BLACK BELT

Welcome to Black Belt!

Congratulations on making it to your Black Belt, the final belt at Code Ninjas! You have worked so hard to reach this point and you should feel incredibly proud of all that you've accomplished so far!

Unlike the previous belts, in Black Belt, you will have the opportunity to create your very own project in Unity, completely from scratch. Using all of the skills and knowledge you've gained from the other belts, you will create an exciting, creative, and collaborative project from the ground up!

Check out projects that Black Belt ninjas have already created on our Code Ninjas Black Belt site (http://codeninjas.com/blackbeltninjas).

In Black Belt, you will go through these 6 phases as you complete your final project:

Planning: Brainstorm and storyboard ideas to include in your project!

Prototyping: Create a simplified, first draft of your project!

Development - Alpha: Begin to program your project and build the bulk of it!

Development - Beta: Playtest your project and make final refinements!

Development - Release Candidate: Submit your project to get feedback from the CN Edu Team!

Going Gold: Share and present your final project!

Throughout Black Belt, you will have lots of opportunities to plan, iterate, and test out ideas for your project. After each phase, you will record a **Dev Diary**, where you can share reflections of your experience as you make progress on your final project.

Whenever you feel stuck, try to break down each task into smaller chunks, look over some of the previous projects you've created, or ask another ninja or Code Sensei for support!

Turn the page to get started on the first phase of your Black Belt project!

Planning Phase

The Planning Phase of your Black Belt project will be entirely **unplugged**! That means that you will not be using a computer for this phase. Instead, you will be planning, brainstorming, and storyboarding – all in your Black Belt Planning Guide – as you think about all of the elements that will be incorporated into your project. Let's get started!

In the Planning Phase, you will:

- **Brainstorm** ideas for your project.
- **Storyboard** your initial ideas by writing or drawing each scene and the different elements you plan to include.
- Begin to think about the **controls and user interface** (**UI**) in your project.
- Record Dev Diary #1 with a Code Sensei's help!

Planning Phase - Brainstorming

While you may already have an idea of what project you want to create, planning and iterating on your ideas will help you develop your vision and lead to a more robust final project!

The first step of any large project is to **brainstorm**. This will involve you starting to think about and jotting down your initial ideas for your Black Belt project. These ideas will change once you start building out your project in Unity, but it's important to start somewhere!

Ninja Planning Document

Complete the **Planning Phase – Brainstorming** portion of your Black Belt Ninja Planning Document to think through initial ideas for your project.

Planning Phase - Storyboarding

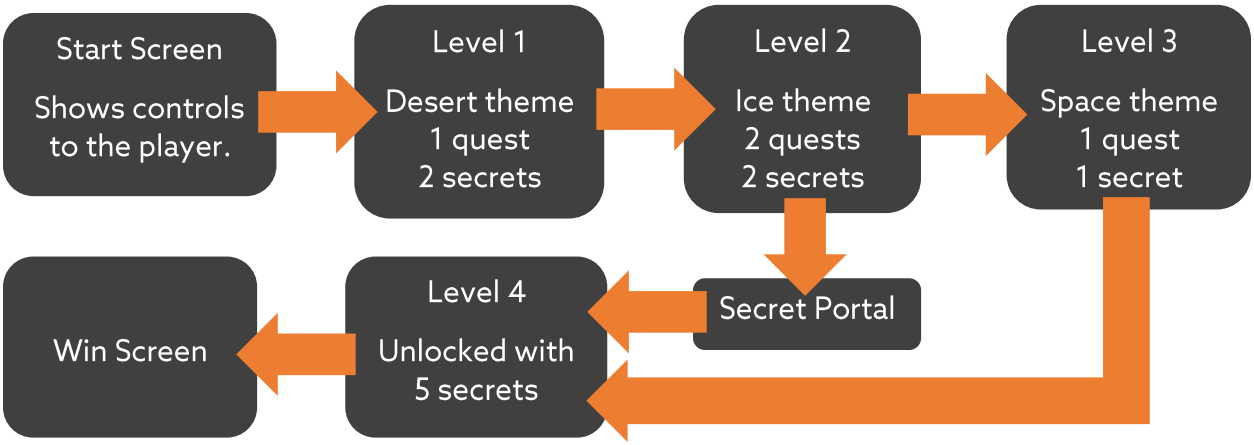
Now that you have thought through the basics of your project, you can start focusing on the details. It is important to have a clear vision for your characters, environment, and story before you start coding, and you can do that with a **storyboard**!

A **storyboard** is a sequence of images and text that act as a visual timeline for what will happen in your project. It should map out your ideas for how a user will progress from start to finish in your project, showing the decisions, obstacles, and goals they will encounter along the way.

A storyboard will help you think through different scenarios, such as:

- What happens at the beginning, middle, and end of the project?
- What different scenes will you need to include in the project?
- What does the user need to do to achieve their goal? What happens when they do or do not achieve it?

Your storyboard might look like this:



Ninja Planning Document

Complete the **Planning Phase – Storyboarding** portion of your Black Belt Ninja Planning Document by writing or sketching the characters, setting, and story of your project.

Planning Phase – Controls and User Interface

Think of different games and projects you've played and created. What type of screen appeared at the start? What information was available to you while playing? These components are all part of the **user interface**, or **UI**.

As you build out your own project, consider what information might be presented to the user at the beginning, middle, and end of the project. Since your UI should be visible without distracting the user from the project, designers often place UI elements around the edges of the screen. Consider the following questions:

- Can the user earn or lose lives or points during the project? Where and how will these be displayed?
- Are there collectables or an inventory that your user needs to keep up with?
- What other information does the user need to know as they interact with your project?

It's also important to plan the **controls**, or how the user will interact with your project. Consider the following questions:

- How will the user move a character around the screen?
- Will the user be able to use letter keys, direction buttons, their mouse, or a combination of these?
- What else will the user be able to control while navigating the project?

Ninja Planning Document

Complete the **Planning Phase – Controls and User Interface** portion of your Black Belt Ninja Planning Document by sketching the controls, the start screen, and user interface of your project.

Finishing the Planning Phase

As you wrap up the Planning Phase of your project, work with a Code Sensei to record a Dev Diary video, then review this checklist to demonstrate that you are ready to move on to the next phase of your Black Belt project!

Dev Diary #1 - Planning

Work with a Code Sensei to record a video in which you:

- Introduce yourself: Tell how old you are, how long you have been at Code Ninjas, and your 3 favorite things about being a ninja.
- Describe the project that you plan to create and something that you're excited about.
- Explain any challenges you anticipate in the prototyping phase, and how you plan to overcome them.

Planning Phase Checklist

- ✓ My Black Belt Ninja Planning Documents (Brainstorming, Storyboarding, and Controls and User Interface) are complete.
- ✓ My project's theme and story have been developed and are appropriate for a Black Belt project.
- ✓ My project's scope is manageable.
- ✓ I recorded Dev Diary #1 with my Code Sensei.

Prototyping Phase

Now that you've planned out your initial project ideas, it's time to bring them to life! In the Prototyping Phase, you will create a simplified **prototype**, or "first draft" of your project in Unity.

In this prototype, you should aim to build out *only* the core elements of your project based on your ideas from the Planning Phase. This will help you to test out your ideas as you create the structure of your project. Don't worry about too many of the fine details—you'll work on those in the Development phases!

As with any big project, it's okay if new ideas arise and you want to add or change anything you came up with during the Planning Phase. This project is completely yours and should change as you develop it!

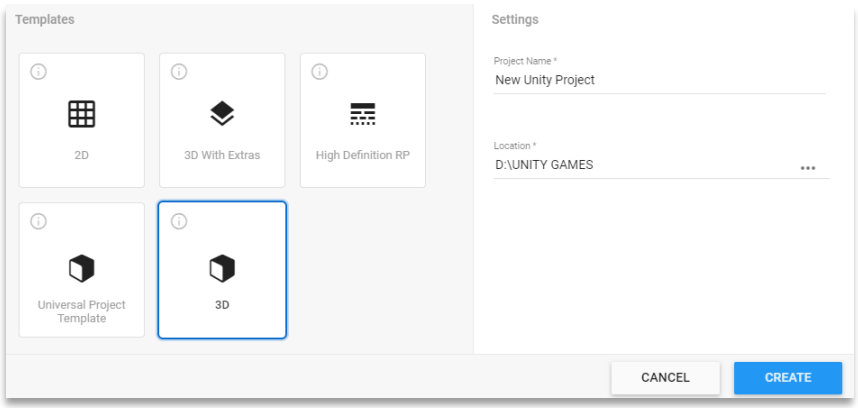
In the Prototyping Phase, you will:

- Create a new project in Unity.
- Add basic shapes to create a **prototype** of your project.
- Plan out your project's game mechanics using **pseudocode**.
- Plan and program the **game mechanics** in your project.
- Have a ninja and a Code Sensei **playtest** your prototype!
- Revise your project based on playtest feedback received.
- Record Dev Diary #2 with a Code Sensei's help!

Prototyping Phase - Starting a Unity Project

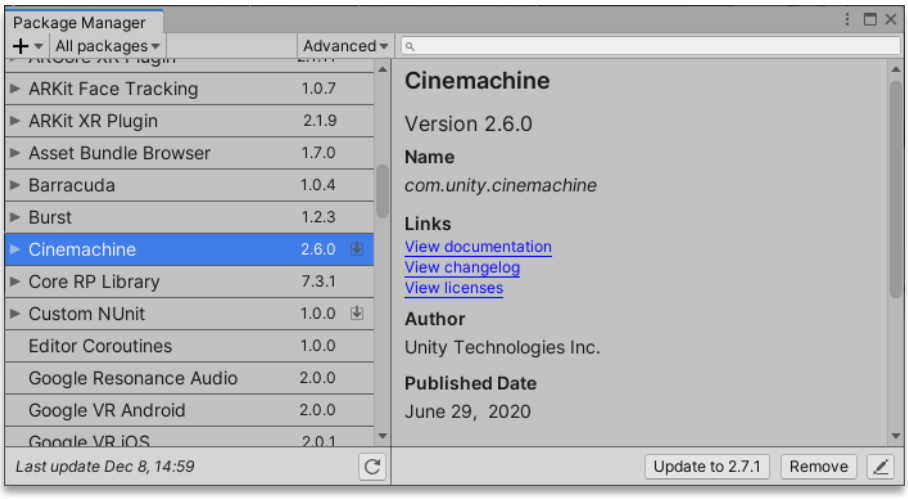
In order to develop a **prototype**, you will first create a new project in Unity. Follow these steps to set up your Black Belt project prototype!

Open the Unity Hub and start a new project. You cannot change the Project Name later, so title it something that you will remember.



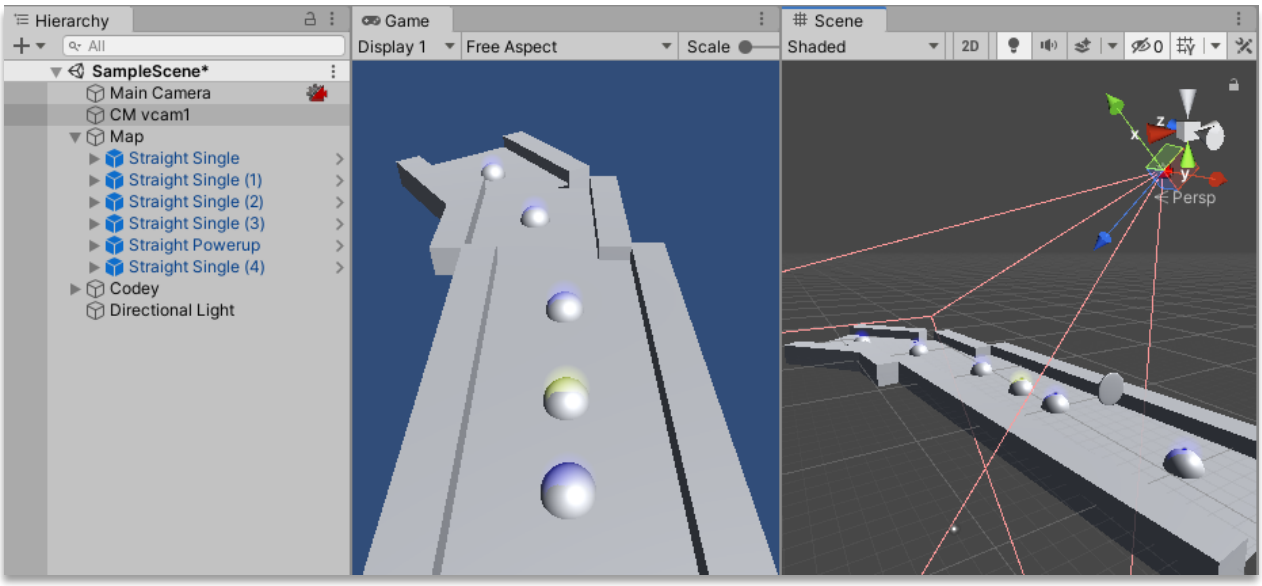
Based on your storyboard, rename your SampleScene.

If you plan to use Cinemachine, now is a good time to add it to your project.



Prototyping Phase – Creating a Prototype

After you've created a new project, use the ideas from the Planning Phase to begin building out your prototype in Unity. Use basic shapes as your assets so that you can focus on the logic and implementation, instead of the look and visuals. This will also allow you to more easily playtest your project and modify your mechanics in the next step.



The picture above is an example of a project prototype. Notice how simple shapes are used and the project does not look "complete". The purpose of the prototype is to test out the core elements of your project, without focusing on any assets, visuals, or effects. This will allow you to build a functioning project before adding in any "extras".

Prototyping Phase – Game Mechanics

All projects are made up of several different mechanics that work together to create a fun and unique experience for the user. For example, a game might combine running, jumping, and climbing to let the user explore a futuristic city.

Throughout the previous Unity belts, you have programmed many different mechanics, but you always focused on the programming because the planning was done already. But now, you get to imagine and plan your own mechanics!

Think about what mechanics you want in your project. What are the different Unity and C# concepts that you will need to use to make them work? Will you need to Instantiate a Prefab? Translate a Game Object? Invoke a Destroy Function? If you aren't sure what these terms mean or how to use them, check the documentation in this guide!

Once you have come up with a mechanic and determined the tools you will use, write **pseudocode**, or plain English text describing what the code will do, to describe the logic and code that your mechanic will use before programming anything in a script.

Ninja Planning Document

Complete the **Prototyping Phase – Game Mechanics #1 & 2** portion of your Black Belt Ninja Planning Document to plan out at least two game mechanics.

After you've planned out your mechanics, begin to add them into your prototype. Use the documentation in this guide to help you program the C# scripts you need to code your mechanics.

Prototyping Phase - Playtesting

As you reach the end of the Prototyping Phase of your project, have your Code Sensei and at least one other ninja **playtest** your project. **Playtesting** happens when other users play your project while it is still in development, and provide you with valuable feedback that can be used to improve your project!

Use the questions in your Black Belt Planning Document to collect playtest feedback from your Code Sensei and at least one other ninja. Then, use the feedback you've received to update your prototype before moving on to the next phase.

Playtesting Questions

- What did you like about my project prototype?
- What could be improved in my project prototype?

Then, create 2 of your own questions to ask your playtesters!

Ninja Planning Document

Complete the **Prototyping Phase – Playtesting** portion of your Black Belt Ninja Planning Document by asking at least 1 other ninja and your Code Sensei to playtest and provide feedback on your prototype.

Finishing the Prototyping Phase

As you wrap up the Prototyping Phase of your project, work with a Code Sensei to record a Dev Diary video, then review this checklist to demonstrate that you are ready to move on to the next phase of your Black Belt project!

Dev Diary #2 - Prototyping

Work with a Code Sensei to record a video in which you:

- Demo your prototype! Tell how you planned, implemented, and modified the mechanics as you added them to your prototype.
- Discuss any challenges you faced while building your prototype, and how you overcame them.
- Share something that you're excited about adding to your project in the Alpha phase!

Prototyping Phase Checklist

- ✓ I used simple shapes like squares and cubes to create my prototype in Unity.
- ✓ I planned out 2 (or more) game mechanics using pseudocode.
- ✓ I added at least 2 game mechanics in my prototype.
- ✓ My prototype was playtested by at least 1 other ninja.
- ✓ My prototype was playtested by at least 1 Code Sensei.
- ✓ My prototype has no bugs.
- ✓ I recorded Dev Diary #2 with my Code Sensei.

Alpha Phase

You've made it to the first Development phase! At this point, you should have a built-out prototype that uses basic shapes to show the structure of your project with a few mechanics added in.

Now that you've received and implemented your first round of playtesting feedback, it's time to build out your project! If you'd like, you can either continue working on the Unity project from the Prototyping Phase, or you can create a new one to start fresh.

During the Alpha Phase, you will replace the basic shapes from your prototype with actual project assets to create the players, characters, enemies, tools, collectables, and other objects in your project, as well as the overall world in which all of these assets will exist. You will also expand on the mechanics you have already begun setting up. Remember to review what you wrote in your planning documents as you are implementing elements and assets in your project.

Extra features, such as music, sound effects, and navigation screens, will be completed in the Beta Phase.

Once you're able to play your project from beginning to end without any bugs, you'll be ready to have your project playtested to move on to the Beta Phase!

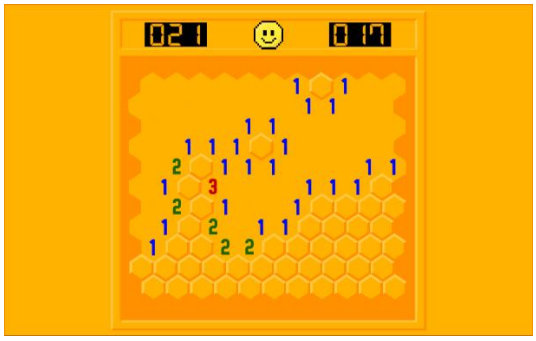
In the Alpha Phase, you will:

- Add onto your prototype or create a new project in Unity.
- Build out the bulk of your project: the **theme**, **assets**, and **game mechanics**.
- Play your project from beginning to end without bugs.
- Have a ninja and a Code Sensei **playtest** your project!
- Revise your project based on playtest feedback received.
- Record Dev Diary #3 with a Code Sensei's help!

Alpha Phase – Project Appearance

Now that you have the core elements of your project built out, it's time to consider the appearance of your project! Next to the mechanics, one of the most important aspects of a project is how it looks.

It's important to make sure your project has a well-defined **theme**. Notice the different themes of these Black Belt projects and how they transport the user instantly into the project.

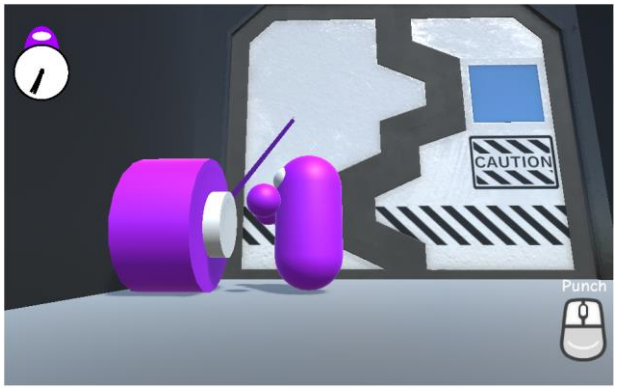


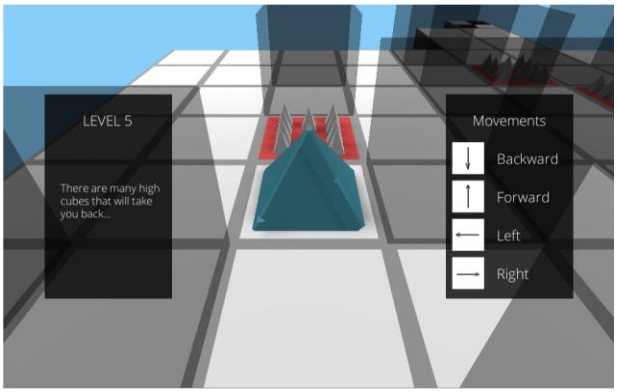
images used from Black Belt projects created by ninjas at Code Ninjas centers

You began thinking about a project theme in your brainstorming and storyboarding planning documents. Now, you'll refine your theme and consider the assets you will use to represent it. Use the following questions to guide your thinking:

- What kind of theme matches the user's goal in your project? How will it affect what happens during your project?
- Will the theme stay the same for the entire project or change across levels?
- What types of assets will you need to create your project theme?

You should also think about **visual clarity**. The user should be able to understand what they can interact with, where they can go, and what they should avoid. Notice how the different assets are used in these Black Belt projects to communicate with the user.





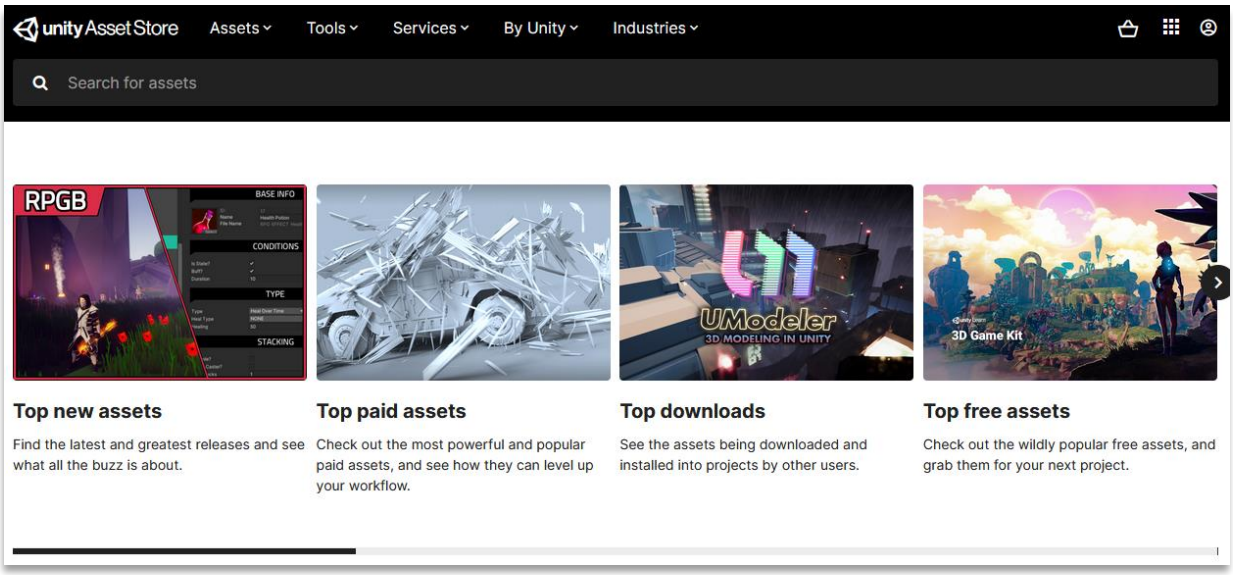
images used from Black Belt projects created by ninjas at Code Ninjas centers

As you think about visual clarity in your project, consider the following questions:

- How will the assets you select help the user understand what they can and cannot interact with?
- What assets will give information to the user? How will this happen?
- What assets will be used to tell the user where they should go and what they should avoid?

Alpha Phase - Assets and Design

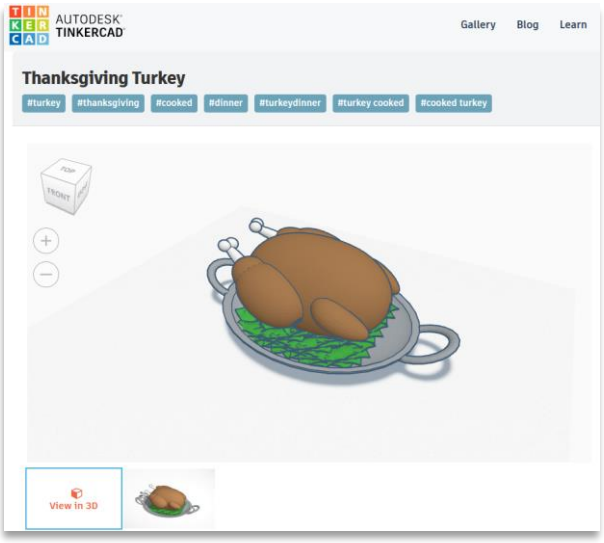
As you already know, you can use the Unity Asset Store to find assets to use in your project. Because your Black Belt project is entirely your own, you should modify assets that you find in the store to make sure your assets match your vision for your project!

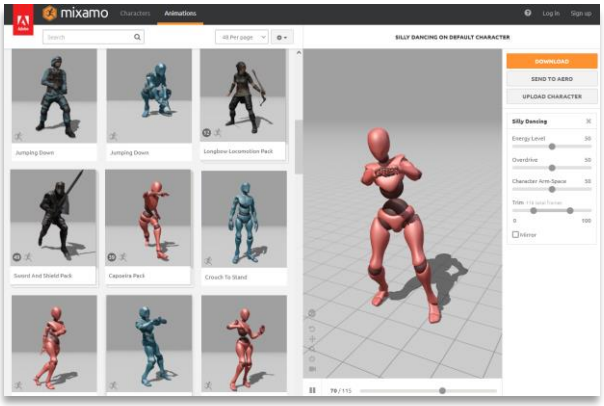


Ninja Planning Document

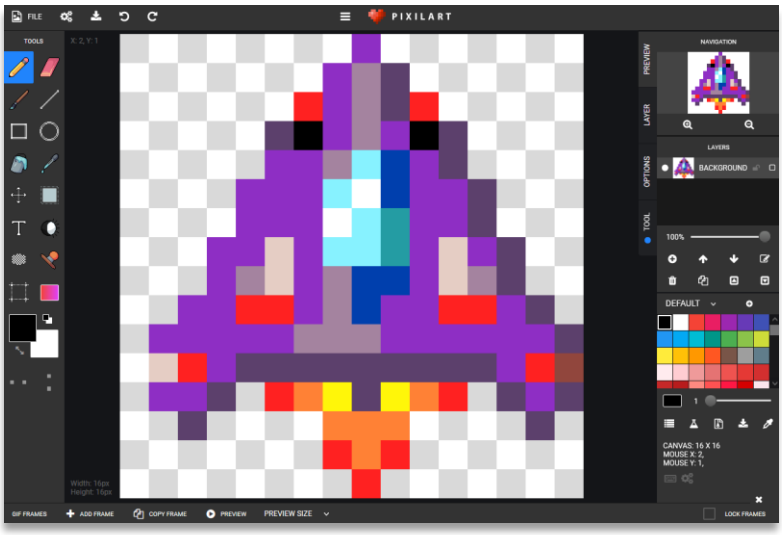
Complete the **Alpha Phase – Assets and Design** portion of your Black Belt Ninja Planning Document by finding at least three different assets and modifying them to use in your project.

You can also create your very own assets for your project. If your project is in 3D, you can use Tinkercad to make models and Mixamo to find free animations for your characters.





If your project is 2D, then you can use Pixilart to create your very own pixel art characters, objects, and environments.



Alpha Phase - Playtesting

As you reach the end of the Alpha Phase of your project, have your Code Sensei and at least one other ninja **playtest** your project. Use the questions in your Black Belt Planning Document to collect playtest feedback, then use their feedback to update your project before moving on to the next phase.

Playtesting Questions

- What did you like about my project?
- What could be improved in my project?

Then, create 2 of your own questions to ask your playtesters!

Ninja Planning Document

Complete the **Alpha Phase – Playtesting** portion of your Black Belt Ninja Planning Document by asking at least 1 other ninja and your Code Sensei to playtest and provide feedback on your project.

Finishing the Alpha Phase

As you wrap up the Alpha Phase of your project, work with a Code Sensei to record a Dev Diary video, then review this checklist to demonstrate that you are ready to move on to the next phase of your Black Belt project!

Dev Diary #3 - Alpha

Work with a Code Sensei to record a video in which you:

- Demo your project! Describe how a user interacts with the project and how you built on your work from the Prototyping phase.
- Share an example of how you have used feedback to develop your project so far.
- Share something that you're excited about adding to your project in the Beta phase!

Alpha Phase Checklist

- ✓ I built out the bulk of my project, so that it is almost fully functional at this point in the process.
- ✓ I used a variety of assets that fit the theme of my project.
- ✓ My project was playtested by at least 1 other ninja.
- ✓ My project was playtested by at least 1 Code Sensei.
- ✓ I revised my project based on feedback I received.
- ✓ My project is bug-free and can be played from start to finish.
- ✓ I recorded Dev Diary #3 with my Code Sensei.

Beta Phase

Now that most of your project is built out, it's time to add any final touches and refinements! During the Beta Phase, focus on fine-tuning the mechanics and overall user experience of your project. Now is also the time to add start/end screens, credits, sound effects, music, and a complete UI.

As you move through the Beta Phase, consider the following questions:

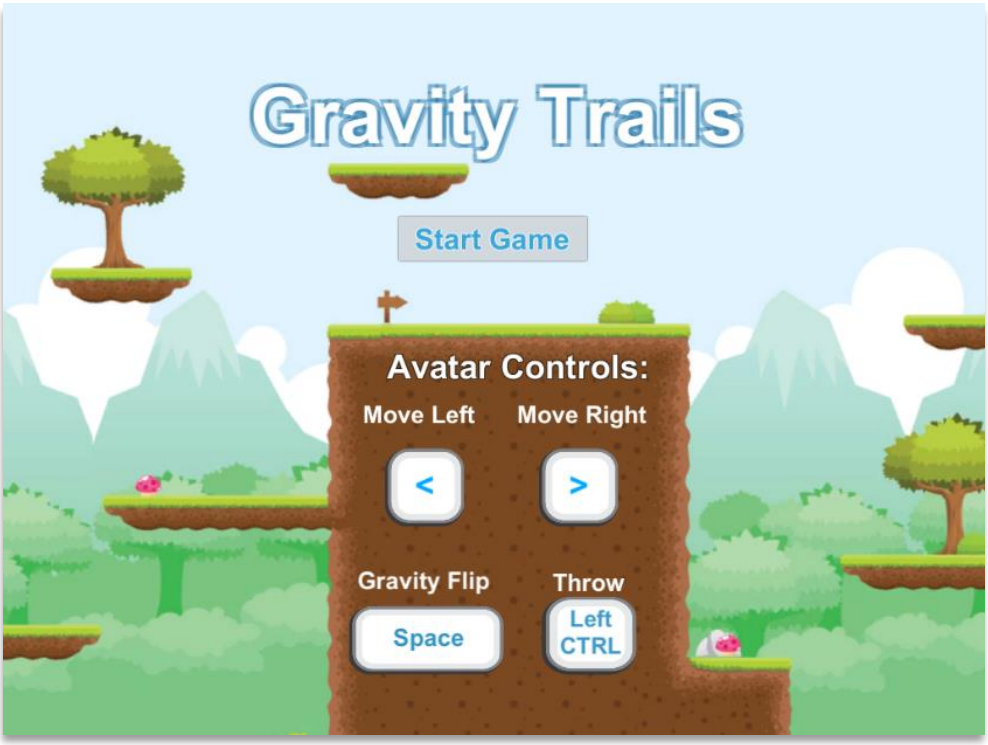
- What type of screen might appear at the beginning of your project and at checkpoints for the user? Where might you place credits?
- What "extras" might you add into your project?
- Is your project a reasonable length? Does it have frequent checkpoints for the user?
- When you playtest your project, does everything work as expected? What mechanics might need to be revised to improve the user experience?
- How might you use the feedback you received to make updates to your project?

In the Beta Phase, you will:

- Make final refinements to your project, such as sound effects, navigation screens, and game mechanic tweaks.
- Play your project from beginning to end to ensure it functions as expected without any bugs.
- Have at least 2 other ninjas and Code Senseis **playtest** your project!
- Revise your project based on playtest feedback received.
- Record Dev Diary #4 with a Code Sensei's help!

Beta Phase – Adding a Start Screen

A common practice is to create a start screen that gets the user excited about your project. A good start screen should contain your project's title, your name, and the controls. You should also include an image of your project that represents its theme.



Use the Creating a Start Screen section from Red Belt's Gravity Trails (p. 72-93, steps 121-167) to help you build a start screen for your Black Belt project.

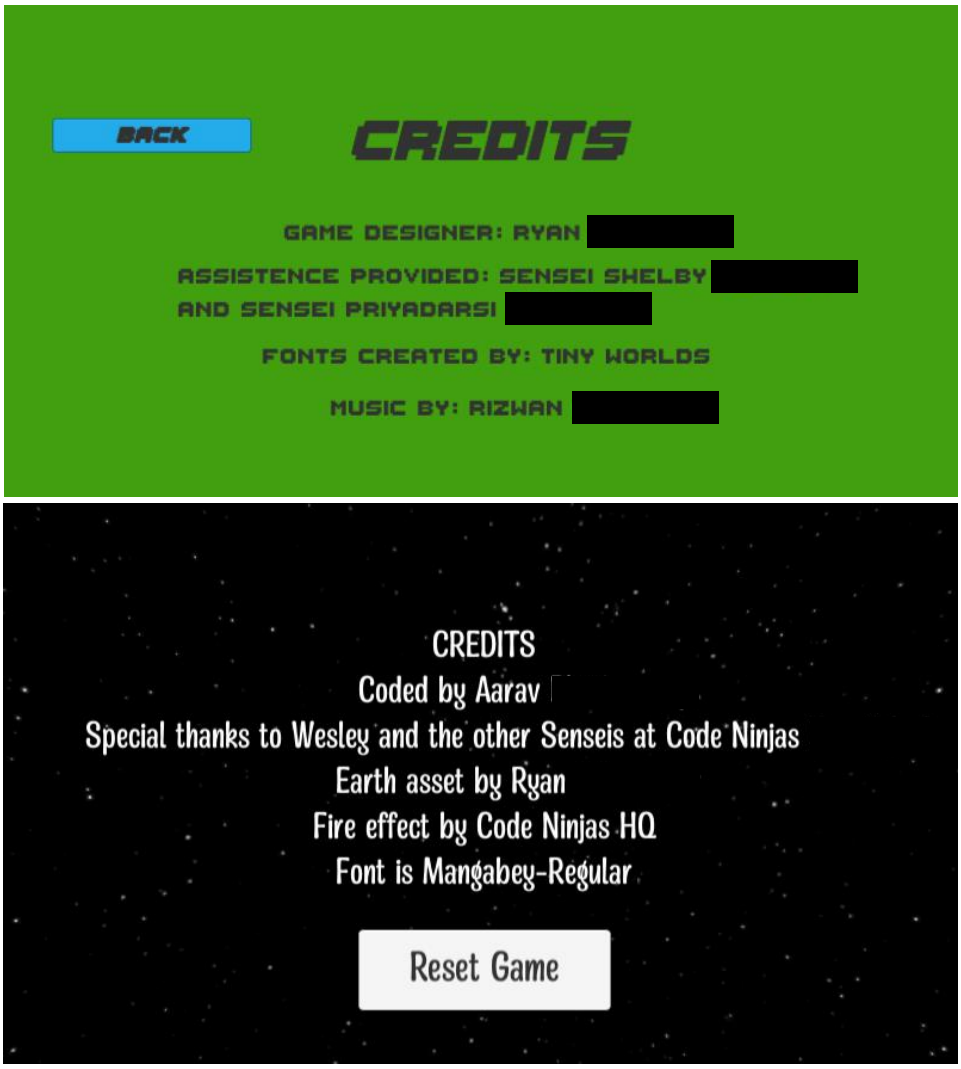
After you've created a start screen, think about what other screens you'll need to include in your project. What will appear when the user moves to a new level or different part of the project? What will appear at the end of the project?

Beta Phase – Credits Screen

Most projects are created by large teams of programmers, designers, and artists, and everyone's contribution should be credited. In addition to using assets from the Unity Asset Store, you will receive help from Code Senseis, fellow ninjas, and others as you create your project.

Adding credits will give you an opportunity to not only say "thanks" to those that helped you, but it will also let you give attribution to the creators of the assets you used.

You can place credits on your start screen or connect a new screen to the start screen or that appears at the end of the project.

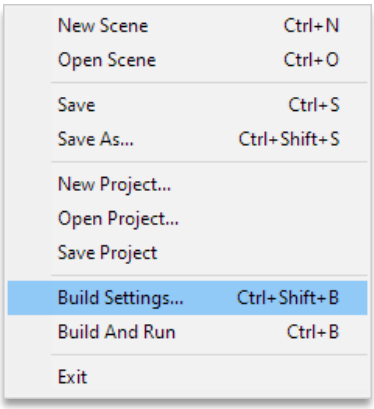


images used from Black Belt projects created by ninjas at Code Ninjas centers

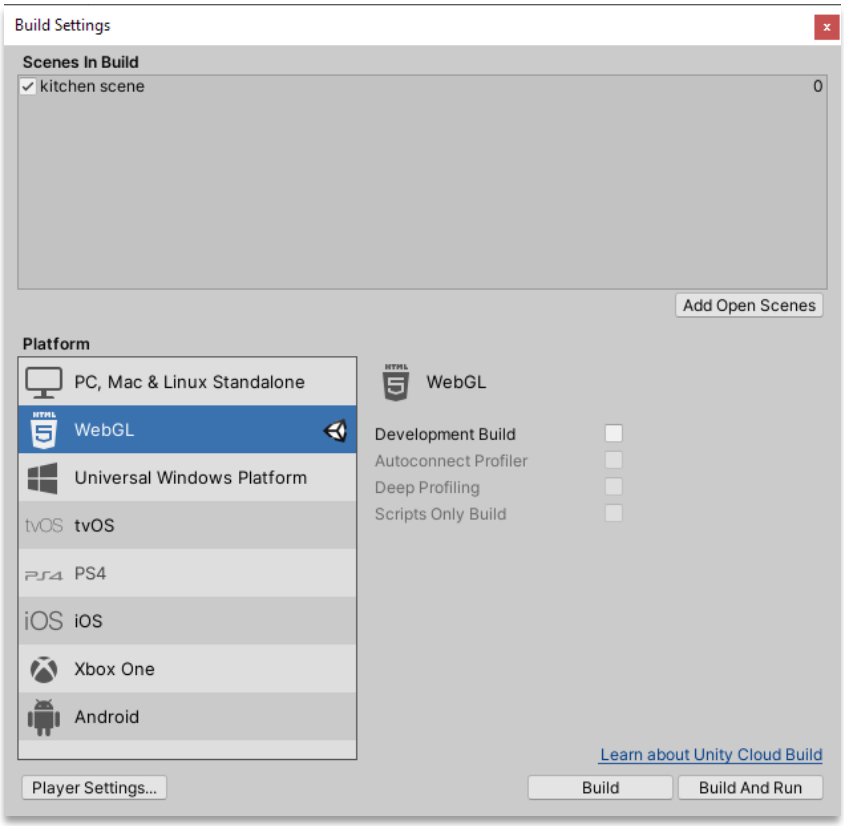
Beta Phase – Loading Screens

Unity lets you control various aspects of your project's loading screen. You can customize it by adding your own logo and adjusting the on-screen animations and colors.

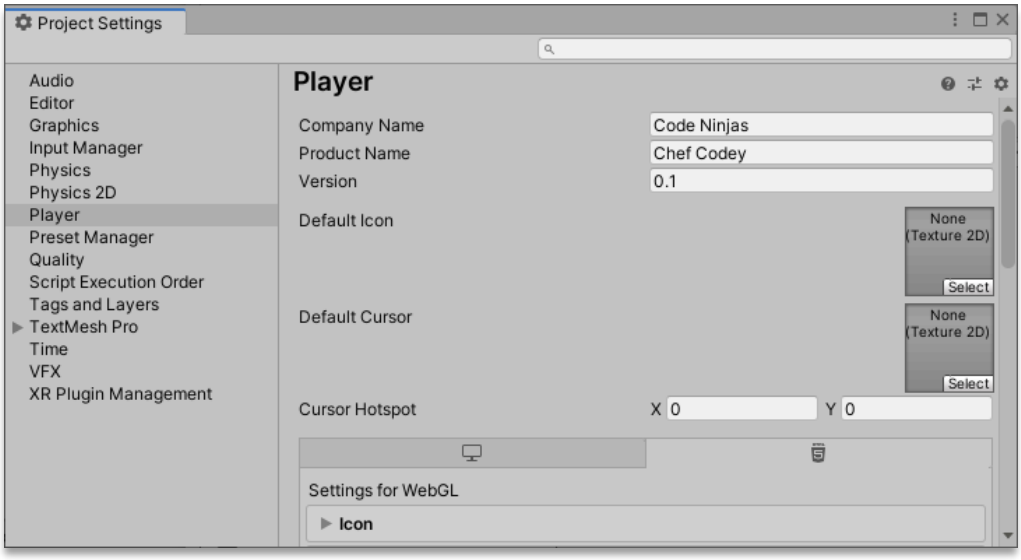
1 First, go to File and then Build Settings.



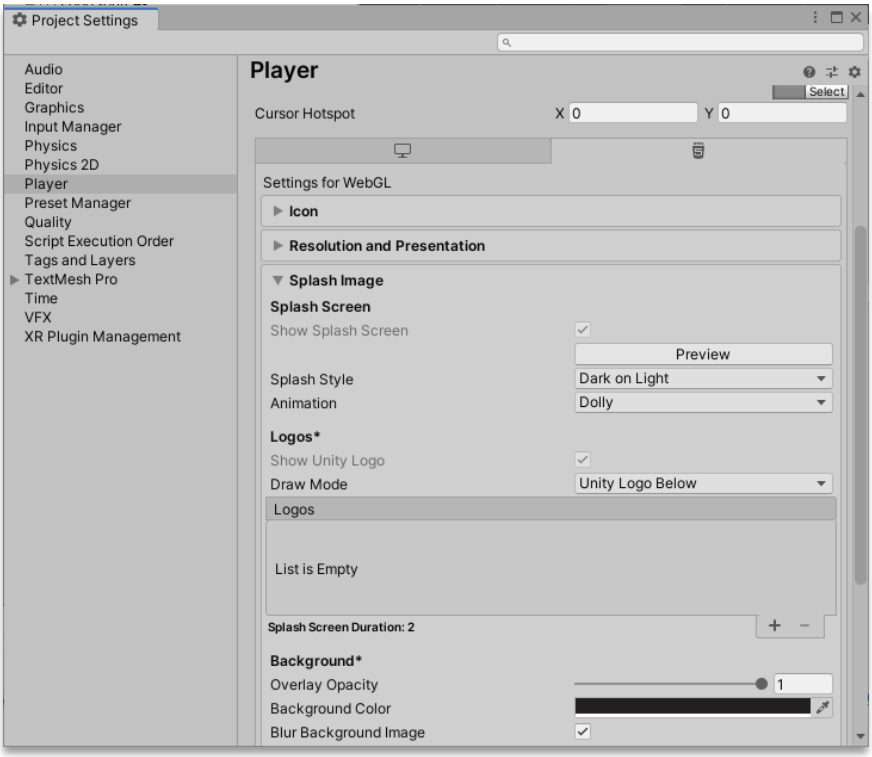
2 Click the Player Settings... button at the bottom.



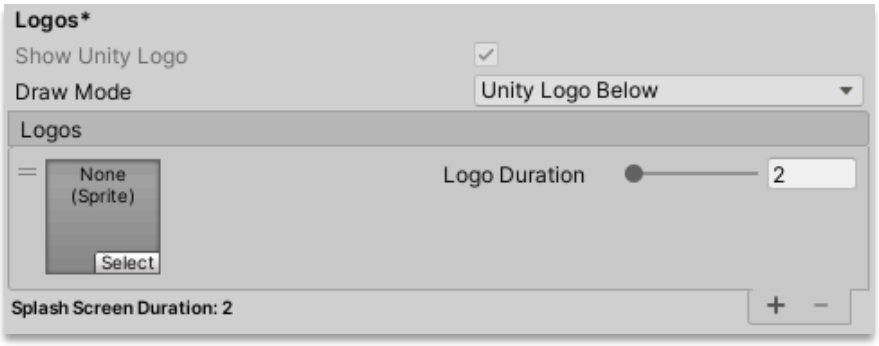
3 Select Player from the menu on the left.



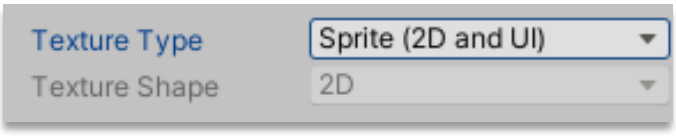
4 In the Settings for WebGL, find the Splash Image section.



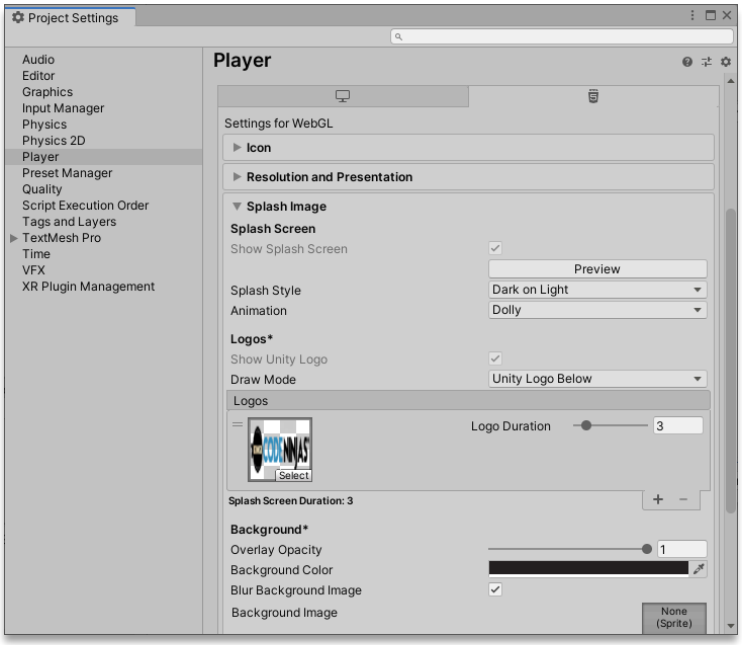
5 Under Logos, click the plus icon to add an image.



6 Click on the Select button on the None (Sprite) Icon to open the Select Sprite menu. You can select any image that is in your Project that has been given a Texture Type of Sprite.



7 Once you have the Sprite selected, you can adjust how long it appears on the loading screen by changing the Logo Duration.



8 You can click on the Preview button to see your logo in the Game window. When you build and host your project, your splash screen will be shown to the user every time the project is initially started.

Beta Phase - Playtesting

To complete the Beta Phase, your project should be playtested by at least 2 other ninjas and 2 Code Senseis. Using their feedback, update your project so that it is playable from beginning to end and is completely bug-free.

Playtesting Questions

- What did you like about my project?
- What could be improved in my project?

Then, create 2 of your own questions to ask your playtesters!

Ninja Planning Document

Complete the **Beta Phase – Playtesting** portion of your Black Belt Ninja Planning Document by asking at least 2 other ninjas and 2 Code Senseis to playtest and provide feedback on your project.

Finishing the Beta Phase

A Code Sensei will fill out the **Black Belt Project Checklist** to determine if your project is ready to advance to the Release Candidate Phase. If your project does not pass the evaluation, work with your Code Sensei to revise your project to fix any problems before moving forward.

As you wrap up the Beta Phase of your project, work with a Code Sensei to record a Dev Diary video, then review this checklist to demonstrate that you are ready to move on to the next phase of your Black Belt project!

Dev Diary #4 - Beta

Work with a Code Sensei to record a video in which you:

- Demo your project! Describe the new additions and changes you've made to your project during the Beta phase.
- Share examples of how you have used feedback to iterate on your project.

Beta Phase Checklist

- ✓ I added start, credits, loading, and other screens to my project.
- ✓ My project has sound effects, music, and other "extras".
- ✓ My project was playtested by at least 2 other ninjas.
- ✓ My project was playtested by at least 2 Code Senseis.
- ✓ I revised my project based on feedback from others.
- ✓ My project is bug-free and can be played from start to finish.
- ✓ I recorded Dev Diary #4 with my Code Sensei.

Release Candidate Phase

Once you have completed the Black Belt Ninja Checklist requirements, you are ready to submit your project as a Release Candidate to the Code Ninjas Education Team! Work with your Code Sensei to fill out and submit the Black Belt Project Release Candidate Submission Form to receive feedback from the Code Ninjas Education Team!

To complete the Release Candidate Submission Form, you will need:

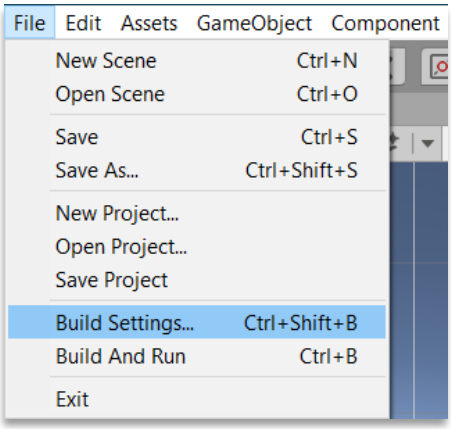
- A **WebGL link from GitHub**, to share your project.
- A link to your **Unity Package**, stored in Google Drive, Dropbox, or Sharepoint.
- A link to your **Ninja Planning Documents**, as a scanned PDF or photos stored in Google Drive, Dropbox, or Sharepoint.
- A link to an optional **Video Walkthrough**, which showcases you demo-ing your project for the Education Team.
- A link to a video or playlist of your **Dev Diary** recordings, which showcases your progress from each stage of the process.

In the Release Candidate Phase, you will:

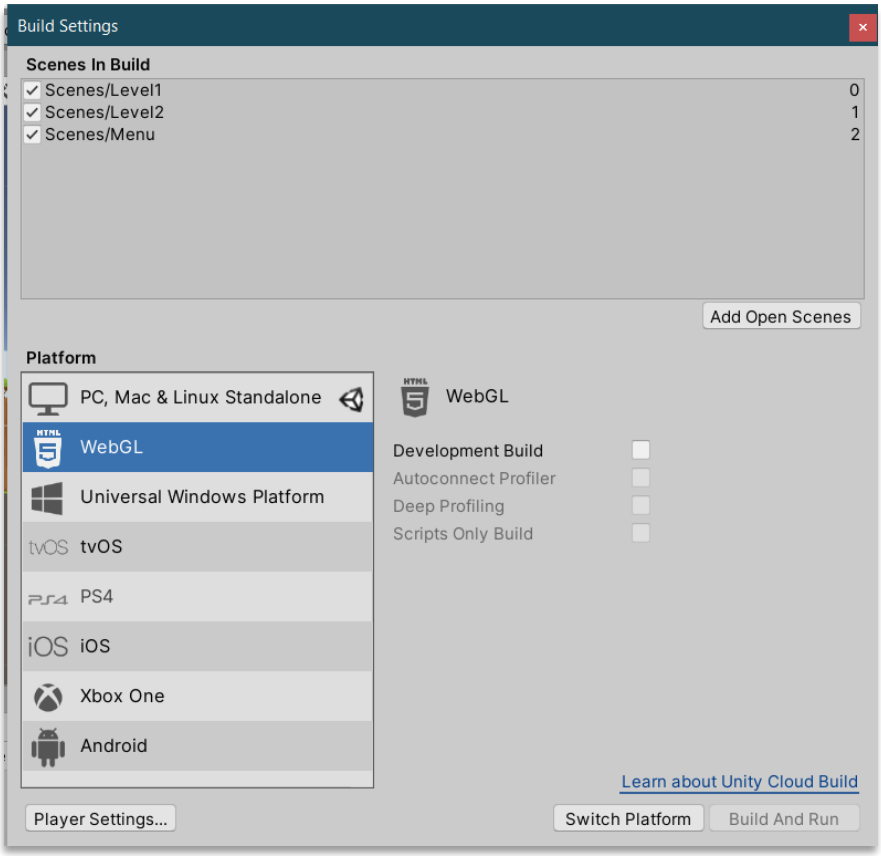
- Collect the required information and submit your project with a Code Sensei through the Release Candidate form.
- Review feedback from the Code Ninjas Education Team and make revisions to your project.
- Have a ninja and a Code Sensei **playtest** your project to confirm that revisions have been made.
- Resubmit your project for additional feedback.
- Record Dev Diary #5 with a Code Sensei's help!

Release Candidate Phase – Publishing

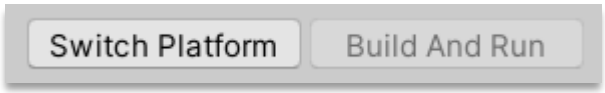
1 In Unity, click File, then Build Settings.



2 Make sure all scenes are checked. Under Platform, click WebGL.

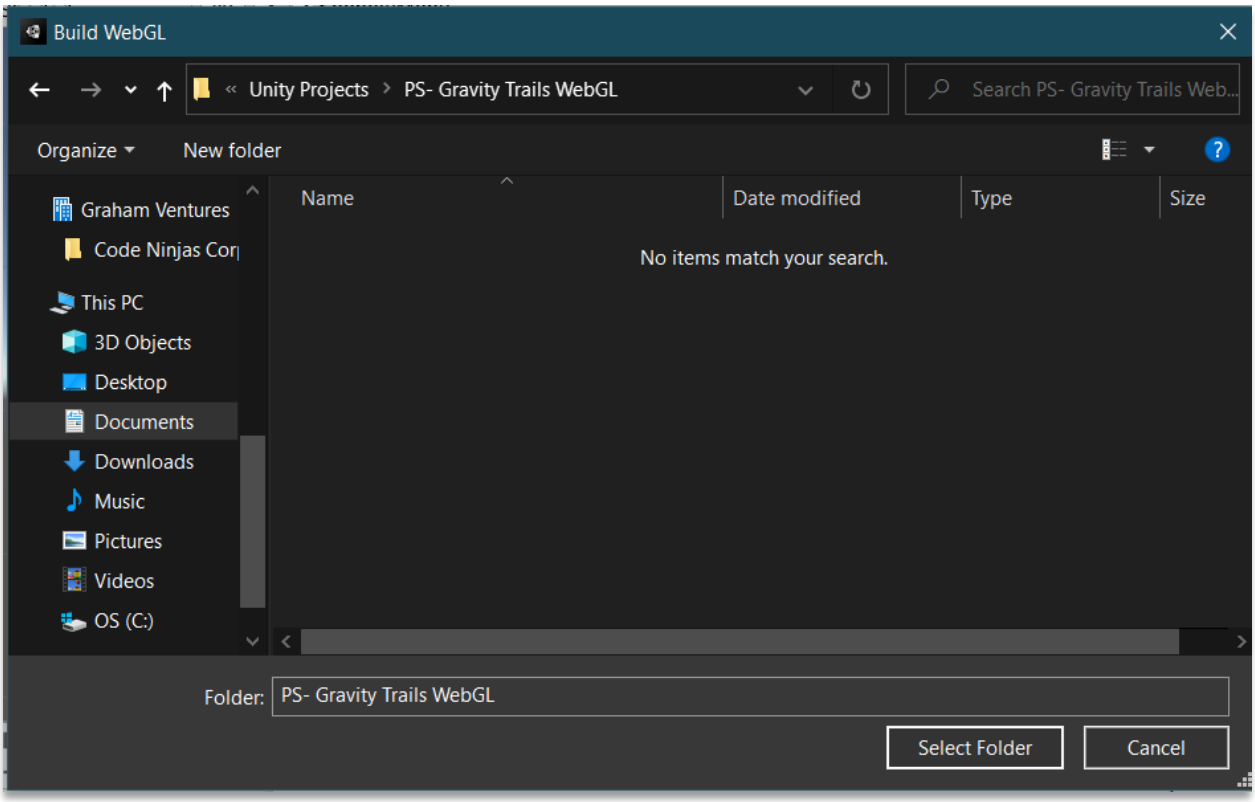


3 Click “Switch Platform.” This may take a few minutes.

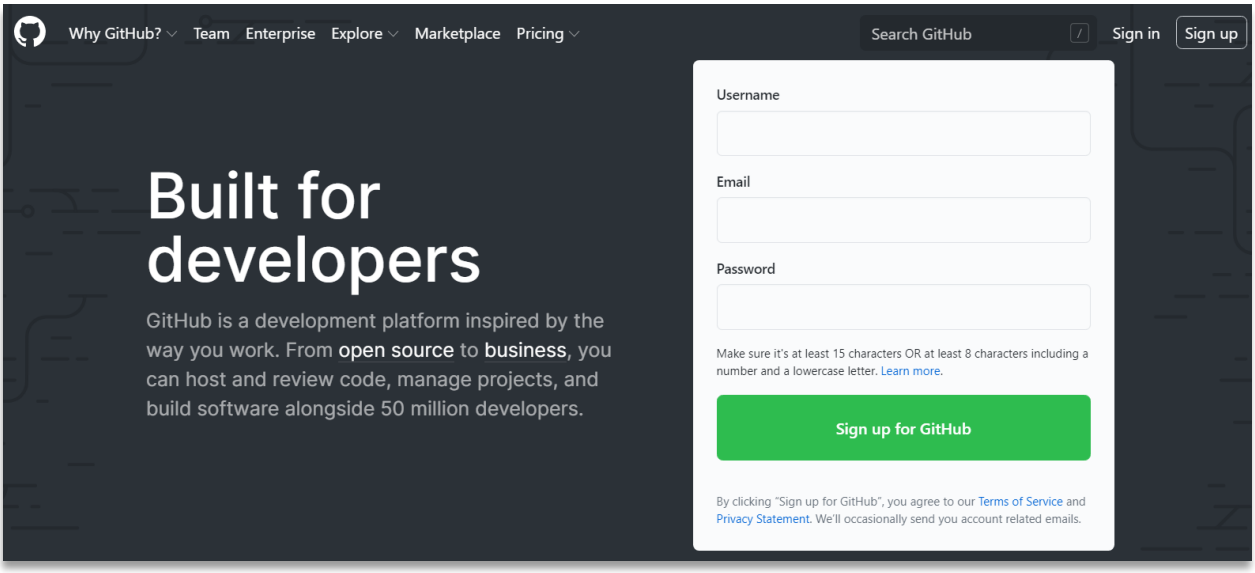


4 Click Build And Run.

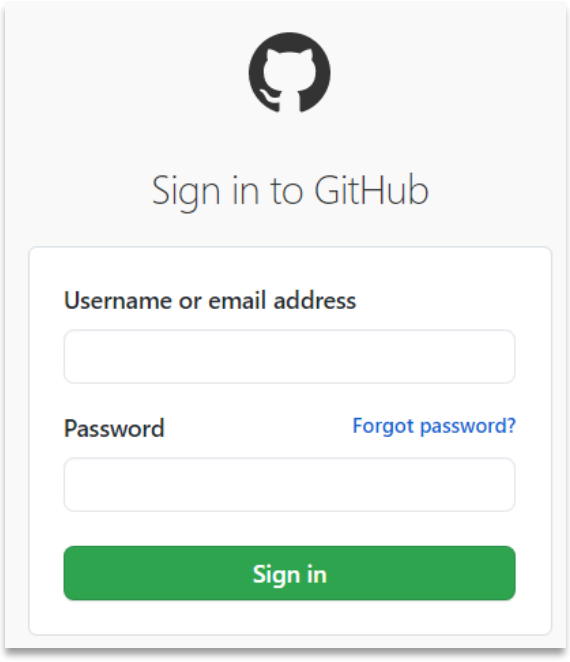
5 Create a folder and save the WebGL files in a location you will remember.



6 Go to GitHub.com and sign in. If you do not already have an account, create a free account.

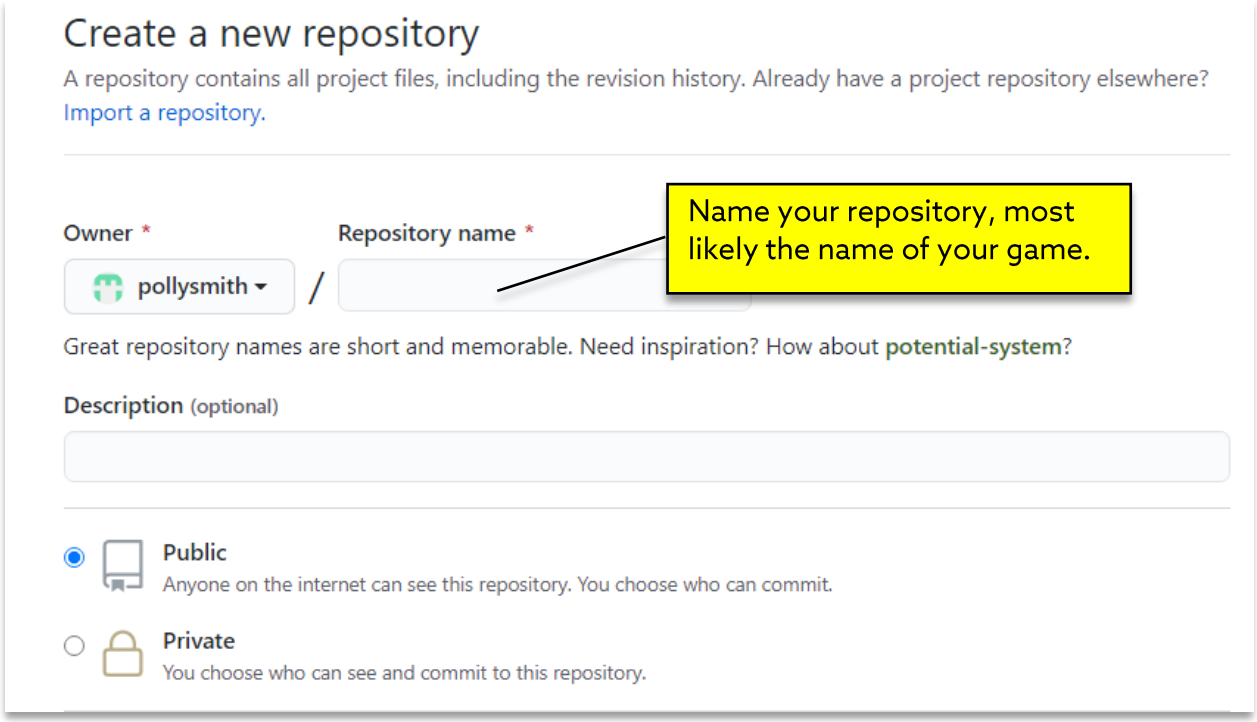


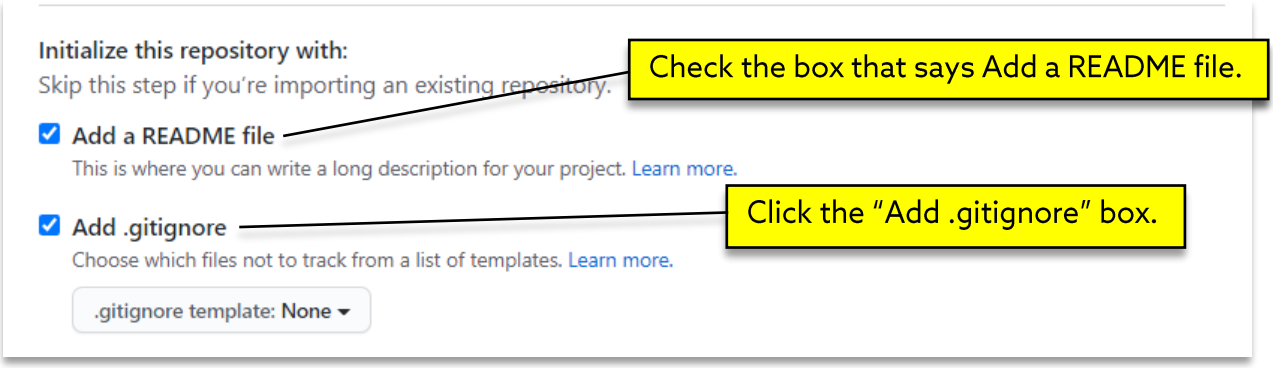
7 Sign into GitHub.



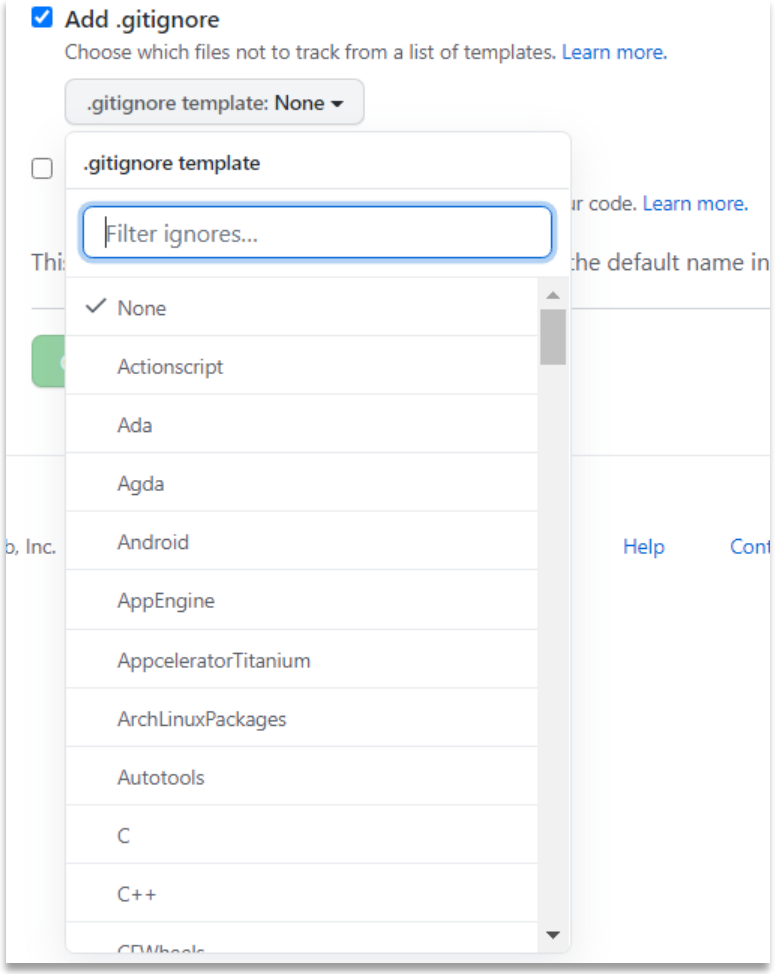
8 In your dashboard, click the "+ New Repository" button.

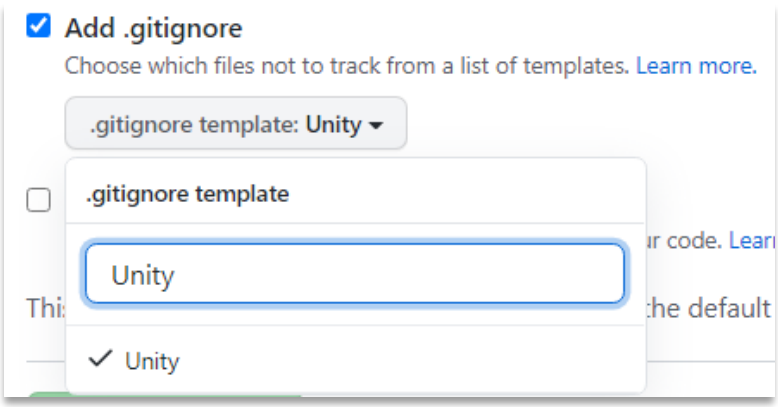
9 Create a new repository.



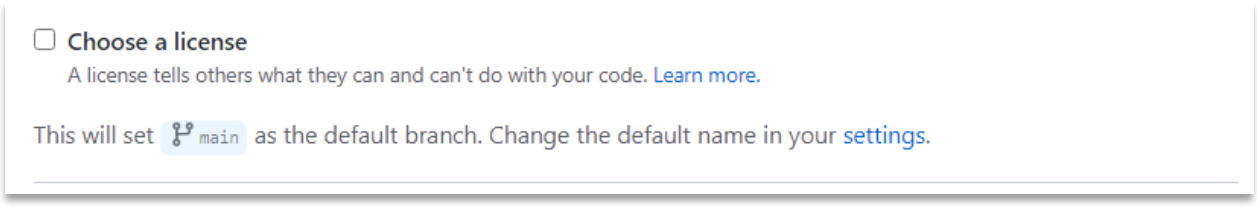


10 Click the .gitignore template: None dropdown menu, and type “Unity”.

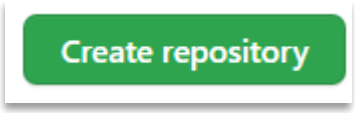


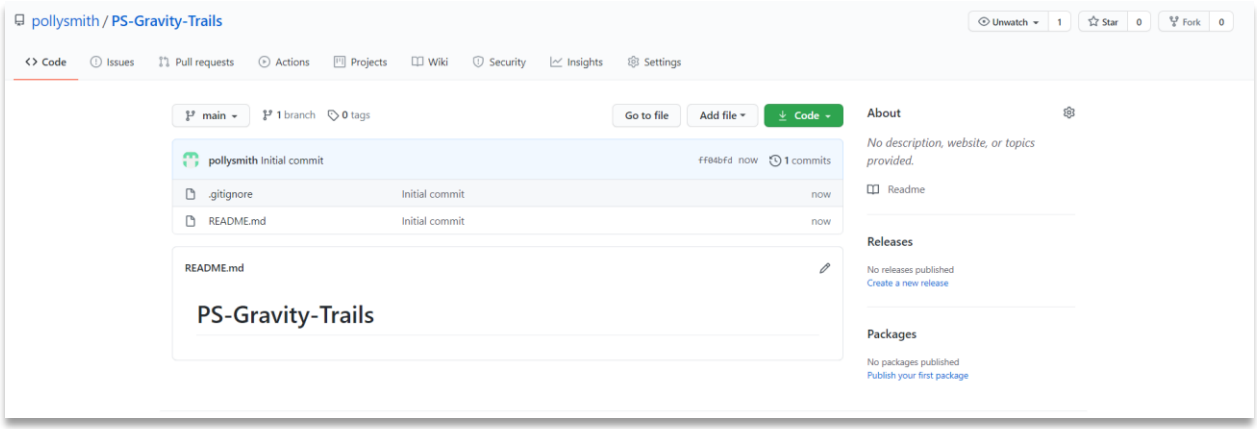


11 Leave the "Add a license" box alone unless you know what kind of license you want to use.

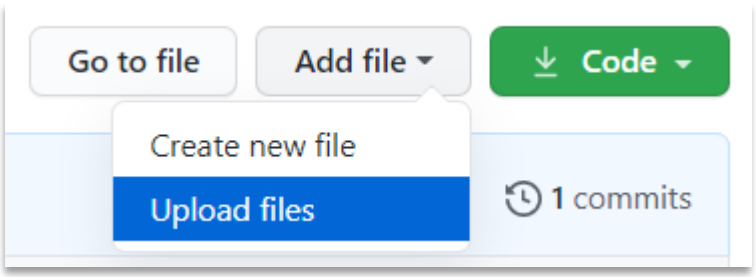


12 Click the Create repository button.

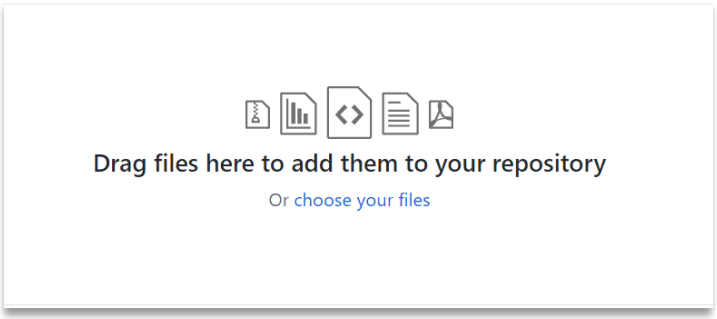




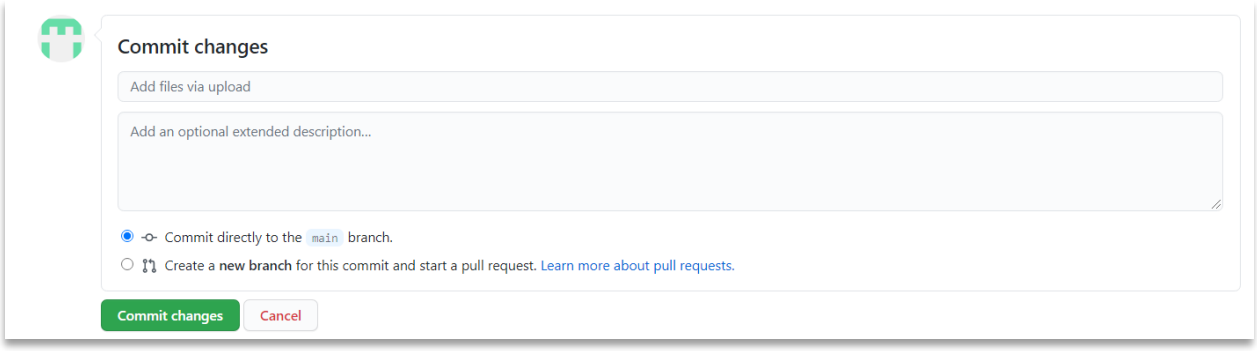
13 Click Add file, then Upload files.



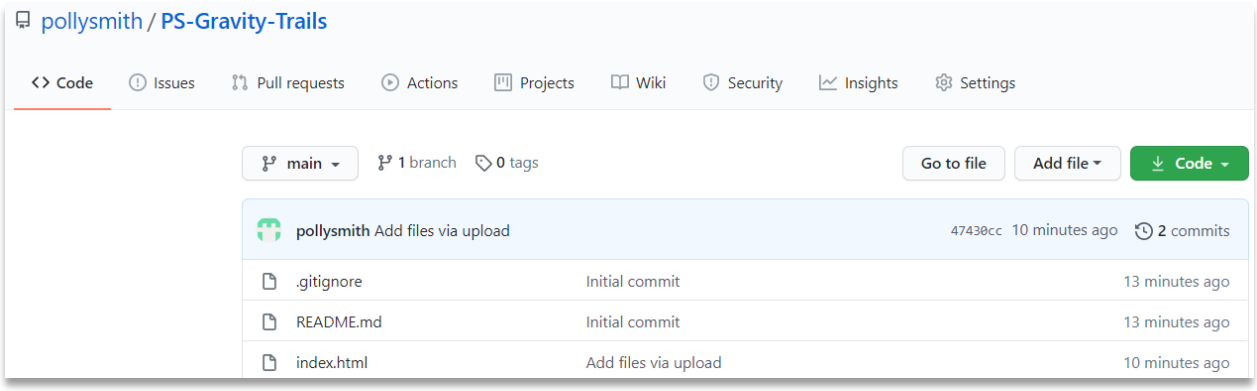
14 Click “choose your files,” then locate the folder where you saved your project’s WebGL files. Upload *index.html*, *Build/*, and *TemplateData/*



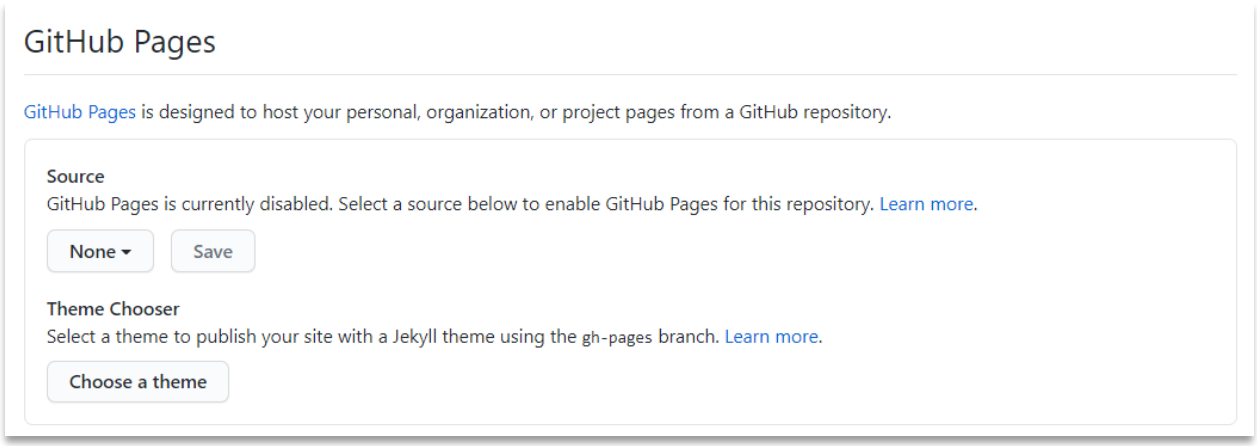
15 Click Commit changes.



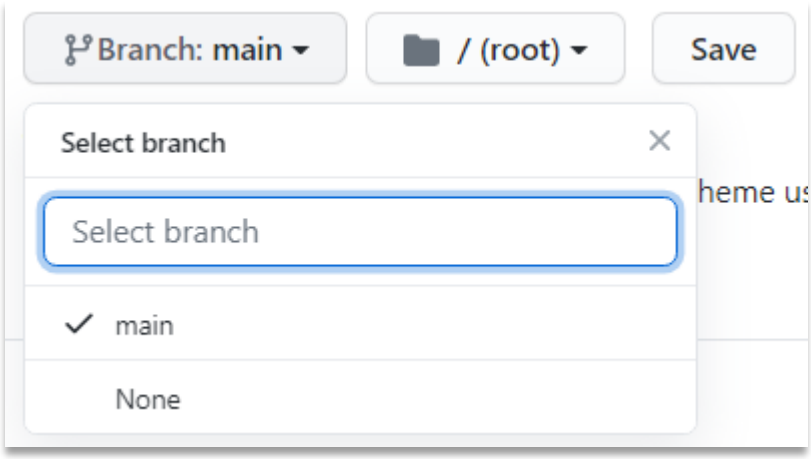
16 Once your files are uploaded, click on “Settings”.



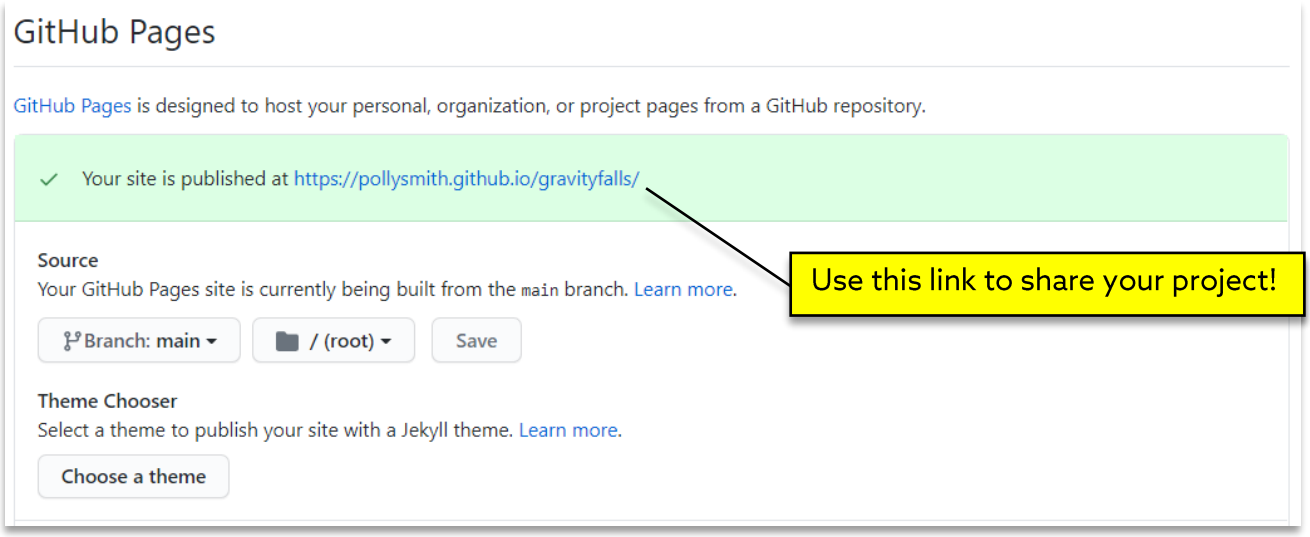
17 Scroll down to “GitHub Pages”.



18 Click the dropdown menu under "Source" and select "Main," then click "Save."



19 Wait while the page is being published, you may have to click refresh until the page is published.



20 Click on the hyperlink to open your project page. Test out your project to make sure all parts work and look the way you want.

21 You will use this GitHub URL when you and your Code Sensei submit your Black Belt project!

Release Candidate Phase – Revisions

Once you've received feedback from the Code Ninjas Education Team, work with your Code Sensei to make any changes based on their recommendations.

When you've made updates to your project, have your project playtested by at least 1 other ninja and 1 Code Sensei to ensure your project is playable from beginning to end and is completely bug-free.

Playtesting Questions

- What did you like about my project?
- What could be improved in my project?

Then, create 2 of your own questions to ask your playtesters!

Ninja Planning Document

Complete the **Release Candidate Phase – Playtesting** portion of your Black Belt Ninja Planning Document by asking at least 1 other ninja and your Code Sensei to playtest and provide feedback on your project.

Once your project has been updated and playtested, work with your Code Sensei to resubmit your project using the Black Belt Project Release Candidate Submission Form.

Continue to revise and resubmit your project until it is approved by the Code Ninjas Education Team.

Finishing the Release Candidate Phase

After your project has been approved by the Code Ninjas Education Team, work with a Code Sensei to record a Dev Diary video, then review this checklist to demonstrate that you are ready to move on to the next phase of your Black Belt project!

Dev Diary #5 – Release Candidate

Work with a Code Sensei to record a video in which you:

- Discuss the feedback you received from the Code Ninjas Education Team and share examples of how you revised your project based on their feedback.
- Give examples of how you have used feedback to iterate on your project.
- Share how you plan to celebrate your big achievement!

Release Candidate Phase Checklist

- ✓ I submitted my project for feedback through the Black Belt Project Release Candidate Submission form.
- ✓ I revised my project based on feedback from the Code Ninjas Education Team.
- ✓ I revised and resubmitted my project until it was approved by the Code Ninjas Education Team.
- ✓ My project is bug-free.
- ✓ I recorded Dev Diary #5 with my Code Sensei.

Going Gold Phase

Congratulations on making it to the Going Gold Phase! You should feel very proud of all your accomplishments and all the progress you have made up to this point, and are hopefully looking forward to sharing your finished project with others!

Once your project is accepted by the Education Team, it has "gone gold" and is ready to be featured on the Black Belt Ninjas site. In order to do this, you'll need to work with your Code Sensei to submit the Black Belt Project Going Gold Form to the Code Ninjas Education Team.

To complete the Going Gold Submission Form, you will need:

- A short **description** of yourself, your project, and your process for developing it.
- A **photo** of yourself, to be displayed on the Black Belt Ninjas site.
- An optional **video** that describes your project and your process, from idea to completion. This can be created from a compliation of your Dev Diaries, or can be made separately.
- An action **screenshot** of your project, to be displayed on the Black Belt Ninjas site.
- An updated **WebGL link from GitHub**, to share your project.

Ninja Planning Document

Complete the **Going Gold Phase – Publishing** portion of your Black Belt Ninja Planning Document by drafting information to be posted alongside your project on the Black Belt Ninjas site.

After compiling everything you need, work with your Code Sensei to submit your project using the Black Belt Project Going Gold Form.

Congratulations on earning your Black Belt!

Unity Concepts

Update

The Update function runs every frame and makes objects in your game move and change. Unity renders frames of your game as fast as it can, meaning that the Update function will run at different rates based on a computer's speed and the scene's complexity. Because you cannot predict exactly how often the Update function will run, it is helpful to use Booleans and Time.deltaTime to control what happens inside the function.

```
void Update()
{
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");
    Vector3 movement = new Vector3(horizontal, 0, vertical);
    transform.position += movement * Time.deltaTime * speed;
}
```

Instructions

1 The Update function is automatically created by Unity in each new script.

```
// Update is called once per frame
0 references
void Update()
{

}
```

2 If you need to access a different game object or component, you should use GetComponent or FindGameObjectWithTag in the Start function and store a reference in a variable. You can then use that variable in your Update function.

```
0 references
void Start()
{
    obstacle = GameObject.FindGameObjectWithTag("obstacle");
    agent = GetComponent<NavMeshAgent>();
}
0 references
void Update()
{
    agent.destination = obstacle.transform.position;
}
```

In this example, we find the obstacle game object and the NavMeshAgent component in the Start function because these two functions take resources to run and their results won't change if we call them again.

In the Update function, we set the agent's destination to the obstacle's changing position each frame.

3 Conditional statements with simple Booleans can help you determine if you want to run a specific section of code on the current frame.

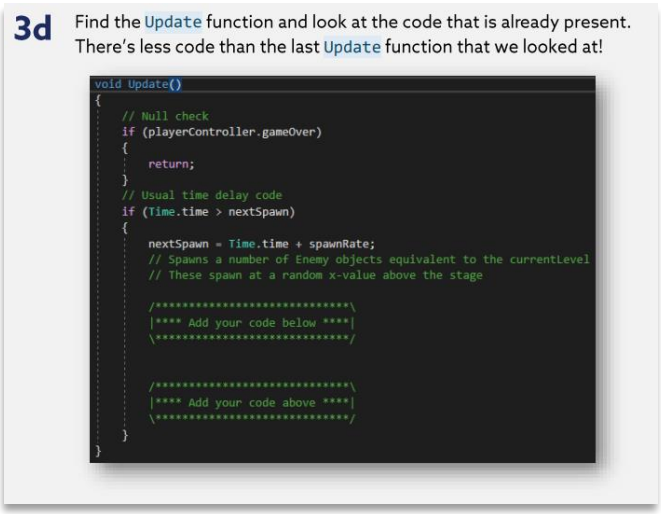
```
0 references
void Update()
{
    if (canMove)
    {
        float vertical = Input.GetAxis("Vertical");
        Vector3 move = Vector3.forward * Speed * Time.deltaTime * vertical;
        transform.Translate(move);
    }
}
```

In this example, we use a canMove Boolean that is calculated outside the Update function. If canMove is false, the code inside does not run. If canMove is true, then Unity polls for Player Input, calculates a movement Vector, and applies a Translate to the transform.

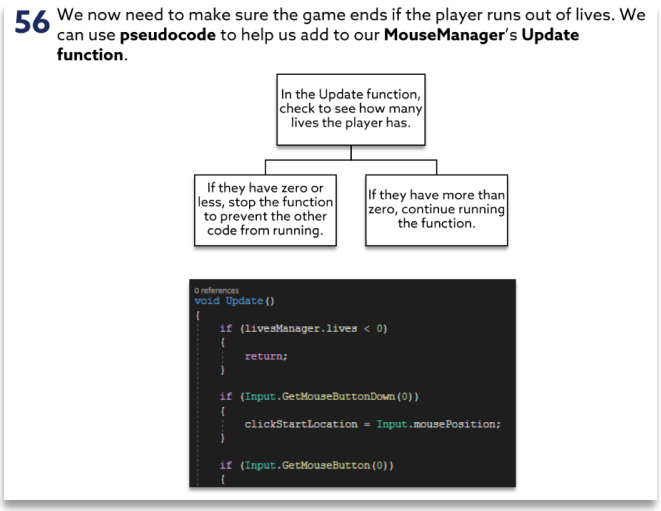
4 Since the Update for each object is run each frame, it is important to make sure your Update code stays relatively simple. If you have 100 enemies that each perform complex tasks in their Update functions, your game will slow down a lot!

Examples

In Shape Jam, we used the Update function to determine when to spawn new enemies. It has several conditional statements and uses Time.time to calculate when new enemies should be Instantiated.



In Sulky Slimes, we used the Update function to check to see if the player had lives remaining. If so, then the function would check for user Input.



Ideas

The Update function is an essential part of game development in Unity. Unity uses this function to modify the game objects inside the scene. You can use it to see how much time has passed or listen for user Input.

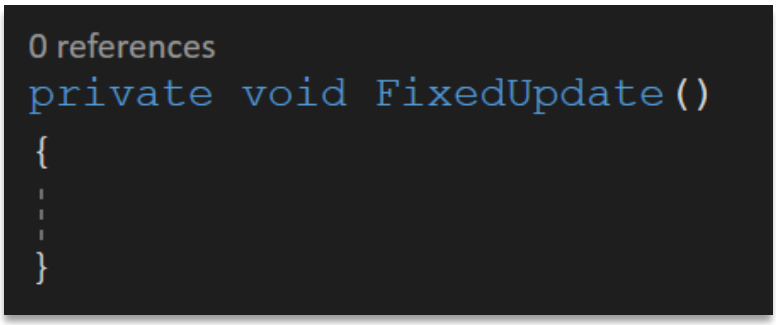
Fixed Update

While the Update function is tied to the game's frame rate, FixedUpdate runs every 0.02 seconds, or 50 times a second. Because it is called at a regular interval, this is the best place for physics calculations. If calculating the effects of gravity happened too fast or too slow, then your game's physics wouldn't feel right. Code within FixedUpdate runs at the exact same rate no matter what.

```
0 references
private void FixedUpdate()
{
    Vector3 propulsion = new Vector3(10, 0, 0);
    myRigidBody.AddForce(propulsion);
}
```

Instructions

5 While the Update function is created automatically in each new script, you must create your own FixedUpdate function.



6 While FixedUpdate and Update behave similarly, they serve different purposes within your code. Things like listening for Player Input should go in the Update function because it is unpredictable when a user might press a button.

On the other hand, calculating the effects of gravity or other physics should go in FixedUpdate because the game objects need to be updated at consistent, or fixed, time intervals.

Note: If you use Rigidbody Components and the Unity Physics Engine, you might not need to rely on FixedUpdate to perform many calculations.

Examples

In Robomania, we used FixedUpdate to apply a force to the enemies.

44b We now need to apply a force to the **Crusher's rigidbody**. Instead of creating our own vector, we can use **Unity's "left" vector** to get a **vector** that has a -1 in the **x coordinate** and a 0 in the **y coordinate**. We can then **multiply** the (-1, 0) **left vector** by the **Crusher's xForce value**. This will bounce the **Crusher** to the left at the same force it **jumps** with.

After `xDirection = -1;` type
`enemyRigidBody.AddForce(Vector2.left * xForce);`
to apply a **force**.

```
private void FixedUpdate()
{
    if (transform.position.x <= -8)
    {
        xDirection = 1;
        enemyRigidBody.AddForce(Vector2.right * xForce);
    }
    if (transform.position.x >= 8)
    {
        xDirection = -1;
        enemyRigidBody.AddForce(Vector2.left * xForce);
    }
}
```

Ideas

You can use FixedUpdate to modify the strength of gravity on specific objects. You could also add a propulsion force to an object to have it accelerate like a space ship.

Time

One advantage video games have over real life is that you can control time! Unity provides useful information and properties about Time in your scene.

```
public Text timerLabel;
0 references
public void UpdateTimer()
{
    timerLabel.text = Time.time.ToString();
}
```

Instructions

1 The Time property is often used to make sure things like speed are scaled properly. Without it, the speed of game objects would be based on how often the Update function runs. By multiplying the speed of an object by Time.deltaTime, you are accounting for how much time has passed since the last time the Update function has run.

```
void Update()
{
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");
    Vector3 direction = new Vector3(horizontal, 0, vertical);
    transform.position += direction * speed * Time.deltaTime;
}
```

2 If you use FixedUpdate instead of the standard Update function, use Time.fixedDeltaTime instead.

```
void FixedUpdate()
{
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");
    Vector3 direction = new Vector3(horizontal, 0, vertical);
    transform.position += direction * speed * Time.fixedDeltaTime;
}
```

3 You can calculate how much time in seconds has passed since the start of the game with Time.time.

```
public Text timerLabel;
0 references
public void UpdateTimeSinceGameStart()
{
    timerLabel.text = Time.time.ToString();
}
```

4 If you want to know how much time in seconds has passed since the scene has loaded, then use Time.timeSinceLevelLoad. This is useful if your game has different levels or scenes.

```
public Text timerLabel;
0 references
public void UpdateTimeSinceSceneStart()
{
    timerLabel.text = Time.time.ToString();
}
```

5 You can control the speed of time in your game by adjusting Time.timeScale. If the value is 1.0f, then time passes as normal. If the value is 0.0f, then time stops completely and all time-dependent objects in your game pause. If the value is between 0.0 and 1.0, then time will slow down.

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Time.timeScale = 0.5f;
    }
    if (Input.GetMouseButtonUp(0))
    {
        Time.timeScale = 1.0f;
    }
}
```

In this example, when the user presses the left mouse button down, the game runs at half speed to create a slow-motion effect. When the user releases the left mouse button, the game returns to its normal speed.

Examples

In Codey Raceway, we used Time.deltaTime to calculate two different timers, a Lap Timer and a Countdown.

111 Having the decimals show up on the screen does not look so good. We can fix that by using the **Mathf.Round** to only display whole numbers. Stop your game and open the **TimersCountdown** script.

Modify your Update function to round both the **totalCountdownTime** and **totalLapTime** variables before converting them to strings.

```
void Update()
{
    totalLapTime -= Time.deltaTime;
    totalCountdownTime -= Time.deltaTime;

    lapTime.text = Mathf.Round(totalLapTime).ToString();
    startCountdown.text = Mathf.Round(totalCountdownTime).ToString();
}
```

In Find the Exit, we used Time.deltaTime to make sure Codey moved at a consistent rate no matter how fast or slow the game was running.

7c In the `Update` function, we want to add one line of code to move our ninja, Codey.

To do this, we will use Unity's `Translate` function on Codey's `transform`. An object's **transform** contains information about its current **position**, **rotation**, and **scale**. Unity's `Translate` function will let us change an object's position. In the `void Update()` function, type `transform.Translate(Vector3.forward * speed * Time.deltaTime);` to tell Codey to move forward at our set speed of 5 at a constant rate of time.

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

Ideas

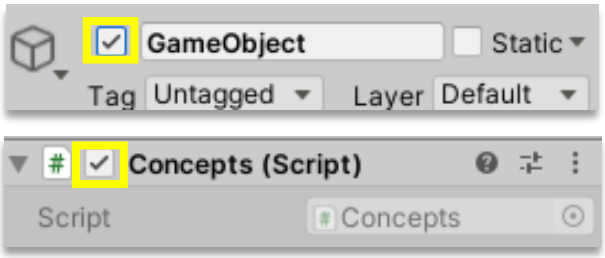
Use Time.deltaTime anywhere you directly move a game object's transform in an Update function. You can use Time.time or Time.timeSinceLevelLoad to create timers that add pressure to the player in your game. Change Time.timeScale to create a cinematic time-slow effect to highlight exciting action like an enemy being defeated.

Start

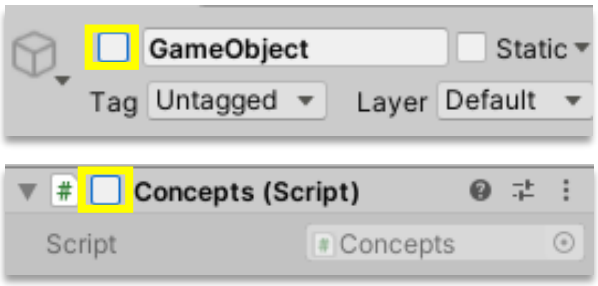
The Start function runs once as soon as your game starts, before any Update functions. As long as both the game object and the script are enabled, your code will execute when you press Play. If you use the same C# script multiple times, the Start function will also run every time there is a new instance of it being used.

```
void Start()
{
    // The code in this function
    // will run first.
}
```

Enabled Game Object and Script: Start function will run.



Disabled Game Object or Script: Start function will not run.



Instructions

1 When you create a script in Unity and open it, the Start function is already made for you.

```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{

}
```

If you deleted the Start function, you can easily recreate it.

2 Since the Start function does not return a value, it must have a return type of void.

```
void Start()
{

}
```

3 We can gather information from other components or scripts on the object using GetComponent, but first we need a variable to store the values.

Create a variable with the type that you want to get outside of your Start function. In this example we created a variable with a Rigidbody type because we want to grab the Rigidbody from our object.

```
public Rigidbody playerRigidbody;
Unity Message | 0 references
void Start()
{

}
```

4 In the Start function, set playerRigidbody to the object's component using the GetComponent function.

```
void Start()
{
    playerRigidbody = GetComponent<>
}
```

5 Type the name of the component inside the <>. In this example, we want a type of Rigidbody. Since this is a function, it needs to end with open and closing parentheses.

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

6 The Start function can do more than get a reference to a component on the object. It can also be used to find other objects or initialize variables.

Examples

You can find the Start function in almost every game. In Gravity trails, we used it to access the Throwable script attached to Codey.

76 Since we are modifying the prefab object, we cannot use the Inspector to attach game objects from the scene.

As soon as the throwable is created, we want to be able to access the variables in the Throwable script. In the **Start()** function, set the value of direction to the Player object's Throwable component.

```
void Start()
{
    direction = GameObject.FindGameObjectWithTag("Player").GetComponent<Throwable>();
}
```

It was also used to determine how many enemies were in the scene.

106 In the Teleport script's **Start()** function we want to use the **FindGameObjectsWithTag** function and **.Length** to get the total number of enemies.

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

Ideas

The Start function is a great way to ensure that your game functions as expected once you hit Play. You can set the value of your variables here so that they are always what you expect them to be.

This function can also be used to have game objects perform tasks in the scene as soon as the game is started.

Awake

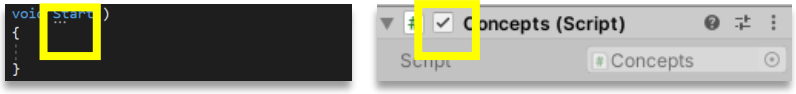
The Awake function is very similar to the Start function, but with some slight differences. Awake is the first function that runs when the game object is created. It runs before both the Start and the Update functions. As long as your game object is enabled, Awake will run even if the script is disabled.

```
0 references
void Awake()
{

}
```

Enabled Game Object and Script: Awake function will run.

Enabled Game Object and Disabled Script: Awake function will run.

Disabled Game Object and Enabled Script: Awake function will NOT run.



Instructions

1 The Awake function is not automatically added into the script. Since the Awake function does not return a value, you must give it a return type of void.

```
0 references
void Awake()
{

}
```

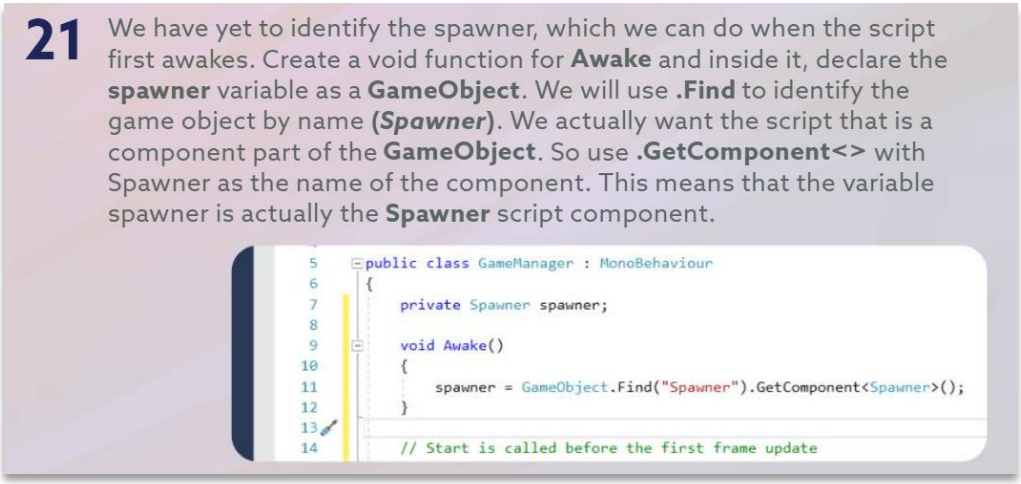
By default, Awake is a private function, but you can also declare it as a public function if you want.

2 You can access game objects in the Awake function just like in the Start function. You need to create a variable with the type you want to get and give it a name.

```
private GameObject finalBoss;
0 references
void Awake()
{
    finalBoss = GameObject.FindGameObjectWithTag("Boss");
}
```

Examples

In step 21 of the Dropping Bombs activity, you used the Awake function so that you could access the Spawner script and all of the code within that script.

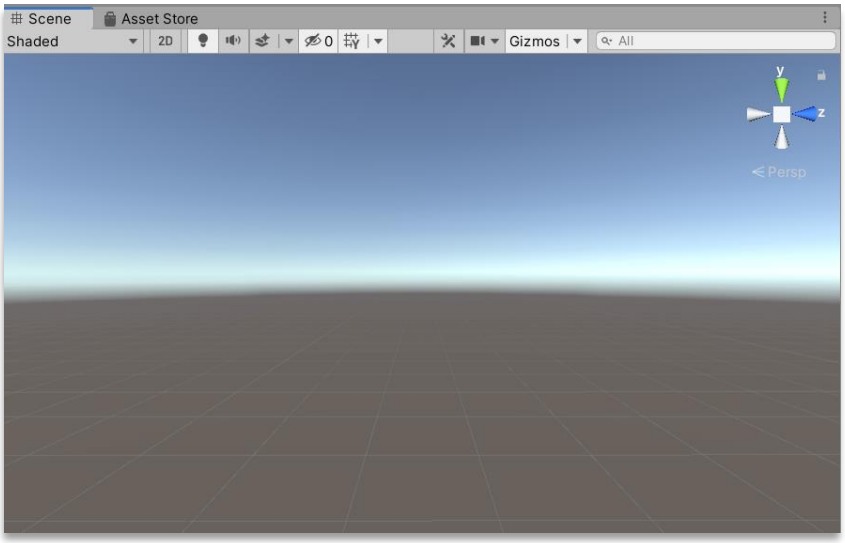


Ideas

You can use the Awake function to look for certain game objects before anything else happens in your game. You can also use Awake to enable certain scripts in a specific order.

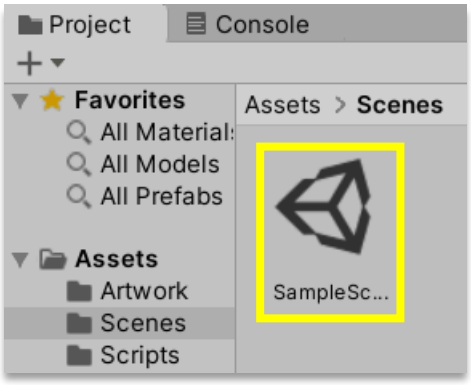
Scenes

All Unity games are made up of Scenes. A Scene contains all of the game objects, UI, lighting, cameras, and scripts you use to design your game.

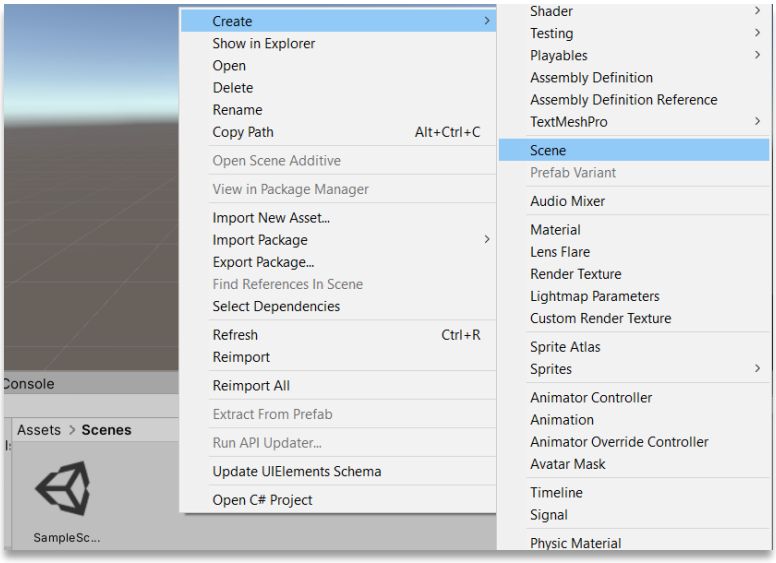


Instructions

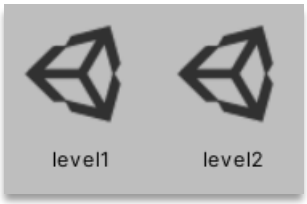
1 Each new Unity project starts with a Sample Scene in the Scenes folder.



2 To create a new scene, right click in the Scenes folder, select Create, and then Scene.



As your game grows in size, you want to make sure your Scenes have recognizable names.



3 Unity's SceneManager object lets you load scenes in your scripts. You must include "using UnityEngine.SceneManagement" at the very top of your code to use these Scene functions.

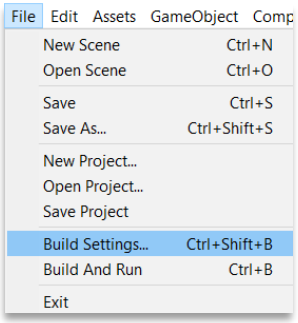
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
```

4 You can load a new scene by name using the SceneManager.LoadScene function. It takes one parameter: a string with the Scene's name.

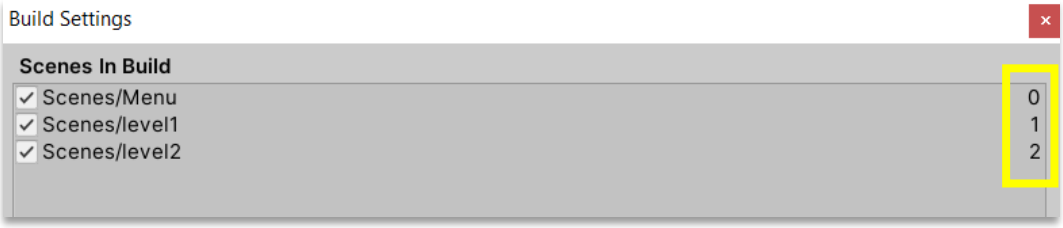
```
0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    SceneManager.LoadScene("Level2");
}
```

5 You can also use the Scene's Index value in the Build Settings menu instead of its name.

Click on File and select Build Settings.



6 Once added to the build, you can see the index of each of your Scenes.



7 You can load a Scene by using its index as the parameter of the SceneManager.LoadScene function.

```
0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    SceneManager.LoadScene(2);
}
```

Examples

In Purple Belt's Stacks, you loaded a Scene by name when the player did not match the pieces.

21 Create a new **IEnumerator X** function and add the following:

```
//IEnumerator X function
IEnumerator X()
{
    //Wait three second then code is
    //executed
    yield return new WaitForSeconds(3f);
    //The scene sample scene is loaded
    SceneManager.LoadScene("SampleScene");
}
```

In Red Belt's Gravity Trails, you used the scene index to load the second level after all the enemies had been removed and the Player character collided with the portal.

111 Back in the conditional statement, use the SceneManager's **LoadScene** function to load the scene that is in build index 1.

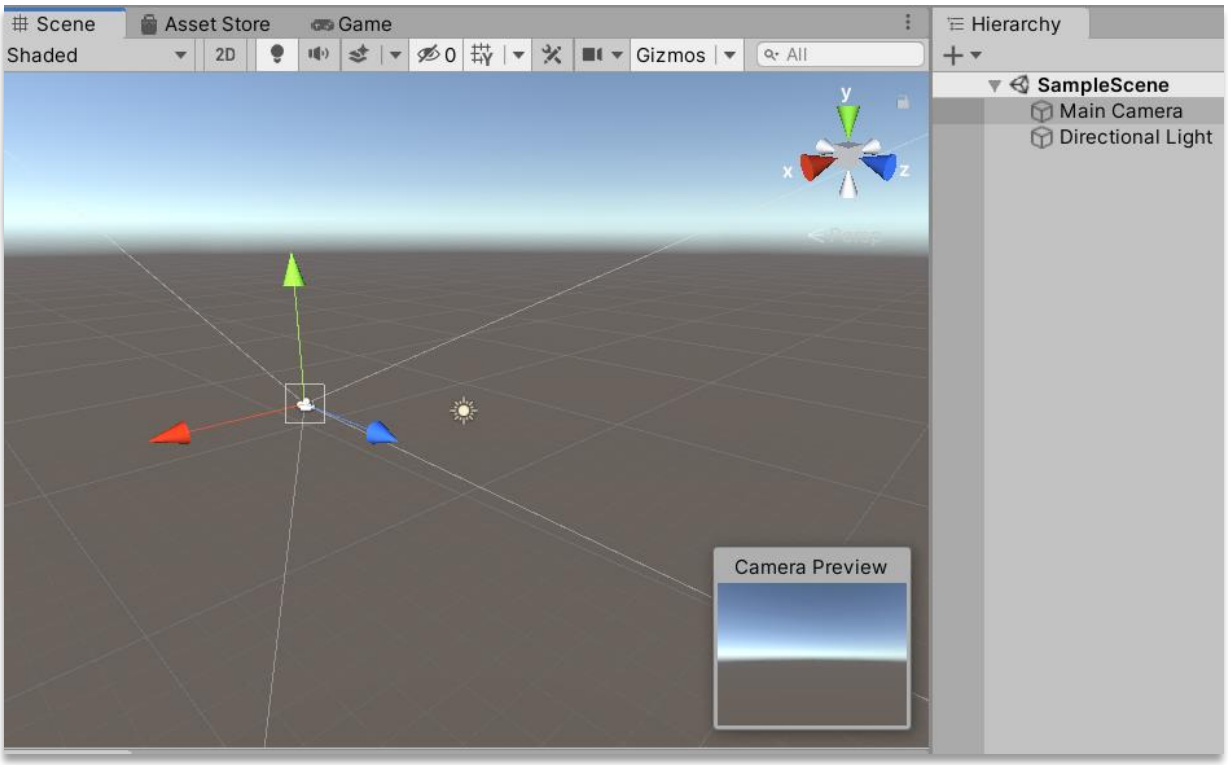

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.gameObject.tag == "Player" && enemyCount == 0)
    {
        SceneManager.LoadScene(1);
    }
}
```

Ideas

Each new level in your game should be in its own Scene. If you want to use game objects or other assets in more than one Scene, you should make it a Prefab. When a Prefab asset is updated, the changes are made everywhere and not just the active Scene.

Camera

The Camera determines what the player sees when they play your game.

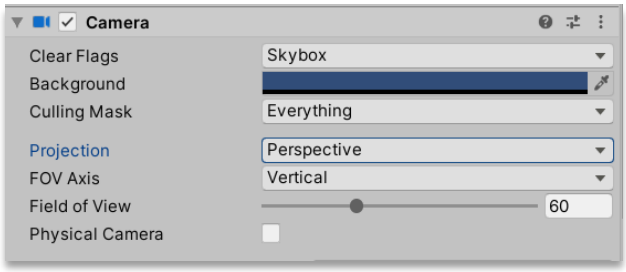


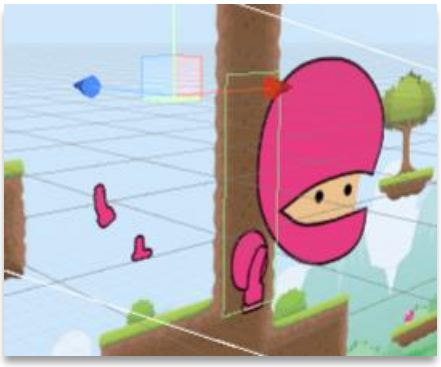
When you select the Main Camera in the Hierarchy, you can see a Camera Preview window in the Scene tab.

Instructions

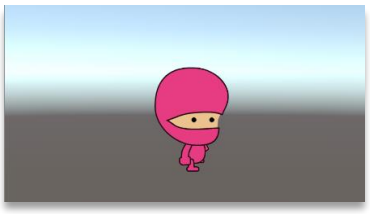
1 The Camera component has two different Projection types that alter how the Camera views your scene, Perspective and Orthographic.

2 When your camera has the Projection set to Perspective, your Scene will have depth. This means that items far away from the camera will look smaller than items closer to the camera. This will typically be used in 3D games, although some 2D games can use it to create an illusion of depth.

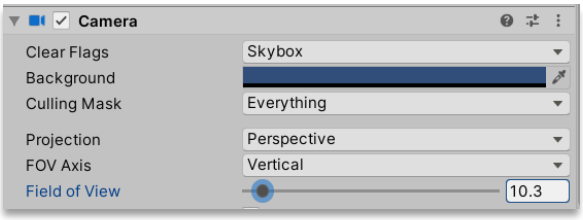




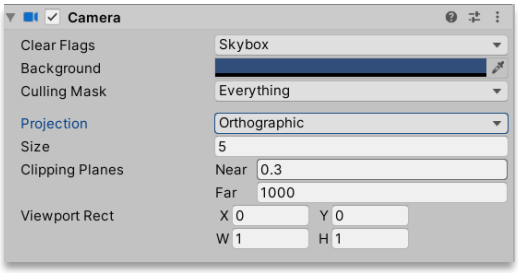
Note: Perspective property can make some 2D sprites look weird or incomplete.

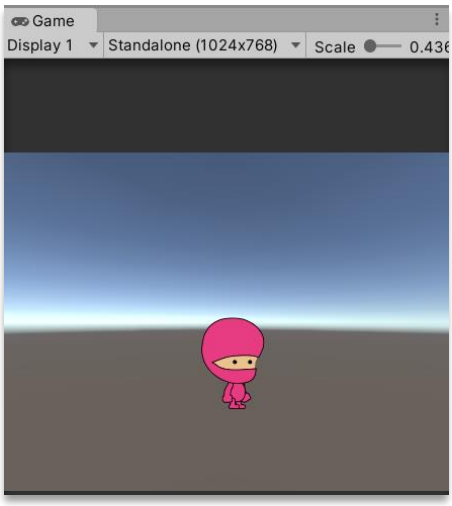


3 You can zoom your camera in or out by using the Field of View property.

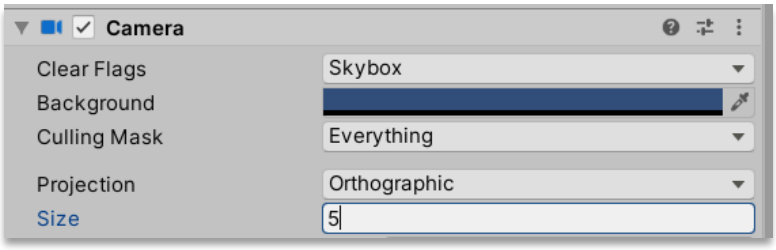


4 The Orthographic Projection removes depth from your scene. This setting is most often used in 2D games.



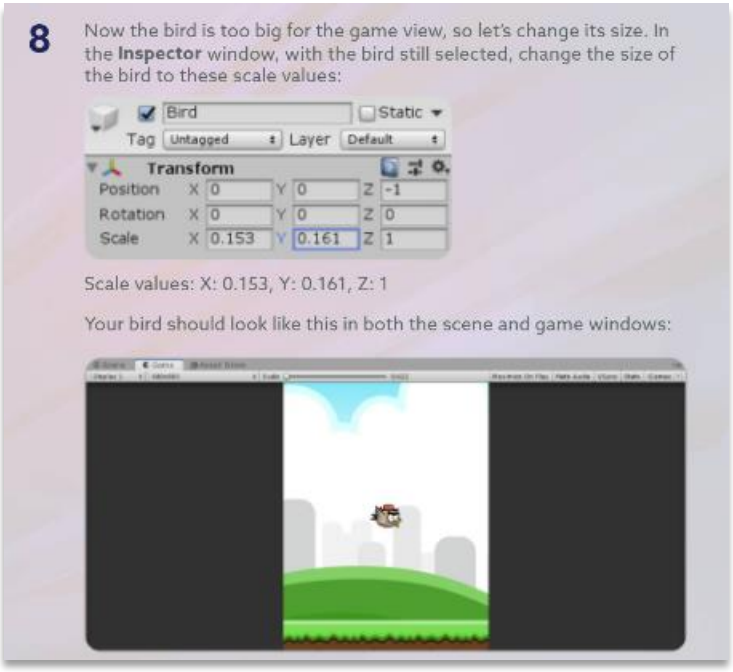


5 The Size property will zoom the camera in or out.



Examples

In Purple Belt's Meany Bird, you learned how to position objects in the view of your camera. You also resized the game objects to ensure they all fit properly.



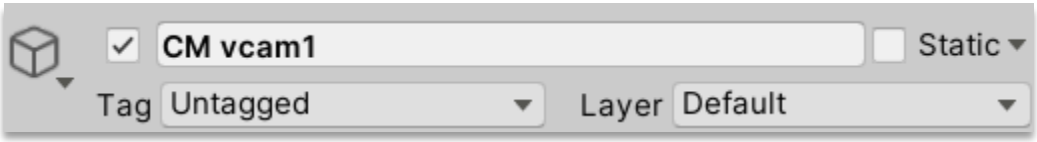
Ideas

If you don't need your camera to constantly follow a game object you can position the Main Camera where it can see the necessary game objects and keep it there.

When you build a 3D game, it is best to set the Projection property to Perspective so the player can be aware of how far or close they are to the objects in the scene.

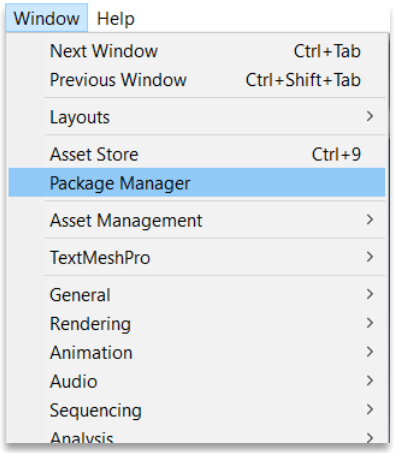
Cinemachine

By default, Unity games have a basic camera that can be difficult to control. By using the Cinemachine package, you can manage your game's camera without programming.

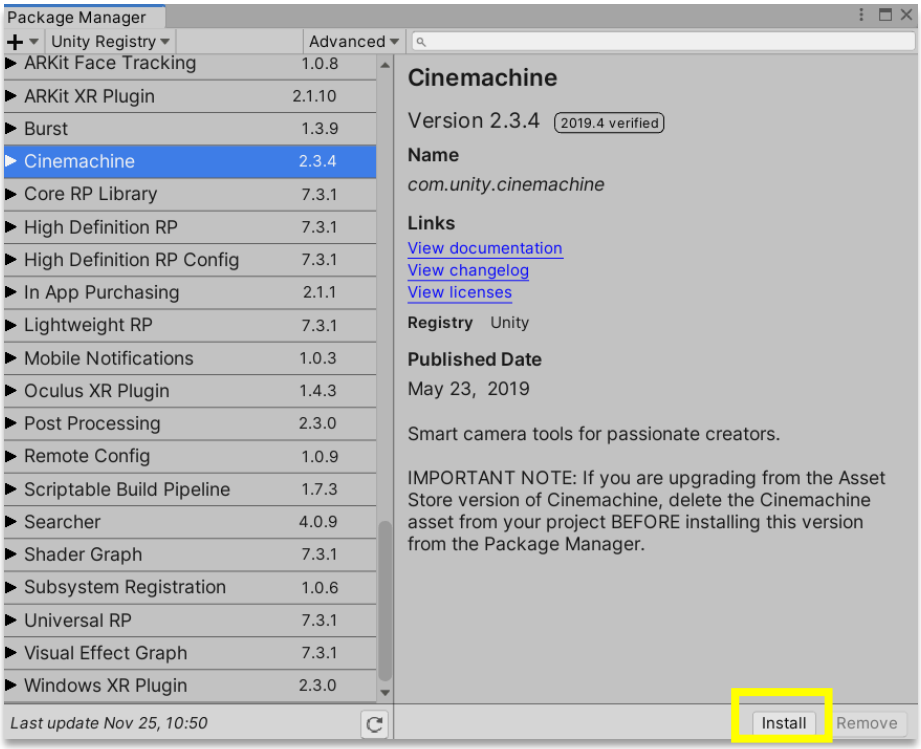


Instructions

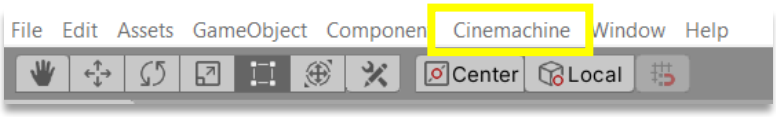
1 New Unity projects do not start with Cinemachine installed. Select the Window menu and click on Package Manager.



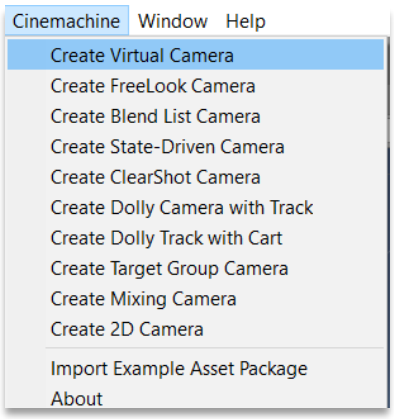
2 A pop-up will appear on your screen. Find and select Cinemachine from the list on the left, then click on the Install button.



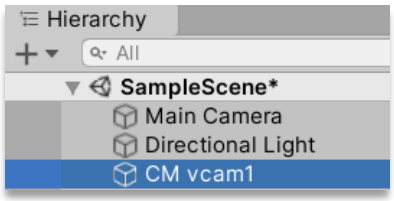
3 After installation, a Cinemachine menu will be added to Unity's menu bar.



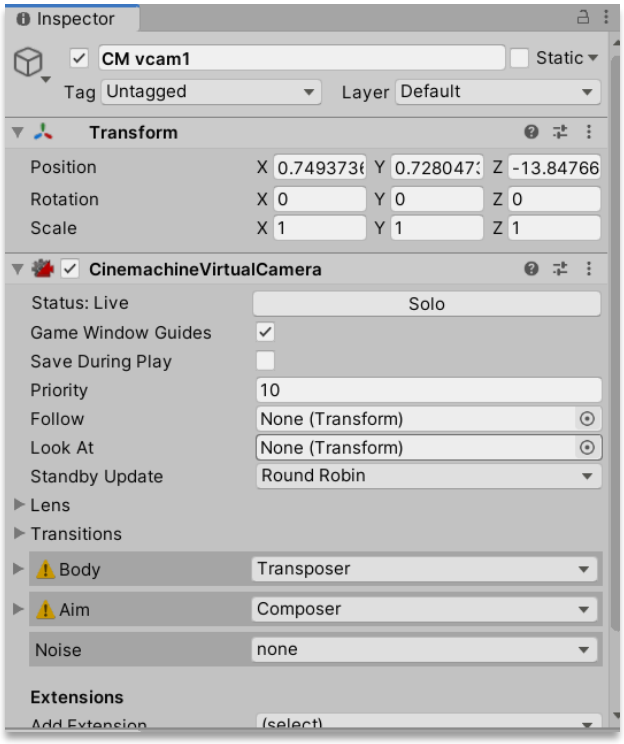
4 Click the new Cinemachine menu and select Create Virtual Camera.



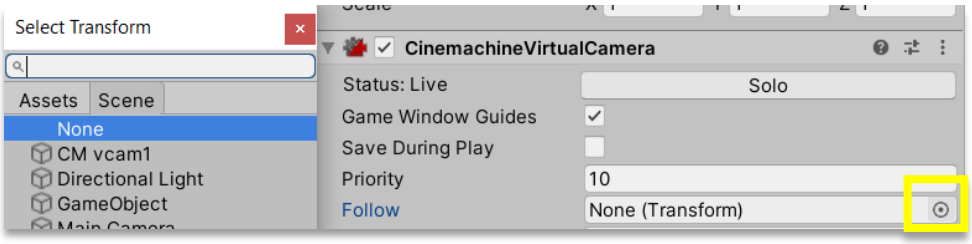
This will create a new game object in your Hierarchy called **CM vcam1.**



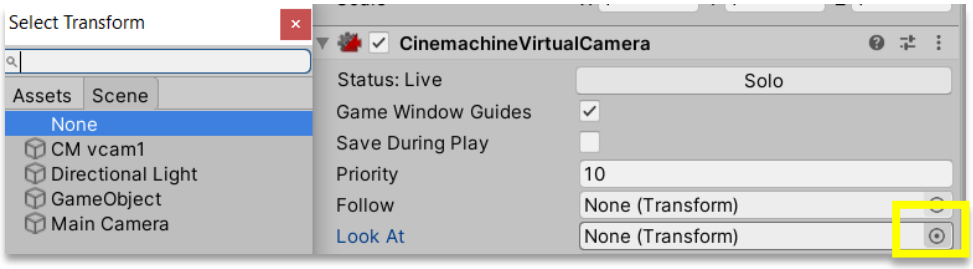
5 Click on CM vcam1 and look in the Inspector. The Cinemachine Virtual Camera component will let you control the behavior of your camera.



6 The Follow property will attach the camera to a specific game object's transform. To change which game object the camera is attached to, click on the small circle to open the Select Transform window.



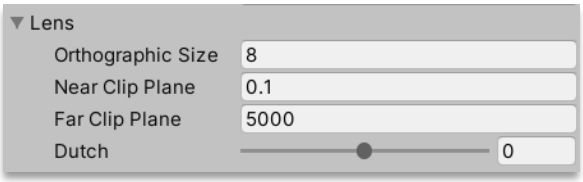
7 To make the camera look at the game object, you need to set the Look At property. Click on the circle to the right and choose a game object from the Select Transform window.



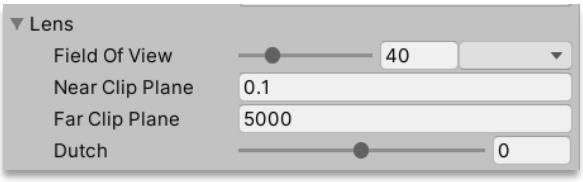
Using both the Follow and Look At properties together will ensure that the camera faces the game object it is following.

8 The Lens property determines the zoom factor of your camera.

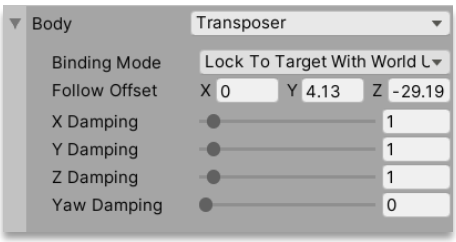
When you use Orthographic View, the smaller the Orthographic Size, the more zoomed in your camera is.



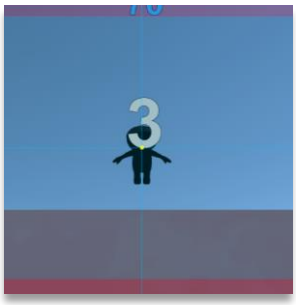
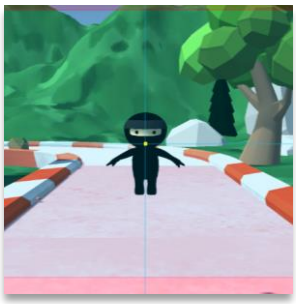
When you use Perspective view, the larger the Field of View, the more zoomed out your camera is.



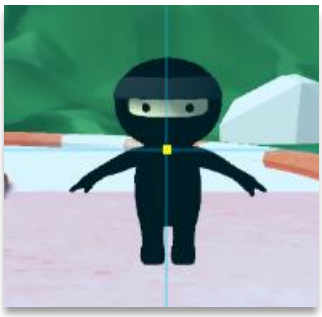
9 If your camera is following a game object, you get an additional set of Body properties.

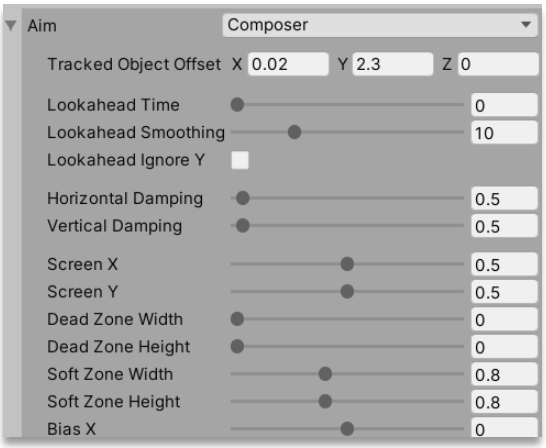


Adjusting the values of the Follow Offset property will change the position of the camera.

	Positive	Negative
X		
Y		
Z		

10 If your camera is looking at a game object, you can adjust it through the Aim property. Notice the small, yellow square in your Game tab. This will help you clearly see where your camera is looking.

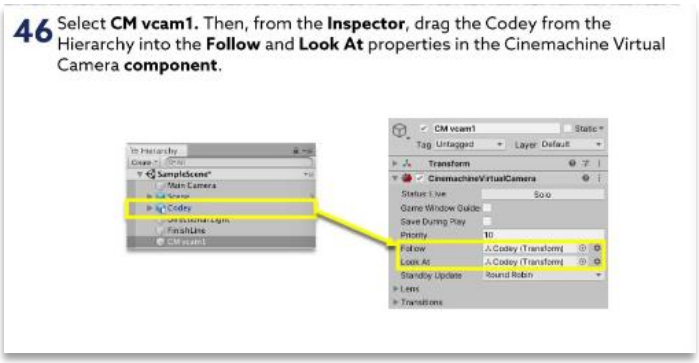




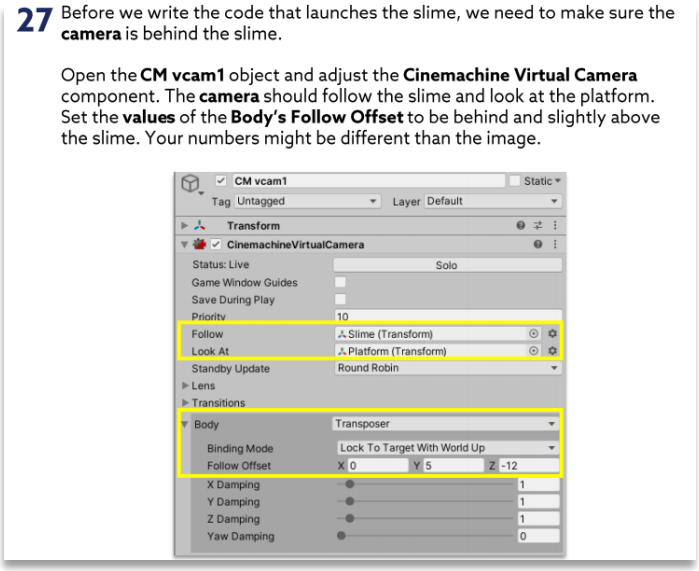
By changing the values of the Tracked Object Offset you can tell the camera exactly where it should be centered.

Examples

The Eyes on the Road section of Codey Raceway details different ways to adjust the values so that your camera is behind Codey at all times.



Sulky Slimes uses Cinemachine to make the camera follow the slime and look at the targets.

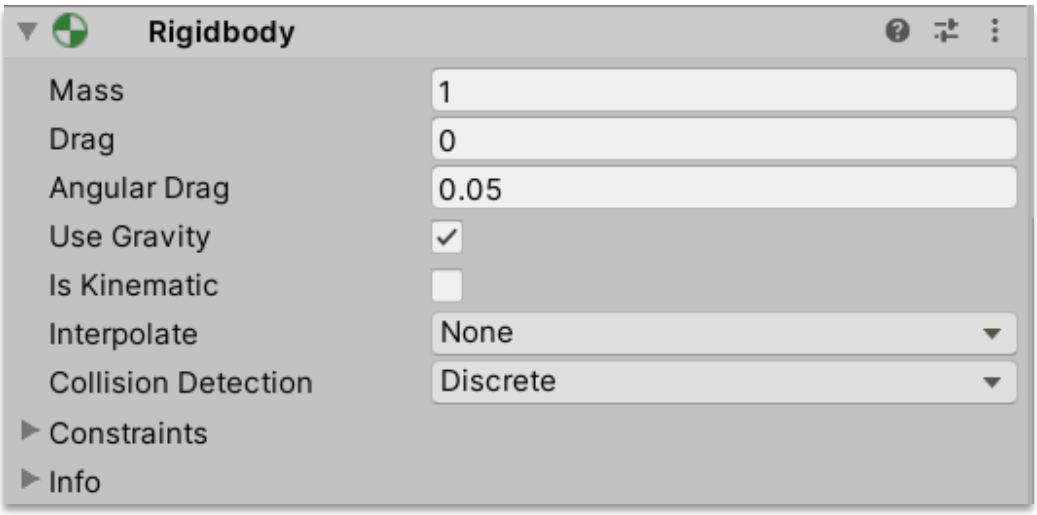


Ideas

Use Cinemachine to focus the camera around your player. There are a lot more properties to play around with and discover!

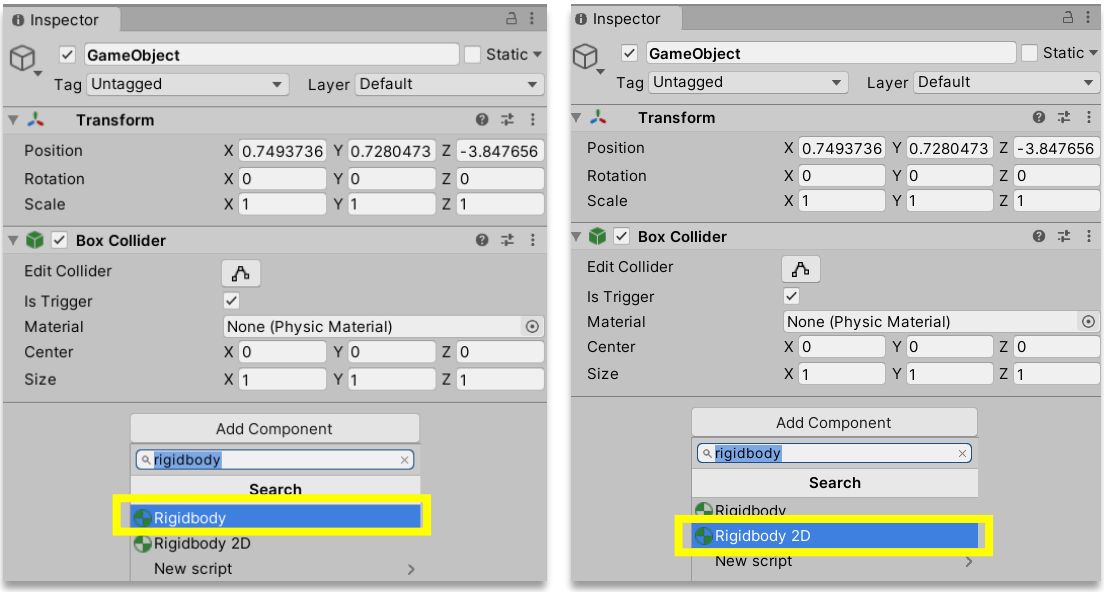
Rigidbody Physics

The Rigidbody component applies physics to our game objects. This means that they will now act as if they are in the real world, affected by gravity and other forces. Objects with Rigidbody components will fall down automatically, have some weight to them, and can interact with other objects in the scene.

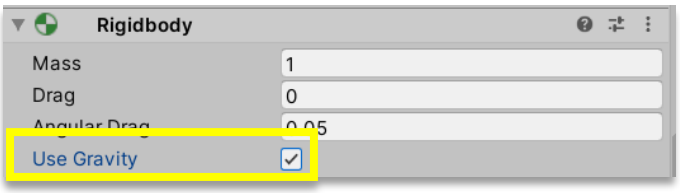


Instructions

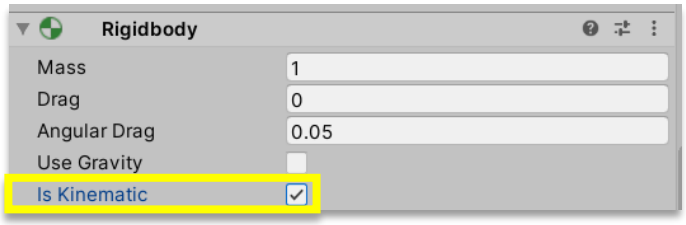
1 Add a Rigidbody component to your game object. Use the Rigidbody component for 3D objects and the Rigidbody 2D component for 2D objects and Sprites.



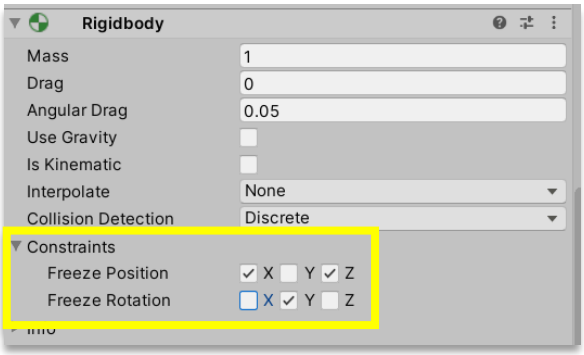
2 The Rigidbody's Use Gravity property determines if your game object will be affected by the Unity Physics Engine's gravity.

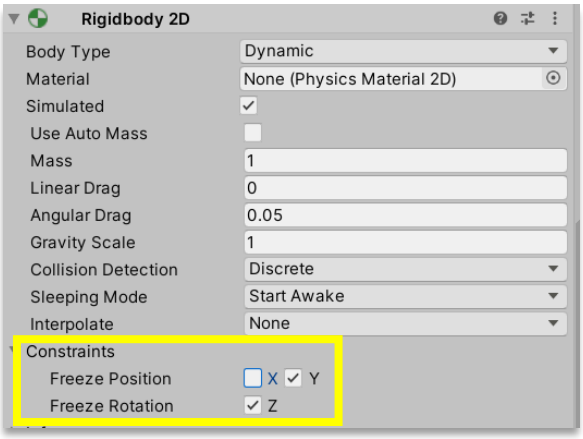


3 The Rigidbody's Is Kinematic property determines if you can apply forces to the game object. If it is not checked, you will not be able to apply force using code.



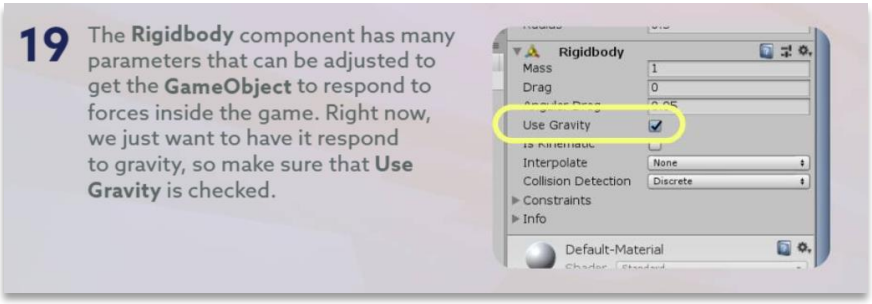
4 Both 2D and 3D Rigidbody Components have a property called Constraints. Adjusting these allow you to restrict the movement of your game objects along an axis. If you do not want your game object to move or rotate in a certain direction you can check the boxes next to the X, Y, or Z property to restrict movement in the given direction.



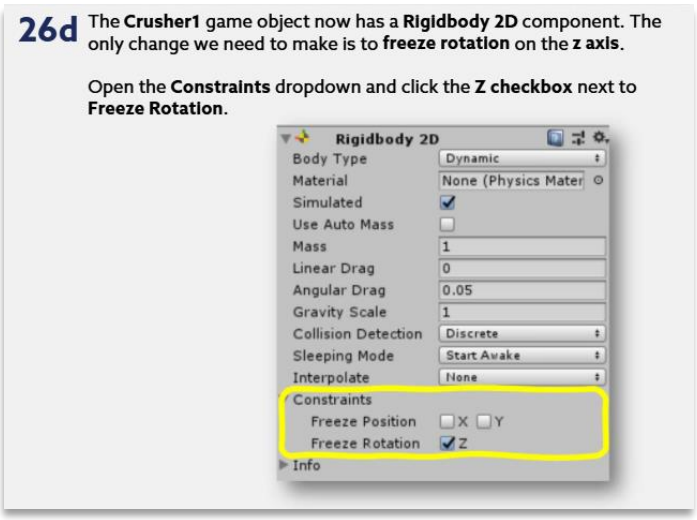


Examples

In Dropping Bombs, you used a Rigidbody and applied gravity to your circles to have them fall down.



In Robomania, you had to freeze the Z rotation to ensure that your crushers did not rotate in circles when they moved.



Ideas

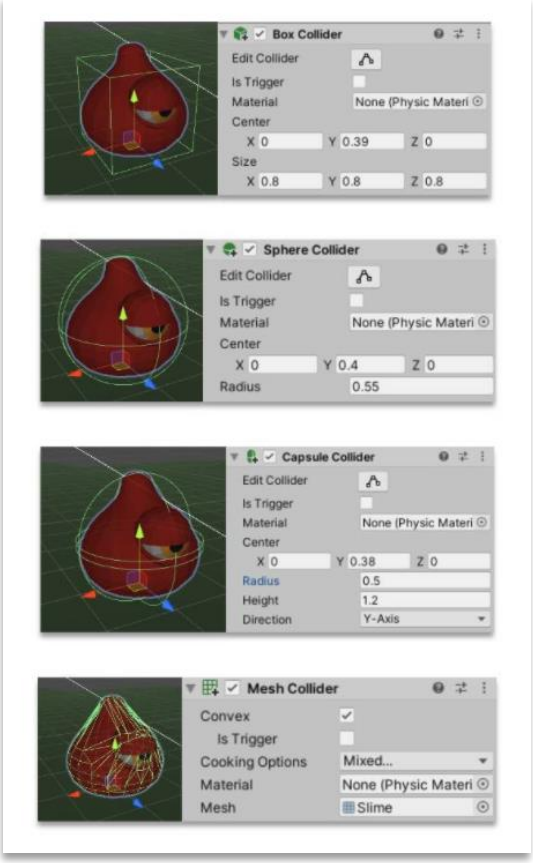
Constraints give you control over how game objects react to your world. You can freeze an object's Y position so it doesn't move up or down, or you can make it so the object doesn't rotate when it's pushed.

You can uncheck Use Gravity on a collectable or other item that you want floating in the air.

Colliders

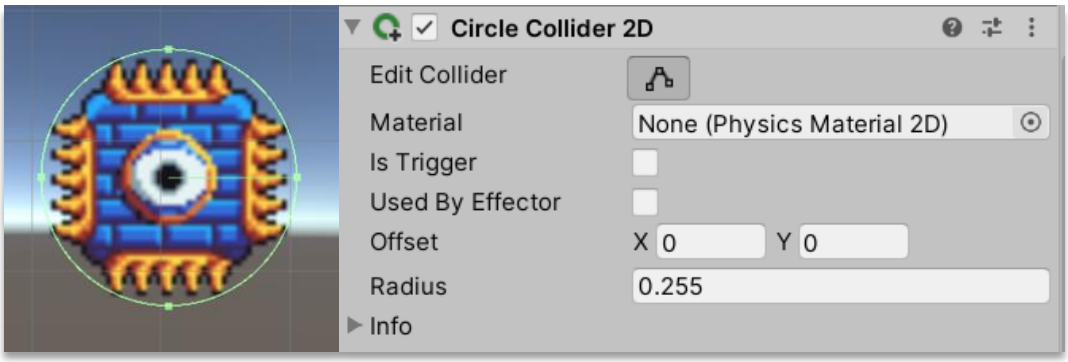
Unity uses colliders to determine when game objects make contact with each other. Colliders give shape to game objects and enable them to interact with one another.

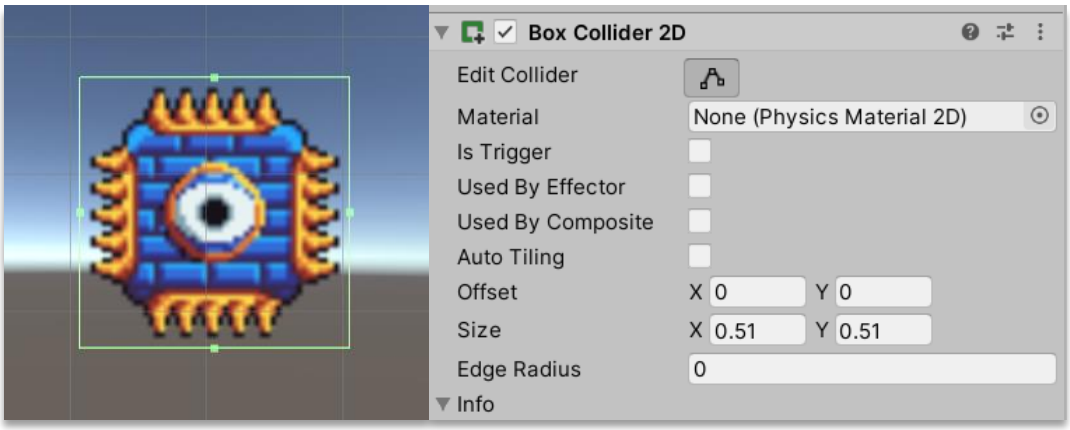
There are many colliders in Unity that you can choose from. Four common 3D colliders are Box, Sphere, Capsule, and Mesh Colliders. In step 10 of Sulky Slimes, you can clearly see the difference between each.

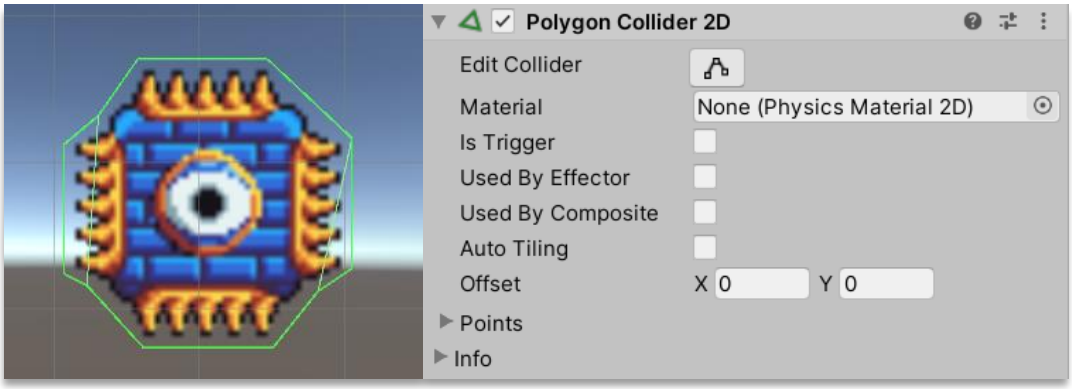


All of the colliders perform the same job, but the collider you chose depends on the shape of your game object. You have more freedom to change the size of your Box, Sphere, and Capsule colliders. This is not true with the Mesh Collider, as it tries to create a collider that fits your game object perfectly. You also must be sure to turn on the **Convex** option with the Mesh Collider to enable it to interact with other colliders.

When working in 2D, you have a different set of colliders to choose from. We have focused on using the Box, Circle, and Polygon Collider 2D.

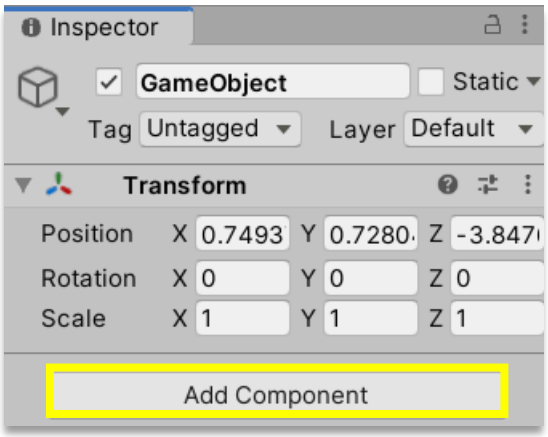




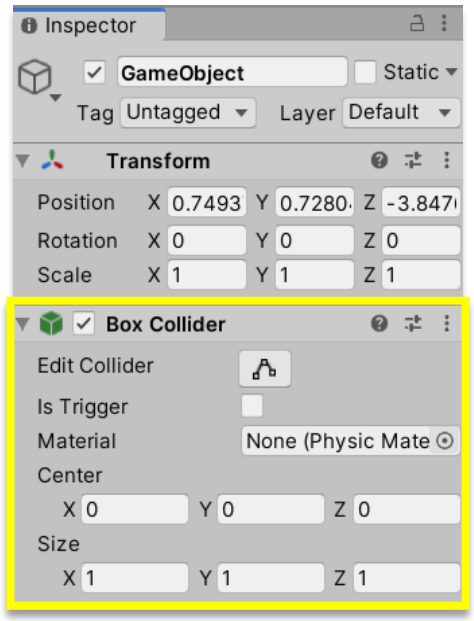


Instructions

1 To detect when game objects collide, they must have Collider components. Go to the Inspector and click on Add Component.



2 Search for the collider that you want and click on it so that it gets added to your game object. We are using a Box Collider as an example. Remember that colliders can only collide with other colliders!



3 Each function has a parameter that represents the object that has been collided with. You can check the different properties like the tag or name of the object and run code if it matches certain conditions.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "enemy")
    {

    }
}
```

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.gameObject.name == "Boss")
    {

    }
}
```

4 Unity has special collision functions for both 3D and 2D colliders.

```
private void OnCollisionEnter(Collision collision)
{

}
Unity Message | 0 references
private void OnCollisionStay(Collision collision)
{

}
Unity Message | 0 references
private void OnCollisionExit(Collision collision)
{

}
```

```
private void OnCollisionEnter2D(Collision2D collision)
{

}
Unity Message | 0 references
private void OnCollisionStay2D(Collision2D collision)
{

}
Unity Message | 0 references
private void OnCollisionExit2D(Collision2D collision)
{

}
```

The only differences between the 3D and 2D colliders is that the 2D collider functions have “2D” at the end of their names and the parameter types.

5 The syntax to check what objects we have collided with in our 3D functions is exactly the same!

```
private void OnCollisionEnter(Collision collision)
{
    if(collision.gameObject.tag == "enemy")
    {

    }
}
```

```
private void OnCollisionEnter(Collision collision)
{
    if(collision.gameObject.name == "Boss")
    {

    }
}
```

6 Let's look at how the different collider functions work.

OnCollisionEnter and OnCollisionEnter2D check for collision on the very first frame when two game objects touch. If you use this function, know that will only register a collision once, when the game objects first touch.

Example: A projectile should hit an enemy only one time.

OnCollisionStay and OnCollisionStay2D check for collision as long as the two objects are touching.

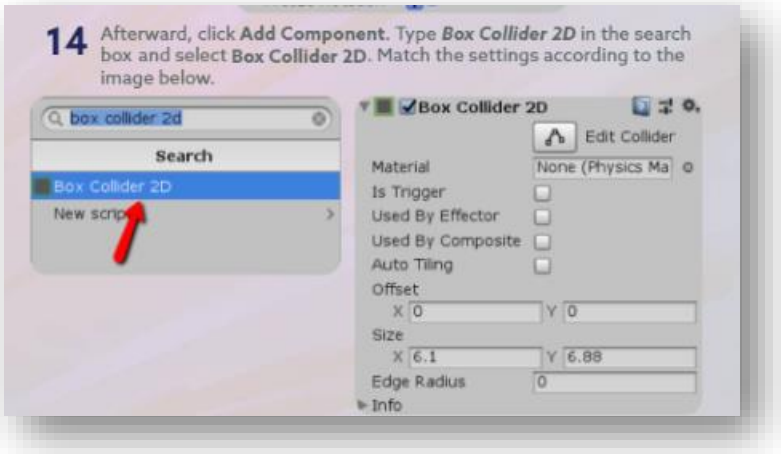
Example: If your player's character is standing on fire, they should take damage the entire time they are standing on it.

OnCollisionExit and OnCollisionExit2D will run on the frame that the two objects stop colliding.

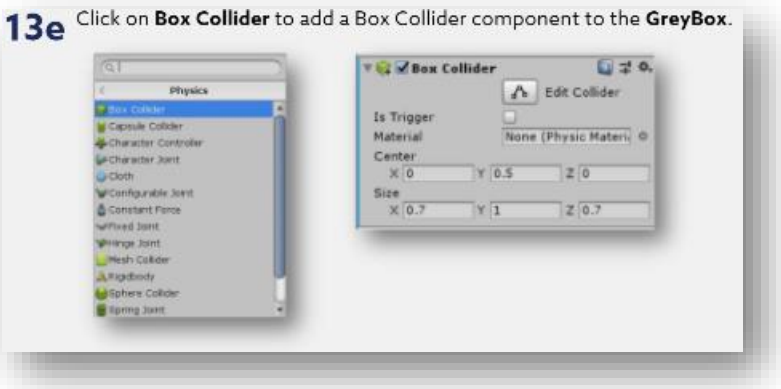
Example: If you have an object that acts as a switch for a countdown, you want the clock to start counting down once the player is off the switch.

Examples

We have used colliders in every game that we have created. On step 14 in Sketch Head, you added a 2D Box Collider to help you determine when the game object landed on a platform.



For a 3D example, look at step 13e and 13f of Evil Fortress of Doctor Worm. Remember how Codey could go straight through the grey box rather than pushing it?

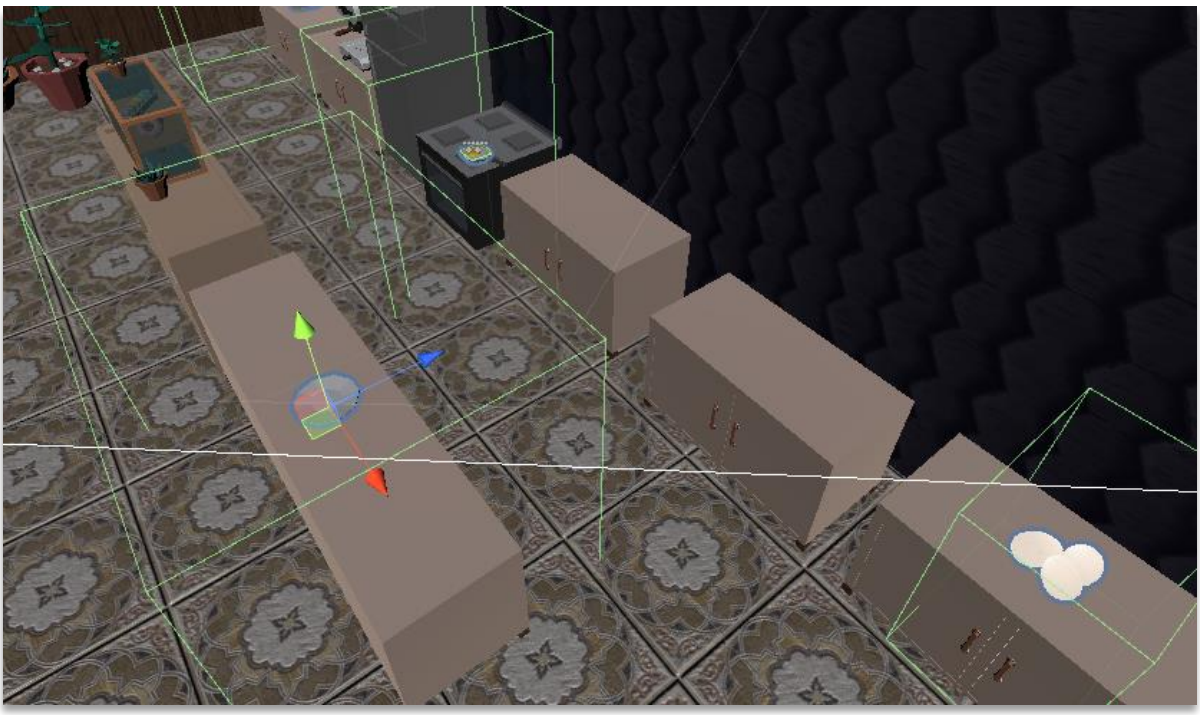


Ideas

You can use colliders to prevent a player from going into areas they can't explore. You could also use them to determine if a player will take damage if they interact with a object in the scene.

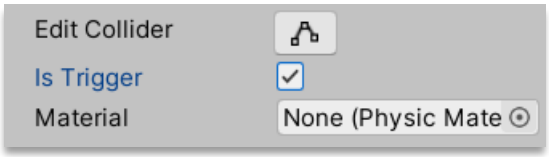
Triggers

A Trigger is a special type of collider that lets you detect collision without the objects stopping each other. Triggers are often used to enable checkpoints and track other kinds of game events.



Instructions

1 When you want an object to act as a trigger, you must check the **Is Trigger** box in the Inspector.



Your game object will recognize when it touches another game object, but will not prevent the object from passing through it.

2 Unity has special functions for both 3D and 2D triggers.

```
0 references
private void OnTriggerEnter(Collider other)
{

}
0 references
private void OnTriggerStay(Collider other)
{

}
0 references
private void OnTriggerExit(Collider other)
{

}
```

```
0 references
private void OnTriggerEnter2D(Collider2D collision)
{

}
0 references
private void OnTriggerStay2D(Collider2D collision)
{

}
0 references
private void OnTriggerExit2D(Collider2D collision)
{

}
```

The only differences between the 3D and 2D triggers is that the 2D trigger functions have "2D" at the end of their names and the parameter types.

3 OnTriggerEnter and OnTriggerEnter2D run on the very first frame when two game objects collide. Your object will still be able to go through it since it is a trigger.

OnTriggerStay and OnTriggerStay2D continue to run as long as the two objects are colliding with one another.

OnTriggerExit and OnTriggerExit2D run on the frame the object and the trigger stop colliding.

4 To ensure that you don't run your OnTrigger functions for every single game object, you can add a conditional that checks the colliding object's name or tag. This works for all six OnTrigger functions.

```
0 references
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.name == "door")
    {
        // code to open the door
    }
}
0 references
private void OnTriggerStay2D(Collider2D collision)
{
    if (collision.CompareTag("offroad"))
    {
        // code to slow movement
    }
}
```

Examples

In Ninja Run we used a trigger to create a pickup item! Step 8c shows with a few of the many things we can do with triggers, such as adding to a UI, destroying objects, and playing animations.

8c Remember this emphasizes that a coin's been picked up, so it should only play when the trigger event is called. We will use the functions `ParticleSystem.Start()` and `ParticleSystem.Stop()` which tells Unity when to play or stop a particle system.
Back in our **Pickup** script, under `Start()`, add:

```
void Start()
{
    Pickup.Stop();
}
```

In `OnTriggerEnter()`, add:

```
void OnTriggerEnter(Collider other)
{
    if(other.gameObject.CompareTag("coin")){
        score++;
        scoreText.text = score.ToString();
        Destroy(other.gameObject);

        Pickup.Play();
    }
}
```

In Chef Codey, Codey interacts with items in the environment through triggers.

40 When Codey enters a **trigger**, we want to set the **triggerName variable** to the **name** of the **collider**. When Codey leaves a **trigger**, we want to reset the **triggerName variable** to an empty **string**.

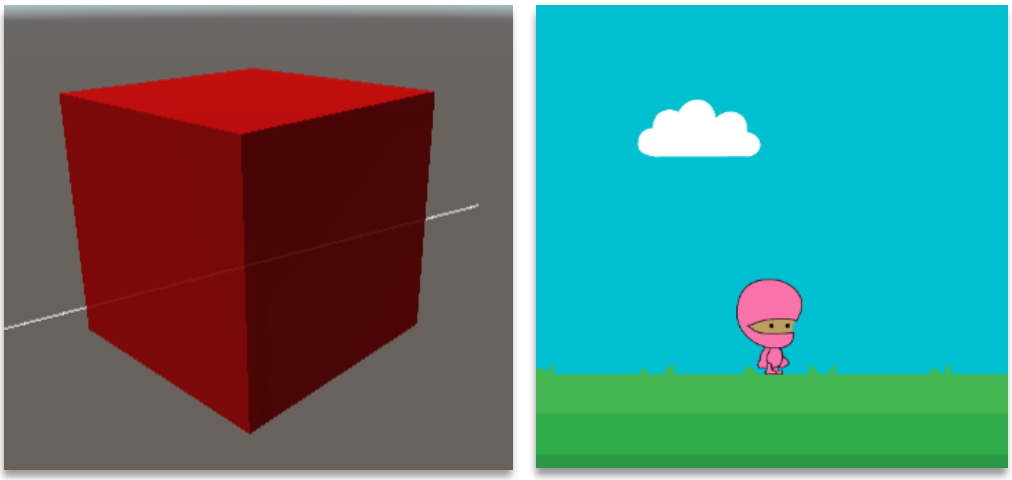
```
0 references
private void OnTriggerEnter(Collider other)
{
    triggerName = other.name;
}

0 references
private void OnTriggerExit(Collider other)
{
    triggerName = "";
}
```

Ideas

Triggers can be used to automatically open or close doors based on a player's position. You can also alter different gameplay elements like speed and gravity when the player is in a special trigger.

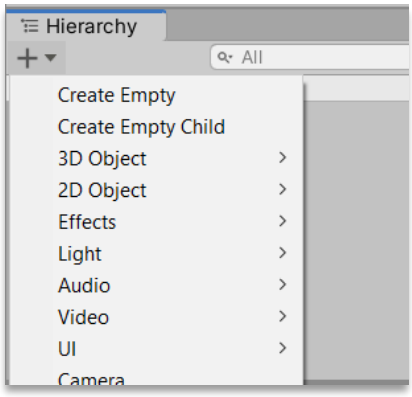
Game Objects

Game Objects are a fundamental part of game development because they make up every game you build. Game Objects can be anything from simple 3D cubes or 2D squares to very detailed 3D models or 2D sprites.

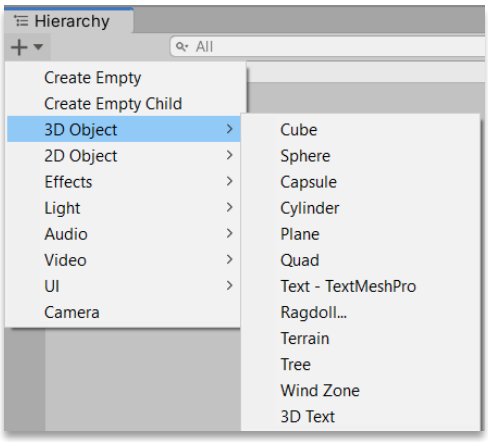


Instructions

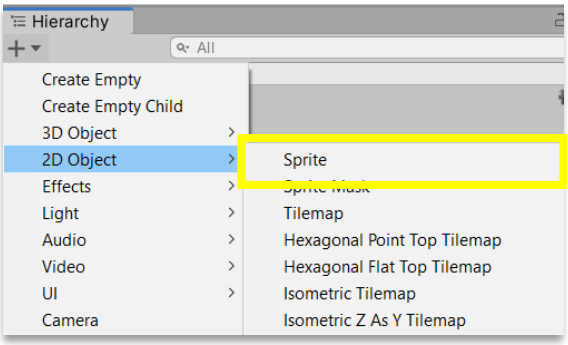
1 You can add a Game Object to your scene by right clicking in the Hierarchy or clicking on the plus sign to open the menu.



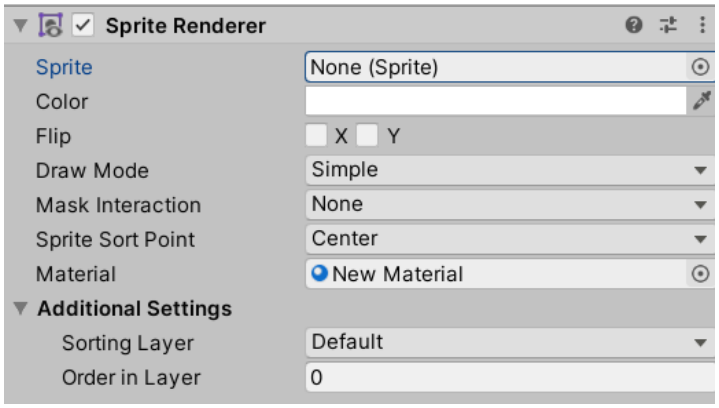
2 If you want to create a 3D object, hover over 3D Object and then select the shape that you want.



3 If you want to create a 2D object, hover over 2D Object and select Sprite.

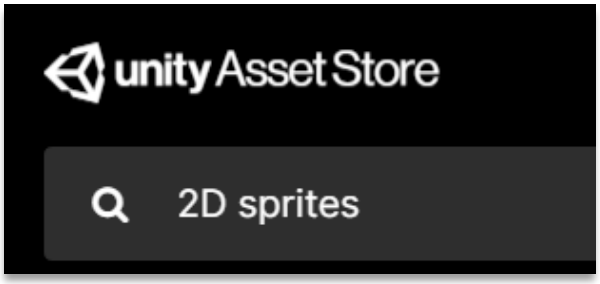


4 To get a 2D game object to appear, you must fill the Sprite Renderer component's Sprite property with an image.



5 You can use the Unity Asset Store to find amazing game objects to add to your game.

You can search for a general type of game object like "2D sprite" or something specific like "tree."



Examples

Every game that you created in Purple, Brown, and Red Belts contained different kinds of game objects! Load your previous projects and export any game objects you want to use again as a .unitypackage file.

Ideas

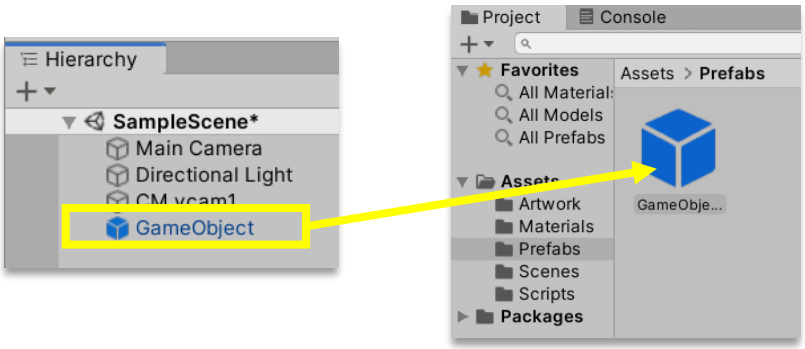
Game objects can be placed in your scene for decoration. Not all game objects have to be interactive.

Prefabs

Prefabs (or Prefabricated Objects) are reusable game objects that can be shared between scenes or games. Assets downloaded from the Unity Asset Store are Prefabs that you can modify in your games.

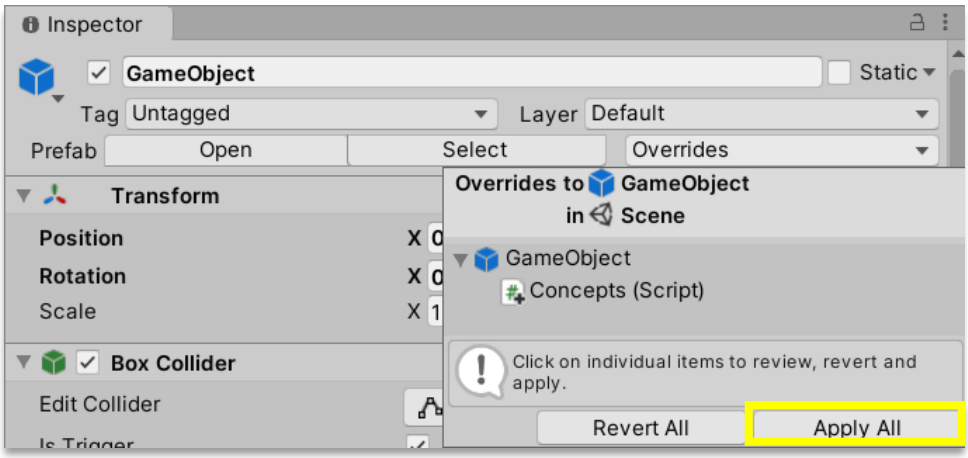
Instructions

1 Prefabs are created by dragging a Game Object from the Hierarchy to the Project tab. You can create a new Prefabs folder to store all your saved game objects. Once the Prefab is created, the name and icon will turn blue to show that it is attached to a Prefab object.



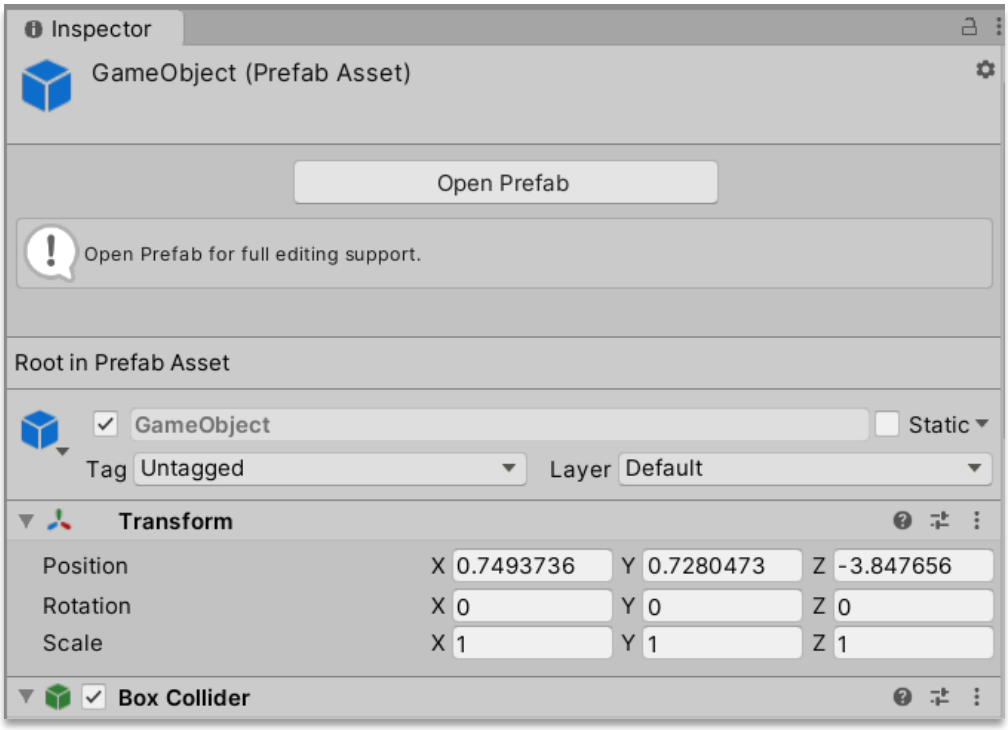
The Prefab game object can be dragged from the Assets folder to the Hierarchy of any scene in your project.

2 By default, changes that you make to the prefab game object in the Hierarchy of your Scene will only affect that instance of the object. To apply your changes to all instances of the Prefab, open the Override menu in the Inspector and click Apply All.



3 Any changes that you make to your Prefab object will be automatically applied to each instance of the game object in the Hierarchy of your Scenes.

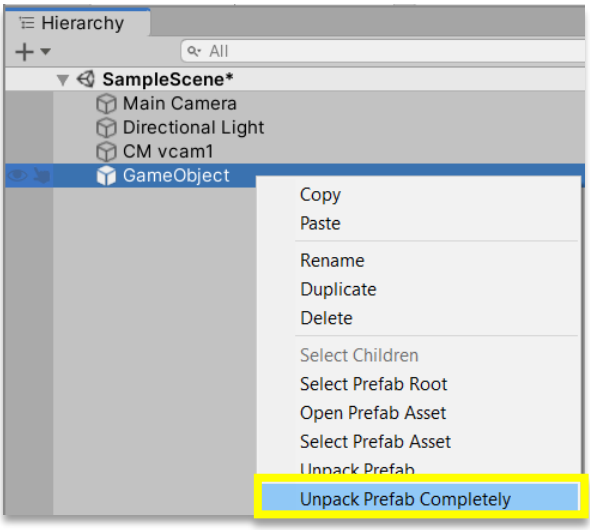
Click on you prefab and make your changes in the Prefab's Inspector.



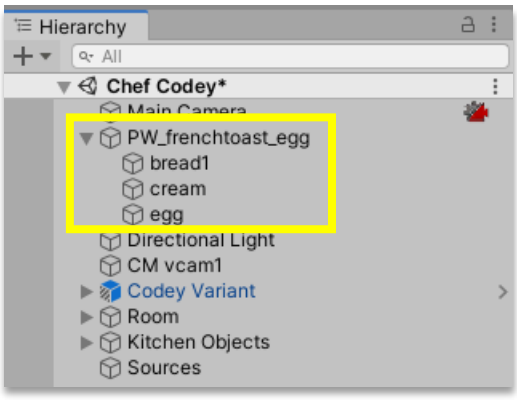
Notice that you no longer have the Overrides option as your changes are applied right away.

4 Sometimes you need to make special changes in only one Scene. You can break a game object's connection with the Prefab by "unpacking" the Prefab in the Hierarchy.

Right-click on the blue object in the Hierarchy and select Unpack Prefab Completely.



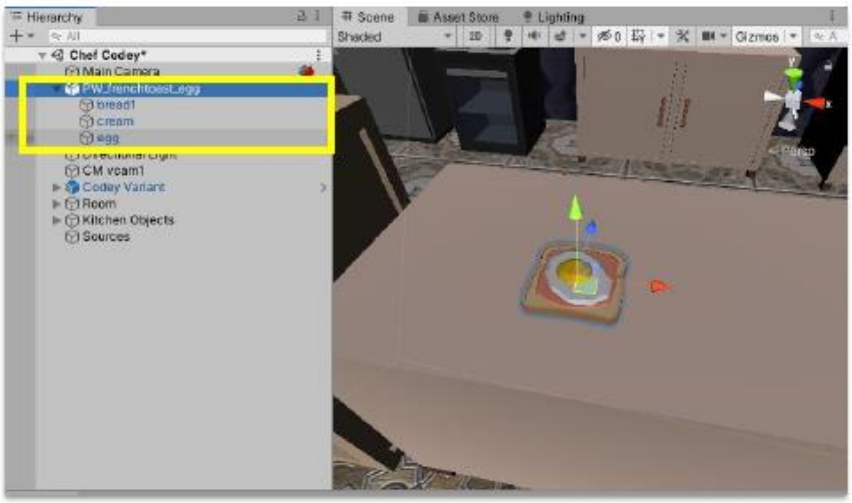
In the Hierarchy, the text color of the game object will return to black and it will no longer be connected to the Prefab asset. Be careful, because this cannot be undone! Once you disconnect the object, you cannot reconnect it later.



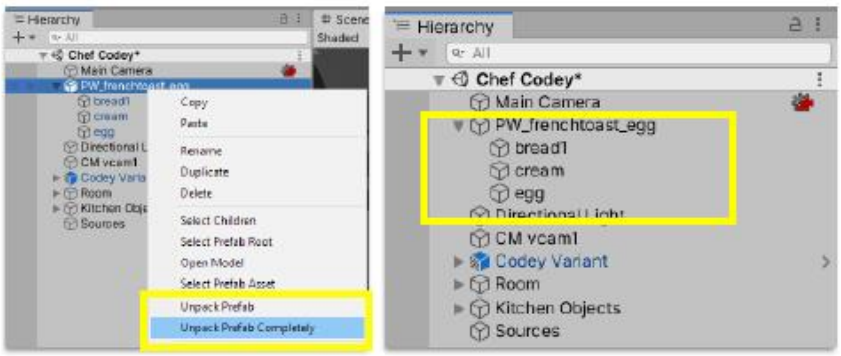
Examples

In Red Belt's Chef Codey, you may have learned how to unpack a Prefab game object from the Asset Store.

27 Once you have your **model**, you might have to separate the pieces to use in your **scene**. If your **object** is not made out of different parts, then the next steps might not be necessary.



28 Right click the blue object in the **hierarchy** and select "Unpack Prefab Completely". Remember to resize your **object** so it looks like it belongs in your **scene**.



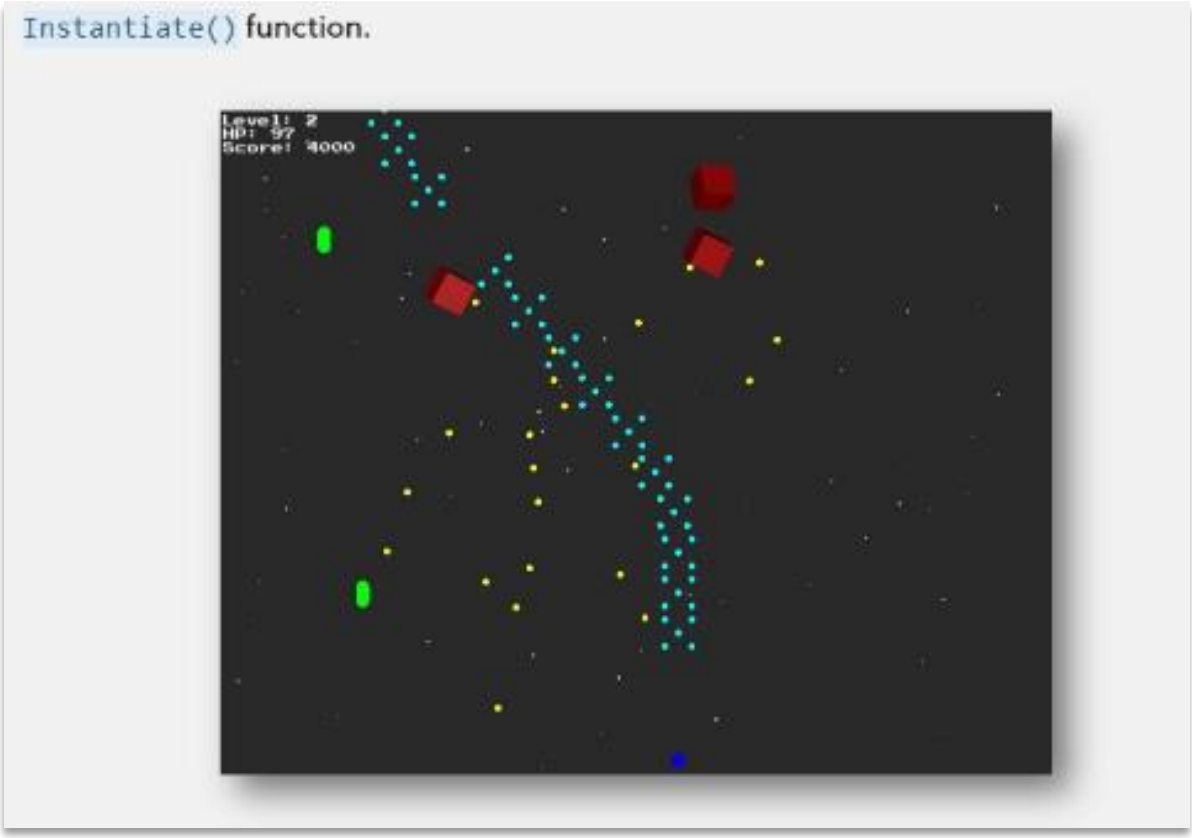
Ideas

You should create Prefabs whenever you want to reuse a game object across multiple Scenes.

The Assets you download from the Unity Asset Store will be Prefabs. You can unpack the Prefabs to make any changes you want.

Instantiate

Instantiating in Unity is a fancy way to say you are cloning an object. The Instantiate function lets you specify which game object you want to clone and the exact location you want to place it in your scene.



Note: There is more than one way to use the Instantiate function. The instructions below cover the most common syntax used for the Instantiate function.

Instructions

1 Create a GameObject variable. You can set the value of this variable to a prefab in the Inspector.

```
public GameObject myGameObject;
```

2 You can call the Instantiate function anywhere in your script. Some possibilities include when a button is pressed or when a collision is detected.

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        Instantiate(myGameObject);
    }
}
```

```
0 references
private void OnCollisionEnter(Collider other)
{
    Instantiate(myGameObject);
}
```

3 Once you decide when to clone the object, you must tell the Instantiate function the game object it should clone, its position, and its rotation.

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        Instantiate(myGameObject, transform.position, transform.rotation);
    }
}
```

Different activities in Purple, Brown, and Red modify the second and third parameters of this function to control the exact position and rotation of the cloned object.

Examples

In step 11f and 11g of Shape Jam, you learned how to clone the projectiles and how to position them either on the right or left side of your player's game object.

11f We now need to add the right and left offsets to the position of our new projectiles. In the first Instantiate, change the second argument to be transform.position + rightOffset.

```
if (currentLevel >= 3)
{
    Vector3 rightOffset = new Vector3(0.2f, 0, 0);
    Instantiate(projectile, transform.position + rightOffset, transform.rotation);
}
```

In step 49 of Chef Codey, we introduced a new way of using the Instantiate function. This form's parameters are: the game object we want to clone, the game object we want as our new object's parent, and either true or false to tell Unity how to position the new object in relation to the world. Once the object is instantiated, you can set its position by using its transform.

49 Codey now has a piece of bread, but it's not in the best **position**. Stop the game and open your Interact **script** again. After the Instantiate line, we can set the heldItem's **position**.

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")
        {
            heldItem = Instantiate(breadPrefab, transform, false);
            heldItem.transform.localPosition = new Vector3(0, 2, 2);
        }
    }
}
```

Ideas

You can Instantiate projectiles for your player or the enemies.

There could be times when you do not want the player to move objects in the scene like in Chef Codey. You can clone the game object you want the player to interact with rather than moving the original game object.

Translate

Translate is used to move our game objects in the direction that we want. It takes three parameters that determine how far it moves on the X, Y, and Z axes.

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        transform.Translate(0, 10, 0);
    }
}
```


Instructions

1 The Translate function is applied to a game object's transform.

```
myGameObject.transform.Translate( );
```

2 The Translate function takes three parameters. The first parameter determines the distance and direction your game object moves along the X axis.

```
myGameObject.transform.Translate(15, -50, 40f);
```

3 The second parameter determines the distance and direction your game object moves along the Y axis.

```
myGameObject.transform.Translate(15, -50, 40f);
```

4 The third parameter determines the distance and direction your game object moves along the Z axis.

```
myGameObject.transform.Translate(15, -50, 40f);
```

Examples

Find the Exit combines user input with the Translate function to move Codey around!

7c In the `Update` function, we want to add one line of code to move our ninja, Codey.

To do this, we will use Unity's `Translate` function on Codey's `transform`. An object's **transform** contains information about its current **position**, **rotation**, and **scale**. Unity's `Translate` function will let us change an object's position. In the `void Update()` function, type `transform.Translate(Vector3.forward * speed * Time.deltaTime);` to tell Codey to move forward at our set speed of 5 at a constant rate of time.

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

All we do when we *Translate* an object in Unity or in the real world is move it from one location to another.

Ideas

You can use the Translate function to move game objects on a consistent path. If you have obstacles that move from left to right, you can use the Translate function inside of the Update function so that they are always moving.

Destroy

When you use Unity's Destroy function, you completely remove a game object from your game.

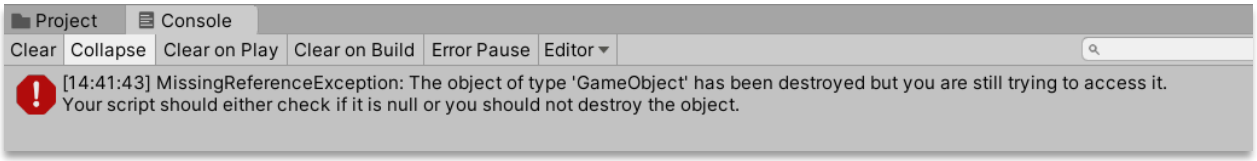
Instructions

1 The Destroy function takes one parameter, the game object you want to remove. Make sure to capitalize the letter D in the function name.

```
Destroy(myGameObject);
```

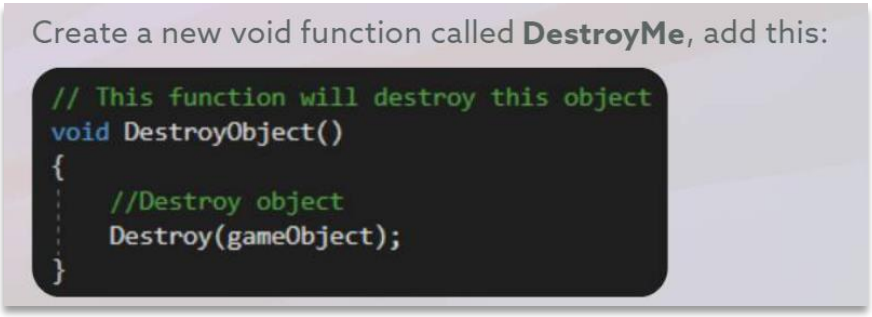
2 You can only call Destroy from of one of the Event Functions or a function you created. Examples of Event Functions include Start, Update, and OnCollisionEnter.

3 If you remove a game object that is needed in another script, Unity will throw an error in the console telling you that other scripts are missing a reference to the object you removed.

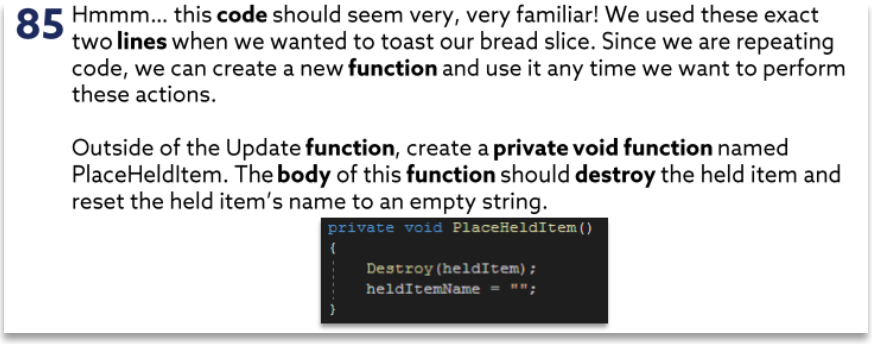


Examples

In step 26 of the Meany Bird activity, you used the Destroy function to remove the Spikes from the scene.



In Chef Codey, you create the illusion that you pick up an item and place it somewhere else by using the Destroy function.

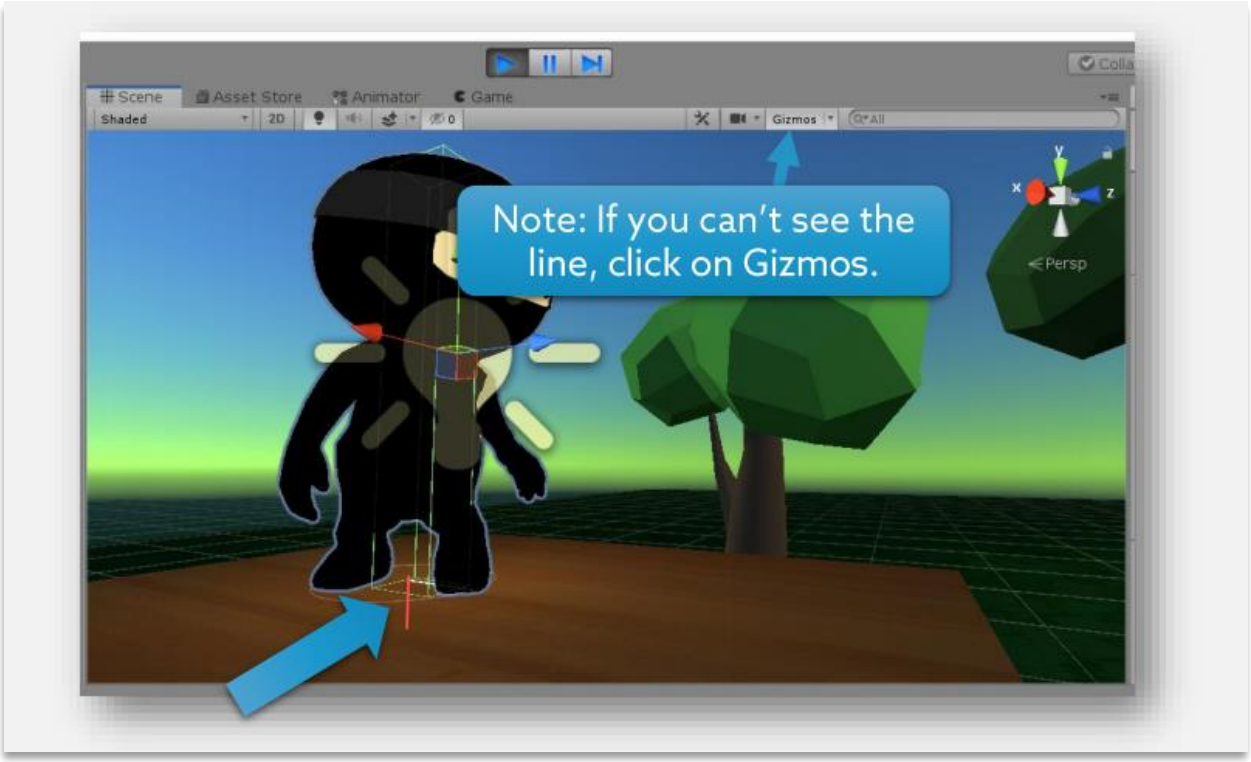


Ideas

It is important to give your player gameplay feedback so they can tell the difference between an interactive game objects and background images. Destroying a collectable is a great way to show your player that they are interacting with the game world.

Raycast

A Raycast is the process of drawing a line, or ray, with a starting point, length, and direction. They are often used to detect game objects touching when a traditional collider won't work.



Instructions

1 The job of a Raycast is to check to see if it collides with another object. Before you use the function, first create a Boolean to store the result.

```
public bool rayDidHit;
```

2 The Raycast function is often used in the Update function to continually check for a hit.

```
public bool rayDidHit;
0 references
void Update()
{
    rayDidHit = Physics.Raycast(  );
}
```

3 The Raycast function needs three parameters: starting position, direction, and length.

The first parameter, the starting position, takes a Vector3. You can start the Raycast at the game object's position with transform.position.

```
public bool rayDidHit;
0 references
void Update()
{
    rayDidHit = Physics.Raycast(transform.position,  );
}
```

4 The second parameter, the direction, also takes a Vector3. Since a Raycast is a line, Unity needs to know which direction to point the Ray. You can create your own Vector3 or use a built-in Vector3 like down, forward, or up.

```
public bool rayDidHit;
0 references
void Update()
{
    rayDidHit = Physics.Raycast(transform.position, Vector3.forward,  );
}
```

5 The third parameter, the length of the Ray, takes a float value. This value will depend on how far away you want to detect objects in your scene.

```
public bool rayDidHit;
0 references
void Update()
{
    rayDidHit = Physics.Raycast(transform.position, Vector3.forward, 25f);
}
```

6 Raycasts are invisible unless you use the Debug.DrawRay function. This function takes four arguments: its starting position, its direction, its color, and its length.

```
public bool rayDidHit;
0 references
void Update()
{
    rayDidHit = Physics.Raycast(transform.position, Vector3.forward, 25f);
    Debug.DrawRay(transform.position, Vector3.forward, Color.red, 25f);
}
```


7 While a simple Boolean will work in a lot of situations, there is an advanced version of the Raycast function that lets you get information about the object that the Raycast collides with.

First, declare a special RaycastHit variable.

```
0 references
void Update()
{
    RaycastHit hitObject;
}
```

8 Then, call the Physics.Raycast function with three parameters: its starting point, its direction, and the keyword "out" followed by your RaycastHit variable.

```
0 references
void Update()
{
    RaycastHit hitObject;
    Physics.Raycast(transform.position, Vector3.forward, out hitObject);
}
```

The "out" keyboard tells Unity to place information about the object the ray hits inside the RaycastHit variable.

9 You can use the different properties on the RaycastHit variable to run different code based on conditionals.

```
0 references
void Update()
{
    RaycastHit hitObject;
    Physics.Raycast(transform.position, Vector3.forward, out hitObject);
    if (hitObject.collider.CompareTag("obstacle"))
    {
        // do something if the ray collides with an obstacle
    }
    if (hitObject.distance < 10f)
    {
        // do something if the ray collides with an object that is close
    }
}
```

Examples

In Brown Belt's Jungle Escape, you used the Raycast function to determine if Codey is on the ground. If the raycast does not detect a game object beneath Codey, then the player cannot jump.

7b Now using what you've learned about `Physics.Raycast()` create a function in `Update` that casts a ray from the player for a short distance. Remember the three parameters: (origin, direction, distance).

Add the parameters as shown here:

```
void Update()
{
    isGrounded = Physics.Raycast(transform.position, Vector3.down, .15f);

    if(Input.GetButtonDown("Jump")){
        rigidbody.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    }
}
```

Ideas

You could build a scene where there are multiple game objects in a small location. You can use the raycast to help identify the specific object that is in front of you.

Enemies can also use the Raycast function to determine if the player is in front of them. If they raycast catches a player in front of them, then they could chase the player.

Invoke

Invoke is a Unity function that allows you to run a function after a delay. You should call Invoke when you want something to happen in the near future.

This function has two parameters:

1) a string parameter that contains the name of the function we want to call, and
2) an int value that specifies how long (in seconds) to wait until calling the function provided in the first parameter.

In the example below, we have called the Invoke function so that our game object is destroyed 3 seconds after the game has started.

```
public GameObject myGameObject;
0 references
void Start()
{
    Invoke("DestroyObject", 3);
}
0 references
private void DestroyObject()
{
    Destroy(myGameObject);
}
```

Instructions

10 Invoke needs to know what function to call. You will need to create the function first.

```
0 references
private void DestroyObject()
{
    Destroy(myGameObject);
}
```

11 Once you have a function you want to use with Invoke, you need to decide when to call it. You can use Invoke in Start, Update, OnCollisionEnter, or any another function.

```
void Start()
{
    Invoke("DestroyObject");
}
```

```
void Update()
{
    Invoke("DestroyObject");
}
```

```
private void OnCollisionEnter(Collision collision)
{
    Invoke("DestroyObject");
}
```

12 The second parameter is the time in seconds to wait before calling the function.

```
void Start()
{
    Invoke("DestroyObject", 21);
}
```

```
void Update()
{
    Invoke("DestroyObject", 7);
}
```

```
private void OnCollisionEnter(Collision collision)
{
    Invoke("DestroyObject", 4);
}
```


Examples

In World of Color, we used Invoke to delay our game from restarting so that the player has a chance to get ready to play again. Notice that we used Invoke within the GameOver function. You can call Invoke from inside just about any function you want.

```
public void GameOver()
{
    player.SetActive(false);
    Invoke("Reload", 2);
}

0 references
void Reload()
{
    SceneManager.LoadScene(0);
}
```

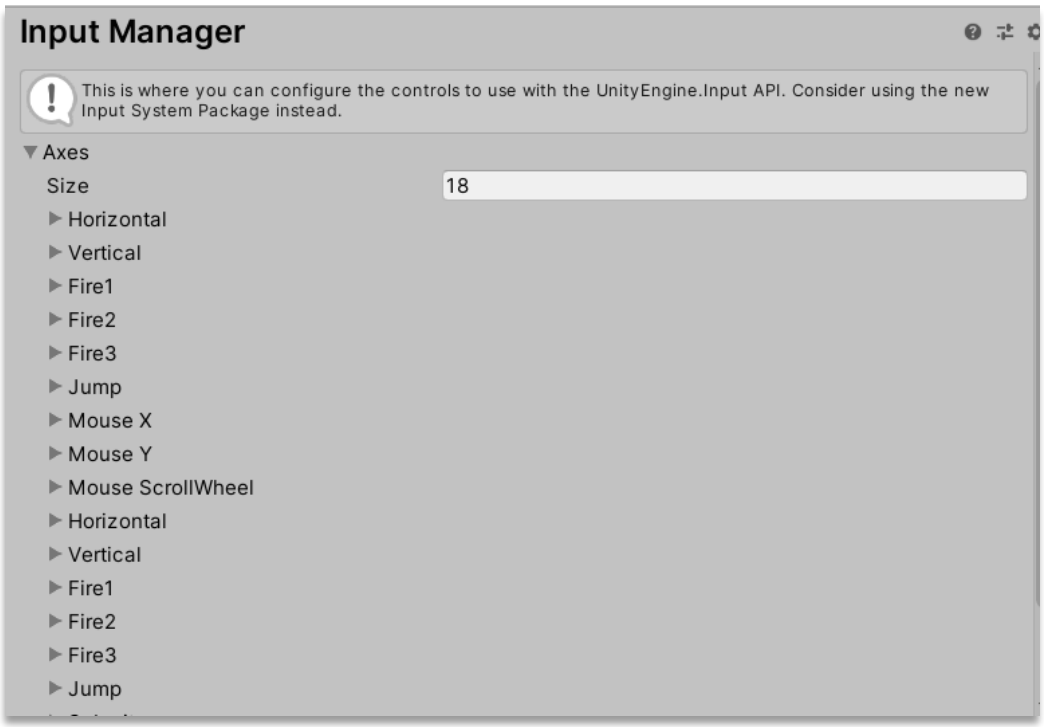
Ideas

Invoke is a great way to create game objects that get added to your game. Just like with World of Color, CyberFu uses Invoke to give players a break after a Game Over (see the PlayerHealth script).

Invoke can also be used to give the player time restrictions. In Chef Codey, the Stove script uses Invoke to simulate the time required to cook.

Player Input

Player Input is essential to enable users to actually play your game. Unity can listen for input from a keyboard, a mouse, or a gamepad for Player Input. You can customize the different types of Player Input in the Input Manager.



Instructions

1 Player Input will return a value if a certain key is pressed at any given moment. We can constantly check if the player has given us the input we are looking for inside of our Update function.

```
0 references
void Update()
{
    if (Input.    )
    {

    }
}
```

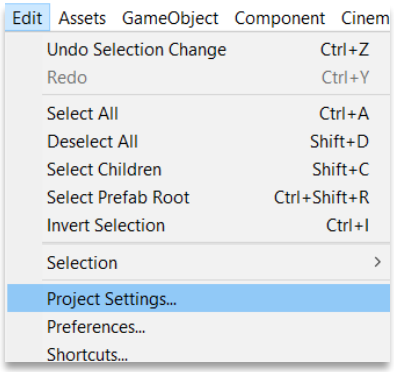
2 The Input object has several different functions to listen for Input from the player.
GetKeyDown, GetKey, and GetKeyUp each take a key's name as a parameter. This parameter can be a string or the KeyCode object.

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {

    }

    if (Input.GetKeyDown(KeyCode.DownArrow))
    {

    }
}
```

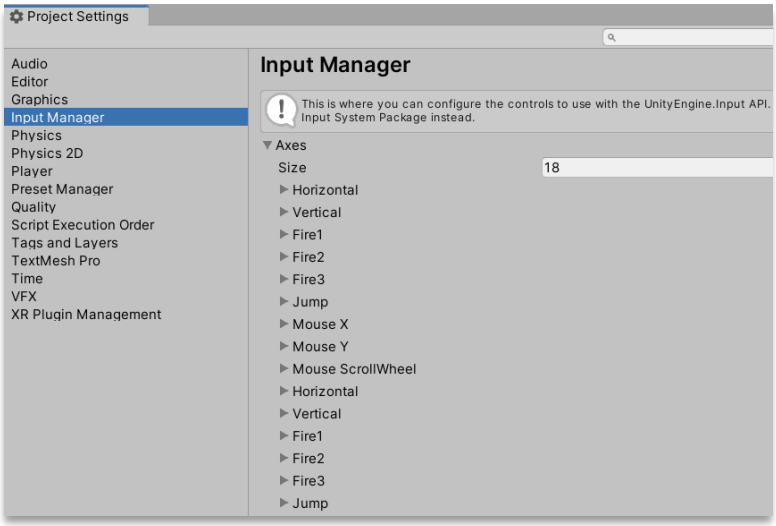
After you type the period after KeyCode, you will see all the key value options.

GetKeyDown will return true on the frame the user presses the key down.
GetKey will return true each frame the key is held down.
GetKeyUp will return true on the frame the user releases the key.

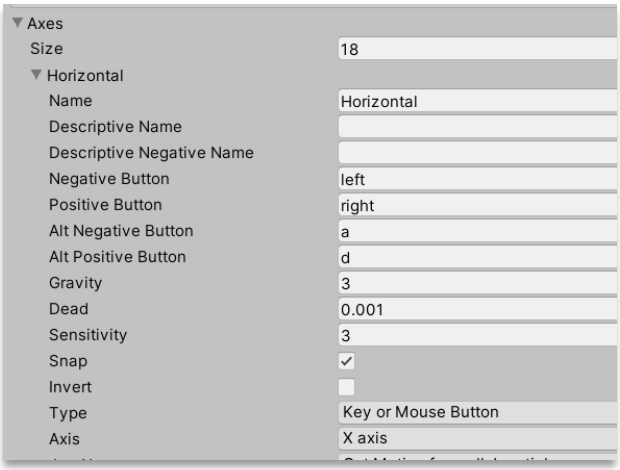
3 GetButtonDown, GetButton, and GetButtonUp take a button's name as a string parameter. You can view and edit the button names in the Project Settings.



Then select the Input Manager to view the input options that you have.



4 To see the exact key values that each Input requires, click on the arrow next to one of the options.



While buttons and keys have two values, just on and off, the Horizontal and Vertical Axes can produce a range of values from -1.0 to 1.0. To account for this, each Axis has a negative button and a positive button.

The Alt Buttons let you define a second key or button.

5 Since the Axes have a range of possible values, you must store the value of the function with a float variable.

```
public float horizontalValue;
0 references
void Update()
{
    horizontalValue = Input.GetAxis("Horizontal");
}
```

6 If you use GetAxis with the keyboard, Unity will return a small value like 0.05f at first. If the key is held, the value will increase slowly to 1.0f (or -1.0f for the negative button).

If you want to get only -1.0f, 0.0f, or 1.0f, then use GetAxisRaw. This function will return only whole number values.

```
public float horizontalValue;
0 references
void Update()
{
    horizontalValue = Input.GetAxisRaw("Horizontal");
}
```

Examples

Brown Belt's Find the Exit uses the GetAxisRaw function to move Codey around the maze.

11f Codey will still ignore user input because we haven't told Codey to move based on the destination vector we just made. In the final line of `Update`, change `Vector3.forward` to `destination`.

```
void Update()
{
    float horizontal = Input.GetAxisRaw("Horizontal");
    float vertical = Input.GetAxisRaw("Vertical");
    Vector3 destination = new Vector3(horizontal, 0, vertical);
    transform.Translate(destination * speed * Time.deltaTime);
}
```

Red Belt's Sulky Slimes uses the three GetMouse functions.

16 Inside each of the **if statements**, put a **print** statement to check to see when each of these will run.

```
0 references
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        print("Click!");
    }
    if (Input.GetMouseButton(0))
    {
        print("Hold!");
    }
    if (Input.GetMouseButtonUp(0))
    {
        print("Release!");
    }
}
```

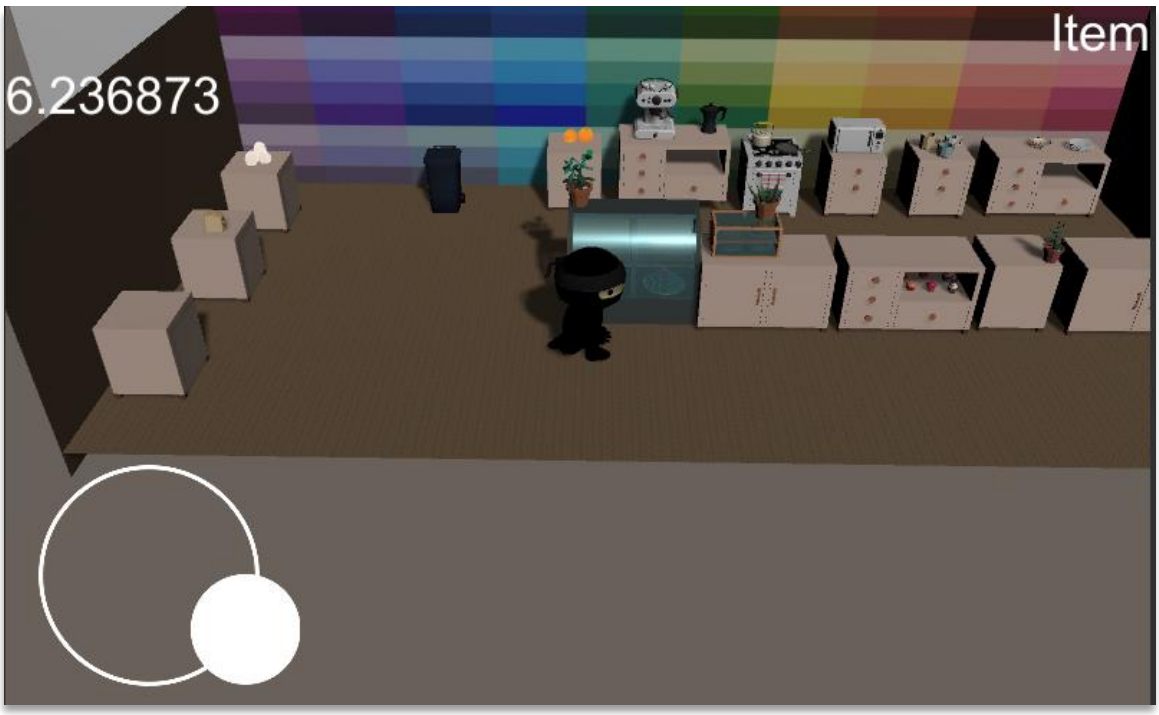
Ideas

Every single game will require player input. You should playtest your game with different control schemes to find the buttons and keys that feel the best.

You should use the Input object any time that you want your game to react to the player. This can be something like moving with the arrow keys, opening a door with a mouse click, or jumping with the space bar.

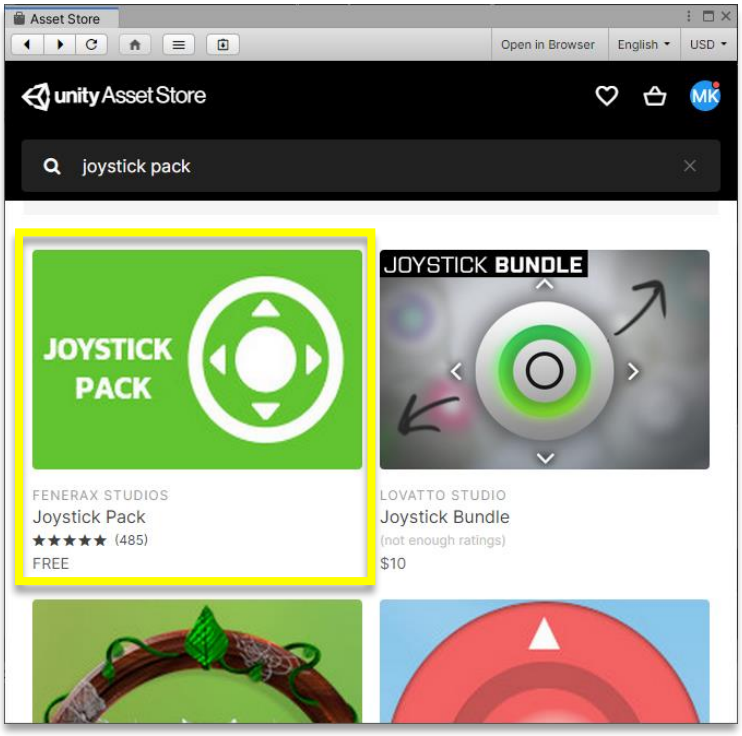
Movement Touch Controls

A virtual joystick can let your players move the characters using touch controls. While you can create your own, using a free asset from the Unity Asset Store can help you easily implement features.

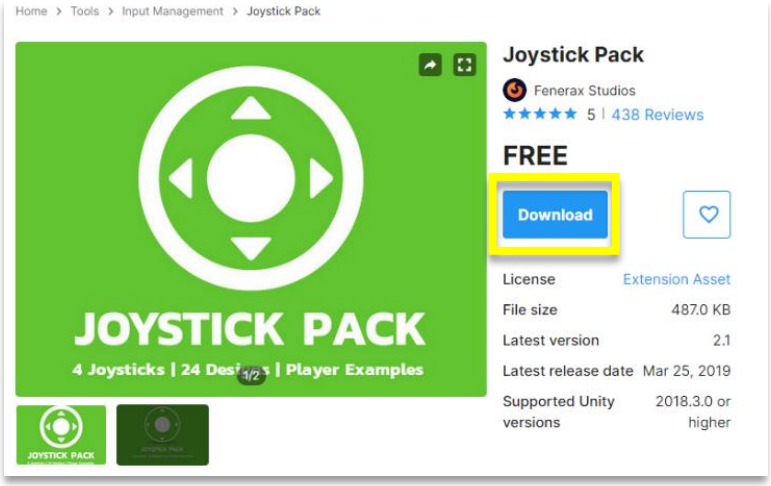


Instructions

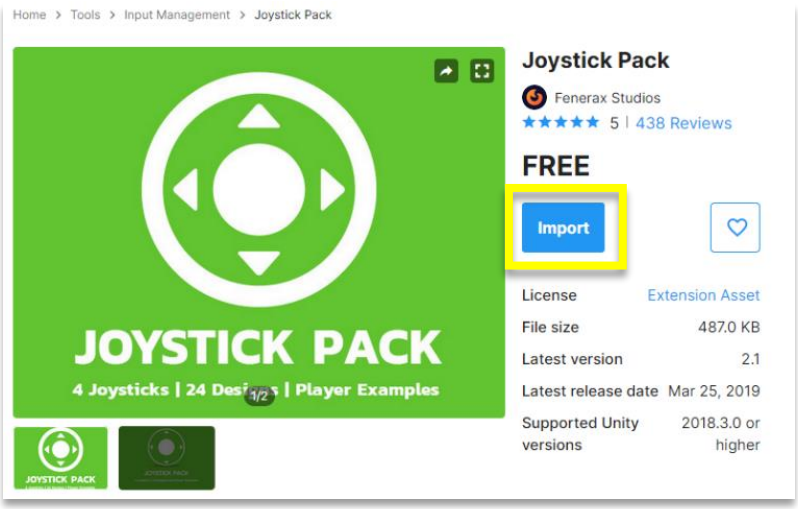
1 Open the Unity Asset Store by going to Window -> Asset Store. Search for "joystick pack" and find the free asset named Joystick Pack by Fenerax Studios.



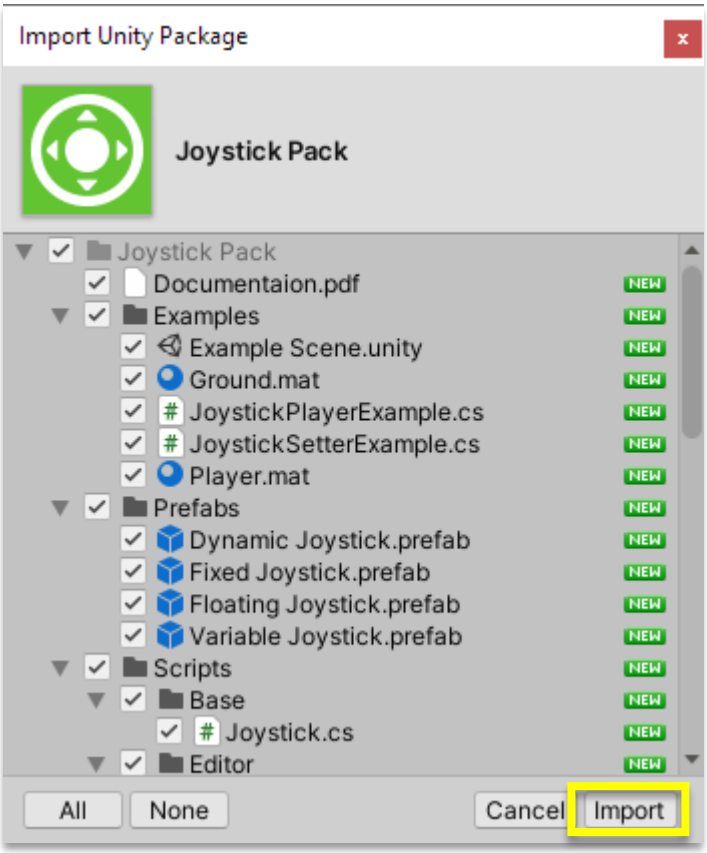
2 Click on Download to add the asset to your Unity account.



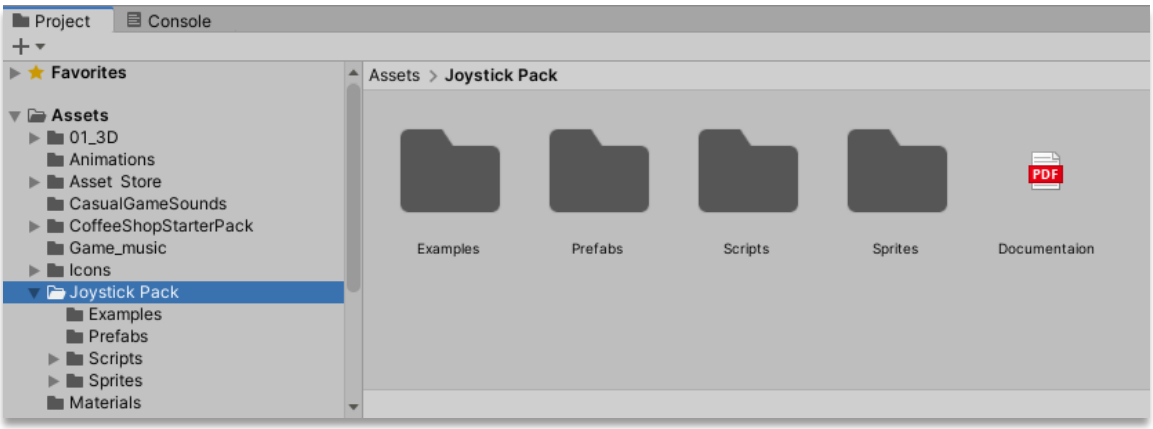
3 Click Import to add it to your current Unity project.



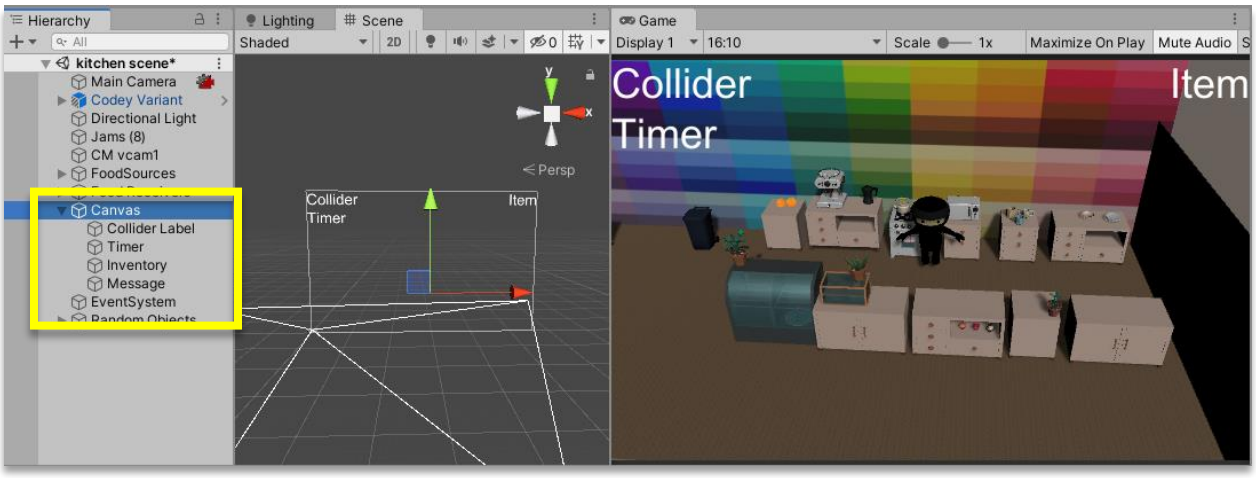
4 On the Import Unity Package window, click Import.



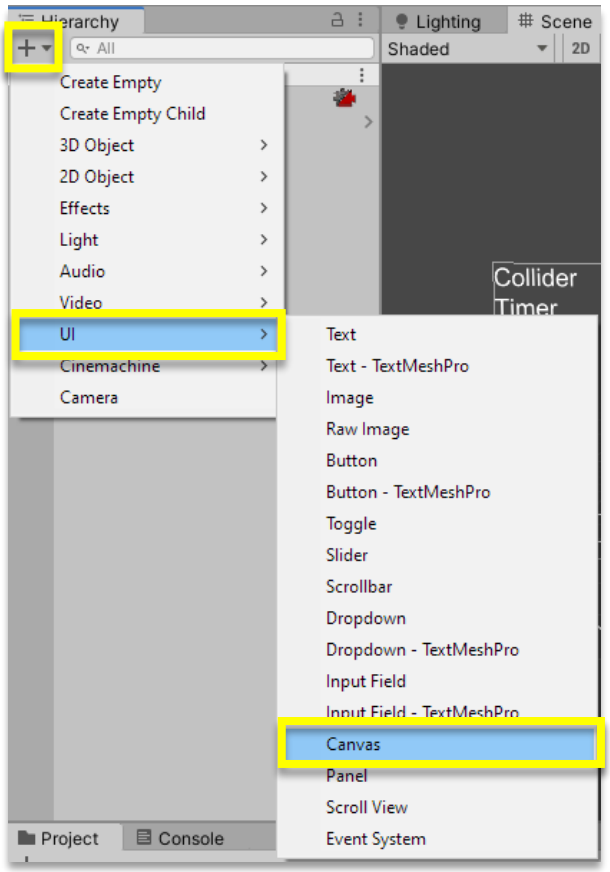
5 The Joystick Pack files are now in your game's Assets folder.



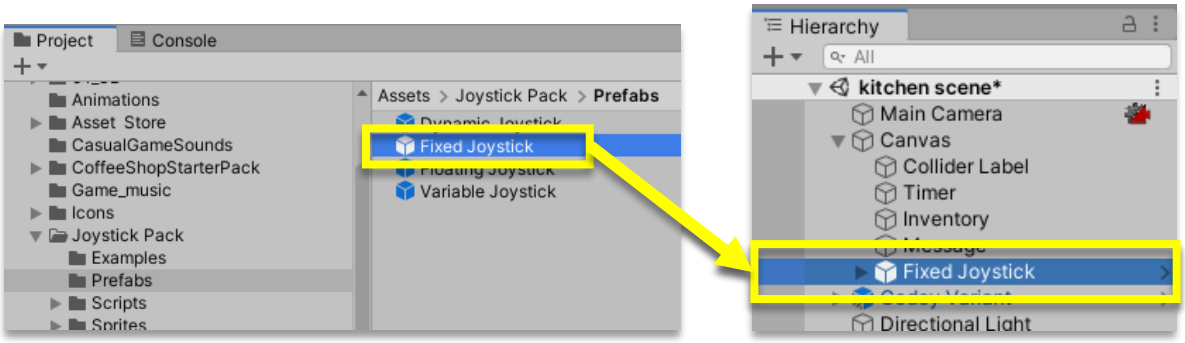
6 Since the onscreen joystick acts like a button, it needs to be placed inside of a canvas game object.



You can use an existing canvas or add one by clicking the Hierarchy's Plus button, then UI, then Canvas.

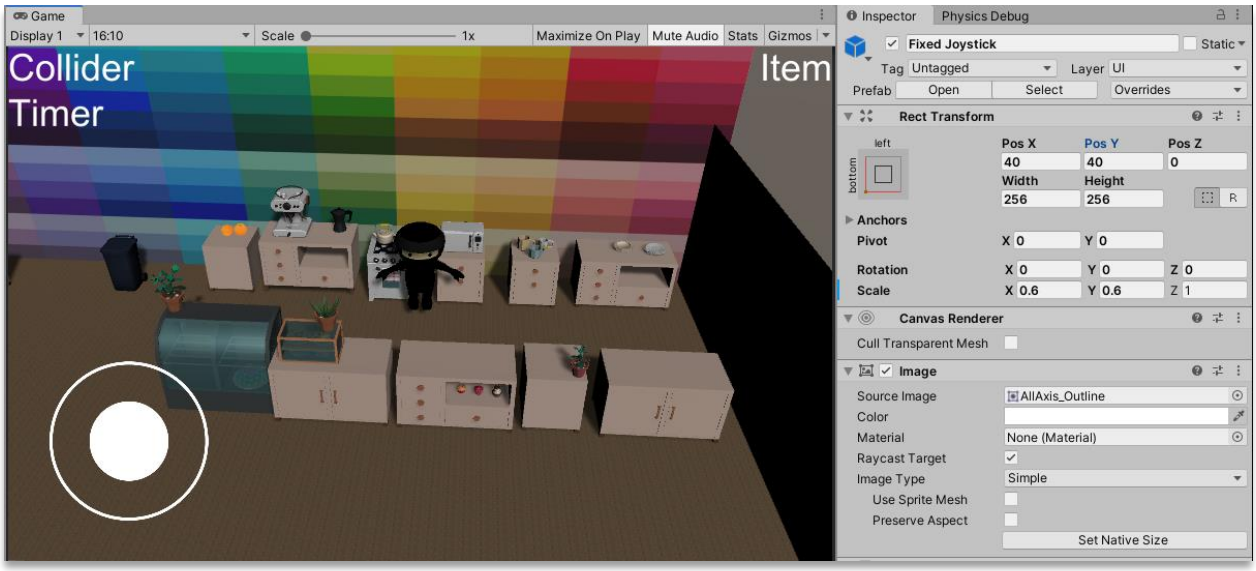


7 Find the Fixed Joystick prefab and place it inside your canvas object in the Hierarchy.



The Joystick Pack also has a dynamic, floating, and variable joysticks available. Look at the documentation and examples to see how you can use these prefabs in your game!

8 Use the Fixed Joystick's Rect Transform component to adjust the anchor, position, and scale of the virtual joystick.



Notice how changing the size of the game affects the joystick.

9 Play your game. Using your mouse, click and drag the joystick.

10 To have the joystick control the character in your game, open your movement script.

Add a new public Joystick variable and give it a name.

```
public class Movement : MonoBehaviour
{
    public Joystick joystick;

    [Header("Movement")]
    [Tooltip("Speed of movement")]
    public float speed;
```


11 Inside the Update or FixedUpdate function, create an if else statement that checks to see if the joystick is defined.

```
void FixedUpdate()
{
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");

    if (joystick)
    {

    }
    else
    {

    }
```

12 Before the if statement, declare your horizontal and vertical variables, but do not give them a value. We want to set the values based on the input type.

```
void FixedUpdate()
{
    float horizontal;
    float vertical;

    if (joystick)
    {

    }
    else
    {

    }
```

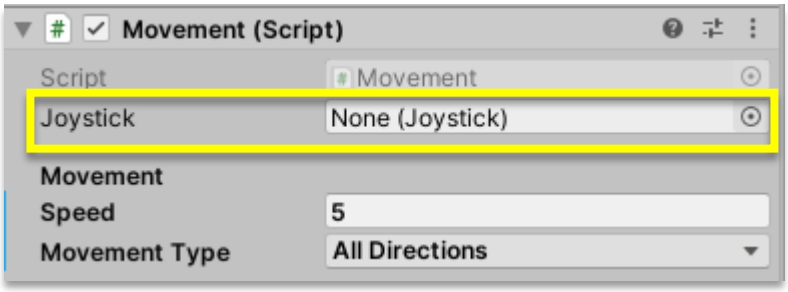
13 If the joystick is defined, then set the horizontal variable to the value of joystick.Horizontal and the vertical variable to the value of joystick.Vertical.

```
void FixedUpdate()
{
    float horizontal;
    float vertical;

    if (joystick)
    {
        horizontal = joystick.Horizontal;
        vertical = joystick.Vertical;
    }
    else
    {

    }
```

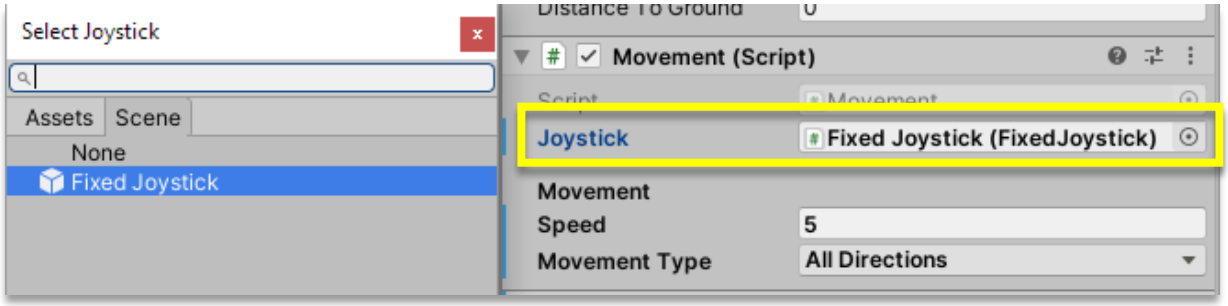
14 If the joystick is not connected, then set the horizontal and vertical variables to the value of Input.GetAxis.

```
void FixedUpdate()
{
    float horizontal;
    float vertical;

    if (joystick)
    {
        horizontal = joystick.Horizontal;
        vertical = joystick.Vertical;
    }
    else
    {
        horizontal = Input.GetAxis("Horizontal");
        vertical = Input.GetAxis("Vertical");
    }
```

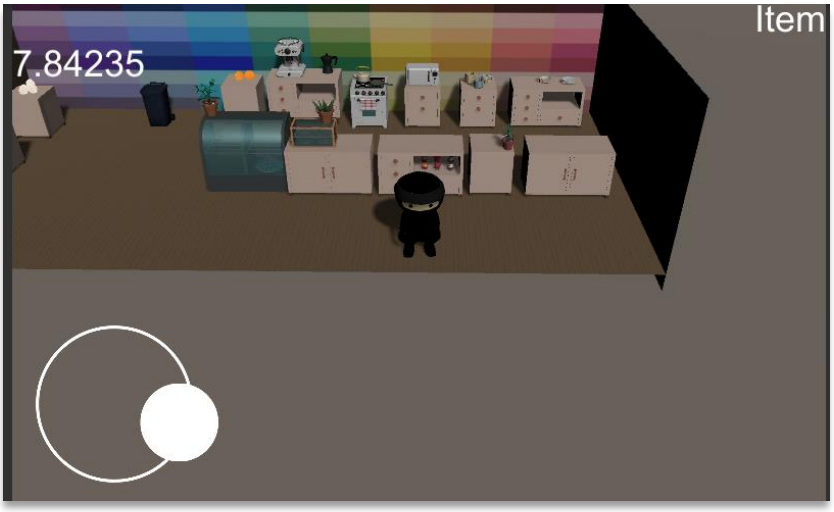
15 Save your script and play your game. Since we have not yet given the script the joystick object, verify that your movement code still works with the keyboard. Make sure you did not delete any variables or code that adjust your character's speed.



16 Set the value of Joystick to your Fixed Joystick by dragging the object from the Hierarchy or using the Select window.



17 Play your game. Verify that the joystick moves your character and the keyboard does not.



18 Based on your game's character and scripts, you might need to add joystick code in other locations.

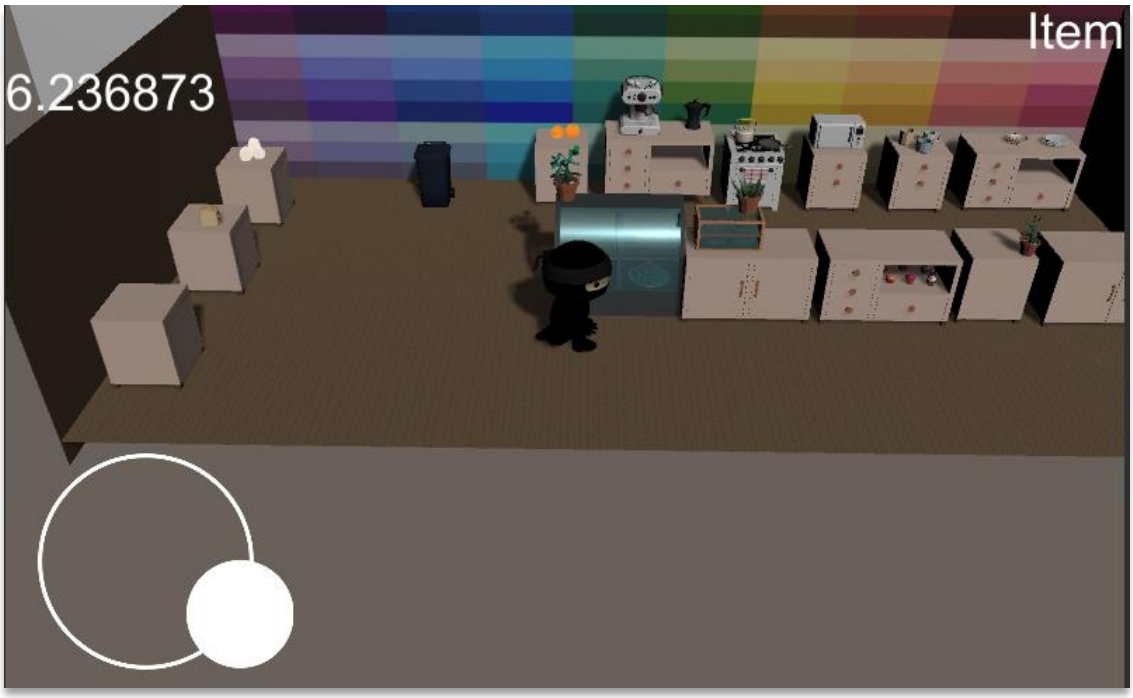
In Chef Codey for example, the character animations are defined in the Player Controller script.

```
void FixedUpdate()
{
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");

    Vector3 movementVector = new Vector3(horizontal, 0, vertical).normalized;

    if (movementVector.magnitude != 0)
    {
        lastLook = Quaternion.LookRotation(movementVector);
    }
```


19 Repeat steps 10 through 17 to add your joystick to any other scripts that use Input.GetAxis.

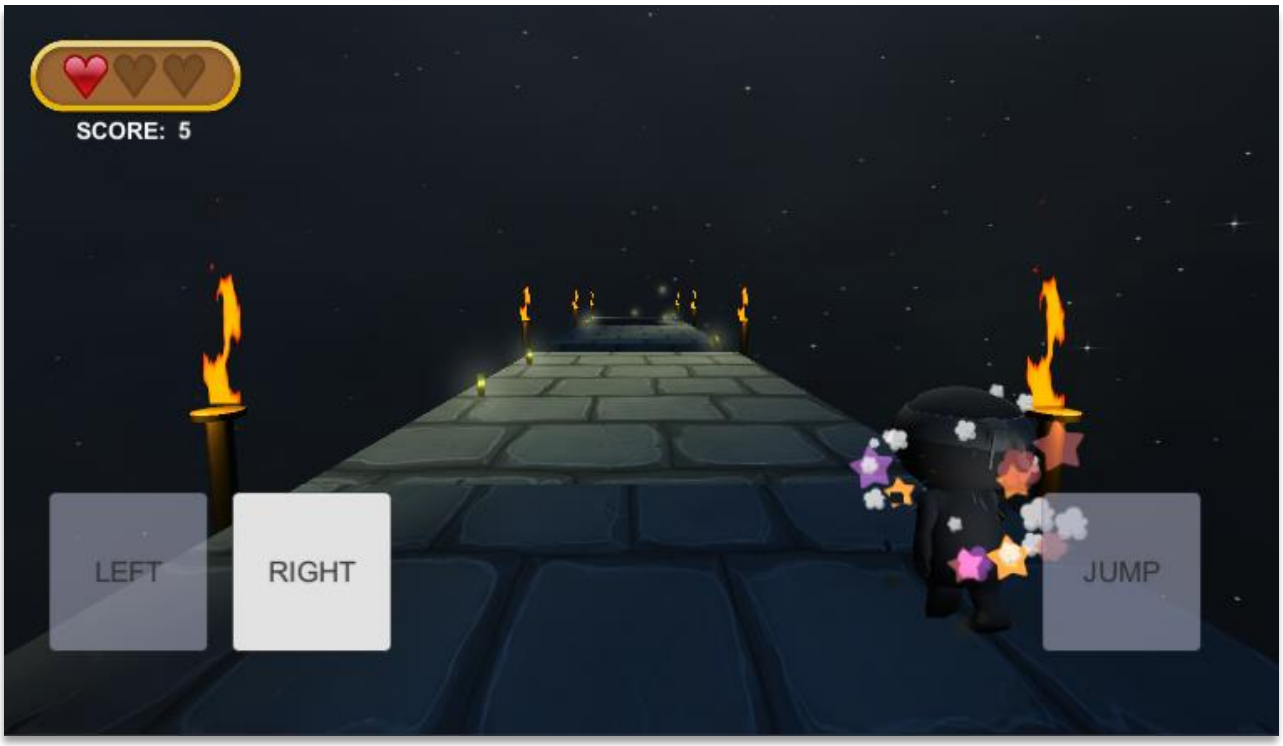

```
public Joystick joystick;
0 references
void FixedUpdate()
{
    float horizontal;
    float vertical;

    if (joystick)
    {
        horizontal = joystick.Horizontal;
        vertical = joystick.Vertical;
    }
    else
    {
        horizontal = Input.GetAxis("Horizontal");
        vertical = Input.GetAxis("Vertical");
    }

    Vector3 movementVector = new Vector3(horizontal, 0, vertical).normalized;

    if (movementVector.magnitude != 0)
    {
        lastLook = Quaternion.LookRotation(movementVector);
    }
```

20 Playtest your game. Verify that the feature you updated works properly with the joystick.



Action Touch Controls

You can add touch controls to your game by creating a simple UI with buttons for each action.



Instructions

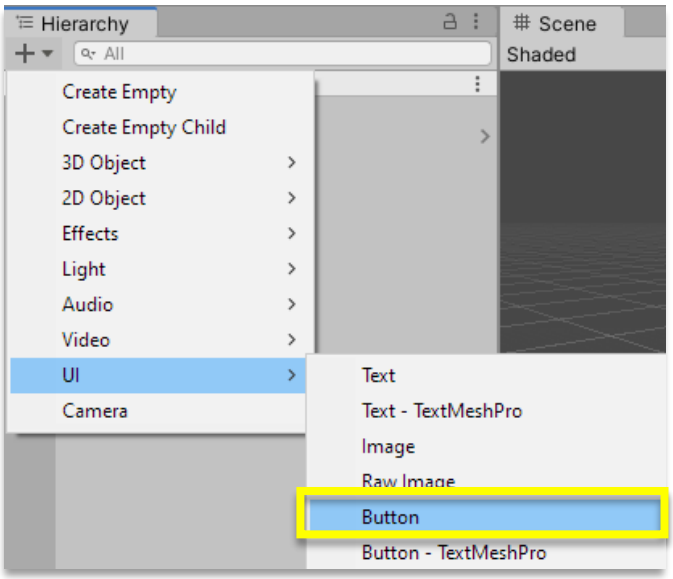
1 This example will modify Brown Belt's Ninja Run game. We want to the user to tap to jump and move lanes.

2 Before you add a UI and Buttons, make sure you have functions for the actions you want to peform. For this game, we need Jump, Move Left, and Move Right functions. Your function names and code should be specific to your game.

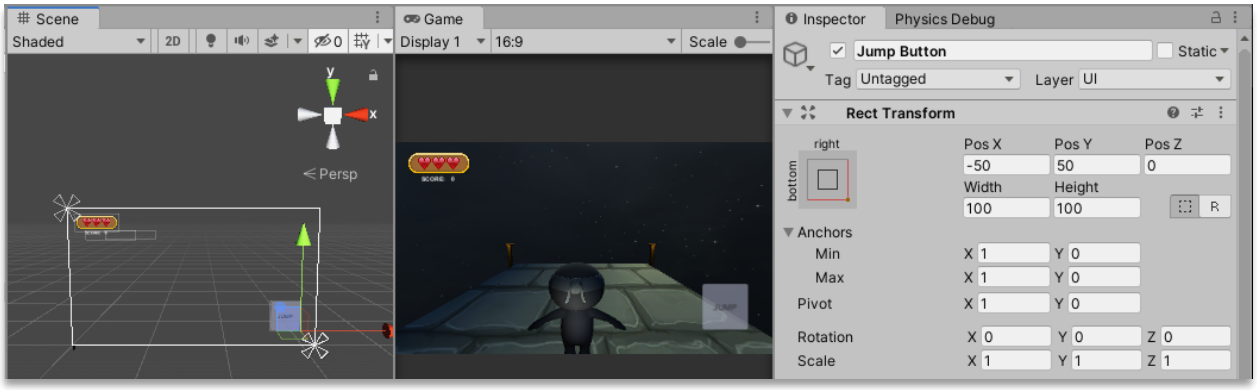
```
0 references
public void JumpButton()
{
    if (!isGrounded) return;
    rigidbody.velocity = Vector3.up * jumpForce;

    dirtParticles.Stop();
    dustParticles.Stop();
    StartCoroutine(particlePlay());
}
0 references
void Update()
{
```

```
2 references
public void MoveLeftButton() {
    moveL = true;
}
2 references
public void MoveRightButton() {
    moveR = true;
}
0 references
void Update()
{
```

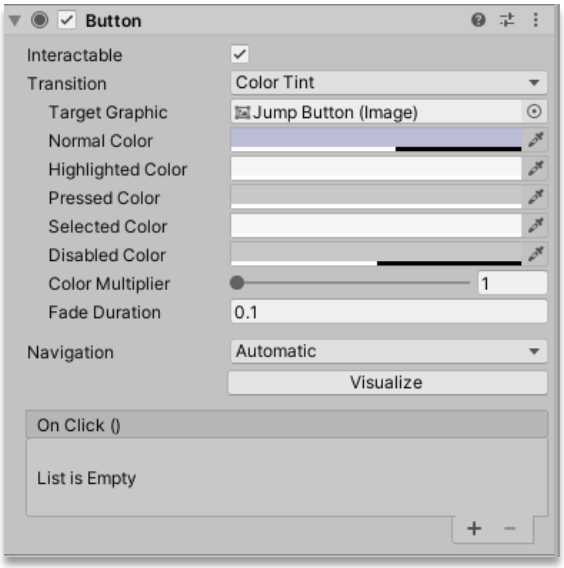
3 Add a Button to your Scene.



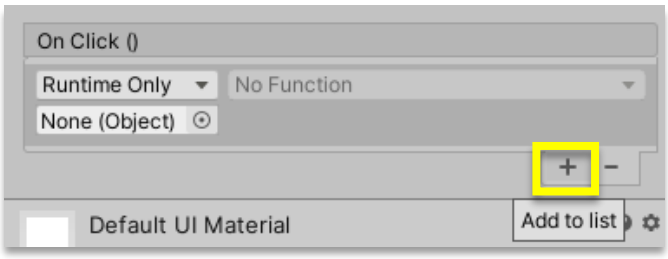
4 Position the button and change its text and any other visuals. Think about how the player will use the button on their touch screen.



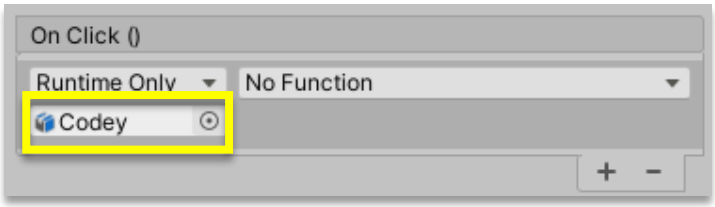
5 Find your Button's Button component in the Inspector.



6 Add an action to the On Click () section by clicking the plus button.

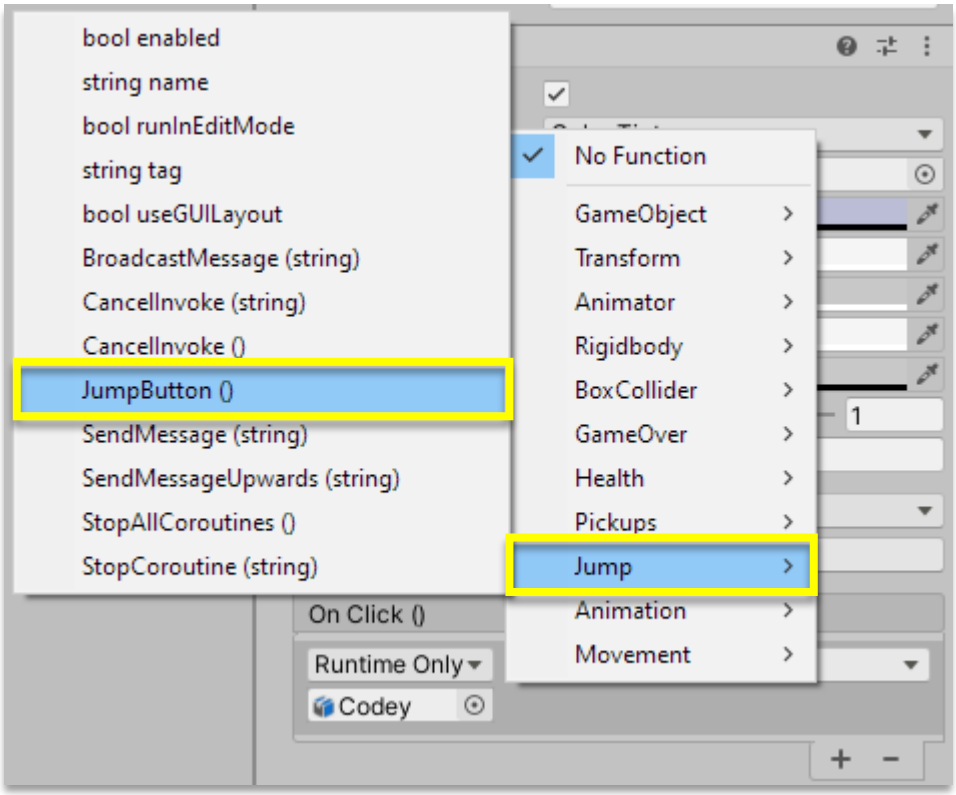


7 Find your object that contains the function you want to use and drag it from the Hierarchy to the None (object) slot of the On Click () section.



You need to use the object with the script and not the script itself because it needs to have an object from the Scene and not an Asset.

8 With the correct object attached, click the No Function dropdown. Find the script and the function you want to trigger when this button is pressed.



9 Verify that you selected the correct function.

10 Playtest your game and click your button. Make sure that the game is behaving how you expect it to!

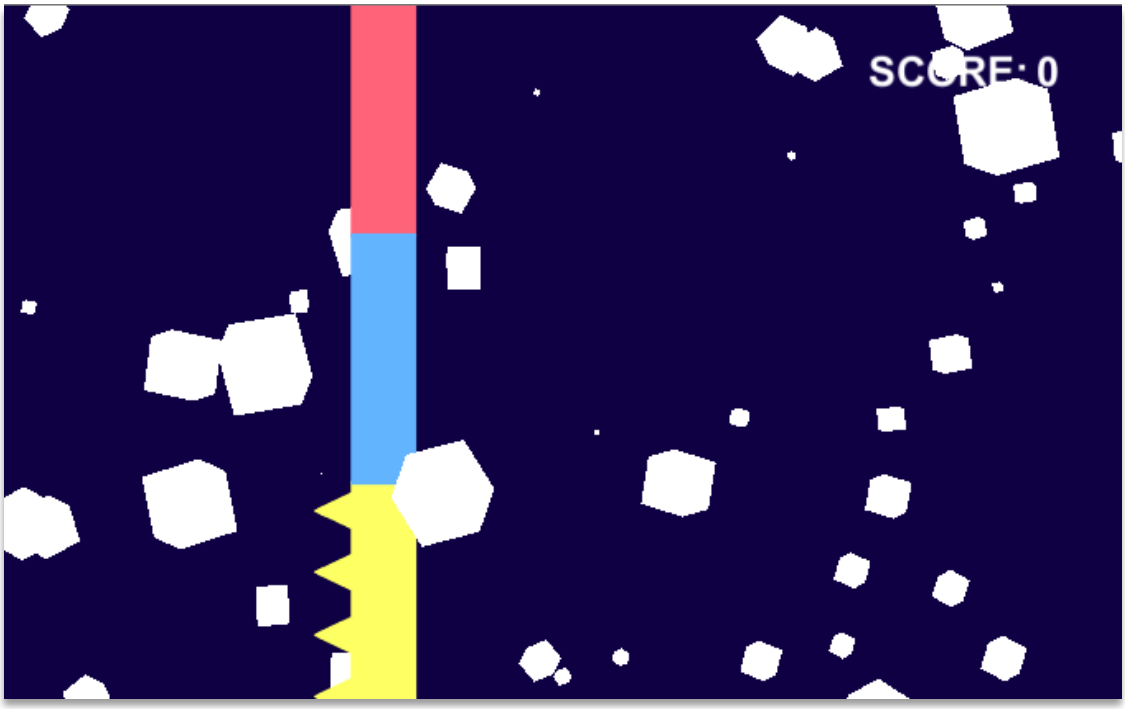
11 Repeat steps 2 through 10 to add buttons for your other actions.

Ideas

You can use UI Buttons to run any function in your game. If you plan on touch being the primary way to play your game, then you need to create buttons for every action required. You can use a script to manage when to enable and disable different buttons.

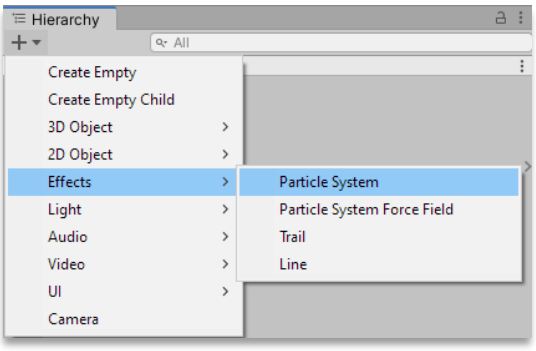
Particles

Particle effects are a way for you to easily add visual flair to your game. They can be used to convey speed, danger, success, and many other concepts to your player.

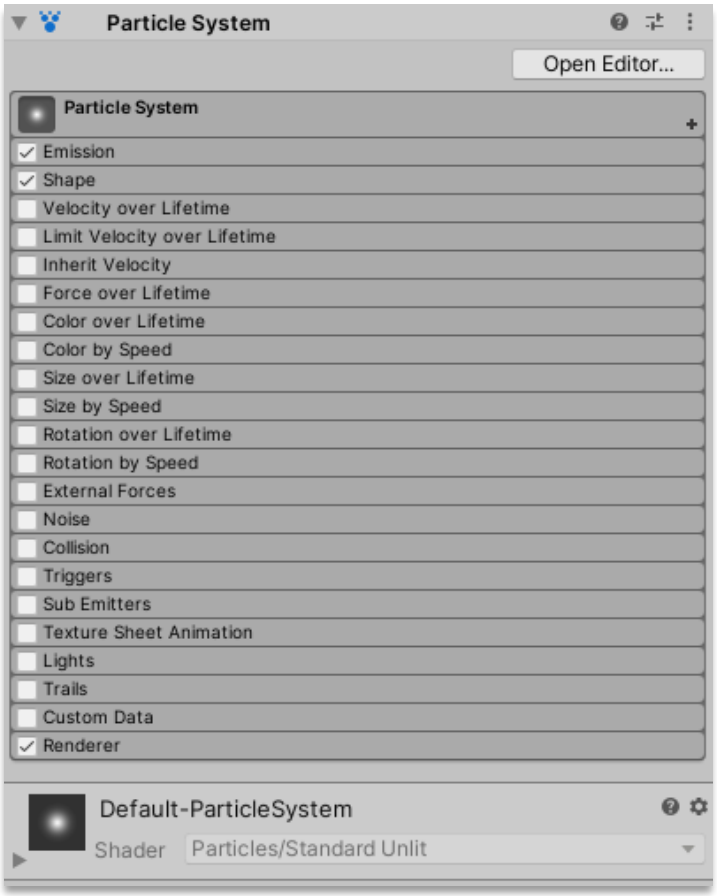


Instructions

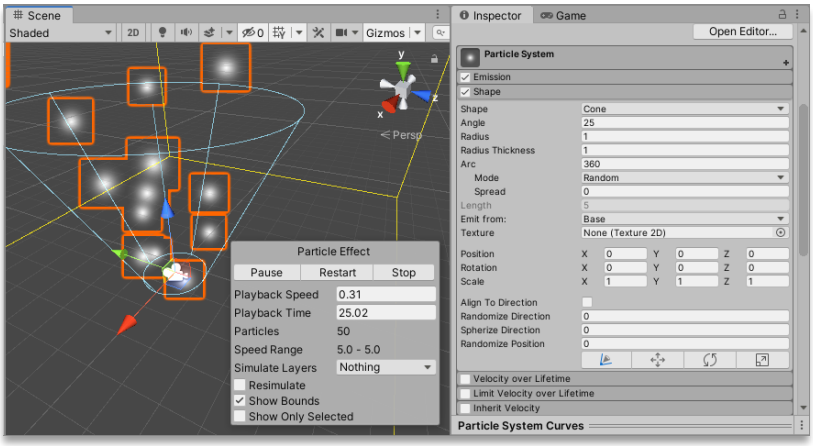
1 A Particle System is a special game object that exists in your scene. To create a new Particle System, right-click an empty space in the Hierarchy, select Effects, and then Particle System.

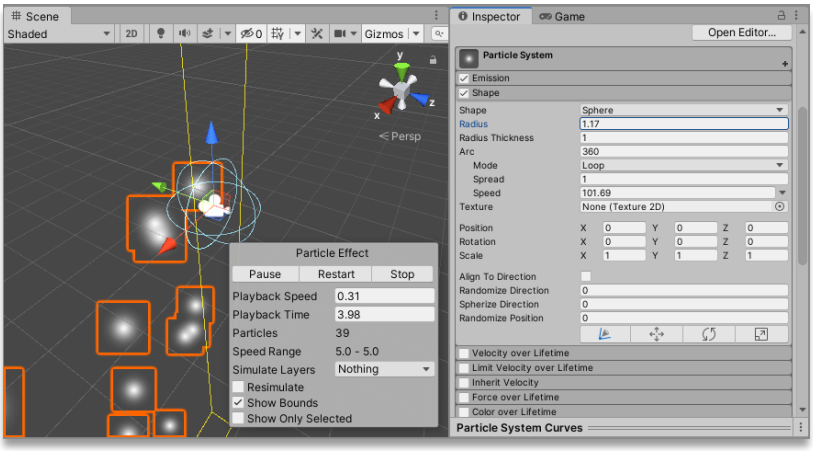


2 The game object that is created has a special Particle System Component with many different properties.

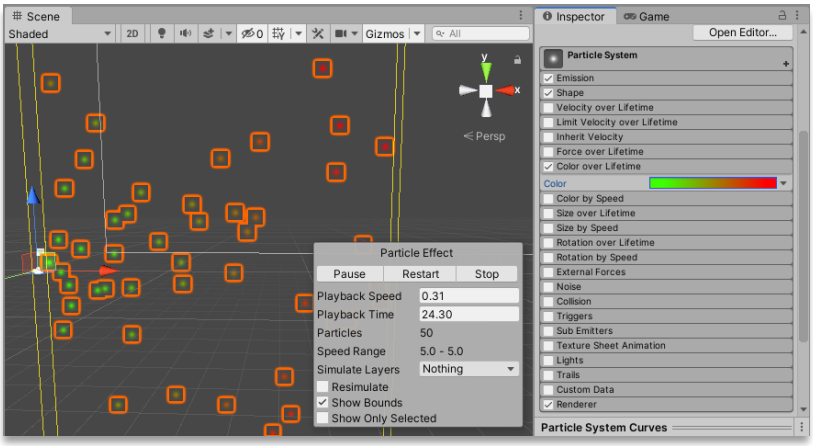


3 Each of these properties controls the behavior of your particle system. For example, changing the shape will modify how your particles spread out from the game object.

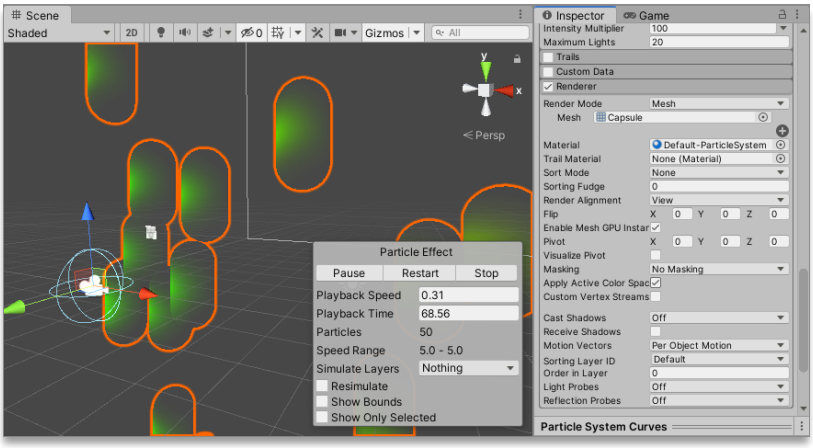


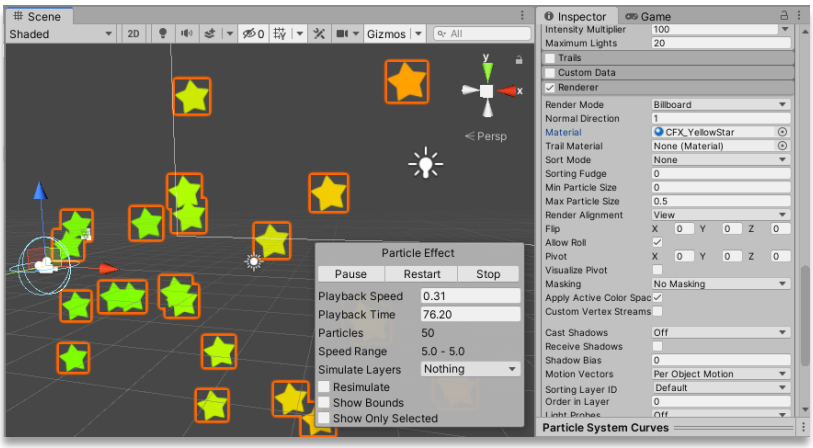


4 Don't be afraid to experiment with each property to see what it does. Your options are almost limitless!

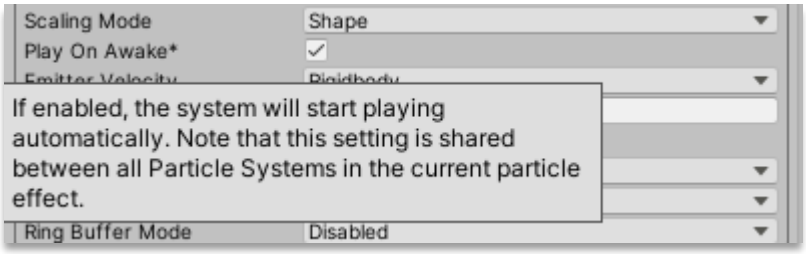


5 You can use a custom mesh object or sprite image by changing the Renderer property's Render Mode.



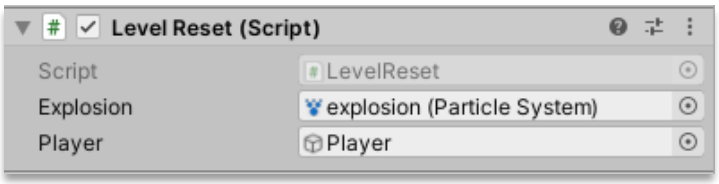


6 By default, particle effects play when the game starts. You can change this by disabling the "Play On Awake" property.



7 There are two simple functions that let you control your particle system. First, you must create a public variable and attach the game object to your script in the Inspector.

Then, you can use the Play function to play your particle effect, and the Stop function to stop your particle effect.




```
public ParticleSystem explosion;
2 references
public void GameOver()
{
    explosion.Play();
    player.SetActive(false);
    Invoke("Reload", 2);
}

0 references
void Reload()
{
    explosion.Stop();
    SceneManager.LoadScene(0);
}
```

Examples

In Chef Codey, particle systems were used to communicate when food was cooking and when it was ready to be picked up by the player.

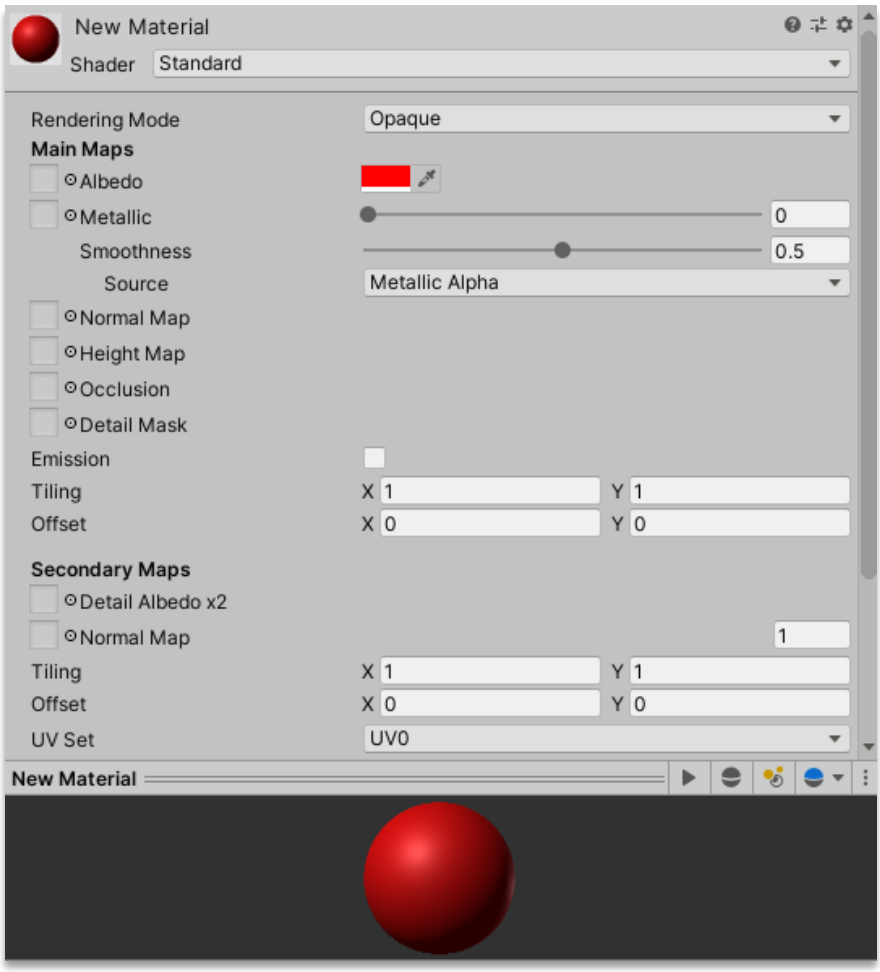
121 Playtest your game and see what happens when you try to make toast or fry an egg.

Ideas

Particle systems can be used in many different ways. You can use a gold glowing particle system to indicate a special key, or add a particle trail to an object that is moving across the scene to give it a sense of speed.

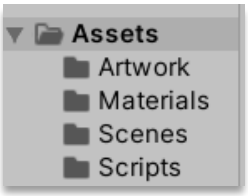
Materials and Textures

Materials can be added to game objects to change how they appear in the scene. A Mesh defines a game object's shape, while a Material defines a game object's appearance. A Texture lets you wrap an image over your game object's surface.

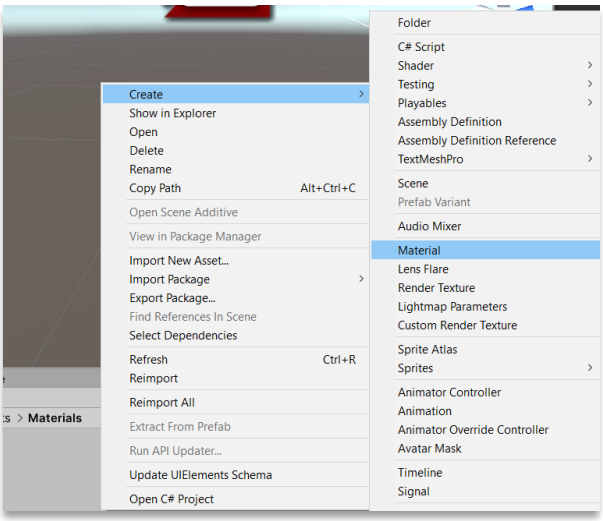


Instructions

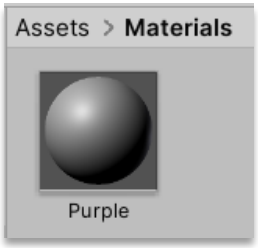
1 To keep things organized, create a folder called Materials to store any Materials that you create or download.



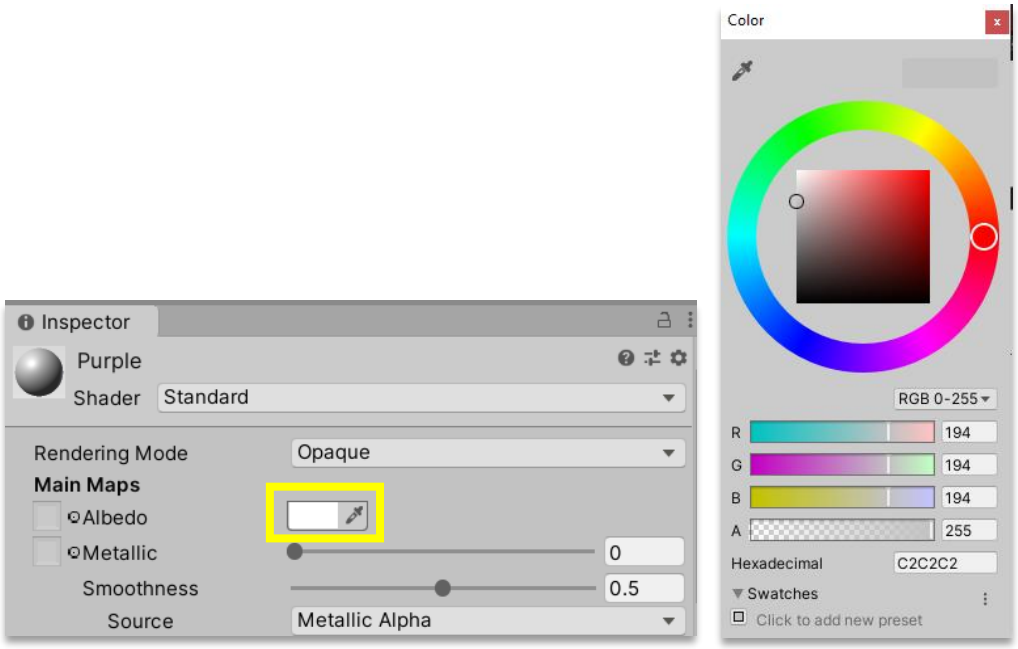
2 You can create a material by right clicking in your desired folder and selecting Create, then Material.



Remember to rename the new Material.

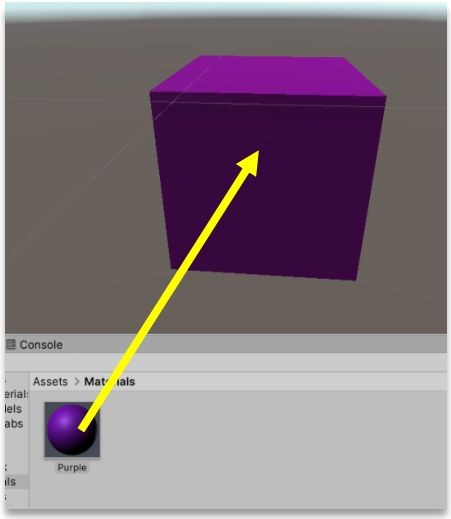


3 To change the color, select the material and look in the Inspector. Click on the rectangle next to Albedo, it will open a Colors tab.

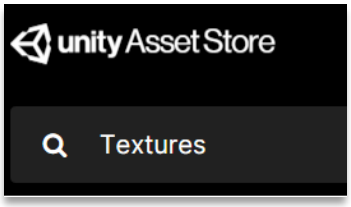


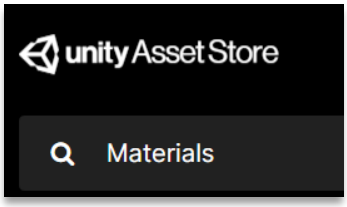
If you're wondering, Albedo is a fancy word that describes how much light a surface reflects!

4 Drag the material onto a game object.



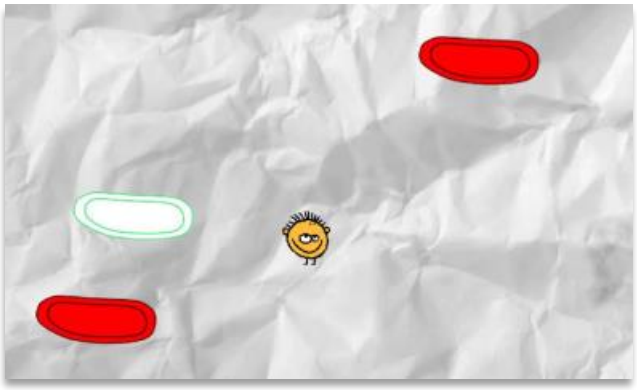
5 You can also find textures with different patterns and colors in the Unity Asset Store.





Examples

In Purple Belt's Sketch Head Prove Yourself activity, you were challenged to create a material to make the fake platforms stand out so the player does not land on the platforms without colliders.



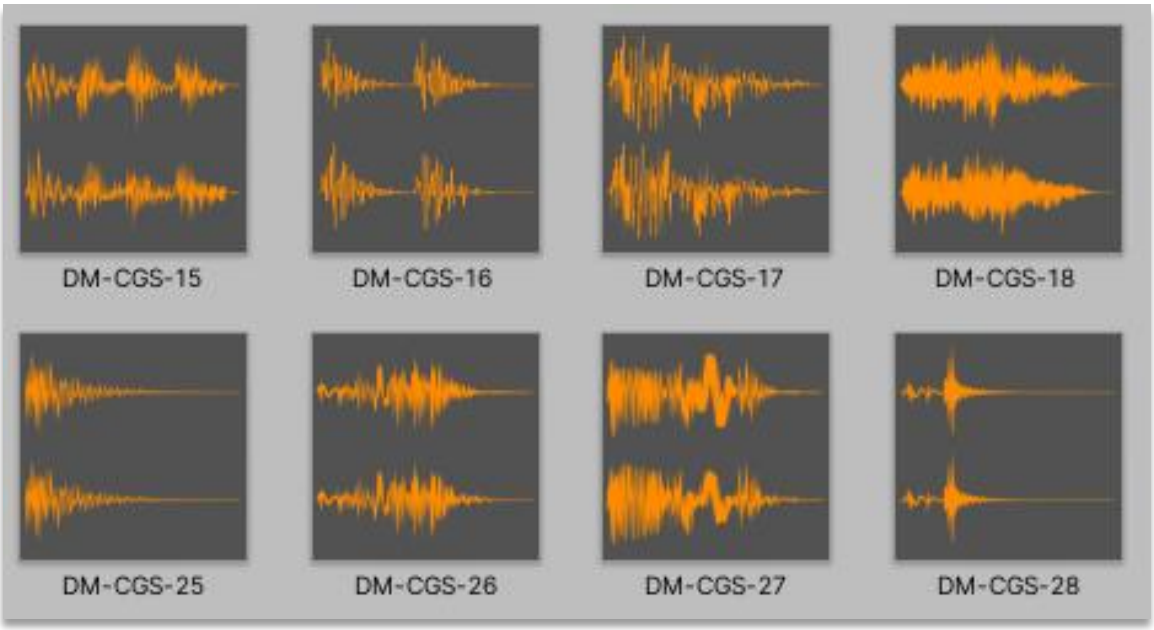
Ideas

Use Materials and Textures to turn your basic cubes and squares into things like bricks and planks.

Not every game object has to look like the real world. Get creative when applying textures – a tree doesn't have to be made of brown wood!

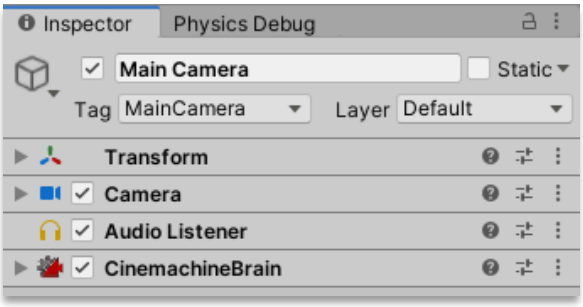
Sound and Music

Adding sound and music to your game will make it feel polished and complete. You can use music to set the mood of the scene, and sound effects to immerse the player in your environment.



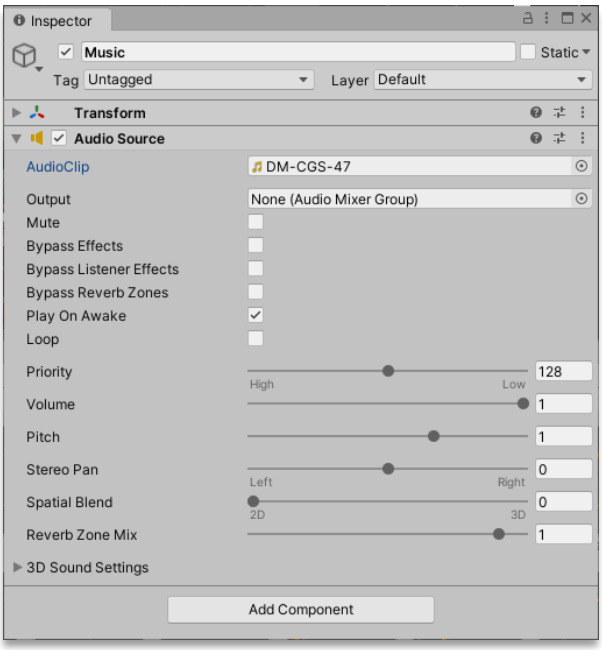
Instructions

1 Before you can hear your sound in your game, you need a game object that has the Audio Listener component.

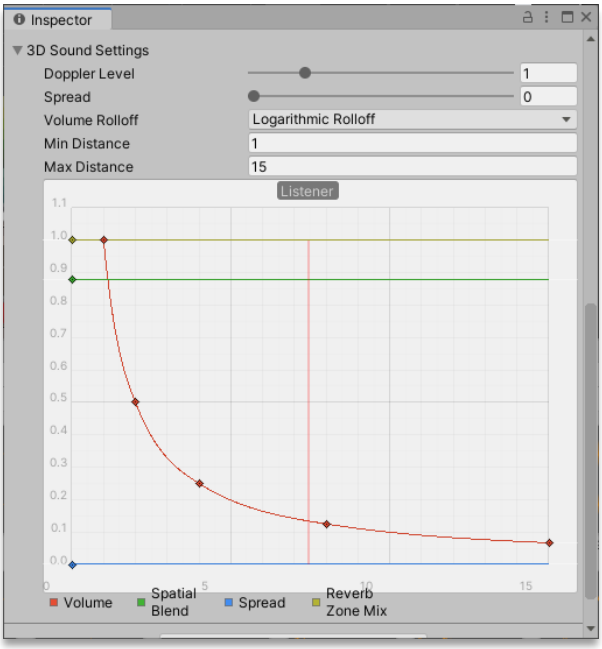


While this can be placed on any object, Unity's Main Camera has one by default. This component has no properties or settings.

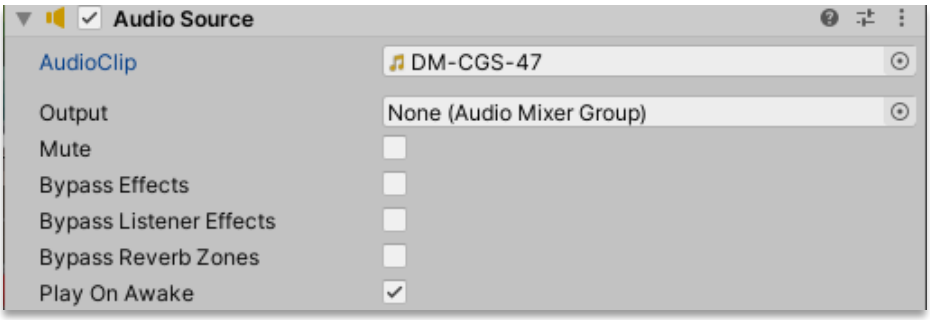
2 To play music in your scene, you need a game object that has an Audio Source component. You can use the Unity Asset Store to find music and effects to use as your Audio Source's AudioClip.



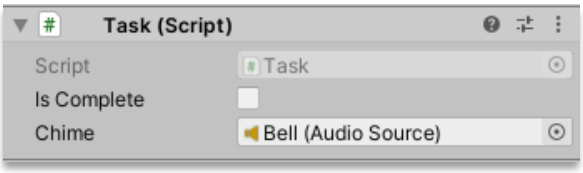
3 Unity uses Spatial Audio to adjust the volume of the sounds based on how far the Listener is from the Source. You can control how the interaction behaves by modifying the 3D Sound Settings.



4 If you want to use a sound effect when a player performs an action, you follow the same steps to create an Audio Source, but make sure to disable the "Play On Awake" option.



5 Your script needs to have a public AudioSource object that you connect to your game object in the Inspector. In the script, you can use the Play function to play your sound effect or music clip.




```
public class Task : MonoBehaviour
{
    public bool isComplete = false;
    public AudioSource chime;
    0 references
    public void CompleteTask()
    {
        chime.Play();
        isComplete = true;
    }
}
```

6 Other useful AudioSource functions include Pause, UnPause, and Stop.

```
public void DJ()
{
    music.Play();
    music.Pause();
    music.UnPause();
    music.Stop();
}
```

Examples

In the Evil Fortress of Dr. Worm, sound effects and background noises were used to make the player feel like they were trapped in an evil lair!

20c We need to add a Boolean to the LaserSwitch script that we can use to inform the laser objects to turn off when Codey steps on the switch.

After `private AudioSource playBeep;` type `public bool lasersAreOff = false;` to create a Boolean with a value of false.

```
private GameObject switchIcon;
private AudioSource playBeep;

public bool lasersAreOff = false;
```

20d We want to turn the lasers off when Codey or the box enters the switch. In the `OnTriggerEnter` function, add `lasersAreOff = true;` after the `playBeep.Play();` line.

```
public void OnTriggerEnter(Collider other)
{
    playBeep.Play();
    lasersAreOff = true;
}
```

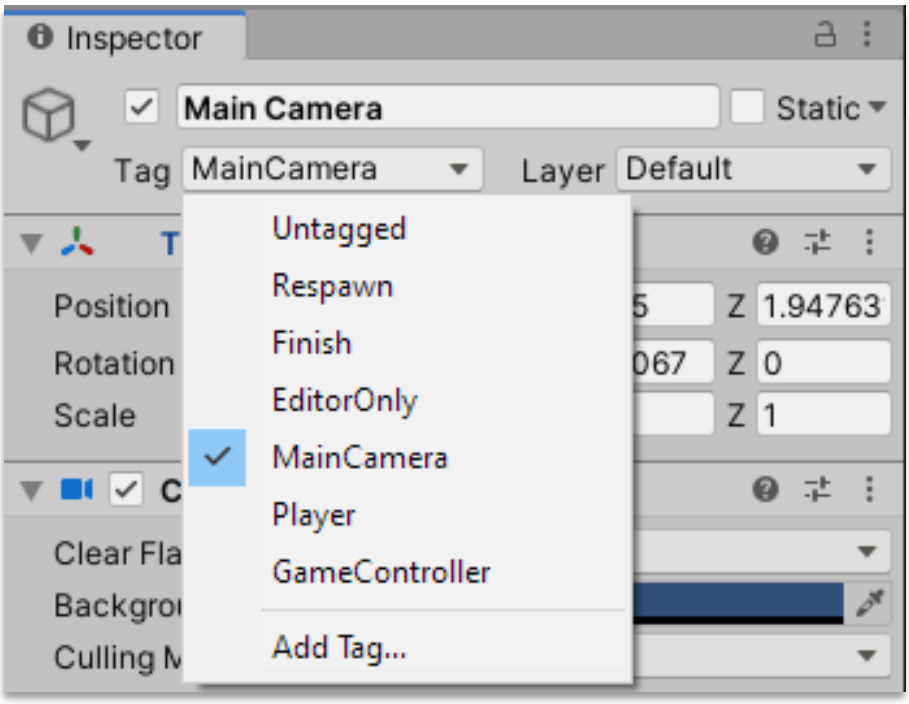
Save your script.

Ideas

Adding sound effects to your game helps the player understand what's going on in the scene. Some examples are a beeping that indicates a time limit or a chime that plays when a door is unlocked. Background music can be used to help transport your player into the setting. For example, a spaceship might have electronic music, while a forest might have soothing pianos.

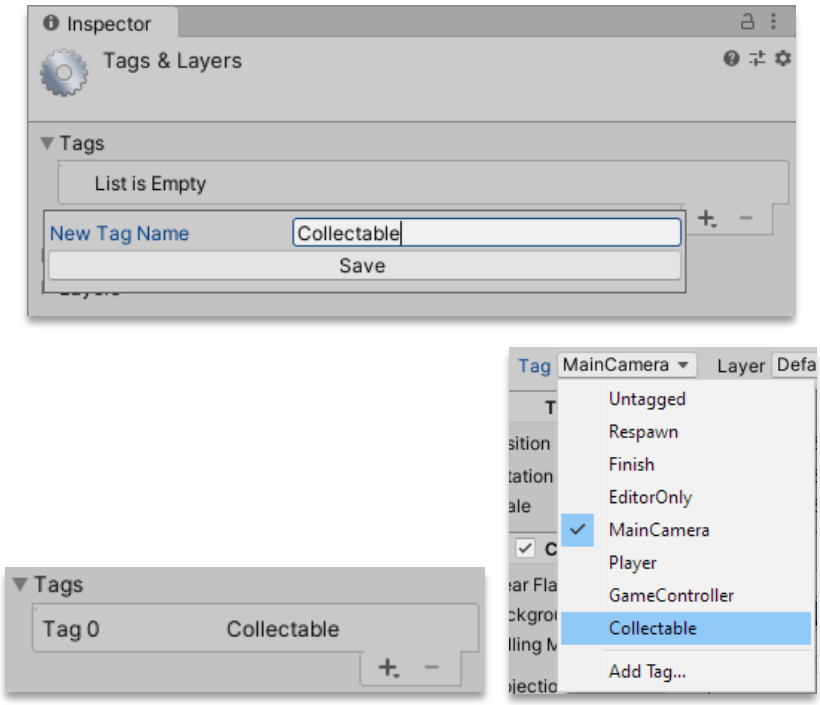
Tags

Tags let you label your game objects so you can find and compare by tag instead of relying on a name, type, or component. They make handling collision easy by letting you ignore all objects except the ones you want to interact with.



Instructions

1 Unity has several default Tags, but you can create your own by selecting "Add Tag..." and adding one in the Tags and Layers menu.



Once created, the tag can be assigned via the Tag drop-down menu in the Inspector.

2 If you need to find a game object based on its tag, you can use the GameObject.FindGameObjectWithTag function to return the most recently edited object with this tag. It is important to remember that if you have more than one object to find, you can't reliably predict which one you will get!

```
public void FindTaggedObject()
{
    GameObject coin = GameObject.FindGameObjectWithTag("Collectable");
}
```

3 If you want to find all objects with a specific tag, you can use the GameObject.FindGameObjectsWithTag function. This will place all the found objects in an array. You can then loop through the array and perform actions on each of the objects.

```
public void FindTaggedObjects()
{
    GameObject[] coins = GameObject.FindGameObjectsWithTag("Collectable");
}
```

4 To limit the objects that you can interact with, you can use a game object's tag property or the CompareTag function inside a Trigger function.

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        count.triggeredCheckpoints++;
    }
}
```

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Player")
    {
        count.triggeredCheckpoints++;
    }
}
```

Examples

Codey Raceway used tags to control the logic of giving Codey a power up.

166 By default, **chosenPowerup** does not have a Game Object attached. After we collide with the itemBox, we want to use **randomNumberInList** to assign a Game Object from the list to the **chosenPowerup** variable.

```
private void OnCollisionEnter(Collision other)
{
    if (other.gameObject.tag == "itemBoxes")
    {
        randomNumberInList = Random.Range(0, powerupList.Count);
        chosenPowerup = powerupList[randomNumberInList];
    }
}
```

Robomania uses tags to make the enemy jump every time it collides with the ground.

36d We need to tell Unity to apply this force to the Crusher's **rigidbody component** by typing `enemyRigidBody.AddForce(jumpForce);`.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Ground")
    {
        enemyRigidBody.AddForce(jumpForce);
    }
}
```

Ideas

Tags can be applied to any game object, so they are a great way to control the logic of objects colliding with each other. Tags are also useful when you need to find similar game objects, but you don't know their names or cannot easily connect them to a public variable in the Inspector.

C# Concepts

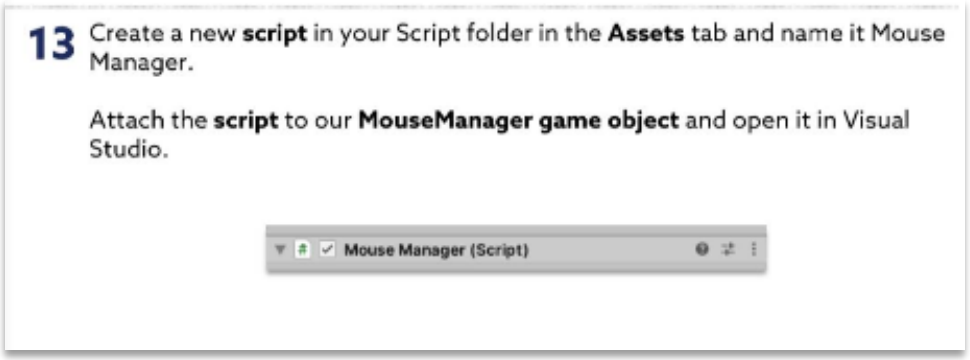
Classes

A Class is used to organize objects and code together based on functionality or purpose. You can design your game objects to have one large class that controls all the logic, or you can create multiple small classes that can be applied to different game objects.

```
0 references
public class SelfDestruct : MonoBehaviour
{
    public float countdown;

    0 references
    void Update()
    {
        countdown -= Time.deltaTime;
        if (countdown < 0)
        {
            Destroy(gameObject);
        }
    }
}
```

Examples

In almost every game you are creating a script that does a specific job. In Red Belt's Sulky Slimes, you created one class containing all the logic required for the player to control the game.



In Red Belt's Codey Raceway, you created a class that controlled the movement of your shell game object.

Ideas

You should use scripts and classes to break up all the different actions that your objects can take. Some classes might be specific to one game object, while others can be attached to almost anything!

Variable Types

All variables in C# need to have a Type. A variable's type determines what kind of data it can store and how it can be used.

```
public GameObject collectable;
public bool isLocked = true;
public int score = 42;
private float distance = 37.1f;
private string message = "The door is locked";
```

Unity has a lot of unique variable types like GameObject, Sprite, and Collider. The values of these variables can easily be set in your script's Inspector.

A bool is a Boolean variable that can be either true or false. These are used in conditionals to control the flow of your game.

An int is a whole number that contains no decimal values. It is useful when you need to count things like the score of a game.

A float is a special type of number that can contain decimal values and ends with the letter f. Floats are used when values need to be calculated or measured.

A string is a how readable text is represented in code. Strings are used when you need to log a message to the console or write out something in your User Interface.

A private variable can only be accessed from inside its script. A public variable can be viewed in Unity's Inspector and used by other scripts.

Variables are used in every game in the Unity belts!

Floats

In C#, a number with a decimal value can be stored by using the Float data type. A float value must always end with the letter f.

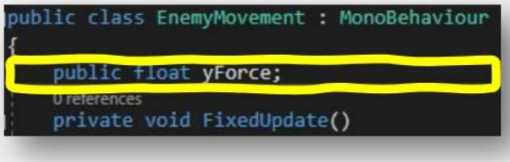
```
public float distanceFromDoor = 12.5f;
```

In the example above, the distance a player is from the door is represented by a float value because it could be a whole number or a decimal number.

Examples

In Brown Belt's Robomania, you created a float variable that determined how much force is applied to your Crushers.

32a We need to tell the **crusher** how high to **jump**. On the line after the open curly brace `{` type `public float yForce;` to create a **variable** that we can change in the **Inspector** tab.



Ideas

You should use Floats any time a value can be between whole numbers. If you can't use whole numbers to count your variable, then you should use a Float.

Functions

A function in C# works like any other function that you have created or used before. Functions allow you to reuse code multiple times without having to retype or copy and paste it over and over again. Functions keep your code clean and easy to read. When you call a function, you are running all of the code that is inside of it.

In the example below, when the function is called it will do two things: remove the enemy and increment our score variable.

```
private int myLevel = 0;
0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    LevelUp();
}
1 reference
private void LevelUp()
{
    myLevel++;
    SceneManager.LoadScene(myLevel);
}
```

Instructions

1 Every C# Function is made up of five parts: its visibility, return type, name, parameters, and body.

First, you must decide if your Function is Public and can be called from any script, or if it is Private and can only be called from its own script.

```
0 references
public void myPublicFunction()
{

}
0 references
private void myPrivateFunction()
{

}
```

If you do not declare your function to be Public or Private, Unity will treat it like a Private function.

2 Every Function runs code, but sometimes you need to calculate a result. C# needs to know that kind of variable your function will return.

The most common return type is void. This means that your Function will not return a value after it is called. Void Functions often modify existing variables rather than create new ones.

```
public int myScore;
public GameObject collectable;
0 references
private void OnCollisionEnter(Collider other)
{
    collectable.SetActive(false);
    myScore++;
}
```

If your function calculates a value or creates a variable that needs to be used outside of the function, you must declare the type that it returns.

```
public int myScore;
public GameObject collectable;
0 references
private void OnCollisionEnter(Collider other)
{
    collectable.SetActive(false);
    myScore += GetRandomNumber();
}
1 reference
private int GetRandomNumber()
{
    int randomNumber = Random.Range(0, 10);
    return randomNumber;
}
```

3 Every Function needs a unique name. You should choose a name that concisely describes what your Function does. If your function Instantiates an enemy, then CreateEnemy is a much better name than something like Spawn.

4 Functions can take any number of parameters. If you do not need to provide any variables or data to your Function, you can leave the inside of the parentheses empty.

```
0 references
void Start()
{

}
```

5 If your Function does need outside variables or data, you must declare the type and provide a name inside the parentheses.

```
0 references
private void UseInventoryItem(GameObject item)
{
    Destroy(item);
}
```

In the example above, the function needs to know what item from the player's inventory to destroy.

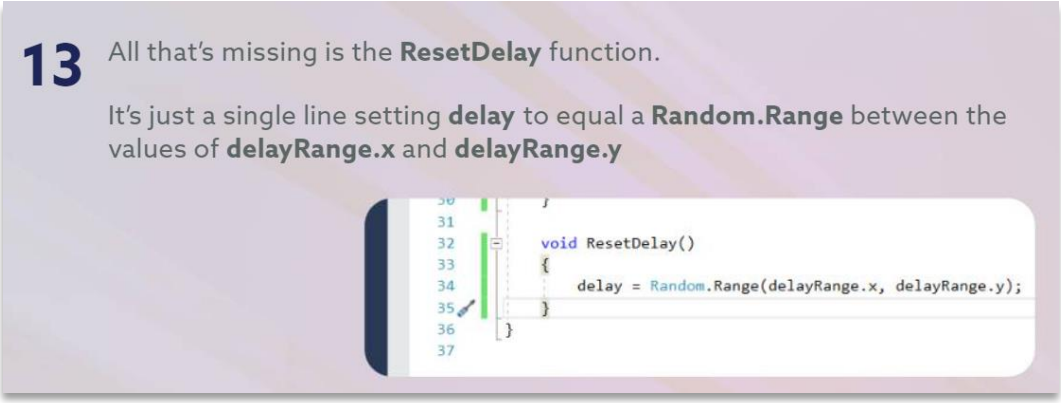
6 If your Function requires more than one parameter, you must separate them with commas.

```
0 references
public void SpawnPowerups(GameObject powerup, int count)
{
    for (int i = 0; i < count; i++)
    {
        Instantiate(powerup);
    }
}
```

In the example above, the Function needs to know what GameObject to Instantiate and how many times to run the loop.

Examples

In Purple Belt's Dropping Bombs Part 3, you used a Function to control how fast items spawned.



In Red Belt's Codey Raceway, you created a Function that used Invoke to reactivate the item boxes.

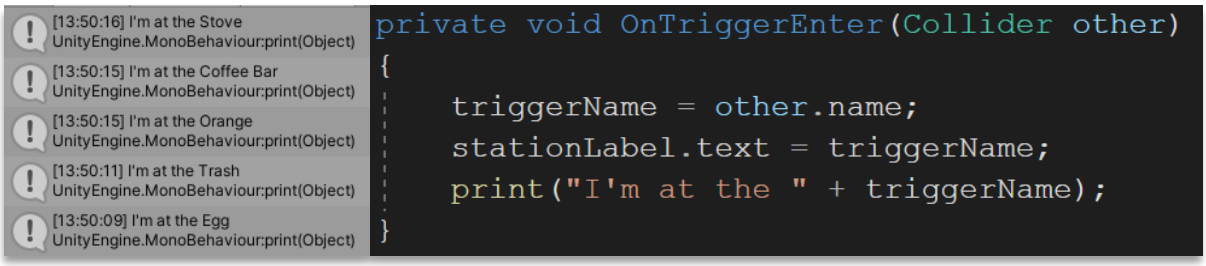
Ideas

Functions are used to easily repeat code without having to rewrite it each time you need it. Any time you find your self thinking "this seems familiar," you should try creating a Function.

You can pair your Functions with the Invoke Function to make something happen at a specified time in the future.

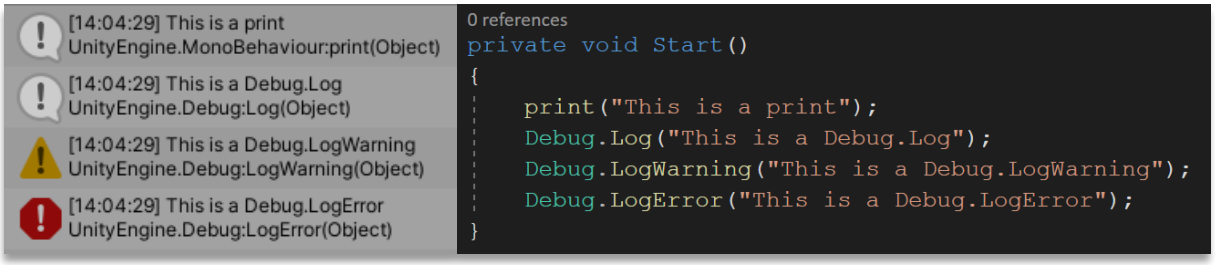
Console

The Console is a useful tool that can help you program and debug your game.



The print function will calculate the result of the parameter and write it out in the console. In the example above, we are printing the location of Codey each time a new trigger is entered.

While the print function will work for most situations, Unity's version of C# lets you use Warnings and Errors to make them stand out.



As you develop your game, you should rely on printing and logging to help you understand what your script is doing when it runs.

If you ever find that your code isn't working like you expect it to, it can be helpful to print the variable(s) that are acting up to diagnose the problem.

Conditionals and Booleans

A Conditional statement lets you control when certain blocks of code run. If your condition is true, then the code inside the body of the statement will run. If the condition is false, then the code will not run. Conditionals are made up of if, else if, and else condition.

```
0 references
private void teleportToLevel(string levelName)
{
    if (levelName == "Level 1")
    {
        SceneManager.LoadScene(1);
    } else if (levelName == "Level 2")
    {
        SceneManager.LoadScene(2);
    } else
    {
        SceneManager.LoadScene(0);
    }

}
```

You can use the conditional operators to create your conditional statements.

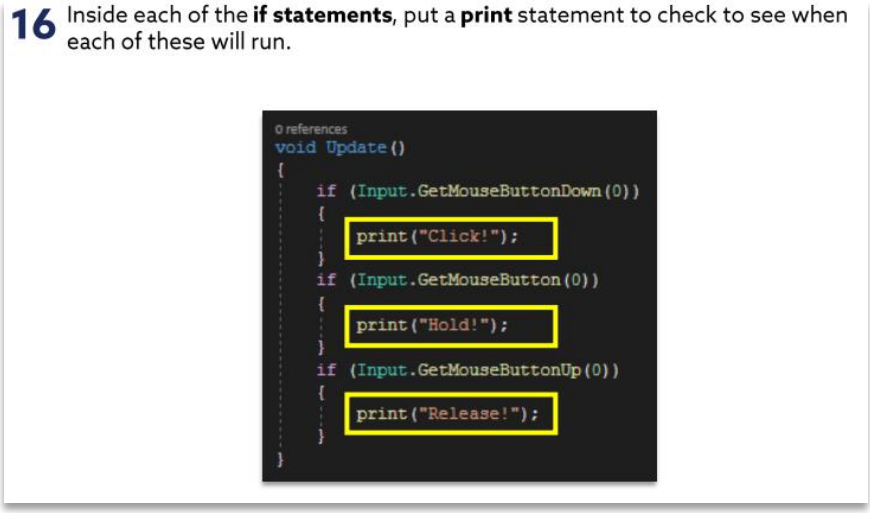
Statement	Definition
A == B	Returns true if A is equal to B.
A != B	Returns true if A is not equal to B.
A > B	Returns true if A is greater than B.
A >= B	Returns true if A is greater than or equal to B.
A < B	Returns true if A is less than B.
A <= B	Returns true if A is less than or equal to B.
A && B	Returns true if both A and B are true.
A || B	Returns true if A is true or B is true.
!A	Returns the opposite of A.

Note: The "|" symbol is found by holding shift and pressing the "\" key.

You can use a combination of all the conditional operators to create complex conditional statements.

Examples

In Red Belt's Sulky Slimes, you used multiple if statements to find out when certain mouse events were performed.



Ideas

You can use conditional statements to unlock a door only if the player has collected a key. You can use a combination of all the conditional operators to create complex conditional statements like in Chef Codey.

Switch Statements

A Switch statement is another way of writing conditional that checks the value of one parameter. It runs different code cases based on the parameter's value.

```
0 references
private void teleportToLevel(string levelName)
{
    switch(levelName)
    {
        case "Level 1":
            SceneManager.LoadScene(1);
            break;
        case "Level 2":
            SceneManager.LoadScene(2);
            break;
        case "Bonus":
            SceneManager.LoadScene(3);
            break;
        default:
            SceneManager.LoadScene(0);
            break;
    }
}
```

Instructions

1 A Switch statement takes one parameter that has distinct possible values. Strings and ints are often used as parameters because they have clear, exact values.

```
0 references
private void teleportToLevel(string levelName)
{
    switch(levelName)
    {

    }
}
```

2 The body of a Switch statement should contain at least one case. Cases are created by using the case keyword, the value you want to look for, and a colon.

```
0 references
private void teleportToLevel(string levelName)
{
    switch(levelName)
    {
        case "Level 1":
    }
}
```

3 If the paramter's value matches the value checked in the case, the code after the case will run until it reaches the break keyword.

```
0 references
private void teleportToLevel(string levelName)
{
    switch(levelName)
    {
        case "Level 1":
            SceneManager.LoadScene(1);
            break;
    }
}
```

4 You can add as many different cases as you want.

```
0 references
private void teleportToLevel(string levelName)
{
    switch(levelName)
    {
        case "Level 1":
            SceneManager.LoadScene(1);
            break;
        case "Level 2":
            SceneManager.LoadScene(2);
            break;
        case "Bonus":
            SceneManager.LoadScene(3);
            break;
    }
}
```

5 All Switch statements should contain a default case that runs if no other case conditions are met.

```
0 references
private void teleportToLevel(string levelName)
{
    switch(levelName)
    {
        case "Level 1":
            SceneManager.LoadScene(1);
            break;
        case "Level 2":
            SceneManager.LoadScene(2);
            break;
        case "Bonus":
            SceneManager.LoadScene(3);
            break;
        default:
            SceneManager.LoadScene(0);
            break;
    }
}
```

Examples

In Brown Belt's Food Frenzy, Switch statements are used to perform a task depending on the level that the player is on.

10e What's left to do is modifying the **subtext** objects depending on which of the three levels is being played. Inside a `SetLevelType` function, we'll be taking the `Level.LevelType type` parameter, and applying a **switch/case** statement to adjust **text** objects accordingly.

```
public void SetLevelType(Level.LevelType type)
{
     switch (type)
     {
         case Level.LevelType.MOVES:
               remainingSubtext.text = "moves remaining";
               targetSubtext.text = "target score";
               break;
         case Level.LevelType.OBSTACLE:
               remainingSubtext.text = "moves remaining";
               targetSubtext.text = "dishes remaining";
               break;
         case Level.LevelType.TIMER:
               remainingSubtext.text = "time remaining";
               targetSubtext.text = "target score";
               break;
     }
}
```

Ideas

While you can always use a series of if and else conditional statements, a Switch statement is useful when you are looking for specific values of one variable. You could use a Switch statement to run code based on the button a user pressed.

Arrays

An Array stores a set of similar data in a specific order. The individual elements inside the Array can be used through their indexes.

Arrays and Lists are very similar, but the size of an Array cannot change. This means Arrays are good to use when you know exactly what elements it needs to contain.

```
0 references
private float GetRandomFraction()
{
    float[] fractions = { 0.5f, 0.25f, 0.8f };
    int randomIndex = Random.Range(0, fractions.Length);
    return fractions[randomIndex];
}
```

Instructions

1 Instead of declaring an Array with the keyword Array, you use the type of data you want to store followed by square brackets.

```
string[] myStrings;

int[] myInts;
```

2 Once you declare an Array and give it a name, you can fill it by placing your data inside curly brackets and separated by commas. Note that in an array, the first item is located at index 0, the second at index 1, and so on. So, in the example bellow, “Hello” is stored at myStrings[0] and “world!” is at myStrings[1].

```
string[] myStrings = { "Hello", "world!" };
```

3 If you try to Add or Remove an item in an Array, you will get an error because the size of the array cannot change.

```
myStrings.Add("Goodbye!");
```

'string[]' does not contain a definition for 'Add' and no accessible extension method 'Add' accepting a first argument of type 'string[]' could be found (are you missing a using directive or an assembly reference?)

4 While an Array does not let you add or remove items, you can modify elements at specific indexes.

```
myStrings[1] = "Goodbye!";

print(myStrings);
// prints out ["Hello", "Goodbye!"]
```

5 Loops can be used to iterate through each element in an Array in order.

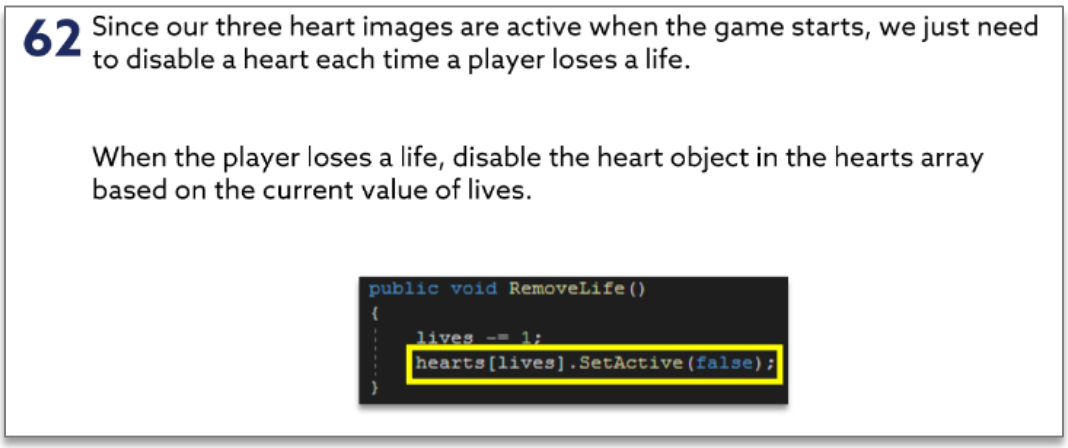
Typically, the loop index will start at 0 and go up to the number of elements in the Array. It should increment by one after each time through the loop.

In the body, you can access each element by using the loop's index variable.

```
for (int i = 0; i < myStrings.Length; i++)
{
    print(myStrings[i]);
}
// prints out "Hello"
// prints out "Goodbye!"
```

Examples

In Sulky Slimes, we used an Array to hold three UI images. Look at steps 57 through 63 to see how to remove specific images from the UI based on their index.



Ideas

You can use an Array to hold references to images used in UI elements or a set of powerups. It is important to remember that once you define the contents of an array, it cannot grow or shrink!

Lists

A List in C# stores pieces of similar data in a specific order. The individual elements inside the List can be accessed using their indexes.

While Arrays and Lists seem very similar, C# treats them as two different types. Most importantly, the size of an Array cannot change dynamcially, but a List can shrink or grow based on your code. Because we are using Unity, you won't notice many behind the scenes differences between the two types.

In the example below, a function uses a list of three strings to return a random food item.

```
private string GetRandomFood()
{
    List<string> items = new List<string>()
    {
        "pizza",
        "tacos",
        "fries"
    };
    int randomIndex = Random.Range(0, items.Count);
    return items[randomIndex];
}
```

Instructions

1 A List is different from most other types because it is just a way to hold other pieces of data. When you want to create a new List, you must use special syntax that declares the type of data it contains.

```
List<string> myStrings;
```

```
List<int> myInts;
```

2 Once you declare a List with a type and give it a name, you need to initialize it. This is another big way that Lists are unique.

First, you must declare a new List of the correct type. Then, you use parentheses to create the List and curly brackets to define the contents of the List. The items in the list need to be the correct type, and they must be separated by commas. The final syntax looks very similar to a Function!

```
List<string> myStrings = new List<string>()
{
    "Hello",
    "world!"
};
```

3 Now that you have your List, you can access what's inside through their indexes.

Like most other things in Computer Science, the first element in a list is at index 0, the second element is at index 1, and so on.

```
print(myStrings[1]); // prints out "world!"

print(myStrings[0]); // prints out "Hello"
```

4 Lists provide you with some special functions that let you read and modify them.

Use Count to return the number of elements contained in the List.

```
print(myStrings.Count); // prints out 2
```

5 In C#, you cannot just use + and – to modify a List.

Use Add to place a new element at the end of the list.

```
myStrings.Add("Goodbye!");

print(myStrings);
// prints out ["Hello", "world!", "Goodbye!"]
```

6 Use RemoveAt to delete an element at a specific index.

```
myStrings.RemoveAt(1);

print(myStrings);
// prints out ["Hello", "Goodbye!"]
```

7 Loops can be used to iterate through each element in a List in order.

The loop index should start at 0 and go up to the number of elements in the List. It should increment by one after each time through the loop.

In the body, you can access each element by using the loop's index variable.

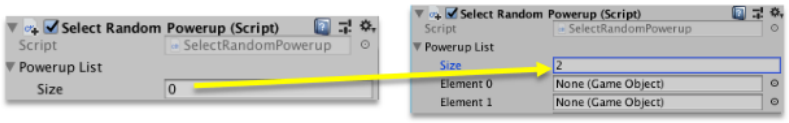
```
for (int i = 0; i < myStrings.Count; i++)
{
    print(myStrings[i]);
}
// prints out "Hello"
// prints out "Goodbye!"
```

Examples

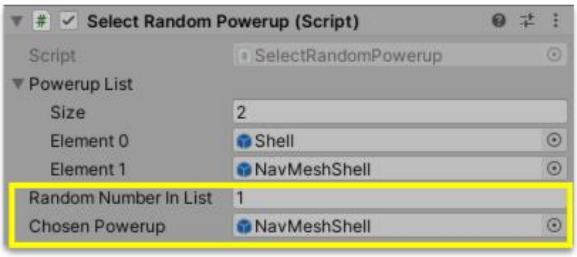
In Codey Raceway, we used a List to hold the powerups a player could use. Note how the Powerup List variable was defined though the Unity Inspector instead of code. Look at steps 159 through 167 to see how powerups were randomly selected from a List.

160 Save your script and return to Unity. Click on Codey and in the Inspector, we need to update the **Select Random Powerup Script** component.

Change the size of the Powerup List from 0 to 2.



Collect an item box and see how the public variables in the script change.



Ideas

A game's inventory system could be represented by a List. The player can add items, remove items, and look through each item. The individual items could have unique properties just like the shells in Codey Raceway. Since the size of the inventory would change throught the game, a List would be a better choice than an Array.

For Loops

A For Loop is made of three parts: a starting index, a run condition, and instructions on how to change the index after each time through the loop.

```
0 references
private void CountToTen()
{
    for (int index = 0; index <= 10; index++)
    {
        print(index);
    }
}
```


Instructions

1 Each For Loop starts with the keyword "for" followed by parentheses.

```
for ()
```

2 Inside the parentheses, you must declare an integer variable to be the index of your Loop. You can name it anything you want, but common names are index and i.
You can name it based on what you are looping through. For example, if you are looping through all the enemies in your Scene, then you could name the index variable enemy.
You must also declare an initial value for your variable. This will typically be 0.

```
for (int enemy = 0;)
```

3 The second parameter is the run condition. Your Loop will run only while this condition evaluates to true. Once it evaluates to false, your Loop will stop running.

Use your index variable in your conditional statement.

```
for (int enemy = 0; enemy < arrayOfEnemies.Length;)
```

4 The last piece of code that goes inside the parentheses is how you want to change your index variable after each Loop. Most often you will add or subtract one.

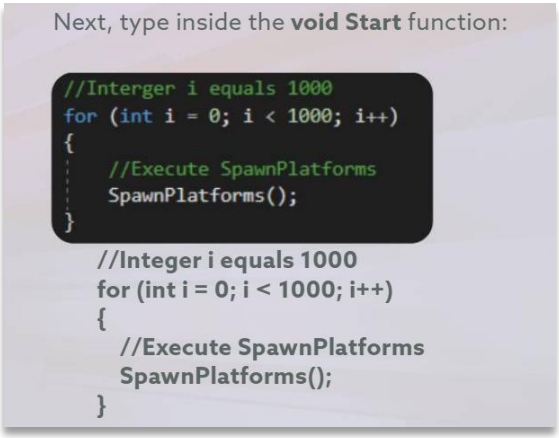
```
for (int enemy = 0; enemy < arrayOfEnemies.Length; enemy++)
```

5 Use curly brackets to make the body of your loop. The code inside will run each time the value of your index variable makes the conditional statement true.

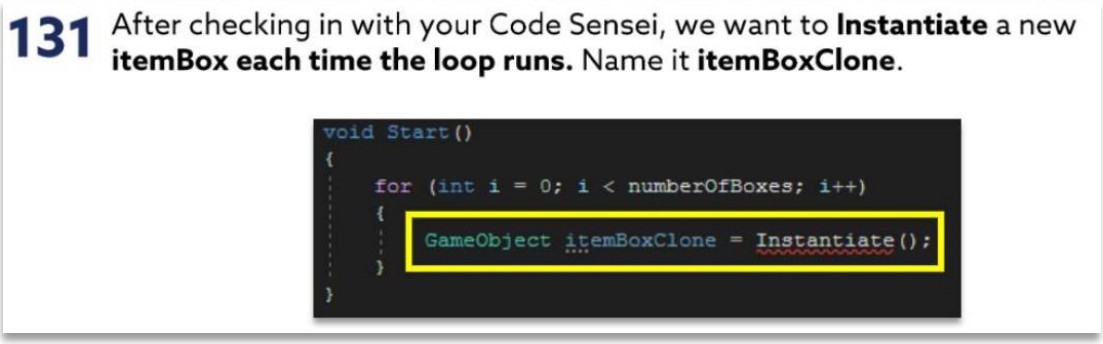
```
for (int enemy = 0; enemy < arrayOfEnemies.Length; enemy++)
{
    // destroy each enemy in the array
    Destroy(arrayOfEnemies[enemy]);
}
```

Examples

In Purple Belt's Sketch Head, you used a For Loop to spawn up to 999 platforms!



In Red Belt's Codey Raceway, we used a For Loop to spawn our item boxes. You could change the number of item boxes by changing the index variable.



Ideas

If you do not want to individually place all the game objects in the scene as you build your game, you can use a for loop to help you position your game objects. This could be different platforms, enemies, obstacles or powerups.

If you store any data or game objects in an array, then you can perform code on each one by writing a For Loop.