ISTD 50.017 Graphics & Visualization

Spring Semester 2017

KALEIDOSCOPE

REPORT

By:
Zhexian Zhang | 1001214
William Koh | 1000929
Dhanya Janaki | 1001288

TABLE OF CONTENTS

1. **INTRODUCTION**

Aim

Through this project, we looked to re-create a Kaleidoscope, a well known toy that makes pretty patterns using the physics of reflection and different mirrors. To do this, we used Unity and Blender to simulate mirrors and glossy surfaces, and the objects we wanted and then coded in Unity to get the desired results. We had to try different approaches to make Unity and Blender work together the way we wanted them to, and these methods will be explained later in the report. The summary of our aim is as follows:

- A basic digital kaleidoscope
- Adjustable object count, size, color, and mirror angles
- Virtual reality viewing capability

Technology

As mentioned briefly above, the technologies used are as follows

- Unity Engine
- Blender
- Google VR API
- Android

2. **LITERATURE** REVIEW

Web-Based Kaleidoscopic Effect (CSS3)[1]

This kaleidoscope effect is produced with SASS-powered CSS rotation and scale transforms.

We learn from the project how mirror rotation and polar pattern may be created based on image segments:

Rotation:

```
@mixin transform-rotate ($degrees) {
   $value: rotate($degrees);
   @include prefix-values('transform',
$value);
}
```

Mirror-Rotation:

```
@mixin transform-mirror-rotate
($degrees) {
   $values: scale(-1, 1)
rotate($degrees);
   @include prefix-values('transform',
$values);
}
```

Kaleidoscopic Effect:

```
$max_segments: 32;
$segments: 0;

    @while $segments <= $max_segments {
        $step: 360deg / ($segments * 2);

        $angle: 0 - $step;
        $i: 0;
```

[1] Source: https://github.com/coldhead/kaleidos

```
        &.n#{$segments} {
                @while $i < $segments * 2 {
                    &.t#{$i} {
                       @include
   transform-rotate($angle + $step);
                    }
                    &.t#{$i+1} {
                       @include
   transform-mirror-rotate($angle + $step);
                    }
                    $angle: $angle + ($step*2);
                    $i: $i + 2;
                }
                .image {
                    @include
   transform-rotate($step);
                }
            }
        }
        $segments: $segments + 1;
    }
```

Custom variables:

- n: a positive integer representing the number of "segments" to create within the kaleidoscope. The project supports up to 32 segments but the effect works best with lower values.
- src: URL for the input image.

Kaleidoscopic Visual Art (Processing.org)[2]

This project is based on Processing[3], a software sketchbook and a language for coding within the context of the visual arts. It serves as good reference of how circular patterns may be created.

Main Drawing Loop (drawArt with step rotation):

```
void draw() {
   buffer.beginDraw();


   buffer.background(0x15FFFFFF);


   if (step > 29) up = false;
   if (step < 3) up = true;
   if (up) step++; else step--;
   drawArc(step);
   buffer.endDraw();


   rotate += 0.005;
     kaleidoscope.draw(buffer, rotate);


}
```

Draw Arc:

```
void drawArc(int steps) {
   colorcycle += 0.04;
   movement += 0.031;
```

[2] Source: https://github.com/dbu/kaleidoscope
[3] Processing: https://processing.org/

```
    movement2 += 0.022;

    buffer.stroke(sin(colorcycle)*255, 0, cos(colorcycle)*255, 255);


    int s = buffer.width;

    float factor = (float) s / steps;

    for (int i=0; i<=steps; i++) {

      buffer.line(i*factor,0,(0.4 + 0.6*abs(sin(movement))) * (s - i *
factor),(0.2*abs(cos(movement2))+0.2) * (s - i * factor));

    }

}
```

Custom variables:

- 0 to 9: set number of mirror axis
- r / R: accelerate/decelerate rotation of the resulting image

Android Kaleidoscope App (Java)[4]

This is the source code of an Kaleidoscope app listed on Google Play Store[5].
Among all the functions, the paint() function is responsible for rendering kaleidoscopic images and it serves our code reference.

Paint (rotation angle based on number of mirrors):

```
 private synchronized void paint(Canvas canvas, Bitmap bitmap) {
                float angle = mStartAngle;
                for (int i = 0; i < mNumberOfMirrors; i++) {
                        canvas.save();
                        canvas.translate(mCenterX, mCenterY);
                        canvas.rotate(angle);
                        canvas.drawBitmap(bitmap, mOffset, mTopOffset, mPaint);
                        canvas.scale(1, -1);
                        canvas.drawBitmap(bitmap, mOffset, mTopOffset, mPaint);
                        canvas.restore();
                        angle += mAngle2;
                }
        }
```

Paint (with pre-defined radius for translation):

```
 private synchronized void paint(Canvas canvas, Bitmap bitmap, int radius) {
                Paint p = new Paint();
                p.setAntiAlias(true);
                float angle = mStartAngle;
                for (int i = 0; i < mNumberOfMirrors; i++) {
                        canvas.save();
                        canvas.translate(radius, radius);
                        canvas.rotate(angle);
                        canvas.drawBitmap(bitmap, mOffset, mTopOffset, mPaint);
                        canvas.scale(1, -1);
                        canvas.drawBitmap(bitmap, mOffset, mTopOffset, mPaint);
                        canvas.restore();
                        angle += mAngle2;
                }
        }
```

[4] Source: https://github.com/prrt714/Kaleidoscope/
[5] App address: https://play.google.com/store/apps/details?id=vnd.bluearat.kaleidoscope6

Unity Kaleidoscopic & Mirror Effect (C#)[6]

Named “KinoMirror”, this project is a mirroring/kaleidoscope image effect for Unity.

The Mirror class is close to what we aim to achieve for mirror effects in the Kaleidoscope project.

Mirror Class (C#):

```
public class Mirror : MonoBehaviour
     {
         #region Public Properties
         [SerializeField]
         int _repeat;
         [SerializeField]
         float _offset;
         [SerializeField]
         float _roll;
         [SerializeField]
         bool _symmetry;
         #endregion


         #region Private Properties
         [SerializeField] Shader _shader;
         Material _material;
         #endregion


         #region MonoBehaviour Functions
         void OnRenderImage(RenderTexture source, RenderTexture destination)
         {
             if (_material == null)
             {
                 _material = new Material(_shader);
                 _material.hideFlags = HideFlags.DontSave;
             }
```

[6] Source: https://github.com/keijiro/KinoMirror

```
            var div = Mathf.PI * 2 / Mathf.Max(1, _repeat);
            _material.SetFloat("_Divisor", div);
            _material.SetFloat("_Offset", _offset * Mathf.Deg2Rad);
            _material.SetFloat("_Roll", _roll * Mathf.Deg2Rad);


            if (_symmetry)
                _material.EnableKeyword("SYMMETRY_ON");
            else
                _material.DisableKeyword("SYMMETRY_ON");


            Graphics.Blit(source, destination, _material);
        }
        #endregion
    }
```

Mirror Shader (.shader): note the angular repeating and reflection portion highlighted

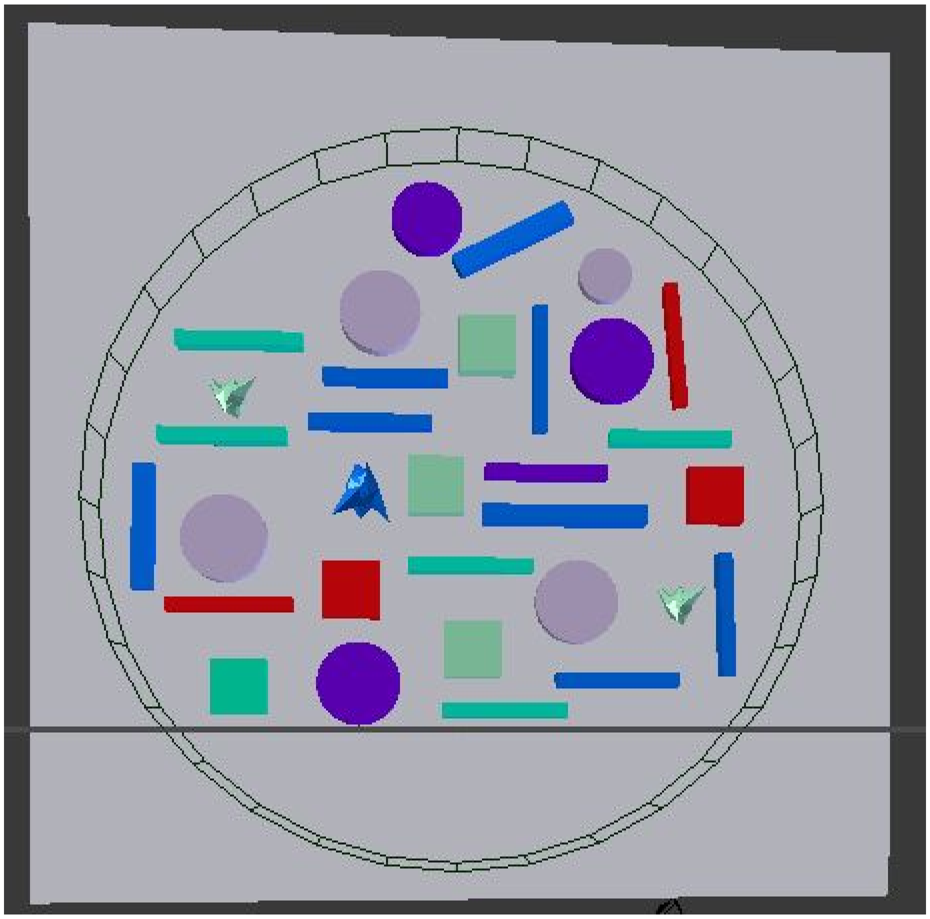
```
Shader "Hidden/Kino/Mirror"
  {
     Properties
     {
         _MainTex ("-", 2D) = "" {}
     }
     CGINCLUDE
     #include "UnityCG.cginc"
     #pragma multi_compile _ SYMMETRY_ON
     sampler2D _MainTex;
     float _Divisor;
     float _Offset;
     float _Roll;


     half4 frag(v2f_img i) : SV_Target
     {
         // Convert to the polar coordinate.
         float2 sc = i.uv - 0.5;
         float phi = atan2(sc.y, sc.x);
         float r = sqrt(dot(sc, sc));


         // Angular repeating.
         phi += _Offset;
         phi = phi - _Divisor * floor(phi / _Divisor);
         #if SYMMETRY_ON
         phi = min(phi, _Divisor - phi);
         #endif
         phi += _Roll - _Offset;


         // Convert back to the texture coordinate.
         float2 uv = float2(cos(phi), sin(phi)) * r + 0.5;


         // Reflection at the border of the screen.
         uv = max(min(uv, 2.0 - uv), -uv);


         return tex2D(_MainTex, uv);
     }


     ENDCG
     SubShader
     {
         Pass
         {
             ZTest Always Cull Off ZWrite Off
             CGPROGRAM
             #pragma vertex vert_img
             #pragma fragment frag
             ENDCG
         }
     }
  }
```

3. **DEVELOPMENT** PROCESS

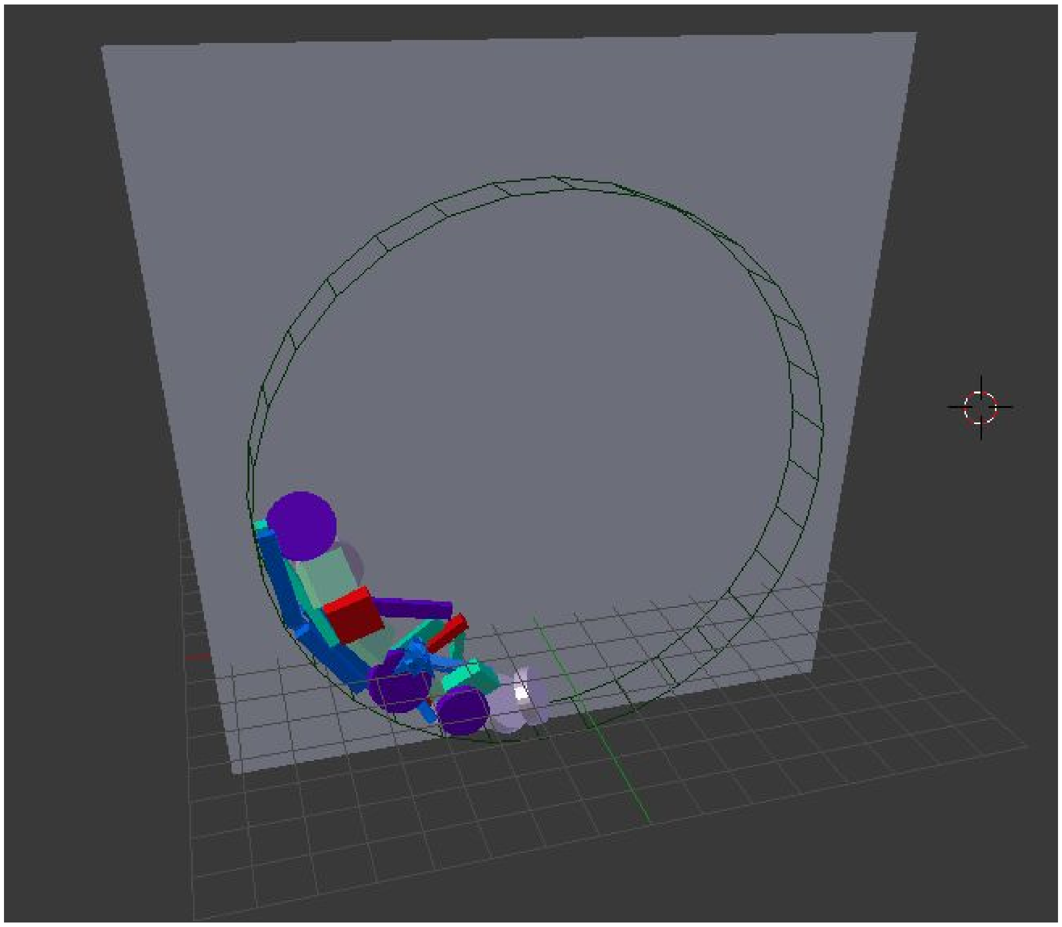
Attempt 1: Blender + Unity

In our first attempt, we tried to follow the tutorial mentioned in the project proposal. To do so, we wanted to Blender to create the objects, the base for the kaleidoscope and the rotational motion. Then we wanted to port them over to Unity, and apply the mirrors to get the Kaleidoscope effect and add in the user controls and VR part.

For the blender part, it initially took a while to get used to the controls. After multiple attempts, we ended up with something that looked like this:
There are multiple shapes as shown below. There is also a rotation that is activated, and the wireframe is the circle around the objects.



This is how it looks like in the middle of its rotations:

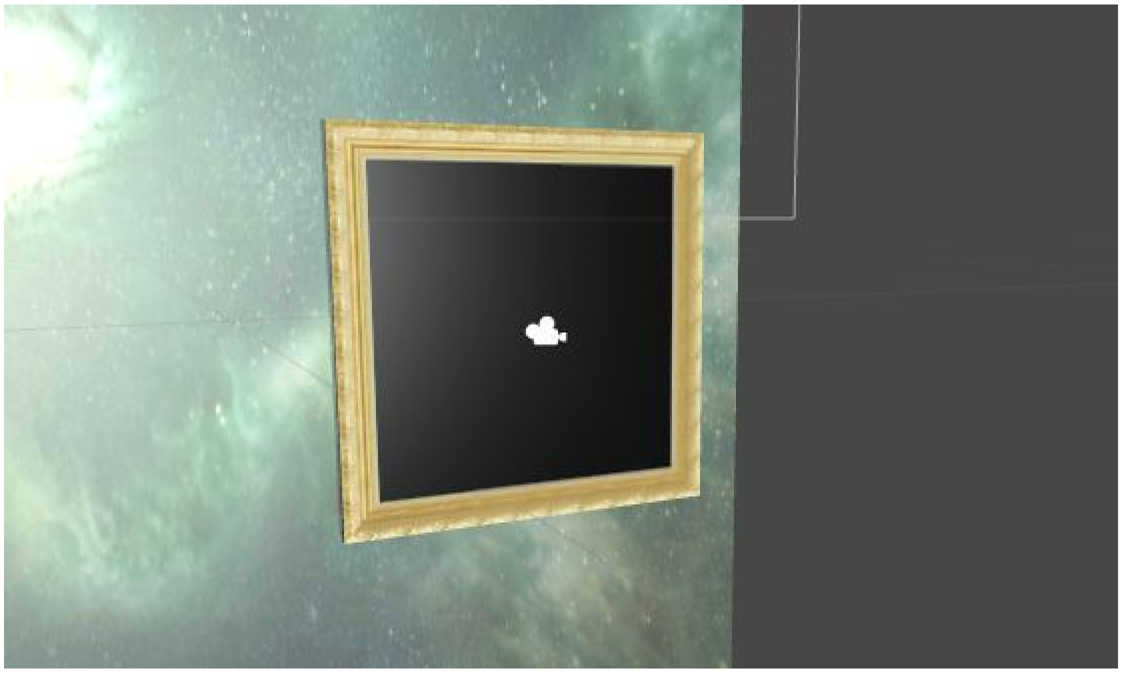


We made the objects Rigid Bodies, and the cylinder (wireframe) a passive object, and enabled collisions between them. We also set the rotation time, and increased the friction and collision constant of the objects to make them

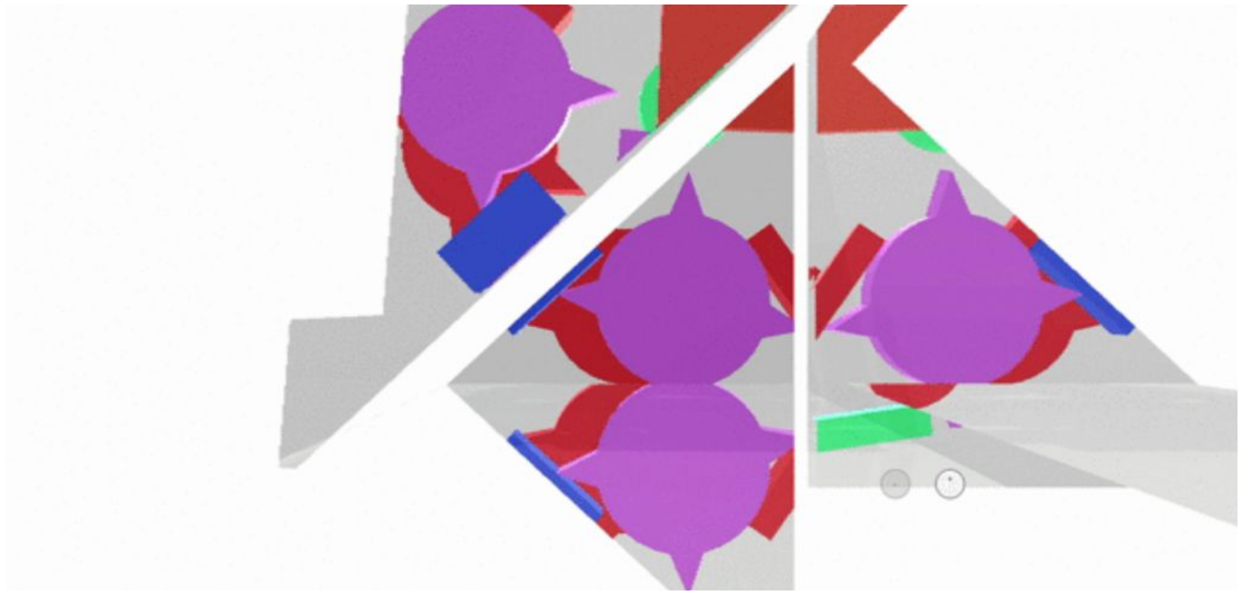
a) Bounce around in a more realistic fashion
b) Have the objects move along with the writeframe for a while before falling so that it would seem more like a kaleidoscope.

The Blender part seemed to work fine, and on rendering, we could see the kaleidoscope working pretty well. We also tried to make the mirror in Blender, but it wouldn't work in

Unity properly because of some material errors, so we decided to make the mirrors in Unity.
How the mirror is done in Unity is through a shader and a secondary camera. Firstly, we create a thin cube which is the physical mirror. Then we created a mirror texture that will reflect whatever is being filmed by the secondary camera. The screenshot below show the structure of mirror.

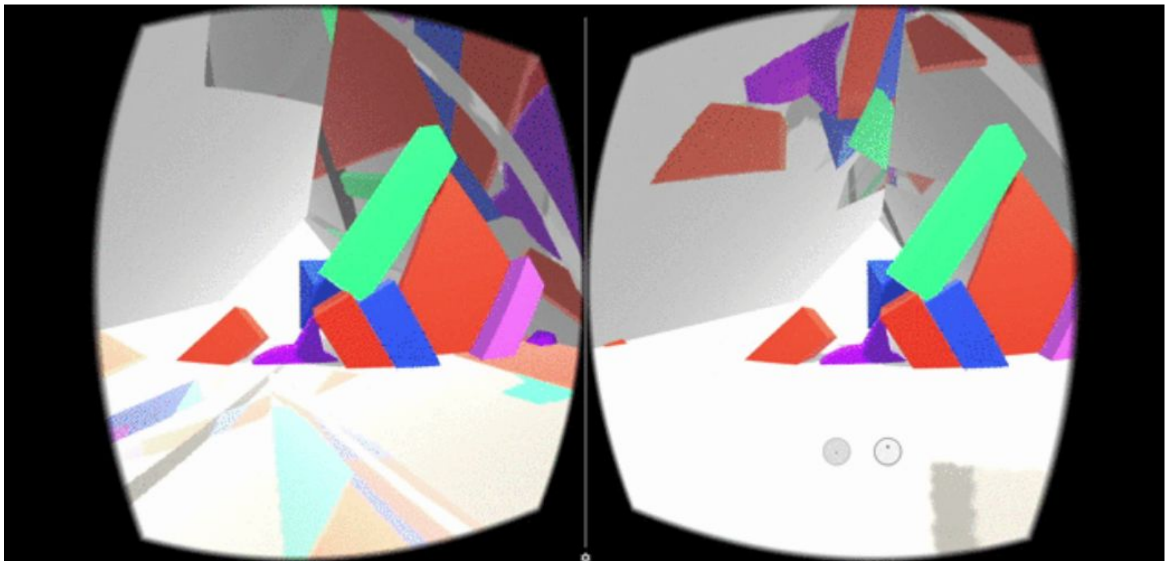


The secondary camera is placed in front of the mirror and the scene that is being filmed will then be reflected on the physical mirror. Next we created 2 more mirrors and rearrange them to right-triangle shape. After that, the main camera is placed at the center of these 3 mirrors to display the effect of kaleidoscope. But the effect is far away from the expectation. The screenshot below shows the outcome of this attempt.



We weren't quite satisfied with these results because:

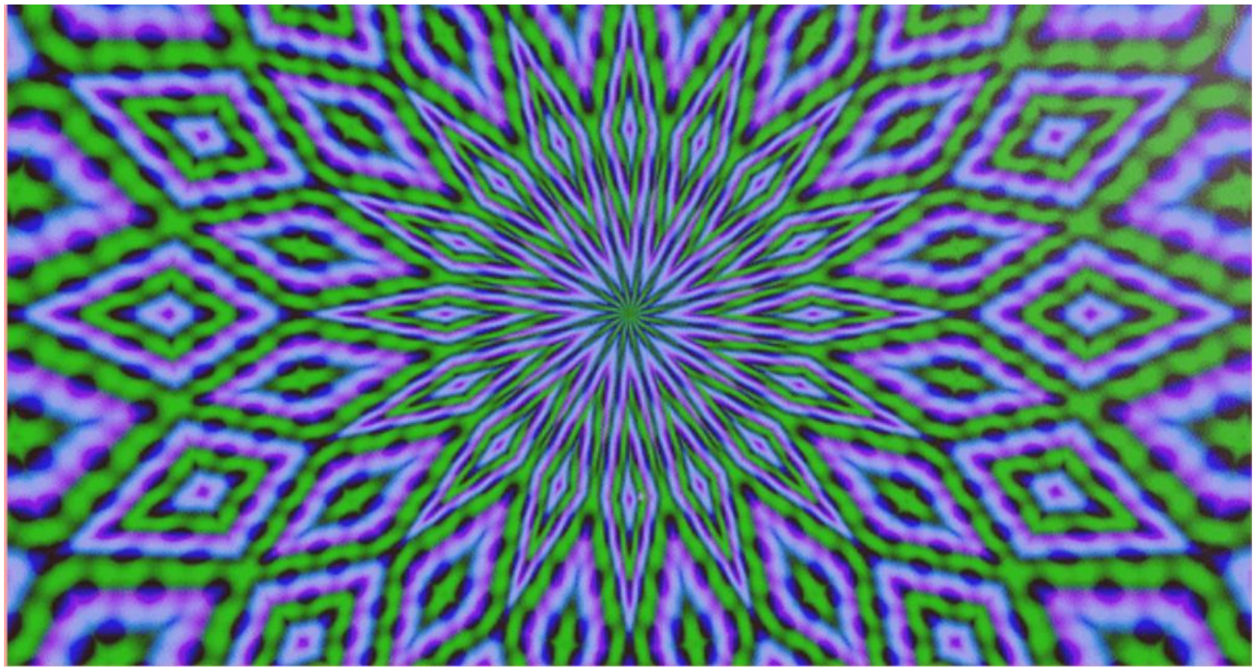
a) The objects didn't look very realistic
b) The glossy surface was clearly not working very well or simulating very good results
c) Moreover, on VR this is how it looked:



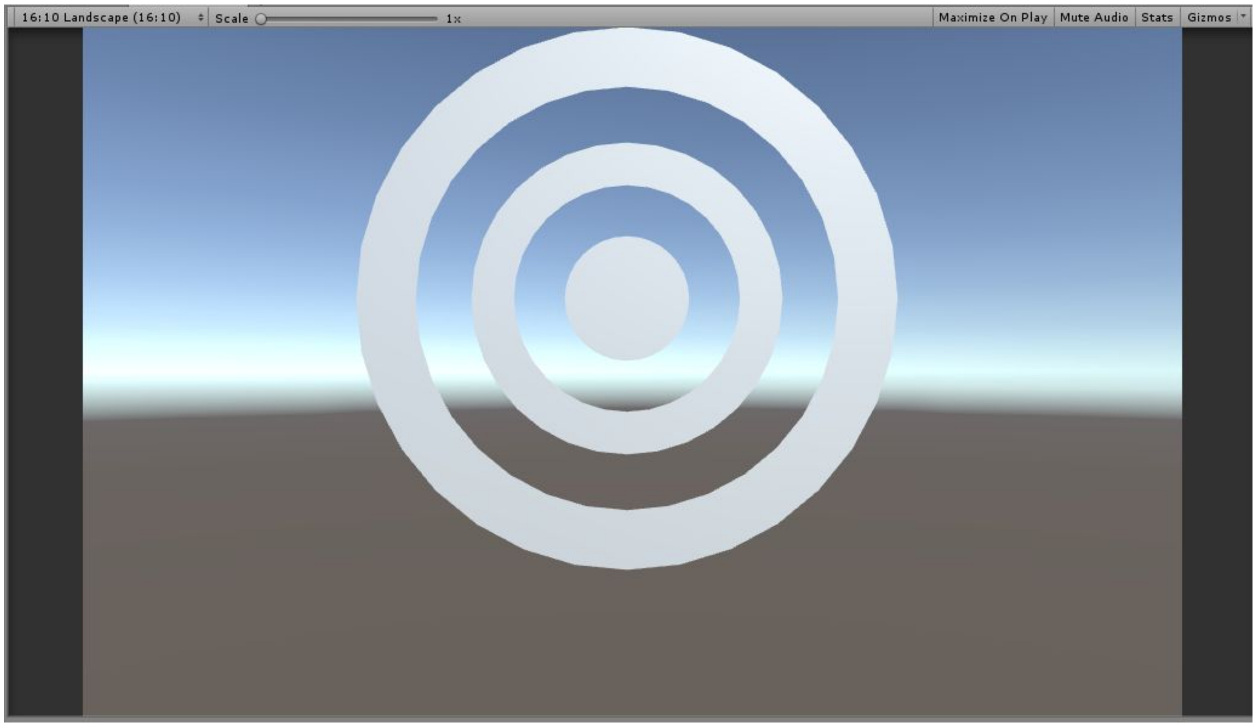
We believed that this did not show an accurate depiction, as it felt more like seeing inside a kaleidoscope and seeing how things rotated. Therefore, we decided to try some other methods.

Attempt 2: Blender

For the second attempt, we tried to use only Blender to create a texture and then import that over to Unity to render. Doing this on Blender involved replicating a glossy surface multiple time to get a sector of a circle, and then copying that 16 times to make it a complete circle, and then giving it a colour, that got reflected. This would also rotate, giving the kaleidoscope effect. In this case, what we had in Blender looked like this:



After the successful attempt on blender, we tried to export it to Unity to see if the rendering effect still works. It turned out that the texture and remained the same as the one in Blender, but the animator that exported to Unity doesn’t work as expected.. The following screenshot is the outcome that we get at the end of the attempt.

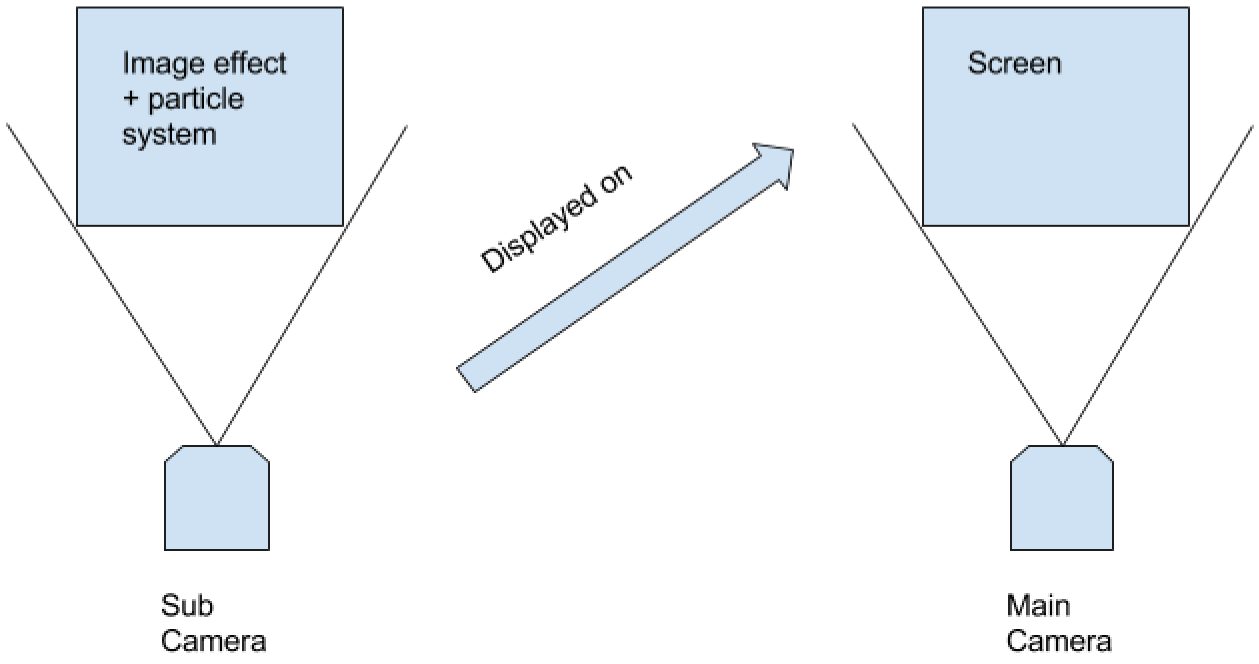


With this failed attempt, we eventually thought of doing image effect on Unity rather than doing on Blender and export.

Attempt 3: Unity

For the final attempt on Unity, rather than trying to build a physical kaleidoscope which contains physical mirrors and small objects, we made full use of the image rendering effect and particle system on Unity to get the similar effect of physical kaleidoscope.

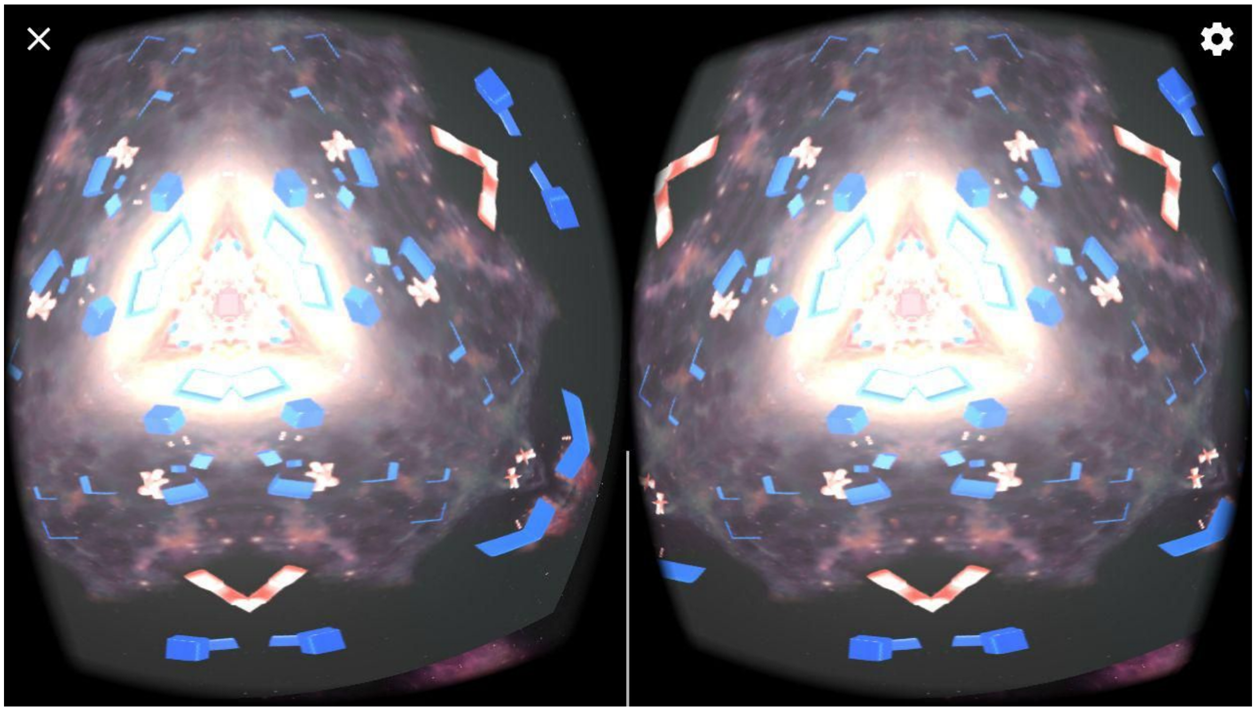
The diagram below illustrates the idea of kaleidoscope effect.

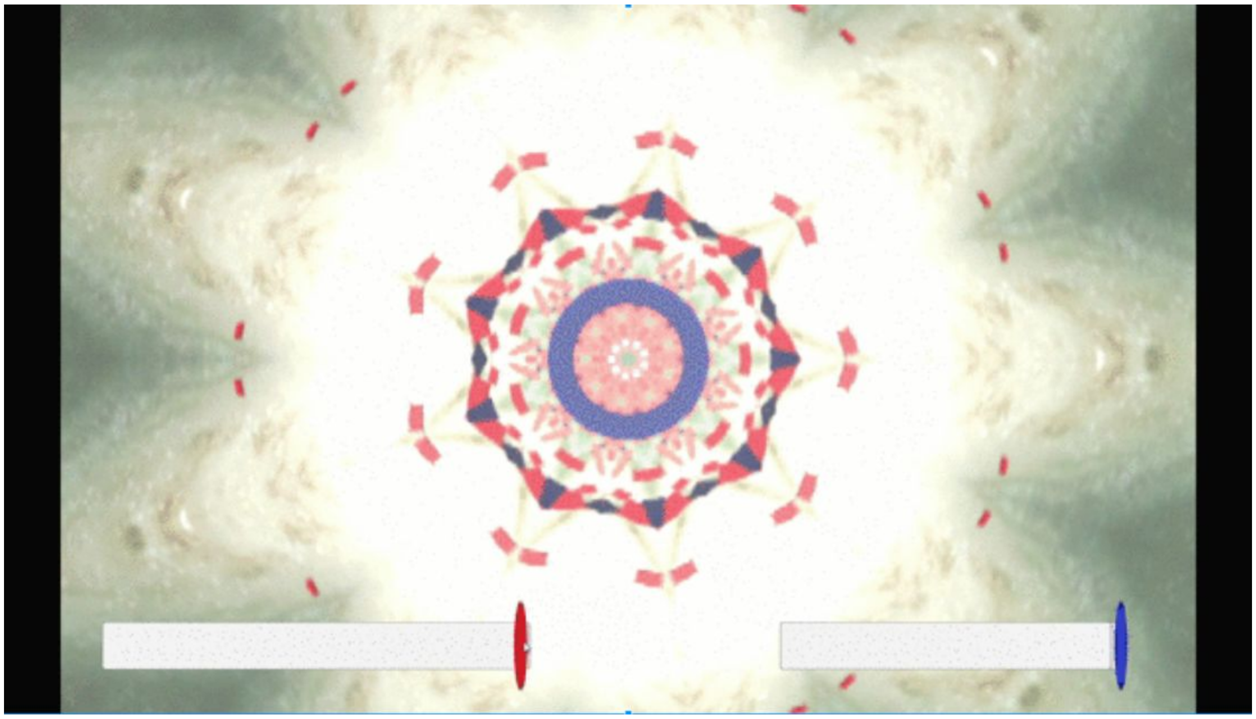


We realized that the image effect that we created did not work on Google VR as Google VR automatically replace the view with VR view when it is in VR mode. So we only can see the original scene in VR view. In order to address the problem, we came out of an idea where we reuse the mirror effect that we created in the first attempt.

From the diagram above, we can see there are 2 camera which is known as sub camera and main camera. The Sub camera film at the skybox and particle systems that are placed in front of the camera. The scene is then process and produce the image effect to the Sub camera.

The screenshot below shows both scene with and without VR mode.





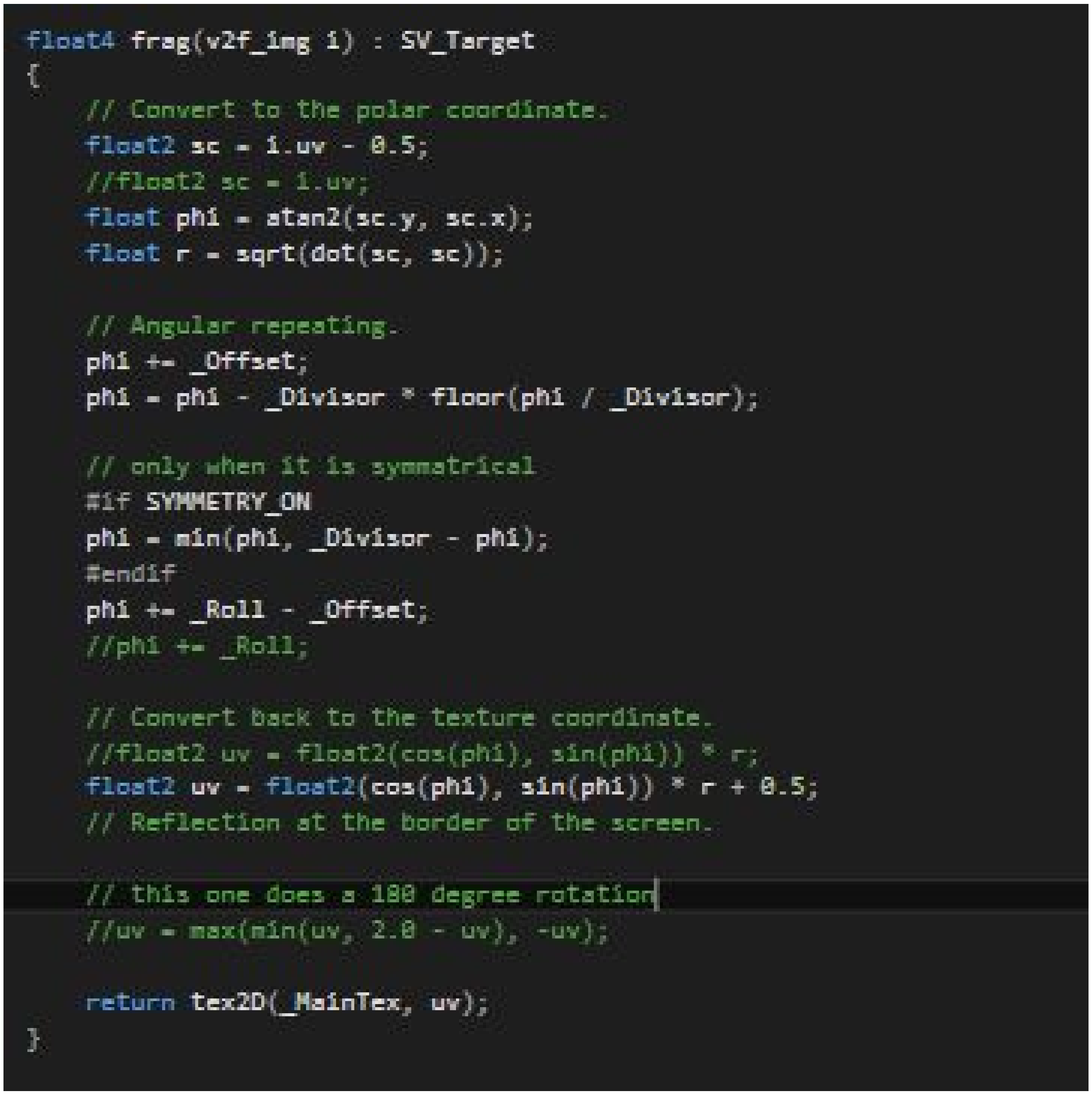
Programming Scripts

Image Effect Processing

The image effect processing consists of 2 important component, which are shader and a processing script.

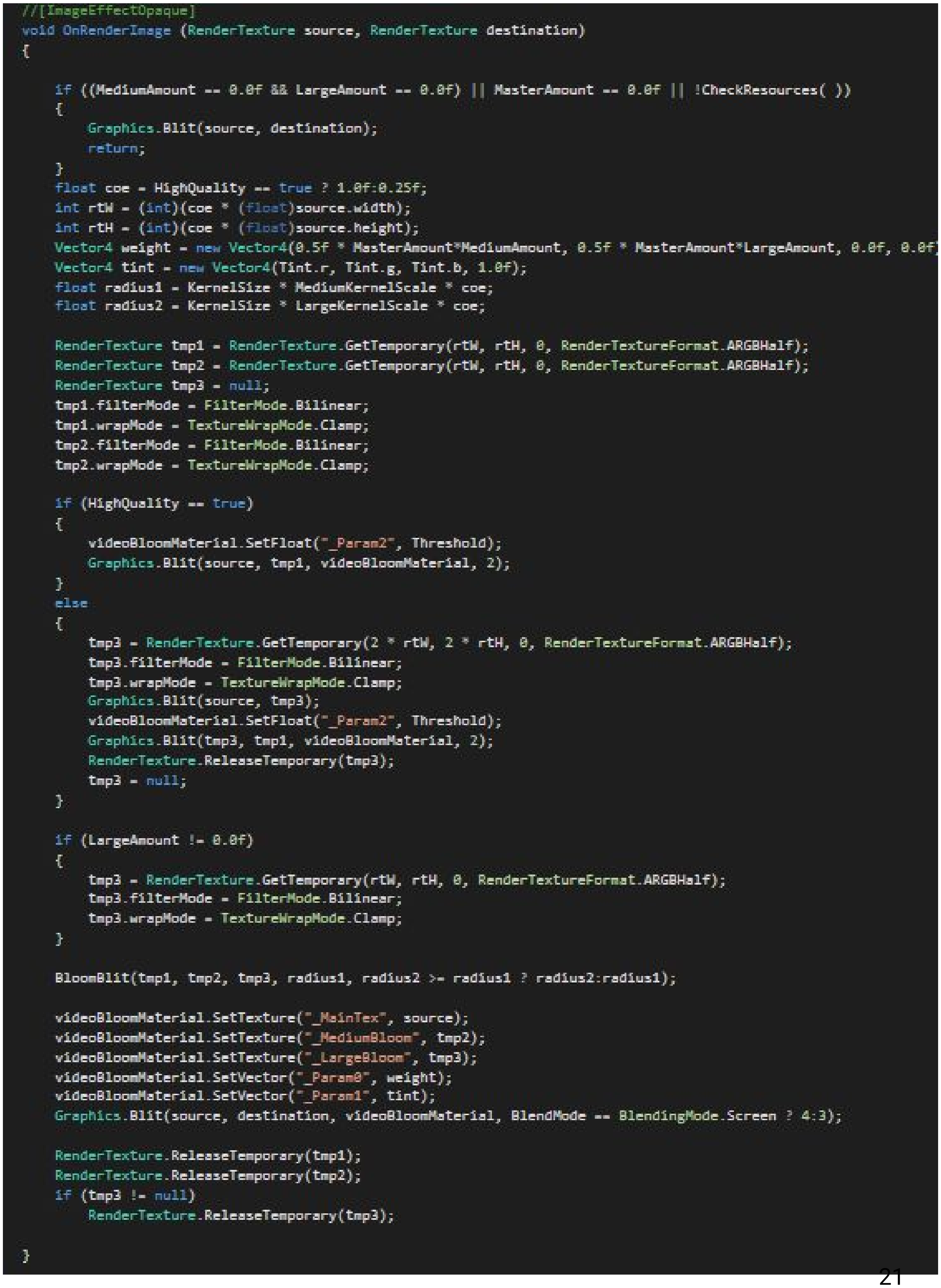
In this context, shader is the core component that provides the mirror effect to the camera. The shader first takes in the skybox and objects in the scene. Then it will get the coordination of their 3D texture and split it into different section, based on the user input (number of mirror). For each section, there is a line that acts as a mirror where it form the reflection effect of them object. Then shader will return the 2D texture of the scene.

While the processing script will get the output from the shader and form the scene and pass to the camera. The screenshots below are the shader and processing script.



```
float4 frag(v2f_img i) : SV_Target
{
    // Convert to the polar coordinate.
    float2 sc = i.uv - 0.5;
    //float2 sc = i.uv;
    float phi = atan2(sc.y, sc.x);
    float r = sqrt(dot(sc, sc));

    // Angular repeating.
    phi += _Offset;
    phi = phi - _Divisor * floor(phi / _Divisor);

    // only when it is symmatrical
    #if SYMMETRY_ON
    phi = min(phi, _Divisor - phi);
    #endif
    phi += _Roll - _Offset;
    //phi += _Roll;

    // Convert back to the texture coordinate.
    //float2 uv = float2(cos(phi), sin(phi)) * r;
    float2 uv = float2(cos(phi), sin(phi)) * r + 0.5;
    // Reflection at the border of the screen.

    // this one does a 180 degree rotation
    //uv = max(min(uv, 2.0 - uv), -uv);

    return tex2D(_MainTex, uv);
}
```

In *frag* field, we calculated the angular motion for each segments and check if the symmetrical mode is on. Then the function converts the angle into texture coordinates and the reflection takes place at the border of the screen.

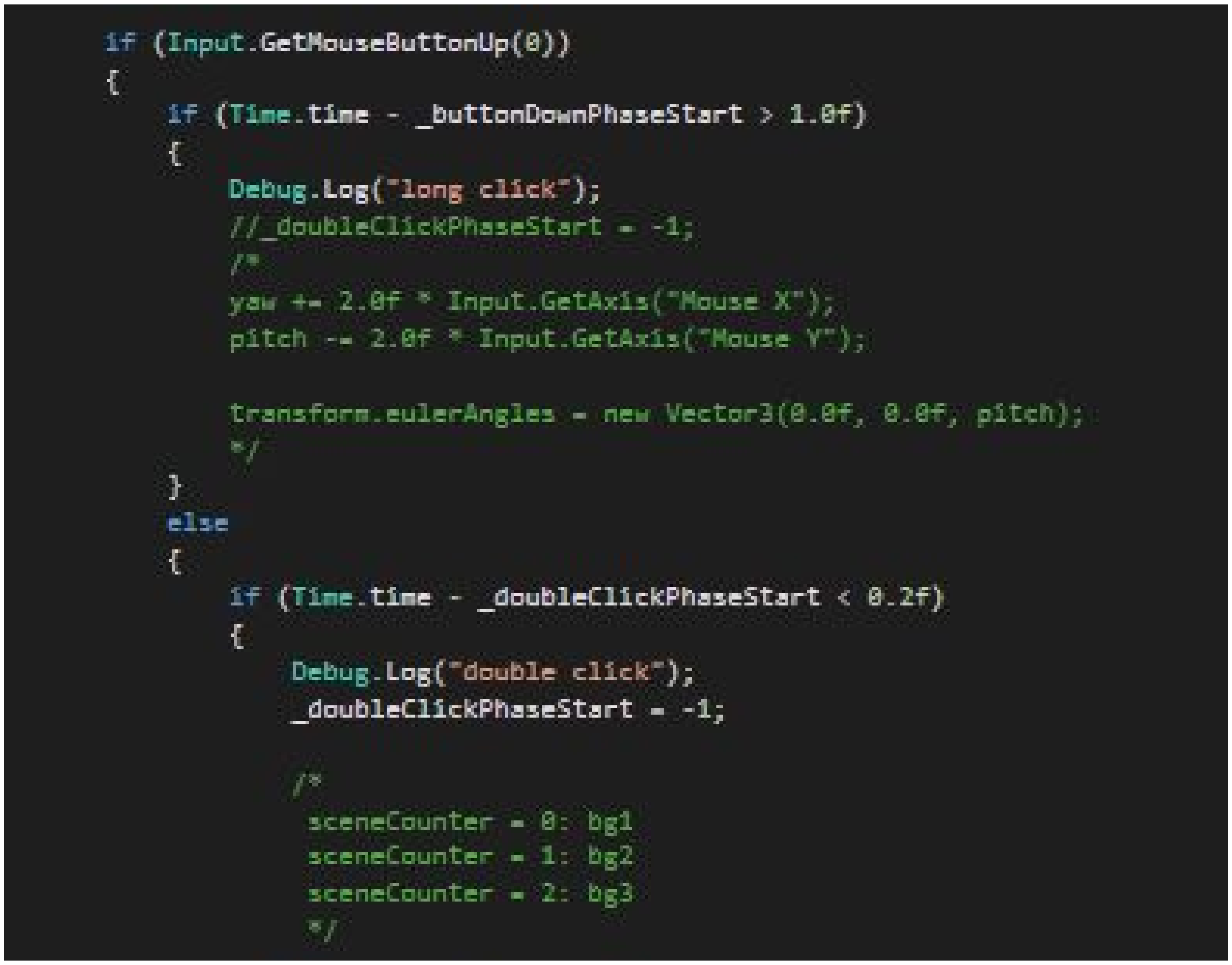


```
//[ImageEffectOpaque]
void OnRenderImage (RenderTexture source, RenderTexture destination)
{

    if ((MediumAmount == 0.0f && LargeAmount == 0.0f) || MasterAmount == 0.0f || !CheckResources( ))
    {
        Graphics.Blit(source, destination);
        return;
    }
    float coe = HighQuality == true ? 1.0f:0.25f;
    int rtW = (int)(coe * (float)source.width);
    int rtH = (int)(coe * (float)source.height);
    Vector4 weight = new Vector4(0.5f * MasterAmount*MediumAmount, 0.5f * MasterAmount*LargeAmount, 0.0f, 0.0f
    Vector4 tint = new Vector4(Tint.r, Tint.g, Tint.b, 1.0f);
    float radius1 = KernelSize * MediumKernelScale * coe;
    float radius2 = KernelSize * LargeKernelScale * coe;

    RenderTexture tmp1 = RenderTexture.GetTemporary(rtW, rtH, 0, RenderTextureFormat.ARGBHalf);
    RenderTexture tmp2 = RenderTexture.GetTemporary(rtW, rtH, 0, RenderTextureFormat.ARGBHalf);
    RenderTexture tmp3 = null;
    tmp1.filterMode = FilterMode.Bilinear;
    tmp1.wrapMode = TextureWrapMode.Clamp;
    tmp2.filterMode = FilterMode.Bilinear;
    tmp2.wrapMode = TextureWrapMode.Clamp;

    if (HighQuality == true)
    {
        videoBloomMaterial.SetFloat("_Param2", Threshold);
        Graphics.Blit(source, tmp1, videoBloomMaterial, 2);
    }
    else
    {
        tmp3 = RenderTexture.GetTemporary(2 * rtW, 2 * rtH, 0, RenderTextureFormat.ARGBHalf);
        tmp3.filterMode = FilterMode.Bilinear;
        tmp3.wrapMode = TextureWrapMode.Clamp;
        Graphics.Blit(source, tmp3);
        videoBloomMaterial.SetFloat("_Param2", Threshold);
        Graphics.Blit(tmp3, tmp1, videoBloomMaterial, 2);
        RenderTexture.ReleaseTemporary(tmp3);
        tmp3 = null;
    }

    if (LargeAmount != 0.0f)
    {
        tmp3 = RenderTexture.GetTemporary(rtW, rtH, 0, RenderTextureFormat.ARGBHalf);
        tmp3.filterMode = FilterMode.Bilinear;
        tmp3.wrapMode = TextureWrapMode.Clamp;
    }

    BloomBlit(tmp1, tmp2, tmp3, radius1, radius2 >= radius1 ? radius2:radius1);

    videoBloomMaterial.SetTexture("_MainTex", source);
    videoBloomMaterial.SetTexture("_MediumBloom", tmp2);
    videoBloomMaterial.SetTexture("_LargeBloom", tmp3);
    videoBloomMaterial.SetVector("_Param0", weight);
    videoBloomMaterial.SetVector("_Param1", tint);
    Graphics.Blit(source, destination, videoBloomMaterial, BlendMode == BlendingMode.Screen ? 4:3);

    RenderTexture.ReleaseTemporary(tmp1);
    RenderTexture.ReleaseTemporary(tmp2);
    if (tmp3 != null)
        RenderTexture.ReleaseTemporary(tmp3);

}
```

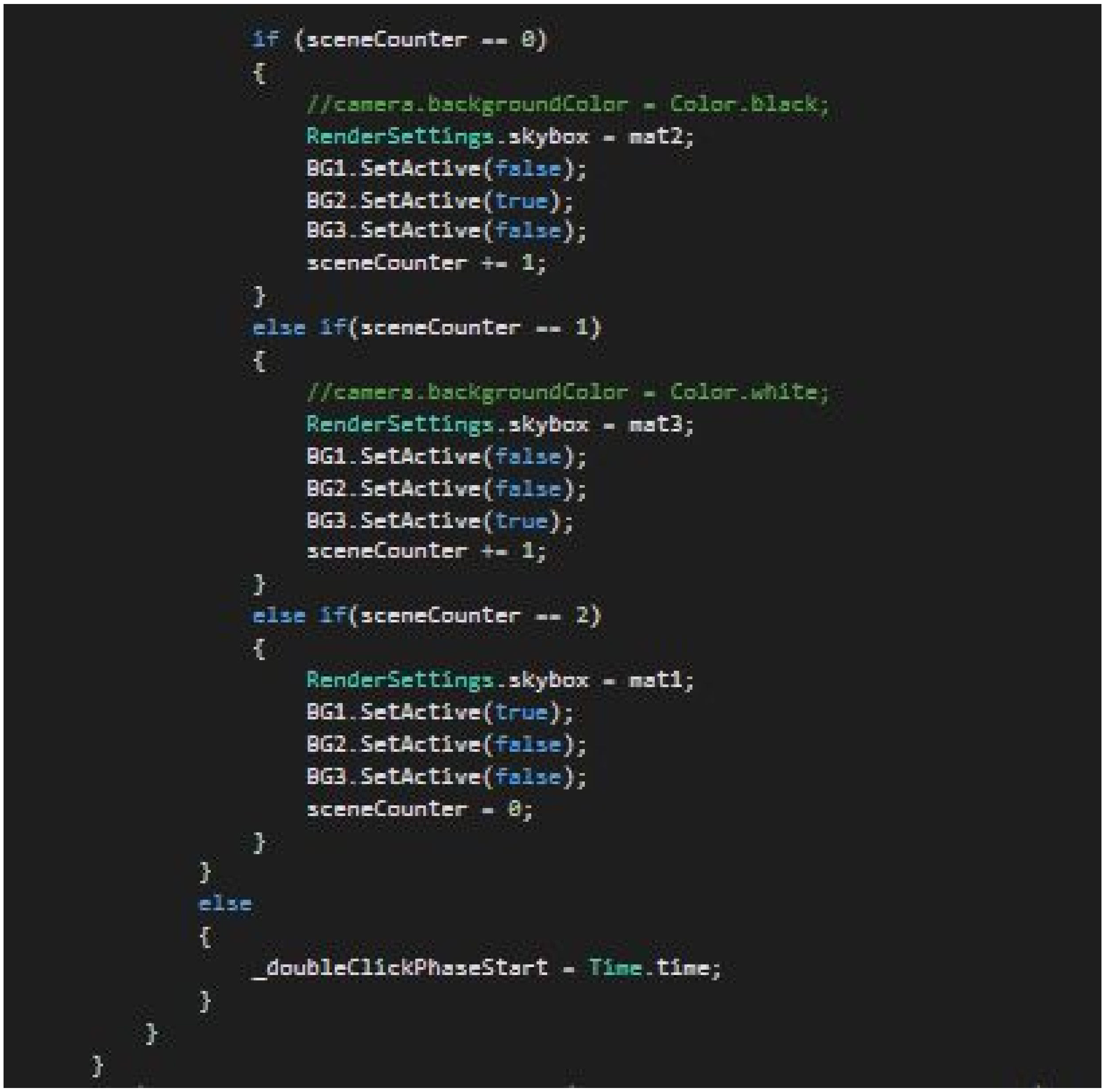
OnRenderImage renders the scene by taking in the output from the shader and parse it to the camera.s

Switch Scene

We have prepared 3 scenes for user to toggle. The scene can be switched by double tapping on the screen. The screenshot below shows the function that detect the double tapping from user.



```
if (Input.GetMouseButtonUp(0))
{
    if (Time.time - _buttonDownPhaseStart > 1.0f)
    {
        Debug.Log("long click");
        //_doubleClickPhaseStart = -1;
        /*
        yaw += 2.0f * Input.GetAxis("Mouse X");
        pitch -= 2.0f * Input.GetAxis("Mouse Y");

        transform.eulerAngles = new Vector3(0.0f, 0.0f, pitch);
        */
    }
    else
    {
        if (Time.time - _doubleClickPhaseStart < 0.2f)
        {
            Debug.Log("double click");
            _doubleClickPhaseStart = -1;

            /*
             sceneCounter = 0: bg1
             sceneCounter = 1: bg2
             sceneCounter = 2: bg3
             */
```



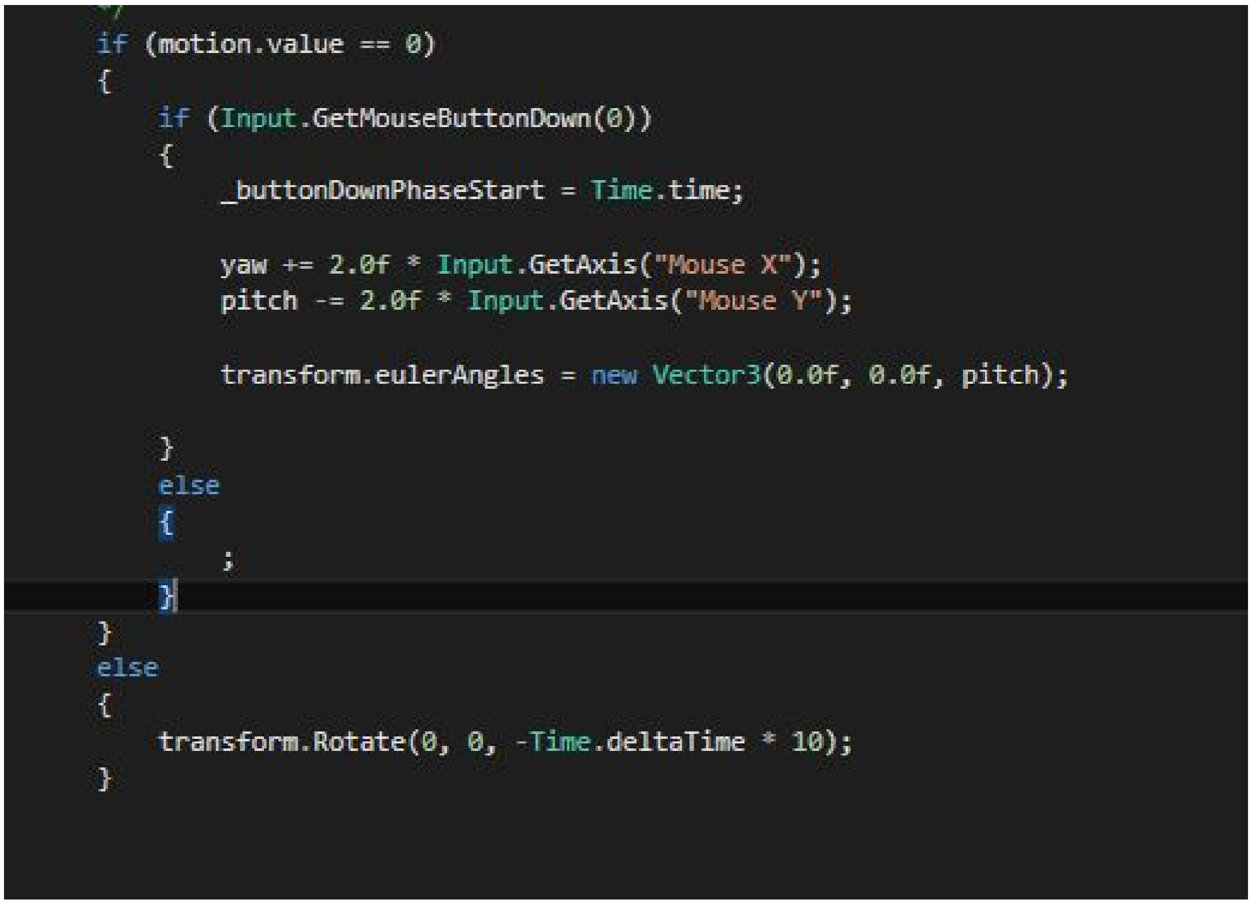
```
                if (sceneCounter == 0)
                {
                    //camera.backgroundColor = Color.black;
                    RenderSettings.skybox = mat2;
                    BG1.SetActive(false);
                    BG2.SetActive(true);
                    BG3.SetActive(false);
                    sceneCounter += 1;
                }
                else if(sceneCounter == 1)
                {
                    //camera.backgroundColor = Color.white;
                    RenderSettings.skybox = mat3;
                    BG1.SetActive(false);
                    BG2.SetActive(false);
                    BG3.SetActive(true);
                    sceneCounter += 1;
                }
                else if(sceneCounter == 2)
                {
                    RenderSettings.skybox = mat1;
                    BG1.SetActive(true);
                    BG2.SetActive(false);
                    BG3.SetActive(false);
                    sceneCounter = 0;
                }
            }
            else
            {
                _doubleClickPhaseStart = Time.time;
            }
        }
    }
```

Other than detecting double tapping, we also use a counter to check which scene to change to.

Manual and Auto Rotation

Rotation is controlled by a binary slider. When value is zero, the auto rotation is off and user can manually rotate the kaleidoscope by sliding up and down on the screen. While in VR mode, the kaleidoscope is in auto-rotation mode because user can't touch the screen.

The following are the codes that control the rotation of kaleidoscope.

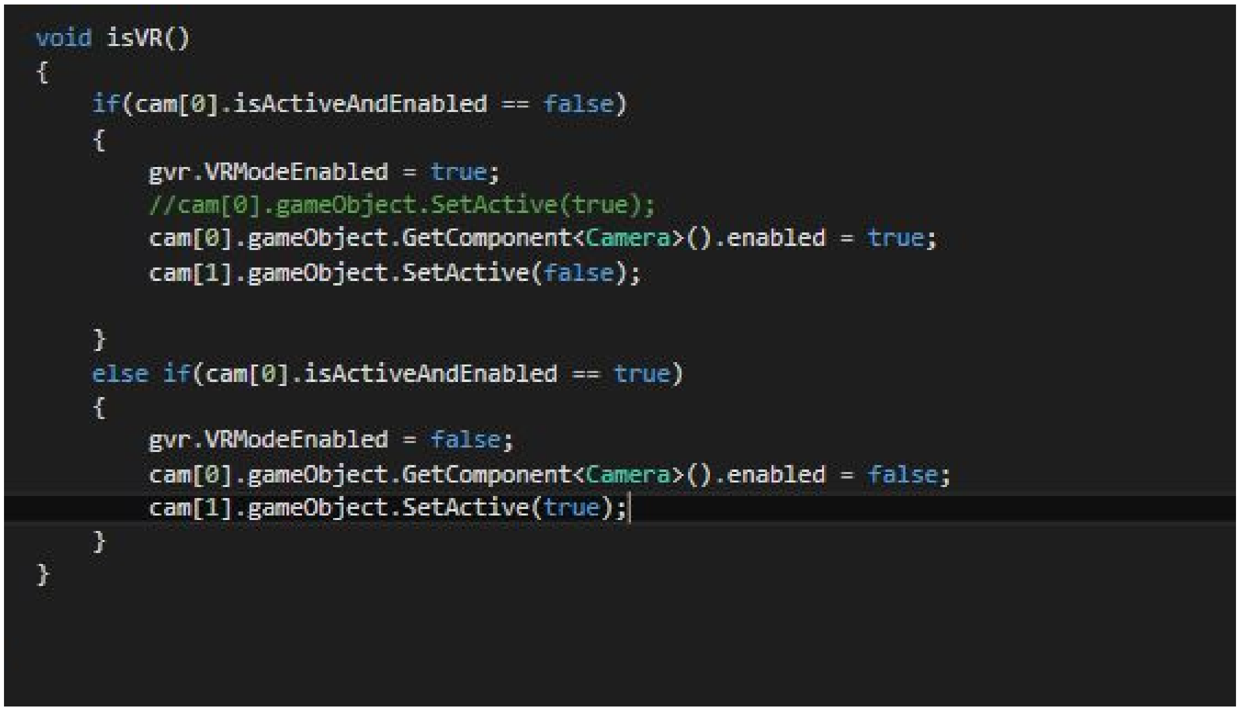


```
if (motion.value == 0)
{
    if (Input.GetMouseButtonDown(0))
    {
        _buttonDownPhaseStart = Time.time;

        yaw += 2.0f * Input.GetAxis("Mouse X");
        pitch -= 2.0f * Input.GetAxis("Mouse Y");

        transform.eulerAngles = new Vector3(0.0f, 0.0f, pitch);

    }
    else
    {
        ;
    }
}
else
{
    transform.Rotate(0, 0, -Time.deltaTime * 10);
}
```

Google VR

For the VR part of the project, we implemented Google VR SDK. This is a Unity plugin from google where it allows users to build virtual reality Android and iOS apps. How this plugin works is there is a VR main controller where it basically controls all the the configuration and display of VR view. In order to use it, we need to drag the main controller prefab into the scene. Then, the VR main controller will automatically take over the control of the scene view by adding a simulation of Google Cardboard.

Since our program have VR mode and normal mode, we created a button for user to toggle between these 2 modes. In order to access the VR controller, we have a script called VRController where it accesses the GvrViewer (script of main controller). Inside VRController, there is a function which controls the on and off of VR mode. We compare the state of the camera and then switch on and off the VR mode accordingly. The following is the screenshot of VRController script.



```
void isVR()
{
    if(cam[0].isActiveAndEnabled == false)
    {
        gvr.VRModeEnabled = true;
        //cam[0].gameObject.SetActive(true);
        cam[0].gameObject.GetComponent<Camera>().enabled = true;
        cam[1].gameObject.SetActive(false);

    }
    else if(cam[0].isActiveAndEnabled == true)
    {
        gvr.VRModeEnabled = false;
        cam[0].gameObject.GetComponent<Camera>().enabled = false;
        cam[1].gameObject.SetActive(true);
    }
}
```

We first check if the program is in VR mode. If is it not in VR mode, we will turn on the VR mode and turn on the main camera as well. Otherwise, we will turn off the VR mode, the main camera and turn on the sub camera so the user will be able to see the normal view.

4. **FINAL** PRODUCT

In the following part, we will explain our final design further by looking through the following parts of the product:

Scene background

There were three different types of scene backgrounds that the user could toggle by double tapping the screen. This would trigger a change in the image that was put in the skybox, and it gave the users different objects and backgrounds to look at.

2D and immersive virtual reality experience

Rather than building a physical kaleidoscope and providing the realistic effect, we use particle system with 2D texture to achieve the similar effect. With the implementation of Google VR, we eventually bring the virtual reality experience to the user which allows them to enjoy playing around with kaleidoscope anytime they would like to.

Components and Parameters

The main components and parameters can be summarized as below:

- **Manual / auto rotation mode:**

  For zoom out view, users are able to manually rotate the kaleidoscope by tapping on the phone screen and drag in clockwise or anti-clockwise direction to manually rotate kaleidoscope and see the changes on image patterns.
  For zoom in view, auto-rotation mode will be activated. At first, users will see a smooth transition from zoom out view to zoom in view. Then users will see the auto rotation that happens inside kaleidoscope which will cover up the entire screen.
  We let the users change this by moving a slider on the screen that will toggle between automatic and manual. In the VR mode, autorotation is enabled by default.

- **Objects inside the kaleidoscope:**

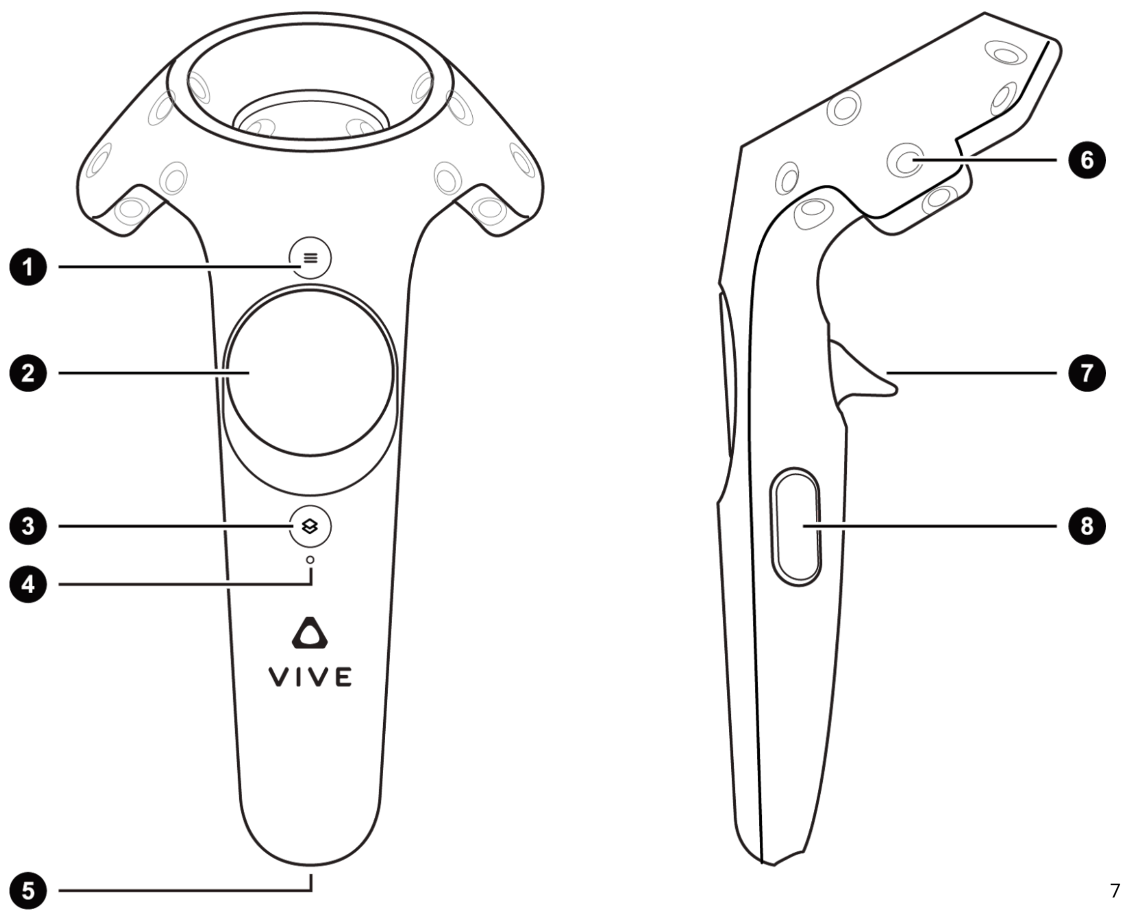
  We have considered a few aspects of the design on the objects in kaleidoscope which includes number, color and geometry of the objects. Objects in the scope can be changed along with the background by double-tapping the screen. There are different objects including some things that one cannot see in real life- such as a cookie cutter shape, amongst others.

- **Number and angle of mirror**
  The number and angle of mirror are changeable where the users are allowed to adjust according to their preferences. This can be changed using a slider present on the bottom right of the screen from 3 mirrors to 9 mirrors. The effects of changing mirrors are very evident.

5. **FUTURE** WORK

Our project simulates virtual reality experience through the use of Google Cardboard, and we feel that this immersive experience may be further improved with the addition of user interactions (e.g. manually controlling the viewing angle, objects, and rotation speed in virtual reality mode).

HTC Vive is one of the virtual reality engine that enables a wide range of interactions, as shown by its controllers below:



[7]

Vive Controller Hardware Feature (Both Left and Right)	Interaction Type
Controller Menu Button (1)	Press
Controller Trackpad (2)	Press
	Touch
	Horizontal Movement
	Vertical Movement
Controller Trigger (7)	Touch
	Squeeze

[7] Image courtesy of developer.viveport.com

Controller Grip Button (8)	Squeeze

There are several ways we propose to integrate HTC Vive interactions in the kaleidoscopic virtual reality world:

- Trigger input: controlling the rotating direction, angle, and speed
- Touchpad input: switching background scene
- Gestures: controlling zoom
- Haptic feedback: providing realistic feedback such as vibration when creating a new pattern or capturing a view

REFERENCES

❖ https://github.com/coldhead/kaleidos
❖ https://github.com/dbu/kaleidoscope
❖ https://github.com/prrt714/Kaleidoscope/
❖ https://developer.viveport.com/sg/develop_portal/
❖ https://docs.unity3d.com/Manual/OpenVRControllers.html
❖ https://itroadblog.wordpress.com/2016/09/21/5-ways-to-interact-with-htc-vive-hand-controls/