# CHAPTER 3 - Processing

### 3.a – PlaySpaceManager.cs

|  |  |
| --- | --- |
| using System.Collections.Generic;  using UnityEngine;  using UnityEngine.Windows.Speech;  using HoloToolkit.Unity;  /// <summary>  /// The SurfaceManager class allows applications to scan the environment for a specified amount of time  /// and then process the Spatial Mapping Mesh (find planes, remove vertices) after that time has expired.  /// </summary>  public class PlaySpaceManager : Singleton<PlaySpaceManager>  {  [Tooltip("When checked, the SurfaceObserver will stop running after a specified amount of time.")]  public bool limitScanningByTime = true;  [Tooltip("How much time (in seconds) that the SurfaceObserver will run after being started; used when 'Limit Scanning By Time' is checked.")]  public float scanTime = 30.0f;  [Tooltip("Material to use when rendering Spatial Mapping meshes while the observer is running.")]  public Material defaultMaterial;  [Tooltip("Optional Material to use when rendering Spatial Mapping meshes after the observer has been stopped.")]  public Material secondaryMaterial;  [Tooltip("Minimum number of floor planes required in order to exit scanning/processing mode.")]  public uint minimumFloors = 1;  [Tooltip("Minimum number of wall planes required in order to exit scanning/processing mode.")]  public uint minimumWalls = 1;  /// <summary>  /// Indicates if processing of the surface meshes is complete.  /// </summary>  private bool meshesProcessed = false;  /// <summary>  /// GameObject initialization.  /// </summary>  private void Start()  {  // Update surfaceObserver and storedMeshes to use the same material during scanning.  SpatialMappingManager.Instance.SetSurfaceMaterial(defaultMaterial);  // Register for the MakePlanesComplete event.  SurfaceMeshesToPlanes.Instance.MakePlanesComplete += SurfaceMeshesToPlanes\_MakePlanesComplete;  }  /// <summary>  /// Called once per frame.  /// </summary>  private void Update()  {  // Check to see if the spatial mapping data has been processed  // and if we are limiting how much time the user can spend scanning.  if (!meshesProcessed && limitScanningByTime)  {  // If we have not processed the spatial mapping data  // and scanning time is limited...  // Check to see if enough scanning time has passed  // since starting the observer.  if (limitScanningByTime && ((Time.time - SpatialMappingManager.Instance.StartTime) < scanTime))  {  // If we have a limited scanning time, then we should wait until  // enough time has passed before processing the mesh.  }  else  {  // The user should be done scanning their environment,  // so start processing the spatial mapping data...  /\* TODO: 3.a DEVELOPER CODING EXERCISE 3.a \*/  // 3.a: Check if IsObserverRunning() is true on the  // SpatialMappingManager.Instace.  if(SpatialMappingManager.Instance.IsObserverRunning())  {  // 3.a: If running, Stop the observer by calling  // StopObserver() on the SpatialMappingManager.Instance.  SpatialMappingManager.Instance.StopObserver();  }  // 3.a: Call CreatePlanes() to generate planes.  CreatePlanes();  // 3.a: Set meshesProcessed to true.  meshesProcessed = true;  }  }  }  /// <summary>  /// Handler for the SurfaceMeshesToPlanes MakePlanesComplete event.  /// </summary>  /// <param name="source">Source of the event.</param>  /// <param name="args">Args for the event.</param>  private void SurfaceMeshesToPlanes\_MakePlanesComplete(object source, System.EventArgs args)  {  /\* TODO: 3.a DEVELOPER CODING EXERCISE 3.a \*/  // Collection of floor planes that we can use to set horizontal items on.  List<GameObject> floors = new List<GameObject>();  // Collection of wall planes that we can use to set vertical items on.  List<GameObject> walls = new List<GameObject>();  // 3.a: Get all floor planes by calling  // SurfaceMeshesToPlanes.Instance.GetActivePlanes().  // Assign the result to the 'floors' list.  floors = SurfaceMeshesToPlanes.Instance.GetActivePlanes(PlaneTypes.Floor);  // 3.a: Get all wall planes by calling  // SurfaceMeshesToPlanes.Instance.GetActivePlanes().  // Assign the result to the 'walls' list.  walls = SurfaceMeshesToPlanes.Instance.GetActivePlanes(PlaneTypes.Wall);  // Check to see if we have enough floors (minimumFloors)  // and walls (minimumWalls), to set holograms on in the world.  if (floors.Count >= minimumFloors && walls.Count >= minimumWalls)  {  // We have enough floors and walls to place our holograms on...  // 3.a: Let's reduce our triangle count by removing triangles  // from SpatialMapping meshes that intersect with our active planes.  // Call RemoveVertices().  // Pass in all activePlanes found by SurfaceMeshesToPlanes.Instance.  RemoveVertices(SurfaceMeshesToPlanes.Instance.ActivePlanes);  // 3.a: We can indicate to the user that scanning is over by  // changing the material applied to the Spatial Mapping meshes.  // Call SpatialMappingManager.Instance.SetSurfaceMaterial().  // Pass in the secondaryMaterial.  SpatialMappingManager.Instance.SetSurfaceMaterial(secondaryMaterial);  // 3.a: We are all done processing the mesh, so we can now  // initialize a collection of Placeable holograms in the world  // and use floor/wall planes to set thier starting positions.  // Call SpaceCollectionManager.Instance.GenerateItemsInWorld().  // Pass in the lists of floors and walls that we found earlier.  SpaceCollectionManager.Instance.GenerateItemsInWorld(floors, walls);  }  else  {  // We do not have enough floors/walls to place our holograms on...  // 3.a: Re-enter scanning mode so the user can find more surfaces by  // calling StartObserver() on the SpatialMappingManager.Instance.  SpatialMappingManager.Instance.StartObserver();  // 3.a: Re-process spatial data after scanning completes by  // re-setting meshesProcessed to false.  meshesProcessed = false;  }  }  /// <summary>  /// Creates planes from the spatial mapping surfaces.  /// </summary>  private void CreatePlanes()  {  // Generate planes based on the spatial map.  SurfaceMeshesToPlanes surfaceToPlanes = SurfaceMeshesToPlanes.Instance;  if (surfaceToPlanes != null && surfaceToPlanes.enabled)  {  surfaceToPlanes.MakePlanes();  }  }  /// <summary>  /// Removes triangles from the spatial mapping surfaces.  /// </summary>  /// <param name="boundingObjects"></param>  private void RemoveVertices(IEnumerable<GameObject> boundingObjects)  {  RemoveSurfaceVertices removeVerts = RemoveSurfaceVertices.Instance;  if (removeVerts != null && removeVerts.enabled)  {  removeVerts.RemoveSurfaceVerticesWithinBounds(boundingObjects);  }  }  /// <summary>  /// Called when the GameObject is unloaded.  /// </summary>  private void OnDestroy()  {  if (SurfaceMeshesToPlanes.Instance != null)  {  SurfaceMeshesToPlanes.Instance.MakePlanesComplete -= SurfaceMeshesToPlanes\_MakePlanesComplete;  }  }  } |  |

# CHAPTER 5 - Occlusion

### 5.a – PlanetOcclusion.cs

|  |  |
| --- | --- |
| using UnityEngine;  using HoloToolkit.Unity;  /// <summary>  /// Determines when the occluded version of the planet should be visible.  /// This script allows us to do selective occlusion, so the occlusionObject  /// will only be rendered when a Spatial Mapping surface is occluding the planet,  /// not when another hologram is responsible for the occlusion.  /// </summary>  public class PlanetOcclusion : MonoBehaviour  {  [Tooltip("Object to display when the planet is occluded.")]  public GameObject occlusionObject;  /// <summary>  /// Points to raycast to when checking for occlusion.  /// </summary>  private Vector3[] checkPoints;  // Use this for initialization  void Start()  {  occlusionObject.SetActive(false);  // Set the check points to use when testing for occlusion.  MeshFilter filter = gameObject.GetComponent<MeshFilter>();  Vector3 extents = filter.mesh.bounds.extents;  Vector3 center = filter.mesh.bounds.center;  Vector3 top = new Vector3(center.x, center.y + extents.y, center.z);  Vector3 left = new Vector3(center.x - extents.x, center.y, center.z);  Vector3 right = new Vector3(center.x + extents.x, center.y, center.z);  Vector3 bottom = new Vector3(center.x, center.y - extents.y, center.z);  checkPoints = new Vector3[] { center, top, left, right, bottom };  }  // Update is called once per frame  void Update()  {  /\* TODO: 5.a DEVELOPER CODING EXERCISE 5.a \*/  // Check to see if any of the planet's boundary points are occluded.  for (int i = 0; i < checkPoints.Length; i++)  {  // 5.a: Convert the current checkPoint to world coordinates.  // Call gameObject.transform.TransformPoint(checkPoints[i]).  // Assign the result to a new Vector3 variable called 'checkPt'.  Vector3 checkPt = gameObject.transform.TransformPoint(checkPoints[i]);  // 5.a: Call Vector3.Distance() to calculate the distance  // between the Main Camera's position and 'checkPt'.  // Assign the result to a new float variable called 'distance'.  float distance = Vector3.Distance(Camera.main.transform.position, checkPt);  // 5.a: Take 'checkPt' and subtract the Main Camera's position.  // Assign the result to a new Vector3 variable called 'direction'.  Vector3 direction = checkPt - Camera.main.transform.position;  // 5.a: Create a RaycastHit variable called 'hitInfo'.  RaycastHit hitInfo;  // Used to indicate if the call to Physics.Raycast() was successful.  bool raycastHit = false;  // 5.a: Check if the planet is occluded by a spatial mapping surface.  // Call Physics.Raycast() with the following arguments:  // - Pass in the Main Camera's position as the origin.  // - Pass in 'direction' for the direction.  // - Pass in 'hitInfo' for the out parameter.  // - Pass in 'distance' for the maxDistance.  // - Pass in SpatialMappingManager.Instance.LayerMask as layerMask.  // Assign the result to 'raycastHit'.  raycastHit = Physics.Raycast(Camera.main.transform.position, direction, out hitInfo, distance, SpatialMappingManager.Instance.LayerMask);  if (raycastHit)  {  // 5.a: Our raycast hit a surface, so the planet is occluded.  // Set the occlusionObject to active.  occlusionObject.SetActive(true);  // At least one point is occluded, so break from the loop.  break;  }  else  {  // 5.a: The Raycast did not hit, so the planet is not occluded.  // Deactivate the occlusionObject.  occlusionObject.SetActive(false);  }  }  }  } |  |