Task 1

1. Setting Up the Project

- **Chosen Engine:** Unity with ARCore
- **Template Used:** AR Core Template

The first step was to create a new Unity project using the AR Core template. This template comes pre-configured with the necessary AR Foundation and ARCore packages, including several basic AR functionalities such as plane detection, light estimation, and object placement. Using this template reduces initial setup time, ensuring that all plugins and dependencies required to run AR applications are installed.

After the project was created:

- **Sample Scene:** A default sample scene was included, which had the basic AR setup ready to detect planes in the environment using the camera.

2. Raycast Example Review

- **Objective:** Understand Raycasting and Modify for Interaction

The project included example scripts that demonstrated raycasting in the AR environment. Raycasting is essential in AR as it allows for detecting where a user interacts with the real-world surfaces through the screen.

I reviewed these scripts to understand how they worked and how to modify them for my use case—specifically, to detect taps on the mesh and return the coordinates of the tapped point.

3. Modifying the Scene

- **Default Object Spawner Removed:** The AR template often comes with object placement functionality, where tapping on a plane would spawn 3D objects. Since object spawning wasn't needed for this task, I removed the object-spawning logic to simplify the scene.

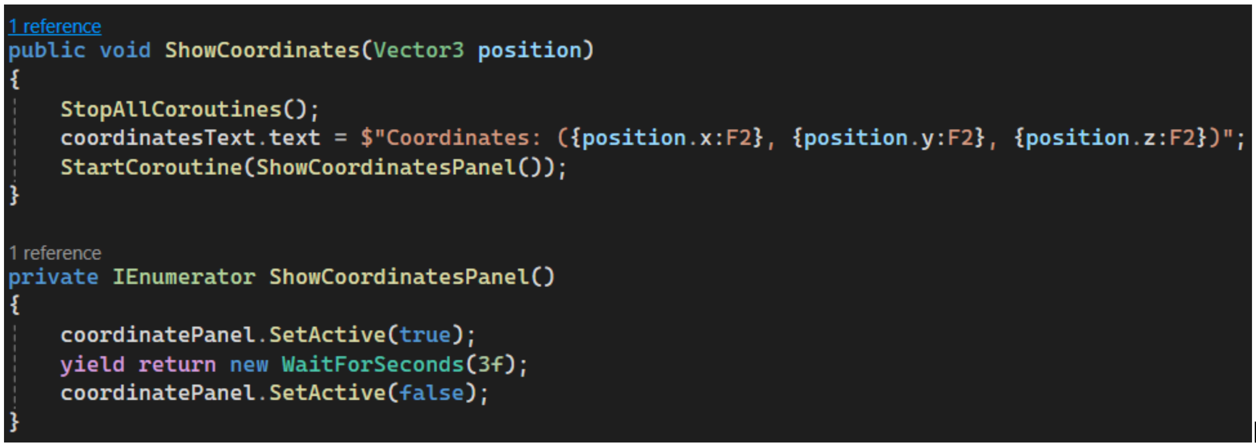
This cleaned up the project, making it easier to focus on the raycast and tap detection functionality.

4. Adding a UI for Coordinate Display

- **Objective:** Display coordinates of tapped points on the screen.

I created a simple UI consisting of a panel that would appear whenever the user taps the AR mesh. The panel would display the 3D coordinates (X, Y, Z) of the location where the user tapped.

Below is the script that handles displaying the coordinates:



```
1 reference
public void ShowCoordinates(Vector3 position)
{
    StopAllCoroutines();
    coordinatesText.text = $"Coordinates: ({position.x:F2}, {position.y:F2}, {position.z:F2})";
    StartCoroutine(ShowCoordinatesPanel());
}

1 reference
private IEnumerator ShowCoordinatesPanel()
{
    coordinatePanel.SetActive(true);
    yield return new WaitForSeconds(3f);
    coordinatePanel.SetActive(false);
}
```

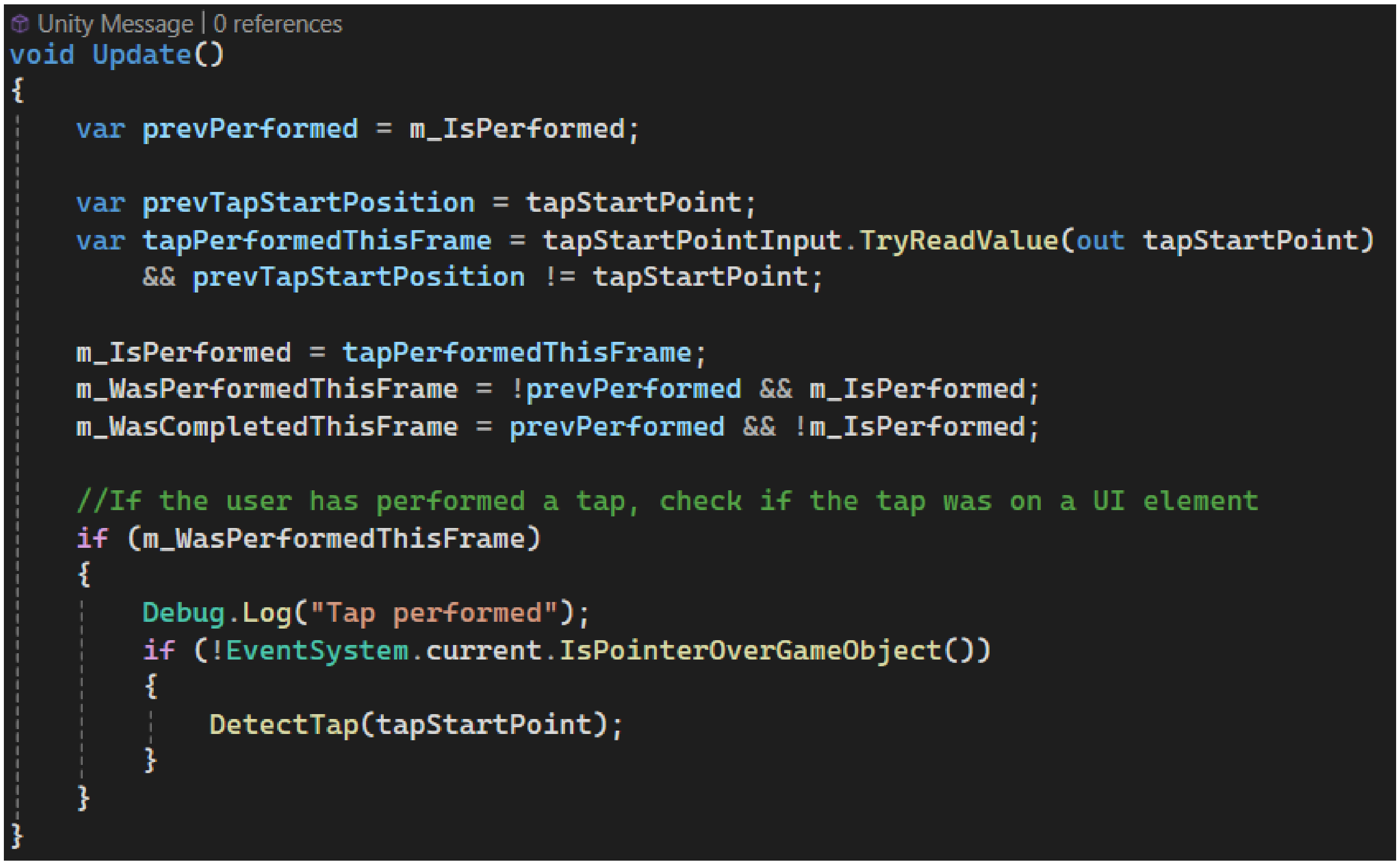
Explanation:

- The ShowCoordinates method is called when a tap on the mesh is detected. It updates the UI text to show the position in world coordinates.
- The coordinates are shown with two decimal precision for clarity.
- The ShowCoordinatesPanel coroutine makes the panel visible for 3 seconds before hiding it again.

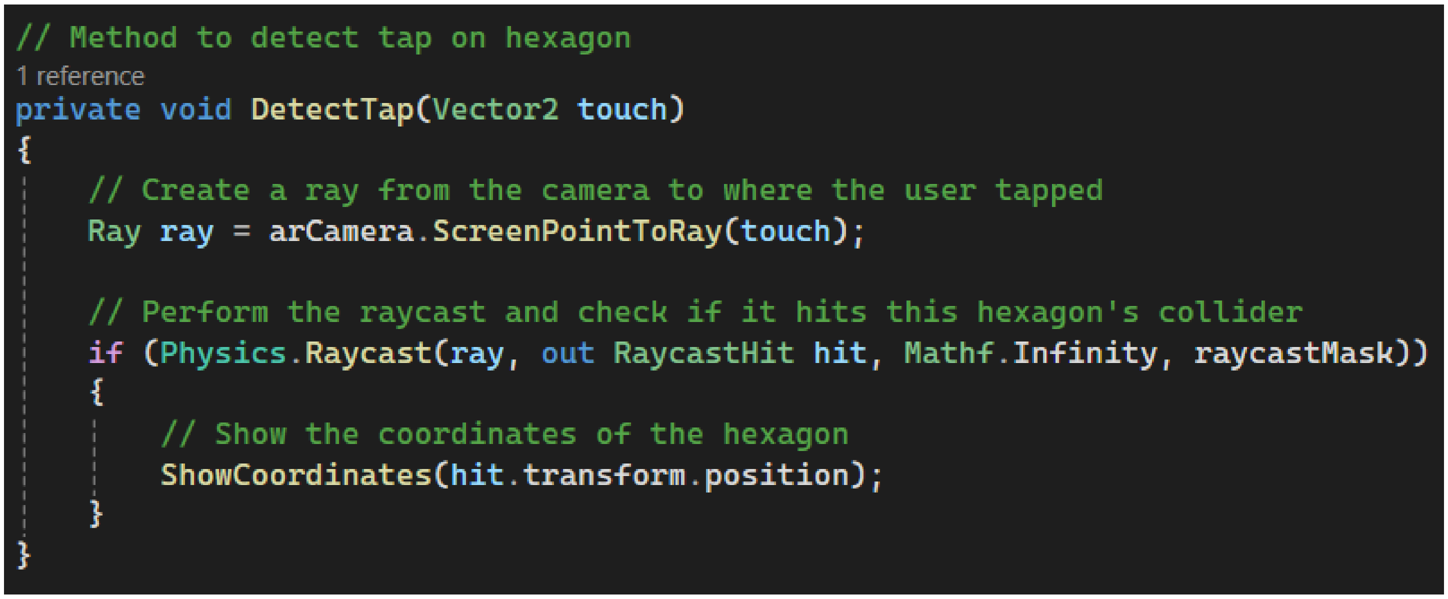
5. Implementing Tap Detection

- **Objective:** Detect user taps on the screen and check if they hit the AR mesh.

The tap detection script checks for user input each frame and ensures that the user's tap is registered. If the tap occurs over a recognized AR plane (mesh), the coordinates of the tapped location are calculated and displayed. Here's the tap detection code:



```
Unity Message | 0 references
void Update()
{
    var prevPerformed = m_IsPerformed;

    var prevTapStartPosition = tapStartPoint;
    var tapPerformedThisFrame = tapStartPointInput.TryReadValue(out tapStartPoint)
        && prevTapStartPosition != tapStartPoint;

    m_IsPerformed = tapPerformedThisFrame;
    m_WasPerformedThisFrame = !prevPerformed && m_IsPerformed;
    m_WasCompletedThisFrame = prevPerformed && !m_IsPerformed;

    //If the user has performed a tap, check if the tap was on a UI element
    if (m_WasPerformedThisFrame)
    {
        Debug.Log("Tap performed");
        if (!EventSystem.current.IsPointerOverGameObject())
        {
            DetectTap(tapStartPoint);
        }
    }
}
```



```
// Method to detect tap on hexagon
1 reference
private void DetectTap(Vector2 touch)
{
    // Create a ray from the camera to where the user tapped
    Ray ray = arCamera.ScreenPointToRay(touch);

    // Perform the raycast and check if it hits this hexagon's collider
    if (Physics.Raycast(ray, out RaycastHit hit, Mathf.Infinity, raycastMask))
    {
        // Show the coordinates of the hexagon
        ShowCoordinates(hit.transform.position);
    }
}
```

Explanation:

- In the Update method, I check each frame whether the user has tapped the screen.
- I use the EventSystem.current.IsPointerOverGameObject() to ensure the tap isn't on a UI element.
- If a valid tap is detected, the DetectTap method is called, which sends a ray from the camera to the tap location on the screen.
- The raycast checks if the ray intersects with the AR mesh (using the plane's collider). If a hit is detected, the coordinates of the hit are passed to ShowCoordinates, which displays them on the UI panel.

6. Final Output:

- **AR Plane Detection and Raycasting Functionality Completed:**
    - Using the default ARPlane prefab, the app detects planes in the real world.
    - Added layer AR_Plane and used a LayerMask to only detect the plane when raycasting
    - The raycast detects user taps on the plane, and the 3D coordinates of the hit location are displayed on the screen via the UI panel.
- **UI Panel:** The coordinates panel briefly appears each time the user taps the plane, showing the X, Y, and Z world coordinates of the point tapped.

Task 2

1. Setting Up Hexagon Assets and Materials

- **Hexagon Asset:** To create the tiling, I sourced a **free hexagon asset** from the Unity Asset Store, which provides a reusable hexagonal prefab.

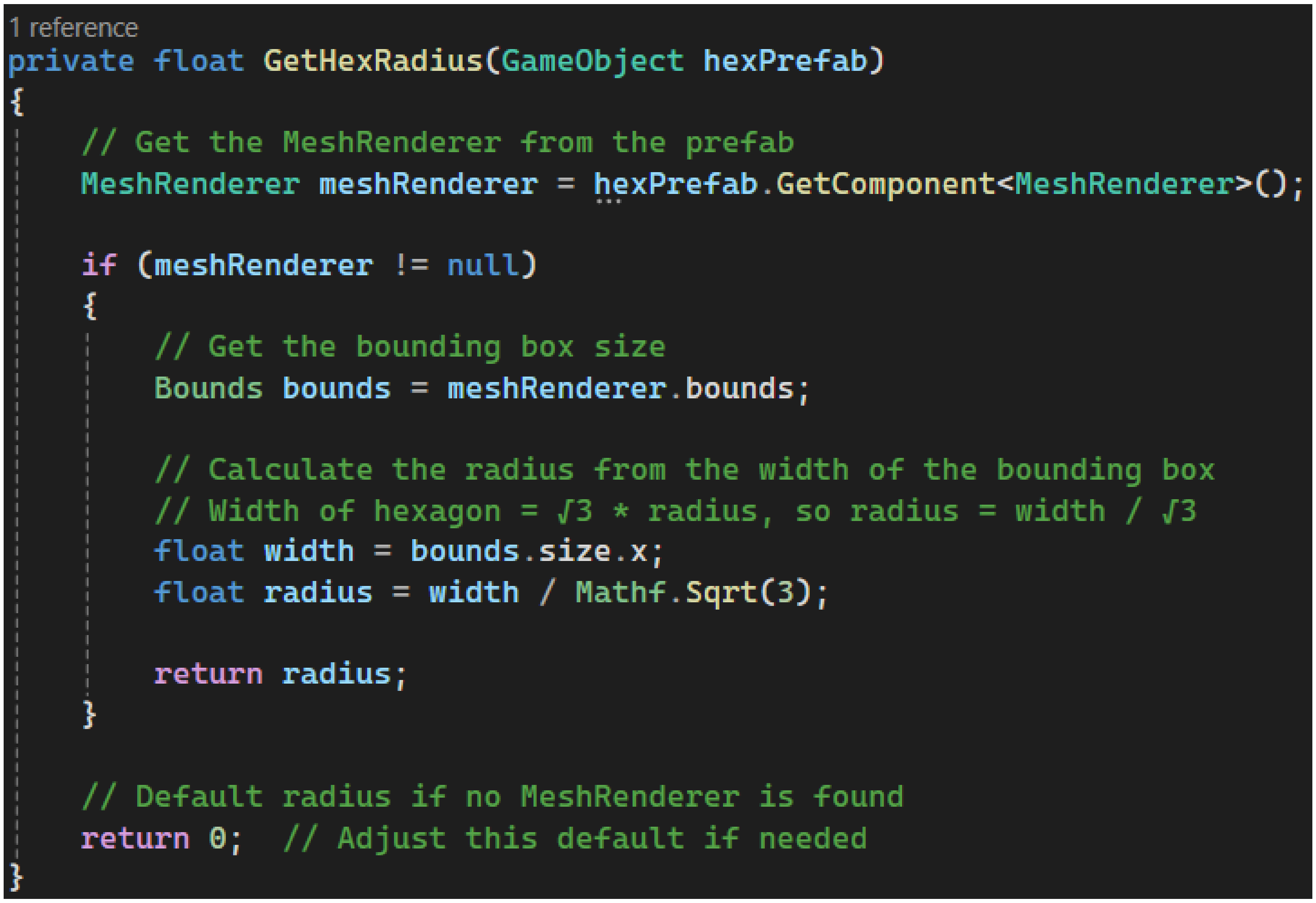
- **Materials:** I created **three semi-transparent materials** (red, green, and blue) to help visually distinguish between adjacent hexagons in the grid. This color-coding simplifies the user's ability to differentiate between individual tiles.

Scaling the Hexagon Prefab:

- After importing the hexagon asset, I found the default size too large for the AR scene. To address this, I **scaled the prefab down to 0.1** of its original size after running size tests in a simulation.

2. Function to Get the Hexagon Radius

- To ensure that hexagons fit properly within the boundaries of the plane, I needed to calculate the radius of the hexagons in world space. I wrote the following function to retrieve the radius from the prefab:



```
1 reference
private float GetHexRadius(GameObject hexPrefab)
{
    // Get the MeshRenderer from the prefab
    MeshRenderer meshRenderer = hexPrefab.GetComponent<MeshRenderer>();

    if (meshRenderer != null)
    {
        // Get the bounding box size
        Bounds bounds = meshRenderer.bounds;

        // Calculate the radius from the width of the bounding box
        // Width of hexagon = √3 * radius, so radius = width / √3
        float width = bounds.size.x;
        float radius = width / Mathf.Sqrt(3);

        return radius;
    }

    // Default radius if no MeshRenderer is found
    return 0;  // Adjust this default if needed
}
```

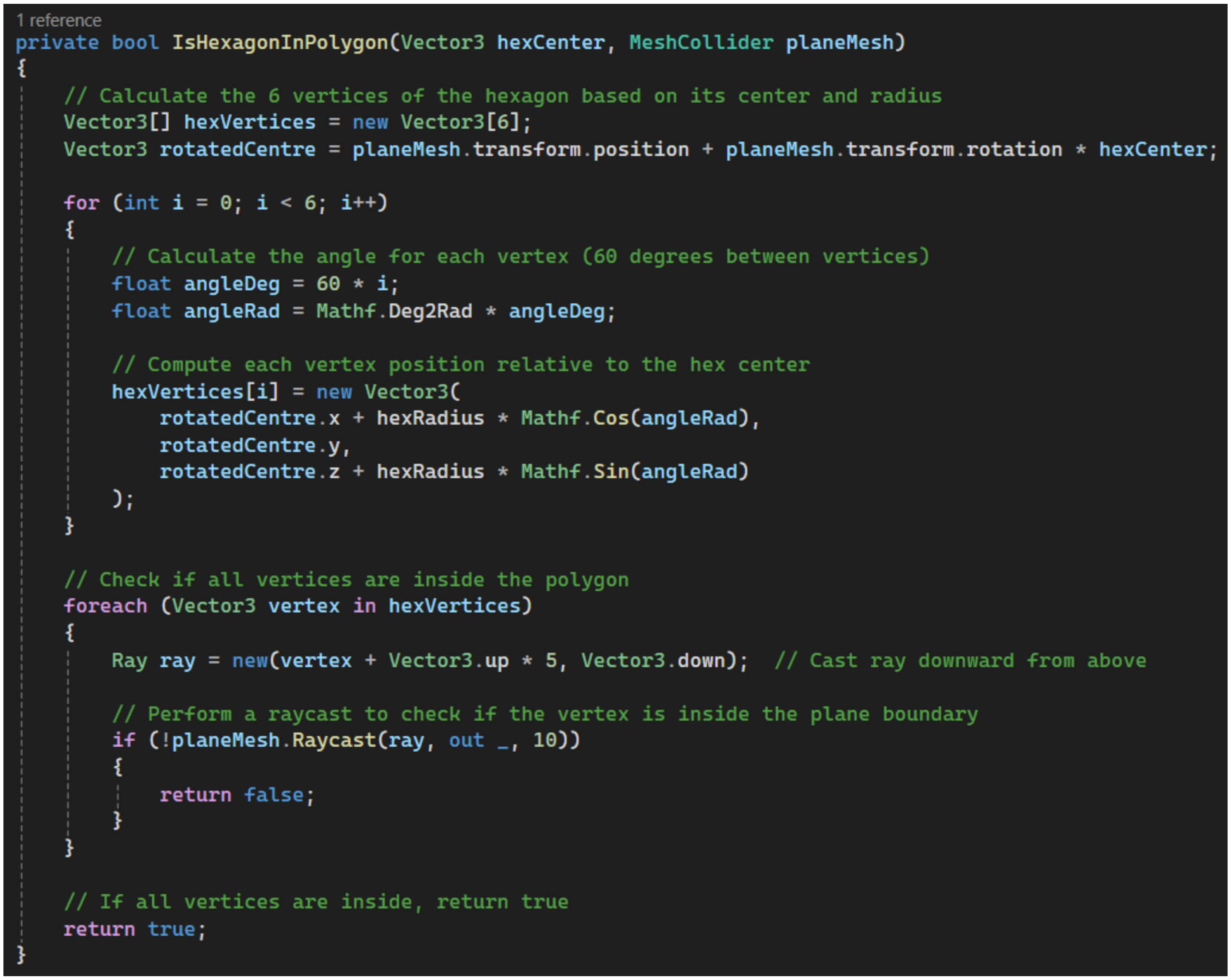
Explanation:

- This function calculates the **radius** of the hexagon based on its **bounding box width**, using the geometric formula for a hexagon's width: Width= √3 * Radius
- The function is vital for correctly spacing and positioning the hexagons within the plane.

3. Checking if a Hexagon Lies Completely Inside the AR Plane

- **Objective:** Ensure that each hexagon tile lies entirely within the plane's boundary before adding it to the grid.

The following function calculates the 6 vertices of the hexagon and checks if all vertices are inside the plane polygon:



```
1 reference
private bool IsHexagonInPolygon(Vector3 hexCenter, MeshCollider planeMesh)
{
    // Calculate the 6 vertices of the hexagon based on its center and radius
    Vector3[] hexVertices = new Vector3[6];
    Vector3 rotatedCentre = planeMesh.transform.position + planeMesh.transform.rotation * hexCenter;

    for (int i = 0; i < 6; i++)
    {
        // Calculate the angle for each vertex (60 degrees between vertices)
        float angleDeg = 60 * i;
        float angleRad = Mathf.Deg2Rad * angleDeg;

        // Compute each vertex position relative to the hex center
        hexVertices[i] = new Vector3(
            rotatedCentre.x + hexRadius * Mathf.Cos(angleRad),
            rotatedCentre.y,
            rotatedCentre.z + hexRadius * Mathf.Sin(angleRad)
        );
    }

    // Check if all vertices are inside the polygon
    foreach (Vector3 vertex in hexVertices)
    {
        Ray ray = new(vertex + Vector3.up * 5, Vector3.down);  // Cast ray downward from above

        // Perform a raycast to check if the vertex is inside the plane boundary
        if (!planeMesh.Raycast(ray, out _, 10))
        {
            return false;
        }
    }

    // If all vertices are inside, return true
    return true;
}
```

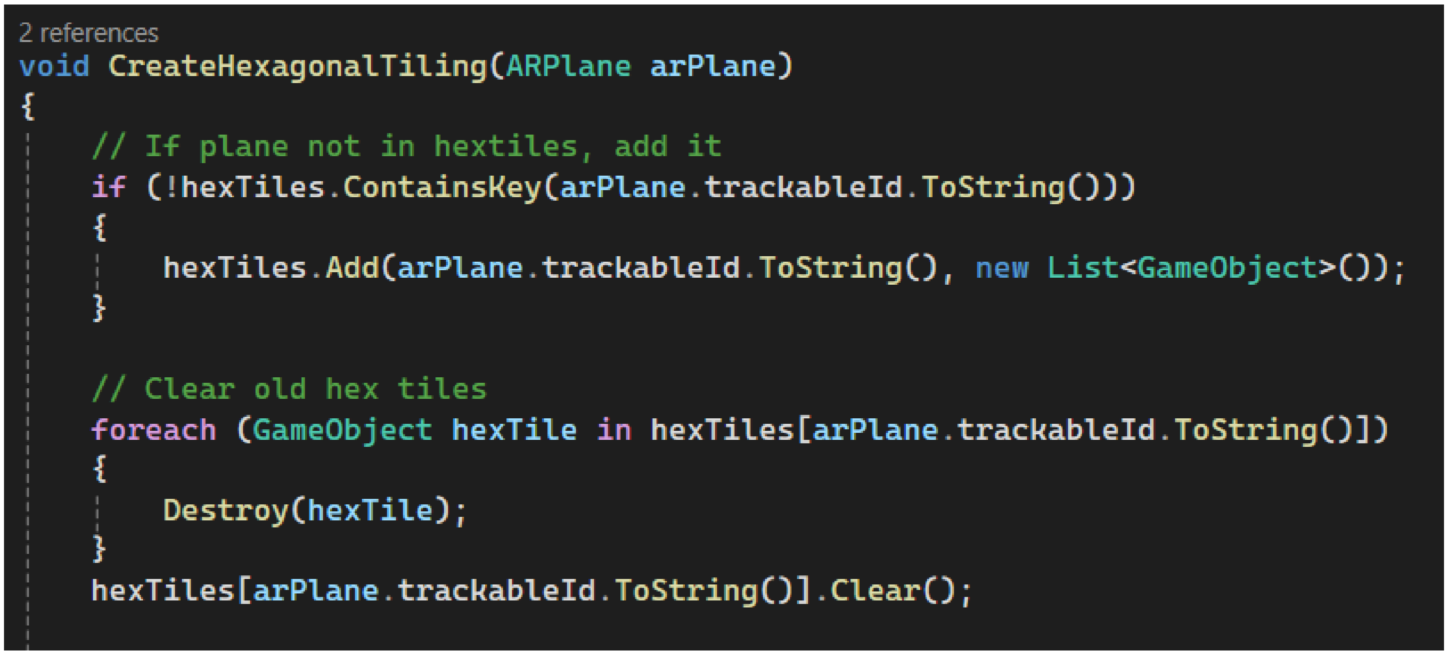
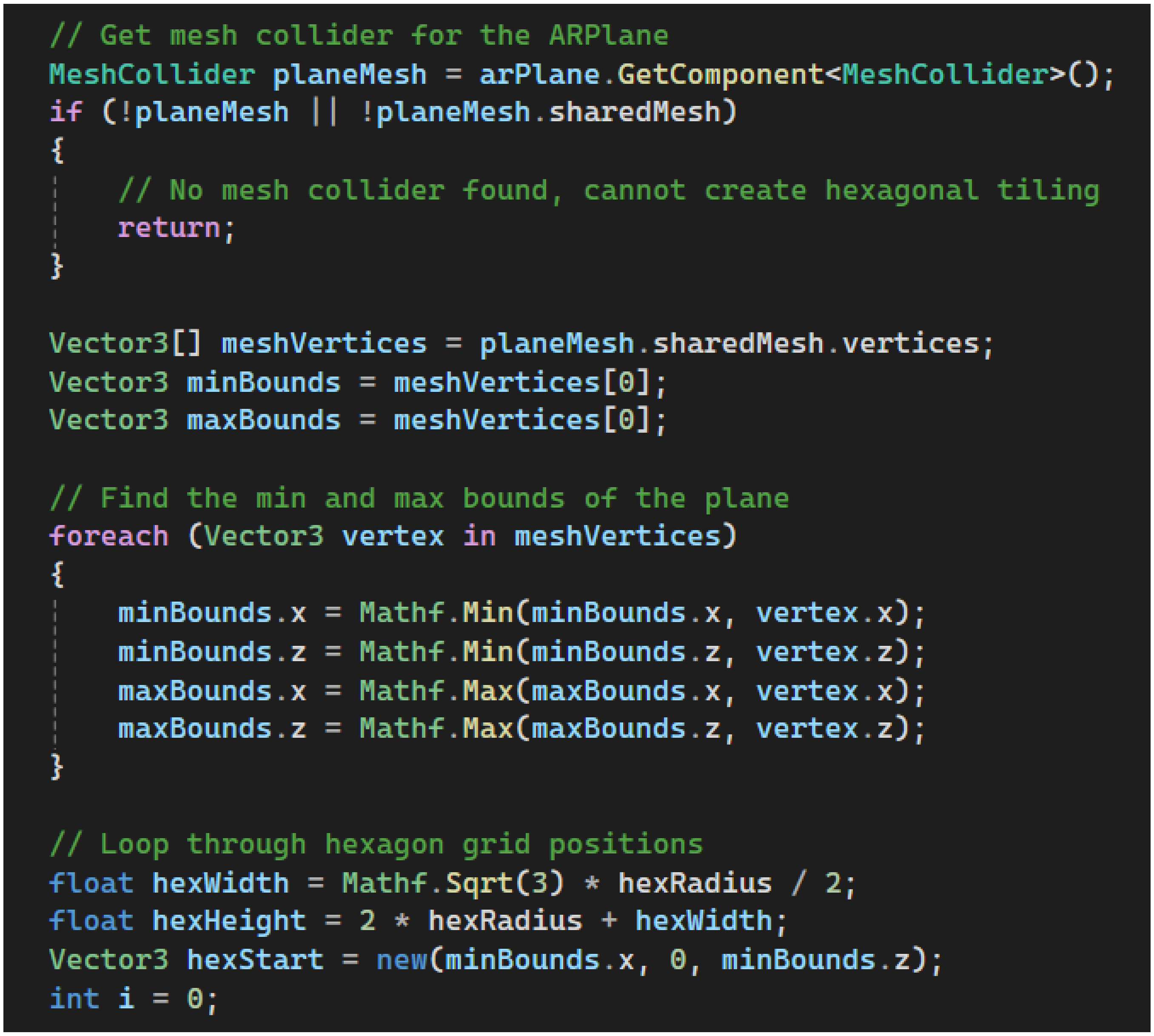
Explanation:

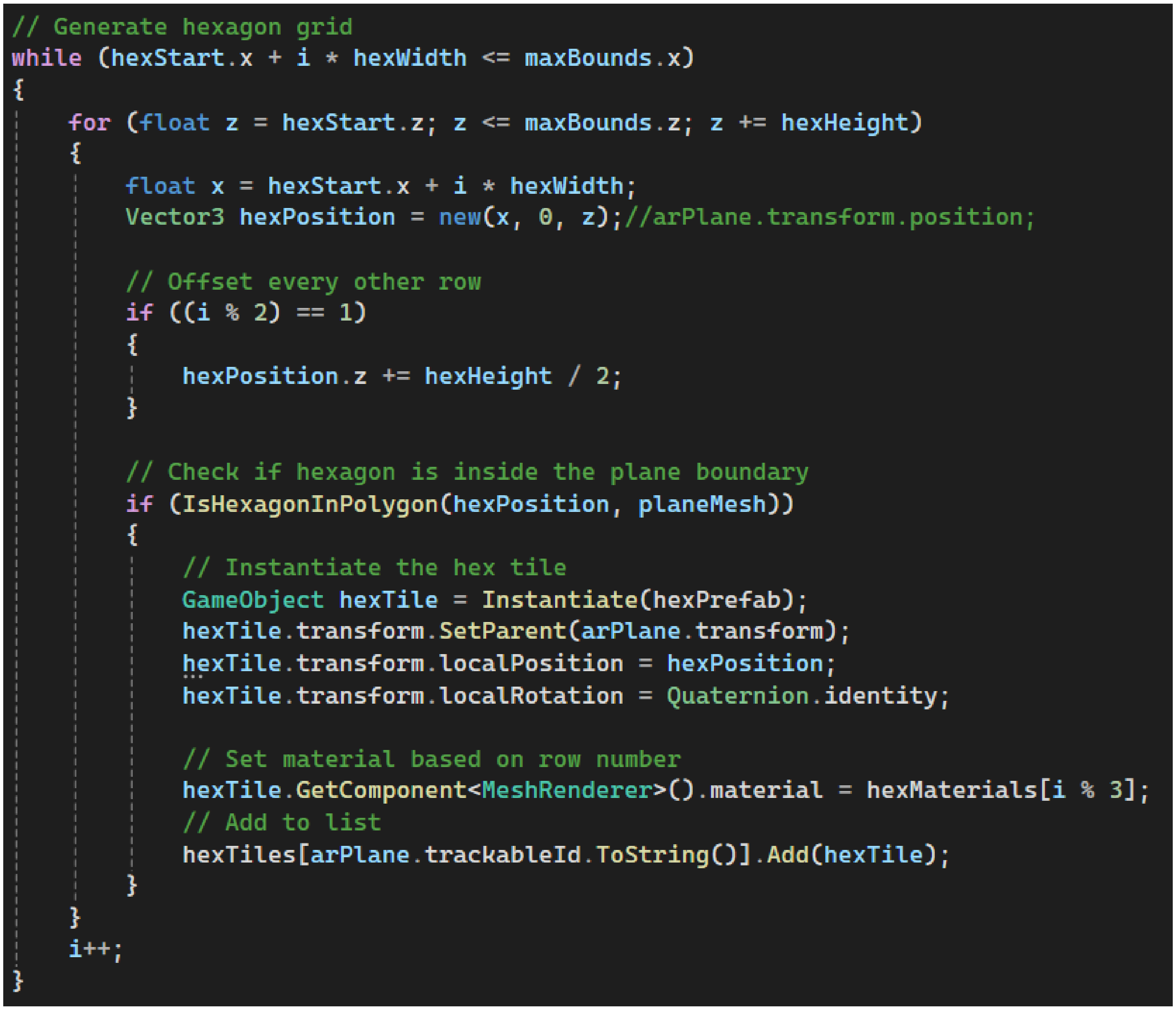
- This function calculates the 6 vertices of the hexagon based on its center and radius. It uses **raycasting** to check whether each vertex lies within the boundaries of the detected plane.
- If any vertex is outside the plane, the function returns false, preventing the placement of the hexagon.

4. Creating the Hexagonal Tiling

- **Objective:** Populate the detected AR planes with a hexagonal grid using the helper functions.

The following function creates a grid of hexagonal tiles for each AR plane:


```
2 references
void CreateHexagonalTiling(ARPlane arPlane)
{
    // If plane not in hextiles, add it
    if (!hexTiles.ContainsKey(arPlane.trackableId.ToString()))
    {
        hexTiles.Add(arPlane.trackableId.ToString(), new List<GameObject>());
    }

    // Clear old hex tiles
    foreach (GameObject hexTile in hexTiles[arPlane.trackableId.ToString()])
    {
        Destroy(hexTile);
    }
    hexTiles[arPlane.trackableId.ToString()].Clear();
```

```
    // Get mesh collider for the ARPlane
    MeshCollider planeMesh = arPlane.GetComponent<MeshCollider>();
    if (!planeMesh || !planeMesh.sharedMesh)
    {
        // No mesh collider found, cannot create hexagonal tiling
        return;
    }

    Vector3[] meshVertices = planeMesh.sharedMesh.vertices;
    Vector3 minBounds = meshVertices[0];
    Vector3 maxBounds = meshVertices[0];

    // Find the min and max bounds of the plane
    foreach (Vector3 vertex in meshVertices)
    {
        minBounds.x = Mathf.Min(minBounds.x, vertex.x);
        minBounds.z = Mathf.Min(minBounds.z, vertex.z);
        maxBounds.x = Mathf.Max(maxBounds.x, vertex.x);
        maxBounds.z = Mathf.Max(maxBounds.z, vertex.z);
    }

    // Loop through hexagon grid positions
    float hexWidth = Mathf.Sqrt(3) * hexRadius / 2;
    float hexHeight = 2 * hexRadius + hexWidth;
    Vector3 hexStart = new(minBounds.x, 0, minBounds.z);
    int i = 0;
```



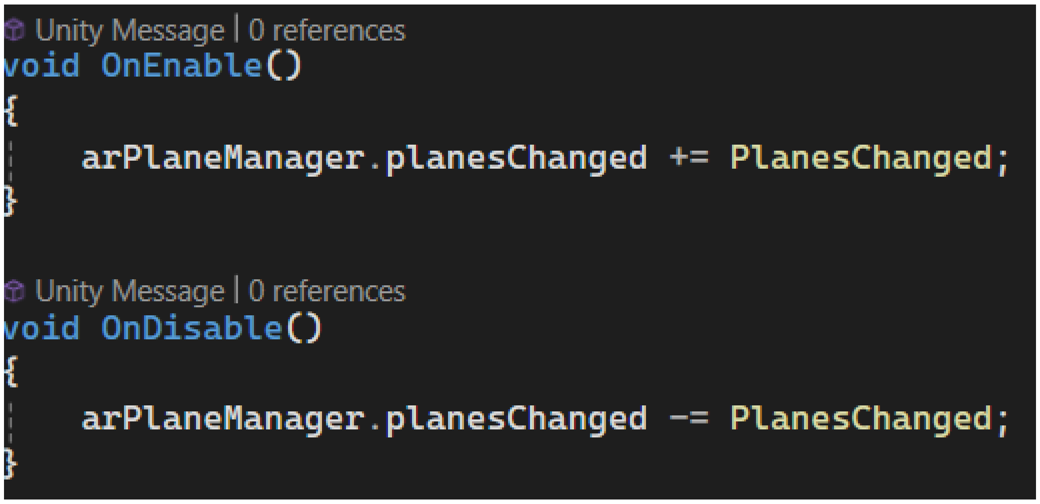
```
// Generate hexagon grid
while (hexStart.x + i * hexWidth <= maxBounds.x)
{
    for (float z = hexStart.z; z <= maxBounds.z; z += hexHeight)
    {
        float x = hexStart.x + i * hexWidth;
        Vector3 hexPosition = new(x, 0, z);//arPlane.transform.position;

        // Offset every other row
        if ((i % 2) == 1)
        {
            hexPosition.z += hexHeight / 2;
        }

        // Check if hexagon is inside the plane boundary
        if (IsHexagonInPolygon(hexPosition, planeMesh))
        {
            // Instantiate the hex tile
            GameObject hexTile = Instantiate(hexPrefab);
            hexTile.transform.SetParent(arPlane.transform);
            hexTile.transform.localPosition = hexPosition;
            hexTile.transform.localRotation = Quaternion.identity;

            // Set material based on row number
            hexTile.GetComponent<MeshRenderer>().material = hexMaterials[i % 3];
            // Add to list
            hexTiles[arPlane.trackableId.ToString()].Add(hexTile);
        }
    }
    i++;
}
```

Explanation:

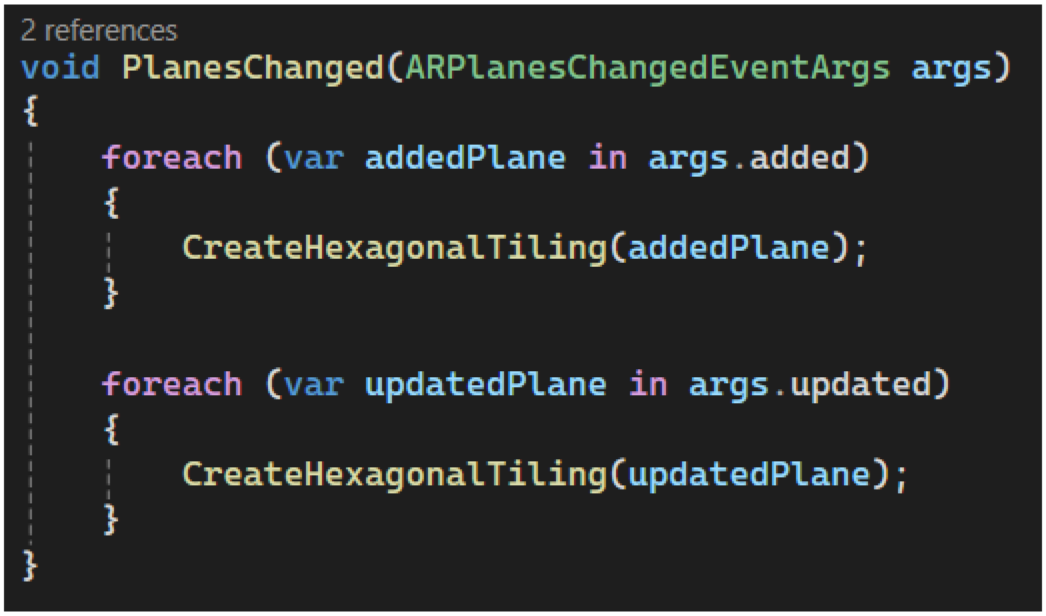
- This function generates a hexagonal grid over the AR plane by looping through potential hex positions within the plane's boundaries.
- The hexagonal gameobjects are stored in a list which is stored in a dictionary with the AR plane's ID as the key.
- The IsHexagonInPolygon function ensures that each hexagon tile is completely within the plane polygon before adding it to the scene.
- Each hex tile is instantiated, assigned a random material (to distinguish adjacent tiles), and stored in a dictionary for future reference.

5. Integrating Tiling with Plane Detection

- I integrated the hexagonal tiling function into the ARPlaneManager's **planesChanged** event to ensure that the hexagonal grid is updated whenever a new plane is detected or an existing plane changes.



```
Unity Message | 0 references
void OnEnable()
{
    arPlaneManager.planesChanged += PlanesChanged;
}

Unity Message | 0 references
void OnDisable()
{
    arPlaneManager.planesChanged -= PlanesChanged;
}
```



```
2 references
void PlanesChanged(ARPlanesChangedEventArgs args)
{
    foreach (var addedPlane in args.added)
    {
        CreateHexagonalTiling(addedPlane);
    }

    foreach (var updatedPlane in args.updated)
    {
        CreateHexagonalTiling(updatedPlane);
    }
}
```

Explanation:

- The planesChanged event triggers whenever AR planes are detected, added, or updated in the AR scene. I used this event to refresh the hexagonal tiling as the plane's shape changes over time.

6. Final Output

At the end of this task, the final output is a fully functioning hexagonal tiling system over detected AR planes with the following features:

- **Hexagonal Grid Tiling:** The AR system now detects planes in real time and dynamically overlays a hexagonal grid on top of them. The hexagons are color-coded (red, green, and blue) to easily differentiate between adjacent tiles. Only hexagons that completely fit inside the plane boundaries are added.

- **Continuous Plane Update:** As the AR planes change in shape or new planes are detected, the hexagonal grid is updated automatically. This ensures that the tiling remains accurate to the detected surfaces in real-time.
- **Tap Detection on Hexagons:** The raycasting system from Task 1 has been updated to detect taps on the hexagonal tiles. When a user taps a hexagon, the coordinates of the hexagon are displayed on the screen through a UI panel, showing the user the precise position of the tapped tile in world space.
- **Smooth User Experience:** The hexagons appear neatly tiled within the plane boundaries, with materials that offer a semi-transparent visual style. This enhances the AR experience by ensuring tiles are visually distinct and interactive.
- **Real-Time Visualization:** The AR application now visualizes the detected planes with a continuously updating hexagonal tiling, replacing the original raw polygon boundaries. This provides a more aesthetically pleasing and interactive way to represent AR planes.