Nguyen D. & Burdea G. Virtual Reality Laboratory Manual, 3rd Ed.

LAB FOUR

IV

Title

Requirements

1. **Unity 3D** (using version 2018.4+)
2. **Blender** (using version 2.8+)
3. **Source-code Editor** (of your choice)

Contents

Immersion, Interaction, Imagination

These are the three pillars of Virtual Reality (also called the "Virtual Reality Triangle"). Over the past few Laboratory Assignments, you learned more and more VR programming skills. You learned how to create 3D environments to immerse the user, how to allow the user to interact and engage with objects around them, and how to make the user imagine that they are part of an alternate reality.

[redacted] In this Lab, we will cover landscape building, wind, particle effects, collision detection & trigger scripting.

Terrains ❓

Unity's Terrain system allows you to add vast landscapes to your scenes. There are many different types of tools, brushes, and settings that allow you to sculpt and model a landscape to your liking. Unity Terrain is highly optimized for rending at run time as well!

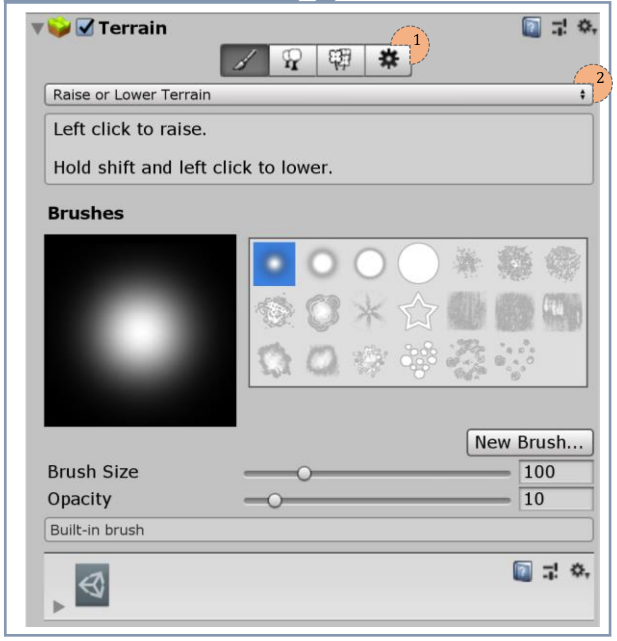
It is recommended to have the Standard Assets Environment files which have terrain assets, trees, and water for you to use and experiment with.

First, create a terrain GameObject, which is listed under 3D Object, and add it to your scene. It is very large, so you may need to zoom out! It may be useful to add a few temporary cubes or objects on top of the terrain to get a sense of its scale. You may think the hill you are making is small, but when you place a first-person controlled character inside, it will look massive!

Of course, you can change the size and resolution of the terrain mesh in the terrain general settings❓. To do so click the cog icon in the inspector window when the terrain is selected. However, you cannot change the scale of its Transform component; this will not work!

Look at the inspector window with the terrain selected. You should be able to observe the terrain component attached as seen in **Figure 1**. This is where all the tools you can use are located.

Terrain Component



Left click to raise.

Hold shift and left click to lower.

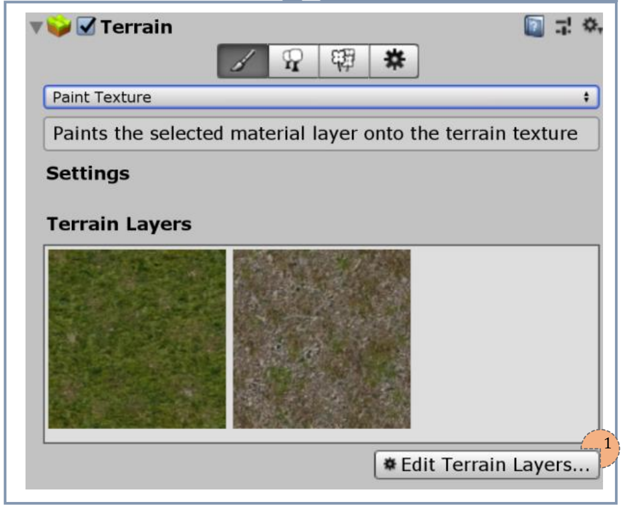
Terrain component showing brush options. Figure 1

In **[Figure 1, #1]**, you can see a toolbar where you can create adjacent Terrain tiles, sculpt and paint your Terrain, add trees, add details such as grass, flowers, and rocks, and change general settings for the selected Terrain. You can hover over the icons if you are unsure what they are used for.

The **Paint Terrain** tool ❷ has a drop-down menu seen in **[Figure 1, #2]**. This drop down has an option for adding adjacent terrain, terrain sculpting, and terrain painting options.

It may be hard to model curvature with a blank white terrain, so we will first apply texture to it. Using the drop-down menu, go to the “Paint Texture” option.

Paint Texture



Paints the selected material layer onto the terrain texture

Paint texture tool. Figure 2

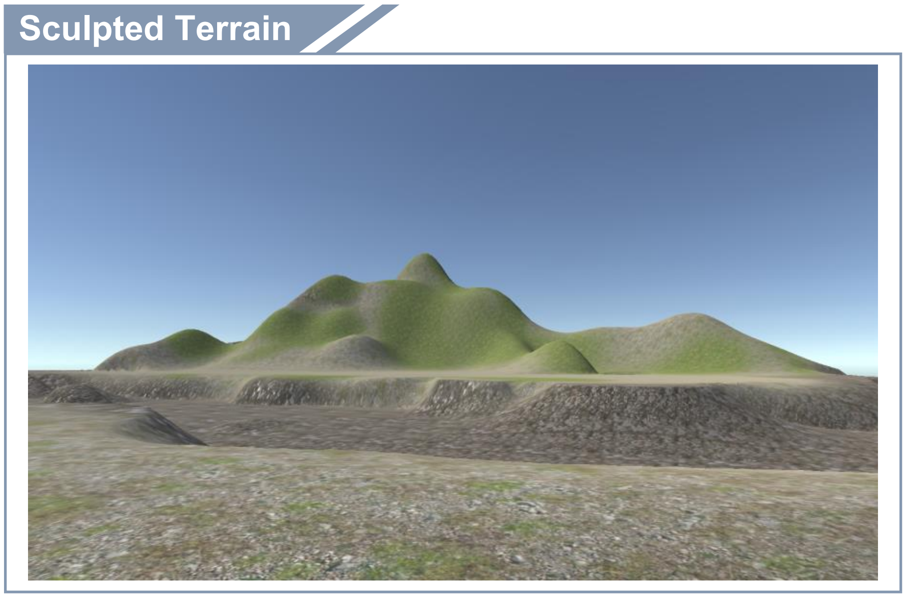
You will not see anything in this tab at first. This is because you need to add **Terrain Layers**. Click on “Edit Terrain Layers” shown in **[Figure 2, #1]** and click on Create Layer. This will create a new layer asset that you will be able to see in your Asset folder.

If you are using the Standard Assets package, find the texture called “GrassHillAlbedo”. There are other texture files that it provides as well, all of which you can find in a folder called “SufaceTextures”. Once you add the first texture, the entire plane will be painted in that chosen texture. Let’s add one more texture called “GrassRockyAlbedo”. We will use this texture to paint on top of the other texture (a process called multi-texturing).

Click on the rocky texture and select any brush and size you want. Then, you should be able to go to the scene window and paint on top of the terrain. You can also click on the grass hill texture and paint with that too.

Now that we have a texture on top of the terrain, we can start sculpting. Go to the drop-down menu again and select “Raise or Lower Terrain”. It will tell you that left click is to raise and shift left click is to lower. Try painting around on the terrain now. Play around with different brush types, sizes, and opacity. There are other terrain sculpting tools you can use as well, such as set height, smooth height, and stamp terrain. Feel free to experiment with those tools as well.

If you need more space, you can add adjacent terrain meshes with the “Create Neighbor Terrains” option.

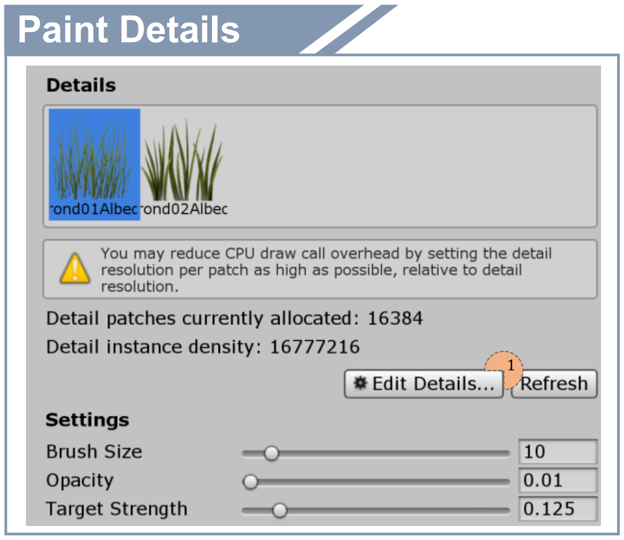


Sculpted terrain with hills, mountains, and valleys. Figure 3

Now that your terrain is sculpted, we can add plants and trees on top of the terrain. Let's start with the **Paint Details** ❷ tool. You can use this tool to paint patches of grass, flowers, or small rocks. In general, this is used for painting in small objects covering the surface of your terrain.

Again, you will not see anything in this tab at first. This is because you need to add Details. Click on "Edit Details" shown in **[Figure 4, #1]** and click on Add Grass Texture. This will open a window where you can select your grass texture along with other settings.

If you are using the Standard Assets package, find the texture called "*GrassFrond01Albedo*". There is another type of grass texture called "GrassFrond02Albedo". After you added both grass texture types, you can always go back to edit them. This will also change all the grass that you have already placed down. You can change their min/max height and width as well as color.



You may reduce CPU draw call overhead by setting the detail resolution per patch as high as possible, relative to detail resolution.

Detail patches currently allocated: 16384
Detail instance density: 16777216

Brush Size 10
Opacity 0.01
Target Strength 0.125

Paint details tool. Figure 4

Once you added the details you want, select your desired brush and start painting in the grass. This grass is also able to react to wind. We will go over wind zones in a later section of this lab.

You may notice that if you move too far away from the grass it will disappear! This is just the rendering distance coming into play, so it does not hinder the performance.

Try out different colors, densities, and use your own grass textures! Use the settings sliders to fine tune how they are painted onto your terrain.

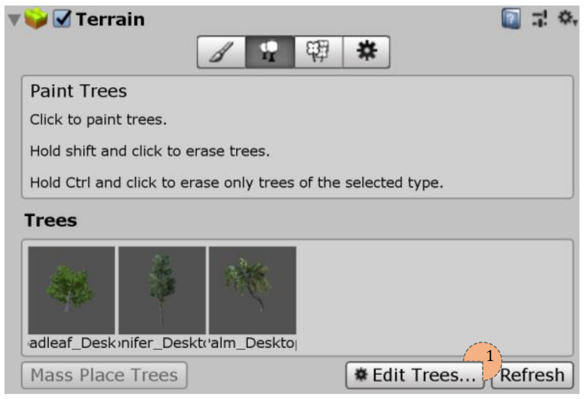
Trees ❓

The last tool you can use on terrain is the **Paint Trees** tool. With it, you can channel your inner Bob Ross and paint happy little trees!

As before, you must first add trees before you can use the paint tool. The Standard Asset package provides 3 types of trees: *Broadleaf*, *Conifer*, and *Palm*.

After you add them, you can change multiple settings including brush size, tree density, and tree height. Then, start painting in your trees!

Paint



Paint Trees
Click to paint trees.
Hold shift and click to erase trees.
Hold Ctrl and click to erase only trees of the selected type.

Paint trees tool. Figure 5

Trees & Grass

Small scene with trees, grass, and textured terrain. Figure 6

And just like that, you can get a nice natural looking landscape in no time at all! Of course, you are encouraged to customize and use your own textures and models for this and not solely rely on the Standard Assets.

Once we add a Wind Zone, the trees will also move and sway in the wind.

With that in mind, we can make our own custom trees with Unity's tree creation system and move onto creating Wind Zones.

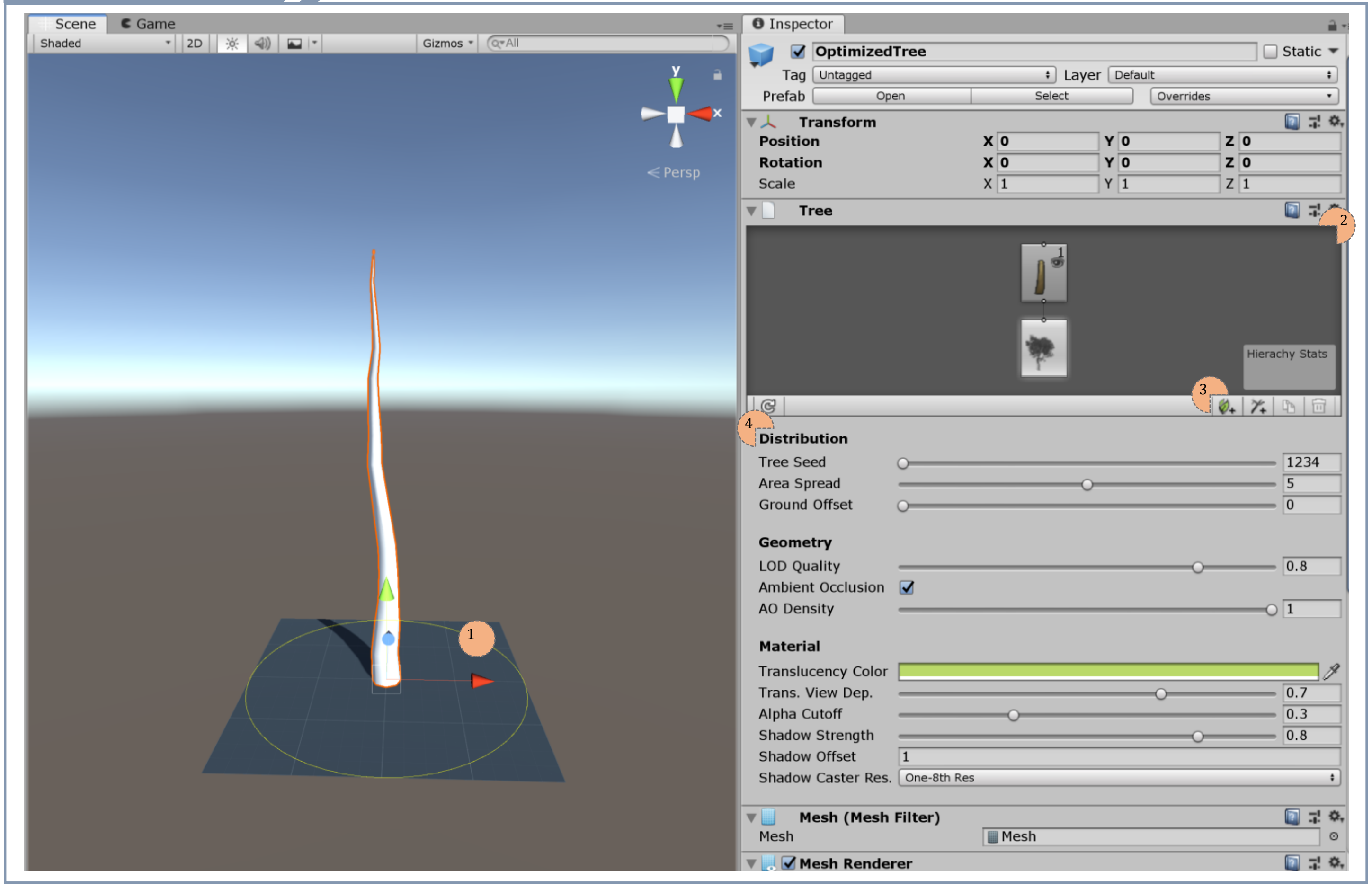
Custom Trees ❓

Unity provides a tool called **Tree Editor** which allows you to design a tree directly in the editor. To begin, create a GameObject called Tree which is under 3D Objects folder. You should now see one long branch in the middle of the scene, as seen in **[Figure 7, #1]**. In the Inspector window, you should be able to see the Tree component with all its different options and the Tree Hierarchy window **[Figure 7, #2]**. There are two nodes present: a root node and a branch node. The root node is the tree's origin point. All other nodes are children of the root node, forming a hierarchy.

To add more nodes, you must first select an existing node on the hierarchy, so to place a new node on top of it. Then click either the *add leaf* or *add branch* buttons as shown in **[Figure 7, #3]**. There are also buttons to duplicate and delete next to them.

For each node, there are different settings, which you can change to modify the orientation, size, frequency, and distribution below the hierarchy **[Figure 7, #4]**.

Custom Tree Start



OptimizedTree — Static
Tag: Untagged — Layer: Default
Prefab: Open — Select — Overrides

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Tree

Distribution

Tree Seed	1234
Area Spread	5
Ground Offset	0

Geometry

LOD Quality	0.8
Ambient Occlusion	✓
AO Density	1

Material

Translucency Color	🟩
Trans. View Dep.	0.7
Alpha Cutoff	0.3
Shadow Strength	0.8
Shadow Offset	1
Shadow Caster Res.	One-8th Res

Mesh (Mesh Filter)
Mesh: Mesh
Mesh Renderer

The beginnings of a custom tree using the Unity Tree Editor. Figure 7

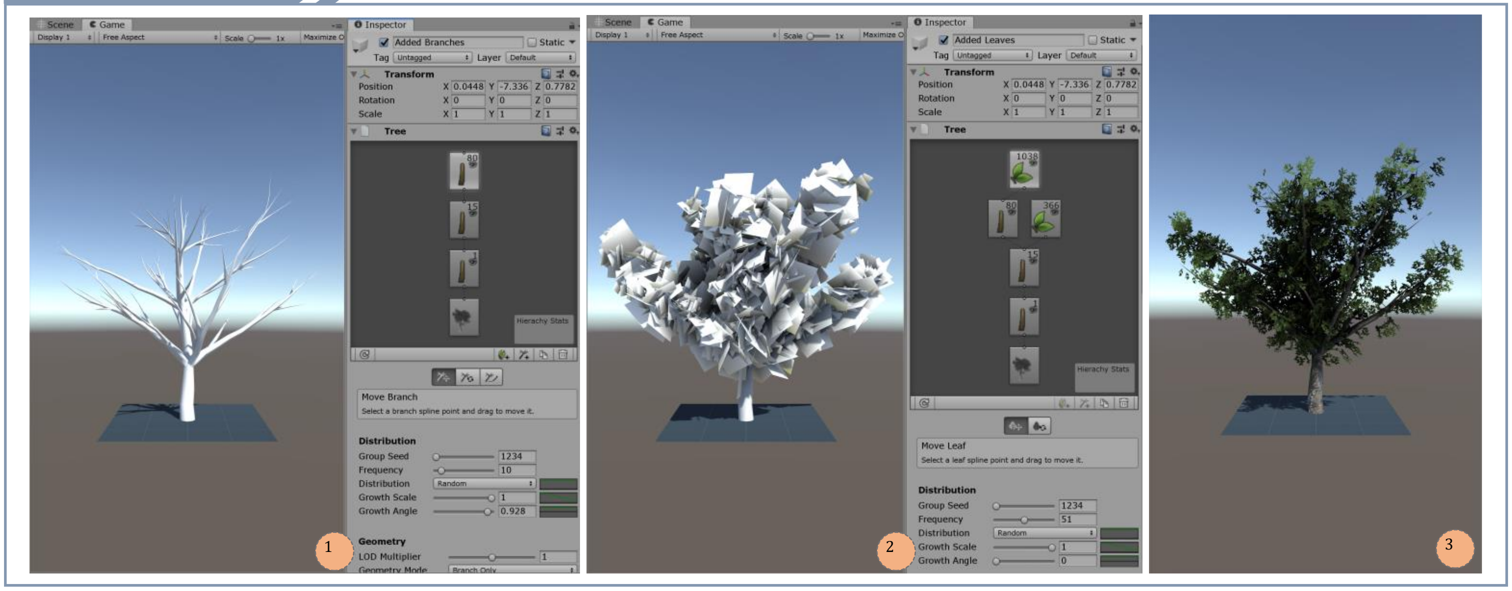
Let's try making a simple tree. First, we want to add a new branch node on top of the current branch node. The current branch node will act as the trunk of the tree. We will now add the branches that stem from the central trunk. When a node is attached higher up in the hierarchy, it will protrude out of its parent's mesh. You can increase the number of branches by increasing the frequency slider and you can see how it generates out around the trunk.

Because the branches and leaves are procedurally generated, there is a lot of randomization involved. However, you have a lot of control over the generation of the tree using different sliders below the hierarchy window.

Try changing the growth angle and distribution until you get the branches the way you like them. Then, you can add another branch node on top of that to extrude smaller branches off the main branches! You should get something like the tree shown in **[Figure 8, #1]**.

When selecting one of the nodes, you may see squares and circles appear on the selected node's branches or leaves. These are spline points where you can manually move and adjust the branches and leaves to your liking. But be careful, because if you want to go back to procedural generation for that specific node, you will lose all the manual adjustments you made.

Tree Steps



Added Branches — Static
Tag: Untagged Layer: Default
Transform
Position X 0.0448 Y -7.336 Z 0.7782
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Move Branch
Select a branch spline point and drag to move it.
Distribution
Group Seed 1234
Frequency 10
Distribution Random
Growth Scale 1
Growth Angle 0.928
Geometry
LOD Multiplier 1
Geometry Mode Branch Only

Added Leaves — Static
Tag: Untagged Layer: Default
Transform
Position X 0.0448 Y -7.336 Z 0.7782
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Move Leaf
Select a leaf spline point and drag to move it.
Distribution
Group Seed 1234
Frequency 51
Distribution Random
Growth Scale 1
Growth Angle 0

(1) Adding branches, (2) Adding leaves, (3) Adding textures. Figure 8

After you are satisfied with the layout of your custom branches, you can add leaves to them. Add leaves to the hierarchy as you did before with branches. You should end up with a bunch of untextured squares (as seen in Figure 8, #2). Again, you can mess around with the distribution and size settings until you are happy with the result.

Now you can add textures/materials to the tree to complete its appearance. Click on either a branch node or a leaf node and find the Geometry section. Here you should be able to find parameters for adding materials. Find and select the material of your choice. The Shader this material uses must be the correct one for the Tree Editor. If the material is not using the correct shader, it will give you a warning and allow you to change it with the click of a button.

Once you have your textures in place you have created your own custom tree! There are endless possibilities and we encourage you to really try different combinations of nodes and settings. (Note that your tree casts a shadow on the ground, unlike monolithic textures of trees).

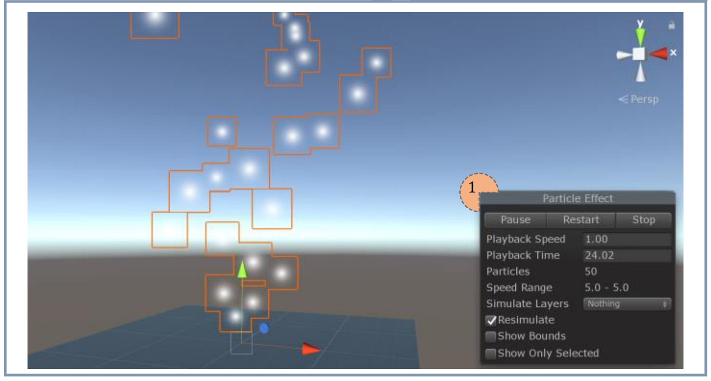
Particles & Effects ❓

Particles are a fantastic way to add cool effects and details to your scene. You can use particles for fire, smoke, magic, bubbles, water, dust, explosions, and more! There are a lot of settings and parameters when using particles and we cannot cover everything in this Lab in a general manner. Instead, we will be doing a tutorial on how to make a campfire with smoke rising from it. This should teach you the basics, however you will need to learn more about particle effects on your own.

First start by creating the particles GameObject which you can find under "**Effects**". This will create a default particle system that spews out circles in a cone. The particle system can play/animate if you have it selected. You can pause or restart it at any time using the controls inside the Scene window as seen in **[Figure 9, #1]**.

For a fire effect, we want to use a cone shape. We want the cone to be narrower than its current size, so we will reduce the Angle down to around 5 and the Radius down to 0.5. We can keep the rest of the settings the same.

Particle



Particle Effect

Pause | Restart | Stop

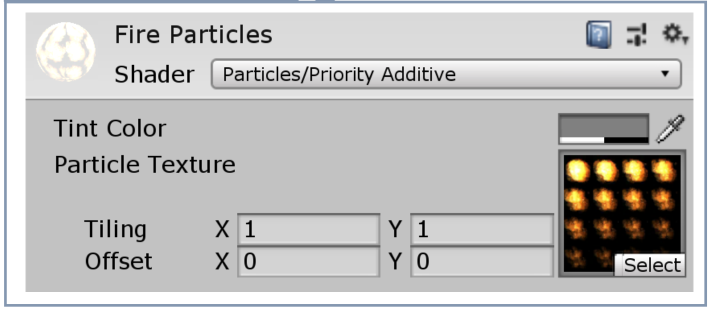
Playback Speed	1.00
Playback Time	24.02
Particles	50
Speed Range	5.0 - 5.0
Simulate Layers	Nothing

✓ Resimulate
Show Bounds
Show Only Selected

Default particle system. Figure 9

Now we want to apply our fire texture/material. You can find sprites of fire online. Import the texture into Unity and apply it to a material with the shader set to **Particles/Priority Additive** as seen in **Figure 10**.

Particles Material



Fire Particles
Shader: Particles/Priority Additive

Tint Color
Particle Texture

Tiling	X	1	Y	1
Offset	X	0	Y	0

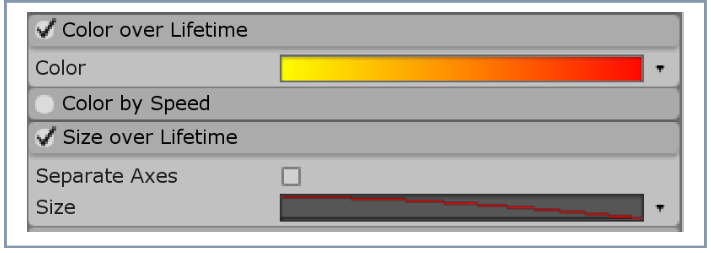
Fire particles material. Figure 10

Once the material is made, go to the last tab (called "Renderer") of the particles component and find the parameter called Material. Use the fire material you just made, and the particles will use the material!

If you use an animated sprite sheet like we did (multiple frames of fire in one image as seen in **Figure 10**), enable the "Texture Sheet Animation" tab and put in the number of tiles (X by Y) and the particles will show the animation.

Now we want to reduce fire duration and change certain properties over its lifetime. Right now, the fire is going too high up (? explain). So, in the very first tab, we will change the "Start Lifetime" to around 1. Here you could change the speed to change how intense the fire effect is. Next, we want it to start from a hot yellow to a cooler red coloration as it goes upward. To do so, we use the "Color over Lifetime" tab to create the yellow to red color gradient as seen in **Figure 11**.

Lifetime

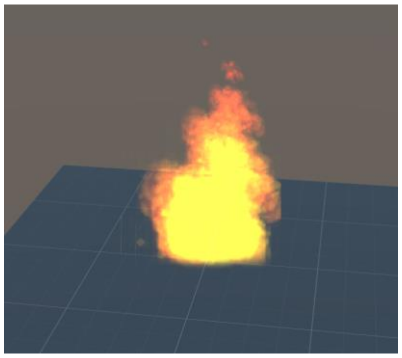


✓ Color over Lifetime
Color
Color by Speed
✓ Size over Lifetime
Separate Axes ☐
Size

Changing properties over lifetime. Figure 11

Then in "Size over Lifetime" we want the fire to taper out, so using the curve, we start at the maximum size on the left side and taper down to zero on the right side.

Fire Particle



Fire! Figure 12

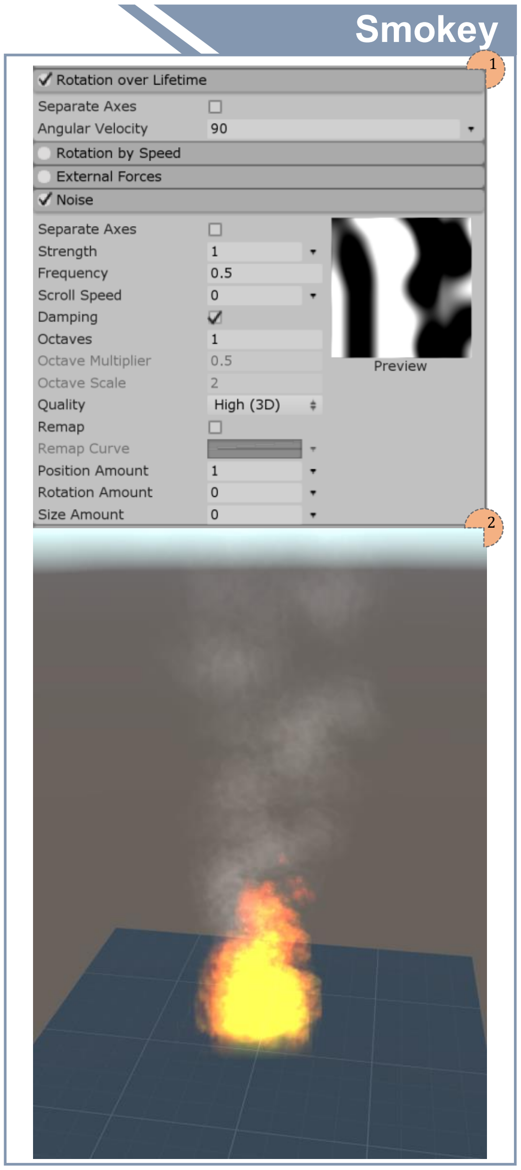
We now have a very convincing fire particle effect! Your fire should look something like **Figure 12**. Now, we can add a second particle effect for the smoke. Create another particle system and place it directly on top of the fire. You can parent the smoke to the fire, so they stay

Do the same thing for the smoke except this time, the smoke should bellow out higher and wider so the Lifetime and Angle should be increased. You can set the Color over Lifetime from opaque to transparent so that it fades out above the fire.

Then, you can use two settings that we have not used before. Changing "Rotation over Lifetime" makes the particles rotate as they travel up, adding to the billowing effect. Pair that with "Noise" randomizes it's movement which helps with the undulating cloud effect as shown in **[Figure 13, #1]**.

Once everything is finished, you should have a nice flame with smoke rising out of it **[see Figure 13, #2]**. You can even add a point light and allow the fire to illuminate the surrounding area. However, a much better way to do this is to enable the "Light" tab and insert a point light. What this does is allow the particles to produce light using the light source as a template. You can change the ratio of particles that will have lighting and change the maximum amount of lights produced by the particles along with intensity of the light. You can set the original light GameObject to be non-active in the scene and the particles will still be able to use it.

This tutorial should give you a basic understanding of how to use particle effects. Other notable settings you can use include Gravity Modifier, External Forces (for it to be affected by wind zones), Collision, and Trails. Explore these on your own and refer to the Unity Manual.



✓ Rotation over Lifetime	
Separate Axes	☐
Angular Velocity	90
Rotation by Speed	
External Forces	
✓ Noise	
Separate Axes	☐
Strength	1
Frequency	0.5
Scroll Speed	0
Damping	✓
Octaves	1
Octave Multiplier	0.5
Octave Scale	2
Quality	High (3D)
Remap	☐
Remap Curve	
Position Amount	1
Rotation Amount	0
Size Amount	0

Added smoke on top of fire. Figure 13

Scripting with Particles ❷

Scripting particle systems allows you to dynamically play or change particle systems. For example, you can script an event so that when a player clicks on an unlit campfire, the fire ignites (fire particles start emerging). You can also change the many different properties and settings of the particle system by scripting them as well.

Let's make a script where we click on campfire logs to activate the fire particles. First, in the parent particle system, turn "Play on Awake" off. Note that this will also affect all child particle systems. You will also want to turn "Prewarm" off. Prewarm, when played, puts particles in a state where they had been simulated one cycle. When played, it will start in that state instead of starting the simulation from the beginning. We want the fire to start as if it had not ignited before, so to achieve the desired effect, we need to turn it off.

Now, create your logs and have one log GameObject have a collider large enough to cover all the other logs. We recommend using a box collider for this. As seen previously in Lab 3, the OnMouseDown() and related functions work when you click on the collider not the object itself.

As seen in **Snippet 1**, we can expose the **ParticleSystem** variable to the editor. Then, you can directly assign the desired particle system to the script component of whatever object it is attached to (in our case a wood log). Alternatively, although not recommended, you can search for the GameObject you want by name or by tag and get the ParticleSystem component that way.

Then, in the **OnMouseUp()** function, in an if/else statement where we check the `ParticleSystem.isStopped` property, we use both the `ParticlesSystem.Play()` and `ParticleSystem.Stop()` methods to turn on and off our fire, respectively. Below in Table 1 are some more common properties and methods that you can use.

```
public class ParticleToggle : MonoBehaviour
{
    [SerializeField] // Assign the particle system in editor.
    private ParticleSystem mParticles;

    /* Alternatively, you can search for it by name
       or if using a different method, by tag.

        void Awake()
        {
            mParticles = GameObject.Find("Fire").GetComponent<ParticleSystem>();
        }

    */

    void OnMouseUp()
    {
        if (mParticles.isStopped) {
            mParticles.Play();
        } else {
            mParticles.Stop();
        }
    }

}
```

SNIPPET 1 *ParticleToggle.cs*

ParticleSystem

ParticleSystem.isPlaying	Property that determines if the particle system is playing.
ParticleSystem.Pause	Method that pauses the particle simulation.
ParticleSytem.Clear	Method that removes all current particles in the simulation.

A few properties and methods in the ParticleSystem class. Table 1

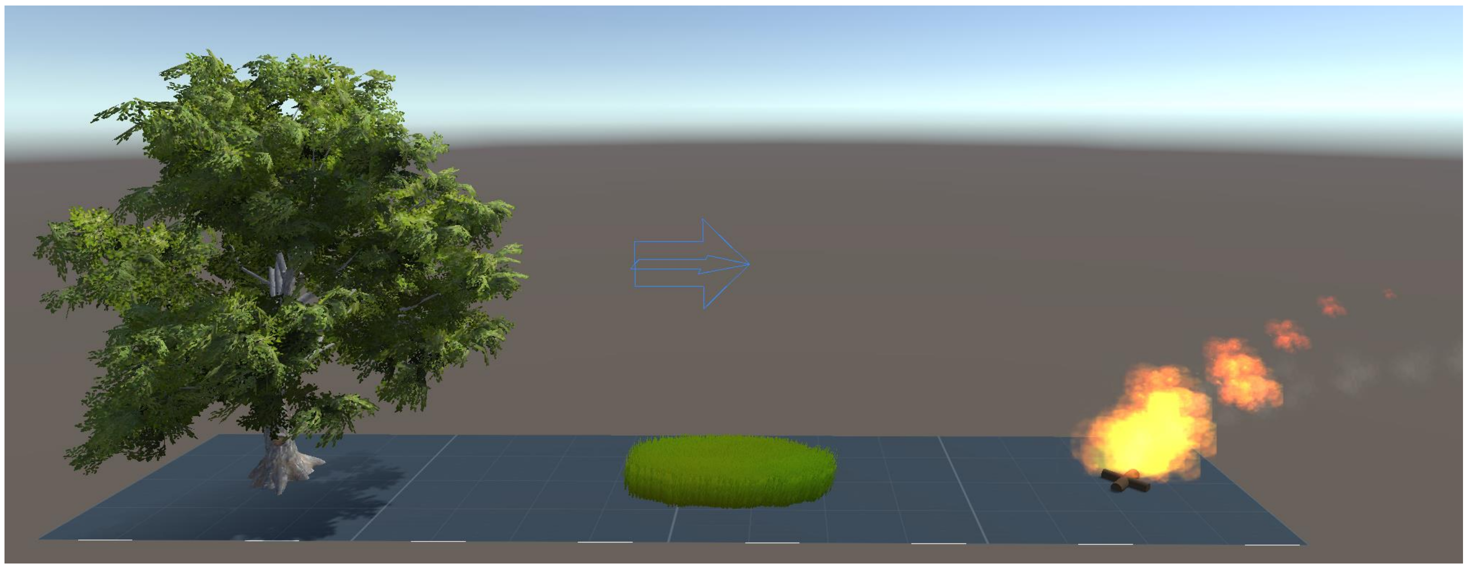
Of course, there are many more properties for the particle system that you could change, so refer to the documentation and experiment with using them in a different script.

Wind Zone

Everything discussed previously in this Lab can be affected by wind zones. A *wind zone* is a simple component that simulates wind. The trees and grass will sway in the wind, while the particle system will swirl.

To set a wind zone up, create it under 3D Objects. The wind zone is shown as a fan icon with an arrow. The direction of the arrow determines the direction of the wind. The component itself only has a few settings which include Mode, Main, and Turbulence. By default, it is set to directional, which means the wind is applied globally and in one direction. In spherical, the wind will only affect what is inside its radius and has falloff closer to the sphere edge. The Main setting is the primary strength of the wind while Turbulence sets the randomized noise.

Wind Zone



The effects of a wind zone as shown on trees, grass, and particle effects. Figure 14

It is very hard to see the effects of a wind zone in a static image, but you can see how the fire particles are pushed to the right of the screen due to the wind zone seen in **Figure 14**. You can experiment with different Main and Turbulence settings to see how these objects react. Try scripting the wind zone to rotate over time to create winds that dynamically shift around periodically.

Collision & Trigger Scripting

One great way to add dynamic interaction in your scenes is through detecting collisions and triggers. A collision is when two colliders touch while a trigger is when a collider passes through a collision zone.

As shown in Lab 3, there are a few callback functions you should know when attempting to script collision or trigger events. Refer to **Table 2** for a refresher on these functions.

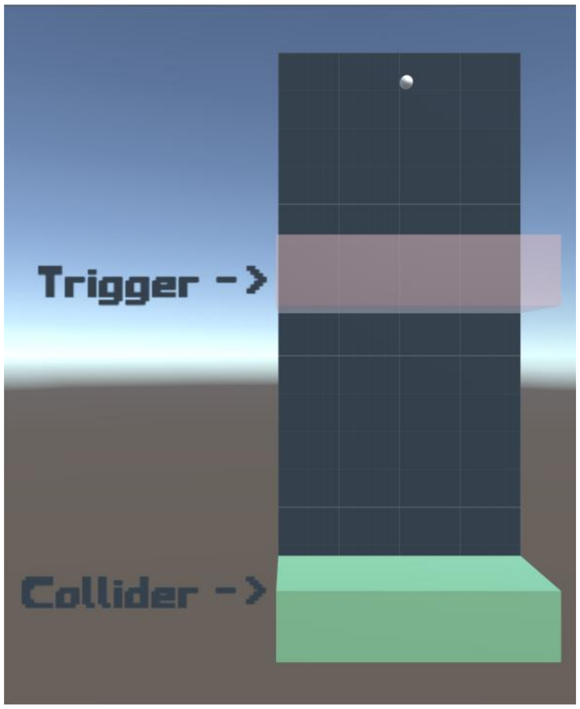
Functions

OnCollisionEnter	Called when this collider/rigidbody has begun touching another rigidbody/collider.
OnCollisionStay	Called once per frame for every collider/rigidbody that is touching rigidbody/collider.
OnCollisionExit	Called when this collider/rigidbody has stopped touching another rigidbody/collider.
OnTriggerEnter	Called when the GameObject collides with another GameObject
OnTriggerStay	Called once per physics update for every Collider '*other*' that is touching the trigger.
OnTriggerExit	Called when the Collider '*other*' has stopped touching the trigger.

Table 2

For collision to work, both GameObjects must have a Collider component attached to them. For trigger to work, one object's collider must have "Is Trigger" set to true. This will allow objects to go through the Collider but will be able to trigger callback functions set for OnTrigger. In **Figure 15**, we have created a simple setup where we will drop a Rigidbody sphere down where it will activate both OnTrigger ❶ and OnCollision ❷ events.

Test Setup



Test setup for OnTrigger and OnCollision. Figure 15

What will happen as the ball falls is a fire particle system will be ignited once the sphere enters the trigger box and an explosion particle system will occur when the sphere collides with the collider box. These particle systems are attached as children of the sphere GameObject where we can find them in our script which will start the particles.

Furthermore, the fire particle system will be looping while the explosion particle system will not loop. We want the fire particles to disappear when the sphere explodes. Thus we need to stop the fire when the sphere triggers the OnCollision function.

We also want the script to detect that the correct GameObject is activating the OnTrigger or OnCollision. To do so, we can simply use an *If* statement (as seen in **Snippet 2**).

```
public class OnTrigger : MonoBehaviour
{
    [SerializeField]
    private GameObject Sphere;

    private Collider mCollider;
    private ParticleSystem Fire;

    void Awake()
    {
        mCollider = Sphere.GetComponent<Collider>();

        // Get fire particle system.
        Fire = Sphere.transform.GetChild(0).GetComponent<ParticleSystem>();
    }

    void OnTriggerEnter(Collider other)
    {
        if (other == mCollider) {
            Fire.Play();
        }
    }
}
```

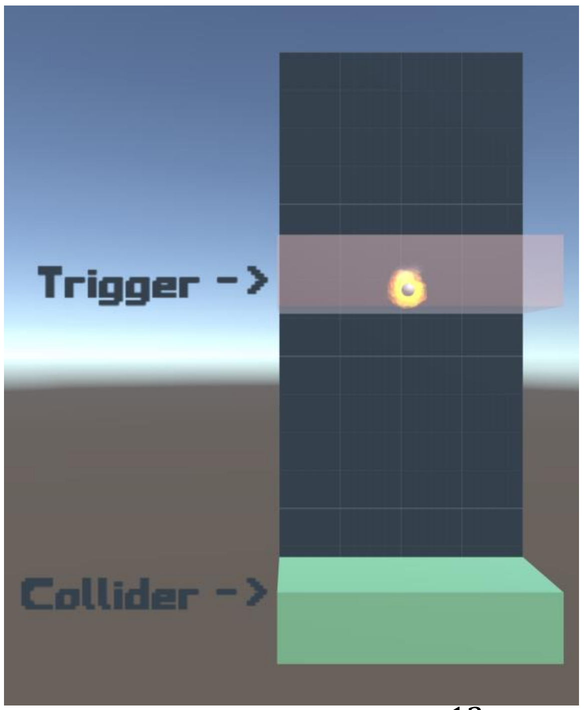
SNIPPET 2 *OnTrigger.cs*

In **Snippet 2**, we obtain the fire particle system by using `Transform.GetChild` (which in this case is located at index zero) and `GameObject.GetComponent`.

Then, in the **OnTriggerEnter()** function, we check if the Collider that entered the trigger box is the same as the one we defined. Because this script is attached to the trigger box, we need to know if the correct GameObject is triggering it. If a different sphere were to enter the trigger, nothing should happen.

Once the sphere passes the *If* statement, the fire particle system will play.

Trigger

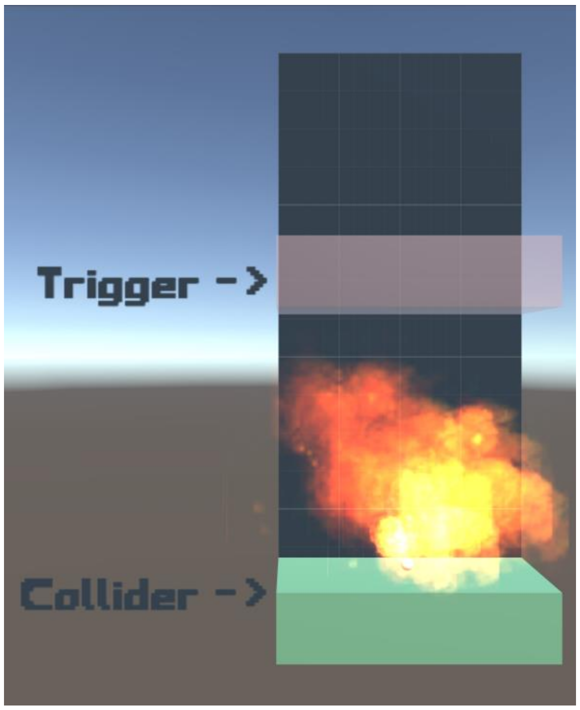


Fire particle system starting. Figure 16

```
public class OnCollision : MonoBehaviour
{
    [SerializeField]
    private GameObject Sphere;

    private ParticleSystem Fire;
    private ParticleSystem Explosion;

    void Awake()
    {
        // Get fire & explosion particle systems
        Fire = Sphere.transform.GetChild(0).GetComponent<ParticleSystem>();
        Explosion = Sphere.transform.GetChild(1).GetComponent<ParticleSystem>();
    }

    void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject == Sphere) {
            Explosion.Play();
            Fire.Stop();
        }
    }
}
```

SNIPPET 3 *OnCollision.cs*

Collision



Explosion on contact. Figure 17

In **Snippet 3**, we again obtain the fire particle system by using `Transform.GetChild` (located at index zero) and `GameObject.GetComponent`. We also want to obtain the explosion particle system (which is located at index one).

Then, in the **OnCollisionEnter()** function, we check if the GameObject that triggered the collision is the same as the one we defined. We attached this to the collider box for consistency with the previous script, but we can attach this collision script to the sphere itself and have it explode on contact with anything!

Additionally, you can remove the GameObject from the scene by using the `Object.Destroy` ❷ method. This simulates the sphere being destroyed during the explosion.

Cloning GameObjects ❓

The cloning of GameObjects is very useful capability when you want to programmatically generate items or projectiles for the player. You can clone rigidbodies and add force to them, allowing you to shoot items at targets, for example.

Creating a clone is quite simple. Using `Object.Instantiate` you can clone any template object.. In **Snippet 4**, we use instantiate on a Rigidbody and apply a force in the direction the camera is facing to shoot the Rigidbody outwards.

```
    . . .

     if(Input.GetButtonDown("Fire2") && Time.time > timeBeforeNextFire){
         GameObject clone = Instantiate(projectile, projectile.transform.position,
                                        myCamera.transform.rotation);
         clone.SetActive(true);
         Rigidbody cloneRB = clone.GetComponent<Rigidbody>();
         cloneRB.AddForce(myCamera.transform.forward * shootForce);
         timeBeforeNextFire = Time.time + fireRate;
     }
```

SNIPPET 4 *FireProjectile.cs*

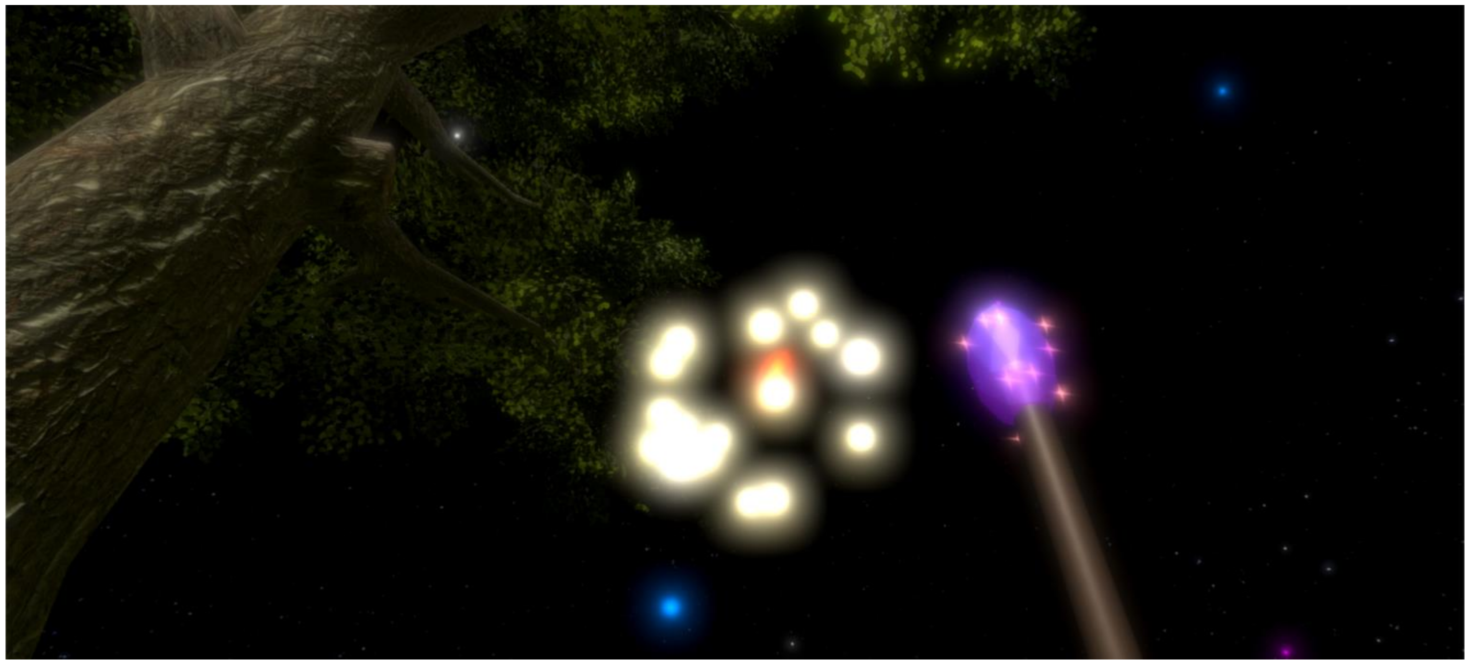
In this snippet the **Update()** function detects a mouse input. In this case the mouse input is *"Fire2"* which maps to the right mouse button. You can find more about `Input.GetButtonDown` ❓ in the Unity Manual.

We have some more logic script here to prevent the player from spamming many projectiles at once. We can have a "cooldown" timer where we can set the fire rate.

The most important piece of this snippet is `Instantiate(Object original, Vector3 position, Quaternion rotation)`. The original is a rigidbody GameObject set to non-active. This will act as the source of the clone GameObjects we will spawn in. For position, we are using the source object's position for the clone's initial position. Alternatively, the source object can be somewhere different, and you can use a predefined position instead. For the rotation, we will be using the camera's rotation in order to shoot the projectile in the direction the camera is facing.

Next, we store the instance of the clone in a variable so we can set it active and add a force to its Rigidbody component in the direction the camera is facing using `Rigidbody.AddForce()`. And just like that, we are launching projectiles!

Wizard in the



Wand casting magic explosion next to a tree. Figure 18

Post Lab Homework Assignment – Option A

[BASIC]

With the gained knowledge of creating terrains, using particle effects, and scripting, your assignment is to create a forest scene and a first-person controlled character that can shoot magic out of a wand. The wand itself should have a passive sparkling effect when not in use, and a brighter shining effect when used. Furthermore, the wand should be able to shoot projectiles that are able to explode on impact and able to be detonated early with a click of a button. You can have the user able to scroll through a selection of spells or have left click be one spell, right click be another spell, and middle click to detonate early (can integrate keyboard controls if preferred). The user should not be able to spam the spells too quickly though, so implement a way to control the rate at which spells can be fired.

Your project must have all the following deliverables:

- Use a first-person controlled character with a wand visible on screen
- Passive and active sparkling effects for the wand.
- At least 2 unique spells (e.g. Fireball & Ice Lance).
- Spells explode on contact or by the player.
- Create a basic landscape and forest for the player to walk around in.
- Ambient stereoscopic sounds.

What to turn in:

- Unity Project File
- Lab Report (Explanation & Images)
- Video (as an embedded link in the report)

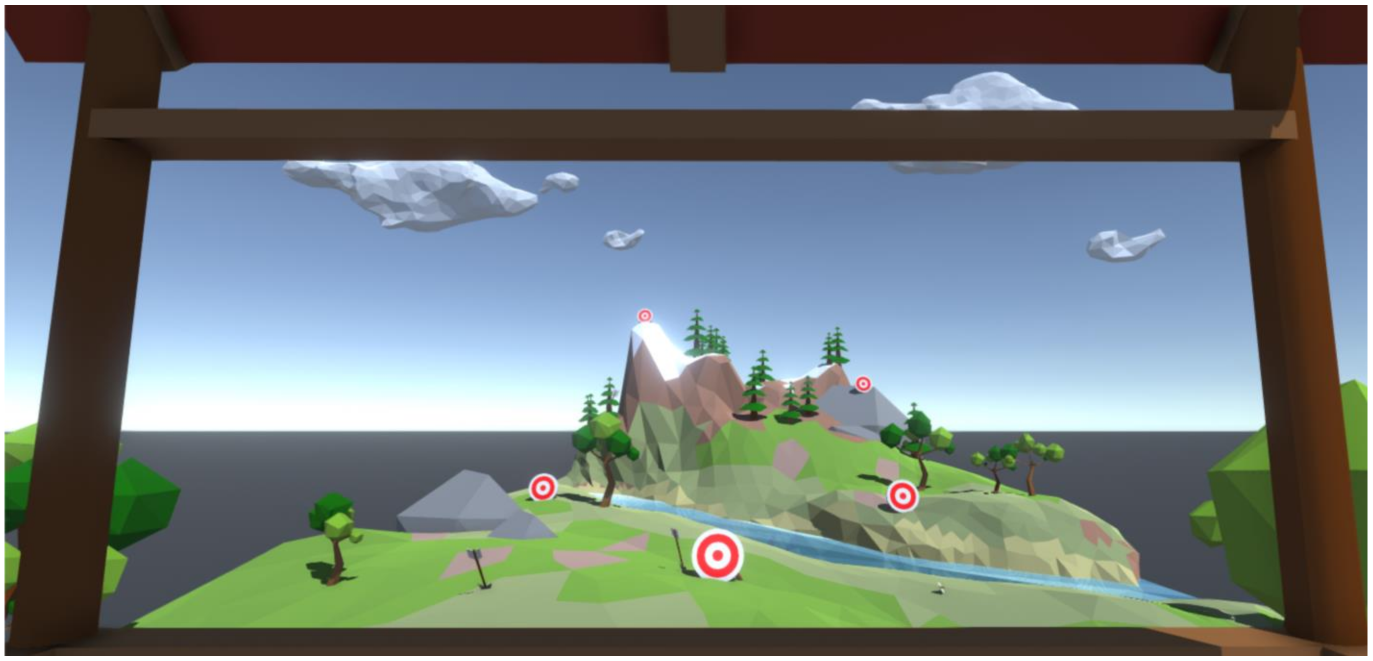
[ADVANCED]

In addition to the basic assignment, the advanced assignment will have you create custom trees and a higher requirement of spell options. The spells must also have trails and sound effects.

Additional deliverables:

- Create 5 unique custom trees (i.e. varying shapes, sizes, and textures).
- An additional 3 unique spells.
- Spells must have trails.
- Stereoscopic sound effects on cast and on impact.
- Create dynamic wind zones that shifts around randomly.
  - One spell idea is to temporarily summon strong gusts of wind in the direction the player is looking.

Archery Range



Archery range with targets.

Figure 19

Post Lab Homework Assignment – Option B

[BASIC] - Same as Option A

[ADVANCED]

You are not required to do the basic assignment for Option B. Instead, this assignment requires you to create a sort of archery range. You should be able to shoot projectiles and earn points the further away the target is. Look into using "TextMesh Pro" ❓ in order to display numbers and text in 3D space or on screen. You should be able to script it to change its text property to display a live score.

Your project must have all the following deliverables:

- A simple crosshair on screen.
- Implement your own scoring system
    - Show live score on screen after each target hit.
    - Player can lose points if they miss.
    - Bonus points if there is a streak without missing.
- Show a trail behind the projectile being shot.
- The player should win after a certain amount of points, which should trigger some sort of particle system fireworks.
- Implement a wind system that can push the projectile from its original path.

What to turn in:

- Unity Project File
- Lab Report (Explanation & Images) Include link to Video in the report

References & Additional Resources

1. Terrain
   a. https://docs.unity3d.com/Manual/script-Terrain.html
   b. https://docs.unity3d.com/Manual/terrain-OtherSettings.html
   c. https://docs.unity3d.com/Manual/terrain-Tools.html
   d. https://docs.unity3d.com/Manual/terrain-Grass.html

2. Trees
   a. https://docs.unity3d.com/Manual/terrain-Trees.html
   b. https://docs.unity3d.com/Manual/class-Tree.html

3. Particles & Effects
   a. https://docs.unity3d.com/Manual/PartSysUsage.html
   b. https://docs.unity3d.com/ScriptReference/ParticleSystem.html

4. Wind Zone
   a. https://docs.unity3d.com/Manual/class-WindZone.html

5. Collisions & Triggers
   a. https://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html
   b. https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html
   c. https://docs.unity3d.com/ScriptReference/Object.Destroy.html

6. Cloning GameObjects
   a. https://docs.unity3d.com/ScriptReference/Object.Instantiate.html
   b. https://docs.unity3d.com/ScriptReference/Object.Instantiate.html