Smart Assets

version 1.0

Please view this document in Google Docs
because this document contains useful GIFs:
https://docs.google.com/document/d/17m4y5yoBm_ukfak3RyERgg6bgui734qsSq4X1_8slVg/edit?usp=sharing

Trajectory Aimer 2D

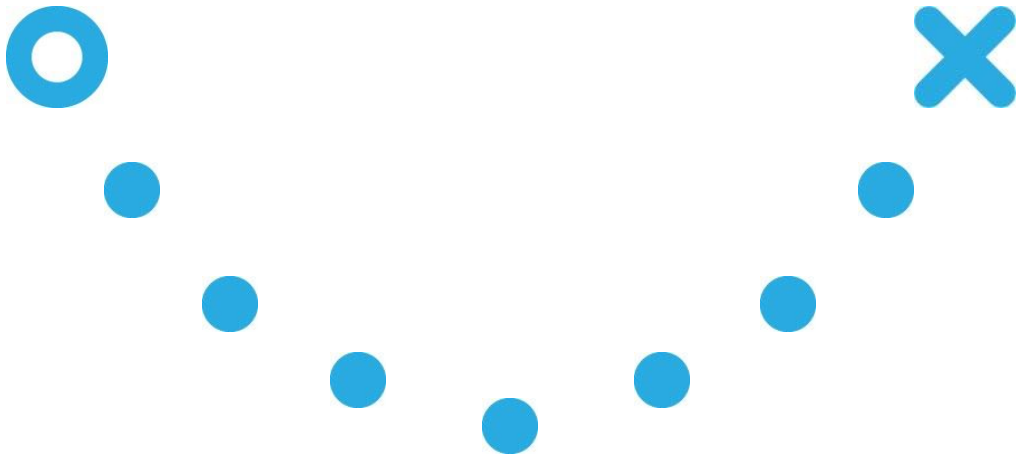
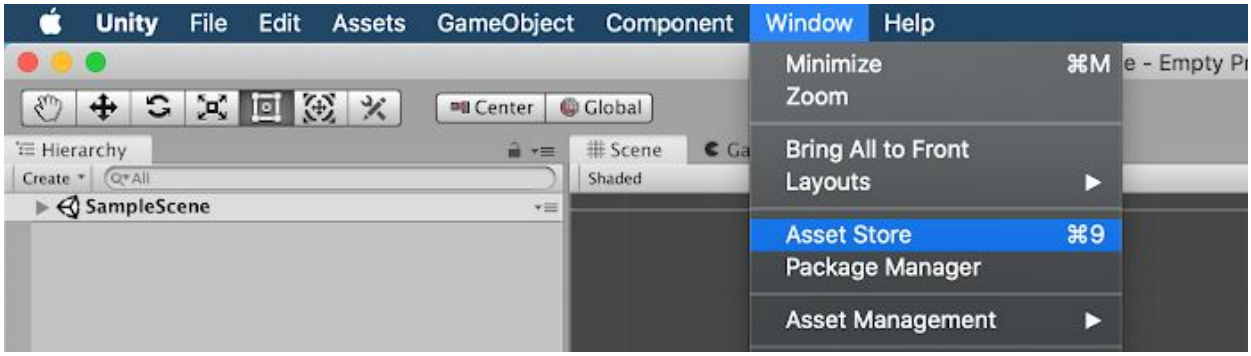


Table of Contents

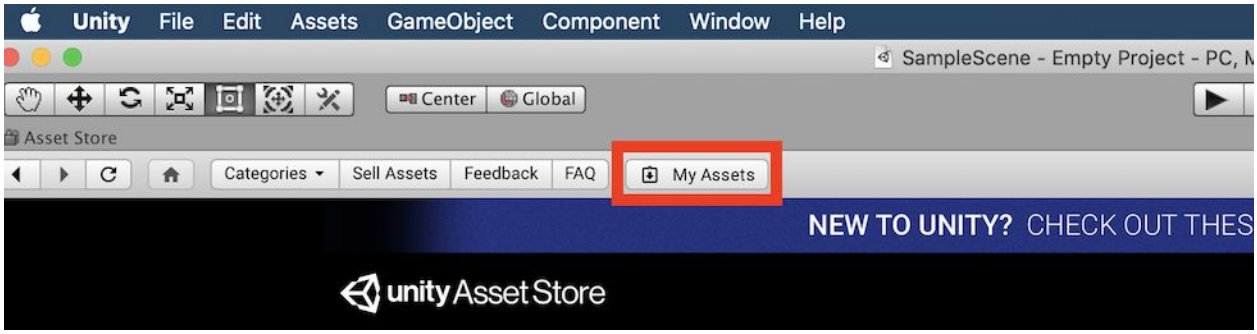
Quick Start

Import

Open your project in the Unity editor. Select **Window** > **Asset Store**.

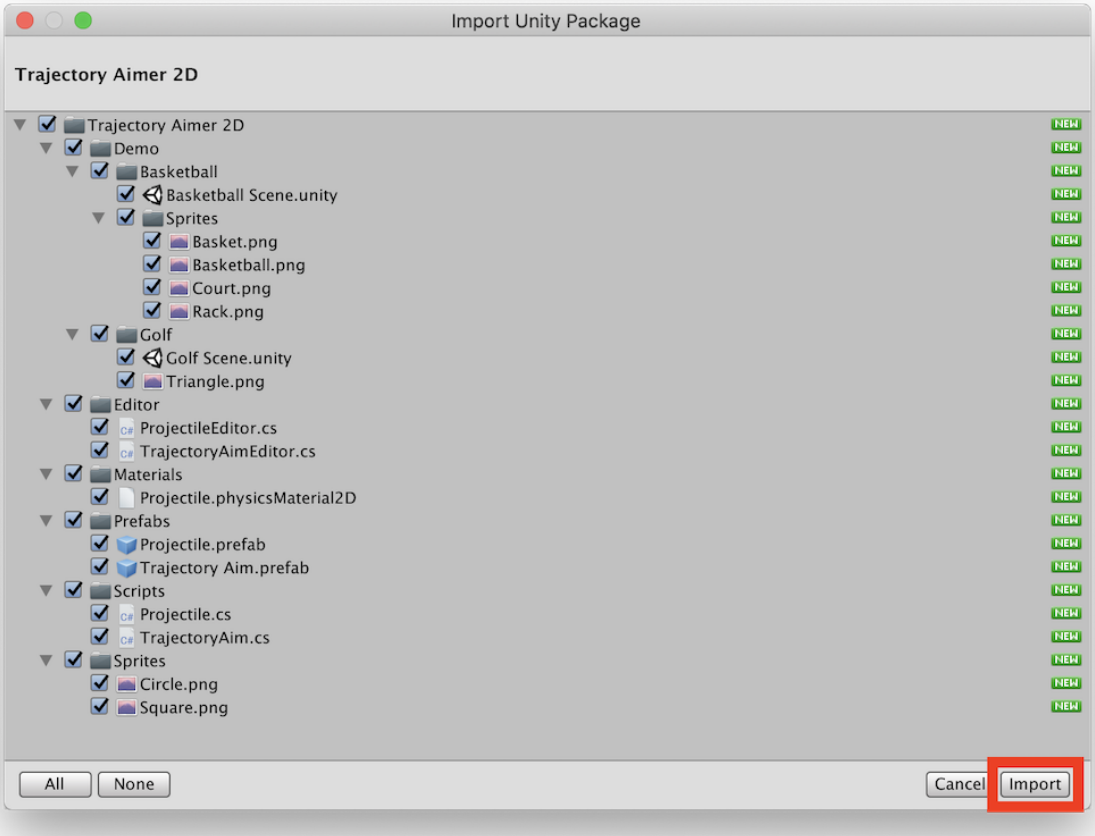


Click the **My Assets** button.



Locate the *Trajectory Aimer 2D* asset and click **Import**. (You may need to click **Update** or **Download** before this button appears)

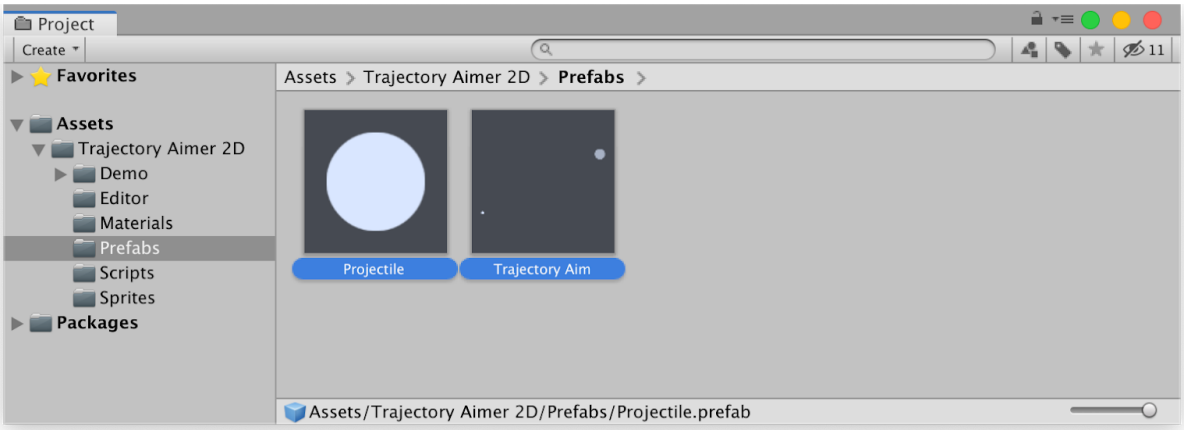
Make sure every file is selected and click **Import**.



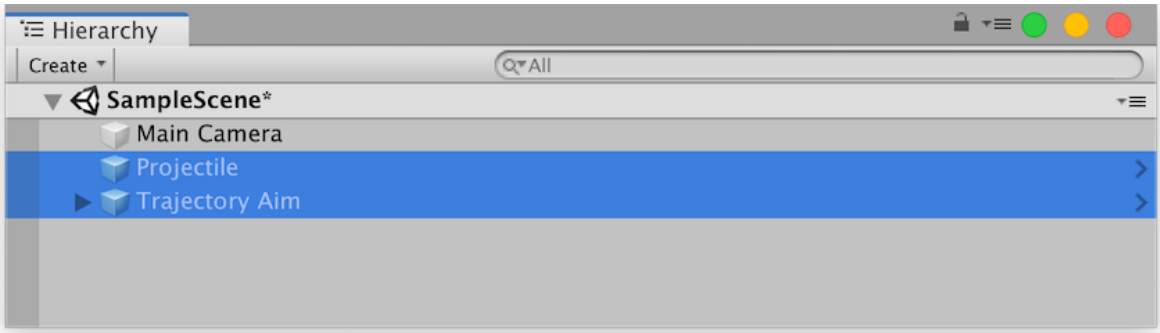
Using Your Own Scene

Add the prefabs to your scene

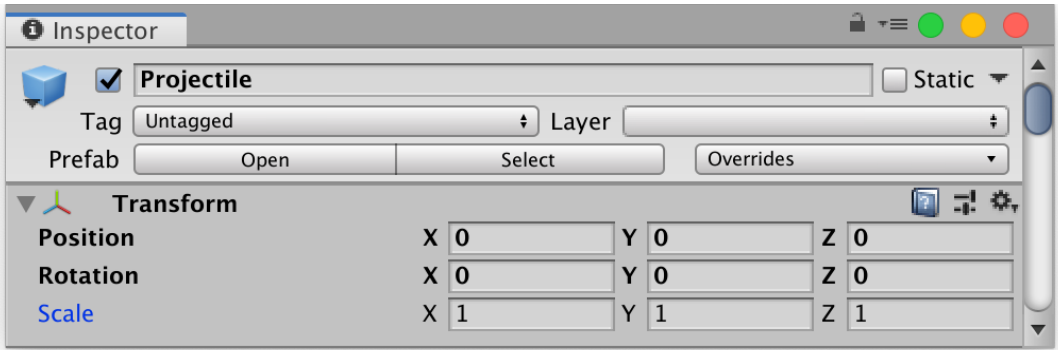
In the **Project** window, find the **Projectile** and **Trajectory Aim** prefabs in the **Assets / Trajectory Aimer 2D / Prefabs** folder.



Drag the **2 prefabs** into the **Hierarchy** window to add them to your scene.



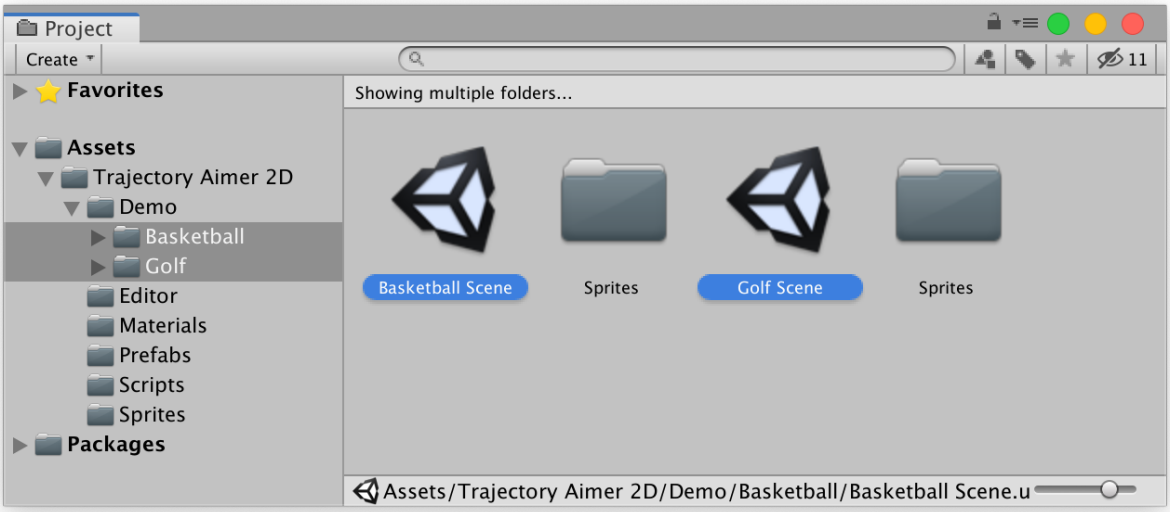
You can select the **Projectile** and adjust it's scale in the **Inspector** window to suit the scale of your scene. For details on changing the other settings, see Inspector Settings.



Using a Demo Scene

Open a demo scene

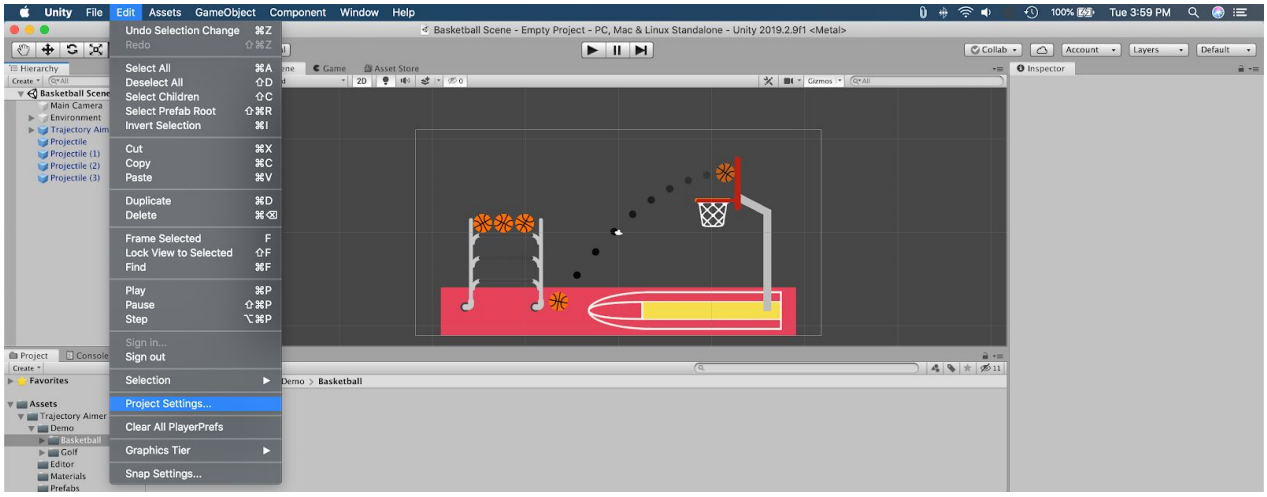
In your **Project** window, double click on a **demo scene** from one of the folders within **Assets / Trajectory Aimer 2D / Demo**.



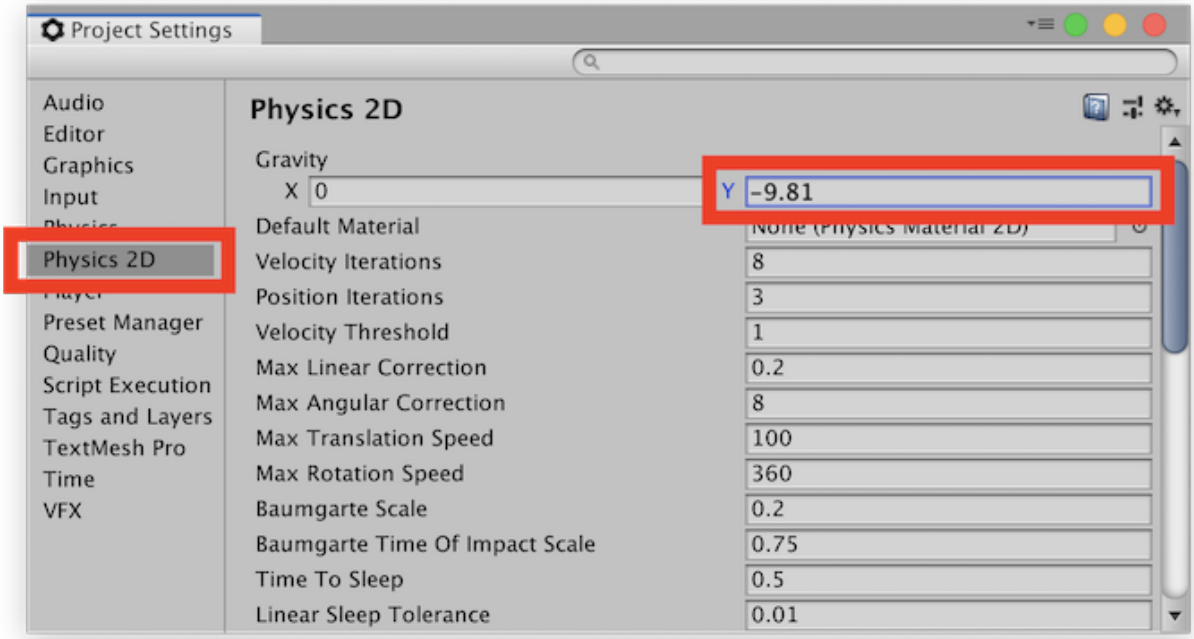
Adjust settings

To get the demo scenes working correctly, we need to set the **Gravity** to **X** = **0** and **Y** = **-9.81** and set the **Game** window aspect ratio to **16:10**.

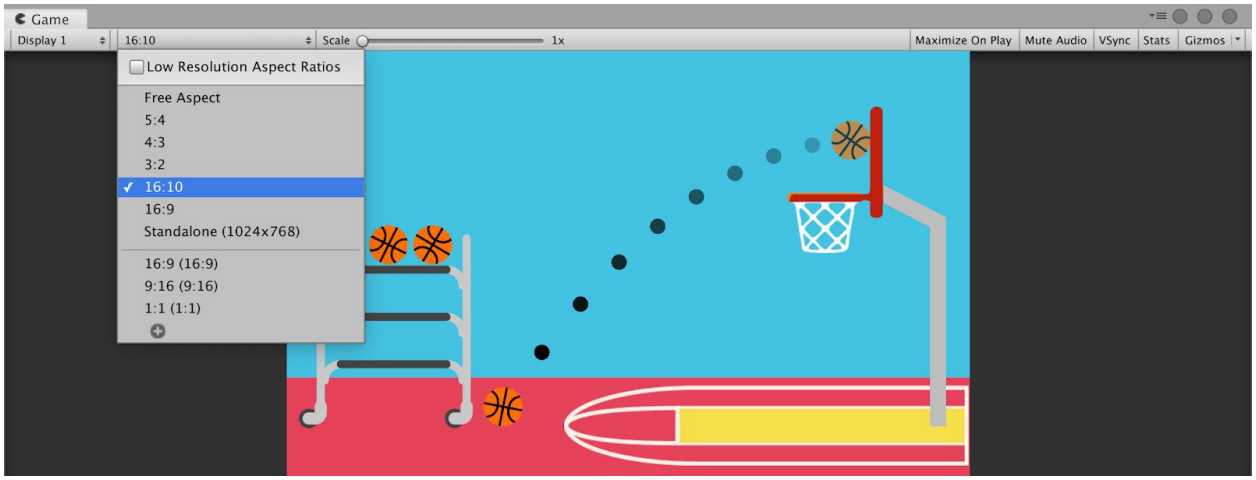
Open the **Project Settings** window in **Edit** > **Project Settings**.



Click on **Physics 2D** and set **Gravity** to **X = 0** and **Y = -9.81**.



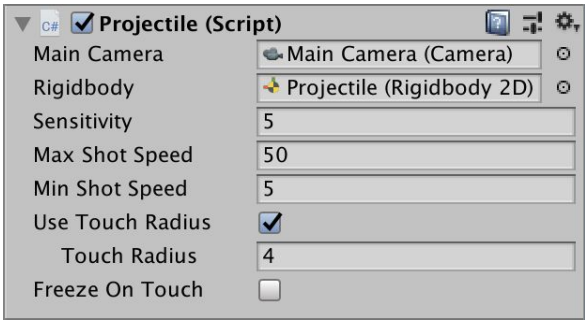
In the **Game** window, set the aspect ratio to **16:10**.



The demo scene is now ready. Click the **Play** button to try it out!

Inspector Settings

Projectile



Main Camera

`Camera Projectile.mainCamera;`

Assign the **Main Camera** of your scene here.

Rigidbody

`Rigidbody2D Projectile.rigidBody;`

Assign the **Rigidbody 2D** of the Projectile here.

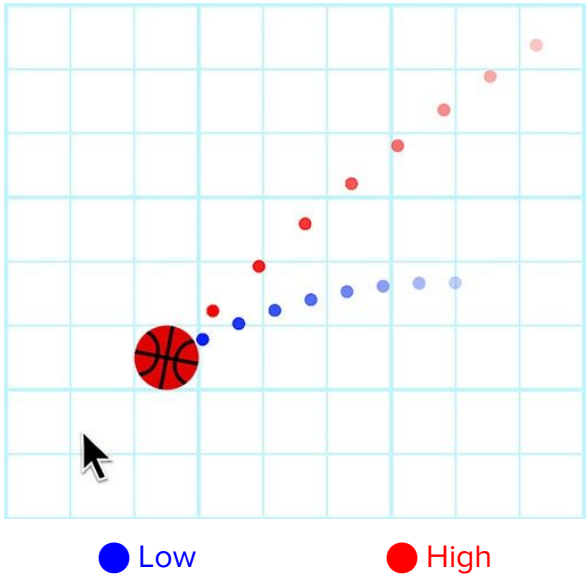
⚠ Warning

The **Linear Drag** of the **Rigidbody 2D** on the Projectile must be set to **0** since drag calculations aren't supported.

Sensitivity

`float Projectile.sensitivity;`

This value determines the amount of velocity to be added to the Projectile while aiming. When clicking and dragging, the same amount of distance dragged will produce a more powerful shot if this value is higher and a less powerful shot if this value is lower.



Max Shot Speed

```
float Projectile.maxShotSpeed;
```

The maximum limit to how much speed can be applied to a shot. When aiming, if the maximum shot speed has been reached, dragging further won't result in a more powerful shot. You will be able to see that the Trajectory Line doesn't stretch any further after this point.

Min Shot Speed

```
float Projectile.minShotSpeed;
```

If a shot is taken and its speed is below this amount, the shot will be **cancelled**. This is the minimum limit to how much speed a shot is required to have to be performed. When aiming, if the shot speed is below this amount, the Trajectory Line will hide and will show once this shot speed has been passed.

Use Touch Radius

```
bool Projectile.useTouchRadius;
```

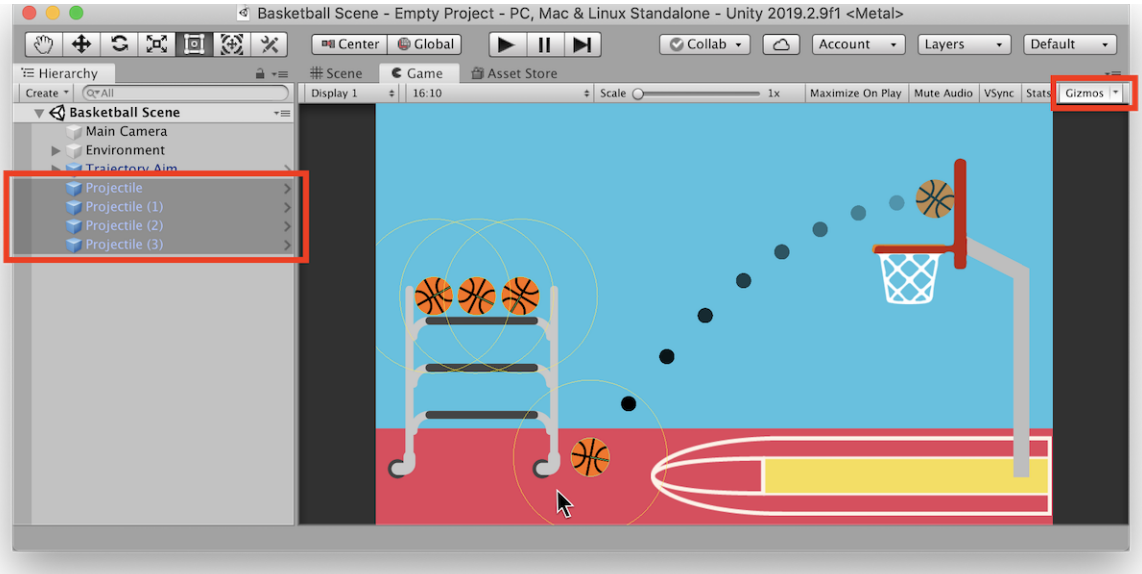
When enabled, a click will need to be performed within the Touch Radius to start a shot. When disabled, a click can be located anywhere to start a shot.

Touch Radius

```
float Projectile.touchRadius;
```

The radius of the circle around the Projectile which a click must be performed within to start a shot. If a click is performed within multiple Projectiles' Touch Radius, the Projectile nearest to the click position will be selected.

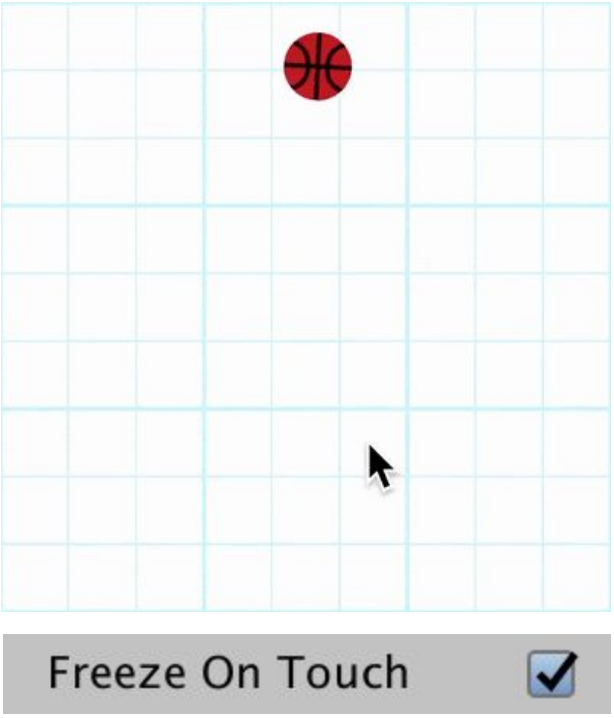
Make sure **Gizmos** is enabled then select a Projectile to see the yellow circle around it representing its Touch Radius.



Freeze On Touch

```
bool Projectile.freezeOnTouch;
```

When enabled, the Projectile will freeze in place when aiming. Rigidbody 2D is set to Kinematic to achieve this.

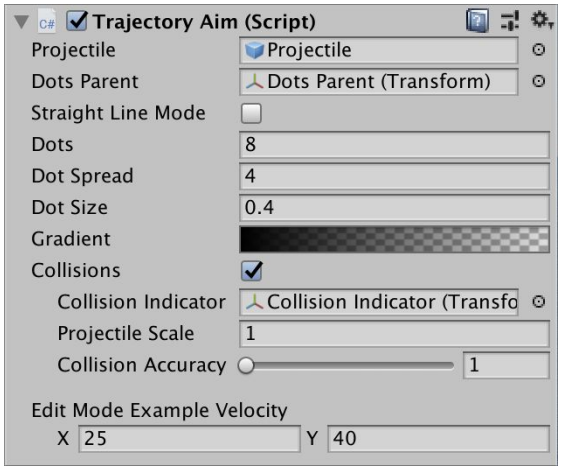


Trajectory Aim

Projectile

```
GameObject TrajectoryAim.instance.Projectile;
```

Assign here the **Projectile GameObject** which will be getting launched.



Dots Parent

```
Transform TrajectoryAim.instance.dotsParent;
```

The **Transform** of the **Dots Parent** GameObject which contains all the dots as its children. It should only contain dots as its children and should always have at least 1 child.

Straight Line Mode

```
bool TrajectoryAim.instance.straightLineMode;
```

When enabled, **gravity calculations will be disabled**. Dots will be arranged into a straight line based on the starting velocity of a shot.

Dots

```
int TrajectoryAim.instance.Dots;
```

The **number of dots** in the trajectory line. In **Edit Mode**, dots will automatically be added/deleted when changing this value. In **Play Mode**, dots will be set inactive/active and instantiated only if not enough dots already exist under the **Dots Parent** Transform. Dots won't be deleted in Play Mode by decreasing this value.

Dot Spread

```
float TrajectoryAim.instance.DotSpread;
```

The amount of **space between the dots**.

Dot Size

```
float TrajectoryAim.instance.DotSize;
```

The **scale of the dots**. All dots' scales will change to this value when it is changed.

Gradient

```
Gradient TrajectoryAim.instance.Gradient;
```

The gradient across the trajectory line.

Optimization Tip: If you plan on changing the number of dots frequently during runtime, setting the gradient's Alpha and Color to be the same across the entire gradient - making it a single color - will increase efficiency. As a result, each dots' colors and alphas won't need to be updated when the number of dots changes.

Collisions

```
bool TrajectoryAim.instance.Collisions;
```

When enabled, the **trajectory path will be scanned for collisions**. By default, only **colliding layers** in the Projectile's **Layer Collision Matrix** will be scanned for collisions. For info on the Layer Collision Matrix and **manually picking which layers are scanned for collisions**, see the section on Layers.

Collision Indicator

```
Transform TrajectoryAim.instance.collisionIndicator;
```

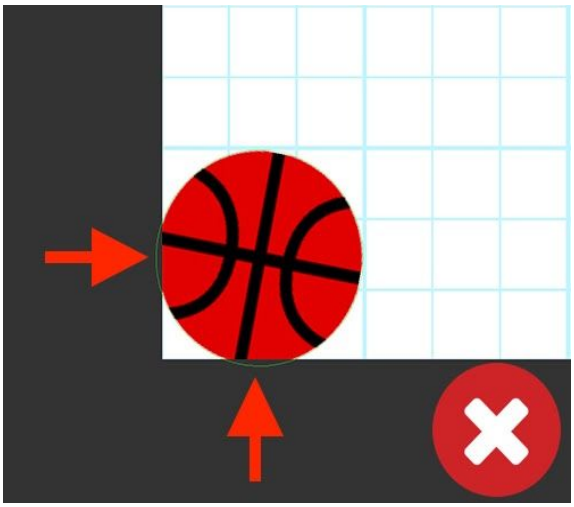
The **Transform** of the **Collision Indicator** GameObject that will be **placed in the position of a detected collision**. The Collision Indicator GameObject can be found under the **Trajectory Aim** GameObject and can be modified in its **Inspector** (ex. changing the color, sprite, scale, etc.).

Projectile Scale

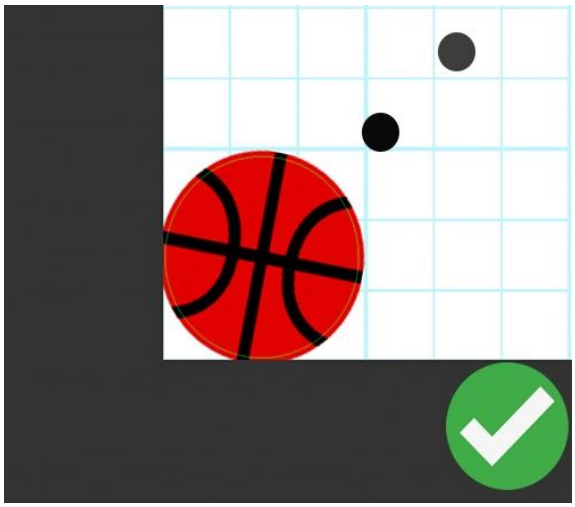
```
float TrajectoryAim.instance.projectileScale;
```

The scale of the Projectile being launched. Indicated as a green circle in the **Scene** and **Game** view when the **Trajectory Aim** GameObject is selected (if Gizmos is on). If the Collider of the Projectile is a **Circle Collider**, this value should be equal to the scale of the Collider. Otherwise, **it is recommended that it should be as large as possible WHILE still being completely inside of the Projectile's Collider**. This value is essentially the thickness of the trajectory line and if any part of the line overlaps with a collidable object, a collision will be detected.

Tip: In Unity, when a collision occurs, there's a brief moment where the colliders of the 2 colliding GameObjects are **overlapping**. If you plan on allowing aiming WHILE your Projectile is moving around and colliding with colliders, set the Projectile Scale slightly smaller than the Collider of the Projectile (around 5% smaller) to avoid allowing the brief moment of overlap to cause a scanned collision.



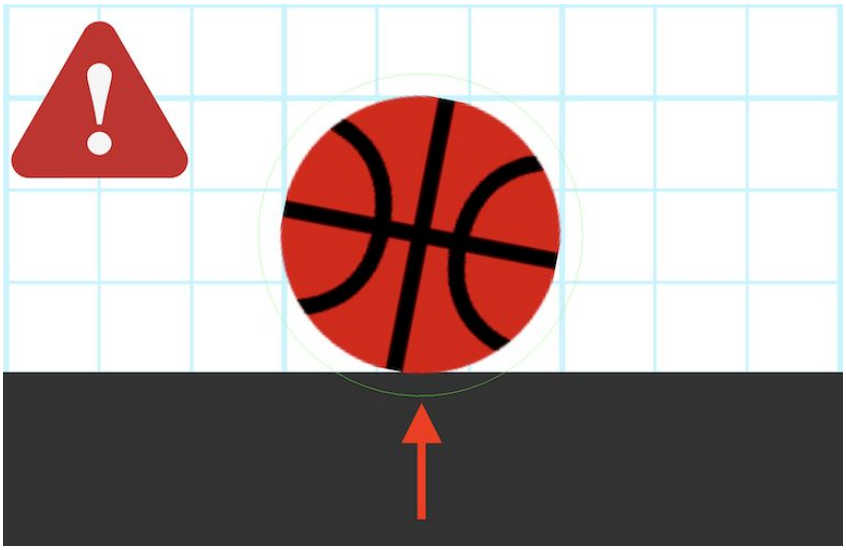
100% of Projectile's Scale



95% Projectile's Scale

⚠ Warning

This value shouldn't be **larger** than the Projectile's Collider. This will result in constantly scanning the overlap as a detected collision. The green circle indicating the Projectile Scale (if Gizmos is on) should be completely within the Projectile's Collider.



Larger than Projectile Collider's scale

Collision Accuracy

```
int TrajectoryAim.instance.collisionAccuracy;
```

The **number of scans** used to accurately find the position of an **already detected collision**. Set this value low for better efficiency.

Example:
Collision Accuracy = **1**:
No additional scans; only 1 scan between each dot of the trajectory line.

Collision Accuracy = **5**:
4 **additional** scans between the 2 dots which have **already detected a collision**.

Edit Mode Example Velocity

```
Vector2 TrajectoryAim.instance.editModeVelocity;
```

The velocity used to draw the **example trajectory line** in **Edit Mode**.

Scripting

The **Inspector settings** mentioned in the previous section can be referenced and modified through script. Under each variable there you can find it's corresponding coding reference.

In this section we will be going over **how to reference the Projectile and Trajectory Aim variables** as well as **additional script functionalities**.

How to Reference a Projectile Script

Just as you could reference any **GameObject** or **Component**, you can assign the Projectile through the **Inspector**.

If you already have a GameObject and script which you want to make a reference with, skip to Create a variable to reference the Projectile.

Create an Empty GameObject

Right click in any empty space of the **Hierarchy** window and select **Create Empty**.

Add a script to your new GameObject

1. Select the new Empty GameObject. In the **Inspector** window click **Add Component**.
2. In the search bar at the top of the **Add Component** window, type a name for your script. I will name mine "MyScript".
3. Click **New script**.
4. Click **Create and Add**.

Open the new script

The script should now be an added component in your **GameObject's Inspector**. In your GameObject's Inspector, double click on your new script's **greyed out name**.

Create a variable to reference the Projectile

Add the following code to your `public class` to create a `private` variable to reference the Projectile:

```
    [SerializeField]
    private Projectile projectile;
```

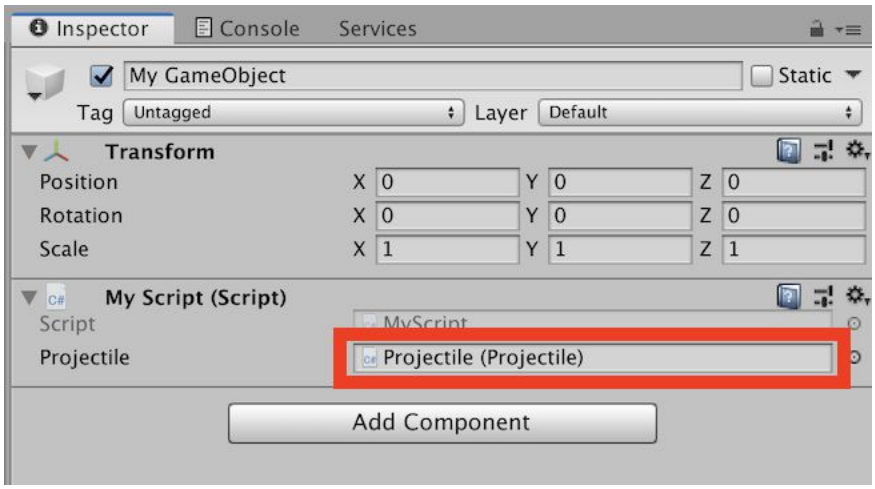
OR you can create a `public` variable to reference the Projectile by writing this instead:

```
    public Projectile projectile;
```

Assign the Projectile to the variable in the Inspector

Go back to the **Unity Editor** and select your new GameObject again. There should be a new variable in the script component.

Drag and drop the Projectile into the new box of the script component.



Or assign the Projectile reference through script

Instead of assigning through the Inspector, you can also reference the Projectile in your scene like this:

```
private void Start()
    {
        projectile = FindObjectOfType<Projectile>();
    }
```

This will assign the Projectile to the `projectile` variable at the start of the scene.

You can now reference the Projectile through your script.

Here's an example of a script which assigns the Projectile and uses the `SetSelectable(bool value)` function at the start of the scene to make the Projectile unselectable.

```
using UnityEngine;

public class MyScript : MonoBehaviour
{
    public Projectile projectile;

    private void Start()
    {
        projectile = FindObjectOfType<Projectile>();
        projectile.SetSelectable(false);
    }
}
```

How to Reference the Trajectory Aim Script

TrajectoryAim.instance

That's it. Just use `TrajectoryAim.instance` from any script to reference it. But that ease of use comes with one warning...

⚠ Warning

There should only be one Trajectory Aim script in a scene at any moment.

Script Functionalities

Here are some other functionalities in addition to the Inspector ones mentioned in the Inspector Settings section.

Projectile

`Projectile.SetSelectable(bool value)`
Makes the Projectile able/unable to be selected and aimed.

Example: After a shot in golf this would be set to false to prevent from shooting the ball while it's moving. When the golf ball stops moving, it's set to true.

`Projectile.Freeze()`
Freezes Projectile in place. Sets isKinematic to true on the Rigidbody2D.

`Projectile.Unfreeze()`
Takes the Projectile out of an aiming state. Sets isKinematic to false on the Rigidbody2D.

`Projectile.Shoot()`
Launches the Projectile by changing the Rigidbody 2D component's velocity to `shotVelocity`.

`Projectile.InitializeAim()`
Begins aiming.

`Projectile.Cancel()`
Cancels a shot while in the aiming state. Called when shot speed is below the Min Shot Speed.

`Vector2 Projectile.currentTouchPos`
The current world-space location of the mouse/touch.

`Vector2 Projectile.initialTouchPos`
The initial world-space location of the mouse/touch. In other words, the click location.

`Vector2 Projectile.shotVelocity`
The current velocity of an aimed shot.

`bool Projectile.aiming`
Whether or not the Projectile is being aimed.

`bool Projectile.shotSpeedAboveMin`
Whether or not the aimed shot speed is currently above the minimum amount.

Trajectory Aim

`TrajectoryAim.instance.Show()`
Makes the Trajectory Aim GameObject active in the hierarchy.

`TrajectoryAim.instance.Hide()`
Makes the Trajectory Aim GameObject inactive in the hierarchy.

`LayerMask TrajectoryAim.instance.layermask`
The layers to scan for collisions. It is automatically set to the colliding layers of the referenced Projectile. For more information on this, see the section on Collision Detection.

`bool TrajectoryAim.instance.autoUpdateLayerMask`
True by default. When true, the layerMask value will be updated automatically. The value of layerMask will be set to reflect the currently selected Projectile's Layer Collision Matrix. Set this to false if you want to assign your own value to layerMask and don't want it to change. For more information on this, see the section on Collision Detection.

`TrajectoryAim.instance.UpdateAim(Vector2 shotVelocity, Vector2 projectilePosition)`
Positions all the visible dots to draw a trajectory line given the velocity of the shot (shotVelocity) and the position of the projectile (projectilePosition).

`class TrajectoryAim.Dot()`

Used in the dotList to reference each dot in order to move and update the color of them.

Properties:

`GameObject gameObject`
`Transform transform`
`SpriteRenderer spriteRenderer`

Constructors:

`GameObject gameObject`

`List<TrajectoryAim.Dot> TrajectoryAim.instance.dotList()`

A list of all the Dots.

`bool TrajectoryAim.instance.editMode`

A boolean indicating whether the Unity Editor is in Edit Mode or not.

`TrajectoryAim.instance.VisualizeParabolaInEditMode(Vector2 shotVelocity, Vector2 projectilePosition)`

Positions all the visible dots to draw a trajectory line given the velocity of the shot (shotVelocity) and the position of the projectile (projectilePosition). It uses editModeVelocity by default.

Adding Your Own Code

Add to the scripts

You can easily add your own code to the TrajectoryAim and Projectile scripts if you want to produce a behavior specific to your needs.

Here's an example of what you can achieve by adding code to the **Projectile** script.

```
//Launches the Projectile
public void Shoot()
    {
        . . .

        //Add code to run when shooting a shot here.

        /* Example:
         *
         * Disable shooting a golf ball after a shot using:
         * SetSelectable(false);
         *
         * Then enable shooting again after the ball has stopped moving using:
         * SetSelectable(true);
         */
    }
```

Using callbacks

You can also use some built in **callbacks** to run a function from one of your own scripts. Here are the built in callbacks:

`TrajectoryAim.instance.OnProjectileChange()`
When the Projectile value changes.

`TrajectoryAim.instance.OnDotsChange()`
When the Dots value changes.

`TrajectoryAim.instance.OnDotsVisibleChange()`
When the dotsVisible value changes. This occurs due to the Dots value changing and due to a detected collision changing the number of dots that are visible.

`TrajectoryAim.instance.OnDotSizeChange()`
When the DotSize value changes.

`TrajectoryAim.instance.OnCollisionsChange()`
When the Collisions value changes.

`TrajectoryAim.instance.OnGradientChange()`
When the Gradient value changes.

Here's an example of a script that **changes the color of the Collision Indicator to match the currently selected projectile** in a scene with **multiple Projectiles**:

```
using UnityEngine;

public class CollisionIndicatorColorUpdater : MonoBehaviour
{
    //Collision Indicator SpriteRenderer
    private SpriteRenderer collisionIndicatorSR;

    private void Start()
    {
        collisionIndicatorSR =
        TrajectoryAim.instance.collisionIndicator.GetComponent<SpriteRenderer>();

        //Subscribe to the event of the Projectile value of the TrajectoryAim
        script changing
        TrajectoryAim.instance.OnProjectileChange += ProjectileChangeHandler;
    }

    //When a change occurs, the following function will run
    private void ProjectileChangeHandler(GameObject newVal)
    {
        //Get the new Projectile's Sprite Renderer
        SpriteRenderer newProjectileSR  newVal.GetComponent<SpriteRenderer>();

        //Change the Collision Indicator's color to match the new Projectile's
        collisionIndicatorSR.color = newProjectileSR.color;
    }
}
```

Collision Detection

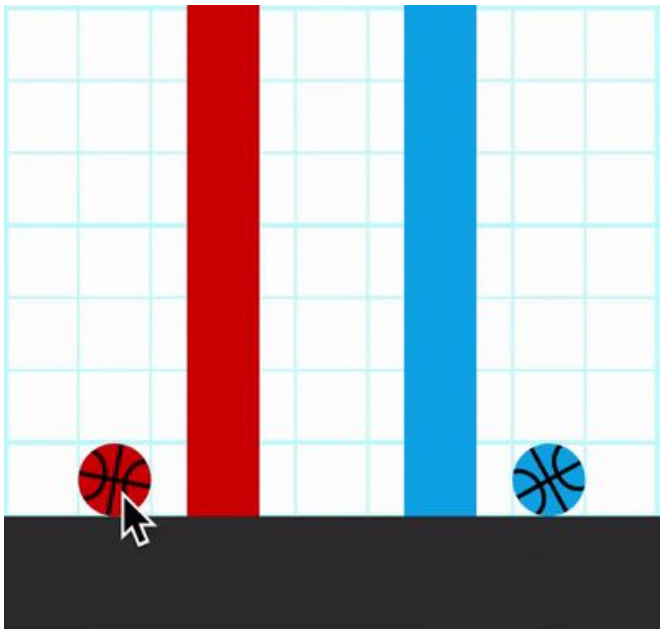
The Layer Collision Matrix is Used by Default

The trajectory Layer Mask adjusts to the currently selected Projectile's Layer Collision Matrix.

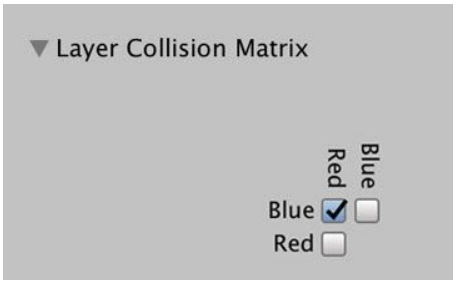
Whether you're using one or multiple Projectile's in your scene, the **TrajectoryAim** script checks a Projectile's **Layer** the moment it becomes selected and begins aiming. It then checks to see which layers are collidable with this Projectile by checking the **Layer Collision Matrix.** Once the collidable layers are found, the `TrajectoryAim.instance.layerMask` value is updated accordingly.

You can see the automatic Layer Mask adjustment in this example:

The blue ball and wall are on the **Blue layer**. The Red ball and wall are on the **Red layer**. Only colliders from **different** layers can collide with each other in this example, the **Layer Collision Matrix** below is responsible for this behaviour.



The TrajectoryAim script only shows a detected collision when the red ball is aimed at the blue wall and **automatically** changes its `layerMask` value when the blue ball is **selected** so that it can show a collision with the red wall.

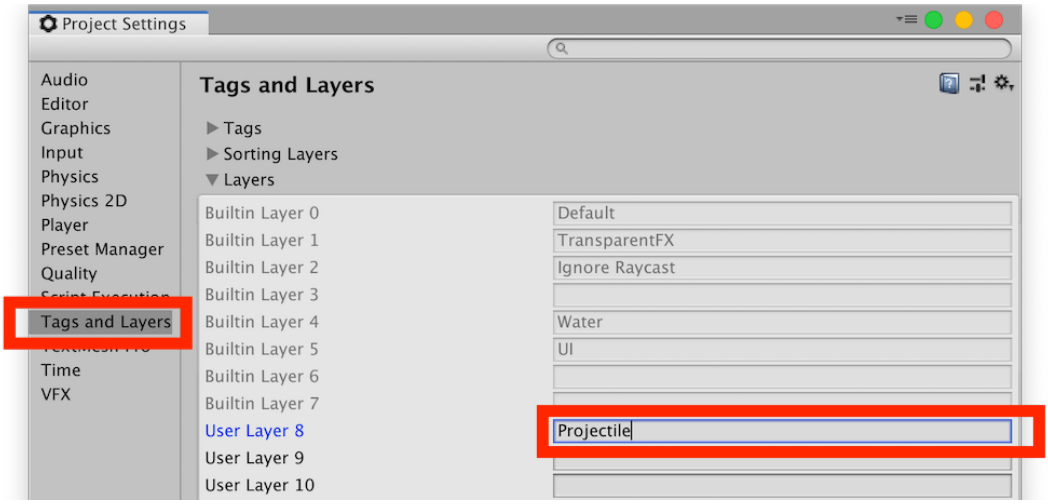


(Layer Collision Matrix simplified in image for clarity)

Layers

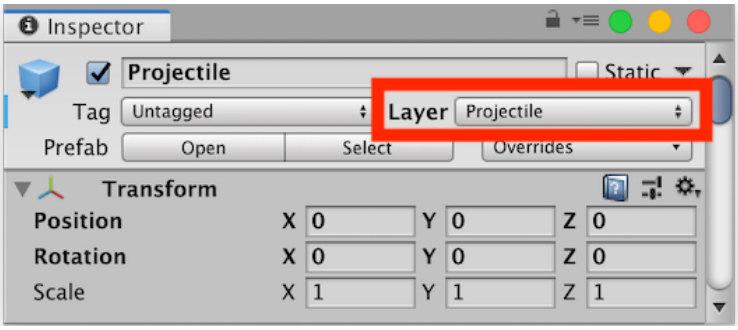
How to create a new Layer

Open **Edit** > **Project Settings** > **Tags and Layers**. In one of the empty layers, type in a name for the new layer.



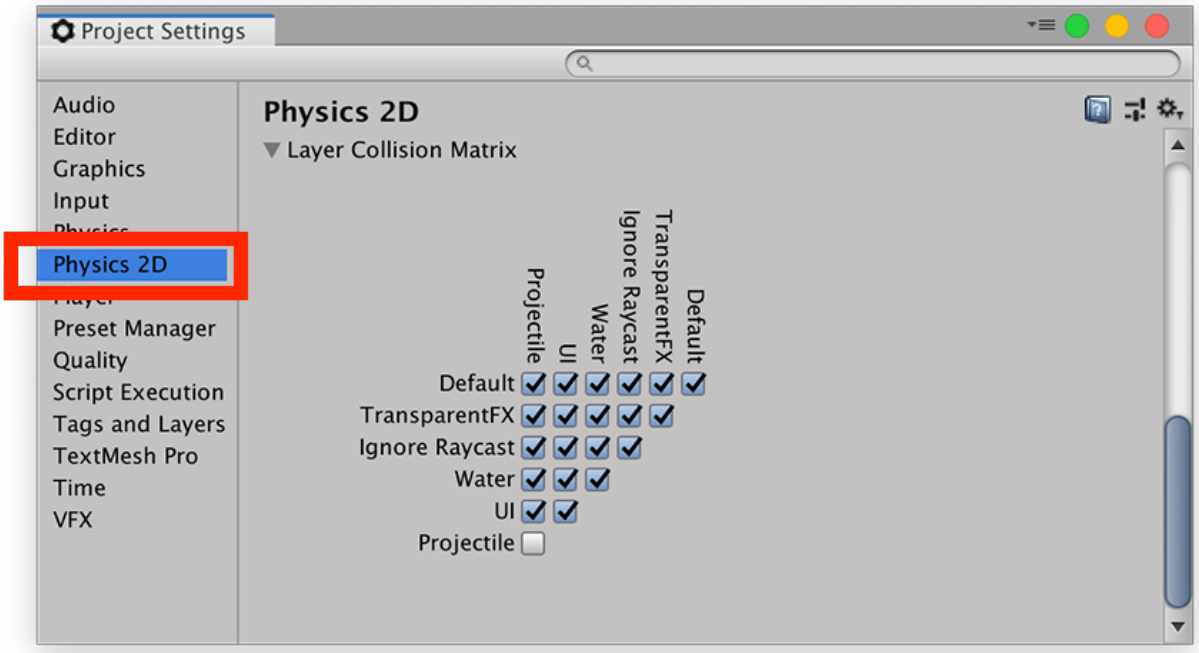
How to assign a Layer

In the **Inspector** window of a **selected GameObject**, click the **Layer** drop-down and select a layer.



How to use the Layer Collision Matrix

Go to **Edit** > **Project Settings** > **Physics 2D**. At the bottom you will see the **Layer Collision Matrix** with all the layers created in your project. Select which layers can collide with which by checking their corresponding toggle box.



Optimization Tip: It's recommended to put your Projectile(s) on their own layer and to **disable** collisions with their own layer. This increases efficiency because when the TrajectoryAim script scans for collisions, it can skip scanning the Projectile itself and having to rescan to see if any other Colliders were detected.

Overriding the Layer Collision Matrix

How to stop automatically updating layerMask

Here's how to disable the automatic behaviour of the layerMask value being changed to the collidable layers in the selected Projectile's Layer Collision Matrix:

```
TrajectoryAim.instance.autoUpdateLayerMask = false;
```

Manually Changing the Layer Mask

To make the TrajectoryAim script detect/ignore collisions differently than the Layer Collision Matrix's settings, you can change the `TrajectoryAim.instance.layerMask` value through script.

Everything

To detect collisions for every layer, use the following:

```
TrajectoryAim.instance.layerMask = ~0;
```

Everything except

To detect collisions for everything **except** a specific layer, use the **inverse operator** (~) to exclude a layer:

```
TrajectoryAim.instance.layerMask = ~LayerMask.GetMask("Layer Name");
```

You can also exclude multiple layers with this method:

```
TrajectoryAim.instance.layerMask = ~LayerMask.GetMask("Layer1", "Layer2",
"Layer3");
```

Only specified layer(s)

To detect collisions for only a specific layer, use the following:

```
TrajectoryAim.instance.layerMask = LayerMask.GetMask("Layer Name");
```

You can also include multiple layers with this method:

```
TrajectoryAim.instance.layerMask = LayerMask.GetMask("Layer1", "Layer2",
"Layer3");
```

Nothing

To stop detecting collisions, turn off Collisions like this

```
TrajectoryAim.instance.Collisions = false;
```