```
// Licensed under the MIT License. See LICENSE in the project root for
license information.

// Licensed under the MIT License. See LICENSE in the project root for
license information.

using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using OpenAI.Audio;
using OpenAI.Chat;
using OpenAI.Images;
using OpenAI.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading;
using System.Threading.Tasks;
using TMPro;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;
using Utilities.Audio;
using Utilities.Encoding.Wav;
using Utilities.Extensions;
using Utilities.WebRequestRest;

namespace OpenAI.Samples.Chat
{
    public class ChatBehaviour : MonoBehaviour
    {

        [SerializeField]
        private bool enableDebug;

        [SerializeField]
        private Button submitButton;

        [SerializeField]
        private Button recordButton;

        [SerializeField]
        public TMP_InputField inputField;

        [SerializeField]
        private RectTransform contentArea;

        [SerializeField]
```

```
        public ScrollRect scrollView;

        [SerializeField]
        private AudioSource audioSource;

        [SerializeField]
        [TextArea(3, 10)]
        private string systemPrompt = "You are a helpful AI debugging
assistant that helps me interface and understand my code with the
Reflection library.\n- If an image is requested then use
\"![Image](output.jpg)\" to display it.";

        private OpenAIClient openAI;

        public Conversation conversation = new Conversation();

        private CancellationTokenSource lifetimeCancellationTokenSource;

        private readonly List<Tool> assistantTools = new List<Tool>
        {
            new Function(
                nameof(GenerateImageAsync),
                "Generates an image based on the user's request.",
                new JObject
                {
                    ["type"] = "object",
                    ["properties"] = new JObject
                    {
                        ["prompt"] = new JObject
                        {
                            ["type"] = "string",
                            ["description"] = "A text description of the
desired image(s). The maximum length is 1000 characters for dall-e-2 and
4000 characters for dall-e-3."
                        },
                        ["model"] = new JObject
                        {
                            ["type"] = "string",
                            ["description"] = "The model to use for
image generation.",
                            ["enum"] = new JArray { "dall-e-2", "dall-e-
3" },
                            ["default"] = "dall-e-2"
                        },
                        ["size"] = new JObject
                        {
                            ["type"] = "string",
                            ["description"] = "The size of the generated
```

```
images. Must be one of 256x256, 512x512, or 1024x1024 for dall-e-2. Must
be one of 1024x1024, 1792x1024, or 1024x1792 for dall-e-3 models.",
                            ["enum"] = new JArray{ "256x256", "512x512",
"1024x1024", "1792x1024", "1024x1792" },
                            ["default"] = "512x512"
                        },
                        ["response_format"] = new JObject
                        {
                            ["type"] = "string",
                            ["enum"] = new JArray { "b64_json" } // hard
coded for webgl
                        }
                    },
                    ["required"] = new JArray { "prompt", "model",
"response_format" }
                })
        };

        private void OnValidate()
        {
            inputField.Validate();
            contentArea.Validate();
            submitButton.Validate();
            recordButton.Validate();
            audioSource.Validate();
        }

        private void Awake()
        {
            OnValidate();
            lifetimeCancellationTokenSource = new
CancellationTokenSource();
            openAI = new OpenAIClient
            {
                EnableDebug = enableDebug
            };
            conversation.AppendMessage(new Message(Role.System,
systemPrompt));
            inputField.onSubmit.AddListener(SubmitChat);
            //submitButton.onClick.AddListener(SubmitChat);
            recordButton.onClick.AddListener(ToggleRecording);
        }

        private void OnDestroy()
        {
            lifetimeCancellationTokenSource.Cancel();
            lifetimeCancellationTokenSource.Dispose();
            lifetimeCancellationTokenSource = null;
```

```
        }


        public void SubmitChat(string _) => SubmitChat();

        private static bool isChatPending;

        public void UpdateChat(string newText)
        {
            conversation.AppendMessage(new Message(Role.Assistant,
newText));
            //inputField.text = newText;
            var assistantMessageContent =
AddNewTextMessageContent(Role.Assistant);
            assistantMessageContent.text = newText;
            scrollView.verticalNormalizedPosition = 0f;

        }



        private async void SubmitChat()
        {
            if (isChatPending ||
string.IsNullOrWhiteSpace(inputField.text)) { return; }
            isChatPending = true;


            inputField.ReleaseSelection();
            inputField.interactable = false;
            submitButton.interactable = false;
            conversation.AppendMessage(new Message(Role.User,
inputField.text));
            var userMessageContent =
AddNewTextMessageContent(Role.User);
            userMessageContent.text = $"User: {inputField.text}";
            inputField.text = string.Empty;
            var assistantMessageContent =
AddNewTextMessageContent(Role.Assistant);
            assistantMessageContent.text = "Assistant: ";

            try
            {
                var request = new ChatRequest(conversation.Messages,
tools: assistantTools, toolChoice: "auto");
                var response = await
openAI.ChatEndpoint.StreamCompletionAsync(request, resultHandler:
deltaResponse =>
```

```
                {
                    if (deltaResponse?.FirstChoice?.Delta == null) {
return; }
                    assistantMessageContent.text +=
deltaResponse.FirstChoice.Delta.ToString();
                    scrollView.verticalNormalizedPosition = 0f;
                }, lifetimeCancellationTokenSource.Token);

conversation.AppendMessage(response.FirstChoice.Message);
if (response.FirstChoice.FinishReason == "tool_calls")
                {
                    response = await ProcessToolCallAsync(response);
                    assistantMessageContent.text +=
response.ToString().Replace("![Image](output.jpg)", string.Empty);
                }

                GenerateSpeech(response);
            }
            catch (Exception e)
            {
                switch (e)
                {
                    case TaskCanceledException:
                    case OperationCanceledException:
                        break;
                    default:
                        Debug.LogError(e);
                        break;
                }
            }
            finally
            {
                if (lifetimeCancellationTokenSource is {
IsCancellationRequested: false })
                {
                    inputField.interactable = true;
EventSystem.current.SetSelectedGameObject(inputField.gameObject);
            submitButton.interactable = true;
                }

                isChatPending = false;
            }

            async Task<ChatResponse> ProcessToolCallAsync(ChatResponse
response)
            {
                var toolCall =
response.FirstChoice.Message.ToolCalls.FirstOrDefault();
```

```
                if (enableDebug)
                {
                    Debug.Log($"{response.FirstChoice.Message.Role}:
{toolCall?.Function?.Name} | Finish Reason:
{response.FirstChoice.FinishReason}");
                    Debug.Log($"{toolCall?.Function?.Arguments}");
                }

                if (toolCall == null || toolCall.Function?.Name !=
nameof(GenerateImageAsync))
                {
                    throw new Exception($"Failed to find a valid tool
call!\n{response}");
                }

                ChatResponse toolCallResponse;

                try
                {
                    var imageGenerationRequest =
JsonConvert.DeserializeObject<ImageGenerationRequest>(toolCall.Function.
Arguments.ToString());
                    var imageResult = await
GenerateImageAsync(imageGenerationRequest);
                    AddNewImageContent(imageResult);
                    conversation.AppendMessage(new Message(toolCall,
"{\"result\":\"completed\"}"));
                    var toolCallRequest = new
ChatRequest(conversation.Messages, tools: assistantTools, toolChoice:
"auto");
                    toolCallResponse = await
openAI.ChatEndpoint.GetCompletionAsync(toolCallRequest);
conversation.AppendMessage(toolCallResponse.FirstChoice.Message);
        }
                catch (RestException restEx)
                {
                    Debug.LogError(restEx);
                    conversation.AppendMessage(new Message(toolCall,
restEx.Response.Body));
                    var toolCallRequest = new
ChatRequest(conversation.Messages, tools: assistantTools, toolChoice:
"auto");
                    toolCallResponse = await
openAI.ChatEndpoint.GetCompletionAsync(toolCallRequest);
conversation.AppendMessage(toolCallResponse.FirstChoice.Message);
        }
```

```
                if (toolCallResponse.FirstChoice.FinishReason ==
"tool_calls")
                {
                    return await ProcessToolCallAsync(toolCallResponse);
                }

                return toolCallResponse;
            }
        }

        public async void GenerateSpeech(string text)
        {
            text = text.Replace("![Image](output.jpg)", string.Empty);
            var request = new SpeechRequest(text, Model.TTS_1);
            var (clipPath, clip) = await
openAI.AudioEndpoint.CreateSpeechAsync(request,
lifetimeCancellationTokenSource.Token);
            audioSource.PlayOneShot(clip);

            if (enableDebug)
            {
                Debug.Log(clipPath);
            }
        }

        public TextMeshProUGUI AddNewTextMessageContent(Role role)
        {
            var textObject = new GameObject($"{contentArea.childCount +
1}_{role}");
            textObject.transform.SetParent(contentArea, false);
            var textMesh = textObject.AddComponent<TextMeshProUGUI>();
            textMesh.fontSize = 24;
            textMesh.enableWordWrapping = true;
            return textMesh;
        }

        private void AddNewImageContent(Texture2D texture)
        {
            var imageObject = new GameObject($"{contentArea.childCount +
1}_Image");
            imageObject.transform.SetParent(contentArea, false);
            var rawImage = imageObject.AddComponent<RawImage>();
            rawImage.texture = texture;
            var layoutElement =
imageObject.AddComponent<LayoutElement>();
            layoutElement.preferredHeight = texture.height / 4f;
            layoutElement.preferredWidth = texture.width / 4f;
            var aspectRatioFitter =
```

```
imageObject.AddComponent<AspectRatioFitter>();
            aspectRatioFitter.aspectMode =
AspectRatioFitter.AspectMode.HeightControlsWidth;
            aspectRatioFitter.aspectRatio = texture.width /
(float)texture.height;
        }

        private async Task<ImageResult>
GenerateImageAsync(ImageGenerationRequest request)
        {
            var results = await
openAI.ImagesEndPoint.GenerateImageAsync(request);
            return results.FirstOrDefault();
        }

        private void ToggleRecording()
        {
            RecordingManager.EnableDebug = enableDebug;

            if (RecordingManager.IsRecording)
            {
                RecordingManager.EndRecording();
            }
            else
            {
                inputField.interactable = false;
                RecordingManager.StartRecording<WavEncoder>(callback:
ProcessRecording);
            }
        }

        private async void ProcessRecording(Tuple<string, AudioClip>
recording)
        {
            var (path, clip) = recording;

            if (enableDebug)
            {
                Debug.Log(path);
            }

            try
            {
                recordButton.interactable = false;
                var request = new AudioTranscriptionRequest(clip,
temperature: 0.1f, language: "en");
                var userInput = await
openAI.AudioEndpoint.CreateTranscriptionAsync(request,
```

```
lifetimeCancellationTokenSource.Token);

                if (enableDebug)
                {
                    Debug.Log(userInput);
                }

                inputField.text = userInput;
                SubmitChat();
            }
            catch (Exception e)
            {
                Debug.LogError(e);
                inputField.interactable = true;
            }
            finally
            {
                recordButton.interactable = true;
            }
        }
    }
}
using UnityEngine.UI;

namespace UnityEngine.XR.Interaction.Toolkit.Samples.StarterAssets
{
    /// <summary>
    /// Add this component to a GameObject and call the <see
cref="IncrementText"/> method
    /// in response to a Unity Event to update a text display to count
up with each event.
    /// </summary>
    public class IncrementUIText : MonoBehaviour
    {
        [SerializeField]
        [Tooltip("The Text component this behavior uses to display the
incremented value.")]
        Text m_Text;

        /// <summary>
        /// The Text component this behavior uses to display the
incremented value.
        /// </summary>
        public Text text
        {
            get => m_Text;
            set => m_Text = value;
        }
```

```
        int m_Count;

        /// <summary>
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        protected void Awake()
        {
            if (m_Text == null)
                Debug.LogWarning("Missing required Text component
reference. Use the Inspector window to assign which Text component to
increment.", this);
        }

        /// <summary>
        /// Increment the string message of the Text component.
        /// </summary>
        public void IncrementText()
        {
            m_Count += 1;
            if (m_Text != null)
                m_Text.text = m_Count.ToString();
        }
    }
}
using System;
using System.Collections.Generic;
using System.Linq;
using Unity.XR.CoreUtils.Editor;
using UnityEngine.XR.Interaction.Toolkit;

namespace UnityEditor.XR.Interaction.Toolkit.Samples
{
    /// <summary>
    /// Unity Editor class which registers Project Validation rules for
the Starter Assets sample package.
    /// </summary>
    class StarterAssetsSampleProjectValidation
    {
        const string k_Category = "XR Interaction Toolkit";
        const string k_StarterAssetsSampleName = "Starter Assets";
        const string k_TeleportLayerName = "Teleport";
        const int k_TeleportLayerIndex = 31;

        static readonly BuildTargetGroup[] s_BuildTargetGroups =
((BuildTargetGroup[])Enum.GetValues(typeof(BuildTargetGroup))).Distinct(
).ToArray();
```

```
        static readonly List<BuildValidationRule> s_BuildValidationRules
= new List<BuildValidationRule>();

        [InitializeOnLoadMethod]
        static void RegisterProjectValidationRules()
        {
            // In the Player Settings UI we have to delay the call one
frame to let the settings provider get initialized
            // since we need to access the settings asset to set the
rule's non-delegate properties (FixItAutomatic).
            EditorApplication.delayCall += AddRules;
        }

        static void AddRules()
        {
            if (s_BuildValidationRules.Count == 0)
            {
                s_BuildValidationRules.Add(
                    new BuildValidationRule
                    {
                        Category = k_Category,
                        Message = $"[{k_StarterAssetsSampleName}]
Interaction Layer {k_TeleportLayerIndex} should be set to
'{k_TeleportLayerName}' for teleportation locomotion.",
                        FixItMessage = $"XR Interaction Toolkit samples
reserve Interaction Layer {k_TeleportLayerIndex} for teleportation
locomotion. Set Interaction Layer {k_TeleportLayerIndex} to
'{k_TeleportLayerName}' to prevent conflicts.",
                        HelpText = "Please note Interaction Layers are
unique to the XR Interaction Toolkit and can be found in Edit > Project
Settings > XR Plug-in Management > XR Interaction Toolkit",
                        FixItAutomatic =
InteractionLayerSettings.Instance.IsLayerEmpty(k_TeleportLayerIndex) ||
IsInteractionLayerTeleport(),
                        Error = false,
                        CheckPredicate = IsInteractionLayerTeleport,
                        FixIt = () =>
                        {
                            if
(InteractionLayerSettings.Instance.IsLayerEmpty(k_TeleportLayerIndex) ||
DisplayTeleportDialog())
InteractionLayerSettings.Instance.SetLayerNameAt(k_TeleportLayerIndex, k
_TeleportLayerName);
                            else
SettingsService.OpenProjectSettings(XRInteractionToolkitSettingsProvider
.k_SettingsPath);
                        },
                    });
```

```
            }

            foreach (var buildTargetGroup in s_BuildTargetGroups)
            {
                BuildValidator.AddRules(buildTargetGroup,
s_BuildValidationRules);
            }
        }

        static bool IsInteractionLayerTeleport()
        {
            return
string.Equals(InteractionLayerSettings.Instance.GetLayerNameAt(k_Telepor
tLayerIndex), k_TeleportLayerName, StringComparison.OrdinalIgnoreCase);
        }

        static bool DisplayTeleportDialog()
        {
            return EditorUtility.DisplayDialog(
                "Fixing Teleport Interaction Layer",
                $"Interaction Layer {k_TeleportLayerIndex} for
teleportation locomotion is currently set to
'{InteractionLayerSettings.Instance.GetLayerNameAt(k_TeleportLayerIndex)
}' instead of '{k_TeleportLayerName}'",
                "Automatically Replace",
                "Cancel");
        }
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine.Events;
using UnityEngine.InputSystem;
using UnityEngine.XR.Interaction.Toolkit.UI;

namespace UnityEngine.XR.Interaction.Toolkit.Samples.StarterAssets
{
    /// <summary>
    /// Use this class to mediate the controllers and their associated
interactors and input actions under different interaction states.
    /// </summary>
    [AddComponentMenu("XR/Action Based Controller Manager")]
    [DefaultExecutionOrder(k_UpdateOrder)]
    public class ActionBasedControllerManager : MonoBehaviour
    {
        /// <summary>
        /// Order when instances of type <see
cref="ActionBasedControllerManager"/> are updated.
```

```
        /// </summary>
        /// <remarks>
        /// Executes before controller components to ensure input
processors can be attached
        /// to input actions and/or bindings before the controller
component reads the current
        /// values of the input actions.
        /// </remarks>
        public const int k_UpdateOrder =
XRInteractionUpdateOrder.k_Controllers - 1;

        [Space]
        [Header("Interactors")]

        [SerializeField]
        [Tooltip("The GameObject containing the interaction group used
for direct and distant manipulation.")]
        XRInteractionGroup m_ManipulationInteractionGroup;

        [SerializeField]
        [Tooltip("The GameObject containing the interactor used for
direct manipulation.")]
        XRDirectInteractor m_DirectInteractor;

        [SerializeField]
        [Tooltip("The GameObject containing the interactor used for
distant/ray manipulation.")]
        XRRayInteractor m_RayInteractor;

        [SerializeField]
        [Tooltip("The GameObject containing the interactor used for
teleportation.")]
        XRRayInteractor m_TeleportInteractor;

        [Space]
        [Header("Controller Actions")]

        [SerializeField]
        [Tooltip("The reference to the action to start the teleport
aiming mode for this controller.")]
        InputActionReference m_TeleportModeActivate;

        [SerializeField]
        [Tooltip("The reference to the action to cancel the teleport
aiming mode for this controller.")]
        InputActionReference m_TeleportModeCancel;

        [SerializeField]
```

```
        [Tooltip("The reference to the action of continuous turning the
XR Origin with this controller.")]
        InputActionReference m_Turn;

        [SerializeField]
        [Tooltip("The reference to the action of snap turning the XR
Origin with this controller.")]
        InputActionReference m_SnapTurn;

        [SerializeField]
        [Tooltip("The reference to the action of moving the XR Origin
with this controller.")]
        InputActionReference m_Move;

        [SerializeField]
        [Tooltip("The reference to the action of scrolling UI with this
controller.")]
        InputActionReference m_UIScroll;

        [Space]
        [Header("Locomotion Settings")]

        [SerializeField]
        [Tooltip("If true, continuous movement will be enabled. If
false, teleport will enabled.")]
        bool m_SmoothMotionEnabled;

        [SerializeField]
        [Tooltip("If true, continuous turn will be enabled. If false,
snap turn will be enabled. Note: If smooth motion is enabled and enable
strafe is enabled on the continuous move provider, turn will be
overriden in favor of strafe.")]
        bool m_SmoothTurnEnabled;

        [Space]
        [Header("UI Settings")]

        [SerializeField]
        [Tooltip("If true, UI scrolling will be enabled.")]
        bool m_UIScrollingEnabled;

        [Space]
        [Header("Mediation Events")]
        [SerializeField]
        [Tooltip("Event fired when the active ray interactor changes
between interaction and teleport.")]
        UnityEvent<IXRRayProvider> m_RayInteractorChanged;
```

```
        public bool smoothMotionEnabled
        {
            get => m_SmoothMotionEnabled;
            set
            {
                m_SmoothMotionEnabled = value;
                UpdateLocomotionActions();
            }
        }

        public bool smoothTurnEnabled
        {
            get => m_SmoothTurnEnabled;
            set
            {
                m_SmoothTurnEnabled = value;
                UpdateLocomotionActions();
            }
        }

        public bool uiScrollingEnabled
        {
            get => m_UIScrollingEnabled;
            set
            {
                m_UIScrollingEnabled = value;
                UpdateUIActions();
            }
        }

        bool m_PostponedDeactivateTeleport;
        bool m_UIScrollModeActive = false;

        const int k_InteractorNotInGroup = -1;

        IEnumerator m_AfterInteractionEventsRoutine;
        HashSet<InputAction> m_LocomotionUsers = new
HashSet<InputAction>();

        /// <summary>
        /// Temporary scratch list to populate with the group members of
the interaction group.
        /// </summary>
        static readonly List<IXRGroupMember> s_GroupMembers = new
List<IXRGroupMember>();

        // For our input mediation, we are enforcing a few rules between
direct, ray, and teleportation interaction:
```

```
        // 1. If the Teleportation Ray is engaged, the Ray interactor is
disabled
        // 2. The interaction group ensures that the Direct and Ray
interactors cannot interact at the same time, with the Direct interactor
taking priority
        // 3. If the Ray interactor is selecting, all locomotion
controls are disabled (teleport ray, move, and turn controls) to prevent
input collision
        void SetupInteractorEvents()
        {
            if (m_RayInteractor != null)
            {
m_RayInteractor.selectEntered.AddListener(OnRaySelectEntered);
     m_RayInteractor.selectExited.AddListener(OnRaySelectExited);
m_RayInteractor.uiHoverEntered.AddListener(OnUIHoverEntered);
    m_RayInteractor.uiHoverExited.AddListener(OnUIHoverExited);
            }

            var teleportModeActivateAction =
GetInputAction(m_TeleportModeActivate);
            if (teleportModeActivateAction != null)
            {
                teleportModeActivateAction.performed += OnStartTeleport;
                teleportModeActivateAction.performed +=
OnStartLocomotion;
                teleportModeActivateAction.canceled += OnCancelTeleport;
                teleportModeActivateAction.canceled += OnStopLocomotion;
            }

            var teleportModeCancelAction =
GetInputAction(m_TeleportModeCancel);
            if (teleportModeCancelAction != null)
            {
                teleportModeCancelAction.performed += OnCancelTeleport;
                teleportModeActivateAction.canceled += OnStopLocomotion;
            }

            var moveAction = GetInputAction(m_Move);
            if (moveAction != null)
            {
                moveAction.performed += OnStartLocomotion;
                moveAction.canceled += OnStopLocomotion;
            }

            var turnAction = GetInputAction(m_Turn);
            if (turnAction != null)
            {
                turnAction.performed += OnStartLocomotion;
```

```
                turnAction.canceled += OnStopLocomotion;
            }
        }

        void TeardownInteractorEvents()
        {
            if (m_RayInteractor != null)
            {
m_RayInteractor.selectEntered.RemoveListener(OnRaySelectEntered);
        m_RayInteractor.selectExited.RemoveListener(OnRaySelectExited);
            }

            var teleportModeActivateAction =
GetInputAction(m_TeleportModeActivate);
            if (teleportModeActivateAction != null)
            {
                teleportModeActivateAction.performed -= OnStartTeleport;
                teleportModeActivateAction.performed -=
OnStartLocomotion;
                teleportModeActivateAction.canceled -= OnCancelTeleport;
                teleportModeActivateAction.canceled -= OnStopLocomotion;
            }

            var teleportModeCancelAction =
GetInputAction(m_TeleportModeCancel);
            if (teleportModeCancelAction != null)
            {
                teleportModeCancelAction.performed -= OnCancelTeleport;
                teleportModeCancelAction.performed -= OnStopLocomotion;
            }

            var moveAction = GetInputAction(m_Move);
            if (moveAction != null)
            {
                moveAction.performed -= OnStartLocomotion;
                moveAction.canceled -= OnStopLocomotion;
            }

            var turnAction = GetInputAction(m_Turn);
            if (turnAction != null)
            {
                turnAction.performed -= OnStartLocomotion;
                turnAction.canceled -= OnStopLocomotion;
            }
        }

        void OnStartTeleport(InputAction.CallbackContext context)
        {
```

```
            m_PostponedDeactivateTeleport = false;

            if (m_TeleportInteractor != null)
                m_TeleportInteractor.gameObject.SetActive(true);

            if (m_RayInteractor != null)
                m_RayInteractor.gameObject.SetActive(false);

            m_RayInteractorChanged?.Invoke(m_TeleportInteractor);
        }

        void OnCancelTeleport(InputAction.CallbackContext context)
        {
            // Do not deactivate the teleport interactor in this
callback.
            // We delay turning off the teleport interactor in this
callback so that
            // the teleport interactor has a chance to complete the
teleport if needed.
            // OnAfterInteractionEvents will handle deactivating its
GameObject.
            m_PostponedDeactivateTeleport = true;

            if (m_RayInteractor != null)
                m_RayInteractor.gameObject.SetActive(true);

            m_RayInteractorChanged?.Invoke(m_RayInteractor);

        }

        void OnStartLocomotion(InputAction.CallbackContext context)
        {
            if (!context.started)
                return;

            m_LocomotionUsers.Add(context.action);
        }

        void OnStopLocomotion(InputAction.CallbackContext context)
        {
            m_LocomotionUsers.Remove(context.action);

            if (m_LocomotionUsers.Count == 0 && m_UIScrollModeActive)
            {
                DisableLocomotionActions();
            }
        }
```

```
        void OnRaySelectEntered(SelectEnterEventArgs args)
        {
            // Disable locomotion and turn actions
            DisableLocomotionActions();
        }

        void OnRaySelectExited(SelectExitEventArgs args)
        {
            // Re-enable the locomotion and turn actions
            UpdateLocomotionActions();
        }

        void OnUIHoverEntered(UIHoverEventArgs args)
        {
            m_UIScrollModeActive = args.deviceModel.isScrollable &&
m_UIScrollingEnabled;
            if (!m_UIScrollModeActive)
                return;

            // If locomotion is occurring, wait
            if (m_LocomotionUsers.Count == 0)
            {
                // Disable locomotion and turn actions
                DisableLocomotionActions();
            }
        }

        void OnUIHoverExited(UIHoverEventArgs args)
        {
            m_UIScrollModeActive = false;

            // Re-enable the locomotion and turn actions
            UpdateLocomotionActions();
        }

        protected void Awake()
        {
            m_AfterInteractionEventsRoutine =
OnAfterInteractionEvents();
        }

        protected void OnEnable()
        {
            if (m_TeleportInteractor != null)
                m_TeleportInteractor.gameObject.SetActive(false);

            SetupInteractorEvents();
```

```
            // Start the coroutine that executes code after the Update
phase (during yield null).
            // Since this behavior has an execution order that runs
before the XRInteractionManager,
            // we use the coroutine to run after the selection events
            StartCoroutine(m_AfterInteractionEventsRoutine);
        }

        protected void OnDisable()
        {
            TeardownInteractorEvents();

            StopCoroutine(m_AfterInteractionEventsRoutine);
        }

        protected void Start()
        {
            // Ensure the enabled state of locomotion and turn actions
are properly set up.
            // Called in Start so it is done after the
InputActionManager enables all input actions earlier in OnEnable.
            UpdateLocomotionActions();
            UpdateUIActions();

            if (m_ManipulationInteractionGroup == null)
            {
                Debug.LogError("Missing required Manipulation
Interaction Group reference. Use the Inspector window to assign the XR
Interaction Group component reference.", this);
                return;
            }

            // Ensure interactors are properly set up in the interaction
group by adding
            // them if necessary and ordering Direct before Ray
interactor.
            var directInteractorIndex = k_InteractorNotInGroup;
            var rayInteractorIndex = k_InteractorNotInGroup;
m_ManipulationInteractionGroup.GetGroupMembers(s_GroupMembers);
  for (var i = 0; i < s_GroupMembers.Count; ++i)
            {
                var groupMember = s_GroupMembers[i];
                if (ReferenceEquals(groupMember, m_DirectInteractor))
                    directInteractorIndex = i;
                else if (ReferenceEquals(groupMember, m_RayInteractor))
                    rayInteractorIndex = i;
            }
```

```
            if (directInteractorIndex == k_InteractorNotInGroup)
            {
                // Must add Direct interactor to group, and make sure it
is ordered before the Ray interactor
                if (rayInteractorIndex == k_InteractorNotInGroup)
                {
                    // Must add Ray interactor to group
                    if (m_DirectInteractor != null)
m_ManipulationInteractionGroup.AddGroupMember(m_DirectInteractor);
             if (m_RayInteractor != null)
m_ManipulationInteractionGroup.AddGroupMember(m_RayInteractor);
      }
                else if (m_DirectInteractor != null)
                {
m_ManipulationInteractionGroup.MoveGroupMemberTo(m_DirectInteractor, ray
InteractorIndex);
                }
            }
            else
            {
                if (rayInteractorIndex == k_InteractorNotInGroup)
                {
                    // Must add Ray interactor to group
                    if (m_RayInteractor != null)
m_ManipulationInteractionGroup.AddGroupMember(m_RayInteractor);
      }
                else
                {
                    // Must make sure Direct interactor is ordered
before the Ray interactor
                    if (rayInteractorIndex < directInteractorIndex)
                    {
m_ManipulationInteractionGroup.MoveGroupMemberTo(m_DirectInteractor, ray
InteractorIndex);
                    }
                }
            }
        }

        IEnumerator OnAfterInteractionEvents()
        {
            while (true)
            {
                // Yield so this coroutine is resumed after the teleport
interactor
                // has a chance to process its select interaction event
during Update.
                yield return null;
```

```
                if (m_PostponedDeactivateTeleport)
                {
                    if (m_TeleportInteractor != null)
m_TeleportInteractor.gameObject.SetActive(false);                                m_P
ostponedDeactivateTeleport = false;
                }
            }
        }

        void UpdateLocomotionActions()
        {
            // Disable/enable Teleport and Turn when Move is
enabled/disabled.
            SetEnabled(m_Move, m_SmoothMotionEnabled);
            SetEnabled(m_TeleportModeActivate, !m_SmoothMotionEnabled);
            SetEnabled(m_TeleportModeCancel, !m_SmoothMotionEnabled);

            // Disable ability to turn when using continuous movement
            SetEnabled(m_Turn, !m_SmoothMotionEnabled &&
m_SmoothTurnEnabled);
            SetEnabled(m_SnapTurn, !m_SmoothMotionEnabled &&
!m_SmoothTurnEnabled);
        }

        void DisableLocomotionActions()
        {
            DisableAction(m_Move);
            DisableAction(m_TeleportModeActivate);
            DisableAction(m_TeleportModeCancel);
            DisableAction(m_Turn);
            DisableAction(m_SnapTurn);
        }

        void UpdateUIActions()
        {
            SetEnabled(m_UIScroll, m_UIScrollingEnabled);
        }

        static void SetEnabled(InputActionReference actionReference,
bool enabled)
        {
            if (enabled)
                EnableAction(actionReference);
            else
                DisableAction(actionReference);
        }
```

```
        static void EnableAction(InputActionReference actionReference)
        {
            var action = GetInputAction(actionReference);
            if (action != null && !action.enabled)
                action.Enable();
        }

        static void DisableAction(InputActionReference actionReference)
        {
            var action = GetInputAction(actionReference);
            if (action != null && action.enabled)
                action.Disable();
        }

        static InputAction GetInputAction(InputActionReference
actionReference)
        {
#pragma warning disable IDE0031 // Use null propagation -- Do not use
for UnityEngine.Object types
            return actionReference != null ? actionReference.action :
null;
#pragma warning restore IDE0031
        }
    }
}
namespace UnityEngine.XR.Interaction.Toolkit.Samples.StarterAssets
{
    /// <summary>
    /// Destroys the GameObject it is attached to after a specified
amount of time.
    /// </summary>
    public class DestroySelf : MonoBehaviour
    {
        [SerializeField]
        [Tooltip("The amount of time, in seconds, to wait after Start
before destroying the GameObject.")]
        float m_Lifetime = 0.25f;

        /// <summary>
        /// The amount of time, in seconds, to wait after Start before
destroying the GameObject.
        /// </summary>
        public float lifetime
        {
            get => m_Lifetime;
            set => m_Lifetime = value;
        }
```

```
        /// <summary>
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        void Start()
        {
            Destroy(gameObject, m_Lifetime);
        }
    }
}
using Unity.XR.CoreUtils;
using UnityEngine.Assertions;

namespace UnityEngine.XR.Interaction.Toolkit.Samples.StarterAssets
{
    /// <summary>
    /// A version of action-based continuous movement that automatically
controls the frame of reference that
    /// determines the forward direction of movement based on user
preference for each hand.
    /// For example, can configure to use head relative movement for the
left hand and controller relative movement for the right hand.
    /// </summary>
    public class DynamicMoveProvider : ActionBasedContinuousMoveProvider
    {
        /// <summary>
        /// Defines which transform the XR Origin's movement direction
is relative to.
        /// </summary>
        /// <seealso cref="leftHandMovementDirection"/>
        /// <seealso cref="rightHandMovementDirection"/>
        public enum MovementDirection
        {
            /// <summary>
            /// Use the forward direction of the head (camera) as the
forward direction of the XR Origin's movement.
            /// </summary>
            HeadRelative,

            /// <summary>
            /// Use the forward direction of the hand (controller) as
the forward direction of the XR Origin's movement.
            /// </summary>
            HandRelative,
        }

        [Space, Header("Movement Direction")]
        [SerializeField]
        [Tooltip("Directs the XR Origin's movement when using the head-
```

```
relative mode. If not set, will automatically find and use the XR Origin
Camera.")]
        Transform m_HeadTransform;

        /// <summary>
        /// Directs the XR Origin's movement when using the head-
relative mode. If not set, will automatically find and use the XR Origin
Camera.
        /// </summary>
        public Transform headTransform
        {
            get => m_HeadTransform;
            set => m_HeadTransform = value;
        }

        [SerializeField]
        [Tooltip("Directs the XR Origin's movement when using the hand-
relative mode with the left hand.")]
        Transform m_LeftControllerTransform;

        /// <summary>
        /// Directs the XR Origin's movement when using the hand-
relative mode with the left hand.
        /// </summary>
        public Transform leftControllerTransform
        {
            get => m_LeftControllerTransform;
            set => m_LeftControllerTransform = value;
        }

        [SerializeField]
        [Tooltip("Directs the XR Origin's movement when using the hand-
relative mode with the right hand.")]
        Transform m_RightControllerTransform;

        public Transform rightControllerTransform
        {
            get => m_RightControllerTransform;
            set => m_RightControllerTransform = value;
        }

        [SerializeField]
        [Tooltip("Whether to use the specified head transform or left
controller transform to direct the XR Origin's movement for the left
hand.")]
        MovementDirection m_LeftHandMovementDirection;

        /// <summary>
```

```
        /// Whether to use the specified head transform or controller
transform to direct the XR Origin's movement for the left hand.
        /// </summary>
        /// <seealso cref="MovementDirection"/>
        public MovementDirection leftHandMovementDirection
        {
            get => m_LeftHandMovementDirection;
            set => m_LeftHandMovementDirection = value;
        }

        [SerializeField]
        [Tooltip("Whether to use the specified head transform or right
controller transform to direct the XR Origin's movement for the right
hand.")]
        MovementDirection m_RightHandMovementDirection;

        /// <summary>
        /// Whether to use the specified head transform or controller
transform to direct the XR Origin's movement for the right hand.
        /// </summary>
        /// <seealso cref="MovementDirection"/>
        public MovementDirection rightHandMovementDirection
        {
            get => m_RightHandMovementDirection;
            set => m_RightHandMovementDirection = value;
        }

        Transform m_CombinedTransform;
        Pose m_LeftMovementPose = Pose.identity;
        Pose m_RightMovementPose = Pose.identity;

        /// <inheritdoc />
        protected override void Awake()
        {
            base.Awake();

            m_CombinedTransform = new GameObject("[Dynamic Move
Provider] Combined Forward Source").transform;
            m_CombinedTransform.SetParent(transform, false);
            m_CombinedTransform.localPosition = Vector3.zero;
            m_CombinedTransform.localRotation = Quaternion.identity;

            forwardSource = m_CombinedTransform;
        }

        /// <inheritdoc />
        protected override Vector3 ComputeDesiredMove(Vector2 input)
        {
```

```
            // Don't need to do anything if the total input is zero.
            // This is the same check as the base method.
            if (input == Vector2.zero)
                return Vector3.zero;

            // Initialize the Head Transform if necessary, getting the
Camera from XR Origin
            if (m_HeadTransform == null)
            {
                var xrOrigin = system.xrOrigin;
                if (xrOrigin != null)
                {
                    var xrCamera = xrOrigin.Camera;
                    if (xrCamera != null)
                        m_HeadTransform = xrCamera.transform;
                }
            }

            // Get the forward source for the left hand input
            switch (m_LeftHandMovementDirection)
            {
                case MovementDirection.HeadRelative:
                    if (m_HeadTransform != null)
                        m_LeftMovementPose =
m_HeadTransform.GetWorldPose();

                    break;

                case MovementDirection.HandRelative:
                    if (m_LeftControllerTransform != null)
                        m_LeftMovementPose =
m_LeftControllerTransform.GetWorldPose();

                    break;

                default:
                    Assert.IsTrue(false, $"Unhandled
{nameof(MovementDirection)}={m_LeftHandMovementDirection}");
                    break;
            }

            // Get the forward source for the right hand input
            switch (m_RightHandMovementDirection)
            {
                case MovementDirection.HeadRelative:
                    if (m_HeadTransform != null)
                        m_RightMovementPose =
m_HeadTransform.GetWorldPose();
```

```
                    break;

                case MovementDirection.HandRelative:
                    if (m_RightControllerTransform != null)
                        m_RightMovementPose =
m_RightControllerTransform.GetWorldPose();

                    break;

                default:
                    Assert.IsTrue(false, $"Unhandled
{nameof(MovementDirection)}={m_RightHandMovementDirection}");
                    break;
            }

            // Combine the two poses into the forward source based on
the magnitude of input
            var leftHandValue =
leftHandMoveAction.action?.ReadValue<Vector2>() ?? Vector2.zero;
            var rightHandValue =
rightHandMoveAction.action?.ReadValue<Vector2>() ?? Vector2.zero;

            var totalSqrMagnitude = leftHandValue.sqrMagnitude +
rightHandValue.sqrMagnitude;
            var leftHandBlend = 0.5f;
            if (totalSqrMagnitude > Mathf.Epsilon)
                leftHandBlend = leftHandValue.sqrMagnitude /
totalSqrMagnitude;

            var combinedPosition =
Vector3.Lerp(m_RightMovementPose.position, m_LeftMovementPose.position,
leftHandBlend);
            var combinedRotation =
Quaternion.Slerp(m_RightMovementPose.rotation,
m_LeftMovementPose.rotation, leftHandBlend);
            m_CombinedTransform.SetPositionAndRotation(combinedPosition,
combinedRotation);

            return base.ComputeDesiredMove(input);
        }
    }
}
using System.Collections.Generic;
using UnityEngine.InputSystem;

namespace UnityEngine.XR.Interaction.Toolkit.Samples.StarterAssets
{
```

```
    /// <summary>
    /// Manages input fallback for <see cref="XRGazeInteractor"/> when
eye tracking is not available.
    /// </summary>
    public class GazeInputManager : MonoBehaviour
    {
        // This is the name of the layout that is registered by
EyeGazeInteraction in the OpenXR Plugin package
        const string k_EyeGazeLayoutName = "EyeGaze";

        [SerializeField]
        [Tooltip("Enable fallback to head tracking if eye tracking is
unavailable.")]
        bool m_FallbackIfEyeTrackingUnavailable = true;

        /// <summary>
        /// Enable fallback to head tracking if eye tracking is
unavailable.
        /// </summary>
        public bool fallbackIfEyeTrackingUnavailable
        {
            get => m_FallbackIfEyeTrackingUnavailable;
            set => m_FallbackIfEyeTrackingUnavailable = value;
        }


        bool m_EyeTrackingDeviceFound;

        /// <summary>
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        protected void Awake()
        {
            // Check if we have eye tracking support
            var inputDeviceList = new List<InputDevice>();
InputDevices.GetDevicesWithCharacteristics(InputDeviceCharacteristics.Ey
eTracking, inputDeviceList);
            if (inputDeviceList.Count > 0)
            {
                Debug.Log("Eye tracking device found!", this);
                m_EyeTrackingDeviceFound = true;
                return;
            }

            foreach (var device in InputSystem.InputSystem.devices)
            {
                if (device.layout == k_EyeGazeLayoutName)
                {
```

```
                    Debug.Log("Eye gaze device found!", this);
                    m_EyeTrackingDeviceFound = true;
                    return;
                }
            }

            Debug.LogWarning($"Could not find a device that supports eye
tracking on Awake. {this} has subscribed to device connected events and
will activate the GameObject when an eye tracking device is connected.",
this);

            InputDevices.deviceConnected += OnDeviceConnected;
            InputSystem.InputSystem.onDeviceChange += OnDeviceChange;

            gameObject.SetActive(m_FallbackIfEyeTrackingUnavailable);
        }

        /// <summary>
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        protected void OnDestroy()
        {
            InputDevices.deviceConnected -= OnDeviceConnected;
            InputSystem.InputSystem.onDeviceChange -= OnDeviceChange;
        }

        void OnDeviceConnected(InputDevice inputDevice)
        {
            if (m_EyeTrackingDeviceFound ||
!inputDevice.characteristics.HasFlag(InputDeviceCharacteristics.EyeTrack
ing))
                return;

            Debug.Log("Eye tracking device found!", this);
            m_EyeTrackingDeviceFound = true;
            gameObject.SetActive(true);
        }

        void OnDeviceChange(InputSystem.InputDevice device,
InputDeviceChange change)
        {
            if (m_EyeTrackingDeviceFound || change !=
InputDeviceChange.Added)
                return;

            if (device.layout == k_EyeGazeLayoutName)
            {
                Debug.Log("Eye gaze device found!", this);
```

```
                m_EyeTrackingDeviceFound = true;
                gameObject.SetActive(true);
            }
        }
    }
}
using System;
using System.Collections.Generic;
using UnityEngine.XR.Interaction.Toolkit.Utilities;

namespace UnityEngine.XR.Interaction.Toolkit.Samples.StarterAssets
{
    /// <summary>
    /// Behavior with an API for spawning objects from a given set of
prefabs.
    /// </summary>
    public class ObjectSpawner : MonoBehaviour
    {
        [SerializeField]
        [Tooltip("The camera that objects will face when spawned. If not
set, defaults to the main camera.")]
        Camera m_CameraToFace;

        /// <summary>
        /// The camera that objects will face when spawned. If not set,
defaults to the <see cref="Camera.main"/> camera.
        /// </summary>
        public Camera cameraToFace
        {
            get
            {
                EnsureFacingCamera();
                return m_CameraToFace;
            }
            set => m_CameraToFace = value;
        }

        [SerializeField]
        [Tooltip("The list of prefabs available to spawn.")]
        List<GameObject> m_ObjectPrefabs = new List<GameObject>();

        /// <summary>
        /// The list of prefabs available to spawn.
        /// </summary>
        public List<GameObject> objectPrefabs
        {
            get => m_ObjectPrefabs;
            set => m_ObjectPrefabs = value;
```

```
        }

        [SerializeField]
        [Tooltip("Optional prefab to spawn for each spawned object. Use a prefab with the Destroy Self component to make " +
            "sure the visualization only lives temporarily.")]
        GameObject m_SpawnVisualizationPrefab;

        /// <summary>
        /// Optional prefab to spawn for each spawned object.
        /// </summary>
        /// <remarks>Use a prefab with <see cref="DestroySelf"/> to make sure the visualization only lives temporarily.</remarks>
        public GameObject spawnVisualizationPrefab
        {
            get => m_SpawnVisualizationPrefab;
            set => m_SpawnVisualizationPrefab = value;
        }

        [SerializeField]
        [Tooltip("The index of the prefab to spawn. If outside the range of the list, this behavior will select " +
            "a random object each time it spawns.")]
        int m_SpawnOptionIndex = -1;

        /// <summary>
        /// The index of the prefab to spawn. If outside the range of <see cref="objectPrefabs"/>, this behavior will
        /// select a random object each time it spawns.
        /// </summary>
        /// <seealso cref="isSpawnOptionRandomized"/>
        public int spawnOptionIndex
        {
            get => m_SpawnOptionIndex;
            set => m_SpawnOptionIndex = value;
        }

        /// <summary>
        /// Whether this behavior will select a random object from <see cref="objectPrefabs"/> each time it spawns.
        /// </summary>
        /// <seealso cref="spawnOptionIndex"/>
        /// <seealso cref="RandomizeSpawnOption"/>
        public bool isSpawnOptionRandomized => m_SpawnOptionIndex < 0 || m_SpawnOptionIndex >= m_ObjectPrefabs.Count;

        [SerializeField]
        [Tooltip("Whether to only spawn an object if the spawn point is
```

```
within view of the camera.")]
        bool m_OnlySpawnInView = true;

        /// <summary>
        /// Whether to only spawn an object if the spawn point is within
view of the <see cref="cameraToFace"/>.
        /// </summary>
        public bool onlySpawnInView
        {
            get => m_OnlySpawnInView;
            set => m_OnlySpawnInView = value;
        }

        [SerializeField]
        [Tooltip("The size, in viewport units, of the periphery inside
the viewport that will not be considered in view.")]
        float m_ViewportPeriphery = 0.15f;

        /// <summary>
        /// The size, in viewport units, of the periphery inside the
viewport that will not be considered in view.
        /// </summary>
        public float viewportPeriphery
        {
            get => m_ViewportPeriphery;
            set => m_ViewportPeriphery = value;
        }

        [SerializeField]
        [Tooltip("When enabled, the object will be rotated about the y-
axis when spawned by Spawn Angle Range, " +
            "in relation to the direction of the spawn point to the
camera.")]
        bool m_ApplyRandomAngleAtSpawn = true;

        /// <summary>
        /// When enabled, the object will be rotated about the y-axis
when spawned by <see cref="spawnAngleRange"/>
        /// in relation to the direction of the spawn point to the
camera.
        /// </summary>
        public bool applyRandomAngleAtSpawn
        {
            get => m_ApplyRandomAngleAtSpawn;
            set => m_ApplyRandomAngleAtSpawn = value;
        }

        [SerializeField]
```

```
        [Tooltip("The range in degrees that the object will randomly be
rotated about the y axis when spawned, " +
            "in relation to the direction of the spawn point to the
camera.")]
        float m_SpawnAngleRange = 45f;

        /// <summary>
        /// The range in degrees that the object will randomly be
rotated about the y axis when spawned, in relation
        /// to the direction of the spawn point to the camera.
        /// </summary>
        public float spawnAngleRange
        {
            get => m_SpawnAngleRange;
            set => m_SpawnAngleRange = value;
        }

        [SerializeField]
        [Tooltip("Whether to spawn each object as a child of this
object.")]
        bool m_SpawnAsChildren;

        /// <summary>
        /// Whether to spawn each object as a child of this object.
        /// </summary>
        public bool spawnAsChildren
        {
            get => m_SpawnAsChildren;
            set => m_SpawnAsChildren = value;
        }

        /// <summary>
        /// Event invoked after an object is spawned.
        /// </summary>
        /// <seealso cref="TrySpawnObject"/>
        public event Action<GameObject> objectSpawned;

        /// <summary>
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        void Awake()
        {
            EnsureFacingCamera();
        }

        void EnsureFacingCamera()
        {
            if (m_CameraToFace == null)
```

```
                m_CameraToFace = Camera.main;
        }

        /// <summary>
        /// Sets this behavior to select a random object from <see
cref="objectPrefabs"/> each time it spawns.
        /// </summary>
        /// <seealso cref="spawnOptionIndex"/>
        /// <seealso cref="isSpawnOptionRandomized"/>
        public void RandomizeSpawnOption()
        {
            m_SpawnOptionIndex = -1;
        }

        /// <summary>
        /// Attempts to spawn an object from <see cref="objectPrefabs"/>
at the given position. The object will have a
        /// yaw rotation that faces <see cref="cameraToFace"/>, plus or
minus a random angle within <see cref="spawnAngleRange"/>.
        /// </summary>
        /// <param name="spawnPoint">The world space position at which
to spawn the object.</param>
        /// <param name="spawnNormal">The world space normal of the
spawn surface.</param>
        /// <returns>Returns <see langword="true"/> if the spawner
successfully spawned an object. Otherwise returns
        /// <see langword="false"/>, for instance if the spawn point is
out of view of the camera.</returns>
        /// <remarks>
        /// The object selected to spawn is based on <see
cref="spawnOptionIndex"/>. If the index is outside
        /// the range of <see cref="objectPrefabs"/>, this method will
select a random prefab from the list to spawn.
        /// Otherwise, it will spawn the prefab at the index.
        /// </remarks>
        /// <seealso cref="objectSpawned"/>
        public bool TrySpawnObject(Vector3 spawnPoint, Vector3
spawnNormal)
        {
            if (m_OnlySpawnInView)
            {
                var inViewMin = m_ViewportPeriphery;
                var inViewMax = 1f - m_ViewportPeriphery;
                var pointInViewportSpace =
cameraToFace.WorldToViewportPoint(spawnPoint);
                if (pointInViewportSpace.z < 0f ||
pointInViewportSpace.x > inViewMax || pointInViewportSpace.x < inViewMin
||
```

```
                    pointInViewportSpace.y > inViewMax ||
pointInViewportSpace.y < inViewMin)
                {
                    return false;
                }
            }

            var objectIndex = isSpawnOptionRandomized ? Random.Range(0,
m_ObjectPrefabs.Count) : m_SpawnOptionIndex;
            var newObject = Instantiate(m_ObjectPrefabs[objectIndex]);
            if (m_SpawnAsChildren)
                newObject.transform.parent = transform;

            newObject.transform.position = spawnPoint;
            EnsureFacingCamera();

            var facePosition = m_CameraToFace.transform.position;
            var forward = facePosition - spawnPoint;
            BurstMathUtility.ProjectOnPlane(forward, spawnNormal, out
var projectedForward);
            newObject.transform.rotation =
Quaternion.LookRotation(projectedForward, spawnNormal);

            if (m_ApplyRandomAngleAtSpawn)
            {
                var randomRotation = Random.Range(-m_SpawnAngleRange,
m_SpawnAngleRange);
                newObject.transform.Rotate(Vector3.up, randomRotation);
            }

            if (m_SpawnVisualizationPrefab != null)
            {
                var visualizationTrans =
Instantiate(m_SpawnVisualizationPrefab).transform;
                visualizationTrans.position = spawnPoint;
                visualizationTrans.rotation =
newObject.transform.rotation;
            }

            objectSpawned?.Invoke(newObject);
            return true;
        }
    }
}
using Unity.Mathematics;
using Unity.XR.CoreUtils.Bindings;
using UnityEngine.XR.Interaction.Toolkit.AffordanceSystem.State;
using UnityEngine.XR.Interaction.Toolkit.Filtering;
```

```
using
UnityEngine.XR.Interaction.Toolkit.Utilities.Tweenables.Primitives;

namespace UnityEngine.XR.Interaction.Toolkit.Samples.StarterAssets
{
    /// <summary>
    /// Follow animation affordance for <see
cref="IPokeStateDataProvider"/>, such as <see cref="XRPokeFilter"/>.
    /// Used to animate a pressed transform, such as a button to follow
the poke position.
    /// </summary>
    [AddComponentMenu("XR/XR Poke Follow Affordance", 22)]
    public class XRPokeFollowAffordance : MonoBehaviour
    {
        [SerializeField]
        [Tooltip("Transform that will move in the poke direction when
this or a parent GameObject is poked." +
                 "\nNote: Should be a direct child GameObject.")]
        Transform m_PokeFollowTransform;

        /// <summary>
        /// Transform that will animate along the axis of interaction
when this interactable is poked.
        /// Note: Must be a direct child GameObject as it moves in local
space relative to the poke target's transform.
        /// </summary>
        public Transform pokeFollowTransform
        {
            get => m_PokeFollowTransform;
            set => m_PokeFollowTransform = value;
        }

        [SerializeField]
        [Range(0f, 20f)]
        [Tooltip("Multiplies transform position interpolation as a
factor of Time.deltaTime. If 0, no smoothing will be applied.")]
        float m_SmoothingSpeed = 16f;

        /// <summary>
        /// Multiplies transform position interpolation as a factor of
<see cref="Time.deltaTime"/>. If <c>0</c>, no smoothing will be applied.
        /// </summary>
        public float smoothingSpeed
        {
            get => m_SmoothingSpeed;
            set => m_SmoothingSpeed = value;
        }
```

```
        [SerializeField]
        [Tooltip("When this component is no longer the target of the
poke, the Poke Follow Transform returns to the original position.")]
        bool m_ReturnToInitialPosition = true;

        /// <summary>
        /// When this component is no longer the target of the poke, the
<see cref="pokeFollowTransform"/> returns to the original position.
        /// </summary>
        public bool returnToInitialPosition
        {
            get => m_ReturnToInitialPosition;
            set => m_ReturnToInitialPosition = value;
        }

        [SerializeField]
        [Tooltip("Whether to apply the follow animation if the target of
the poke is a child of this transform. " +
                 "This is useful for UI objects that may have child
graphics.")]
        bool m_ApplyIfChildIsTarget = true;

        /// <summary>
        /// Whether to apply the follow animation if the target of the
poke is a child of this transform.
        /// This is useful for UI objects that may have child graphics.
        /// </summary>
        public bool applyIfChildIsTarget
        {
            get => m_ApplyIfChildIsTarget;
            set => m_ApplyIfChildIsTarget = value;
        }

        [SerializeField]
        [Tooltip("Whether to keep the Poke Follow Transform from moving
past a maximum distance from the poke target.")]
        bool m_ClampToMaxDistance;

        /// <summary>
        /// Whether to keep the <see cref="pokeFollowTransform"/> from
moving past <see cref="maxDistance"/> from the poke target.
        /// </summary>
        public bool clampToMaxDistance
        {
            get => m_ClampToMaxDistance;
            set => m_ClampToMaxDistance = value;
        }
```

```
        [SerializeField]
        [Tooltip("The maximum distance from this transform that the Poke
Follow Transform can move.")]
        float m_MaxDistance;

        /// <summary>
        /// The maximum distance from this transform that the <see
cref="pokeFollowTransform"/> can move when
        /// <see cref="clampToMaxDistance"/> is <see langword="true"/>.
        /// </summary>
        public float maxDistance
        {
            get => m_MaxDistance;
            set => m_MaxDistance = value;
        }

        /// <summary>
        /// The original position of this interactable before any pushes
have been applied.
        /// </summary>
        public Vector3 initialPosition
        {
            get => m_InitialPosition;
            set => m_InitialPosition = value;
        }

        IPokeStateDataProvider m_PokeDataProvider;
        IMultiPokeStateDataProvider m_MultiPokeStateDataProvider;

        readonly Vector3TweenableVariable m_TransformTweenableVariable =
new Vector3TweenableVariable();
        readonly BindingsGroup m_BindingsGroup = new BindingsGroup();
        Vector3 m_InitialPosition;
        bool m_IsFirstFrame;

        /// <summary>
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        protected void Awake()
        {
            m_MultiPokeStateDataProvider =
GetComponentInParent<IMultiPokeStateDataProvider>();
            if(m_MultiPokeStateDataProvider == null)
                m_PokeDataProvider =
GetComponentInParent<IPokeStateDataProvider>();
        }

        /// <summary>
```

```
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        protected void Start()
        {
            if (m_PokeFollowTransform != null)
            {
                m_InitialPosition = m_PokeFollowTransform.localPosition;
m_BindingsGroup.AddBinding(m_TransformTweenableVariable.Subscribe(OnTran
sformTweenableVariableUpdated));

                if(m_MultiPokeStateDataProvider != null)
m_BindingsGroup.AddBinding(m_MultiPokeStateDataProvider.GetPokeStateData
ForTarget(transform).Subscribe(OnPokeStateDataUpdated));
                else if(m_PokeDataProvider != null)
m_BindingsGroup.AddBinding(m_PokeDataProvider.pokeStateData.SubscribeAnd
Update(OnPokeStateDataUpdated));
            }
            else
            {
                enabled = false;
                Debug.LogWarning($"Missing Poke Follow Transform
assignment on {this}. Disabling component.", this);
            }
        }

        /// <summary>
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        protected void OnDestroy()
        {
            m_BindingsGroup.Clear();
            m_TransformTweenableVariable?.Dispose();
        }

        /// <summary>
        /// See <see cref="MonoBehaviour"/>.
        /// </summary>
        protected void LateUpdate()
        {
            if (m_IsFirstFrame)
            {
                m_TransformTweenableVariable.HandleTween(1f);
                m_IsFirstFrame = false;
                return;
            }
            m_TransformTweenableVariable.HandleTween(m_SmoothingSpeed >
0f ? Time.deltaTime * m_SmoothingSpeed : 1f);
        }
```

```
        void OnTransformTweenableVariableUpdated(float3 position)
        {
            m_PokeFollowTransform.localPosition = position;
        }

        void OnPokeStateDataUpdated(PokeStateData data)
        {
            var pokeTarget = data.target;
            var applyFollow = m_ApplyIfChildIsTarget
                ? pokeTarget != null && pokeTarget.IsChildOf(transform)
                : pokeTarget == transform;

            if (applyFollow)
            {
                var targetPosition =
pokeTarget.InverseTransformPoint(data.axisAlignedPokeInteractionPoint);
                if (m_ClampToMaxDistance && targetPosition.sqrMagnitude
> m_MaxDistance * m_MaxDistance)
                    targetPosition =
Vector3.ClampMagnitude(targetPosition, m_MaxDistance);

                m_TransformTweenableVariable.target = targetPosition;
            }
            else if (m_ReturnToInitialPosition)
            {
                m_TransformTweenableVariable.target = m_InitialPosition;
            }
        }

        public void ResetFollowTransform()
        {
            if (!m_ClampToMaxDistance || m_PokeFollowTransform == null)
                return;

            m_PokeFollowTransform.localPosition = m_InitialPosition;
        }
    }
}
using UnityEngine;

public class ColorChangeScript : MonoBehaviour
{
    public GameObject targetObject; // Reference to the object to change
color
    public float changeInterval = 30.0f; // Time interval for color
change
    private float timeSinceLastChange = 0.0f; // Time elapsed since the
```

```
last color change
    private Renderer objectRenderer; // Reference to the target object's
renderer

    void Start()
    {
        // Get the Renderer component of the target object
        objectRenderer = targetObject.GetComponent<Renderer>();

        // Initialize the time since last change to a random value
within the interval
        timeSinceLastChange = Random.Range(0.0f, changeInterval);
    }

    void Update()
    {
        // Update the time elapsed
        timeSinceLastChange += Time.deltaTime;

        // Check if it's time to change the color
        if (timeSinceLastChange >= changeInterval)
        {
            // Generate a random color
            ChangeColor();

            // Reset the time since last change
            timeSinceLastChange = 0.0f;
        }
    }

    public void ChangeColor()
    {
        Color randomColor = new Color(Random.value, Random.value,
Random.value);

        // Change the target object's material color to the random color
        objectRenderer.material.color = randomColor;
    }
}
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using UnityEngine;

public class ComponentRuntimeController : MonoBehaviour
{
```

```
    [SerializeField] public UnityEngine.Object customObject;
    public Dictionary<string, FieldInfo> publicVariables;
    public Dictionary<string, MethodInfo> publicMethods;
    //public List<FieldInfo> publicVariables;
    public List<FieldInfo> privateVariables;

    private void Awake() {
        publicVariables = new Dictionary<string, FieldInfo>();
        publicMethods = new Dictionary<string, MethodInfo>();
        privateVariables = new List<FieldInfo>();
        //if (customObject == null) customObject =
GetComponent<UnityEngine.Object>();
    }

    public void SetCustomObject(UnityEngine.Object obj) {
        customObject = obj;
        PopulateClassProperties();
    }

    void PopulateClassProperties()
    {
        if (customObject != null) {
            foreach (FieldInfo ft in
customObject.GetType().GetFields(BindingFlags.Public |
BindingFlags.NonPublic |
                          BindingFlags.Instance)) {
                Debug.Log($"Public variable name {ft.Name} and type
{ft.FieldType}");
                publicVariables.Add(ft.Name, ft);
            }
            foreach (FieldInfo ft in
customObject.GetType().GetFields(BindingFlags.NonPublic |
                          BindingFlags.Instance)) {
                Debug.Log($"Private variable name {ft.Name} and type
{ft.FieldType}");
                privateVariables.Add(ft);
            }
            foreach (MethodInfo mI in
customObject.GetType().GetMethods(BindingFlags.Public |
BindingFlags.Instance))
            {
                Debug.Log($"public function name {mI.Name} and type
{mI.ReturnType}");
                // For now, only add if not duplicate instance
                if (!publicMethods.ContainsKey(mI.Name))
                    publicMethods.Add(mI.Name, mI);
            }
            //return from p in customObject.GetType().GetFields()
```

```
            //       where p.FieldType == typeof(T)
            //       select new KeyValuePair<string, T>(p.Name,
(T)p.GetValue(obj));
        }
    }


    public string GetValueOfPublicVariable(string varName) {
        return
publicVariables[varName].GetValue(customObject).ToString();
    }

    public void SetValueOfPublicVariable(string varName, object newVal)
{
        publicVariables[varName].SetValue(customObject, newVal);
    }

    public Type GetTypeOfPublicVariable(string varName) {
        return
publicVariables[varName].GetValue(customObject).GetType();
    }
    /** METHODS **/
    public MethodInfo GetPublicMethod(string methodName) {
        return publicMethods[methodName];
    }

    public ParameterInfo[] GetParameterTypesOfPublicMethod(string
methodName) {
        return publicMethods[methodName].GetParameters();
    }

    public void InvokePublicMethod(string methodName) {
        // for now, only invoke if method contains no parameters
        if (GetParameterTypesOfPublicMethod(methodName).Length == 0) {
            publicMethods[methodName].Invoke(customObject, new object[]
{ });
        }
    }

    public Type GetReturnTypeOfPublicMethod(string methodName) {
        return publicMethods[methodName].ReturnType;
    }
}
using UnityEngine;
using UnityEditor;
using System.IO;
using System.Text;
using iTextSharp.text;
```

```
using iTextSharp.text.pdf;
using Mono.Cecil.Cil;
using System;

public class GPTDocsGenerator : EditorWindow
{
    [MenuItem("Tools/Generate PDF from C# Files")]
    public static void ShowWindow()
    {
        GetWindow<GPTDocsGenerator>("PDF Generator");
    }

    private string inputDirectory = "Assets"; // Default input directory
is the "Assets" folder
    private string outputFilePath = "GeneratedPDF.pdf"; // Default
output PDF file path

    private void OnGUI()
    {
        GUILayout.Label("PDF Generator", EditorStyles.boldLabel);

        GUILayout.Space(10);

        inputDirectory = EditorGUILayout.TextField("Input Directory",
inputDirectory);
        outputFilePath = EditorGUILayout.TextField("Output PDF File",
outputFilePath);

        GUILayout.Space(20);

        if (GUILayout.Button("Generate PDF"))
        {
            GeneratePDF();
        }
    }

    private void GeneratePDF()
    {
        iTextSharp.text.Document doc = new iTextSharp.text.Document();

        try
        {
            // Get the path to the StreamingAssets folder
            //string streamingAssetsPath =
Path.Combine(Application.dataPath, "StreamingAssets");
            //string outputPath = Path.Combine(streamingAssetsPath,
outputFilePath);
            string streamingAssetsPath =
```

```
Application.streamingAssetsPath;
            string outputPath = Path.Combine(streamingAssetsPath,
outputFilePath);

            // Create the StreamingAssets folder if it doesn't exist
            if (!Directory.Exists(streamingAssetsPath))
            {
                Directory.CreateDirectory(streamingAssetsPath);
            }

            PdfWriter writer = PdfWriter.GetInstance(doc, new
FileStream(outputPath, FileMode.Create));

            doc.Open();

            AppendCSharpFilesToPDF(doc, inputDirectory);

            doc.Close();

            EditorUtility.DisplayDialog("PDF Generated", "PDF file
created and saved to: " + outputPath, "OK");
        }
        catch (Exception e)
        {
            Debug.LogError("Error: " + e.Message);
        }
    }


    private void AppendCSharpFilesToPDF(iTextSharp.text.Document doc,
string directory)
    {
        string[] csharpFiles = Directory.GetFiles(directory, "*.cs");

        foreach (string csharpFile in csharpFiles)
        {
            StreamReader reader = new StreamReader(csharpFile,
Encoding.UTF8);
            string fileContent = reader.ReadToEnd();
            reader.Close();

            Paragraph paragraph = new Paragraph();
            paragraph.Font = FontFactory.GetFont(FontFactory.COURIER,
12f);
            paragraph.Add(fileContent);

            doc.Add(paragraph);
        }
```

```
        string[] subDirectories = Directory.GetDirectories(directory);

        foreach (string subDirectory in subDirectories)
        {
            AppendCSharpFilesToPDF(doc, subDirectory);
        }
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using OpenAI.Chat;
using OpenAI;
using System.Threading;
using Unity.Collections.LowLevel.Unsafe;
using System;
using UnityEngine.EventSystems;
using OpenAI.Models;
using System.Threading.Tasks;
using System.Reflection;
using System.Text;
using System.Linq;
using OpenAI.Samples.Chat;
using UnityEngine.UIElements;
using System.Text.RegularExpressions;
using OpenAI.Threads;
using UnityEditor.VersionControl;
using Utilities.WebRequestRest;

public class GPTReflectionAnalysis : MonoBehaviour
{
    public ChatBehaviour chatBehaviour;
    public ReflectionRuntimeController componentController; // Reference
to your component controller
    private OpenAIClient openAI; // OpenAI Client

    private void Start()
    {
        // Initialize the OpenAI Client
        openAI = new OpenAIClient();
    }

    private void Update()
    {
        // Check if the 'Q' key is pressed
        if (Input.GetKeyDown(KeyCode.Q))
        {
```

```
            // Call the AnalyzeComponents method
            Debug.Log("running task");
            AnalyzeComponents();
        }
    }

    private async void AnalyzeComponents()
    {
        // Format the data from your ComponentRuntimeController into a
string for GPT analysis
        string dataForGPT =
FormatDataForGPT(componentController.classCollection);

        // Pre-prompt for the GPT query
        string gptPrompt = "Given the following snapshot of the runtime
environment with classes, methods, and variables, can you analyze the
relationships among these components and their runtime values?";

        // Combine the prompt with the data
        string combinedMessage = $"{gptPrompt}\n{dataForGPT}";

        Debug.Log(combinedMessage);
        // Create a message list for the chat request
        var messages = new List<OpenAI.Chat.Message>
        {
            new OpenAI.Chat.Message(Role.System, combinedMessage),
        };

        try
        {
            var chatRequest = new ChatRequest(messages,
Model.GPT3_5_Turbo);
            var result = await
openAI.ChatEndpoint.GetCompletionAsync(chatRequest);
            var response = result.ToString();

            //Debug.Log(response);

            ProcessGPTResponse(response);
        }
        catch (Exception e)
        {
            Debug.LogError(e);
        }
        finally
        {
            //if (lifetimeCancellationTokenSource != null) {}
            //isChatPending = false;
```

```
        }

    }


    /// <summary>
    /// invoke via button press
    /// </summary>
    public void SubmitChat()
    {
        if (ParseKeyword(chatBehaviour.inputField.text))
        {
//componentController.SearchFunctions(ParseFunctionName(chatBehaviour.in
putField.text));
            Debug.Log($"Keyword found");
        } else
        {
            chatBehaviour.SubmitChat(chatBehaviour.inputField.text);
        }
    }

    private string FormatDataForGPT(Dictionary<string, ClassInfo>
classCollection)
    {
        StringBuilder formattedData = new StringBuilder();

        foreach (var classEntry in classCollection)
        {
            formattedData.AppendLine($"Class: {classEntry.Key}");

            formattedData.AppendLine("Methods:");
            foreach (var method in classEntry.Value.Methods)
            {
                formattedData.AppendLine($"- {method.Key}: Parameters:
{string.Join(", ", method.Value.GetParameters().Select(p =>
p.ParameterType.Name + " " + p.Name))}, Return Type:
{method.Value.ReturnType.Name}");
            }

            formattedData.AppendLine("Variables:");
            foreach (var variable in classEntry.Value.Variables)
            {
                // Retrieve the runtime value of the variable
                object value =
classEntry.Value.VariableValues.TryGetValue(variable.Key, out object
val) ? val : "Unavailable";
                formattedData.AppendLine($"- {variable.Key}: Type:
{variable.Value.FieldType.Name}, Value: {value}");
```

```
            }

            formattedData.AppendLine(); // Separator for readability
        }

        return formattedData.ToString();
    }


    private void ProcessGPTResponse(string gptResponse)
    {
        // Process the GPT response to extract useful information
        // ...

        Debug.Log("GPT Analysis:\n" + gptResponse);
        chatBehaviour.UpdateChat(gptResponse);
    }


    public void UpdateChat(string newText)
    {
        chatBehaviour.conversation.AppendMessage(new
OpenAI.Chat.Message(Role.Assistant, newText));
        //inputField.text = newText;
        var assistantMessageContent =
chatBehaviour.AddNewTextMessageContent(Role.Assistant);
        assistantMessageContent.text = newText;
        chatBehaviour.scrollView.verticalNormalizedPosition = 0f;

    }

    /// <summary>
    /// Anytime submitchat is invoked, we first search for keywords
    /// </summary>
    /// <param name="_tex"></param>
    /// <returns></returns>
    public bool ParseKeyword(string _text)
    {
        Debug.Log($"Input text {_text}");
        if (_text.Contains("invoke function "))
        {
            string _func = ParseFunctionName(_text);
            if (!string.IsNullOrEmpty(_func))
            {
                Debug.Log($"Function name {_func}");
                componentController.SearchFunctions(_func);
                return true;
            }
```

```
        }
        else if (_text.Contains("view variables of "))
        {
            string className = ParseClassName(_text, "view variables of
");
            if (!string.IsNullOrEmpty(className))
            {
                Debug.Log($"Viewing variables of class {className}");
                //componentController.PrintAllVariableValues(className);
                string localQueryResponse =
componentController.GetAllVariableValuesAsString(className);
                UpdateChat(localQueryResponse);
                //chatBehaviour.GenerateSpeech(localQueryResponse);
                return true;
            }
        }
        else if (_text.Contains("view variable "))
        {
            string variableName = ParseVariableName(_text, "view
variable ");
            if (!string.IsNullOrEmpty(variableName))
            {
                Debug.Log($"Viewing variable {variableName}");
componentController.PrintVariableValueInAllClasses(variableName);
        return true;
            }
        }
        return false;
    }


    public string ParseFunctionName(string input)
    {
        // Define a regular expression pattern to match "invoke
function" followed by a function name in parentheses
        string pattern = @"invoke\s+function\s+([A-Za-z_][A-Za-z0-
9_]*)\s*\(";

        // Use Regex to find a match
        Match match = Regex.Match(input, pattern);
        Debug.Log("attempt to regex match");
        if (match.Success)
        {
            // Extract and return the function name from the matched
group
            return match.Groups[1].Value;
        }
        else
```

```
        {
            // If no match is found, return null or an empty string,
depending on your preference
            return null;
        }
    }

    public string ParseClassName(string input, string patternStart)
    {
        string pattern = patternStart + @"([A-Za-z_][A-Za-z0-9_]*)";
        Match match = Regex.Match(input, pattern);
        if (match.Success)
        {
            return match.Groups[1].Value;
        }
        return null;
    }

    public string ParseVariableName(string input, string patternStart)
    {
        return ParseClassName(input, patternStart); // Reusing the same
logic as class name parsing
    }


}
using UnityEngine;
using System.Reflection;
using System.Collections.Generic;
using System;
using System.Text.RegularExpressions;
using System.Text;
using System.Collections;

public class ReflectionRuntimeController : MonoBehaviour
{
    public float detectionRadius = 5f; // Radius for proximity detection
    public LayerMask detectionLayer; // Layer mask to filter which
objects to detect
    [SerializeField] public Dictionary<string, ClassInfo>
classCollection = new Dictionary<string, ClassInfo>();
    [SerializeField] public UnityEngine.Object customObject;

    void Update()
    {
        // Check for space bar press
        if (Input.GetKeyDown(KeyCode.Space))
        {
```

```
            ScanAndPopulateClasses();
        }
        //if (Input.GetKeyDown(KeyCode.RightArrow))
        //{
        //    SetCustomObject(FindObjectOfType<ColorChangeScript>());
        //    InvokePublicMethod("ColorChangeScript", "ChangeColor");
        //}
    }

    void ScanAndPopulateClasses()
    {
        Debug.Log("Scanning...");
        // Clearing existing data
        classCollection.Clear();

        // Find all colliders within the specified radius
        Collider[] colliders = Physics.OverlapSphere(transform.position,
detectionRadius, detectionLayer);
        foreach (Collider collider in colliders)
        {
            GameObject obj = collider.gameObject;
            PopulateClassInfo(obj);
        }
        Debug.Log($"Total {colliders.Length}");
    }

    void PopulateClassInfo(GameObject obj)
    {
        MonoBehaviour[] monoBehaviours =
obj.GetComponents<MonoBehaviour>();
        foreach (MonoBehaviour monoBehaviour in monoBehaviours)
        {
            var type = monoBehaviour.GetType();
            var classInfo = new ClassInfo();

            // Populate methods
            foreach (var method in type.GetMethods(BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance |
BindingFlags.DeclaredOnly))
            {
                classInfo.Methods[method.Name] = method;
            }

            //foreach (var field in type.GetFields(BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance |
BindingFlags.DeclaredOnly))
            foreach (var field in type.GetFields(BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance))
```

```
            {
                classInfo.Variables[field.Name] = field;
                // Retrieve and store the current value of the variable
                object value = field.GetValue(monoBehaviour);
                classInfo.VariableValues[field.Name] = value;
            }

            // Add to class collection
            if (!classCollection.ContainsKey(type.Name))
            {
                classCollection[type.Name] = classInfo;
            }
        }
    }


    public void ParseKeyword(string _tex)
    {
        if (_tex.Contains("invoke function "))
        {
            string _func = ParseFunctionName(_tex);
            if (_func != null || _func != "")
            {
                Debug.Log($"Function name {_func}");

            }

        }
    }

    public string ParseFunctionName(string input)
    {
        // Define a regular expression pattern to match "invoke
function" followed by a function name in parentheses
        string pattern = @"invoke\s+function\s+([A-Za-z_][A-Za-z0-
9_]*)\s*\(";

        // Use Regex to find a match
        Match match = Regex.Match(input, pattern);
        Debug.Log("attempt to regex match");
        if (match.Success)
        {
            // Extract and return the function name from the matched
group
            return match.Groups[1].Value;
        }
        else
        {
```

```
            // If no match is found, return null or an empty string,
depending on your preference
            return null;
        }
    }

    public void SearchFunctions(string func)
    {
        Debug.Log($"Searching functions {func}");
        foreach (string _class in classCollection.Keys)
        {
            foreach (string _func in
classCollection[_class].Methods.Keys)
            {
                if (_func == func)
                {
                    Debug.Log($"Found function {_func}");
SetCustomObject(FindObjectOfType(Type.GetType(_class)));
   InvokePublicMethod(_class, _func);
                }
            }
        }
    }

    // TODO: Make it where ChatGPT can return responses informing the
runtime as to what methods to invoke if
    // user asks what methods to call for invoking a chain of events
    protected ParameterInfo[] GetParameterTypesOfPublicMethod(string
className, string methodName)
    {
        return
classCollection[className].Methods[methodName].GetParameters();
    }

    public void InvokePublicMethod(string className, string methodName)
    {
        // for now, only invoke if method contains no parameters
        if (GetParameterTypesOfPublicMethod(className,
methodName).Length == 0)
        {
classCollection[className].Methods[methodName].Invoke(customObject, new
object[] { });
        }
    }

    public void PrintAllVariableValues(string className)
    {
        if (classCollection.TryGetValue(className, out ClassInfo
```

```
classInfo))
        {
            Debug.Log($"All variables in class {className}:");
            foreach (var variable in classInfo.Variables)
            {
                object value =
classInfo.VariableValues.TryGetValue(variable.Key, out object val) ? val
: "Unavailable";
                Debug.Log($"- {variable.Key}: {value}");
            }
        }
        else
        {
            Debug.Log($"Class {className} not found.");
        }
    }


    public string GetAllVariableValuesAsString(string className)
    {
        if (classCollection.TryGetValue(className, out ClassInfo
classInfo))
        {
            StringBuilder variableValues = new StringBuilder();
            variableValues.AppendLine($"All variables in class
{className}:");

            foreach (var variable in classInfo.Variables)
            {
                object variableValue =
classInfo.VariableValues.TryGetValue(variable.Key, out object val) ? val
: "Unavailable";

                // Check if the variable is a dictionary
                if (variableValue is IDictionary dictionary)
                {
                    variableValues.AppendLine($"- {variable.Key}
(Dictionary):");
                    foreach (DictionaryEntry entry in dictionary)
                    {
                        variableValues.AppendLine($"    - Key:
{entry.Key}, Value: {entry.Value}");
                    }
                }
                else
                {
                    variableValues.AppendLine($"- {variable.Key}:
{variableValue}");
```

```
                }
            }

            return variableValues.ToString();
        }
        else
        {
            return $"Class {className} not found.";
        }
    }



    public void PrintVariableValueInAllClasses(string variableName)
    {
        bool variableFound = false;

        foreach (var classEntry in classCollection)
        {
            string className = classEntry.Key;
            ClassInfo classInfo = classEntry.Value;

            if (classInfo.Variables.TryGetValue(variableName, out
FieldInfo fieldInfo))
            {
                object value =
classInfo.VariableValues.TryGetValue(variableName, out object val) ? val
: "Unavailable";
                Debug.Log($"Variable {variableName} found in class
{className}: {value}");
                variableFound = true;
            }
        }

        if (!variableFound)
        {
            Debug.Log($"Variable {variableName} not found in any
class.");
        }
    }



    public void SetCustomObject(UnityEngine.Object obj) => customObject
= obj;

}

[System.Serializable]
```

```
public class ClassInfo
{
    public Dictionary<string, MethodInfo> Methods { get; set; }
    public Dictionary<string, FieldInfo> Variables { get; set; }
    public Dictionary<string, object> VariableValues { get; set; } //
Store runtime values

    public ClassInfo()
    {
        Methods = new Dictionary<string, MethodInfo>();
        Variables = new Dictionary<string, FieldInfo>();
        VariableValues = new Dictionary<string, object>();
    }
}
using UnityEngine;

public class RotationScript : MonoBehaviour
{
    public float rotationSpeed = 30.0f; // Adjust the rotation speed in
the Unity Editor

    void Update()
    {
        // Rotate the object around its Y-axis
        transform.Rotate(Vector3.up * rotationSpeed * Time.deltaTime);
    }
}
```