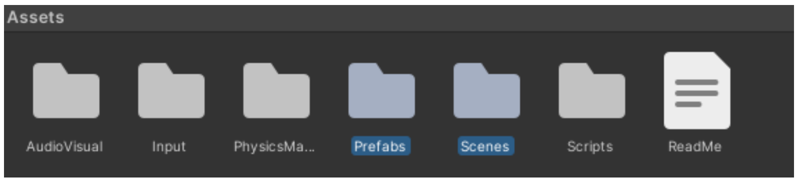
Contents

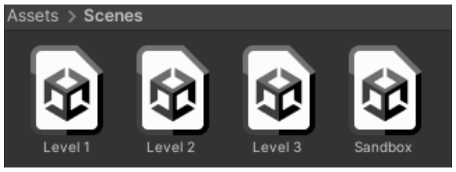
Creating Levels

In the project *Roll_a_Ball_Design_Challenge*, we have provided you with various tools to create your own roll a ball game without needing to do any scripting. There are a handful of folders in the game's *assets* directory, but you only need to focus on two: *Scenes* and *Prefabs*.

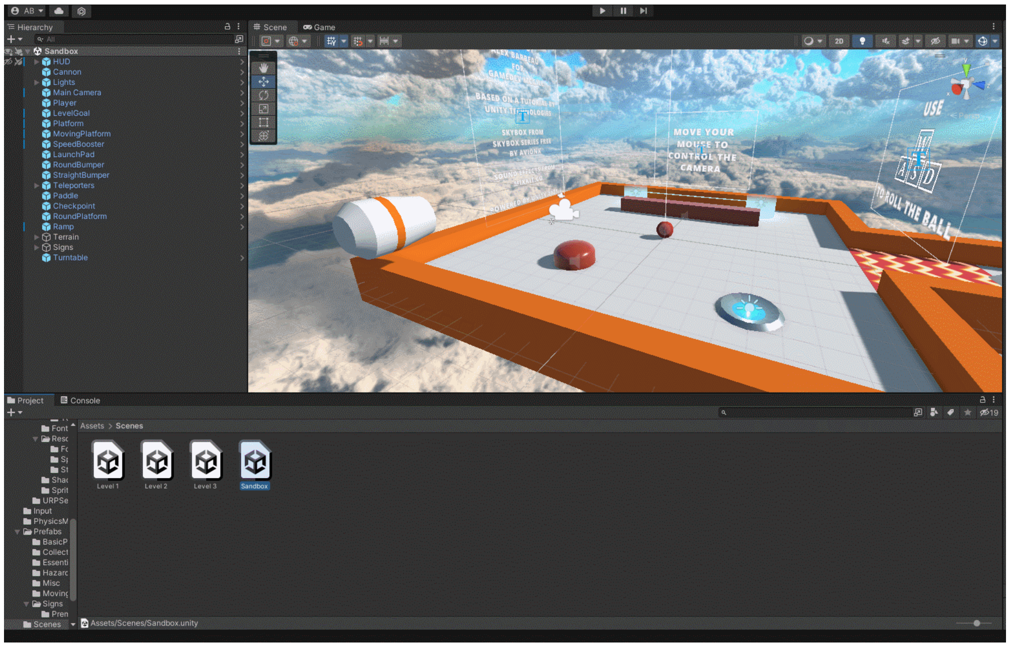


Scenes

The *Scenes* folder contains 4 scenes, *Sandbox*, *Level 1*, *Level 2*, and *Level 3*.



These scenes represent levels in our game. By default, the *Sandbox* scene should be open in the Scene Editor. By double clicking (or right clicking and selecting `open`) on any of these scenes, you can switch to editing it.



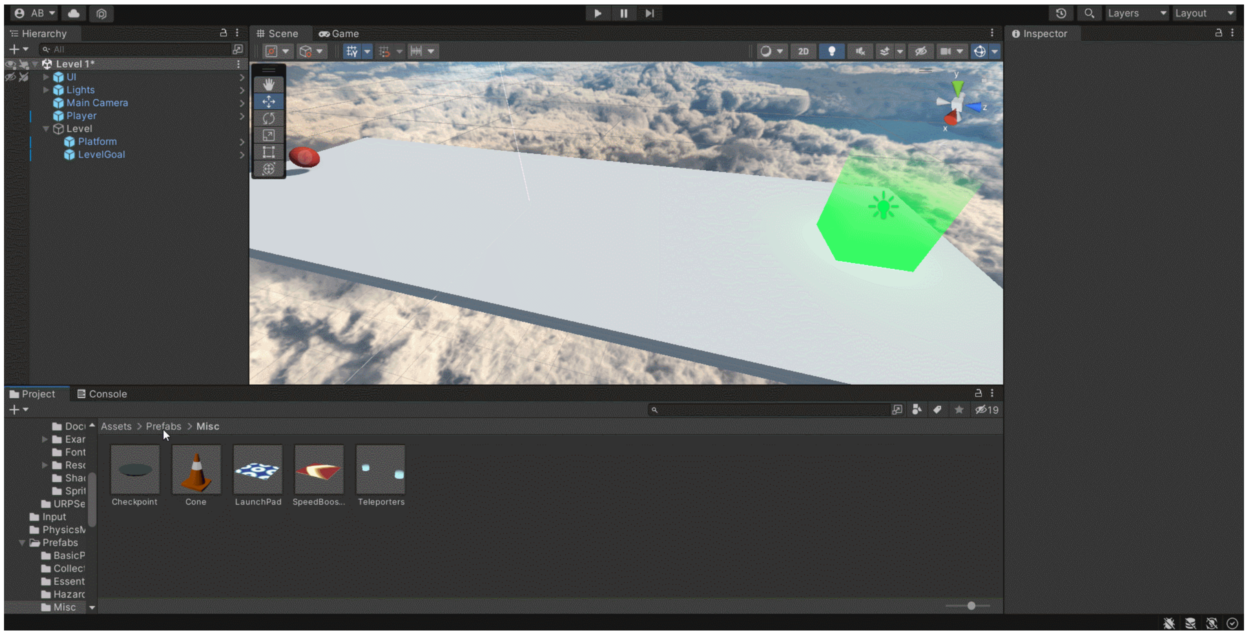
opening the scene Level 1

We've provided all the scenes you need to do the design task. However, if you want to add your own levels, you can as well. In order to create a new scene, right click and select `Create>Scene`. After creating a scene, you'll want to open it, and add a **Player**, a **Main Camera**, a **HUD**, and **Lights**, all of which can be found in the *Essential* subfolder of the *Prefabs* folder (explained in further detail in the next section). You'll also probably want to add a **Platform** so the ball doesn't fall straight into the void.

Prefabs

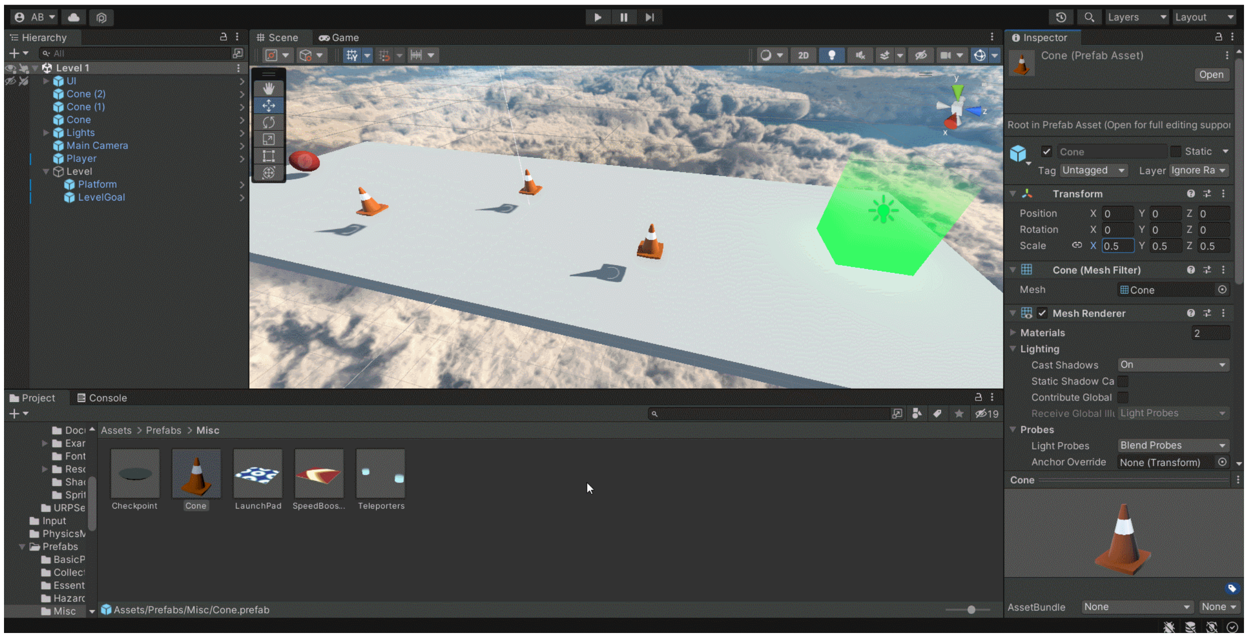
The *Prefabs* folder contains a plethora of **game objects** you can use to create your own levels. These are divided into *Essential, Basic Platforms*, *Moving Platforms*, *Collectables*, *Hazards*, *Signs*, and *Misc*. This next section will go over each of these **prefabs** in depth, detailing how you can use them in your levels, and how to edit their **components** for unique results.

In order to add a **prefab** to a scene, simply click and drag it from the *Project* window into the *Scene* window, or into the *Hierarchy* window. This creates an **instance** of the **prefab** within our current scene.



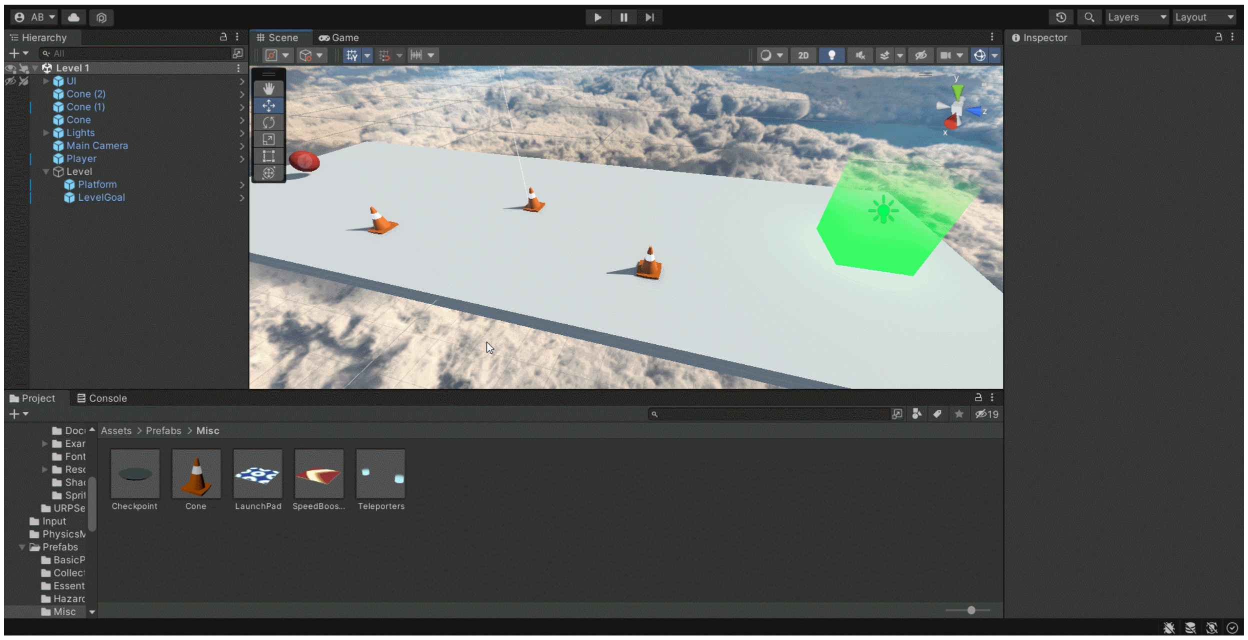
adding a ***Cone*** *to the scene*

When you select a **prefab** in the project window, that **prefab's** **components** appear in the *Inspector* window. If you make a change to any of these, it is reflected in every instance of the **prefab** in every scene. This is very powerful if you want to make large changes to the game. For example, when editing the **Player**, you will almost always want to edit the prefab directly, so that the **Player** behaves the same from one scene to the next. The only exceptions are the **Position** and **Rotation** values of the **prefabs'** **transforms**.



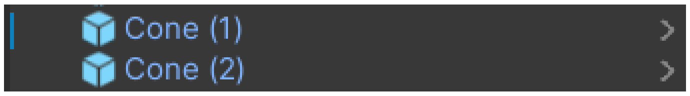
*changing the scale of the **Cone prefab** changes the scale of every **Cone** instance*

You can select an individual instance by clicking on it in the *Scene* window or in the *Hierarchy* window (instances are also selected automatically upon creation). When you select it, the instance's **components** appear in the *Inspector* window. Modifying them here does not affect any other instances of the same prefab.

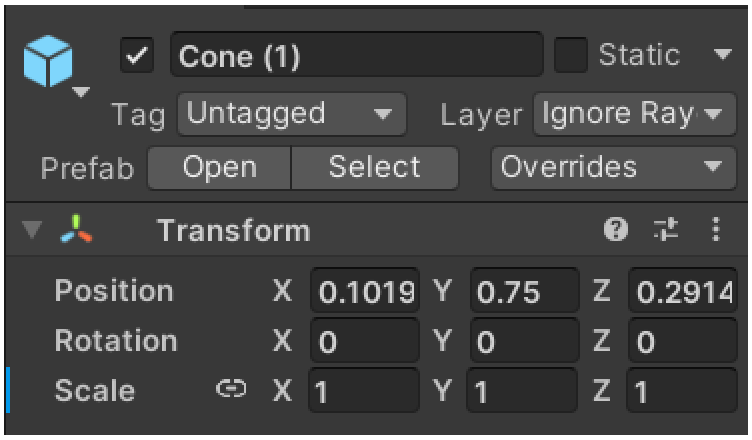


*selecting a **Cone instance** and modifying the **position** and **scale** of its **transform** component*

After you modify an instance of a **prefab**, it is said to override that prefab. If an instance is overriding the original **prefab**, a small blue line appears in the *Hierarchy* to indicate that. A similar line appears in the *Inspector* next to the overridden value.



***Cone (1)** overrides the prefab while **Cone (2)** does not*



Position X 0.1019 Y 0.75 Z 0.2914
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Cone (1)** overrides the **Scale** value of the **Cone prefab's transform

If a value is overridden by an **instance**, changing that value in the **prefab** will not affect that **instance**. This allows you to safely edit both individual **instances** and original **prefabs** without worrying about one conflicting with the other. If you want an **instance** to stop overriding the **prefab**, select `Overrides` in the *Inspector* window, and either `Revert` or `Apply` depending on whether you want the **instance** value to switch to the **prefab**'s value, or vice versa (affecting all other **instances** in the latter case).

Essentials

The following **prefabs** are absolutely essential to the game. Every level should have at least one **instance** (and, in most cases, exactly one) of each of these elements.

Player

One of the two game objects the user can directly control (the other being the Main Camera). The user can move the **Player** using the WASD keys on their keyboard, and (if enabled in the **Player Controller**), use the Ctrl key to brake and the spacebar to jump.

Warning: the **Player Controller** component is designed such that there can only be one in each scene while the game is running. As such, even if you put multiple **Players** in the scene, when you play only one will function.

Important **Components**:

- **Player Controller**
  This gives the **Player** almost all of its functionality, such as movement and being respawned after falling.
  - **Base Speed:** The **Player**'s max speed, before any modifiers such as Speed Boosters.
  - **Acceleration:** The **Player**'s acceleration. Setting this significantly lower than speed gives the **Player**'s movement more weight, and setting it higher than speed makes it very responsive.
  - **Air Control Multiplier:** How much control the **Player** has in the air. A value lower than 1 results in less air control than ground control, and a value greater than 1 results in greater air control. A value of 0 gives the **Player** no control in the air."
  - **Movement Mode:** Controls which directions the **Player** can move in, relative to the position of the Main Camera.
    - Back And Forth: The **Player** can only move away from and towards the camera.

        - Side To Side: The **Player** can only move left and right vis a vis the camera.
        - Rook Movement: The **Player** can move forwards, backwards, left, and right, but not diagonally.
        - Omnidirectional: The **Player** can move in any direction.
    - **Death Height:** y level at which the **Player** "dies" and has to be respawned. The **Player** will not die if it is under this height for less than a second. -50 by default.
    - **Allow Jumping:** If checked, allows the **Player** to jump by hitting the spacebar or left clicking.
    - **Jump Count:** How many times the **Player** can jump.
      At 1, the **Player** can only jump once and only on the ground. At 2, the player can jump once on the ground and once in the air. At n jumps, the player can jump once on the ground and n-1 times in the air.
    - **Jump Force:** How much force the **Player** jumps with. Higher values allow the player to reach greater heights when jumping.
    - **Allow Braking:** Determines whether or not the **Player** can hit control or right click to brake.
        - Never: The **Player** cannot brake.
        - Grounded Only: The **Player** can only brake on the ground.
        - Any Time: The **Player** can brake both on the ground and in the air
    - **Brake Amount:** How much braking slows the **Player** down. The greater the number, the more the **Player** slows down.
- **Rigidbody**
  Gives the attached object physics. For full information see the **Unity Manual**

page.

Warning: when the game starts, the **Player Controller** resets certain values of the attached **Rigidbody**, these being **Mass**, **Drag**, **Angular Drag**, and **Is Kinematic**. Changing these values should have no effect on your game.

- **Freeze Position:** If checked, freezes that component of the **object**'s **position** (for example, if **y** is checked, the **object** is unable to move up or down).
- **Freeze Rotation:** If checked, freezes that component of the **object**'s **rotation**.

Main Camera

One of the two game objects the user can directly control (the other being the **Player**). The user can move the **Main Camera** using the mouse (if enabled in the **Camera Controller**).

Warning: the **Camera Controller** component is designed such that there can only be one in each scene while the game is running. As such, even if you put multiple **Main Cameras** in the scene, when you play only one will function.

- **Camera**
  Allows the **game object** to serve as a camera. For more information see the Unity Manual page.
- **Camera Controller**
  Grants the user the ability to control the **Main Camera** with the mouse. Also gives options to restrict this control.
  - **Camera Mode:** Determines how much control the player has over the camera.
    - Fixed: No camera control.

        - Horizontal Control: The player can move the mouse to turn the camera left and right.
        - Full Control: The player can move the camera in any direction with the mouse.
    - **Offset:** Offsets the position the camera follows
    - **Camera Distance:** The distance the camera follows the player from.
    - **Start Rotation:** The initial rotation of the camera when the game starts.
    - **Sensitivity:** How mouse movement should be translated into camera movement. Higher sensitivity means faster camera movement.
    - **Reset Camera Transform:** Pressing this button immediately sets the **Main Camera**'s position and rotation to those dictated by the **Offset**, **Camera Distance**, and **Start Rotation**. This allows you to preview where the camera will be when the scene starts from within the editor.

HUD

This **game object** controls the game's GUI. This includes the score counter, the speedometer, and the black overlay that is displayed when the player falls off the edge.

The **HUD** has one child, the **Canvas**, which represents the area on which the GUI is drawn. The **Canvas** in turn has three children, **Darkness**, **Score**, and **Speed**, which are all elements of the UI.

Warning: the **HUD** component is designed such that there can only be one in each scene while the game is running. As such, even if you put multiple **HUDs** in the scene, when you play only one will function.

- **Hud**
  Updates the UI elements.

- **Enable Score Text:** If checked, the score text is shown and is updated when the player gets points.
- **Score Text:** Refers to a Text Mesh Pro UGUI component (essentially, text). This component is used to display the score. By default, this is the Text Mesh Pro - Text attached to the Score child of the HUD.
- **Enable Speedometer:** If checked, the speedometer is shown and is updated to match the player's speed.
- **Speedometer:** Refers to a Text Mesh Pro UGUI component (essentially, text). This component is used to display the player's speed. By default, this is the Text Mesh Pro - Text attached to the Speed child of the HUD.
- **Enable Darkness:** If checked, the **Darkness** Image gradually fades in as the Player falls past the **Death Height** (as defined in the Player Controller attached to the Player).
- **Darkness:** Refers to an Image component. This image fades in when the player falls to their doom. By default, this is the Image attached to the Darkness child of the HUD.

Lights

Contains one child, SunLight, which in turn has a Light attached to it. One of these should be in every scene, providing basic lighting. Modifying the prefab by modifying the existing Light or attaching more children allows you to quickly modify the lighting in every scene containing one.

Level Goal

The goal of the level. You will implement functionality so that this allows you to proceed to another scene. Essential for allowing the game to progress from one level to another.

- **Box Collider**
  The **Level Goal** script requires at least one **Collider** to be attached to the same **game object** with the **Is Trigger** property checked in order to function correctly. For more information, see the **Unity Manual page** on collision.

- **Level Goal**
  Provides all basic functionality.
  - **Require Score:** If checked, the goal is inactive so long as the player has not reached the **Score Threshold**.
  - **Score Threshold:** How many points the player must have to proceed to the next level.

Platforms

In order to make an actual level, you need something for the **Player** to actually roll on. The *Platforms* folder contains the most basic types of platforms, in order to make basic levels.

Every platform has some kind of **Mesh Renderer** and **Collider** attached, and we've provided multiple platforms with useful shapes. If you're up to the challenge and want more variety in platforms, feel free to import your own assets to make your own platforms, though this is not required. You can make a great level using only the shapes provided.

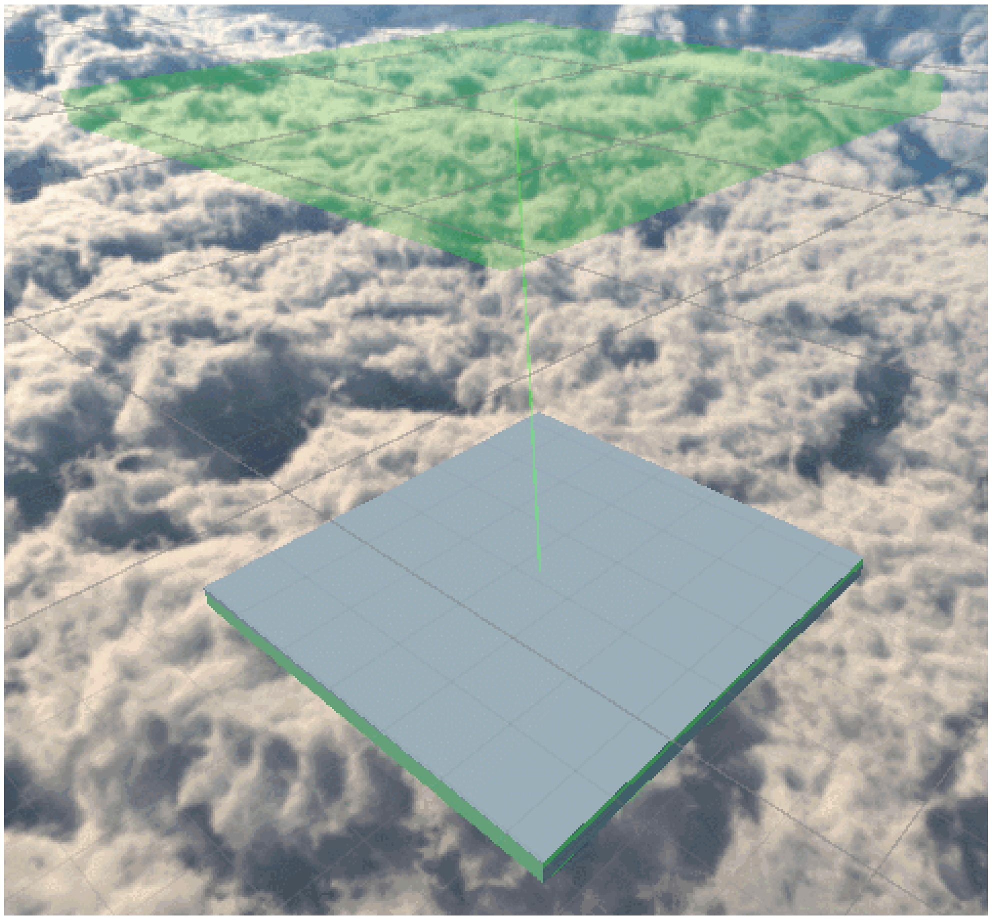
Moving Platforms

The *Moving Platforms* folder is full of platforms which move and spin.

Moving Platform

The **Moving Platform** is a versatile object capable of moving and rotating. In the *Scene* window, the start and end position of the platform are previewed in green.

The **Moving Platform** always starts at the position and rotation specified in its **Transform**, and ends at the position and rotation given in its **Moving Platform** component.



a ***Moving Platform*** *moving between its start and end positions*

- **Rigidbody**
  Gives the attached object physics. For full information see the **Unity Manual page**.
  An attached **Rigidbody** is required by the **Moving Platform** script. Because the latter fully controls the object's movement, editing the **Rigidbody** directly is essentially pointless.

- **Moving Platform**
  Controls the movement of the **Moving Platform**. The platform always takes the

most direct path to its target position and rotation. The movement is always cyclical.

  - **Smoothing:** How smooth the platform's movement should be. 0 is completely linear, and 1 is completely smooth.
  - **Movement:** How far the end of the movement should be from the beginning. For example, a platform at (1, 2, 3) with a movement value of (4, 5, 6) would have a final position of (1+4, 2+5, 3+6) = (5, 7, 9).
  - **End Rotation:** The final rotation of the platform. When it reaches the end of its movement, it will have this rotation.
  - **Move Duration:** Time in seconds it takes the platform to reach the end of its movement, after which it will return to its original position.

See Saw

The **See Saw** is a simple moving platform controlled by Unity's physics and the built-in **Hinge Joint** component. The **See Saw** tilts along the x-axis in response to weight and other forces acting on it, and stabilizes when there are no unbalanced forces acting on it.

- **Rigidbody**
  Gives the attached object physics. For full information see the Unity Manual page.
  An attached **Rigidbody** is required by the **Hinge Joint** component.
- **Hinge Joint**
  Limits the attached **object's** rotation and position such that the **Anchor** is always in the same place, and its rotation is limited to the provided **Axis**. Also provides useful parameters for adding a motor-like or spring-like behaviour. For more information, see the Unity Manual page.
  The **See Saw** uses the **Spring** of the **Hinge Joint**

Turntable

The **Turntable** is a platform that is constantly rotating about the y-axis. It is controlled by Unity's physics and the built-in **Hinge Joint** component.

- **Rigidbody**
  Gives the attached object physics. For full information see the Unity Manual page.
  An attached **Rigidbody** is required by the **Hinge Joint** component.

- **Hinge Joint**
  Limits the attached **object's** rotation and position such that the **Anchor** is always in the same place, and its rotation is limited to the provided **Axis**. Also provides useful parameters for adding a motor-like or spring-like behaviour. For more information, see the Unity Manual page.
  The **Turntable** uses the **Motor** of the **Hinge Joint** to control its spinning.

Windmill

The **Windmill** follows the same principle as the **Turntable**, but turns on a different axis, and has more complex geometry. In order to achieve the latter, the **Windmill** has two children, each of which has a **Box Collider** attached, one representing the vertical parts of the **Windmill** and the other the horizontal parts.

- **Rigidbody**
  Gives the attached object physics. For full information see the Unity Manual page.
  An attached **Rigidbody** is required by the **Hinge Joint** component.

- **Hinge Joint**
  Limits the attached **object's** rotation and position such that the **Anchor** is always in the same place, and its rotation is limited to the provided **Axis**. Also provides useful parameters for adding a motor-like or spring-like behaviour. For more

information, see the Unity Manual page.The **Windmill** uses the **Motor** of the **Hinge Joint** to control its spinning.

Collectables

The *Collectables* folder contains collectables which increase the score when the **Player** touches them. There are three **prefabs** provided, though they have identical functionality (the only differences being colour and number of points awarded).

Collectable

- **Sphere Collider**
  The **Collectable Behaviour** script requires at least one **Collider** to be attached to the same **game object** with the **Is Trigger** property checked in order to function correctly. For more information, see the Unity Manual page on collision.
- **Collectable Behaviour**
  Gives the attached **object** the ability to be picked up by the **Player** for points.
  - **Value:** How many points the player gains after picking up the **Collectable**.
- **Bob and Rotate**
  Makes the attached **object** bob up and down and spin. This should only be used for aesthetic purposes.
  - **Bob Multiplier:** How much the object bobs. Greater values mean more bobbing.
  - **Spin:** If true, the object spins.
  - **Bob:** If true, the object bobs.

5x Collectable

See **Collectable**.

25x Collectable

See Collectable.

Hazards

The *Hazards* folder is full of obstacles that can impede the Player's progress by knocking them around.

Cannon

The **Cannon** fires a steady stream of projectiles capable of launching the Player off course.

The **Cannon** automatically attaches a **Cannonball** and a **Rigidbody** component to any projectile it launches. The former automatically deletes projectiles after a certain amount of time (for performance), and the latter provides physics. For more information, see Cannonball.

- **Cannon**
  Unsurprisingly, the **Cannon's** functionality is provided by the **Cannon** component.
    - **Projectile:** Which object the canon fires. These objects should always have a Rigidbody attached.
      By default, value is a reference to the **Cannonball** prefab. Changing this value may have unexpected results, but feel free to experiment by setting this to different **prefabs**.
    - **Force:** How much force the projectile is launched with.
    - **Offset:** How far from the cannon the projectile is initially created. Greater values mean further forward.
    - **Reload Time:** How many seconds pass between shots.

    - **Start Phase:** Offsets the time before the first shot. A canon with a Start Phase of 1 fires 1 second before one with a Start Phase of 0. This can be used to stagger shots from canon while still keeping them synchronized.

Cannonball

The default projectile fired by the **Cannon**. The **Cannonball** is very heavy and very bouncy, making it a very effective projectile.

- **Rigidbody**
  Gives the attached object physics. For full information see the **Unity Manual page**.
  An attached **Rigidbody** is required by the **Cannonball** script.

- **Cannonball**
  The primary purpose of the **Cannonball** script is to destroy **Cannonballs** after a certain amount of time. This is essential for performance, as the more physics objects there are in the scene, the more work needs to be done to simulate them.
  The **Cannonball** script also causes a sound to play when the attached object collides with something, assuming a sound is provided.

    - **Lifetime:** The amount of time, in seconds, that the **Cannonball** persists for.

    - **Bounce Sound:** The sound that plays when the **Cannonball** hits something.

Paddle

The **Paddle** is an object that spins in circles, potentially knocking the **Player** off course in the case of a collision. The **Paddle** functions using the same principles as the **Windmill** (found in the *Moving Platforms* folder).

- **Rigidbody**
  Gives the attached object physics. For full information see the **Unity Manual**

page.
An attached **Rigidbody** is required by the **Hinge Joint** component.

- **Hinge Joint**
  Limits the attached **object's** rotation and position such that the **Anchor** is always in the same place, and its rotation is limited to the provided **Axis**. Also provides useful parameters for adding a motor-like or spring-like behaviour. For more information, see the Unity Manual page.
  The **Paddle** uses the **Motor** of the **Hinge Joint** to control its spinning.

Round Bumper

The **Round Bumper** is one of two **prefabs** which have the purpose of bouncing the **Player** on collision (the other being the **Straight Bumper**). When the **Player** touches the bumper, they are immediately launched away from it, even if they touched it very gently.

Note that the **Player** is launched mainly with horizontal force, with very little vertical force being applied. If you want an **object** that launches the **Player** upwards or downwards, use the **Launch Pad** instead.

- **Bumper**
  Launches the **Player** when they collide with the attached **object's** collider(s). Also provides a small animation of the **object** springing (by contracting and expanding in scale) as well as the sound effect played when it launches the **Player**.
  - **Force Amount:** How much force the bumper imparts on the **Player**.

Straight Bumper

When the **Player** touches the bumper, they are immediately launched away from it, even if they touched it very gently. For more information, see **Round Bumper**, which functions identically.

Misc.

All the **prefabs** which didn't fit neatly into another category. Despite that, most of these are still very useful.

Checkpoint

By default, when the **Player** "dies" by falling too far, they respawn where they started in the scene. After touching a **Checkpoint**, the **Player** instead respawns at the **Checkpoint's** location. This is a useful tool to alleviate frustration in longer levels, since most people would rather not repeat an entire level because they made a small mistake near the end.

- **Sphere Collider**
  The **Checkpoint** script requires at least one **Collider** to be attached to the same **game object** with the **Is Trigger** property checked in order to function correctly. For more information, see the **Unity Manual page** on collision.
  The **Checkpoint** prefab uses an oversized **Sphere Collider** to trigger the **Checkpoint** script. This allows for some leniency as to what counts as "touching" the **Checkpoint**, which is done to make the game slightly more forgiving. You can adjust the size of this **collider** or replace it with another collider with **Is Trigger** enabled as you see fit.

- **Checkpoint**
  Provides the basic functionality of the **Checkpoint**. When the **Player** enters the **Checkpoint's Sphere Collider**, the **Checkpoint** becomes activated. When activated, the **Checkpoint** glows green (rather than blue), and when the **Player** respawns, they respawn at the activated **Checkpoint**. When a new **Checkpoint** is activated, the previously activated **Checkpoint** is deactivated automatically.

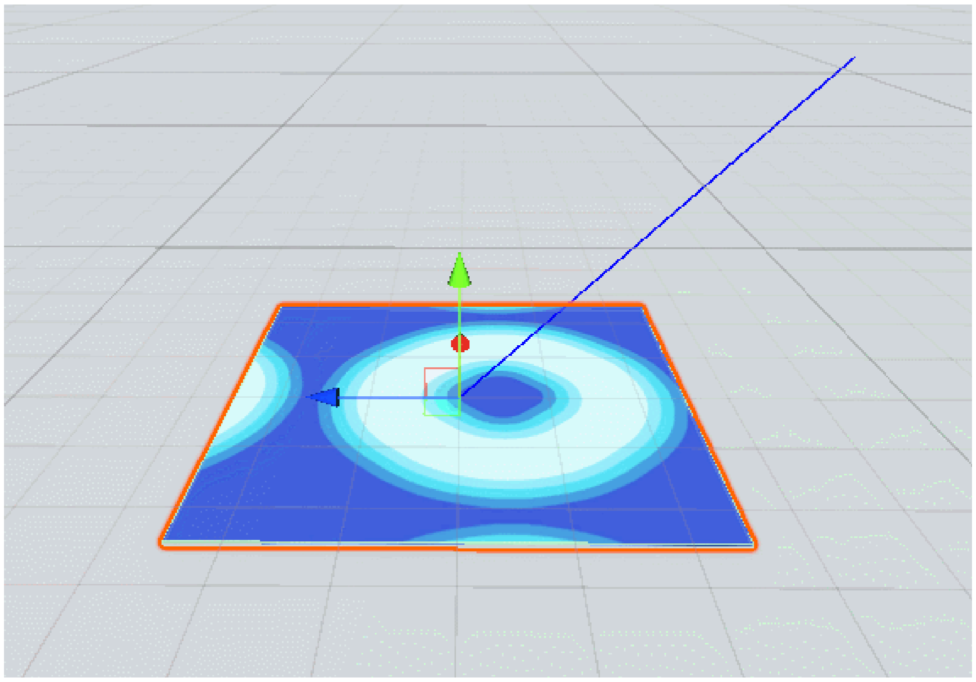
Cone

The **Cone** is a simple **object** using physics. Use these for decoration, as obstacles, or just for fun. For more information on how these actually work, see the Unity Manual pages on **Collision** and **Rigidbody Physics**.

Launch Pad

The **Launch Pad** is an object that instantly launches the **Player** in a predetermined direction when they touch it. This can be used as a tool for the **Player** that launches them to an otherwise inaccessible part of the level, or as an obstacle which launches them to their doom.

When the **Launch Pad** is selected in the *Scene* view, the direction that it launches the **Player** in is visualized by a blue line. When the game is playing, there is also an animation to indicate this direction. The direction of the launch is relative to the **Launch Pad**; if the **Launch Pad** is rotated, the launch direction changes to match.



*a **Launch Pad** which will launch the **Player** upwards and to the right when touched*

- **Box Collider**
  The **Launch Pad** script requires at least one **Collider** to be attached to the same **game object** with the **Is Trigger** property checked in order to function correctly. For more information, see the **Unity Manual page** on collision.
- **Launch Pad**
  This script provides the actual functionality of the **Launch Pad**.
  - **Force:** The force imparted on the **Player** when it touches the launchpad.

Speed Booster

When the **Player** is in contact with a **Speed Booster**, their maximum movement speed and acceleration are multiplied. This can potentially allow the **Player** to reach otherwise inaccessible parts of the level, be used as an obstacle that makes the **Player** more difficult to control, or be used purely for excitement.

The **Speed Booster** plays a scrolling animation that goes faster the greater the **Boost Factor** is. While this animation is directional, the **Speed Booster** itself is not; the **Player** travels just as fast on a **Speed Booster** regardless of which direction they are travelling in.

- **Box Collider**
  The **Speed Booster** script requires at least one **Collider** to be attached to the same **game object** with the **Is Trigger** property checked in order to function correctly. For more information, see the **Unity Manual page** on collision.
- **Speed Booster**
  This script provides the actual functionality of the **Speed Booster**.
  - **Boost Factor:** How much the booster speeds up the **Player**. A booster with a boost factor of 2 causes the speed of the **Player** to double, 3 causes it to triple, and so on.

Teleporters

The **Teleporters prefab** has no **components** other than a **transform**, but has two **children**, **Teleporter a** and **Teleporter b**. When the **Player** touches either of these **Teleporters**, they are immediately teleported to the other **Teleporter** (from a to b or from b to a). The **Player** maintains their velocity and rotation when exiting a **Teleporter**.

The following **components** are attached to **Teleporter a** and **Teleporter b** rather than to **Teleporters** itself.

- **Capsule Collider**
  The **Teleporter** script requires at least one **Collider** to be attached to the same **game object** with the **Is Trigger** property checked in order to function correctly. For more information, see the **Unity Manual page** on collision.

- **Teleporter**
  Provides the **Teleporter's** main functionality of sending the **Player** to a corresponding **Teleporter**.

  - **Linked Teleporter:** The teleporter this teleporter sends the player to. When two **Teleporters** are linked, a line is drawn between them in the *Scene* view. This line also previews the **Color** of the **Teleporters**.

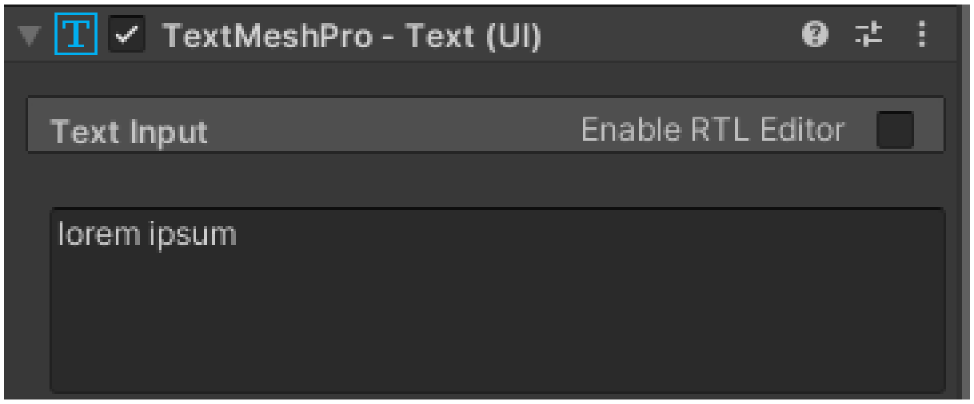
  - **Color:** The color of the **Teleporter** (this only appears when the game is being played). Linked **Teleporters** are always the same color.

Signs

The *Signs* folder is full of signs you can use to convey information. Each of these **prefabs** is made up of a **parent** and one or several **children**.

The **parent object** always has a **Canvas** attached. This represents the area in which the sign's contents are displayed.

The **children** objects all have either a **Text Mesh Pro - Text** attached, or a **Sprite**. The former of these is used for displaying text, while the latter is used to display images.



lorem ipsum

*the text displayed by a sign is controlled by the **Text Input** field of its **Text Mesh Pro - Text***

For the sake of brevity, this section will only cover the components unique to each sign.

Basic Sign

A simple sign. The **parent** has a **Canvas** attached, and the **child** (**Text TMP**) has a **Text Mesh Pro - Text** attached.

Arrow Sign

Like the **Basic Sign**, but with a second **child** (**Arrow**). **Arrow** has a **Sprite** attached which draws an arrow on the **Canvas**, and a **Bob and Rotate** script attached which makes the arrow bob up and down (for more information on **Bob and Rotate**, see **Collectable**).

Rotating Sign

Like the **Basic Sign**, but the **parent** has one extra **component**: **Face Camera**. This causes the **parent** (and the attached **Canvas**) to face the **Main Camera** at all times.

Credits

Contains the game's credits. See **Basic Sign**.

Mouse Sign

Tells the user how to control the **Main Camera**. See **Basic Sign**.

WASD Sign

Tells the user how to control the **Player**. Like the **Basic Sign**, but with a second **child** (**Image**). **Image** has a **Sprite** attached which draws a picture of the WASD keys on the **Canvas**.