Universal Text

Project & Software Specification
UW Reality Labs

Tech Lead/PM	Nathan Reilly, Justin Lin
GitHub	https://github.com/uwrealitylabs/universal-text-unity
Scrum Board	
Expected Delivery	EOT W25

Changes to Spec:

Change Date	Change Author	Change Reason
Aug. 17, 2024	Justin Lin	Initial Author
Aug. 31, 2024	Nathan Reilly & Justin Lin	Technical revisions for Text Label composition. Added Introduction
Sep 27, 2024	Nathan Reilly	Large-scale revision of the implementation
Jan 20, 2025	Nathan Reilly	Revision of the UTT and UTS implementation & other updates for W25

Point Persons:

Role	Name	Contact Info
Sedra Lead	Peter Teertstra	peter.teertstra@uwaterloo.ca
Team Lead	Justin Lin Nathan Reilly	jk3lin@uwaterloo.ca nathanmreilly@gmail.com
UW Reality Labs Leads	Vincent Xie Kenny Na Justin Lin	vincent.xie@uwaterloo.ca kenny.na@uwaterloo.ca justin.lin1@uwaterloo.ca

Table of Contents

Introduction

When you prompt a virtual assistant (for example Meta AI on Raybans glasses), what happens when you ask “What am I looking at”? Currently, the pipeline seems rather simplistic. The cameras on the glasses take a picture, that picture is passed through a model that can assign text labels to images, and finally that text label describing the whole image is passed into an LLM. This process, especially the step where a model must describe everything in an image using words, is often inaccurate.

What if we could build a system that…

- …provides a richer text summary of a virtual environment, complete with descriptions of how objects compose each other, are placed within/next to/on top of each other?
- …also describes how you, the user, is interacting with that environment at any moment? Could we assign additional text to describe that you are pointing at a specific object, or reaching out for one?
- …runs in real time, that is, can constantly update every frame to provide an updated description. That way, we wouldn’t have to wait for text generation, and we could create a live captioning system?
- …runs entirely on-device, meaning this information is never sent to the cloud?

If we created this, we could use it for…

- …in-application virtual assistants that make use of a rich text summary for high-accuracy responses
- …virtual science labs where users could receive detailed auto-generated scientific explanations about tools and objects they interact with

- …dynamic VR scene descriptions for the visually impaired, describing layout and objects, or even what they're holding, pointing at or nearby to
- …and so much more

Universal Text aims to explore this. We will develop a structured software package for Unity, which is composed of several scripts. We will begin with fully-virtual environments—artificial scenes that we build and label ourselves. The goal is to create a system that allows Unity VR developers to easily label their GameObjects with descriptions, and seamlessly integrate tutorials, live captioning for accessibility, or virtual assistants into their application.

Technical Specification

Tech Stack

The tech stack for this project involves Unity, OpenAI Whisper, and Meta's Llama 3 LLM running locally. C# will be the primary scripting language.
The target platform is Meta Quest, inclusive of Meta Quest 2, Meta Quest 3, and Meta Quest Pro headsets. The project will build on top of Meta's XR All-in-One SDK (UPM) and therefore it will be a dependency.

The Real-Time Text Representation (Our Goal)

The main components of the Universal Text project are dedicated to building a dynamic textual representation of a user's virtual environment and their interactions within it. To implement this, we take advantage of Unity's object hierarchy system. In a Scene, GameObjects represent entities that a user can perceive and interact with, such as characters, props and scenery. Using Universal Text, developers first create descriptions for relevant GameObjects, optionally including custom attributes and relationships with other GameObjects. During runtime, our tool parses GameObjects that the user is currently interacting with, nearby to, or is able to perceive, combining these individual descriptions into one cohesive, **real-time textual representation**.

From here we denote this real-time textual representation of a user's virtual environment as **RTR**. The main focus of the project is creating a tool that is capable of generating an **RTR** during runtime whenever requested.

Universal Text Tag

Overview

This is the core component of Universal Text: a C# script named **Universal Text Tag (*UTT*)**. This script contains all of the data needed by the **Universal Text Scanner** to describe the GameObject it is attached to. This data includes a GameObject's basic *description*, and its *attributes*.

Fields

Description: string

A basic description of the GameObject

- E.g. "A small colorful fridge magnet in the shape of a fruit"

Attributes: List<Attribute>

A list of this GameObject's attributes. Attributes can be thought of as descriptive statements of the GameObject, which are generally capable of changing during runtime. For example, an attribute of a water bottle could be "is 90% filled with water", and later on this may change to something like "is 40% filled with water".

- E.g. [{"Stuck to", <Refrigerator Universal Text Tag>}]

Implementation

The Universal Text Tag script is implemented as a C# class inheriting from MonoBehaviour so that it can be attached to GameObjects. Its fields are to be easily customizable by developers, likely with the help of custom *PropertyDrawers* for the *Attributes* field, as developers will need to provide delegates to initialize instances of the *Attribute* class (and thus the derived *Relation* class).

The UTT also implements the ToString() method, which is used by the UTS when a string representation of the GameObject is needed. When converting itself to a string, the UTT first converts each *Attribute* into a string using its ToString() method, and then appends the list of all attributes to the *Description*.

```
/// <summary>
/// A tag that marks this GameObject to be recognized by the UTS and thus included in the RTR
/// </summary>
Unity Script (12 asset references) | 16 references
public class UniversalTextTag : MonoBehaviour
{
    /// <summary>
    /// Description of the Object
    /// </summary>
    public string Description;

    /// <summary>
    /// Array containing attributes of the Object
    /// </summary>
    public List<Attribute> Attributes = new List<Attribute>();

    3 references
    public override string ToString()
    {
        string representation = $"{String.Copy(Description)}";

        // Clean out invalid attributes
        List<Attribute> cleanedAttributes = new List<Attribute>();
        foreach (Attribute attribute in Attributes)
        {
            if (attribute.Valid) cleanedAttributes.Add(attribute);
        }

        if (cleanedAttributes.Count == 0)
        {
            return representation += ".";
        }
        representation += ", which ";
        if (cleanedAttributes.Count == 1)
        {
            return representation += $"{cleanedAttributes.First()}.";
        }
        foreach (Attribute attribute in cleanedAttributes)
        {
            if (attribute == cleanedAttributes.Last())
            {
                representation += $"and {attribute}.";
            } else
            {
                representation += $"{attribute}, ";
            }
        }
```

The following are the two inner classes that the UTT uses for its *Attributes* field:

- **Attribute:** a class that represents a single attribute, i.e. some descriptive statement about the GameObject. The properties of this class are:
    - Description: string: A formatted string that contains a placeholder for the attribute's *Value*
    - Value: object: The current value of this attribute, to be inserted at the placeholder of the *Description,* e.g. if our *Description* is "{0}% full", and our *Value* is *30*, then we get "30% full"

- Valid: boolean: Whether or not this Attribute is currently relevant to this GameObject. This will be false whenever this attribute is not worth mentioning when describing our object.

Whenever the *Value* and *Valid* properties are accessed, the class calls their associated **delegates** to retrieve their current value. This is to permit fully customizable attributes to the developer, as the mechanism with which the *Attribute* instance retrieves the necessary data to generate its description is left up to the developer alone.

When an Attribute is converted to a string using ToString(), it places whatever retrieved from the Value property into the placeholder inside the Description property.

```
/// <summary>
/// Describes an attribute of this UniversalTextTag's Object. Attribute values can either be initialized with a value type or
/// a function that returns the current value of the attribute when called.
/// </summary>
9 references
public class Attribute
{
    /// <summary>
    /// Description of the attribute
    /// </summary>
    protected string _description;

    /// <summary>
    /// Delegate referencing a method that returns this Attribute's current value
    /// </summary>
    private Func<object> _valueGetter;

    /// <summary>
    /// Delegate referencing a method that returns whether or not this Attribute is currently valid (i.e. relevant to the RTR)
    /// </summary>
    private Func<bool> _validGetter;

    /// <summary>
    /// Current value of this Attribute
    /// </summary>
    4 references
    protected object Value { get => _valueGetter(); }

    /// <summary>
    /// Validity of this Attribute (true if relevant to the RTR, false otherwise)
    /// </summary>
    1 reference
    public bool Valid { get => _validGetter(); }

    /// <summary>
    /// Initializes an Attribute with provided 'formattedDescription', and delegates referencing getter functions for
    /// the Attribute's current value and validity.
    /// </summary>
    /// <param name="formattedDescription">Formatted description of the Attribute, must be compatible with String.Format(formattedDescription, valueGetter()) when Value is not null</param>
    /// <param name="valueGetter">Reference to getter function that retrieves the associated attribute's current value</param>
    /// <param name="validGetter">Reference to getter function that returns a bool value representing whether or not this Attribute is currently relevant to the RTR</param>
    1 reference
    public Attribute(string formattedDescription, Func<object> valueGetter = null, Func<bool> validGetter = null)
    {
        _description = formattedDescription;

        if (valueGetter != null) _valueGetter = valueGetter;
        else _valueGetter = () => null;

        if (validGetter != null) _validGetter = validGetter;
        else _validGetter = () => true;
    }

    1 reference
    public override string ToString()
    {
        if (Value == null)
        {
            return _description;
        }
        return String.Format(_description, Value);
    }
}
```

- **Relation (inherits from Attribute)**: a class that represents a relation with other GameObject(s) (which must also have a UTT). Given that this class inherits from *Attribute*, it contains all of the same data and is initialized with the same delegates, and it also includes the following property:
    - Tags: List<UTT>: A list of all of the UTTs attached to GameObjects that this relation is held with. E.g. a pencil case GameObject might hold the relation “contains” with several other pen and pencil GameObjects.

The value of the *Tags* property is also retrieved using a **delegate**, similar to the *Value* and *Valid* properties.

When a Relation is converted to a string using ToString(), it will combine the inherited *Description* and *Value (if used)* fields, as well as the string representations of the *Tags* that the relation is held with.

This may look something like “contains a **red pen**, a **blue sharpie**, and a **pink eraser**”, where “contains” is our *Description* field, our *Value* field is unused (as it will often be for Relations, since we usually only care about the *Tags*), and the *Tags* field contains the UTTs referencing the red pen, blue sharpie and pink eraser GameObjects.

Universal Text Scanner

Overview

The **Universal Text Scanner (*UTS*)** is a singleton class that, when requested, finds the UTTs of any GameObjects currently relevant to the RTR, and aggregates their text representations into a single text description of the user's current environment and interactions. This is the final step in the creation of our *RTR* (*for now*).

This script handles the aggregation of all *relevant* GameObjects into our final *RTR*. To accomplish this goal one might ask: *How do we know which GameObjects are relevant?* And, for each of these GameObjects, *in what **way** is it relevant?* For example, a hammer may be relevant because the user is hammering a nail with it, or an NPC may be relevant because the user is speaking to it. To generate a rich, detailed *RTR* we want to be able to answer these questions.

To do so, we will abstract the notions of "*which GameObjects"* and "*why this GameObject*" into the concept of a **Search Point.** For our purposes, a Search Point will represent a context (*that usually holds some relation with the user*) within which GameObjects can either be a part of, or not a part of. For example, a useful Search Point could be "All of the GameObjects that the user is nearby to", or "All of the GameObjects the user is holding".

Now, we will consider all GameObjects that are a part of any Search Point's context to be *relevant*. With this, the UTS needs only to collect all GameObjects from the contexts of all Search Points, and aggregate them to create the *RTR*.

Then, to describe the **way** in which each GameObject is *relevant*, we describe the context of the GameObject's Search Point, prepending it to the GameObject's description. This produces something like "The user is holding a screwdriver, and a small screw", where our GameObjects are the "screwdriver" and "small screw", and the Search Point containing both of these has the context: "All of the GameObjects the user is holding".

Thus, the UTS produces the *RTR* by iterating through each Search Point, prepending the list of the UTTs provided by each one by the description of the Search Point's context, and combining the results of all of the Search Points together into one large string.

Implementation

Although our UTS class (*more on it later*) bears the name of the system, our **SearchPoint** is the true star player. Within each different Search Point contains the functionality to, during runtime, fetch a list of GameObjects currently within its respective context.

When implementing our SearchPoint, we want to provide maximum flexibility to developers using our package. It would be ideal if developers could define their own SearchPoints, with custom descriptions and searching functionality that may be tailored specifically for their

game/app. For example, a survival game may want a Search Point that fetches all the available food in the vicinity of the user. We can't create a Search Point for every possible context, so instead we give our developers the tools necessary for them to create their own Search Points as needed, using an interface.

Our SearchPoint interface, ISearchPoint, will look something like this:

```
0 references
public interface ISearchPoint
{
    /// <summary>
    /// A description of the SearchPoint's context
    /// </summary>
    0 references
    string Description { get; }

    /// <summary>
    /// Fetches all GameObjects within the context of this SearchPoint
    /// </summary>
    0 references
    List<UniversalTextTag> Search();
}
```

This provides the developer the ability to customize the description of a SearchPoint's context, as well as the freedom to implement their own Search() function, returning a list of all the GameObjects within the context.

Additionally, our package may also include its own default SearchPoints, such as all GameObjects the user is *pointing at*, *nearby to*, *looking at* or *grabbing*. To implement our default SearchPoints, we will use the same interface we provide to the developers.

Usage of our ISearchPoint interface (or any C# interface really) is straightforward, and looks something like this:

```
0 references
public class GrabbingSearchPoint : ISearchPoint
{
    1 reference
    public string Description { get => "The user is grabbing"; }

    1 reference
    public List<UniversalTextTag> Search()
    {
        List<UniversalTextTag> grabbedGameObjects = new List<UniversalTextTag>();

        // (implementation)

        return grabbedGameObjects;
    }
}
```

Here, we created a new SearchPoint, a class named “GrabbingSearchPoint”, which covers the context of all GameObjects the user is *grabbing*. Here, we say that GrabbingSearchPoint *implements* ISearchPoint. To our UTS, any class that implements ISearchPoint will be considered a valid search point that can be included in the RTR.

Now, with a flexible interface for the creation of Search Points, all we have left to do is to actually use them! This will be the job of our Universal Text Scanner.

In the UniversalTextScanner singleton, we will store a list containing the SearchPoints that we will use to generate the RTR. Additionally, we define a function called **Generate()**, which aggregates the contents of all SearchPoints into one large string, and returns it as the RTR. Our implementation of the UTS will thus look something like this:

```
public class UniversalTextScanner : MonoBehaviour
{
    private List<ISearchPoint> _searchPoints;

    /// <summary>
    /// Generates RTR by aggregating all search points
    /// </summary>
    0 references
    public string Generate()
    {
        string rtr = "";
        foreach (ISearchPoint searchPoint in _searchPoints)
        {
            List<UniversalTextTag> tags = searchPoint.Search();
            string desc = searchPoint.Description;
            // ...
        }
        // ...
        return rtr;
    }

    0 references
    public void AddSearchPoint(ISearchPoint searchPoint)
    {
        _searchPoints.Add(searchPoint);
    }
}
```

And with that, we've covered the core functionality of Universal Text!

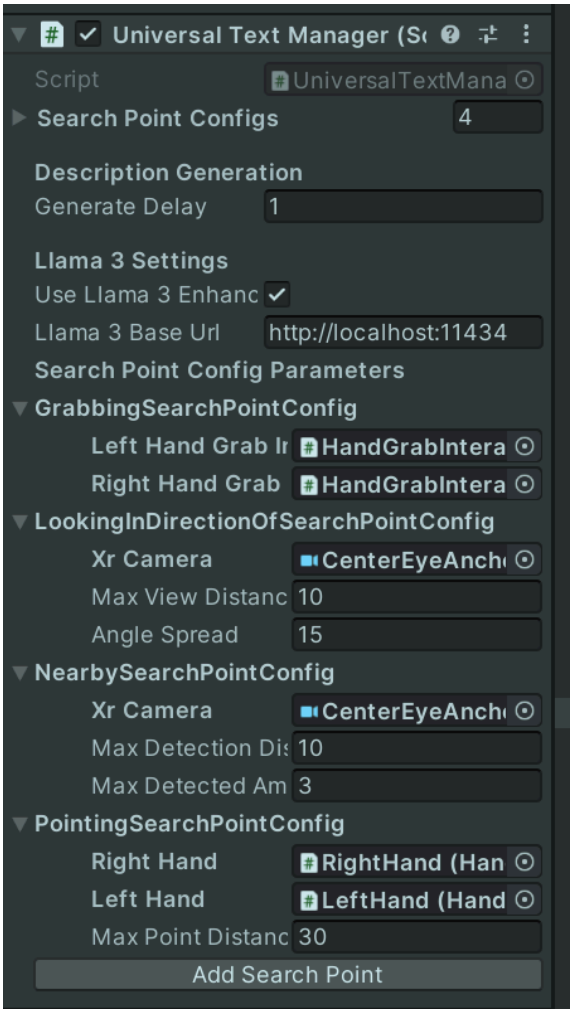
Universal Text Manager

Overview

The **Universal Text Manager (UTM)** is a Monobehaviour class that acts as a configuration interface to the developer. In a sense, this is the *final* component of our package, as it is responsible for referencing all of the core components of our package and exposing their parameters to the developer.

As such, although its responsibility is quite significant, it is quite a simple component. All the hard work is already done within the likes of the **Universal Text Scanner** and **Universal Text Tag**, so all this script needs to do is provide our developer with the interface needed to integrate their package within their VR application through the Inspector Window.

The UTM script is attached to the identically named *UniversalTextManager* prefab, found under *Assets/Scripts/Prefabs.* The intention is that the developer will have an instance of this prefab in any scene where they want the Universal Text package to be active. Additionally, they can use the inspector window to configure the package as needed by selecting their instance of the UniversalTextManager prefab.



The figure to the left shows the current (2025-05-22) inspector view of the UTM. We have exposed parameters for each search point that the developer wants to include in the RTR, and an option to add/remove them. Additionally, there is parameters relating to how often to automatically generate a new RTR, and the Llama3 integration for additional features (e.g. **Universal Text Prompter**).

In addition to exposing functionality to the developer, this script can be conveniently used for prototyping as you work on adding features to the package. You can simply invoke your new features from the UTM's Start or Update functions (a benefit of inheriting from Monobehaviour), so that you can more easily debug them.

Any core features to the package that are intended to be invoked in some way during runtime can be done so through the UTM.

Universal Text Prompter & Additional Scripts

The final feature of Universal Text involves making this description available as an input prompt for virtual assistants. Using the output of our scripts, user's could prompt an LLM with a question such as "Is this safe to eat? If not, which one of these should I use for disposal?" (referring to the UTS output example above). A simple script named **Universal Text Prompter** would append the textual description at the end of the user's question and prompt a local instance of Llama 3 (using a simple API call). This output could be returned and displayed to the user within Unity.

User voice input can be captured using OpenAI's Whisper API for voice-to-text transcription. We would need to build a simple demo voice recognition script, named **Universal Text Voice,** that activates either when the user says a key phrase or makes a gesture. It would then use Whisper to transcribe text from the Quest microphone, and pass it to the Universal Text Prompter. The Prompter would get a response from Llama 3 and display the output in Unity.

Running Llama 3 locally on Windows

Acceptance Criteria

Listed below are more general, overarching criteria for what the software should accomplish. These criteria will become more specific and technical over time.

1. Universal Text can display a detailed text description of a virtual environment, and describe how the user interacts with it (for a certain amount of preset gestures).
2. This process can occur rapidly, so that from an observer's point of view, the text description is always up-to-date with what is happening in the virtual environment.
3. A virtual assistant demo has been created, where users can prompt an LLM that runs on-device. The assistant should be able to produce accurate and helpful answers that take into consideration the virtual environment of the user.
4. The finished software package is hosted on GitHub, and is easily importable into a new Unity Project using the Package Manager.

Resources Required

A team of 5-10 developers is the projected team size for this project.

Meta Quest 2 and 3 headsets are required for testing the project in Unity. We may need 2-3 total headsets available to be used by the team during work sessions/accessible from a locker to be used on developer's own time within the team space. Not all headsets are required to be the more expensive Quest 3: hand tracking functionality should be the same across models