Alex Lin
Project 4: In-Editor Spawner

Project Details

The design team likes the run-time prop spawner from Project 3 and has requested an edit mode (not play mode) spawner tool which will allow designers to place multiple spawners in a scene. They need editable control over item count, spawn area, and randomness for different props. The spawner settings need to be editable and automatically place objects on collision surfaces.

Create an edit mode prop spawner which:

- *Adds a spawner object to the scene using a custom menu item*
- *The spawner object must provide controls which allow a designer to:*
  - *Attach a prefab*
  - *Select a quantity*
  - *Control spawning area in the scene*
  - *Control randomness and scatter*
  - *Automatically place spawned objects on collision surfaces*
  - *Edit all settings*
  - *Manually delete or move individual objects*
  - *Nice inspector layout and controls which prevent unreasonable settings*
  - *Spawn objects must not intersect*

The following assets were used in this project:

- *BitGem 3d Pixel Dungeon kit*
- *ShaderForge*
- *Lava 2 Normal Map from Textures.com*

Video: https://youtu.be/aYw4vHqlJ2o

Analysis

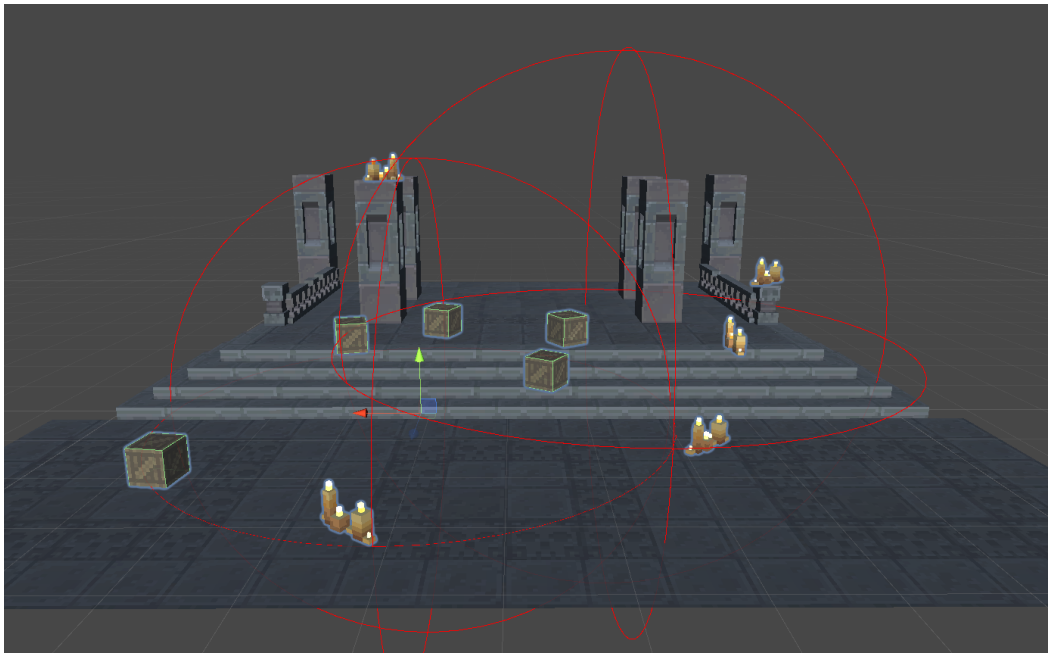
Due to the required features, some questions that came up included:

- How will the system handle spawning of the objects on surfaces?
- How should collision detection function?
- Which user settings best fit the given needs?

Approach

After doing research into collision checking and brainstorming, I decided to use the following methods:

- Spawn the objects using loops in the OnValidate() function and a list
- Check for collisions in a deeper loop using boxCasting and Unity's layer system and spawn the objects on the collision point
- Update positions on objects by raycasting in the OnDrawGizmosSelected() and moving the objects to the hit point



An example of two separate spawner objects with different prefabs.

Spawn()

The spawner system is a function, Spawn () that is run while values are changed in the editor using the OnValidate() function. The idea is that the item(s) instantiated are using the prefab that the user inputs in the editor. The instances are held in a list for ease of access and maneuverability. The function first checks if the Randomize Rotation option is on in the editor and adjusts the rotation of the instantiated objects further into the function. A while loop contains the rest of the function, the first real parts of the function are generating a Vector 3 which randomizes the x and z, and having the y above based on the size of the spawn radius, and some RaycastHit objects that hold the information of the hit point to use that position to generate the instantiated object.

```
if (randomizeRotation)
{
    while (i < count)
    {
        Vector3 pos = new Vector3 (Random.Range(transform.position.x + -spawnRadius, transform.position.x +
                spawnRadius), transform.position.y + spawnRadius, Random.Range(transform.position.z + -
                spawnRadius, transform.position.z + spawnRadius));
        RaycastHit hit;
        RaycastHit boxHit;

        bool groundHit = (Physics.BoxCast(pos, prefab.GetComponent<MeshRenderer>().bounds.extents*2,
            Vector3.down, out hit, Quaternion.identity,
            50, groundLayer, QueryTriggerInteraction.UseGlobal));
        bool objectHit = (Physics.BoxCast(pos, prefab.GetComponent<MeshRenderer>().bounds.extents*2,
            Vector3.down, out boxHit, Quaternion.identity,
            50, objectLayer, QueryTriggerInteraction.UseGlobal));

        if (groundHit && !objectHit)
        {
            instances.Add(Instantiate(prefab, hit.point, Quaternion.Euler(0, Random.Range(0, 360), 0),
                transform));
            instances[i].transform.SetParent(transform);
            i++;
        }
    }
}
```

The spawning system when the Randomize Rotation option is checked, mainly using boxCasts and random x, z cords for placement.

After that, the 2 actual boxCasts are generated from that randomized Vector3 into bools that detect collisions on the Ground Layer for instantiating and one on the Object Layer for detecting collisions. An if statement that checks if the Ground boxCast is touching an object on the Ground Layer and the Object boxCast is not touching an object on the on the Object Layer and instantiating (with randomized rotation or not depending on the user input in the editor) and adding the prefab into the list that holds the objects and also parents the instantiated object to the Spawner GameObject.

OnValidate()

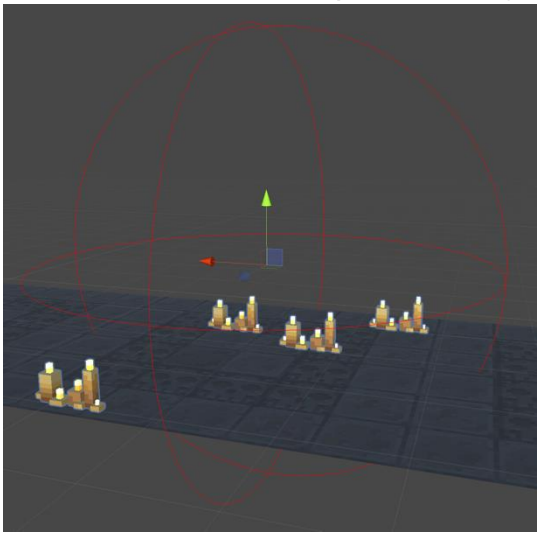
```
void OnValidate()
{
    // Trick for making Destroy work
    EditorApplication.delayCall += () =>
    {
        foreach (GameObject g in instances) { DestroyImmediate(g); }
        instances = new List<GameObject>();

        Spawn();
    };

    Random.InitState(seed);
}
```

The OnValidate() function, which calls the spawn function, destroys duplicate instances, and contains the seed generation.

This function is run with every value change in the editor, which was very valuable in this project, but limiting the refreshing of the function with EditorApplication.delayCall (which makes sure the inspectors are fully done updating) was necessary. Within this, a foreach loop with the instances list was used to destroy the duplicates that could be generated and then creates a fresh list for the instancing every time. Finally, the Spawn() function is called using the fresh instances list. The last part is just using the in-editor modifiable integer “seed” with Random.InitState to basically hold the rng of the instancing.

OnDrawGizmosSelected()

This function refreshes every frame that a gizmo is selected in the editor, perfect for modifying objects live as a user moves the spawner object around. The first thing tackled with this function is drawing a wire sphere to indicate the spawning area within the scene, simply drawing it at the current location of the spawner object.



The wire sphere drawn to indicate spawning area

To handle the refreshing of the placement when moving the spawner GameObject, using a simple Raycast to check for another ground surface was the cleanest option. Firstly, a foreach loop is used to gather the instanced objects in the list, then a new Vector3 is created using the position of each object and a new Raycast is cast down with that new Vector3 position. The transform.position of that object is then set equal to the position of the hit.

```
void OnDrawGizmosSelected()
{
    Gizmos.color = Color.red;
    Gizmos.DrawWireSphere(transform.position, spawnRadius);

    foreach (GameObject g in instances)
    {
        Vector3 newPos = new Vector3(g.transform.position.x, g.transform.position.y + spawnRadius,
            g.transform.position.z);
        RaycastHit newHit;

        if(Physics.Raycast(newPos, Vector3.down, out newHit, spawnRadius, groundLayer))
        {
            g.transform.position = newHit.point;
        }
}
```

The OnValidate() function, which calls the spawn function, destroys duplicate instances, and contains the seed generation.

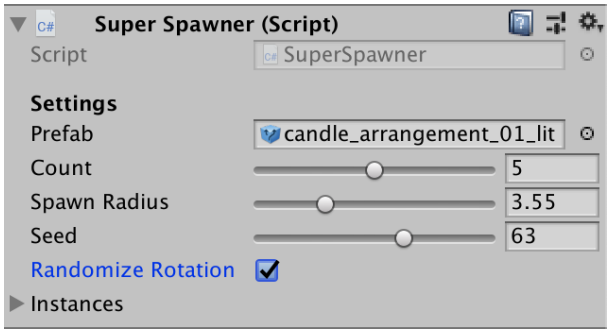
UI

The UI for interacting with the spawner options are very simple with tooltips designating the uses of each setting. Each setting has a set range to prevent crazy numbers and breaking of the system.

Recording was handled with OBS.

```
[Header("Settings")]

[Tooltip("Insert prefab of the object you wish to spawn here")]
public GameObject prefab;

[Range(0, 10)]
[Tooltip("Number of objects to spawn")]
public int count;

[Range(1f, 10f)]
[Tooltip("Spawn distance from this object in scene units")]
public float spawnRadius;

[Range(0, 100)]
[Tooltip("Change the seed")]
public int seed;

[Tooltip("Randomizes rotation of spawned objects")]
public bool randomizeRotation;
```

The user-facing settings with ranges to prevent unreasonable use.



The user-facing settings with ranges to prevent unreasonable use.