
Unity Accessibility Extensions Documentation

Release 0.1

Sam Hebditch

May 30, 2019

Contents:

1	Introduction	3
1.1	Why?	3
1.2	What are others doing on this front?	3
1.3	How?	4
2	Rationale	5
2.1	Raycasting	5
2.2	Script Modularity	6
2.3	Object Descriptions	7
2.4	Priority Queue - <i>event driven branch only</i>	7
3	Installation Process	9
3.1	Initial Setup	9
3.2	Additional Options	9

This set of documents is designed to outline and explain the principles behind the creation of the Unity Accessibility Extensions, alongside documentation of the code and underlying technologies that enable this functionality.

The extensions, are still in development, as is this documentation. Over time, pages will be added, removed, tweaked, and changed as necessary, and things are bound to change. Consider this v0a, an initial version that we will build off.

Sam Hebditch, 16/05/19

CHAPTER 1

Introduction

Think of this as a bit of a primer. In this page, I'm aiming to cover why exactly we built these extensions, alongside how a brief, high level overview of how everything **currently** works.

1.1 Why?

This project was initially born out of a question asked during a demonstration of an augmented reality application that we'd built. This question was simply, "How can we make this accessible - What features does Unity have for accessibility?". This was something I had no idea about, and as such, proceeded to dive into and research.

What I found, was that Unity did nothing in the way to work with Native accessibility APIs on a users device. And whilst there were projects such as [UAP](#), it only offered options to make the UI accessible, and not much else. This then led me to question and explore how we could make augmented reality accessible, and what could be done with Unity to enable it to hook into native, on-device APIs for Text-To-Speech etc. and create an inclusive experience.

Being a person that is Visually Impaired, my key focus has been building out the extensions to enable those with visual impairments to access Augmented and Virtual Reality, alongside existing Unity 3D games and applications. However, there is no reason why these extensions could not be built out to be inclusive of other disabilities. From the outside, Unity might appear like a tool that is nigh on impossible to make accessible. However, it's versatile nature, and ability to interact with user created native code, make it suprisingly extensible.

1.2 What are others doing on this front?

Over the course of development, I've seen efforts from research teams at Microsoft to make VR accessible. However, these appear to tie-in to the Graphics Rendering Stack/DirectX API on Windows, to provide a one-size-fits-all solution that requires no developer input or modification of the code. The team are also building out extensions for Unity that tie into these low-level tools, to provide additional levels of interaction.

However, this has been one of the few examples I've seen when it comes to making mixed reality and Unity applications accessible. In a brief study of some of most popular AR applications, such as Pokémon Go, IKEA Place, and Google's Measure AR, most were lacking in any kind of accomidations or modifications for those with specialised

requirements such as text-to-speech, descriptions of objects, larger text etc. Upon testing these apps out, and exploring how they worked with services like TalkBack on, this revealed a large issue to me, **Nobody was making AR/VR apps accessible!**

1.3 How?

I'll be saving a deep dive into the technologies and techniques behind the extension later on in this documentation, but here's a high level overview to whet your appetite for the time being.

We use Raycasting to provide object detection, distance estimation. Once an object has been hit by a ray, we pull several bits of data from it, including it's name, a description (via a custom component that allows a developer to include a long string.), and it's distance. We feed these in, alongside camera rotation data (as it's safe to assume that in AR and VR, the camera in the scene is located in roughly the same position or perspective that the users head or viewpoint will be), into a script to be parsed and turned into fully descriptive strings (such as, "The object is 1.5m away from you, double tap to hear the description attached to it"), which get fed into a script that handles passing over the data to native code which taps into the Text-to-Speech Engines on both Android and iOS.

Currently, as of writing (16th May) - I'm exploring and creating a queuing system for information passed to the TTS, so that a developer, user can choose which event they want to be spoken first, and also to ensure that the TTS isn't flooded with requests to handle rotation, object description etc. all at the same time.

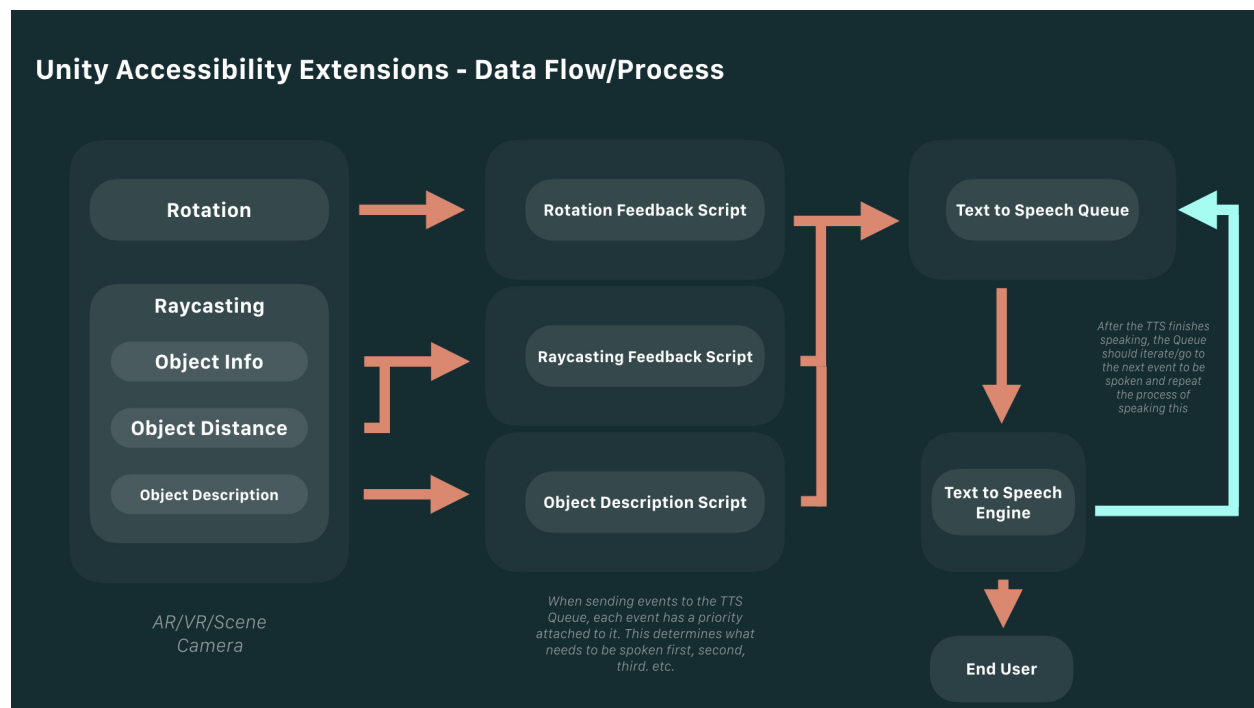


Fig. 1: A flow chart outlining the flow of data/information from the camera in the scene, to it's endpoint, the text to speech engine.

The aim of this page is to cover, outline and describe the design decisions taken during the development of these extensions for Unity. The aim of this page is also to describe some of the assumptions that we make in the scripts to produce something that is easy to drop in, regardless of project size or configuration, and with minimal configuration from the developer/user.

As mentioned above, one of the key goals for this project is to produce something that is modular, and easy to drop in for a developer, with little to no input required or needed from them. Accessibility shouldn't just be in the form of making games and projects accessible, but making the tools to do so easy to implement and an easy process that encourages people to factor in accessibility.

2.1 Raycasting

Raycasting is currently the only method we use for determining objects in a scene, it was chosen, as it's part of the standard Unity engine (as part of the physics system), it's also got little to no performance impact. Whilst not tested, I also believe that Unity is able to handle multiple rays, meaning that it's a solution that could integrate easily into existing games that use Raycasting as part of their object collision/detection/physics systems.

2.1.1 Raycast source

Initially, I experimented with using rays fired from the camera in the scene, however I found that using some augmented reality platforms don't quite work. This led me to creating the `Casting Cube` component, which when enabled and set up, will follow/mirror the direction of the main camera. From here, we cast the ray in a forward direction, using `transform.forward`.

When describing this functionality, I allude, and liken it to a cane for a blind or visually impaired person, as it allows the user to sweep across the scene using their device, much like a cane would be used to sweep in the real world. When paired up with the other scripts and functionality I've built, the user gets feedback, just like they would when the cane hits something in the real world.

We do assume that the Camera is going to be paired up and configured to match the devices rotation and movement, since we've focused on Augmented Reality so far, this typically makes perfect sense and has been the case in all of our tests so far.

2.1.2 Raycast setup

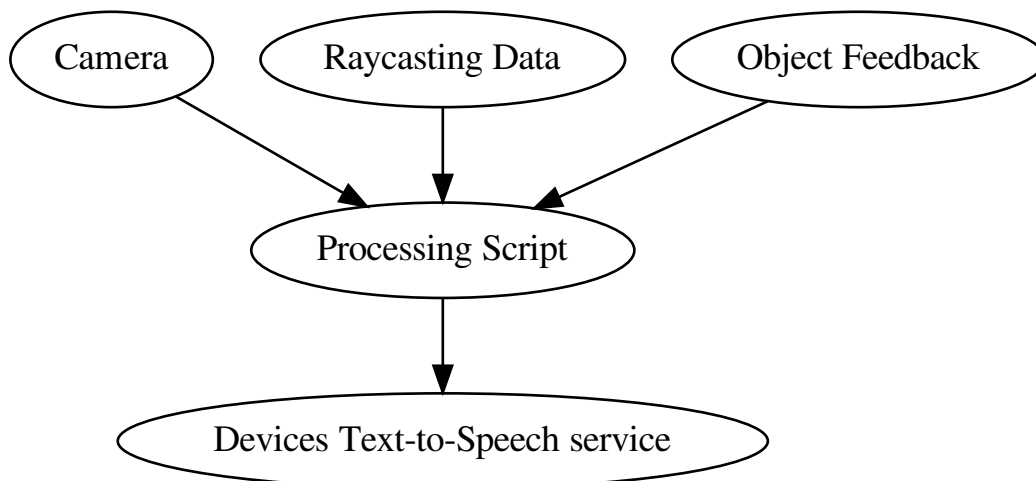
In the setup script, we again, make a few assumptions to allow things to work somewhat seamlessly, regardless of the set up that the developer has in place. We use tags to identify object, and rely on some pre-existing tags in Unity. Primarily, we rely on the `MainCamera` tag initially to determine the camera in the scene, and place all of the components required for raycasting as children of it.

It is worth noting though, that whilst we initially rely on the `MainCamera` tag, we do shift it over to a `ScnCamera` tag that gets set up, and referenced throughout the scripts that have been created.

It's also worth noting, that as we're using Raycasting, all objects that you want to be detectable by the end user require some form of collider on them to function/be picked up by the raycasting script.

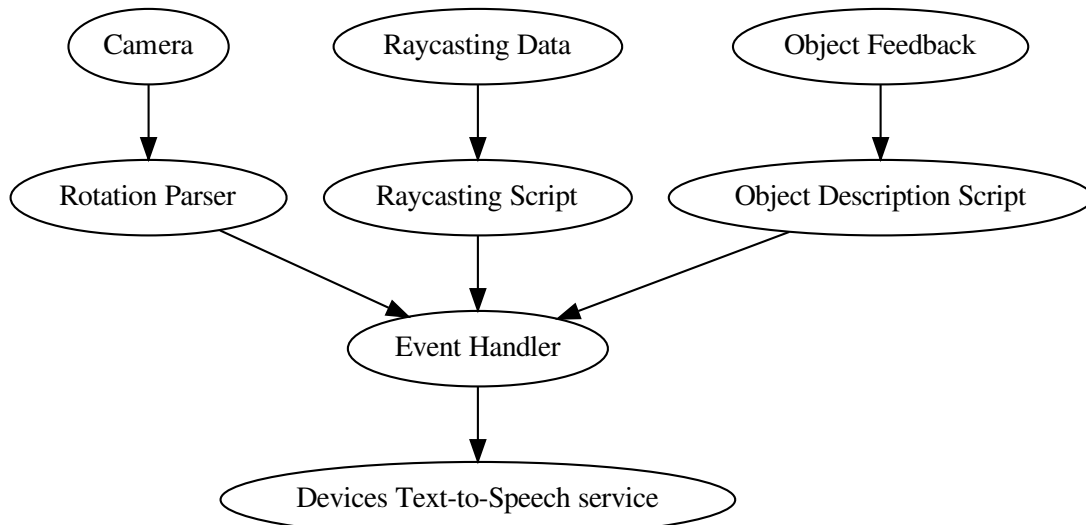
2.2 Script Modularity

Initially, the scripts and code for this project was all handled within a singular script, there was no communication between scripts, and things became very messy and hard to debug and modify without fear of breaking something else. Below is an example of how this behaved:



Flow chart illustrating processing flow

However, since then, we've moved away from this approach to something more modular, that allows for information to be referenced and pulled from across the various scripts, and piped in to whatever may require it. This looks like this:



Flow chart illustrating processing flow

This allows us to pull data from the various scripts easily, and create bespoke functionality that only relies on certain functions, without having to invoke and work with the entire accessibility extension codebase. Having modular, yet centralised points to pull from has been successful, however, I'm not sure how performant it'd be in the long term or on larger projects. We're continuously investigating things such as ECS, or more event driven systems however.

2.3 Object Descriptions

Rather than write some bespoke structure or format for object descriptions, I've settled upon using long strings with an Editor UI to accommodate holding longer strings and wrapping them to the size of the editor. This decision was done to make things easier, and also saving converting between types etc. when passing data to the event handler, and then the Text-to-Speech engine on a users device.

As a general rule of thumb for object descriptions, try and make them as descriptive as possible, but succinct. It's worth testing out how your descriptions sound on a device with TalkBack or VoiceOver enabled, just to see if they're too long or if they potentially get in the way of a user receiving other bits of information.

2.4 Priority Queue - *event driven branch only*

In the event driven branch, there is configuration tied to each event that determines the priority of an event. As a developer, you can remap and change the priority levels, if you feel it makes sense to do so. Currently the priority levels are as follows:

- Priority 1: Raycasting Feedback: This will always take priority, as the main means for the user to interact with the AR/VR/MR world.
- Priority 2: Rotation Feedback: An additional bit of information that will help a blind or visually impaired user orient themselves, however, not as important as the raycast feedback

- Priority 3: Object Description: As this is a bit of feedback that requires user action to trigger it, it's the lowest priority currently.

It's possible to add an unlimited amount of priorities, there is a custom struct set up so that an int, alongside a string can be passed along through Unity's messaging system. This int is used to define the priority of the event, and is passed on as such to the queuing system itself.

CHAPTER 3

Installation Process

As mentioned on the rationale page, the aim for these extensions was to be able to easily set up and configure these extensions with little to no developer input (reconfiguration of scenes, refactoring code, etc.) As such, the installation process is as automated as possible, nonetheless, it's worth documenting it, including current quirks and oddities when it comes to configuring the extensions at the moment.

3.1 Initial Setup

Initial set up is relatively easy, download the latest package from the releases page, and import it as you would any other unity package.

image placeholder

Once this is done, you'll notice a new item appear within your Unity menu bar, much like the above screenshot. To start the initial setup, click *placeholder*. A simple dialog will appear that gives you options for configuring the components required in the script, along with the status of each of these components.

screenshot of status window

There is a two step process to setting up the components, first, you'll need to set up the global configuration object, which is done simply by pressing *placeholder name*, then you'll need to setup the camera and create the `Casting Cube`, which is also done via simple button push

3.2 Additional Options

There are some additional options contained within the initial setup window, including a toggle for a system wide debug mode, which will relay debugging information as required.