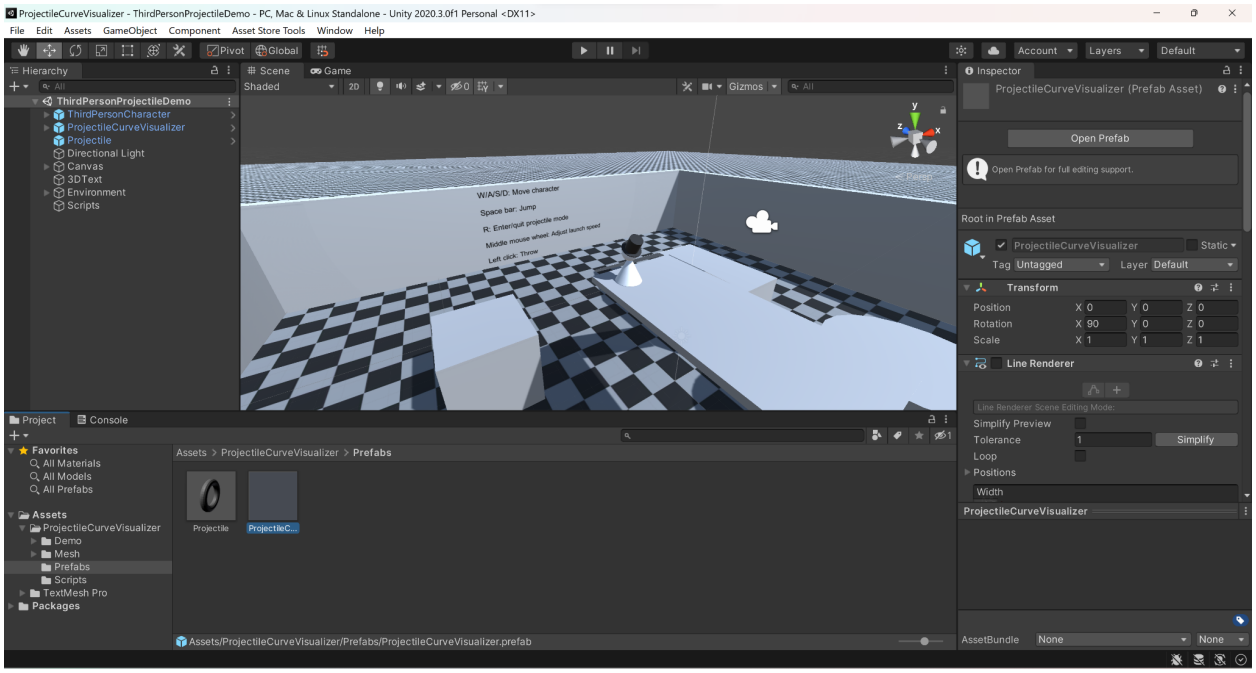
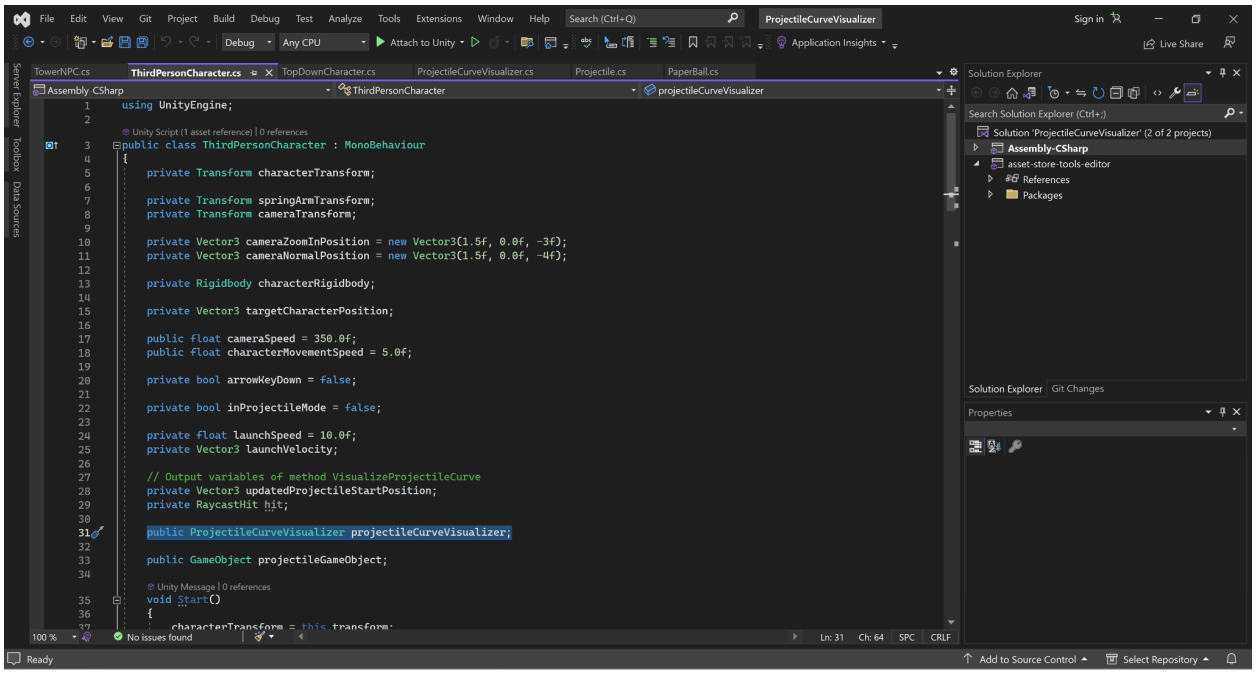
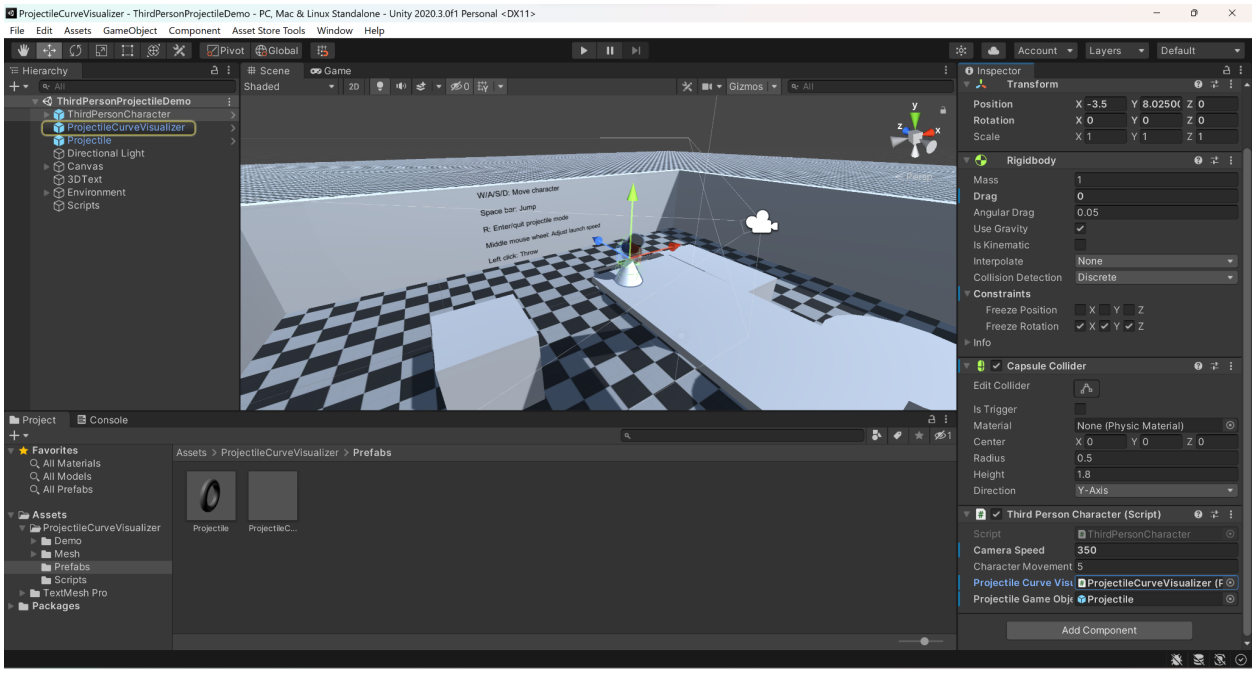
Projectile Curve Visualizer Documentation

Setup

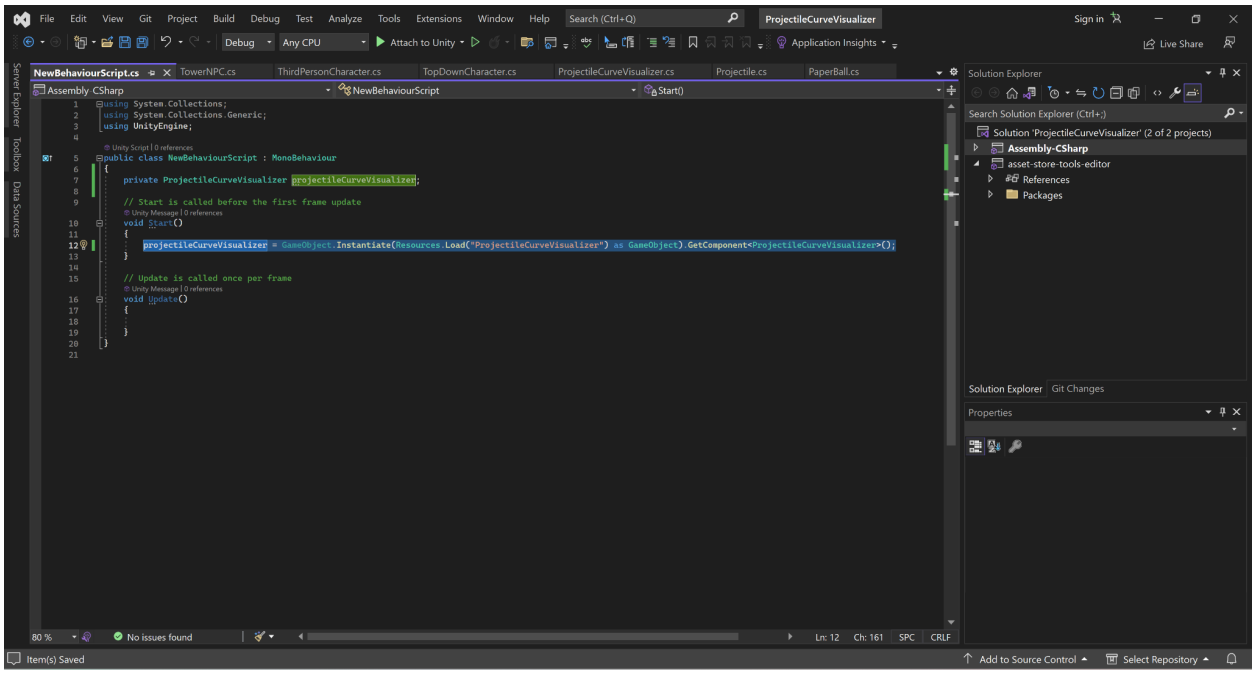
1. Import the plugin.
2. To create an instance of “ProjectileCurveVisualizer”, there could be 2 ways:
   (1) Navigate to Assets\ProjectileCurveVisualizer\Prefabs folder, drag the “ProjectileCurveVisualizer” prefab into the scene, declare a public variable with ProjectileCurveVisualizer type in first-person/third-person character C# script, on the Inspector panel drag the ProjectileCurveVisualizer prefab in Hierarchy and assign to the variable.







(2) Create “Resources” folder inside Assets folder, move the “ProjectileCurveVisualizer” prefab to Resources folder, use “GameObject.Instantiate(Resources.Load("ProjectileCurveVisualizer") as GameObject).GetComponent<ProjectileCurveVisualizer>()” method in C# to spawn an instance and get the ProjectileCurveVisualizer class object reference.



3. Depending on the need, either the function "VisualizeProjectileCurve" or "VisualizeProjectileCurveWithTargetPosition" function can be called. If the curve needs to be calculated per frame, it should be placed in Update() method or other update function.
    (1) VisualizeProjectileCurve: The target position is unknown and launch velocity is adjustable. Usually for first-person mode or third-person mode.

    Example of VisualizeProjectileCurve function

```
private float launchSpeed = 10.0f;

public ProjectileCurveVisualizer projectileCurveVisualizer;

// Output variables of method VisualizeProjectileCurve
private Vector3 updatedProjectileStartPosition;
private RaycastHit hit;

void Start()
{
        characterTransform = this.transform;
}

void Update()
{
```

```
        projectileCurveVisualizer.VisualizeProjectileCurve(characterTransform.
        position, 1.0f, characterTransform.forward * launchSpeed, 0.25f, 0.1f,
        true, out updatedProjectileStartPosition, out hit);
}
```

Input parameters

projectileStartPosition	The world position of the projectile start point.
projectileStartPositionForwardOffset	This will add an offset towards the target position. If the value is 100, the actual projectile start position will be 100 units ahead. Adding an offset can make sure the projectile is not too close to the character, then collides with the character.
launchVelocity	It includes the speed and direction.
projectileRadius	The radius of projectile mesh.
distanceOffsetAboveHitPosition	This will add an offset towards the hit normal direction for the target mesh(the mesh at the end of the curve, with the target icon texture). Sometimes setting the target mesh exactly the same as the hit position will cause Z-fighting, giving it a small offset can fix the issue.
debugMode	For debug use, to show the Raycast paths of the projectile curve. In Scene view, Raycast paths are green and hit normal is red.

Output parameters

updatedProjectileStartPosition	The actual start location, result of projectileStartPosition + ProjectileStartPositionForwardOffset.
hit	The result of RaycastHit, contains information like hit.point and hit.hitNormal.

(2) VisualizeProjectileCurveWithTargetPosition: The target position is known, launch velocity needs to be calculated. Usually for top down mode.

Example of VisualizeProjectileCurveWithTargetPosition function

```
private float launchSpeed = 10.0f;

private Ray ray;
private RaycastHit mouseRaycastHit;

private bool canHitTarget = false;

public ProjectileCurveVisualizer projectileCurveVisualizer;

// Output variables of method VisualizeProjectileCurveWithTargetPosition
private Vector3 updatedProjectileStartPosition;
private Vector3 projectileLaunchVelocity;
private Vector3 predictedTargetPosition;
private RaycastHit hit;

void Start()
{
        characterTransform = this.transform;
}

void Update()
{
        if (Physics.Raycast(ray, out mouseRaycastHit))
        {
           canHitTarget =
        projectileCurveVisualizer.VisualizeProjectileCurveWithTargetPosition(ch
        aracterTransform.position, 1.5f, mouseRaycastHit.point, launchSpeed,
        Vector3.zero, Vector3.zero, 0.05f, 0.1f, true, out
        updatedProjectileStartPosition, out projectileLaunchVelocity, out
        predictedTargetPosition, out hit);

           if(canHitTarget)
           {
              // Throw
           }
        }
}
```

Input parameters

projectileStartPosition	The same as described in the previous function(see above).
projectileStartPositionForwardOffset	The same as described in the previous function(see above).
projectileEndPosition	The target position to be hit.
launchSpeed	The larger the value, the further hitable position will be.
throwerVelocity	The velocity of the thrower. If the thrower is not moving, just input Vector3.zero. The velocity needs to be calculated manually, velocity = deltaDistance / Time.deltaTime.
targetObjectVelocity	The velocity of the target object. If the target is not moving, just input Vector3.zero. Similarly, the velocity needs to be calculated manually, velocity = deltaDistance / Time.deltaTime.
projectileRadius	The same as described in the previous function(see above).
distanceOffsetAboveHitPosition	The same as described in the previous function(see above).
debugMode	The same as described in the previous function(see above).

Output parameters

updatedProjectileStartPosition	The same as described in the previous function(see above).
projectileLaunchVelocity	The necessary velocity in order to hit the target position. This velocity can be used when throwing a projectile.
predictedTargetPosition	If throwerVelocity or targetObjectVelocity is not (0, 0, 0), it will predict the possible target position at the next moment, based on the velocity information. If both thrower and target are static, the

	predictedTargetPosition would be exactly the target position(no prediction).
hit	The result of RaycastHit, contains information like hit.point and hit.hitNormal.

4. Call “HideProjectileCurve” function to hide the curve when not needed.

   Example of HideProjectileCurve

   ```
   projectileCurveVisualizer.HideProjectileCurve();
   ```

5. Optional: The Assets\ProjectileCurveVisualizer\Prefabs folder contains a prefab “Projectile” for the actual projectile mesh to be thrown.

   To throw a projectile: Spawn a prefab “Projectile”, set its position the same as updatedProjectileStartPosition, then call the “Throw” function from Projectile and pass:
   - launchVelocity(if using VisualizeProjectileCurve function)
     or
   - projectileLaunchVelocity(if using VisualizeProjectileCurveWithTargetPosition function).

     Example of VisualizeProjectileCurveWithTargetPosition function

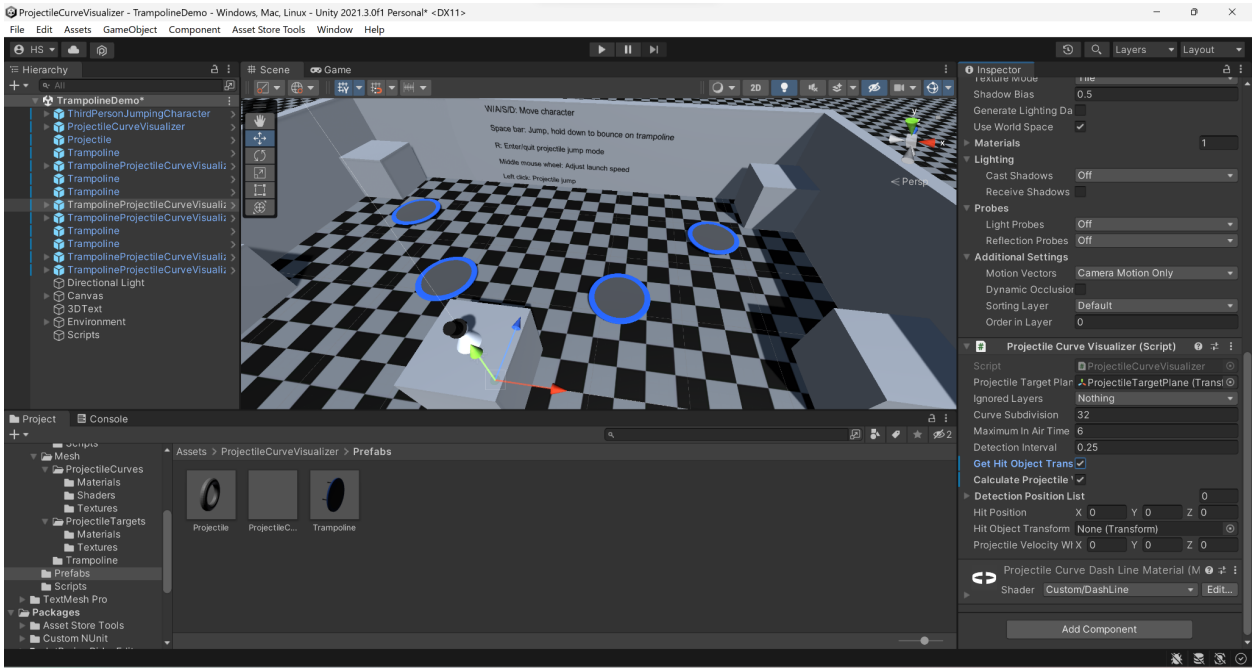
```
private float launchSpeed = 10.0f;

private Ray ray;
private RaycastHit mouseRaycastHit;

public ProjectileCurveVisualizer projectileCurveVisualizer;

// Spawn a projectile and move to the start position
Projectile projectile =
GameObject.Instantiate(projectileGameObject).GetComponent<Projectile>();
projectile.transform.position = updatedProjectileStartPosition;

// Throw projectile
projectile.Throw(launchVelocity);
```

Trampoline

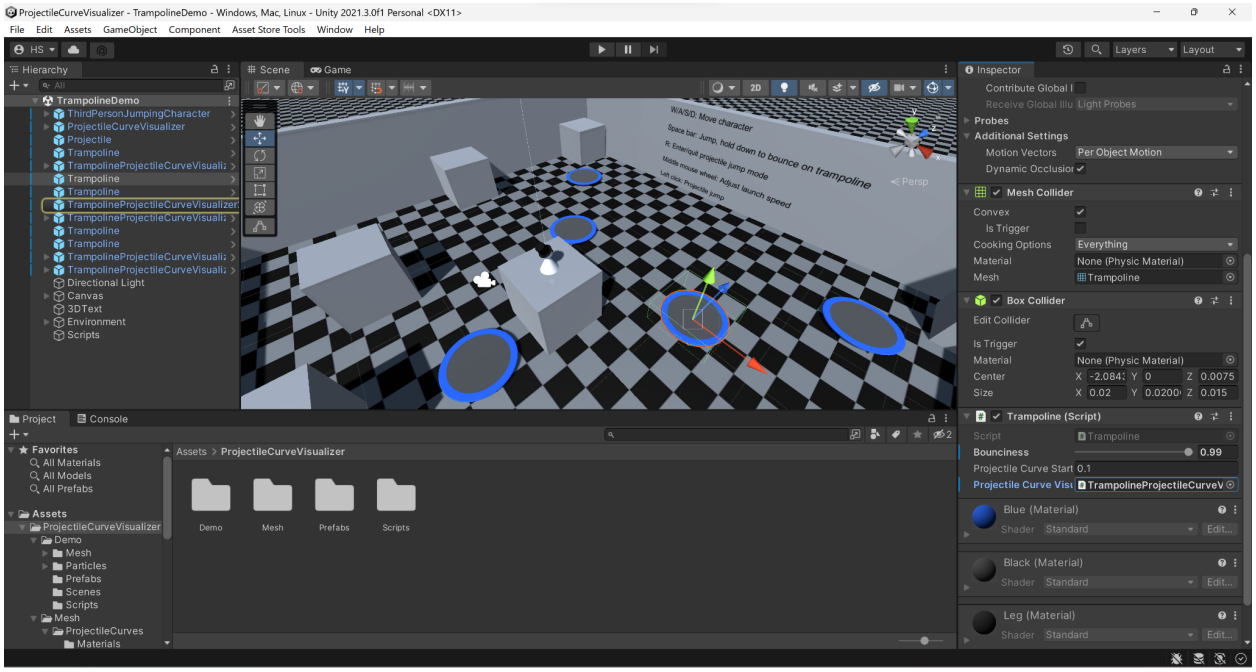
Utility: Character can jump on it and bounce, it can bounce higher by holding down spacebar to give an extra force when landing on the trampoline. Multiple trampolines can be placed in a line to allow the character to jump and bounce to move forward on each trampoline, with the visualization of predicted curves. Plus, if the character throws a projectile to a trampoline, it will bounce as well.

The attribute bounciness(range: 0 ~ 1) controls how bouncy the trampoline is. If bounciness is 0.5, when a character jumps from a 10 meters high place and lands on the trampoline, it will bounce to 10* 0.5 = 5 meters high for the first bounce, second bounce is 0.5 * 5 = 2.5 meters, and so on.

Navigate to Assets\ProjectileCurveVisualizer\Prefabs folder, drag the “Trampoline” prefab into the scene.



To make the trampoline able to visualize a predicted curve, a ProjectileCurveVisualizer prefab is needed. Drag the “ProjectileCurveVisualizer” prefab into the scene, tick “GetHitObjectTransform” and “CalculateProjectileVelocityWhenHit”.



Drag the ProjectileCurveVisualizer object and assign to the corresponding slot of the Trampoline script.

Currently the way to check if the hit object is a trampoline is by comparing name == “Trampoline”, so each trampoline object should be named as “Trampoline” to meet the condition. A better way is by comparing the tag(https://docs.unity3d.com/Manual/Tags.html), so the following lines of code can be modified:

ThirdPersonJumpingCharacter.cs line 172: if (projectileCurveVisualizer.hitObjectTransform.name == "Trampoline")

Trampoline.cs line 56: if (projectileCurveVisualizer.hitObjectTransform.name == "Trampoline")

Notice: The default Trampoline 3D model was made in Blender, so it has different rotation in Unity(-90.0f, 0.0f, 0.0f). This will affect the calculation of reflection vector for the bounce, for this model “trampolineTransform.forward” represents up vector. If this 3D model is replaced by a model with rotation of (0.0f, 0.0f, 0.0f), “trampolineTransform.forward” should be replaced by ““trampolineTransform.up”” for the following line of code:

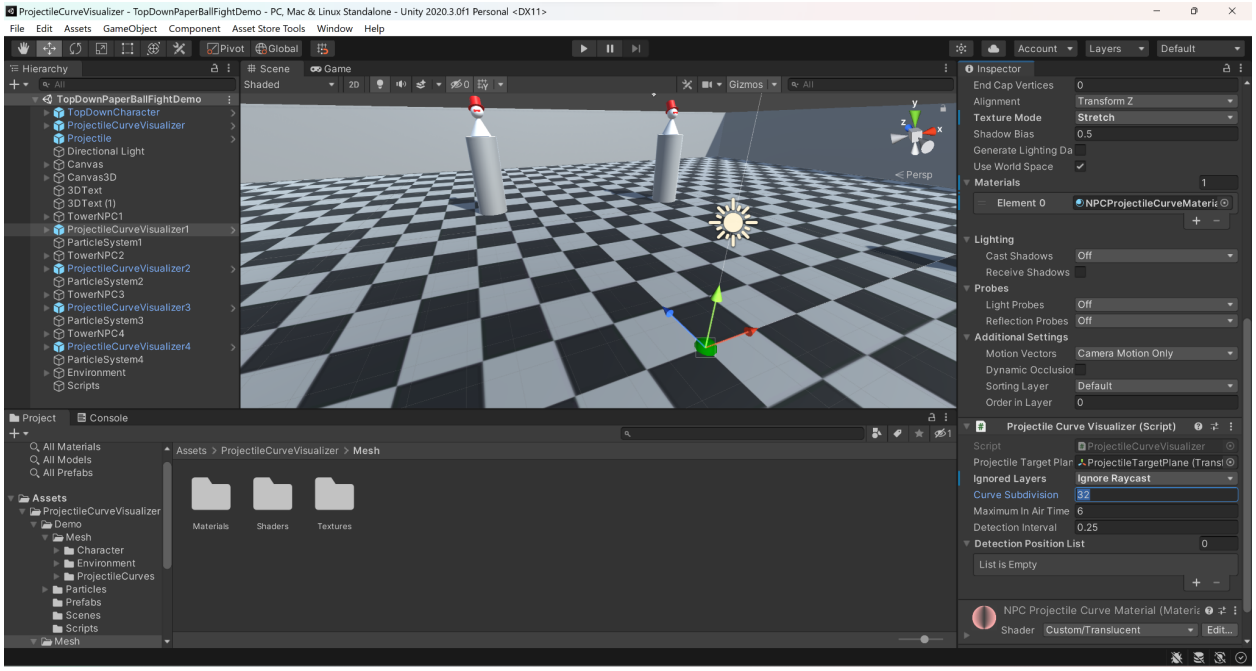
Trampoline.cs line 84: return Vector3.Normalize(incidenceVector - 2f * Vector3.Dot(incidenceVector, trampolineTransform.forward) * trampolineTransform.forward) * incidenceVectorLength;

Customization

Projectile Curve Attributes

There are 4 adjustable attributes in the ProjectileCurveVisualizer script:

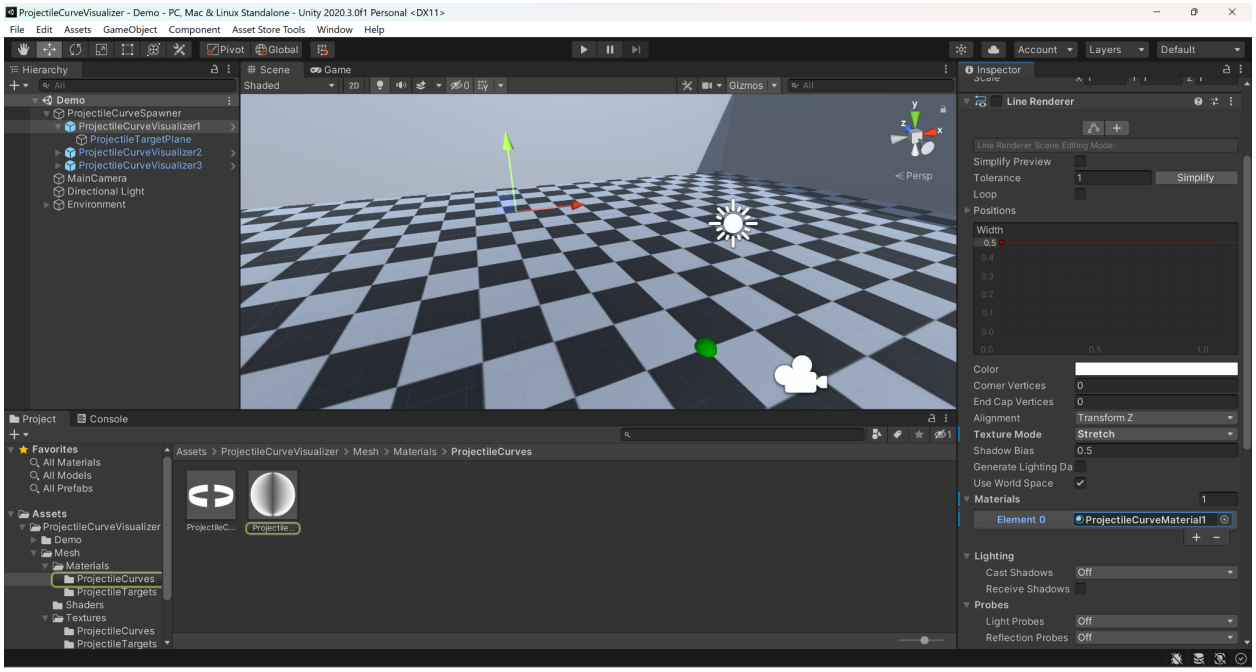
- IgnoredLayers: The layers to be ignored. If the Water layer is selected, the projectile curve won't hit any object on that layer.
- CurveSubdivision: The higher the smoother the curve will be. If the value is 100, there will be 100 vertices for the curve(LineRenderer).
- MaximumInAirTime: Maximum projectile travel time in seconds. If the value is 8, then the trail of the first 8 seconds will be rendered, the part greater than 8 will be ignored. For the consideration of performance, it can be limited below 10 in most of the cases.
- DetectionInterval: The frequency of physics detection in seconds. If the value is 0.25, it will calculate the projectile position every 0.25 second, and perform Raycast to check if there is an obstacle blocked between them. Limiting this value can save some performance. Normally values between 0.1 and 0.25 are sufficient for most of the cases.



Projectile Curve Material

The textures and colors of the curve mesh and hit target mesh are adjustable, even the entire material can be replaced based on personal preference. Similarly, the actual projectile mesh can be customized.

Materials and textures are located in Assets\ProjectileCurveVisualizer\Mesh and Assets\ProjectileCurveVisualizer\Demo\Mesh\ProjectileCurves\Materials. By default, the Texture Mode of LineRenderer should be “Tile” to match the DashLine material, but it can be changed to “Stretch” to match the style of different materials. The width of the LineRenderer can be adjusted directly on the Inspector panel.



Size

To reduce the size, the Demo folder in Assets\ProjectileCurveVisualizer folder can be completely removed without affecting any functionality of this plugin.

The extra materials, textures and shaders in Assets\ProjectileCurveVisualizer\Mesh folder can be removed if they are not needed, only the necessary materials for the projectile curve and target mesh remain.

Universal Render Pipeline (URP) & High Definition Render Pipeline (HDRP)

The materials can be converted by following this tutorial:
https://www.youtube.com/watch?v=aJ1OpirisGM

Explanation

VisualizeProjectileCurve Function

The physics test is done by calling built-in function Physics.Raycast() to check if there is an obstacle blocked between each detection position pair and get sampling points, after that the visual part(LineRenderer) will be rendered to match the points.

A fast approximation formula is used to calculate the control point of the Bézier curve:

controlPointX = 2 * thirdPointOnCurveX - startPointX/2 - endPointX/2;
controlPointY = 2 * thirdPointOnCurveY - startPointY/2 - endPointY/2;

Source:
https://stackoverflow.com/questions/22237780/how-to-model-quadratic-equation-using-a-bezier-curve-calculate-control-point

https://jsfiddle.net/m1erickson/6jNCM/

The limitation is that, despite the 100% accuracy of start point and end point, the curve body is not always correct. How accurate it is depends on how close the third point is to the middle point.

VisualizeProjectileCurveWithTargetPosition Function

The prediction(heuristic) is executed for the situation of moving thrower or moving target, or both.

Possible target position = current target position + target object velocity * projectile travel time + opposite direction of thrower velocity * projectile travel time

Source:
https://www.youtube.com/watch?v=mHofF0ddts0

Example: See the TopDownMovingThrowerDemo scene in Assets\ProjectileCurveVisualizer\Demo\Scenes folder. If the thrower is on a platform moving to the right, the target is static, then it should aim at the left of the target instead of aiming exactly at the target in order to cancel out the relative movement.

If the thrower is on a moving platform, it needs to manually calculate the velocity for the thrower object. Velocity = delta distance / delta time. See the example of TopDownMovingThrowerDemo scene, TowerNPC C# script and TopDownCharacter C# script.

The prediction provides certain accuracy, but still difficult to hit the target if the movement speed is too fast or the target has irregular movement. Part of the reason is due to the nature of projectile, once it is launched, the hit position is fixed, no change can be made.