CS135 Winter 2022 - Lab 2

Lab 2 was a Collaboration between Diego Vega (dvega007) and Sumachai Suksanguan (ssuks001)

Parts and Explanation

All Assignments were completed and tested within Unity. Parts 1, 2, and 3 were all done and tested with the `OVRCameraRig`

Part 1 - VR Mirror

This part of the assignment was to create a script that allowed us to manipulate Minecraft Steve's head by way of mimicing our current head rotation and position.

CameraReset, CameraFlip, and Exit

We were asked to implement simple basic functions such as reset, flip, and exit. Doing so was easy, as it just required us to bind those commands to their respective buttons, as shown below:

```
if(Input.GetKeyDown(KeyCode.Tab)){
  transform.position = new Vector3(0, 0, 0);
}
if(Input.GetKeyDown(KeyCode.F)){
  transform.Rotate(0, 180.0f, 0);
}
if(Input.GetKeyDown(KeyCode.Esc)){
  #if UNITY_EDITOR
    UnityEditor.EditorApplication.isPlaying = false;
  #else
    Application.Quit;
  #endif
}
```

> *Note: These should be in their respective files, as `CameraReset.cs` contains the reset function, and `CameraFlip.cs` contains the flip and exit function.*

Mirror Functionality

Designing the mirror was also straight forward once you realize that you can directly reference the head object and have it's positional and rotational values "mimic" ours.

Start by initializing two flags, `match` and `mirror` to `false`, so that our program starts with the box not mimicing us at all.

When input `M` is received, determine whether to turn on matching or mirroring. When the program first starts, this will turn on matching. Throughout the program runtime though, pressing `M` will toggle between matching and mirroring.

It is important to note, that the only difference between matching and mirror logic, is the orientation of the face, and the direction it moves towards when the camera approaches it. Therefore there is an unconditional transformation in the `if` statements of both `match` and `mirror`:

```
if(matching){
  head.transform.rotation = transform.rotation;
  ...
}
if(mirror){
  head.transform.rotation = transform.rotation;
  ...
}
```

Matching Logic

When the head is matching our movements, it simulates the same movement as us moving **away** from the main camera. Therefore, we should see the **back** of the head, and it should move **away** from us as we move towards it.

```
if(matching){
  head.trasform.rotation = transform.rotation;
  head.transform.Rotate = (0, -90.0f, 0);
  head.transform.position = new Vector3(transform.localPosition.x,
transform.localPosition.y, (2 + transform.localPosition.z));
}
```

> *Rotation Note: Rotating -90 Degrees in the Y-Axis will turn the head away from us.*

> *Position Note: `transform.localPosition.x` and `transform.localPosition.y` will mimic our movement from the origin, however `transform.localPosition.z` must be shifted AWAY from Z = 2, as that is the head's Z-Origin*

Mirror Logic

When the head is mirroring our movements, it simulates the same movement as us moving **towards** from the main camera. Therefore, we should see the **front** of the head, and it should move **towards** from us as we move towards it.

```
if(mirror){
  head.trasform.rotation = transform.rotation;
  head.transform.Rotate = (0, 90.0f, 0);
  head.transform.position = new Vector3(transform.localPosition.x,
transform.localPosition.y, (2 - transform.localPosition.z));
}
```

> *Rotation Note: Rotating 90 Degrees in the Y-Axis will turn the head away toward us.*

> *Position Note: `transform.localPosition.x` and `transform.localPosition.y` will mimic our movement from the origin, however `transform.localPosition.z` must be shifted TOWARDS from Z = 2, as that is the head's Z-Origin*

Part 2 - Disabling Position and Rotation Tracking

This part of the assignment was to disable Position and Rotation Tracking. The solution was create by creating a script `ToggleTracking.cs` that would adjust the parent camera to "counter-act" the child cameras movements.

Users should be able to toggle turning off tracking for Position and Rotation by pressing `P` and `R` respectively. Therefore, to start, create and initialize your `rotationTracking` and `positionTracking` to `true`, as the program should always start with both Rotation and Position Tracking enabled.

Use the script below to toggle between the two via keyboard input:

```
if(Input.GetKeyDown(KeyCode.R)){
  rotationTracking = !rotationTracking;
}
if(Input.GetKeyDown(KeyCode.P)){
  positionTracking = !positionTracking;
}
```

Disabling Positional Tracking

To disable position tracking, add the same movement to the parent camera that the child camera, `CenterEyeAnchor` is being shifted by. The reason being for this, is that if the child is shifted 2 to the right, if we add that same movement to the parent, we end up shifting the entire camera setup by 2 to the right, which consequently moves the child camera setup 2 to the left, thus negating any camera movement.

When position tracking is enabled, the camera should just default to the origin, `{0, 0, 0}`.

```
if(!positionTracking){
  transform.position = (childCamera.transform.localPosition);
}
else{
  transform.position = new Vector3(0, 0, 0);
}
```

Disabling Rotational Tracking

Disabling Rotational Tracking is a bit more complex in terms of implementation. If the child is shift 10 degrees in the X-Direction, the parent should shift -10 degrees in the X-Direction to offset the angular movement. Using `localEulerAngles.axis` will give you the rotation difference in regards to the parent camera, so all we need to do is inverse it, and then re-assign that as the main camera angle.

When rotation tracking is enabled, the camera should just default to it's base rotation, `{0, -180.0f, 0}`.

```
if(!rotationTracking){
  float x_rotate = -(childCamera.transform.localEulerAngles.x);
  float y_rotate = -(childCamera.transform.localEulerAngles.y);
  float z_rotate = -(childCamera.transform.localEulerAngles.z);

  Vector3 currentAngles = new Vector3(x_rotate, y_rotate + 180, z_rotate);
  transform.eulerAngles = currentAngles;
}
else{
  transform.eulerAngles = new Vector3(0, 180.0f, 0);
}
```

> *Note: We must rotate the Y-Axis, from it's original rotation of -180.0f, thus the `y_rotate + 180`.*

Part 3 - Depth Perception and Relative Size

The final part of the assignment was creating a script that dynamically alters the size of the two bue spheres so that no matter what position and orientation the camera is in, the spheres "appear" to be the same depth as the red sphere.

Start by creating your `resizeToggle` and `appearToggle` flags, as these will control whether the spheres are appearing/disappearing, and whether they all appaer the same or not.

Initialize `appearToggle` to `true` as the scene will begin with all spheres present. Then initialize `resizeToggle` to `false` as the scene should begin with all spheres looking different depth-wise.

Then, create a `s_pressed` and `done` flag that will allow the program to determine whether `S` was pressed, while `done` determines if the program is making the spheres appear/disappear. Initilialize `s_pressed` to `false` and `done` to `true`.

Control Stimuli and Appearance

Users will be able to control whether the spheres appear and disappear by pressing the `S` key. The first time the user presses the `S` key, the `resizeToggle` flag should be set to true, and then at no point in the program is the `resizeToggle` is changed.

When `S` is pressed, the program should determine whether to turn `redSphere` on or off. We can do this with one line, as the bool `appearToggle` will let us know if the spheres are in the game or not. We should then also set `timer` to 0, so that we can start counting.

The system should also not do anything if `done` is `false`, as that means that the system is in the process of making the blue spheres appear/disappear.

```
if(Input.GetKeyDown(KeyCode.S)){
  resizeToggle = true;
  s_pressed = true;
  done = false;

  redSphere.SetActive(!appearToggle);
  timer = 0.0f;
}
```

> *Note: Since `appearToggle` determines whether the spheres are appearing or not, doing `redSphere.SetActive(!appearToggle)` will turn on the sphere if the sphers are inactive, and turn them off if they are active.*

The blue spheres are also scripted to turn on and off after 2 seconds have elapsed, and we can determine time using `Time.deltaTime`. Therefore, if the `s_pressed` flag is `true`, we should count time elapsed, and if time elapsed is 2 or more seconds, enable or disble the blue spheres.

```
if(s_pressed){
  timer += time.deltaTime;
  if(timer >= 2.0f){
    blueSphere1.SetActive(!appearToggle);
    blueSphere2.SetActive(!appearToggle);
    s_pressed = false;
    done = true;
  }
}
```

> *Note: Once we are done manipulating the blue spheres, we should set `s_pressed` to `false` so we don't enter the loop again. Likewise, we should also set `done` to `true` so that the system can begin accepting input again.*

Dynamically Alter Blue Sphere Depth

When `S` is pressed for the first time, the `resizeToggle` flag should be set to `true`, which will continually alter the blue spheres depth so that it appears to have the same depth as the red orb.

Note, that the appearance of depth can be perceived as: (`Object_Size` / `Object_Distance_From_You`). Therefore, if we want the objects to have similar depths, we want them to satisfy the equation:

(`Object1_size` / `Object1_Distance_From_You`) = (`Object2_size` / `Object2_Distance_From_You`)

So, since distance is constantly being updated by the camera, we should dynamically be altering blue spheres size, meaning we can rewrite the function as:

`Object1_size` = ((`Object2_size` * `Object1_Distance_From_You`) / `Object2_Distance_From_You`)

The only static object size we should be calculating is `redSphere`, as `blueSphere1` and `blueSphere2` should be dynamically calculated each. Thus, `red_ratio` can be given by `redSphere.transform.localScale.x`.

All the spheres distance from us are also dynamically altered by the camera movement, but their only purpose is to be used in calculations, thus we can get `sphere_distance` by `Vector3.Distance(camera.transform.position, sphere.transform.position)`.

Thus, the code should work as follows:

```
if(resizeToggle){
  red_dist = Vector3.Distance(camera.transform.position,
redSphere.transform.position);
  red_ratio = redSphere.transform.localScale.x;
  blue1_dist = Vector3.Distance(camera.transform.position,
blueSphere1.transform.position);
  blue2_dist = Vector3.Distance(camera.transform.position,
blueSphere2.transform.position);

  blue1_temp_ratio = (red_ratio * blue1_dist) / red_dist;
  blue2_temp_ratio = (red_ratio * blue2_dist) / red_dist;

  blueSphere1.transform.localScale = new Vector3(blue1_temp_ratio, blue1_temp_ratio,
blue1_temp_ratio);
  blueSphere2.transform.localScale = new Vector3(blue2_temp_ratio, blue2_temp_ratio,
blue2_temp_ratio);
}
```

Closing Notes

Please note that several factors go into depth perception besides scale and actual depth of the object. So while it isn't completely obvious that the objects appear to be the same size, they *look* similar enough.