Create with Code

Unit 3 Lesson Plans

unity

3.1 Jump Force

Steps:

Step 1: Open prototype and change background

Step 2: Choose and set up a player character

Step 3: Make player jump at start

Step 4: Make player jump if spacebar pressed

Step 5: Tweak the jump force and gravity

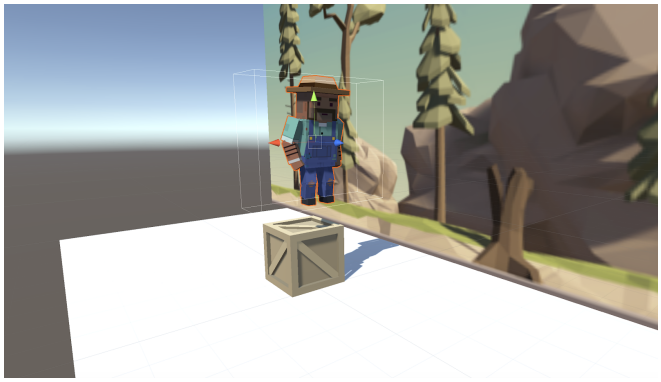
Step 6: Prevent player from double-jumping

Step 7: Make an obstacle and move it left

Step 8: Create a spawn manager

Step 9: Spawn obstacles at intervals

Example of project by end of lesson



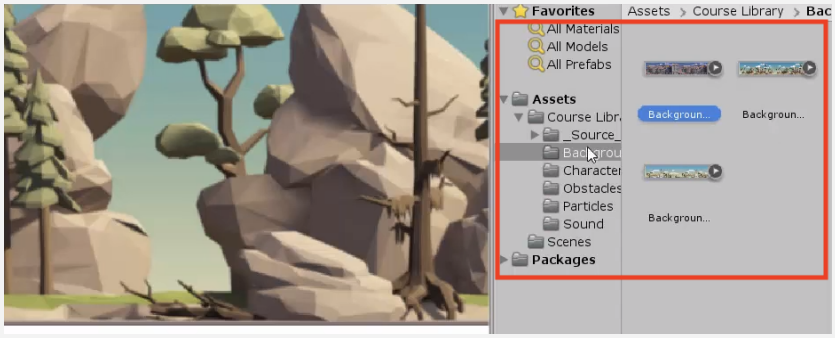
Length:	90 minutes
Overview:	The goal of this lesson is to set up the basic gameplay for this prototype. We will start by creating a new project and importing the starter files. Next we will choose a beautiful background and a character for the player to control, and allow that character to jump with a tap of the spacebar. We will also choose an obstacle for the player, and create a spawn manager that throws them in the player's path at timed intervals.
Project Outcome:	The character, background, and obstacle of your choice will be set up. The player will be able to press spacebar and make the character jump, as obstacles spawn at the edge of the screen and block the player's path.
Learning Objectives:	By the end of this lesson, you will be able to: - Use GetComponent to manipulate the components of GameObjects - Influence physics of game objects with ForceMode.Impulse - Tweak the gravity of your project with Physics.gravity - Utilize new operators and variables like && - Use Bool variables to control the number of times something can be done - Constrain the RigidBody component to halt movement on certain axes

Step 1: Open prototype and change background

The first thing we need to do is set up a new project, import the starter files, and choose a background for the game.

1. Open **Unity Hub** and create an empty "Prototype 3" project in your course directory on the correct Unity version.
   If you forget how to do this, refer to the instructions in Lesson 1.1 - Step 1
2. Click to download the Prototype 3 Starter Files, **extract** the compressed folder, and then **import** the .unitypackage into your project.
   If you forget how to do this, refer to the instructions in Lesson 1.1 - Step 2
3. Open the Prototype 3 scene and **delete** the **Sample Scene** without saving
4. Select the **Background object** in the hierarchy, then in the **Sprite Renderer** *component > Sprite,* select the *_City, _Nature, or _Town image*

- **New Concept:** Sprites / Sprite Renderer
- **Tip:** Browse all of the Player and Background options before choosing either - some work better with others

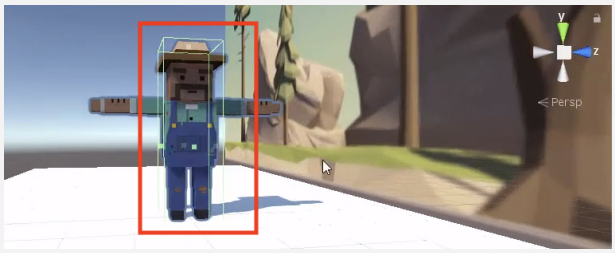


Step 2: Choose and set up a player character

Now that we've started the project and chosen a background, we need to set up a character for the player to control.

1. From *Course Library > Characters*, **Drag** a character into the hierarchy, **rename it** "Player", then **rotate it** on the Y axis to face to the right
2. Add a **Rigid Body** component
3. Add a **box collider**, then **edit** the collider bounds
4. Create a new "Scripts" folder in Assets, create a "PlayerController" script inside, and **attach** it to the player

- **Don't worry:** We will get the player and the background moving soon
- **Warning:** Keep isTrigger UNCHECKED!
- **Tip:** Use isometric view and the gizmos to cycle around and edit the collider with a clear perspective



Step 3: Make player jump at start

Until now, we've only called methods on the entirety of a gameobject or the transform component. If we want more control over the force and gravity of the player, we need to call methods on the player's Rigidbody component, specifically.

1. In **PlayerController.cs**, declare a new ***private Rigidbody playerRb;*** variable
2. In ***Start()***, initialize ***playerRb = GetComponent<Rigidbody>();***
3. In ***Start()***, use the AddForce method to make the player jump at the start of the game

- **New Function:** GetComponent
- **Tip:** The playerRb variable could apply to anything, which is why we need to specify using GetComponent

```
private Rigidbody playerRb;

void Start()
{
  playerRb = GetComponent<Rigidbody>();
  playerRb.AddForce(Vector3.up * 1000);
}
```

Step 4: Make player jump if spacebar pressed

We don't want the player jumping at start - they should only jump when we tell it to by pressing spacebar.

1. In **Update()** add an **if-then statement** checking if the spacebar is pressed
2. **Cut and paste** the AddForce code from Start() into the if-statement
3. Add the ***ForceMode.Impulse*** parameter to the ***AddForce*** call, then **reduce** force multiplier value

- **Warning:** Don't worry about the slow jump double jump, or lack of animation, we will fix that later
- **Tip:** Look at Unity documentation for method overloads here
- **New Function:** ForceMode.Impulse and optional parameters

```
void Start()
{
  playerRb = GetComponent<Rigidbody>();
  playerRb.AddForce(Vector3.up * 100);
}
```

```
void Update() {
  if (Input.GetKeyDown(KeyCode.Space)) {
    playerRb.AddForce(Vector3.up * 100, ForceMode.Impulse); } }
```

Step 5: Tweak the jump force and gravity

We need to give the player a perfect jump by tweaking the force of the jump, the gravity of the scene, and the mass of the character.

1. **Replace** the hardcoded value with a new ***public float jumpForce*** variable
2. Add a new ***public float gravityModifier*** variable and in ***Start()***, add ***Physics.gravity *= gravityModifier;***
3. In the inspector, tweak the ***gravityModifer***, ***jumpForce***, and **Rigidbody** mass values to make it fun

- **New Function:** the students about something
- **Warning:** Don't make gravityModifier too high - the player could get stuck in the ground
- **New Concept:** Times-equals operator *=

```
private Rigidbody playerRb;
public float jumpForce;
public float gravityModifier;

void Start() {
  playerRb = GetComponent<Rigidbody>();
  Physics.gravity *= gravityModifier; }

void Update() {
  if (Input.GetKeyDown(KeyCode.Space)) {
    playerRb.AddForce(Vector3.up * 10 jumpForce, ForceMode.Impulse); } }
```

Step 6: Prevent player from double-jumping

The player can spam the spacebar and send the character hurtling into the sky. In order to stop this, we need an if-statement that makes sure the player is grounded before they jump.

1. Add a new ***public bool isOnGround*** variable and set it equal to ***true***
2. In the if-statement making the player jump, set ***isOnGround = false***, then **test**
3. Add a condition ***&& isOnGround*** to the **if-statement**
4. Add a new ***void onCollisionEnter*** method, set ***isOnGround = true*** in that method, then **test**

- **New Concept:** Booleans
- **New Concept:** "And" operator (&&)
- **New Function:** OnCollisionEnter
- **Tip:** When assigning values, use one = equal sign. When comparing values, use == two equal signs

```
public bool isOnGround = true

void Update() {
  if (Input.GetKeyDown(KeyCode.Space) && isOnGround) {
    playerRb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    isOnGround = false; } }

private void OnCollisionEnter(Collision collision) {
  isOnGround = true; }
```

Step 7: Make an obstacle and move it left

We've got the player jumping in the air, but now they need something to jump over. We're going to use some familiar code to instantiate obstacles and throw them in the player's path.

1. From *Course Library > Obstacles*, add an obstacle, rename it "Obstacle", and **position** it where it should spawn
2. Apply a **Rigid Body** and **Box Collider** component, then **edit** the collider bounds to fit the obstacle
3. Create a new "Prefabs" folder and drag it in to create a new **Original Prefab**
4. Create a new "MoveLeft" script, **apply** it to the obstacle, and **open** it
5. In MoveLeft.cs, write the code to **Translate** it to the left according to the speed variable
6. Apply the MoveLeft script to the **Background**

- **Warning:** Be careful choosing your obstacle in regards to the background. Some obstacles may blend in, making it difficult for the player to see what they're jumping over.
- **Tip:** Notice that when you drag it into hierarchy, it gets placed at the spawn location

```
private float speed = 30;

void Update() {
  transform.Translate(Vector3.left * Time.deltaTime * speed);
}
```

Step 8: Create a spawn manager

Similar to the last project, we need to create an empty object Spawn Manager that will instantiate obstacle prefabs.

1. Create a new "Spawn Manager" empty object, then apply a new **SpawnManager.cs** script to it
2. In **SpawnManager.cs**, declare a new ***public GameObject obstaclePrefab;***, then **assign** your prefab to the new variable in the inspector
3. Declare a new ***private Vector3 spawnPos*** at your spawn location
4. In ***Start()***, **Instantiate** a new obstacle prefab, then **delete** your prefab from the scene and test

- **Don't worry:** We're just instantiating on Start for now, we will have them repeating later
- **Tip:** You've done this before! Feel free to reference code from the last project

```
public GameObject obstaclePrefab;
private Vector3 spawnPos = new Vector3(25, 0, 0);

void Start() {
  Instantiate(obstaclePrefab, spawnPos, obstaclePrefab.transform.rotation); }
```

Step 9: Spawn obstacles at intervals

Our spawn manager instantiates prefabs on start, but we must write a new function and utilize InvokeRepeating if it to spawn obstacles on a timer. Lastly, we must modify the character's RigidBody so it can't be knocked over.

1. Create a new ***void SpawnObstacle*** method, then move the **Instantiate** call inside it
2. Create new **float variables** for ***startDelay*** and ***repeatRate***
3. Have your obstacles spawn on **intervals** using the ***InvokeRepeating()*** method
4. In the Player RigidBody component, expand **Constraints**, then **Freeze** all but the Y position

- **New Concept:** RigidBody constraints

```
private float startDelay = 2;
private float repeatRate = 2;

void Start() {
  InvokeRepeating("SpawnObstacle", startDelay, repeatRate);
  Instantiate(obstaclePrefab, spawnPos, obstaclePrefab.transform.rotation); }

void SpawnObstacle () {
  Instantiate(obstaclePrefab, spawnPos, obstaclePrefab.transform.rotation); }
```

Lesson Recap

New Functionality	• Player jumps on spacebar press • Player cannot double-jump • Obstacles and Background move left • Obstacles spawn on intervals
New Concepts and Skills	• GetComponent • ForceMode.Impulse • Physics.Gravity • Rigidbody constraints • Rigidbody variables • Booleans • Multiply/Assign ("*) Operator • And (&&) Operator • OnCollisionEnter()
Next Lesson	• We're going to fix that weird effect we created by moving the background left by having it actually constantly scroll using code!

unity

3.2 Make the World Whiz By

Steps:

Example of project by end of lesson

Length: 70 minutes

Overview: We've got the core mechanics of this game figured out: The player can tap the spacebar to jump over incoming obstacles. However, the player appears to be running for the first few seconds, but then the background just disappears! In order to fix this, we need to repeat the background seamlessly to make it look like the world is rushing by! We also need the game to halt when the player collides with an obstacle, stopping the background from repeating and stopping the obstacles from spawning. Lastly, we must destroy any obstacles that get past the player.

Project Outcome: The background moves flawlessly at the same time as the obstacles, and the obstacles will despawn when they exit game boundaries. With the power of script communication, the background and spawn manager will halt when the player collides with an obstacle. Colliding with an obstacle will also trigger a game over message in the console log, halting the background and the spawn manager.

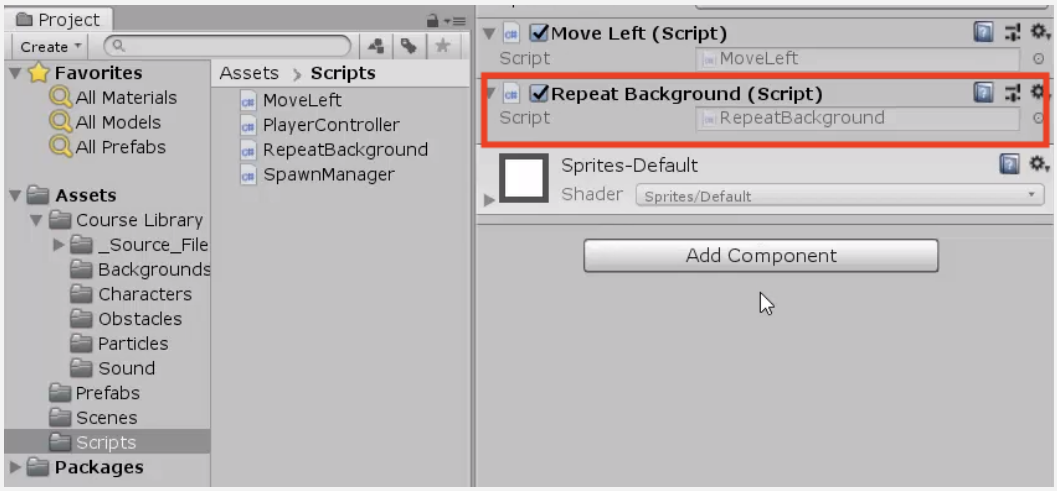
Learning Objectives: By the end of this lesson, you will be able to:
- Use tags to label game objects and call them in the code
- Use script communication to access the methods and variables of other scripts

Step 1: Create a script to repeat background

We need to repeat the background and move it left at the same speed as the obstacles, to make it look like the world is rushing by. Thankfully we already have a move left script, but we will need a new script to make it repeat.

1. Create a new script called **RepeatBackground.cs** and attach it to the **Background Object**

- **Tip:** Think through what needs to be done: when the background moves half of its length, move it back that distance



Step 2: Reset position of background

In order to repeat the background and provide the illusion of a world rushing by, we need to reset the background object's position so it fits together perfectly.

1. Declare a new variable ***private Vector3 startPos;***
2. In ***Start()***, set the ***startPos*** variable to its actual starting position by assigning it ***= transform.position;***
3. In ***Update()***, write an **if-statement** to reset position if it moves a certain distance

- **Don't worry:** We're setting it at 40 for now, just to test basic functionality. You could probably get it right with trial and error… but what would happen if you changed the size?

```
private Vector3 startPos;

void Start() {
  startPos = transform.position; }

void Update() {
  if (transform.position.x < startPos.x - 50) {
    transform.position = startPos; } }
```

Step 3: Fix background repeat with collider

We've got the background repeating every few seconds, but the transition looks pretty awkward. We need make the background loop perfectly and seamlessly with some new variables.

1. Add a **Box Collider** component to the **Background**
2. Declare a new ***private float repeatWidth*** variable
3. In ***Start()***, get the width of the **box collider**, divided by 2
4. Incorporate the ***repeatWidth*** variable into the **repeat function**

- **Don't worry:** We're only adding a box collider to get the size of the background
- **New Function:** .size.x

```
private Vector3 startPos;
private float repeatWidth;

void Start() {
  startPos = transform.position;
  repeatWidth = GetComponent<BoxCollider>().size.x / 2; }

void Update() {
  if (transform.position.x < startPos.x - 50 repeatWidth) {
    transform.position = startPos; } }
```

Step 4: Add a new game over trigger

When the player collides with an obstacle, we want to trigger a "Game Over" state that stops everything In order to do so, we need a way to label and discern all game objects that the player collides with.

1. In the inspector, add a "Ground" tag to the **Ground** and an "Obstacle" tag to the **Obstacle prefab**
2. In PlayerController, declare a new ***public bool gameOver;***
3. In ***OnCollisionEnter***, add the **if-else statement** to test if player collided with the "Ground" or an "Obstacle"
4. If they collided with the "Ground", set ***isOnGround = true***, and if they collide with an "Obstacle", set ***gameOver = true***

- **New Concept:** Tags
- **Warning:** New tags will NOT be automatically added after you create them. Make sure to add them yourself once they are created.
- **Tip:** No need to say gameOver = false, since it is false by default

```
public bool gameOver = false;

private void OnCollisionEnter(Collision collision) {
  isOnGround = true;
  if (collision.gameObject.CompareTag("Ground")) {
    isOnGround = true;
  } else if (collision.gameObject.CompareTag("Obstacle")) {
    gameOver = true;
    Debug.Log("Game Over!"); }
}
```

Step 5: Stop MoveLeft on gameOver

We've added a gameOver bool that seems to work, but the background and the objects continue to move when they collide with an obstacle. We need the MoveLeft script to communicate with the PlayerController, and stop once the player triggers gameOver.

1. In **MoveLeft.cs**, declare a new ***private PlayerController playerControllerScript;***
2. In ***Start()***, initialize it by finding the **Player** and getting the PlayerController component
3. Wrap the **translate method** in an **if-statement** checking if game is not over

- **New Concept:** Script Communication
- **Warning:** Make sure to spell the "Player" tag correctly

```
private float speed = 30;
private PlayerController playerControllerScript;

void Start() {
  playerControllerScript =
  GameObject.Find("Player").GetComponent<PlayerController>(); }

void Update() {
  if (playerControllerScript.gameOver == false) {
    transform.Translate(Vector3.left * Time.deltaTime * speed); } }
```

Step 6: Stop obstacle spawning on gameOver

The background and the obstacles stop moving when gameOver == true, but the Spawn Manager is still raising an army of obstacles! We need to communicate with the Spawn Manager script and tell it to stop when the game is over.

1. In **SpawnManager.cs**, get a reference to the **playerControllerScript** using the same technique you did in MoveLeft.cs
2. Add a condition to only instantiate objects if ***gameOver == false***

```
private PlayerController playerControllerScript;

void Start() {
  InvokeRepeating("SpawnObstacle", startDelay, repeatRate);
  playerControllerScript =
  GameObject.Find("Player").GetComponent<PlayerController>(); }

void SpawnObstacle () {
  if (playerControllerScript.gameOver == false) {
    Instantiate(obstaclePrefab, spawnPos, obstaclePrefab.transform.rotation);
} }
```

Step 7: Destroy obstacles that exit bounds

Just like the animals in Unit 2, we need to destroy any obstacles that exit boundaries. Otherwise they will slide into the distance… forever!

1. In **MoveLeft**, in ***Update()***; write an if-statement to **Destroy** Obstacles if their position is less than a ***leftBound*** variable
2. Add any **comments** you need to make your code more **readable**

- **Tip:** Reference your code from MoveLeft

```
private float leftBound = -15;

void Update() {
  if (playerControllerScript.gameOver == false) {
    transform.Translate(Vector3.left * Time.deltaTime * speed); }

  if (transform.position.x < leftBound && gameObject.CompareTag("Obstacle")) {
    Destroy(gameObject); } }
```

Lesson Recap

New Functionality	• Background repeats seamlessly • Background stops when player collides with obstacle • Obstacle spawning stops when player collides with obstacle • Obstacles are destroyed off-screen
New Concepts and Skills	• Repeat background • Get Collider width • Script communication • Equal to (==) operator • Tags • CompareTag()
Next Lesson	• Our character, while happy on the inside, looks a little too rigid on the outside, so we're going to do some work with animations

unity

3.3 Don’t Just Stand There

Steps:

Step 1: Explore the player’s animations

Step 2: Make the player start off at a run

Step 3: Set up a jump animation

Step 4: Adjust the jump animation

Step 5: Set up a falling animation

Step 6: Keep player from unconscious jumping

Example of project by end of lesson

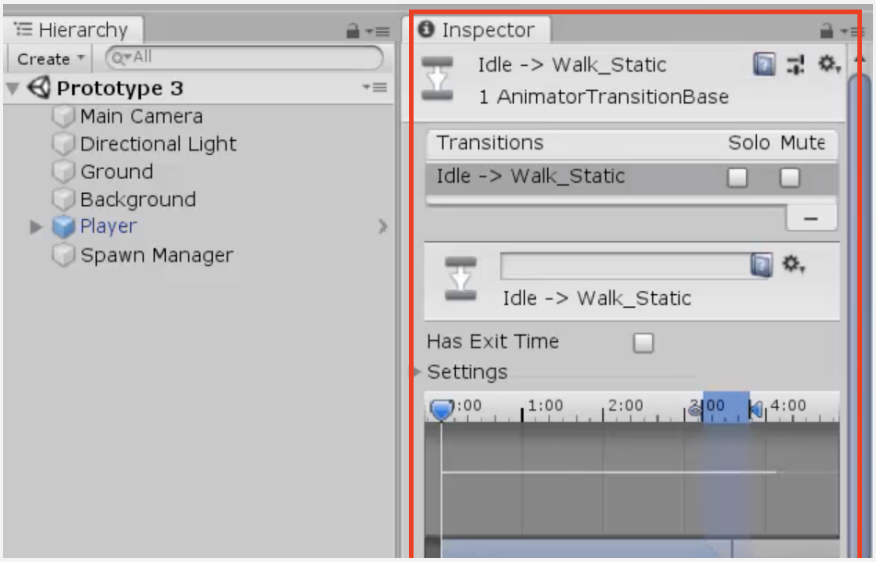
Length:	60 minutes
Overview:	The game is looking great so far, but the player character is a bit… lifeless. Instead of the character simply sliding across the ground, we’re going to give it animations for running, jumping, and even death! We will also tweak the speed of these animations, timing them so they look perfect in the game environment.
Project Outcome:	With the animations from the animator controller, the character will have 3 new animations that occur in 3 different game states. These states include running, jumping, and death, all of which transition smoothly and are timed to suit the game.
Learning Objectives:	By the end of this lesson, you will be able to: - Manage basic animation states in the Animator Controller - Adjust the speed of animations to suit the character or the game - Set a default animation and trigger others with anim.SetTrigger - Set a permanent state for “Game Over” with anim.SetBool

Step 1: Explore the player's animations

In order to get this character moving their arms and legs, we need to explore the Animation Controller.

1. Double-click on the Player's **Animation Controller**, then explore the different **Layers**, double-clicking on **States** to see their animations and **Transitions** to see their conditions

- **New Concept:** Animator Controller
- **New Concept:** States and Conditions

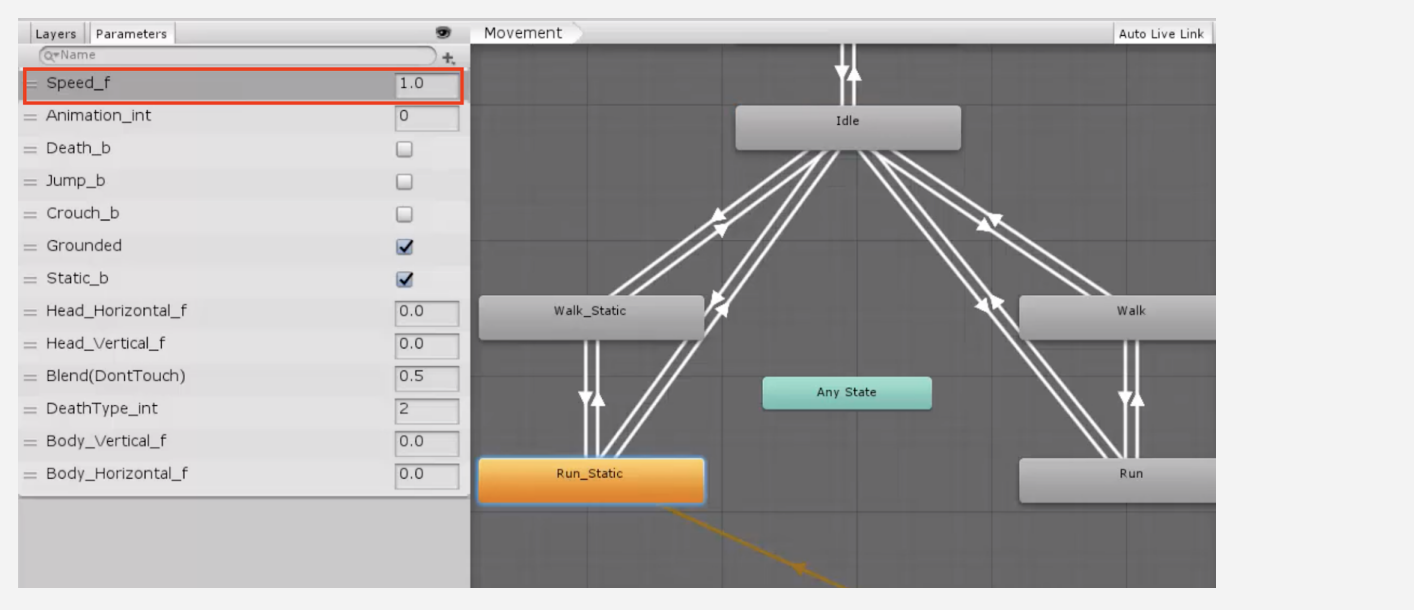


Step 2: Make the player start off at a run

Now that we're more comfortable with the animation controller, we can tweak some variables and settings to make the player look like they're really running.

1. In the **Parameters tab**, change the **Speed_f** variable to 1.0
2. **Right-click** on *Run_Static > Set as Layer Default State*
3. **Single-click** the the Run_Static state and adjust the **Speed** value in the inspector to match the speed of the **background**

- **Tip:** Notice how it transitions from idle to walk to Run - looks awkward - that's why need to make run default



Step 3: Set up a jump animation

The running animation looks good, but very odd when the player leaps over obstacles. Next up, we need to add a jumping animation that puts a real spring in their step.

1. In **PlayerController.cs**, declare a new ***private Animator playerAnim;***
2. In ***Start()***, set ***playerAnim = GetComponent<Animator>();***
3. In the **if-statement** for when the player jumps, trigger the jump: ***animator.SetTrigger("Jump_trig");***

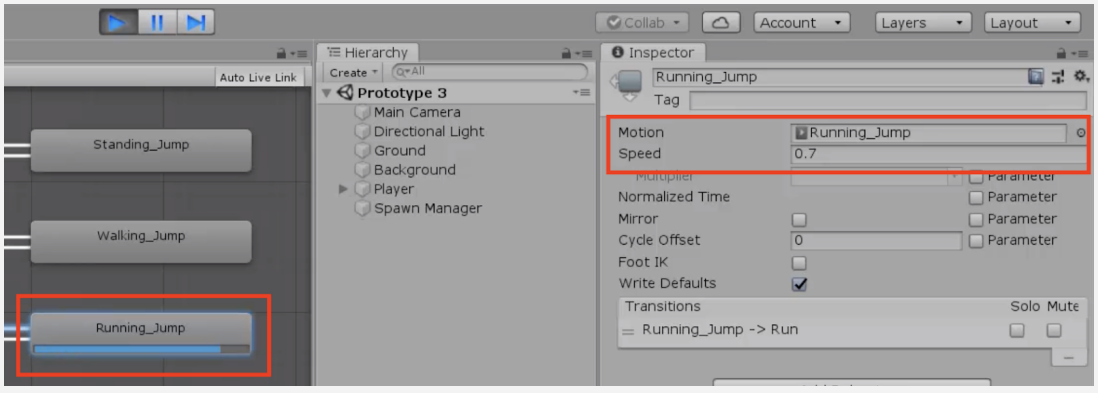
- **New Function:** anim.SetTrigger
- **Tip:** SetTrigger is helpful when you just want something to happen once then return to previous state (like a jump animation)

```
private Animator playerAnim;

void Start() {
  playerRb = GetComponent<Rigidbody>();
  playerAnim = GetComponent<Animator>();
  Physics.gravity *= gravityModifier; }

void Update() {
  if (Input.GetKeyDown(KeyCode.Space) && isOnGround) {
    playerRb.AddForce(Vector3.up * 10 jumpForce, ForceMode.Impulse);
    isOnGround = false;
    playerAnim.SetTrigger("Jump_trig"); } }
```

Step 4: Adjust the jump animation

The running animation plays, but it's not perfect yet, we should tweak some of our character's physics-related variables to get this looking just right.

1. In the Animator window, click on the **Running_Jump** state, then in the inspector and **reduce its Speed** value to slow down the animation
2. Adjust the player's **mass**, jump **force**, and **gravity** modifier to get your jump just right



Step 5: Set up a falling animation

The running and jumping animations look great, but there's one more state that the character should have an animation for. Instead of continuing to sprint when it collides with an object, the character should fall over as if it has been knocked out.

1. In the **condition** that player collides with Obstacle, set the **Death bool** to **true**
2. In the same **if-statement**, set the **DeathType** integer to 1

- **New Function:** anim.SetBool
- **New Function:** anim.SetInt

```
public bool gameOver = false;

private void OnCollisionEnter(Collision collision) {
  if (collision.gameObject.CompareTag("Ground")) {
       isOnGround = true;
  } else if (collision.gameObject.CompareTag("Obstacle")) {
       Debug.Log("Game Over")
       gameOver = true;
       playerAnim.SetBool("Death_b", true);
       playerAnim.SetInteger("DeathType_int", 1);
  }
}
```

Step 6: Keep player from unconscious jumping

Everything is working perfectly, but there's one small disturbing bug to fix: the player can jump from an unconscious position, making it look like the character is being defibrillated.

1. To prevent the player from jumping while unconscious, add ***&& !gameOver*** to the **jump condition**

- **New Concept:** ! "Does not" and != "Does not equal" operators
- **Tip:** gameOver != true is the same as gameOver == false

```
void Update() {
  if (Input.GetKeyDown(KeyCode.Space) && isOnGround && !gameOver) {
    playerRb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    isOnGround = false;
    animator.SetTrigger("Jump_trig");
  }
}
```

Lesson Recap

New Functionality	• The player starts the scene with a fast-paced running animation • When the player jumps, there is a jumping animation • When the player crashes, the player falls over
New Concepts and Skills	• Animation Controllers • Animation States, Layers, and Transitions • Animation parameters • Animation programming • SetTrigger(), SetBool(), SetInt() • Not (!) operator
Next Lesson	• We'll really polish this game up to make it look nice using particles and sound effects!

unity

3.4 Particles and Sound Effects

Steps:

Step 1: Customize an explosion particle

Step 2: Play the particle on collision

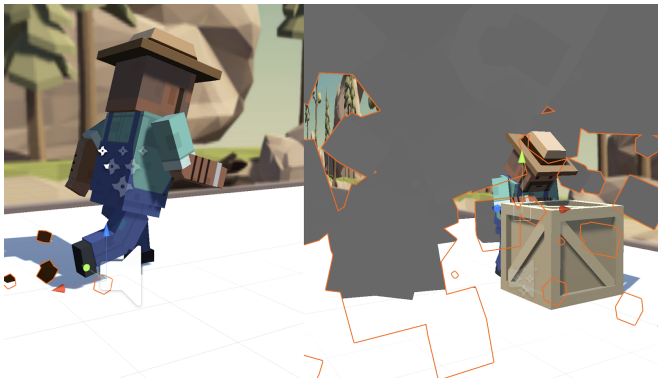
Step 3: Add a dirt splatter particle

Step 4: Add music to the camera object

Step 5: Declare variables for Audio Clips

Step 6: Play Audio Clips on jump and crash

Example of project by end of lesson



Length: 60 minutes

Overview: This game is looking extremely good, but it's missing something critical: Sound effects and Particle effects! Sounds and music will breathe life into an otherwise silent game world, and particles will make the player's actions more dynamic and eye-popping. In this lesson, we will add cool sounds and particles when the character is running, jumping, and crashing.

Project Outcome: Music will play as the player runs through the scene, kicking up dirt particles in a spray behind their feet. A springy sound will play as they jump and a boom will play as they crash, bursting in a cloud of smoke particles as they fall over.

Learning Objectives: By the end of this lesson, you will be able to:

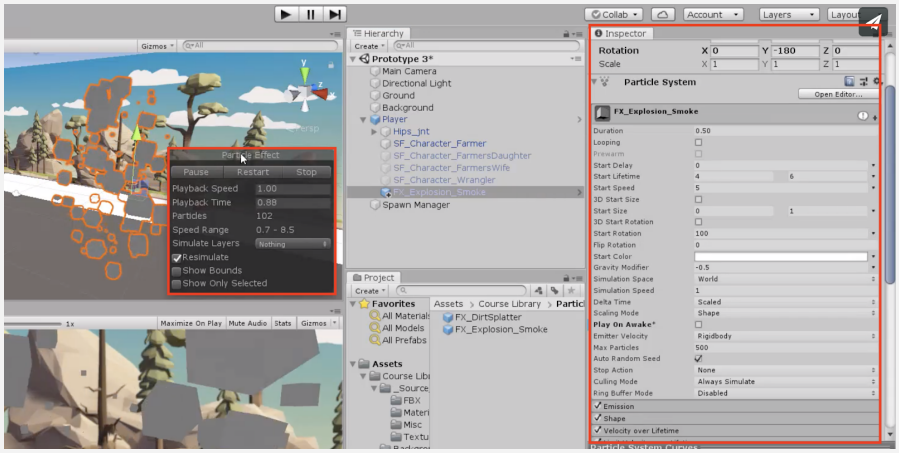
- Attach particle effects as children to game objects
- Stop and play particle effects to correspond with character animation states
- Work with Audio Sources and Listeners to play background music
- Add sound effects to add polish to your project

Step 1: Customize an explosion particle

The first particle effect we should add is an explosion for when the player collides with an obstacle.

1. From the *Course Library > Particles*, drag **FX_Explosion_Smoke** into the hierarchy, then use the **Play** / **Restart** / **Stop** buttons to preview it
2. Play around with the **settings** to get your **particle system** the way you want it
3. Make sure to **uncheck** the **Play on Awake** setting
4. Drag the **particle** onto your player to make it a **child object**, then position it relative to the player

- **New Concept:** Particle Effects
- **Warning:** Don't go crazy customizing your particle effects, you could easily get sidetracked
- **New Concept:** Child objects with relative positions
- **Tip:** Hovering over the settings while editing your particle provides great tool tips



Step 2: Play the particle on collision

We discovered the particle effects and found an explosion for the crash, but we need to assign it to the Player Controller and write some new code in order to play it.

1. In **PlayerController.cs**, declare a new ***public ParticleSystem explosionParticle;***
2. In the Inspector, assign the **explosion** to the **explosion particle** variable
3. In the **if-statement** where the player collides with an obstacle, call ***explosionParticle.Play();***, then test and tweak the **particle properties**

- **New Function:** particle.Play()

```
public ParticleSystem explosionParticle;

private void OnCollisionEnter(Collision collision other) {
  if (other.gameObject.CompareTag("Ground")) {
    isOnGround = true;
  } else if (other.gameObject.CompareTag("Obstacle")) {
    ... explosionParticle.Play(); } }
```

Step 3: Add a dirt splatter particle

The next particle effect we need is a dirt splatter, to make it seem like the player is kicking up ground as they sprint through the scene. The trick is that the particle should only play when the player is on the ground.

1. Drag **FX_DirtSplatter** as the Player's **child object**, reposition it, rotate it, and edit its settings
2. Declare a new ***public ParticleSystem dirtParticle;***, then **assign** it in the Inspector
3. Add ***dirtParticle.Stop();*** when the player jumps or collides with an **obstacle**
4. Add ***dirtParticle.Play();*** when the player lands on the **ground**

- **New Function:** particle.Stop()

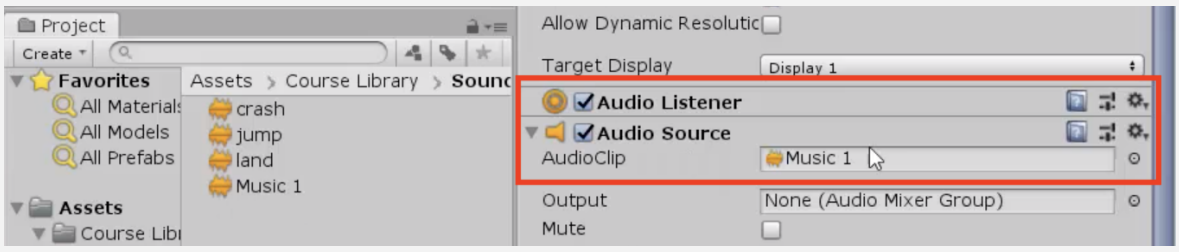
```
public ParticleSystem dirtParticle

void Update() {
  if (Input.GetKeyDown(KeyCode.Space) && isOnGround && !gameOver) {
    ... dirtParticle.Stop(); } }

private void OnCollisionEnter(Collision collision other) {
  if (other.gameObject.CompareTag("Ground")) { ... dirtParticle.Play();
  } else if (other.gameObject.CompareTag("Obstacle")) {  ... dirtParticle.Stop(); } }
```

Step 4: Add music to the camera object

Our particle effects are looking good, so it's time to move on to sounds! In order to add music, we need to attach sound component to the camera. After all, the camera is the eyes AND the ears of the scene.

1. Select the Main **Camera** object, then *Add Component > Audio Source*
2. From *Course Library > Sound*, drag a **music clip** onto the **AudioClip** variable in the inspector
3. Reduce the **volume** so it will be easier to hear **sound effects**
4. Check the **Loop** checkbox

- **New Concept:** Audio Listener and Audio Sources
- **Tip:** Music shouldn't appear to come from a particular location in 3D space, which is why we're adding it directly to the camera

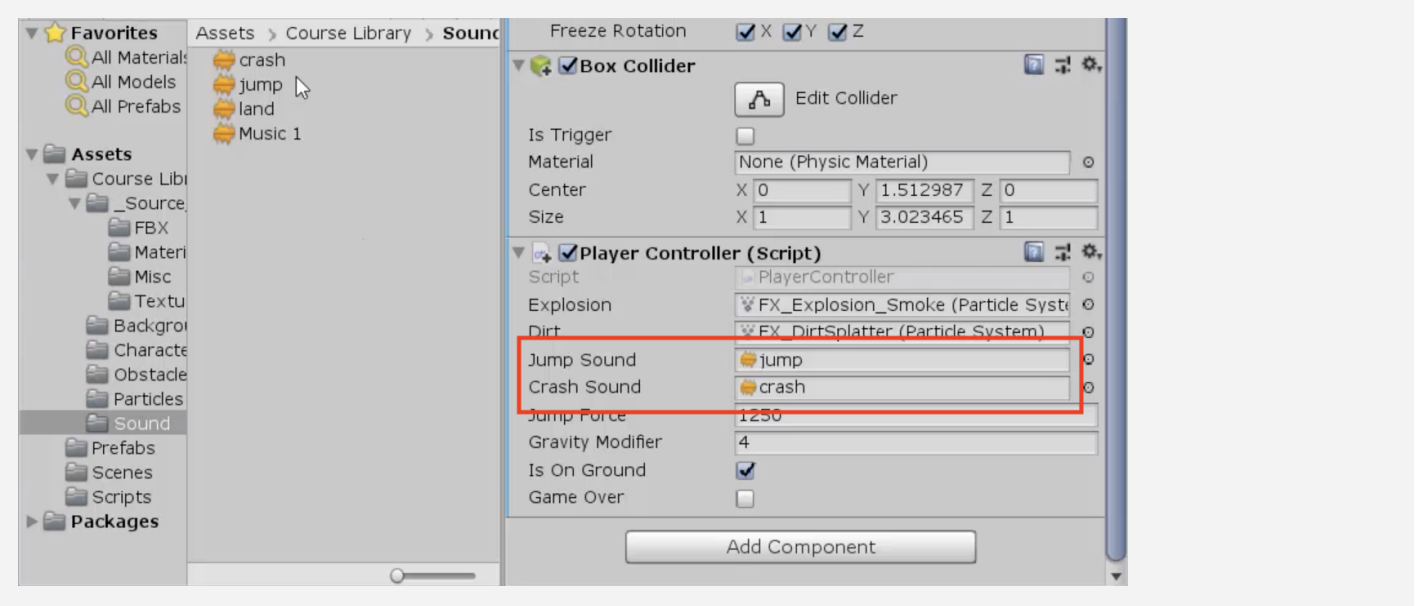


Step 5: Declare variables for Audio Clips

Now that we've got some nice music playing, it's time to add some sound effects. This time audio clips will emanate from the player, rather than the camera itself.

1. In **PlayerController.cs**, declare a new ***public AudioClip jumpSound;*** and a new ***public AudioClip crashSound;***
2. From *Course Library > Sound*, drag a clip onto each new **sound** variable in the inspector

- **Tip:** Adding sound effects is not as simple as adding music, because we need to trigger the events in our code



Step 6: Play Audio Clips on jump and crash

We've assigned audio clips to the jump and the crash in PlayerController. Now we need to play them at the right time, giving our game a full audio experience

1. Add an **Audio Source** component to the **player**
2. Declare a new ***private AudioSource playerAudio;*** and initialize it as ***playerAudio = GetComponent<AudioSource>();***
3. Call ***playerAudio.PlayOneShot(jumpSound, 1.0f);*** when the character **jumps**
4. Call ***playerAudio.PlayOneShot(crashSound, 1.0f);*** when the character **crashes**

- **Don't worry:** Declaring a new AudioSource variable is just like declaring a new Animator or RigidBody

```
private AudioSource playerAudio;

void Start() {
  ... playerAudio = GetComponent<AudioSource>(); }

void Update() {
  if (Input.GetKeyDown(KeyCode.Space) && isOnGround && !gameOver) {
    ... playerAudio.PlayOneShot(jumpSound, 1.0f); } }

private void OnCollisionEnter(Collision collision other) {
  ...
  } else if (other.gameObject.CompareTag("Obstacle"))
  {  ... playerAudio.PlayOneShot(crashSound, 1.0f); } }
```

Lesson Recap

New Functionality	• Music plays during the game • Particle effects at the player's feet when they run • Sound effects and explosion when the player hits an obstacle
New Concepts and Skills	• Particle systems • Child object positioning • Audio clips and Audio sources • Play and stop sound effects

Challenge 3

Balloons & Booleans

Challenge Overview:

Apply your knowledge of physics, scrolling backgrounds, and special effects to a balloon floating through town, picking up tokens while avoiding explosives. You will have to do a lot of troubleshooting in this project because it is riddled with errors.

Challenge Outcome:

- The balloon floats upwards as the player holds spacebar
- The background seamlessly repeats, simulating the balloon's movement
- Bombs and Money tokens are spawned randomly on a timer
- When you collide with the Money, there's a particle and sound effect
- When you collide with the Bomb, there's an explosion and the background stops

Challenge Objectives:

In this challenge, you will reinforce the following skills/concepts:

- Declaring and initializing variables with the GetComponent method
- Using booleans to trigger game states
- Displaying particle effects at a particular location relative to a gameobject
- Seamlessly scrolling a repeating background

Challenge Instructions:

- Open your **Prototype 3** project
- **Download** the "Challenge 3 Starter Files" from the Tutorial Materials section, then double-click on it to **Import**
- In the *Project Window > Assets > Challenge 3 >* ***Instructions*** folder, use the "Challenge 3 - Instructions" and Outcome video as a guide to complete the challenge

Challenge		Task	Hint
1	The player can't control the balloon	The balloon should float up as the player presses spacebar	There is a "NullReferenceExcepton" error on the player's rigidBody variable - it has to be assigned in Start() using the ***GetComponent<>*** method
2	The background only moves when the game is over	The background should move at start, then *stop* when the game is over	In MoveLeftX.cs, the objects should only Translate to the left if the game is *NOT* over
3	No objects are being spawned	Make bombs or money objects spawn every few seconds	There is an error message saying, "Trying to Invoke method: SpawnManagerX.***PrawnsObject*** couldn't be called" - spelling matters
4	Fireworks appear to the side of the balloon	Make the fireworks display at the balloon's position	The fireworks particle is a child object of the Player - but its location still has to be set at the same location
5	The background is not repeating properly	Make the background repeat seamlessly	The ***repeatWidth*** variable should be half of the background's *width*, not half of its *height*

Bonus Challenge		Task	Hint
X	The balloon can float way too high	Prevent the player from floating their balloon too high	Add a boolean to check if the balloon ***isLowEnough***, then only allow the player to add upwards force if that boolean is true
Y	The balloon can drop below the ground	Make the balloon appear to bounce off of the ground, preventing it from leaving the bottom of the screen. There should be a sound effect when this happens, too!	Figure out a way to test if the balloon collides with the ground object, then add an impulse force upward if it does

Challenge Solution

1 In PlayerControllerX.cs, in Start(), assign ***playerRb*** just like the playerAudio variable:

```
playerAudio = GetComponent<AudioSource>();
playerRb = GetComponent<Rigidbody>();
```

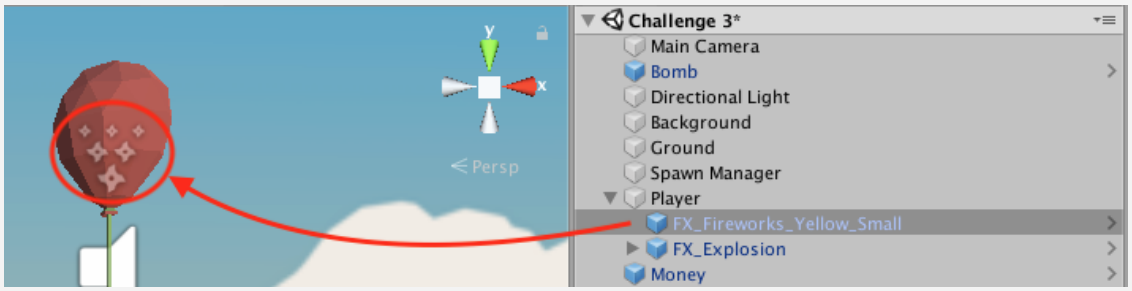
2 In MoveLeftX.cs, the objects should only Translate to the left if the game is NOT over - it's currently checking if the game IS over:

```
if (! playerControllerScript.gameOver) {
  transform.Translate(Vector3.left * speed * Time.deltaTime, Space.World);
}
```

3 In SpawnManagerX.cs, in Start(), the InvokeRepeating method is using an incorrect spelling of "SpawnObjects" - correct the spelling error

```
void Start() {
  InvokeRepeating("PrawnsObjectSpawnObjects", spawnDelay, spawnInterval);
  ...
}
```

4 Select the Fireworks child object and reposition it to the same location as the Player



5 In RepeatBackgroundX.cs, in Start(), the repeatWidth should be dividing the X size (width) of the box collider by 2, not the Y size (height)

```
repeatWidth = GetComponent<BoxCollider>().size.y x / 2;
```

Bonus Challenge Solution

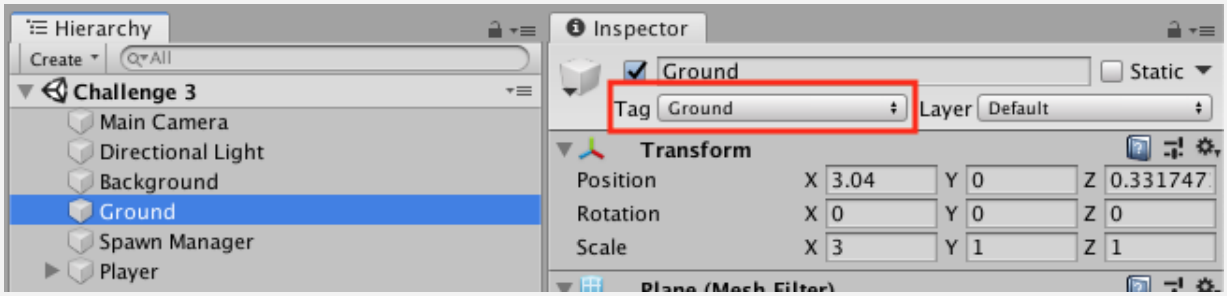
X1 In PlayerControllerX.cs create a boolean to track whether the player is low enough to float upwards, then in Update(), set it to ***false*** if the player is above a certain Y value and, else, set it to ***true***

```
public bool isLowEnough;

void Update() {
  if (transform.position.y > 13) {
    isLowEnough = false;
  } else {
    isLowEnough = true;
  }
}
```

X2 In the if-statement testing for the player pressing spacebar, add a condition testing that the ***isLowEnough*** boolean is true:

```
if (Input.GetKey(KeyCode.Space) && isLowEnough && !gameOver) {
  playerRb.AddForce(Vector3.up * floatForce
}
```

Y1 Add a tag to the Ground object so that you can easily test for a collision with it



Y2 In PlayerControllerX.cs, in the OnCollisionEnter method, add a third else-if checking if the balloon collided with the ground during the game, and if so, to add an impulse force upwards

```
private void OnCollisionEnter(Collision other) {
...
} else if (other.gameObject.CompareTag("Ground") && !gameOver)
{
  playerRb.AddForce(Vector3.up * 10, ForceMode.Impulse);
}
```

Y3 To add a sound effect, declare a new AudioClip variable and assign it in the inspector, then use the PlayOneShot method when the player collides with the ground.

```
public AudioClip moneySound;
public AudioClip explodeSound;
public AudioClip bounceSound;

private void OnCollisionEnter(Collision other) {
...
} else if (other.gameObject.CompareTag("Ground") && !gameOver)
{
  rigidBody.AddForce(Vector3.up * 10, ForceMode.Impulse);
  playerAudio.PlayOneShot(bounceSound, 1.5f);
}
```

Unit 3 Lab

Player Control

Steps:

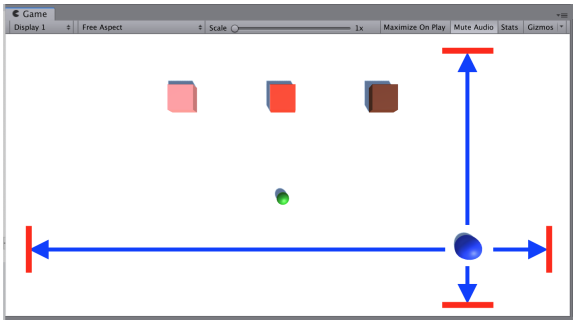
Step 1: Create PlayerController and plan your code

Step 2: Basic movement from user input

Step 3: Constrain the Player's movement

Step 4: Code Cleanup and Export Backup

Example of progress by end of lab



Length: 60 minutes

Overview: In this lesson, you program the player's basic movement, including the code that limits that movement. Since there are a lot of different ways a player can move, depending on the type of project you're working on, you will not be given step-by-step instructions on how to do it. In order to do this, you will need to do research, reference other code, and problem-solve when things go wrong.

Project Outcome: The player will be able to move around based on user input, but *not* be able to move where they shouldn't.

Learning Objectives: By the end of this lab, you will be able to:
- Program the type of player movement you want based on user input
- Restrict player movement in the manner that is appropriate, depending on the needs of the project
- Troubleshoot issues and find workarounds related to player movement

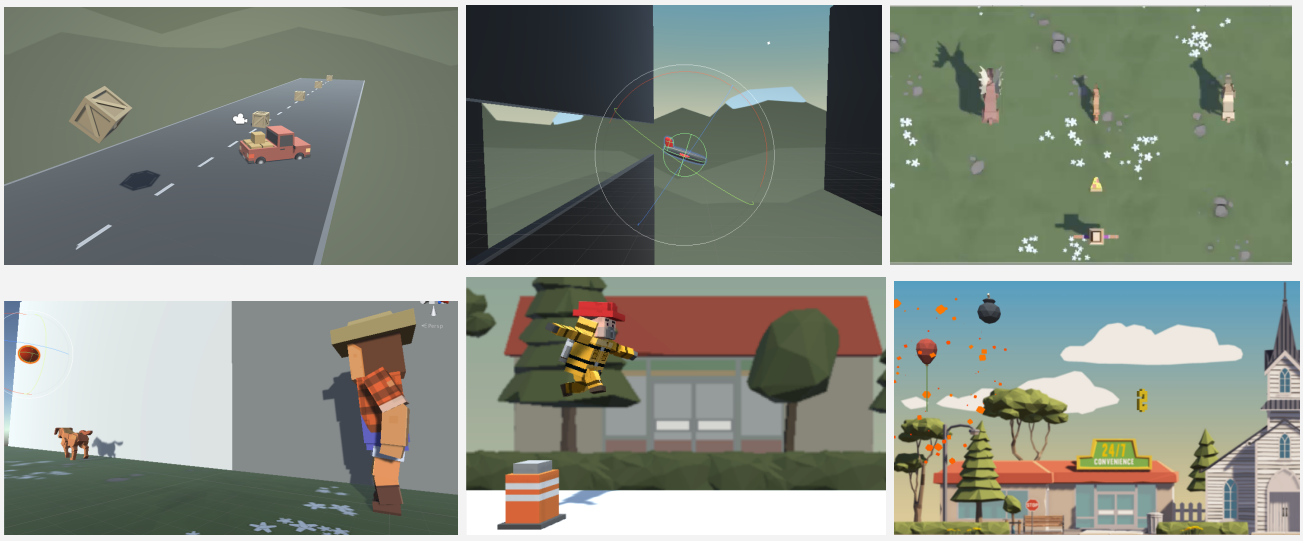
Step 1: Create PlayerController and plan your code

Regardless of what type of movement your player has, it'll definitely need a PlayerController script

1. Select your Player and add a **Rigidbody** component (with or without gravity enabled)
2. In your Assets folder, create a new "Scripts" folder
3. Inside the new "Scripts" folder, create a new "PlayerController" C# script
4. **Attach** it to the player, then **open** it
5. Determine what type of programming will be required for your Player

- **Tip:** Rigidbody is usually helpful - also detect triggers
- **Tip:** Think about all the movement we've done so far:
  - Prototype 1 - forward/back and rotate based on up/down and left/right arrows
  - Challenge 1 - plane moving constantly, rotated direction based on arrows
  - Prototype 2 - side-to-side movement and spacebar to fire a projectile
  - Challenge 2 - No player movement, but projectile launch on spacebar
  - Prototype 3 - background move, and player jumps on spacebar press
  - Challenge 3 - background move and player floats up when spacebar down
- **Don't worry:** If you want your player to move like the ball in Prototype 4, just use basic alternative for now

References to the various types of movement programmed up to this point in the course



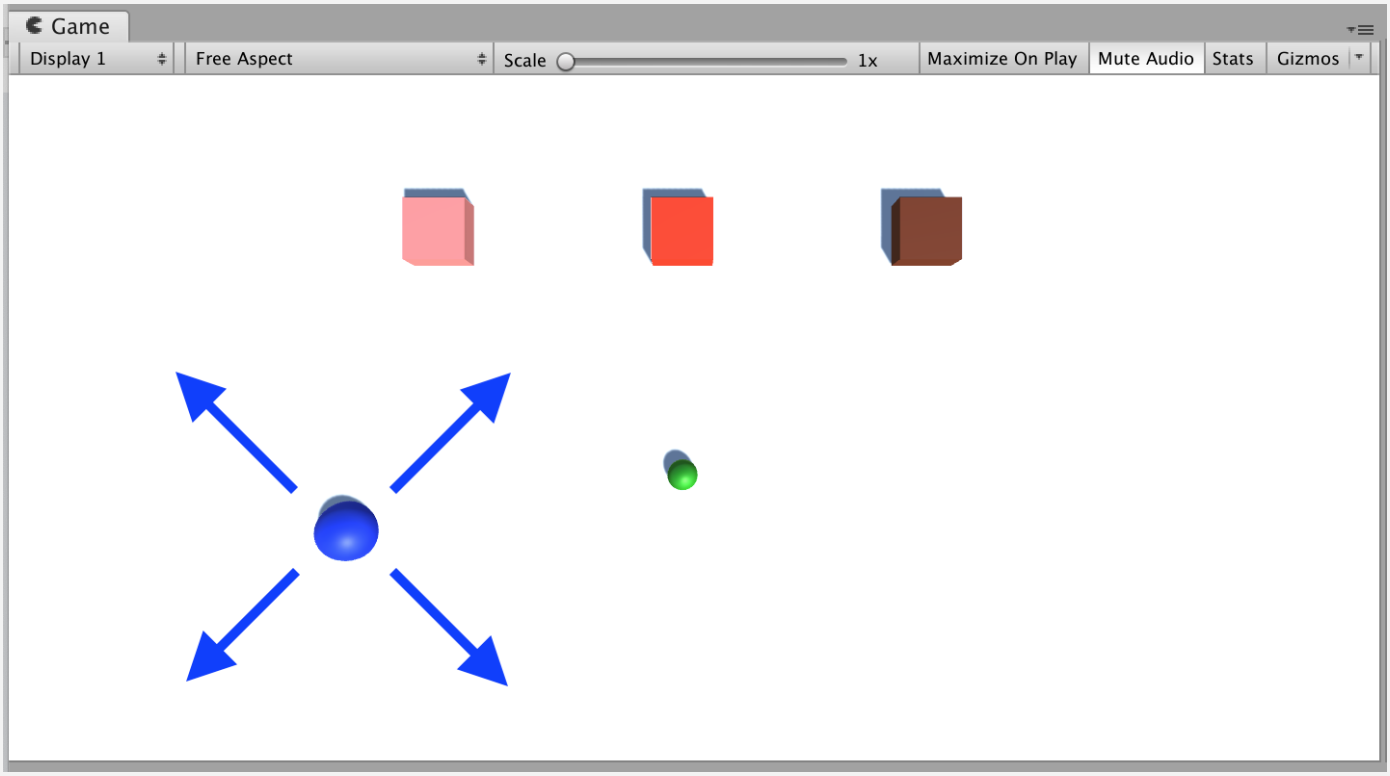
By the end of this step, you should have a new Script open and a solid plan for what will go in it.

Step 2: Basic movement from user input

The first thing we'll program is the player's very basic movement based on user input

1. Declare a new ***private float speed*** variable
2. If using physics, declare a new ***Rigidbody playerRb variable*** for it and initialize it in Start()
3. If using arrow keys, declare new ***verticalInput*** and/or ***horizontalInput*** variables
4. If basing your movement off a key press, create the **if-statement** to test for the **KeyCode**
5. Use either the ***Translate*** method or ***AddForce*** method (if using physics) to move your character

- **Explanation:** Rigidbody movement with AddForce is different than Translate - looks more similar to real world movement with force being applied
- **Don't worry:** If your player is colliding with the ground or other objects in weird ways - we'll fix that soon
- **Tip:** You can look through your old code for references to how you did things



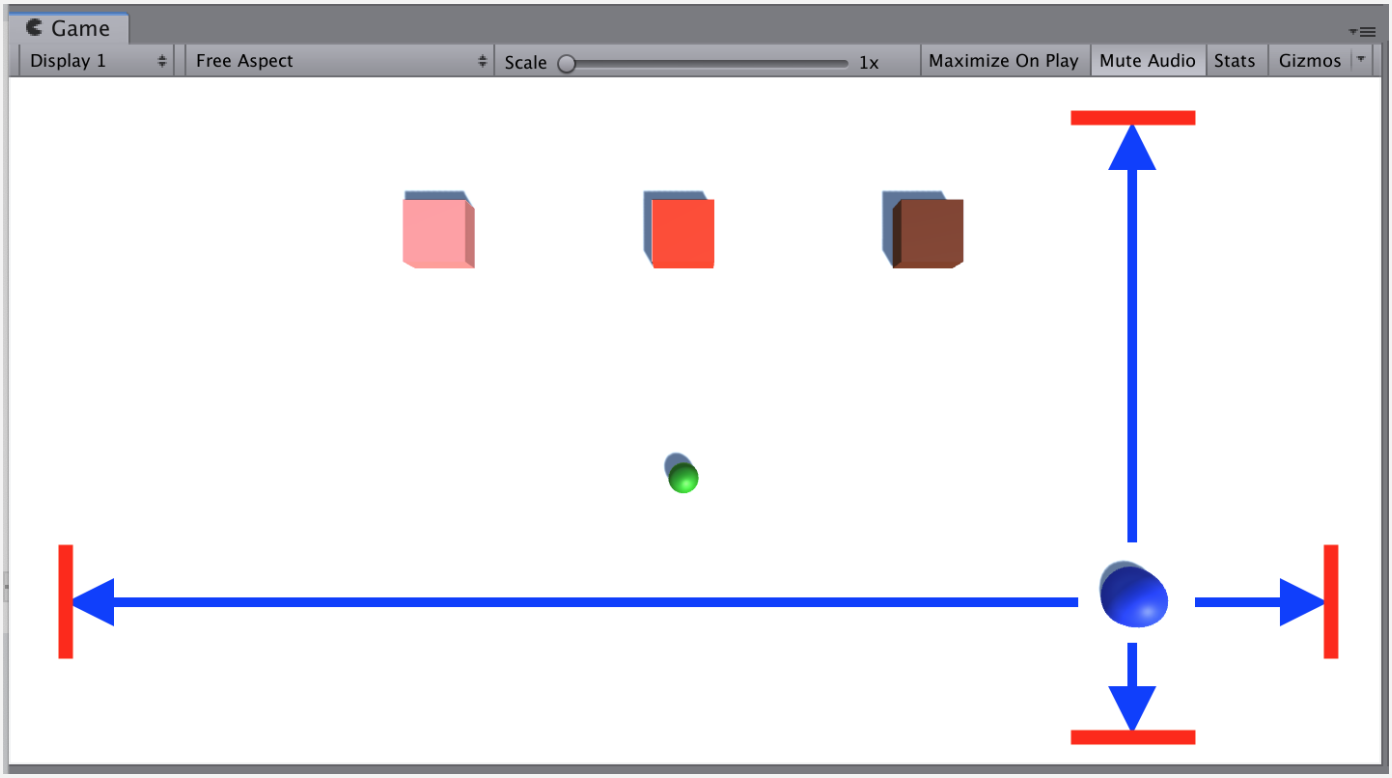
By the end of this step, the player should be able to move the way that you want based on user input.

Step 3: Constrain the Player's movement

No matter what kind of movement your player has, it needs to be limited for gameplay

1. If your player is colliding with objects they shouldn't (including the ground), check the "**Is trigger**" box in the Collider component
2. If your player's position or rotation should be constrained, expand the **constraints** in the Rigidbody component and constrain certain axes
3. If your Player can go **off the screen**, write an **if-statement** checking and resetting the position
4. If the Player can double-jump or fly off-screen, create a **boolean variable** that limits the user's ability to do so
5. If your player should be constrained by physical barriers along the outside of the play area, create more primitive **Planes** or **Cubes** and scale them to form walls

- **Tip:** Check the Global/Local checkbox above scene view to see the rotation of the player
- **Tip:** Look back at Prototype 2 for the if-then statement to keep the player on screen
- **Tip:** Look back at Prototype 3 and Challenge 3 for examples of booleans to prevent double-jumping or going too high



By the end of this step, the player's movement should be constrained in such a way that makes your game playable.

Step 4: Code Cleanup and Export Backup

Now that we have the basic functionality working, let's clean up our code and make a backup.

1. Create new **Empty** game objects and nest objects inside them to **organize** your hierarchy
2. Clean up your Update methods by moving the blocks of code into new void functions (e.g. "MovePlayer()" or "ConstrainPlayerPosition()")
3. Add comments to make your code more readable
4. **Test** to make sure everything still works, then **save** your scene
5. Right-click on your *Assets folder* > ***Export Package*** then save a new version in your **Backups** folder

- **Tip:** You always want to keep your Update() functions clean or they can become overwhelming - it should be easy to see what actions are happening every frame

```
// Move the player left/right and up/down based on arrow keys
void MovePlayer() {
  ...
}

// Prevent the player from leaving the screen top/bottom
void ConstrainPlayerPosition() {
  ...
}
```

By the end of this step, your code should be commented, organized, and backed up.

Lesson Recap

New Progress	• Player can move based on user input • Player movement is constrained to suit the requirements of the game
New Concepts and Skills	• Program in C# independently • Troubleshoot issues independently

unity

Quiz Unit 3

QUESTION	CHOICES
1 You are trying to STOP spawning enemies when the player has died and have created the two scripts below to do that. However, there is an error on the underlined code, “isAlive” in the EnemySpawner script. What is causing that error?	a. The “p” should be capitalized in “playerController.isAlive” b. The “bool” in the PlayerController class needs a “public” access modifier c. The if-statement cannot be in the Update method d. “isAlive” must start with a capital “I” (“IsAlive”)

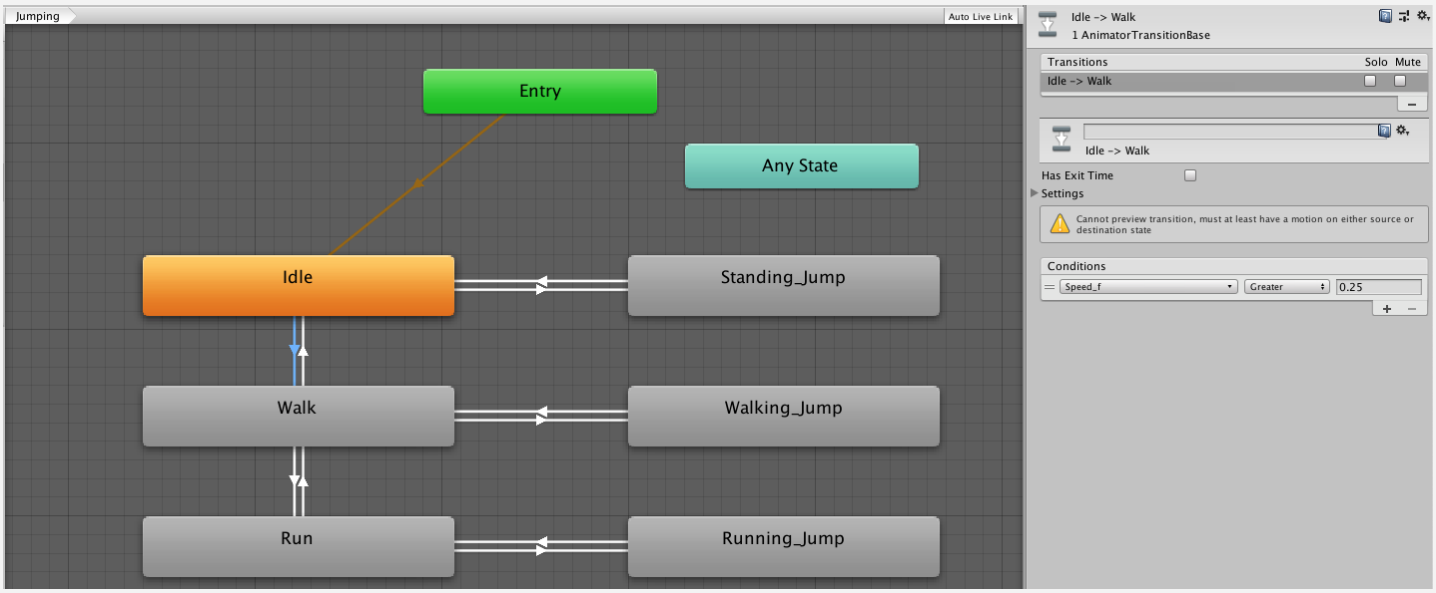
```
public class PlayerController : MonoBehaviour {
  bool isAlive;
  ...
}

public class EnemySpawner : MonoBehaviour {
  void Start() {
    playerController = GameObject.Find("Player").GetComponent<PlayerController>();
  }
  void Update() {
    if (playerController.isAlive == false) {
       StopSpawning();
    }
  }
}
```

2 Match the following animation methods with its set of parameters	1. anim.SetBool(_____);	A. “Celebrate”
	2. anim.SetTrigger(_____);	B. “Alive”, true
	3. anim.SetInt(_____);	C. “ThrowType”, 2

a. 1A, 2B, 3C
b. 1A, 2C, 3B
c. 1B, 2A, 3C
d. 1C, 2A, 3B

3 Given the animation controller / state machine below, which code will make the character transition from the “Idle” state to the “Walk” state?

a. setFloat(“Speed_f”, 0.3f);
b. setInt(“Speed_f”, 1);
c. setTrigger(“Speed_f”);
d. setFloat(“Speed_f”, 0.1f);



4 Which of these is the correct way to get a reference to an AudioSource component on a GameObject?

```
A. audio = GetComponent<AudioSource>();
B. audio = GetComponent(AudioSource)<>;
C. audio = AudioSource.GetComponent<>();
D. audio = GetComponent.Audio<Source>;
```

a. Line A
b. Line B
c. Line C
d. Line D

5 When you run a project with the code below, you get the following error: “NullReferenceException: Object reference not set to an instance of an object.” What is most likely the problem?

```
public class Enemy : MonoBehaviour {
  void Start() {
    player = GameObject.Find("Player");
  }
  void OnTriggerEnter(Collider other) {
    if (player.transform.position.z > 10) {
       Destroy(other.gameObject);
    }
  }
}
```

a. The Player object does not have a collider
b. The Enemy object does not have a Rigidbody component
c. The “Start” method should actually be “Update”
d. There is no object named “Player” in the scene

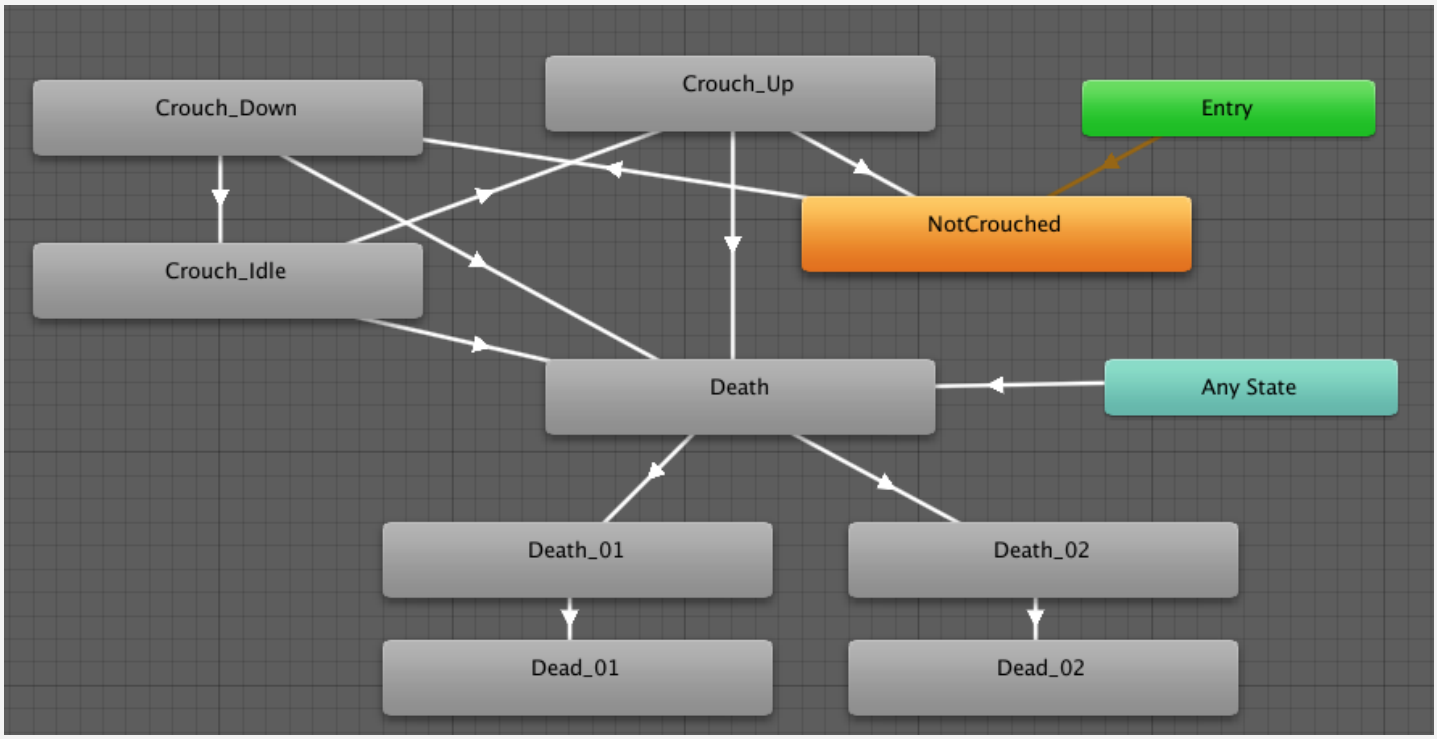
6 Which of the following conditions properly tests that the game is NOT over and the player IS on the ground

```
A. if (gameOver == false AND isOnGround)
B. if (gameOver && isOnGround == true)
C. if (gameOver != true && isOnGround)
D. if (gameOver != false && isOnGround == true)
```

a. Line A
b. Line B
c. Line C
d. Line D

7 By default, what will be the first state used by this Animation Controller?

a. “Any State”
b. “NotCrouched”
c. “Death”
d. “Crouch_Up”



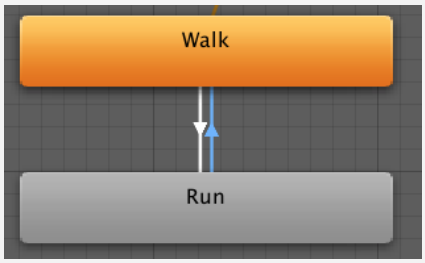
8 Which of the following variable declarations observes Unity’s standard naming conventions (especially as it relates to capitalization)?

```
1. private Animator anim;
2. private player Player;
3. Float JumpForce = 10.0f;
4. bool gameOver = True;
5. private Vector3 startPos;
6. Public gameObject ObstaclePrefab;
```

a. 2 and 4
b. 3 and 6
c. 4 and 5
d. 1 and 5

9 Which of the following is most likely the condition for the transition between “Run” and “Walk” shown below?

a. Jump_b is true
b. Speed_f is Less than 0.5
c. Speed_f is Greater than 0.5
d. Animation_int is Less than 10



A.

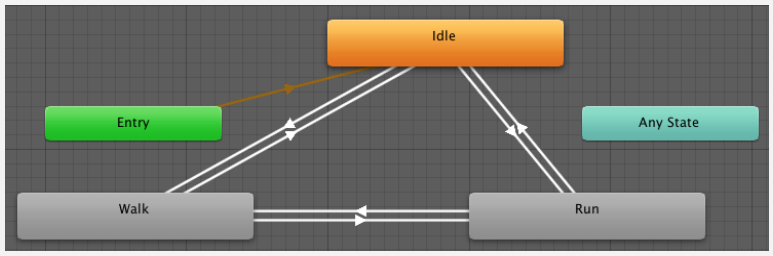

B.

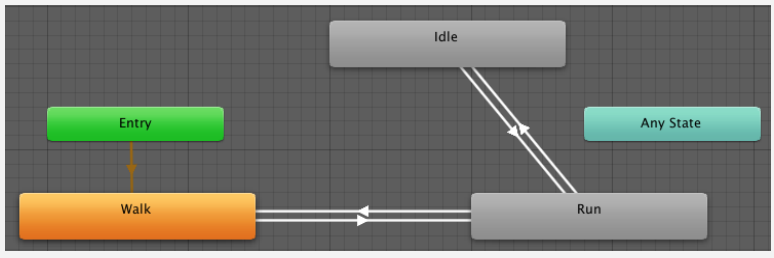

C.

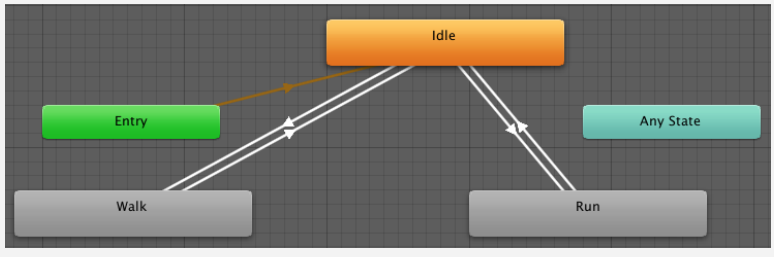

D.


10 Which of the following do you think makes the most sense for a simple movement state machine?

a. Image A
b. Image B
c. Image C

A.


B.


C.


Quiz Answer Key

#	ANSWER	EXPLANATION
1	B	In order to access a variable from another class, that variable needs to be “public”. By default, if there is no access modifier, variables are private and cannot be accessed by another class
2	C	SetInt would require an integer parameter, SetBool would require a boolean parameter, and SetTrigger only requires the trigger name/id
3	A	You can see in the inspector that the condition for this transition is that “Speed_f is greater than 0.25”. You can tell it’s a float because it uses decimal points and it must be higher than 0.25.
4	A	“GetComponent<AudioSource>();” is the correct way to use the GetComponent method
5	D	If you try to “Find” an object that is not in the scene, you will get a “NullReferenceException” error.
6	C	!= means “does not equal to”, so “gameOver != true” is testing that the game is *not* over. If you just use the boolean’s name like “isOnGround,” this tests whether that boolean is true. The syntax for testing two conditions is “&&”.
7	B	The default starting state is the one that the “Entry” state connects to.
8	D	1. private Animator anim; - this is correct 2. private player Player; - should be “private Player player” 3. Float JumpForce = 10.0f; - should be “float jumpForce = 10.0f” 4. bool gameOver = True; - should be “true” (lowercase “t”) 5. private Vector3 startPos; - this is correct 6. Public gameObject ObstaclePrefab; - should be “public GameObject obstaclePrefab”
9	B	If you are transitioning from Running to Walking, that most likely is a result of reducing speed, so checking if “Speed_f is *less* than 0.5” is most likely
10	A	You should start with “Idle” as the default state, then be able to transition between any of the states (Idling, Walking, Running). There should definitely be a transition between Walk and Run.

unity

Bonus Features 3 - Share your Work

Steps:

Length: 60 minutes

Overview: In this tutorial, you can go way above and beyond what you learned in this Unit and share what you've made with your fellow creators.

There are four bonus features presented in this tutorial marked as Easy, Medium, Hard, and Expert. You can attempt any number of these, put your own spin on them, and then share your work!

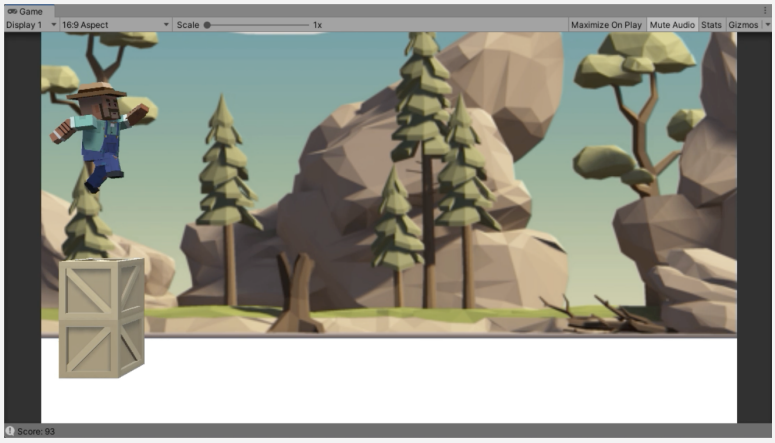
This tutorial is entirely optional, but highly recommended for anyone wishing to take their skills to a new level.

Step 1: Overview

This tutorial outlines four potential bonus features for the Run and Jump Prototype at varying levels of difficulty:

- **Easy**: Randomize obstacles
- **Medium**: Double jump
- **Hard**: Dash ability and score
- **Expert**: Game start animation

Here's what the prototype could look like if you complete all four features:



The Easy and Medium features can probably be completed entirely with skills from this course, but the Hard and Expert features will require some additional research.

Since this is optional, you can attempt none of them, all of them, or any combination in between. You can come up with your own original bonus features as well!

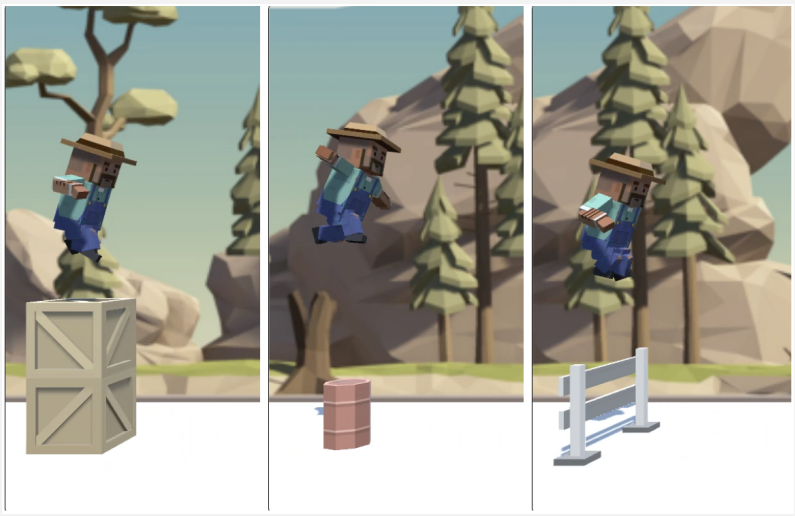
Then, at the end of this tutorial, there is an opportunity to share your work.

We highly recommend that you attempt these using relentless Googling and troubleshooting, but if you do get completely stuck, there are hints and step-by-step solutions available below.

Good luck!

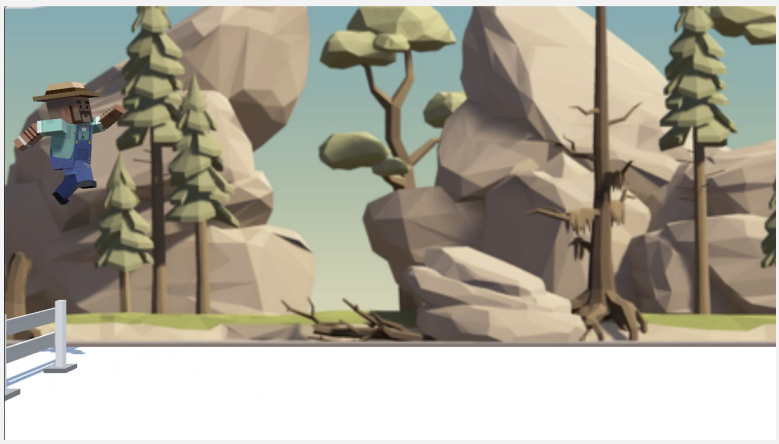
Step 2: Easy: Randomize obstacles

Randomly select from a variety of obstacles to spawn.
You could even have piles of obstacles instead of single ones, forcing the player to pay closer attention.



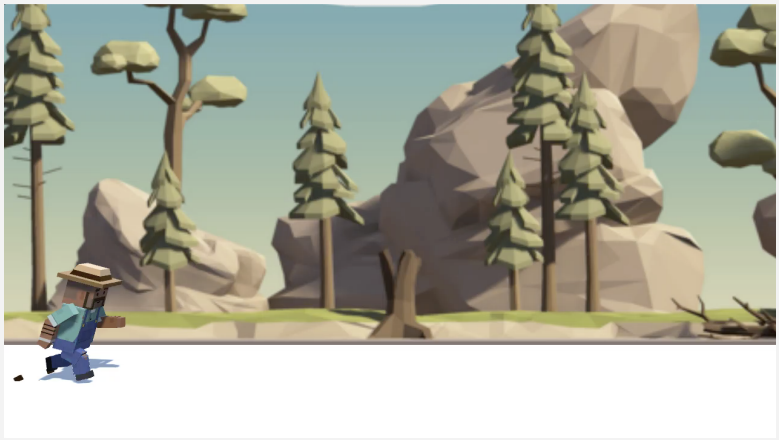
Step 3: Medium: Double jump

Program a double-jump, so the player can jump one additional time once already in the air. Along with this, you could create a new extra tall obstacle that requires a double-jump (maybe two obstacles stacked on top of each other).
This adds a completely new gameplay mechanic. And who doesn't love a double-jump?



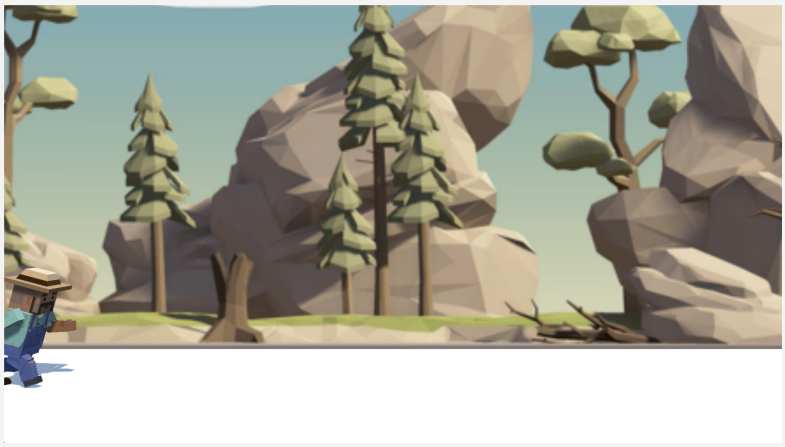
Step 5: Hard: Dash ability and score

Add a “dash” / “super speed” ability where, if the player is holding a certain key, the player runs faster through the world, matched by a faster running animation. Use Debug.Log to track the player’s increasing score from 0, which should increase twice as fast during “dash” mode, and then stop counting when the game is over, reflecting the player’s score.
This adds a completely new strategic element to the game, where players might implement different tactics to maximize their score.



Step 6: Expert: Game start animation

Rather than your player starting off running in place with the background moving, have the player walk into frame from the left, then begin running in place alongside the moving background.
This gives the player a moment to collect themselves rather than being thrown straight into gameplay.



Step 7: Hints and solution walkthrough

Hints:

- Easy: Randomize obstacles
    - Convert the prefab holder inside the SpawnManager to an array.
- Medium: Double jump
    - Try using a boolean to limit the player to double-jumping once
- Hard: Dash ability and score
    - Try using a boolean to determine whether the player is running fast
- Expert: Game start animation
    - Try lerping the players position.

Solution walkthrough
If you are really stuck, download the step-by-step solution walkthrough.
Note that there are likely many ways to implement these features - this is only one suggestion.

Step 8: Share your work

Have you implemented any of these bonus features? Have you added any new, unique features? Have you applied these new features to another project?

We would love to see what you've created!

Please take a screenshot of your project or do a screen-recording walking us through it, then post it here to share what you've made.

We highly recommend that you comment on at least one other creator's submission. What do you like about the project? What would be a cool new feature they might consider adding?