```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class MessageColorMode : MonoBehaviour
{
    public enum MessageType
    {
        Sender,
        Reciever
    }

    [SerializeField]
    public MessageUIProperties senderMessage, receiverMessage;
    public RectTransform rectTransform;
    public TextMeshProUGUI messageText;

    public MessageType messageType;
    public Image image;
    public void SetMessageColor() => image.color = messageType ==
MessageType.Sender ? senderMessage.color : receiverMessage.color;
    public void SetMode(MessageType _messageType)
    {
        messageType = _messageType;
        SetMessageColor();
        // rectTransform.offsetMin = messageType == MessageType.Sender ?
senderMessage.offset : receiverMessage.offset;

        // Get the current anchored position
        if (messageType == MessageType.Sender)
        {
            Vector2 anchoredPosition = rectTransform.anchoredPosition;
            anchoredPosition.x += 80f;
            rectTransform.anchoredPosition = anchoredPosition;
        }
    }

    public void SetMessageText(string text)
    {
        messageText.SetText(text);
    }


    public void Awake()
    {
        if (image == null) image = GetComponent<Image>();
        if (rectTransform == null) rectTransform =
GetComponent<RectTransform>();
    }

    private void Start() => SetMode(messageType);

    public string debugText;
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.C)) SetMessageText(debugText);
    }
}

[System.Serializable]
public class MessageUIProperties
{
    public Color32 color;
    public Vector2 offset;
}
using UnityEngine;
using UnityEditor;
using System;
using System.Linq;
using OpenAI;
using OpenAI.Assistants;
using System.Threading.Tasks;
using OpenAI.Files;
using OpenAI.Threads;

[Serializable]
public class FileReference
{
    public string assetPath;
    public bool markedForRemoval;
}

public class GPTAssistantBuilder : EditorWindow
{
    private FileReference[] files = new FileReference[0];
    private const string EditorPrefKey = "GPTAssistantFiles";
    private float feedbackTimer = 0f;
    private float feedbackDuration = 1.0f;

    private string assistantId;
    private string fileID;

    [MenuItem("Tools/GPT Assistant Builder")]
    public static void ShowWindow()
    {
        GetWindow<GPTAssistantBuilder>("GPT Assistant Builder");
    }

    private void OnEnable()
    {
        LoadFilesFromEditorPrefs();
        LoadAssistantId();
        if (!string.IsNullOrEmpty(assistantId))
        {
            DisplayAssistantDetails(assistantId);
        }
        else
        {
            ListAssistants();
        }
    }

    private void OnDisable()
    {
        SaveFilesToEditorPrefs();
    }

    private void OnGUI()
    {
        GUILayout.Label("GPT Assistant Builder",
EditorStyles.boldLabel);

        GUILayout.Space(10);

        EditorGUILayout.LabelField("Drag and drop files here:");

        Rect dropArea = GUILayoutUtility.GetRect(0f, 50f,
GUILayout.ExpandWidth(true));
        GUI.Box(dropArea, "Drag and drop files here");
        UnityEngine.Event currentEvent = UnityEngine.Event.current;

        if (currentEvent.type == EventType.DragUpdated)
        {
            DragAndDrop.visualMode = DragAndDropVisualMode.Copy;
        }

        if (currentEvent.type == EventType.DragPerform)
        {
            DragAndDrop.AcceptDrag();
            feedbackTimer = Time.realtimeSinceStartup;

            foreach (UnityEngine.Object draggedObject in
DragAndDrop.objectReferences)
            {
                string assetPath =
AssetDatabase.GetAssetPath(draggedObject);

                if (!files.Any(file => file.assetPath == assetPath))
                {
                    Array.Resize(ref files, files.Length + 1);
                    files[files.Length - 1] = new FileReference {
assetPath = assetPath };
                }
            }
        }

        if (Time.realtimeSinceStartup - feedbackTimer <
feedbackDuration)
        {
            EditorGUI.DrawRect(dropArea, new Color(0.5f, 1f, 0.5f,
0.5f));
        }

        GUILayout.Space(20);

        for (int i = 0; i < files.Length; i++)
        {
            EditorGUILayout.BeginHorizontal();

            files[i].markedForRemoval =
EditorGUILayout.Toggle(files[i].markedForRemoval, GUILayout.Width(20));
            EditorGUILayout.LabelField("Added File:",
AssetDatabase.LoadAssetAtPath<UnityEngine.Object>(files[i].assetPath).na
me);

            if (GUILayout.Button("Remove", GUILayout.Width(80)))
            {
                RemoveFileAtIndex(i);
            }

            EditorGUILayout.EndHorizontal();
        }

        GUILayout.Space(20);

        if (GUILayout.Button("Save Files to EditorPrefs"))
        {
            SaveFilesToEditorPrefs();
        }
        // Display assistant details
        if (!string.IsNullOrEmpty(assistantId))
        {
            GUILayout.Label($"Assistant ID: {assistantId}");
            // Display other assistant details
        }

        if (GUILayout.Button("Create New Assistant"))
        {
            CreateAssistant();
        }

        if (GUILayout.Button("Attach Files to Assistant"))
        {
            AttachFilesToAssistant();
        }

        if (!string.IsNullOrEmpty(assistantId) &&
GUILayout.Button("Delete Assistant"))
        {
            DeleteAssistant();
        }
    }

    private void RemoveFileAtIndex(int index)
    {
        if (index >= 0 && index < files.Length)
        {
            FileReference[] newFiles = new FileReference[files.Length -
1];
            int newIndex = 0;

            for (int i = 0; i < files.Length; i++)
            {
                if (i != index)
                {
                    newFiles[newIndex] = files[i];
                    newIndex++;
                }
            }

            files = newFiles;
        }
    }

    private void SaveFilesToEditorPrefs()
    {
        for (int i = 0; i < files.Length; i++)
        {
            Debug.Log("Saved File: " +
AssetDatabase.LoadAssetAtPath<UnityEngine.Object>(files[i].assetPath).na
me);
        }

        string[] assetPaths = files.Select(file =>
file.assetPath).ToArray();
        bool[] removalStates = files.Select(file =>
file.markedForRemoval).ToArray();

        EditorPrefs.SetString(EditorPrefKey + "Paths", string.Join(",",
assetPaths));
        EditorPrefs.SetString(EditorPrefKey + "States", string.Join(",",
removalStates));
        EditorPrefs.SetInt(EditorPrefKey + "Count", files.Length);
    }

    private void LoadFilesFromEditorPrefs()
    {
        if (EditorPrefs.HasKey(EditorPrefKey + "Paths") &&
EditorPrefs.HasKey(EditorPrefKey + "States") &&
EditorPrefs.HasKey(EditorPrefKey + "Count"))
        {
            string[] assetPaths = EditorPrefs.GetString(EditorPrefKey +
"Paths").Split(',');
            string[] removalStateStrings =
EditorPrefs.GetString(EditorPrefKey + "States").Split(',');
            int count = EditorPrefs.GetInt(EditorPrefKey + "Count");

            FileReference[] loadedFiles = new FileReference[count];

            for (int i = 0; i < count; i++)
            {
                bool markedForRemoval =
bool.Parse(removalStateStrings[i]);
                loadedFiles[i] = new FileReference { assetPath =
assetPaths[i], markedForRemoval = markedForRemoval };
            }

            files = loadedFiles;
        }
    }

    // Method to save and load assistant ID
    private void LoadAssistantId()
    {
        assistantId = EditorPrefs.GetString("AssistantId", "");
    }

    private void DisplayAssistantDetails(string assistantId)
    {
        // Fetch and display details like ID, name, date created, file
names
        // You'll need to call RetrieveAssistant and ListAssistantFiles
    }

    public async void ListAssistants()
    {
        var api = new OpenAIClient();
        var assistantsList = await
api.AssistantsEndpoint.ListAssistantsAsync();

        foreach (var assistant in assistantsList.Items)
        {
            Debug.Log($"{assistant} -> {assistant.CreatedAt}, ID
{assistant.Id}");
        }
    }

    public async void CreateAssistant()
    {
        var api = new OpenAIClient();
        var retrievalTools = new OpenAI.Tool[1];
        retrievalTools[0] = OpenAI.Tool.Retrieval;
        var request = new CreateAssistantRequest(model: "gpt-4-1106-
preview", name: "test", tools: retrievalTools);
        var assistant = await
api.AssistantsEndpoint.CreateAssistantAsync(request);
        Debug.Log($"Assistant ID {assistant.Id}");

        // Save the assistant ID
        EditorPrefs.SetString("AssistantId", assistant.Id);
        assistantId = assistant.Id;

        await Task.Delay(2000);
        /*
        // Implement file upload and attachment logic
        foreach (var fileRef in files)
        {
            //var fileUploadRequest = new
FileUploadRequest(fileRef.assetPath, "assistant");
            //var file = await
api.FilesEndpoint.UploadFileAsync(fileUploadRequest);
            //var assistantFile = await
api.AssistantsEndpoint.AttachFileAsync(assistantId, file);
```

```
            var assistantFile = await
assistant.UploadFileAsync(fileRef.assetPath);
            await Task.Delay(1000);
        }*/
        Debug.Log("Sent off api requests to create assistant");
    }



    public async Task<AssistantResponse> RetrieveAssistant()
    {
        var api = new OpenAIClient();
        var assistant = await
api.AssistantsEndpoint.RetrieveAssistantAsync(assistantId);
        Debug.Log($"{assistant} -> {assistant.CreatedAt}, ID
{assistant.Id}");
        return assistant;
    }

    public async void DeleteAssistant()
    {
        var api = new OpenAIClient();
        if (assistantId != string.Empty)
        {
            var isDeleted = await
api.AssistantsEndpoint.DeleteAssistantAsync(assistantId);
            Debug.Log($"Deleted assistant {assistantId}");
        }
        assistantId = string.Empty;
        EditorPrefs.SetString("AssistantId", "");
    }

    public async void ListAssistantFiles()
    {
        var api = new OpenAIClient();
        var filesList = await
api.AssistantsEndpoint.ListFilesAsync(assistantId);

        foreach (var file in filesList.Items)
        {
            Debug.Log($"{file.AssistantId}'s file -> {file.Id}");
        }
    }

    public async void AttachFilesToAssistant()
    {
        /*var api = new OpenAIClient();
        foreach (var fileRef in files)
        {
            string absoluteFilePath =
System.IO.Path.GetFullPath(fileRef.assetPath);
            Debug.Log($"Uploading file: {absoluteFilePath}");

            var fileUploadRequest = new
FileUploadRequest(absoluteFilePath, "assistant");
            var file = await
api.FilesEndpoint.UploadFileAsync(fileUploadRequest);
            Debug.Log($"Uploaded file ID: {file.Id}");

            var assistantFile = await
api.AssistantsEndpoint.AttachFileAsync(assistantId, file);
            Debug.Log($"Attached file {file.Id} to assistant
{assistantId}");

            await Task.Delay(1000); // Delay for 1 second
        }*/
        var api = new OpenAIClient();
        var fileData = await
api.FilesEndpoint.UploadFileAsync(files[0].assetPath, "assistants");
        var assistantFile = await
api.AssistantsEndpoint.AttachFileAsync(assistantId, new
FileResponse(fileData.Id, fileData.Object, fileData.Size,
fileData.CreatedUnixTimeSeconds, fileData.FileName, fileData.Purpose,
fileData.Status));
        Debug.Log($"Attached file {fileData.Id} to assistant
{assistantId}");
    }



}
using UnityEngine;
using UnityEditor;
using System.IO;
using System.Text;
using iTextSharp.text;
using iTextSharp.text.pdf;
using System;
using System.Threading.Tasks;

public class GPTDocsGenerator : EditorWindow {
    [MenuItem("Tools/DopeCoder/Scan Codebase")]
    public static void ShowWindow() {
        GetWindow<GPTDocsGenerator>("PDF Generator");
    }
    private string inputDirectory = "Assets"; // Default input directory
is the "Assets" folder
    private string outputFileName = "GeneratedPDF.pdf"; // Default
output PDF file name
    private string outputFolderPath; // Output folder path
    private float progress = 0;
    private bool isGenerating = false;

    private void OnEnable() {
        // Determine the script's directory
        string scriptPath =
AssetDatabase.GetAssetPath(MonoScript.FromScriptableObject(this));
        string scriptDirectory = Path.GetDirectoryName(scriptPath);

        // Set the output folder path to a "FileScans" folder within the
script's directory
        outputFolderPath = Path.Combine(scriptDirectory, "FileScans");

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

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

        GUILayout.Space(10);

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

        GUILayout.Space(20);

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

        if (isGenerating) {
            EditorGUILayout.LabelField("Progress:",
EditorStyles.boldLabel);
            Rect rect = EditorGUILayout.GetControlRect(false, 20);
            EditorGUI.ProgressBar(rect, progress, "Generating PDF");
            GUILayout.Space(20);
        }
    }
    private async void GeneratePDFAsync() {
        isGenerating = true;

        var progressIndicator = new Progress<float>(value => {
            progress = value;
            Repaint();
        });

        await Task.Run(() => {
            GeneratePDF(progressIndicator);
        });

        isGenerating = false;
        progress = 0;
        EditorUtility.DisplayDialog("PDF Generated", "PDF file created
successfully!", "OK");
        Repaint();
    }

    private void GeneratePDF(IProgress<float> progress) {
        iTextSharp.text.Document doc = new iTextSharp.text.Document();

        try {
            string outputPath = Path.Combine(outputFolderPath,
outputFileName);

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

            doc.Open();

            AppendCSharpFilesToPDF(doc, inputDirectory, progress);

            doc.Close();
        } catch (Exception e) {
            Debug.LogError("Error: " + e.Message);
        }
    }

    private void AppendCSharpFilesToPDF(iTextSharp.text.Document doc,
string directory, IProgress<float> progress) {
        string[] csharpFiles = Directory.GetFiles(directory, "*.cs");
        string[] subDirectories = Directory.GetDirectories(directory);
        int totalFiles = csharpFiles.Length + subDirectories.Length;
        int processedFiles = 0;

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

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

                doc.Add(paragraph);
            }

            processedFiles++;
            progress.Report((float)processedFiles / totalFiles);
        }

        foreach (string subDirectory in subDirectories) {
            AppendCSharpFilesToPDF(doc, subDirectory, progress);
        }
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;
using LogicUI.FancyTextRendering;
using NaughtyAttributes;

namespace FancyTextRendering.Demo
{
    public class DemoRenderUpdater : MonoBehaviour
    {
        [SerializeField] TMP_InputField MarkdownSourceInputField;
        [SerializeField] MarkdownRenderer MarkdownRenderer;

        private void Start()
        {
            MarkdownSourceInputField.onValueChanged.AddListener(_ =>
UpdateRender());
        }

        [Button]
        private void UpdateRender()
        {
            MarkdownRenderer.Source = MarkdownSourceInputField.text;
        }
    }
}
```

```
using System.Text;

namespace LogicUI.FancyTextRendering
{
    /// <summary>
    /// Allows efficient pre-processing of text before the main markdown
processing is applied. Use with <see
cref="Markdown.MarkdownToRichText(string, MarkdownRenderingSettings,
ICustomTextPreProcessor[])"/> or
    /// </summary>
    public interface ICustomTextPreProcessor
    {
        void ProcessLine(StringBuilder lineBuilder);
    }
}
using JimmysUnityUtilities;
using NaughtyAttributes;
using UnityEngine;

namespace LogicUI.FancyTextRendering
{
    [RequireComponent(typeof(MarkdownRenderer))]
    public class LoadMarkdownFromResources : MonoBehaviour
    {
        [SerializeField] string MarkdownResourcesPath;

        private void Awake()
        {
            LoadMarkdown();
        }

        [Button]
        private void LoadMarkdown()
        {
            string markdown =
ResourcesUtilities.ReadTextFromFile(MarkdownResourcesPath);
            GetComponent<MarkdownRenderer>().Source = markdown;
        }
    }
}
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using JimmysUnityUtilities;
using LogicUI.FancyTextRendering.MarkdownLogic;
using TMPro;
using UnityEngine;
using UnityEngine.Profiling;

namespace LogicUI.FancyTextRendering
{
    /// <summary>
    /// Converts markdown into TMP rich text tags.
    /// Very unfinished and experimental. Not even close to being a
complete markdown renderer.
    /// </summary>
    public static class Markdown
    {
        // Useful links for anyone who wants to finish this
        // http://digitalnativestudios.com/textmeshpro/docs/rich-text/
        // https://github.com/adam-p/markdown-here/wiki/Markdown-
Cheatsheet



        public static void RenderToTextMesh(string markdownSource,
TMP_Text textMesh)
            => RenderToTextMesh(markdownSource, textMesh,
MarkdownRenderingSettings.Default);

        public static void RenderToTextMesh(string markdownSource,
TMP_Text textMesh, MarkdownRenderingSettings settings, params
ICustomTextPreProcessor[] customTextPreProcessors)
        {
            string richText = MarkdownToRichText(markdownSource,
settings, customTextPreProcessors);

            textMesh.text = richText;
            UpdateTextMesh(textMesh);
        }

        public static void UpdateTextMesh(TMP_Text textMesh)
        {
            ResetLinkInfo(); // TextMeshPro doesn't reset the link infos
automatically, so we have to do it manually in situations where it will
be changed

            textMesh.ForceMeshUpdate();
            textMesh.GetComponent<TextLinkHelper>()?.LinkDataUpdated();


            void ResetLinkInfo()
            {
                if (textMesh.textInfo != null) // Make sure the text is
initialized; required as of TMP 2.1
                {
                    textMesh.textInfo.linkInfo =
Array.Empty<TMP_LinkInfo>();
                    textMesh.textInfo.linkCount = 0;
                }
            }
        }


        public static string MarkdownToRichText(string source)
            => MarkdownToRichText(source,
MarkdownRenderingSettings.Default);

        public static string MarkdownToRichText(string source,
MarkdownRenderingSettings settings, params ICustomTextPreProcessor[]
customTextPreProcessors)
        {
            if (source.IsNullOrEmpty())
                return String.Empty;


            Profiler.BeginSample(nameof(MarkdownToRichText));

            var lines = new List<MarkdownLine>();

            using (var reader = new StringReader(source))
            {
                string line;

                while ((line = reader.ReadLine()) != null)
                {
                    lines.Add(new MarkdownLine()
                    {
                        Builder = new StringBuilder(line)
                    });
                }
            }


            foreach (var processor in customTextPreProcessors)
            {
                foreach (var line in lines)
                    processor.ProcessLine(line.Builder);
            }


            foreach (var processor in BuiltInLineProcessors)
                processor.Process(lines, settings);
            var builder = new StringBuilder();

            foreach (var line in lines)
            {
                if (!line.DeleteLineAfterProcessing)
                    builder.AppendLine(line.Finish());
            }

            Profiler.EndSample();
            return builder.ToString();
        }


        private static readonly IReadOnlyList<MarkdownLineProcessorBase>
BuiltInLineProcessors = new MarkdownLineProcessorBase[]
        {
            // Order of processing here does matter, be mindful when
adding to this list.
            new AutoLinksHttp(),
            new AutoLinksHttps(),
            new UnorderedLists(),
            new OrderedLists(),
            new Bold(),
            new Italics(),
            new Strikethrough(),
            new SuperscriptChain(), // Important to process chain before
single!
            new SuperscriptSingle(),
            new Monospace(),
            new Headers(),
            new Links(),
        };
    }
}
using TMPro;
using UnityEngine;

namespace LogicUI.FancyTextRendering
{
    [RequireComponent(typeof(TMP_Text))]
    public class MarkdownRenderer : MonoBehaviour
    {
        [SerializeField]
        [TextArea(minLines: 10, maxLines: 50)]
        string _Source;

        public string Source
        {
            get => _Source;
            set
            {
                _Source = value;
                RenderText();
            }
        }

        TMP_Text _TextMesh;
        public TMP_Text TextMesh
        {
            get
            {
                if (_TextMesh == null)
                    _TextMesh = GetComponent<TMP_Text>();

                return _TextMesh;
            }
        }

        private void OnValidate()
        {
            RenderText();
        }


        public MarkdownRenderingSettings RenderSettings =
MarkdownRenderingSettings.Default;

        private void RenderText()
        {
            Markdown.RenderToTextMesh(Source, TextMesh, RenderSettings);
        }
    }
}
using System;
using NaughtyAttributes;
using UnityEngine;

namespace LogicUI.FancyTextRendering
{
    [Serializable]
    public class MarkdownRenderingSettings
    {
        public static MarkdownRenderingSettings Default => new
MarkdownRenderingSettings();
```

```
        public BoldSettings Bold = new BoldSettings();
        [Serializable] public class BoldSettings
        {
            public bool RenderBold = true;
        }

        public ItalicSettings Italics = new ItalicSettings();
        [Serializable] public class ItalicSettings
        {
            public bool RenderItalics = true;
        }

        public StrikethroughSettings Strikethrough = new
StrikethroughSettings();
        [Serializable] public class StrikethroughSettings
        {
            public bool RenderStrikethrough = true;
        }

        public MonospaceSettings Monospace = new MonospaceSettings();
        [Serializable] public class MonospaceSettings
        {
            public bool RenderMonospace = true;

            [Space]
            [ShowIf(nameof(RenderMonospace)), AllowNesting]
            public bool UseCustomFont = true;

            [ShowIf(EConditionOperator.And, nameof(RenderMonospace),
nameof(UseCustomFont)), AllowNesting]
            public string FontAssetPathRelativeToResources = "Noto/Noto
Mono/NotoMono-Regular";

            [Space]
            [ShowIf(nameof(RenderMonospace)), AllowNesting]
            public bool DrawOverlay = true;

            [ShowIf(EConditionOperator.And, nameof(RenderMonospace),
nameof(DrawOverlay)), AllowNesting]
            public Color OverlayColor = new Color32(0, 0, 0, 60);

            [ShowIf(EConditionOperator.And, nameof(RenderMonospace),
nameof(DrawOverlay)), AllowNesting]
            public float OverlayPaddingPixels = 25;

            [Space]
            [ShowIf(nameof(RenderMonospace)), AllowNesting]
            public bool ManuallySetCharacterSpacing = false;

            [ShowIf(EConditionOperator.And, nameof(RenderMonospace),
nameof(ManuallySetCharacterSpacing)), AllowNesting]
            public float CharacterSpacing = 0.69f;

            [Space]
            [ShowIf(nameof(RenderMonospace)), AllowNesting]
            public bool AddSeparationSpacing = true;

            [ShowIf(EConditionOperator.And, nameof(RenderMonospace),
nameof(AddSeparationSpacing)), AllowNesting]
            public float SeparationSpacing = 0.3f;
        }


        public ListSettings Lists = new ListSettings();
        [Serializable] public class ListSettings
        {
            public bool RenderUnorderedLists = true;
            public bool RenderOrderedLists = true;

            [Space]
            [ShowIf(nameof(RenderUnorderedLists)), AllowNesting]
            public string UnorderedListBullet = "•";

            [ShowIf(nameof(RenderOrderedLists)), AllowNesting]
            public string OrderedListNumberSuffix = ".";

            [Space]
            [ShowIf(EConditionOperator.Or, nameof(RenderUnorderedLists),
nameof(RenderOrderedLists)), AllowNesting]
            public float VerticalOffset = 0.76f;

            [ShowIf(EConditionOperator.Or, nameof(RenderUnorderedLists),
nameof(RenderOrderedLists)), AllowNesting]
            public float BulletOffsetPixels = 100f;

            [ShowIf(EConditionOperator.Or, nameof(RenderUnorderedLists),
nameof(RenderOrderedLists)), AllowNesting]
            public float ContentSeparationPixels = 20f;
        }

        public LinkSettings Links = new LinkSettings();
        [Serializable] public class LinkSettings
        {
            public bool RenderLinks = true;
            public bool RenderAutoLinks = true;
            [ShowIf(EConditionOperator.Or, nameof(RenderLinks),
nameof(RenderAutoLinks)), AllowNesting]
            [ColorUsage(showAlpha: false)]
            public Color LinkColor = new Color32(29, 124, 234, a:
byte.MaxValue);
        }

        public HeaderSettings Headers = new HeaderSettings();
        [Serializable] public class HeaderSettings
        {
            public bool RenderPoundSignHeaders = true;
            public bool RenderLineHeaders = true;

            [Space]
            // [ShowIf(nameof(RenderHeaders)), AllowNesting]
            // Can't use ShowIf here yet --
https://github.com/dbrizov/NaughtyAttributes/issues/142
            public HeaderData[] Levels = new HeaderData[]
            {
                new HeaderData(2f, true, true, 0.45f),
                new HeaderData(1.7f, true, true, 0.3f),
                new HeaderData(1.5f, true, false),
                new HeaderData(1.3f, true, false),
            };


            [Serializable]
            public class HeaderData
            {
                public float Size;
                public bool Bold;
                public bool Underline;
                public HeaderCase Case = HeaderCase.None;
                public float VerticalSpacing;



                public HeaderData() { } // Needs a default constructor
so it can be deserialized by SUCC
                public HeaderData(float size, bool bold, bool underline,
float verticalSpacing = 0)
                {
                    Size = size;
                    Bold = bold;
                    Underline = underline;
                    VerticalSpacing = verticalSpacing;
                }
                public enum HeaderCase
                {
                    None = 0,
                    Uppercase,
                    Smallcaps,
                    Lowercase
                }
            }
        }

        public SuperscriptSettings Superscript = new
SuperscriptSettings();
        [Serializable] public class SuperscriptSettings
        {
            public bool RenderSuperscript = false;

            [ShowIf(nameof(RenderSuperscript)), AllowNesting]
            public bool RenderChainSuperscript = true;
        }
    }
}
using System;
using System.Collections.Generic;
using UnityEngine;

namespace LogicUI.FancyTextRendering
{
    /// <summary>
    /// Allows links in TextMeshPro text objects to be clicked on, and
gives them custom colors when they are hovered or clicked.
    /// </summary>
    [RequireComponent(typeof(TextLinkHelper))]
    [DisallowMultipleComponent]
    public class SimpleLinkBehavior : MonoBehaviour
    {
        private void Awake()
        {
            GetComponent<TextLinkHelper>().OnLinkClicked += ClickOnLink;
        }

        private void ClickOnLink(string linkID)
        {
            if (CustomLinks.TryGetValue(linkID, out var action))
                action?.Invoke();
            else
                Application.OpenURL(linkID);
            Debug.Log($"clicked on link: {linkID}");
        }


        private Dictionary<string, Action> CustomLinks = new
Dictionary<string, Action>();

        /// <summary>
        /// Sets some code to be run when a link is clicked on.
        /// If a link doesn't have a custom action set, we will use <see
cref="Application.OpenURL(string)"/> on it.
        /// </summary>
        /// <param name="linkID"></param>
        /// <param name="linkAction"></param>
        public void SetCustomLink(string linkID, Action linkAction)
        {
            CustomLinks[linkID] = linkAction;
        }
    }
}
using JimmysUnityUtilities;
using System;
using TMPro;
using UnityEngine;
using UnityEngine.EventSystems;

namespace LogicUI.FancyTextRendering
{
    /// <summary>
    /// Allows links in TextMeshPro text objects to be clicked on, and
gives them custom colors when they are hovered or clicked.
    /// </summary>
    [RequireComponent(typeof(TMP_Text))]
    public class TextLinkHelper : MonoBehaviour, IPointerClickHandler,
IPointerDownHandler, IPointerUpHandler, IPointerEnterHandler,
IPointerExitHandler
    {
        private TMP_Text _Text;
        public TMP_Text Text
        {
            get
            {
                if (_Text == null)
                    _Text = GetComponent<TMP_Text>();

                return _Text;
            }
        }
        private void OnValidate()
        {
            SetAllLinksToNormalColor();
        }

        private void OnEnable()
        {
            previouslyHoveredLinkIndex = -1;

            // I'm not sure why the frame delay is necessary, but it is.
            // I suspect that TMP changes the colors in LateUpdate or
something
            CoroutineUtility.RunAfterOneFrame(SetAllLinksToNormalColor);
        }

        private void OnDisable()
        {
            HoverEnded();
        }


        [ColorUsage(showAlpha: false), SerializeField]
        Color32 LinkNormalColor = new Color32(29, 124, 234, 255);

        [ColorUsage(showAlpha: false), SerializeField]
        Color32 LinkHoveredColor = new Color32(72, 146, 231, 255);

        [ColorUsage(showAlpha: false), SerializeField]
        Color32 LinkClickColor = new Color32(38, 108, 190, 255);

        public void SetColors(Color32 linkNormalColor, Color32
linkHoveredColor, Color32 linkClickColor)
        {
            LinkNormalColor = linkNormalColor;
            LinkHoveredColor = linkHoveredColor;
            LinkClickColor = linkClickColor;

            SetAllLinksToNormalColor();
        }

        public void LinkDataUpdated()
        {
            previouslyHoveredLinkIndex = -1;
            SetAllLinksToNormalColor();
        }
        public event Action<string> OnLinkClicked;

        void IPointerClickHandler.OnPointerClick(PointerEventData
eventData)
        {
            int linkIndex = TMP_TextUtilities.FindIntersectingLink(Text,
eventData.pressPosition, eventData.pressEventCamera);
            if (linkIndex > -1)
            {
                var link = Text.textInfo.linkInfo[linkIndex];
                OnLinkClicked?.Invoke(link.GetLinkID());
            }
        }



        void IPointerDownHandler.OnPointerDown(PointerEventData
eventData)
        {
            PointerIsDown = true;
            SetLinkToColor(previouslyHoveredLinkIndex, LinkClickColor);
        }

        void IPointerUpHandler.OnPointerUp(PointerEventData eventData)
        {
            PointerIsDown = false;
            SetLinkToColor(previouslyHoveredLinkIndex, LinkNormalColor);
            previouslyHoveredLinkIndex = -1; // Reset the link hovered
caching so that in Update() it's set back to the hovered color
        }

        void IPointerEnterHandler.OnPointerEnter(PointerEventData
eventData)
        {
            CurrentlyHoveredOver = true;
            PointerIsDown = false;
            cachedCamera = eventData.enterEventCamera;
        }

        void IPointerExitHandler.OnPointerExit(PointerEventData
eventData)
        {
            CurrentlyHoveredOver = false;
            HoverEnded();
            SetAllLinksToNormalColor();
        }
        Camera cachedCamera;

        bool CurrentlyHoveredOver;
        bool PointerIsDown;
        void Update()
        {
            if (!CurrentlyHoveredOver)
                return;

            if (PointerIsDown)
                return;


            int linkIndex = TMP_TextUtilities.FindIntersectingLink(Text,
Input.mousePosition, cachedCamera);
            if (linkIndex < 0)
                HoverEnded();
            else
                HoverOnLink(linkIndex);
        }


        private void SetAllLinksToNormalColor()
        {
            if (Text.textInfo == null || Text.textInfo.linkInfo == null)
// Text object isn't initialized yet; required as of TMP 2.1
                return;

            for (int i = 0; i < Text.textInfo.linkInfo.Length; i++)
                SetLinkToColor(i, LinkNormalColor);
        }


        public event Action<string> OnLinkHovered;
        public event Action OnLinkHoverEnded;

        private int previouslyHoveredLinkIndex = -1;
        private void HoverOnLink(int linkIndex)
        {
            if (linkIndex < 0 || linkIndex >=
Text.textInfo.linkInfo.Length)
                return;

            if (linkIndex == previouslyHoveredLinkIndex)
                return;

            previouslyHoveredLinkIndex = linkIndex;
            SetLinkToColor(linkIndex, LinkHoveredColor);
            string linkID =
Text.textInfo.linkInfo[linkIndex].GetLinkID();
            OnLinkHovered?.Invoke(linkID);
        }

        private void HoverEnded()
        {
            SetLinkToColor(previouslyHoveredLinkIndex, LinkNormalColor);

            if (previouslyHoveredLinkIndex > -1)
                OnLinkHoverEnded?.Invoke();

            previouslyHoveredLinkIndex = -1;
        }


        private void SetLinkToColor(int linkIndex, Color32 color)
        {
            if (linkIndex < 0 || linkIndex >=
Text.textInfo.linkInfo.Length)
                return;

            TMP_LinkInfo linkInfo = Text.textInfo.linkInfo[linkIndex];

            if (linkInfo.linkTextfirstCharacterIndex +
linkInfo.linkTextLength - 1 >= Text.textInfo.characterInfo.Length)
                return;


            for (int i = 0; i < linkInfo.linkTextLength; i++)
            {
                int characterIndex =
linkInfo.linkTextfirstCharacterIndex + i;
                var charInfo =
Text.textInfo.characterInfo[characterIndex];
                int meshIndex = charInfo.materialReferenceIndex;
                int vertexIndex = charInfo.vertexIndex;

                var characterVertexColors =
Text.textInfo.meshInfo[meshIndex].colors32;

                if (charInfo.isVisible)
                {
                    characterVertexColors[vertexIndex + 0] = color;
                    characterVertexColors[vertexIndex + 1] = color;
                    characterVertexColors[vertexIndex + 2] = color;
                    characterVertexColors[vertexIndex + 3] = color;
                }
            }

            Text.UpdateVertexData(TMP_VertexDataUpdateFlags.All);
        }
    }
}
using System;
using System.Collections.Generic;
using System.Text;
using JimmysUnityUtilities;

using HeaderCase =
LogicUI.FancyTextRendering.MarkdownRenderingSettings.HeaderSettings.Head
erData.HeaderCase;

namespace LogicUI.FancyTextRendering.MarkdownLogic
{
    class Headers : MarkdownLineProcessorBase
    {
        protected override void
ProcessInternal(IReadOnlyList<MarkdownLine> lines,
MarkdownRenderingSettings settings)
        {
            bool inWakeOfHeader = false; // Used to delete empty lines
after headers
            for (int i = 0; i < lines.Count; i++)
            {
                MarkdownLine line = lines[i];
                StringBuilder builder = line.Builder;

                if (line.DisableFutureProcessing)
                    continue;

                if (inWakeOfHeader)
                {
                    if (line.Builder.IsEmpty())
                    {
                        line.DeleteLineAfterProcessing = true;
                        continue;
                    }
                    else
                    {
                        inWakeOfHeader = false;
                    }
                }
                if (settings.Headers.RenderPoundSignHeaders)
                {
                    int leadingNumberSignsCount = 0;
                    while (leadingNumberSignsCount < builder.Length &&
builder[leadingNumberSignsCount] == '#')
                        leadingNumberSignsCount++;

                    // Require a space between the # and the header
contents. This is different from most markdown implementations,
                    // but it means you can type #relatable and it will
actually be a hashtag and not a big header
                    if (builder.Length > leadingNumberSignsCount &&
builder[leadingNumberSignsCount] == ' ')
                    {
                        if (CanMakeLineHeader(line,
leadingNumberSignsCount))
                        {
                            builder.Remove(startIndex: 0, length:
leadingNumberSignsCount);
                            MakeLineHeader(line,
leadingNumberSignsCount);
                            continue;
                        }
                    }
                }

                if (settings.Headers.RenderLineHeaders)
                {
                    if (i > 0) // Line headers can't apply to the first
line
                    {
                        int targetHeaderLevel = -1;

                        if (builder.IsExclusively('='))
                            targetHeaderLevel = 1;
                        else if (builder.IsExclusively('-'))
                            targetHeaderLevel = 2;

                        if (CanMakeLineHeader(lines[i - 1],
targetHeaderLevel))
                        {
                            MakeLineHeader(lines[i - 1],
targetHeaderLevel);
                            line.DeleteLineAfterProcessing = true;
                        }
                    }
                }
                bool CanMakeLineHeader(MarkdownLine line_, int
headerLevel)
                {
                    if (line_.DisableFutureProcessing)
                        return false;

                    if (headerLevel < 1 || headerLevel >
settings.Headers.Levels.Length)
                        return false;

                    return true;
                }

                void MakeLineHeader(MarkdownLine line_, int headerLevel)
                {
                    inWakeOfHeader = true;

                    var builder_ = line_.Builder;
                    var info = settings.Headers.Levels[headerLevel - 1];

                    builder_.TrimStart(' ');

                    builder_.PrependChain("<size=",
info.Size.ToString(), "em>");
                    builder_.Append("</size>");

                    if (info.Bold)
                        builder_.Prepend("<b>").Append("</b>");

                    if (info.Underline)
                        builder_.Prepend("<u>").Append("</u>");

                    switch (info.Case)
                    {
                        default:
                        case HeaderCase.None:
                            break;

                        case HeaderCase.Uppercase:
builder_.Prepend("<uppercase>").Append("</uppercase>");
          break;

                        case HeaderCase.Smallcaps:
builder_.Prepend("<smallcaps>").Append("</smallcaps>");
          break;

                        case HeaderCase.Lowercase:
```

```
builder_.Prepend("<lowercase>").Append("</lowercase>");
          break;
                    }

line_.AddVerticalWhitespaceAfter(info.VerticalSpacing);                  }
           }
        }

        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Headers.RenderPoundSignHeaders ||
settings.Headers.RenderLineHeaders;
    }
}
using System;
using System.Text;
using UnityEngine;
using JimmysUnityUtilities;

namespace LogicUI.FancyTextRendering.MarkdownLogic
{
    // Important note about link coloring: yes, link colors are assigned
by LinkTextHelper. However:
    //    A) This can only be done after a delay of one frame due to TMP
bullshit. Unless we also set the color in the rich
    //       text tags, the color will flicker for a frame.
    //    B) If we want to change the alpha of the text (such as fading
chat messages), that resets the LinkTextHelper colors,
    //       because TMP resets the mesh vertex colors when the base
color changes.
    // So, TextLinkHelper handles colors for hovering and clicking on
the link, but we ensure links always appear with the
    // appropriate color by also doing it in rich text.

    class Links : SimpleMarkdownLineProcessor
    {
        protected override void ProcessLine(MarkdownLine line,
MarkdownRenderingSettings settings)
        {
            StringBuilder builder = line.Builder;

            int linkTextStart = line.UnescapedIndexOf('[');

            while (linkTextStart > -1)
            {
                int linkTextEnd = line.UnescapedIndexOf(']');
                if (linkTextEnd < 0)
                    return;
                int linkContentStart = line.UnescapedIndexOf('(',
startIndex: linkTextEnd);
                if (linkContentStart < 0)
                    return;

                if (linkContentStart != linkTextEnd + 1)
                    return;

                int linkContentEnd = line.UnescapedIndexOf(')',
startIndex: linkContentStart); // Escaping this doesn't work properly
for some reason -- todo investigate
                if (linkContentEnd < 0)
                    return;

                if (linkContentEnd - linkContentStart < 2)
                    return;



                string linkText = builder.Snip(linkTextStart + 1,
linkTextEnd - 1);
                string linkContent = builder.Snip(linkContentStart + 1,
linkContentEnd - 1);

                builder.Remove(linkTextStart, linkContentEnd -
linkTextStart + 1);
                builder.InsertChain(linkTextStart, out int
insertionEndIndex,
                    "<color=#",
ColorUtility.ToHtmlStringRGBA(settings.Links.LinkColor), ">",
                    "<link=\"", linkContent, "\">",
                    linkText,
                    "</link></color>");



                linkTextStart = builder.IndexOf('[', startIndex:
insertionEndIndex);
            }
        }



        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Links.RenderLinks;
    }

    // Auto-links don't override regular links because they only apply
to links without a character in front of them, and auto links always
```

```
have
    // the quotation marks.
    abstract class AutoLinks : SimpleMarkdownLineProcessor
    {
        protected abstract string LinkStartString { get; }

        protected override void ProcessLine(MarkdownLine line,
MarkdownRenderingSettings settings)
        {
            StringBuilder builder = line.Builder;

            int linkStartIndex = builder.IndexOf(LinkStartString,
ignoreCase: true);

            while (linkStartIndex > -1)
            {
                if (linkStartIndex == 0 || builder[linkStartIndex -
1].IsWhitespaceOrNonBreakingSpace())
                {
                    int linkEndIndex =
builder.IndexOfWhitespace(linkStartIndex + LinkStartString.Length) - 1;
                    if (linkEndIndex < 0)
                        linkEndIndex = builder.Length - 1;

                    int linkLength = linkEndIndex - linkStartIndex;
                    if (linkLength <= LinkStartString.Length)
                    {
                        linkStartIndex =
builder.IndexOf(LinkStartString, linkStartIndex +
LinkStartString.Length, ignoreCase: true);
                        continue;
                    }

                    int nextDotIndex = builder.IndexOf('.',
linkStartIndex + LinkStartString.Length);

                    if (nextDotIndex < 0 || nextDotIndex >=
linkEndIndex)
                    {
                        linkStartIndex =
builder.IndexOf(LinkStartString, linkEndIndex, ignoreCase: true);
                        continue;
                    }


                    string linkText = builder.Snip(linkStartIndex,
linkEndIndex);
```

```
                    builder.InsertChain(linkStartIndex, out int
insertionEndIndex,
                        "<color=#",
ColorUtility.ToHtmlStringRGBA(settings.Links.LinkColor), ">",
                        "<link=\"", linkText, "\">");

                    builder.Insert(insertionEndIndex + linkText.Length,
"</link></color>");

                    linkStartIndex = builder.IndexOf(LinkStartString,
insertionEndIndex + 15);
                }
                else
                {
                    linkStartIndex = builder.IndexOf(LinkStartString,
linkStartIndex + LinkStartString.Length);
                }
            }
        }


        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Links.RenderAutoLinks;
    }

    class AutoLinksHttps : AutoLinks
    {
        protected override string LinkStartString => "https://";
    }

    class AutoLinksHttp : AutoLinks
    {
        protected override string LinkStartString => "http://";
    }
}
using System;
using System.Collections.Generic;
using System.Text;
using JimmysUnityUtilities;

namespace LogicUI.FancyTextRendering.MarkdownLogic
{
    abstract class MarkdownListProcessor : MarkdownLineProcessorBase
    {
        protected abstract bool LineIsPartOfList(StringBuilder
lineBuilder, MarkdownRenderingSettings settings);
        protected abstract void FormatBeginningOfListLine(StringBuilder
lineBuilder, int listLineIndex, MarkdownRenderingSettings settings);

        protected abstract void InsertListBulletContents(StringBuilder
builder, int index, int listLineIndex, MarkdownRenderingSettings
settings, out int bulletTextLength);

        protected override void
ProcessInternal(IReadOnlyList<MarkdownLine> lines,
MarkdownRenderingSettings settings)
        {
            bool previousLineWasListLine = false;
            int listLineCount = 0;

            // We want to have the list bullets right-aligned and the
list contents left-aligned. We're looking for this effect:
            //
            //    8. On top of the empire state building
            //    9. In the alley behind the Applebees
            //   10. In an airplane bathroom
            //
            // TMP doesn't support two different alignments on the same
line, so we hack it by using two lines and setting one to
            // 0% height. This makes the two lines appear on top of each
other.
            //
            // Currently there's a bug with TMP where font units don't
work for width. TODO switch to font units when it's fixed.
            // https://forum.unity.com/threads/font-units-do-not-work-
with-the-width-rich-text-tag.1006504/

            string listPrependBeforeBullet = $"<line-
height=0%><width={settings.Lists.BulletOffsetPixels}><align=right>";
            string listPrependAfterBullet = $"</width></align>\n</line-
height><indent={settings.Lists.BulletOffsetPixels +
settings.Lists.ContentSeparationPixels}>";

            for (int i = 0; i < lines.Count; i++)
            {
                if (lines[i].DisableFutureProcessing)
                    goto lineNotPartOfList;


                StringBuilder builder = lines[i].Builder;

                if (builder.IsEmptyOrWhitespace())
                    goto lineNotPartOfList;

                if (LineIsPartOfList(builder, settings))
                {
                    if (previousLineWasListLine)
                        listLineCount++;


                    FormatBeginningOfListLine(builder, listLineCount, 
settings);
                    builder.TrimStart(' ');


                    // Write the stuff at the beginning of the line
                    // We go through this absolute hell because it's 
marginally faster than PrependChain
                    builder.Insert(0, listPrependBeforeBullet);
                    InsertListBulletContents(builder, index: 
listPrependBeforeBullet.Length, listLineCount, settings, out int 
bulletTextLength);
                    builder.Insert(listPrependBeforeBullet.Length + 
bulletTextLength, listPrependAfterBullet);

                    // Write the stuff at the end of the line
                    builder.Append("</indent>");


                    previousLineWasListLine = true;
                }
                else
                {
                    goto lineNotPartOfList;
                }


                continue;

lineNotPartOfList:

                if (previousLineWasListLine)
                    HandleListEnds(listEndLineIndex: i - 1);

                previousLineWasListLine = false;
                listLineCount = 0;
                ResetVariables();
                continue;
            }


            if (previousLineWasListLine)
                HandleListEnds(listEndLineIndex: lines.Count - 1);
            // Handles the vertical offset of the list from its
surrounding content
            void HandleListEnds(int listEndLineIndex)
            {
                int listStartLineIndex = listEndLineIndex -
listLineCount;

                // If the list doesn't start on the first line, add
whitespace before the first list line.
                if (listStartLineIndex > 0)
lines[listStartLineIndex].AddVerticalWhitespaceBefore(settings.Lists.Ver
ticalOffset);

                // If the list doesn't end on the last line, add
whitespace after the last line.
                if (listEndLineIndex < lines.Count - 1)
lines[listEndLineIndex].AddVerticalWhitespaceAfter(settings.Lists.Vertic
alOffset);


                // Remove extra whitespace before and after the list.

                for (int i = listStartLineIndex - 1; i > 0; i--)
                {
                    // The use of IsEmpty() instead of
IsEmptyOrWhitespace() is intentional. It allows you to manually add
vertical offset to lists by putting
                    // a single space on the line. However, this is a
deviation from standard markdown implementations.
                    if (lines[i].Builder.IsEmpty())
                        lines[i].DeleteLineAfterProcessing = true;
                    else
                        break;
                }

                for (int i = listEndLineIndex + 1; i < lines.Count; i++)
                {
                    if (lines[i].Builder.IsEmpty())
                        lines[i].DeleteLineAfterProcessing = true;
                    else
                        break;
                }
            }
        }

        protected virtual void ResetVariables() { }
```

```
    }



    class UnorderedLists : MarkdownListProcessor
    {
        protected override bool LineIsPartOfList(StringBuilder
lineBuilder, MarkdownRenderingSettings settings)
            => lineBuilder.StartsWith("- ") || lineBuilder.StartsWith("*
");

        protected override void FormatBeginningOfListLine(StringBuilder
lineBuilder, int listLineIndex, MarkdownRenderingSettings settings)
        {
            // Remove the '-' or '*'
            lineBuilder.Remove(0, 1);
        }

        protected override void InsertListBulletContents(StringBuilder
builder, int index, int _, MarkdownRenderingSettings settings, out int
bulletTextLength)
        {
            builder.Insert(index, settings.Lists.UnorderedListBullet);
            bulletTextLength =
settings.Lists.UnorderedListBullet.Length;
        }

        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Lists.RenderUnorderedLists;
    }


    class OrderedLists : MarkdownListProcessor
    {
        int dotIndex;
        int parsedNumber;
        bool currentListIsAutoNumbered;
        string potentialNumberString;

        protected override void ResetVariables()
        {
            dotIndex = 0;
            parsedNumber = 0;
            currentListIsAutoNumbered = false;
            potentialNumberString = string.Empty;
        }
        protected override bool LineIsPartOfList(StringBuilder
lineBuilder, MarkdownRenderingSettings settings)
        {
            dotIndex = lineBuilder.IndexOf('.');

            if (dotIndex < 1)
                return false;

            if (lineBuilder.Length < dotIndex + 2 ||
lineBuilder[dotIndex + 1] != ' ')
                return false;


            potentialNumberString = lineBuilder.Snip(0, dotIndex - 1);
            bool thisLineIsListLine =
int.TryParse(potentialNumberString, out parsedNumber);

            return thisLineIsListLine;
        }

        protected override void FormatBeginningOfListLine(StringBuilder
lineBuilder, int listLineIndex, MarkdownRenderingSettings settings)
        {
            lineBuilder.Remove(0, dotIndex + 1);
        }


        protected override void InsertListBulletContents(StringBuilder
builder, int index, int listLineIndex, MarkdownRenderingSettings
settings, out int bulletTextLength)
        {
            // Our logic here differs from other markdown
implementations: if you use all 1s for your ordered list, it auto-
numbers them. Otherwise,
            // it uses the numbers you provide.

            string numberToDisplayString;

            int listLineNumber = listLineIndex + 1;
            bool isFirstLineOfList = listLineIndex == 0;
            if (isFirstLineOfList)
            {
                if (parsedNumber == 1)
                {
                    currentListIsAutoNumbered = true;
                    numberToDisplayString = listLineNumber.ToString();
                }
                else
                {
                    numberToDisplayString = potentialNumberString;
                }
            }
            else
            {
                if (currentListIsAutoNumbered && parsedNumber == 1)
                {
                    numberToDisplayString = listLineNumber.ToString();
                }
                else
                {
                    numberToDisplayString = potentialNumberString;
                    currentListIsAutoNumbered = false;
                }
            }

            builder.InsertChain(index, numberToDisplayString,
settings.Lists.OrderedListNumberSuffix);
            bulletTextLength = numberToDisplayString.Length +
settings.Lists.OrderedListNumberSuffix.Length;
        }

        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Lists.RenderOrderedLists;
    }
}
using UnityEngine;

namespace LogicUI.FancyTextRendering.MarkdownLogic
{
    class Bold : SimpleMarkdownTag
    {
        protected override string MarkdownIndicator => "**";
        protected override string RichTextOpenTag => "<b>";
        protected override string RichTextCloseTag => "</b>";

        protected override char? IgnoreContents => '*';

        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Bold.RenderBold;
    }

    class Italics : SimpleMarkdownTag
    {
        protected override string MarkdownIndicator => "*";
        protected override string RichTextOpenTag => "<i>";
        protected override string RichTextCloseTag => "</i>";

        protected override char? IgnoreContents => '*';

        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Italics.RenderItalics;
    }

    class Strikethrough : SimpleMarkdownTag
    {
        protected override string MarkdownIndicator => "~~";
        protected override string RichTextOpenTag => "<s>";
        protected override string RichTextCloseTag => "</s>";

        protected override char? IgnoreContents => '~';

        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Strikethrough.RenderStrikethrough;
    }

    class Monospace : MarkdownTag
    {
        protected override string
GetMarkdownOpenTag(MarkdownRenderingSettings settings) => "`";
        protected override string
GetMarkdownCloseTag(MarkdownRenderingSettings settings) => "`";

        // Optimization not critical, as this is only called once per
markdown render
        protected override string
GetRichTextOpenTag(MarkdownRenderingSettings settings)
        {
            string tag = string.Empty;

            if (settings.Monospace.AddSeparationSpacing)
                tag +=
$"<space={settings.Monospace.SeparationSpacing}em>";

            if (settings.Monospace.UseCustomFont)
                tag +=
$"<font=\"{settings.Monospace.FontAssetPathRelativeToResources}\">";

            if (settings.Monospace.DrawOverlay)
            {
                var padding = settings.Monospace.OverlayPaddingPixels;
                tag +=
$"<mark=#{ColorUtility.ToHtmlStringRGBA(settings.Monospace.OverlayColor)
} padding=\"{padding},{padding},0,0\">";
            }

            if (settings.Monospace.ManuallySetCharacterSpacing)
                tag +=
$"<mspace={settings.Monospace.CharacterSpacing}em>";

            return tag;
        }

        protected override string
GetRichTextCloseTag(MarkdownRenderingSettings settings)
        {
            string tag = string.Empty;

            if (settings.Monospace.UseCustomFont)
                tag += "</font>";

            if (settings.Monospace.DrawOverlay)
                tag += "</mark>";

            if (settings.Monospace.ManuallySetCharacterSpacing)
                tag += "</mspace>";

            if (settings.Monospace.AddSeparationSpacing)
                tag +=
$"<space={settings.Monospace.SeparationSpacing}em>";

            return tag;
        }


        protected override char? IgnoreContents => '`';

        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Monospace.RenderMonospace;
    }



    class SuperscriptSingle : MarkdownTag
    {
        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Superscript.RenderSuperscript;
        protected override string
GetMarkdownOpenTag(MarkdownRenderingSettings settings) => "^";
        protected override string
GetMarkdownCloseTag(MarkdownRenderingSettings settings) => " ";

        protected override string
GetRichTextOpenTag(MarkdownRenderingSettings settings) => "<sup>";
        protected override string
GetRichTextCloseTag(MarkdownRenderingSettings settings) => "</sup> ";


        protected override char? IgnoreContents => '^';
        protected override bool TreatLineEndAsCloseTag => true;
    }

    class SuperscriptChain : MarkdownTag
    {
        protected override bool
AllowedToProces(MarkdownRenderingSettings settings)
            => settings.Superscript.RenderSuperscript &&
settings.Superscript.RenderChainSuperscript;


        protected override string
GetMarkdownOpenTag(MarkdownRenderingSettings settings) => "^(";
        protected override string
GetMarkdownCloseTag(MarkdownRenderingSettings settings) => ")";

        protected override string
GetRichTextOpenTag(MarkdownRenderingSettings settings) => "<sup>";
        protected override string
GetRichTextCloseTag(MarkdownRenderingSettings settings) => "</sup>";
    }
}
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Text;
using JimmysUnityUtilities;

namespace LogicUI.FancyTextRendering.MarkdownLogic
{
    internal class MarkdownLine
    {
        public StringBuilder Builder;
        public bool DeleteLineAfterProcessing { get; set; }

        public bool DisableFutureProcessing
        {
            get => _DisableFutureProcessing ||
DeleteLineAfterProcessing;
            set => _DisableFutureProcessing = value;
        }
        private bool _DisableFutureProcessing;



        // To create vertical whitespace, separating some text from the
stuff above or below it, we have an empty line at a specific size.
        // The line must have some contents for TMP to apply the custom
size (I think this may be a TMP bug). Therefore the line contains a
single zero-width space.
        public void AddVerticalWhitespaceBefore(float spacingFontUnits)
        {
            if (!spacingFontUnits.IsPrettyCloseTo(0))
                Builder.Prepend("<size=" +
spacingFontUnits.ToString(CultureInfo.InvariantCulture) + "em>" +
SpecialStrings.ZeroWidthSpace + "</size>\n");
        }
        public void AddVerticalWhitespaceAfter(float spacingFontUnits)
        {
            if (!spacingFontUnits.IsPrettyCloseTo(0))
                Builder.Append("\n<size=" +
spacingFontUnits.ToString(CultureInfo.InvariantCulture) + "em>" +
SpecialStrings.ZeroWidthSpace + "</size>");
        }


        public string Finish()
        {
            if (DeleteLineAfterProcessing)
                throw new Exception("You shouldn't process this line --
it's supposed to be deleted!");

            ReplaceEscapes();
            return Builder.ToString();


            void ReplaceEscapes()
            {
                int countingIndex = 0;
                while (countingIndex < Builder.Length)
                {
                    int index = Builder.IndexOf(EscapeCharacater,
countingIndex);

                    if (index < 0)
                        return;

                    // Remove the escape character
                    Builder.Remove(startIndex: index, length: 1);

                    // Skip one chracter. This allows you to escape
escapes, i.e. '\\' becomes '\'. Big brain moment
                    countingIndex = index + 1;
                }
            }
        }


        public static readonly char EscapeCharacater = '\\';
        public int UnescapedIndexOf(string find, int startIndex = 0)
        {
            int countingIndex = startIndex;

            while (countingIndex < Builder.Length)
            {
                int index = Builder.IndexOf(find, countingIndex);

                if (index < 0)
                    return -1;

                // Count how many escape characters come before the
index. If it's odd, that means the index is escaped.
                // If it's even, that means the escape is escaped, or
there are no escapes.
                if (CountPrecedingEscapesCharacters(index).IsOdd())
                {
                    countingIndex = index + find.Length;
                    continue;
                }

                return index;
            }

            return -1;
        }
        public int UnescapedIndexOf(char find, int startIndex = 0)
        {
            int countingIndex = startIndex;
            while (countingIndex < Builder.Length)
            {
                int index = Builder.IndexOf(find, countingIndex);

                if (index < 0)
                    return -1;

                // Count how many escape characters come before the
index. If it's odd, that means the index is escaped.
                // If it's even, that means the escape is escaped, or
there are no escapes.
                if (CountPrecedingEscapesCharacters(index).IsOdd())
                {
                    countingIndex = index + 1;
                    continue;
                }

                return index;
            }

            return -1;
        }


        int CountPrecedingEscapesCharacters(int index)
        {
            if (index < 0 || index > Builder.Length)
                throw new IndexOutOfRangeException();

            if (index == 0)
                return 0;


            int count = 0;

            for (int i = index; i > 0; i--)
            {
                if (Builder[index - 1] == EscapeCharacater)
                    count++;
                else
                    break;
            }

            return count;
        }
    }
}
using System;
using System.Collections.Generic;
using System.Text;
using JimmysUnityUtilities;

namespace LogicUI.FancyTextRendering.MarkdownLogic
{
    internal abstract class MarkdownLineProcessorBase
    {
        public void Process(IReadOnlyList<MarkdownLine> lines,
MarkdownRenderingSettings settings)
        {
            if (!AllowedToProces(settings))
                return;

            ProcessInternal(lines, settings);
        }

        protected virtual bool AllowedToProces(MarkdownRenderingSettings
settings) => true;
        protected abstract void
ProcessInternal(IReadOnlyList<MarkdownLine> lines,
MarkdownRenderingSettings settings);
    }
}
using System;
using System.Collections.Generic;
using System.Text;
using JimmysUnityUtilities;
using UnityEngine;

namespace LogicUI.FancyTextRendering.MarkdownLogic
{
    internal abstract class SimpleMarkdownTag : MarkdownTag
    {
        protected abstract string MarkdownIndicator { get; }
        protected abstract string RichTextOpenTag { get; }
        protected abstract string RichTextCloseTag { get; }

        protected override string
GetMarkdownOpenTag(MarkdownRenderingSettings _) => MarkdownIndicator;
        protected override string
GetMarkdownCloseTag(MarkdownRenderingSettings _) => MarkdownIndicator;

        protected override string
GetRichTextOpenTag(MarkdownRenderingSettings _) => RichTextOpenTag;
        protected override string
GetRichTextCloseTag(MarkdownRenderingSettings _) => RichTextCloseTag;
    }
    internal abstract class MarkdownTag : MarkdownLineProcessorBase
    {
        protected abstract string
GetMarkdownOpenTag(MarkdownRenderingSettings settings);
        protected abstract string
GetMarkdownCloseTag(MarkdownRenderingSettings settings);

        protected abstract string
GetRichTextOpenTag(MarkdownRenderingSettings settings);
        protected abstract string
GetRichTextCloseTag(MarkdownRenderingSettings settings);

        protected virtual char? IgnoreContents => null;
        protected virtual bool TreatLineEndAsCloseTag => false;



        protected override void
ProcessInternal(IReadOnlyList<MarkdownLine> lines,
MarkdownRenderingSettings settings)
        {
            // It's more efficient to get these values once than for
every line. That's why this class inherits from
            // MarkdownLineProcessorBase instead of
SimpleMarkdownLineProcessor.

            string markdownOpenTag = GetMarkdownOpenTag(settings);
            string markdownCloseTag = GetMarkdownCloseTag(settings);

            string richTextOpenTag = GetRichTextOpenTag(settings);
            string richTextCloseTag = GetRichTextCloseTag(settings);

            foreach (MarkdownLine line in lines)
            {
                if (line.DisableFutureProcessing)
                    continue;

                ProcessLine(line);
            }



            void ProcessLine(MarkdownLine line)
            {
                var lineBuilder = line.Builder;

                int index = 0;
                while (index < lineBuilder.Length -
markdownOpenTag.Length)
```

```
                {
                    int openTagIndex =
line.UnescapedIndexOf(markdownOpenTag, index);

                    if (openTagIndex < 0)
                        break;


                    int closeTagIndex =
line.UnescapedIndexOf(markdownCloseTag, openTagIndex +
markdownOpenTag.Length);

                    if (closeTagIndex > -1) // If there's a remaining
tag pair in the line
                    {
                        if (openTagIndex + markdownOpenTag.Length ==
closeTagIndex)
                        {
                            // Tags have to actually apply to some text.
You can write '**' and it doesn't just turn invisible, it shows the
asterisks.
                            index = openTagIndex + 1;
                        }
                        else if
(IsSpaceBetweenIndexesInvalidForTagging(startIndex: openTagIndex +
markdownOpenTag.Length, endIndex: closeTagIndex - 1))
                        {
                            index = openTagIndex + 1; ;
                        }
                        else
                        {
                            lineBuilder.ReplaceFirst(markdownOpenTag,
richTextOpenTag, openTagIndex);
                            lineBuilder.ReplaceFirst(markdownCloseTag,
richTextCloseTag, out int richTextCloseTagStartIndex, openTagIndex +
richTextOpenTag.Length - 1);

                            index = richTextCloseTagStartIndex +
richTextCloseTag.Length;
                        }
                    }
                    else if (TreatLineEndAsCloseTag)
                    {
                        if (openTagIndex + markdownOpenTag.Length ==
lineBuilder.Length)
                            break;

                        if
```

```
(IsSpaceBetweenIndexesInvalidForTagging(startIndex: openTagIndex +
markdownOpenTag.Length, endIndex: lineBuilder.Length - 1))
                        {
                            index = openTagIndex + 1;
                            continue;
                        }

                        lineBuilder.ReplaceFirst(markdownOpenTag,
richTextOpenTag, openTagIndex);
                        lineBuilder.Append(richTextCloseTag);

                        break;
                    }
                    else
                    {
                        break;
                    }
                }


                bool IsSpaceBetweenIndexesInvalidForTagging(int
startIndex, int endIndex)
                {
                    // First, we have to check that the text doesn't
begin with the markdown indicator. To solve conflicts, we must always
use the last
                    // markdown indicator in a chain. I.e.: in
*****example*** we should use the last two asterisks from the first
clump as the bold
                    // indicator.
                    if (lineBuilder[startIndex] == markdownOpenTag[0])
                        return true;

                    // You can put spaces between stuff to make it not
apply. I.e. '* example*' is not italicized.
                    if (lineBuilder[startIndex].IsWhitespace() ||
lineBuilder[endIndex].IsWhitespace())
                        return true;

                    // Many tags have a defined character that is
ignored. So for example you can write "~~~~~" and it's five tildes, not
one crossed out tilde.
                    // Furthermore, we don't tag just escape characters.
                    for (int i = startIndex; i <= endIndex; i++)
                    {
                        if (lineBuilder[i] == IgnoreContents)
                            continue;
```

```
                        if (lineBuilder[i] ==
MarkdownLine.EscapeCharacater)
                            continue;

                        return false;
                    }

                    return true;
                }
            }
        }
    }
}
using System.Collections.Generic;

namespace LogicUI.FancyTextRendering.MarkdownLogic
{
    internal abstract class SimpleMarkdownLineProcessor :
MarkdownLineProcessorBase
    {
        protected override void
ProcessInternal(IReadOnlyList<MarkdownLine> lines,
MarkdownRenderingSettings settings)
        {
            foreach (var line in lines)
            {
                if (line.DisableFutureProcessing)
                    continue;

                ProcessLine(line, settings);
            }
        }

        protected abstract void ProcessLine(MarkdownLine line,
MarkdownRenderingSettings settings);
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AudioManager : MonoBehaviour
{
    public AudioSource AudioSource;

    public void TogglePlayPause() {
        if (AudioSource.isPlaying) AudioSource.Pause();
        else if (AudioSource.clip != null && !AudioSource.isPlaying)
```

```
AudioSource.UnPause();
    }
}
// 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;
using LogicUI.FancyTextRendering;
using OpenAI;

public class ChatWindow : MonoBehaviour
{
    public GameObject pfb_chatMsgUI;

    [SerializeField]
    public GameObject audioPanel;

    [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.";


    public delegate void OnSpeechToText(string text);
    public static OnSpeechToText onSTT;

    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, MessageColorMode.MessageType
msgType)
    {
        //inputField.text = newText;
        var assistantMessageContent = AddNewTextMessageContent(msgType
== MessageColorMode.MessageType.Sender ? Role.User : Role.Assistant);

        assistantMessageContent.GetComponent<MarkdownRenderer>().Source
= newText;
        scrollView.verticalNormalizedPosition = 0f;
        //if (audioPanel.activeInHierarchy &&
!newText.Contains("User:")) GenerateSpeech(newText);
    }



    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);
        Debug.Log($"Speech request firing");
        var (clipPath, clip) = await
openAI.AudioEndpoint.CreateSpeechAsync(request,
lifetimeCancellationTokenSource.Token);
        Debug.Log($"Speech request fired {clip.name}");
        audioSource.clip = clip;
        audioSource.Play();

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

    public TextMeshProUGUI AddNewTextMessageContent(Role role)
    {
        var textObject = Instantiate(pfb_chatMsgUI, contentArea);
        textObject.name = $"{contentArea.childCount + 1}_{role}";
        //var textObject = new GameObject($"{contentArea.childCount +
1}_{role}");
        //textObject.transform.SetParent(contentArea, false);
        var msgColorMode = textObject.GetComponent<MessageColorMode>();
        if (role == Role.User)
msgColorMode.SetMode(MessageColorMode.MessageType.Sender);
        else
```

```
msgColorMode.SetMode(MessageColorMode.MessageType.Reciever);

        var textMesh = msgColorMode.messageText;
        MarkdownRenderer mr =
textMesh.gameObject.AddComponent<MarkdownRenderer>();
        mr.RenderSettings.Monospace.UseCustomFont = false;
        mr.RenderSettings.Lists.BulletOffsetPixels = 40;
        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();
    }



    /// <summary>
    /// invoked via settings toggle menu
    /// </summary>
    public void EnableAudioInterface(bool status) {
        audioPanel.SetActive(status);
        //recordButton.SetActive(status);
        audioSource.enabled = status;
    }
    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;
            UpdateChat(userInput, MessageColorMode.MessageType.Sender);
            onSTT?.Invoke(userInput);
            inputField.interactable = true;
        }
        catch (Exception e)
        {
            Debug.LogError(e);
            inputField.interactable = true;
        }
        finally
        {
            recordButton.interactable = true;
        }
    }


}
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 OpenAI.Chat;
using OpenAI;
using System.Collections;
using System.Collections.Generic;
using System.Threading;
using UnityEngine;
using Newtonsoft.Json;
using OpenAI.Images;
using System.Linq;
using System.Threading.Tasks;
using System;
using UnityEngine.EventSystems;
using UnityEngine.UI;
using UnityEngine.UIElements;
using Utilities.WebRequestRest;

public class DopeCoderController : MonoBehaviour
{
    public static DopeCoderController Instance { get; private set; }
    [SerializeField]
    public DopeCoderSettings Settings;

    [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.";

    public KeywordEventManager KeywordEventManager;
    public SphereController sphereController;
    public ReflectionRuntimeController componentController;
    public UI_Controller uiController;
    public GPTInterfacer gptInterfacer;
    [Header("Speech Controller Properties")]
    public AudioSource speechControllerAudio;
    public GameObject speechControllerAudioPanel;
    public SpeechController speechController;

    private static bool isChatPending;



    private void Awake()
    {
        if (Instance != null && Instance != this)
        {
            Destroy(this);
            return;
        }

        Instance = this;

        uiController = GetComponentInChildren<UI_Controller>();
        speechController = new SpeechController(speechControllerAudio,
speechControllerAudioPanel);
    }

    private void Start()
    {
        gptInterfacer.openAI.EnableDebug = Settings.debugMode;
        gptInterfacer.conversation.AppendMessage(new
Message(Role.System, systemPrompt));
        //inputField.onSubmit.AddListener(SubmitChat);
        //submitButton.onClick.AddListener(SubmitChat);
uiController.recordButton.onClick.AddListener(speechController.ToggleRec
ording);

        sphereController.SetMode(SphereController.SphereMode.Idle);
        GPTInterfacer.onGPTMessageReceived += UpdateChat;
        SpeechController.onSTT += UpdateChatSTT;
    }

    public void ToggleTTS(bool toggle)
    {
        Settings.tts = toggle;
        speechController.EnableAudioInterface(toggle);
    }

    public void UpdateChat(string newText, MessageColorMode.MessageType
msgType = MessageColorMode.MessageType.Sender)
    {
        uiController.UpdateChat(newText, msgType);
        if (msgType == MessageColorMode.MessageType.Reciever)
sphereController.SetMode(SphereController.SphereMode.Talking);
        if (msgType == MessageColorMode.MessageType.Sender)
sphereController.SetMode(SphereController.SphereMode.Listening);

    }

    public void UpdateChatSTT(string text)
    {
        uiController.UpdateChat(text,
MessageColorMode.MessageType.Sender);
        gptInterfacer.SubmitChatStreamRequst(text);
    }

    private void OnDestroy()
    {
        GPTInterfacer.onGPTMessageReceived -= UpdateChat;
        SpeechController.onSTT -= UpdateChatSTT;
        speechController.Destroy();
    }



    /// <summary>
    /// invoked via gui elements/event maps
    /// </summary>
    /// <param name="_">user input query via unity input field</param>
    public void SubmitChat(string _) => SubmitChatRequest();
    public void SubmitChat() => SubmitChatRequest();
    private async void SubmitChatRequest()
    {
        sphereController.SetMode(SphereController.SphereMode.Listening);
        if (isChatPending ||
string.IsNullOrWhiteSpace(uiController.inputField.text)) { return; }
        isChatPending = true;


        uiController.ToggleInput(false);

        var userMessageContent =
uiController.AddNewTextMessageContent(Role.User);
        userMessageContent.text = $"User:
{uiController.inputField.text}";

        if (KeywordEventManager != null &&
KeywordEventManager.ParseKeyword()) {
//componentController.SearchFunctions(ParseFunctionName(chatBehaviour.in
putField.text));
```

```
            Debug.Log("Keyword found, invoking event");
            uiController.ToggleInput(true); // bc chat request is async
in else block, we toggle ui back here for local commands
        }
        else {
            //chatBehaviour.SubmitChat(chatBehaviour.inputField.text);
gptInterfacer.SubmitChatStreamRequst(uiController.inputField.text);
  }
        uiController.inputField.text = string.Empty;
    }

    /// <summary>
    /// invoked externally via button press/mapping
    /// </summary>
    public void AnalyzeComponents()
    {
        // Format the data from your ComponentRuntimeController into a
string for GPT analysis
        string dataForGPT =
gptInterfacer.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? " +
            "Please leverage your knowledge of the code base as well
using the documentation that was given to you, specifically looking at
the classes specified in this message with respect to your
documentation.";

        // Combine the prompt with the data
        string combinedMessage = $"{gptPrompt}\n{dataForGPT}";
        Debug.Log(combinedMessage);
        try {
gptInterfacer.SubmitAssistantResponseRequest(combinedMessage); }
        catch (Exception e) { Debug.LogError(e); }
        finally
        {
            //if (lifetimeCancellationTokenSource != null) {}
            isChatPending = false;
        }

    }
}


[System.Serializable]
public class DopeCoderSettings
{
    public bool debugMode;
    public bool tts;
    public bool saveLogs;
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public class ExpandUI : MonoBehaviour
{
    public RectTransform rectTransform;
    public TextMeshProUGUI uiButton;
    private float originalLeft;

    protected bool isOriginalWidth;


    public void Start()
    {
        isOriginalWidth = false;
        originalLeft = rectTransform.offsetMin.x; // This should be 0 if
the panel is stretched full width to start with
    }

    public void ToggleWidth()
    {
        isOriginalWidth = !isOriginalWidth;
        if (isOriginalWidth)
        {
            // Set to half width
            rectTransform.offsetMin = new Vector2(originalLeft +
rectTransform.rect.width / 2, rectTransform.offsetMin.y);
            uiButton.text = "<";
        }
        else
        {
            // Set back to full width
            rectTransform.offsetMin = new Vector2(originalLeft,
rectTransform.offsetMin.y);
            uiButton.text = ">";
        }

        Debug.Log($"New Left Offset: {rectTransform.offsetMin.x}");
    }


}
```

```
using Newtonsoft.Json.Linq;
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Text;
using UnityEngine;
using OpenAI.Threads;
using OpenAI.Chat;
using OpenAI;
using System.Threading;
using System.Linq;
using System.Threading.Tasks;
using UnityEngine.EventSystems;
using Utilities.WebRequestRest;

public class GPTInterfacer : MonoBehaviour
{
    public string AssistantID;
    public string AssistantIDGPT3;
    public string AssistantIDGPT4;
    private bool isChatPending;

    #region OpenAI_Variables
    public OpenAIClient openAI;
    public Conversation conversation = new Conversation();
    public CancellationTokenSource lifetimeCancellationTokenSource;
    #endregion

    public delegate void GPTMessageReceived(string text,
MessageColorMode.MessageType messageType);
    public static GPTMessageReceived onGPTMessageReceived;

    #region GPTAssistantVariables
    private string threadID;
    private string messageID;
    private string runID;
    protected ThreadResponse GPTthread;
    private const int GPT4_CHARACTERLIMIT = 32768;
    #endregion

    public Dictionary<string, MessageResponse> gptDebugMessages;

    private void Awake()
    {
        openAI = new OpenAIClient(new
OpenAIAuthentication().LoadFromEnvironment());
        lifetimeCancellationTokenSource = new CancellationTokenSource();
        gptDebugMessages = new Dictionary<string, MessageResponse>();
    }


    protected void Start()
    {
DopeCoderController.Instance.uiController.inputField.onSubmit.AddListene
r(DopeCoderController.Instance.SubmitChat);
    }

    protected void OnDestroy()
    {
        if (DopeCoderController.Instance.Settings.saveLogs)
WriteConversationToFile();

        gptDebugMessages = null;
DopeCoderController.Instance.uiController.inputField.onSubmit.RemoveAllL
isteners();

        lifetimeCancellationTokenSource.Cancel();
        lifetimeCancellationTokenSource.Dispose();
        lifetimeCancellationTokenSource = null;

    }


    public async void SubmitAssistantResponseRequest(string msg = "What
exactly is all the code doing around me and what relationships do the
scripts have with one another?")
    {
        isChatPending = true;
        var assistant = await
openAI.AssistantsEndpoint.RetrieveAssistantAsync(AssistantID);
        Debug.Log($"{assistant} -> {assistant.CreatedAt}");

        if (threadID == null || threadID == string.Empty)
        {
            GPTthread = await
openAI.ThreadsEndpoint.CreateThreadAsync();
            threadID = GPTthread.Id;
        }

        CreateMessageRequest request;
        MessageResponse message;

        // check if char count exceed, if so make file, then submit to
GPTAssistant
        if (msg.Length > GPT4_CHARACTERLIMIT)
        {
            msg += "Please first make sure to read the file attached to
this message before responding.";
            MemoryStream ms = await WriteGPTQueryToStream(msg);
            // Calculate the number of files needed
            int numberOfFiles = (int)Math.Ceiling((double)ms.Length /
GPT4_CHARACTERLIMIT);
            FileReference[] files = new FileReference[numberOfFiles];
            string[] fileIDs = new string[numberOfFiles];
            for (int i = 0; i < numberOfFiles; i++)
            {
                string tempFilePath =
Path.Combine(Application.temporaryCachePath,
$"tempFile_{i}_{DateTime.Now:yyyy-MM-dd_HH-mm-ss}.txt");

                using (FileStream file = new FileStream(tempFilePath,
System.IO.FileMode.Create, FileAccess.Write))
                {
                    // Copy the portion of the MemoryStream into the
file
                    ms.Position = i * GPT4_CHARACTERLIMIT;
                    byte[] buffer = new
byte[Math.Min(GPT4_CHARACTERLIMIT, ms.Length - ms.Position)];
                    ms.Read(buffer, 0, buffer.Length);
                    file.Write(buffer, 0, buffer.Length);
                }
                files[i] = new FileReference { assetPath = tempFilePath
};

                var fileData = await
openAI.FilesEndpoint.UploadFileAsync(files[i].assetPath, "assistants");
                Debug.Log($"Exceeded character count, creating file
upload req {fileData.Id}");
                fileIDs[i] = fileData.Id;
                // Optionally, delete the temporary file after upload
                //File.Delete(tempFilePath);
            }
            msg = "okay, based on everything in the text files I've
given you in this message thread, what are some of the most important
classes you've identified? Please explain how everything works";
            //request = new CreateMessageRequest(msg, new[] {
fileIDs[0], fileIDs[1], fileIDs[2] });
            request = new CreateMessageRequest(msg, fileIDs);

            message = await
openAI.ThreadsEndpoint.CreateMessageAsync(threadID, request);
            //var messageFileMsg = await
DopeCoderController.Instance.openAI.ThreadsEndpoint.CreateMessageAsync(t
hreadID, requestFileMsg);
        }
        else
        {
            request = new CreateMessageRequest(msg);
            message = await
openAI.ThreadsEndpoint.CreateMessageAsync(threadID, request);
        }


        messageID = message.Id;
        Debug.Log($"{message.Id}: {message.Role}:
{message.PrintContent()}");

        if (!gptDebugMessages.ContainsKey(message.Id))
        {
            gptDebugMessages.Add(message.Id, message);
            onGPTMessageReceived?.Invoke($"User:
{message.PrintContent()}", MessageColorMode.MessageType.Sender);
        }

        var run = await GPTthread.CreateRunAsync(assistant);
        Debug.Log($"[{run.Id}] {run.Status} | {run.CreatedAt}");
        runID = run.Id;

        var messageList = await RetrieveAssistantResponseAsync();
        for (int index = messageList.Items.Count - 1; index >= 0; index-
-)
        {
            var _message = messageList.Items[index];
            Debug.Log($"{_message.Id}: {_message.Role}:
{_message.PrintContent()}");
            if (!gptDebugMessages.ContainsKey(_message.Id))
            {
                gptDebugMessages.Add(_message.Id, _message);
                onGPTMessageReceived?.Invoke($"{_message.Role}:
{_message.PrintContent()}",
                    _message.Role == Role.User ?
MessageColorMode.MessageType.Sender :
MessageColorMode.MessageType.Reciever);
            }
        }
    }

    public async void SubmitChatStreamRequst(string text)
    {
        conversation.AppendMessage(new OpenAI.Chat.Message(Role.User,
text));
```

```
        var assistantMessageContent =
DopeCoderController.Instance.uiController.AddNewTextMessageContent(Role.
Assistant);
        assistantMessageContent.text = "Assistant: ";

        try
        {
            var request = new ChatRequest(conversation.Messages);
            var response = await
openAI.ChatEndpoint.StreamCompletionAsync(request, resultHandler:
deltaResponse =>
            {
                if (deltaResponse?.FirstChoice?.Delta == null) { return;
}

                assistantMessageContent.text +=
deltaResponse.FirstChoice.Delta.ToString(); // populate response text
DopeCoderController.Instance.uiController.scrollView.verticalNormalizedP
osition = 0f;    // set ui to align with new message
            }, lifetimeCancellationTokenSource.Token);

            conversation.AppendMessage(response.FirstChoice.Message);

            onGPTMessageReceived?.Invoke(response,
MessageColorMode.MessageType.Reciever);
        }
        catch (Exception e)
        {
            switch (e)
            {
                case TaskCanceledException:
                case OperationCanceledException:
                    break;
                default:
                    Debug.LogError(e);
                    break;
            }
        }
        finally
        {
            if (lifetimeCancellationTokenSource is {
IsCancellationRequested: false })
            {
DopeCoderController.Instance.uiController.inputField.interactable = true
;
EventSystem.current.SetSelectedGameObject(DopeCoderController.Instance.u
iController.inputField.gameObject);
DopeCoderController.Instance.uiController.submitButton.interactable = tr
ue;
            }

            isChatPending = false;
        }

    }

    private async Task<ListResponse<MessageResponse>>
RetrieveAssistantResponseAsync()
    {
        var run = await
openAI.ThreadsEndpoint.RetrieveRunAsync(threadID, runID);
        Debug.Log($"[{run.Id}] {run.Status} | {run.CreatedAt}");
        RunStatus status = run.Status;
        while (status != RunStatus.Completed)
        {
            run = await
openAI.ThreadsEndpoint.RetrieveRunAsync(threadID, runID);
            Debug.Log($"[{run.Id}] {run.Status} | {run.CreatedAt}");
            status = run.Status;
            await System.Threading.Tasks.Task.Delay(500);
        }
        var messageList = await
openAI.ThreadsEndpoint.ListMessagesAsync(threadID);

        return messageList;
    }

    /// <summary>
    /// When character count is exceeded, return a MemoryStream
containing the GPT query text.
    /// </summary>
    /// <param name="text">Query Text for GPT</param>
    /// <returns>MemoryStream containing the text</returns>
    private async Task<MemoryStream> WriteGPTQueryToStream(string text)
    {
        var memoryStream = new MemoryStream();
        using (StreamWriter writer = new StreamWriter(memoryStream,
Encoding.UTF8, 1024, leaveOpen: true))
        {
            await writer.WriteAsync(text);
            await writer.FlushAsync(); // Ensure all data is written to
the stream

            // Reset the position of the stream to the beginning
            memoryStream.Position = 0;
            return memoryStream;
        }
    }


    public 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();
    }

    /// <summary>
    /// TODO: Format writing of file to be more properly ordered and
formatted (distinguish user/assistant)
    /// </summary>
    private void WriteConversationToFile()
    {
        // iterate over all messages and create a long string for now
        StringBuilder formattedData = new StringBuilder();
        foreach (MessageResponse mr in gptDebugMessages.Values)
        {
            formattedData.AppendLine($"{mr.Role}: {mr.PrintContent()}");
        }
        // Fire and forget method to write to a file asynchronously
        System.Threading.Tasks.Task.Run(async () => {
            string filename = $"ConversationData_{DateTime.Now:yyyy-MM-
dd_HH-mm-ss}.txt";
            string path = Path.Combine(Application.streamingAssetsPath,
filename);
            using (StreamWriter writer = new StreamWriter(path, false))
            {
                await writer.WriteAsync(formattedData.ToString());
                Debug.Log($"Wrote conversation file to {path}");
            }
        });
    }
}
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 Utilities.WebRequestRest;
//using Unity.VisualScripting;
using System.IO;
using OpenAI.Files;

public class GPTReflectionAnalysis : MonoBehaviour
{
    public DopeCoderController dopeCoderController;
    public ReflectionRuntimeController componentController; // Reference
to your component controller
    private OpenAIClient openAI; // OpenAI Client
    public string AssistantID;
    public string AssistantIDGPT3;
    public string AssistantIDGPT4;
    public bool textToSpeech;

    public SphereController sphereController;

    public Dictionary<string, MessageResponse> gptDebugMessages;

    public KeywordEvent[] keywordEvents;


    protected bool saveMode;
    private static bool isChatPending;  // manage state of chat requests
to prevent spamming


    #region GPTAssistantVariables
    private string threadID;
    private string messageID;
    private string runID;
    protected OpenAI.Threads.ThreadResponse GPTthread;
    private const int GPT4_CHARACTERLIMIT = 32768;
    #endregion

    private void Awake()
    {
        // Initialize the OpenAI Client
        openAI = new OpenAIClient();
        gptDebugMessages = new Dictionary<string, MessageResponse>();
    }


    protected void Start()
    {
dopeCoderController.uiController.inputField.onSubmit.AddListener(dopeCod
erController.SubmitChat);
        SpeechController.onSTT += ProcessVoiceInput;

        sphereController.SetMode(SphereController.SphereMode.Idle);
    }

    public void ProcessVoiceInput(string voiceInput) =>
RetrieveAssistant(voiceInput);

    /// <summary>
    /// invoked via settings button
    /// </summary>
    public void SetSaveMode(bool status) => saveMode = status;

    /// <summary>
    /// invoked via settings button
    /// </summary>
    public void SetGPT3Mode(bool status) {
        if (status) AssistantID = AssistantIDGPT3;
        else AssistantID = AssistantIDGPT4;
    }

    protected void OnDestroy()
    {
        if (saveMode) {
            WriteConversationToFile();
        }
        gptDebugMessages = null;
dopeCoderController.uiController.inputField.onSubmit.RemoveAllListeners(
);
        SpeechController.onSTT -= ProcessVoiceInput;
    }

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

    public void ToggleTTS() => textToSpeech = !textToSpeech;

    /// <summary>
    /// invoked externally via button press/mapping
    /// </summary>
    public 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? " +
            "Please leverage your knowledge of the code base as well
using the documentation that was given to you, specifically looking at
the classes specified in this message with respect to your
documentation.";

        // Combine the prompt with the data
        string combinedMessage = $"{gptPrompt}\n{dataForGPT}";
        Debug.Log(combinedMessage);
        try { RetrieveAssistant(combinedMessage); }
        catch (Exception e) { Debug.LogError(e); }
        finally {
            //if (lifetimeCancellationTokenSource != null) {}
            isChatPending = false;
        }

    }



    private async void RetrieveAssistant(string msg = "What exactly is
all the code doing around me and what relationships do the scripts have
with one another?") {
        sphereController.SetMode(SphereController.SphereMode.Listening);
        isChatPending = true;
        var assistant = await
openAI.AssistantsEndpoint.RetrieveAssistantAsync(AssistantID);
        Debug.Log($"{assistant} -> {assistant.CreatedAt}");

        if (threadID == null || threadID == string.Empty) {
            GPTthread = await
openAI.ThreadsEndpoint.CreateThreadAsync();
            threadID = GPTthread.Id;
        }

        CreateMessageRequest request;
        MessageResponse message;

        if (msg.Length > GPT4_CHARACTERLIMIT) { // check if char count
exceed, if so make file, then submit to GPTAssistant
            msg += "Please first make sure to read the file attached to
this message before responding.";
            MemoryStream ms = await WriteGPTQueryToStream(msg);
            // Calculate the number of files needed
            int numberOfFiles = (int)Math.Ceiling((double)ms.Length /
GPT4_CHARACTERLIMIT);
            FileReference[] files = new FileReference[numberOfFiles];
            string[] fileIDs = new string[numberOfFiles];
            for (int i = 0; i < numberOfFiles; i++) {
                string tempFilePath =
Path.Combine(Application.temporaryCachePath,
$"tempFile_{i}_{DateTime.Now:yyyy-MM-dd_HH-mm-ss}.txt");

                using (FileStream file = new FileStream(tempFilePath,
System.IO.FileMode.Create, FileAccess.Write)) {
                    // Copy the portion of the MemoryStream into the
file
                    ms.Position = i * GPT4_CHARACTERLIMIT;
                    byte[] buffer = new
byte[Math.Min(GPT4_CHARACTERLIMIT, ms.Length - ms.Position)];
```

```
                    ms.Read(buffer, 0, buffer.Length);
                    file.Write(buffer, 0, buffer.Length);
                }
                files[i] = new FileReference { assetPath = tempFilePath
};

                var fileData = await
openAI.FilesEndpoint.UploadFileAsync(files[i].assetPath, "assistants");
                Debug.Log($"Exceeded character count, creating file
upload req {fileData.Id}");
                fileIDs[i] = fileData.Id;
                // Optionally, delete the temporary file after upload
                //File.Delete(tempFilePath);
            }
            msg = "okay, based on everything in the text files I've
given you in this message thread, what are some of the most important
classes you've identified? Please explain how everything works";
            //request = new CreateMessageRequest(msg, new[] {
fileIDs[0], fileIDs[1], fileIDs[2] });
            request = new CreateMessageRequest(msg, fileIDs);

            message = await
openAI.ThreadsEndpoint.CreateMessageAsync(threadID, request);
            //var messageFileMsg = await
openAI.ThreadsEndpoint.CreateMessageAsync(threadID, requestFileMsg);
        } else {
            request = new CreateMessageRequest(msg);
            message = await
openAI.ThreadsEndpoint.CreateMessageAsync(threadID, request);
        }


        messageID = message.Id;
        Debug.Log($"{message.Id}: {message.Role}:
{message.PrintContent()}");

        if (!gptDebugMessages.ContainsKey(message.Id)) {
            gptDebugMessages.Add(message.Id, message);
            UpdateChat($"User: {message.PrintContent()}");
        }

        var run = await GPTthread.CreateRunAsync(assistant);
        Debug.Log($"[{run.Id}] {run.Status} | {run.CreatedAt}");
        runID = run.Id;

        var messageList = await RetrieveAssistantResponseAsync();
        for (int index = messageList.Items.Count - 1; index >= 0; index-
-) {
```

```
            var _message = messageList.Items[index];
            Debug.Log($"{_message.Id}: {_message.Role}:
{_message.PrintContent()}");
            if (!gptDebugMessages.ContainsKey(_message.Id)) {
                gptDebugMessages.Add(_message.Id, _message);
                if (textToSpeech)
                {
                    Debug.Log("Attempting to invoke speech req");
//dopeCoderController.speechController.GenerateSpeech(_message.PrintCont
ent());
                }
                UpdateChat($"{_message.Role}:
{_message.PrintContent()}", _message.Role == Role.User ?
MessageColorMode.MessageType.Sender :
MessageColorMode.MessageType.Reciever);
            }
        }
    }


    private async Task<ListResponse<MessageResponse>>
RetrieveAssistantResponseAsync()
    {
        var run = await
openAI.ThreadsEndpoint.RetrieveRunAsync(threadID, runID);
        Debug.Log($"[{run.Id}] {run.Status} | {run.CreatedAt}");
        RunStatus status = run.Status;
        while (status != RunStatus.Completed)
        {
            run = await
openAI.ThreadsEndpoint.RetrieveRunAsync(threadID, runID);
            Debug.Log($"[{run.Id}] {run.Status} | {run.CreatedAt}");
            status = run.Status;
            await System.Threading.Tasks.Task.Delay(500);
        }
        var messageList = await
openAI.ThreadsEndpoint.ListMessagesAsync(threadID);

        return messageList;
    }


    private async void RetrieveAssistantResponse()
    {
        //var message = await
openAI.ThreadsEndpoint.RetrieveMessageAsync(threadID, messageID);

        //Debug.Log($"{message.Id}: {message.Role}:
```

```
{message.PrintContent()}");
        var run = await
openAI.ThreadsEndpoint.RetrieveRunAsync(threadID, runID);
        Debug.Log($"[{run.Id}] {run.Status} | {run.CreatedAt}");
        var messageList = await
openAI.ThreadsEndpoint.ListMessagesAsync(threadID);

        foreach (var message in messageList.Items)
        {
            Debug.Log($"{message.Id}: {message.Role}:
{message.PrintContent()}");
        }
    }

    /// <summary>
    /// When character count is exceeded, return a MemoryStream
containing the GPT query text.
    /// </summary>
    /// <param name="text">Query Text for GPT</param>
    /// <returns>MemoryStream containing the text</returns>
    private async Task<MemoryStream> WriteGPTQueryToStream(string text)
{
        // Create a MemoryStream
        var memoryStream = new MemoryStream();

        // Use StreamWriter to write the text to the MemoryStream
        using (StreamWriter writer = new StreamWriter(memoryStream,
Encoding.UTF8, 1024,leaveOpen: true)) {
            await writer.WriteAsync(text);
            await writer.FlushAsync(); // Ensure all data is written to
the stream

            // Reset the position of the stream to the beginning
            memoryStream.Position = 0;

            return memoryStream;
        }
    }

    /// <summary>
    /// TODO: Format writing of file to be more properly ordered and
formatted (distinguish user/assistant)
    /// </summary>
    private void WriteConversationToFile() {
        // iterate over all messages and create a long string for now
        StringBuilder formattedData = new StringBuilder();

        foreach (MessageResponse mr in gptDebugMessages.Values) {
            formattedData.AppendLine($"{mr.Role}: {mr.PrintContent()}");

        }
        // Fire and forget method to write to a file asynchronously
        System.Threading.Tasks.Task.Run(async () => {
            string filename = $"ConversationData_{DateTime.Now:yyyy-MM-
dd_HH-mm-ss}.txt";
            string path =
Path.Combine(Application.streamingAssetsPath,filename);
            using (StreamWriter writer = new StreamWriter(path, false))
{
                await writer.WriteAsync(formattedData.ToString());
                Debug.Log($"Wrote conversation file to {path}");
            }
        });
    }

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



    public void UpdateChat(string newText, MessageColorMode.MessageType
msgType = MessageColorMode.MessageType.Sender)
    {
        dopeCoderController.uiController.UpdateChat(newText, msgType);
        if (msgType == MessageColorMode.MessageType.Reciever)
sphereController.SetMode(SphereController.SphereMode.Talking);
        if (msgType == MessageColorMode.MessageType.Sender)
sphereController.SetMode(SphereController.SphereMode.Listening);

    }




    /// <summary>
    /// Anytime submitchat is invoked, we first search for keywords
    /// Example:
    /// invoke function ScanAndPopulateClasses()
    /// view variables of ChatBehavior
    /// view variables of ColorChangeScript
    /// </summary>
    /// <param name="_tex"></param>
    /// <returns></returns>

    private bool ParseKeyword()
    {
        string input = dopeCoderController.uiController.inputField.text;
        foreach (KeywordEvent k in keywordEvents)
        {
            if (input.Contains(k.Keyword,
StringComparison.OrdinalIgnoreCase))
            {
                Debug.Log($"Keyword {k.Keyword}");
                k.keywordEvent.Invoke();
                return true;
            }
        }
        return false;
    }


    public void GetHelpText()
    {
        StringBuilder helpTextStrBuilder = new StringBuilder();
        helpTextStrBuilder.AppendLine("## Available Commands\n");

        foreach (KeywordEvent k in keywordEvents)
        {
            helpTextStrBuilder.AppendLine($"- **{k.Keyword}**:
{k.Description}");
        }

        string helpText = helpTextStrBuilder.ToString();
        UpdateChat(helpText);
    }


    public void InvokeFunction()
    {
        string _text = dopeCoderController.uiController.inputField.text;
        string _func = ParseFunctionName(_text);
        if (!string.IsNullOrEmpty(_func))
        {
            Debug.Log($"Function name {_func}");
            componentController.SearchFunctions(_func);
        }
    }

    public void ViewClassVariables()
    {
        string _text = dopeCoderController.uiController.inputField.text;
        string className = ParseClassName(_text, "view variables of ");
        if (!string.IsNullOrEmpty(className))
        {
            Debug.Log($"Viewing variables of class {className}");
            //componentController.PrintAllVariableValues(className);
            // update references
            componentController.ScanAndPopulateClasses();
            string localQueryResponse =
componentController.GetAllVariableValuesAsString(className);
            UpdateChat(localQueryResponse);
        }
    }

    public void ViewVariable()
    {
        string _text = dopeCoderController.uiController.inputField.text;
        string variableName = ParseVariableName(_text, "view variable
");
        if (!string.IsNullOrEmpty(variableName))
        {
            // update references
            componentController.ScanAndPopulateClasses();
            Debug.Log($"Viewing variable {variableName}");
componentController.PrintVariableValueInAllClasses(variableName);
}
    }




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

public class KeywordEventManager : MonoBehaviour
{
    [SerializeField] public KeywordEvent[] keywordEvents;

    /// <summary>
    /// Anytime submitchat is invoked, we first search for keywords
    /// Example:
    /// invoke function ScanAndPopulateClasses()
    /// view variables of ChatBehavior
    /// view variables of ColorChangeScript
    /// </summary>
    /// <param name="_tex"></param>
    /// <returns></returns>

    public bool ParseKeyword()
    {
        string input =
DopeCoderController.Instance.uiController.inputField.text;
        foreach (KeywordEvent k in keywordEvents)
        {
            if (input.Contains(k.Keyword,
StringComparison.OrdinalIgnoreCase))
            {
                Debug.Log($"Keyword {k.Keyword}");
                k.keywordEvent.Invoke();
                return true;
            }
        }
        return false;
    }


    public void GetHelpText()
    {
        StringBuilder helpTextStrBuilder = new StringBuilder();
        helpTextStrBuilder.AppendLine("## Available Commands\n");

        foreach (KeywordEvent k in keywordEvents)
        {
            helpTextStrBuilder.AppendLine($"- **{k.Keyword}**:
{k.Description}");
        }

        string helpText = helpTextStrBuilder.ToString();
        DopeCoderController.Instance.UpdateChat(helpText,
MessageColorMode.MessageType.Reciever);
    }


    public void InvokeFunction()
    {
        string _text =
DopeCoderController.Instance.uiController.inputField.text;
        string _func = ParseFunctionName(_text);
        if (!string.IsNullOrEmpty(_func))
        {
            Debug.Log($"Function name {_func}");
DopeCoderController.Instance.componentController.SearchFunctions(_func);
       }
    }

    public void ViewClassVariables()
    {
        string _text =
DopeCoderController.Instance.uiController.inputField.text;
        string className = ParseClassName(_text, "view variables of ");
        if (!string.IsNullOrEmpty(className))
        {
            Debug.Log($"Viewing variables of class {className}");
            //componentController.PrintAllVariableValues(className);
            // update references
DopeCoderController.Instance.componentController.ScanAndPopulateClasses(
);
            string localQueryResponse =
DopeCoderController.Instance.componentController.GetAllVariableValuesAsS
tring(className);
            DopeCoderController.Instance.UpdateChat(localQueryResponse,
MessageColorMode.MessageType.Reciever);
        }
    }

    public void ViewVariable()
    {
        string _text =
DopeCoderController.Instance.uiController.inputField.text;
        string variableName = ParseVariableName(_text, "view variable
");
        if (!string.IsNullOrEmpty(variableName))
        {
            // update references
DopeCoderController.Instance.componentController.ScanAndPopulateClasses(
);
            Debug.Log($"Viewing variable {variableName}");
DopeCoderController.Instance.componentController.PrintVariableValueInAll
Classes(variableName);
        }
    }




    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
    }


}

[System.Serializable]
public class KeywordEvent
{
    public string Keyword;
    public string Description;
    public UnityEngine.Events.UnityEvent keywordEvent;
}

[Serializable]
public class FileReference
{
    public string assetPath;
    public bool markedForRemoval;
}
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 bool ScanCollidersOnly;

    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;


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

        if (ScanCollidersOnly) {    // 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($"Total {colliders.Length}");
        } else {
            PopulateClassInfo();
            Debug.Log($"Total {classCollection.Count}");
        }

    }

    void PopulateClassInfo() {

        MonoBehaviour[] scripts =
UnityEngine.Object.FindObjectsOfType<MonoBehaviour>();
        foreach (MonoBehaviour script in scripts) {
            Type type = script.GetType();
            var scope = type.Namespace;
            if (scope == null || (!scope.StartsWith("Unity") &&
!scope.StartsWith("UnityEngine.UI") && !scope.StartsWith("TMPro"))) {
```

```
                Debug.Log(script.GetType().FullName + " is used within "
+ script.gameObject.name);
                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(script);
                    classInfo.VariableValues[field.Name] = value;
                }

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

            }
        }
    }

    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}");
                }
            }
            Debug.Log(variableValues.ToString());
            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;

    public void SetColliderTrigger(bool status) => ScanCollidersOnly =
status;

}

[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 OpenAI;
using OpenAI.Audio;
using OpenAI.Models;
using System.Collections;
using System.Collections.Generic;
using System.Threading;
using UnityEngine;
using System;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading.Tasks;
using UnityEngine.UI;
using Utilities.Audio;
using Utilities.Encoding.Wav;
using Utilities.Extensions;

public class SpeechController
{
    public AudioSource audioSource;
    public GameObject audioPanel;

    public delegate void OnSpeechToText(string text);
    public static OnSpeechToText onSTT;

    public SpeechController(AudioSource audioSource, GameObject
audioPanel)
    {
        this.audioSource = audioSource;
        this.audioPanel = audioPanel;
        Init();
    }

    public void Init()
    {
        OnValidate();
        GPTInterfacer.onGPTMessageReceived += GenerateSpeech;
    }

    public void OnValidate()
    {
        audioSource.Validate();
    }

    public void Destroy()
    {
        GPTInterfacer.onGPTMessageReceived -= GenerateSpeech;
    }

    /// <summary>
    /// invoked via settings toggle menu
    /// </summary>
    public void EnableAudioInterface(bool status)
    {
        audioPanel.SetActive(status);
        //recordButton.SetActive(status);
        audioSource.enabled = status;
    }

    public async void GenerateSpeech(string text,
MessageColorMode.MessageType messageType)
    {
        if (!DopeCoderController.Instance.Settings.tts) return;
        //text = text.Replace("![Image](output.jpg)", string.Empty);
        var request = new SpeechRequest(text, Model.TTS_1);
        var (clipPath, clip) = await
DopeCoderController.Instance.gptInterfacer.openAI.AudioEndpoint.CreateSp
eechAsync(request,
DopeCoderController.Instance.gptInterfacer.lifetimeCancellationTokenSour
ce.Token);
        Debug.Log($"Speech request fired {clip.name}");
        audioSource.clip = clip;
        audioSource.Play();
        if (DopeCoderController.Instance.Settings.debugMode)
Debug.Log(clipPath);

    }

    public void ToggleRecording()
    {
        RecordingManager.EnableDebug =
DopeCoderController.Instance.Settings.debugMode;

        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 (DopeCoderController.Instance.Settings.debugMode)
Debug.Log(path);
        Debug.Log("Processing recording");

        try
        {
DopeCoderController.Instance.uiController.recordButton.interactable = fa
lse;
            var request = new AudioTranscriptionRequest(clip,
temperature: 0.1f, language: "en");
            var userInput = await
DopeCoderController.Instance.gptInterfacer.openAI.AudioEndpoint.CreateTr
anscriptionAsync(request,
DopeCoderController.Instance.gptInterfacer.lifetimeCancellationTokenSour
ce.Token);

            if (DopeCoderController.Instance.Settings.debugMode)
Debug.Log($"voice input {userInput}");

            DopeCoderController.Instance.uiController.inputField.text =
userInput;
            onSTT?.Invoke(userInput);
DopeCoderController.Instance.uiController.inputField.interactable = true
;
        }
        catch (Exception e)
        {
            Debug.LogError(e);
DopeCoderController.Instance.uiController.inputField.interactable = true
;
        }
        finally
        {
DopeCoderController.Instance.uiController.recordButton.interactable = tr
ue;
        }
    }

}
using System.Collections;
using UnityEngine;

public class SphereController : MonoBehaviour
{
    public enum SphereMode
    {
        Idle,
        Talking,
        Listening
    }

    [System.Serializable]
    public class ModeSettings
    {
        public Color color;
        public float colorTransitionDuration;
        public float scaleChangeDuration;
        public float minScale;
        public float maxScale;
        public float oscillationSpeed;
    }

    public SphereMode mode = SphereMode.Idle;
    private SphereMode currentMode;
    public ModeSettings idleSettings;
    public ModeSettings talkingSettings;
    public ModeSettings listeningSettings;
    private Renderer rend;
    private Color targetColor;
    private ModeSettings currentSettings;

    private Vector3 initialScale;
    private Vector3 targetScale;
    private Coroutine colorTransitionCoroutine;
    private Coroutine scaleChangeCoroutine;
    private bool isScaling = false;

    void Start()
    {
        rend = GetComponent<Renderer>();
        initialScale = transform.localScale;
        SetMode(mode);
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            SetMode(mode);
        }

        switch (currentMode)
        {
            case SphereMode.Idle:
                UpdateIdleState();
                break;
            case SphereMode.Talking:
                if (!isScaling)
                    UpdateTalkingState();
                break;
            case SphereMode.Listening:
                if (!isScaling)
                    UpdateListeningState();
                break;
        }
    }

    // TOOD: Switch these to abstract and extended classes with their
own update method
    void UpdateIdleState() => ScaleOscillation(idleSettings);
    void UpdateTalkingState() => ScaleOscillation(talkingSettings);

    void UpdateListeningState() => ScaleOscillation(listeningSettings);

    void ScaleOscillation(ModeSettings settings)
    {
        //Debug.Log($"Attempting to scale oscillation, state 
{isScaling}");
        float lerpParameter = Mathf.PingPong(Time.time * 
settings.oscillationSpeed, 1f);
        float scaleFactor = Mathf.Lerp(settings.minScale, 
settings.maxScale, lerpParameter);
        transform.localScale = initialScale * scaleFactor;
    }

    public void SetMode(SphereMode newMode)
    {
        currentMode = mode = newMode;
        switch (mode)
        {
            case SphereMode.Idle:
                ApplyModeSettings(idleSettings);
                break;
            case SphereMode.Talking:
                ApplyModeSettings(talkingSettings);
                break;
            case SphereMode.Listening:
                ApplyModeSettings(listeningSettings);
                break;
        }
    }

    void ApplyModeSettings(ModeSettings settings)
    {
        currentSettings = settings;
        targetColor = settings.color;

        // Start color transition coroutine
        if (colorTransitionCoroutine != null)
            StopCoroutine(colorTransitionCoroutine);
        colorTransitionCoroutine = 
StartCoroutine(ColorTransitionCoroutine(targetColor, 
settings.colorTransitionDuration));

        StartScaling();
    }

    IEnumerator ColorTransitionCoroutine(Color targetColor, float 
duration)
    {
        Color initialColor = rend.material.color;
        float timer = 0f;
        while (timer < duration)
        {
            timer += Time.deltaTime;
            float lerpValue = timer / duration;
            rend.material.color = Color.Lerp(initialColor, targetColor,
lerpValue);
            yield return null;
        }

        rend.material.color = targetColor;
    }

    void StartScaling()
    {
        // Set target scale
        targetScale = Vector3.one * currentSettings.minScale;
        // Start scaling coroutine
        if (scaleChangeCoroutine != null)
            StopCoroutine(scaleChangeCoroutine);
        isScaling = true;
        scaleChangeCoroutine =
StartCoroutine(ScaleChangeCoroutine(currentSettings.minScale,
currentSettings.scaleChangeDuration));
    }

    IEnumerator ScaleChangeCoroutine(float targetScale, float duration)
    {
        Vector3 initialScale = transform.localScale;
        float timer = 0f;

        while (timer < duration)
        {
            timer += Time.deltaTime;
            float lerpValue = timer / duration;
            transform.localScale = Vector3.Lerp(initialScale,
this.targetScale, lerpValue);
            yield return null;
        }

        transform.localScale = this.targetScale;
        isScaling = false;
    }
}
using LogicUI.FancyTextRendering;
using OpenAI;
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.UI;
using Utilities.Extensions;

public class UI_Controller : MonoBehaviour
{
    public GameObject pfb_chatMsgUI;

    [SerializeField]
    public Button submitButton;

    [SerializeField]
    public Button recordButton;
    public GameObject startRecordingIcon;
    public GameObject stopRecordingIcon;

    [SerializeField]
    public TMP_InputField inputField;

    [SerializeField]
    public RectTransform contentArea;

    [SerializeField]
    public ScrollRect scrollView;


    private void Awake()
    {
        OnValidate();
    }

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


    public void ToggleInput(bool status)
    {
        inputField.ReleaseSelection();
        inputField.interactable = status;
        submitButton.interactable = status;
    }

    public void ToggleMicIcon()
    {
        if (startRecordingIcon.activeSelf)
        {
            startRecordingIcon.SetActive(false);
            stopRecordingIcon.SetActive(true);
        } else
        {
            startRecordingIcon.SetActive(true);
            stopRecordingIcon.SetActive(false);
        }
    }

    public void UpdateChat(string newText, MessageColorMode.MessageType
msgType)
    {
        //inputField.text = newText;
        var assistantMessageContent = AddNewTextMessageContent(msgType
== MessageColorMode.MessageType.Sender ? Role.User : Role.Assistant);
        assistantMessageContent.GetComponent<MarkdownRenderer>().Source
= newText;
        scrollView.verticalNormalizedPosition = 0f;
        //if (audioPanel.activeInHierarchy &&
!newText.Contains("User:")) GenerateSpeech(newText);
    }

    public TextMeshProUGUI AddNewTextMessageContent(Role role)
    {
        var textObject = Instantiate(pfb_chatMsgUI, contentArea);
        textObject.name = $"{contentArea.childCount + 1}_{role}";
        //var textObject = new GameObject($"{contentArea.childCount +
1}_{role}");
        //textObject.transform.SetParent(contentArea, false);
        var msgColorMode = textObject.GetComponent<MessageColorMode>();
        if (role == Role.User)
msgColorMode.SetMode(MessageColorMode.MessageType.Sender);
        else
msgColorMode.SetMode(MessageColorMode.MessageType.Reciever);

        var textMesh = msgColorMode.messageText;
        MarkdownRenderer mr =
textMesh.gameObject.AddComponent<MarkdownRenderer>();
        mr.RenderSettings.Monospace.UseCustomFont = false;
        mr.RenderSettings.Lists.BulletOffsetPixels = 40;
        textMesh.fontSize = 24;
        textMesh.enableWordWrapping = true;
        return textMesh;
    }
```

```
}
// 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 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);
    }
}
```