Yuanqin Fan, Yan Kai, Yian Li, Junlin Wang

114 - Final Project Report

Project Introduction

For this project, what we did is to create a simple game by using Unity 3D, a game engine, with several feature of computer graphics. Since this class is a graphics class, we decided to make the game scene as good as possible to highlight the graphic's importance in this project rather than paying attention on the gameplay. You will be free to wonder and explore the landscape we have created, and even interact with components in the game.

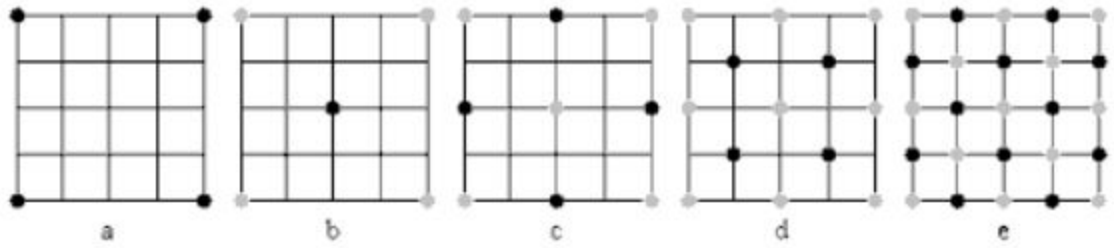
And these are the algorithms we implemented in this project:
1. Fractal Terrain
2. Fractal Tree
3. QuadTree/OcTree Dynamic Loading
4. Ripple Water
5. Others
6. Gameplay

Ps. The game file will be submitted by using google drive sharing.

Fractal Terrain ---- By Yuanqin Fan

There is a very important concept for any fractal things which is called self-similarity. If the replication of an object is exactly the same or similar at every scale, it can be called as a fractal object. Back to topic, the fractal terrain generator that I implemented in this project is mainly relies on an algorithm called Diamond Square. Suppose we have a two-dimensional array with many points in it. For simplicity, we take a 5X5 array as an example and set every point around the corner in same value (every dimension of this array should be 2^n+1), which is shown as Figure a. And than, this algorithm will be

divided into two phases.



Phase 1 – Diamond
By using four corner points in this square, we can generate a random value in the center of this square which is also the intersection of two diagonals (the random value's range can be defined by ourselves). Then, we make the center point's value equal to the average value of four corner points plus the random value we generated before. Thus, we can generate four different diamonds as the Figure b shows.

Phase 2 -- Square
Next, we will use the four corner point of each diamond to generate a random value in the range we defined in every center points of each diamond. Just like what we did in phase 1, the center point's value also equal to the average of corner points in each diamond plus the random value. By doing so, we will get four squared as the Figure c shows.

Therefore, the number of squares generated will equal to 2^(I+2), where I is the number of iterations through the recursive subdivision routine.

Let's take a look at what we did in very first round of this algorithm. By connecting the nine points of Figure c, we can get our first terrain as Figure f shows. And if we finished second round of this algorithm, our terrain will be more complex as the Figure g. shows. Also, if we assign a lager array and do more rounds, we can get a very detailed terrain just like Figure h.

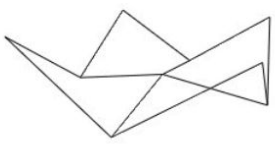


Figure f.

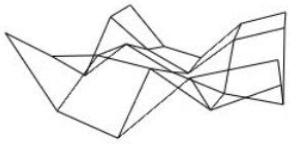


Figure g.

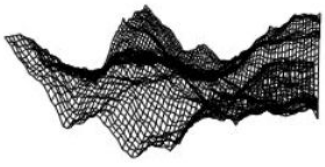


Figure h.

Script Code: terrain.cs

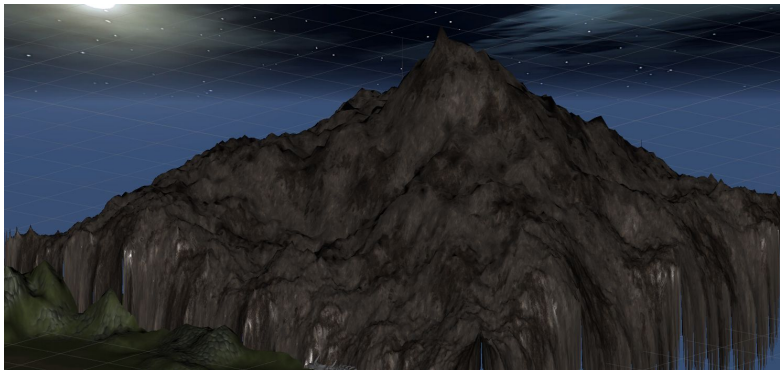


Figure i. Final Result

Fractal Trees ---- By Yan Kai

The algorithm for growing a tree is roughly like so:

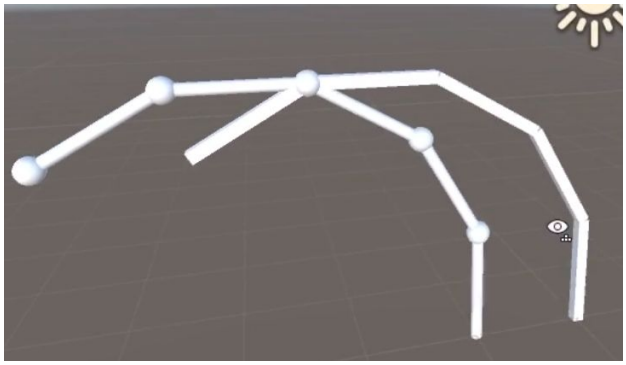
1. The tree sprouts.
2. The sprout eventually splits into branches.
3. These branches themselves split into further branches.

At each step of this process it is as if two new smaller trees emerge. These smaller trees can be conceptualized as the trunks of a new generation of trees. This repetition of branching that forms the tree is also the cause for trees approximate self-similarity. In nature this process eventually stops and the end products are no longer fractals.

Each branch are generated recursively by using Recursive Instantiator:

```
        for (int i = 0; i < splitNumber; ++i) {
                        if (recurse >= 0)
                        {
                                var copy = Instantiate(gameObject);
                                var recursive =
copy.GetComponent<RecursiveInstantiator>();
                                int[] data = {i,splitNumber, info};
                                recursive.SendMessage("Generated", data);
                        }
                }
```

The function sent a generate message to the game object and pass to the tree base object. The base object includes script that can receive the “Generated” message and use the data to calculate the scale, location, color, rotation of the next tree base object.


The tree is able to randomly generate branch in 3d coordinate by rotating the next generated branch:
this.transform.localScale *= scalar;
this.transform.rotation *= Quaternion.Euler((angle * ((depth*2) -1))-offset, Random.Range(constraint)*depth, Random.Range(constraint)*index[0]); // the overall appearance and feature of the tree will decide by the constraints function

Result:

Scripts that implement the data:
Grower.cs
Rotation.cs
Scale.cs
RecursiveInstantiator.cs

QuadTree/OcTree Dynamic Loading ---- By Junlin Wang

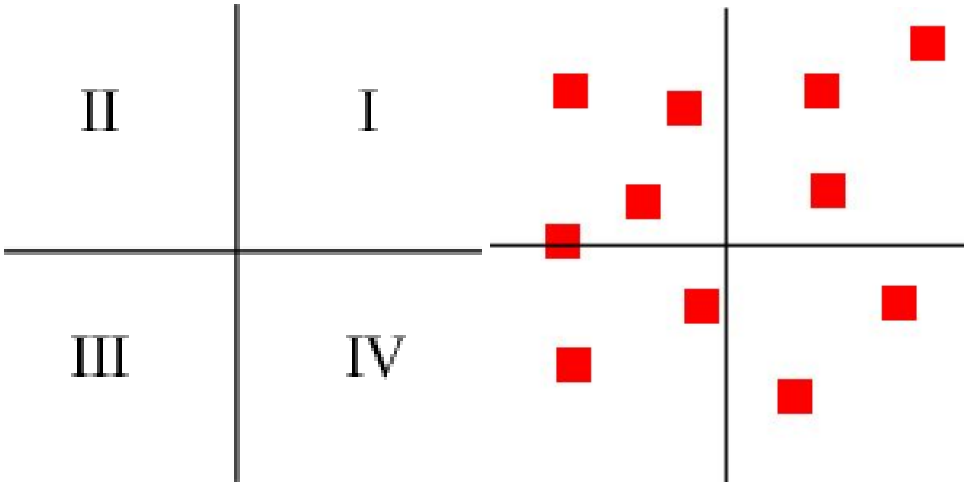
We implemented QuadTree to dynamically load objects into the scene. Unity only provide basic object loading and resource management. This idea is inspired by the need that sometimes we don't want to load every objects into the scene, but rather want to load objects that are visible by the player. And everytime the visible bounding box changes, the objects in the new bounding box will be loaded, and outside will be deleted. And objects no longer in the bounding box will be deleted from the scene. Therefore, we need a data structure that would allow us to quickly search objects in an area of interest, such as QuadTree/OcTree/KdTree.

We picked QuadTree because we choose to segregate our objects into 2d spaces rather than 3d. So we only need to divide each space into 4 subdivisions in QuadTree rather than 8 subdivisions in OcTree. We didn't use Kd tree because the aspect ratio would be sometimes too high. This means sometimes it would generate subdivisions that are too narrow since Kd tree divides by the median of the data points. This would create unnecessary inefficiency in searches, or unexpected results.

Construct a QuadTree

1. Insert and object into the QuadTree object
2. Traverse to the QuadTree node that contains this object. We can do this since each node will store its bound, and each object will have a (x,y) coordinate.
    a. Once we find the Quadtree node, we insert the object into the node's object list.
    b. If the number of object in the current QuadTree node exceeds a certain amount, the QuadTree splits the current 2d space into 4 equivalent regions (often squares). Each region is a new node with corresponding objects within its bound.

SceneSeperateTreeNode.cs handles the logic of QuadTree.



We load all objects into QuadTree when the game starts. Once we get everything loaded. We create a bounding box around our character, and only objects within this bounding box will be displayed. Example.cs is going to init the tree and the controller.

Functions Written

- ➔ QuadTreeNode
  - ◆ Constructor
  - ◆ Insert: This function will insert the object into the object list in the tree node, if the list is full, we split the tree node.
  - ◆ Remove: Removes the object from the treenode. We don't delete it, since in the future it may be used again. Also it makes the implementation cleaner and simpler.
  - ◆ Contain: check if the tree node contain that object
  - ◆ Trigger: this is called everytime we want to check the character bounding box in the QuadTree. It will be triggered by the refresh technique described below. The character bounding box will be passed to this function, and this function will return a list of objects within character bounding box.

You can see more in SceneSeprarteTreeNode.cs

Unity Provided Class

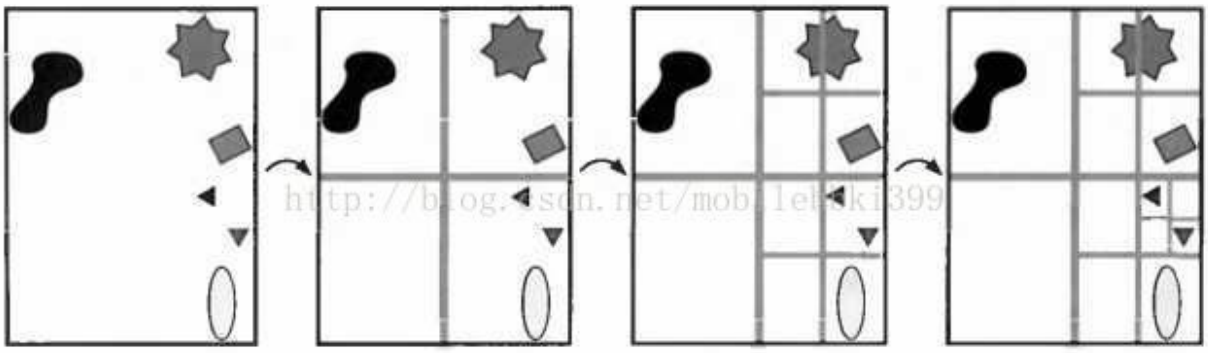
There is a really nice build-in class called Bounds in Unity that we can utilize. It makes tracking the properties and the intersections of bounding boxes simpler. We used an extended Bounds wrapper that could help me simplify the process. The code can be found in Utility/Boundsex.cs.

In order to better utilize Unity's API, we created some API files to handle some events. IDetector.cs handles the character bounding box, and when an intersection happens. ISceneObject.cs handles the bounding box for the objects. You can also call the onHide and onShow functions to hide or show the objects depending on the bounding box. SceneObject.cs provides a wrapper for the object class. It includes more information that makes it easier to handle the queue of objects.

Refresh Technique

The game will only search in the QuadTree once our character bounding box moves. We also keep track of time, so that there is a minimum interval that the game will search in the QuadTree. So sometimes even though the character bounding box has moved, if the time interval has not passed, it will not do a search. This technique is used to prevent excessive search to save computation power. There is also an interval to delete out of bounds objects. It will only deletes objects from the scene once the interval is reached. This part of logic is handled in SceneObjectLoadOcntroller.cs.

Edge Case

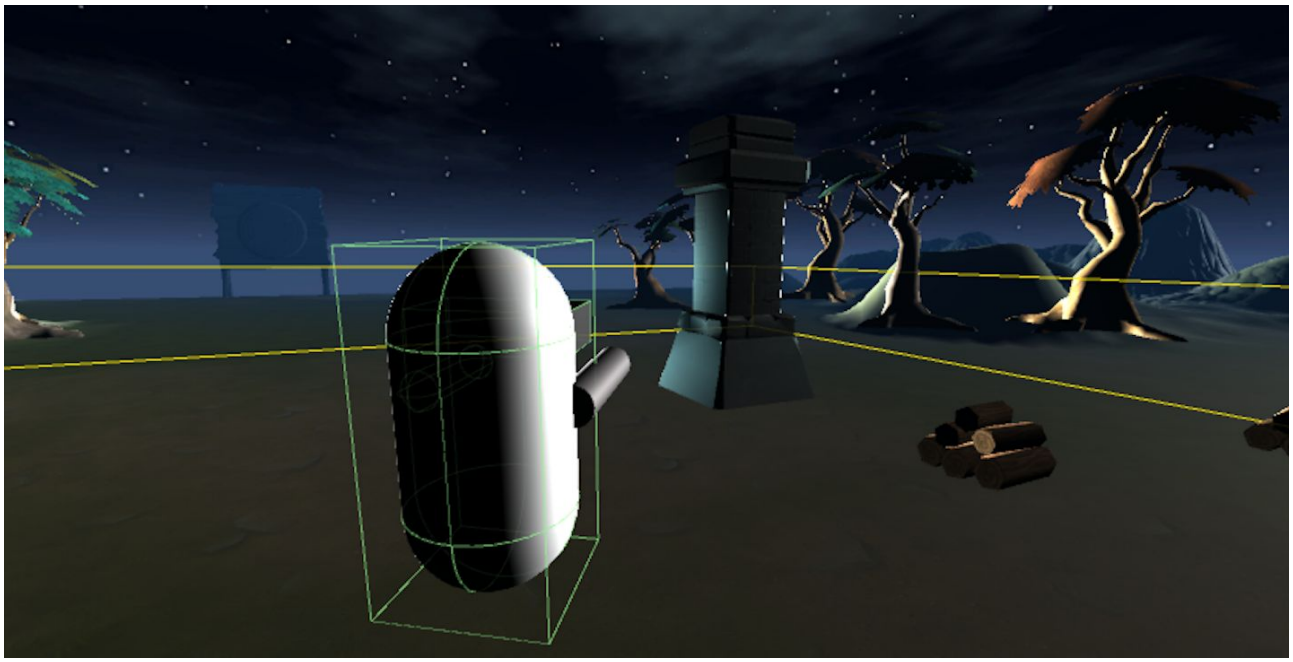


So sometimes an object can be in two/more sub regions. So basically, this means an object overlap one/multiple edges. There are two ways to solve this problem:

1. We store this object in both or all QuadTree nodes.
2. We store it in one node and only in the parent node.

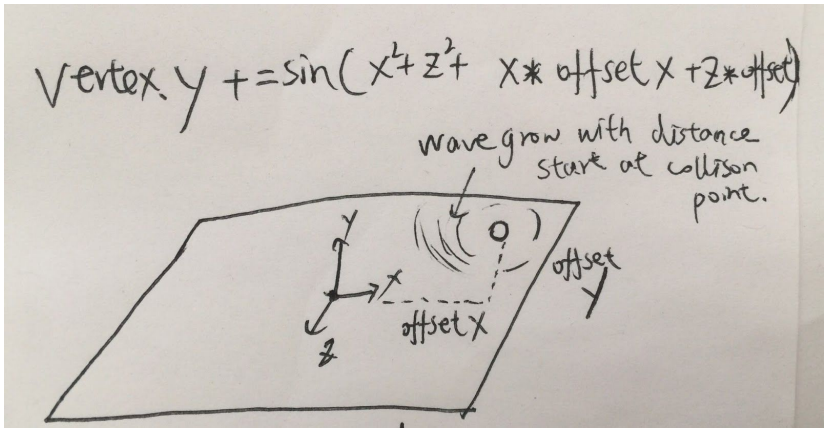
We use the second solution, because the first solution is hard to manage, since all the nodes and their children will contain this object. If we use the second solution, the implementation will be cleaner. Object overlapping many regions will only be stored in the parent will make it easy to manage.

Results



Ripple Water ---- By Yian Li

----------Ripple Shader----------



create 8 wave amplitude in array
and 8 distance offset
{
value = scale* Sin(Time*Speed* Frequency

+X^2 +Y^2+
(vertex.X +Offset_X) (offset calculated in the C# below)
+ (vertex.Z +Offset_Z)
}
The Offset_X and Y above will trace to the collision point and make the water ripple begin at the collision point with the ball
(other wise the ripple will begin at the center of the plane)

v.vertex.y += value *amplitude
v.normal.y += value *amplitude

repeat above 8 times with the array structure to support multiple ball drop at the same time

----------C# script attached to the Ball----------
Void update() set shader's each waveamplitude *= 0.98
To lower it until reach 0

OnCollisionEnter(){
Calculate the offset distance from Water Plane Object's X and Z axis to the collision point with the Ball object
Thus, we got distance X and Distance Z

Pass the (distance _X and Z/ size of the object) to the Shader's "Offset_X" and "Offset_Z" value
PASS waveAmplitude=(collision_velocity.magnitude/divider) to the shader's "Amplitude"

}

----------Problem----------
so far…… If 2 of more ball hit the water together the Ripple effect of the previous dropped ball will be interrupt and stopped
We do not want the ripple to be stopped.

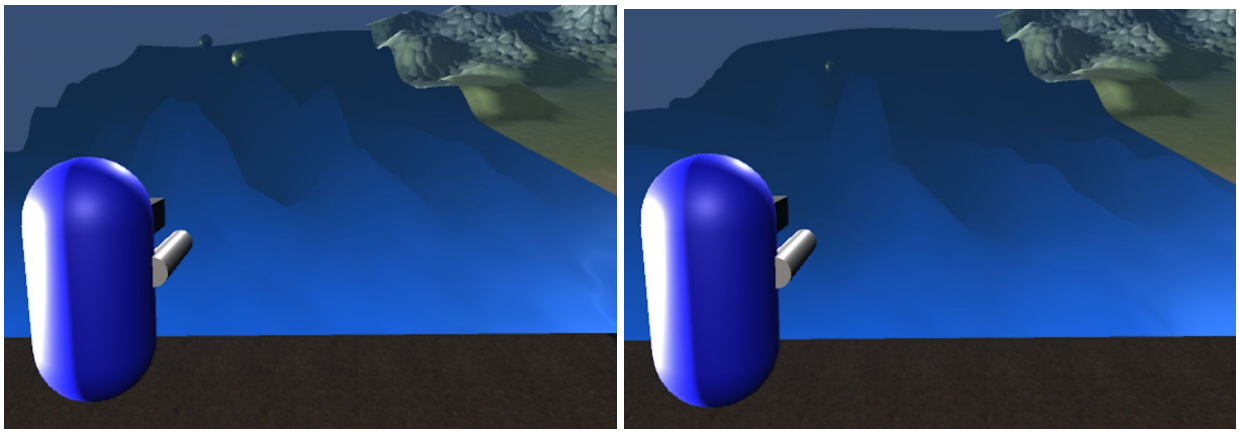
----------Fix----------
Next…… Calculate distance of point ofimpact so the Wave will grow over and doesn't impact the whole plane

Ball's C# script --- collision script ---------
Add a vector 2[]impact begin point----- X and Z -----make the collision point as the center point and the wave will be spread out

Add a float Distance----distance of impactwill grow bigger so will effect more vertex points
At the update() function
If amplitude> 0 distance += wave Speed

----------Shader----------
because collision point is in world space so :
float3 worldPosition=mul(object2world, v.vertex).xyz ------for water plane vertices
if sqrt((worldPos.x – X_impact)^2 + (WorldPos.Y –Y_impact)^2)) < Distance
{
v.vertex.y += value *amplitude
v.normal.y += value * amplitude
}
Now Wave will spread like a Round wave slowly
Because the distance of impact will grow bigger every frame
Every time distance grow bigger we apply:
v.vertex.y += value *amplitude
v.normal.y += value * amplitude
we need to do this 8 times to support up to 8 ball bounce on the water
Finally add a diagonal wave to the water constantly value=sin(x+z)

Result:



Other Stuffs:

Deform Snow
Bind rendering texture to the camera
What the camera view will be a real time texture.
It is changed within the camera view. Such as Minimap can record player's changing position change

collision area on the snow surface will be painted read in the rendering texture
In vertex shader , read the read pixel on the image, and vertex.y minus its red pixel.

Detailed snow deform requires Tessellation: More triangle Mesh - interpolate vertex
Displace the vertices toward the normal direction according to depth

3D Modeling
In addition to the terrain, tree and water we did, I used some online 3D models from "www.turbosquid.com" to improve our game scene.

Gameplay

The gameplay for this game is very simple. Player can control the character by using **"W","A","S","D" and Mouse (just like FPS game)**. By walking through, players can see the environment model are loading around themselves which is the "Dynamic Loading". And the Fractal Tree is biggest tree in the center of this map, players take a look at it when they passed by. Also player can **press "T" key** to trigger the fractal terrain generator and the terrain's shape be changed. When the player are closed to the Water, player can **press "C" key** to shoot the ball and see the effect of Ripple water.