HOLOLENS

Hololens Academy

- The HoloLens comprises of three processing units, the CPU + GPU + HPU.
- 2 Speakers - Present above each ear.
- Inertial Measurement Unit (IMU) - Consists of : accelerometer, gyroscope, magnetometer. The IMU is used to track movements.
- A laser camera and laser project are located at the front of the HoloLens, they help in mapping out the local environment.

The Unity platform will be used to develop applications for the HoloLens.

- https://developer.microsoft.com/en-us/windows/mixed-reality/academy
- https://unity3d.com/learn/tutorials

The Unity environment needs to be configured before beginning the development of the application. Basic Unity components are discussed below.

SETTING UP THE SCENE

Unity views the world with a camera, the camera corresponds to the HoloLens wearers point of view (i.e User’s head). The scene needs to be set before building the application. This involves taking the following steps.

- Edit -> Project Settings -> Player -> Other Settings -> Virtual Reality Supported -> Windows Mixed Reality
- Edit -> Project Settings -> Player -> Publishing Settings -> InternetClientServer
- Edit -> Project Settings -> Player -> Publishing Settings -> PrivateNetworkClientServer
- Edit -> Project Settings -> Player -> Publishing Settings -> Microphone (Voice Applications)
- Edit -> Project Settings -> Player -> Publishing Settings -> SpatialPerception (Spatial Mapping)

- Edit -> Project Settings -> Quality -> Windows Icon -> *Select* Fastest
- Edit -> Audio -> Spatializer Plugin -> *Enable* MS HRTF Spatializer (Spatial Sound)
- Inspector Window -> Layers -> User Layer 31 -> *Rename to* "SpatialMapping"

MAIN CAMERA

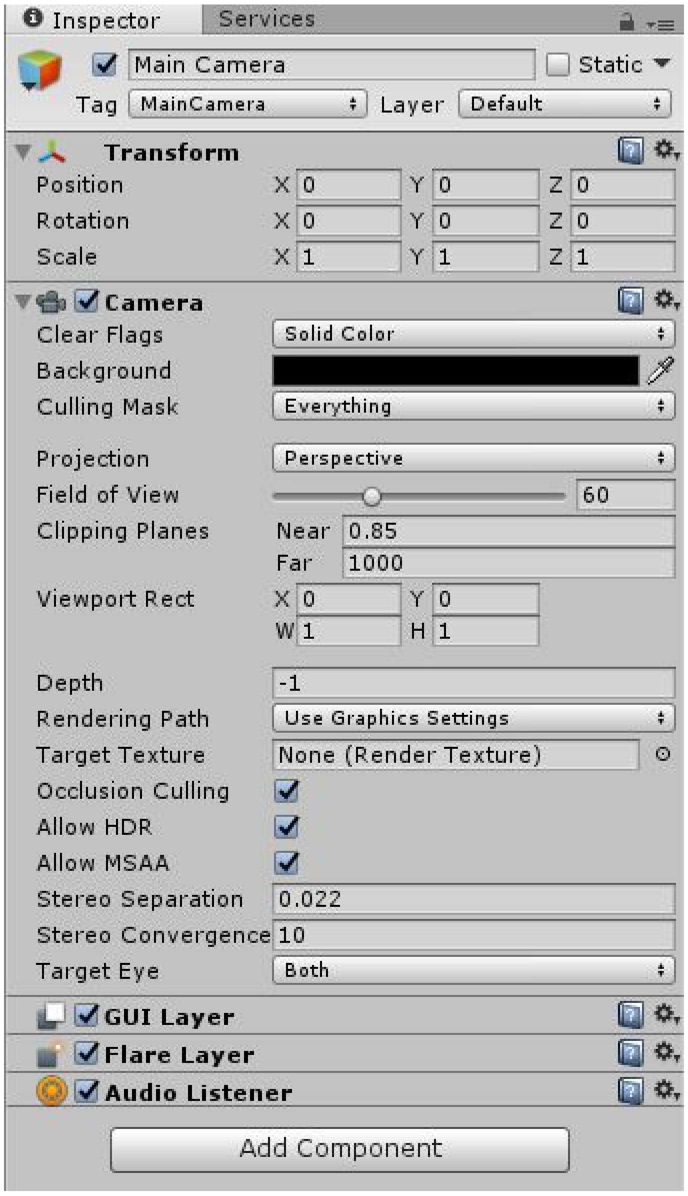
Setting up the Main Camera, the main camera tracks the user's head.

Position of the user's head = Position of the Main Camera.

Change coordinates to X:0, Y:0, Z:0. [X : Left/Right, Y : Up/Down, Z : Front/Back, with respect to the user. Example X : 0.3, Y : -0.5, Z : 2, 0.3 metres to the right of the user's head, 0.5 metres below the user's head, 2 metres in front of the user's head.]

[Left Hand Rule - Thumb = X, Index Finger = Y, Middle Finger = Z]

- Main Camera -> Inspector Panel -> Clear Flags -> *Change to* Solid Color
- Main Camera -> Inspector Panel -> *Select* Background Color -> *Update RGBA values to* (0, 0, 0, 0). [A = Opacity, 0 = Transparent, 1 = Opaque. 0 = Black Pixels. Everything rendered in black will be rendered transparent (0) when viewed in the device]
- Main Camera -> Inspector Panel -> Near Clip Plane -> *Change from* 0.3 to 0.85.



```
✓ Main Camera                         ☐ Static
Tag  MainCamera        Layer  Default

Transform
Position              X 0      Y 0      Z 0
Rotation              X 0      Y 0      Z 0
Scale                 X 1      Y 1      Z 1

✓ Camera
Clear Flags           Solid Color
Background            ■
Culling Mask          Everything

Projection            Perspective
Field of View         60
Clipping Planes       Near 0.85
                      Far  1000
Viewport Rect         X 0      Y 0
                      W 1      H 1

Depth                 -1
Rendering Path        Use Graphics Settings
Target Texture        None (Render Texture)
Occlusion Culling     ✓
Allow HDR             ✓
Allow MSAA            ✓
Stereo Separation     0.022
Stereo Convergence    10
Target Eye            Both

✓ GUI Layer
✓ Flare Layer
✓ Audio Listener
```

Figure 1 : Main Camera Component

CREATING GAME OBJECTS

Click on Create and select the Gameobject. Assign the desired values in the Inspector Panel.

Game Objects can be singular or nested (Parent - Child Relationship). Game objects can be enabled and disabled as per the application. Various components can be added to a gameobject, such as scripts, audiosource, rigidbody etc.

- Singular Game Objects : The values correspond to the world coordinate system.

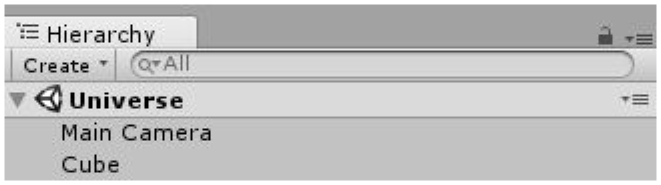


Figure 2 : Cube Game Object

- Nested Game Objects : The values correspond to the local coordinate system, that is with respect to the parent object.

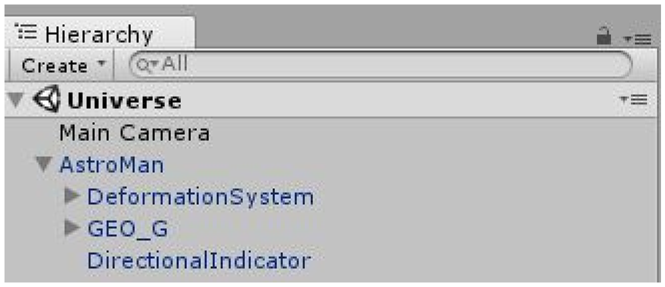
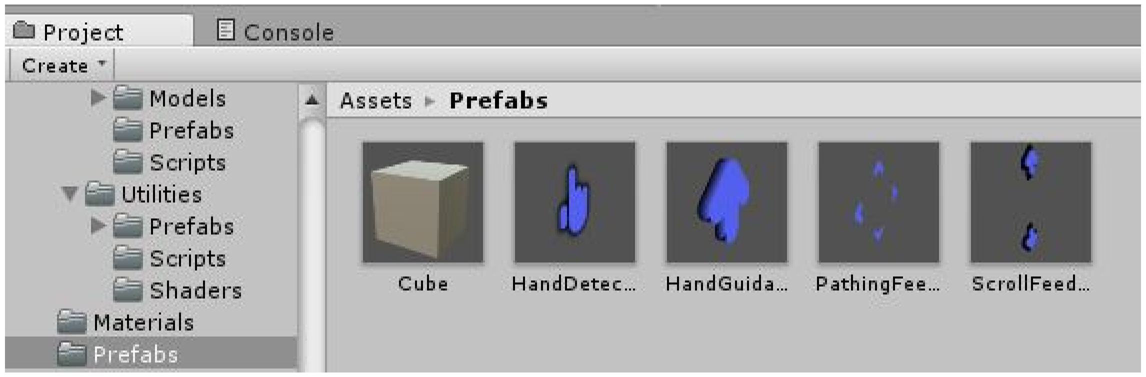


Figure 3 : Astroman (Parent Game Object)

DeformationSystem, GEO_G…(Child Game Objects)

PREFABS

An asset type that allows you to store a gameobject object complete with components and properties. The prefab acts as a template from which you can create new object instances in the scene.



Creating a prefab :

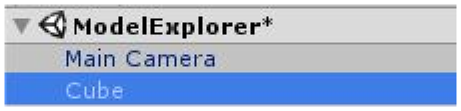
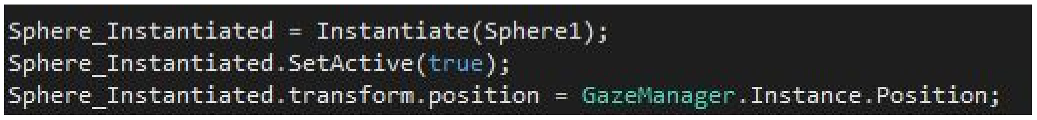


Figure 1 : Drag and drop cube

Drag and drop the cube gameobject into any of the folders in the Project panel, which is present in the assets folder. The cube becomes a prefab and you can use multiple instances of this prefab (drag and drop the prefab into the scene or instantiate).

If you want the properties of the prefab to remain the same, the prefab needs to be instantiated before adding it to the scene. This ensures that any change to the instantiated object does not affect the prefab.



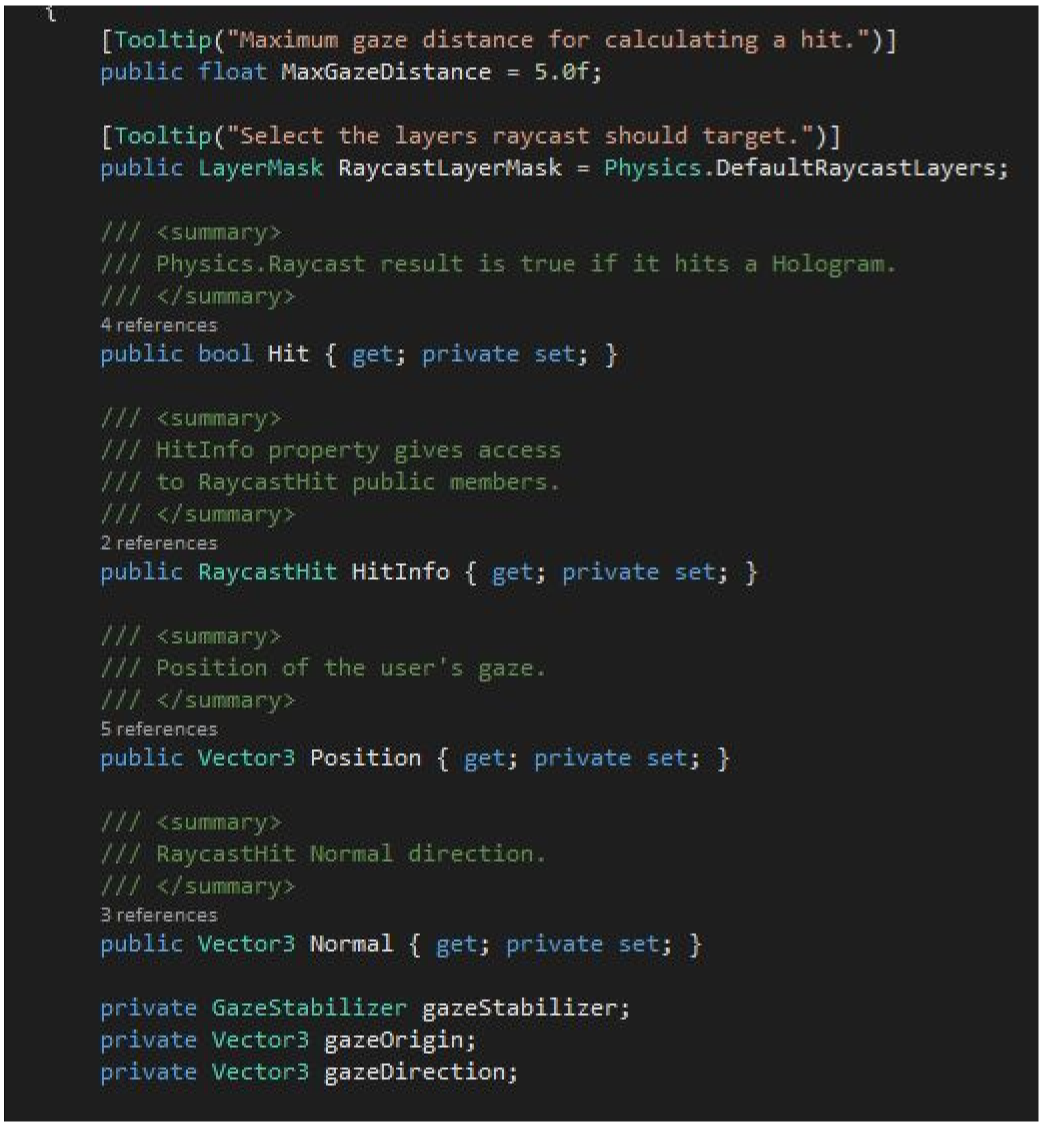
```
Sphere_Instantiated = Instantiate(Sphere1);
Sphere_Instantiated.SetActive(true);
Sphere_Instantiated.transform.position = GazeManager.Instance.Position;
```

Figure 1 : Instantiate a prefab

The default position and rotation of the instantiated object will be the same as that of the prefab. The transform properties can then be changed as required and it will not affect the prefab.

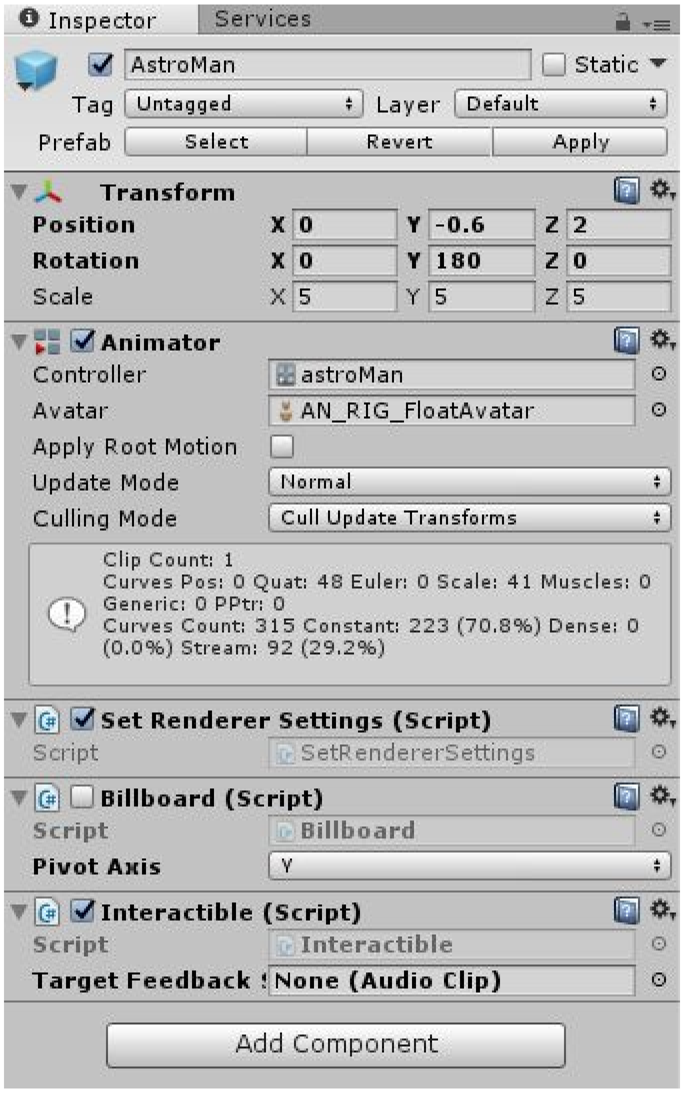
C# SCRIPTS

The scripts are used to provide additional functionality to the game objects and to the scene. They are added as components or can be added to the scene directly. Initialize the variables appropriately, example :



```
    [Tooltip("Maximum gaze distance for calculating a hit.")]
    public float MaxGazeDistance = 5.0f;

    [Tooltip("Select the layers raycast should target.")]
    public LayerMask RaycastLayerMask = Physics.DefaultRaycastLayers;

    /// <summary>
    /// Physics.Raycast result is true if it hits a Hologram.
    /// </summary>
    4 references
    public bool Hit { get; private set; }

    /// <summary>
    /// HitInfo property gives access
    /// to RaycastHit public members.
    /// </summary>
    2 references
    public RaycastHit HitInfo { get; private set; }

    /// <summary>
    /// Position of the user's gaze.
    /// </summary>
    5 references
    public Vector3 Position { get; private set; }

    /// <summary>
    /// RaycastHit Normal direction.
    /// </summary>
    3 references
    public Vector3 Normal { get; private set; }

    private GazeStabilizer gazeStabilizer;
    private Vector3 gazeOrigin;
    private Vector3 gazeDirection;
```

Figure 4 : Defining variables in a script



AstroMan ✓ Static ☐
Tag: Untagged
Layer: Default
Prefab: Select, Revert, Apply

Transform
Position X 0 Y -0.6 Z 2
Rotation X 0 Y 180 Z 0
Scale X 5 Y 5 Z 5

Animator ✓
Controller: astroMan
Avatar: AN_RIG_FloatAvatar
Apply Root Motion ☐
Update Mode: Normal
Culling Mode: Cull Update Transforms

```
Clip Count: 1
Curves Pos: 0 Quat: 48 Euler: 0 Scale: 41 Muscles: 0
Generic: 0 PPtr: 0
Curves Count: 315 Constant: 223 (70.8%) Dense: 0
(0.0%) Stream: 92 (29.2%)
```

Set Renderer Settings (Script) ✓
Script: SetRendererSettings

Billboard (Script) ☐
Script: Billboard
Pivot Axis: Y

Interactible (Script) ✓
Script: Interactible
Target Feedback : None (Audio Clip)

Figure 5 : Scripts attached to a gameobject

SetRendererSettings, Billboard, and Interactible are three scripts that are components of the astroman gameobject, here the billboard script is disabled, it can be enabled by checking the box or from a script.

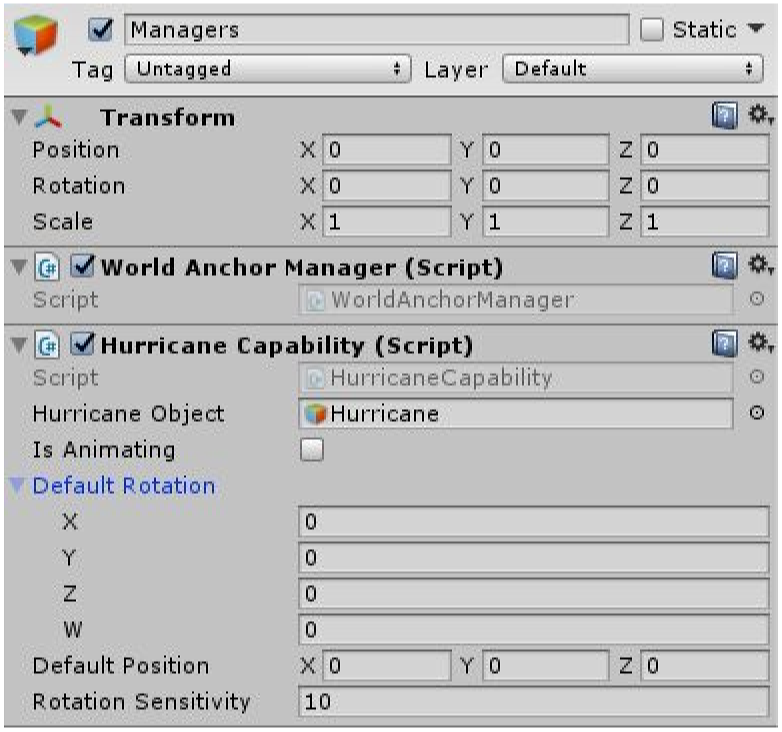
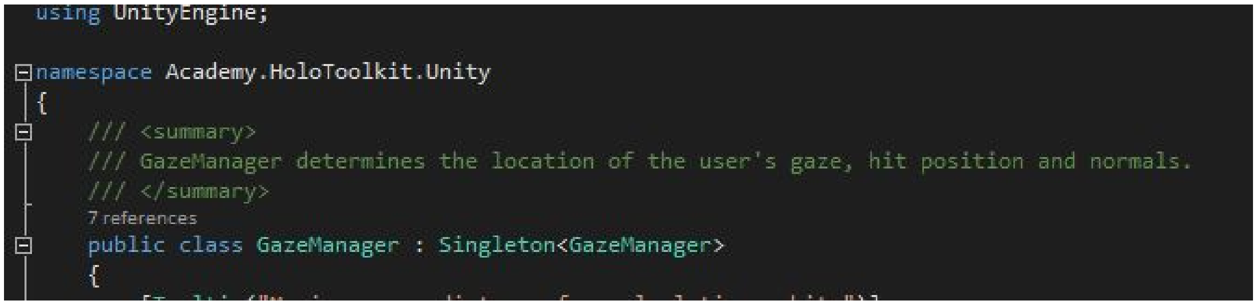

Managers
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
World Anchor Manager (Script)
Script WorldAnchorManager
Hurricane Capability (Script)
Script HurricaneCapability
Hurricane Object Hurricane
Is Animating
Default Rotation
X 0
Y 0
Z 0
W 0
Default Position X 0 Y 0 Z 0
Rotation Sensitivity 10

Figure 6 : Public variables in a script can be changed from the inspector window

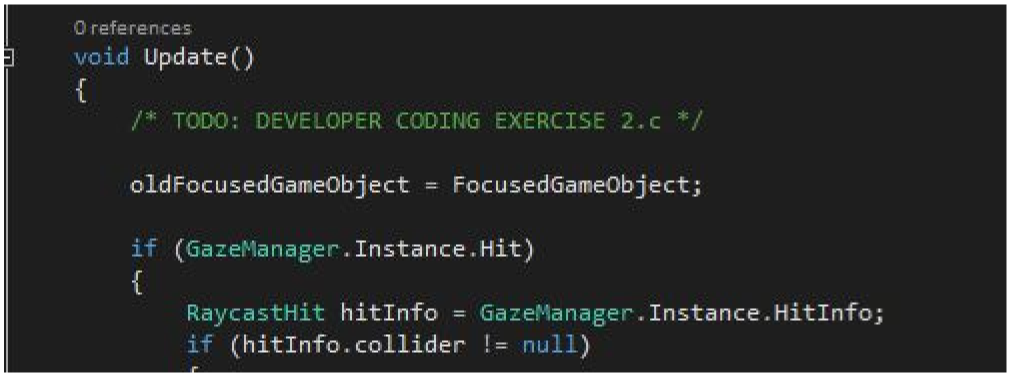
The two types of scripts are MonoBehaviour and Singleton.

- Singleton: There is only one Instance of the class. To access any variable from another script, ClassName.Instance.VariableName is used to get the value of that variable at that instance. Using the singleton class in another script, it gets the value of the variable of the singleton class at that instant.



```
using UnityEngine;

namespace Academy.HoloToolkit.Unity
{
    /// <summary>
    /// GazeManager determines the location of the user's gaze, hit position and normals.
    /// </summary>
    7 references
    public class GazeManager : Singleton<GazeManager>
    {
```

Figure 7 : Singleton *Script*



```
0 references
void Update()
{
    /* TODO: DEVELOPER CODING EXERCISE 2.c */

    oldFocusedGameObject = FocusedGameObject;

    if (GazeManager.Instance.Hit)
    {
        RaycastHit hitInfo = GazeManager.Instance.HitInfo;
        if (hitInfo.collider != null)
```

Figure 8 : Using Singleton (Instance) script in another class

- MonoBehaviour: The 3 main components are the Awake(), Start(), and Update() functions. The scripts runs in the following way

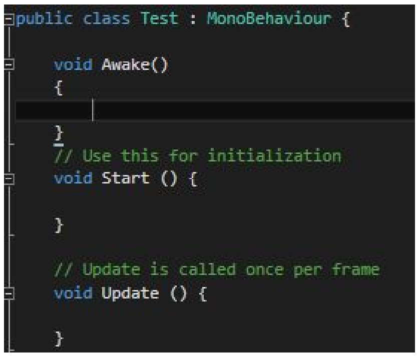
1. Awake() : This function is called first. The function is used for references between scripts, initialisation.
2. Start(): This function is called next. This function is used for assigning the initial values to the variables.
3. Update(): This function is called next, and continues to be called once per frame.

Awake() and Start() functions are called once in the lifetime of a gameobject.

In the inspector panel of the Game Object there is a checkbox that can be used to enable or disable the Game Object. When enabled all the 3 functions will be called, but when disabled the Awake() function will still be called.

Update() : Called once per frame. Used for regular updates such as: Moving non physics objects, Simple timer, Receiving input etc. Update time intervals may vary, if the time for processing frames are different.

FixedUpdate() : Called every physics step. Update intervals are consistent. Used for regular updates such as adjusting physics (Rigidbody) objects. Used for physics calculations.



```
public class Test : MonoBehaviour {

    void Awake()
    {

    }
    // Use this for initialization
    void Start () {

    }

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

    }
```

DEVELOPMENT

The main characteristics of a holographic application are:

- Gaze
- Gesture
- Voice
- Spatial Sound
- Spatial Mapping

Gaze *(Holograms 210)*

Gaze interactivity is used to determine what hologram the user wants to interact with. A visual feedback is needed to inform the user about the direction of his/her gaze, therefore a cursor object is used. The cursor denotes the gaze of the user, that is where the user is looking.

A Raycast is used to identify where the user is looking. A Raycast is the process of shooting an invisible ray from a point in a specified direction to detect whether any colliders lay in the path of the ray. Game Object will have a collider component, this component is necessary if we want to have gaze and gesture interaction with the object.

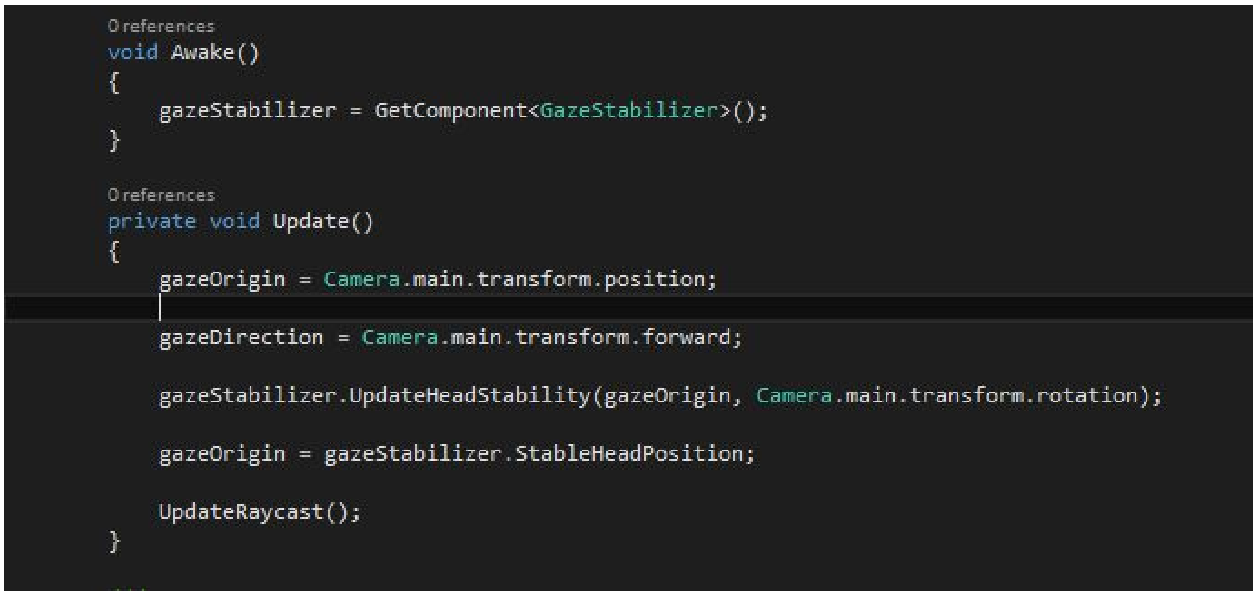
The raycast determines if the gaze strikes a hologram, if it does, update the position of the cursor onto the hologram.

Gaze depends on two components:

- var gazeOrigin = Camera.main.Transform.position [Gives the position of the user's head]
- Var gazeDirection = Camera.main.Transform.forward [A unit vector in the direction of the Z - axis, hence the direction in which the user is looking]

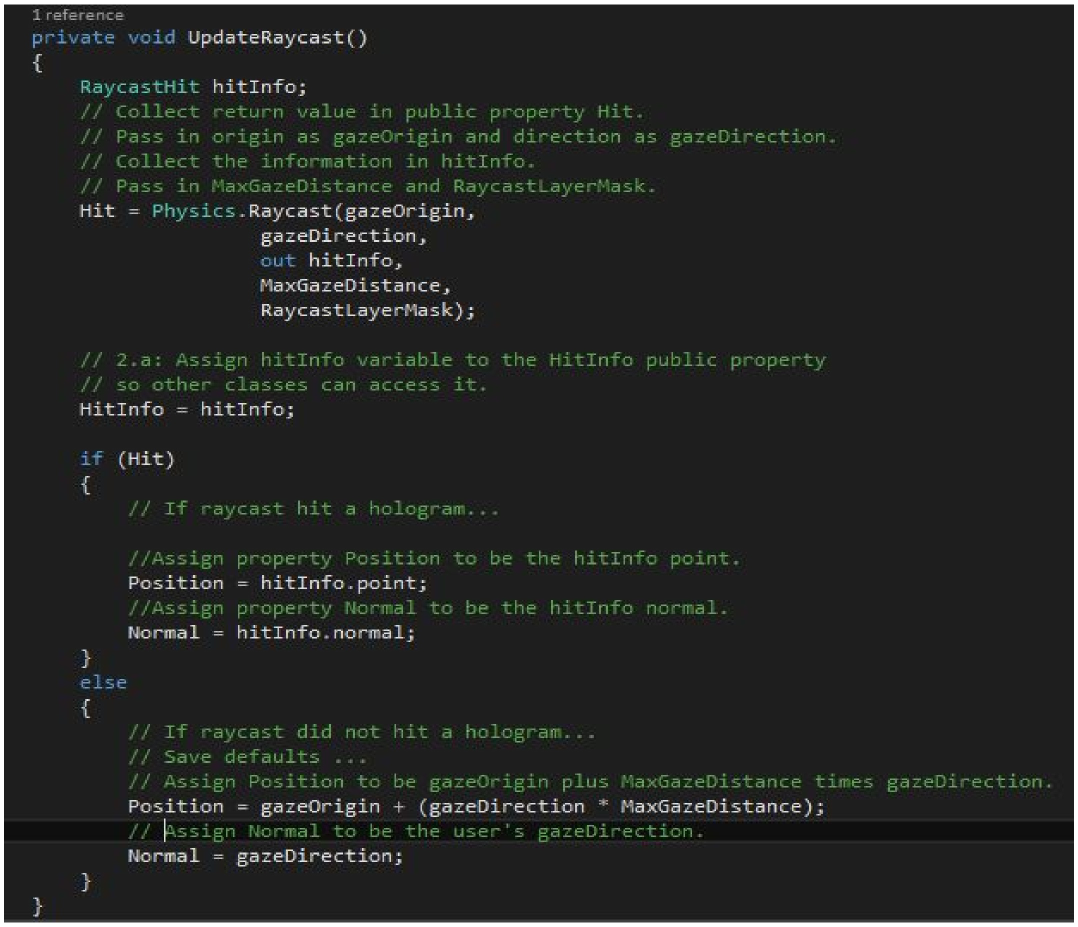
MaxGazeDistance is a value associated with the maximum distance a user's gaze will reach.

Physics.Raycast(gazeOrigin, gazeDirection, out hitInfo, MaxGazeDistance, RaycastLayerMask) - hitInfo will contain the point of interaction if the gaze struck a hologram, it will contain the position and the normal at which the gaze struck.



```
0 references
void Awake()
{
    gazeStabilizer = GetComponent<GazeStabilizer>();
}

0 references
private void Update()
{
    gazeOrigin = Camera.main.transform.position;

    gazeDirection = Camera.main.transform.forward;

    gazeStabilizer.UpdateHeadStability(gazeOrigin, Camera.main.transform.rotation);

    gazeOrigin = gazeStabilizer.StableHeadPosition;

    UpdateRaycast();
}
```

Figure 9 : Updates the Origin and Direction of the Gaze



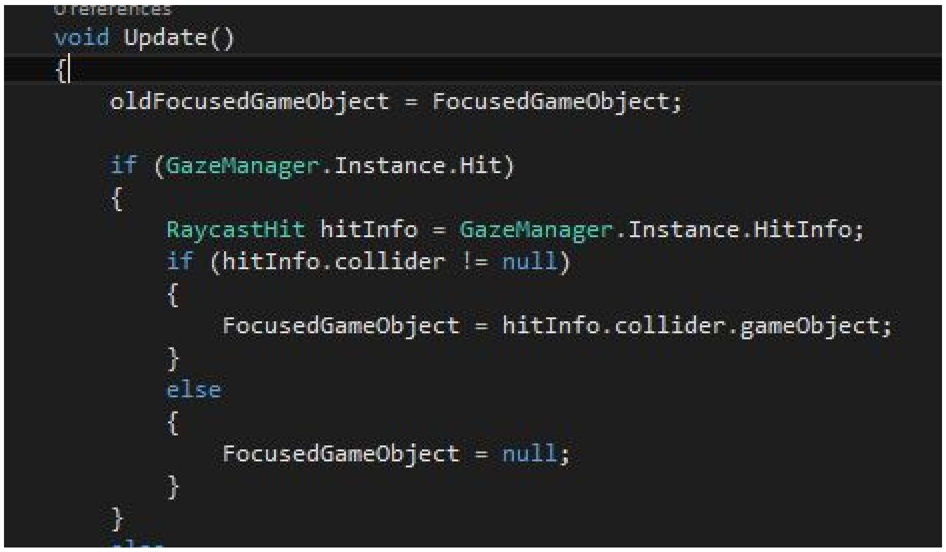
```
1 reference
private void UpdateRaycast()
{
    RaycastHit hitInfo;
    // Collect return value in public property Hit.
    // Pass in origin as gazeOrigin and direction as gazeDirection.
    // Collect the information in hitInfo.
    // Pass in MaxGazeDistance and RaycastLayerMask.
    Hit = Physics.Raycast(gazeOrigin,
                gazeDirection,
                out hitInfo,
                MaxGazeDistance,
                RaycastLayerMask);

    // 2.a: Assign hitInfo variable to the HitInfo public property
    // so other classes can access it.
    HitInfo = hitInfo;

    if (Hit)
    {
        // If raycast hit a hologram...

        //Assign property Position to be the hitInfo point.
        Position = hitInfo.point;
        //Assign property Normal to be the hitInfo normal.
        Normal = hitInfo.normal;
    }
    else
    {
        // If raycast did not hit a hologram...
        // Save defaults ...
        // Assign Position to be gazeOrigin plus MaxGazeDistance times gazeDirection.
        Position = gazeOrigin + (gazeDirection * MaxGazeDistance);
        // Assign Normal to be the user's gazeDirection.
        Normal = gazeDirection;
    }
}
```

Figure 10 : Calculating where the user’s gaze is at

Gaze Stabilizer is a built-in script that stabilises the head movements of the user.

- Hit is a boolean variable, will be true if the Raycast hits a hologram
- HitInfo is a public variable and can be accessed from other classes, while hitInfo is private.

We use another script to store our focused gameobject and perform operations on it.



```
0 references
void Update()
{
    oldFocusedGameObject = FocusedGameObject;

    if (GazeManager.Instance.Hit)
    {
        RaycastHit hitInfo = GazeManager.Instance.HitInfo;
        if (hitInfo.collider != null)
        {
            FocusedGameObject = hitInfo.collider.gameObject;
        }
        else
        {
            FocusedGameObject = null;
        }
    }
```

Figure 11 : Storing the gameobject that the user is gazing at

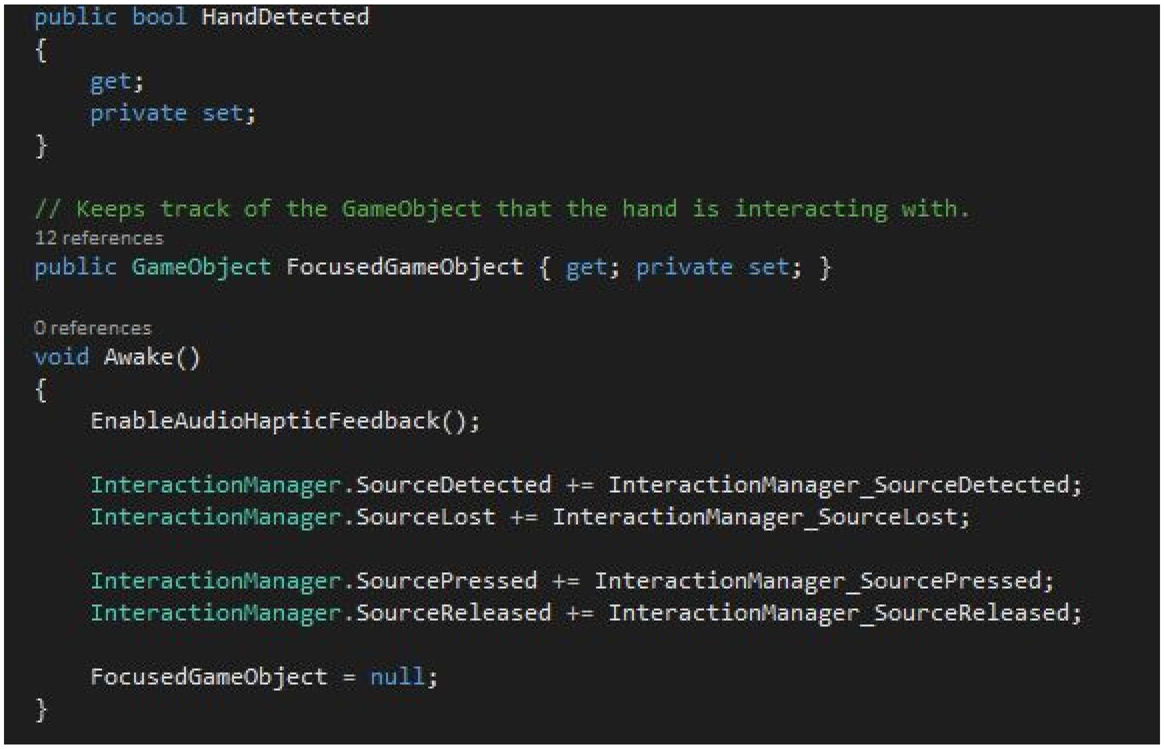
Gestures ***(Holograms 211)***

The HoloLens recognizes hand gestures by tracking the position of either or both hands that are visible to the device. The HoloLens recognizes our hands when they are in either the ready state (back of the hand facing you with index finger up) or the pressed state (back of the hand facing you with the index finger down).

Before detecting gestures, it is useful to know if the hand is detected and in view. This is done by setting up an InteractionManager event

The events are :

- InteractionManager.SourceDetected
- InteractionManager.SourceLost
- InteractionManager.SourcePressed
- InteractionManager.SourceReleased



```
public bool HandDetected
{
    get;
    private set;
}

// Keeps track of the GameObject that the hand is interacting with.
12 references
public GameObject FocusedGameObject { get; private set; }

0 references
void Awake()
{
    EnableAudioHapticFeedback();

    InteractionManager.SourceDetected += InteractionManager_SourceDetected;
    InteractionManager.SourceLost += InteractionManager_SourceLost;

    InteractionManager.SourcePressed += InteractionManager_SourcePressed;
    InteractionManager.SourceReleased += InteractionManager_SourceReleased;

    FocusedGameObject = null;
}
```

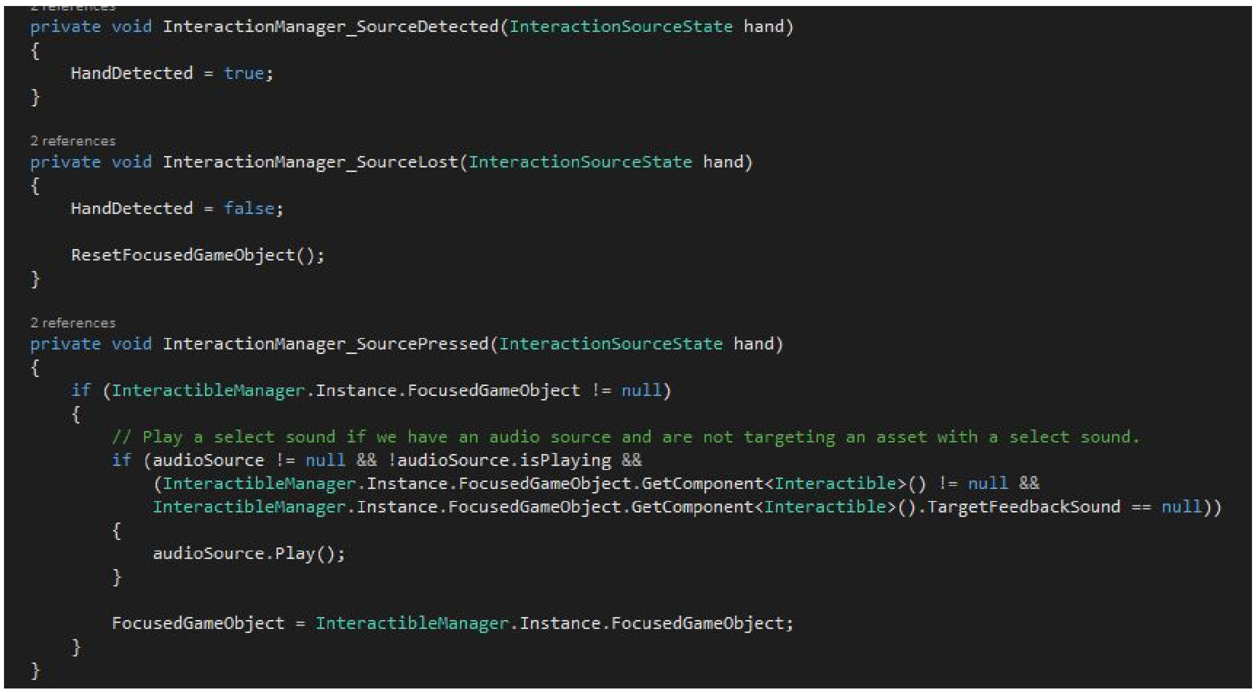
Figure 12 : Initialising the events

- When a hand is detected by the hololens the first event is triggered, thereby calling the InteractionManager_SourceDetected() function
- When a hand goes out of view the second event is triggered
- When the hand is in the pressed state the third event is triggered
- When the hand is released from the pressed state the fourth event is triggered

Example:

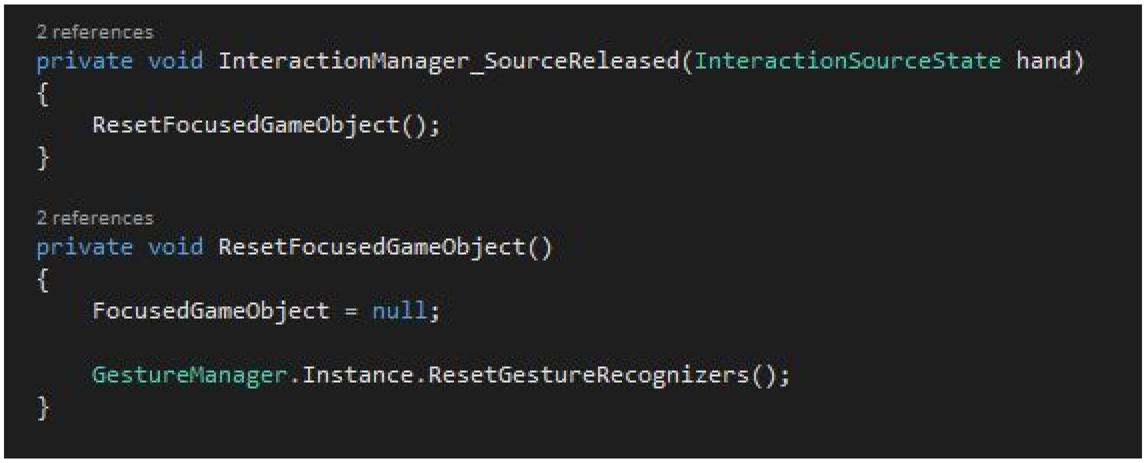
When a hand comes in view, the InteractionManager.SourceDetected event is triggered, so the InteractionManager_SourceDetected() function is called, setting HandDetected = true. If the hand moves into the pressed state the InteractionManager_SourcePressed event is triggered, and the FocusedGamebject variable is assigned, to perform actions on. In the pressed state if the position of the hand changes the InteractionManager.SourceUpdated event is triggered which notes the change in position. When the hand is released the InteractionManager.SourceReleased event is triggered, we reset the FocusedGamebject and start recapturing the gestures. When the hand goes out of view the InteractionManager.SourceLost event is triggered and the FocsedGameObject is reset, and then set HandDetected = false and recapture gestures.

Note : Whenever the hand is lost or released, the gesture recognizer needs to be reset, which means the gestures need to be recaptured.



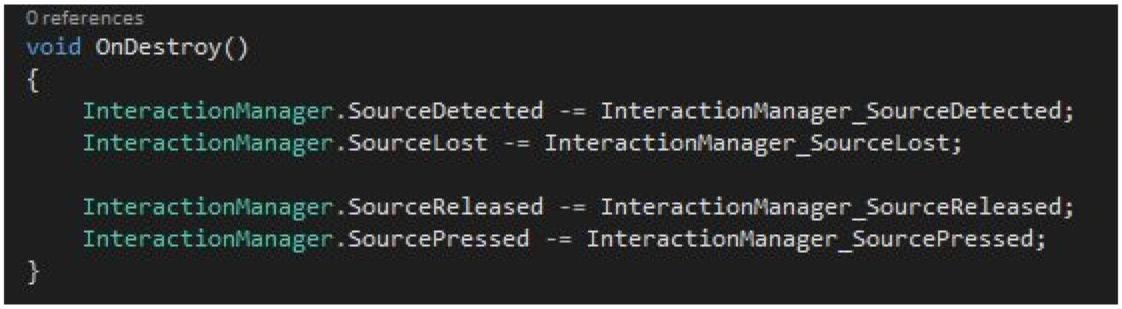
```
private void InteractionManager_SourceDetected(InteractionSourceState hand)
{
    HandDetected = true;
}

2 references
private void InteractionManager_SourceLost(InteractionSourceState hand)
{
    HandDetected = false;

    ResetFocusedGameObject();
}

2 references
private void InteractionManager_SourcePressed(InteractionSourceState hand)
{
    if (InteractibleManager.Instance.FocusedGameObject != null)
    {
        // Play a select sound if we have an audio source and are not targeting an asset with a select sound.
        if (audioSource != null && !audioSource.isPlaying &&
            (InteractibleManager.Instance.FocusedGameObject.GetComponent<Interactible>() != null &&
            InteractibleManager.Instance.FocusedGameObject.GetComponent<Interactible>().TargetFeedbackSound == null))
        {
            audioSource.Play();
        }

        FocusedGameObject = InteractibleManager.Instance.FocusedGameObject;
    }
}
```

Figure 13 : Performing actions with respect to the events



```
2 references
private void InteractionManager_SourceReleased(InteractionSourceState hand)
{
    ResetFocusedGameObject();
}

2 references
private void ResetFocusedGameObject()
{
    FocusedGameObject = null;

    GestureManager.Instance.ResetGestureRecognizers();
}
```

Figure 14 : Resetting and recapturing gestures



```
0 references
void OnDestroy()
{
    InteractionManager.SourceDetected -= InteractionManager_SourceDetected;
    InteractionManager.SourceLost -= InteractionManager_SourceLost;

    InteractionManager.SourceReleased -= InteractionManager_SourceReleased;
    InteractionManager.SourcePressed -= InteractionManager_SourcePressed;
}
```

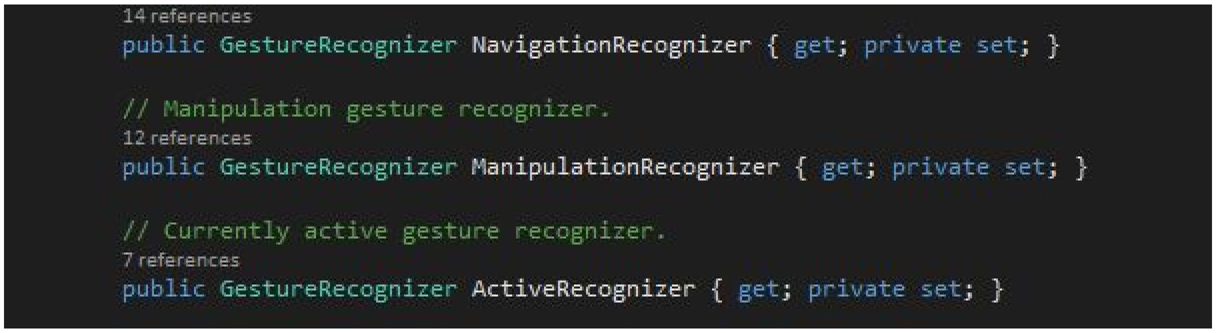
Figure 15 : Events are no longer needed

This function will be called when we no longer want the hand gestures to be detected, this function destroys the events and hence the events will not respond to the hand movements.

There types of hand events are :

- Tap: A Select press and release.
- Hold: Holding a Select press beyond the system's Hold threshold.
- Manipulation: A Select press, followed by absolute movement of your hand through 3-dimensional world.
- Navigation: A Select press, followed by relative movement of your hand or the controller within a 3-dimensional unit cube, potentially on axis-aligned rails.

More on this below.

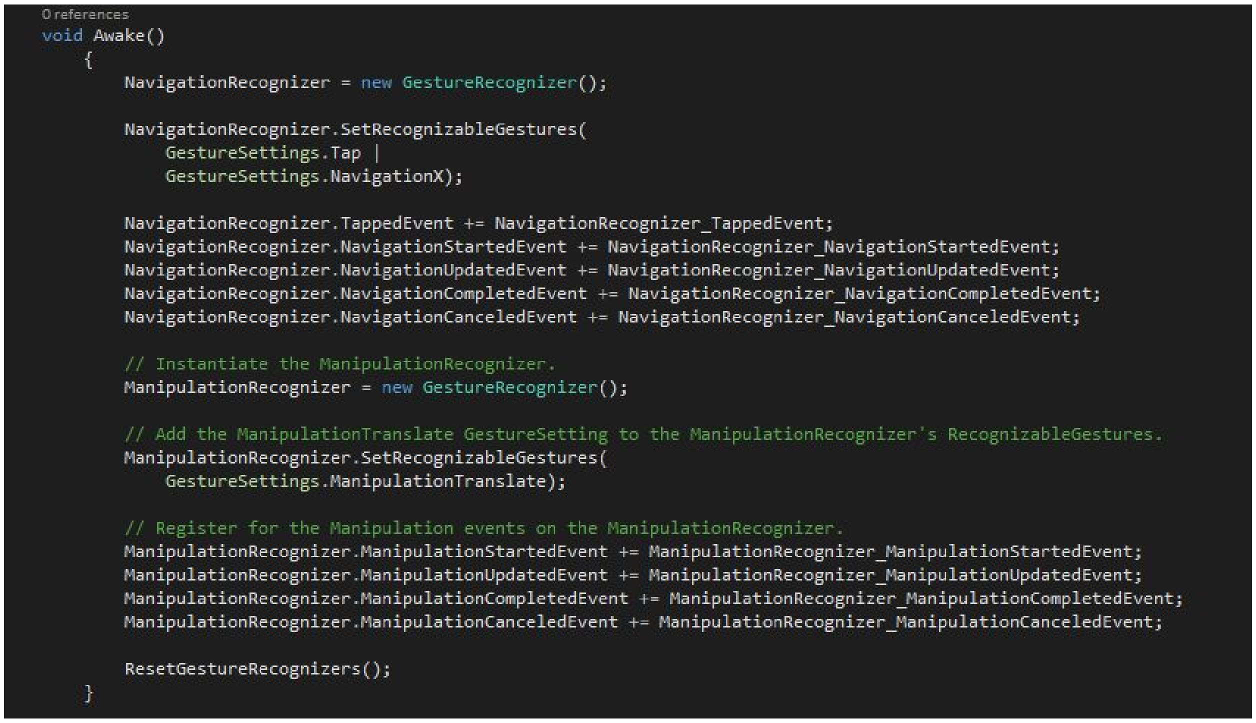


```
    14 references
    public GestureRecognizer NavigationRecognizer { get; private set; }

    // Manipulation gesture recognizer.
    12 references
    public GestureRecognizer ManipulationRecognizer { get; private set; }

    // Currently active gesture recognizer.
    7 references
    public GestureRecognizer ActiveRecognizer { get; private set; }
```

Figure 16 : Setting up recognizers for the events

To make use of gestures we initialize a gesture recognizer in the Awake() or Start() function.

The navigation and manipulation events work in the same way as before (detecting the hand). The default recognizer is the navigation recognizer and therefore only those events will be functioning until a switch (discussed later) is made to the manipulation recognizer, then the navigation recognizer stops and the manipulation recognizer events are captured. The tapped event (NavigationRecognizer.TappedEvent) is triggered on performing the airtap gesture.



```
0 references
void Awake()
    {
        NavigationRecognizer = new GestureRecognizer();

        NavigationRecognizer.SetRecognizableGestures(
            GestureSettings.Tap |
            GestureSettings.NavigationX);

        NavigationRecognizer.TappedEvent += NavigationRecognizer_TappedEvent;
        NavigationRecognizer.NavigationStartedEvent += NavigationRecognizer_NavigationStartedEvent;
        NavigationRecognizer.NavigationUpdatedEvent += NavigationRecognizer_NavigationUpdatedEvent;
        NavigationRecognizer.NavigationCompletedEvent += NavigationRecognizer_NavigationCompletedEvent;
        NavigationRecognizer.NavigationCanceledEvent += NavigationRecognizer_NavigationCanceledEvent;

        // Instantiate the ManipulationRecognizer.
        ManipulationRecognizer = new GestureRecognizer();

        // Add the ManipulationTranslate GestureSetting to the ManipulationRecognizer's RecognizableGestures.
        ManipulationRecognizer.SetRecognizableGestures(
            GestureSettings.ManipulationTranslate);

        // Register for the Manipulation events on the ManipulationRecognizer.
        ManipulationRecognizer.ManipulationStartedEvent += ManipulationRecognizer_ManipulationStartedEvent;
        ManipulationRecognizer.ManipulationUpdatedEvent += ManipulationRecognizer_ManipulationUpdatedEvent;
        ManipulationRecognizer.ManipulationCompletedEvent += ManipulationRecognizer_ManipulationCompletedEvent;
        ManipulationRecognizer.ManipulationCanceledEvent += ManipulationRecognizer_ManipulationCanceledEvent;

        ResetGestureRecognizers();
    }
```

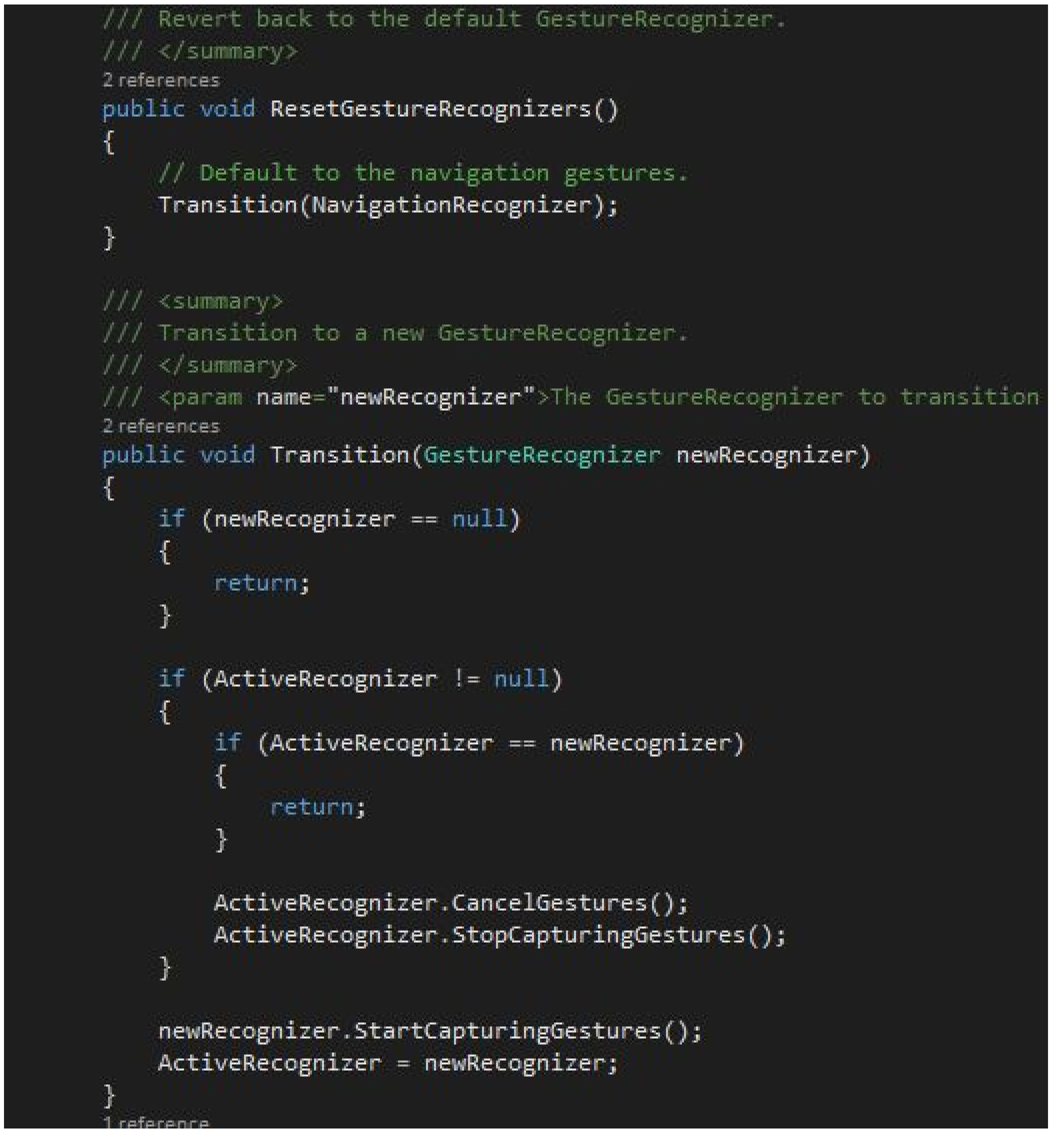
Figure 17 : Initialise the recognizers and capture their events

In both the Navigation and Manipulation events :

- Started Event : Triggered when the hand moves into the pressed state.
- Updated Event : When the hand is in the pressed state and the hand changes its position from the initial position.
- Completed Event : When the hand moves from the pressed state to the released state.

GestireSettings.NavigationX : The hand movements along the X-direction are recognized and updated.

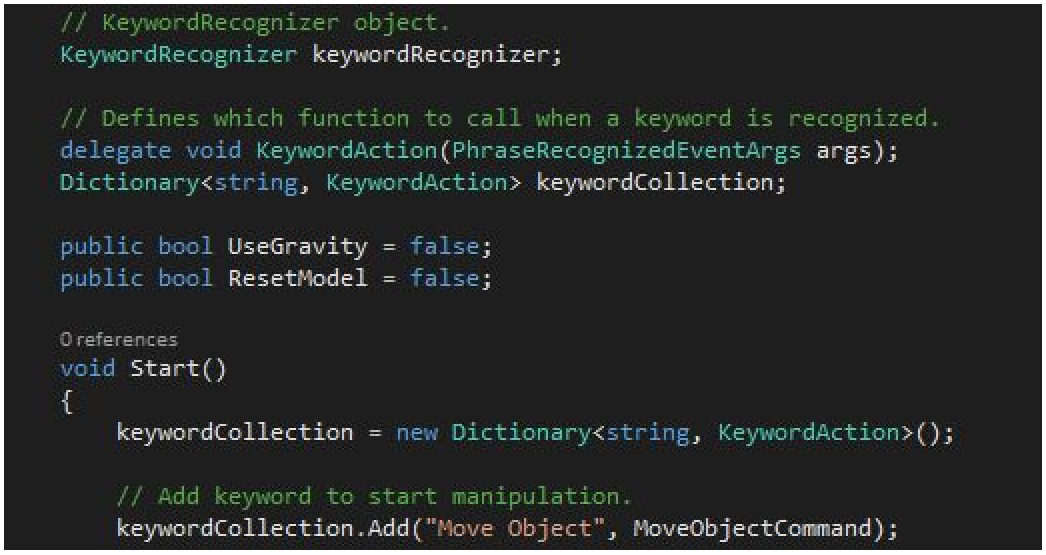
When the manipulation recognizer events finish running, the ResetGestureRecognizers() function is called, which transitions the gesture recognizer back to the default gesture recognizer, the navigation recognizer.



```
    /// Revert back to the default GestureRecognizer.
    /// </summary>
    2 references
    public void ResetGestureRecognizers()
    {
        // Default to the navigation gestures.
        Transition(NavigationRecognizer);
    }

    /// <summary>
    /// Transition to a new GestureRecognizer.
    /// </summary>
    /// <param name="newRecognizer">The GestureRecognizer to transition
    2 references
    public void Transition(GestureRecognizer newRecognizer)
    {
        if (newRecognizer == null)
        {
            return;
        }

        if (ActiveRecognizer != null)
        {
            if (ActiveRecognizer == newRecognizer)
            {
                return;
            }

            ActiveRecognizer.CancelGestures();
            ActiveRecognizer.StopCapturingGestures();
        }

        newRecognizer.StartCapturingGestures();
        ActiveRecognizer = newRecognizer;
    }
    1 reference
```

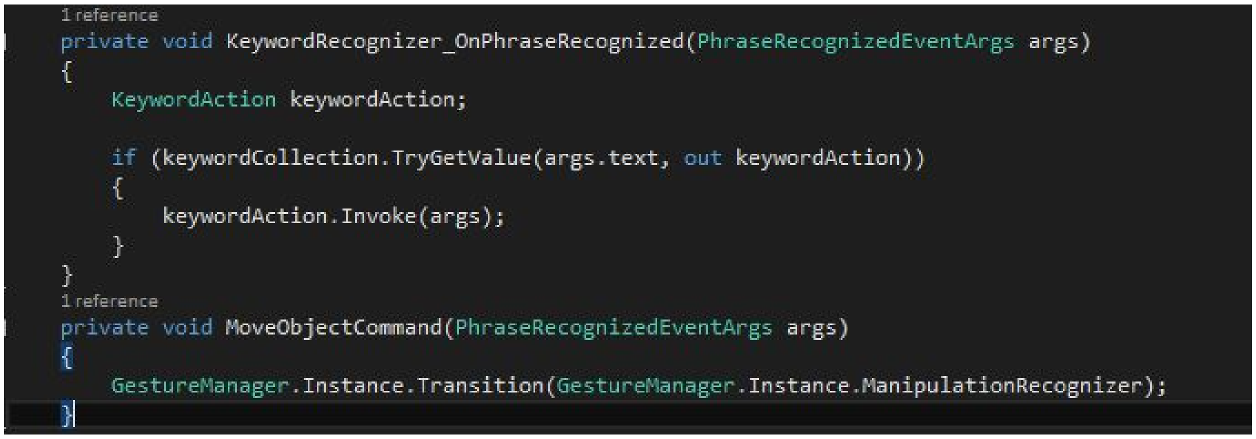
Figure 18 : Resetting the gesture recognizer

We can have only one active recognizer at an instance, therefore we switch between recognizer using a command, generally a voice command. In the above piece of code, the navigation recognizer is the default recognizer, to switch to the manipulation recognizer we use a voice command (“Move Object”).



```
    // KeywordRecognizer object.
    KeywordRecognizer keywordRecognizer;

    // Defines which function to call when a keyword is recognized.
    delegate void KeywordAction(PhraseRecognizedEventArgs args);
    Dictionary<string, KeywordAction> keywordCollection;

    public bool UseGravity = false;
    public bool ResetModel = false;

    0 references
    void Start()
    {
        keywordCollection = new Dictionary<string, KeywordAction>();

        // Add keyword to start manipulation.
        keywordCollection.Add("Move Object", MoveObjectCommand);
```

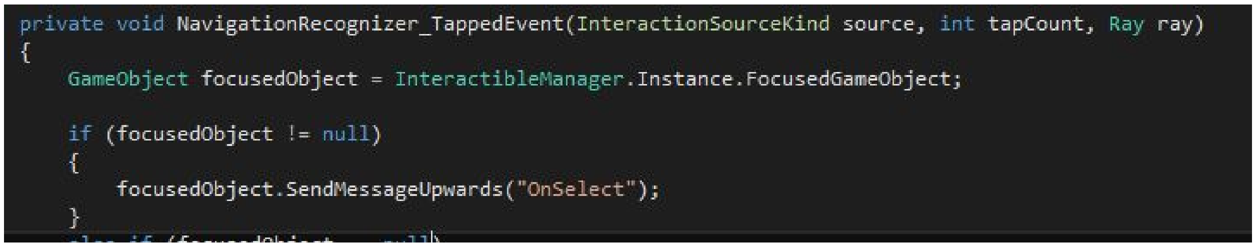
Figure 19 : Keyword Recognizer, that recognizes user's speech



```
    1 reference
    private void KeywordRecognizer_OnPhraseRecognized(PhraseRecognizedEventArgs args)
    {
        KeywordAction keywordAction;

        if (keywordCollection.TryGetValue(args.text, out keywordAction))
        {
            keywordAction.Invoke(args);
        }
    }
    1 reference
    private void MoveObjectCommand(PhraseRecognizedEventArgs args)
    {
        GestureManager.Instance.Transition(GestureManager.Instance.ManipulationRecognizer);
    }
```

Figure 20 : Switch from Navigation to Manipulation

The Transition() function transitions the recognizers from on recognizer to the other (Navigation -> Manipulation/Manipulation -> Navigation) *[Figure 18]*

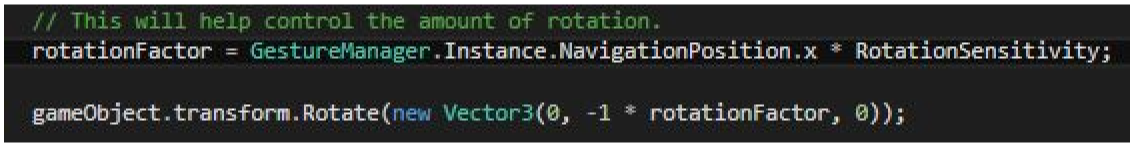


```
private void NavigationRecognizer_TappedEvent(InteractionSourceKind source, int tapCount, Ray ray)
{
    GameObject focusedObject = InteractibleManager.Instance.FocusedGameObject;

    if (focusedObject != null)
    {
        focusedObject.SendMessageUpwards("OnSelect");
    }
```

Figure 21 : Recognising the Air-Tap gesture

Using the navigation and manipulation gestures to move an object :

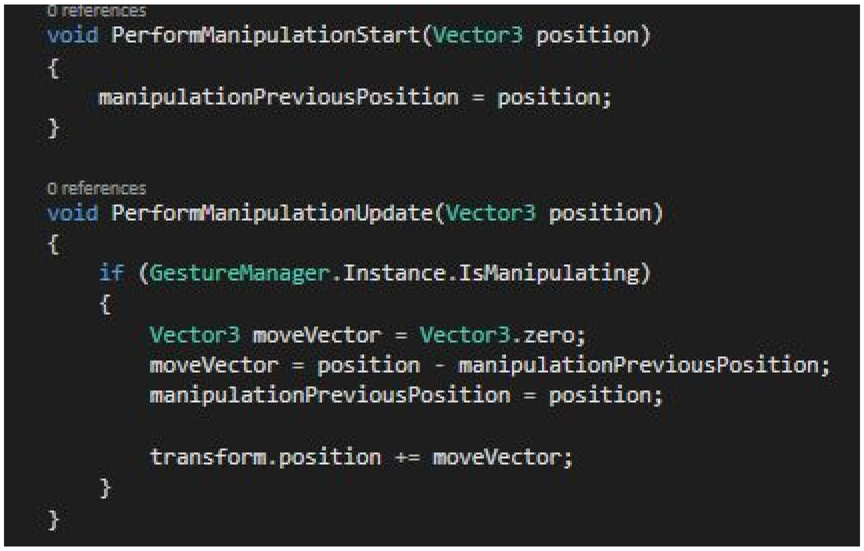
Navigation : The .x variable is the amount the hand moves in the x direction(-1,1)



```
// This will help control the amount of rotation.
rotationFactor = GestureManager.Instance.NavigationPosition.x * RotationSensitivity;

gameObject.transform.Rotate(new Vector3(0, -1 * rotationFactor, 0));
```

Figure 22 : Rotate the gameobject

Manipulation : Move the gameobject using hand movements.



```
0 references
void PerformManipulationStart(Vector3 position)
{
    manipulationPreviousPosition = position;
}

0 references
void PerformManipulationUpdate(Vector3 position)
{
    if (GestureManager.Instance.IsManipulating)
    {
        Vector3 moveVector = Vector3.zero;
        moveVector = position - manipulationPreviousPosition;
        manipulationPreviousPosition = position;

        transform.position += moveVector;
    }
}
```

Figure 23 : Move the gameobject

The position variable is with respect to the world coordinate system and not the hand, so in PerformManipulationStart we have the original position of the hand and finally we get the relative position by subtracting the final position.

Voice *(Holograms 212)*

Two types of voice commands :

- Global Voice Commands : Will trigger irrespective of what the user is looking at.
- Gaze Aware Voice Commands : WIll target the the hand gestures, based on the user's gaze, it will trigger events with respect to what the user is looking at.

Two important points to note :

- Make the user aware of what voice commands are available.
- Acknowledge that we have heard the user's voice command.

Types of recognizers :

- Dictation Recognizer : Understand what the user is saying. The hololens must be connected to the WiFi for the dictation recognizer to work.
- Grammar recognizer : Natural Language, SRGS - Speech Recognition Grammar Specification (Deriving the (semantics) meaning of what the user is saying).
- Keyword Recognizer : Will recognize the keywords and take appropriate action as specified in the script or inspector window.

Dictation Recognizer converts the user's speech to text, and will also show the hypothesis and the final result in the communicator. The hypothesis is what the recognizer thinks it has heard so far and is relayed back to the user. The final result is displayed when the user pauses, such as the end of a sentence (it works like how Siri or Google now would recognize voice commands). Microphone manager contains code for controlling microphone and dictation recognizer

At a time only the dictation recognizer or the keyword recognizer can run. The RestartSpeechSystem helps start our keyword recognizer again, it turns off the dictation recognizer and switches to the keyword recognizer.

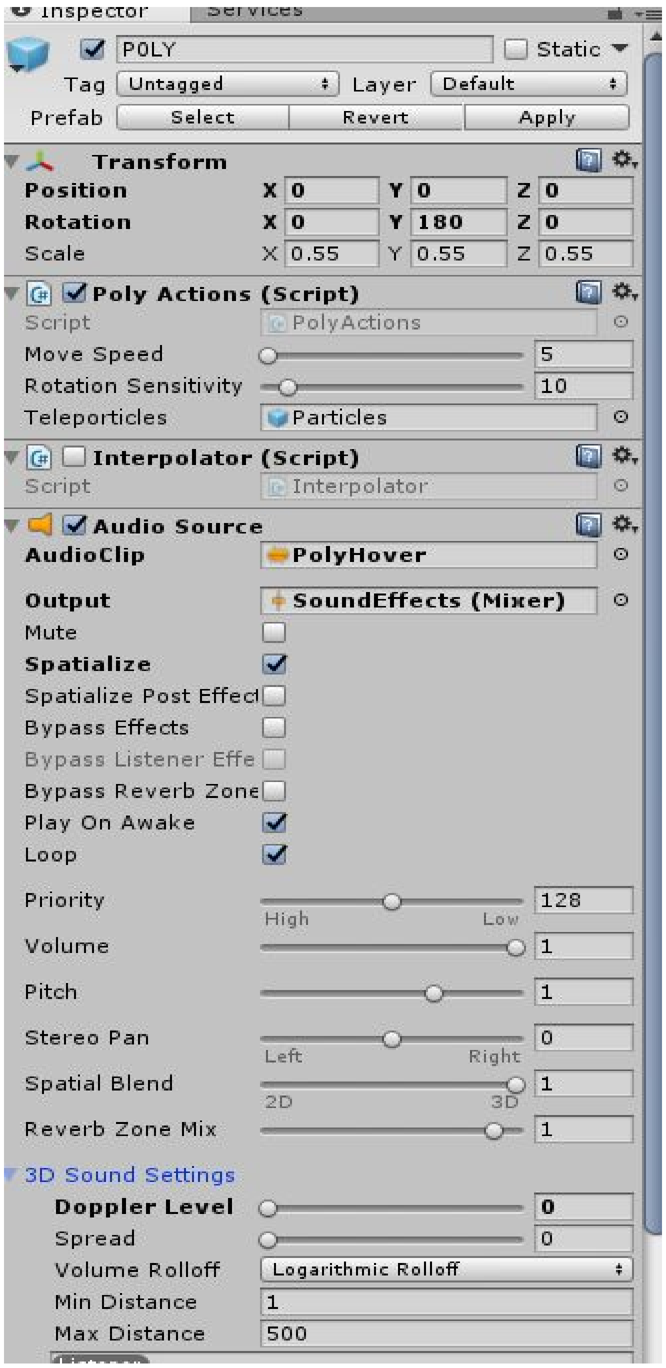
To understand grammar we need an SRGS file and this file needs to be in the streamingassets folder, it gives us the semantic meaning. SRGS is what Cortana works on to understand the meaning behind what the user has said.

Spatial Sound *(Holograms 220)*

Points to keep in mind when using spatial sound in your application:

- It enhances hologram realism.
- Do not overwhelm the user with overly loud sounds, subtle sounds work best.
- You can direct the user's gaze using sound. We can use sound to call attention to important holograms.
- It is important to place the sound source at an appropriate location(example - for a human at the mouth and not at the feet).

When we attach an audio source component and want to spatialize the sound to achieve the 3D effect (spatial sound), the Spatialize box needs to be checked in the inspector panel, for the object associated with the spatial sound, in the audio source component. Along with this the Spatial Blend value needs to be updated to 3-D/1.

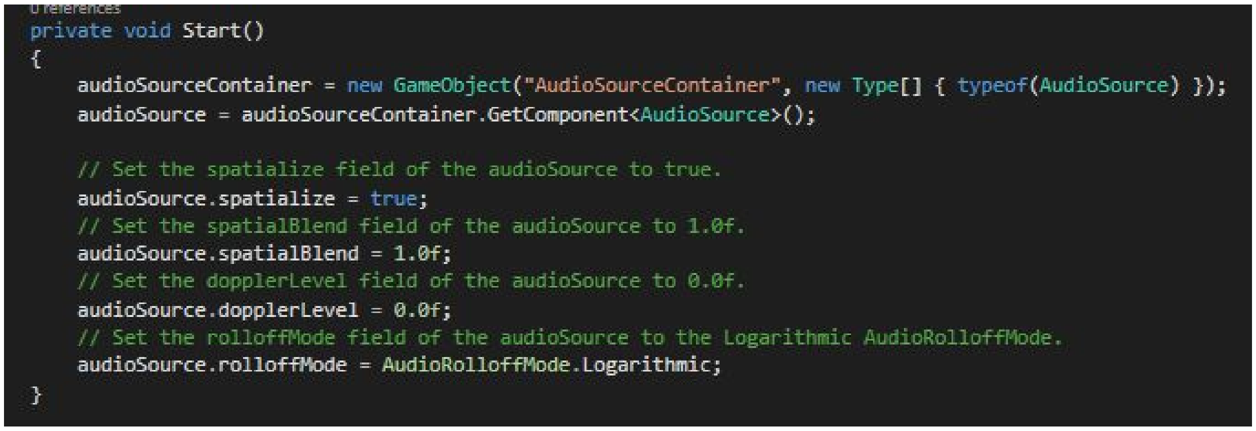


```
[✓] POLY                         [ ] Static
Tag      Untagged        Layer   Default
Prefab   Select   Revert   Apply

Transform
Position            X 0      Y 0      Z 0
Rotation            X 0      Y 180    Z 0
Scale               X 0.55   Y 0.55   Z 0.55

[✓] Poly Actions (Script)
Script                 PolyActions
Move Speed             5
Rotation Sensitivity   10
Teleporticles          Particles

[ ] Interpolator (Script)
Script                 Interpolator

[✓] Audio Source
AudioClip              PolyHover
Output                 SoundEffects (Mixer)
Mute                   [ ]
Spatialize             [✓]
Spatialize Post Effect [ ]
Bypass Effects         [ ]
Bypass Listener Effe   [ ]
Bypass Reverb Zone     [ ]
Play On Awake          [✓]
Loop                   [✓]
Priority               128   (High ... Low)
Volume                 1
Pitch                  1
Stereo Pan             0     (Left ... Right)
Spatial Blend          1     (2D ... 3D)
Reverb Zone Mix        1
3D Sound Settings
  Doppler Level        0
  Spread               0
  Volume Rolloff       Logarithmic Rolloff
  Min Distance         1
  Max Distance         500
```

Figure 24 : Audio source component

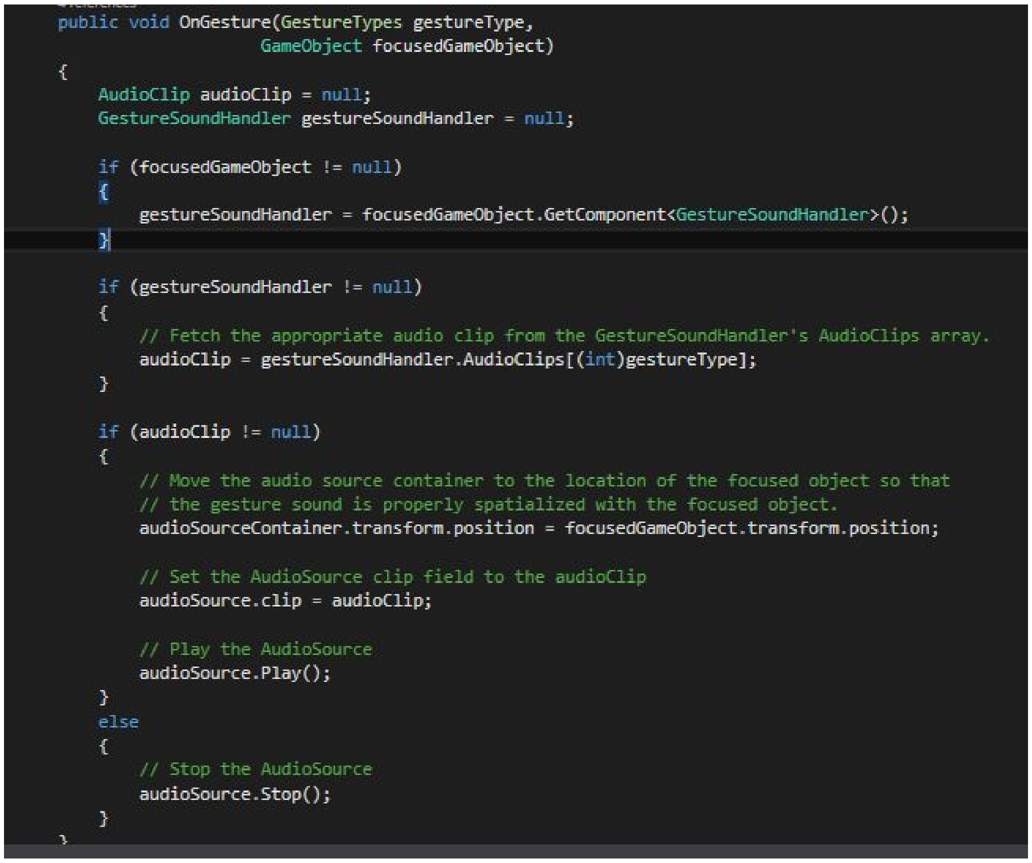
In the audio source component, the audio clip is the sound that is going to be played. Setting Doppler Level to 0 disables the changes in pitch caused by motion.

Initialising an audio source in a script and having it move with a gameobject :



```
0 references
private void Start()
{
    audioSourceContainer = new GameObject("AudioSourceContainer", new Type[] { typeof(AudioSource) });
    audioSource = audioSourceContainer.GetComponent<AudioSource>();

    // Set the spatialize field of the audioSource to true.
    audioSource.spatialize = true;
    // Set the spatialBlend field of the audioSource to 1.0f.
    audioSource.spatialBlend = 1.0f;
    // Set the dopplerLevel field of the audioSource to 0.0f.
    audioSource.dopplerLevel = 0.0f;
    // Set the rolloffMode field of the audioSource to the Logarithmic AudioRolloffMode.
    audioSource.rolloffMode = AudioRolloffMode.Logarithmic;
}
```

Figure 25 : Initialise audio source and audio source container



```
public void OnGesture(GestureTypes gestureType,
                      GameObject focusedGameObject)
{
    AudioClip audioClip = null;
    GestureSoundHandler gestureSoundHandler = null;

    if (focusedGameObject != null)
    {
        gestureSoundHandler = focusedGameObject.GetComponent<GestureSoundHandler>();
    }

    if (gestureSoundHandler != null)
    {
        // Fetch the appropriate audio clip from the GestureSoundHandler's AudioClips array.
        audioClip = gestureSoundHandler.AudioClips[(int)gestureType];
    }

    if (audioClip != null)
    {
        // Move the audio source container to the location of the focused object so that
        // the gesture sound is properly spatialized with the focused object.
        audioSourceContainer.transform.position = focusedGameObject.transform.position;

        // Set the AudioSource clip field to the audioClip
        audioSource.clip = audioClip;

        // Play the AudioSource
        audioSource.Play();
    }
    else
    {
        // Stop the AudioSource
        audioSource.Stop();
    }
}
```

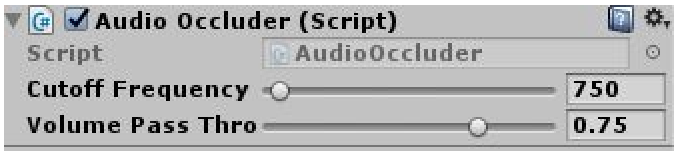
Figure 26 : Move the audio source container with the object. Play and stop the audio clip.

Audio source container is used to move the audio source with the focused gameobject. This project uses an Unity Audio mixer.

Spatial sound can work with spatial mapping:

- We can have interaction between holograms and real world using sound.
- We can occlude sound using the physical world. Sound like light can be occluded. For sound occlusion we use the audio occluder and audio emitter script. Audio Emitter scripts takes care of the audio influencers.

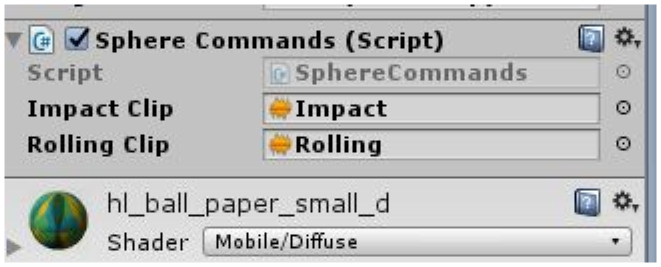
The audio occluder values need to be set accordingly to achieve a realistic effect.



Audio Occluder (Script)
Script: AudioOccluder
Cutoff Frequency: 750
Volume Pass Thro: 0.75

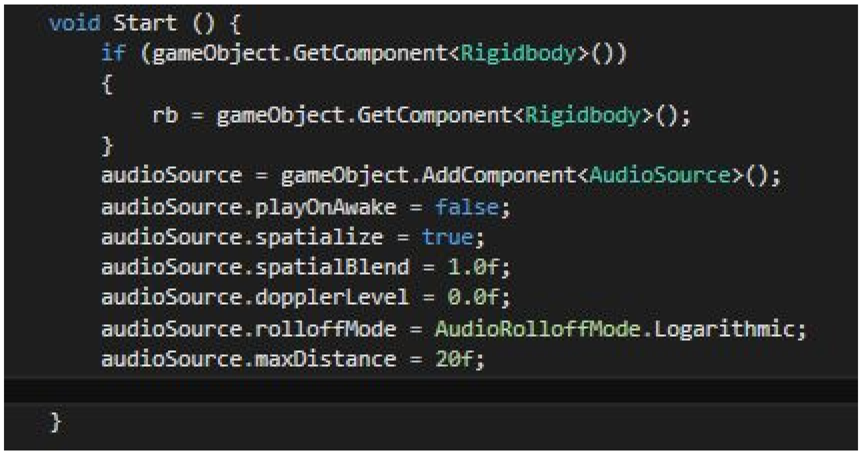
Figure 27 : Audio Occluder script component

Example of spatial sound working with spatial mapping : We play an impact clip when the sphere falls on the ground and a rolling clip while the sphere/ball (rigidbodies) is rolling on the ground.



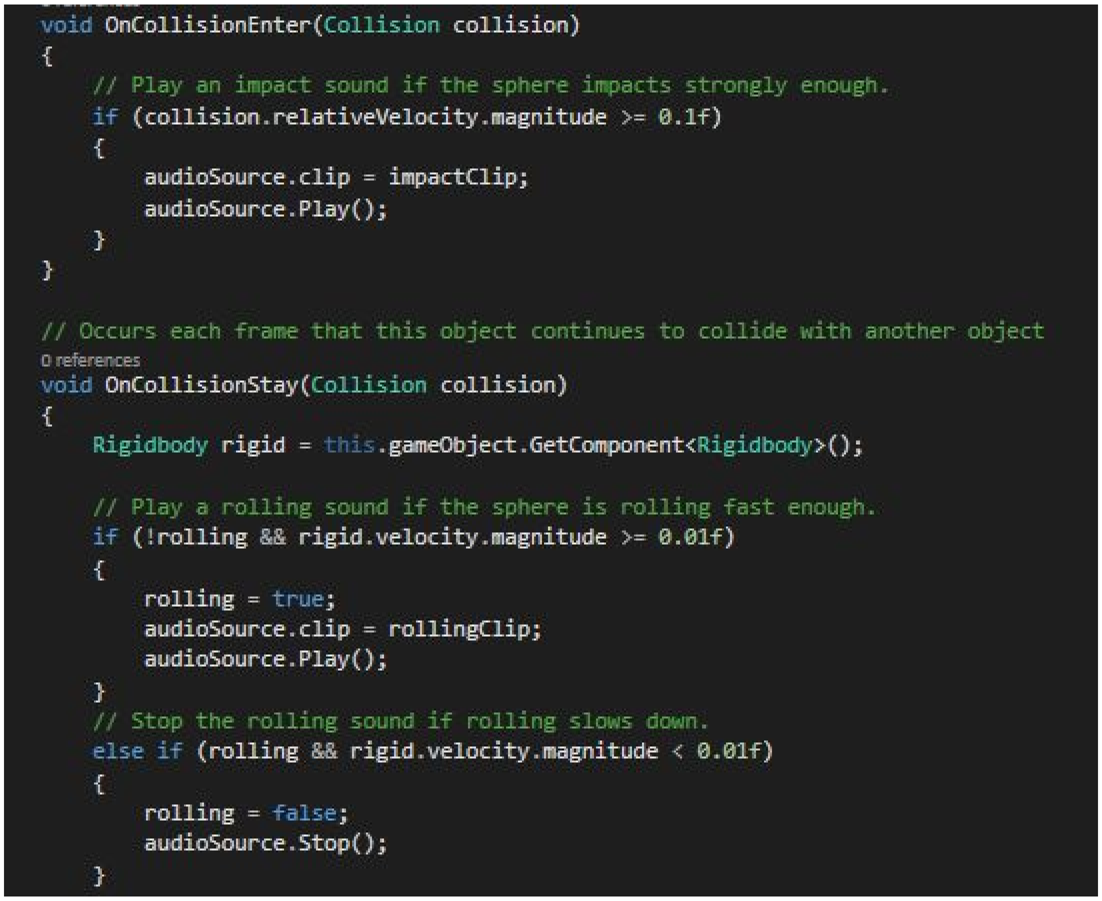
Sphere Commands (Script)
Script: SphereCommands
Impact Clip: Impact
Rolling Clip: Rolling
hl_ball_paper_small_d
Shader: Mobile/Diffuse

Figure 28 : Set the audio clips



```
void Start () {
    if (gameObject.GetComponent<Rigidbody>())
    {
        rb = gameObject.GetComponent<Rigidbody>();
    }
    audioSource = gameObject.AddComponent<AudioSource>();
    audioSource.playOnAwake = false;
    audioSource.spatialize = true;
    audioSource.spatialBlend = 1.0f;
    audioSource.dopplerLevel = 0.0f;
    audioSource.rolloffMode = AudioRolloffMode.Logarithmic;
    audioSource.maxDistance = 20f;

}
```

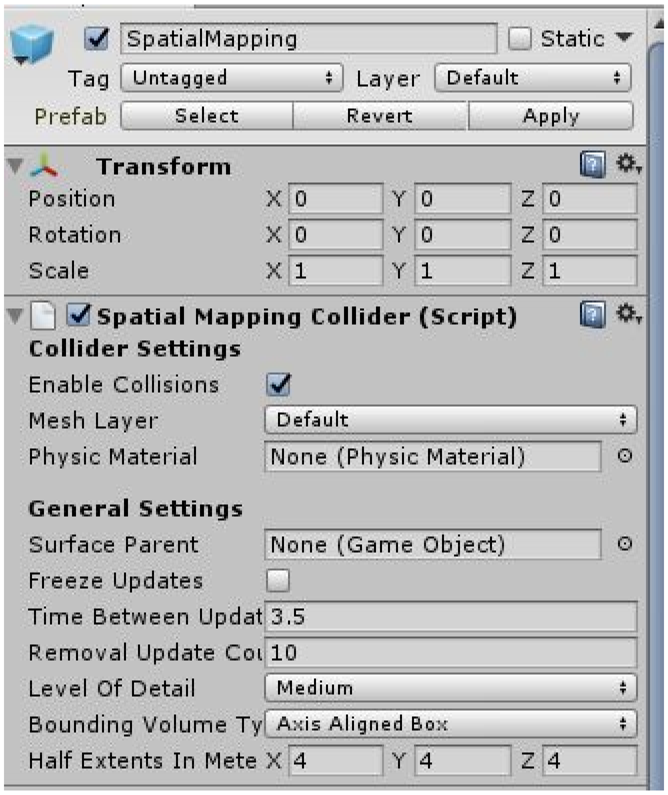
Figure 29 : Initialise audio source



```
void OnCollisionEnter(Collision collision)
{
    // Play an impact sound if the sphere impacts strongly enough.
    if (collision.relativeVelocity.magnitude >= 0.1f)
    {
        audioSource.clip = impactClip;
        audioSource.Play();
    }
}

// Occurs each frame that this object continues to collide with another object
0 references
void OnCollisionStay(Collision collision)
{
    Rigidbody rigid = this.gameObject.GetComponent<Rigidbody>();

    // Play a rolling sound if the sphere is rolling fast enough.
    if (!rolling && rigid.velocity.magnitude >= 0.01f)
    {
        rolling = true;
        audioSource.clip = rollingClip;
        audioSource.Play();
    }
    // Stop the rolling sound if rolling slows down.
    else if (rolling && rigid.velocity.magnitude < 0.01f)
    {
        rolling = false;
        audioSource.Stop();
    }
```

Figure 30 : Play the sounds

OnCollisionEnter is called by Unity when the ball makes contact with the spatial mapping surface. OnCollisionStay is called as long as the ball is in contact with the spatial mapping surface. For these functions to work we need to enable physic interaction between holograms and the spatial mapping surface. This is done by adding the Spatial Mapping Collider to the Spatial Manager prefab.


SpatialMapping ☐ Static
Tag: Untagged | Layer: Default
Prefab: Select | Revert | Apply

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

✓ Spatial Mapping Collider (Script)
Collider Settings
Enable Collisions ✓
Mesh Layer: Default
Physic Material: None (Physic Material)

General Settings
Surface Parent: None (Game Object)
Freeze Updates ☐
Time Between Updat 3.5
Removal Update Coı 10
Level Of Detail: Medium
Bounding Volume Ty: Axis Aligned Box
Half Extents In Mete X 4 Y 4 Z 4

Figure 31 : Spatial Mapping Collider for physics collisions

Spatial Mapping *(Holograms 230)*

Spatial mapping involves :

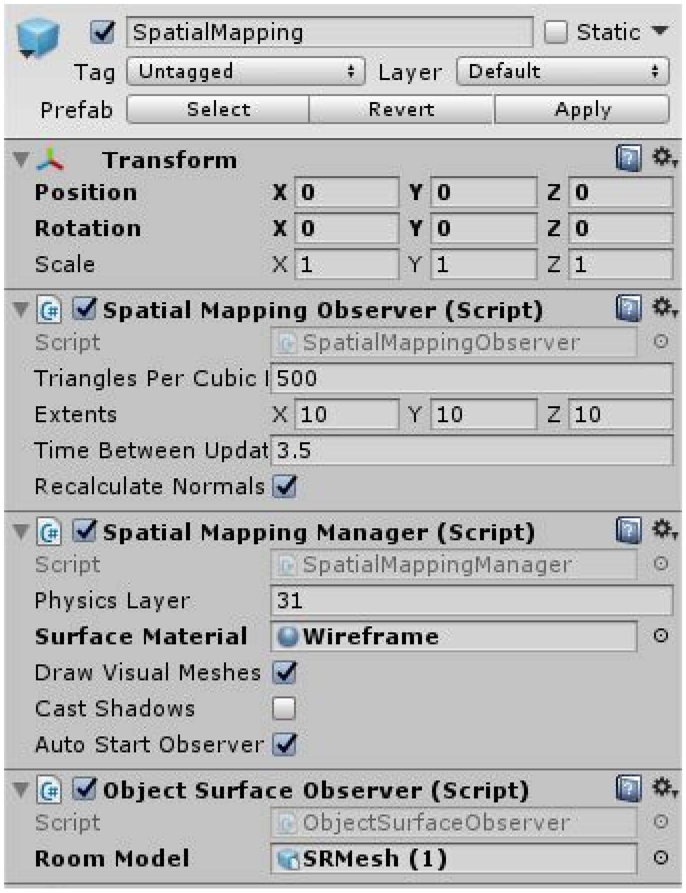
- Gaining an understanding of the environment.
- Ability to visualize our world.
- To place holograms on real world surfaces.
- Interact with the world using physics.
- To occlude holograms and navigate the world.

There are different techniques to process spatial mapping data (hole filling, smoothing, plane finding, vertex removal)

Saving a room model with the spatial mapping data of the room.

Browser -> IP Address of the HoloLens -> Windows Device Portal -> 3D View -> Update -> Surface Reconstruction -> Save -> FileName(SRMesh.obj) -> Assets folder and drag and drop in room model of the Object Surface Observer

This room model will be used for spatial mapping now.



```
☑ SpatialMapping                 ☐ Static
Tag     Untagged        Layer  Default
Prefab  Select   Revert   Apply

Transform
Position              X 0    Y 0    Z 0
Rotation              X 0    Y 0    Z 0
Scale                 X 1    Y 1    Z 1

☑ Spatial Mapping Observer (Script)
Script                SpatialMappingObserver
Triangles Per Cubic I 500
Extents               X 10   Y 10   Z 10
Time Between Updat    3.5
Recalculate Normals   ☑

☑ Spatial Mapping Manager (Script)
Script                SpatialMappingManager
Physics Layer         31
Surface Material      Wireframe
Draw Visual Meshes    ☑
Cast Shadows          ☐
Auto Start Observer   ☑

☑ Object Surface Observer (Script)
Script                ObjectSurfaceObserver
Room Model            SRMesh (1)
```

Figure 32 : Setting the room model

We can use planes for hologram placement, that is placing the holograms only on vertical or horizontal planes. We can classify planes as floor, tables, walls or ceilings, we can also destroy the planes that we don't want. The surface meshes to planes scripts does this.

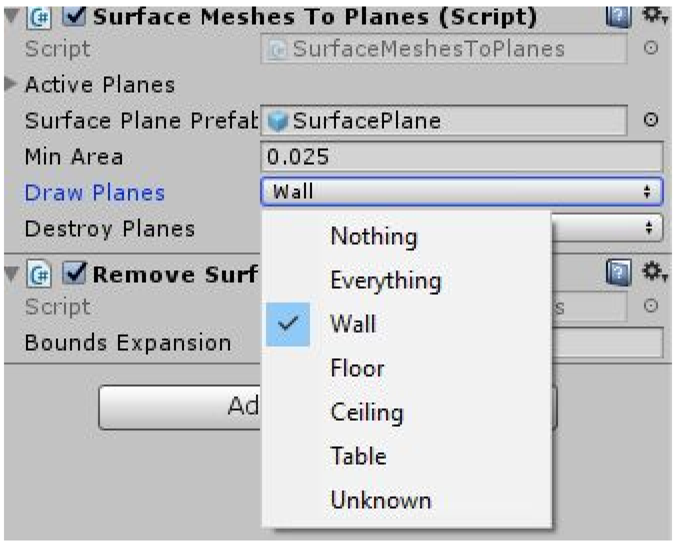

Surface Meshes To Planes (Script)
Script: SurfaceMeshesToPlanes
Active Planes
Surface Plane Prefab: SurfacePlane
Min Area: 0.025
Draw Planes: Wall
Destroy Planes
Remove Surf
Script
Bounds Expansion

Figure 33 : Finding planes

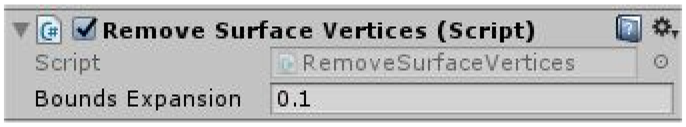

Remove Surface Vertices (Script)
Script: RemoveSurfaceVertices
Bounds Expansion: 0.1

Figure 34 : Remove vertices

Surface meshes to planes will find and generate planes based on the spatial mapping data. Remove Surface vertices will remove vertices from the spatial mapping mesh. This can be used to create holes in the mesh or to remove excess triangles that are no longer needed.

The play space manager and the placeable scripts are used to identify the horizontal and vertical planes and place an object on the appropriate plane

Occlusion :

- Determine if a hologram is occluded by a spatial mapping mesh.
- Apply different occlusion techniques.
- Hiding holograms behind real world objects.
- Setting up an occlusion material for occluded objects, the occluded object will be displayed with this material.

Other components of an application are described below :

Cursor

When the user's gaze is directed at a hologram we can set a cursor to be active and display the cursor, while the user is not gazing at any hologram we can set another cursor to be active and displayed at the same time, disabling the other cursor. The gazeManager/Raycast helps in identifying whether the user is gazing at a hologram or not.

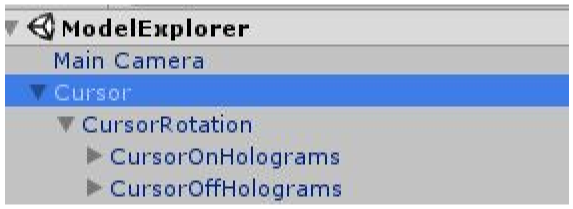
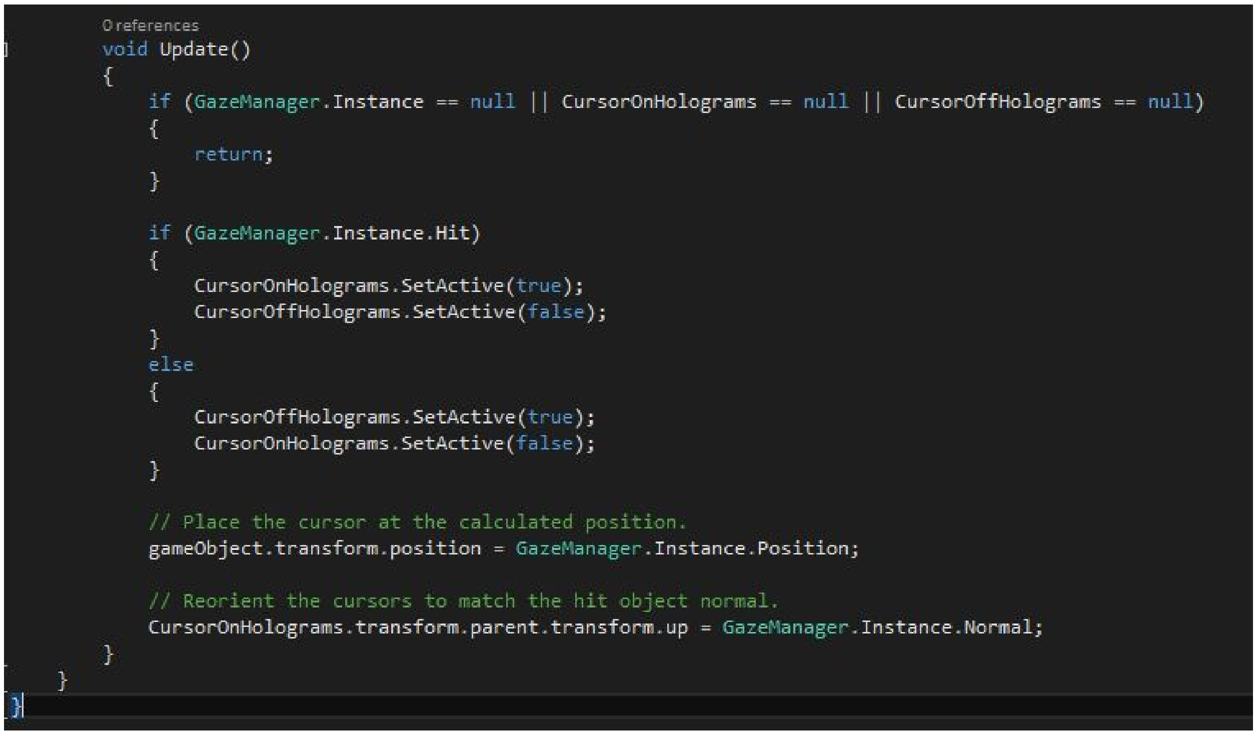


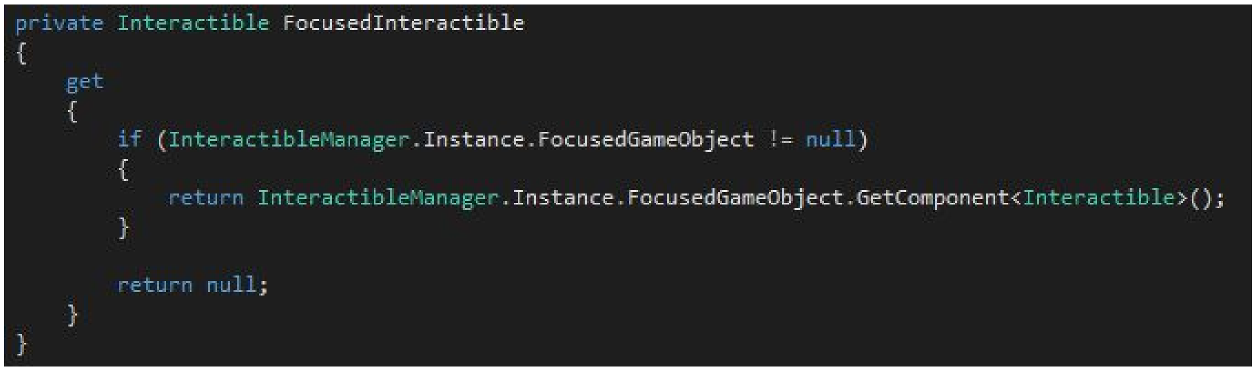
Figure 35 : Cursor gameobject



```
        0 references
        void Update()
        {
            if (GazeManager.Instance == null || CursorOnHolograms == null || CursorOffHolograms == null)
            {
                return;
            }

            if (GazeManager.Instance.Hit)
            {
                CursorOnHolograms.SetActive(true);
                CursorOffHolograms.SetActive(false);
            }
            else
            {
                CursorOffHolograms.SetActive(true);
                CursorOnHolograms.SetActive(false);
            }

            // Place the cursor at the calculated position.
            gameObject.transform.position = GazeManager.Instance.Position;

            // Reorient the cursors to match the hit object normal.
            CursorOnHolograms.transform.parent.transform.up = GazeManager.Instance.Normal;
        }
    }
}
```

Figure 36 : CursorOnHolograms is activated when the user is gazing at a hologram

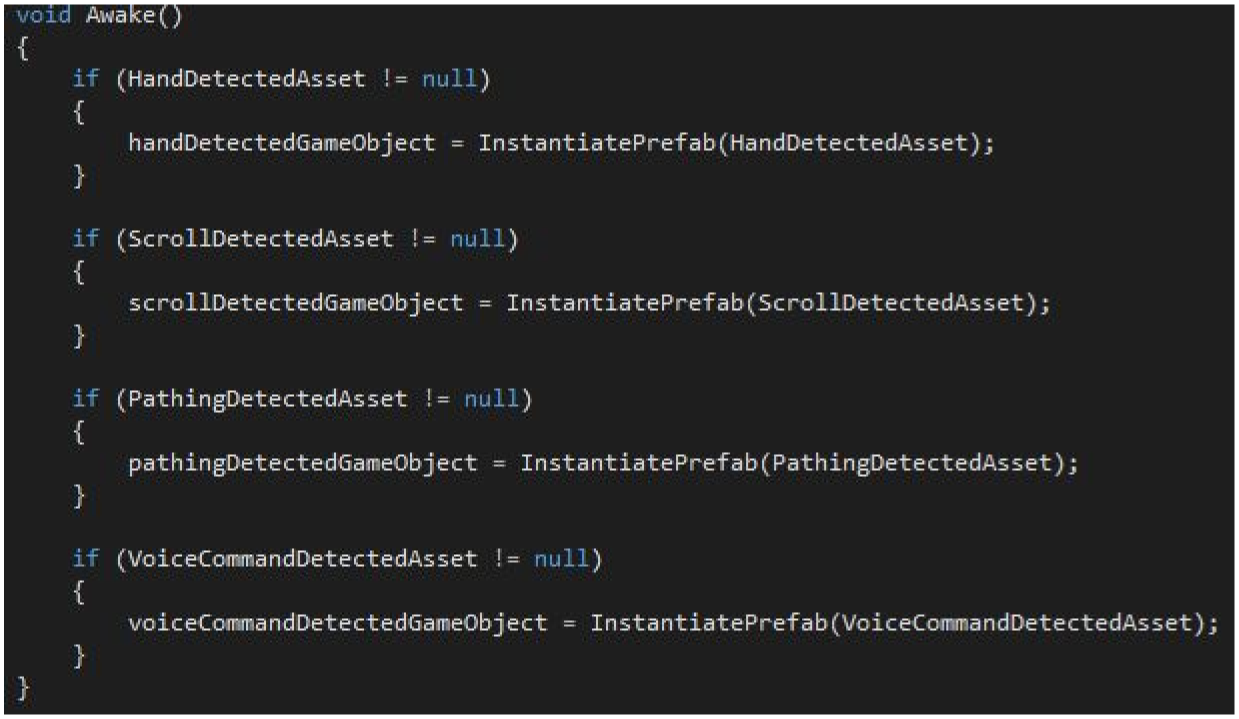
We can use the cursor to indicate other states as well :



```
private Interactible FocusedInteractible
{
    get
    {
        if (InteractibleManager.Instance.FocusedGameObject != null)
        {
            return InteractibleManager.Instance.FocusedGameObject.GetComponent<Interactible>();
        }

        return null;
    }
}
```

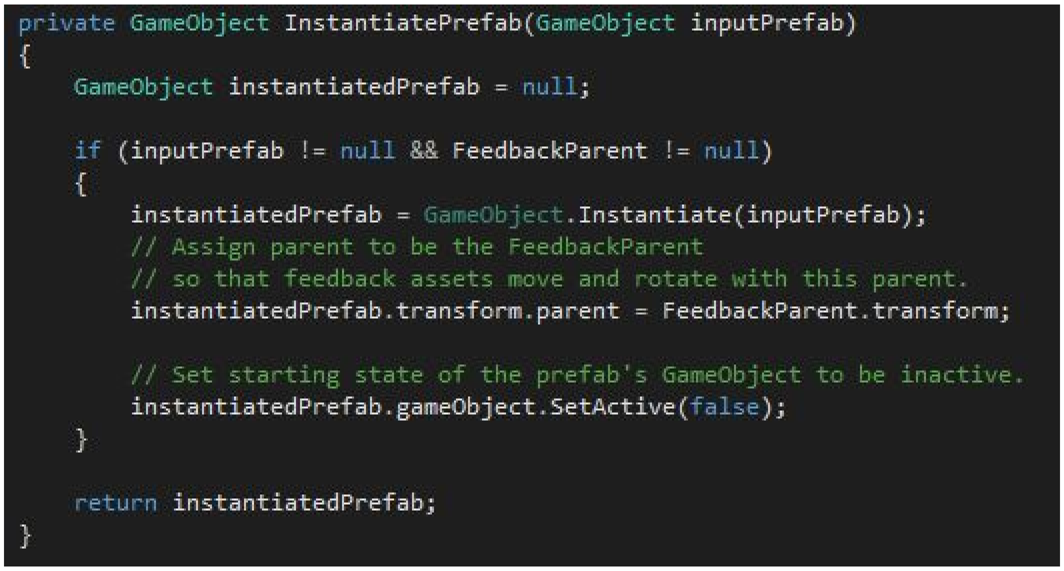
Figure 37 : Check if object can be interacted with

This is done to check if we are interacting with an gameobject that has the intercatible script attached to it, if it does then it means we can navigate and manipulate this gameobject, and therefore only for such objects will we display the Scroll and Pathing feedback cursors.



```
void Awake()
{
    if (HandDetectedAsset != null)
    {
        handDetectedGameObject = InstantiatePrefab(HandDetectedAsset);
    }

    if (ScrollDetectedAsset != null)
    {
        scrollDetectedGameObject = InstantiatePrefab(ScrollDetectedAsset);
    }

    if (PathingDetectedAsset != null)
    {
        pathingDetectedGameObject = InstantiatePrefab(PathingDetectedAsset);
    }

    if (VoiceCommandDetectedAsset != null)
    {
        voiceCommandDetectedGameObject = InstantiatePrefab(VoiceCommandDetectedAsset);
    }
}
```

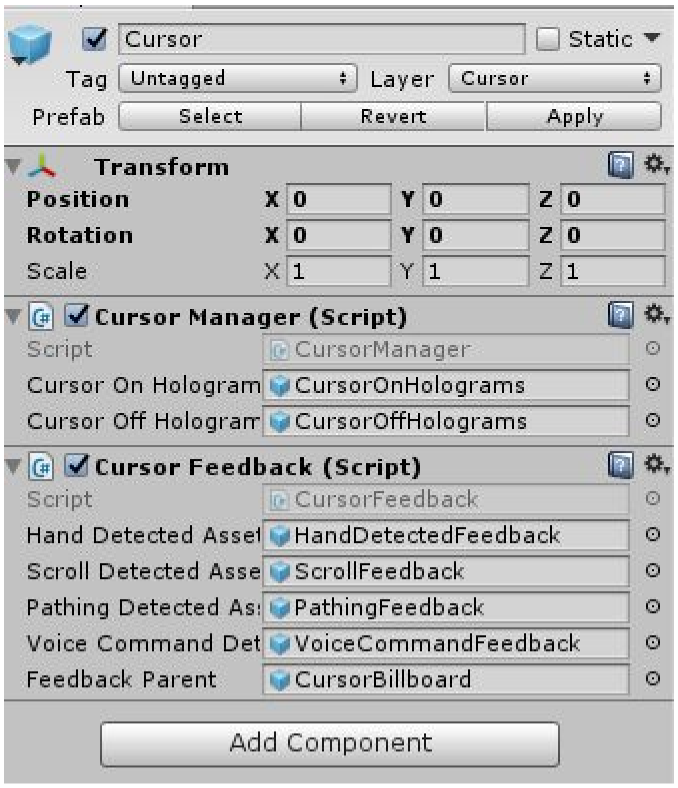
Figure 38 : States of the cursor



```
private GameObject InstantiatePrefab(GameObject inputPrefab)
{
    GameObject instantiatedPrefab = null;

    if (inputPrefab != null && FeedbackParent != null)
    {
        instantiatedPrefab = GameObject.Instantiate(inputPrefab);
        // Assign parent to be the FeedbackParent
        // so that feedback assets move and rotate with this parent.
        instantiatedPrefab.transform.parent = FeedbackParent.transform;

        // Set starting state of the prefab's GameObject to be inactive.
        instantiatedPrefab.gameObject.SetActive(false);
    }

    return instantiatedPrefab;
}
```

Figure 39 : Instantiate a state of the cursor.

We instantiate the prefabs and set the position to the position of the cursor gameobject.



✓ Cursor			Static
Tag	Untagged	Layer	Cursor
Prefab	Select	Revert	Apply

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

✓ Cursor Manager (Script)

Script	CursorManager
Cursor On Hologram	CursorOnHolograms
Cursor Off Hologram	CursorOffHolograms

✓ Cursor Feedback (Script)

Script	CursorFeedback
Hand Detected Asset	HandDetectedFeedback
Scroll Detected Asse	ScrollFeedback
Pathing Detected As	PathingFeedback
Voice Command Det	VoiceCommandFeedback
Feedback Parent	CursorBillboard

Figure 40 : The cursor gameobject

HandDetectedFeedback, ScrollFeedback, PathingFEedback, VoiceCommandFeedback are all prefabs in the Assets/Prefabs folder.

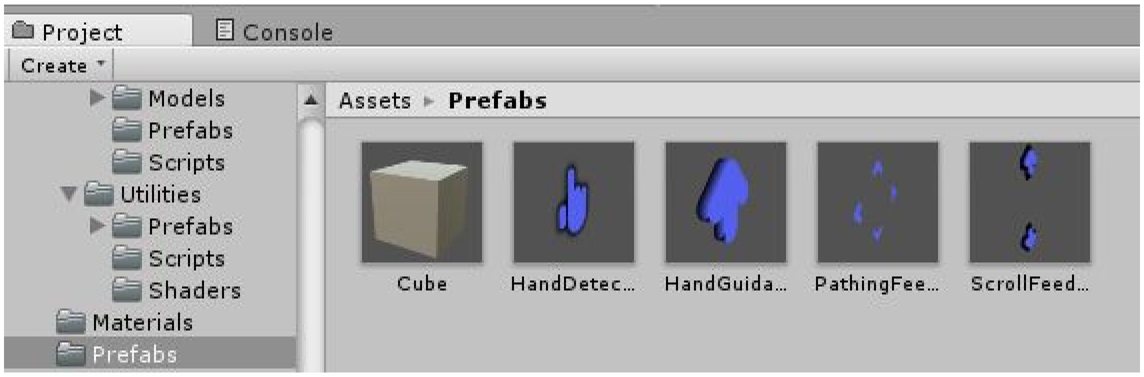
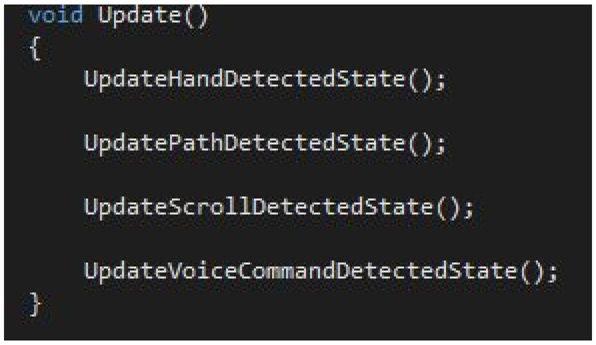


Figure 41 : Cursor state prefabs

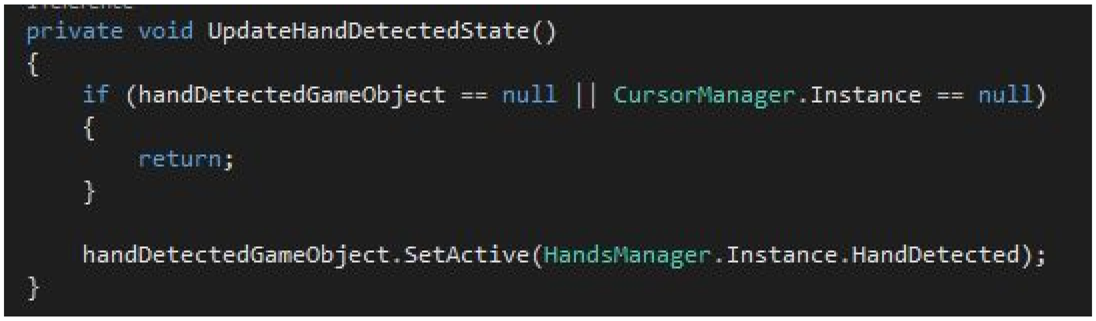
The feedback parent ensures that the visual feedback is attached to the cursor gameobject and since it is associated with the billboard script, the cursor will always face the user.



```
void Update()
{
    UpdateHandDetectedState();

    UpdatePathDetectedState();

    UpdateScrollDetectedState();

    UpdateVoiceCommandDetectedState();
}
```

Figure 42 : Update cursor state.

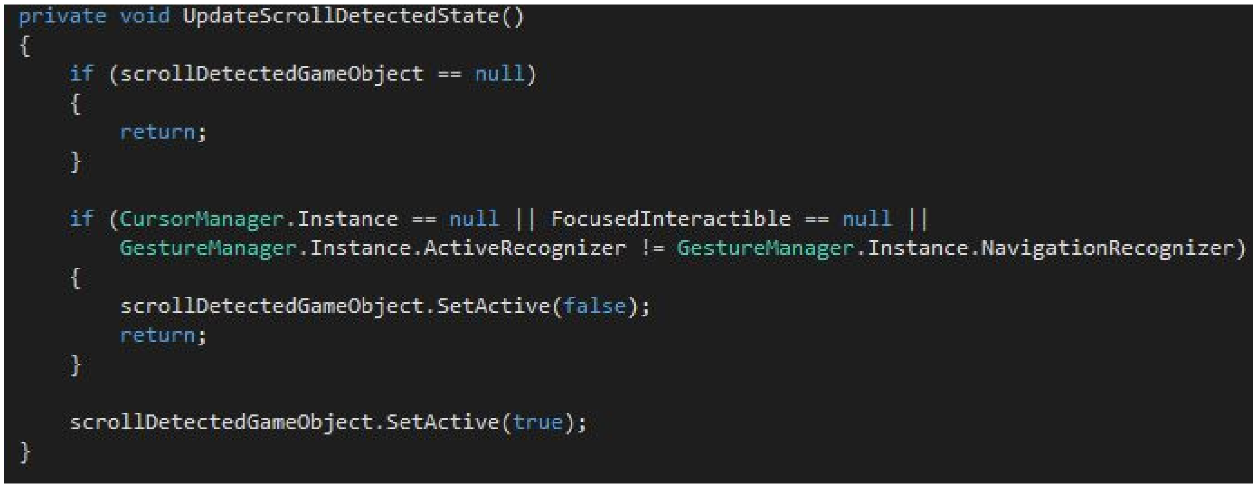
Hand Detected State : Detects whether the user's hand is being tracked and provides feedback to the user.



```
private void UpdateHandDetectedState()
{
    if (handDetectedGameObject == null || CursorManager.Instance == null)
    {
        return;
    }

    handDetectedGameObject.SetActive(HandsManager.Instance.HandDetected);
}
```

Figure 43 : Hand detected state of the cursor.

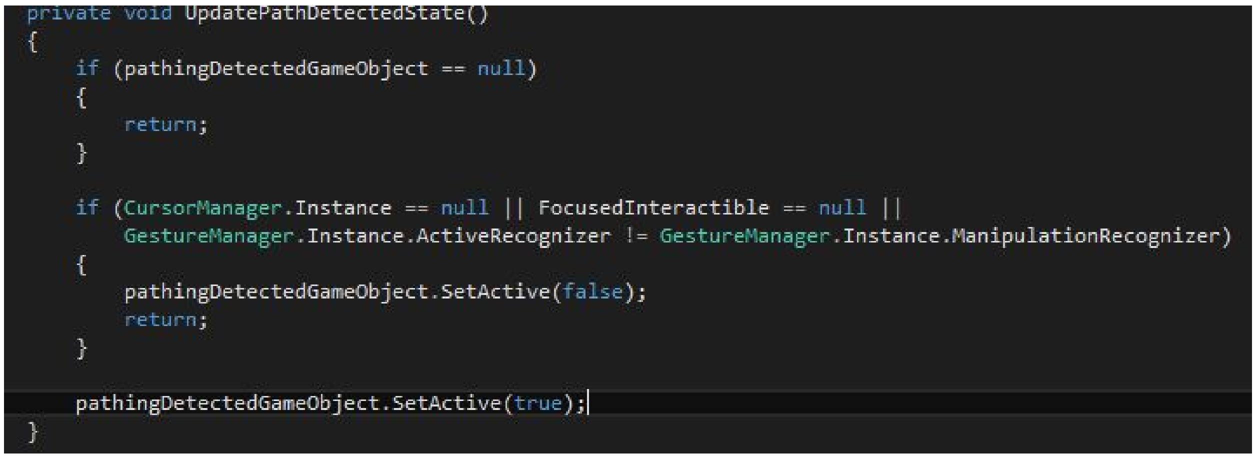
Scroll Feedback : Detects whether the user is using the navigation recognizer.



```
private void UpdateScrollDetectedState()
{
    if (scrollDetectedGameObject == null)
    {
        return;
    }

    if (CursorManager.Instance == null || FocusedInteractible == null ||
        GestureManager.Instance.ActiveRecognizer != GestureManager.Instance.NavigationRecognizer)
    {
        scrollDetectedGameObject.SetActive(false);
        return;
    }

    scrollDetectedGameObject.SetActive(true);
}
```

Figure 44 : Scroll feedback state of the cursor

Pathing Feedback : Detect whether the user is using the manipulation recognizer.



```
private void UpdatePathDetectedState()
{
    if (pathingDetectedGameObject == null)
    {
        return;
    }

    if (CursorManager.Instance == null || FocusedInteractible == null ||
        GestureManager.Instance.ActiveRecognizer != GestureManager.Instance.ManipulationRecognizer)
    {
        pathingDetectedGameObject.SetActive(false);
        return;
    }

    pathingDetectedGameObject.SetActive(true);
}
```

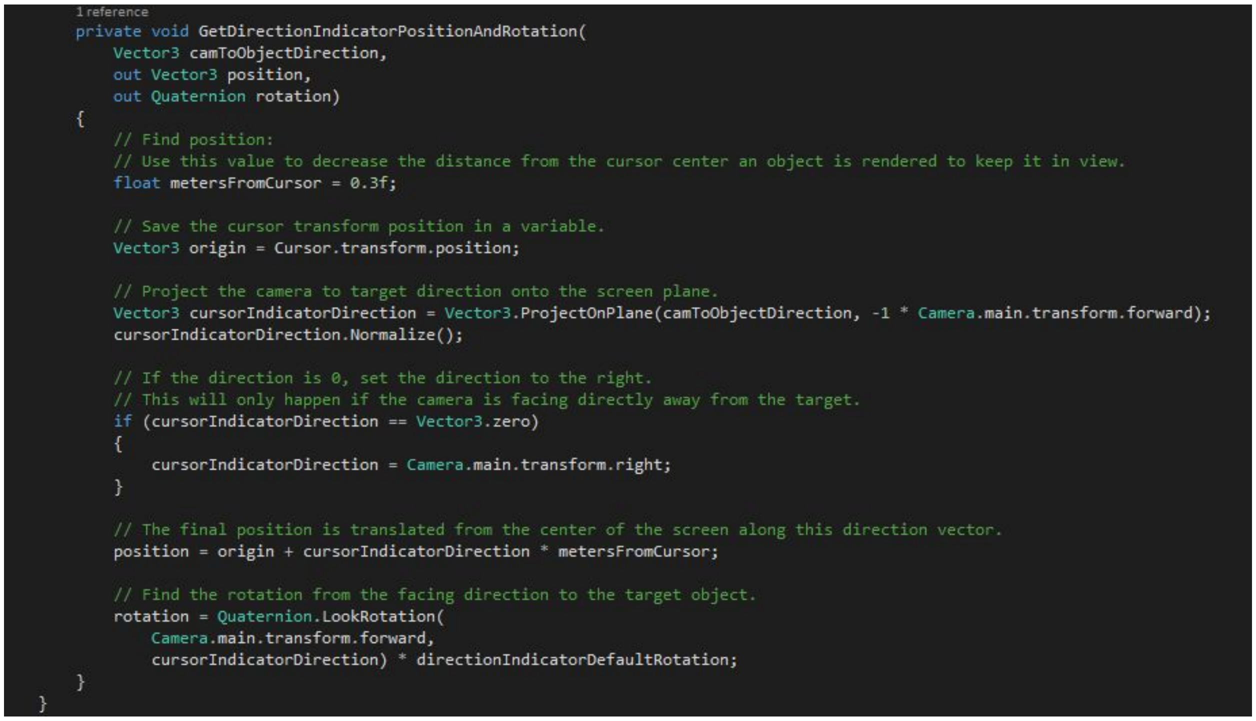
Figure 45 : Pathing feedback state of the user

These prefabs need to be activated at the appropriate moment, that is when the user needs to be given the visual feedback informing the user which state he/she is in.

We can also provide visual feedback for other actions as necessary. For example voice feedback can be provided, so we can have a voice prefab that is displayed on the hololens when the user is peaking, thereby providing visual feedback to the user that his/her voice is being heard.

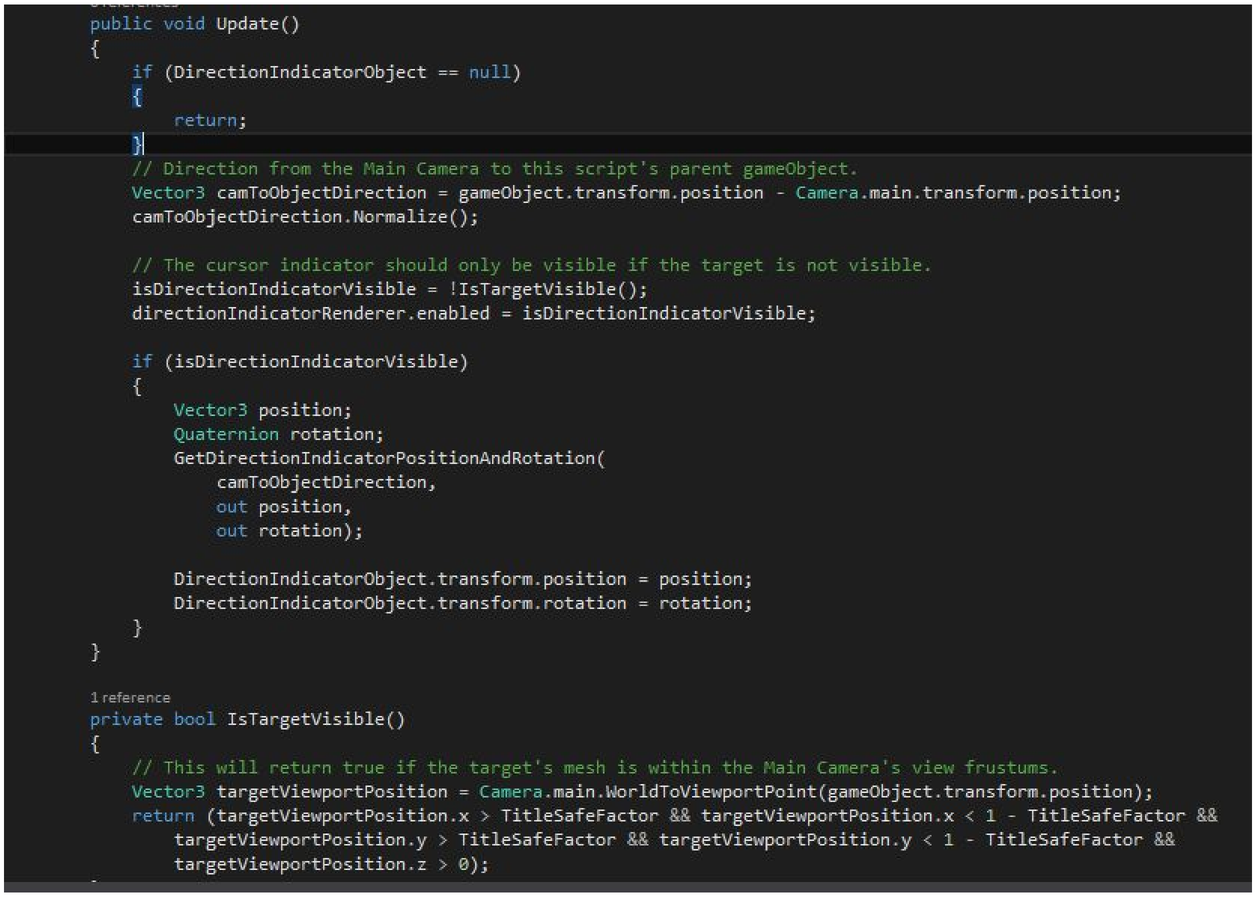
Directional Indicator

An indicator alongside the cursor that indicates which direction the user needs to move in order to see the hologram the indicator is associated with. Remove the collider and rigidbody component so that the indicator does not interfere with Unity's physics system.



```
        1 reference
        private void GetDirectionIndicatorPositionAndRotation(
            Vector3 camToObjectDirection,
            out Vector3 position,
            out Quaternion rotation)
        {
            // Find position:
            // Use this value to decrease the distance from the cursor center an object is rendered to keep it in view.
            float metersFromCursor = 0.3f;

            // Save the cursor transform position in a variable.
            Vector3 origin = Cursor.transform.position;

            // Project the camera to target direction onto the screen plane.
            Vector3 cursorIndicatorDirection = Vector3.ProjectOnPlane(camToObjectDirection, -1 * Camera.main.transform.forward);
            cursorIndicatorDirection.Normalize();

            // If the direction is 0, set the direction to the right.
            // This will only happen if the camera is facing directly away from the target.
            if (cursorIndicatorDirection == Vector3.zero)
            {
                cursorIndicatorDirection = Camera.main.transform.right;
            }

            // The final position is translated from the center of the screen along this direction vector.
            position = origin + cursorIndicatorDirection * metersFromCursor;

            // Find the rotation from the facing direction to the target object.
            rotation = Quaternion.LookRotation(
                Camera.main.transform.forward,
                cursorIndicatorDirection) * directionIndicatorDefaultRotation;
        }
    }
```

Figure 46 : Direction Indicator position and rotation



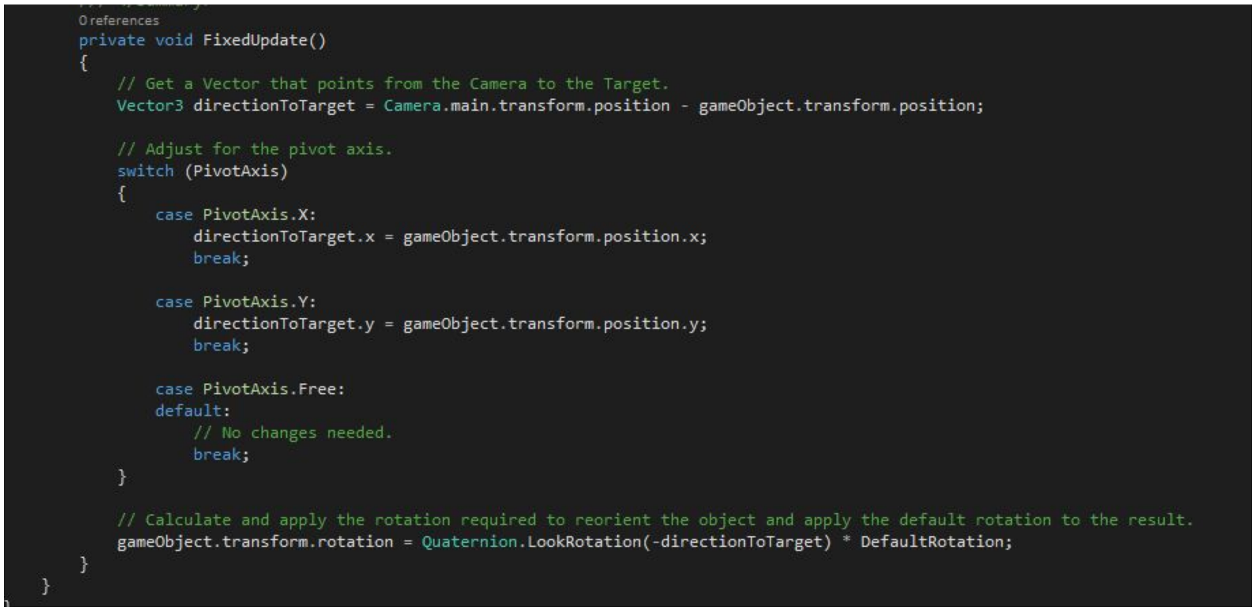
```
public void Update()
{
    if (DirectionIndicatorObject == null)
    {
        return;
    }
    // Direction from the Main Camera to this script's parent gameObject.
    Vector3 camToObjectDirection = gameObject.transform.position - Camera.main.transform.position;
    camToObjectDirection.Normalize();

    // The cursor indicator should only be visible if the target is not visible.
    isDirectionIndicatorVisible = !IsTargetVisible();
    directionIndicatorRenderer.enabled = isDirectionIndicatorVisible;

    if (isDirectionIndicatorVisible)
    {
        Vector3 position;
        Quaternion rotation;
        GetDirectionIndicatorPositionAndRotation(
            camToObjectDirection,
            out position,
            out rotation);

        DirectionIndicatorObject.transform.position = position;
        DirectionIndicatorObject.transform.rotation = rotation;
    }
}

1 reference
private bool IsTargetVisible()
{
    // This will return true if the target's mesh is within the Main Camera's view frustums.
    Vector3 targetViewportPosition = Camera.main.WorldToViewportPoint(gameObject.transform.position);
    return (targetViewportPosition.x > TitleSafeFactor && targetViewportPosition.x < 1 - TitleSafeFactor &&
        targetViewportPosition.y > TitleSafeFactor && targetViewportPosition.y < 1 - TitleSafeFactor &&
        targetViewportPosition.z > 0);
```

Figure 47 : Update direction indicator

First we check if we need to display the direction indicator, if we need to we calculate the position and rotation of the direction indicator object.

Bill Boarding

Billboarding is the technique of always ensuring the hologram faces the user. The billboarding script is available in the HoloToolKit, the script needs to be attached to the hologram as a component. The the axis of rotation needs to be changed to Y. This means that the hologram will rotate about the Y-axis and hence the X and Z axis will move around in such a way that the hologram will always face the user. The script needs to be attached to the object that you want to billboard, and the Pivot Axis needs to be changed to Y in the inspector panel.



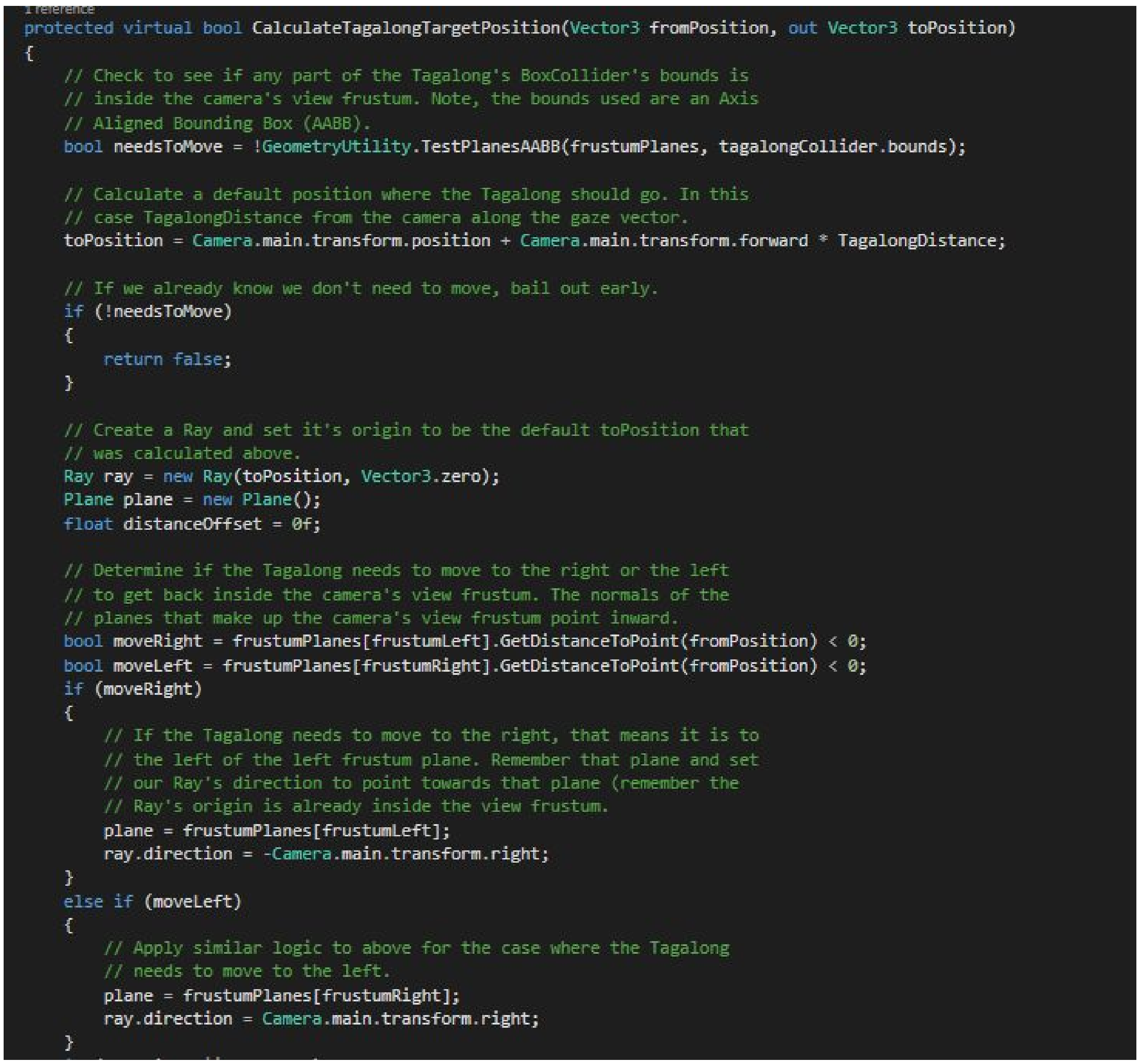
```
        0 references
        private void FixedUpdate()
        {
            // Get a Vector that points from the Camera to the Target.
            Vector3 directionToTarget = Camera.main.transform.position - gameObject.transform.position;

            // Adjust for the pivot axis.
            switch (PivotAxis)
            {
                case PivotAxis.X:
                    directionToTarget.x = gameObject.transform.position.x;
                    break;

                case PivotAxis.Y:
                    directionToTarget.y = gameObject.transform.position.y;
                    break;

                case PivotAxis.Free:
                default:
                    // No changes needed.
                    break;
            }

            // Calculate and apply the rotation required to reorient the object and apply the default rotation to the result.
            gameObject.transform.rotation = Quaternion.LookRotation(-directionToTarget) * DefaultRotation;
        }
    }
```

Figure 48 : Billboarding script

Tag Along

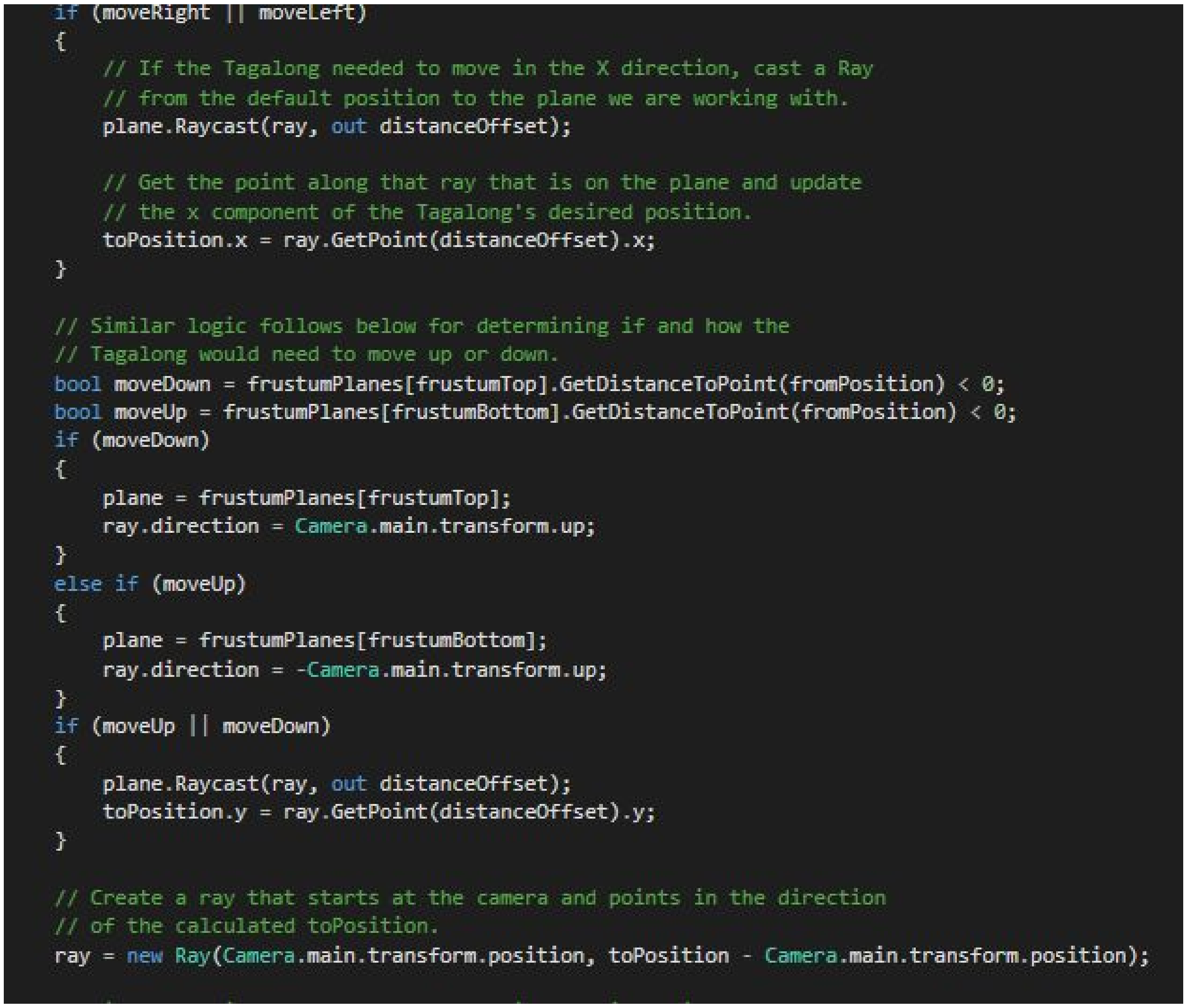
Tag along is having the hologram follow the user around the room at a relatively fixed position.

A tag along script component is attached to the object you want to tag along, as soon as the hologram leaves the frustum planes/view of the user, a position is calculated such that the hologram is brought back into the user's view, that is back into the frustum planes.



```
1 reference
protected virtual bool CalculateTagalongTargetPosition(Vector3 fromPosition, out Vector3 toPosition)
{
    // Check to see if any part of the Tagalong's BoxCollider's bounds is
    // inside the camera's view frustum. Note, the bounds used are an Axis
    // Aligned Bounding Box (AABB).
    bool needsToMove = !GeometryUtility.TestPlanesAABB(frustumPlanes, tagalongCollider.bounds);

    // Calculate a default position where the Tagalong should go. In this
    // case TagalongDistance from the camera along the gaze vector.
    toPosition = Camera.main.transform.position + Camera.main.transform.forward * TagalongDistance;

    // If we already know we don't need to move, bail out early.
    if (!needsToMove)
    {
        return false;
    }

    // Create a Ray and set it's origin to be the default toPosition that
    // was calculated above.
    Ray ray = new Ray(toPosition, Vector3.zero);
    Plane plane = new Plane();
    float distanceOffset = 0f;

    // Determine if the Tagalong needs to move to the right or the left
    // to get back inside the camera's view frustum. The normals of the
    // planes that make up the camera's view frustum point inward.
    bool moveRight = frustumPlanes[frustumLeft].GetDistanceToPoint(fromPosition) < 0;
    bool moveLeft = frustumPlanes[frustumRight].GetDistanceToPoint(fromPosition) < 0;
    if (moveRight)
    {
        // If the Tagalong needs to move to the right, that means it is to
        // the left of the left frustum plane. Remember that plane and set
        // our Ray's direction to point towards that plane (remember the
        // Ray's origin is already inside the view frustum.
        plane = frustumPlanes[frustumLeft];
        ray.direction = -Camera.main.transform.right;
    }
    else if (moveLeft)
    {
        // Apply similar logic to above for the case where the Tagalong
        // needs to move to the left.
        plane = frustumPlanes[frustumRight];
        ray.direction = Camera.main.transform.right;
    }
```

Figure 49 : Determine position of object with respect to frustum planes



```
if (moveRight || moveLeft)
{
    // If the Tagalong needed to move in the X direction, cast a Ray
    // from the default position to the plane we are working with.
    plane.Raycast(ray, out distanceOffset);

    // Get the point along that ray that is on the plane and update
    // the x component of the Tagalong's desired position.
    toPosition.x = ray.GetPoint(distanceOffset).x;
}

// Similar logic follows below for determining if and how the
// Tagalong would need to move up or down.
bool moveDown = frustumPlanes[frustumTop].GetDistanceToPoint(fromPosition) < 0;
bool moveUp = frustumPlanes[frustumBottom].GetDistanceToPoint(fromPosition) < 0;
if (moveDown)
{
    plane = frustumPlanes[frustumTop];
    ray.direction = Camera.main.transform.up;
}
else if (moveUp)
{
    plane = frustumPlanes[frustumBottom];
    ray.direction = -Camera.main.transform.up;
}
if (moveUp || moveDown)
{
    plane.Raycast(ray, out distanceOffset);
    toPosition.y = ray.GetPoint(distanceOffset).y;
}

// Create a ray that starts at the camera and points in the direction
// of the calculated toPosition.
ray = new Ray(Camera.main.transform.position, toPosition - Camera.main.transform.position);
```

Figure 50 : Calculate the position of the hologram, to be in view

plane.Raycast : where the ray intersects the plane, and the distance between the origin of the ray and the plane is stored in distanceOffset.

ray.GetPoint gives the point that is at distance of disatnceOffset on the ray in the x-direction.

****

These are the basics to the Hololens Academy tutorials. The scripts, techniques and components in the Hololens academy may or may not similar to the ones in the HoloToolkit. The HoloToolkit has its own set of scripts and components that may differ in their functioning as compared to the Hololens Academy.

****

<u>Thank You</u>