Curiosity Cube

In this hands-on session we will recreate the Curiosity cube with a simple generative algorithm and some 3D mathematics. The cubes are generated by an algorithm given the layer number where layer 1 is the cube in the middle, layer 2 is the layer of cubes encasing 1 and so on. As shown in Figure 1 there is a mathematical relationship between the layer number and the amount of cubes in that layer.

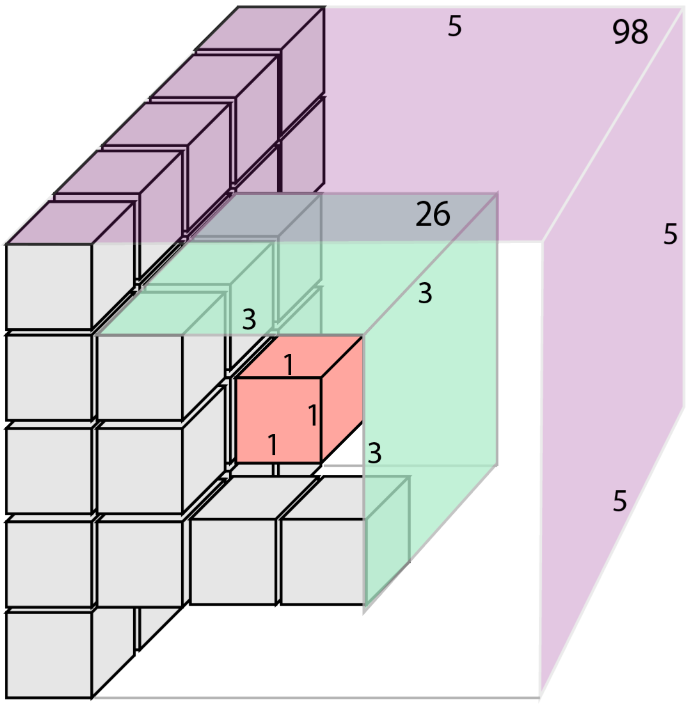

Figure 1. A cube made of layers of smaller cubes where each layer fully encases the next.

In layer 1 there is 1 cube, in layer 2 there are 27 cubes in layer 3 there are 125 cubes. The relationship between the layer number and the cubes in that layer is:

$$(\text{layer number} + 2)^3 - \text{layer number}^3$$

Therefore in layer 3 there are $(3 + 2)^3 - 3^3 = 125 - 27 = 98$ cubes. Although in layer 3 the length of a side is 5 cubes, which would make the total 125 if the entire structure were composed of smaller cubes, the smaller cubes are only surface depth and therefore the number of cubes making up the interior must be subtracted.

If each small cube is made from a standard Unity cube of length 1 the math to determine the position of each is quite elementary. The centre cube is placed at (0,0,0). With a size of 1 it extends from -0.5 to 5 along each x, y and z axis. The position of the next layer begins at a distance of 1 from the previous layer. The distance from the centre for the position of any layer is equal to the layer number. For example, layer 3 would begin at a distance of 3 from the center.

The number of cubes making up the side of a layer can be calculated by:

layers x 2 -1

For example on layer 3, the length of a side in cubes is equal to 5 and on layer 4, the length is 7. Once we know how many cubes will make up the length of the surface we need to draw and considering the position of each cube in any axis will be a multiple of 1 based on the layer, its only a matter of implementing a nested for loop that creates and places cubes according to their row, column and depth position. In addition, we need to take care not to draw any interior cubes and not to repeatedly draw cubes on the edges and corners.

Step 1. Start a new Unity Project.

Step 2. Create a new C# script called createCubeLayer and add the following:

```
using UnityEngine;
using System.Collections;


public class createCubeLayer : MonoBehaviour {


	public GameObject cubePrefab;
	public int layers = 5;		//layers from centre
	float cubeSpacing = 0.05f;	//small gap between cubes
	GameObject largeC;		//the large middle cube


	Vector3 lastMousePos;
	float movementSensitivity = 0.1f;
float pinchSensitivity = 0.01f;
float previousDistance;


Vector3 CameraZoomMax;
Vector3 CameraZoomMin;



void Start ()
{
      int maxCubes = layers * 2 - 1;


      //generate a large inner cube
      Vector3 pos2 = new Vector3(
                  (maxCubes -1)/2.0f + cubeSpacing*layers,
                  (maxCubes -1)/2.0f + cubeSpacing*layers,
                  (maxCubes -1)/2.0f + cubeSpacing*layers);
      largeC = Instantiate(cubePrefab, pos2,
                  Quaternion.identity) as GameObject;
      largeC.transform.localScale = new Vector3(
                  maxCubes-2 + cubeSpacing*layers,
                  maxCubes-2 + cubeSpacing*layers,
                  maxCubes-2 + cubeSpacing*layers);
      largeC.renderer.material.color = Color.red;
      largeC.transform.gameObject.tag = "Untagged";


      //generate outer cubes
      for(int row = 0; row < maxCubes; row++)
      {
            for(int col = 0; col < maxCubes; col++)
            {
                  for(int dep = 0; dep < maxCubes; dep++)
                  {
                        //if we are generating a cube on
                        //the outside
                        if(row == 0 || col == 0 || dep == 0 ||
                    row == maxCubes-1 || col == maxCubes-1 ||
                    dep == maxCubes-1)
                        {
                              Vector3 pos = new Vector3(
                                    row + row*cubeSpacing,
                                    col + col*cubeSpacing,
                                    dep + dep*cubeSpacing);
                              GameObject c =
                                Instantiate(cubePrefab,
                                    pos,
                                    Quaternion.identity)
                                          as GameObject;

                              //give the cube a name relative
                              //to its position
                              c.name =
                                "Cube_"+row+"_"+col+"_"+dep;


                              //child to the centre cube
                              c.transform.parent =
                                    largeC.transform;
                        }
                    }
                }
            }

            largeC.transform.position = Vector3.zero;


            CameraZoomMax = new Vector3
                  (0,0,(layers + layers*cubeSpacing)*5f);
            CameraZoomMin = new Vector3
                  (0,0,(layers + layers*cubeSpacing)*2f);


            //set starting position of the camera
            this.transform.position =
                        new Vector3(0,0,CameraZoomMax.z);


            this.transform.LookAt(largeC.transform.position);



      }
```

This code is quite long and its worthwhile pausing at this spot to consider what has already been added. As you will see the nested for loops have been included to position smaller cubes as an enclosing skin around a single larger cube. The centre cube is positioned at (0,0,0) and each small cube is childed to the centre cube. This makes it much easier to rotate the entire structure as a whole in the game environment.

The second part of the code (following) takes care of cube rotation using finger swipes or the mouse and camera zooming in and out.

```
      // Update is called once per frame
      void Update ()
      {

         //drag and rotate
         if(Input.GetMouseButtonDown(0) || (Input.touchCount > 0 &&
                  Input.GetTouch(0).phase == TouchPhase.Began))
         {
            lastMousePos = Input.mousePosition;
         }
         else if(Input.GetMouseButtonUp(0) ||
               (Input.touchCount > 0 &&
              Input.GetTouch(0).phase == TouchPhase.Ended))
         {
    }
    else if((Input.touchCount > 0 &&
             Input.GetTouch(0).phase == TouchPhase.Moved)||
             Input.GetMouseButton(0))
    {
       float mouseChangeX = (Input.mousePosition.x -
             lastMousePos.x) * movementSensitivity;
       float mouseChangeY = (Input.mousePosition.y -
             lastMousePos.y) * movementSensitivity;


       largeC.transform.RotateAround(
             largeC.transform.position,
             Vector3.up, -mouseChangeX  * Time.deltaTime);
       largeC.transform.RotateAround(
             largeC.transform.position,
             Vector3.right, -mouseChangeY  * Time.deltaTime);
     }


    //pinch and zoom
    if(Input.touchCount == 2 &&
             (Input.GetTouch(0).phase == TouchPhase.Began ||
             Input.GetTouch(1).phase == TouchPhase.Began) )
 {
             //calibrate previous distance
             previousDistance =
                   Vector2.Distance(Input.GetTouch(0).position,
                   Input.GetTouch(1).position);
 }
 else if (Input.touchCount == 2 &&
         (Input.GetTouch(0).phase == TouchPhase.Moved ||
          Input.GetTouch(1).phase == TouchPhase.Moved) )
       {
         float distance;
        Vector2 touch1 = Input.GetTouch(0).position;
        Vector2 touch2 = Input.GetTouch(1).position;
        distance = Vector2.Distance(touch1, touch2);

        float zChange = (distance - previousDistance);


        previousDistance = distance;
        this.transform.Translate(
             new Vector3(0f,0f,zChange*pinchSensitivity));


        if(this.transform.position.z > CameraZoomMax.z)
             this.transform.position = CameraZoomMax;
        else if(this.transform.position.z < CameraZoomMin.z)
             this.transform.position = CameraZoomMin;



 }
       //zoom with mouse wheel
 else if (Input.GetAxis("Mouse ScrollWheel") != 0)
 {
         this.transform.Translate(new
             Vector3(0f,0f,Input.GetAxis("Mouse ScrollWheel")
                ));
                if(this.transform.position.z > CameraZoomMax.z)
                    this.transform.position = CameraZoomMax;
                else if(this.transform.position.z < CameraZoomMin.z)
                          this.transform.position = CameraZoomMin;


        }
        }
}
```

Save the script. Attach to the Main Camera. Add a directional light to the scene to illuminate the cube.

Before playing add a default Unity cube to the scene and then make a prefab from it. Give the prefab a tag of "cube". Delete the original cube. With the Main Camera selected in the Hierarchy, locate the createCubeLayer script and the exposed Cube Prefab. Drag the newly created cube prefab from your assets into this exposed variable.

By default the layer number is set to 5 as you will see. You can increase this, however if you select a layer greater than 10 you'll begin to see a dramatic performance decrease. The distance the camera can zoom in and out is adjusted according to the layer.

When you run the code, either in the editor or on a mobile device you will be able to rotate the cube and zoom in and out.

Step 3. As the objective of the game is to remove the outer layer of cubes by tapping on them, we will now add the appropriate code by which to do so. Because we are already touching the screen in order to rotate the cube, we don't want to destroy cubes at the same time as it might not be the players intention. Rather, we will have a cube destroyed at the end of a touch on the provision that during the touch a move action did not occur. Modify the createCubeLayer script thus:

```
// Update is called once per frame
bool moving = true;
```

```
void Update ()
{
      //drag and rotate
      if(Input.GetMouseButtonDown(0) || (Input.touchCount > 0 &&
        Input.GetTouch(0).phase == TouchPhase.Began))
      {
            lastMousePos = Input.mousePosition;
            moving = false;
      }
      else if((Input.GetMouseButtonUp(0) ||
         (Input.touchCount > 0 &&
          Input.GetTouch(0).phase == TouchPhase.Ended)) && !moving)
      {
            RaycastHit hit;
            Ray ray =
               Camera.main.ScreenPointToRay(Input.mousePosition);
            if (!Physics.Raycast (ray, out hit, 10000))
                        return;


            if(hit.transform.gameObject.tag == "cube")
            {
                  Destroy(hit.transform.gameObject);
            }


         }
         else if((Input.touchCount > 0 &&
                  Input.GetTouch(0).phase == TouchPhase.Moved)||
                  (Input.GetMouseButton(0) &&
                  Vector3.Distance(Input.mousePosition,lastMousePos)
                 > 0.5))
         {
            ...
            moving = true;
         }
...
```

Save and run. Cubes will be destroyed when you touch on them, but not while rotating the cube.

Step 4. After all the outer cubes have been removed from a layer, we want the next layer down to become covered in cubes. To achieve this we rerun the start script after decreasing the number of layers by 1 and resetting other key variables. Modify the createCubeLayer script thus:

```
...
int totalCubes;
int cubesDestroyed;

// Use this for initialization
void Start ()
```

```
{
      totalCubes = (int)(Mathf.Pow(layers+2, 3) -
                        Mathf.Pow(layers, 3));
      cubesDestroyed = 0;
      int maxCubes = layers*2-1;


      //generate a large inner cube
...

...

void Update()

{

...

      if(cubesDestroyed == totalCubes)
     {
      Destroy(largeC.gameObject);
      layers--;
      Start();
     }
}
```

Save and play. You might like to test by setting the layers to a small number such as 3 otherwise it will take you quite a while to remove the outer layer to see the next layer created.