is greater than 0.1f, either left or right), then we set our creepCamReversedButPlayerIsStillHoldingSameDir bool to true. It maintains true as long as the user holds down the key. This allows the Player to continue moving in one direction as long as the key is pressed, even if the camera flips from far to near. To set the creepCamReversedButPlayerIsStillHoldingSameDir bool to false, we check for two cases. The first case is that our horizontal direction is less than 0.1f; if so, we release a button. The second case is it switches to false if the player has switched directions, in which case, it checks the current eCamMode and repositions it accordingly.

Activity 4: Environmental Interactions

Challenge Prompt

It's time to add some more obstacles to the Scene and make them and the existing enemy robot actually sense the Player. In this challenge, implement a SecurityGate, a Desk panel to disable the SecurityGate, and automated SecurityCameras.

Tasks to Complete

Desk

- Create a Desk with a panel that the Player must find and activate to disable the SecurityGate.
- The Desk should glow green when the Player is within the trigger attached to it. The green border of the Desk's SecurityRoomDesk Material should pulsate as described in the video.
- When the Player is within the trigger and presses the space bar, the Security Gate should be disabled.
- The Desk should glow green before the Player disables the Security Gate, but should not glow after the Security Gate has been disabled.
- When the Player interacts with the Desk, the laser beam effect on the Security Gate should be disabled.

Security Camera and Light Cone

- Create a Security Camera that pans back and forth, looking for the Player.
- Implement the SecurityCamera's back and forth motion via a PlayableDirector and Timeline Playable.
- The SecurityCamera should make use of the same faux Volumetric Lighting cone that the EnemyBot uses.
- Read the code and comments for this LightCone to determine how to use it to sense when the Player is seen by the SecurityCamera.
- When the SecurityCamera sees the Player, the game should switch to Alert mode.

Tips: *An InteractingPlayer Component has been added to the Player GameObject. This can make it very easy for triggers the Player walks into to show they are colliding with the Player. Please make use of it. Also, the AlertModeManager script has already been written and attached to the _GameManager GameObject. Please make use of, and expand, the scripts contained in the __Scripts > Player Interaction folder.*

Solution

Security Gate

To disable the Security Gate, we created a PlayerAction_DeactivateGameObject script, which derives from our PlayerAction script, and placed it on the Desk. The script gets called when the Player interacts with the Desk by pressing the space bar. Within the script, there is an int and a reference to a GameObject. The int is our timesLeftToActivate, which will tell our script that, after being used once, to no longer be on our list of actions — the gameObjectToDeactivate will be our Security Gate. We have an Action() that deactivates our gameObjectToDeactivate as long as it's not null, which ultimately is the same as disabling the Security Gate from within the Inspector.

To make the Security Gate alarm go off when the Player enters it before the Desk has been activated, we used the PlayerInteractable script, which checks to see if the Player is within the trigger and, if so, we execute the Alarm, which sets our game to Alert Mode and tells the robots to search for our player.

Desk

We created multiple scripts within our Desk object. The first is HighlightWhenPlayerInTrigger, which we based off the PlayerInteractable script instead of using OnTriggerEnter / OnTriggerExit. That way, we make use of the PlayerInteractable functionality that we'd already created. When we've selected a gameObjectToHighlight, such as the Desk, we look for specific values within the toon Shader. Using the Project window to find the Shader file, we use the Inspector to figure out what values need to be changed. Within the HighlightWhenPlayerInTrigger script, we set the gameObjectToHighlight to have no outline, and we set the size of the border to 0 to ensure that the Desk's border won't show up on Start(). Within the Update(), if the Player is within the trigger defined by our PlayerInteractable, we turn on the outline and set the border color to the highlight color that's set in the Inspector.

We set the startTime to be the current time, then set the outline width based off a cosine value. This gives our Desk the nice green border that pulses when the Player is within its trigger area. We made sure to set the outline and border back to zero upon leaving the trigger and set the startTime to -1, which will cause it to reset the next time the Player enters the trigger. Once there are no more actions to take with the Desk, the PlayerWithinTrigger sets to false, stopping the Desk from highlighting further.

Security Camera and Light Cone

To make the light cone interact with the Player without having a conventional OnTrigger event, we made a PlayerInteractable_LightCone script that has no Triggering Key and a bool that tells us if our Player is within the triggerable area. We start the script by grabbing a reference to the LightCone component using GetComponentInchildren. We chose to use GetComponnentInChildren because it allows us to be more flexible with the location of the LightCone component. Within the FixedUpdate(), we get the Raycast hits that are being sent out by the LightCone in the form of an array. If the hits are not null, we iterate through each one and check if the Collider is not null. If the Collider is not null, we compare that Collider to the Collider of the InteractingPlayer. If that hit returns true, we then know we've hit the

Player. We do it this way because GetComponentInParent can be slow, and we can avoid a performance hit that may be noticeable otherwise. Within the Update, we check to see if the PlayerWithinTrigger is true; if so, we then call ExecuteActions() which executes our Alarm.

To make the Security Cameras pan back and forth, we opened our Timeline and selected the Camera_Cam object to see its timeline and keyframes within the Animation window. We set the rotation keyframes from -45 to positive 45 to make the Camera pan back and forth. We made sure our Animation is set to loop and will play on start.

Chapter 4: 3D Art & Audio Pipeline

Activity 1: Red Alert

Challenge Prompt

In this challenge, you will make both graphical and coding changes to implement a “Red Alert” in the game. Luckily, the AlertModeManager script makes it very easy for scripts to register themselves to be called whenever the game switches into or out of Alert Mode.

Tasks to Complete

All of the following changes should occur when Alert Mode is activated and revert when Alert Mode is deactivated:

Code

- Add a script to each GameObject that will register itself with the AlertModeManager and will change into or out of its AlertMode state when the alertModeStatusChangeDelegate() is called.

Ambient Light, Walls, and Light Cones

- The emissive color of the lights at the bottom of each wall segment should switch from cyan to bright red.
- The ambient light of the Scene should also switch from blue to a dim yellowish-green that contrasts well with the red of the rest of the Scene.
- The glow on the Desk should change from green to red.
- A new holographic Company Logo Prefab has been added to the Scene in several places. This hologram should switch from the logo (in white, split into three channels: red, green, & blue) to an ALERT! graphic in red (IntruderAlert.psd).
- The LightCones of the Enemy ACS-17 and the Security Camera should switch from amber to red.

Enemy ACS-17

- In addition to the graphical changes listed above, the Enemy ACS-17 should leave its patrol route and chase after the Player. To do this, you must implement the chase and stopChase modes in the EnemyNav script.
- When Alert Mode is disabled, the Enemy ACS-17 should return to its patrol route.

Solution

Code

We set up AlertModeStatusChangeDelegate(), which is called any time the Alert Mode changes, whether it is newly true or false.

Ambient Light, Walls, and Light Cones

We then set up the AlertModeAmbientLightModifier script, which holds the value of our original ambient light color and the color we want to switch to when the Alert Mode is activated. Within the AlertModeAmbientLightModifier script we initialize the original color and then we register our AlertModeStatusChange() method within the delegate, being sure to de-register the method once we call OnDestroy(). Within the AlertModeStatusChange() method we then set our new Alert mode color based off whether the Alert Mode is set to true or false.

To change the color of the walls, we created an AlertModeColorModifier script. First the script checks to make sure our walls have the material and the parameter emission color. After ensuring that the object that is changing color has the specified material and parameter we want, we pull the original color out of the shader. We register the script within the AlertModeStatusChangeDelegate to indicate that we have gone into Alert Mode, and we change the color once we have.

To change the color of the light cone, we created a new AlertModeLightConeModifier() script, which grabs the original color by looping through the cone. We set the color to our new Alert color once the Alert delegate is called.

To change the color of the hologram, we created a AlertModeSpriteModifier script, which grabs the Sprite Renderer of the hologram, referencing the sprite and individual color of the hologram. We set the sprite to the Alert sprite and then set the color to the Alert Mode color using a Lerp using a colorBlend value we defined. By setting the the colorBlend between 0 and 1, we can get a slight difference in the color for the panels, allowing us to see the three individual hologram layers and glitching better.

Enemy ACS-17

To make the ACS-17 chase the player in Alert Mode, we created a script AlertModeEnemyStateModifier, which uses our delegate to determine whether we are in Alert Mode, and then sets the eMode to either EnemyNav.eMode.chase or EnemyNav.eMode.stopChase, depending on the Alert Mode setting.

Activity 2: Audio

Challenge Prompt

Sound is a critical part of modern games, and you need to understand how to implement and modify sound within Unity. This challenge includes multiple methods of implementing and modifying sound in Unity projects. You will add background music and ambient sound, including the challenging task of adding footsteps.

Tasks to Complete

Audio Listener and Reverb Zones

- Remove the Audio Listener on the Main Camera and add one to the Player.
- Create Reverb Zones for the Hallways and give the carpeted part of the room a different kind of reverb from the polished floor sections.

Audio Mixer 1 & 2

- Create an Audio Mixer named _AudioMixer to manage all of the audio sources. You should create groups under Master named Music, Ambient, Footsteps, Alarm, and SecurityGateBeams.
- Create two different Snapshots for the Audio Mixer, so that when the Player is against cover, the music and footsteps are quieter and the low frequencies of the footsteps and high frequencies of the music are muted.

Background Music and Ambient Sound

- Add the Lingo ED1 music by Andre Bowman as background music. Because this is pervasive ambient sound, it makes sense to make it a child of _GameManager. This should output to the Music _AudioMixer group. By making it a child of _GameManager, it will later be part of the _PersistentScene and therefore not destroyed when various levels are loaded and unloaded. The _PersistentScene is part of the multiple levels challenge later in this course.
- To create ambient sound, add the SecurityRoomAmbient sound as a GameObject with no parent (this is because we may want to have different ambient sounds for different levels, so it should not be part of the _PersistentScene). This should output to the Ambient _AudioMixer group.

Alert Sound

- To add an Alert, create a child of _GameManager that has an Audio Source for the Alarm from AaS_VerbProcA_100-03 by MusicRadar. This source should only sound when Alert Mode is active, so you should implement an AlertModeAudioPlayer script to manage this.

Location-Based 3D Sound

- Generate location-based 3D sound by adding an Audio Source component for the laser_hum sound effect to the SecurityGateBeams GameObject. Make sure the Spatial Blend is set so this

is a 3D sound. As a component of SecurityGateBeams, this sound will be disabled with the beams are turned off.

Footsteps

- Create an AudioFootstepSoundsScriptableObject Scriptable Object to hold all four footstep clips from the Standard Assets in a public List<AudioClip> clips. This script should include a public AudioClip GetClip() method that returns a random AudioClip from clips and ensures that it never returns the same AudioClip twice in a row.
- Create a new AudioGroundSound script and attach it as a component of the *Hallway_Floor* child of the Hallway prefab. This component should have a public AudioFootstepSoundsScriptableObject AudioFootstepSO field and a public AudioClip GetClip() method. GetClip() calls GetClip() on AudioFootstepsSO.
- Attach both an Audio Source and a new script named AudioFootstepCollider to each FootCollider. The Audio Source should have no clip assigned, should output to the Footsteps Audio Mixer group, and should have *Play On Awake* not checked.
- When AudioFootstepCollider collides with another GameObject, make it look for an AudioGroundSound component attached to the GameObject and, if found, make it call GetClip() on that component and play the clip using the AudioSource component on the same FootCollider.
- Implement AudioFootstepCollider such that the footstep sound doesn't play when a new level loads.

Links and Resources

Music created by Andre Bowman. You can contact Andre via his LinkedIn account: https://www.linkedin.com/in/andre-bowman-12602116

Solution

Audio Listener, Reverb Zones

We first removed the Audio Listener from the Main Camera and placed it on our Player. We can have only one Audio Listener per scene, so we wanted to be sure we hear what's near the Player, not the Camera. To add our Reverb Zones, we created a new empty GameObject, which will become our Reverb Zone folder, and then we placed four new GameObjects within it. We placed a Reverb Zone on each of the GameObjects and moved them to their preferred location in the Scene, setting each of their Reverb Presets as desired. Placing them within a folder GameObject keeps the Hierarchy a little bit cleaner.

Audio Mixer 1 & 2

We opened our Audio Mixer from within the Windows menu and then created a new Mixer named _AudioMixer. To create the individual channels, we used the Groups section of the Mixer, adding new groups for each group specified by the challenge. By ensuring that all of the new groups were children of the Master group, we can use the Master group to control the audio levels of all child groups.

To create two different Snapshots to modify the sound when the Player is against cover, we used the Snapshots section of the Audio Mixer window. After creating the Snapshots — one for cover and the other for no cover — we set the music and footsteps levels to the desired volume for the Snapshot. We then lowered the Lowpass Cutoff Frequency of our Cover Snapshot Music group from 22000 to 5000, removing the crisper sounds from our music. Alternatively, for the Cover Snapshot Footsteps Group, we changed the Cutoff Frequency of the Highpass from 10 to 1000, allowing only higher-pitched sounds to come through, making the steps sound lighter. By lowering the music and footsteps volume and changing the Highpass and Lowpass, we gave the game a more stealthy feel.

Background Music and Ambient Sound

Once we set up our groups, we added the background music and ambient sound to our Scene. For the background music, we added a new empty GameObject under the _GameManager that has an Audio Source component. Within this Audio Source, we placed our background music and set Play on Awake as well as Loop to true, which ensures that our background music will play when the player starts a game. For the ambient audio, we created a new empty GameObject named Audio_SecurityRoomAmbient and placed it within our Security Room. We added an Audio Source component and set it to play our SecurityRoomAmbient noise, making sure both Play on Awake and Loop were checked. We chose to place the Background music object under the _GameManager to ensure that the music will persist within our Scene, whereas the Audio_SecurityRoomAmbient, being an object of our Scene, will not persist through level transitions.

Alert Sound

We copied the same procedure we used to set up the background music. We placed a new GameObject as a child of our _GameManager object to ensure it persists, and added an AudioSource. Within the AudioSource, we set the Alarm sound as the AudioClip, set Loop to true, and made sure that Play On

Awake was set to false. That way, the Alarm will not sound immediately upon starting. We created an AlertModeAudioPlayer script and placed it on our Audio_Alarm object.

Within our script we have an Audio Source reference that's the source we placed on our Audio_Alarm object. We used the AlertModeManager delegate that we previously set up to tell our script whether we are in Alert mode. Then, if our Alert Mode status changes to true, we play the Audio Source; otherwise, when false, it stops the Audio Source.

Location-Based 3D Sound

After getting our Alert Sound set up, we set up our location-based 3D sound by placing our Audio Source on our SecurityGateBeams object. By placing it on the SecurityGateBeams child and not the SecurityGate itself, we guarantee the audio will stop playing as soon as the beams are disabled. Within our Audio Source, we set Play on Awake and Loop to true, allowing this sound to play when the game starts. In the Output field, we placed our SecurityGameBeams group from our Audio Mixer. By changing the Output to the Mixer, we give the Mixer control over the audio levels of our beam. We moved the Spatial Blend slider all the way over from 2D to 3D, which causes our SecurityGateBeams audio to change based off the Player's position relative to the sound.

Footsteps

We set up a ScriptableObject to hold all our footstep sounds, created an AudioGroundSound script to get our sounds, and set up our foot Colliders in a way that plays the audio appropriately. We placed a small sphere Collider within the toe of each foot, making sure each Collider barely extends below the foot to touch the ground when the character walks. Within our Collider object, we placed the script AudioFootStepCollider, which looks for an OnTriggerEnter event and then tries to grab the AudioGroundSound component of whatever the Players's foot stepped on. If the event returns null, we just return; otherwise, we grab the Audio Clip from the AudioGroundSound script and play it.

To get the AudioClip, the AudioGroundSound script holds a reference to the footstep ScriptableObject. It then asks it which footstep sound should be played by returning AudioFootstepsSO.GetClip(). Within the AudioFootstepSoundsScriptableObject, we call the GetClip() method to iterate through all the available clips in the AudioClips list and return a random clip. It checks that clip against the lastClip to make sure the same sound doesn't play twice.

We set up our footsteps so the sound doesn't play immediately upon starting the game. To prevent footsteps sounds from playing immediately upon start, we set a bool called muteNextStep that initializes as true. Upon the first footstep, when the character is spawned and first makes contact with the floor, we turn that bool to false, stopping the loud double footstep sound we were getting when the game started.

Activity 3: Adding Multiple Levels

Challenge Prompt

In this challenge, you'll create a new level for the Stealth game and create a Game Manager that will handle the transitions between levels and when the Player is captured.

Create a second level and link it to Level 1. The game should automatically load Level 1 on start, and transition to Level 2 once the Player reaches the _LevelGoal of Level 1. It should then transition back to Level 1 once the player reaches the _LevelGoal of Level 2.

If the player is caught, the LevelAdvancePanel should show the same level again, and then reload the current level.

Tasks to Complete

Persistent Scene Setup

- Add the _GameManager, StealthPlayerCamera, MiniMap, and Player to the _PersistentScene Scene. The LevelAdvancePanel from **AsteraX** has already been added to the _PersistentScene to provide a smooth transition between levels.

Game Management and Level Setup

- Remove the following GameObjects from both the Level 1 and Level 2 Scenes:
    - _GameManager
    - StealthPlayerCamera
    - MiniMap
    - Player
- Author a GameManager.cs script (as a component of _GameManager) that manages the additive loading and subtractive unloading of scenes. This GameManager script should also have a delegate event for LEVEL_START_EVENT (when the level starts) and LEVEL_END_EVENT (when the level ends).
- Make the GameManager handle level failure (when the Player is caught by an Enemy) and restart the same level in that case.
- Whenever loading a level, the GameManager should call AudioFootstepCollider.MUTE_NEXT_STEPS() to prevent the footstep sounds when the level

loads. (This could also be done by each AudioFootstepCollider registering for LEVEL_START_EVENT and having a non-static MuteNextStep() method.)

- Modify the EnemyNav script so that it causes the level to be failed when the Enemy is chasing the player and catches her.
- The LevelAdvancePanel should be implemented by the GameManager.
- Give each level a GameObject named _PlayerStart that gives the Player her initial position and orientation in the level.
- Give each level a GameObject named _LevelGoal with a trigger volume that triggers successful level completion when the Player runs into it.

Solution

Persistent Scene Setup

We started by ensuring that our Scenes had been set up appropriately within the Build Settings, starting with our PersistentScene, then followed by Level 1 and Level 2. We made sure the PersistentScene was the zeroth Scene so it will load first. We created the GameManager script on our _GameManager object, in which we set up a public delegate for setting up our events LEVEL_START_EVENT and LEVEL_END_EVENT. We created an enum, eGameManageState, which contains the states for idle, preLevel, level, and postLevel. In preLevel and postLevel, the LevelAdvancePanel will be showing and we will additively and subtractively manage our Scenes, allowing for smoother transitions.

Game Management and Level Setup

In the Start() method, we set the state to postLevel and then called LevelAdvancePanel.FadeIntoEndLevel(), passing in LoadLevel as our callback. When this callback happens, LoadLevel() is called without a parameter, which then calls the LoadLevel() with a parameter of -1, causing it to use the level that's the current level Num. At the end of LoadLevel(), the coroutine LoadSceneAndSetActive() lets us pass in the name of the Scene to load. To load and unload the Scene, we use the coroutine to check within SceneManager to see if there's more than one level, then check if the Scene count is greater than 1 and has a Scene with the passed-in name. If those conditions are met, we unload the Scene asynchronously. We then moved the Player outside the Scene to prevent any collision issues while loading the next Scene. Next, using a yield, we load the Scene asynchronously, ensuring that we have additive mode set. By doing this, we can wait for the Scene to finish loading before moving to the next line. Once the Scene has loaded, we set it to a newlyLoadedScene variable and set the active Scene to the newlyLoadedScene. By doing, this we can unload this Scene later when we're ready to load in another new Scene. After everything has finished loading, we set the game state to preLevel, which then calls the FadeOutToBeginLevel() on the LevelAdvancePanel with a callback of StartLevel().

In the StartLevel() method, we made sure to switch Alert Mode off so the level doesn't start with a loud alarm going off. We then found the _PlayerStart Transform, which is the starting point for the Player within the Scene, and moved the Player to that position. We reset the Stealth Camera to the far position to make sure it's not zoomed in right away. We set the game state to the "level" state and called the LEVEL_START_EVENT event so we can properly initialize the MiniMap and not cause any bugs. To do so, we edited the previously created MiniMap script getting rid of the original FillCameraWithLevel() call and replacing it with GameManager.LEVEL_START_EVENT += FillCameraWithLevel, which means any time LEVEL_START_EVENT is called by the GameManager, the FillCameraWithLevel() will be called as well. Within the MiniMap_Manager script, to ensure the MiniMap blip is still work as intended, we added our AssignBlips() method to the LEVEL_START_EVENT delegate so our blips will be added as soon as the game starts. To make sure we remove the blips, we registered the DestroyActiveBlips() method with the GameManager.LEVEL_END_EVENT, which is called when the level ends. By doing this, we avoid issues such as an object being tracked that has already been unloaded.

To ensure the level change, we placed the LevelGoal script on a trigger at the end of the level, which tells the GameManager that the Player has entered the goal. In the EnemyNav script, we created a trigger event to notify us when the enemy has caught the Player, and notify the GameManager to restart the level. With that, we've now implemented multiple levels.

Activity 4: Self-Assessment

Challenge Prompt

Test your project and evaluate it according to the rubric.

Grading Criteria

GameManager Script - 1 Point
The programmer has created a GameManager script that manages the additive loading and subtractive unloading of levels from _PersistentScene.

Level Restart - 1 Point
When the player's character is “caught” by the enemy robot, the level restarts.

Next Level Loading - 1 Point
When the player’s character reaches the level goal area, the LevelAdvancePanel appears and the next level loads (or the first level, if the player has just completed the last level in the game).

Starting Position - 1 Point
When a new level loads, the player’s character is placed in the correct starting position and orientation for that level.

MiniMap Reload - 1 Point
When a new level loads, the MiniMap reloads and represent the new level and its contents accurately.

Closing

We hope that you've learned a lot by participating in this workshop, and it will help you on your path as a Unity developer. Take some time to update your Learning Action Plan, and be sure to share your experience with your peers!

Unity offers several in-person workshops like the one you experienced today. If you are interested in learning about our other training sessions, check out the Unity Training Workshops page.