



0.1 Install Unity Software

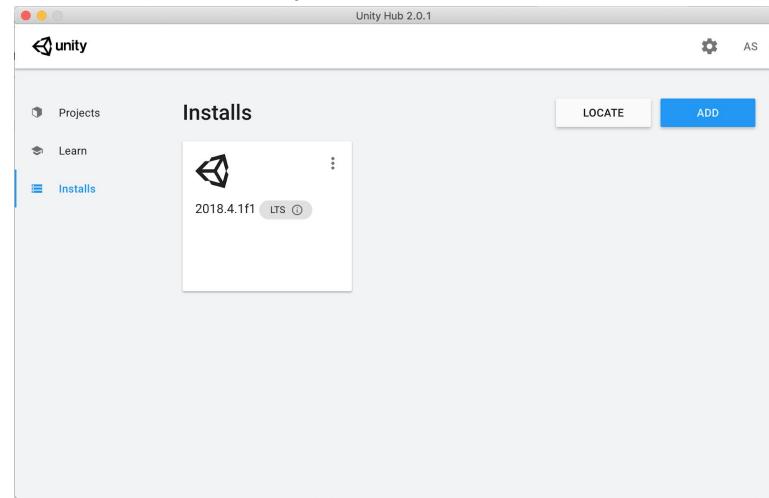
Steps:

Step 1: Download and install Unity Hub

Step 2: Install a new version of Unity

Step 3: Sign in or create a new Unity ID

Example of progress by end of lesson



Length: 20 minutes

Overview: If you do not already have Unity installed on your computer, the first thing you need to do before you get started on the course is install it. In order to do so, if you don't have one already, you will need to create a Unity ID. When you install the software, you will install Unity Hub, which allows you to manage your installations and projects, the Unity engine itself, and Visual studio, the Integrated Development Environment (or IDE) you will use to code in C#.

Project Outcome: Unity Hub, the Unity Editor, and Visual Studio will all be installed on your computer.

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

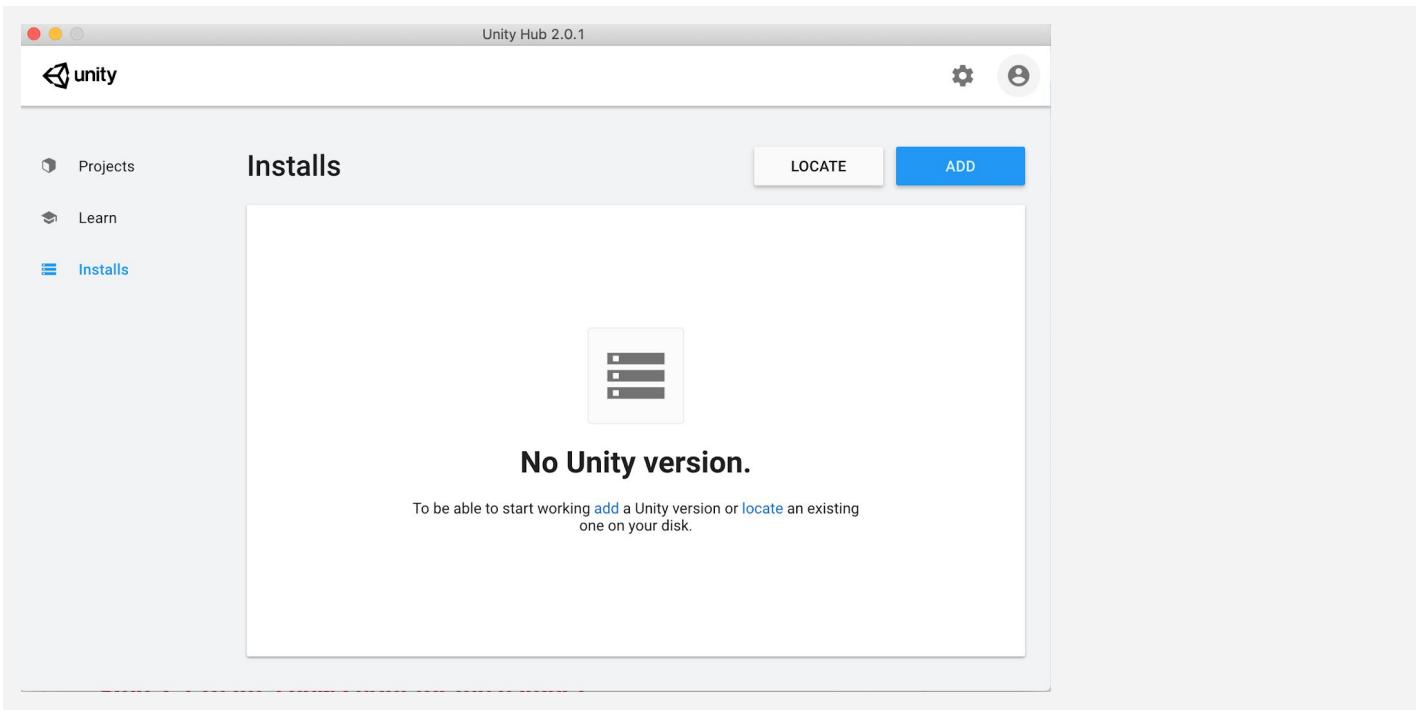
- Use Unity Hub to install and manage versions of Unity on your computer
- Create a new Unity ID to be able to access of all Unity's services

Step 1: Download and install Unity Hub

In order to most effectively download, install, and manage the versions of Unity on our computer, we will use something called Unity Hub.

1. In a new tab, either Google “Download Unity Hub” or go to <https://unity3d.com/get-unity/download>, then click to **Download Unity Hub**
2. From your Downloads folder, **double-click** on the **Unity Hub Setup** file to begin the installation
3. Agree to Unity Terms of Service and follow the instructions to install Unity Hub
4. **Open** Unity Hub for the first time and click on the **Projects, Learn, and Installs** tabs

- **Warning:** If you already have a version of Unity that is version 2018+, you do not need to complete this lesson
- **Warning:** Don’t download Unity directly - download *Unity Hub*
- **Warning:** Will be different on a Mac vs a PC - on a Mac, you just have to drag the icon into the Applications folder, then open it from there
- **Don’t worry:** There might be old projects or other versions in the list

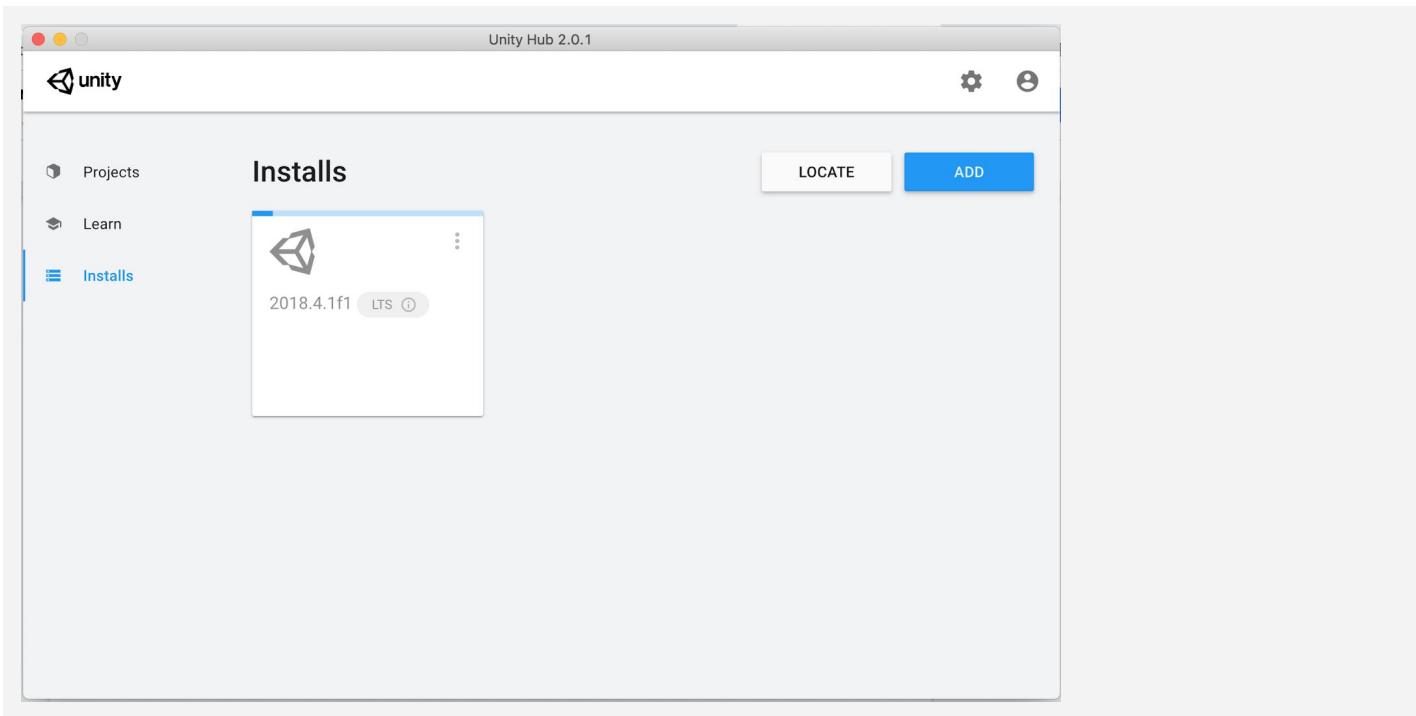


Step 2: Install a new version of Unity

Now that *Unity Hub* is installed, we need to actually *install* a new version of *Unity* and our code editor, *Visual Studio*

1. In the **Installs** tab click to **Add** a new Unity Version
2. Choose either **2018.4.1f1** (if you would like your version of Unity to look exactly the same as the videos) or anything **higher** than that
3. Choose to install **Visual Studio** (for Mac or PC)
4. Accept any necessary terms and conditions and begin installation

- **Don't worry:** This may take a very long time, depending on the speed of your computer and internet connection as it has to first *download*, then *install* both *Unity* and *Visual Studio*
- **Warning:** You will likely be asked to provide your computer's admin password
- **New Concept:** LTS stands for Long-term support, which means *Unity* will officially support it and keep it up-to-date for 2 years
- **Tip:** You can continue onto next step of creating a *Unity ID* while it is installing

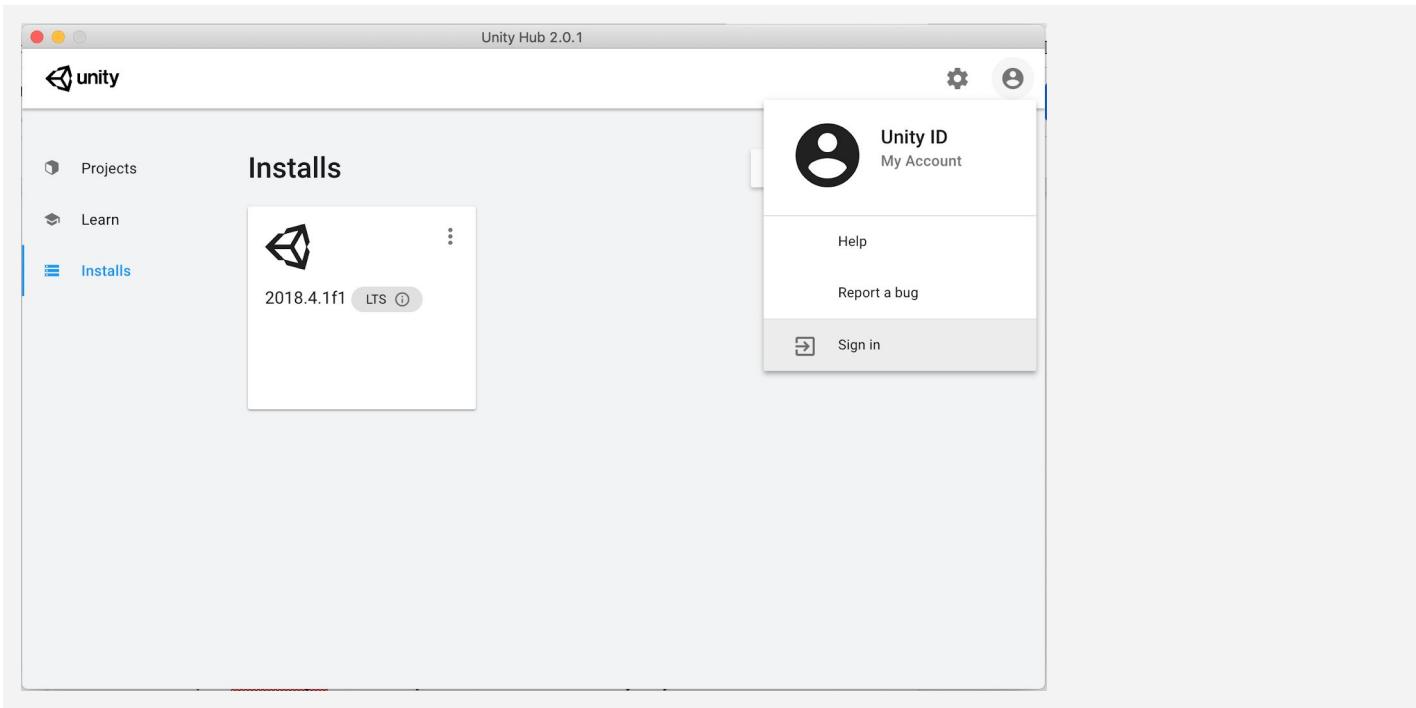


Step 3: Sign in or create a new Unity ID

In order to access a lot of important Unity services, including the Unity Asset Store, we need to be signed in with a Unity ID

1. From the **Account** menu in Unity Hub, click to **Sign in**
2. If you already have an account, **sign in** - otherwise, you can sign in quickly through **Google** or **Facebook** or Create a **New Unity ID**

- **New Concept:** What is a Unity ID?
- **Warning:** If you create a new Unity ID, you will be asked to complete a questionnaire



Lesson Recap

New Progress

- Unity Hub, Unity Editor 2018+, and Visual Studio installed
- Signed into Unity Hub

New Concepts and Skills

- Unity Hub and its features
- Editor versions, including LTS releases
- Visual Studio
- Unity IDs

Next Lesson

- We will actually create a new project and open the Unity Editor to start creating



1.1 Start your 3D Engines

Steps:

Step 1: Make a course folder and new project

Step 2: Import assets and open Prototype 1

Step 3: Add your vehicle to the scene

Step 4: Add an obstacle and reposition it

Step 5: Locate your camera and run the game

Step 6: Move the camera behind the vehicle

Step 7: Customize the interface layout

Example of project by end of lesson



Length: 70 minutes

Overview: In this lesson, you will create your very first game project in Unity Hub. You will choose and position a vehicle for the player to drive and an obstacle for them to hit or avoid. You will also set up a camera for the player to see through, giving them a perfect view of the scene. Throughout this process, you will learn to navigate the Unity Editor and grow comfortable moving around in 3D Space. Lastly, you will customize your own window layout for the Unity Editor.

Project Outcome: You will have a vehicle and obstacle positioned on the road and the camera set up perfectly behind the vehicle. You will also have a new custom Unity layout, perfectly optimized for editing.

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

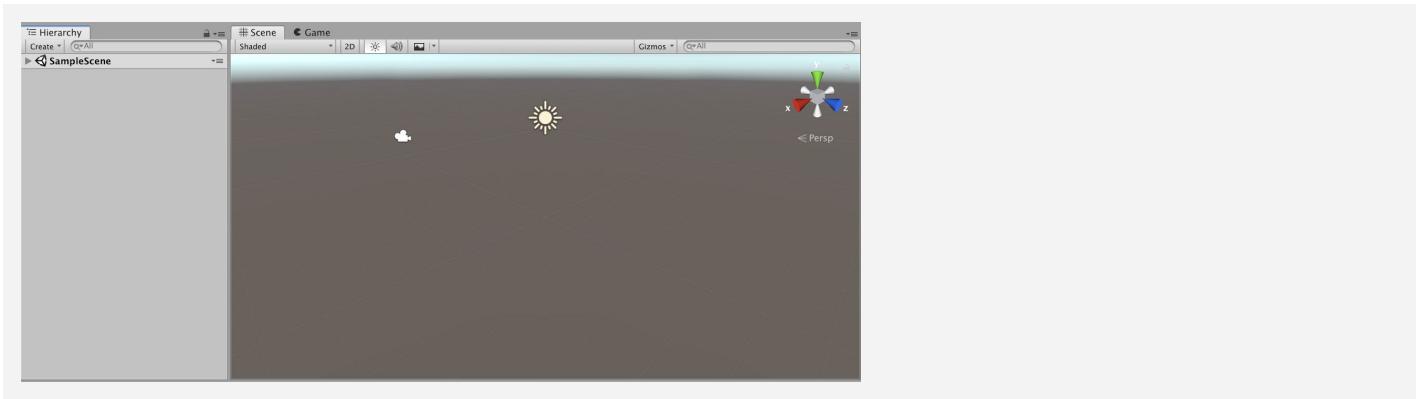
- Create a new project through Unity Hub
- Navigate 3D space and the Unity Editor comfortably
- Add and manipulate objects in the scene to position them where you want
- Position a camera in an ideal spot for your game
- Control the layout of Unity Editor to suit your needs

Step 1: Make a course folder and new project

The first thing we need to do is create a folder that will hold all of our course projects, then create a new Unity project inside it for Prototype 1.

1. On your **desktop** (or somewhere else you will remember),
Right-click > create **New Folder**, then name it “Create with Code”
2. Open **Unity Hub** and click **New**
3. Name the project “Prototype 1”, select the correct **version of Unity**, set the location to the new “**Create with Code**” folder, and select the **3D** template
4. Click **Create Project**, then wait for Unity to open

- **Don't worry:** Unity might take a while to open, so just give it some time

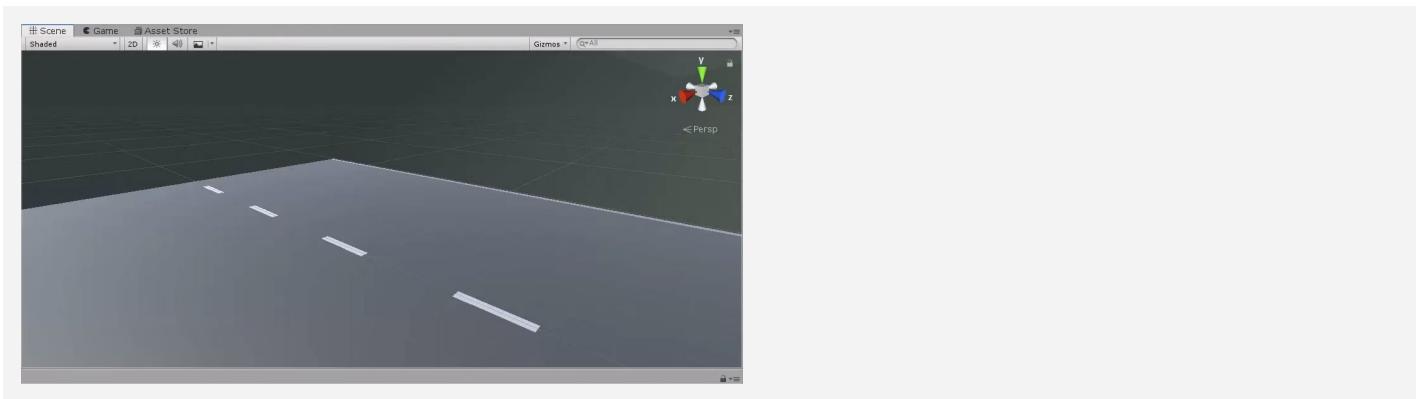


Step 2: Import assets and open Prototype 1

Now that we have an empty project open, we need to import the assets for Prototype 1 and open the scene

1. Click on one of the links to access the Prototype 1 starter files, then **download** and **import** them into Unity
2. In the **Project** window, in **Assets > Scenes** > double-click on the **Prototype 1 scene** to open it
3. Delete the **Sample Scene** without saving
4. **Right-click + drag** to look around at the start of the road

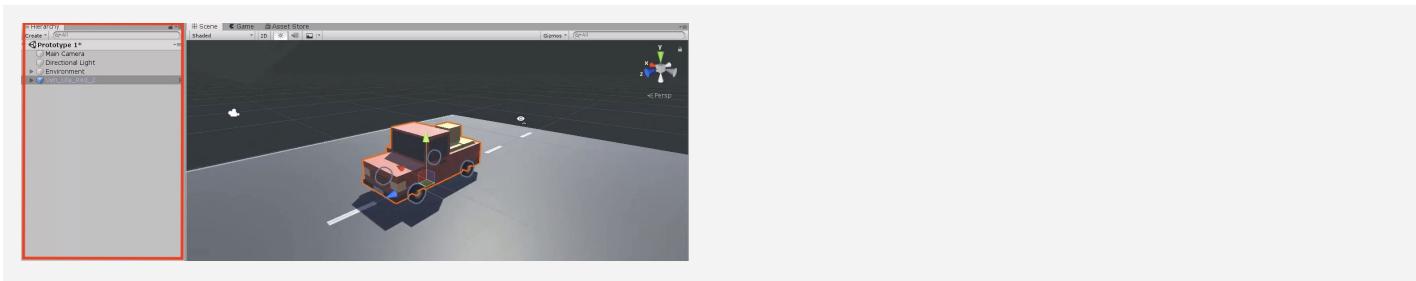
- **Warning:** You're free to look around, but don't try moving yet
- **Warning:** Be careful playing with this interface, don't click on anything else yet
- **New Concept:** Project Window



Step 3: Add your vehicle to the scene

Since we're making a driving simulator, we need to add our own vehicle to the scene.

1. In the **Project Window**, open Assets > Course Library > Vehicles, then drag a vehicle into the **Hierarchy**
2. **Hold right-click + WASD** to fly to the vehicle, then try to rotate around it
3. **Press F** in the Scene view to focus on it, then use the **scroll wheel** to zoom in and out and **hold the scroll wheel** to pan
4. Press F to focus on it, **hold alt+left-click** to rotate around it perfectly
5. If anything goes wrong, press **Ctrl/Cmd+Z** to Undo until it's fixed

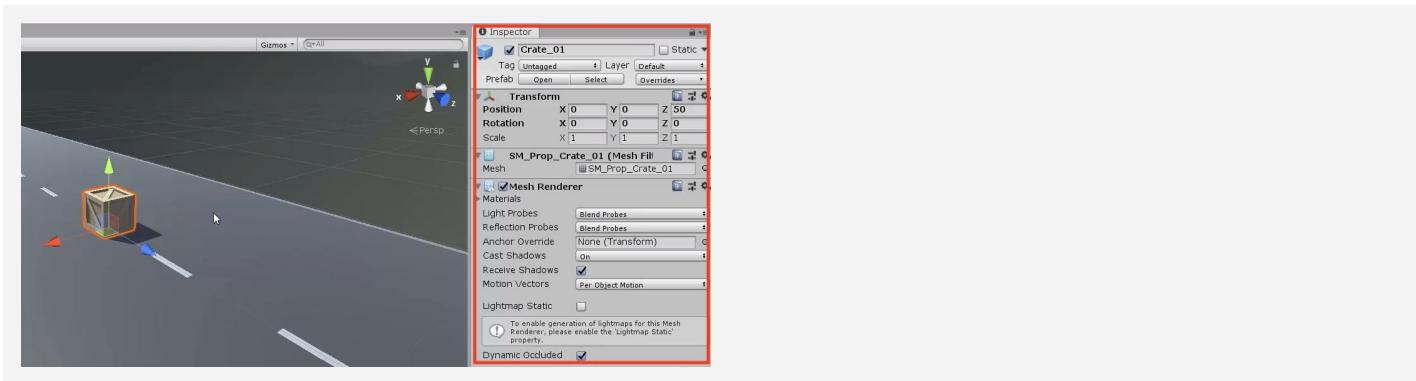


Step 4: Add an obstacle and reposition it

The next thing our game needs is an obstacle! We need to choose one and position it in front of the vehicle.

1. Go to Course Library > Obstacles and drag an obstacle directly into **scene view**
2. In the Inspector for your obstacle, in the top-right of the Transform component, click the **Gear Icon** > **Reset Position**
3. In the **Inspector**, change the **XYZ Location** to **0,0,25**
4. In the hierarchy, **Right-click** > **Rename** your two objects as "Vehicle" and "Obstacle"

- **New: Hierarchy**
- **New: Undo** (Cmd/Ctrl + Z) and **Redo** (Cmd+Shift+Z / Ctrl+Y)
- **Warning:** Mouse needs to be in scene view for F/focus to work
- **New Technique:** Scroll Wheel for Zoom and Pan



Step 5: Locate your camera and run the game

Now that we've set up our vehicle and obstacle, let's try running the game and looking through the camera.

1. Select the **Camera** in the hierarchy, then **press F** to focus on it
2. Press the **Play button** to run your Game, then press Play again to **stop** it

- **New Concept:** Game View vs Scene View
- **New Technique:** Stop/Play (Cmd/Ctrl + P)

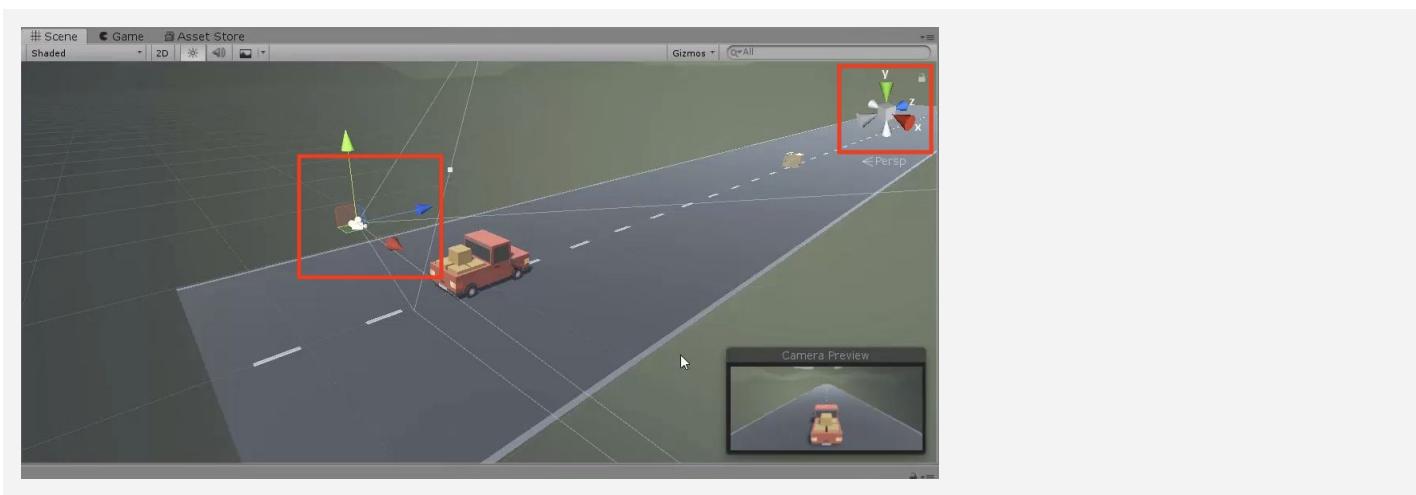


Step 6: Move the camera behind the vehicle

In order for the player to properly view our game, we should position and angle the camera in a good spot behind the vehicle

1. Use the **Move** and **Rotate tools** to move the camera behind the vehicle looking down on it
2. **Hold Ctrl/Cmd** to move the camera by whole units

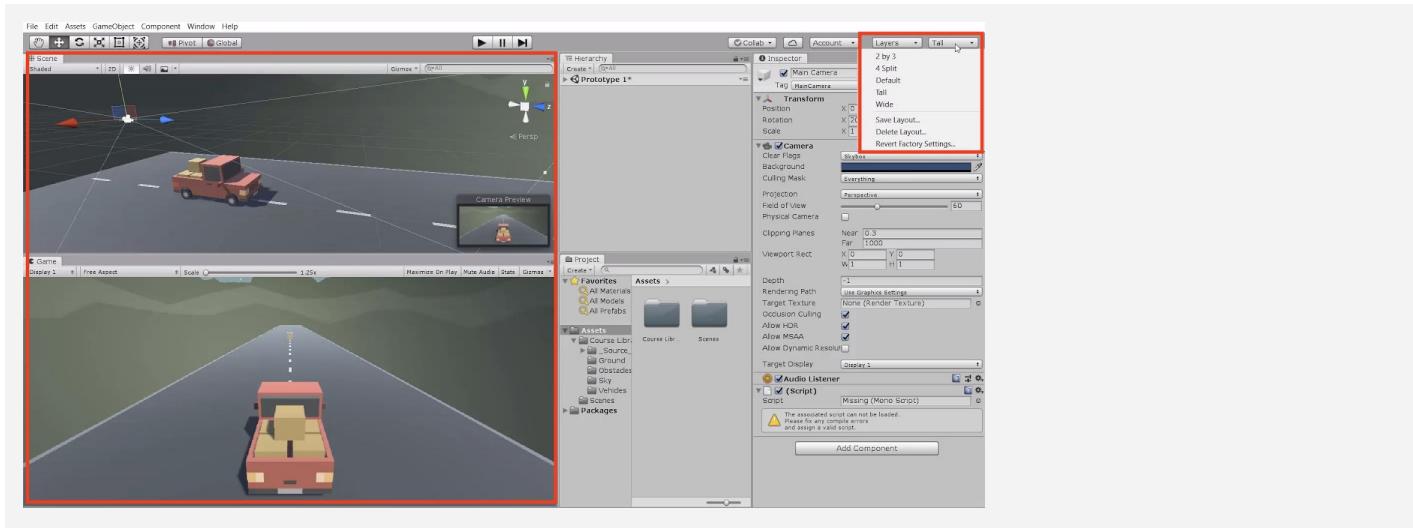
- **New Technique:** Snapping (Cmd/Ctrl + Drag)
- **New Concept:** Rotation on the XYZ Axes



Step 7: Customize the interface layout

Last but not least, we need to customize the Unity Editor layout so that it's perfect for editing our project.

1. In the top-right corner, change the layout from "Default" to "Tall", **- New Concept: Layouts**
2. Move **Game view** beneath Scene view
3. In the **Project** window, click on the little drop-down menu in the top-right and choose "**One-column layout**"
4. In the layout Dropdown, **save a new Layout** and call it "My Layout"



Lesson Recap

New Functionality

- Project set up with assets imported
- Vehicle positioned at the start of the road
- Obstacle positioned in front of the vehicle
- Camera positioned behind vehicle

New Concepts and Skills

- Create a new project
- Import assets
- Add objects to the scene
- Game vs Scene view
- Project, Hierarchy, Inspector windows
- Navigate 3D space
- Move and Rotate tools
- Customize the layout

Next Lesson

- We'll really make this interactive by writing our first line of code in C# to make the vehicle move and have it collide with other objects in the scene

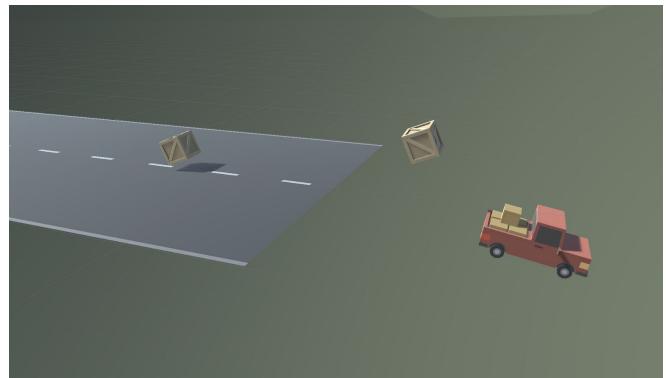


1.2 Pedal to the Metal

Steps:

- Step 1: Create and apply your first script
- Step 2: Add a comment in the Update() method
- Step 3: Give the vehicle a forward motion
- Step 4: Use a Vector3 to move forward
- Step 5: Customize the vehicle's speed
- Step 6: Add RigidBody components to objects
- Step 7: Duplicate and position the obstacles

Example of project by end of lesson



Length: 70 minutes

Overview: In this lesson you will make your driving simulator come alive. First you will write your very first lines of code in C#, changing the vehicle's position and allowing it to move forward. Next you will add physics components to your objects, allowing them to collide with one another. Lastly, you will learn how to duplicate objects in the hierarchy and position them along the road.

Project Outcome: You will have a moving vehicle with its own C# script and a road full of objects, all of which may collide with each other using physics components.

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

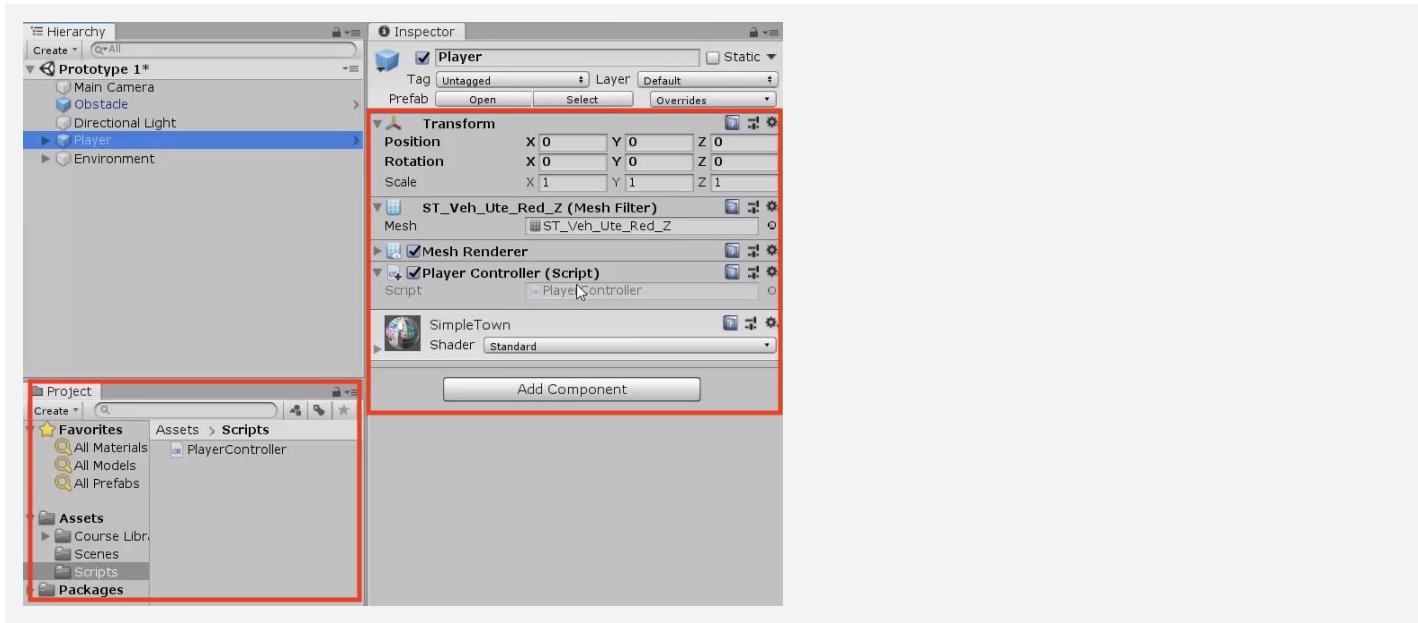
- Create C# scripts and apply them to objects
- Use Visual Studio and a few of its basic features
- Write comments to make your code more readable
- Utilize fundamental C# methods and classes like transform.Translate and Vector3
- Add Rigidbody and Collider components to allow objects to collide realistically
- Duplicate objects in the hierarchy to populate your scene

Step 1: Create and apply your first script

We will start this lesson by creating our very first C# script that will control the vehicle's movement.

1. In the Project window, Right-click > Create > **Folder** named “Scripts”
2. In the “Scripts” folder, Right-click > Create > **C# Script** named “PlayerController”
3. Drag the new script onto the **Vehicle object**
4. Click on the Vehicle object to make sure it was added as a **Component** in the Inspector

- **New Concept:** C# Scripts
- **Warning:** Type the script name as soon as the script is created, since it adds that name to the code. If you want to edit the name, just delete it and make a new script
- **New Concept:** Components



Step 2: Add a comment in the Update() method

In order to make the vehicle move forward, we have to first open our new script and get familiar with the development environment.

1. Double-click on the script to open it in **Visual Studio**
2. In the **Update()** method, add a comment that you will: **// Move the vehicle forward**

- **New:** Start vs Update functions
- **New:** Comments

```
void Update()
{
    // Move the vehicle forward
}
```

Step 3: Give the vehicle a forward motion

Now that we have the comment saying what we *WILL* program - we have to write a *line* of code that will actually move the vehicle forward.

- Under your new comment, type ***transform.tr***, then select **Translate** from the autocomplete menu
- Type **(**, add **0, 0, 1** between the parentheses, and complete the line with a semicolon **(;)**
- Press **Ctrl/Cmd + S** to save your script, then run your game to test it

- **New Function:** `transform.Translate`
- **New Concept:** Parameters
- **Warning:** Don't use decimals yet. Only whole numbers!

```
void Update()
{
    // Move the vehicle forward
    transform.Translate(0, 0, 1);
}
```

Step 4: Use a Vector3 to move forward

We've programmed the vehicle to move along the Z axis, but there's actually a cleaner way to code this.

- Delete the **0, 0, 1** you typed and use auto-complete to replace it with **Vector3.forward**

- **New Concept:** Documentation
- **New Concept:** `Vector3`
- **Warning:** Make sure to save time and use Autocomplete! Start typing and VS Code will display a popup menu with recommended code.

```
void Update()
{
    // Move the vehicle forward
    transform.Translate(0, 0, 1 Vector3.forward);
}
```

Step 5: Customize the vehicle's speed

Right now, the speed of the vehicle is out of control! We need to change the code in order to adjust this.

1. Add * **Time.deltaTime** and run your game
2. Add * **20** and run your game

- **New Concept:** Math symbols in C#
- **New Function:** Time.deltaTime

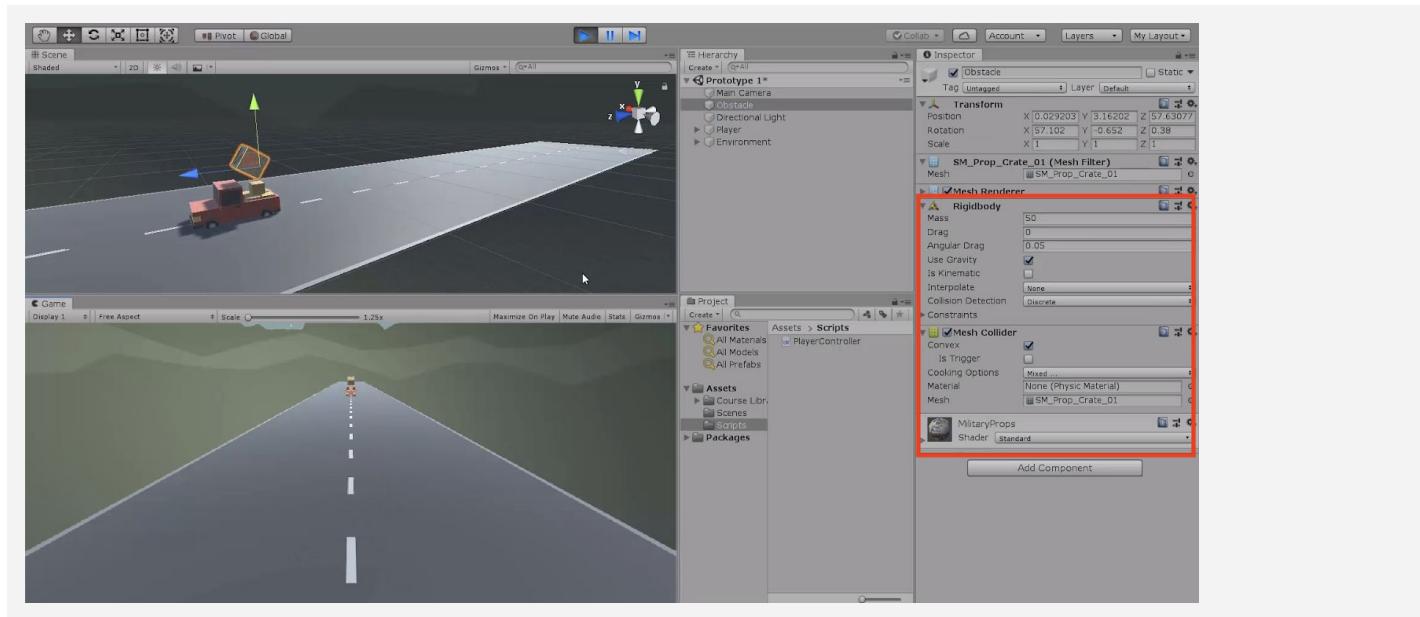
```
void Update()
{
    // Move the vehicle forward
    transform.Translate(Vector3.forward * Time.deltaTime * 20);
}
```

Step 6: Add RigidBody components to objects

Right now, the vehicle goes right through the box! If we want it to be more realistic, we need to add physics.

1. Select the **Vehicle**, then in the hierarchy click **Add Component** and select **RigidBody**
2. Select the **Obstacle**, then in the hierarchy click **Add Component** and select **RigidBody**
3. In the RigidBody component properties, increase the **mass** of vehicle and obstacle to be about what they would be in **kilograms** and test again

- **New Concept:** Rigidbody Component
- **New Concept:** Collider Component
- **Tip:** Adjust the mass of the vehicle and the obstacle, and test the collision results

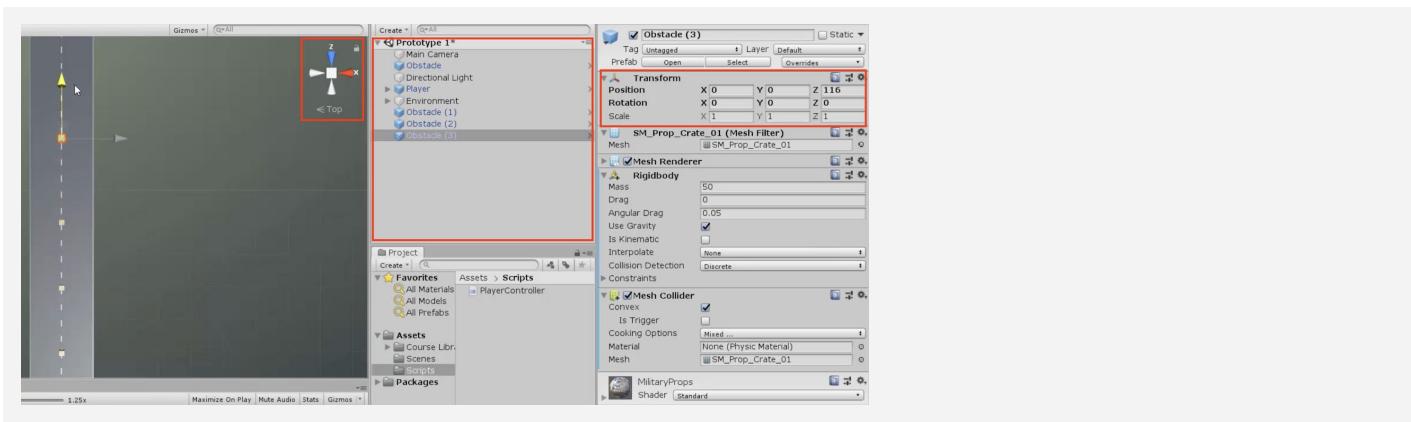


Step 7: Duplicate and position the obstacles

Last but not least, we should duplicate the obstacle and make the road more treacherous for the vehicle.

1. Click and drag your obstacle to the **bottom of the list** in the hierarchy
2. Press **Ctrl/Cmd+D** to duplicate the obstacle and move it down the **Z axis**
3. Repeat this a few more times to create more obstacles
4. After making a few duplicates, select one in the hierarchy and **hold ctrl + click** to select multiple obstacles, then **duplicate** those

- **New Technique:** Duplicate (**Ctrl/Cmd+D**)
- **Tip:** Try using top-down view to make this easier
- **Tip:** Try using the inspector to space your obstacles exactly 25 apart



Lesson Recap

New Functionality

- Vehicle moves down the road at a constant speed
- When the vehicle collides with obstacles, they fly into the air

New Concepts and Skills

- C# Scripts
- Start vs Update
- Comments
- Methods
- Pass parameters
- Time.deltaTime
- Multiply (*) operator
- Components
- Collider and RigidBody

Next Lesson

- We'll add some code to our camera, so that it follows the player as they drive along the road.



1.3 High Speed Chase

Steps:

Step 1: Add a speed variable for your vehicle

Step 2: Create a new script for the camera

Step 3: Add an offset to the camera position

Step 4: Make the offset into a Vector3 variable

Step 5: Edit the playmode tint color

Example of project by end of lesson



Length: 50 minutes

Overview: Keep your eyes on the road! In this lesson you will code a new C# script for your camera, which will allow it to follow the vehicle down the road and give the player a proper view of the scene. In order to do this, you'll have to use a very important concept in programming: variables.

Project Outcome: The camera will follow the vehicle down the road through the scene, allowing the player to see where it's going.

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

- Declare variables properly and understand that variables can be different data types (float, Vector3, GameObject)
- Initialize/assign variables through code or through the inspector to set them with appropriate values
- Use appropriate access modifiers (public/private) for your variables in order to make them easier to change in the inspector

Step 1: Add a speed variable for your vehicle

We need an easier way to change the vehicle's speed and allow it to be accessed from the inspector. In order to do so what we need is something called a **variable**.

1. In PlayerController.cs, add **public float speed = 5.0f;** at the top of the **class**
2. Replace the **speed value** in the Translate method with the **speed variable**, then test
3. **Save** the script, then edit the speed value in the **inspector** to get the speed you want

- **New Concept:** Floats and Integers
- **New Concept:** Assigning Variables
- **New Concept:** Access Modifiers

```
public float speed = 20;

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

Step 2: Create a new script for the camera

The camera is currently stuck in one position. If we want it to follow the player, we have to make a new script for the camera.

1. Create a new **C# script** called FollowPlayer and attach it to the **camera**
2. Add **public GameObject player;** to the top of the script
3. Select the **Main Camera**, then, drag the player object onto the **empty player variable** in the Inspector
4. In **Update()**, assign the camera's position to the player's position, then test

- **Warning:** Remember to capitalize your script name correctly and rename it as soon as the script is created!
- **Warning:** It's really easy to forget to assign the player variable in the inspector
- **Don't worry:** The camera will be under the car... weird! We will fix that soon

```
public GameObject player;

void Update()
{
    transform.position = player.transform.position;
}
```

Step 3: Add an offset to the camera position

We need to move the camera's position above the vehicle so that the player can have a decent view of the game.

1. In the line in the Update method add **+ new Vector3(0, 5, -7)**, then test

- **New Concept:** Vector3 in place of coordinates
- **Tip:** You need "new Vector3()" because 3 numbers in a row could mean anything
- **New Concept:** FixedUpdate
- **Warning:** Remember to update your comments and maintain their accuracy!

```
public GameObject player;

void Update()
{
    transform.position = player.transform.position + new Vector3(0, 5, -7);
}
```

Step 4: Make the offset into a Vector3 variable

We've fixed the camera's position, but we may want to change it later! We need an easier way to access the offset.

1. At the top of **FollowPlayer.cs**, declare **private Vector3 offset;**
2. Copy the **new Vector3()** code and **assign** it to that variable
3. **Replace** the original code with the **offset** variable
4. **Test and save**

- **Don't worry:** Pay no mind to the read only warning
- **Tip:** Whenever possible, make variables! You never want hard values in the middle of your code

```
public GameObject player;
private Vector3 offset = new Vector3(0, 5, -7);

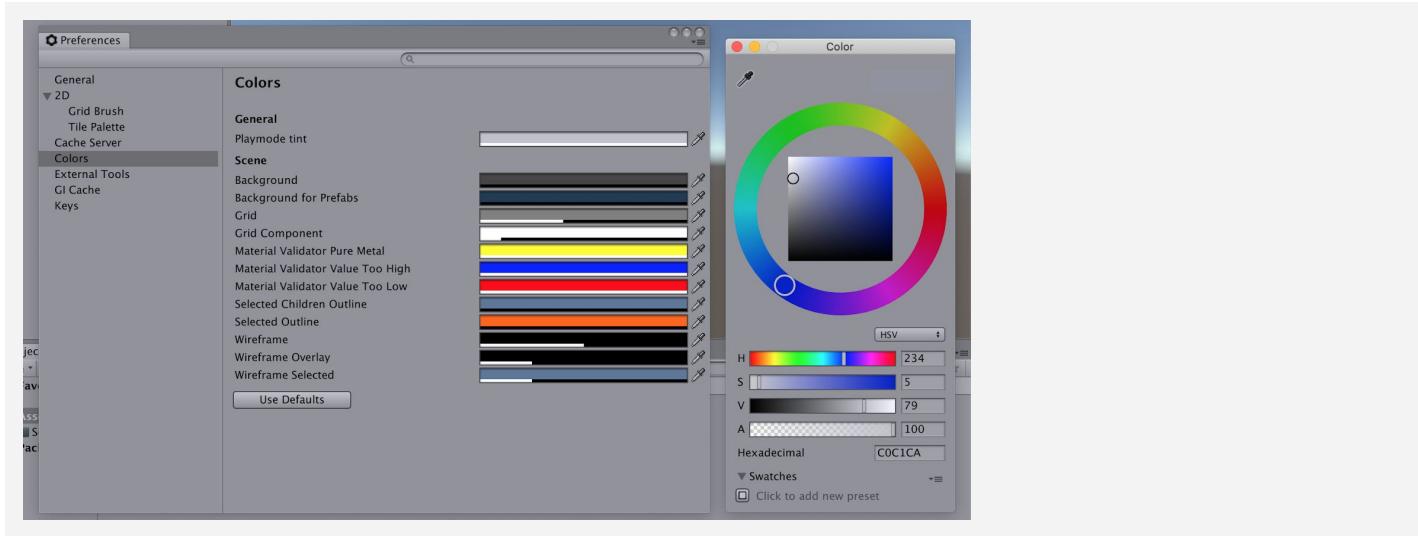
void Update()
{
    transform.position = player.transform.position + new Vector3(0, 5, -7)
offset;
}
```

Step 5: Edit the playmode tint color

If we're going to be creating and editing variables, we need to make sure we don't accidentally try to make changes when in "Play mode"

1. From the top menu, go to *Edit > Preferences (Windows)* or *Unity > Preferences (Mac)*
2. In the left menu, choose **Colors**, then edit the "**Playmode tint**" color to have a *slight* color
3. **Play** your project to test it, then close your preferences

- **Tip:** Try editing a variable in play mode, then stopping - it will revert
- **Warning:** Don't go crazy with the colors or it will be distracting



Lesson Recap

New Functionality

- Camera follows the vehicle down the road at a set offset distance

New Concepts and Skills

- Variables
- Data types
- Access Modifiers
- Declare and initialize variables

Next Lesson

- In the next lesson, we'll add our last lines of code to take control of our car and be able to drive it around the scene.



1.4 Step into the Driver's Seat

Steps:

Step 1: Allow the vehicle to move left/right

Step 2: Base left/right movement on input

Step 3: Take control of the vehicle speed

Step 4: Make vehicle rotate instead of slide

Step 5: Clean your code and hierarchy

Example of project by end of lesson



Length: 50 minutes

Overview: In this lesson, we need to hit the road and gain control of the vehicle. In order to do so, we need to detect when the player is pressing the arrow keys, then accelerate and turn the vehicle based on that input. Using new methods, Vectors, and variables, you will allow the vehicle to move forwards or backwards and turn left to right.

Project Outcome: When the player presses the up/down arrows, the vehicle will move forward and backward. When the player presses the left/right arrows, the vehicle will turn.

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

- Gain user input with Input.GetAxis, allowing the player to move in different ways
- Use the Rotate function to rotate an object around an axis
- Clean and organize your hierarchy with Empty objects

Step 1: Allow the vehicle to move left/right

Until now, the vehicle has only been able to move straight forward along the road. We need it to be able to move left and right to avoid the obstacles.

1. At the top of PlayerController.cs, add a **public float turnSpeed**; variable - **New Function:** Vector3.right
2. In **FixedUpdate()**, add
`transform.Translate(Vector3.right * Time.deltaTime * turnSpeed);`
3. Run your game and use the **turnSpeed** variable slider to move the vehicle left and right

```
public float turnSpeed;

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

Step 2: Base left/right movement on input

Currently, we can only control the vehicle's left and right movement in the inspector. We need to grant some power to the player and allow them to control that movement for themselves.

1. In PlayerController.cs, add a new **public float horizontalInput** variable
2. In **FixedUpdate**, assign **horizontalInput = Input.GetAxis("Horizontal")**; then test to see it in inspector
3. Add the **horizontalInput** variable to your left/right **Translate** method to gain control of the vehicle
4. In the inspector, edit the **turnSpeed** and **speed** variables to tweak the feel

- **New:** Input.GetAxis
- **Tip:** Edit > Project Settings > Input and expand the Horizontal Axis to show everything about it
- **Warning:** Spelling is important in string parameters. Make sure you spell and capitalize "Horizontal" correctly!

```
public float horizontalInput;

void Update()
{
    horizontalInput = Input.GetAxis("Horizontal");

    transform.Translate(Vector3.forward * Time.deltaTime * speed);
    transform.Translate(Vector3.right * Time.deltaTime * turnSpeed * horizontalInput);
}
```

Step 3: Take control of the vehicle speed

We've allowed the player to control the steering wheel, but we also want them to control the gas pedal and brake.

1. Declare a new public ***forwardInput*** variable
2. In ***FixedUpdate***, assign ***forwardInput = Input.GetAxis("Vertical");***
3. Add the ***forwardInput*** variable to the ***forward Translate method***, then test

- **Tip:** It can go backwards, too!
- **Warning:** This is slightly confusing with *forwardInput* and vertical axis

```
public float horizontalInput;
public float forwardInput;

void Update()
{
    horizontalInput = Input.GetAxis("Horizontal");
    forwardInput = Input.GetAxis("Vertical");

    transform.Translate(Vector3.forward * Time.deltaTime * speed * forwardInput);
    transform.Translate(Vector3.right * Time.deltaTime * turnSpeed * horizontalInput);
}
```

Step 4: Make vehicle rotate instead of slide

There's something weird about the vehicle's movement... it's slides left to right instead of turning. Let's allow the vehicle to turn like a real car!

1. In ***FixedUpdate***, call ***transform.Rotate(Vector3.up, horizontalInput)***, then test
2. **Delete** the line of code that ***translates Right***, then test
3. Add **** turnSpeed * Time.deltaTime***, then test

- **New:** *transform.Rotate*
- **Tip:** You can always trust the official Unity scripting API documentation

```
void Update()
{
    horizontalInput = Input.GetAxis("Horizontal");
    forwardInput = Input.GetAxis("Vertical");

    transform.Translate(Vector3.forward * Time.deltaTime * speed * forwardInput);
    transform.Rotate(Vector3.up, turnSpeed * horizontalInput * Time.deltaTime);
    transform.Translate(Vector3.right * Time.deltaTime * turnSpeed * horizontalInput);
}
```

Step 5: Clean your code and hierarchy

We added lots of new stuff in this lesson. Before moving on and to be more professional, we need to clean our scripts and hierarchy to make them more organized.

1. In the hierarchy, Right-click > Create Empty and rename it “Obstacles”, then drag all the obstacles into it
2. **Initialize** variables with values in **PlayerController**, then make all variables **private** (except for the **player** variables)
3. Use // to add **comments** to each section of code

- **New:** Empty Object
- **Tip:** You don't actually need to type “private”, it defaults to that
- **Tip:** Comments are important, especially for your future self

```
public private float speed = 20.0f;
public private float turnSpeed = 45.0f;
public private float horizontalInput;
public private float forwardInput;

void Update() {
    horizontalInput = Input.GetAxis("Horizontal");
    forwardInput = Input.GetAxis("Vertical");
    // Moves the car forward based on vertical input
    transform.Translate(Vector3.forward * Time.deltaTime * speed * forwardInput);
    // Rotates the car based on horizontal input
    transform.Rotate(Vector3.up, turnSpeed * horizontalInput * Time.deltaTime);
}
```

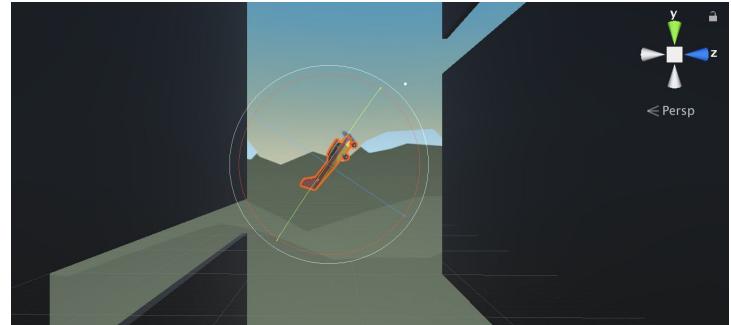
Lesson Recap

New Functionality	<ul style="list-style-type: none"> • When the player presses the up/down arrows, the vehicle will move forward and backward • When the player presses the left/right arrows, the vehicle turns
New Concepts and Skills	<ul style="list-style-type: none"> • Empty objects • Get user input • Translate vs Rotate
Next Lesson	<ul style="list-style-type: none"> • We made our first project! We learned a lot about how Unity works, we wrote our first lines of code, and we made a driving game where our player has full control over this vehicle!



Challenge 1

Plane Programming



Challenge Overview: Use the skills you learned in the driving simulation to fly a plane around obstacles in the sky. You will have to get the user's input from the up and down arrows in order to control the plane's pitch up and down. You will also have to make the camera follow alongside the plane so you can keep it in view.

Challenge Outcome:

- The plane moves forward at a constant rate
- The up/down arrows tilt the nose of the plane up and down
- The camera follows along beside the plane as it flies

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

- Using the Vector3 class to move and rotate objects along/around an axis
- Using Time.deltaTime in the Update() method to move objects properly
- Moving and rotating objects in scene view to position them the way you want
- Assigning variables in the inspector and initializing them in code
- Implementing Input variables to control the movement/rotation of objects based on User input

Challenge Instructions:

- Open your **Prototype 1** project
- **Download** the "Challenge 1 Starter Files" from the Tutorial Materials section, then double-click on it to **Import**
- In the *Project Window > Assets > Challenge 1 > Instructions* folder, use the Outcome video as a guide to complete the challenge

Challenge	Task	Hint
1 The plane is going backwards	Make the plane go forward	Vector3.back makes an object move backwards, Vector3.forward makes it go forwards
2 The plane is going too fast	Slow the plane down to a manageable speed	If you multiply a value by Time.deltaTime, it will change it from 1x/frame to 1x/second
3 The plane is tilting automatically	Make the plane tilt only if the user presses the up/down arrows	In PlayerControllerX.cs, in Update(), the verticalInput value is assigned, but it's never actually used in the Rotate() call
4 The camera is <i>in front</i> of the plane	Reposition it so it's beside the plane	For the camera's position, try X=30, Y=0, Z=10 and for the camera's rotation, try X=0, Y=-90, Z=0
5 The camera is not following the plane	Make the camera follow the plane	In FollowPlayerX.cs, neither the plane nor offset variables are assigned a value - assign the plane variable in the camera's inspector and assign the offset = new Vector3(0, 30, 10) in the code

Bonus Challenge	Task	Hint
X The plane's propeller does not spin	Create a script that spins the plane's propeller	There is a "Propeller" child object of the plane - you should create a new "SpinPropellerX.cs" script and make it rotate every frame around the Z axis.

Challenge Solution

- 1 In PlayerControllerX.cs, in Update, change `Vector3.back` to `Vector3.forward`

```
// move the plane forward at a constant rate
transform.Translate(Vector3.back.forward * speed);
```

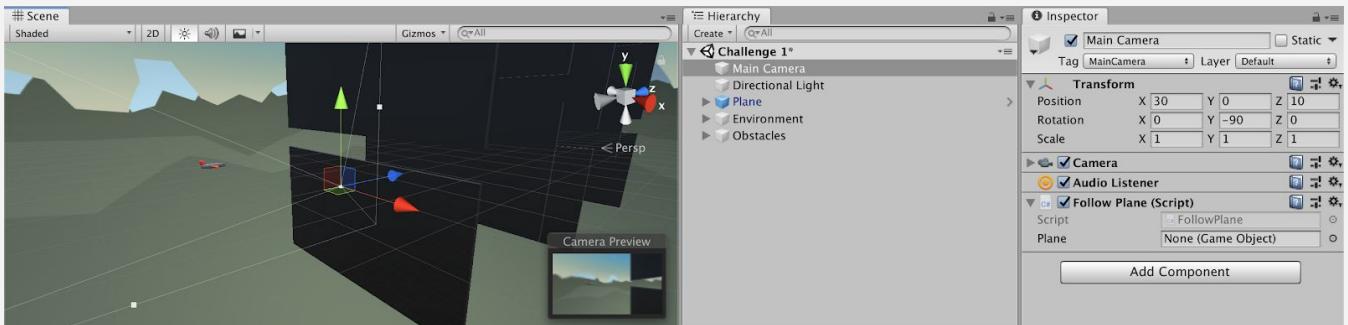
- 2 In PlayerControllerX.cs, in Update, add `* Time.deltaTime` to the Translate call

```
// move the plane forward at a constant rate
transform.Translate(Vector3.forward * speed * Time.deltaTime);
```

- 3 In PlayerControllerX.cs, include the `verticalInput` variable to the Rotate method:

```
// tilt the plane up/down based on up/down arrow keys
transform.Rotate(Vector3.right * rotationSpeed * verticalInput * Time.deltaTime);
```

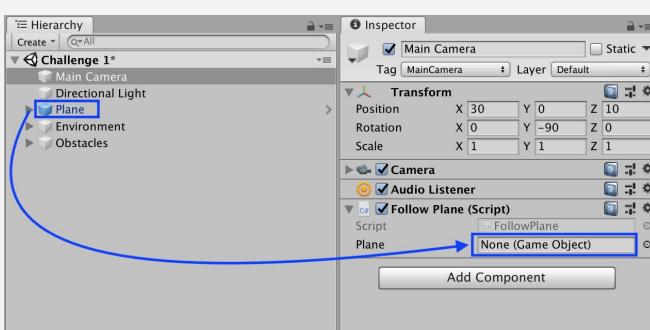
- 4 Change the camera's position to (30, 0, 10) and its rotation, to (0, -90, 0)



- 5 To assign the `plane` variable, select **Main Camera** in the hierarchy, then drag the **Plane** object onto the "Plane" variable in the inspector

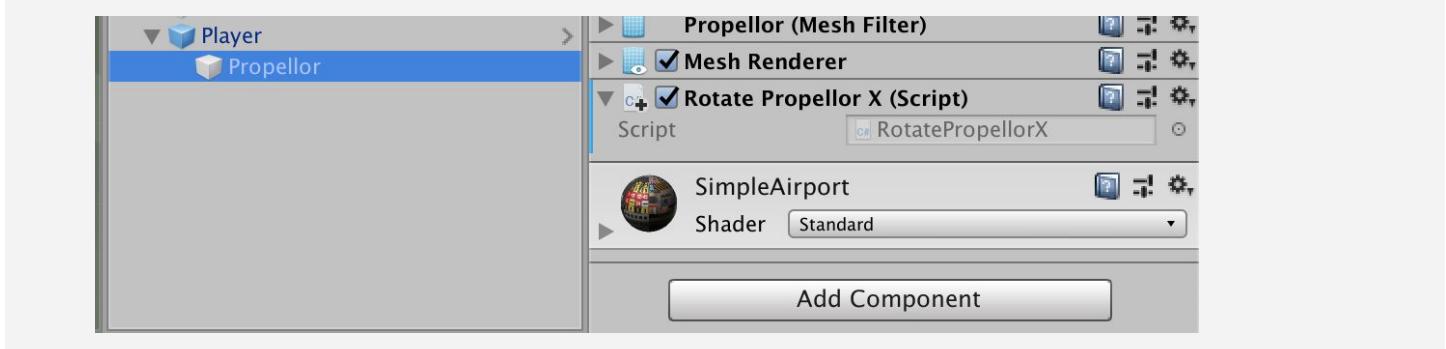
To assign the `offset variable`, add the value as a new Vector3 at the top of `FollowPlane.cs`:

```
private Vector3 offset = new Vector3(30,
0, 10);
```



Bonus Challenge Solution

- X1 Create a new Script called “SpinPropellerX.cs” and attach it to the “Propellor” object (which is a child object of the Plane):



- X2 In RotatePropellerX.cs, add a new propellorSpeed variable and Rotate the propeller on the Z axis

```
private float propellorSpeed = 1000;

void Update() {
    transform.Rotate(Vector3.forward, propellorSpeed * Time.deltaTime);
}
```



Unit 1 Lab

Project Design Document

Steps:

Step 1: Understand what a Personal Project is

Step 2: Review Design Doc examples

Step 3: Complete your Project Concept V1

Step 4: Complete your Project Timeline

Step 5: Complete your MVP sketch

Example of progress by end of lab

Capstone Project Plan

03/21/2019
Aaron Sharp

Project Concept	
1 Player Control	You control a frog in this game where the arrow keys make the player move vertically and horizontally in 1-space increments.
2 Basic Gameplay	During the game, cars and floating logs appear from the sides of the screen and the goal of the game is to Get the frog to the top of the screen without being hit by a car or falling in the water.
3 Sound & Effects	There will be sound effects every time the frog moves, gets to the other side, or is destroyed and particle effects when the frog splashes in the water or gets hit by a car. (optional) There will also be fun music in the background and animated water.

Length: 60 minutes

Overview: In this first ever Lab session, you will begin the preliminary work required to successfully create a personal project in this course. First, you'll learn what a personal project is, what the goals for it are, and what the potential limitations are. Then you will take the time to come up with an idea and outline it in detail in your Design Document, including a timeline for when you hope to complete certain features. Finally, you will take some time to draw a sketch of your project to help you visualize it and share your idea with others.

Project Outcome: The Design Document will be filled out, including the concept, the timeline, and a preliminary sketch of the minimum viable product.

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

- Come up with an idea for a project with a scope appropriate to your time and available resources
- Think through a project's concept in order to better understand its requirements
- Plan out a project's milestones with due dates to better understand the production cycle and to hold yourself more accountable
- Create a simple sketch / storyboard in order to better communicate your ideas

Step 1: Understand what a Personal Project is

Before we get started on our personal projects, we should make sure we understand our primary goals.

Explain	<p>What Personal Projects (PP's) are:</p> <ul style="list-style-type: none"> • Projects they will be working on on their own with less direct instruction • A chance to create a project they really care about with their own creative choices • An opportunity to apply and solidify skills they learned in lessons and challenges
Demo	<p>The Core Functionality and skills they will learn from each of the 5 Units by showcasing completed versions of each Prototype:</p> <ol style="list-style-type: none"> 1. Driving Simulation: player control through user input 2. Feed the Animals: basic gameplay by spawning random objects on an interval and trying to collect them, avoid them, or fire projectiles at them 3. Run and Jump: sound and effects, and animation (of background or player) 4. Sumo Battle: gameplay mechanics, powerups and/or increasing difficulty 5. Quick Click: user interface with title screen, game over screen, and score display
Explain	<p>Goal / Evaluation of the PP's are based on:</p> <ul style="list-style-type: none"> • <u>Completeness</u> - how much of what you set out to complete did you actually finish • <u>Uniqueness / Application</u> - how much did you add new design and dev features, extending and applying your skills in novel and creative ways <p>NOTE - These two priorities are at odds and it's up to you to find the balance</p>
Explain	<p>You just need a Minimum Viable Product (an MVP) - doesn't have to be polished</p> <ul style="list-style-type: none"> • Definition: a product with just enough features to satisfy early customers, and to provide feedback for future product development • This will allow them to focus on the core of the project and not get distracted by flashy features and graphics that don't matter as much
Warning	<p>There will be a temptation to try and do too much that is completely different from what anything in the course (e.g. "I want to make "Madden + Facebook + Google!")</p> <ul style="list-style-type: none"> • There's <i>lots</i> of time to try and do really ambitious crazy projects in the future, but for now on this first project, try to stick closely to the core functionality you're learning • The <i>only</i> limitation is time - with enough time, they could make anything!
Discuss	<p>Make sure students understand what the Personal Project is, allowing them to ask questions</p>

Step 2: Review Design Doc examples

Now that we have some idea of what a Personal Project is, let's look at a couple examples

1. Click on the link to open the “**Project Design Doc [EXAMPLE]**” and read through the **Project Concept**
2. Click on the link to open a new “**Project Design Doc**” as either a **Google Doc Copy**, **Word Doc** or **PDF**
3. Think through how you would fill out a design doc for other games

- **Warning:** you will need to be signed into a Google account to be able to make a copy of the Google Doc version
- **Tip:** Search YouTube for “gameplay” of the classic game you want
- **Explanation:** Notice that sections correspond to what you’ll be learning with each unit/prototype

Capstone Project Plan		
03/21/2019 Aaron Sharp		
Project Concept		
1 Player Control	You control a frog in this top Down game where the arrow keys makes the player Move vertically and horizontally in 1-space increments	
2 Basic Gameplay	During the game, cars and floating logs appear from the sides of the screen and the goal of the game is to Get the frog to the top of the screen without being hit by a car or falling in the water	
3 Sound & Effects	There will be sound effects every time the frog moves, gets to the other side, or is destroyed and particle effects when the frog splashes in the water or gets hit by a car (optional) There will also be fun music in the background and animated water	
4 Gameplay Mechanics	As the game progresses, new logs will appear that occasionally sink making it more difficult for the player to time their jumps (optional) There will also be ladybugs on random logs that give the player some extra points	
5 Object Spawning	The score will increase whenever the player moves or gets to the other end At the start of the game, the title "Frogger" will appear and the game will end when the timer runs out	
6 Other Features	There will be 5 places where the frog can “land” at the other end. Every time they get into one of these, the timer will be extended, allowing them to get more points	

Step 3: Complete your Project Concept V1

Now that we've seen some examples, let's try to come up with our own project concept.

1. Add your **name** and **date** in the top-right corner
2. **Fill in the blanks** for your project concept
3. **Share** your project concept with someone else to make sure it makes sense to them

- **Explanation:** In the Course Library, you've got human characters, animals, vehicles, foods, sports balls, other random things, but you can always use “primitives” as placeholders in a MVP, then go to the Unity Asset store to get real graphics
- **Tip:** This is good opportunity to catch yourself if you're being too ambitious
- **Don't worry:** This is just a best guess right now, if you want to change your project completely next lab, you could

Step 4: Complete your Project Timeline

Now that we know the basic concept of our project, let's figure out how we're going to get it done.

1. Fill in **milestone descriptions** based on your schedule for the course, including self-imposed due dates
2. Add features that will *not* be included in your MVP to the “**Backlog**”

- **Warning:** This is a MVP, so don't be afraid to put objects on backlog that you'll get to in version 2
- **Explanation:** In Lab 2 you will be setting up your project, in Lab 3 you will do basic player movement, in Lab 4 you will add basic gameplay, and Lab 5 you will add graphics - that would be a good start in filling this out
- **Tip:** This will depend heavily on the schedule you're following for this course - you should leave a significant amount of time to work on it at the end when you've completed all 5 units
- **Don't worry:** It will be hard to do this accurately, since you don't know how long things take - this can change
- **Don't worry:** You don't need to use all milestones - can add more or leave blank rows you are not using
- **Tip:** These should be worded as “Completed functionality” - as in: “Frog can move side-to-side based on left/right arrow keys”

Project Timeline		
Milestone	Description	Due
#1	- Basic frog movement left to right and side to side	03/01
#2	- Random cars spawning from left and right sides, in 3 discrete lanes - When the player is hit by one of the cars, it will be repositioned at start	04/01
#3	- Logs generating from left/right sides of the screens - The player can ride on these logs and be moved along with them - When player falls in the water (not on log), the are repositioned at start	05/01
Backlog	- 5 discrete areas for the frog to land, each one extending the timer - Random things that pop up and block the 5 landing spots - Ladybug powerup that gives the player extra points and invincibility	09/01

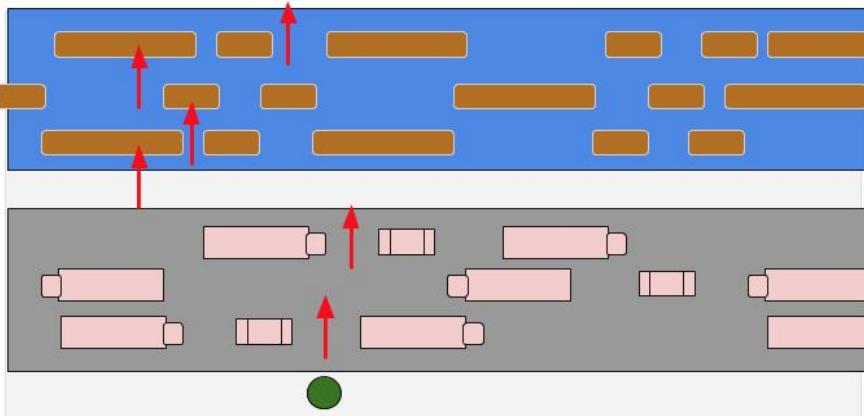
Step 5: Complete your MVP sketch

To help visualize our *minimum viable product*, it's always helpful to have a sketch.

1. Look at sketch in the **example**
2. Using Google Docs, some other online simple drawing program, or pencil and paper, draw a sketch of your MVP and add it to your doc

- **Warning:** Do not spend forever on this - it's just a sketch - use circles, squares, and arrows
- **Explanation:** This should just be a sketch of your MVP - what you hope to accomplish by the end of the course - *not* the fully fledged product

Minimum Viable Product Sketch



Lesson Recap

- New Progress** • Completed your project concept and production timeline

- New Concepts and Skills**
- Personal Projects
 - Design Documents
 - Project Timelines,
 - Project Milestones and Backlogs
 - Minimum Viable Products



Quiz Unit 1

QUESTION

CHOICES

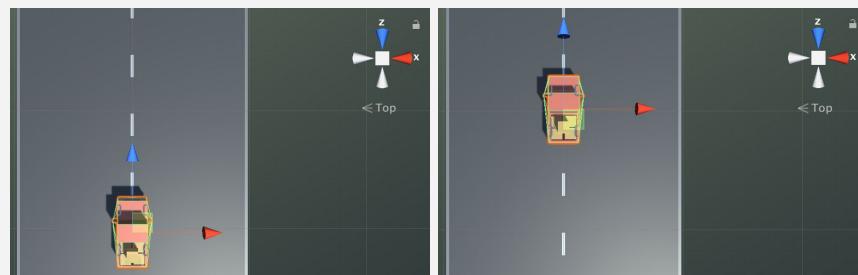
1 Which Unity window contains a list of all the game objects currently in your scene?

- a. Scene view
- b. Project window
- c. Hierarchy
- d. Inspector

2 True or False:
Visual Studio is not a part of Unity. You could use a different code editor to edit your C# scripts if you wanted to.

- a. True
- b. False

3 What best describes the difference between the below images, where the car is in the second image is further along the road?



- a. The second car's X location value is higher than the first car's
- b. The second car's Y location value is higher than the first car's
- c. The second car's Z location value is higher than the first car's
- d. The second car's Transform value is higher than the first car's.

4 In what order do you put the words when you are declaring a new variable?

```
public float speed = 20.0f;
```

- a. [data type] [access modifier] [variable value] [variable name]
- b. [access modifier] [data type] [variable name] [variable value]
- c. [data type] [access modifier] [variable name] [variable value]
- d. [variable name] [data type]

[access modifier] [variable value]

- 5 Which of the following variables would be visible in the Inspector?

```
public float speed;
float turnSpeed = 45.0f;
private float horizontalInput;
private float forwardInput;
```

- a. speed
- b. turnSpeed
- c. speed & turnSpeed
- d. horizontalInput & forwardInput

- 6 What is a possible value for the horizontalInput variable?

```
horizontalInput = Input.GetAxis("Horizontal");
```

- a. -10
- b. 0.52
- c. "Right"
- d. Vector3.Up

- 7 What is true about the following two lines of code?

```
transform.Translate(Vector3.forward);
transform.Translate(1, 0, 0);
```

- a. They will both move an object at the same speed
- b. They will both move an object in the same direction
- c. They will both move an object along the same axis
- d. They will both rotate an object, but along different axes

- 8 Which of the following lines of code is using standard Unity naming conventions?

```
/* a */ Public Float Speed = 40.0f;
/* b */ public float Speed = 40.0f;
/* c */ public float speed = 40.0f;
/* d */ public float speed = 40.0f;
```

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

- 9 Which comment would best describe the code below?

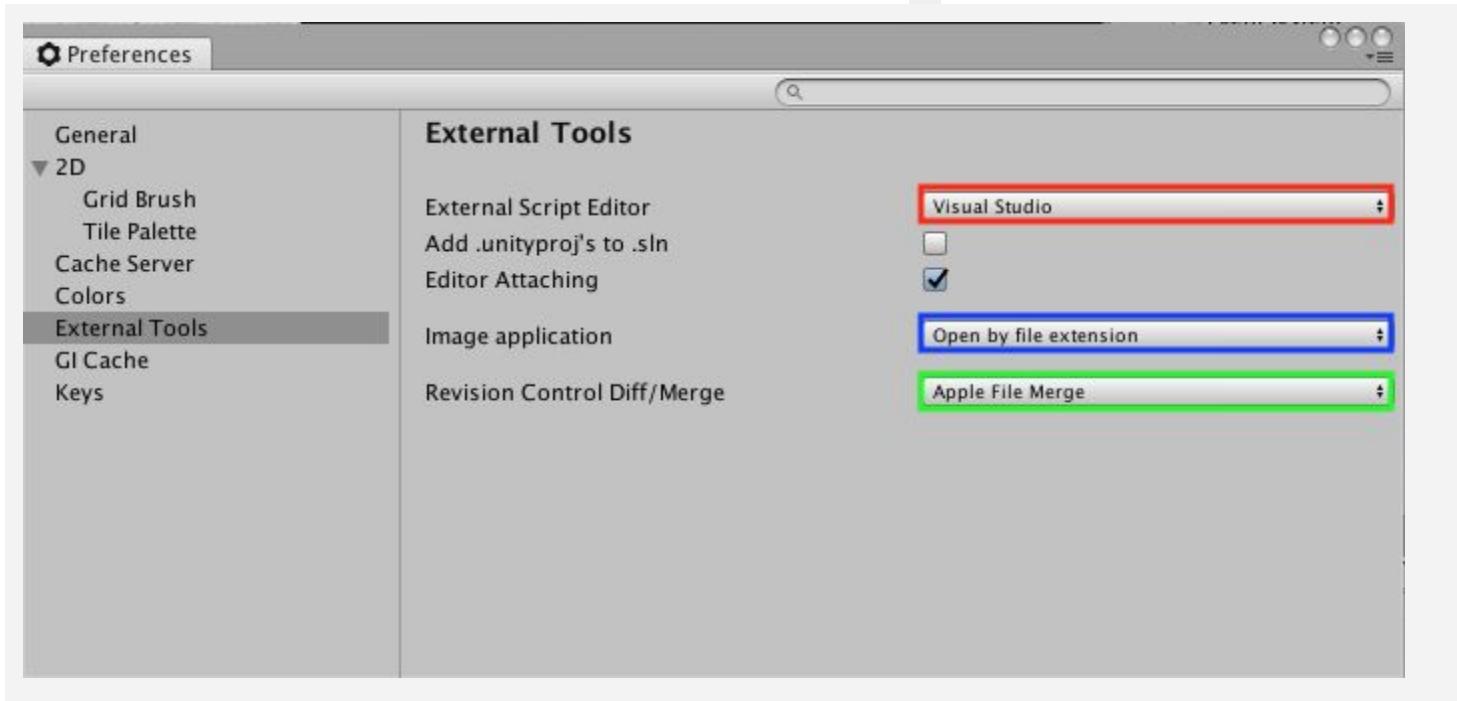
```
horizontalInput = Input.GetAxis("Horizontal");
transform.Rotate(Vector3.up, horizontalInput);
```

- a. // Rotates around the Y axis based on left/right arrow keys
- b. // Rotates around the Z axis based on up/down arrow keys
- c. // Rotates in an upward direction based on left/right

- arrow keys
d. // Moves object up/down based on the the left/right arrow keys

10 The image below shows the preferences window that allows you to change which script editing tool (or IDE) you want to use. Where would you click to choose an alternative code editing tool?

- a. The red box
b. The blue box
c. The green box



Quiz Answer Key

#	ANSWER	EXPLANATION
1	C	The Hierarchy window contains a list of every GameObject in the current Scene. As objects are added and removed in the Scene, they will appear and disappear from the Hierarchy as well.
2	B	True. Visual Studio is just one of many editors you could use to edit your code, including editors like Atom, Sublime, or even a basic Text Editor.
3	C	You can tell which axis the car has moved along using the XYZ directional gizmo in the top-right, which shows the blue axis pointing forwards down the road.
4	B	Variables are always declared in the order: [access modifier] - public, private, etc [data type] - float, int, GameObject, etc [variable name] - speed, turnSpeed, player, offset, etc [variable value] - 1.0f, 2, new Vector3(0, 1, 0), etc
5	A	“public float speed” would be visible because it has the “public” modifier applied to it
6	B	Input.GetAxis returns a float value between -1 and 1, which means 0.52 is a possible value
7	A	Vector3.forward is the equivalent of (0, 0, 1), which has the same magnitude as (1, 0, 0), even though they’re in different directions, so they would both move an object at the same speed, but along different axes
8	D	“public float speed = 40.0f;” uses the correct naming conventions because all three of these terms should start with lowercase letters
9	A	Vector3.up is the Y axis and it’s using the Horizontal input value, so it would rotate around the Y axis when the user presses the left/right arrows
10	A	You would click on the Red box to change the “External Script Editor” from Visual Studio to another tool.



2.1 Player Positioning

Steps:

Step 1: Create a new Project for Prototype 2

Step 2: Add the Player, Animals, and Food

Step 3: Get the user's horizontal input

Step 4: Move the player left-to-right

Step 5: Keep the player inbounds

Step 6: Clean up your code and variables

Example of project by end of lesson



Length: 60 minutes

Overview: You will begin this unit by creating a new project for your second Prototype and getting basic player movement working. You will first choose which character you would like, which types of animals you would like to interact with, and which food you would like to feed those animals. You will give the player basic side-to-side movement just like you did in Prototype 1, but then you will use if-then statements to keep the Player in bounds.

Project Outcome: The player will be able to move left and right on the screen based on the user's left and right key presses, but will not be able to leave the play area on either side.

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

- Adjust the scale of an object proportionally in order to get it to the size you want
- More comfortably use the GetInput function in order to use user input to control an object
- Create an if-then statement in order to implement basic logic in your project, including the use of greater than ($>$) and less than ($<$) operators
- Use comments and automatic formatting in order to make their code more clean and readable to other programmers

Step 1: Create a new Project for Prototype 2

The first thing we need to do is create a new project and import the Prototype 2 starter files.

1. Open **Unity Hub** and create a **New** project named “Prototype 2” in your course directory
2. Click on the **link** to access the Prototype 2 starter files, then **import** them into Unity
3. Open the **Prototype 2 scene** and **delete** the SampleScene without saving
4. In the top-right of the Unity Editor, change your Layout from **Default** to your custom layout

- **Don't worry:** Unit 2 has far more assets than Unit 1, so the package might take a while to import.



Step 2: Add the Player, Animals, and Food

Let's get all of our objects positioned in the scene, including the player, animals, and food.

1. If you want, drag a different **material** from Course Library > Materials onto the Ground object
2. Drag 1 **Human**, 3 **Animals**, and 1 **Food** object into the Hierarchy
3. Rename the human “Player”, then **reposition** the animals and food so you can see them
4. Adjust the XYZ **scale** of the food so you can easily see it from above

- **New Technique:** Adjusting Scale
- **Warning:** Don't choose people for anything but the player, they don't have walking animations
- **Tip:** Remember, dragging objects into the hierarchy puts them at the origin



Step 3: Get the user's horizontal input

If we want to move the Player left-to-right, we need a variable tracking the user's input.

1. In your **Assets** folder, create a "Scripts" folder, and a "PlayerController" script inside
2. **Attach** the script to the Player and open it
3. At the top of PlayerController.cs, declare a new **public float horizontalInput**
4. In **Update()**, set **horizontalInput = Input.GetAxis("Horizontal")**, then test to make sure it works in the inspector

- **Warning:** Make sure to create your Scripts folder inside of the assets folder
- **Don't worry:** We're going to get VERY familiar with this process
- **Warning:** If you misspell the script name, just delete it and try again.

```
public float horizontalInput;

void Update()
{
    horizontalInput = Input.GetAxis("Horizontal");
}
```

Step 4: Move the player left-to-right

We have to actually use the horizontal input to translate the Player left and right.

1. Declare a new **public float speed = 10.0f;**
2. In **Update()**, Translate the player side-to-side based on **horizontalInput** and **speed**

- **Tip:** You can look at your old scripts for code reference

```
public float horizontalInput;
public float speed = 10.0f;

void Update()
{
    horizontalInput = Input.GetAxis("Horizontal");
    transform.Translate(Vector3.right * horizontalInput * Time.deltaTime * speed);
}
```

Step 5: Keep the player inbounds

We have to prevent the player from going off the side of the screen with an **if-then** statement.

1. In **Update()**, write an **if-statement** checking if the player's left X position is **less than** a certain value
2. In the if-statement, set the player's position to its current position, but with a **fixed X location**

- **Tip:** Move the player in scene view to determine the x positions of the left and right bounds
- **New Concept:** If-then statements
- **New Concept:** Greater than > and Less Than < operators

```
void Update() {
    if (transform.position.x < -10) {
        transform.position = new Vector3(-10, transform.position.y, transform.position.z);
    }
}
```

Step 6: Clean up your code and variables

We need to make this work on the right side, too, then clean up our code.

1. Repeat this process for the **right side** of the screen
2. Declare new **xRange** variable, then replace the hardcoded values with them
3. Add **comments** to your code

- **Warning:** Whenever you see hardcoded values in the body of your code, try to replace it with a variable
- **Warning:** Watch your greater than / less than signs!

```
public float xrange = 10;

void Update()
{
    // Keep the player in bounds
    if (transform.position.x < -10 -xrange)
    {
        transform.position = new Vector3(-10 -xrange, transform.position.y, transform.position.z);
    }
    if (transform.position.x > xrange)
    {
        transform.position = new Vector3(xrange, transform.position.y, transform.position.z);
    }
}
```

Lesson Recap

New Functionality

- The player can move left and right based on the user's left and right key presses
- The player will not be able to leave the play area on either side

New Concepts and Skills

- Adjust object scale
- If-statements
- Greater/Less than operators

Next Lesson

- We'll learn how to create and throw endless amounts of food to feed our animals!



2.2 Food Flight

Steps:

- Step 1: Make the projectile fly forwards
- Step 2: Make the projectile into a prefab
- Step 3: Test for spacebar press
- Step 4: Launch projectile on spacebar press
- Step 5: Make animals into prefabs
- Step 6: Destroy projectiles offscreen
- Step 7: Destroy animals offscreen

Example of project by end of lesson



Length: 70 minutes

Overview: In this lesson, you will allow the player to launch the projectile through the scene. First you will write a new script to send the projectile forwards. Next you will store the projectile along with all of its scripts and properties using an important new concept in Unity called Prefabs. The player will be able to launch the projectile prefab with a tap of the spacebar. Finally, you will add boundaries to the scene, removing any objects that leave the screen.

Project Outcome: The player will be able to press the Spacebar and launch a projectile prefab into the scene, which destroys itself when it leaves the game's boundaries. The animals will also be removed from the scene when they leave the game boundaries.

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

- Transform a game object into a prefab that can be used as a template
- Instantiate Prefabs to spawn them into the scene
- Override Prefabs to update and save their characteristics
- Get user input with GetKey and KeyCode to test for specific keyboard presses
- Apply components to multiple objects at once to work as efficiently as possible

Step 1: Make the projectile fly forwards

The first thing we must do is give the projectile some forward movement so it can zip across the scene when it's launched by the player.

1. Create a new "MoveForward" script, attach it to the food object, then open it

- **Don't worry:** You should all be super familiar with this method now... getting easier, right?

2. Declare a new **public float speed** variable;

3. In **Update()**, add

```
transform.Translate(Vector3.forward *
Time.deltaTime * speed);
```

4. In the **Inspector**, set the projectile's **speed** variable, then test

```
public float speed = 40;

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

Step 2: Make the projectile into a prefab

Now that our projectile has the behavior we want, we need to make it into a prefab so it can be reused anywhere and anytime, with all its behaviors included.

1. Create a new "Prefabs" folder, drag your food into it, and choose **Original Prefab**

- **New Concept:** Prefabs

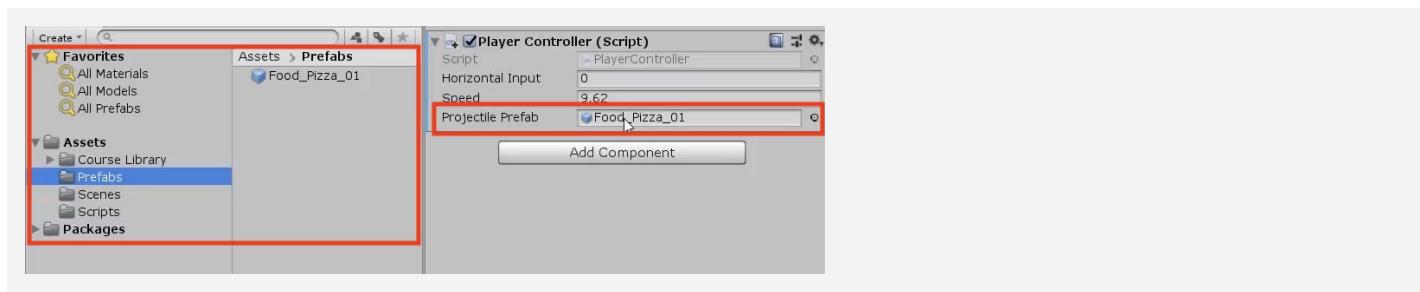
2. In PlayerController.cs, declare a new **public GameObject projectilePrefab**; variable

- **New Concept:** Original vs Variant Prefabs

3. **Select** the Player in the hierarchy, then **drag** the object from your Prefabs folder onto the new **Prefab Variant box** in the inspector

- **Tip:** Notice that this your projectile already has a move script if you drag it in

4. Try **dragging** the projectile into the scene at runtime to make sure they fly



Step 3: Test for spacebar press

Now that we have a projectile prefab assigned to *PlayerController.cs*, the player needs a way to launch it with the space bar.

1. In *PlayerController.cs*, in ***Update()***, add an **if-statement** checking for a spacebar press:
`if (Input.GetKeyDown(KeyCode.Space)) {`
2. Inside the if-statement, add a comment saying that you should **// Launch a projectile from the player**

- **Tip:** Google a solution. Something like "How to detect key press in Unity"
- **New Functions:** Input.GetKeyDown, GetKeyUp, GetKey
- **New Function:** KeyCode

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        // Launch a projectile from the player
    }
}
```

Step 4: Launch projectile on spacebar press

We've created the code that tests if the player presses spacebar, but now we actually need spawn a projectile when that happens

1. Inside the if-statement, use the **Instantiate** method to spawn a projectile at the player's location with the prefab's rotation

- **New Concept:** Instantiation

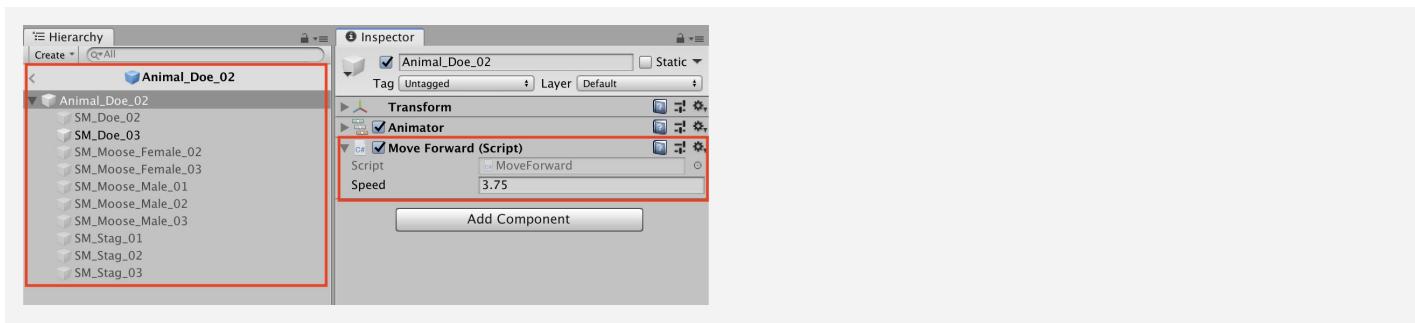
```
if (Input.GetKeyDown(KeyCode.Space))
{
    // Launch a projectile from the player
    Instantiate(projectilePrefab, transform.position, projectilePrefab.transform.rotation);
}
```

Step 5: Make animals into prefabs

The projectile is now a prefab, but what about the animals? They need to be prefabs too, so they can be instantiated during the game.

1. **Rotate** all animals on the Y axis by **180 degrees** to face down
2. **Select** all three animals in the hierarchy and **Add Component > Move Forward**
3. Edit their **speed values** and **test** to see how it looks
4. Drag all three animals into the **Prefabs folder**, choosing “Original Prefab”
5. **Test** by dragging prefabs into scene view during gameplay

- **Tip:** You can change all animals at once by selecting all them in the hierarchy while holding Cmd/Ctrl
- **Tip:** Adding a Component from inspector is same as dragging it on
- **Warning:** Remember, anything you change while the game is playing will be reverted when you stop it



Step 6: Destroy projectiles offscreen

Whenever we spawn a projectile, it drifts past the play area into eternity. In order to improve game performance, we need to destroy them when they go out of bounds.

1. Create “DestroyOutOfBounds” script and apply it to the **projectile**
2. Add a new **private float topBound** variable and initialize it = **30**;
3. Write code to destroy if out of top bounds **if (transform.position.z > topBound) { Destroy(gameObject); }**
4. In the Inspector **Overrides** drop-down, click **Apply all** to apply it to prefab

- **Warning:** Too many objects in the hierarchy will slow the game
- **Tip:** Google “How to destroy gameobject in Unity”
- **New Function:** Destroy
- **New Technique:** Override prefab

```
private float topBound = 30;

void Update() {
    if (transform.position.z > topBound) {
        Destroy(gameObject);
    }
}
```

Step 7: Destroy animals offscreen

If we destroy projectiles that go out of bounds, we should probably do the same for animals. We don't want critters getting lost in the endless abyss of Unity Editor...

1. Create a new **private float lowerBound** variable and initialize it = -10;
2. Create **else-if statement** to check if objects are beneath **lowerBound**:
else if (transform.position.z > topBound)
3. **Apply** the script to all of the animals, then **Override** the prefabs

- **New Function:** Else-if statement
- **Warning:** Don't make topBound too tight or you'll destroy the animals before they can spawn

```
private float topBound = 30;
private float lowerBound = -10;

void Update() {
    if (transform.position.z > topBound)
    {
        Destroy(gameObject);
    } else if (transform.position.z < lowerBound) {
        Destroy(gameObject);
    }
}
```

Lesson Recap

New Functionality	<ul style="list-style-type: none"> • The player can press the Spacebar to launch a projectile prefab, • Projectile and Animals are removed from the scene if they leave the screen
New Concepts and Skills	<ul style="list-style-type: none"> • Create Prefabs • Override Prefabs • Test for Key presses • Instantiate objects • Destroy objects • Else-if statements
Next Lesson	<ul style="list-style-type: none"> • Instead of dropping all these animal prefabs onto the scene, we'll create a herd of animals roaming the plain!



2.3 Random Animal Stampede

Steps:

Step 1: Create a spawn manager

Step 2: Spawn an animal if S is pressed

Step 3: Spawn random animals from array

Step 4: Randomize the spawn location

Step 5: Change the perspective of the camera

Example of project by end of lesson



Length: 50 minutes

Overview: Our animal prefabs walk across the screen and get destroyed out of bounds, but they don't actually appear in the game unless we drag them in! In this lesson we will allow the animals to spawn on their own, in a random location at the top of the screen. In order to do so, we will create a new object and a new script to manage the entire spawning process.

Project Outcome: When the user presses the S key, a randomly selected animal will spawn at a random position at the top of the screen, walking towards the player.

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

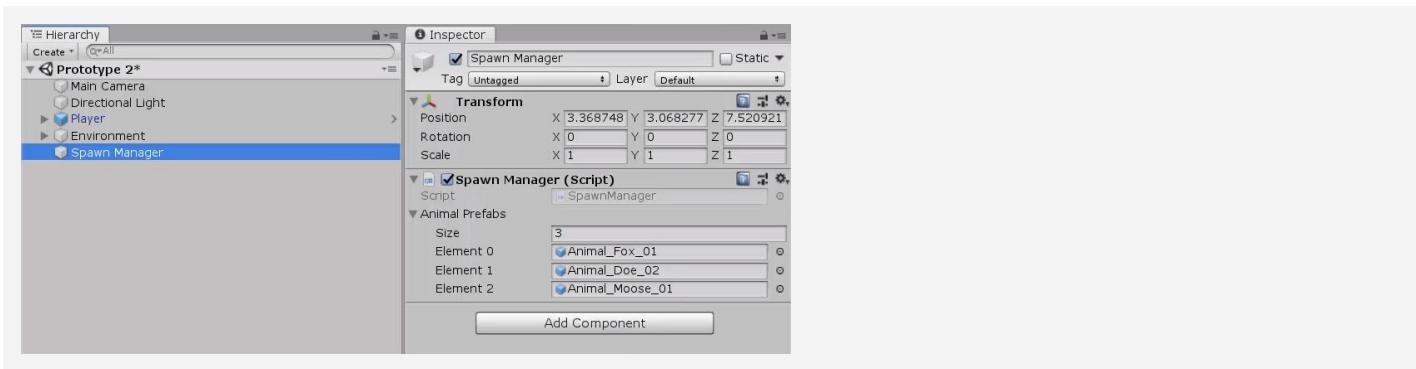
- Create an empty object with a script attached
- Use arrays to create an accessible list of objects or values
- Use integer variables to determine an array index
- Randomly generate values with Random.Range in order to randomize objects in arrays and spawn positions
- Change the camera's perspective to better suit your game

Step 1: Create a spawn manager

If we are going to be doing all of this complex spawning of objects, we should have a dedicated script to manage the process, as well as an object to attach it to.

1. In the hierarchy, create an **empty object** called “Spawn Manager”
2. Create a new script called “SpawnManager”, attach it to the **Spawn Manager**, and open it
3. Declare new **public GameObject[] animalPrefabs;**
4. In the inspector, change the **Array size** to match your animal count, then **assign** your animals by **dragging** them in

- **Tip:** Empty objects can be used to store objects or used to store scripts
- **Warning:** You can use spaces when naming your empty object, but make sure your script name uses PascalCase!
- **New Concept:** Arrays



Step 2: Spawn an animal if S is pressed

We've created an array and assigned our animals to it, but that doesn't do much good until we have a way to spawn them during the game. Let's create a temporary solution for choosing and spawning the animals.

1. In **Update()**, write an if-then statement to **instantiate** a new animal prefab at the top of the screen if **S** is pressed
2. Declare a new **public int animalIndex** and incorporate it in the **Instantiate** call, then test editing the value in the Inspector

- **New Concept:** Array Indexes
- **Tip:** Array indexes start at 0 instead of 1. An array of 3 animals would look like [0, 1, 2]
- **New Concept:** Integer Variables
- **Don't worry:** We'll declare a new variable for the Vector3 and index later

```
public GameObject[] animalPrefabs;
public int animalIndex;

void Update() {
    if (Input.GetKeyDown(KeyCode.S)) {
        Instantiate(animalPrefabs[animalIndex], new Vector3(0, 0, 20),
        animalPrefabs[animalIndex].transform.rotation);
    }
}
```

Step 3: Spawn random animals from array

We can spawn animals by pressing S, but doing so only spawns an animal at the array index we specify. We need to randomize the selection so that S can spawn a random animal based on the index, without our specification.

1. In the if-statement checking if S is pressed, generate a random **int animalIndex** between 0 and the length of the array
2. Remove the global **animalIndex** variable, since it is only needed locally in the **if-statement**

- **Tip:** Google “how to generate a random integer in Unity”
- **New Function:** Random.Range
- **New Function:** .Length
- **New Concept:** Global vs Local variables

```
public GameObject[] animalPrefabs;
public int animalIndex;

void Update() {
    if (Input.GetKeyDown(KeyCode.S)) {
        int animalIndex = Random.Range(0, animalPrefabs.Length);
        Instantiate(animalPrefabs[animalIndex], new Vector3(0, 0, 20),
                    animalPrefabs[animalIndex].transform.rotation); }}
```

Step 4: Randomize the spawn location

We can press S to spawn random animals from **animalIndex**, but they all pop up in the same place! We need to randomize their spawn position, so they don't march down the screen in a straight line.

1. Replace the X value for the Vector3 with **Random.Range(-20, 20)**, then test
2. Within the **if-statement**, make a new local **Vector3 spawnPos** variable
3. At the top of the class, create **private float** variables for **spawnRangeX** and **spawnPosZ**

- **Tip:** Random.Range for floats is inclusive of all numbers in the range, while Random.Range for integers is exclusive!
- **Tip:** Keep using variables to clean your code and make it more readable

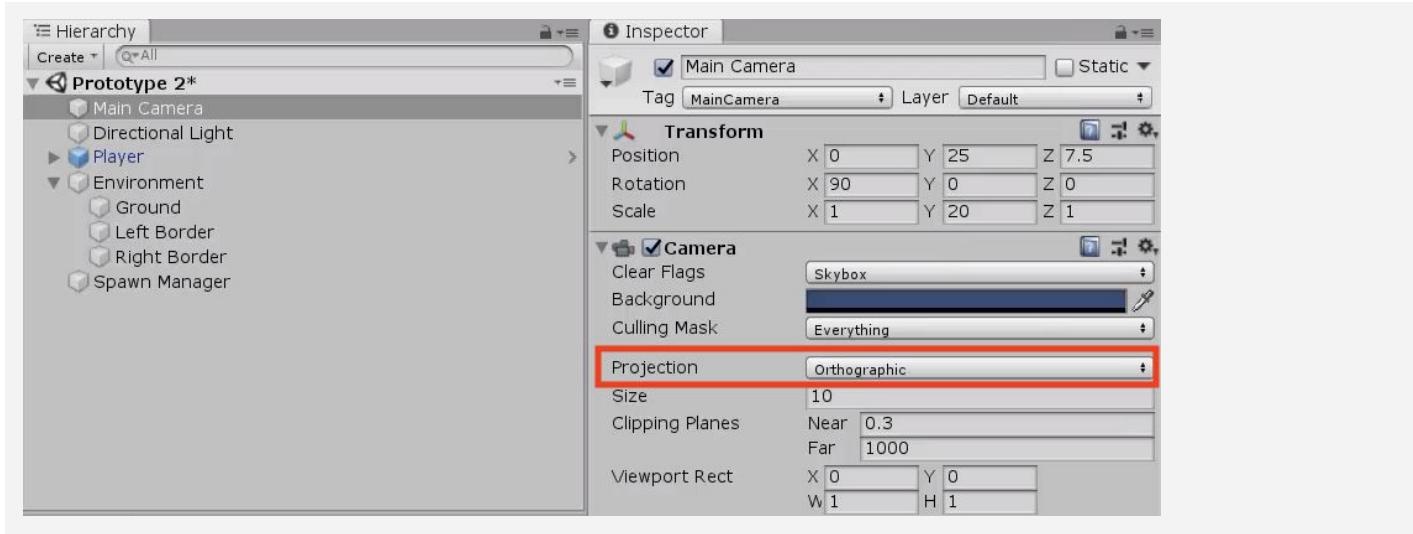
```
private float spawnRangeX = 20;
private float spawnPosZ = 20;

void Update() {
    if (Input.GetKeyDown(KeyCode.S)) {
        // Randomly generate animal index and spawn position
        Vector3 spawnPos = new Vector3(Random.Range(-spawnRangeX, spawnRangeX),
                                         0, spawnPosZ);
        int animalIndex = Random.Range(0, animalPrefabs.Length);
        Instantiate(animalPrefabs[animalIndex], spawnPos,
                    animalPrefabs[animalIndex].transform.rotation); }}
```

Step 5: Change the perspective of the camera

Our **Spawn Manager** is coming along nicely, so let's take a break and mess with the camera. Changing the camera's perspective might offer a more appropriate view for this top-down game.

1. Toggle between **Perspective** and **Isometric** view in Scene view to appreciate the difference
 2. Select the **camera** and change the **Projection** from "Perspective" to "Orthographic"
- **New:** Orthographic vs Perspective Camera Projection
 - **Tip:** Test the game in both views to appreciate the difference



Lesson Recap

New Functionality

- The player can press the S to spawn an animal
- Animal selection and spawn location are randomized
- Camera projection (perspective/orthographic) selected

New Concepts and Skills

- Spawn Manager
- Arrays
- Keycodes
- Random generation
- Local vs Global variables
- Perspective vs Isometric projections

Next Lesson

- Using collisions to feed our animals!



2.4 Collision Decisions

Steps:

Step 1: Make a new method to spawn animals

Step 2: Spawn the animals at timed intervals

Step 3: Add collider and trigger components

Step 4: Destroy objects on collision

Step 5: Trigger a "Game Over" message

Example of project by end of lesson



Length: 50 minutes

Overview: Our game is coming along nicely, but there are some critical things we must add before it's finished. First off, instead of pressing S to spawn the animals, we will spawn them on a timer so that they appear every few seconds. Next we will add colliders to all of our prefabs and make it so launching a projectile into an animal will destroy it. Finally, we will display a "Game Over" message if any animals make it past the player.

Project Outcome: The animals will spawn on a timed interval and walk down the screen, triggering a "Game Over" message if they make it past the player. If the player hits them with a projectile to feed them, they will be destroyed.

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

- Repeat functions on a timer with InvokeRepeating
- Write custom functions to make your code more readable
- Edit Box Colliders to fit your objects properly
- Detect collisions and destroy objects that collide with each other
- Display messages in the console with Debug Log

Step 1: Make a new method to spawn animals

Our Spawn Manager is looking good, but we're still pressing S to make it work! If we want the game to spawn animals automatically, we need to start by writing our very first custom function.

1. In **SpawnManager.cs**, create a new **void** **SpawnRandomAnimal()** {} function beneath **Update()**
2. Cut and paste the code from the **if-then statement** to the **new function**
3. Call **SpawnRandomAnimal();** if **S** is pressed

- **New Concept:** Custom Void Functions
- **New Concept:** Compartmentalization / Abstraction

```
void Update() {
    if (Input.GetKeyDown(KeyCode.S)) {
        SpawnRandomAnimal();
        Vector3 spawnpos... (Cut and Pasted Below) }}

void SpawnRandomAnimal() {
    Vector3 spawnpos = new Vector3(Random.Range(-xSpawnRange,
    xSpawnRange), 0, zSpawnPos);
    int animalIndex = Random.Range(0, animalPrefabs.Length);
    Instantiate(animalPrefabs[animalIndex], new Vector3(0, 0, 20) spawnpos,
    animalPrefabs[animalIndex].transform.rotation); }
```

Step 2: Spawn the animals at timed intervals

We've stored the spawn code in a custom function, but we're still pressing S! We need to spawn the animals on a timer, so they randomly appear every few seconds.

1. In **Start()**, use **InvokeRepeating** to spawn the animals based on an interval, then **test**.
2. Remove the **if-then statement** that tests for S being pressed
3. Declare new **private startDelay** and **spawnInterval** variables then playtest and tweak variable values

- **Tip:** Google "Repeating function in Unity"
- **New Function:** InvokeRepeating

```
private float startDelay = 2;
private float spawnInterval = 1.5f;

void Start() {
    InvokeRepeating("SpawnRandomAnimal", startDelay, spawnInterval); }

void Update() {
    if (Input.GetKeyDown(KeyCode.S)) {
        SpawnRandomAnimal(); } }
```

Step 3: Add collider and trigger components

Animals spawn perfectly and the player can fire projectiles at them, but nothing happens when the two collide! If we want the projectiles and animals to be destroyed on collision, we need to give them some familiar components - “colliders.”

1. Double-click on one of the **animal prefabs**, then *Add Component > Box Collider*
2. Click **Edit Collider**, then **drag** the collider handles to encompass the object
3. Check the “**Is Trigger**” checkbox
4. Repeat this process for each of the **animals** and the **projectile**
5. Add a **RigidBody component** to the projectile and uncheck “use gravity”

- **New Component:** Box Colliders
- **Warning:** Avoid Box Collider 2D
- **Tip:** Use isometric view and the gizmos to cycle around and edit the collider with a clear perspective
- **Tip:** For the Trigger to work, at least one of the objects needs a rigidbody component



Step 4: Destroy objects on collision

Now that the animals and the projectile have Box Colliders with triggers, we need to code a new script in order to destroy them on impact.

1. Create a new **DetectCollisions.cs** script, add it to each animal prefab, then **open** it
2. Before the final } add **OnTriggerEnter** function using **autocomplete**
3. In **OnTriggerEnter**, put **Destroy(gameObject);**, then test
4. In **OnTriggerEnter**, put **Destroy(other.gameObject);**

- **New Concept:** Overriding Functions
- **New Function:** OnTriggerEnter
- **Tip:** The “other” in OnTriggerEnter refers to the collider of the other object
- **Tip:** Use VS’s Auto-Complete feature for OnTriggerEnter and any/all override functions

```
void OnTriggerEnter(Collider other) {
    Destroy(gameObject);
    Destroy(other.gameObject); }
```

Step 5: Trigger a “Game Over” message

The player can defend their field against animals for as long as they wish, but we should let them know when they've lost with a “Game Over” message if any animals get past the player.

1. In DestroyOutOfBounds.cs, in the **else-if condition** that checks if the animals reach the bottom of the screen, add a Game Over message:

Debug.Log("Game Over!")

2. Clean up your code with **comments**
3. If using Visual Studio, Click *Edit > Advanced > Format document* to fix any indentation issues
(On a **Mac**, click *Edit > Format > Format Document*)

- **New Functions:** Debug.Log, LogWarning,.LogError
- **Tip:** Tweak some values to adjust the difficulty of your game. It might too easy!

```
void Update() {
    if (transform.position.z > topBound)
    {
        Destroy(gameObject);
    } else if (transform.position.z < lowerBound)
    {
        Debug.Log("Game Over!");
        Destroy(gameObject);
    }
}
```

Lesson Recap

New Functionality

- Animals spawn on a timed interval and walk down the screen
- When animals get past the player, it triggers a “Game Over” message
- If a projectile collides with an animal, both objects are removed

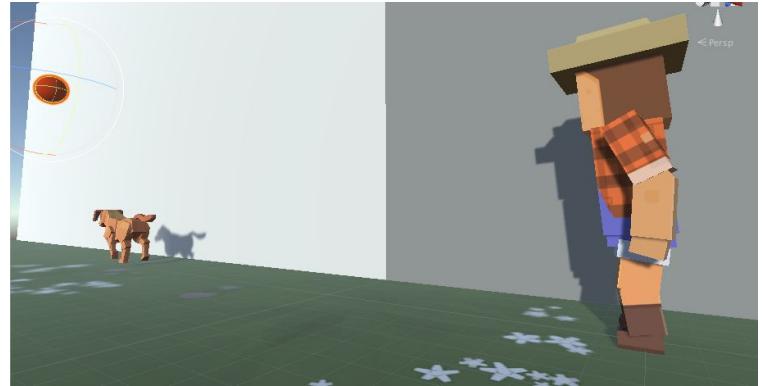
New Concepts and Skills

- Create custom methods/functions
- InvokeRepeating() to repeat code
- Colliders and Triggers
- Override functions
- Log Debug messages to console



Challenge 2

Play Fetch



Challenge Overview:

Use your array and random number generation skills to program this challenge where balls are randomly falling from the sky and you have to send your dog out to catch them before they hit the ground. To complete this challenge, you will have to make sure your variables are assigned properly, your if-statements are programmed correctly, your collisions are being detected perfectly, and that objects are being generated randomly.

Challenge Outcome:

- A random ball (of 3) is generated at a random x position above the screen
- When the user presses spacebar, a dog is spawned and runs to catch the ball
- If the dog collides with the ball, the ball is destroyed
- If the ball hits the ground, a “Game Over” debug message is displayed
- The dogs and balls are removed from the scene when they leave the screen

Challenge Objectives:

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

- Assigning variables and arrays in the inspector
- Editing colliders to the appropriate size
- Testing xyz positions with greater/less than operators in if-else statements
- Randomly generating values and selecting objects from arrays

Challenge Instructions:

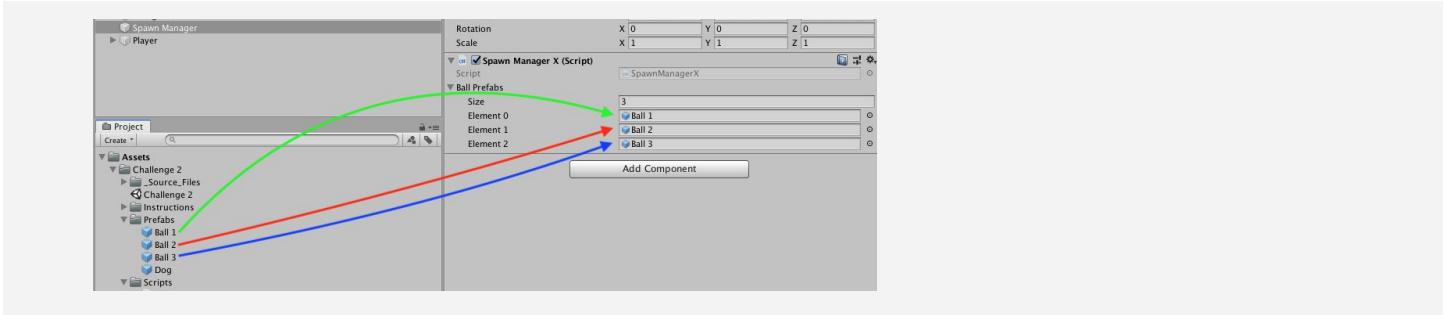
- Open your **Prototype 2** project
- **Download** the “Challenge 2 Starter Files” from the Tutorial Materials section, then double-click on it to **Import**
- In the *Project Window > Assets > Challenge 1 > Instructions* folder, use the “Challenge 2 - Instructions” and “Outcome” video as a guide to complete the challenge

Challenge	Task	Hint
1 Dogs are spawning at the top of the screen	Make the balls spawn from the top of the screen	Click on the Spawn Manager object and look at the "Ball Prefabs" array
2 The player is spawning green balls instead of dogs	Make the player spawn dogs	Click on the Player object and look at the "Dog Prefab" variable
3 The balls are destroyed if anywhere near the dog	The balls should only be destroyed when coming into direct contact with a dog	Check out the box collider on the dog prefab
4 Nothing is being destroyed off screen	Balls should be destroyed when they leave the bottom of the screen and dogs should be destroyed when they leave the left side of the screen	In the DestroyOutOfBounds script, double-check the lowerLimit and leftLimit variables, the greater than vs less than signs, and which position (x,y,z) is being tested
5 Only one type of ball is being spawned	Ball 1, 2, and 3 should be spawned randomly	In the SpawnRandomBall() method, you should declare a new random int index variable, then incorporate that variable into the Instantiate call

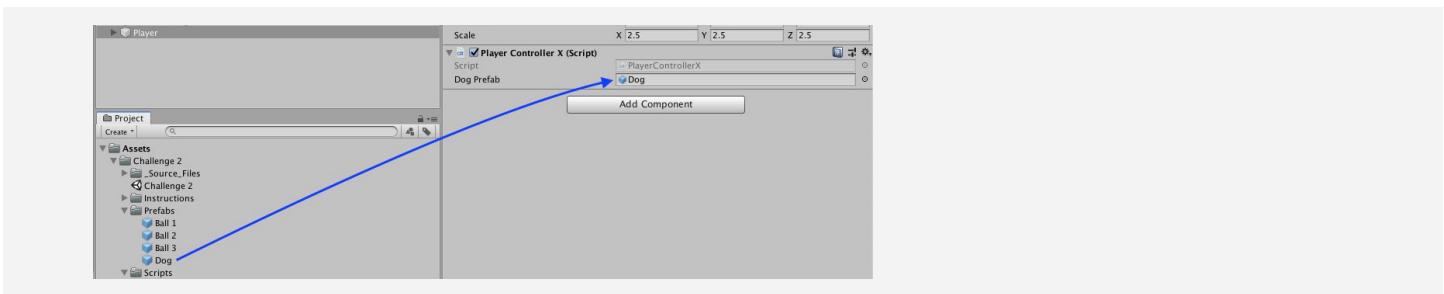
Bonus Challenge	Task	Hint
X The spawn interval is always the same	Make the spawn interval a random value between 3 seconds and 5 seconds	Set the spawnInterval value to a new random number between 3 and 5 seconds in the SpawnRandomBall method
Y The player can "spam" the spacebar key	Only allow the player to spawn a new dog after a certain amount of time has passed	Search for <code>Time.time</code> in the Unity Scripting API and look at the example. And don't worry if you can't figure it out - this is a very <i>difficult</i> challenge.

Challenge Solution

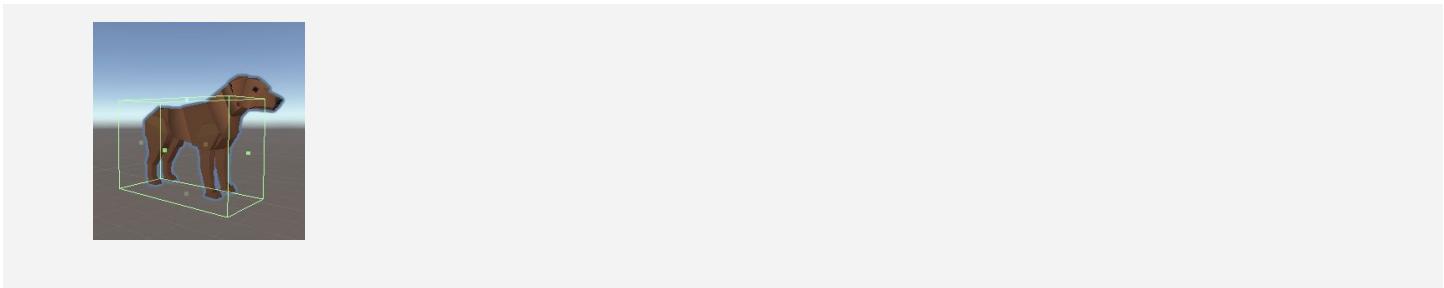
- 1 Select the Spawn Manager object and expand the “Ball Prefabs” array, then drag the **Ball 1, 2, 3** prefabs from *Assets > Challenge 2 > Prefabs* onto **Element 0, 1, 2**



- 2 Select the Player object and drag the **Dog** prefab from *Assets > Challenge 2 > Prefabs* onto the “Dog Prefab” variable



- 3 Double-click on the Dog prefab, then in the Box Collider component, click **Edit Collider**, and reduce the collider to be the same size as the dog



- 4 In *DestroyOutOfBoundsX.cs*, make the leftLimit a negative value, change the greater than to a less than when testing the x position, and test the y value instead of the z for the bottom limit

```
private float leftLimit = -30;
private float bottomLimit = -5;

void Update() {
    if (transform.position.x < leftLimit) {
        Destroy(gameObject);
    } else if (transform.position.z - y < bottomLimit) {
        Destroy(gameObject);
    }
}
```

- 5 In the SpawnRandomBall() method, declare a new random **int index** variable between 0 and the length of the Array, then incorporate that index variable into the the Instantiate call

```
void SpawnRandomBall ()  
{  
    // Generate random ball index and random spawn position  
    int index = Random.Range(0, ballPrefabs.Length);  
    Vector3 spawnPos = new Vector3(Random.Range(spawnXLeft, spawnXRight), spawnPosY, 0);  
  
    // instantiate ball at random spawn location  
    Instantiate(ballPrefabs[0-index], spawnPos, ballPrefabs[0-index].transform.rotation);  
}
```

Bonus Challenge Solution

- X1** In SpawnManagerX.cs, in SpawnRandomBall(), randomly set **spawnInterval** using Random.Range()

```
void SpawnRandomBall ()
{
    spawnInterval = Random.Range(2, 4);
    ...
}
```

- Y1** In PlayerControllerX.cs, declare and initialize new fireRate and nextFire variables. Your “fireRate” will represent the time the player has to wait in seconds, and the nextFire variable will indicate the time (in seconds since the game started) at which the player will be able to fire again (starting at 0.0)

```
public GameObject dogPrefab;
private float fireRate = 1; // time the player has to wait to fire again
private float nextFire = 0; // time since start after which player can fire again
```

- Y2** In the if-statement checking if the player pressed spacebar, add a new condition to check that Time.time (the time in seconds since the game started) is greater than nextFire (which represents the time after which the player is allowed to fire). If so, nextFire should be reset to the current time plus the fireRate.

```
// On spacebar press, if enough time has elapsed since last fire, send dog
if (Input.GetKeyDown(KeyCode.Space) && Time.time > nextFire)
{
    nextFire = Time.time + fireRate; // reset nextFire to current time +
    fireRate
    Instantiate(dogPrefab, transform.position, dogPrefab.transform.rotation);
}
```



Unit 2 Lab

New Project with Primitives

Steps:

Step 1: Create a new Unity Project

Step 2: Create a background plane

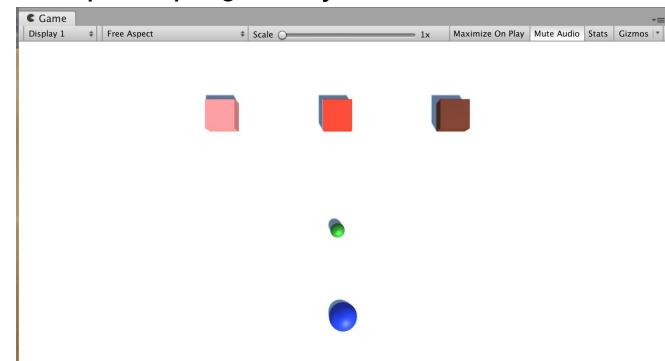
Step 3: Create primitive Player and material

Step 4: Position camera based on project type

Step 5: Enemies, obstacles, and projectiles

Step 6: Export a Unity Package backup file

Example of progress by end of lab



Length: 60 minutes

Overview: You will create and set up the project that will soon transform into your very own Personal Project. For now, you will use “primitive” shapes (such as spheres, cubes, and planes) as placeholders for your objects so that you can add functionality as efficiently as possible without getting bogged down by graphics. To make it clear which object is which, you will also give each object a unique colored material.

Project Outcome: All key objects are in the scene as primitive objects with the camera positioned properly for your project type.

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

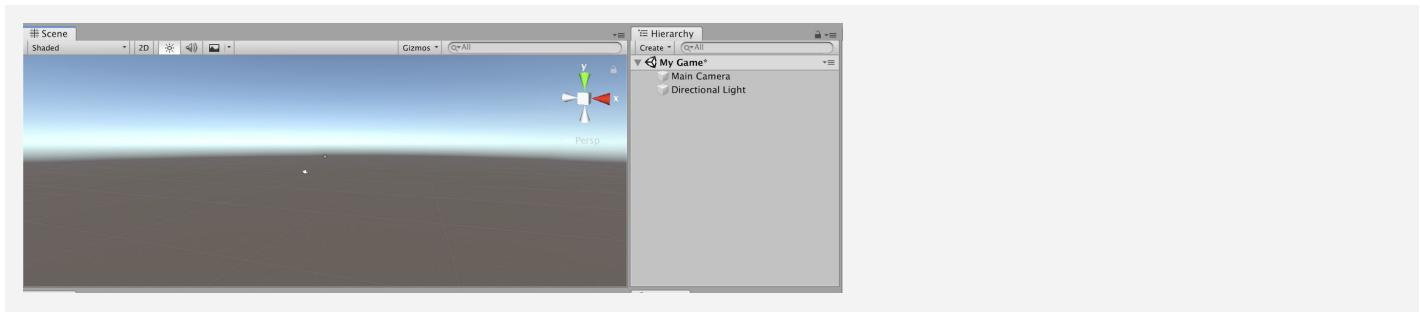
- Create a simple plane as a background for your project
- Position the camera, background, and player appropriately depending on the type of project you are creating
- Create primitive shapes to serve as placeholders for your gameobjects
- Create new colored materials and apply them to distinguish gameobjects

Step 1: Create a new Unity Project

Just like we did with the Prototype, the first thing we need to do is create a new blank project

1. Open Unity Hub and click **New**
2. Name the project “[Your Name] - Personal Project”, select the correct **version of Unity**, make sure the location is set to the new “Create with Code” folder, and that you are using the **3D** template
3. Click **Create Project**, wait for Unity to open, then select your custom **Layout**
4. In the Project window, Assets > Scenes, rename “**SampleScene**” to “My Game”

- **Tip:** If there are multiple people with the same name using the computer, might want to add last initial
- **Don't worry:** There will just be a Main camera and directional light in there

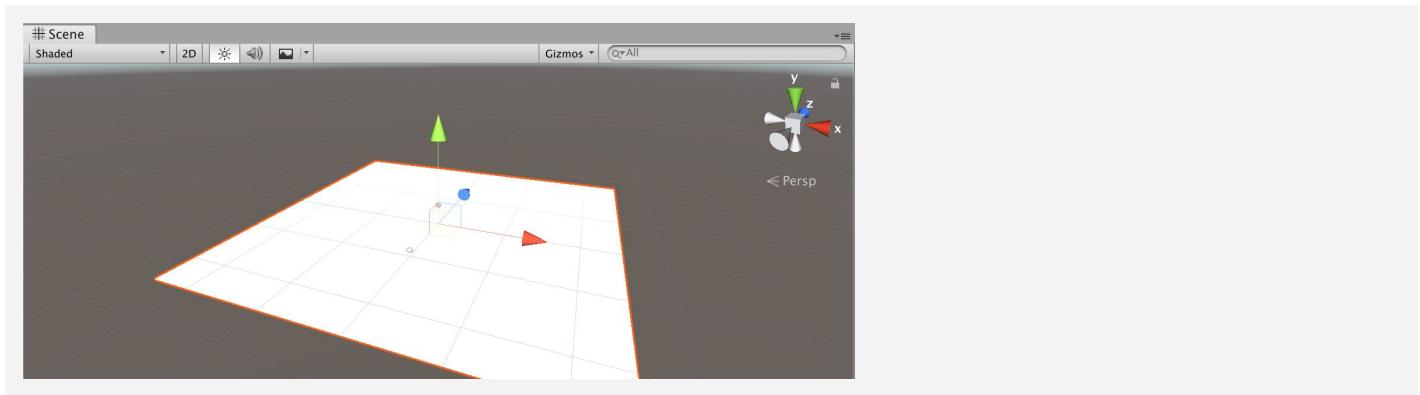


Step 2: Create a background plane

To orient yourself in the scene and not feel like you're floating around in mid-air, it's always good to start by adding a background / ground object

1. In the Hierarchy, Right-click > **3D Object** > **Plane** to add a plane to your scene
2. In the Plane's Inspector, in the top-right of the Transform component, click on the **Gear icon** > **Reset**
3. Increase the **XYZ scale** of the plane to **(5, 1, 5)**
4. Adjust your position in Scene view so you have a good view of the Plane

- **Explanation:** Working with **primitives** - these are simple objects that allow you to work faster

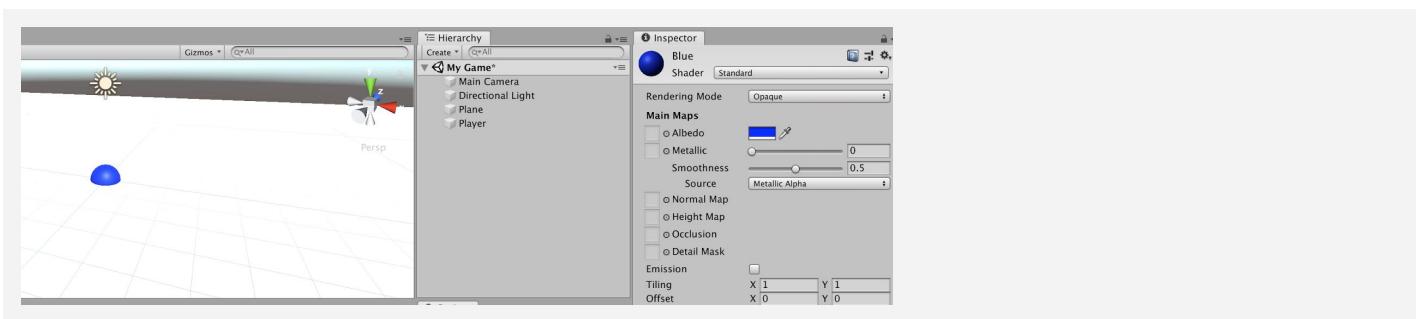


Step 3: Create primitive Player and material

Now that we have the empty plane object set up, we can add the star of the show: the player object

1. In the Hierarchy, Right-click > 3D Object > **Sphere**, then rename it “Player”
2. In Assets, Right-click > Create > **Folder** named “Materials”
3. Inside “Materials”, Right-click > Create > **Material** and rename it “Blue”
4. In Blue’s Inspector, click on the **Albedo color** box and change it to a blue
5. **Drag** the material from your Assets onto the Player object

- **Tip:** Using primitives doesn’t let graphics distract you and get in the way of core features,
- **Explanation:** Albedo is a reference to astronomical light reflection properties - but it’s basically just the material’s color
- **Warning:** Stick with blue right now so it’s easy to follow - you’ll be replacing it later

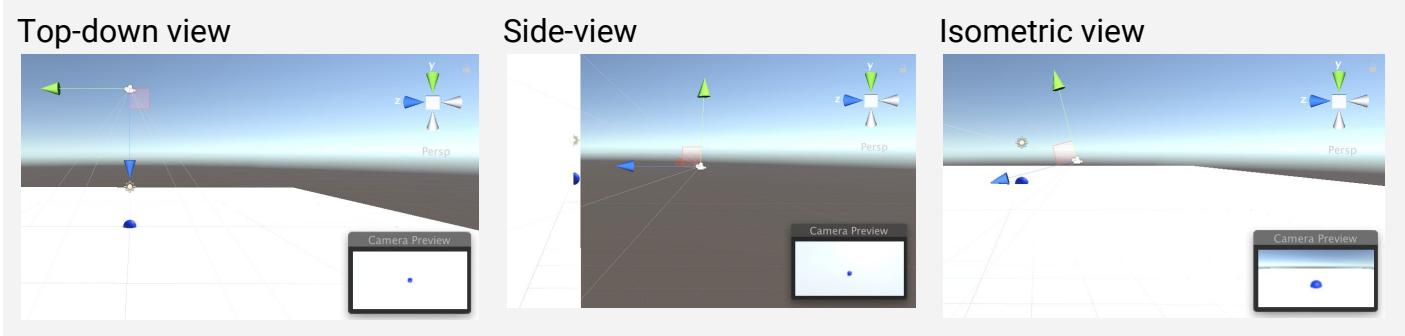


Step 4: Position camera based on project type

Now that we have the player in there, we need the best view of it, depending on our type of project

1. For a **top-down** game, position the camera at **(0, 10, 0)** directly over the player and rotate it **90** degrees on the **X axis**
2. For a **side-view** game, rotate the **Plane** by **-90** degrees on the **X axis**
3. For a **third-person** view game, move the camera up on the **Y and Z axes** and increase its **rotation on the X axis**

- **Tip:** Side view looks like top view, but it’ll make a big diff when you apply gravity
- **Don’t worry:** You might not know exact view yet - just go with what’s in your design doc

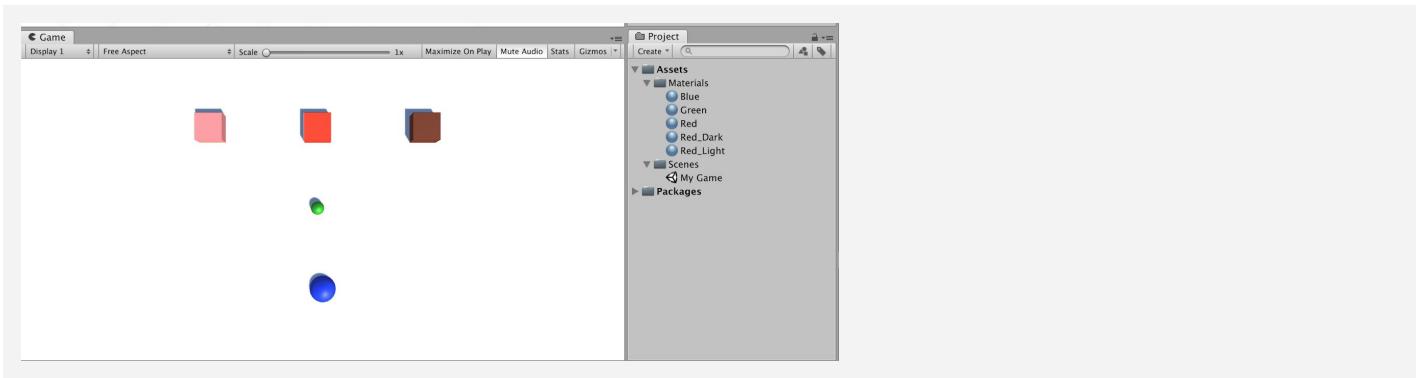


Step 5: Enemies, obstacles, and projectiles

Now that we know how to make primitives, let's go ahead and make one for each object in our project

1. In the Hierarchy, create new **Cubes**, **Spheres**, and **Capsules** for all other main objects, **renaming** them, **repositioning** them, and **scaling** them
2. In your Materials folder, create **new materials** for as many colors as you have unique objects, editing their color to match their name, then **apply** those materials to your objects
3. Position all of your objects in locations relative to each other that make sense

- **Tip:** If you plan on having variants of certain objects (e.g. multiple animals), create dark/light shades of the same color
- **Tip:** Good to make enemies red - easy if everyone uses the same conventions

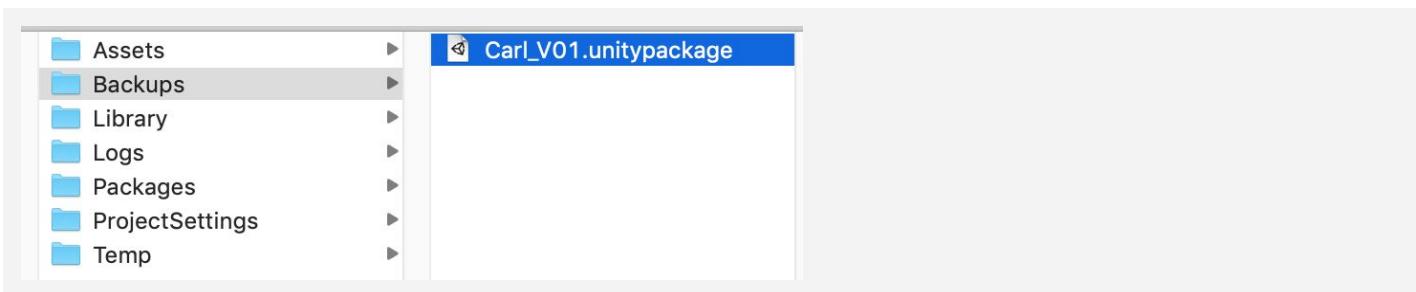


Step 6: Export a Unity Package backup file

Since we're going to be putting our hearts and souls into this project, it's always good to make backups

1. **Save** your Scene
2. In the Project window, Right-click on the "Assets" folder > **Export Package**, then click Export
3. Create a **new "Backups" folder** in your Personal Project folder, then **save** it with your name and the version number (e.g. Carl_V0.1.unitypackage")

- **Explanation:** The "include dependencies" checkbox will include any files that are tied to / used by anything else we're exporting
- **Tip:** This is the same file type that you imported at the start of Prototype 1



Lesson Recap

New Progress

- New project for your Personal Project
- Camera positioned and rotated based on project type
- All key objects in scene with unique materials

New Concepts and Skills

- Primitives
- Create new materials
- Export Unity packages



Quiz Unit 2

QUESTION

CHOICES

- 1 If it says, "Hello there!" in the console, what was the code used to create that message?

Hello there!

- a. Debug("Hello there!");
- b. Debug.Log("Hello there!");
- c. Debug.Console("Hello there!");
- d. Debug.Log>Hello there!);

- 2 If you want to destroy an object when its **health reaches 0**, what code would be best in the blank below?

```
private int health = 0;

void Update() {
    if (_____) {
        Destroy(gameObject);
    }
}
```

- a. health > 0
- b. health.0
- c. health < 1
- d. health < 0

- 3 The code below creates an error that says, "error CS1503: Argument 1: cannot convert from 'UnityEngine.GameObject[]' to 'UnityEngine.Object''. What could you do to remove the errors?

```
1. public GameObject[] enemyPrefabs;
2.
3. void Start()
4. {
5.     Instantiate(enemyPrefabs);
6. }
```

- a. On line 1, change "GameObject[]" to "GameObject"
- b. On line 1, change "enemyPrefabs" to "enemyPrefabs[0]"
- c. On line 3, change "Start()" to "Update()"
- d. On line 5, change "enemyPrefabs" to "enemyPrefabs[0]"
- e. Either A or D
- f. Both A and D
- g. Both B and C

4 Which comment best describes the following code?

```
public class PlayerController : MonoBehaviour
{
    // Comment
    private void OnTriggerEnter(Collider other) {
        Destroy(other.gameObject);
    }
}
```

- a. // If player collides with another object, destroy player
- b. // If enemy collides with another object, destroy the object
- c. // If player collides with a trigger, destroy trigger
- d. // If player collides with another object, destroy the object

5 If you want to move the character **up continuously** as the player presses the **up arrow**, what code would be best in the two blanks below:

```
if (Input._____())
{
    transform.Translate(Vector3.up);
}
```

- a. GetKey(KeyCode.UpArrow)
- b. GetKeyDown(KeyCode.UpArrow)
- c. GetKeyUp(KeyCode.Up)
- d. GetKeyHeld(KeyCode.Up)

6 Read the documentation from the Unity Scripting API and the code below. Which of the following are possible values for the randomFloat and randomInt variables?

```
public static float Range(float min, float max);
```

Description

Return a random float number between `min` [inclusive] and `max` [inclusive] (Read Only).

Note `max` is inclusive. `Random.Range(0.0f, 1.0f)` can return `1.0` as the value. The `Random.Range` distribution is uniform. `Range` is a Random Number Generator.

```
public static int Range(int min, int max);
```

Description

Return a random integer number between `min` [inclusive] and `max` [exclusive] (Read Only).

Note `max` is exclusive. `Random.Range(0, 10)` can return a value between `0` and `9`. Return `min` if `max` equals `min`. The `Random.Range` distribution is uniform. `Range` is a Random Number Generator.

```
float randomFloat = Random.Range(0, 100);
int randomInt = Random.Range(0, 100);
```

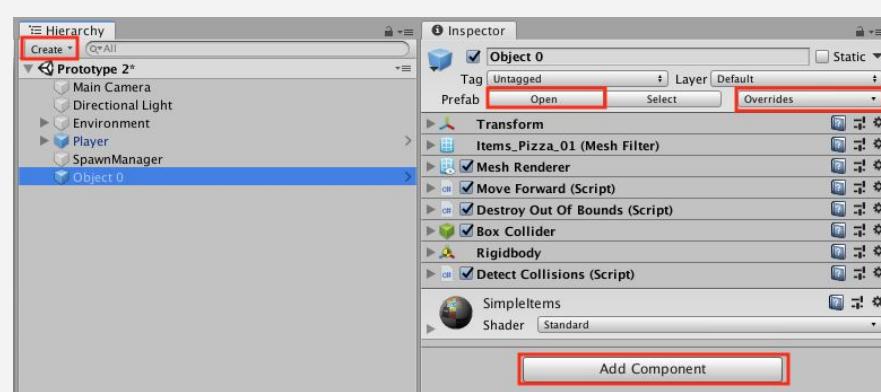
- a. `randomFloat = 100.0f;`
`randomInt = 0;`
- b. `randomFloat = 100.0f;`
`randomInt = 100;`
- c. `randomFloat = 50.5f;`
`randomInt = 100;`
- d. `randomFloat = 0.0f;`
`randomInt = 50.5;`

7 Your game is running and you see an error in the console that says there was an “error at Assets/Scripts/SpawnManager.cs:5. IndexOutOfRangeException: Index was outside the bounds of the array.” Which line of code needs to be edited to fix this?

1. `public GameObject[] randomObjects;`
- 2.
3. `void SpawnRandomObject() {`
4. `int objectIndex = Random.Range(0, 3);`
5. `Instantiate(randomObjects[objectIndex]);`
6. `}`

- a. Line 2
- b. Line 3
- c. Line 4
- d. Line 5

8 If you have made changes to a prefab in the scene and you want to apply those changes to all prefabs, what should you click?



- a. The “Create” drop-down at the top of the Hierarchy
- b. The “Open” button at the top of the Inspector
- c. The “Override” drop-down at the top of the Inspector
- d. The “Add Component” button at the bottom of the Inspector

9 Read the documentation from the Unity Scripting API below. Which of the following is a correct use of the `InvokeRepeating` method.

```
public void InvokeRepeating(string methodName, float time, float repeatRate);
```

Description

Invokes the method `methodName` in `time` seconds, then repeatedly every `repeatRate` seconds.

- a. `InvokeRepeating("Spawn", 0.5f, 1.0f");`
- b. `InvokeRepeating("Spawn", 0.5f, 1.0f);`
- c. `InvokeRepeating("Spawn", gameObject, 1.0f);`
- d. `InvokeRepeating(0.5f, 1.0f, "Spawn");`

10 You're trying to create some logic that will tell the user to speed up if they're going too slow or to slow down if they're going too fast. How should you arrange the lines of code below to accomplish that?

```
1. Debug.Log(speedUp); }  
2. else if (speed > 60) {  
3.     private string speedUp = "Speed up!";  
4.     void Update() {  
5.         Debug.Log(slowDown); }  
6.     if (speed < 10) {  
7.         private float speed;  
8.         private string slowDown = "Slow down!";  
9.     }
```

- a. 4, 6, 1, 2, 5, 9, 7, 8, 3
- b. 6, 1, 2, 5, 7, 8, 3, 4, 9
- c. 7, 8, 3, 4, 6, 5, 2, 1, 9
- d. 7, 8, 3, 4, 6, 1, 2, 5, 9

Quiz Answer Key

#	ANSWER	EXPLANATION
1	B	Debug.Log() prints messages to the console and can accept String parameters between quotation marks, such as "Hello there!"
2	C	Since the "health" variable is an int, anything less than 1 would be "0". The sign for "less than" is "<".
3	E	"GameObject[]" is a GameObject array. You cannot instantiate an array, but you <i>can</i> instantiate an object inside an array. So you could either remove the array and have Instantiate use an individual object (option A) or you could use an GameObject index of that Array (option D), but both would not work.
4	D	Since it's inside the PlayerController class, and it is destroying other.gameObject , it is destroying something that the player collides with.
5	A	"Input.GetKey" tests for the user holding down a key (as opposed to KeyKeyDown, which test for a single press down of a Key).
6	A	As it says in the documentation, Random.Range does <i>not</i> include the maximum value for integers, but <i>does</i> include the maximum value for floats. This means that randomInt <i>cannot</i> be 100, but randomFloat can be.
7	C	Line 4, which generates the objectIndex, must be generating an index value that is too high for the number of objects in the array. The best thing to do would be to change it to "Random.Range(0, randomObjects.Length);"
8	C	The "Override" drop-down will allow you to apply any changes you've made to your individual prefab to the original prefab object.
9	B	According to the Scripting API, InvokeRepeating requires a string parameter, then two floats.
10	D	All variables should be declared first, then the void method, then the if-condition telling them to speed up, then the else condition telling them to slow down.



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

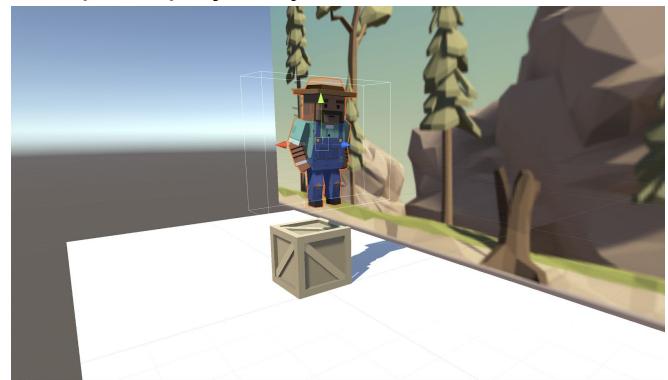
Step 6: Prevent player from double-jumping

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.

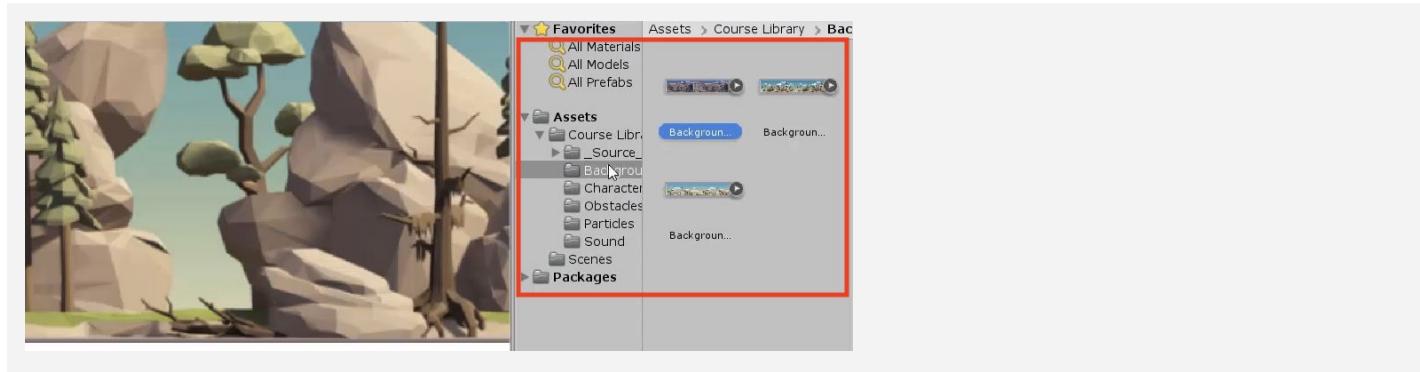
Learning Objectives: By the end of this lesson, you will be able to:

- 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.

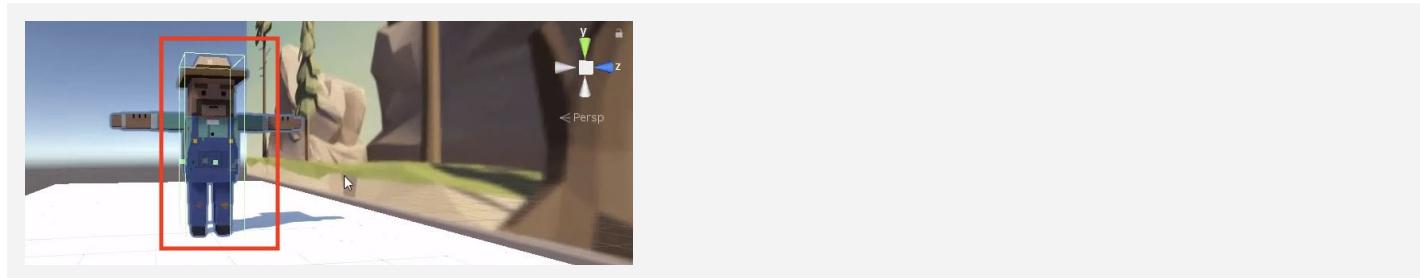
1. Open **Unity Hub** and create a new “Prototype 3” project in your course directory
2. Click on the **link** to access the Prototype 3 starter files, then **download and import** them into Unity
3. Open the Prototype 3 scene and **delete** the **Sample Scene** without saving
4. Select the **Background object** in the hierarchy, then in the **Sprite Renderer component > Sprite**, select the _City, _Nature, or _Town image



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



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

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 **rigidRb = 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 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 * 10 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**

```
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 - **New Concept:** `RigidBody` constraints 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 Rigid Body component, expand **Constraints**, then **Freeze** all but the Y position

```
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	<ul style="list-style-type: none"> • Player jumps on spacebar press • Player cannot double-jump • Obstacles and Background move left • Obstacles spawn on intervals
New Concepts and Skills	<ul style="list-style-type: none"> • GetComponent • ForceMode.Impulse • Physics.Gravity • Rigidbody constraints • Rigidbody variables • Booleans • Multiply/Assign ("*") Operator • And (&&) Operator • OnCollisionEnter()
Next Lesson	<ul style="list-style-type: none"> • We're going to fix that weird effect we created by moving the background left by having it actually constantly scroll using code!



3.2 Make the World Whiz By

Steps:

- Step 1: Create a script to repeat background
- Step 2: Reset position of background
- Step 3: Fix background repeat with collider
- Step 4: Add a new game over trigger
- Step 5: Stop MoveLeft on gameOver
- Step 6: Stop obstacle spawning on gameOver
- Step 7: Destroy obstacles that exit bounds

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:

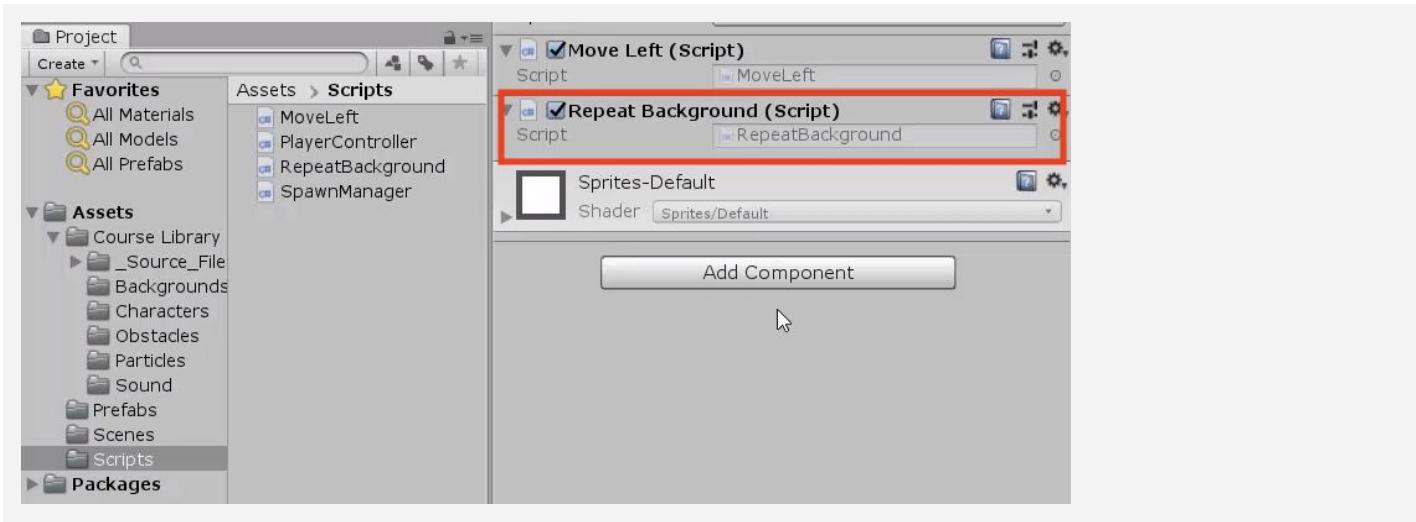
- Use tags to label game objects and call them in the code
- Use script communication to access the methods and variables of other scripts

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**

```
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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Repeat background • Get Collider width • Script communication • Equal to (==) operator • Tags • CompareTag()
Next Lesson	<ul style="list-style-type: none"> • 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



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:

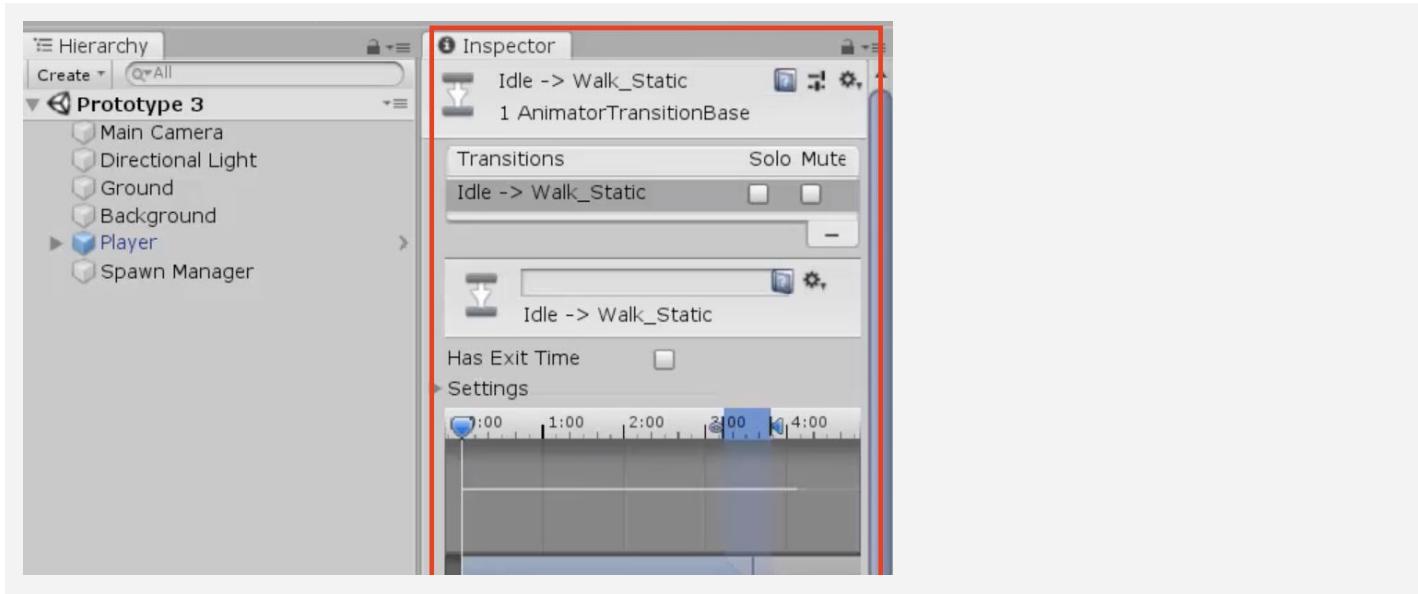
- Manage basic animation states in the Animator Controller
- Adjust the speed of animations to suit the character or the game
- Set a default animation and trigger others with `anim.SetTrigger`
- Set a permanent state for "Game Over" with `anim.SetBool`

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

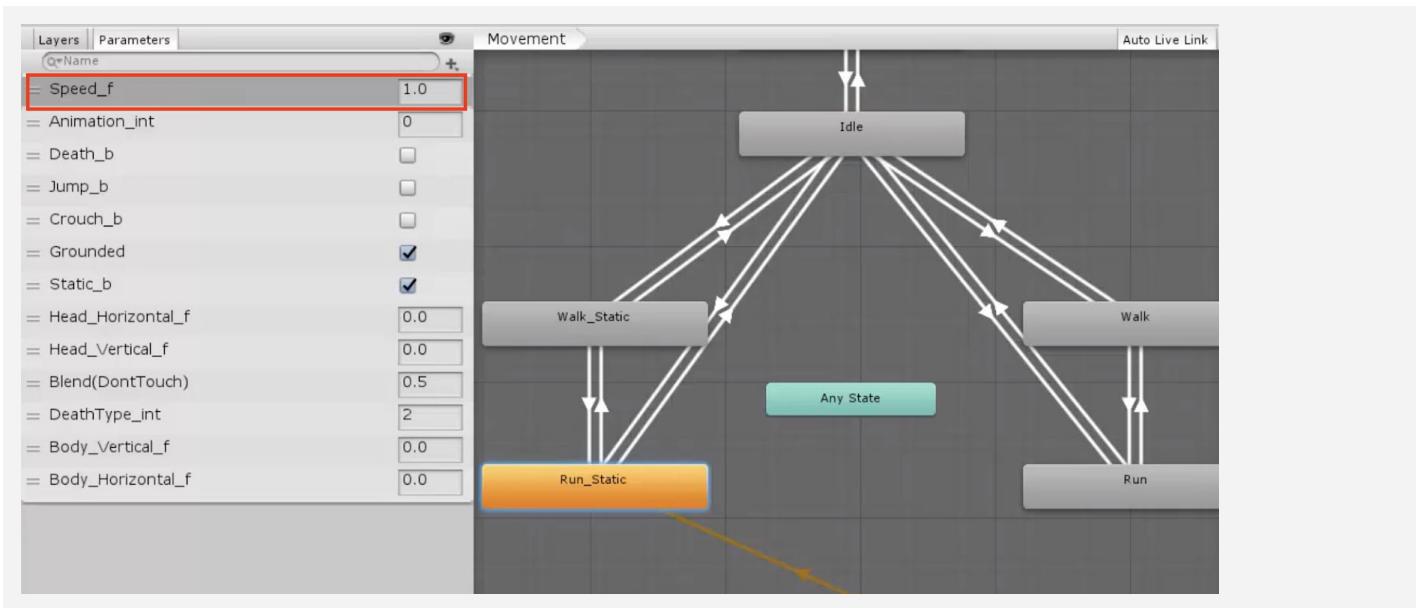


Step 2: Make the player start off at a run

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;

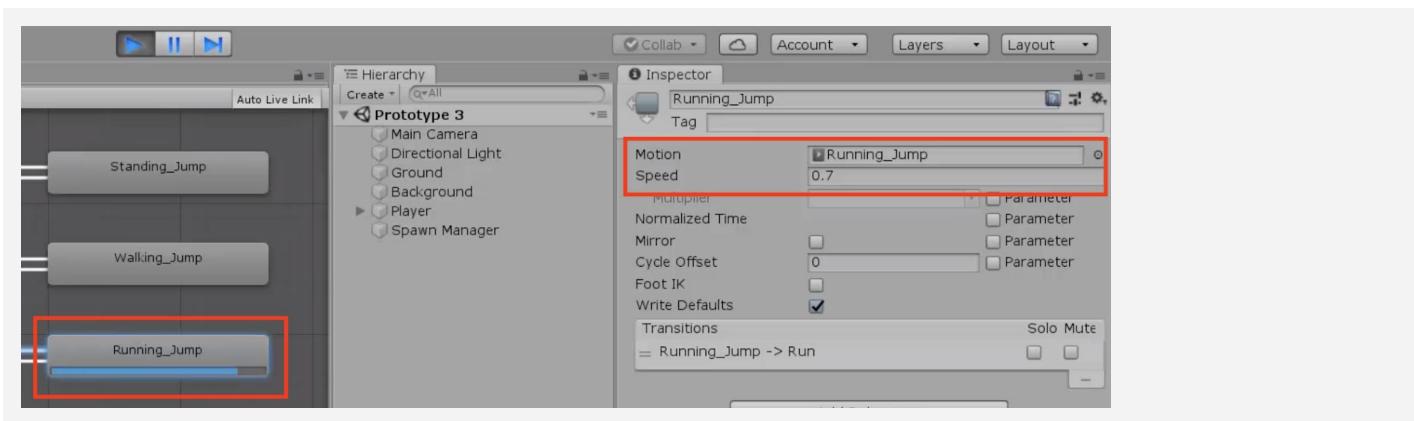
void Start() {
    playerRb = GetComponent<Rigidbody>();
    playerAnim = GetComponent<Animator>();
    Physics.gravity *= gravityModifier; }

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
 - **New Function:** anim.SetBool
 - **New Function:** anim.SetInt
2. In the same **if-statement**, set the **DeathType** integer to 1

```
public bool gameOver = false;

private void OnCollisionEnter(Collision collision other) {
    if (other.gameObject.CompareTag("Ground")) {
        isOnGround = true;
    } else if (other.gameObject.CompareTag("Obstacle")) {
        Debug.Log("Game Over")
        gameOver = true;
        playerAnim.SetBool("Death_b", true);
        playerAnim.SetInt("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 * 10 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!



3.4 Particles and Sound Effects

Steps:

Step 1: Customize an explosion particle

Step 2: Play the particle on collision

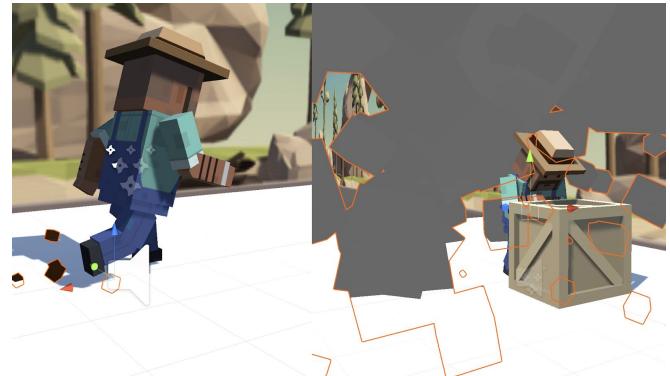
Step 3: Add a dirt splatter particle

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:

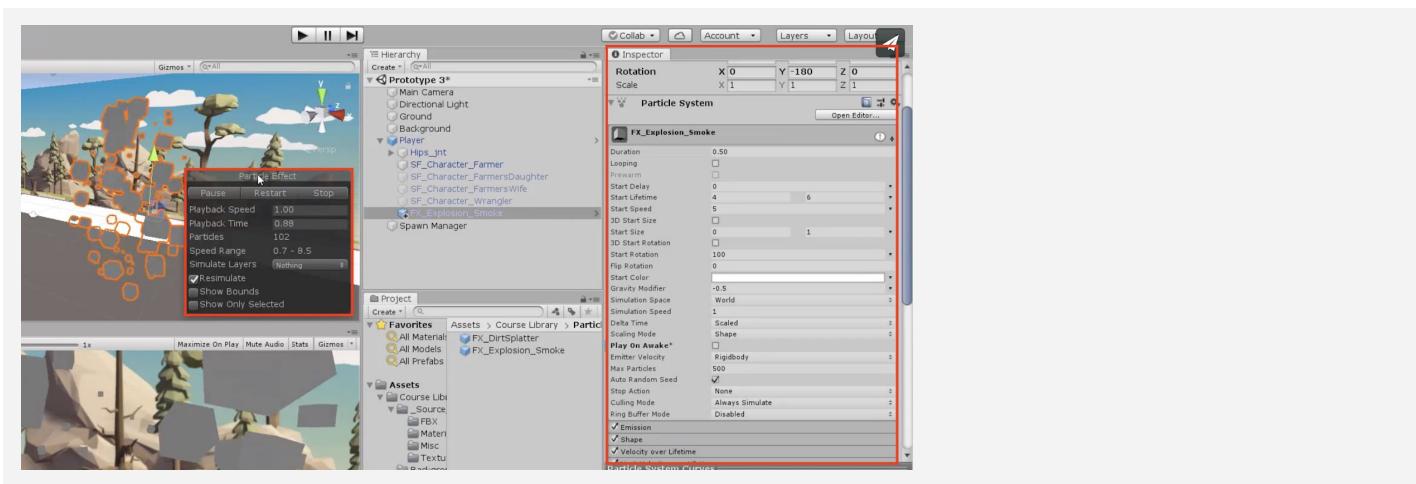
- Attach particle effects as children to game objects
- Stop and play particle effects to correspond with character animation states
- Work with Audio Sources and Listeners to play background music
- Add sound effects to add polish to your project

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 **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

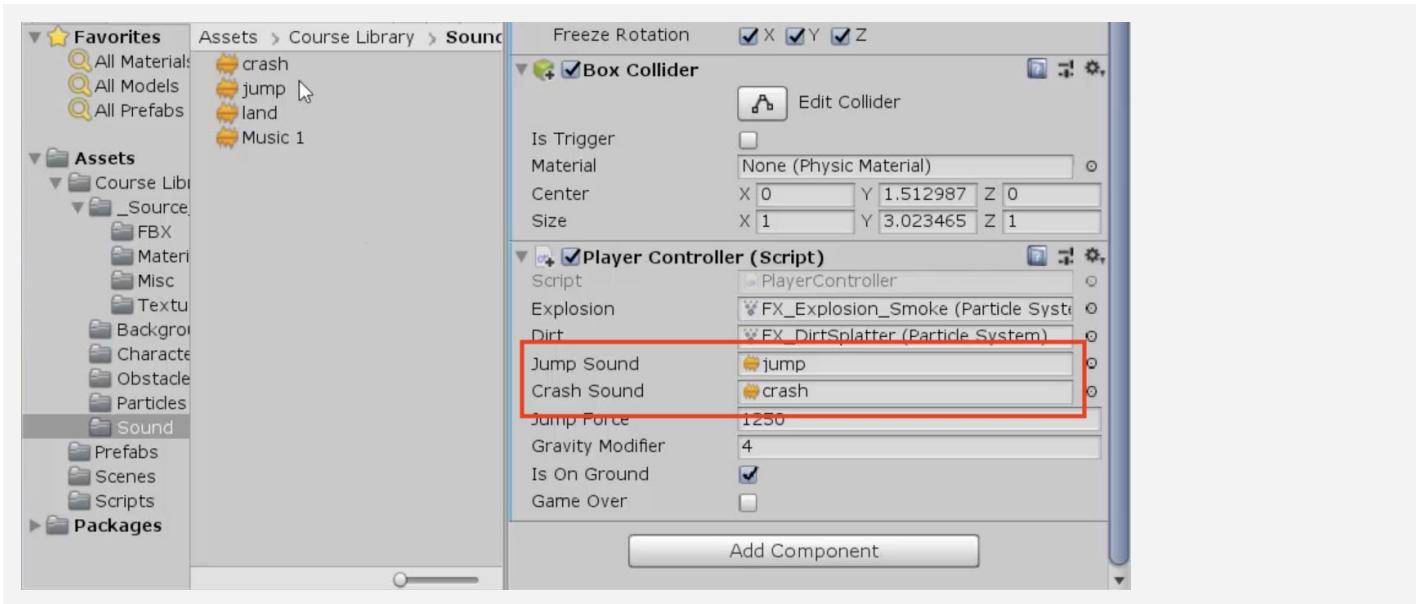


Step 5: Declare variables for Audio Clips

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 **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<**
 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<

```

Lesson Recap

New Functionality	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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:	<ul style="list-style-type: none"> - 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:	<p>In this challenge, you will reinforce the following skills/concepts:</p> <ul style="list-style-type: none"> - 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:	<ul style="list-style-type: none"> - 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 <i>Project Window > Assets > Challenge 3 > Instructions</i> 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 "NullReferenceException" error on the player's <code>rigidBody</code> variable - it has to be assigned in <code>Start()</code> using the <code>GetComponent<></code> method
2 The background only moves when the game is over	The background should move at start, then <i>stop</i> when the game is over	In <code>MoveLeftX.cs</code> , the objects should only <code>Translate</code> to the left if the game is <i>NOT</i> 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: <code>SpawnManagerX.PrawnsObject</code> 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 <code>repeatWidth</code> variable should be half of the background's <code>width</code> , not half of its <code>height</code>

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 <code>isLowEnough</code> , 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 >();
```

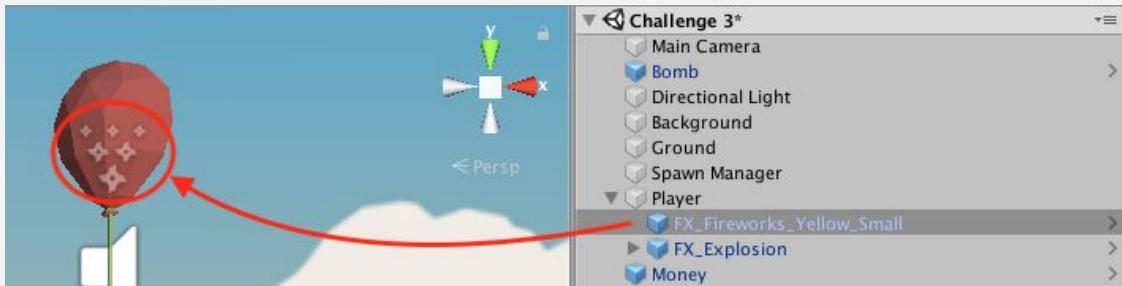
- 2 In MoveLeftX.cs, the objects should only Translate to the left if the game is NOT over - it's currently checking if the game IS over:

```
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 / 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**

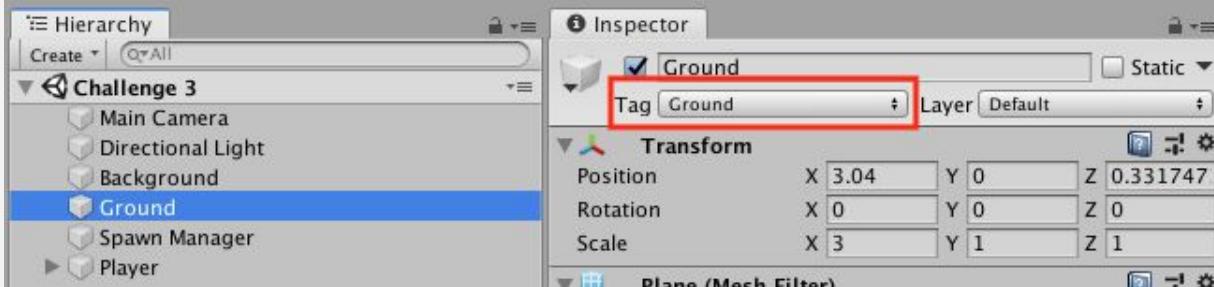
```
public bool isLowEnough;

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.GetKeyDown(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);  
}
```



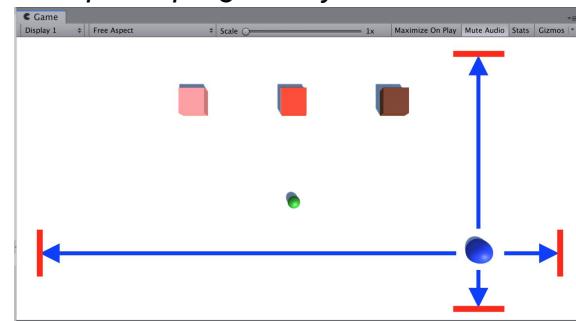
Unit 3 Lab

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

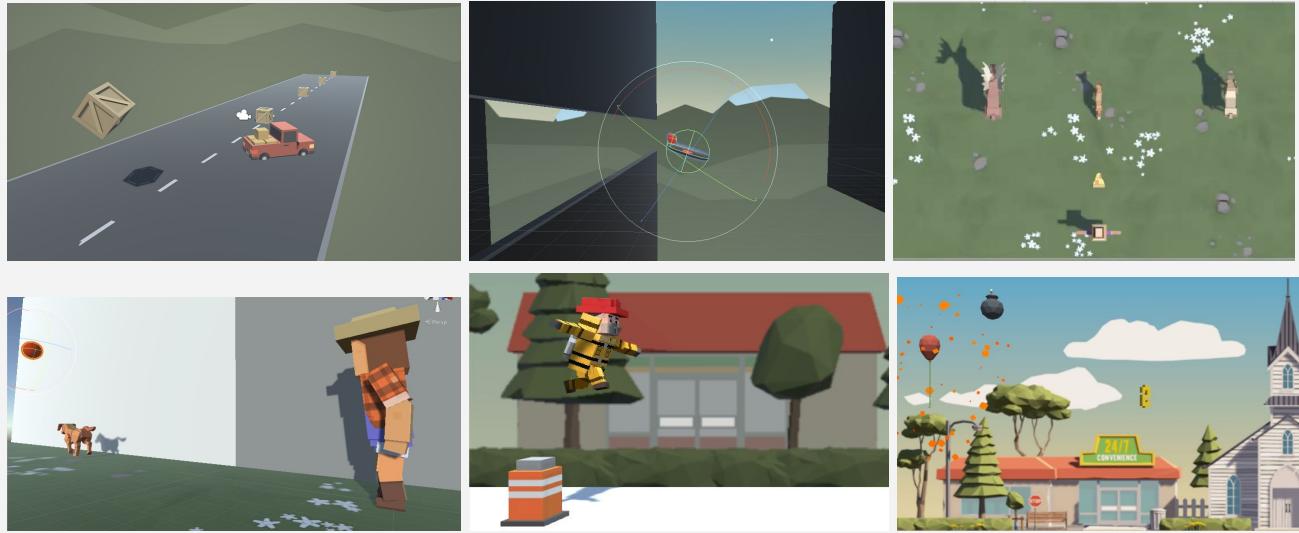
Step 1: Create PlayerController and plan your code

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



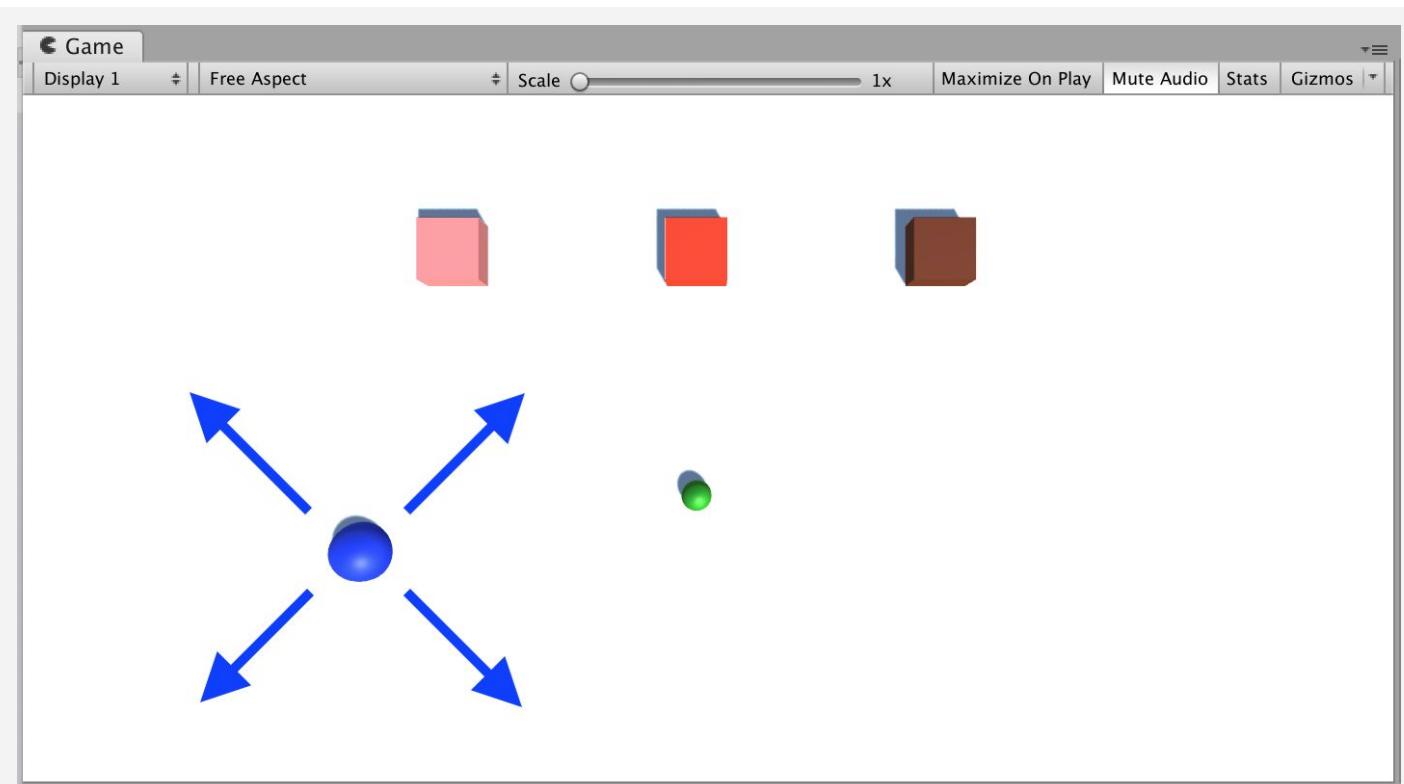
By the end of this step, you should have a new Script open and a solid plan for what will go in it.

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



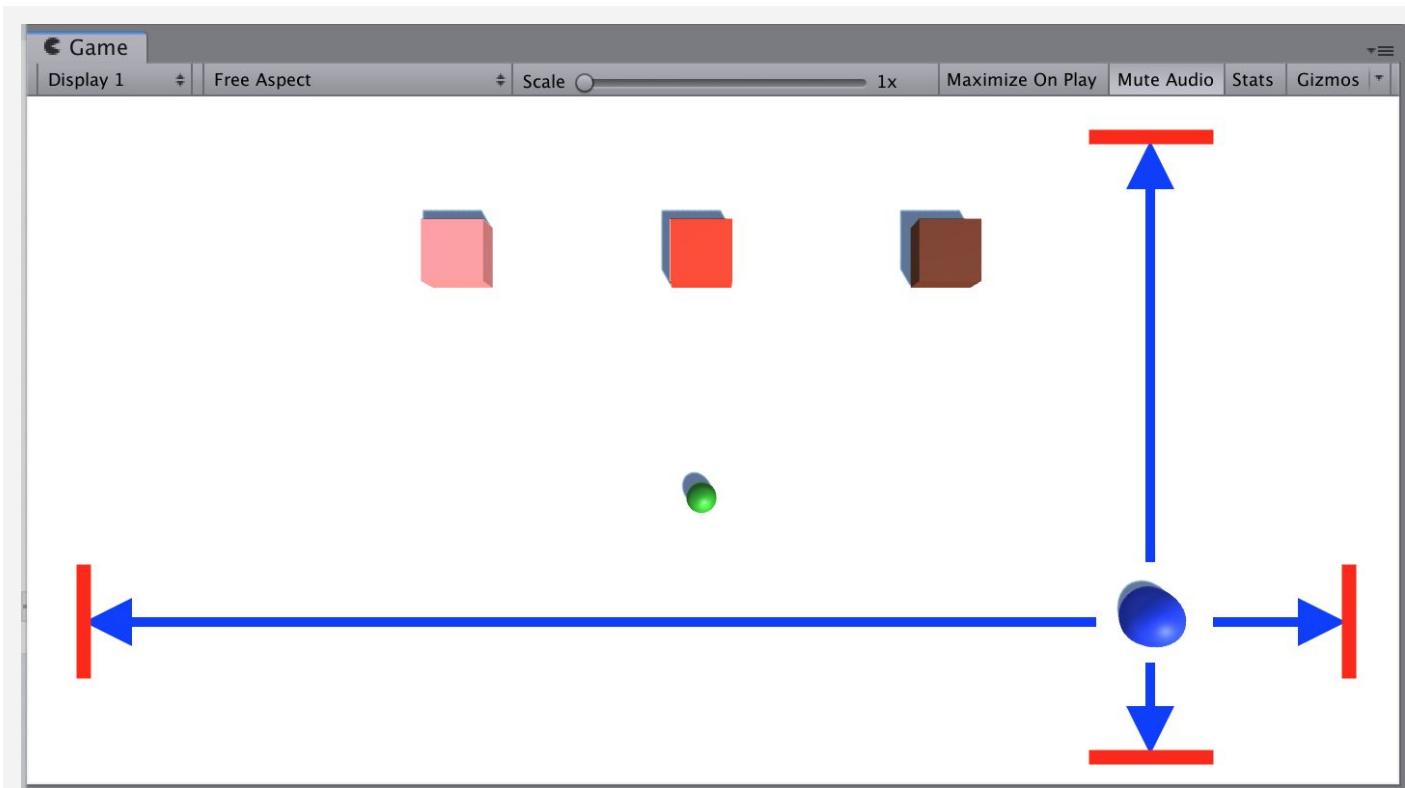
By the end of this step, the player should be able to move the way that you want based on user input.

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



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();
        }
    }
}
```

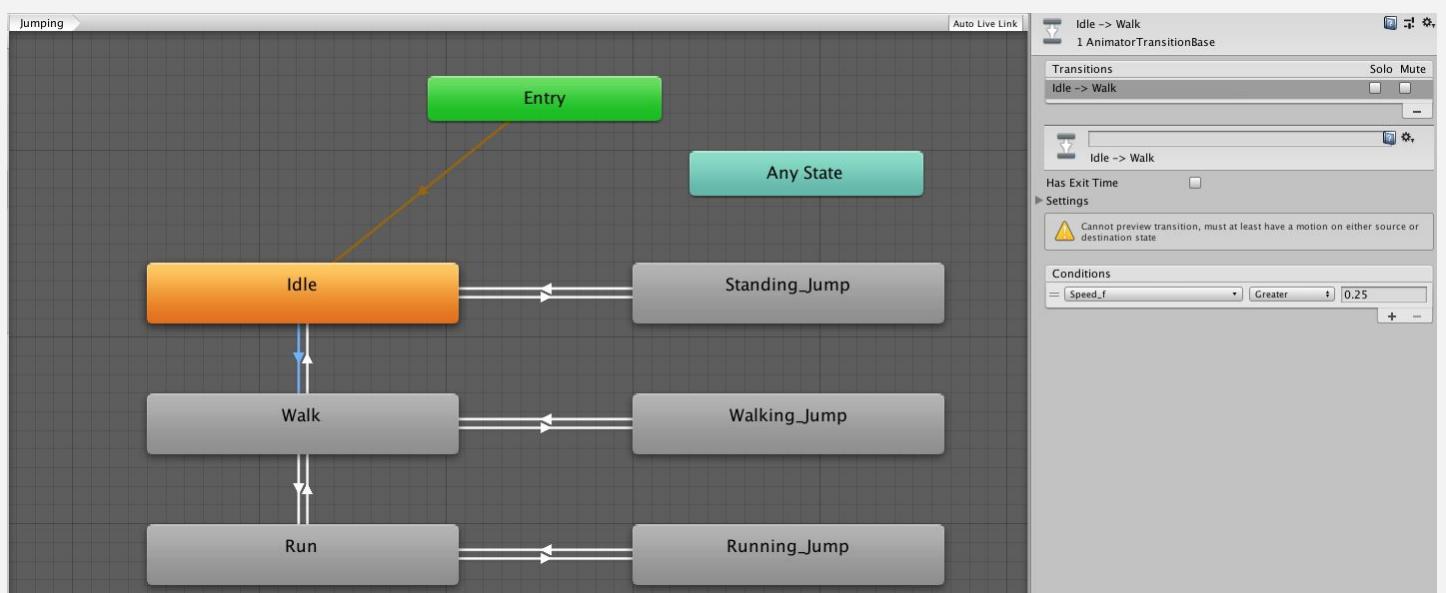
- 2** Match the following animation methods with its set of parameters

- | | |
|----------------------------|-------------------|
| 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

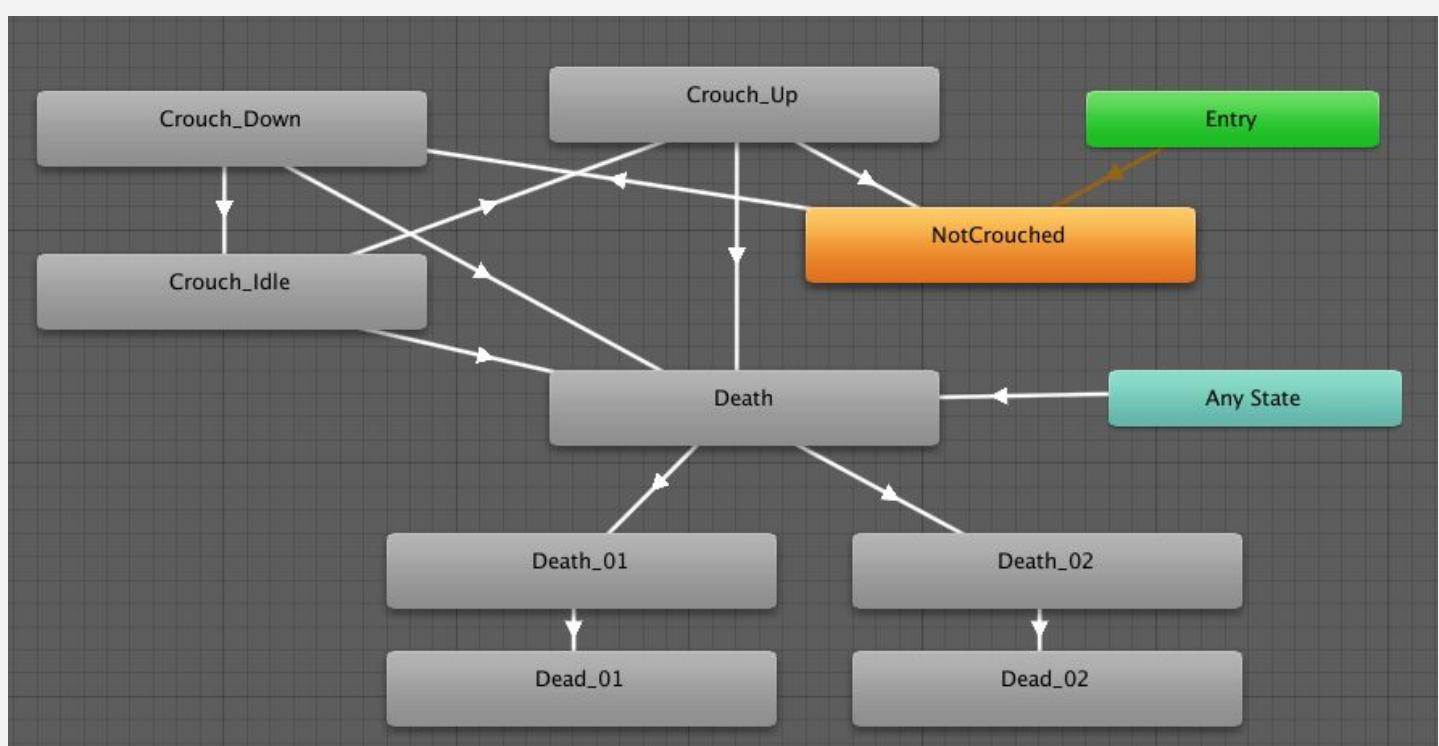
6 Which of the following conditions properly tests that the game is NOT over and the player IS on the ground

- 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"

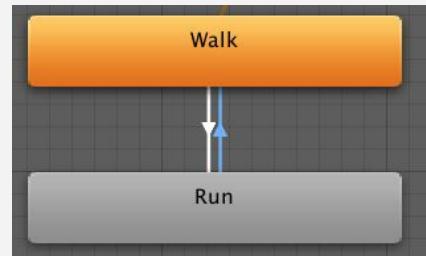


8 Which of the following variable declarations observes Unity's standard naming conventions (especially as it relates to capitalization)?

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?

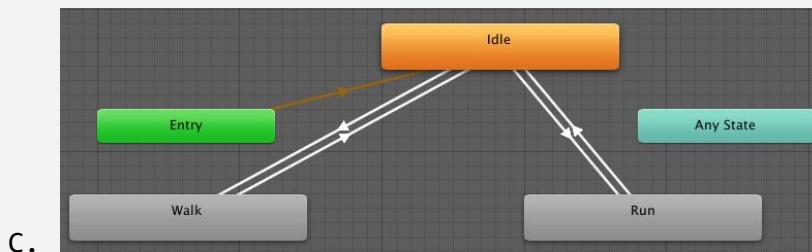
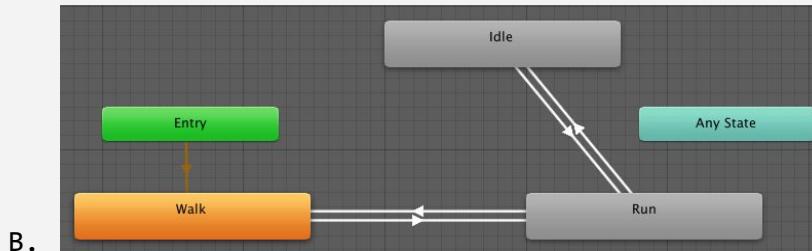
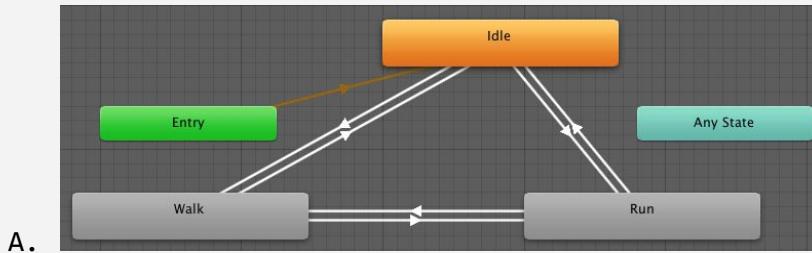


- a. Jump_b is true
- b. Speed_f is Less than 0.5
- c. Speed_f is Greater than 0.5
- d. Animation_int is Less than 10

- A. = Jump_b true
- B. = Speed_f Less 0.5
- C. = Speed_f Greater 0.5
- D. = Animation_int Less 10

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



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	<code>!=</code> means “does not equal to”, so “gameOver != true” is testing that the game is <i>not</i> 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 “ <code>&&</code> ”.
7	B	The default starting state is the one that the “Entry” state connects to.
8	D	<ol style="list-style-type: none"> 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 <i>less</i> 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.



4.1 Watch Where You're Going

Steps:

Step 1: Create project and open scene

Step 2: Set up the player and add a texture

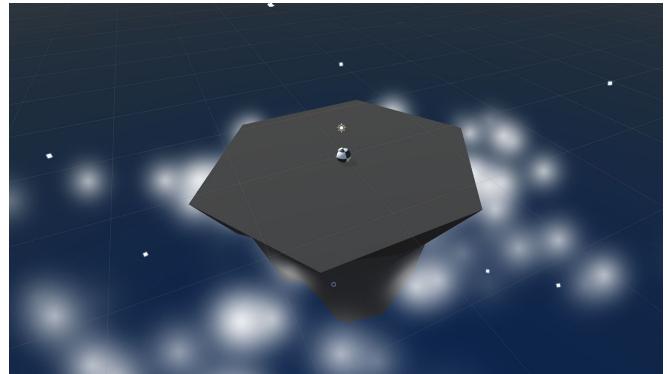
Step 3: Create a focal point for the camera

Step 4: Rotate the focal point by user input

Step 5: Add forward force to the player

Step 6: Move in direction of focal point

Example of project by end of lesson



Length: 60 minutes

Overview: First thing's first, we will create a new prototype and download the starter files! You'll notice a beautiful island, sky, and particle effect... all of which can be customized! Next you will allow the player to rotate the camera around the island in a perfect radius, providing a glorious view of the scene. The player will be represented by a sphere, wrapped in a detailed texture of your choice. Finally you will add force to the player, allowing them to move forwards or backwards in the direction of the camera.

Project Outcome: The camera will evenly rotate around a focal point in the center of the island, provided a horizontal input from the player. The player will control a textured sphere, and move them forwards or backwards in the direction of the camera's focal point.

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

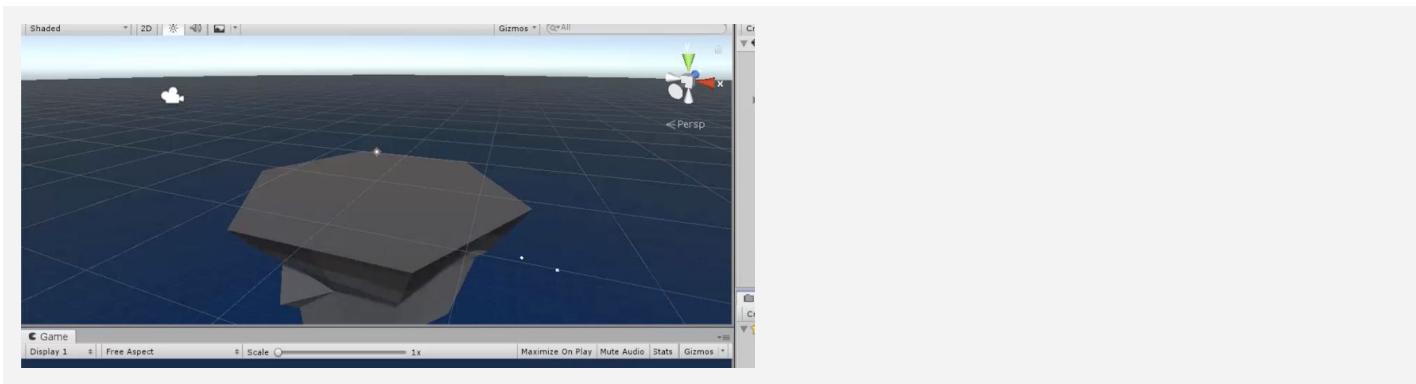
- Apply Texture wraps to objects
- Attaching a camera to its focal point using parent-child relationships
- Transform objects based on local XYZ values

Step 1: Create project and open scene

You've done it before, and it's time to do it again... we must start a new project and import the starter files.

1. Open **Unity Hub** and create "Prototype 4" in your course directory
2. Click on the link to access the Prototype 4 **starter files**, then **download and import** them into Unity
3. Open the **Prototype 4 scene** and delete the **Sample Scene** without saving
4. Click **Run** to see the particle effects

- **Don't worry:** You can change texture of floating island and the color of the sky later
- **Don't worry:** We're in isometric/orthographic view for a reason: It just looks nicer when we rotate around the island

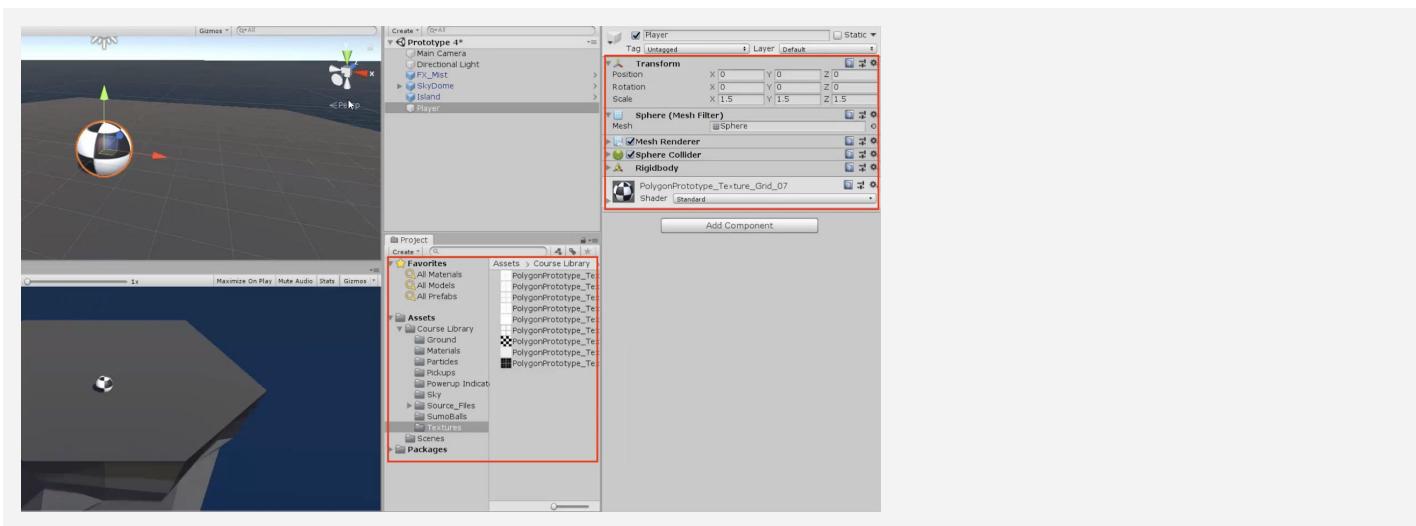


Step 2: Set up the player and add a texture

We've got an *island* for the game to take place on, and now we need a *sphere* for the player to control and roll around.

1. In the **Hierarchy**, create **3D Object > Sphere**
2. Rename it "Player", reset its **position** and increase its **XYZ scale** to 1.5
3. Add a **RigidBody** component to the **Player**
4. From the **Library > Textures**, drag a **texture** onto the **sphere**

- **New Concept:**
Texture wraps

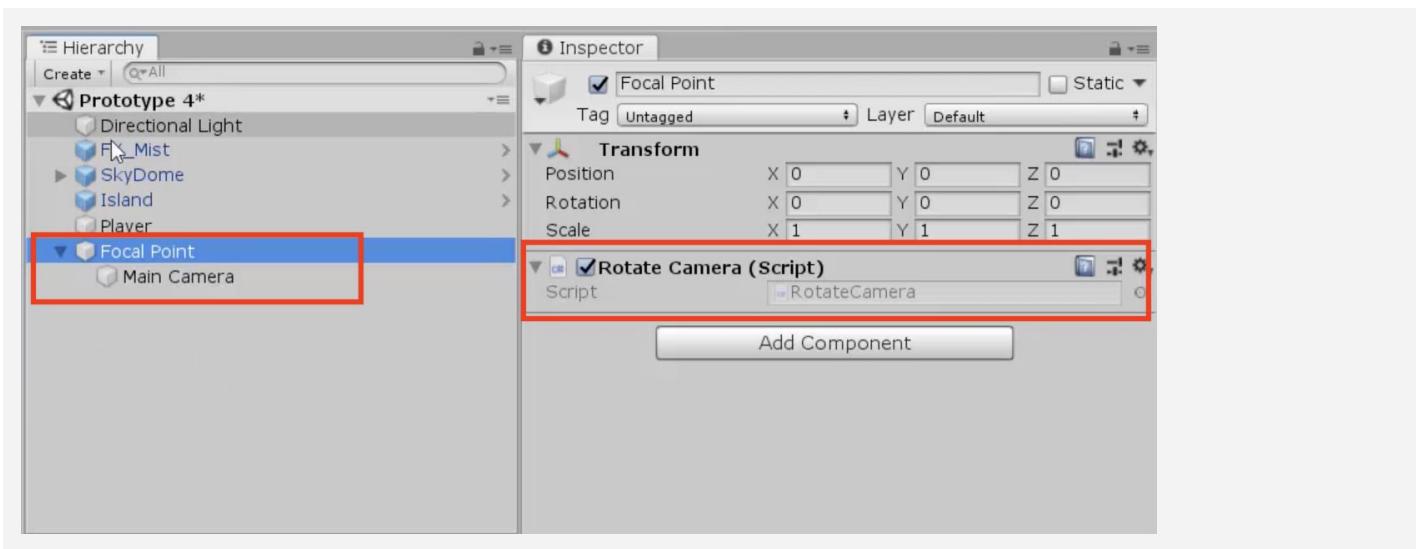


Step 3: Create a focal point for the camera

If we want the camera to rotate around the game in a smooth and cinematic fashion, we need to pin it to the center of the island with a focal point.

1. Create a new **Empty Object** and rename it “Focal Point”,
2. Reset its position to the origin (0, 0, 0), and make the Camera a **child object** of it
3. Create a new “Scripts” folder, and a new “RotateCamera” script inside it
4. **Attach** the “RotateCamera” script to the **Focal Point**

- **Don't worry:** This whole “focal point” business may be confusing at first, but it will make sense once you see it in action
- **Tip:** Try rotating the Focal point around the Y axis and see the camera rotate around in scene view



Step 4: Rotate the focal point by user input

Now that the camera is attached to the focal point, the player must be able to rotate it - and the camera child object - around the island with horizontal input.

1. Create the code to rotate the camera based on ***rotationSpeed*** and ***horizontalInput***
2. Tweak the ***rotation speed*** value to get the speed you want

- **Tip:** Horizontal input should be familiar, we used it all the way back in Unit 1! Feel free to reference your old code for guidance.

```
public float rotationSpeed;

void Update()
{
    float horizontalInput = Input.GetAxis("Horizontal");
    transform.Rotate(Vector3.up, horizontalInput * rotationSpeed * Time.deltaTime);
}
```

Step 5: Add forward force to the player

The camera is rotating perfectly around the island, but now we need to move the player.

1. Create a new “PlayerController” script, apply it to the **Player**, and open it
2. Declare a new **public float speed** variable and initialize it
3. Declare a new **private Rigidbody playerRb** and initialize it in **Start()**
4. In **Update()**, declare a new **forwardInput** variable based on “Vertical” input
5. Call the **AddForce()** method to move the player forward based **forwardInput**

- **Tip:** Moving objects with **Rigidbody** and **Addforce** should be familiar, we did it back in Unit 3! Feel free to reference old code.
- **Don't worry:** We don't have control over its direction yet - we'll get to that next

```
private Rigidbody playerRb;
public float speed;

void Start() {
    playerRb = GetComponent<Rigidbody>(); }

void Update() {
    float forwardInput = Input.GetAxis("Vertical");
    playerRb.AddForce(Vector3.forward * speed * forwardInput); }
```

Step 6: Move in direction of focal point

We've got the ball rolling, but it only goes forwards and backwards in a single direction! It should instead move in the direction the camera (and focal point) are facing.

1. Declare a new **private GameObject focalPoint**; and initialize it in **Start()**: **focalPoint = GameObject.Find("Focal Point")**;
2. In the **AddForce** call, Replace **Vector3.forward** with **focalPoint.transform.forward**

- **New Concept:** Global vs Local XYZ
- **Tip:** Global XYZ directions relate to the entire scene, whereas local XYZ directions relate to the object in question

```
private GameObject focalPoint;

void Start() {
    rb = GetComponent<Rigidbody>();
    focalPoint = GameObject.Find("Focal Point"); }

void Update() {
    float forwardInput = Input.GetAxis("Vertical");
    playerRb.AddForce(Vector3.forward * focalPoint.transform.forward
        * speed * Time.deltaTime); }
```

Lesson Recap

New Functionality	<ul style="list-style-type: none">• Camera rotates around the island based on horizontal input• Player rolls in direction of camera based on vertical input
New Concepts and Skills	<ul style="list-style-type: none">• Texture Wraps• Camera as child object• Global vs Local coordinates• Get direction of other object
Next Lesson	<ul style="list-style-type: none">• In the next lesson, we'll add more challenge to the player, by creating enemies that chase them in the game.



4.2 Follow the Player

Steps:

- Step 1: Add an enemy and a physics material
- Step 2: Create enemy script to follow player
- Step 3: Create a lookDirection variable
- Step 4: Create a Spawn Manager for the enemy
- Step 5: Randomly generate spawn position
- Step 6: Make a method return a spawn point

Example of project by end of lesson



Length: 60 minutes

Overview: The player can roll around to its heart's content... but it has no purpose. In this lesson, we fill that purpose by creating an enemy to challenge the player! First we will give the enemy a texture of your choice, then give it the ability to bounce the player away... potentially knocking them off the cliff. Lastly, we will let the enemy chase the player around the island and spawn in random positions.

Project Outcome: A textured and spherical enemy will spawn on the island at start, in a random location determined by a custom function. It will chase the player around the island, bouncing them off the edge if they get too close.

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

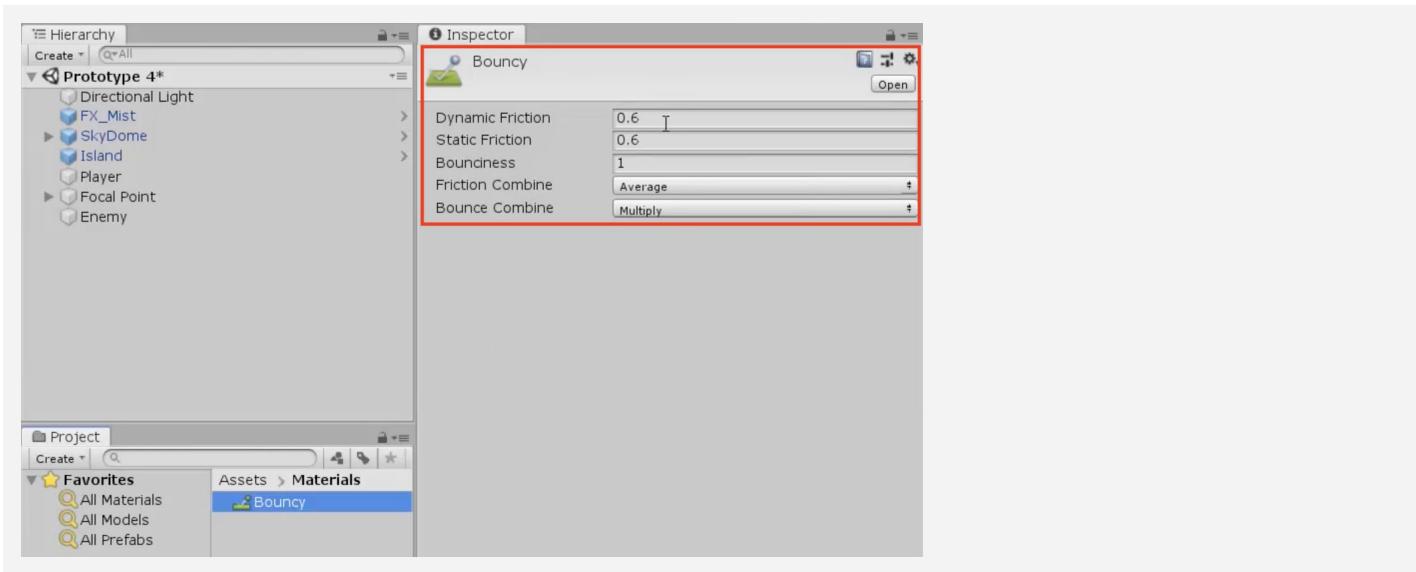
- Apply Physics Materials to make game objects bouncy
- Normalize vectors to point the enemy in the direction of the player
- Randomly spawn with Random.Range on two axes
- Write more advanced custom functions and variables to make your code clean and professional

Step 1: Add an enemy and a physics material

Our camera rotation and player movement are working like a charm. Next we're going to set up an enemy and give them some special new physics to bounce the player away!

1. Create a new **Sphere**, rename it "Enemy" reposition it, and drag a **texture** onto it
2. Add a new **Rigidbody** component and adjust its **XYZ scale**, then test
3. In a new "Physics Materials" folder, Create > **Physics Material**, then name it "Bouncy"
4. Increase the **Bounciness** to "1", change **Bounce Combine** to "Multiply", apply it to your player and enemy, then **test**

- **Don't worry:** If your game is lagging, uncheck the "Active" checkbox for your clouds
- **New Concept:** Physics Materials
- **New Concept:** Bounciness property and Bounce Combine



Step 2: Create enemy script to follow player

The enemy has the power to bounce the player away, but only if the player approaches it. We must tell the enemy to follow the player's position, chasing them around the island.

1. Make a new “[Enemy](#)” script and attach it to the **Enemy**
2. Declare 3 new variables for **Rigidbody enemyRb**, **GameObject player**, and **public float speed**;
3. Initialize **enemyRb = GetComponent< Rigidbody >();** and **player = GameObject.Find("Player");**
4. In **Update()**, AddForce towards in the direction between the Player and the Enemy

- **Tip:** Imagine we’re generating this new vector by drawing an arrow from the enemy to the player.
- **Tip:** We should start thinking ahead and writing our variables in advance. Think... what are you going to need?
- **Tip:** When normalized, a vector keeps the same direction but its length is 1.0, forcing the enemy to try and keep up

```
public float speed;
private Rigidbody enemyRb;
private GameObject player;

void Start() {
    enemyRb = GetComponent< Rigidbody >();
    player = GameObject.Find("Player");
}

void Update() {
    enemyRb.AddForce((player.transform.position
        - transform.position).normalized * speed);
}
```

Step 3: Create a lookDirection variable

The enemy is now rolling towards the player, but our code is a bit messy. Let’s clean up by adding a variable for the new vector.

1. In **Update()**, declare a new **Vector3 lookDirection** variable
2. Set **Vector3 lookDirection = (player.transform.position - transform.position).normalized;**
3. Implement the **lookDirection** variable in the **AddForce** call

- **Tip:** As always, adding variables makes the code more readable

```
void Update() {
    Vector3 lookDirection = (player.transform.position
        - transform.position).normalized;

    enemyRb.AddForce(lookDirection (player.transform.position
        - transform.position).normalized * speed);
}
```

Step 4: Create a Spawn Manager for the enemy

Now that the enemy is acting exactly how we want, we're going to turn it into a prefab so it can be instantiated by a Spawn Manager.

1. Drag **Enemy** into the Prefabs folder to create a new **Prefab**, then delete **Enemy** from scene
2. Create a new “Spawn Manager” **object**, attach a new “SpawnManager” **script**, and open it
3. Declare a new **public GameObject enemyPrefab** variable then assign the prefab in the **inspector**
4. In **Start()**, instantiate a new **enemyPrefab** at a predetermined location

```
public GameObject enemyPrefab;

void Start()
{
    Instantiate(enemyPrefab, new Vector3(0, 0, 6),
    enemyPrefab.transform.rotation); }
```

Step 5: Randomly generate spawn position

The enemy spawns at start, but it always appears in the same spot. Using the familiar **Random** class, we can spawn the enemy in a random position.

1. In **SpawnManager.cs**, in **Start()**, create new **randomly generated X and Z**
2. Create a new **Vector3 randomPos** variable with those random X and Z positions
3. Incorporate the new **randomPos** variable into the **Instantiate** call
4. Replace the hard-coded **values** with a **spawnRange** variable
5. **Start** and **Restart** your project to make sure it's working

- **Tip:** Remember, we used **Random.Range** all the way back in Unit 2! Feel free to reference old code.

```
public GameObject enemyPrefab;
private float spawnRange = 9;

void Start() {
    float spawnPosX = Random.Range(-9, 9 - spawnRange, spawnRange);
    float spawnPosZ = Random.Range(-9, 9 - spawnRange, spawnRange);
    Vector3 randomPos = new Vector3(spawnPosX, 0, spawnPosZ);
    Instantiate(enemyPrefab, randomPos, enemyPrefab.transform.rotation); }
```

Step 6: Make a method return a spawn point

The code we use to generate a random spawn position is perfect, and we're going to be using it a lot. If we want to clean the script and use this code later down the road, we should store it in a custom function.

1. Create a new function **Vector3 GenerateSpawnPosition()**
2. Copy and Paste the **spawnPosX** and **spawnPosZ** variables into the new method
3. Add the line to **return randomPos;** in your new method
4. Replace the code in your **Instantiate call** with your new function name: **GenerateSpawnPosition()**

- **Tip:** This function will come in handy later, once we randomize a spawn position for the powerup
- **New Concept:** Functions that return a value
- **Tip:** This function is different from "void" calls, which do not return a value. Look at "GetAxis" in PlayerController for example - it returns a float

```
void Start() {
    Instantiate(enemyPrefab, GenerateSpawnPosition()
        new Vector3(spawnPosX, 0, spawnPosZ), enemyPrefab.transform.rotation);
    float spawnPosX = Random.Range(-spawnRange, spawnRange);
    float spawnPosZ = Random.Range(-spawnRange, spawnRange); }

private Vector3 GenerateSpawnPosition () {
    float spawnPosX = Random.Range(-spawnRange, spawnRange);
    float spawnPosZ = Random.Range(-spawnRange, spawnRange);
    Vector3 randomPos = new Vector3(spawnPosX, 0, spawnPosZ);
    return randomPos; }
```

Lesson Recap

New Functionality	<ul style="list-style-type: none"> • Enemy spawns at random location on the island • Enemy follows the player around • Spheres bounce off of each other
New Concepts and Skills	<ul style="list-style-type: none"> • Physics Materials • Defining vectors in 3D space • Normalizing values • Methods with return values
Next Lesson	<ul style="list-style-type: none"> • In our next lesson, we'll create ways to fight back against these enemies using Powerups!

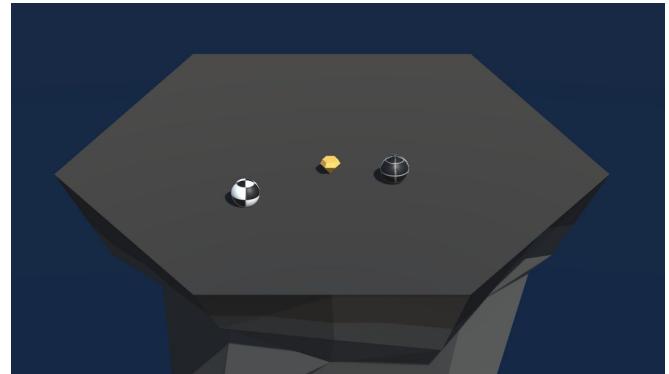


4.3 PowerUp and CountDown

Steps:

- Step 1: Choose and prepare a powerup
- Step 2: Destroy powerup on collision
- Step 3: Test for collision with a powerup
- Step 4: Apply extra knockback with powerup
- Step 5: Create Countdown Routine for powerup
- Step 6: Add a powerup indicator

Example of project by end of lesson



Length: 60 minutes

Overview: The enemy chases the player around the island, but the player needs a better way to defend themselves... especially if we add more enemies. In this lesson, we're going to create a powerup that gives the player a temporary strength boost, shoving away enemies that come into contact! The powerup will spawn in a random position on the island, and highlight the player with an indicator when it is picked up. The powerup indicator and the powerup itself will be represented by stylish game assets of your choice.

Project Outcome: A powerup will spawn in a random position on the map. Once the player collides with this powerup, the powerup will disappear and the player will be highlighted by an indicator. The powerup will last for a certain number of seconds after pickup, granting the player super strength that blasts away enemies.

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

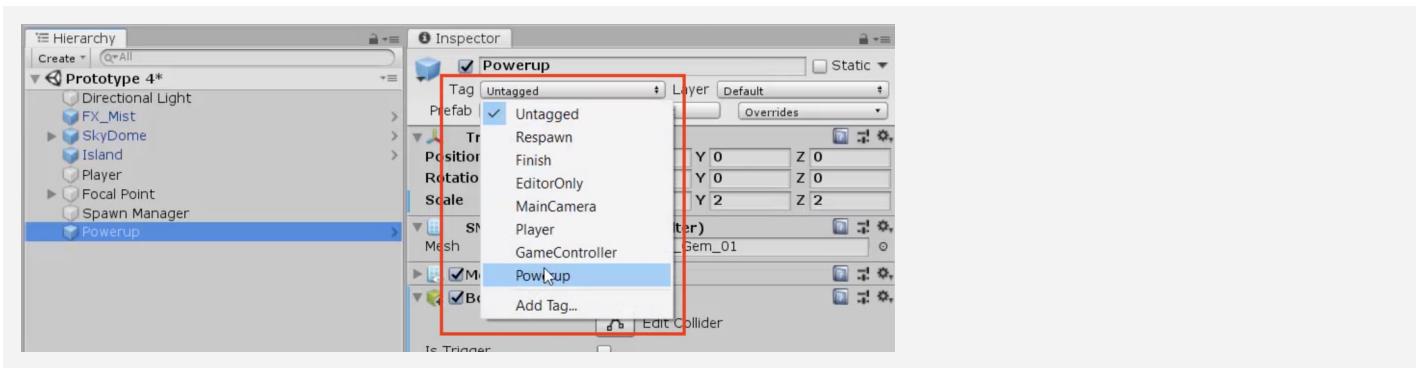
- Write informative debug messages with Concatenation and variables
- Repeat functions with the power of Ienumerator and Coroutines
- Use SetActive to make game objects appear and disappear from the scene

Step 1: Choose and prepare a powerup

In order to add a completely new gameplay mechanic to this project, we will introduce a new powerup object that will give the player temporary superpowers.

1. From the **Library**, drag a **Powerup** object into the scene, rename it “Powerup” and edit its **scale & position**
2. Add a **Box Collider** to the powerup, click **Edit Collider** to make sure it fits, then check the “**Is Trigger**” checkbox
3. Create a new “Powerup” **tag** and apply it to the **powerup**
4. Drag the **Powerup** into the **Prefabs folder** to create a new “Original Prefab”

- **Warning:** Remember, you still have to apply the tag after it has been created.



Step 2: Destroy powerup on collision

As a first step to getting the powerup working, we'll make it disappear when the player hits it and set up a new boolean variable to track that the player got it.

1. In **PlayerController.cs**, add a new **OnTriggerEnter()** method
2. Add an **if-statement** that destroys **other.CompareTag("Powerup")** powerup on collision
3. Create a new **public bool hasPowerup;** and set **hasPowerup = true;** when you **collide** with the Powerup

- **Don't worry:** If this doesn't work, make sure that the Powerup's collider “Is trigger” and player's collider is NOT
- **Tip:** Make sure **hasPowerup = true** in the inspector when you collide

```
public bool hasPowerup

private void OnTriggerEnter(Collider other) {
    if (other.CompareTag("Powerup")) {
        hasPowerup = true;
        Destroy(other.gameObject); } }
```

Step 3: Test for collision with a powerup

The powerup will only come into play in a very particular circumstance: when the player has a powerup AND they collide with an enemy - so we'll first test for that very specific condition.

1. Create a new “Enemy” tag and apply it to the **Enemy Prefab**
2. In **PlayerController.cs**, add the ***OnCollisionEnter()*** function
3. Create the **if-statement** with the **double-condition**
4. Create a **Debug.Log** to make sure it's working

- **Tip:** `OnTriggerEnter` is good for stuff like picking up powerups, but you should use `OnCollisionEnter` when you want something to do with physics
- **New Concept:** Concatenation in Debug messages
- **Tip:** When you concatenate a variable in a debug message, it will return its VALUE not its name

```
private void OnCollisionEnter(Collision collision) {
    if (collision.gameObject.CompareTag("Enemy") && hasPowerup) {
        Debug.Log("Player collided with " + collision.gameObject
        + " with powerup set to " + hasPowerup); } }
```

Step 4: Apply extra knockback with powerup

With the condition for the powerup set up perfectly, we are now ready to program the actual powerup ability: when the player collides with an enemy, the enemy should go flying!

1. In ***OnCollisionEnter()*** declare a new **local variable** to get the Enemy's **Rigidbody** component
2. Declare a new variable to get the **direction** away from the **player**
3. Add an **impulse force** to the **enemy**
4. **Replace** the hard-coded value with a new **powerupStrength** variable

- **Tip:** Reference the code in `Enemy.cs` that makes the enemy follow the player. In a way, we're reversing that code in order to push the enemy away.
- **Don't worry:** No need to use `.Normalize`, since they're colliding

```
private float powerupStrength = 15.0f;

private void OnCollisionEnter(Collision collision) {
    if (collision.gameObject.CompareTag("Enemy") && hasPowerup) {

        Rigidbody enemyRigidbody = collision.gameObject.GetComponent<Rigidbody>();
        Vector3 awayFromPlayer = (collision.gameObject.transform.position
        - transform.position);

        Debug.Log("Player collided with " + collision.gameObject
        + " with powerup set to " + hasPowerup);
        enemyRigidbody.AddForce(awayFromPlayer
        * -10 powerupStrength, ForceMode.Impulse); } }
```

Step 5: Create Countdown Routine for powerup

It wouldn't be fair to the enemies if the powerup lasted forever - so we'll program a countdown timer that starts when the player collects the powerup, removing the powerup ability when the timer is finished.

1. Add a new **IEnumerator**
PowerupCountdownRoutine () {}
2. Inside the **PowerupCountdownRoutine**, wait 7 seconds, then **disable** the powerup
3. When player **collides** with powerup, start the **coroutine**

- **New Concept:** IEnumerator
- **New Concept:** Coroutines
- **Tip:** WaitForSeconds()

```
private void OnTriggerEnter(Collider other) {
    if (other.CompareTag("Powerup")) {
        hasPowerup = true;
        Destroy(other.gameObject);
        StartCoroutine(PowerupCountdownRoutine()); } }

IEnumerator PowerupCountdownRoutine() {
    yield return new WaitForSeconds(7); hasPowerup = false; }
```

Step 6: Add a powerup indicator

To make this game a lot more playable, it should be clear when the player does or does not have the powerup, so we'll program a visual indicator to display this to the user.

1. From the *Library*, drag a **Powerup object** into the scene, rename it **"Powerup Indicator"**, and edit its **scale**
2. **Uncheck** the "Active" checkbox in the inspector
3. In **PlayerController.cs**, declare a new **public GameObject powerupIndicator** variable, then assign the **Powerup Indicator** variable in the inspector
4. When the player collides with the powerup, set the indicator object to **Active**, then set to **Inactive** when the powerup expires
5. In **Update()**, set the Indicator position to the player's position + an **offset value**

- **New Function:**
SetActive
- **Tip:** Make sure the indicator is turning on and off before making it follow the player

```
public GameObject powerupIndicator

void Update() {
    ... powerupIndicator.transform.position = transform.position
    + new Vector3(0, -0.5f, 0); }

private void OnTriggerEnter(Collider other) {
    if (other.CompareTag("Powerup")) {
        ... powerupIndicator.gameObject.SetActive(true); } }

IEnumerator PowerupCountdownRoutine() {
    ... powerupIndicator.gameObject.SetActive(false); }
```

Lesson Recap

New Functionality	<ul style="list-style-type: none"> • When the player collects a powerup, a visual indicator appears • When the player collides with an enemy while they have the powerup, the enemy goes flying • After a certain amount of time, the powerup ability and indicator disappear
New Concepts and Skills	<ul style="list-style-type: none"> • <i>Debug concatenation</i> • <i>Local component variables</i> • <i>IEnumerators and WaitForSeconds()</i> • <i>Coroutines</i> • <i>SetActive(true/false)</i>
Next Lesson	<ul style="list-style-type: none"> • We'll start generating waves of enemies for our player to fend off!



4.4 For-Loops For Waves

Steps:

Step 1: Write a for-loop to spawn 3 enemies

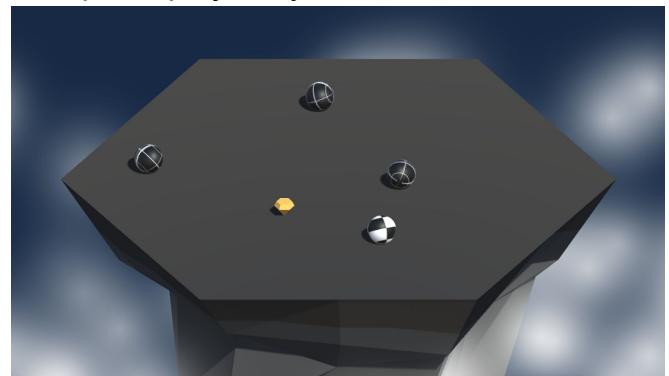
Step 2: Give the for-loop a parameter

Step 3: Destroy enemies if they fall off

Step 4: Increase enemyCount with waves

Step 5: Spawn Powerups with new waves

Example of project by end of lesson



Length: 60 minutes

Overview: We have all the makings of a great game; A player that rolls around and rotates the camera, a powerup that grants super strength, and an enemy that chases the player until the bitter end. In this lesson we will wrap things up by putting these pieces together!

First we will enhance the enemy spawn manager, allowing it to spawn multiple enemies and increase their number every time a wave is defeated. Lastly we will spawn the powerup with every wave, giving the player a chance to fight back against the ever-increasing horde of enemies.

Project Outcome: The Spawn Manager will operate in waves, spawning multiple enemies and a new powerup with each iteration. Every time the enemies drop to zero, a new wave is spawned and the enemy count increases.

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

- Repeat functions with For-loops
- Increment integer values in a loop with the `++` operator
- Target objects in a scene with `FindObjectsOfType`
- Return the length of an array as an integer with `.Length`

Step 1: Write a for-loop to spawn 3 enemies

We should challenge the player by spawning more than one enemy. In order to do so, we will repeat enemy instantiation with a loop.

1. In `SpawnManager.cs`, in `Start()`, replace single **Instantiation** with a **for-loop** that spawns 3 enemies
2. Move the for-loop to a new **void** `SpawnEnemyWave()` function, then call that function from `Start()`

- **New Concept:** For-loops
- **Don't worry:** Loops are a bit confusing at first, but they make sense eventually. Loops are powerful tools that programmers use often
- **New Concept:** `++` Increment Operator

```
void Start() {
    SpawnEnemyWave();
    for (int i = 0; i < 3; i++) {
        Instantiate(enemyPrefab, GenerateSpawnPosition(),
        enemyPrefab.transform.rotation); } }

void SpawnEnemyWave() {
    for (int i = 0; i < 3; i++) {
        Instantiate(enemyPrefab, GenerateSpawnPosition(),
        enemyPrefab.transform.rotation); } }
```

Step 2: Give the for-loop a parameter

Right now, `SpawnEnemyWave` spawns exactly 3 enemies, but if we're going to dynamically increase the number of enemies that spawn during gameplay, we need to be able to pass information to that method.

1. Add a parameter **`int enemiesToSpawn`** to the `SpawnEnemyWave` function
2. Replace `i < _` with `i < enemiesToSpawn`
3. Add this new variable to the function call in `Start()`: `SpawnEnemyWave(_);`

- **New Concept:** Custom methods with parameters
- **Tip:** `GenerateSpawnPosition` returns a value, `SpawnEnemyWave` does not. `SpawnEnemyWave` takes a parameter, `GenerateSpawnPosition` does not.

```
void Start() {
    SpawnEnemyWave(3); }

void SpawnEnemyWave(int enemiesToSpawn) {
    for (int i = 0; i < 3; i++) {
        Instantiate(enemyPrefab, GenerateSpawnPosition(),
        enemyPrefab.transform.rotation); } }
```

Step 3: Destroy enemies if they fall off

Once the player gets rid of all the enemies, they're left feeling a bit lonely. We need to destroy enemies that fall, and spawn a new enemy wave once the last one is vanquished!

1. In `Enemy.cs`, **destroy** the enemies if their position is less than a **-Y value**
2. In `SpawnManager.cs`, declare a new **public int** `enemyCount` variable
3. In `Update()`, set `enemyCount = FindObjectsOfType<Enemy>().Length;`
4. Write the **if-statement** that if `enemyCount == 0` then `SpawnEnemyWave`, then delete it from `Start()`

```
void Update() {
    ... if (transform.position.y < -10) { Destroy(gameObject); } }

<----->
public int enemyCount

void Update() {
    enemyCount = FindObjectsOfType<Enemy>().Length;
    if (enemyCount == 0) { SpawnEnemyWave(1); } }
```

Step 4: Increase `enemyCount` with waves

Now that we control the amount of enemies that spawn, we should increase their number in waves. Every time the player defeats a wave of enemies, more should rise to take their place.

1. Declare a new **public int** `waveCount = 1;`, then implement it in `SpawnEnemyWave(waveCount)`
2. In the if-statement that tests if there are 0 enemies left, **increment** `waveCount` by 1

- **Tip:** Incrementing with the `++` operator is very handy, you may find yourself using it in the future

```
public int waveCount = 1;

void Start() {
    SpawnEnemyWave(3 waveCount); }

void Update() {
    enemyCount = FindObjectsOfType<Enemy>().Length;
    if (enemyCount == 0) { waveCount++; SpawnEnemyWave(4 waveCount); } }
```

Step 5: Spawn Powerups with new waves

Our game is almost complete, but we're missing something. Enemies continue to spawn with every wave, but the powerup gets used once and disappears forever, leaving the player vulnerable. We need to spawn the powerup in a random position with every wave, so the player has a chance to fight back.

1. In `SpawnManager.cs`, declare a new ***public GameObject powerupPrefab*** variable, assign the ***prefab*** in the inspector and ***delete*** it from the scene
2. In ***Start()***, ***Instantiate*** a new Powerup
3. Before the ***SpawnEnemyWave()*** call, ***Instantiate*** a new Powerup

- **Tip:** Now that we have a very playable game, let's test and tweak values

```
public GameObject powerupPrefab;

void Start() {
    ... Instantiate(powerupPrefab, GenerateSpawnPosition(),
    powerupPrefab.transform.rotation); }

void Update() {
    ... if (enemyCount == 0) { ... Instantiate(powerupPrefab,
    GenerateSpawnPosition(), powerupPrefab.transform.rotation); } }
```

Lesson Recap

New Functionality	<ul style="list-style-type: none"> • Enemies spawn in waves • The number of enemies spawned increases after every wave is defeated • A new power up spawns with every wave
New Concepts and Skills	<ul style="list-style-type: none"> • For-loops • Increment (++) operator • Custom methods with parameters • <code>FindObjectsOfType</code>



Challenge 4

Soccer Scripting



Challenge Overview:

Use the skills you learned in the Sumo Battle prototype in a completely different context: the soccer field. Just like in the prototype, you will control a ball by rotating the camera around it and applying a forward force, but instead of knocking them off the edge, your goal is to knock them into the opposing net while they try to get into your net. Just like in the Sumo Battle, after every round a new wave will spawn with more enemy balls, putting your defense to the test. However, almost nothing in this project is functioning! It's your job to get it working correctly.

Challenge Outcome:

- Enemies move towards your net, but you can hit them to deflect them away
- Powerups apply a temporary strength boost, then disappear after 5 seconds
- When there are no more enemy balls, a new wave spawns with 1 more enemy

Challenge Objectives:

- In this challenge, you will reinforce the following skills/concepts:
- Defining Vectors by subtracting one location in 3D space from another
 - Track the number of objects of a certain type in a scene to trigger certain events
 - Using Coroutines to perform actions based on a timed interval
 - Using for-loops and dynamic variables to run code a particular number of times
 - Resolving errors related to null references of unassigned variables

Challenge Instructions:

- Open your **Prototype 4** project
- **Download** the "Challenge 4 Starter Files" from the Tutorial Materials section, then double-click on it to **Import**
- In the *Project Window > Assets > Challenge 4 > Instructions* folder, use the resources as a guide to complete this challenge

Challenge	Task	Hint
1 Hitting an enemy sends it back towards you	When you hit an enemy, it should send it away from the player	In PlayerControllerX.cs, to get a Vector away from the player, you should subtract the [enemy position] minus the [player's position] - not the reverse
2 A new wave spawns when the player gets a powerup	A new wave should spawn when all enemy balls have been removed	In SpawnManagerX.cs, check that the enemyCount variable is being set correctly
3 The powerup never goes away	The powerup should only last for a certain duration, then disappear	In PlayerControllerX.cs, the PowerupCoolDown Coroutine code looks good, but this coroutine is never actually called with the StartCoroutine() method
4 2 enemies are spawned in every wave	One enemy should be spawned in wave 1, two in wave 2, three in wave 3, etc	In SpawnManagerX.cs, the for-loop that spawns enemy should make use of the enemiesToSpawn parameter
5 The enemy balls are not moving anywhere	The enemy balls should go towards the "Player Goal" object	There is an error in EnemyX.cs: "NullReferenceException: Object reference not set to an instance of an object". It looks like the playerGoal object is never assigned.

Bonus Challenge	Task	Hint
X The player needs a turbo boost	The player should get a speed boost whenever the player presses spacebar - and a particle effect should appear when they use it	In PlayerController, add a simple if-statement that adds an "impulse" force if spacebar is pressed. To add a particle effect, first attach it as a child object of the Focal Point.
Y The enemies never get more difficult	The enemies' speed should increase in speed by a small amount with every new wave	You'll need to track and increase the enemy speed in SpawnManagerX.cs. Then in EnemyX.cs, reference that speed variable and set it in Start().

Challenge Solution

- 1 In PlayerControllerX.cs, in OnCollisionEnter(), the ***awayFromPlayer*** Vector3 is in the opposite direction it should be.

```
Vector3 awayFromPlayer = transform.position -  
other.gameObject.transform.position;  
= other.gameObject.transform.position -  
transform.position;
```

- 2 In SpawnManagerX.cs, the ***enemyCount*** variable is counting the number of objects with a “Powerup” tag - it should be counting the number of objects with an “Enemy” tag

```
void Update() {  
    enemyCount = GameObject.FindGameObjectsWithTag("Powerup Enemy").Length;  
    ...  
}
```

- 3 In PlayerControllerX.cs, in the OnTriggerEnter() method, you need to initiate the ***PowerupCooldown*** Coroutine in order to begin the countdown process

```
private void OnTriggerEnter(Collider other) {  
    if (other.gameObject.CompareTag("Powerup")) {  
        ...  
        StartCoroutine(PowerupCooldown());  
    }  
}
```

- 4 In SpawnManagerX.cs, the for-loop that spawns enemy should make use of the ***enemiesToSpawn*** parameter

```
for (int i = 0; i < 2-enemyPrefab; i++) {  
    Instantiate(enemyPrefab, GenerateSpawnPosition(), ...  
}
```

- 5 In EnemyX.cs, the ***playerGoal*** variable is not initialized - initialize it in the Start() method

```
void Start() {  
    enemyRb = GetComponent<Rigidbody>();  
    playerGoal = GameObject.Find("Player Goal");  
}
```

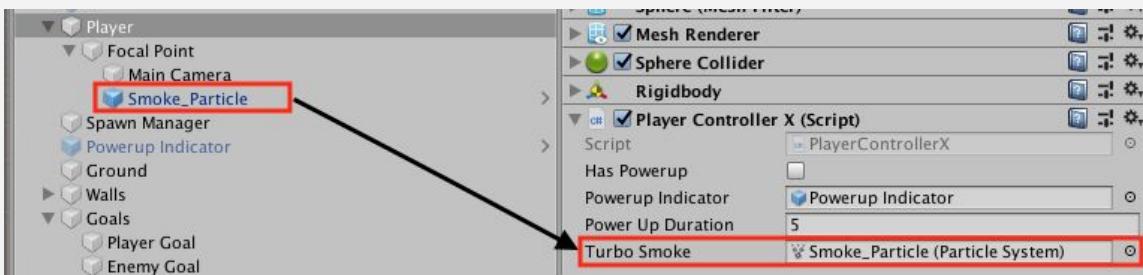
Bonus Challenge Solution

- X1** To add a turbo boost, In PlayerControllerX.cs, declare a new **turboBoost** float variable, then in Update(), add a simple if-statement that adds an “impulse” force in the direction of the focal point if spacebar is pressed:

```
private float turboBoost = 10;

void Update() {
    ...
    if (Input.GetKeyDown(KeyCode.Space)) {
        playerRb.AddForce(focalPoint.transform.forward * turboBoost, ForceMode.Impulse);
    }
}
```

- X2** Add the Smoke_Particle prefab as a child object of the focal point (next to the camera), then in PlayerControllerX.cs, declare a new turboSmoke particle variable and assign it in the inspector



- X3** In PlayerControllerX.cs, in the if-statement checking if the player presses spacebar, play the particle

```
if (Input.GetKeyDown(KeyCode.Space)) {
    playerRb.AddForce(focalPoint.transform.forward * turboBoost, ForceMode.Impulse);
    turboSmoke.Play();
}
```

- Y1** In SpawnManagerX.cs, declare and initialize a new public **enemySpeed** variable, then increase it by a certain amount every time a wave is spawned:

```
public int enemyCount;
public float enemySpeed = 50;

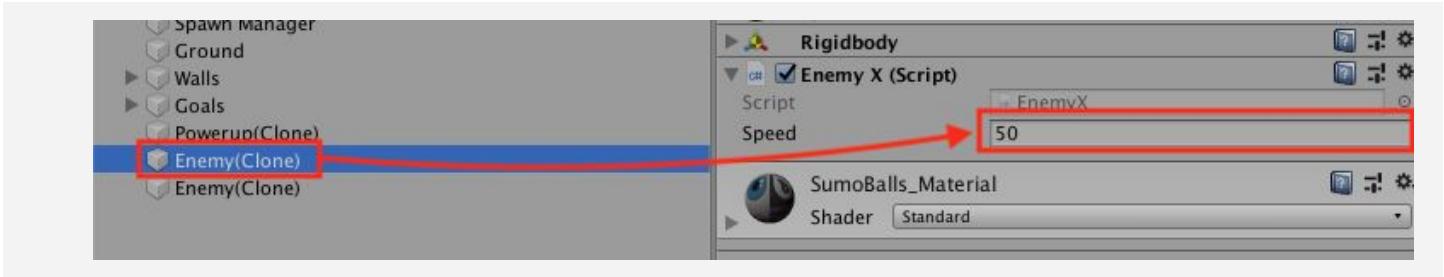
void SpawnEnemyWave(int enemiesToSpawn) {
    ...
    waveCount++;
    enemyCount += 25;
}
```

- Y2** In EnemyX.cs, declare a new spawnManagerXScript variable, get a reference to it in Start(), then set the enemy's **speed** variable to your new **enemySpeed** variable

```
private GameObject playerGoal;
private SpawnManagerX spawnManagerXScript;

void Start() {
    enemyRb = GetComponent<Rigidbody>();
    playerGoal = GameObject.Find("Player Goal");
    spawnManagerXScript = GameObject.Find("Spawn Manager").GetComponent<SpawnManagerX>();
    speed = spawnManagerXScript.enemySpeed;
}
```

- Y3** To test, make the **speed** variable in EnemyX.cs public and check the enemies' speed when they are spawned in different waves





Unit 4 Lab

Basic Gameplay

Steps:

Step 1: Give objects basic movement

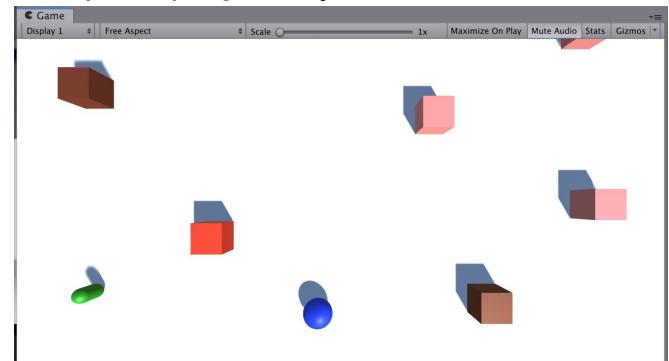
Step 2: Destroy objects off-screen

Step 3: Handle object collisions

Step 4: Make objects into prefabs

Step 5: Make SpawnManager spawn Prefabs

Example of progress by end of lab



Length: 60 minutes

Overview: In this lab, you will work with all of your non-player objects in order to bring your project to life with its basic gameplay. You will give your projectiles, pickups, or enemies their basic movement and collision detection, make them into prefabs, and have them spawned randomly by a spawn manager. By the end of this lab, you should have a glimpse into the core functionality of your game.

Project Outcome: Non-player objects are spawned at appropriate locations in the scene with basic movement. When objects collide with each other, they react as intended, by either bouncing or being destroyed.

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

- More comfortably program basic movement
- More comfortably handle object collisions
- More comfortably spawn object prefabs on timed intervals

Step 1: Give objects basic movement

Before you spawn objects into your scene, they should move the way you want.

1. If relevant, add **Rigidbody** components to your non-player objects
2. Create a **new script(s)** for the objects that will be instantiated during gameplay and attach them to their respective objects (including projectiles or pickups)
3. Program the **basic movement** for your objects and test that they work

- **Tip:** Make sure you uncheck "use gravity" if you don't want them to fall
- **Tip:** In the collider component, check "is trigger" if you don't want actual collisions

By the end of this step, all objects should basically move the way they should in the game.

Step 2: Destroy objects off-screen

To make sure our hierarchy doesn't get too cluttered, let's make sure these objects get destroyed when they leave the screen.

1. Either create a new script or add code to your existing script to make sure objects are destroyed when they leave the screen

- **Tip:** Move your objects in scene view to determine the xyz positions objects should be destroyed

By the end of this step, objects should be removed from the hierarchy when they are no longer in play.

Step 3: Handle object collisions

Now that you have all these moving objects, they're bound to start colliding with each other - we need to program what should happen when everything collides.

1. If relevant, edit the **Rigidbody mass** of your objects
2. If relevant, to change the way your objects collide, create a new **Physics material** for your objects
3. Add **tags** to your objects so you can accurately test for which objects are colliding with which
4. Use **OnCollisionEnter()** (for Rigidbody collisions) or **OnTriggerEnter()** (for trigger-based collisions) to **Destroy** or **Log** messages to the console what should happen when certain collisions occur

- **Don't worry:** If you collide with a powerup or pickup, the actual functionality does *not* need to be programmed, just the effect
- **Tip:** Should use OnTriggerEnter if objects are being destroyed - but remember that "Is Trigger" must be checked for this to work!

By the end of this step, objects should destroy, bounce, or do nothing based on collisions.

Step 4: Make objects into prefabs

Now that the objects are basically behaving the way they should, if they're going to be instantiated during gameplay, they need to be prefabs

1. In the Assets directory, create a new **Folder** called "Prefabs"
2. **Drag** in each object to create a **new prefab** for it
3. After all objects have been turned into prefabs, **delete** them from the scene
4. **Test** the objects' behavior by dragging them from the Prefabs folder into the scene while the game is running

- **Tip:** When creating new prefabs, you have to drag them one at a time
- **Tip:** Notice that their icons turn blue when they are prefabs

By the end of this step, all objects that will be spawned during gameplay should be prefabs and should no longer be in your scene.

Step 5: Make SpawnManager spawn Prefabs

Now that we have all of our prefabs set up, we can create a spawn manager to spawn them at intervals and, if we want, in random locations.

1. Create an Empty "Spawn Manager" object and attach a new `SpawnManager.cs` script to it
2. Create individual **GameObject** or **GameObject array** variables for your prefabs, then **assign** them in the inspector
3. Use the **`Instantiate()`**, **`Random.Range()`**, and the **`InvokeRepeating()`** methods to spawn objects at intervals (random objects, random locations, or both)
4. Right-click on your Assets folder > **Export Package** then save a new version in your **Backups** folder

- **Tip:** Name your variables "__Prefab" so you know it requires a prefab value
- **Don't worry:** If it's not perfect yet or if there are some minor bugs - just get the general idea working

By the end of this step, objects should be spawned automatically from the appropriate location.

Lesson Recap

New Progress

- Non-player objects prefabs have basic movement
- Objects are destroyed when they leave the screen
- Collisions between objects are handled appropriately
- Objects are spawned at the appropriate locations on time-based intervals

New Concepts and Skills

- Creating basic gameplay for a project independently



Quiz Unit 4

QUESTION

CHOICES

- 1** You're trying to write some code that creates a random age between 1 and 100 and prints that age, but there is an error. What would fix the error?

```

1. private int age;
2.
3. void Start() {
4.     Debug.Log(GenerateRandomAge());
5. }
6.
7. private int GenerateRandomAge() {
8.     age = Random.Range(1, 101);
9. }
```

- 2** The following message was displayed in the console: "Monica has 20 dollars". Which of lines in the PrintNames function produced it?

- a. Change line 1 to "private float age"
- b. Add the word "int" to line 8, so it says "int age = ..."
- c. On line 7, change the word "private" to "void"
- d. Add a new line after line 8 that says "return age;"

- a. Option A
- b. Option B
- c. Option C
- d. Option D

```

string[] names = new string[] { "Steve", "Monica", "Eric" };
int money = 5;

void Start() {
    money *= 2;
    PrintNames();
}

void PrintNames () {
    A. Debug.Log("Monica has " + money/2 + " dollars");
    B. Debug.Log(names[1] + " has " + money*2 + " dollars");
    C. Debug.Log(names[2] + " has " + money*4 + " dollars");
    D. Debug.Log(names[Monica] + " has " + money/2 + " dollars");
}
```

3 The code below produces “error CS0029: Cannot implicitly convert type ‘float’ to ‘UnityEngine.Vector3’”. Which of the following would remove the error?

1. `private Vector3 startingVelocity;`
2. `void Start() {`
3. `startingVelocity = 2.0f;`
4. `}`

- a. On line 1, change “Vector3” to “float”
- b. On line 3, change “=” to “+”
- c. Either A or B
- d. None of the above

4 Which of the following follows Unity’s naming conventions (especially as it relates to capitalization)?

- A. `float forwardInput = Input.GetAxis("Vertical");`
- B. `float ForwardInput = input.GetAxis("Vertical");`
- C. `Float forwardInput = Input.GetAxis("Vertical");`
- D. `float forwardInput = input.getAxis("vertical");`

5 You are trying to assign the powerup variable in the inspector, but it is not showing up in the Player Controller component. What is the problem?

```
public class PlayerController : MonoBehaviour
{
    private GameObject powerup;
}
```

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

- a. You cannot declare a powerup variable in the Player Controller Script
- b. You cannot assign GameObject type variables in the inspector
- c. The powerup variable should be public instead of private
- d. The PlayerController class should be private instead of public

6 Your game has just started and you see the error, “UnassignedReferenceException: The variable playerIndicator of PlayerController has not been assigned.” What is likely the solution to the problem?

```
public class PlayerController : MonoBehaviour
{
    public GameObject playerIndicator;
    void Update()
    {
        playerIndicator.transform.position.y = 10;
    }
}
```

- a. PlayerController variable in the playerIndicator script needs to be declared
- b. The playerIndicator variable needs to be made private
- c. The PlayerController script must be assigned to the player object
- d. An object needs to be dragged onto the playerIndicator variable in the inspector

7 You are trying to create a new method that takes a number and multiplies it by two. Which method would do that?

- a. Method A
- b. Method B
- c. Method C
- d. Method D

- A.

```
private float DoubleNumber() {`  
    return number *= 2;  
}
```
- B.

```
private float DoubleNumber(float number) {`  
    return number *= 2;  
}
```
- C.

```
private void DoubleNumber(float number) {`  
    return number *= 2;  
}
```
- D.

```
private void DoubleNumber() {`  
    return number *= 2;  
}
```

8 Which comment best describes the code below?

```
public class Enemy : MonoBehaviour  
{  
    // Comment  
    private void OnTriggerEnter(Collider other) {  
        if(other.CompareTag("Spike")) {  
            Destroy(other.gameObject);  
        }  
    }  
}
```

- a. // If the player collides with an enemy, destroy the enemy
- b. // If the enemy collides with a spike, destroy the spike
- c. // If the enemy collides with a spike, destroy the enemy
- d. // If the player collides with a spike, destroy the spike

9 The code below produces the error, “error CS0029: Cannot implicitly convert type ‘UnityEngine.GameObject’ to ‘UnityEngine.Rigidbody’”. What could be done to fix this issue?

1. `void OnCollisionEnter(Collision collision) {`
2. `if(collision.gameObject.CompareTag("Enemy")) {`
3. `Rigidbody enemyRb = collision.gameObject;`
4. `}`
5. `}`

- a. On line 1, change “collision” to “Rigidbody”
- b. On line 2, change “gameObject” to “Rigidbody”
- c. On line 3, delete “.gameObject”
- d. On line 3, add “.GetComponent< Rigidbody>()” before the semicolon

10 Which of the following statements about functions/methods are correct:

- A. Functions/methods must be passed at least one parameter
- B. Functions/methods with a "void" return type cannot be passed parameters
- C. A Function/method with an "int" return type could include the code, "return 0.5f;"
- D. If there was a function/method declared as "private void RenameObject(string newName)", you could call that method with "RenameObject();"

- a. A and B are correct
- b. Only B is correct
- c. B and C are correct
- d. Only D is correct
- e. None are correct

Quiz Answer Key

#	ANSWER	EXPLANATION
1	D	Since the method has an “int” return type “private int GenerateRandomAge()”, it must <i>return</i> an int.
2	B	Debug.Log(names[1] + " has " + money*2 + " dollars"); is correct. Arrays start with index 0, so “Monica” has the index value of “1” (names[1]). In start, money is multiplied by 2, making it 10, so “money*2” would give you the value of 20.
3	A	Changing “Vector3” to “float” would work because you would just be multiplying a flat by another float. Changing “=” to “+” would not work because you can’t add a float to a Vector3.
4	A	Lowercase “float”, camelCase variables, Capitalized class & method names
5	C	Making a variable public will make it appear in the inspector.
6	D	If the consoles says a variable is not assigned, you most likely forgot to assign that variable by dragging on object onto it in the inspector.
7	B	Since it needs to “return” a value, it should have a return type of “private float ” as opposed to “private void .” Since it needs to take a number, it needs a float parameter (“float number”).
8	B	Since this is the “Enemy” class, we are testing for the enemy colliding with something. Since it destroys “ other.gameObject ”, it will destroy the spike.
9	D	The code cannot convert a Rigidbody type variable to a GameObject type variable, so you have to get the Rigidbody component from the gameObject
10	E	<ul style="list-style-type: none"> A. Functions/methods do not necessarily require parameters B. Functions/methods with a “void” return type <i>can</i> be passed parameters C. A Function/method with an “int” return type could not include the code, “return 0.5f;”, since 0.5f is a float D. If there was a function/method declared as “private void RenameObject(string newName)”, you would have to pass it a string parameter, such as RenameObject(“Steve”);



5.1 Clicky Mouse

Steps:

Step 1: Create project and switch to 2D view

Step 2: Create good and bad targets

Step 3: Toss objects randomly in the air

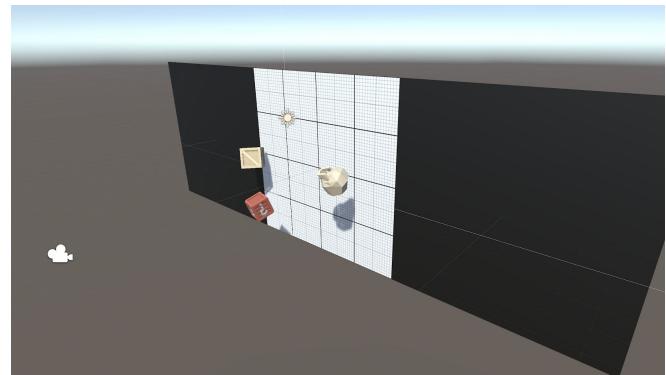
Step 4: Replace messy code with new methods

Step 5: Create object list in Game Manager

Step 6: Create a coroutine to spawn objects

Step 7: Destroy target with click and sensor

Example of project by end of lesson



Length: 60 minutes

Overview: It's time for the final unit! We will start off by creating a new project and importing the starter files, then switching the game's view to 2D. Next we will make a list of target objects for the player to click on: Three "good" objects and one "bad". The targets will launch spinning into the air after spawning at a random position at the bottom of the map. Lastly, we will allow the player to destroy them with a click!

Project Outcome: A list of three good target objects and one bad target object will spawn in a random position at the bottom of the screen, thrusting themselves into the air with random force and torque. These targets will be destroyed when the player clicks on them or they fall out of bounds.

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

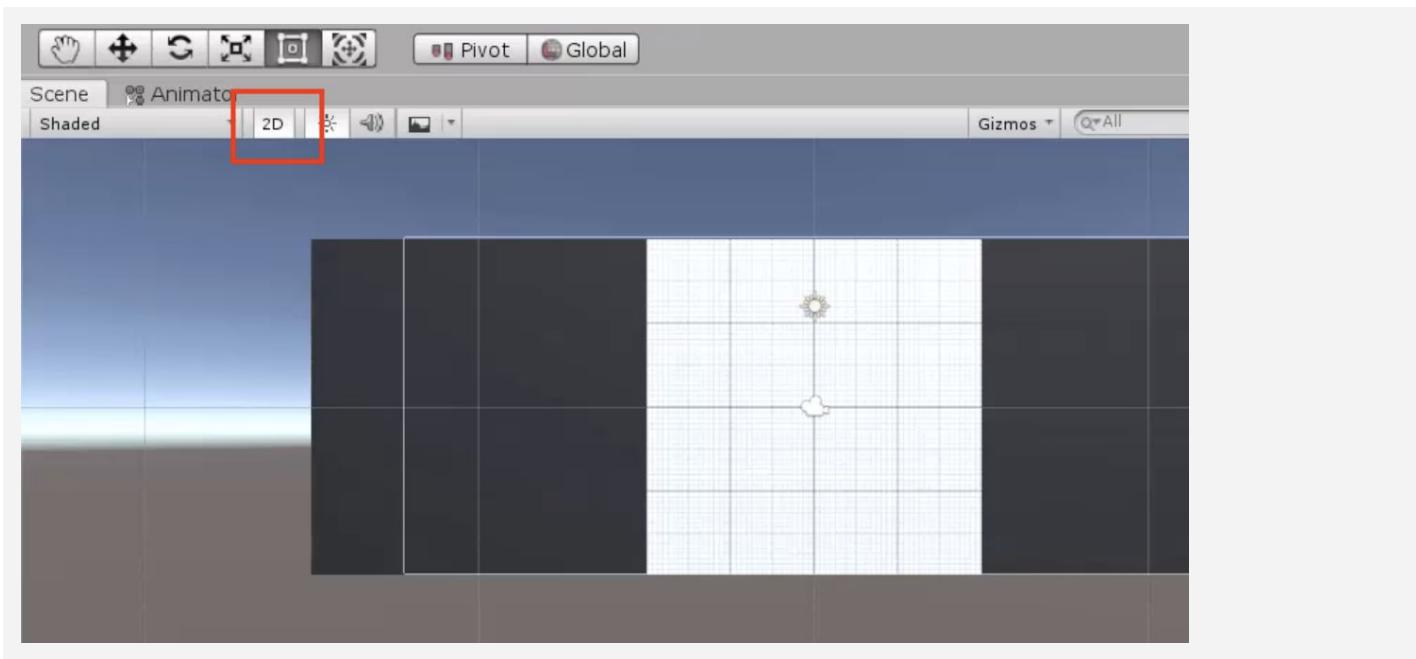
- Switch the game to 2D view for a different perspective
- Add torque to the force of an object
- Create a Game Manager object that controls game states as well as spawning
- Create a List of objects and return their length with Count
- Use While Loops to repeat code while something is true
- Use OnMouseDown to enable the player to click on things

Step 1: Create project and switch to 2D view

One last time... we need to create a new project and download the starter files to get things up and running.

1. Open **Unity Hub** and create “**Prototype 5**” in your course directory on correct version in 3D
2. Click on the link to access the Prototype 5 **starter files**, then **download and import** them into Unity
3. Open the **Prototype 5** scene, then delete the **sample scene** without saving
4. Click on the **2D icon** in Scene view to put Scene view in **2D**
5. (optional) Change the texture and color of the **background** and the color of the **borders**

- **New Concept:** 2D View
- **Demo:** Notice in 2D view: You can't rotate around objects or move them in the Z direction

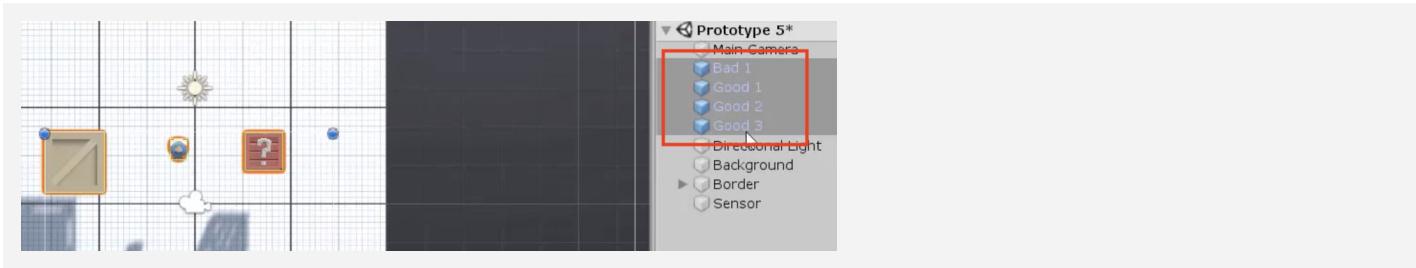


Step 2: Create good and bad targets

The first thing we need in our game are three good objects to collect, and one bad object to avoid. It'll be up to you to decide what's good and what's bad.

1. From the **Library**, drag 3 "good" objects and 1 "bad" object into the Scene, rename them "Good 1", "Good 2", "Good 3", and "Bad 1"
2. Add **Rigid Body** and **Box Collider** components, then make sure that Colliders surround objects properly
3. Create a new Scripts folder, a new "Target.cs" script inside it, attach it to the **Target objects**
4. Drag all 4 targets into the **Prefabs folder** to create "original prefabs", then **delete** them from the scene

- **Tip:** The bigger the collider boxes, the easier it will be to hit them
- **Tip:** Try selecting multiple objects and applying scripts/components - very handy



Step 3: Toss objects randomly in the air

Now that we have 4 target prefabs with the same script, we need to toss them into the air with a random force, torque, and position.

1. In **Target.cs**, declare a new **private Rigidbody targetRb**; and initialize it in **Start()**
2. In **Start()**, add an **upward force** multiplied by a **randomized speed**
3. Add a **torque** with randomized **xyz values**
4. Set the **position** with a randomized **X value**

- **New Function:** AddTorque
- **Tip:** Test with different values by dragging them in during runtime
- **Don't worry:** We're going to fix all these hard-coded values next

```
private Rigidbody targetRb;

void Start() {
    targetRb = GetComponent<Rigidbody>();
    targetRb.AddForce(Vector3.up * Random.Range(12, 16), ForceMode.Impulse);
    targetRb.AddTorque(Random.Range(-10, 10), Random.Range(-10, 10),
    Random.Range(-10, 10), ForceMode.Impulse);
    transform.position = new Vector3(Random.Range(-4, 4), -6); }
```

Step 4: Replace messy code with new methods

Instead of leaving the random force, torque, and position making our `Start()` function messy and unreadable, we're going to store each of them in brand new clearly named custom methods.

1. Declare and initialize new private float variables for `minSpeed`, `maxSpeed`, `maxTorque`, `xRange`, and `ySpawnPos`;
2. Create a new function for `Vector3 RandomForce()` and call it in `Start()`
3. Create a new function for `float RandomTorque()`, and call it in `Start()`
4. Create a new function for `RandomSpawnPos()`, have it return a new `Vector3` and call it in `Start()`

```
private float minSpeed = 12;    private float maxSpeed = 16;
private float maxTorque = 10;   private float xRange = 4;
private float ySpawnPos = -6;

void Start() {
    ... targetRb.AddForce(... RandomForce(), ForceMode.Impulse);
    targetRb.AddTorque(... RandomTorque(), RandomTorque(),
    RandomTorque(), ForceMode.Impulse);
    transform.position = new Vector3(... RandomSpawnPos); }

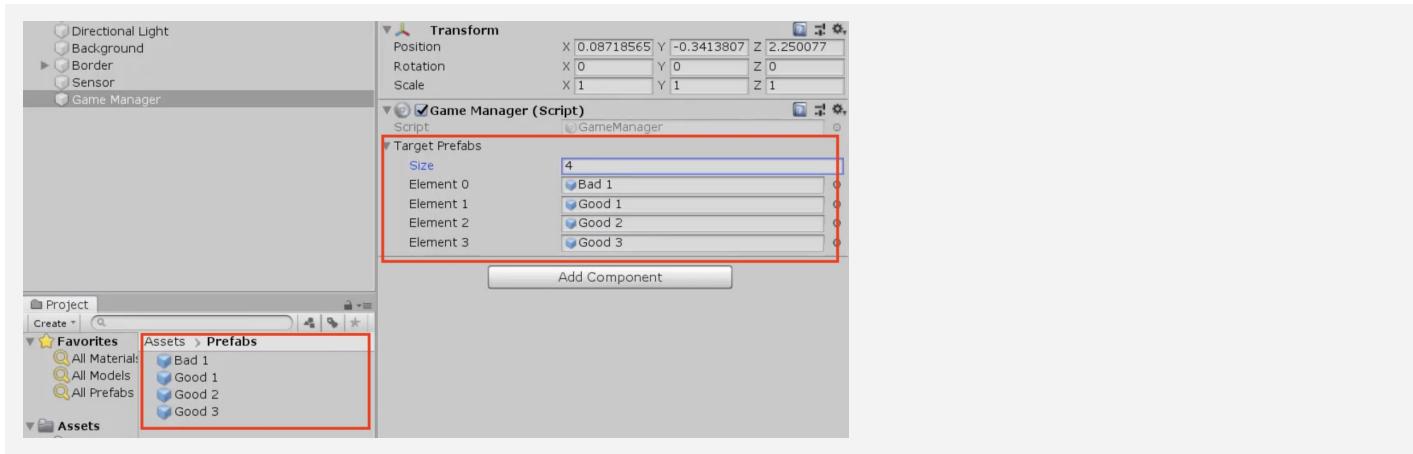
Vector3 RandomForce() { return Vector3.up * Random.Range(minSpeed, maxSpeed); }
float RandomTorque() { return Random.Range(-maxTorque, maxTorque); }
Vector3 RandomSpawnPos() { return new Vector3(Random.Range(-xRange, xRange),
ySpawnPos); }
```

Step 5: Create object list in Game Manager

The next thing we should do is create a list for these objects to spawn from. Instead of making a **Spawn Manager** for these spawn functions, we're going to make a **Game Manager** that will also control game states later on.

1. Create a new “**Game Manager**” **Empty object**, attach a new **GameManager.cs** script, then open it
2. Declare a new **public List<GameObject> targets;**, then in the Game Manager inspector, change the list **Size** to 4 and assign your **prefabs**

- **New Concept:** Lists
- **New Concept:** Game Manager
- **Demo:** Feel free to reference old code: We used an array instead of a list to spawn the animals in Unit 2



Step 6: Create a coroutine to spawn objects

Now that we have a *list of object prefabs*, we should instantiate them in the game using coroutines and a new type of loop.

1. Declare and initialize a new **private float spawnRate** variable
2. Create a new **IEnumerator SpawnTarget ()** method
3. Inside the new method, **while(true)**, wait **1 second**, generate a **random index**, and spawn a random **target**
4. In **Start()**, use the **StartCoroutine** method to begin spawning objects

- **Tip:** Feel free to reference old code: we used coroutines for the powerup cooldown in Unit 4
- **Tip:** Arrays return an integer with **.Length**, while Lists return an integer with **.Count**
- **New Concept:** While Loops

```
private float spawnRate = 1.0f;

void Start() { StartCoroutine(SpawnTarget()); }

IEnumerator SpawnTarget() {
    while (true) {
        yield return new WaitForSeconds(spawnRate);
        int index = Random.Range(0, targets.Count);
        Instantiate(targets[index]); } }
```

Step 7: Destroy target with click and sensor

Now that our targets are spawning and getting tossed into the air, we need a way for the player to destroy them with a click. We also need to destroy any targets that fall below the screen.

1. In **Target.cs**, add a new method for **private void OnMouseDown()** {}, and inside that method, destroy the gameObject
2. Add a new method for **private void OnTriggerEnter(Collider other)** and inside that function, destroy the gameObject

- **New Function:** OnMouseDown
- **Tip:** There is also OnMouseUp, and OnMouseEnter, but Down is definitely the one we want
- **Tip:** You could use Update and check if target y position is lower than a certain value, but a sensor is better because it doesn't run all the time

```
private void OnMouseDown() {
    Destroy(gameObject);
}

private void OnTriggerEnter(Collider other) {
    Destroy(gameObject);
}
```

Lesson Recap

New Functionality

- Random objects are tossed into the air on intervals
- Objects are given random speed, position, and torque
- If you click on an object, it is destroyed

New Concepts and Skills

- 2D View
- AddTorque
- Game Manager
- Lists
- While Loops
- Mouse Events

Next Lesson

- We'll add some effects and keep track of score!



5.2 Keeping Score

Steps:

Step 1: Add Score text position it on screen

Step 2: Edit the Score Text's properties

Step 3: Initialize score text and variable

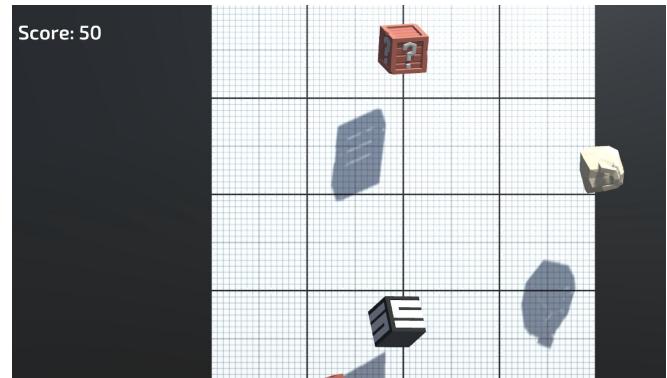
Step 4: Create a new UpdateScore method

Step 5: Add score when targets are destroyed

Step 6: Assign a point value to each target

Step 7: Add a Particle explosion

Example of project by end of lesson



Length: 60 minutes

Overview: Objects fly into the scene and the player can click to destroy them, but nothing happens. In this lesson, we will display a score in the user interface that tracks and displays the player's points. We will give each target object a different point value, adding or subtracting points on click. Lastly, we will add cool explosions when each target is destroyed.

Project Outcome: A "Score: " section will display in the UI, starting at zero. When the player clicks a target, the score will update and particles will explode as the target is destroyed. Each "Good" target adds a different point value to the score, while the "Bad" target subtracts from the score.

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

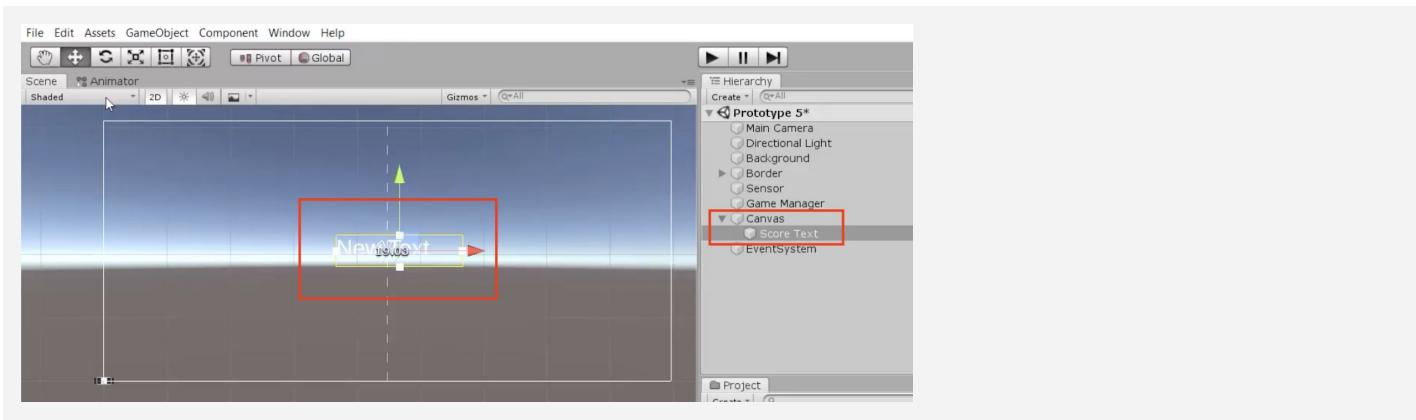
- Create UI Elements in the Canvas
- Lock elements and objects into place with Anchors
- Use variables and script communication to update elements in the UI

Step 1: Add Score text position it on screen

In order to display the score on-screen, we need to add our very first UI element.

1. Create > UI > **TextMeshPro text**, then if prompted click the button to **Import TMP Essentials**
2. Rename the new object “Score Text”, then **zoom out** to see the **canvas** in Scene view
3. Change the **Anchor Point** so that it is anchored from the **top-left corner**
4. In the inspector, change its **Pos X** and **Pos Y** so that it is in the top-left corner

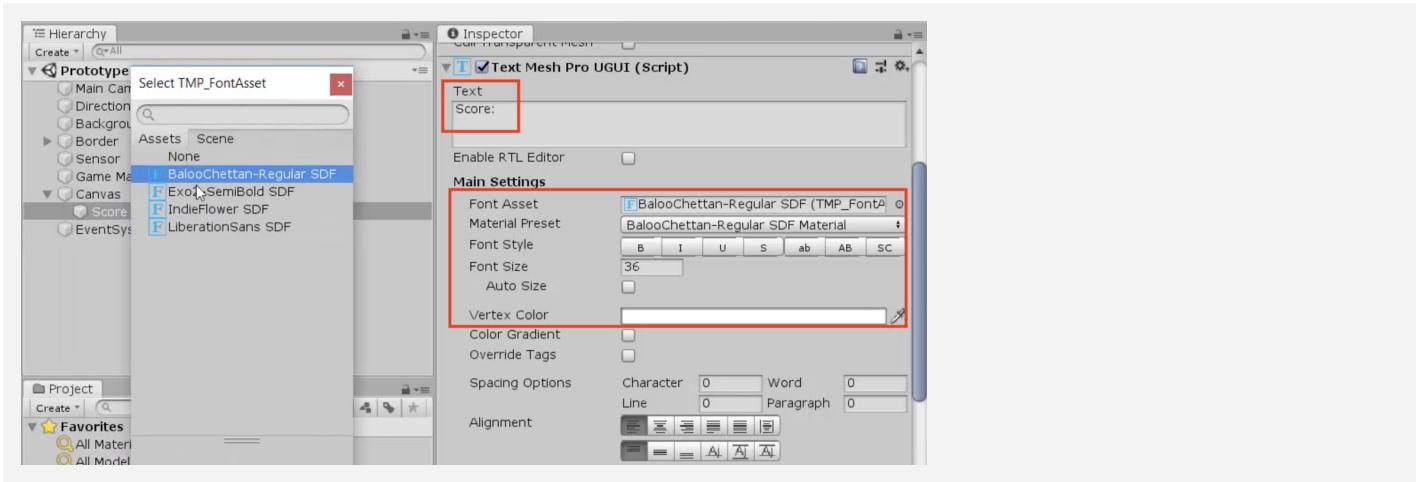
- **New Concept:** Text Mesh Pro / TMPro
- **New Concept:** Canvas
- **New Concept:** Anchor Points
- **Tip:** Look at how it displays in scene vs game view. It may be hard to see white text depending on the background



Step 2: Edit the Score Text's properties

Now that the basic text is in the scene and positioned properly, we should edit its properties so that it looks nice and has the correct text.

1. Change its text to “Score:”
2. Choose a **Font Asset**, **Style**, **Size**, and **Vertex color** to look good with your background



Step 3: Initialize score text and variable

We have a great place to display score in the UI, but nothing is displaying there! We need the UI to display a score variable, so the player can keep track of their points.

1. At the top of **GameManager.cs**, add “`using TMPro;`”
2. Declare a new **public TextMeshProUGUI scoreText**, then assign that variable in the inspector
3. Create a new **private int score** variable and initialize it in **Start()** as `score = 0;`
4. Also in **Start()**, set `scoreText.text = "Score: " + score;`

```
private int score;
public TextMeshProUGUI scoreText;

void Start() {
    StartCoroutine(SpawnTarget());
    score = 0;
    scoreText.text = "Score: " + score; }
```

- **New Concept:** Importing Libraries

Step 4: Create a new UpdateScore method

The score text displays the score variable perfectly, but it never gets updated. We need to write a new function that racks up points to display in the UI.

1. Create a new **private void UpdateScore()** method
2. Cut and paste `scoreText.text = "Score: " + score;` into the new method, then call **UpdateScore()** in **Start()**
3. Add the parameter **int scoreToAdd** to the **UpdateScore** method, then fix the error in **Start()** by passing it a value of **zero**
4. In **UpdateScore()**, increase the score by setting `score += scoreToAdd;`
5. Call **UpdateScore(5)** in the **spawnTarget()** function

- **Don't worry:** It doesn't make sense to add to score when spawned, this is just temporary

```
void Start() {
    ...
    score = 0;
    scoreText.text = "Score: " + score; UpdateScore(0); }

IEnumerator SpawnTarget() {
    while (true) { ... UpdateScore(5); }

private void UpdateScore(int scoreToAdd) {
    score += scoreToAdd;
    scoreText.text = "Score: " + score; }
```

Step 5: Add score when targets are destroyed

Now that we have a method to update the score, we should call it in the target script whenever a target is destroyed.

1. In Target.cs, create a reference to **private GameManager gameManager;**
2. Initialize GameManager in **Start()** using the **Find()** method
3. In GameManager.cs, make the **UpdateScore** method **public**
4. When a target is **destroyed**, call **UpdateScore(5);** then **delete** the method call from SpawnTarget()

- **Tip:** Feel free to reference old code: We used script communication in Unit 3 to stop the game on GameOver
- **Warning:** If you try to call UpdateScore while it's private, it won't work

```
private GameManager gameManager;

void Start() {
    ... gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();}

private void OnMouseDown() {
    Destroy(gameObject);    gameManager.UpdateScore(5);}

private-public void UpdateScore(int scoreToAdd) { ... }
```

Step 6: Assign a point value to each target

The score gets updated when targets are clicked, but we want to give each of the targets a different value. The good objects should vary in point value, and the bad object should subtract points.

1. In Target.cs, create a new **public int pointValue** variable
2. In each of the **Target** prefab's inspectors, set the **Point Value** to whatever they're worth, including the bad target's **negative value**
3. Add the new variable to **UpdateScore(pointValue);**

- **Tip:** Here's the beauty of variables at work. Each target can have their own unique pointValue!

```
public int pointValue;

private void OnMouseDown() {
    Destroy(gameObject);
    gameManager.UpdateScore(5 pointValue); }
```

Step 7: Add a Particle explosion

The score is totally functional, but clicking targets is sort of... unsatisfying. To spice things up, let's add some explosive particles whenever a target gets clicked!

1. In Target.cs, add a new **public ParticleSystem explosionParticle** variable
2. For each of your target prefabs, assign a **particle prefab** from Course Library > Particles to the **Explosion Particle** variable
3. In the **OnMouseDown()** function, **instantiate** a new explosion prefab

```
public ParticleSystem explosionParticle;

private void OnMouseDown() {
    Destroy(gameObject);
    Instantiate(explosionParticle, transform.position,
    explosionParticle.transform.rotation);
    gameManager.UpdateScore(5 pointValue); }
```

Lesson Recap

New Functionality	<ul style="list-style-type: none"> • There is a UI element for score on the screen • The player's score is tracked and displayed by the score text when hit a target • There are particle explosions when the player gets an object
New Concepts and Skills	<ul style="list-style-type: none"> • TextMeshPro • Canvas • Anchor Points • Import Libraries • Custom methods with parameters • Calling methods from other scripts
Next Lesson	<ul style="list-style-type: none"> • We'll use some UI elements again - this time to tell the player the game is over and reset our game!



5.3 Game Over

Steps:

- Step 1: Create a Game Over text object
- Step 2: Make GameOver text appear
- Step 3: Create GameOver function
- Step 4: Stop spawning and score on GameOver
- Step 5: Add a Restart button
- Step 6: Make the restart button work
- Step 7: Show restart button on game over

Example of project by end of lesson



Length: 60 minutes

Overview: We added a great score counter to the game, but there are plenty of other game-changing UI elements that we could add. In this lesson, we will create some “Game Over” text that displays when a “good” target object drops below the sensor. During game over, targets will cease to spawn and the score will be reset. Lastly, we will add a “Restart Game” button that allows the player to restart the game after they have lost.

Project Outcome: When a “good” target drops below the sensor at the bottom of the screen, the targets will stop spawning and a “Game Over” message will display across the screen. Just underneath the “Game Over” message will be a “Reset Game” button that reboots the game and resets the score, so the player can enjoy it all over again.

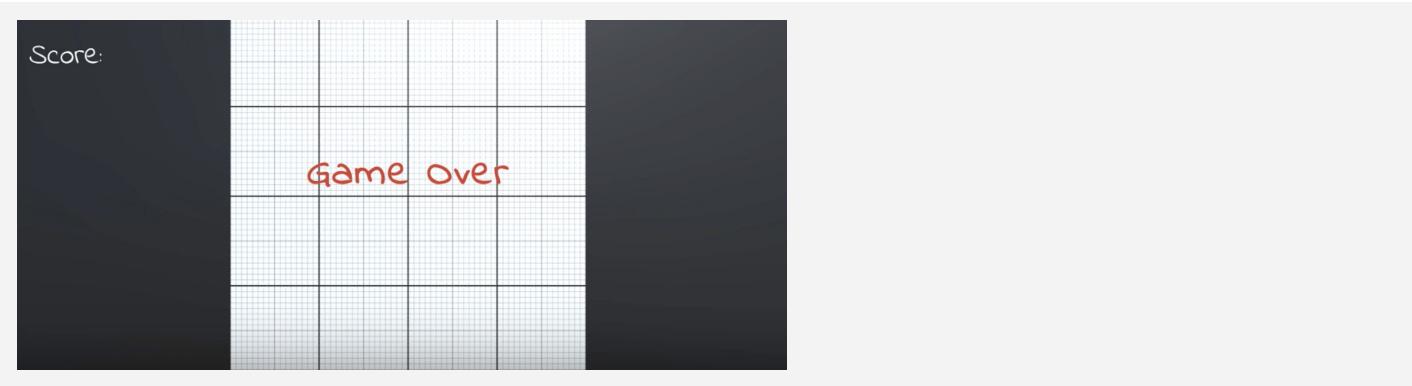
Learning Objectives: By the end of this lesson, you will be able to:

- Make UI elements appear and disappear with `.SetActive`
- Use Script Communication and Game states to have a working “Game Over” screen
- Restart the game using a UI button and Scene Management

Step 1: Create a Game Over text object

If we want some “Game Over” text to appear when the game ends, the first thing we’ll do is create and customize a new UI text element that says “Game Over”.

1. Right-click on the **Canvas**, create a new **UI > TextMeshPro - Text** object, and rename it “Game Over Text”
 2. In the inspector, edit its **Text, Pos X, Pos Y, Font Asset, Size, Style, Color, and Alignment**
 3. Set the “Wrapping” setting to “Disabled”
- **Tip:** The center of the screen is the best place for this Game Over message - it grabs the player’s attention



Step 2: Make GameOver text appear

We’ve got some beautiful Game Over text on the screen, but it’s just sitting and blocking our view right now. We should deactivate it, so it can reappear when the game ends.

1. In **GameManager.cs**, create a new **public TextMeshProUGUI gameOverText;** and assign the **Game Over** object to it in the inspector
 2. **Uncheck** the Active checkbox to **deactivate** the Game Over text by default
 3. In **Start()**, activate the Game Over text
- **Don’t worry:** We’re just doing this temporarily to make sure it works

```
public TextMeshProUGUI gameOverText;

void Start() {
    ...
    gameOverText.gameObject.SetActive(true);
}
```

Step 3: Create GameOver function

We've temporarily made the "Game Over" text appear at the start of the game, but we actually want to trigger it when one of the "Good" objects is missed and falls.

1. Create a new **public void GameOver()** function, and **move** the code that activates the game over text inside it
2. In Target.cs, call **gameManager.GameOver()** if a target collides with the **sensor**
3. Add a new "Bad" tag to the **Bad object**, add a condition that will only trigger game over if it's *not* a bad object

```
void Start() {
    ...
    gameOverText.gameObject.SetActive(true); }

public void GameOver() {
    gameOverText.gameObject.SetActive(true); }

<----->
private void OnTriggerEnter(Collider other) {
    Destroy(gameObject);
    if (!gameObject.CompareTag("Bad")) { gameManager.GameOver(); } }
```

Step 4: Stop spawning and score on GameOver

The "Game Over" message appears exactly when we want it to, but the game itself continues to play. In order to truly halt the game and call this a "Game Over", we need to stop spawning targets and stop generating score for the player.

1. Create a new **public bool isGameActive**;
2. As the **first line** in **Start()**, set **isGameActive = true**; and in **GameOver()**, set **isGameActive = false**;
3. To prevent spawning, in the **SpawnTarget()** coroutine, change **while (true)** to **while (isGameActive)**
4. To prevent scoring, in Target.cs, in the **OnMouseDown()** function, add the condition **if (gameManager.isGameActive) {**

```
public bool isGameActive;

void Start() { ... isGameActive = true; }

public void GameOver() { ... isGameActive = false; }

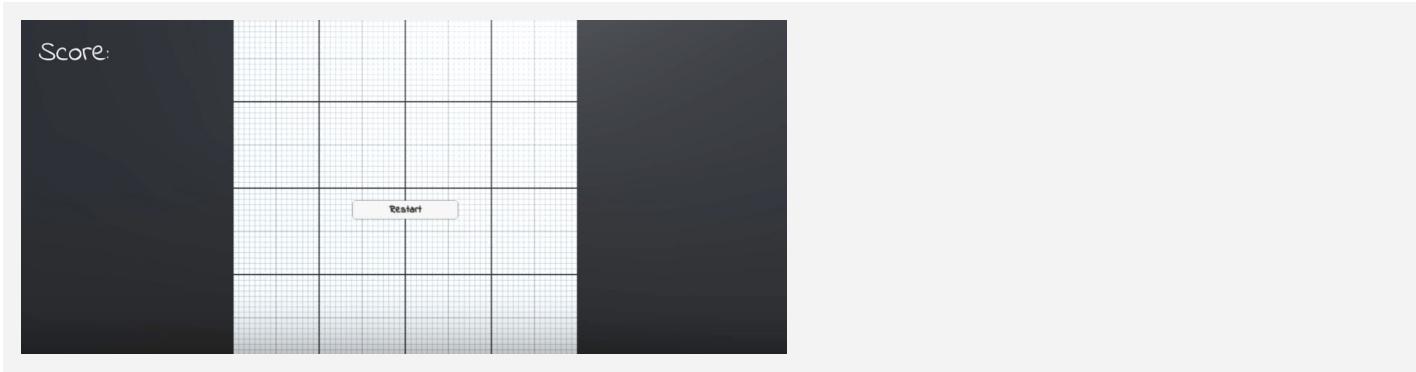
IEnumerator SpawnTarget() { while (true isGameActive) { ... }
<----->
private void OnMouseDown() {
    if (gameManager.isGameActive) { ... [all function code moved inside] }}
```

Step 5: Add a Restart button

Our Game Over mechanics are working like a charm, but there's no way to replay the game. In order to let the player restart the game, we will create our first UI button

1. Right-click on the **Canvas** and Create > UI > Button
2. Rename the button "Restart Button"
3. Temporarily **reactivate** the Game Over text in order to reposition the Restart Button nicely with the text, then **deactivate** it again
4. Select the Text child object, then edit its **Text** to say "Restart", its **Font**, **Style**, and **Size**

- **New Concept:**
Buttons



Step 6: Make the restart button work

We've added the Restart button to the scene and it LOOKS good, but now we need to make it actually work and restart the game.

1. In GameManager.cs, add **using UnityEngine.SceneManagement;**
2. Create a new **void RestartGame()** function that reloads the current scene
3. In the **Button's** inspector, click + to add a new **On Click event**, drag it in the **Game Manager** object and select the **GameManager.RestartGame** function

- **New Concept:** Scene Management
- **New Concept:** On Click Event
- **Don't worry:** The restart button is just sitting there for now, but we will fix it later

```
using UnityEngine.SceneManagement;

void RestartGame() {
    SceneManager.LoadScene(SceneManager.GetActiveScene().name); }
```

Step 7: Show restart button on game over

The Restart Button looks great, but we don't want it in our faces throughout the entire game. Similar to the "Game Over" message, we will turn off the Restart Button while the game is active.

1. At the top of GameManager.cs add **using UnityEngine.UI;**
2. Declare a new **public Button restartButton;** and assign the **Restart Button** to it in the inspector
3. **Uncheck** the "Active" checkbox for the **Restart Button** in the inspector
4. In the **GameOver** function, activate the **Restart Button**

- **Tip:** Adding "using UnityEngine.UI" allows you to access the Button class

```
using UnityEngine.UI;

public Button restartButton;

public void GameOver() { ...
    restartButton.gameObject.SetActive(true); }
```

Lesson Recap

New Functionality

- A functional Game Over screen with a Restart button
- When the Restart button is clicked, the game resets

New Concepts and Skills

- Game states
- Buttons
- On Click events
- Scene management Library
- UI Library
- Booleans to control game states

Next Lesson

- In our next lesson, we'll use buttons to really add some difficulty to our game



5.4 What's the Difficulty?

Steps:

- Step 1: Create Title text and menu buttons
- Step 2: Add a DifficultyButton script
- Step 3: Call SetDifficulty on button click
- Step 4: Make your buttons start the game
- Step 5: Deactivate Title Screen on StartGame
- Step 6: Use a parameter to change difficulty

Example of project by end of lesson



Length: 60 minutes

Overview: It's time for the final lesson! To finish our game, we will add a Menu and Title Screen of sorts. You will create your own title, and style the text to make it look nice. You will create three new buttons that set the difficulty of the game. The higher the difficulty, the faster the targets spawn!

Project Outcome: Starting the game will open to a beautiful menu, with the title displayed prominently and three difficulty buttons resting at the bottom of the screen. Each difficulty will affect the spawn rate of the targets, increasing the skill required to stop "good" targets from falling.

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

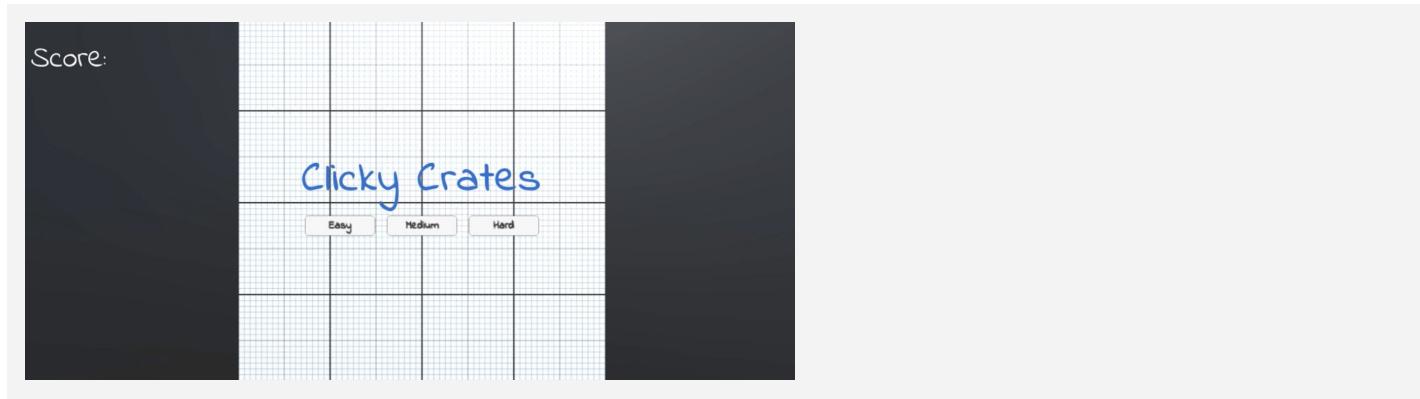
- Store UI elements in a parent object to create Menus, UI, or HUD
- Add listeners to detect when a UI Button has been clicked
- Set difficulty by passing parameters into game functions like SpawnRate

Step 1: Create Title text and menu buttons

The first thing we should do is create all of the UI elements we're going to need. This includes a big title, as well as three difficulty buttons.

1. Duplicate your **Game Over text** to create your **Title Text**, editing its name, text and all of its attributes
2. Duplicate your **Restart Button** and edit its attributes to create an "Easy Button" button
3. Edit and duplicate the new Easy **button** to create a "Medium Button" and a "Hard Button"

- **Tip:** You can position the title and buttons however you want, but you should try to keep them central and visible to the player



Step 2: Add a DifficultyButton script

Our difficulty buttons look great, but they don't actually do anything. If they're going to have custom functionality, we first need to give them a new script.

1. For all 3 new buttons, in the Button component, in the **On Click ()** section, click the **minus** (-) button to remove the `RestartGame` functionality
2. Create a new **DifficultyButton.cs** script and attach it to **all 3 buttons**
3. Add **`using UnityEngine.UI`** to your imports
4. Create a new **`private Button button;`** variable and initialize it in `Start()`

```
using UnityEngine.UI;

private Button button;

void Start() {
    button = GetComponent<Button>();
}
```

Step 3: Call SetDifficulty on button click

Now that we have a script for our buttons, we can create a **SetDifficulty** method and tie that method to the click of those buttons

1. Create a new **void SetDifficulty** function, and inside it, **Debug.Log(gameObject.name + " was clicked");**
2. Add the **button listener** to call the **SetDifficulty** function

- **New Function:** AddListener
- **Don't worry:** onClick.AddListener is similar what we did in the inspector with the Restart button
- **Don't worry:** We're just using Debug for testing, to make sure the buttons are working

```
void Start() {
    button = GetComponent<Button>();
    button.onClick.AddListener(SetDifficulty); }

void SetDifficulty() {
    Debug.Log(gameObject.name + " was clicked"); }
```

Step 4: Make your buttons start the game

The Title Screen looks great if you ignore the target objects bouncing around, but we have no way of actually starting the game. We need a **StartGame** function that can communicate with **SetDifficulty**.

1. In GameManager.cs, create a new **public void StartGame()** function and move everything from **Start()** into it
2. In DifficultyButton.cs, create a new **private GameManager gameManager;** and initialize it in **Start()**
3. In the **SetDifficulty()** function, call **gameManager.startGame();**

- **Don't worry:** Title objects don't disappear yet - we'll do that next

```
void Start() { ... }

public void StartGame() {
    isGameActive = true;
    score = 0;
    StartCoroutine(SpawnTarget());
    UpdateScore(0); }

<----->
private GameManager gameManager;

void Start () { ...
    gameManager = GameObject.Find("Game Manager").GetComponent<GameManager>(); }

SetDifficulty() { ... gameManager.startGame(); }
```

Step 5: Deactivate Title Screen on StartGame

If we want the title screen to disappear when the game starts, we should store them in an empty object rather than turning them off individually. Simply deactivating the single empty parent object makes for a lot less work.

1. Right-click on the Canvas and Create > Empty Object, rename it “Title Screen”, and drag the **3 buttons** and **title** onto it
2. In GameManager.cs, create a new **public GameObject titleScreen**; and assign it in the inspector
3. In **StartGame()**, deactivate the title screen object

```
public GameObject titleScreen;

StartGame() {
    ... titleScreen.gameObject.SetActive(false); }
```

Step 6: Use a parameter to change difficulty

The difficulty buttons start the game, but they still don’t change the game’s difficulty. The last thing we have to do is actually make the difficulty buttons affect the rate that target objects spawn.

1. In DifficultyButton.cs, create a new **public int difficulty** variable, then in the Inspector, assign the **Easy** difficulty as 1, **Medium** as 2, and **Hard** as 3
2. Add an **int difficulty** parameter to the **StartGame()** function
3. In **StartGame()**, set **spawnRate /= difficulty;**
4. Fix the error in DifficultyButton.cs by passing the difficulty parameter to **StartGame(int difficulty)**

- New Concept:
/= operator

```
public int difficulty;

void SetDifficulty() {
    ... gameManager.startGame(difficulty); }

<----->
public void StartGame(int difficulty) {
    spawnRate /= difficulty; }
```

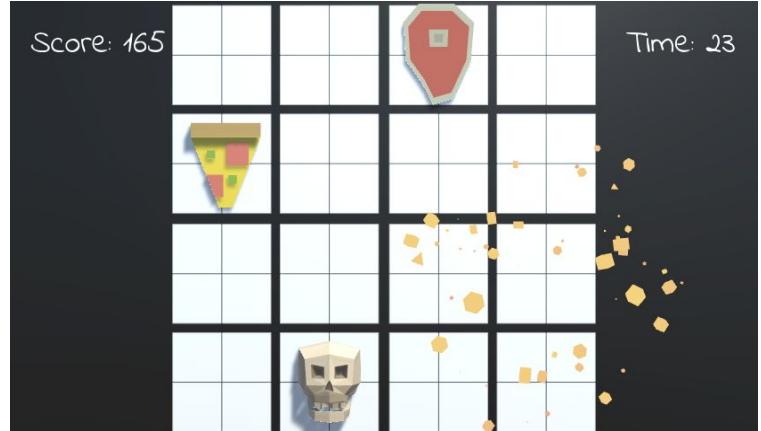
Lesson Recap

New Functionality	<ul style="list-style-type: none">• Title screen that lets the user start the game• Difficulty selection that affects spawn rate
New Concepts and Skills	<ul style="list-style-type: none">• AddListener()• Passing parameters between scripts• Divide/Assign (/=) operator• Grouping child objects
Next Lesson	<ul style="list-style-type: none">• In our next lesson, we'll use buttons to really add some difficulty to our game



Challenge 5

Whack-a-Food



Challenge Overview:

Put your User Interface skills to the test with this whack-a-mole-like challenge in which you have to get all the food that pops up on a grid while avoiding the skulls. You will have to debug buttons, mouse clicks, score tracking, restart sequences, and difficulty setting to get to the bottom of this one.

Challenge Outcome:

- All of the buttons look nice with their text properly aligned
- When you select a difficulty, the spawn rate changes accordingly
- When you click a food, it is destroyed and the score is updated in the top-left
- When you lose the game, a restart button appears that lets you play again

Challenge Objectives:

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

- Working with text and button objects to get them looking the way you want
- Using Unity's various mouse-related methods appropriately
- Displaying variables on text objects properly using concatenation
- Activating and deactivating objects based on game states
- Passing information between scripts using custom methods and parameters

Challenge Instructions:

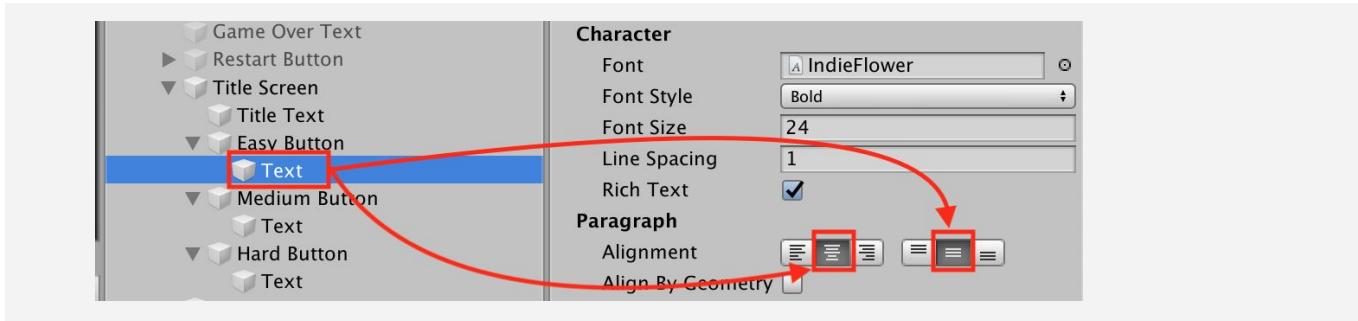
- Open your **Prototype 5** project
- **Download** the "Challenge 5 Starter Files" from the Tutorial Materials section, then double-click on it to **Import**
- In the *Project Window > Assets > Challenge 5 > Instructions* folder, use the "Challenge 5 - Outcome" video as a guide to complete the challenge

Challenge	Task	Hint
1 The difficulty buttons look messy	Center the text on the buttons horizontally and vertically	If you expand one of the button objects in the hierarchy, you'll see a "Text" object inside - you have to edit the properties of that "Text" object
2 The food is being destroyed too soon	The food should only be destroyed when the player clicks on it, not when the mouse touches it	OnMouseEnter() detects when the mouse <i>enters</i> an object's collider - OnMouseDown() detects when the mouse <i>clicks</i> on an object's collider
3 The Score is being replaced by the word "score"	It should always say, "Score: __" with the value displayed after "Score:"	When you set the score text, you have to add (concatenate) the word "Score: " <i>and</i> the actual score value
4 When you lose, there's no way to Restart	Make the Restart button appear on the game over screen	In the GameOver() method, make sure the restart button is being reactivated
5 The difficulty buttons don't do anything	When you click Easy, the spawnRate should be slower - if you click Hard, the spawnRate should be faster	There is no information (or parameter) being passed from the buttons' script to the Game Manager's script - you need to implement a difficulty parameter

Bonus Challenge	Task	Hint
X The game can go on forever	Add a "Time: __" display that counts down from 60 in whole numbers (i.e. 59, 58, 57, etc) and triggers the game over sequence when it reaches 0.	Google, "Unity Count down timer C#". It will involve subtracting "Time.deltaTime" and using the Mathf.Round() method to display only whole numbers.

Challenge Solution

- 1 Expand each of the “Easy”, “Medium”, and “Hard” buttons to access their “Text” object properties, then select the horizontal and vertical alignment buttons in the “Paragraph” properties



- 2 In TargetX.cs, change OnMouseEnter() to OnMouseDown()

```
private void OnMouseEnterDown() {
```

- 3 In GameManagerX.cs, in UpdateScore(), concatenate the word “Score: ” with the score value:

```
public void UpdateScore(int scoreToAdd) {
    score += scoreToAdd;
    scoreText.text = "score" "Score: " + score;
}
```

- 4 In GameManagerX.cs, in GameOver(), change SetActive(false) to “true”

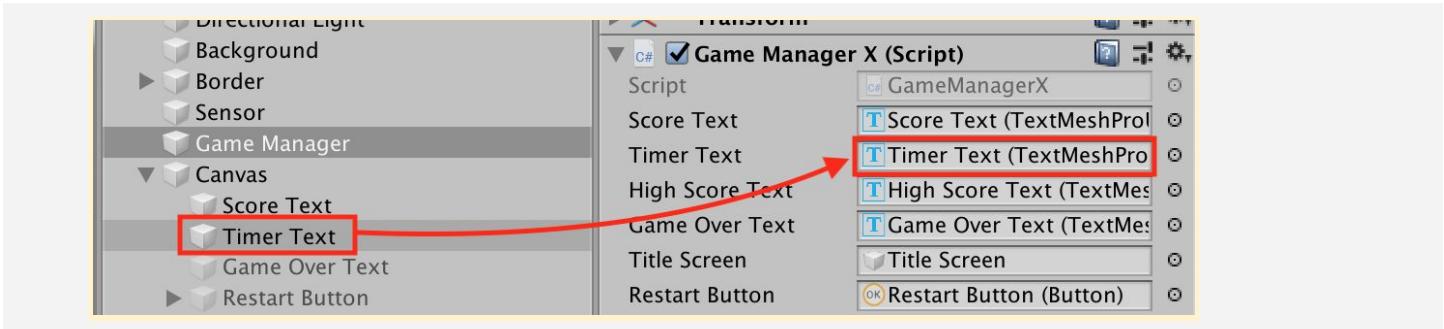
```
public void GameOver() {
    gameOverText.gameObject.SetActive(true);
    restartButton.gameObject.SetActive(false);
    ...
}
```

- 5 In GameManagerX.cs, in StartGame(), add an “int difficulty” parameter and divide the spawnRate by it. Then in DifficultyButtonX.cs, in SetDifficulty(), pass in the “difficulty” value from the buttons.

<code>GameManager.cs</code> <pre>public void StartGame(int difficulty){ spawnRate /= 5 - difficulty; ... }</pre>	<code>DifficultyButtonX.cs</code> <pre>void SetDifficulty() { ... gameManagerX.StartGame(difficulty); }</pre>
---	--

Bonus Challenge Solution

- X1** Duplicate the “Score Text” object in the hierarchy to create a new “Timer text” object, then in GameManagerX.cs declare a new **TextMeshProUGUI timerText** variable and assign it in the inspector



- X2** In GameManagerX.cs, in StartGame(), set your new timerText variable to your starting time

```
public void StartGame(int difficulty) {
    ...
    timeLeft = 60;
}
```

- X3** In GameManagerX.cs, add an Update() function that, if the game is active, subtracts from the timeLeft and sets the timerText to a rounded version of that timeLeft. Then, if timeLeft is less than zero, calls the game over method.

```
private void Update() {
    if (isGameActive) {
        timeLeft -= Time.deltaTime;
        timerText.SetText("Time: " + Mathf.Round(timeLeft));
        if (timeLeft < 0) {
            GameOver();
        }
    }
}
```



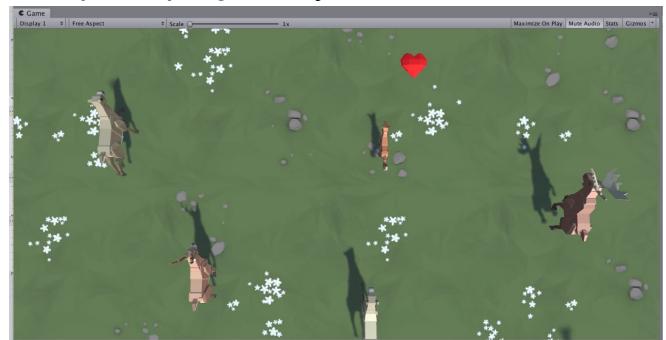
Unit 5 Lab

Swap out your Assets

Steps:

- Step 1: Import and browse the asset library
- Step 2: Replace player with new asset
- Step 3: Browse the Asset store
- Step 4: Replace all non-player primitives
- Step 5: Replace the background texture

Example of progress by end of lab



Length: 90 minutes

Overview: In this lab, you will finally replace those boring primitive objects with beautiful dynamic ones. You will either use assets from the provided course library or browse the asset store for completely new ones to give your game exactly the look and feel that you want. Then, you will go through the process of actually swapping in those new assets in the place of your placeholder primitives. By the end of this lab, your project will be looking a *lot* better.

Project Outcome: All primitive objects are replaced by actual 3D models, retaining the same basic gameplay functionality.

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

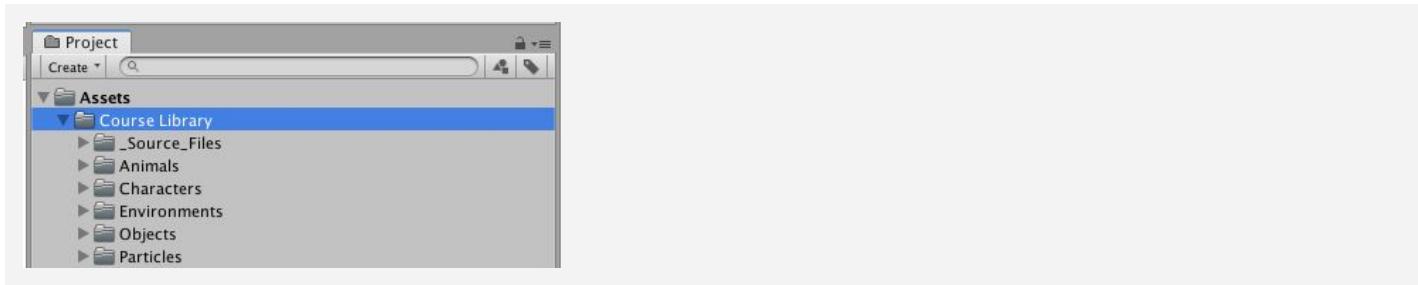
- Browse the asset store to find the perfect assets for your project
- Use Nested Prefabs to swap out placeholder objects with real assets
- Adjust material settings to get the resolution and look you want

Step 1: Import and browse the asset library

If we are going to swap out our primitive shapes with cool new assets, we need to import those assets first.

1. Click on the **link** to download the **Course Library** asset files, then **import** them into your project
2. **Close** the **Asset Store** window
3. Browse through the library to find the assets you would like to replace your Player and non-player objects with

- **Don't worry:** It will take longer than normal to import these files because it's a lot more files
- **Don't worry:** Even if you don't think you're going to use one of these assets for your player, just choose something for now to get used to the process

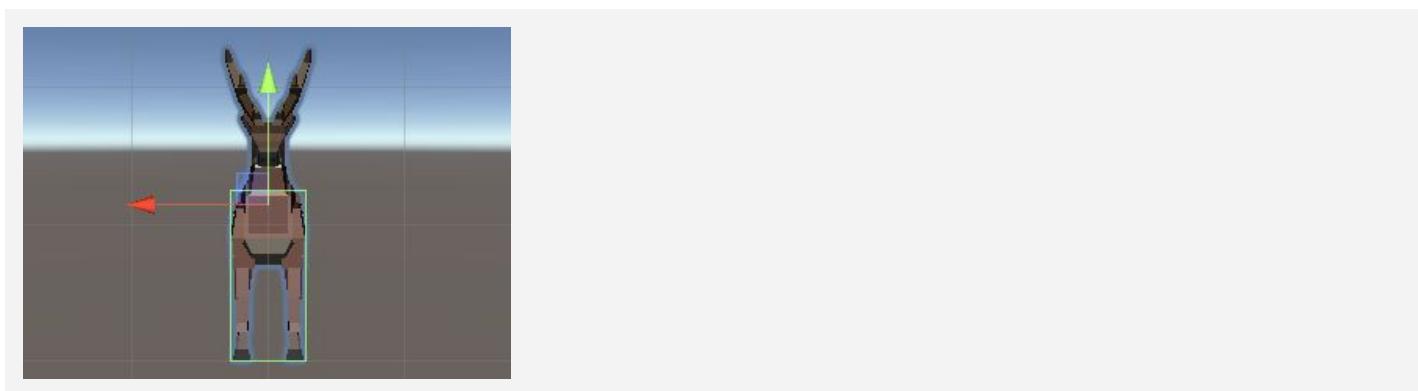


Step 2: Replace player with new asset

Now that we have the assets ready to go, the first thing we'll do is replace the Player object

1. **Drag** the Player object into the "Prefabs" folder to make it a prefab, then **double-click** on it to open the prefab editor
2. **Drag** the asset you want into the **hierarchy** to make it a nested prefab of the Player, then **scale** and **position** it so that it is around the same size and location
3. On the parent **Player** object itself, either **Edit** the collider to be the size of the new asset or **replace** it with a different type of collider (e.g. Box)
4. **Test** testing to make sure it works, then **uncheck** the **Mesh Renderer** component of the primitive

- **New:** Nested Prefabs
- **Tip:** Notice how the asset updates automatically in game view
- **Tip:** Isometric view is useful when resizing and repositioning child objects

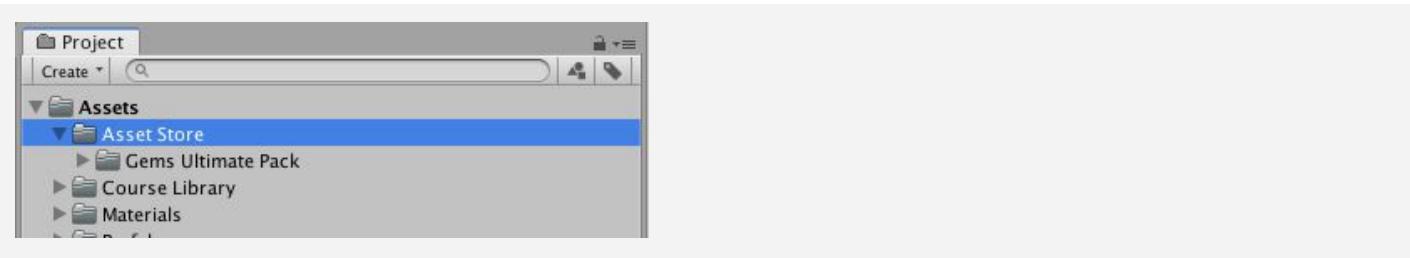


Step 3: Browse the Asset store

Even though we have a *really great asset library*, there may be certain assets you want that aren't in there. In that case, it might be good to try and find assets in the Unity Asset Store.

1. From the top menu, click **Window > Asset Store** to open the Asset Store window in Unity, then right-click on the tab and **Maximize** it to make it easier to browse
2. In the **Publisher** filter, search for "Synty Studios", then browse some of their asset packs
3. In the **Pricing** filter, drag the right handle back to only view "Free" assets, **remove** the Synty Studios filter, and search for "Low Poly"
4. If you see something you want to include in your project, **download** and **import** it into your project
5. **Drag** the imported assets into a new folder called "Asset Store", then browse through the imported assets

- **Warning:** This will only be possible if you can sign into a Unity account
- **Explain:** The assets for this course were made by Synty Studios, which are really good - as you can see, you normally have to pay for them
- **New:** Unity Asset Store
- **New:** "Low Poly" assets
- **Warning:** Only download "Low Poly" assets or your project will become huge, then not web- or mobile-friendly
- **Don't worry:** Even if you think you have all the assets you need, it's still good to take a look



Step 4: Replace all non-player primitives

Now that we know the basic concept of our project, let's figure out how we'll get it done.

1. Repeat the process you used to replace the player prefab with your other non-player objects
2. Test to make sure everything is working as expected

- **Warning:** Make sure that, if you are editing prefabs in the scene, to Override any changes you make

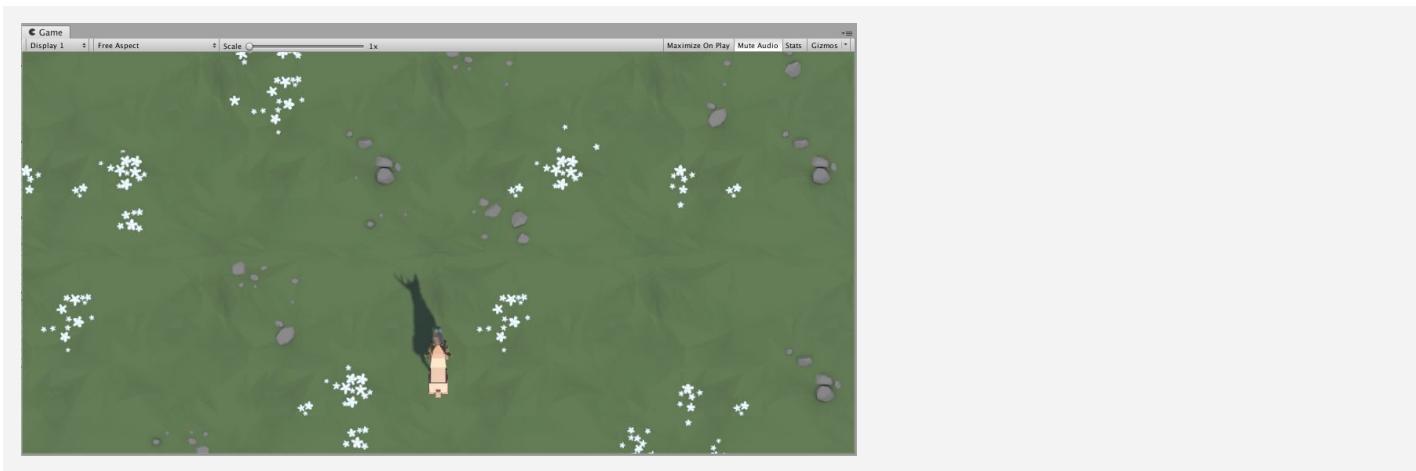


Step 5: Replace the background texture

Now that our dynamic objects have a new look, we should update the ground / background too.

1. From the Course Library > **Textures**, (or from a Unity Asset Store package), drag a new material onto the Ground / Background object
2. To adjust the material's **resolution**, in the Material properties (with the sphere next to it), change the Main Map Tiling X and Y values
3. To make the material less shiny, in the Material properties, **unchecked** the "Specular highlights" and "Reflections" settings

- **Tip:** You might want to adjust the resolution/tiling of the material, depending on the scale of the objects
- **Tip:** Natural ground materials like grass or dirt do not tend to show highlights or reflections



Lesson Recap

New Progress

- Primitive objects replaced with new assets that function the same way

New Concepts and Skills

- Art workflow
- High vs. Low Poly
- Asset Store
- Nested Prefabs
- Material properties



Quiz Unit 5

QUESTION

CHOICES

1 Which of the following follows Unity naming conventions (especially as they relate to capitalization)?

1. `public void MultiplyScore(int currentScore) { }`
2. `public void multiplyScore(int CurrentScore) { }`
3. `public Void MultiplyScore(Int currentScore) { }`
4. `public Void MultiplyScore(int CurrentScore) { }`

2 If there is a boolean in script A that you want to access in script B, which of the following are true:

1. You need a reference to script A in script B
2. The boolean needs to be public
3. The boolean must be true
4. The boolean must be included in the Update method

3 Which code to fill in the blank will result in the object being destroyed?

```
string name = "player"
bool isDead;
float health = 3;

if (_____) {
    Destroy(gameObject);
}
```

- a. Line 1
- b. Line 2
- c. Line 3
- d. Line 4

- a. 1 only
- b. 1 and 2 only
- c. 2 and 3 only
- d. 3 and 4 only
- e. 1, 2, and 3 only
- f. All are true

- a. `name = "player" && isDead && health < 5`
- b. `name != "player" && isDead != true && health > 5`
- c. `name == "player" && !isDead && health < 5`
- d. `name == "player" && isDead != true && health > 5`

- 4** You run your game and get the following error message in the console, “NullReferenceException: Object reference not set to an instance of an object”. Given the image and code below, what would resolve the problem?
- a. In the hierarchy, rename “Game Manager” to “gameManager”
 - b. In the hierarchy, rename “Game Manager” as “GameManager”
 - c. On Line 1, rename “GameManager” as “Game Manager”
 - d. On Line 3, remove the GetComponent code



```

1. private GameManager gameManager;
2. void Start() {
3.     gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();
4. }
```

- 5** Read the Unity documentation below about the OnMouseDown event and the code beneath it. What will the value of the “counter” variable be if the user clicked and held down the mouse over an object with a collider for 10 seconds?
- a. 0
 - b. 1
 - c. 99
 - d. 100
 - e. A value over 100

MonoBehaviour.OnMouseDown()

[Leave feedback](#) [Other Versions](#)

Description

OnMouseDown is called when the user has clicked on a [GUIElement](#) or [Collider](#) and is still holding down the mouse.

OnMouseDown is called every frame while the mouse is down.

```

int counter = 0;
void OnMouseDown() {
    if (counter < 100) {
        counter++;
    }
}
```

- 6** Based on the code below, what will be displayed in the console when the button is clicked?

- a. "Welcome, Robert Smith"
- b. "Welcome, firstName Smith"
- c. "Button is ready"
- d. "Welcome + Robert + Smith"

```
private Button button;
private string firstName = "Robert";

void Start() {
    button = GetComponent<Button>();
    button.onClick.AddListener(DisplayWelcomeMessage);
    Debug.Log("Button is ready");
}

void DisplayWelcomeMessage() {
    Debug.Log("Welcome, " + firstName + " Smith");
}
```

- 7** You have declared a new Button variable as “private Button start;”, but there’s an error under the word “Button” that says “error CS0246: The type or namespace name ‘Button’ could not be found (are you missing a using directive or an assembly reference?)”

- a. You can’t name a button “start” because that’s the name of a Unity Event Function
- b. “Button” should be lowercase “button”
- c. You are missing “using UnityEngine.UI;” from the top of your class
- d. New Button variables must be made public

- 8** Look at the documentation and code below. Which of the following lines would *NOT* produce an error?

- a. Line 5
- b. Line 6
- c. Line 7
- d. Line 8

```
public void AddForceAtPosition(Vector3 force, Vector3 position, ForceMode mode = ForceMode.Force);
```

Parameters

force Force vector in world coordinates.

position Position in world coordinates.

Description

Applies force at position. As a result this will apply a torque and force on the object.

1. `public Vector3 explosion;`
2. `Vector3 startPos;`
3. `float startSpeed;`
4. `void Start {`
5. `AddForceAtPosition(50, 0, ForceMode.Impulse)`
6. `AddForceAtPosition(100, startPos, ForceMode.Impulse)`
7. `AddForceAtPosition(startSpeed, startPos, ForceMode.Impulse)`
8. `AddForceAtPosition(explosion, new Vector3(0, 0, 0), ForceMode.Impulse)`
9. `}`

- 9 If you wanted a button to display the message, "Hello!" when a button was clicked, what code would you use to fill in the blank?

- a. `(SendMessage);`
- b. `(SendMessage("Hello"));`
- c. `(SendMessage(string Hello));`
- d. `(SendMessage>Hello);`

```
private Button button;
void Start {
    button = GetComponent<Button>();
    button.onClick.AddListener_____;
}
void SendMessage() {
    Debug.Log("Hello!");
}
```

- 10 Which of the following is the correct way to declare a new List of game objects named "enemies"?

- a. Line 1
- b. Line 2
- c. Line 3
- d. Line 4

1. `public List[GameObjects] enemies;`
2. `public List(GameObject) "enemies";`
3. `public List<GameObjects> "enemies";`
4. `public List<GameObject> enemies;`

Quiz Answer Key

#	ANSWER	EXPLANATION
1	A	<code>public void MultiplyScore(int currentScore)</code> The “public”, “void”, and “int” keywords should be lowercase. Method names (like “MultiplyScore”) should be Title Case. variable names (like “currentScore”) should be camelCase.
2	B	You always need a variable reference to the script you’re trying to access and that variable must be public.
3	C	To compare a string, two ==’s are needed. By default, booleans are false unless declared as true and adding an exclamation mark before !isDead checks that it’s false. Since health = 3, checking “health < 5” is true.
4	B	<code>GameObject.Find("GameManager")</code> is returning a NullReferenceException error because there’s no object in the scene named that. If you renamed the “Game Manager” in the hierarchy to have no spaces, it would be fixed.
5	D	Since the function is called “every frame” the mouse is held, it will be called hundreds of times in 10 seconds. However, the condition will only be true if the counter is less than 99, meaning it will no longer increase after 100.
6	B	If you wanted it to say “Robert Smith”, you would have needed to use the variable name, <i>firstName</i> , <i>without</i> quotation marks.
7	C	In order to use some of the UI classes like “Button,” you need to include the “UnityEngine.UI” library
8	D	The first two required parameters are Vector3 variables. Only option D uses Vector3 variables for those parameters.
9	A	<code>SendMessage</code> does not require any parameters - it prints “Hello” no matter what when it is called. Also, when adding a listener, you just need to include the method’s name - no parentheses are required.
10	D	<code>public List<GameObject> enemies</code> is correct. <code><GameObject></code> should be in angle brackets. You don’t need “GameObject” to be plural because it’s the type of object it is. Variable names are never declared with quotation marks around them.



6.1 Project Optimization

Techniques:

1: Variable attributes

2: Unity Event Functions

3: Object Pooling

Length: 30 minutes

Overview: In this lesson, you will learn about a variety of different techniques to optimize your projects and make them more performant. You may not notice a huge difference in these small prototype projects, but when you're exporting a larger project, especially one for mobile or web, every bit of performance improvement is critical.

Project Outcome: Several of your prototype projects will have improved optimization, serving as examples for you to implement in your personal projects

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

- Recognize and use new variable attributes to keep values private, but still editable in the inspector
- Use the appropriate Unity Event Functions (e.g. Update vs. FixedUpdate vs. LateUpdate) to make your project run as smoothly as possible
- Understand the concept of Object Pooling, and appreciate when it can be used to optimize your project

1: Variable attributes

In the course, we only ever used “public” or “private” variables, but there are a lot of other variable attributes you should be familiar with.

1. Open your **Prototype 1** project and open the **PlayerController.cs** script
2. Replace the keyword “private” with **[SerializeField]**, then edit the values in the inspector
3. In **FollowPlayer.cs**, add the **[SerializeField]** attribute to the Vector3 **offset** variable
4. Try applying the “**readonly**”, “**const**”, or “**static**” attributes, noticing that all have the effect of removing the variable from the inspector

```
[SerializeField] private float speed = 30.0f;
[SerializeField] private float turnSpeed = 50.0f;

[SerializeField]-private Vector3 offset = new Vector3(0, 5, -7);
```

- **New Concept:** using **[SerializeField]** instead of public attribute
- **Tip:** “protected” is very similar to “private”, but would also allow access to derived classes

2: Unity Event Functions

In the course we only ever used the *default Update()* and *Start()* event functions, but there are others you might want to use in different circumstances

1. **Duplicate** your main Camera, rename it “Secondary Camera”, then **deactivate** the Main Camera
2. **Reposition** the Secondary camera in a first-person view, then edit the **offset variable** to match that position
3. Run your project and notice how choppy it is
4. In **PlayerController.cs**, change “Update” to “ FixedUpdate”, and in **FollowPlayer.cs**, change “Update” to “LateUpdate”, then **test again**
5. **Delete** the *Start()* function in both scripts, then reactivate your Main Camera

- **New Concept:** “Event Functions” are Unity’s default methods that run in a very particular order over the life of a script (e.g. Start and Update)
- **New Concept:** Update vs FixedUpdate vs LateUpdate
- **New Concept:** Awake vs Start
- **Tip:** If you’re not using Start or Update, it’s cleaner to delete them

```
PlayerController.cs
void FixedUpdate() { ... }
```

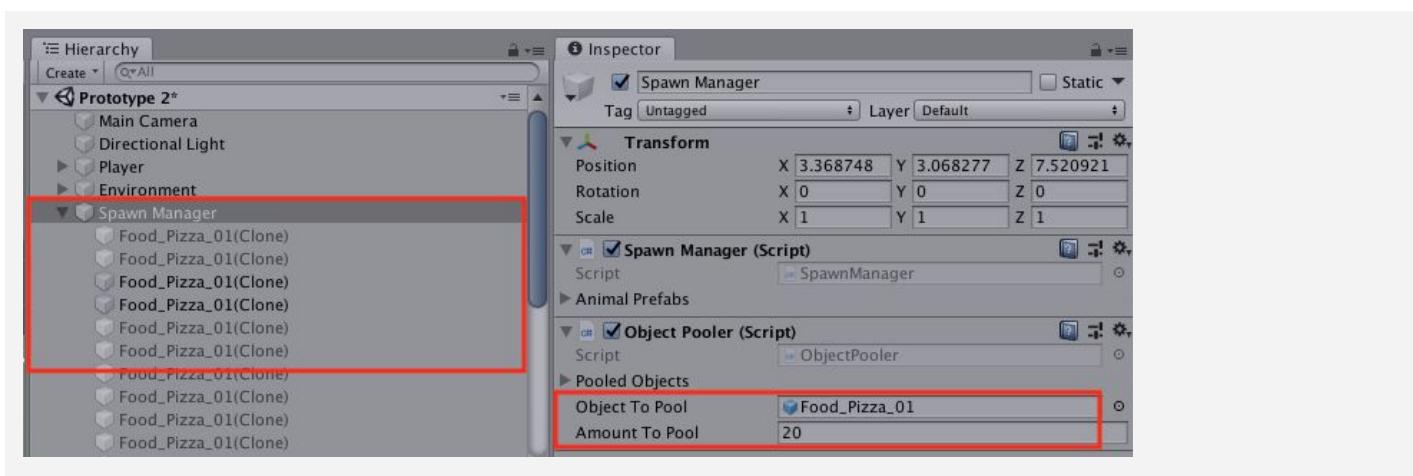
```
FollowPlayer.cs
void LateUpdate() { ... }
```

3: Object Pooling

Throughout the course, we've created a lot of prototypes that instantiated and destroyed objects during gameplay, but there's actually a more performant / efficient way to do that called **Object Pooling**.

1. Open **Prototype 2** and create a backup
2. Download the **Object Pooling** unity package and import it into your scene
3. Reattach the **PlayerController** script to your player and reattach the **DetectCollisions** script to your animal prefabs (**not** to your food prefab)
4. Attach the **ObjectPooler** script to your Spawn Manager, drag your projectile into the “**Objects To Pool**” variable, and set the “**Amount To Pool**” to 20
5. Run your project and see how the projectiles are activated and deactivated

- **Warning:** You will be overwriting your old work with this new system, so it's important to make a backup first in case you want to revert back
- **New Concept:** Object Pooling: creating a reusable “pool” of objects that can be activated and deactivated rather than instantiated and destroyed, which is much more performant
- **Tip:** Try reading through the new code in the ObjectPooler and PlayerController scripts
- **Don't worry:** If your project is small enough that you're not experiencing any performance issues, you probably don't have to implement this



Lesson Recap

New Concepts and Skills

- Optimization
- Serialized Fields
- readonly / const / static / protected
- Event Functions
- FixedUpdate() vs. Update() vs. LateUpdate()
- Awake() vs. Start()
- Object Pooling



6.2 Research and Troubleshooting

Steps:

Step 1: Make the vehicle use forces

Step 2: Prevent car from flipping over

Step 3: Add a speedometer display

Step 4: Add an RPM display

Step 5: Prevent driving in mid-air

Example of project by end of lesson



Length: 75 minutes

Overview: In this lesson, you will attempt to add a speedometer and RPM display for your vehicle in Prototype 1. In doing so, you will learn the process of doing online research when trying to implement new features and troubleshoot bugs in your projects. As you will find out, adding a new feature is very rarely as simple as it initially seems - you inevitably run into unexpected complications and errors that usually require a little online research. In this lesson, you will learn how to do that so that you can do it with your own projects.

Project Outcome: By the end of this lesson, the vehicle will behave with more realistic physics, and there will be a speedometer and Revolution per Minute (RPM) display.,

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

- Use Unity Forums, Unity Answers, and the online Unity Scripting Documentation to implement new features and troubleshoot issues with your projects

Step 1: Make the vehicle use forces

If we're going to implement a speedometer, the first thing we have to do is make the vehicle accelerate and decelerate more like a real car, which uses forces - as opposed to the Translate method.

1. Open your **Prototype 1** project and make a backup
2. Replace the Translate call with an AddForce call on the vehicle's Rigidbody, renaming the "speed" variable to "horsePower"
3. Increase the **horsePower** to be able to actually move the vehicle
4. To make the vehicle move in the appropriate direction, change AddForce to AddRelativeForce

- **New Concept:** using Unity Documentation
- **New Concept:** using Unity Answers
- **New Concept:** AddRelativeForce
- **Don't worry:** Still a big issue where the vehicle can drive in air and that it flips over super easily!

```
private Rigidbody playerRb;

void Start() {
    playerRb = GetComponent<Rigidbody>();
}

void FixedUpdate() {
    transform.Translate(Vector3.forward * speed * verticalInput);
    playerRb.AddRelativeForce(Vector3.forward * verticalInput * horsePower);
}
```

Step 2: Prevent car from flipping over

Now that we've implemented real physics on the vehicles, it is very easy to overturn. We need to figure out a way to make our vehicle safer to drive.

1. Add wheel colliders to the wheels of your vehicle and edit their radius and center position, then disable any other colliders on the wheels
2. Create a new **GameObject centerOfMass** variable, then in Start(), assign the playerRb variable to the centerOfMass position
3. Create a new **Empty Child** object for the vehicle called "Center Of Mass", reposition it, and assign it to the **Center Of Mass** variable in the inspector
4. **Test** different center of mass positions, speed, and turn speed values to get the car to steer as you like

- **New Concept:** Wheel colliders
- **New Concept:** Center of Mass
- **Don't Worry:** We can still drive the vehicle when it's sideways or upside down
- **Warning:** This is still not the *proper* way to do vehicles - should actually be rotating / turning the wheels

```
[SerializeField] GameObject centerOfMass;

void Start() {
    playerRb.centerOfMass = centerOfMass.transform.position;
}
```

Step 3: Add a speedometer display

Now that we have our vehicle in a semi-drivable state, let's display the speed on the User Interface.

1. Add a new **TextMeshPro - Text** object for your "Speedometer Text"
2. Import the **TMPro library**, then create and assign new create a new **TextMeshProUGUI** variable for your **speedometerText**
3. Create a new float variables for your **speed**
4. In Update(), **calculate** the speed in mph or kph then **display** those values on the UI

- **Warning:** Will be going fast through adding the text, since we did this in prototype 5
- **New Concept:** RoundToInt

```
using TMPro;

[SerializeField] TextMeshProUGUI speedometerText;
[SerializeField] float speed;

private void Update() {
    speed = Mathf.Round(playerRb.velocity.magnitude * 2.237f); // 3.6 for kph
    speedometerText.SetText("Speed: " + speed + "mph");
}
```

Step 4: Add an RPM display

One other cool feature that a lot of car simulators have is a display of the RPM (Revolutions per Minute) - the tricky part is figuring out how to calculate it.

1. Create a new “RPM Text” object, then **create** and **assign** a new **rpmText variable** for it
2. In **Update()**, calculate the the RPMs using the Modulus/Remainder operator (%), then display that value on the UI

- **New Concept:** Modulus / Remainder (%) operator

```
[SerializeField] TextMeshProUGUI rpmText;
[SerializeField] float rpm;

private void Update() {
    rpm = Mathf.Round((speed % 30)*40);
    rpmText.SetText("RPM: " + rpm);
}
```

Step 5: Prevent driving in mid-air

Now that we have a mostly functional vehicle, there's one other big bug we should try to fix: the car can still accelerate/decelerate, turn, and increase in speed/rpm in mid-air!

1. Declare a new **List** of **WheelColliders** named **allWheels** (or **frontWheels/backWheels**), then assign each of your wheels to that list in the inspector
2. Declare a new **int wheelsOnGround**
3. Write a **bool IsOnGround()** method that returns true if all wheels are on the ground and false if not
4. Wrap the **acceleration**, **turning**, and **speed/rpm** functionality in if-statements that check if the car is on the ground

- **New Concept:** looping through lists
- **New Concept:** custom methods with bool returns
- **Tip:** if you use **frontWheels** or **backWheels**, make sure you only drag in two wheels and only test that **wheelsOnGround == 2**

```
[SerializeField] List<WheelCollider> allWheels;
[SerializeField] int wheelsOnGround;

if (IsOnGround()) {[ACCELERATION], [ROTATION], [SPEED/RPM]}

bool IsOnGround () {
    wheelsOnGround = 0;
    foreach (WheelCollider wheel in allWheels) {
        if (wheel.isGrounded) {
            wheelsOnGround++;
        }
    }
    if (wheelsOnGround == 2) {
        return true;
    } else {
        return false;
    }
}
```

Lesson Recap

New Concepts and Skills

- Searching on Unity Answers, Forum, Scripting API
- Troubleshooting to resolve bugs
- AddRelativeForce, Center of Mass, RoundToInt
- Modulus/Remainder (%) operator
- Looping through lists
- Custom methods with bool return



6.3 Sharing your Projects

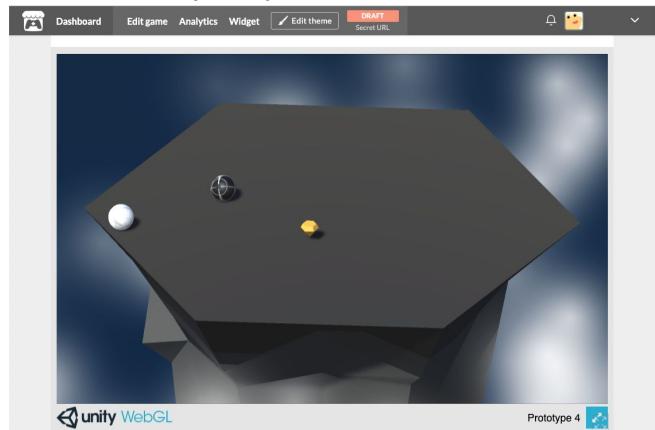
Steps:

Step 1: Install export Modules

Step 2: Build your game for Mac or Windows

Step 3: Build your game for WebGL

Example of project by end of lesson



Length: 30 minutes

Overview: In this lesson, you will learn how to build your projects so that they're playable outside of the Unity interface. First, you will install the necessary export modules to be able to publish your projects. After that, you will build your project as a standalone app to be played on Mac or PC computers. Finally, you will export your project for WebGL and even upload it to a game sharing site so that you can send it to your friends and family.

Project Outcome: Your project will be exported and playable as a standalone app on Mac/PC or for embedding online.

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

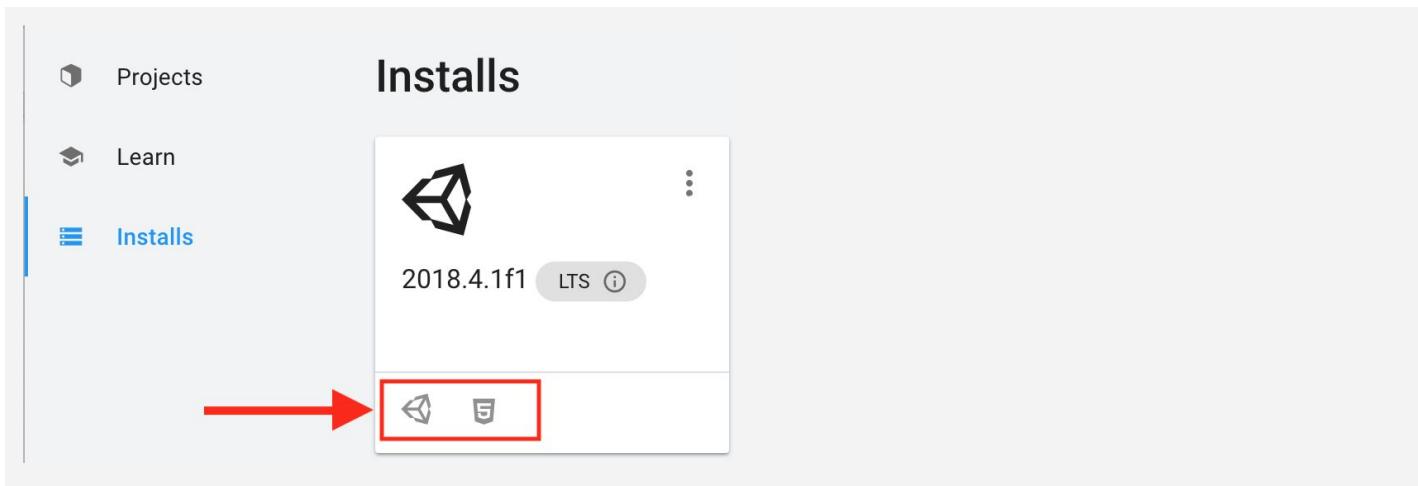
- Add and manage export modules for your Unity installs so you can choose which platforms to build for
- Build your projects for Mac or PC so they can be played as standalone apps
- Build your projects for WebGL so they can be uploaded and embedded online and shared with a single URL

Step 1: Install export Modules

Before we can export our projects, we need to add the "Export Modules" that will allow us to export for particular platforms.

1. Open **Unity Hub** and click to **Add Modules** to the version of Unity you have used in the course
2. Select **WebGL Build Support**, and either **Mac** or **Windows** build support, then click **Done** and wait for the installation to complete

- **Tip** - Mac and Windows will create apps for your computer and WebGL will allow you to publish online
- **Tip** - you should see little icons appear when it is complete
- **Tip** - WebGL is nice because you can more easily share it online and it is platform-independent

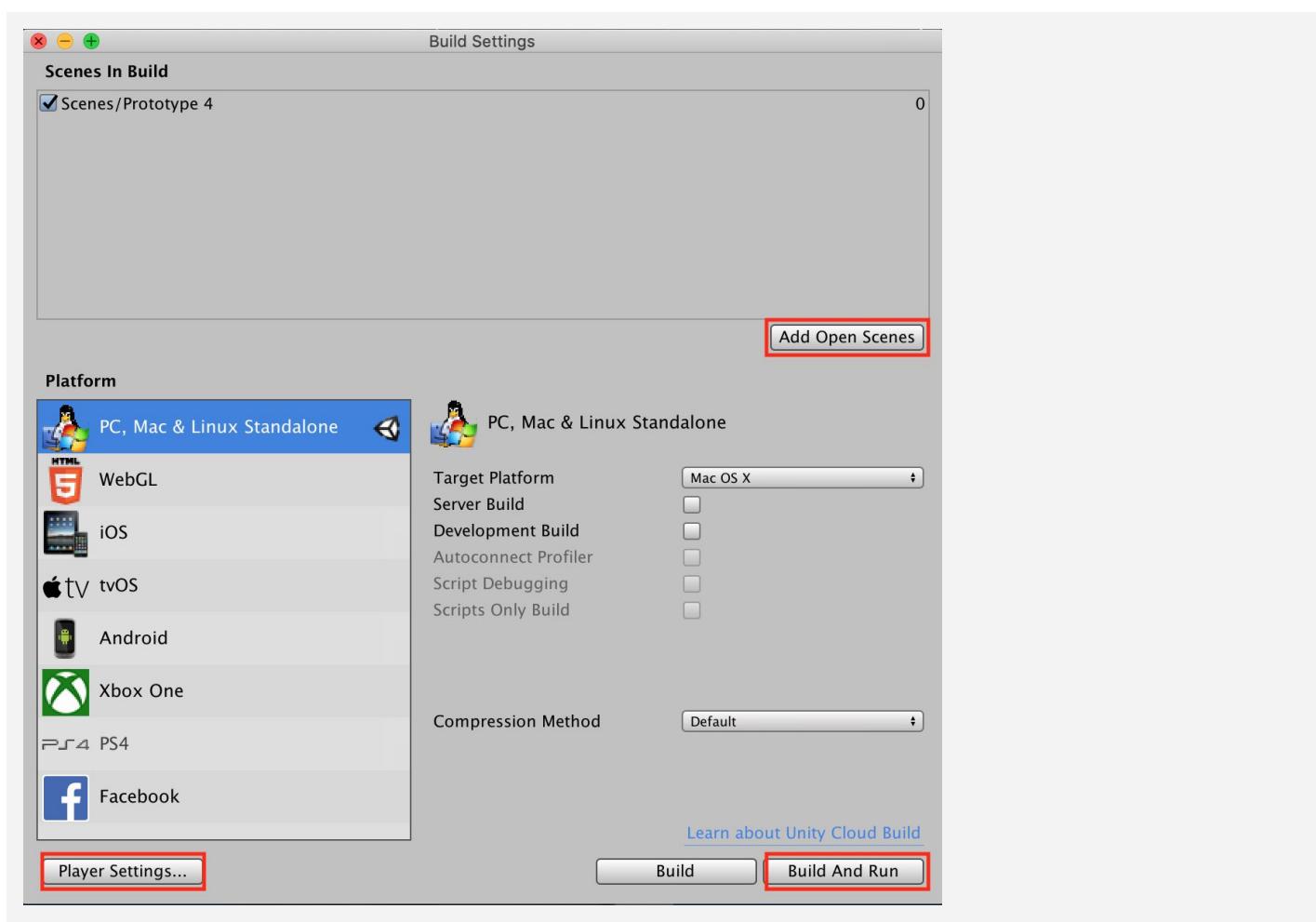


Step 2: Build your game for Mac or Windows

Now that we have the export modules installed, we can put them to use and export one of our projects

1. Open the project you would like to build (could be a prototype or your personal project)
2. In Unity, click **File > Build Settings**, then click **Add Open Scenes** to add your scene
3. Click **Player Settings** and adjust any settings you want, including making it "Windowed", "Resizable", and whether or not you want to enable the "Display Resolution Dialog"
4. Click **Build**, name your project, and save it inside a new folder inside your Create with Code folder called "Builds"
5. Play your game to test it out, then if you want, rebuild it with different settings

- **Don't worry** - a prototype that's not fully playable will be problematic when you share it because the user will have to close and reopen it to play it again, but that's OK for now
- **Tip** - since it's just a mini-game, it might be better to use "Windowed"
 - this also allows the player to more easily exit since we don't have a full UI to do that
- **Don't worry** - on Windows, you have an .exe file and a Data folder - on Mac, you just have a .app file
- **Warning** - it's kind of hard to distribute these as is because most email clients are cautious of executables like this

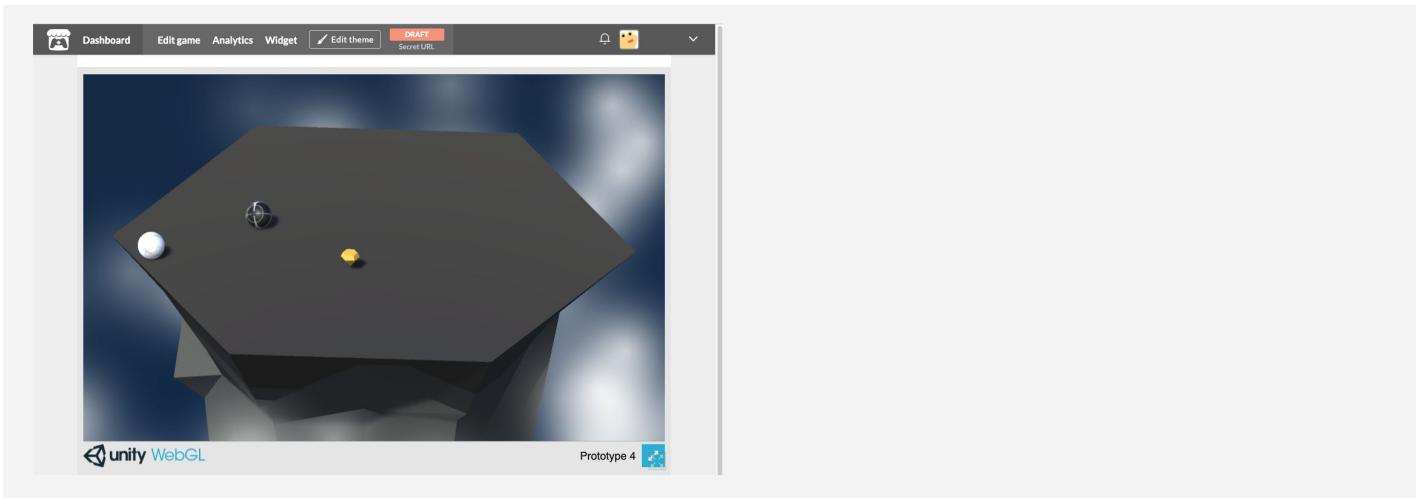


Step 3: Build your game for WebGL

Since it is pretty hard to distribute your games on Mac or Windows, it's a good idea to make your projects available online by building for WebGL.

1. Reopen the **Build Settings** menu, select **WebGL**, then click **Switch Platform**
2. Click **Build**, then save in your “Builds” folder with “[- WebGL](#)” in the name
3. Try clicking on **index.html** to run your project (you may have to try opening with different browsers)
4. Right-click on your WebGL build folder and **Compress/Zip** it into a .zip file
5. If you want, **upload** it to a game sharing site like [itch.io](#)

- **Warning** - it's easy to forget to click “Switch platform” and can be confusing
- **Don't worry** - building for WebGL can take a long time
- **Warning** - some browsers do not support opening WebGL programs from your computer
- **Tip** - If uploading your game to a site like [itch.io](#), make sure to choose “HTML” format and to “Play in browser”



Lesson Recap

New Concepts and Skills

- Installing export modules
- Building for Mac/PC
- Building for WebGL/HTML