Unity Accessibility Extensions Documentation

Release Alpha

Sam Hebditch

Oct 28, 2019

Contents:

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.

These extensions are designed to aid visually impaired or blind people access Augmented, Virtual and Mixed Reality experiecnes that have been created within Unity. One of the design goals for these extensions, along with making experiecnes accessible, was to require as little input, or initial bring up to encourage developers to implement them. Unity itself, is slowly making strides towards accessibility, with a new accessibility and inclusivity focus and thread being set up on their community forums. however, it's currently very lacking in featuers to make games and experiences created within Unity accessible.

These tools make use of native operating system APIs, as outlined within the "Native Overview" page to provide native fucntionality to aid the user.

I hope that these extensions prove a useful starting point, resouce, or reference and when paired with this documentation, can provide you with a good idea of the methodology and process behind making an experience accesible for those who are visually impaired or blind.

Sam Hebditch.

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.

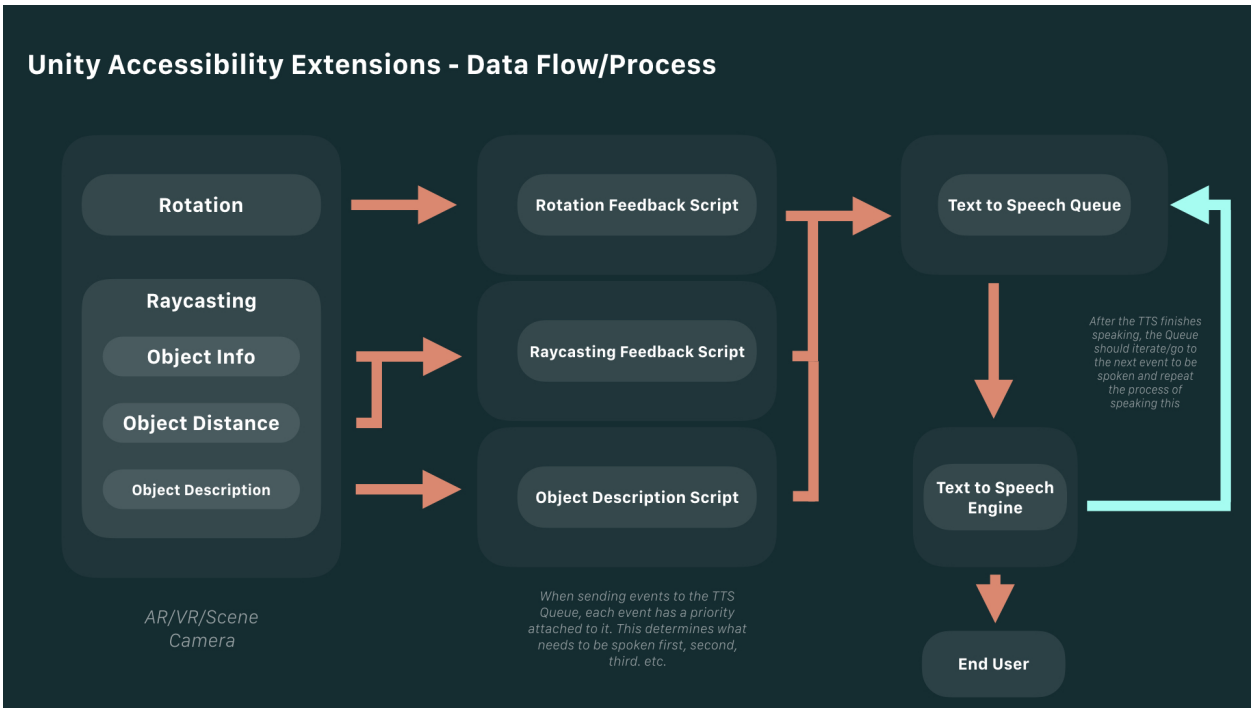


Fig. 1: A flow chart outlining the flow of data/information from the camera in the scene, to it's endpoint, the text to speech engine.

CHAPTER 2

Rationale

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 mutiple 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 foward direction, using `transform.forward`.

When describing this functionality, I allude, and liken it to a cane for a blind or visualy 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

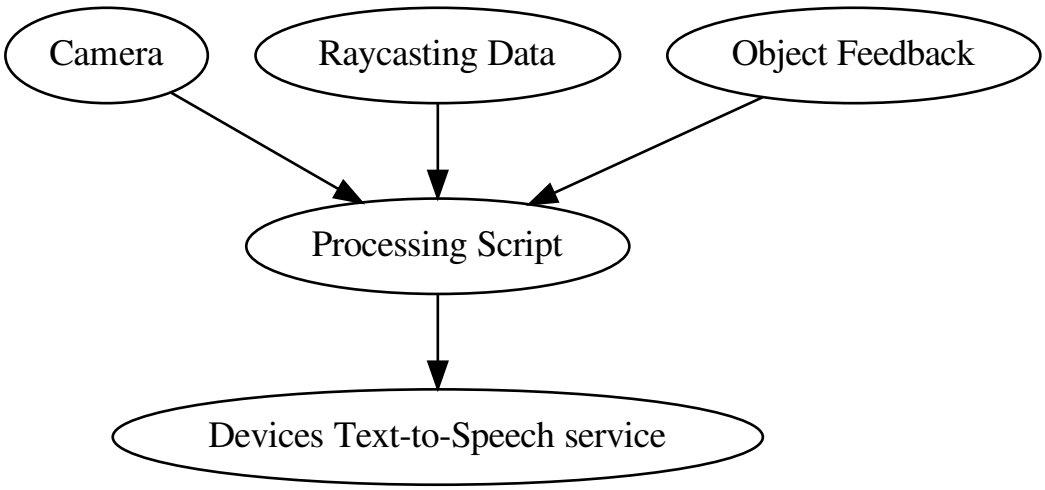
In the setup script, we again, make a few assumptions to allow things to work somewhat seamlessly, regardless of the set up that the developer has in place. We use tags to identify object, and rely on some pre-existing tags in Unity. Primarily, we rely on the `MainCamera` tag initially to determine the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

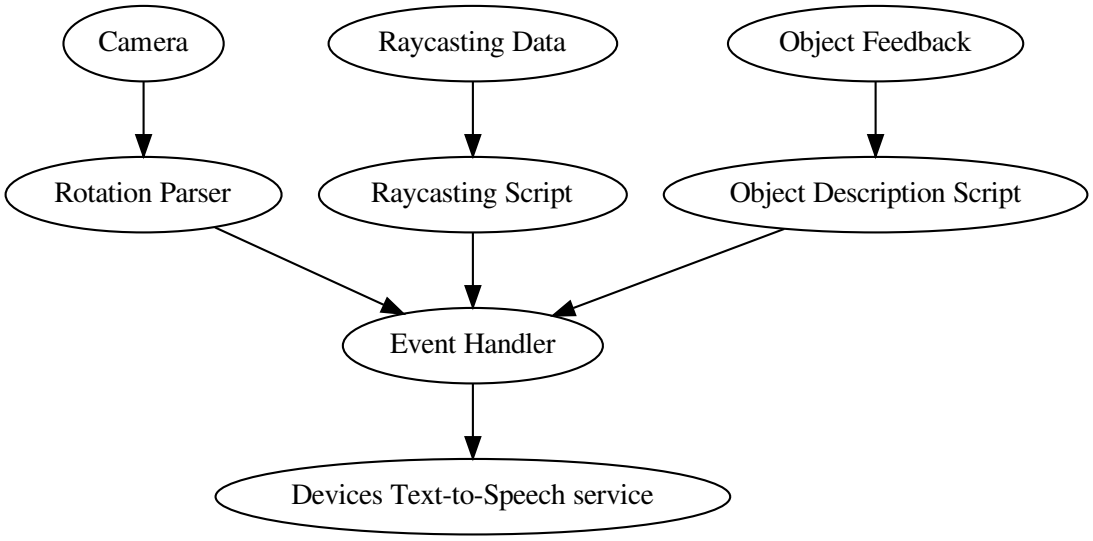
2.2 Script Modularity

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 ssize 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

As of the 30th May 2019 - SH has merged Master and ExperimentalEventDelegation Branches, making this the default behaviour as of now (until further tweaks and changes)

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 informtation that will help a blind or visually impaired user orient theirselves, 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.

It's worth noting though that the priority queueing system currently adds in a lot of latency. SH, as of 12th Sept. '19 has experimented with potential resolutions and fixes for this, including using a fixed size prioirty queue, but has not had much luck at the moment.

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 custom Unity package.

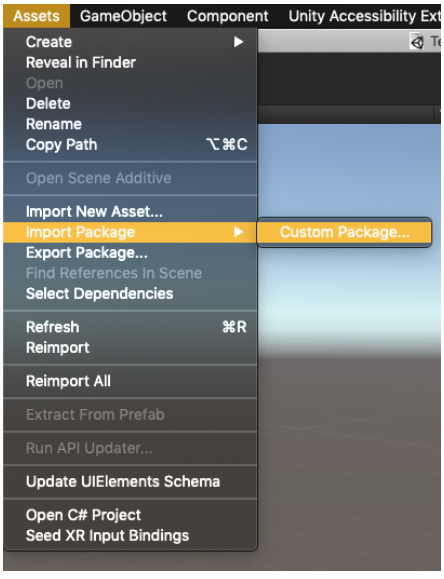


Fig. 1: A screenshot depeicting the menus needed to go through to import custom packages.

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.

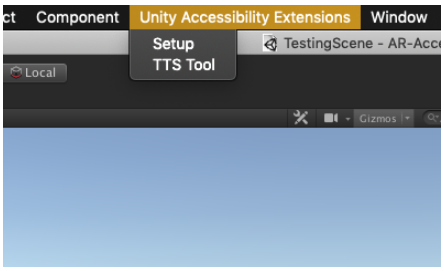
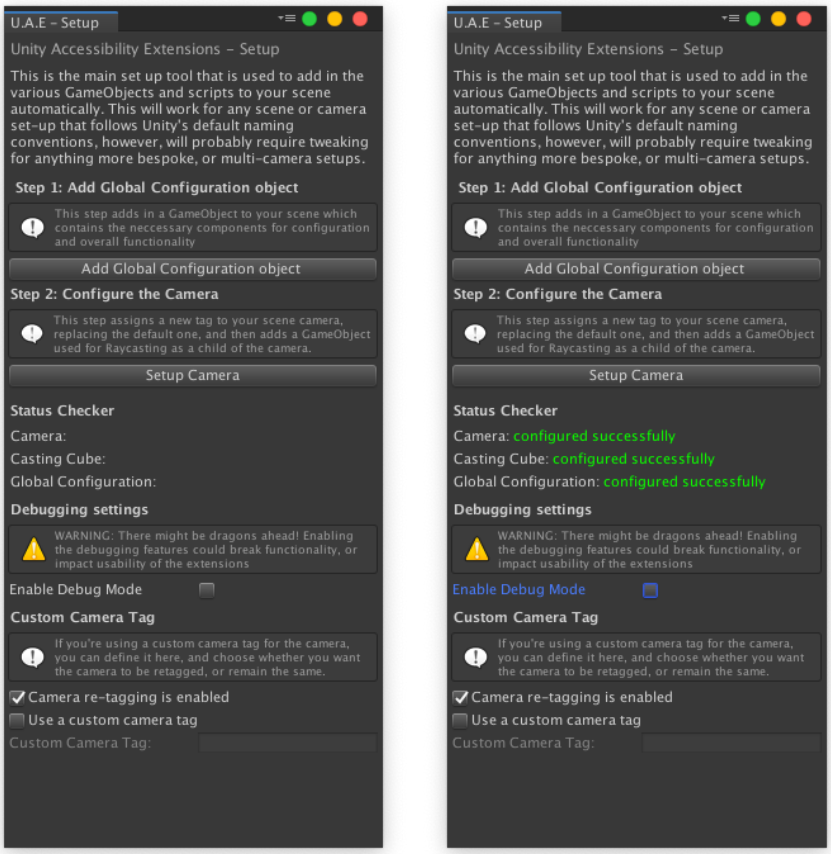


Fig. 2: The menu items that appear once the package import is complete

There is a two step process to setting up the extensions, handily outlined within the status window. We firstly set up the global configuration object, which handles all of the main scripts, before re-tagging the camera and adding in our raycasting source object.



The above images showcase the setup dialog before, and after the setup process has been completed.

3.2 Additional Options

There are some additional options contained within the initial setup window, below is a brief outline and explanation of these.

- **Debug Mode:** This is mode enables debugging messaging, and other information to be visible when working within the editor. It will also enable debugging lines when working with Raycasting in the editor, alowing you to view the directions that the raycast is going in, whether it's intersecting or hitting an object, or casting infinitely into the distance.
- **Custom Camera Tag** Currently, the script will create a new tag called `ScnCam` and apply this to camera, these options can circumvent this, and disable this functionality, instead allowing the user to use the default Unity camera tag, or a completely bespoke camera tag, allowing for integration (as some render pipelines might be dependant on custom tags etc.)

CHAPTER 4

Native Code Overivew

This is designed to give you an overivew of the Native Code and integrations that are used throughout the accesibility extensions, and explain how, and why, we use them (Although, I'd like to thing some of it its fairly self explanatory!)

4.1 What platforms utilise native code?

Currently, due to Unity platform limitations, most, if not all of the current platforms that we support and work with (Android, iOS, MacOS etc.) use some degree of native code to enable certain elements of functionality that are simply not possible within Unity. The key bit of fucntionality enabled by these elements of native code is Text-to-Speech and speech synthesis, something with Unity is not aware of, yet capable of interfacing with, itself. Alongside this, we also use native code to detect things like the Locale of the users device, and whether they've got accessibility services enabled. I have plans to use native code in the future to provide things such as haptic feedback for devices that support it.

4.2 Structure and Layout

We utilise and follow Unity's guidelines for creating plugins where possible. Following the /Plugins/ folder structure where required. Currently, the plugins we use are spread across multiple different scripts and files, however, there is the potential for this to be consolidated and tidied up in the future (However, not now, as having things slightly spread and disconnected like this makes it easier to debug).

4.3 A note on macOS

Whilst it was initially planned that I'd use native code, to tie into the native `NSSpeechSynthesizer` - However, this has been slightly difficult to achieve, and has preseented more problems than the TTS implementation on iOS did.

Instead, we use the C# `System.Diagnostics.Process` functionality, to trigger off system processes. In this case, this works well, as macOS has a built in `say` command, which can be triggered via the terminal.

There are also additional plans to implement a picker for the various voices availible on a users device, to enable developers to test out how their strings sound in different locales, currently, this has been a p.i.t.a. due to serialization fun.

4.4 A further note on Windows

Initially, I had planned to use `System.Speech.Synthesis` in C#, however, I found that Untiy and Mono don't include this functionality, and importing and including the DLLs seems to be a massive pain, with inconsistent results.

Looking for alternatives, I found a wrapper for the native C++ Speech API, created by Chad Weisshaar (this can be found here). I did look to build upon this, by exposing some additional functionality that native functionality has on other platforms (e.g. rate/speed of speech, pitch control), however found that I in the time frame I set myself, I couldn't get this to work. Also, limited knowledge of SAPI, C++ and C, seemed to get in the way.

It's also worth noting that the Windows native code is currently 64-Bit only. Chad does provide source for rebuilding to 32-Bit though, meaning it's possible to build and recompile this if required.

4.5 Native Plugin Overview

As mentioned above, both iOS and Android utilise native plugins to provide text-to-speech functionality. We also use tiny bits of native code to detect locale of the device, as well as the status of accessibility services (e.g. whether the user has accessibility services enabled, or not).

Windows platforms currently use native code, but to a limited extent, with simple TTS relying on the system default settings being the only supported functionality.

Each bit of native code is hooked up and contained within a couple of scripts, that expose simple APIs that can be used to do things such as speak out a string, or return the devices location value, or the status of the native text-to-speech engine.

Native code can be found in the following areas:

- **`TTS.cs`**
  - All native code is contained within one string, which uses ifdefs to call upon the appropriate code, and return a ICU locale value, which gets passed on where needed.
- **`TextToSpeech.cs`**
  - Native code is spread across a couple of different functions within this script, to handle setting up the text-to-speech engine where needed, actually speaking out the appropriate strings, and stopping the speech where needed.
- **`WindowsVoice.cs`**
  - This provides native TTS under windows, native functions from the C++ wrapper/plugin for the speech API, are passed through, and exposed in here.
- **`OSXSaySpeech.cs`**
  - This handles the say command, and command execution and lifespan on macOS.

CHAPTER 5

Useful Links and Resources

Starting out this project, I didn't find much in the way of useful links and resources for making Unity experiences accessible, yet alone making any kind of XR experience accessible.

Since starting this journey, however, I've found several links, tools, resources and references that may prove useful for someone diving into this, or curious about this!

5.1 Communities

- **Unity: Accessibility and Inclusivity Thread** https://forum.unity.com/threads/accessibility-and-inclusion.694477/

  Currently a bit dead, but a worthwhile insight that I'm hoping will get updated as the Unity team implement and improve upon accessiblity features within the editor, and at the API/engine level.

- **XR Access Initiative** https://www.xraccess.org/

  A community focused around making AR, VR and XR accessibile to those with visual impairements. Offers a wide variety of links to tools, guidelines, and players within the XR accesibility space.

- **W3C Immersive Web Working Group** https://www.w3.org/immersive-web/

  Whilst this a community centered around immersive web technologies, such as WebXR, there is some crossover as the group begin to explore how current web APIs, or future APIs can be used to make these accessible. In November 2019, the W3C are planning on hosting an inclusive design workshop for the immersive web, which should produce some interesting results for those looking to eplore web based immserive tech, links to that can be found here.

5.2 Tools and Resources