# CHAPTER 2

### 2.a – HandsManager.cs:

|  |  |
| --- | --- |
| using HoloToolkit;  using UnityEngine.VR.WSA.Input;  using UnityEngine;  /// <summary>  /// HandsManager keeps track of when a hand is detected.  /// </summary>  public class HandsManager : Singleton<HandsManager>  {  [Tooltip("Audio clip to play when Finger Pressed.")]  public AudioClip FingerPressedSound;  private AudioSource audioSource;  /// <summary>  /// Tracks the hand detected state.  /// </summary>  public bool HandDetected  {  get;  private set;  }  // Keeps track of the GameObject that the hand is interacting with.  public GameObject FocusedGameObject { get; private set; }  void Awake()  {  EnableAudioHapticFeedback();  InteractionManager.SourceDetected += InteractionManager\_SourceDetected;  InteractionManager.SourceLost += InteractionManager\_SourceLost;  /\* TODO: DEVELOPER CODE ALONG 2.a \*/  // 2.a: Register for SourceManager.SourcePressed event.  InteractionManager.SourcePressed += InteractionManager\_SourcePressed;  // 2.a: Register for SourceManager.SourceReleased event.  InteractionManager.SourceReleased += InteractionManager\_SourceReleased;  // 2.a: Initialize FocusedGameObject as null.  FocusedGameObject = null;  }  private void EnableAudioHapticFeedback()  {  // If this hologram has an audio clip, add an AudioSource with this clip.  if (FingerPressedSound != null)  {  audioSource = GetComponent<AudioSource>();  if (audioSource == null)  {  audioSource = gameObject.AddComponent<AudioSource>();  }  audioSource.clip = FingerPressedSound;  audioSource.playOnAwake = false;  audioSource.spatialBlend = 1;  audioSource.dopplerLevel = 0;  }  }  private void InteractionManager\_SourceDetected(InteractionSourceState hand)  {  HandDetected = true;  }  private void InteractionManager\_SourceLost(InteractionSourceState hand)  {  HandDetected = false;  // 2.a: Reset FocusedGameObject.  ResetFocusedGameObject();  }  private void InteractionManager\_SourcePressed(InteractionSourceState hand)  {  if (InteractibleManager.Instance.FocusedGameObject != null)  {  // Play a select sound if we have an audio source and are not targetting 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();  }  // 2.a: Cache InteractibleManager's FocusedGameObject in FocusedGameObject.  FocusedGameObject = InteractibleManager.Instance.FocusedGameObject;  }  }  private void InteractionManager\_SourceReleased(InteractionSourceState hand)  {  // 2.a: Reset FocusedGameObject.  ResetFocusedGameObject();  }  private void ResetFocusedGameObject()  {  // 2.a: Set FocusedGameObject to be null.  FocusedGameObject = null;  // 2.a: On GestureManager call ResetGestureRecognizers  // to complete any currently active gestures.  GestureManager.Instance.ResetGestureRecognizers();  }  void OnDestroy()  {  InteractionManager.SourceDetected -= InteractionManager\_SourceDetected;  InteractionManager.SourceLost -= InteractionManager\_SourceLost;  // 2.a: Unregister the SourceManager.SourceReleased event.  InteractionManager.SourceReleased -= InteractionManager\_SourceReleased;  // 2.a: Unregister for SourceManager.SourcePressed event.  InteractionManager.SourcePressed -= InteractionManager\_SourcePressed;  }  } |  |

### 2.b – GestureManager.cs:

|  |  |
| --- | --- |
| using HoloToolkit;  using UnityEngine;  using UnityEngine.VR.WSA.Input;  public class GestureManager : Singleton<GestureManager>  {  // Tap and Navigation gesture recognizer.  public GestureRecognizer NavigationRecognizer { get; private set; }  // Manipulation gesture recognizer.  public GestureRecognizer ManipulationRecognizer { get; private set; }  // Currently active gesture recognizer.  public GestureRecognizer ActiveRecognizer { get; private set; }  public bool IsNavigating { get; private set; }  public Vector3 NavigationPosition { get; private set; }  public bool IsManipulating { get; private set; }  public Vector3 ManipulationPosition { get; private set; }  void Awake()  {  /\* TODO: DEVELOPER CODING EXERCISE 2.b \*/  // 2.b: Instantiate the NavigationRecognizer.  NavigationRecognizer = new GestureRecognizer();  // 2.b: Add Tap and NavigationX GestureSettings to the NavigationRecognizer's RecognizableGestures.  NavigationRecognizer.SetRecognizableGestures(  GestureSettings.Tap |  GestureSettings.NavigationX);  // 2.b: Register for the TappedEvent with the NavigationRecognizer\_TappedEvent function.  NavigationRecognizer.TappedEvent += NavigationRecognizer\_TappedEvent;  // 2.b: Register for the NavigationStartedEvent with the NavigationRecognizer\_NavigationStartedEvent function.  NavigationRecognizer.NavigationStartedEvent += NavigationRecognizer\_NavigationStartedEvent;  // 2.b: Register for the NavigationUpdatedEvent with the NavigationRecognizer\_NavigationUpdatedEvent function.  NavigationRecognizer.NavigationUpdatedEvent += NavigationRecognizer\_NavigationUpdatedEvent;  // 2.b: Register for the NavigationCompletedEvent with the NavigationRecognizer\_NavigationCompletedEvent function.  NavigationRecognizer.NavigationCompletedEvent += NavigationRecognizer\_NavigationCompletedEvent;  // 2.b: Register for the NavigationCanceledEvent with the NavigationRecognizer\_NavigationCanceledEvent function.  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();  }  void OnDestroy()  {  // 2.b: Unregister the Tapped and Navigation events on the NavigationRecognizer.  NavigationRecognizer.TappedEvent -= NavigationRecognizer\_TappedEvent;  NavigationRecognizer.NavigationStartedEvent -= NavigationRecognizer\_NavigationStartedEvent;  NavigationRecognizer.NavigationUpdatedEvent -= NavigationRecognizer\_NavigationUpdatedEvent;  NavigationRecognizer.NavigationCompletedEvent -= NavigationRecognizer\_NavigationCompletedEvent;  NavigationRecognizer.NavigationCanceledEvent -= NavigationRecognizer\_NavigationCanceledEvent;  // Unregister the Manipulation events on the ManipulationRecognizer.  ManipulationRecognizer.ManipulationStartedEvent -= ManipulationRecognizer\_ManipulationStartedEvent;  ManipulationRecognizer.ManipulationUpdatedEvent -= ManipulationRecognizer\_ManipulationUpdatedEvent;  ManipulationRecognizer.ManipulationCompletedEvent -= ManipulationRecognizer\_ManipulationCompletedEvent;  ManipulationRecognizer.ManipulationCanceledEvent -= ManipulationRecognizer\_ManipulationCanceledEvent;  }  /// <summary>  /// Revert back to the default GestureRecognizer.  /// </summary>  public void ResetGestureRecognizers()  {  // Default to the navigation gestures.  Transition(NavigationRecognizer);  }  /// <summary>  /// Transition to a new GestureRecognizer.  /// </summary>  /// <param name="newRecognizer">The GestureRecognizer to transition to.</param>  public void Transition(GestureRecognizer newRecognizer)  {  if (newRecognizer == null)  {  return;  }  if (ActiveRecognizer != null)  {  if (ActiveRecognizer == newRecognizer)  {  return;  }  ActiveRecognizer.CancelGestures();  ActiveRecognizer.StopCapturingGestures();  }  newRecognizer.StartCapturingGestures();  ActiveRecognizer = newRecognizer;  }  private void NavigationRecognizer\_NavigationStartedEvent(InteractionSourceKind source, Vector3 relativePosition, Ray ray)  {  // 2.b: Set IsNavigating to be true.  IsNavigating = true;  // 2.b: Set NavigationPosition to be relativePosition.  NavigationPosition = relativePosition;  }  private void NavigationRecognizer\_NavigationUpdatedEvent(InteractionSourceKind source, Vector3 relativePosition, Ray ray)  {  // 2.b: Set IsNavigating to be true.  IsNavigating = true;  // 2.b: Set NavigationPosition to be relativePosition.  NavigationPosition = relativePosition;  }  private void NavigationRecognizer\_NavigationCompletedEvent(InteractionSourceKind source, Vector3 relativePosition, Ray ray)  {  // 2.b: Set IsNavigating to be false.  IsNavigating = false;  }  private void NavigationRecognizer\_NavigationCanceledEvent(InteractionSourceKind source, Vector3 relativePosition, Ray ray)  {  // 2.b: Set IsNavigating to be false.  IsNavigating = false;  }  private void ManipulationRecognizer\_ManipulationStartedEvent(InteractionSourceKind source, Vector3 position, Ray ray)  {  if (HandsManager.Instance.FocusedGameObject != null)  {  IsManipulating = true;  ManipulationPosition = position;  HandsManager.Instance.FocusedGameObject.SendMessageUpwards("PerformManipulationStart", position);  }  }  private void ManipulationRecognizer\_ManipulationUpdatedEvent(InteractionSourceKind source, Vector3 position, Ray ray)  {  if (HandsManager.Instance.FocusedGameObject != null)  {  IsManipulating = true;  ManipulationPosition = position;  HandsManager.Instance.FocusedGameObject.SendMessageUpwards("PerformManipulationUpdate", position);  }  }  private void ManipulationRecognizer\_ManipulationCompletedEvent(InteractionSourceKind source, Vector3 position, Ray ray)  {  IsManipulating = false;  }  private void ManipulationRecognizer\_ManipulationCanceledEvent(InteractionSourceKind source, Vector3 position, Ray ray)  {  IsManipulating = false;  }  private void NavigationRecognizer\_TappedEvent(InteractionSourceKind source, int tapCount, Ray ray)  {  GameObject focusedObject = InteractibleManager.Instance.FocusedGameObject;  if (focusedObject != null)  {  focusedObject.SendMessageUpwards("OnSelect");  }  }  } |  |

### 2.c – GestureAction.cs:

|  |  |
| --- | --- |
| using UnityEngine;  /// <summary>  /// GestureAction performs custom actions based on  /// which gesture is being performed.  /// </summary>  public class GestureAction : MonoBehaviour  {  [Tooltip("Rotation max speed controls amount of rotation.")]  public float RotationSensitivity = 10.0f;  private Vector3 manipulationPreviousPosition;  private float rotationFactor;  void Update()  {  PerformRotation();  }  private void PerformRotation()  {  if (GestureManager.Instance.IsNavigating &&  (!ExpandModel.Instance.IsModelExpanded ||  (ExpandModel.Instance.IsModelExpanded && HandsManager.Instance.FocusedGameObject == gameObject)))  {  /\* TODO: DEVELOPER CODING EXERCISE 2.c \*/  // 2.c: Calculate rotationFactor based on GestureManager's NavigationPosition.X and multiply by RotationSensitivity.  // This will help control the amount of rotation.  rotationFactor = GestureManager.Instance.NavigationPosition.x \* RotationSensitivity;  // 2.c: transform.Rotate along the Y axis using rotationFactor.  transform.Rotate(new Vector3(0, -1 \* rotationFactor, 0));  }  }  void PerformManipulationStart(Vector3 position)  {  manipulationPreviousPosition = position;  }  void PerformManipulationUpdate(Vector3 position)  {  if (GestureManager.Instance.IsManipulating)  {  /\* TODO: DEVELOPER CODING EXERCISE 4.a \*/  Vector3 moveVector = Vector3.zero;  // 4.a: Calculate the moveVector as position - manipulationPreviousPosition.  // 4.a: Update the manipulationPreviousPosition with the current position.  // 4.a: Increment this transform's position by the moveVector.  }  }  } |  |

# CHAPTER 4

### 4.a – GestureAction.cs:

|  |
| --- |
| using UnityEngine;  /// <summary>  /// GestureAction performs custom actions based on  /// which gesture is being performed.  /// </summary>  public class GestureAction : MonoBehaviour  {  [Tooltip("Rotation max speed controls amount of rotation.")]  public float RotationSensitivity = 10.0f;  private Vector3 manipulationPreviousPosition;  private float rotationFactor;  void Update()  {  PerformRotation();  }  private void PerformRotation()  {  if (GestureManager.Instance.IsNavigating &&  (!ExpandModel.Instance.IsModelExpanded ||  (ExpandModel.Instance.IsModelExpanded && HandsManager.Instance.FocusedGameObject == gameObject)))  {  /\* TODO: DEVELOPER CODING EXERCISE 2.c \*/  // 2.c: Calculate rotationFactor based on GestureManager's NavigationPosition.X and multiply by RotationSensitivity.  // This will help control the amount of rotation.  rotationFactor = GestureManager.Instance.NavigationPosition.x \* RotationSensitivity;  // 2.c: transform.Rotate along the Y axis using rotationFactor.  transform.Rotate(new Vector3(0, -1 \* rotationFactor, 0));  }  }  void PerformManipulationStart(Vector3 position)  {  manipulationPreviousPosition = position;  }  void PerformManipulationUpdate(Vector3 position)  {  if (GestureManager.Instance.IsManipulating)  {  /\* TODO: DEVELOPER CODING EXERCISE 4.a \*/  Vector3 moveVector = Vector3.zero;  // 4.a: Calculate the moveVector as position - manipulationPreviousPosition.  moveVector = position - manipulationPreviousPosition;  // 4.a: Update the manipulationPreviousPosition with the current position.  manipulationPreviousPosition = position;  // 4.a: Increment this transform's position by the moveVector.  transform.position += moveVector;  }  }  } |

# Chapter 5

### 5.a – AstronautManager.cs:

|  |
| --- |
| using HoloToolkit;  using System.Collections.Generic;  using System.Linq;  using UnityEngine;  using UnityEngine.WSA.Speech;  public class AstronautManager : Singleton<AstronautManager>  {  float expandAnimationCompletionTime;  // Store a bool for whether our astronaut model is expanded or not.  bool isModelExpanding = false;  // KeywordRecognizer object.  KeywordRecognizer keywordRecognizer;  // Defines which function to call when a keyword is recognized.  delegate void KeywordAction(PhraseRecognizedEventArgs args);  Dictionary<string, KeywordAction> keywordCollection;  void Start()  {  keywordCollection = new Dictionary<string, KeywordAction>();  // Add keyword to start manipulation.  keywordCollection.Add("Move Astronaut", MoveAstronautCommand);  /\* TODO: DEVELOPER CODING EXERCISE 5.a \*/  // 5.a: Add keyword Expand Model to call the ExpandModelCommand function.  keywordCollection.Add("Expand Model", ExpandModelCommand);  // 5.a: Add keyword Reset Model to call the ResetModelCommand function.  keywordCollection.Add("Reset Model", ResetModelCommand);  // Initialize KeywordRecognizer with the previously added keywords.  keywordRecognizer = new KeywordRecognizer(keywordCollection.Keys.ToArray());  keywordRecognizer.OnPhraseRecognized += KeywordRecognizer\_OnPhraseRecognized;  keywordRecognizer.Start();  }  private void KeywordRecognizer\_OnPhraseRecognized(PhraseRecognizedEventArgs args)  {  KeywordAction keywordAction;  if (keywordCollection.TryGetValue(args.text, out keywordAction))  {  keywordAction.Invoke(args);  }  }  private void MoveAstronautCommand(PhraseRecognizedEventArgs args)  {  GestureManager.Instance.Transition(GestureManager.Instance.ManipulationRecognizer);  }  private void ResetModelCommand(PhraseRecognizedEventArgs args)  {  // Reset local variables.  isModelExpanding = false;  // Disable the expanded model.  ExpandModel.Instance.ExpandedModel.SetActive(false);  // Enable the idle model.  ExpandModel.Instance.gameObject.SetActive(true);  // Enable the animators for the next time the model is expanded.  Animator[] expandedAnimators = ExpandModel.Instance.ExpandedModel.GetComponentsInChildren<Animator>();  foreach (Animator animator in expandedAnimators)  {  animator.enabled = true;  }  ExpandModel.Instance.Reset();  }  private void ExpandModelCommand(PhraseRecognizedEventArgs args)  {  // Swap out the current model for the expanded model.  GameObject currentModel = ExpandModel.Instance.gameObject;  ExpandModel.Instance.ExpandedModel.transform.position = currentModel.transform.position;  ExpandModel.Instance.ExpandedModel.transform.rotation = currentModel.transform.rotation;  ExpandModel.Instance.ExpandedModel.transform.localScale = currentModel.transform.localScale;  currentModel.SetActive(false);  ExpandModel.Instance.ExpandedModel.SetActive(true);  // Play animation. Ensure the Loop Time check box is disabled in the inspector for this animation to play it once.  Animator[] expandedAnimators = ExpandModel.Instance.ExpandedModel.GetComponentsInChildren<Animator>();  // Set local variables for disabling the animation.  if (expandedAnimators.Length > 0)  {  expandAnimationCompletionTime = Time.realtimeSinceStartup + expandedAnimators[0].runtimeAnimatorController.animationClips[0].length \* 0.9f;  }  // Set the expand model flag.  isModelExpanding = true;  ExpandModel.Instance.Expand();  }  public void Update()  {  if (isModelExpanding && Time.realtimeSinceStartup >= expandAnimationCompletionTime)  {  isModelExpanding = false;  Animator[] expandedAnimators = ExpandModel.Instance.ExpandedModel.GetComponentsInChildren<Animator>();  foreach (Animator animator in expandedAnimators)  {  animator.enabled = false;  }  }  }  } |