Klemens Strasser, BSc

Unity Accessibility Toolkit

Enhancing Accessibility of Video Games for People with Vision Impairments

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. BSc Pirker Johanna

Institute of Interactive Systems and Data Science
Head: Univ.-Prof. Dipl.-Inf. Dr. Stefanie Lindstaedt

Graz, May 2021

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Abstract

Gaming with a vision impairment has already been researched in several studies, but most of the times by creating or modifying games that are not available outside the study. It's rarely discussed which games visually impaired players can really play. The root causes why so many games are not accessible to this demographic are also hardly ever researched.

This work is an attempt to shed a light on what and how people with vision impairments play video games and how more accessible games can be built. It shows that most games are made with cross-platform engines and compares them in terms of vision accessibility tools. The most popular engine, Unity, has almost no accessibility features built-in. So the work points out different areas where the vision accessibility of the Unity engine could be extended through reusable plugins. This research led to the design and implementation of such a reusable plugin in the form of the Unity Accessibility Toolkit. It is a first step to provide Unity developers with tools to create more visually accessible games by offering a navigation agent and a screen reader. Both are designed to integrate well with existing technologies, like Unity UI, to make them easy to use for the developer. The work also shows concrete examples of how these tools can be used to make menu navigation, environment observation, discrete and continuous navigation more accessible.

A first evaluation of the screen reader was also conducted. Developers familiar with Unity used the Unity Accessibility Toolkit to make a simple match-3 game accessible. Although none of the participants had experience with a screen reader prior to the evaluation, most of them performed well. Issues like insufficient labeling could have been avoided with a simple feedback loop. Such an extra feedback step should be a part in future studies. The overall response was also good. Participants showed interest in the topic and would use the toolkit to make their own projects accessible.

Contents

List of Figures

1 Introduction

Many of us use video games to socially interact with others and as a form of escapism from the difficulties of our daily lives (Calleja, 2010). Together with friends, family, and strangers, we strive through gorgeous digital worlds, drive around a race track or beat that next level in a puzzle game (AbleGamers, 2020). But in many of those experiences, we exclude people with impairments. (Porter, 2014). One group that is particularly hard hit are people with visual impairments, even though they make up a significant part of our society. According to a report from the World Health Organization (2010), there were an estimated 285 million people with visual impairments in 2010, of whom 39 million are blind.

This inaccessibility comes down to one of video games' most appealing features: visuals. Most video games rely heavily on visuals to present information to the player and often require a timely reaction to this information (Andrade et al., 2019). With limited or without any sight, this information can never reach the player. There have been several proposals in the past to translate this information in an auditory form, but they rarely reach the mainstream games (Atkinson et al., 2006). This problem sparked the idea for this master thesis.

1.1 Research Goals

This thesis' goals are to explore how people with visual impairments can play games, why most games aren't accessible to the visually impaired, and what can be done to change this. Therefore, our research goals are:

1. Analysing how people with visual impairments interact with computing devices and how they can play games

2. Researching the tools used to make video games and if any of these tools support building visually accessible games
3. Discussing what can be done to make vision accessibility features more common in video games

Besides exploring these questions in theory we will also present the implementation of a package for the Unity game engine that can help developers make games more accessible to visually impaired gamers. This packages' focus is both on making navigation in a game world more accessible and giving better access to menus and other user interface elements.

1.2 Structure

We will start in Chapter 2 by explaining the difference between impairment and disability and also discuss other terms commonly used in this thesis. After that, we will examine accessibility tools for blind and low vision users that can be used to interact with different computing devices. Next will be an overview of gaming with a visual impairment. This includes a discussion of both games with vision accessibility features and games without such features that are still playable by visually impaired gamers through some workarounds. We will then transition to game development by discussing the tools used to make games, game engines, and comparing them in terms of their vision accessibility features. Finally, we will pick out one specific game engine, Unity, explore existing extensions to make visually accessible games with this engine, and present other areas where extensions are possible.

In the third chapter, we will discuss the plugin itself. We will first talk about the design of the two main parts, the screen reader and the navigation agent. The screen reader can help navigate menus and make certain, simpler games accessible, while the navigation agent helps with navigating in a unconstraint game environment. While the design of both parts could be implemented for any game engine, we will present an implementation of them as a Unity package. Finally, we will show how to make use of the native screen reader of the iOS platform to increase acceptance by visually impaired users.

The fourth chapter is all about the evaluation of the plugin. We've invited developers to use our Unity package to make a simple demo project accessible. We, as domain experts, then evaluated how well the developers implemented the screen reader support in the sample project. We also surveyed the developers about their experience with the toolkit and accessibility tools in general. All the results are presented in this chapter.

In the fifth chapter, we will discuss the different lessons learned during the research of this thesis and the design, implementation, and evaluation of the package.

The thesis will come to a close in chapter six. Here, we will point at the most promising areas for future extensions of the Unity plugin.

2 Background and Related Work

Video games are highly complex pieces of software experienced with highly complex computing devices. People of all ages and all physical and cognitive abilities often struggle to interact with them. This interaction gets even more complicated when a person has an impairment of one of their senses, like their visual perception. So throughout this chapter, we explore how people with visual impairments can interact with computing devices, play accessible games and how developers can create them. We will start by discussing some definitions that will be used throughout this thesis.

2.1 Terminology

With the first part of this chapter, we want to clarify and give some context to different terms used throughout the thesis. This includes a differentiation between disability and impairment and a brief look at the classification of different visual impairments.

2.1.1 Impairment and Disability

When writing about accessibility measurements, both the terms "impairment" and "disability" are often used as if they describe the same group of people when in fact they do not. This is why we need to give a definition of both terms and make it more clear when it is appropriate to use which. The following was heavily influenced by Gerry Ellis's chapter on *"Impairment and Disability: Challenging Concepts of 'Normality'"* in the book *"Researching Audio Description: New Approaches"* (Ellis, 2016).

Impairment The International Classification of Functioning, Disability, and Health (ICF) was released in 2001 by the World Health Organization (WHO) with the goal to *"provide a unified and standard language and framework for the description of health and health-related states"* (World Health Organization, 2001). According to this document, impairments are defined as *"problems in body function or structure as a significant deviation or loss"* and that they *"represent a deviation from certain generally accepted population standards in the biomedical status of the body and its functions"* (World Health Organization, 2001). This refers not only to physical conditions but also mental ones. The ICF also states explicitly that these conditions can be temporary and can change over time.

Disability Article 1 of the United Nation's Convention on the Rights of Persons with Disabilities (CRPD) states that *"Persons with disabilities include those who have long-term physical, mental, intellectual or sensory impairments which in interaction with various barriers may hinder their full and effective participation in society on an equal basis with others."* (UN General Assembly, 2007). This means that a disability is caused by barriers in the environment.

These two definitions should make it clear that impairment and disability are two separate things. An impairment is a mental or physical condition that makes people with those conditions deviate from a generally accepted norm. A disability is caused by a barrier that makes something not accessible to a person, no matter of the mental or physical condition of this person. For video games, this means that if we do not offer the right tools and settings to make our games adjust to the needs of players, we disable them to play our games. These players could be players with impairments, but also older adults, kids, non-gamers, and any other person who would never describe themselves as disabled.

2.1.2 Impairments Concerning the Visual System

Since this thesis will concern itself with technical measurements for people with visual impairments, we need to give some context to this term. In the

International Classification of Diseases 11 (ICD-11), the WHO categorizes vision impairment in terms of visual acuity (World Health Organization, 2018). The same publication also lists other impairments concerning the visual field, contrast vision, light sensitivity, or color vision. In the following, we will describe the categorization of vision impairments from the ICD-11 and take a look at one specific vision impairment: color vision deficiency.

Vision Impairment Categorization The vision impairment categorization of the ICD-11 gives standardized measurements of how affected one's perception of visual details is. It classifies binocular vision impairment based on both a person's distance visual acuity and near visual acuity (World Health Organization, 2018). In both tests, one's vision is evaluated with both eyes open and by presenting a correction if needed.

The distance visual acuity is measured using Snellen chart, Figure 2.1, or a logMAR chart, Figure 2.2. In the following, we describe how a Snellen chart works.

A Snellen chart consists of multiple lines of letters in different sizes. Each line has a meter indication. A person with no visual impairment can read the letters on the 6 meters indication from a distance of 6 meters, letters on the 12 meters indication from a distance of 12 meters and so on (Hussain et al., 2006). This is why having no visual impairment is called 6/6 vision. For a mild visual impairment (worse than 6/12, equal or better than 6/18), this means that the person needs to be closer than 6 meters to read the letters on the 12 meters indication but can read letters on the 18 meters line from at least 6 meters distance. Moderate and serve visual impairment are often grouped under the term low vision (National Eye Institute, 2019).

Using such a Snellen chart, the distance visual acuity is defined by the ICD-11 as follows:

- No vision impairment: Equal or better than 6/12
- Mild vision impairment: worse than 6/12, equal or better than 6/18
- Moderate vision impairment: worse than 6/18 equal or better than 6/60
- Severe vision impairment: worse than 6/60 equal or better than 3/60
- Blindness: worse than 3/60 or no light perception

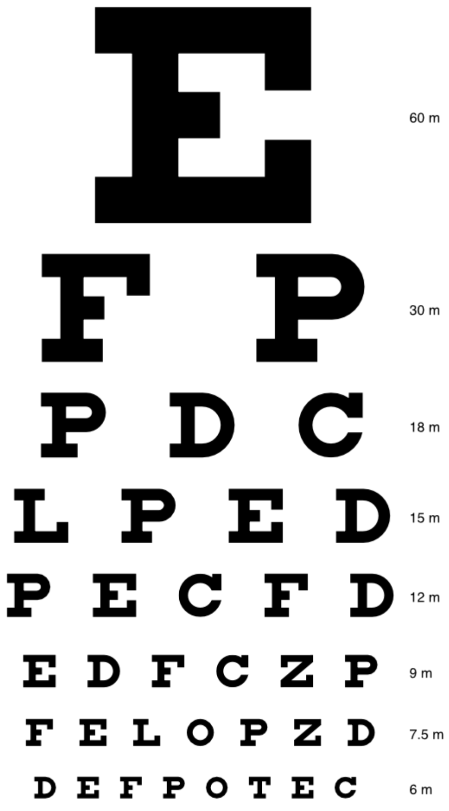


Figure 2.1: Snellen chart with metric measurements, based on a public domain drawing by Schneider (2002)

Near vision impairment is defined as having a near vision acuity worse than N6 with existing correction (World Health Organization, 2018). This value is based on the ability to read letters on a near vision logMAR chart at a standardized distance of 40cm (WHO Programme for the Prevention of Blindness and Deafness, 2003).

Color Vision Deficiency When thinking of visual impairments, most people think about some kind of impairment to the visual acuity, as described in the former section. Something that is far more common, affecting 8% of

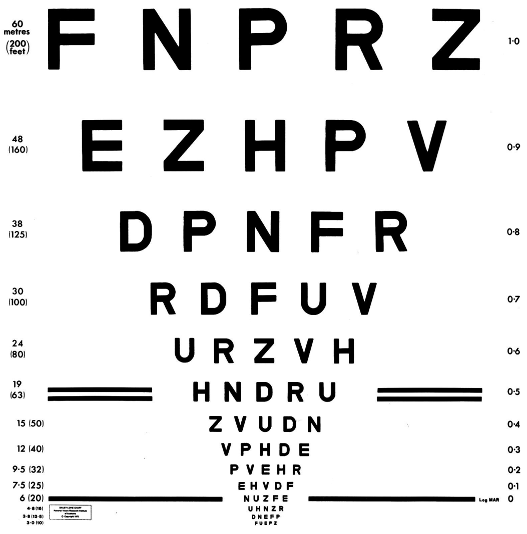


Figure 2.2: A logMAR chart with British Standard 5 x 4 letters for a standard testing distance of 6m (Bailey and Lovie-Kitchin, 2013).

males and 0.5% of females, is color vision deficiency, also known as color blindness vision (Simunovic, 2010). This impairment makes people perceive colors differently than people with perfect vision. Color blindness is most commonly caused by an anomaly or lack of working red-, green- or blue-sensitive cones in the eye. The most common forms of color blindness are Protanomaly/Protanopia (red cones), Deuteranomaly/Deuteranopia (green cones) and Tritanomaly/Tritanopia (blue cones). The ending -anomaly stands for an anomaly in the cones, where the -anopia stands for the total lack of working cones (Huang et al., 2007). Figure 2.3 shows how these three forms affect the vision of a person.

2.1.3 Usage of the Terms

In this thesis, we will mostly discuss measurements for visually impaired people in terms of visual acuity. When we talk about people with visual impairments we mean any visual impairment, from mild to blindness. Note that every tool that can help a blind person with a better perception of

[1]https://michelf.ca/projects/sim-daltonism/

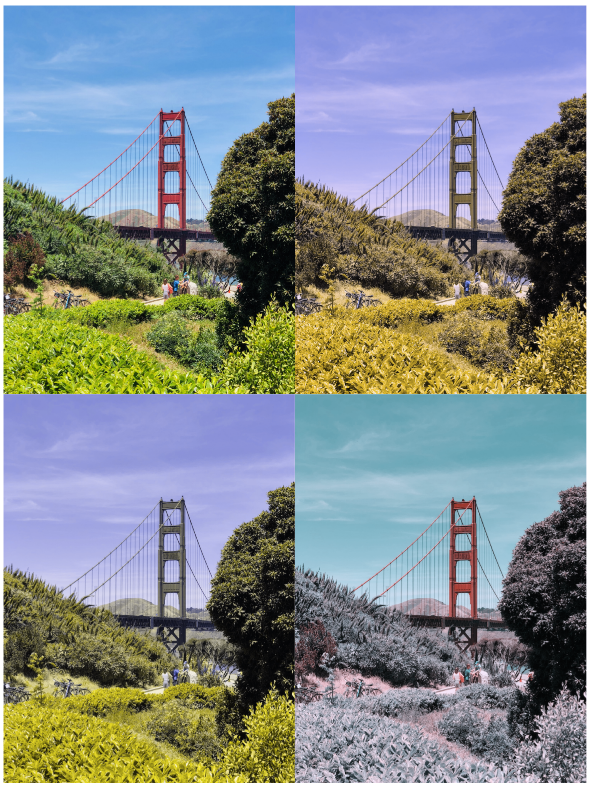


Figure 2.3: An image of the Golden Gate Bridge, as seen with different forms of color blindness. Top left: Original image, top right: Deuteranopia, bottom left: Protanopia, bottom left: Tritanopia. The images where altered using Sim Daltonism[1].

something on the screen will also help a person with any kind of visual impairment, even though it might not be the most efficient tool for that person. If any mentioned tool only concerns people with a certain visual impairment, like low vision, we will point that out.

We now know what a visual impairment is and how strongly affected one's vision is to be considered as visually impaired. What is still hard to grasp is how people with these kinds of impairments can interact with modern computing devices, which rely heavily on visual stimuli. We will discuss this in the following section.

2.2 Vision Accessibility of Computing Devices

The history of personal computing devices shows a static rise of importance for visuals and a decline of haptics. The first operating systems were text-only and ran on bulky devices. In subsequent years, the graphical user interface (GUI) emphasized visual representation and became a standard for operating systems. While we still use a GUI today, our devices became much smaller and replaced most haptic buttons with virtual ones. Virtual and augmented reality devices go even a step further. They use a head-mounted display and sometimes even take away any physical input by replacing it with hand gestures (Hiliges et al., 2017).

This trend brought immense challenges when it comes to making computing devices accessible to the visually impaired. While the accessibility in virtual and augmented reality devices is a topic of current research, all other previously mentioned devices can already be used by the visually impaired (Zhao et al., 2019). To understand how this is possible, we will briefly consider the interaction framework by Abowd and Beale (Dix et al., 2003).

2.2.1 The Interaction Framework

The interaction framework breaks down an interactive system into *system*, *user*, *input*, and *output*. The user has a goal and forms a task to achieve that goal. This task is articulated to the input and then translated for the system. The system transforms itself as a response and presents the results via the output. Finally, the user observes the result from the output. Through this observation, the user can adjust the task at hand or form a new task. This will lead to another round of the interactive cycle. A visual representation of this cycle can be seen in Figure 2.4.

2.2.2 Vision Accessibility of the Interaction Framework

The combination of input and output in the interaction framework is called the *interface*. For most modern computing devices, the interface is a GUI. The

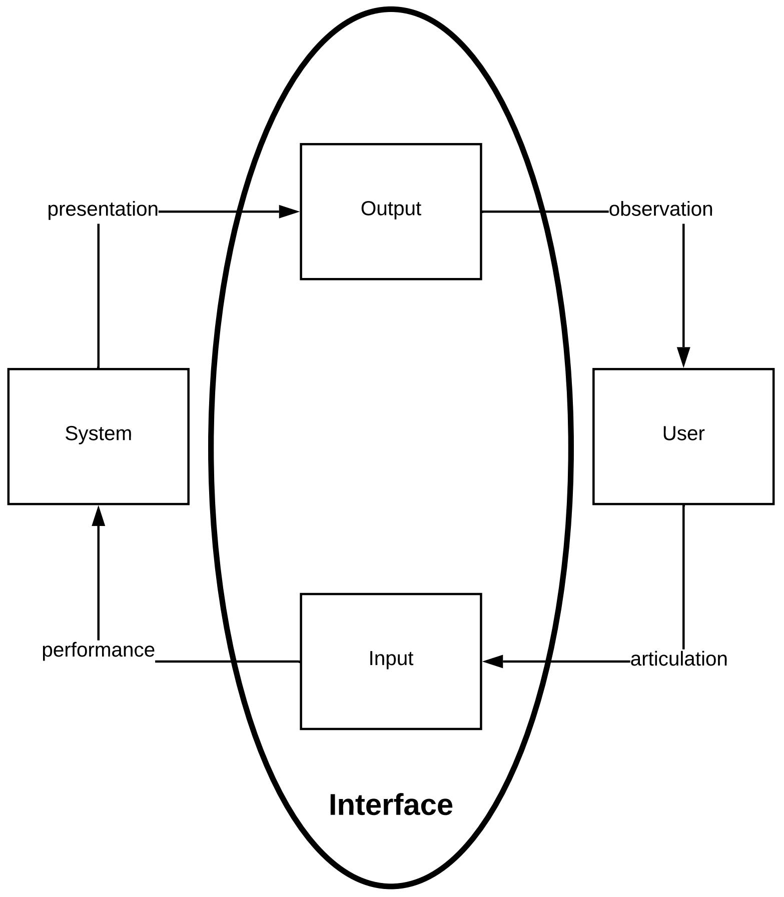


Figure 2.4: The interaction framework with transition between the components. Based on a drawing by Dix et al. (2003).

input for a GUI is most of the time articulated via a mouse, keyboard, or a touch screen, and the output is observed via a display. This interface will not work for most visually impaired persons without adjustments. Depending on the degree of visual impairment, the user will not be able to observe the output. This break in the interaction cycle does not only influence the observation of the output, but also the articulation of the task from the user to the input. How should a user articulate the task, if they cannot even know basic things like which elements are on the screen or where the mouse pointer is currently positioned? This question leaves us with three options to make a computing device accessible to a visually impaired user:

I Adjust or Extend the output of the GUI
II Add an additional layer between interface and the user
III Use an alternate interface

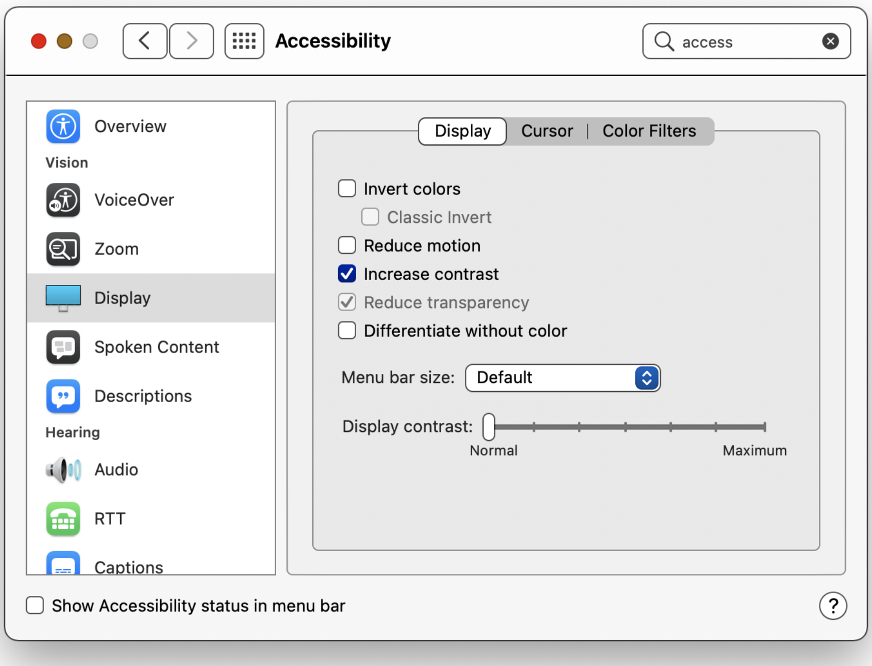
The following subsections will explore different available technologies that fall in either one of these categories.

(I) Adjust or Extend the Output of the GUI

Depending on the form of visual impairment, simple system settings can be enough to make the output perceivable to a visually impaired user. There are impairments for example which only affect the perception of contrast and not the visual acuity (Hyvärinen, Laurinen, and Romvmo, 1983). Modern operating systems like Windows or macOS have options to increase the contrast of text and icons or even allow changing the theme of the operating system to something legible to the visually impaired user. Other options let the user invert colors, adjust the mouse cursor size, text size, text weight, or reduce transparency. Figure 2.5 shows different display accessibility options that are available in macOS Big Sur.

The issue with having to rely on these system settings alone is that not all applications will either not support them at all or will not support them consistently with the rest of the system. Especially on desktops, many applications are built with non-native technologies like Java or Electron that either adjust differently or do not adjust at all to these settings. So people need other tools to make the GUI accessible.

Refreshable Braille Displays One such tool is a refreshable braille display. Braille is a tactile writing system invented in 1825 by Louis Braille (Oliveira et al., 2011). It uses a 3x2 matrix of dots to represent different letters, numbers, and punctuation characters. This system has been brought to computing devices via the refreshable braille displays (Schmidt et al., 2002). A refreshable braille display consists of multiple matrices of braille dots that can be extended and retracted to represent a text that would usually



- [ ] Invert colors
  - [ ] Classic Invert
- [ ] Reduce motion
- [x] Increase contrast
- [x] Reduce transparency
- [ ] Differentiate without color

Menu bar size: Default

Display contrast: Normal

- [ ] Show Accessibility status in menu bar

Figure 2.5: Display accessibility settings in macOS Big Sur

be shown on a screen. Therefore, it's an alternate output to the traditional screen that allows visually impaired users to get a text translated in a perceptible form. Some braille displays can even function as alternate input by including extra buttons that enable system navigation (HumanWare, 2019). One example of such a display can be seen in Figure 2.6.

Zoom/Magnifier A more trivial but highly effective approach of making the output legible to people with a mild form of a visual impairment is a virtual magnifier or a zoom option. These tools have the huge advantage that they work with every application across the whole operating system (Irvine et al., 2014). How such a tool looks like in use is shown in Figure 2.7. Besides only working for people with mild visual impairment, the big downside of such a system is that the user has to drag around the

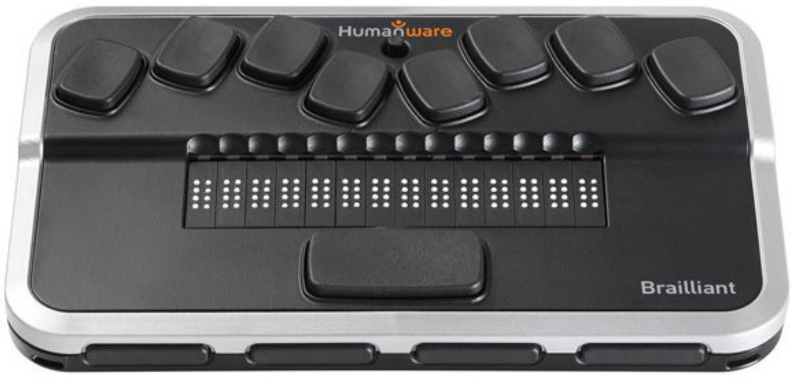
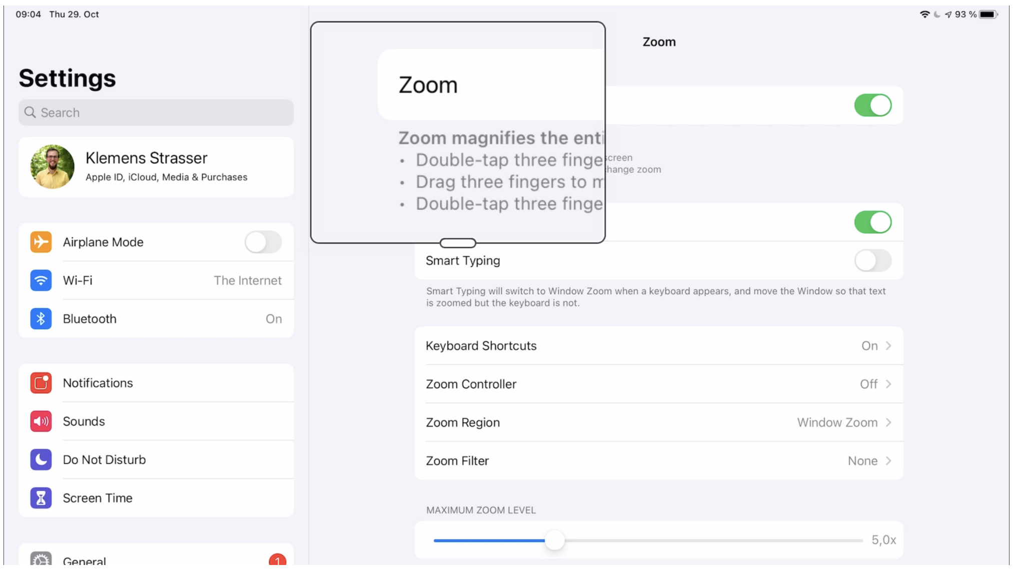


Figure 2.6: Stock image of the Brailliant BI 14 refreshable braille display by HumanWare (2019)

magnifier or zoom in and out all the time. This can make interacting with the computing device very tedious.

Speech Synthesizers Speech synthesizers are yet another technology that can be found in many modern operating systems. These systems take a written text as an input and create artificially spoken text from it (Taylor, 2009). This enables a visually impaired person to mark certain portions of the screen and let it read out loud to them, effectively replacing the visual output with a vocal one. This can be particularly useful when combined with Optical Character Recognition (OCR). OCR tools can convert an image of text back to text Dix et al. (2003). In the context of interacting with computing devices, one can mark any section of a screen with tools like EasyScreenOCR[2], get a textual representation, and then use a speech synthesizer to convert this text to spoken word.

[2]https://easyscreenocr.com


09:04 Thu 29. Oct

93 %

Settings

Search

Klemens Strasser
Apple ID, iCloud, Media & Purchases

Airplane Mode	Off
Wi-Fi	The Internet
Bluetooth	On
Notifications	
Sounds	
Do Not Disturb	
Screen Time	
General	1

Zoom

Zoom

Zoom magnifies the enti
- Double-tap three finge
- Drag three fingers to m
- Double-tap three finge

creen
hange zoom

Smart Typing: Off

Smart Typing will switch to Window Zoom when a keyboard appears, and move the Window so that text is zoomed but the keyboard is not.

Keyboard Shortcuts	On
Zoom Controller	Off
Zoom Region	Window Zoom
Zoom Filter	None

MAXIMUM ZOOM LEVEL
5,0x

Figure 2.7: Zoom accessibility settings on an iPad Pro under iOS 12.3

Additional Layer between Interface and the User (II)

The aforementioned tools can work for people with some kind of mild to mediate blindness. A fully blind person is not able to see anything on the screen and therefore, cannot make use of settings or magnifiers. They also cannot select any text which could then be represented via a braille display or spoken out with a speech synthesizer; at least not without an additional layer between the interface and the blind user. This layer comes in the form of a screen reader.

Screen Reader The first screen readers already existed before the GUI became popular, like the IBM Screen Reader for DOS (Thatcher, 1994). The capabilities of these systems evolved, but the basic idea stayed the same. A screen reader is an *additional interface*, and thus, an *additional input* and *additional output*, that sits between the user and the GUI. A visual representation can be seen in Figure 2.8. The screen reader helps the user with the articulation of a task to the input and makes the presentation of the output perceptible. This is done by fetching all interface elements on the screen and representing them in a serial manner (Borodin et al., 2010). The visually impaired user can then step through elements one by one, go in and out of groups of elements, or activate each element with keyboard shortcuts or swipe gestures (i.e. articulation to the sdditional input). To know which element is currently highlighted and what can be done with it, a screen reader uses the aforementioned speech synthesizer or braille display to convey this information (i.e. translation to the additional output). To make this concept clearer, Figure Figure 2.9 shows how a user interface is serialized by the VoiceOver[3] screen reader under iOS.

Every major operating system now includes a screen reader, like the Narrator[4] on Windows, VoiceOver on macOS and iOS and TalkBack[5] on Android.

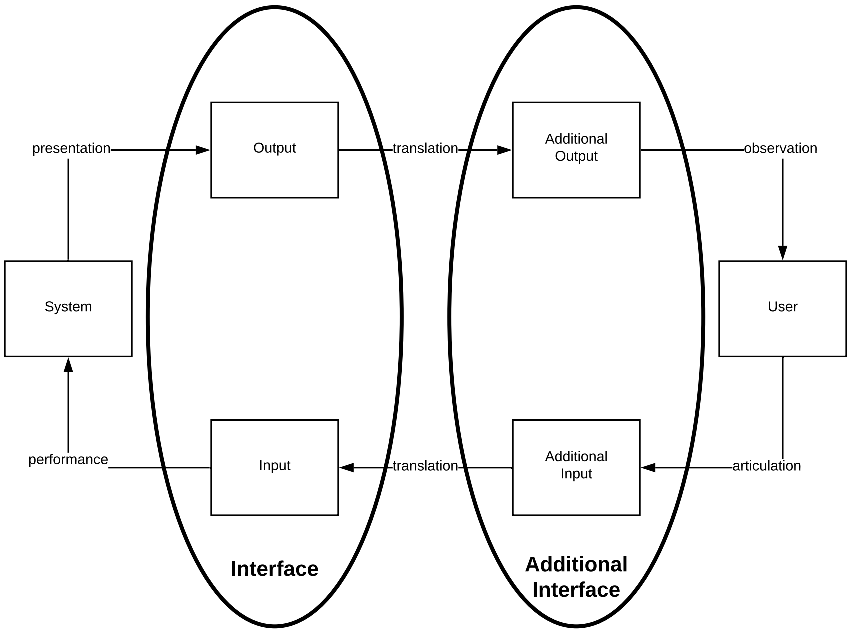


Figure 2.8: The interaction framework with transition between the components, extended with the additional interface layer, the scree reader. Based on a drawing by Dix et al. (2003).

[3]https://www.apple.com/accessibility/iphone/vision/

[4]https://support.microsoft.com/en-us/help/22798/windows-10-complete-guide-to-narrator

[5]https://support.google.com/accessibility/android/answer/6283677?hl=en

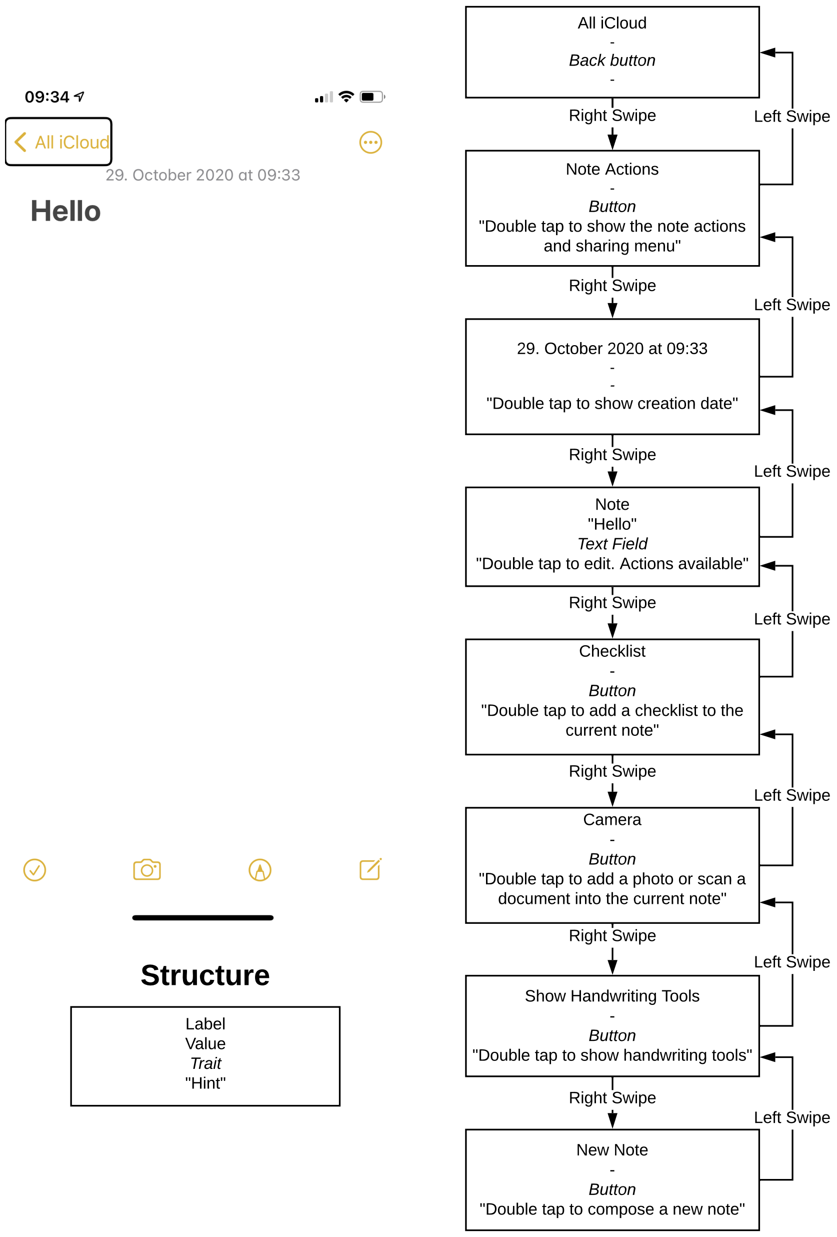

09:34
All iCloud
29. October 2020 at 09:33
Hello
Structure
Label
Value
Trait
"Hint"
All iCloud
-
Back button
-
Right Swipe
Left Swipe
Note Actions
-
Button
"Double tap to show the note actions and sharing menu"
Right Swipe
Left Swipe
29. October 2020 at 09:33
-
-
"Double tap to show creation date"
Right Swipe
Left Swipe
Note
"Hello"
Text Field
"Double tap to edit. Actions available"
Right Swipe
Left Swipe
Checklist
-
Button
"Double tap to add a checklist to the current note"
Right Swipe
Left Swipe
Camera
-
Button
"Double tap to add a photo or scan a document into the current note"
Right Swipe
Left Swipe
Show Handwriting Tools
-
Button
"Double tap to show handwriting tools"
Right Swipe
Left Swipe
New Note
-
Button
"Double tap to compose a new note"

Figure 2.9: Example of user interface serilization in VoiceOver under iOS 14.0.1

There are also third-party alternatives for specific systems like NVDA[6] or JAWS[7] (Vtyurina et al., 2019). While screen readers are a tremendous achievement for vision accessibility, all of them face a similar problem to the one discussed in 2.2.2. Depending on how the application is built, a screen reader might not be able to find all interactive elements of this application (Borodin et al., 2010). We will encounter this specific problem again when discussing the accessibility of video games.

Use an Alternate Interface (III)

The most prominent alternate interface comes in the form of Intelligent Personal Assistants (IPAs). An IPA is *"an application that uses input such as the user's voice, vision (images), and contextual information to provide assistance by answering questions in natural language, making recommendations"* (Baber, 1993). These applications are not only available on modern Smartphone operating systems like Siri[8] on iOS or Google Assistant[9] on Android, but also in homes within standalone devices like the Amazon Echo Dot[10] or the Nest Audio from Google[11] (Cowan et al., 2017). IPAs are often used in order to save time or when the users' hands are occupied (Luger and Sellen, 2016). Common tasks are responding to messages, creating reminders, or starting a navigation.

Many of the standalone IPA devices do not feature a screen and therefore lack any kind of GUI. In fact, studies suggest that it is perceived negatively by the users if a GUI is available and the IPA shows a graphical element that is not reflected in spoken text (Cowan et al., 2017). Not having a mandatory graphical representation makes these devices accessible to visually impaired users out of the box and thus, are perfectly suited as an alternate interface (Pradhan, Mehta, and Findlater, 2018). To see how well suited this interface is for people with visual impairments, Abdolrahmani, Kuber, and Branham

[6]https://www.nvaccess.org
[7]https://www.freedomscientific.com/products/software/jaws/
[8]https://www.apple.com/siri/
[9]https://assistant.google.com
[10]https://www.amazon.com/Echo-Dot/dp/B07FZ8S74R
[11]https://store.google.com/product/nest_audio

(2018) interviewed fourteen legally blind people about their usage of IPAs. The study shows that time-saving and hands-free interactions are again important factors. IPAs can for example ease or enable multitasking and interaction with smart home devices. But the participants also expressed frustrations over being misinterpreted by IPAs and having to adjust how a question is formulated for it to be understood. The study also pointed out that some standalone devices still use visual cues for indicating things like a mutated microphone or a notification that have no audio representation. Even though these issues exist, the *"Accessibility by Accident"* study shows that the adaption of IPAs visually impaired users is already high and will potentially grow as these interfaces become richer (Pradhan, Mehta, and Findlater, 2018).

2.2.3 Practical Usage of the Accessibility Technologies

What should be clear from the descriptions of all the technologies is that there is no single solution to solve interaction with computing devices for visually impaired people. All the different technologies solve different problems. IPAs are great for small, quick tasks like starting to play music or turning on the lights, but they often fail to work for more complex tasks (Abdolrahmani, Kuber, and Branham, 2018). Magnifiers or speech synthesizers with OCR technology can make any screen element more accessible, but they make interacting with the device tedious and will not work for every kind of visual impairment. And a screen reader can help with navigation in an application, but it might fail to find all interactive elements, depending on how the application is built (Borodin et al., 2010).

So the real power of all these technologies is their combined usage and shared availability and support on a system. Having access to as many technologies as possible can make it possible to interact with many complex applications. One important group of highly complex applications are video games. So the question is - can these discussed technologies be used to interact with video games?

2.3 Gaming with Vision Impairment

The assistive technologies we've discussed in the last part can help a visually impaired user to navigate and interact with a computing device. To determine if the same is true when it comes to video gaming, we need to look at something we call the *Gaming Workflow*.

2.3.1 The Gaming Workflow

We define the *Gaming Workflow* as a set of steps a user must take to play any new game. Figure 2.10 gives a visual representation of these steps and the actions needed to move between them. We can separate the steps into two phases, *Pre-Game* and *In-Game*. Pre-Game includes the process from wanting to play a new game to having everything set up to actually start the game. For a single platform, the Pre-Game experience is mostly identical for every game because the experience lies in the hands of the platform maker or the creator of a software distributing platform. For the In-Game experience, which is everything that happens within one single game, the experience can differ tremendously between games. The In-Game phase is mostly determined by the game developer.

While the Gameplay step is definitely the most interesting part of this workflow when discussing how people play games, we will need to take a quick look at everything else first. The Gameplay will be further discussed in Section 2.3.4.

2.3.2 Vision Accessibility of the Pre-Game Phase

Video games are enjoyed on different platforms. The most common devices used to play games are according to the *"2020 Essential Facts About the Video Game Industry"* report mobile devices (smartphones), game consoles, and personal computers (Entertainment Software Association, 2020). We will take a look at the Pre-Game experience for each of them.

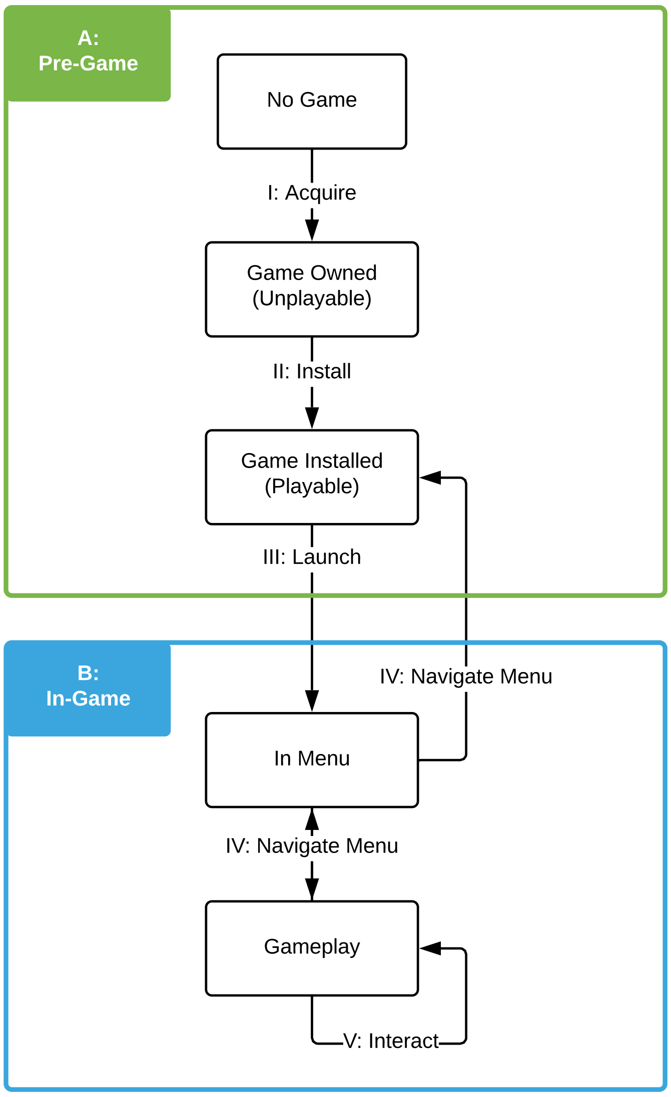


Figure 2.10: The Gaming Workflow: A visualisation of the steps a user has to go through when wanting to play any new game.

Mobile devices We have previously mentioned in Section 2.2.2 that the two common operating systems, Android and iOS, come with built-in screen readers. Those screen readers make the operating system accessible to visually impaired users, including everything from interacting with built-in applications to launching installed apps (Apple Inc., 2020c). Because games and applications can both be installed and downloaded through built-in applications, via the App Store[12] on iOS and the Google Play Store[13] on Android, having these powerful screen readers already makes the whole Pre-Game part of the Gaming Workflow accessible to visually impaired users. To make discovery even easier, the iOS App Store features a specific article about accessible apps and games, as can be seen in Figure 2.11.

Game Consoles The accessibility of acquiring, installing, and launching games on consoles varies between platforms. As of April 2021, the most recent consoles of the three major console vendors, Sony, Nintendo, and Microsoft, are the Playstation 5 (PS5), the Nintendo Switch (Switch), and Xbox Series S/X (Xbox Series). Games on these systems are bought either on physical mediums or, with growing popularity, digitally through built-in digital stores (Dring, 2019).

For the PS5 and the Xbox Series, these stores and the rest of the system can be adjusted to be more visually accessible. This includes a high contrast mode and a magnifier on the Xbox Series[14], bolder and larger text on the PS5[15]. Both also feature a screen reader, a feature that the Switch is lacking. As of this writing, the only vision accessibility feature provided by the Switch is a Magnifier[16], making it the least visually accessible console of the three.

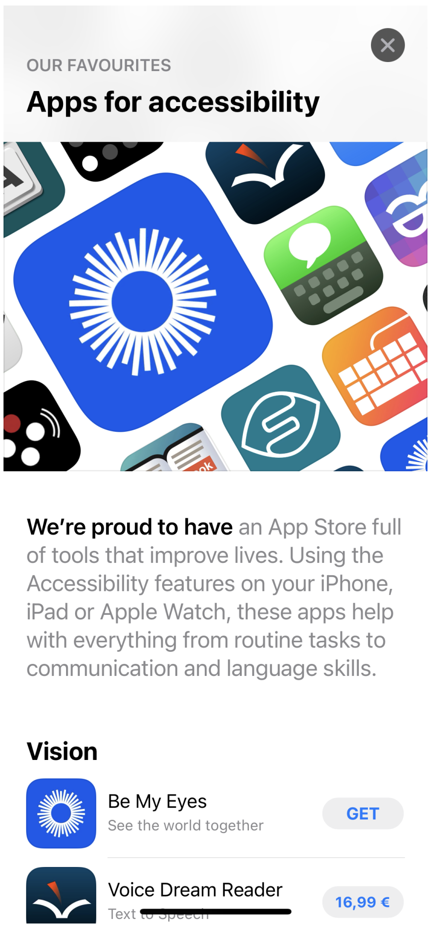
[12]https://www.apple.com/app-store/

[13]https://play.google.com/store

[14]https://support.xbox.com/en-US/help/account-profile/accessibility/ease-of-access-settings

[15]https://www.playstation.com/en-us/support/hardware/ps5-accessibility-settings/

[16]https://en-americas-support.nintendo.com/app/answers/detail/a_id/44641/ /how-to-use-zoom


OUR FAVOURITES

Apps for accessibility

We're proud to have an App Store full of tools that improve lives. Using the Accessibility features on your iPhone, iPad or Apple Watch, these apps help with everything from routine tasks to communication and language skills.

Vision

Be My Eyes
See the world together
GET

Voice Dream Reader
Text to Speech
16,99 €

Figure 2.11: List of selected accessibility apps in the App Store (Apple Inc., 2020d)

Personal Computers The openness of the computer platform makes it possible to download games from multiple different sources, so the accessibility of the Pre-Game phase highly depends on where the game is available. Looking at all different ways would go beyond the scope of this work. So for this analysis, we've chosen the leading digital distributor, Steam[17], and one of its competitors, the Epic Games Launcher and Store[18].

Steam and the Epic Game Launcher roughly work in the same way. Both

[17]https://store.steampowered.com
[18]https://www.epicgames.com/store/en-US/

have desktop applications that are used to acquire games through a built-in store, install games from the user's library, or launch them, also through the library. Sadly, both desktop applications are not accessible through a screen reader (Andrade et al., 2019; IllegallySighted, 2017). Therefore, low vision users have to rely on other tools like magnifiers to interact with those applications. Fully blind users are not able to download or launch any games through the clients without external help or through the laborious process of using a speech synthesizer with an OCR tool to learn the layout of the client.

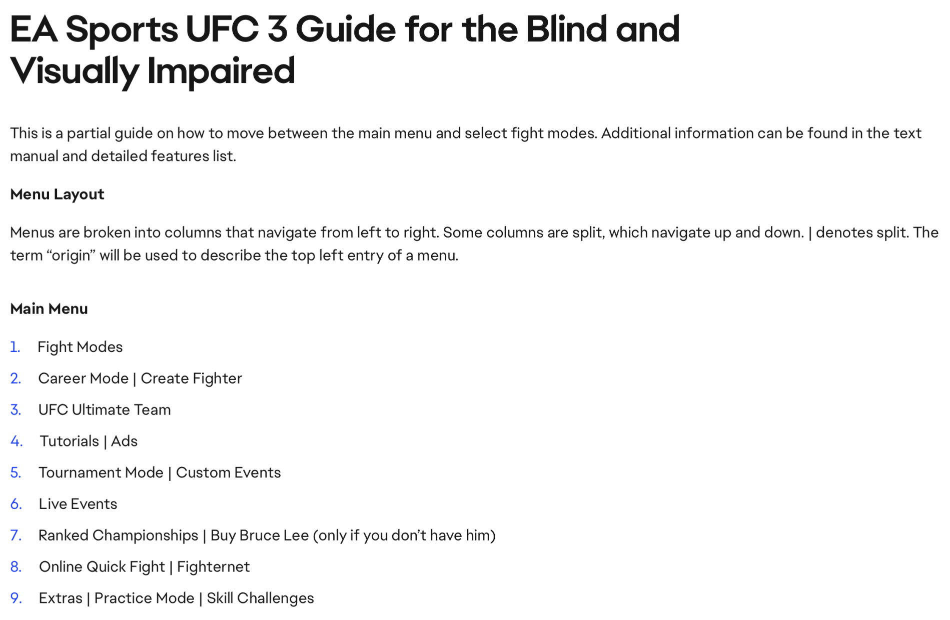
2.3.3 Vision Accessibility of the In-Menu Step

When the actual game is launched, the next hurdle is game menu navigation (Game Accessibility Guidelines, 2019a). No matter if desktop, mobile, or console game, modern games for all platforms are often built with cross-platform game engines like Unity[19] or the Unreal Engine[20]. Most of these engines render the user interface in a way that its elements are not exposed to a screen reader. Thus, menu navigation is not possible with a screen reader in these games (Hamilton, 2013; Andrade et al., 2019). The aforementioned system wide zooming options and OCR tools, if available on the respective platform, can help people with mild visual impairment to overcome this hurdle. Some games still found a way of making the menu accessible by imitating screen reader like behavior. Here, every menu element is communicated to the user through spoken word (RARECSM, 2019). These settings are not always turned on by default though, so if they aren't, visually impaired users need to find a way of enabling them.

One such way could be a written menu guide (Stevens, 2018). A written menu guide is a text representation of all menus and settings in a game with a description of how to reach each of them. Guides like these are mostly done by the community, but Electronic Arts (EA) for example has created official ones for games like UFC 3, as can be seen in Figure 2.12 (Electronic Arts Inc., 2019). For these guides to work, the game menu has to be deterministic (Stevens, 2018). While a user with perfect vision can see

[19]https://unity.com
[20]https://www.unrealengine.com/

changes in the menu layout immediately, a visually impaired user can get lost if the menu guide suddenly does not correlate with the current layout anymore. It is also very helpful to get audio cues for moving the focus in the menu, for selecting something and when the menu wraps, so the user knows if an input changed something on the screen (Stevens, 2018).


EA Sports UFC 3 Guide for the Blind and Visually Impaired

This is a partial guide on how to move between the main menu and select fight modes. Additional information can be found in the text manual and detailed features list.

Menu Layout

Menus are broken into columns that navigate from left to right. Some columns are split, which navigate up and down. | denotes split. The term "origin" will be used to describe the top left entry of a menu.

Main Menu

1. Fight Modes
2. Career Mode | Create Fighter
3. UFC Ultimate Team
4. Tutorials | Ads
5. Tournament Mode | Custom Events
6. Live Events
7. Ranked Championships | Buy Bruce Lee (only if you don't have him)
8. Online Quick Fight | Fighternet
9. Extras | Practice Mode | Skill Challenges

Figure 2.12: An excerpt from the official menu guide for UFC 3 (Electronic Arts Inc., 2019)

If no screen reader support, no screen reader imitation and no guide exist for a certain game, visually impaired, and especially blind gamers, have to rely on their sighted friends to assist them with the navigation (Icel, 2017).

2.3.4 Vision Accessibility of the Gameplay Step

We have now established a knowledge of how a visually impaired user can go from wanting to play a game to the actual point of playing the game. While researching games that can be played by visually impaired gamers, we found four different categories of games:

I Games that are not accessible at all

II Games that were not intended to be accessible, but the visually impaired community found ways to play them nonetheless. Thus, these games can be called Unintentionally Accessible Experiences.

III Games that communicate the content first and foremost via audio instead of visuals. These games are called Audio Games

IV Games that heavily rely on visuals, but have either special accessibility settings or support tools like screen readers to make them accessible to visually impaired players. We call them Intentionally Accessible Mainstream Experiences

Besides those categories, there are also games that have been specifically built for scientific studies but were never made available outside the studies. Examples are Blind Hero (Yuan and, 2008) and Rock Vibe (Allman et al., 2009), imitations of popular music games Guitar Hero (Harmonix, 2007a) and Rock Band (Harmonix, 2007b), which require special hardware that was only built for the study. It is definitely valuable to think about the usage of special hardware to make games more accessible, but this lies beyond the scope of this thesis. We want to understand how people with visual impairments play games that are already available to them. So to get this understanding, we will now look into such video games based on our categorization.

II: Unintentionally Accessible Experiences

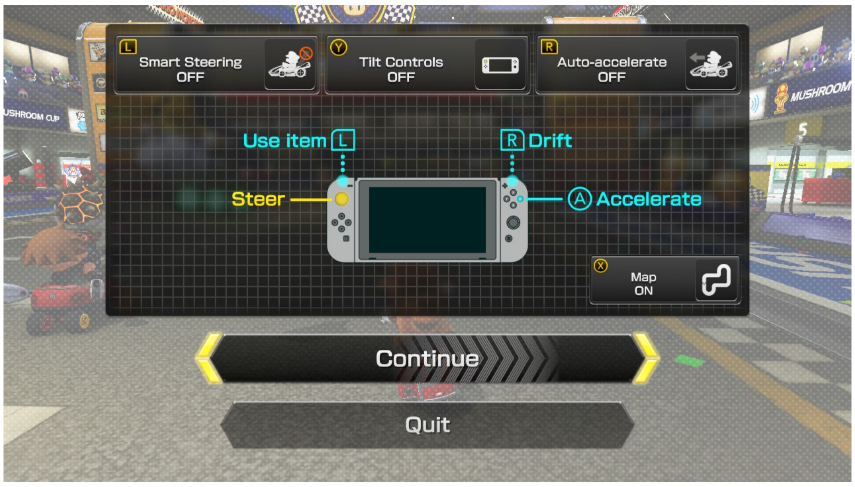
The category of unintentionally accessible experiences includes all games where the creators didn't intentionally offer or support any tools to make them accessible for people with visual impairments, but the community has still found ways to play them. Most of the time, this is possible because the game had already a realistic or carefully crafted sound design, something like an assist mode for less skilled players, or have a limited control scheme (Stevens, 2018). A prime example of games with a realistic sound design are sports games, like Madden NFL (EA Tiburon, 2020).

Madden NFL Madden NFL is an American football video game series (EA Tiburon, 2020). Like many other sports video games, the developers decided to include a sports commentary to make the experience feel like an interactive sports broadcast. The commentary includes all kinds of useful information, from things like the current score or yard line to detailed information about the whereabouts of the players team members and the members of the opponent team. This fact alone can give visually impaired users all the information they need to interact with the game. In addition, the goal of the game in its default settings is basically to move upwards (Stevens, 2018). This movement is simplified once more through an artificial intelligence that automatically steers the player until they take action themselves. Therefore, the amount of input a user has to make to be successful is limited.

Mortal Kombat Similar to Madden NFL, the fighting game series Mortal Kombat also features a limited directional movement (NetherRealm Studios, 2019). But this fact alone doesn't make it accessible. It is a highly detailed sound design that makes this game, along with many other fighting games, accessible to visually impaired players (Andrade et al., 2019). How this is possible is demonstrated in the HBO documentary *"This Is How To Play Video Games If You're Totally Blind"* (Vice News, 2019). This documentary also shows that blind players often need assistance when playing fighting games for the first time. The game features a vast variety of different moves, so when training modes lack a voice over, the visually impaired player cannot correlate the auditory feedback with the impact in the game when initially playing it.

Mario Kart 8 In contrast to fighting games, racing games like Mario Kart 8 Deluxe feature a smaller amount of possible inputs and auditory feedback and are therefore easier to learn without assistance (Nintendo, 2017). Information like the motor becoming louder when the vehicle speed increases or tires squealing when the vehicle takes a turn can be used to paint a mental picture of the racing course by following and listening to AI players (Stevens, 2018). This approach works for every racing game with a detailed sound design and AI players. We've chosen Mario Kart 8

Deluxe for this discussion because of its assist options, that can be seen in Figure 2.13 (Game Accessibility Guidelines, 2019b). The Smart Steering and Auto-Accelerations options can help a player to easier traverse the course and learn its layout quicker. But having these options enabled also adds a small disadvantage, disabling people to reach certain shortcuts or use a special boost. Nevertheless, having a disadvantageous gaming experience is still better than having no gaming experience at all (Stevens, 2018).


Smart Steering OFF

Tilt Controls OFF

Auto-accelerate OFF

Map ON

Figure 2.13: The assist options of Mario Kart 8 Deluxe (Nintendo, 2017)

All of those unintentionally accessible games are accessible because they have good sound design besides their visuals. Good sound design is also key to the next category of accessible games.

III: Audio Games

All information the user needs to play an audio game is provided via audio (Preece, 2013). It is possible that these games feature some visuals, but *"it is always the sounds that are central to the game"* (Friberg and Gärdenfors, 2004). Tutorials, story bits, and other texts are typically read by a voice actor or are converted to audio via Text-To-Speech technology (Araújo et al., 2017). Game elements are usually conveyed via two types of sounds: Auditory icons and earcons (Andrade et al., 2019). Auditory icons are natural sounds and are used if the game element has a natural equivalent (Gaver, 1986). An example would be the sounds of someone walking on grass or dirt to convey the current terrain the player is walking on. Earcons, on the other hand, are artificial sounds (Blattner, Sumikawa, and Greenberg, 1989). The player has to learn these sounds for every game separately because there is no standard set of earcons (White, Fitzpatrick, and McAllister, 2008). The Blindfold games by ObjectiveEd[21] show how these earcons can effectively be used to turn well-known games into audio games.

Blindfold The range of games turned into accessible audio games under the Blindfold label reach from well-known video games like Bejeweled (PopCap Games, 2001) to physical games like Sudoku or Pinball. All Blindfold games that we've tested can roughly be grouped into two kinds: Games with discrete and games with continuous changes. Blindfold Pinball is an example for the latter category (Schultz, 2016). Here, the player releases a ball into the playing field, which then moves continuously towards the bottom. While the ball moves around the playing field, different obstacles can be hit to get points. Upon hitting, each obstacle emits an earcon, so that the obstacle can be located through stereo sound. The current location of that ball is communicated by a sound that gets louder when the ball is getting closer to the bottom. In addition to this, a countdown will start when the ball is getting very close to the bottom. The player can then trigger the two flippers at the bottom of the screen via tapping to prevent the ball from falling into a hole. The game ends when the ball has fallen into this hole. So to actually succeed in the game, the player has to constantly listen to the different earcons and react timely to them.

This gameplay substantially differs from the games with discrete changes. In games with discrete changes, the elements stay the same until the user performs an action. After the action, the game updates and becomes static again until the next action is performed. For example, in Blindfold Sudoku, the user has to fill out numbers in a 9x9 playing field. The field only changes when the user enters a new number into a cell and stays static otherwise (Schultz, 2014). The user moves between those cells of this field via swiping. Every time a highlighted element changes, its location and value are spoken

[21]https://blindfoldgames.org

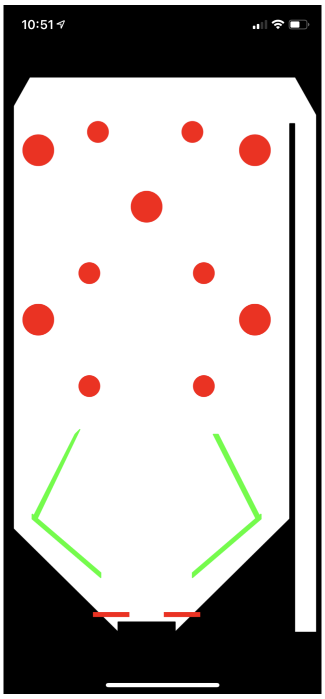
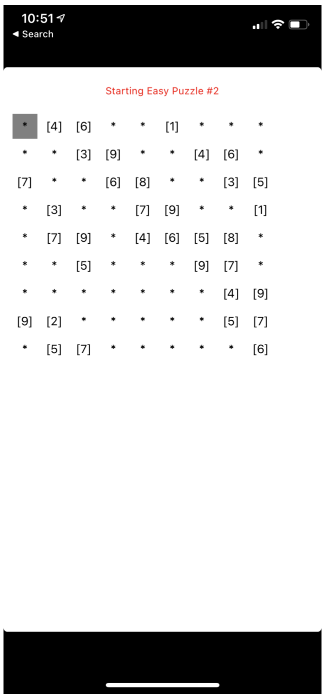


Figure 2.14: Blindfold Pinball (Schultz, 2016)



Starting Easy Puzzle #2

```
*    [4]  [6]  *    *    [1]  *    *    *
*    *    [3]  [9]  *    *    [4]  [6]  *
[7]  *    *    [6]  [8]  *    *    [3]  [5]
*    [3]  *    *    [7]  [9]  *    *    [1]
*    [7]  [9]  *    [4]  [6]  [5]  [8]  *
*    *    [5]  *    *    *    [9]  [7]  *
*    *    *    *    *    *    *    [4]  [9]
[9]  [2]  *    *    *    *    *    [5]  [7]
*    [5]  [7]  *    *    *    *    *    [6]
```

Figure 2.15: Blindfold Sudoku (Schultz, 2014)

out, which is very reminiscent of screen reading technologies. With this information, players can paint a mental picture of the scene in their heads and react to it in their own time.

We could go through numerous other Blindfold games and explain how they use earcons to turn different kinds of games into audio games. But since they all follow roughly the same approach as the two games mentioned above, we will shift our focus to a game that was built from the ground up as an audio game.

A Blind Legend A Blind Legend is an action-adventure game by the French developer DOWiNO released for multiple platforms (DOWINO, 2016). We have used the iOS version[22] for the following description. In the game, the

[22]https://itunes.apple.com/fr/app/a-blind-legend/id973483154

player takes on the role of Edward, a blind swordsman on a quest to save his wife. The experience is possible through auditory icons, earcons, and audio recordings from voice actors, all of which are conveyed to the user via binaural sound. Binaural recordings can be created with an artificial human head and two high-fidelity microphones embedded into the dummy head's ears (Andrade et al., 2019). Using this method, the *"microphones record the audio stream as it is heard and processed by the human"* (Kovács et al., 2015). To perceive this audio recording correctly in the game, the player has to wear stereo headphones. Even a player with perfect sight cannot play the game without stereo headphones, since there is also no visual representation of the gameplay, as can be seen in Figure 2.16.

Figure 2.16: Screenshot of A Blind Legend, showing that the game doesn't have any visual representation of the gameplay (DOWINO, 2016)

The gameplay is split into two parts, namely movement and fighting. Movement happens in a 3D environment with the help of the protagonist's daughter, Louise. Tapping the screen will make Louise speak up. Since her voice is conveyed via binaural sound, the user can perceive her location through audio and move towards her. To move, the user swipes the screen to turn left, right forward, or backward. If the player has reached Louise, she will go to the next point and the same procedure repeats until a final location is reached. The second gameplay part consists of fighting against various enemies. The position of the enemies is again conveyed through binaural sound. Auditory icons are used to warn the player when the enemy strikes. As soon as the icon is noticed, the player has to swipe in the direction of the sound to strike back. Attacks can also be blocked by pinching on the screen, so the player can listen and get a feeling for the attack pattern of the enemy. Edwards current health is represented by another auditory icon, a heartbeat, that increases in speed when it is lowered.

The games we've just described can be experienced by all players independent from how good their sight is. But there are indeed examples of audio games that are not necessarily accessible (Araújo et al., 2017). Zombies, Run! for example requires the player to physically run in the real world, which not every gamer can do (Six to Start, 2012). But inaccessible audio games are rare. The bigger problem with audio games is their acceptance in the general gaming community. Audio games often have a lesser appeal to sighted players than they do to visually impaired (Andrade et al., 2019). This is made worse by the fact that audio games are often overly simplistic, probably because developers underestimate the skills of their audience (Andrade et al., 2019; Arielle M. Silverman and Van Boven, 2014). So players with full sight will rather play more complex and visually appealing games; and visually impaired players would want to play the same games as their sighted friends (Smith and Nayar, 2018). This shows that audio games are not a solution to include visually impaired gamers in the general gaming community. So it's the task of all game developers to create visual experiences that are accessible to all. This brings us to our next category of games.

IV: Intentionally Accessible Mainstream Experiences

Intentionally accessible mainstream experiences are games that are high on visuals but still built with vision accessibility in mind. Sadly, we could not find many games during our research that fit this category. Most games we've found are already very limited in interaction and scope, which enabled the developers to make them playable through a screen reader. This include virtual board games like Shredder Chess (Skizzix, 2009) and word games like Subwords (Strasser, 2018).

The group of complex visually accessible games is very sparse. We think one reason is that complex games often fail to translate highly complex visuals into audio. Studies have shown that navigation in a virtual environment through audio cues only is theoretically possible, but it is less accurate than audio-visual navigation (Lokki and Grohn, 2005). This lack of accuracy occurs because developers do not want to overwhelm the player with too much audio signals (Yuan, Folmer, and Harris, 2011; Kruger and Zijl, 2014).

While we were already deep into the writing of this thesis, a new game was released that is highly complex, has options to enhance visuals for low vision users, and found a way around the problem of translating information into an auditory form. This game is The Last of Us Part II (Naughty Dog, 2020).

The Last of Us Part II The Last of Us Part II by Naughty Dog has become the proclaimed *"most accessible game ever"* (Molloy and Carter, 2020). What makes a game accessible to a certain person is highly subjective, so this statement should be taken with a grain of salt, but The Last of Us Part II is still a remarkable game in this category. It can be called a mainstream video game, having sold more than 4 million copies within the first few days after its release, and still has more accessibility options than any other game we've found during our research (Sony Interactive Entertainment Europe, 2020; Sony Interactive Entertainment LLC, 2020).

In this 3D action-adventure game, the user controls a character through a post-apocalyptical world. Throughout the game, the user has to traverse a 3D environment, solve puzzles, and fight various enemies. All of these interactions are highly complex and usually would be hardly playable for users with visual impairments, if playable at all. But Naughty Dog has included a vast range of tools and settings to improve vision accessibility. One set of tools is about magnification and visual aids. Users can scale up elements heads-up display (HUD), change its color and appearance. There is even a special colorblind mode for the HUD. Besides that, a screen magnifier can be used to scale up all the content of the game upon usage. We've discussed similar tools in the context of computing devices in section 2.2.2. A more advanced tool is the *High Contrast Display* option of the game. The setting mutes environment colors, highlights edges of the obstacles in the game and tints interactive elements in distinguishable colors. Figure 2.17 shows the high contrast display option applied to the game.

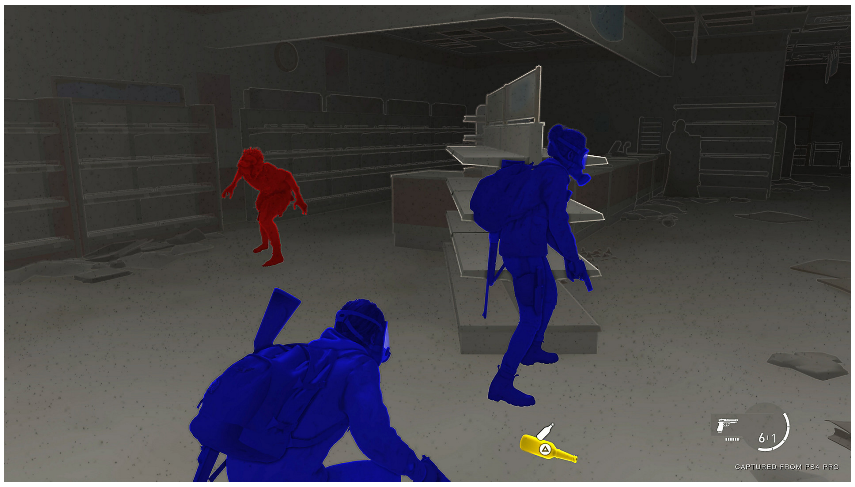


Figure 2.17: The Last Of Us Part II with vision accessibility presets applied (Sony Interactive Entertainment LLC, 2020; Naughty Dog, 2020)

These visual aids are helpful for low-vision users, but cannot assist blind players to play the game. So for blind players, Naughty Dog included the *Enhanced Listen Mode* and audio cues (Saylor, 2020). With the Enhanced Listen Mode enabled, users can scan the world and get both visual feedback and audio cues for different objects, like enemies or items. In the scientific literature, features like this are often referred to as a Sonar, because of its similarities to the underwater navigation technique of the same name (White, Fitzpatrick, and McAllister, 2008). Users of The Last Of Us Part II can use this sonar to either manually change between scanning and moving around to find a certain location or use a second feature, called *Navigation Assistance*, to automatically move towards the last scanned item. The game also has options to skip parts that are not easily accessible, like difficult jumps or puzzles. All of this makes the game fully experienceable for blind players.

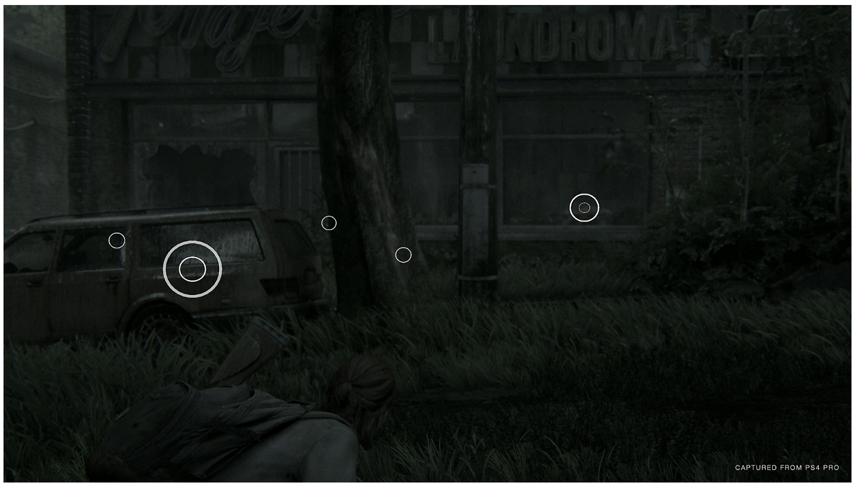


Figure 2.18: The Last Of Us Part II with enhanced listen mode feature applied (Sony Interactive Entertainment LLC, 2020; Naughty Dog, 2020)

As we've already mentioned, building in accessibility features like the ones just discussed is a highly complex task and requires many resources. The Last Of Us II was the follow-up to a highly successful game and funded by a multinational company, Sony, and thus, had access to those resources. But what about other game developers? Do any of the standard tools they use to build games include special options to make accessible games? And what tools do they even use? This will be the topic of the next section.

2.4 Game Engines and Vision Accessibility

In the previous section, we've looked into how visually impaired people can play some video games, but far from all games. To get a better understanding of why many games do not work with tools like screen readers and if we can improve this situation, we have to take a look at how most video games are built nowadays: using game engines.

2.4.1 Definition of Game Engines

Gregory (2014) describes game engines as *"software that is extensible and can be used as the foundation for many different games without major modification"*. Another definition is given by the creators of the Unity engine, who describe a game engine as *"the software that provides game creators with the necessary set of features to build games quickly and efficiently"* (Unity Technologies, 2019b).

While the exact set of features differ between game engines, some common parts are a rendering engine, a physics system, an audio system, tools to build a graphical user interface, and an artificial intelligence system (Gregory, 2014; Unity Technologies, 2019b). Having all those tools in place instead of building them from scratch for every game makes game development drastically faster and more approachable (Christopoulou and Xinogalos, 2017). Modern game engines even come with powerful graphical editors that enable people with little to no coding skills to develop their own games (Unity Technologies, 2019c).

As already mentioned, each game engine has a different set of features. There are publications, like *"Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices"* by Christopoulou and Xinogalos (2017), which compare those features across engines. But we could not find any work that includes a comparison in terms of vision accessibility features. So that's what we are going to do in the following.

2.4.2 Vision Accessibility Features of Different Game Engines

For our comparison, we will take a look at the vision accessibility features of five different game engines: Unity[23], Unreal Engine[24], Godot[25], SpriteKit[26] and SceneKit[27]. We have selected the first two engines because of their

[23]https://unity.com
[24]https://www.unrealengine.com/
[25]https://godotengine.org/
[26]https://developer.apple.com/spritekit/
[27]https://developer.ape.com/scenekit/

popularity, Godot because of its open-source nature and SpriteKit/SceneKit because its developer, Apple, is known for their strong focus on accessibility. We also left out game engines like eAdventure because they limit themselves to building experiences for visually impaired players only (Torrente et al., 2014).

Note that when we talk about vision accessibility features, we only talk about the features and tools offered by the game engine to make games suited for the visually impaired without having to add any additional software or plugins. We also do not look into how accessible the tools for building games are for people with vision impairments because this goes beyond the scope of this thesis.

Unity

The Unity engine was first introduced in 2005 as a game engine for Mac OS X (Higgins, 2010). Since then, it rose from a niche product Mac games to one of the biggest game engines in the industry, supporting 18 different platforms and powering countless games (Unity Technologies, 2020d). Unity claims that more than 50% of games released in 2020 are powered by Unity (PC/console/top mobile games combined) (Unity Technologies, 2019d).

This huge market share comes with a big problem for vision accessibility. As of version 2021.2, Unity does not offer any advanced vision accessibility features. The only feature available is a color-blind safe color palette generator, which generates a palette of colors that should be distinguishable no matter if and what kind of color vision deficiency a user has (Unity Technologies, 2019e). Many users in the official Unity forums requested features like screen reader support for years, but these features did not make it to the engine yet (Schaller, 2006). The only indication of a potential change in the future is a forum post about *"Accessibility and inclusion"* by the Unity R&D UX Department (Unity R&D UX, 2019). So it looks like Unity has acknowledged that there is a need for better vision accessibility support, but the addition of features like a screen reader support might still be years away.

Unreal Engine

The first version of the Unreal Engine was developed by Epic MegaGames, Inc., known today as Epic Game, Inc., for their 1998 First-Person-Shooter Unreal (Thomsen, 2012; Epic MegaGames, 1998). Since then, it has been the foundation for countless popular games like Fornite (Epic Games, 2017) or Minecraft: Dungeons (Mojang Studios, 2020).

When this thesis was already started, Epic has released version 4.23 of the Unreal Engine. Prior to this version, the only vision accessibility feature of the engine was a way to simulate the three most common color vision deficiencies (Epic Games, Inc, 2020a). This can help developers to find out if all elements are visible and easy to distinguish for people with a color vision deficiency. But since 4.23, the Unreal Engine now allows developers to expose the UI elements to a screen reader (Epic Games, Inc, 2020c). This feature supports the previously mentioned screen readers VoiceOver on iOS and JAWS, NVDA and others on Windows . But it is important to note that this is only working with the Unreal Motion Graphics UI Designer (UMG). Everything else that is rendered in the game is not accessible through a screen reader. Furthermore, screen reader support is still an experimental feature. Epic does not recommend shipping projects with this feature turned on (Epic Games, Inc, 2020b).

Godot

With its first stable version being released in 2014, Godot is a fairly new game engine (Linietsky, 2014). It is an open-source project, funded by donations and free to use for everyone (Juan Linietsky, Ariel Manzur, 2019). Because of Godot's open-source nature, we can get a glimpse into current developments surrounding the accessibility of the Godot engine. Sadly, in the current version (3.2) there are no vision accessibility features implemented in Godot. There has been a major discussion ongoing since 2017 to make both the engine and the engine editor more accessible for visually impaired gamers and developers, but nothing has been implemented yet (ghost, 2017). Other than that, there is a proposal to make a color vision deficiency simulator similar to the one the Unreal Engine has and another proposal to add an accessibility description to UI elements (Two-Tone, 2018; Darilek, 2018). The latter would be the first step towards screen reader support.

SpriteKit and SceneKit

SpriteKit and SceneKit are two engines developed by Apple for 2D (SpriteKit) and 3D (SceneKit) game development (Apple Inc., 2019d; Apple Inc., 2019b). Both engines are limited to Apple platforms iOS, macOS, watchOS, and tvOS, which restricts its reach for both developers and users.

We've grouped the two engines for this discussion because of their similarity when it comes to vision accessibility. Both are built atop of the same standard UI frameworks of the Apple platforms, AppKit on macOS and UIKit on iOS, watchOS, and tvOS. Developers can therefore leverage the accessibility features of those UI frameworks within the games they built in SpriteKit or SceneKit. This includes access to system settings like a high contrast mode or larger font size, easy usage of the systems speech synthesizer, and most importantly, being able to expose every game element to the VoiceOver screen reader (Apple Inc., 2019a; Apple Inc., 2019c). Still, the engines lacks more advanced features for vision accessibility in games, like a Sonar or automatic navigation feature we've seen in The Last of Us II, section 2.3.4.

Direct Comparison

The result of our research in the visual accessibility of the different engines can be seen in Table 2.1. It shows that SpriteKit and SceneKit are the most advanced engines when it comes to vision accessibility features. But SpriteKit and SceneKit might not be an option for a developer if they want to reach gamers beyond the Apple Platforms. Unreal Engine seems to be the next best choice, but its screen reader support is limited to UI elements and is still an experimental feature. Besides that, none of the engines do support a Sonar system or Auto Navigation.

	Unity	**Unreal Engine**	**Godot**	**SpriteKit SceneKit**
Color Vision Deficiency Tools	**yes (Palette Generator)**	**yes (Simulation)**	no	no
Screen Reader Support	no	Partially (UI Only)	no	**yes**
Speech Synthesizer Access	no	no	no	**yes**
System Settings Access	no	no	no	**yes**
Sonar	no	no	no	no
Auto Navigation	no	no	no	no

Table 2.1: Comparison of the different engines based on their vision accessibility features

So these results leave developers who want to make a cross-platform game vision accessible with two options: Build everything from scratch or search if other developers have created plugins to extend the selected engine. So in the next section, we will look into work that has been done by other developers to extend the functionality of the Unity engine in terms of vision accessibility. We've selected Unity because of its popularity and its lack of any substantial vision accessibility features, as we've shown in this section.

2.5 Vision Accessibility Extensions for Unity

The most common way to extend the functionality to a Unity project is through asset packages (Unity Technologies, 2019a). Asset packages are a bundle of files like scripts or materials that can be reused in other projects. Developers can then distribute those Asset packages themselves or sell them to other developers through the Unity Asset Store[28].

[28]https://assetstore.unity.com

2.5.1 Asset Packages for Vision Accessibility

In the following, we will take a brief look at several such asset packages that concern themselves with vision accessibility.

Colorblind Effect

The Colorblind Effect[29] asset package is a free addition to Unity that adds a new camera effect to Unity to simulate Protanopia, Deuteranopia, or Tritanopia. It enables a developer to test out if all elements in the game are distinguishable for a person with one of these color deficiencies. This functionality is similar to the previously discussed color deficiency simulation built into the Unreal Engine (Epic Games, Inc, 2020a).

UI Accessibility Plugin

The goal of the UI Accessibility Plugin[30] (UAP) is to make UI and UI-heavy games accessible by providing a screen reader. The way it works is that the developer has to add a specific accessibility component to every object that should be exposed to the screen reader. The screen reader then searches for all objects with this accessibility component and lets a user step through them sequentially and interact with them via the keyboard or touch gestures. This interaction is only working with the UAP screen reader and does not work with any system wide screen readers like VoiceOver or Narrator.

The UAP is available through the Unity Asset Store as a paid extension.

[29] https://assetstore.unity.com/packages/vfx/shaders/fullscreen-camera-effects/colorblind-effect-76360

[30] https://assetstore.unity.com/packages/tools/gui/ui-accessibility-plugin-uap-87935

Responsive Spatial Audio for Immersive Gaming

The Responsive Spatial Audio for Immersive Gaming[31] asset package is a project by Microsoft which can be downloaded for free in the Unity Asset Store. The package adds tools to help with unconstrained navigation and observation in a Unity game using audio. Developers can tag objects with a description and make that description accessible to a player by letting them scan the scene and getting feedback through earcons and text-to-speech. This is similar to Enhanced Listen Mode of The Last Of Us II. But the package goes a step further with a tool called Accessible Navigation. This tool guides a user to a selected object by using an agent that moves around obstacles and emits spatial earcons (Huston, 2019). Thus, the tool allows to fully navigate a 3D-environment without the possibility of getting stuck.

Accessibility Features for Game Creator

Game Creator[32] is a set of tools that makes game development in Unity even easier. It includes tools like a visual scripting language, a save/load system, and a module manager to extend its functionality further. One of these modules, Accessibility Features for Game Creator[33], concerns itself with visual, auditory, motoric, and cognitive accessibility . For the visual accessibility part, the package allows to dynamically increase the text size and apply an adjustment or replacement of all colors within a scene (Pivec Labs, 2020). It also includes a shader to show the outlines of all objects, allowing for an effect similar to the High Contrast Display in The Last Of Us II, which we've discussed in section 2.3.4.

The Accessibility Features for Game Creator is available in the Unity Asset Store. It is a paid product and only works in combination with Game Creator.

[31]https://assetstore.unity.com/packages/tools/game-toolkits/responsive-spatial-audio-for-immersive-gaming-a-microsoft-garage-144702

[32]https://assetstore.unity.com/packages/tools/game-toolkits/game-creator-89443

[33]https://assetstore.unity.com/packages/tools/utilities/accessibility-features-for-game-creator-149171

2.5.2 Areas of Improvement

We have now reached a point where we've seen how people with vision disabilities interact with computing devices and with games. We've also discussed the tools and extensions to Unity that allow building vision accessible games. Based on all this knowledge, we've created Figure 2.19, 2.20 and 2.21 to point out different areas where vision accessibility can be enhanced for games created in Unity. The measurements are split into enhancements for individuals with a color vision deficiency, people with low vision, and enhancements for any form of vision impairment. We also gave suggestions to make enhancements in all of these areas. Note that not all suggestions in the Figures can be created as an extension for Unity. Some of them are deeply interconnected with every game and thus, have to be created specifically for each one of them. An example would be the addition of a realistic sound design. Areas where a plugin could not help to create an accessibility feature are dashed in the Figures.

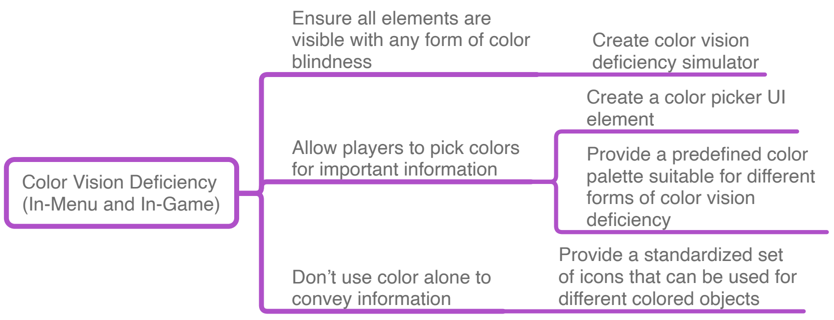

Color Vision Deficiency (In-Menu and In-Game)
Ensure all elements are visible with any form of color blindness
Create color vision deficiency simulator
Allow players to pick colors for important information
Create a color picker UI element
Provide a predefined color palette suitable for different forms of color vision deficiency
Don't use color alone to convey information
Provide a standardized set of icons that can be used for different colored objects

Figure 2.19: Areas of improvement for individuals with a color vision deficiency

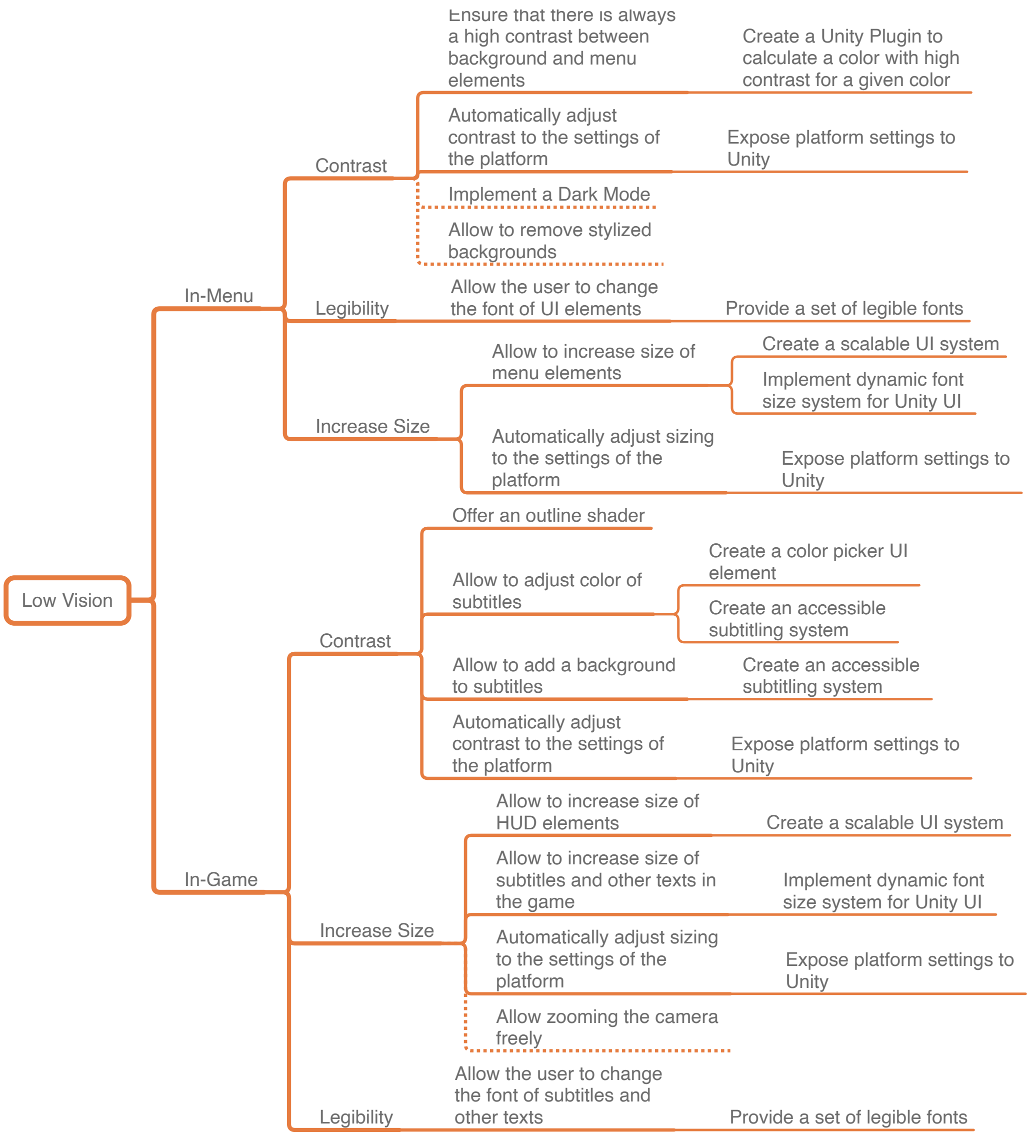

Low Vision
In-Menu
Contrast
Ensure that there is always a high contrast between background and menu elements
Create a Unity Plugin to calculate a color with high contrast for a given color
Automatically adjust contrast to the settings of the platform
Expose platform settings to Unity
Implement a Dark Mode
Allow to remove stylized backgrounds
Legibility
Allow the user to change the font of UI elements
Provide a set of legible fonts
Increase Size
Allow to increase size of menu elements
Create a scalable UI system
Implement dynamic font size system for Unity UI
Automatically adjust sizing to the settings of the platform
Expose platform settings to Unity
In-Game
Contrast
Offer an outline shader
Allow to adjust color of subtitles
Create a color picker UI element
Create an accessible subtitling system
Allow to add a background to subtitles
Create an accessible subtitling system
Automatically adjust contrast to the settings of the platform
Expose platform settings to Unity
Increase Size
Allow to increase size of HUD elements
Create a scalable UI system
Allow to increase size of subtitles and other texts in the game
Implement dynamic font size system for Unity UI
Automatically adjust sizing to the settings of the platform
Expose platform settings to Unity
Allow zooming the camera freely
Legibility
Allow the user to change the font of subtitles and other texts
Provide a set of legible fonts

Figure 2.20: Areas of improvement for individuals with low vision

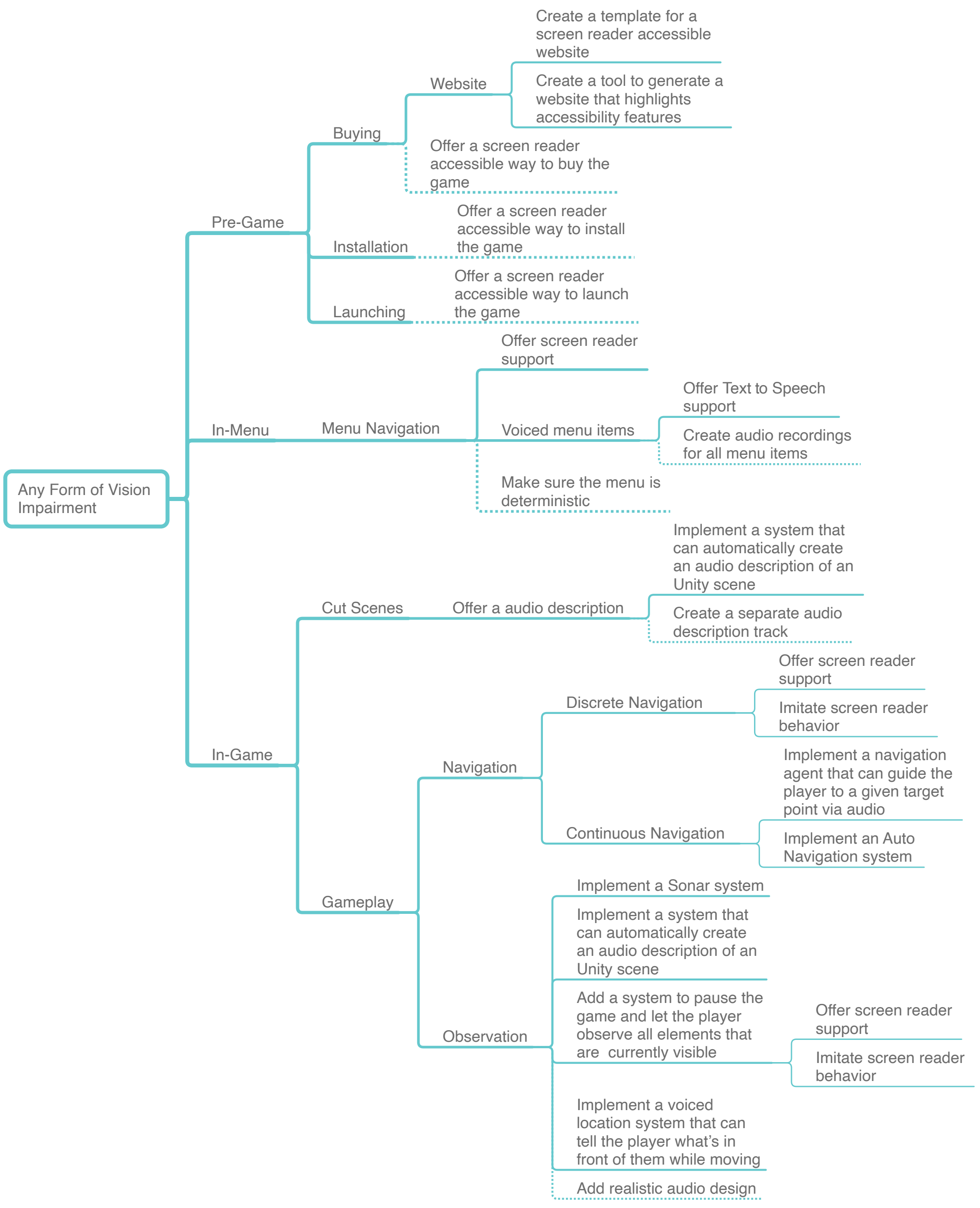

Any Form of Vision Impairment
Pre-Game
Buying
Website
Create a template for a screen reader accessible website
Create a tool to generate a website that highlights accessibility features
Offer a screen reader accessible way to buy the game
Installation
Offer a screen reader accessible way to install the game
Launching
Offer a screen reader accessible way to launch the game
In-Menu
Menu Navigation
Offer screen reader support
Voiced menu items
Offer Text to Speech support
Create audio recordings for all menu items
Make sure the menu is deterministic
In-Game
Cut Scenes
Offer a audio description
Implement a system that can automatically create an audio description of an Unity scene
Create a separate audio description track
Gameplay
Navigation
Discrete Navigation
Offer screen reader support
Imitate screen reader behavior
Continuous Navigation
Implement a navigation agent that can guide the player to a given target point via audio
Implement an Auto Navigation system
Observation
Implement a Sonar system
Implement a system that can automatically create an audio description of an Unity scene
Add a system to pause the game and let the player observe all elements that are currently visible
Offer screen reader support
Imitate screen reader behavior
Implement a voiced location system that can tell the player what's in front of them while moving
Add realistic audio design

Figure 2.21: Areas of improvement for individuals with any form of vision impairment

2.6 Summary

This chapter gave a broad overview of how visually impaired people can experience the digital world, how they can play video games, and what developers can do to enable this community to enjoy more games.

We've started with some terminology and pointed out the difference between disability and impairment and how to use them. We then listed different types of visual impairments, how the WHO classifies them in terms of visual acuity, and what the term low vision means. Color vision deficiency was also briefly discussed.

The second section listed different software and hardware solutions that enable visually impaired people to interact with the highly visual computing devices of today. This included, among others, settings to enlarge screen content, refreshable braille displays, and screen readers. Intelligent Personal Assistants (IPAs) were also mentioned, which can speed up access to a wide range of simple tasks through a voice interface.

We continued with vision accessible video gaming in the third section. The Gaming Workflow was defined as the steps a user needs to go through from wanting to play a game to actually playing it. We then talked about how visually accessible the Pre-Game part of the Gaming Workflow is on different platforms and about problems in the In-Menu phase. After that, we continued with Audio Games, which are audio-only experiences often designed specifically for visually impaired gamers. Next, we've discussed games that were not designed for people with vision impairments, but are playable because of their realistic audio design. And finally, we've looked at games that were intentionally made visually accessible by the developers, like The Last of Us II (Naughty Dog, 2020).

Such games are rare and we wanted to find out why this is by looking at how video games are built. So the fourth section concerned itself with video game engines. We defined game engines as a set of tools to ease the development of video games. After that, we compared some engines in terms of vision accessibility features and found that the only engines that have extensive vision accessibility support are limited to the Apple platforms. The far more popular, cross-platform engine Unity is very limited in terms of built-in vision accessibility features.

But there are ways to extend the Unity engine and we've looked at some of those extensions in the fifth section of the chapter. Most of these extensions are paid, limited, or not frequently maintained, which limits their potential. Based on what plugins we've seen and all the knowledge from the previous chapters, we've created Figure 2.19, 2.20 and 2.21 to highlight different areas of how Unity could be extended to build vision accessible games.

All of this shows that there is still an urgent need for further work in the area of vision accessible gaming. So we have decided to create our own Asset Package for Unity to ease development for the visually impaired gaming community. The next chapter will concern itself with our idea, the design, and the implementation of this package.

3 The Unity Accessibility Toolkit: Design and Implementation

In the last chapter, we have discussed the different ways of how visually impaired players can interact with their computing devices and play video games. We've seen the tools that are used to create video games and showed that the popular Unity game engine has a significant lack of features to create games for the visually impaired. This is why we started our work on the *Unity Accessibility Toolkit*, an asset package to help make Unity games more visually accessible. In this chapter, we want to present the design and implementation of this package.

Before starting with any specifics, we want to briefly discuss our intentions behind this project. Our goal is to provide a toolkit that can be integrated into as many games as possible. This makes our target group all Unity developers, though more specifically, developers that have an interest in making games accessible, but lack the resources to build something like a screen reader themselves. Therefore, our main requirements for the toolkit are:

- *Ease of Usage*: The tools should be easy to integrate into a game. This should minimize the work overhead to make a game accessible and thus, increase adaption.
- *Base on existing technologies*: Use existing and familiar technologies wherever possible. This should ease adaption by the developer and acceptance by the gamer.
- *Extensibility*: The toolkit should lay a foundation that can be reused for other accessibility technologies included in future versions of the toolkit.

- *Open-Source*: The toolkit should be available to everyone free of charge as an open-source project.

With these requirements in mind, we turn back to Figure 2.21. In this Figure, we showed that there are tons of possible areas of improvement for vision accessibility. Implementing all of them would go beyond the scope of the thesis, so we've focused on those we consider most important. This lead to selection of the following four areas:

- *Menu Navigation*: Help navigation through the menu of a game to start the actual gameplay or to adjust settings
- *Environment Observation*: Pausing the game and letting the user observe all the elements that are currently visible
- *Discrete Navigation*: Navigating elements or a character on a fixed grid
- *Continuous Navigation*: Freely moving a character around an environment

Menu navigation, discrete navigation and environment observation have a single thing in common: They can all be realised by creating a screen reader. So the first part of our package will concern itself with providing such a screen reader. For the continuous navigation, we've decided to implement a navigation agent. The design of both technologies will be discussed in the next section of this chapter.

3.1 Design

As a basis for all our accessibility tools, we needed an auditory signifier that a developer can attach to any game object. Norman (2002) refers to signifiers as *"any mark or sound, any perceivable indicator that communicates appropriate behavior to a person"*. Our accessibility signifier does have a dual purpose. Firstly, it is used to store details about a game object and possible interactions with it. Secondly, it lets our accessibility tools know which game objects are even accessible. The last point comes from the fact that both our screen reader and the navigation agent system behaves as an additional layer in the interaction framework between interface and the user, as we've discussed in Section 2.2.2. The additional layer will collect information about which game objects should be presented in an auditory form.

For the concrete design of our accessibility signifier, we drew inspiration from the `UIAccessibility` protocol from UIKit, which is a standard UI framework for iOS (Apple Inc., 2020b). Similar to objects in UIKit, each of our signifiers communicates the information about the game object it's attached to using the following four properties:

- *Label*: A short but concise label of the object.
- *Traits*: Indication of how the game object behaves or should be treated.
- *Value*: The value of the game object.
- *Description*: Detailed description of the function of the game object. This should give additional context if the function of the object and how to interact with it if this isn't obvious through label, value, and traits.

Note that these properties are available for every signifier of a game object, but not every game object has to provide a value for every property. Figure 3.1 should make this point clearer by showing an example of how the values can be used for a button and a slider.

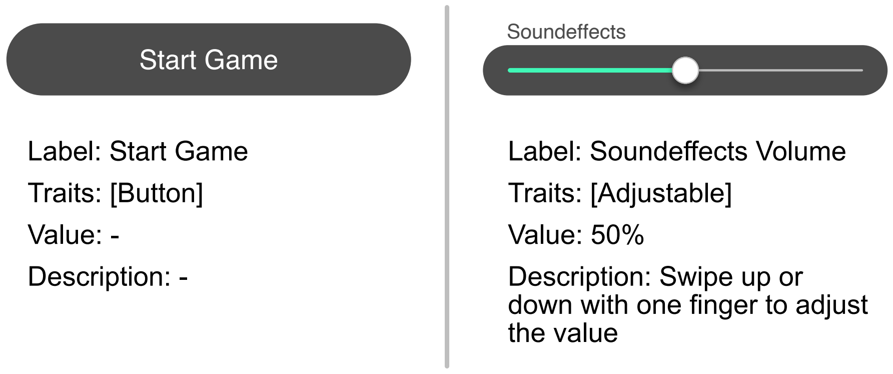


Figure 3.1: Examples of how to use the values of the accessibility signifier with a button (left) and a slider (right)

The signifier for a button needs a label and traits, but doesn't have value. It also might not need a description, if its label already states its function clearly enough. A signifier for a slider on the other hand has besides its label and traits also a value indicating the position of the slide handle. The slider also might need description of how to interact with it, if the interaction is not common.

The example also gives a glimpse at how traits can be use. Thr following lists all the traits that we provide:

- *Button*: Marking it as a button.
- *Toggle*: Marking it as a toggle.
- *Link*: Marking it as a link to somewhere outside the game, like the developers' website.
- *Image*: Marking it an image.
- *Static Text*: Marking it as uneditable text.
- *Header*: Marking it as a header text.
- *Summary Element*: Indicating that it gives a summary of multiple game objects .
- *Adjustable*: Marking it as an adjustable game object, like a slider.
- *Selected*: Indicating that the game object has been selected. An example usage would be marking a checkbox button as checked.
- *Not enabled*: Indicating that the game object has been disabled.
- *Allows Direct Interaction*: Indicating that the game object wants to accept direct touch or mouse interaction, instead of forwarding this interaction to the accessibility tool.
- *Hide From Screen Reader*: Indicating that the game object is an accessible element, but should be hidden from the screen reader.

Our selection of these traits is based on both the `UIAccessibilityTraits` (Apple Inc., 2020a) UIKit and the built-in controls that Unity features (Unity Technologies, 2020b). Developers can also to add custom traits to fit the needs of their game.

As we've already mentioned, these accessibility traits and everything else the accessibility signifier holds should be used to show our accessibility tools which game objects in a game are accessible and how to handle them. So next we're going to discuss what our screen reader should do with this information.

3.1.1 Design of the Screen Reader

The main reason why there is even a need for this screen reader is that game objects in Unity are not visible to other screen readers like VoiceOver or Windows Narrator, as we've pointed out in section 2.4.2. It would be preferable to use those screen readers over a self-built one because they are already familiar to the gamer in both interaction and auditory output, i.e. the synthesized voice. This is something we want to consider for our design. So instead of directly building a screen reader, we will have a *screen reader manager* that captures all the accessible game objects on the screen. This manager then reads the system settings to see if a system screen reader is available and can be accessed. If it can be accessed, the information about the accessible game objects is forwarded to the so-called *native screen reader*, an interface to a system screen reader. If the system screen is not available, the information is forwarded to the *custom screen reader*, a screen reader built from the ground up for Unity. Figure 3.2 gives a visual representation of this design. Both the native and the custom screen reader will be discussed in the following.

Custom Screen Reader Like most other screen readers, our custom screen reader gives access to all accessible game objects in a serial manner. We use the screen location of each accessible object to sort them first by their y-coordinate. If two game objects have the same y-coordinate, those objects are sorted by the x-coordinate. Figure 3.3 shows how this sorting works on an example UI.

When entering a screen, the first game object in the sequence will be highlighted by the screen reader. The screen reader will then announce the different properties of our accessibility signifier in the following order: label, traits, value and finally the description. This auditory announcement is done via the speech synthesizer of the system. We also use the *screen reader visualizer* to draw a rectangle around the currently highlighted element, to give people with a mild vision impairment an additional hint.

The highlighting can be changed either sequentially or via browsing. The sequential change moves the highlighting either to the next or the previous element in our sequence. The browsing strategy changes the highlighting

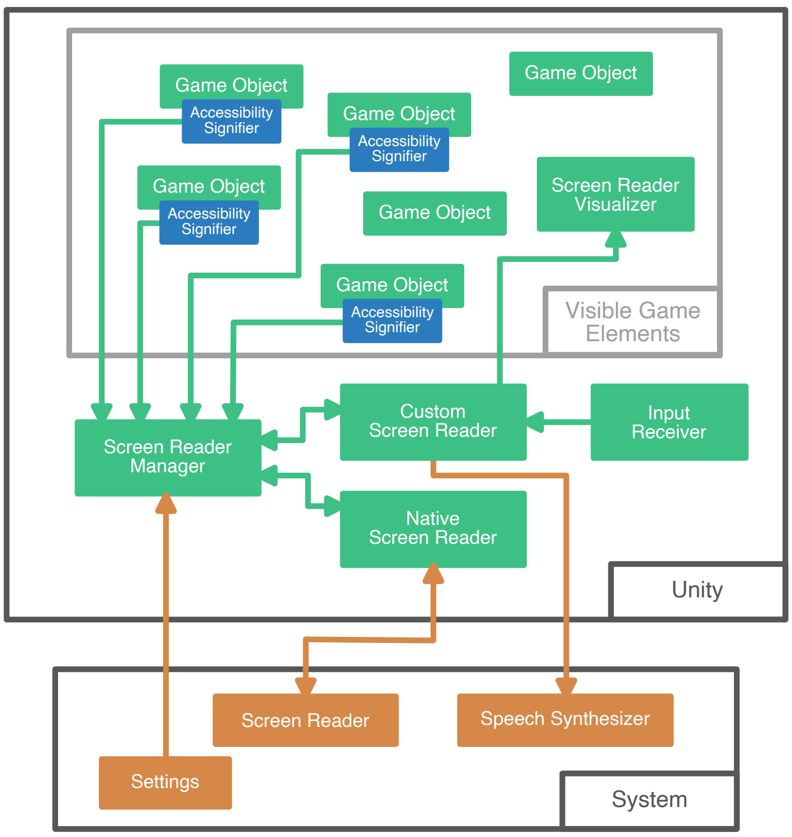


Figure 3.2: Architecture of the screen reader

via a pointing device, like a mouse or a finger. To indicate that the highlighting was successfully changed, an earcon will be played and the different properties of the accessibility signifier will be announced. For the sequential change, the user might try to move to the next element when the last element is highlighted or to the previous element if the very first element is highlighted. If that happens, another earcon will be played to signal that a bound of the sequence has been reached and the highlighting didn't change. This should not trigger our screen reader to announce the properties of the signifier again.

The input for changing the highlighting will come from our *input receiver*. This element can either forward touch gestures or keyboard and mouse

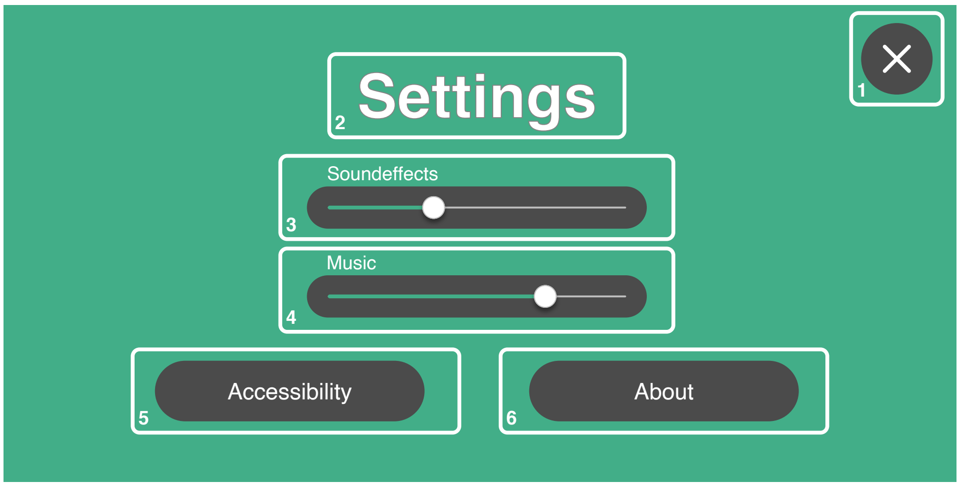


Figure 3.3: Example of how our screen reader will sort the game object with the accessibility signifier

input to our screen reader, depending on the platform. Besides changing the highlighting, it is also responsible to forward interactions like activating an object, and, if suitable, increase and decrease the value of it.

Native Screen Reader The sole purpose of the native screen reader is to forward the accessible game objects and their signifiers to a system screen reader and to communicate the interaction from that system screen reader back to the game. The actual serialization of the interface, the interaction, and announcing the properties of the signifier is up to the system screen reader.

We think having both the native and a custom screen reader in place will give us the benefits of familiarity, if the system screen reader can be accessed, combined with broad platform support if the system screen reader can't be accessed. This should ensure a great experience for gamers and broader adaptation by developers.

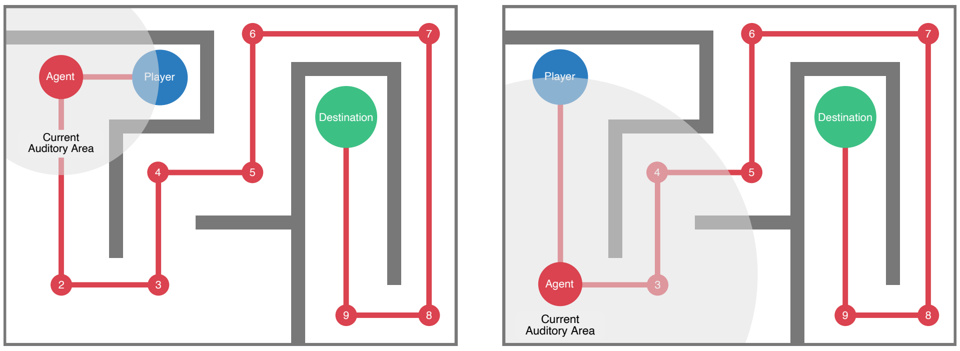


Figure 3.4: Sketch of how continuous navigation is eased via the navigation agent. Left: Initial position of player and navigation agent. Right: The first intersection was reached and the agent moved to the second one.

3.1.2 Design of the Navigation Agent

The idea of the navigation agent is to ease the continuous navigation from any location to any reachable point of interest. To make this work, the developer has to provide a user interface to chose this destination and forward it to our agent. When it has been selected, we will calculate a path from the users' current position to this location. To ease navigation, the path only consists of straight lines. Our navigation agent will then move to the first intersection between two lines. The user should then be able to locate the navigation agent by triggering a positional earcon upon hitting a key or performing a touch gesture. The location of the agent relative to the player's position should be determinable through stereo sound, the distance to it through the loudness of the earcon.

After reaching the agent, it will emit another earcon to confirm that it has been reached. The agent will then move to the next intersection of the path. This process will repeat until the final destination has been reached. Figure 3.4 shows a graphical representation of this interaction flow.

While following the navigation agent, it might happen that the player gets stuck or cannot hear the earcon anymore because they went in the wrong direction. To resolve such a scenario, users can trigger a recalculation of the path at any time. How this is intended to work can be seen in Figure 3.5.

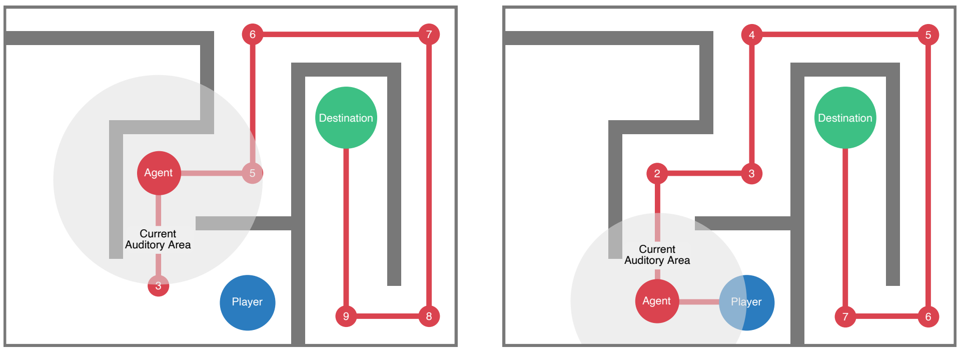


Figure 3.5: Sketch of how recalculating the path of the navigation agent is working. Left: Player has moved off the intended path. Right: User has triggered the recalculation of the path and the agent was re-positioned.

This concludes the design of both the screen reader and the navigation agent. The design we've presented in this section could be theoretically implemented for any kind of game engine. But we've decided to build our toolkit for Unity, so next we're going to present how the proposed toolkit has been implemented for Unity.

3.2 Basics of the Implementation

The basic building blocks of Unity games are `GameObjects` and `Components`. Every object within a game is a `GameObject`. But `GameObjects` themselves are nothing else then empty containers; they get their properties from `Components`. An example for a basic `Component` is the `Transform Component`, which adds position, rotation and scale properties to the `GameObject`.

For our plugin, we came up with an `Component` called `UA11YElement`. The `UA11YElement Component` marks a `GameObject` as an accessible element and gives it the basic properties needed for every accessibility technology that we implement. So the `UA11YElement` will act as our accessibility signifier that we've described in Section 3.1. The simplified class diagram of the `UA11YElement` can be seen in Figure 3.6.

The `traits` and the other properties like `label` and `description` cover

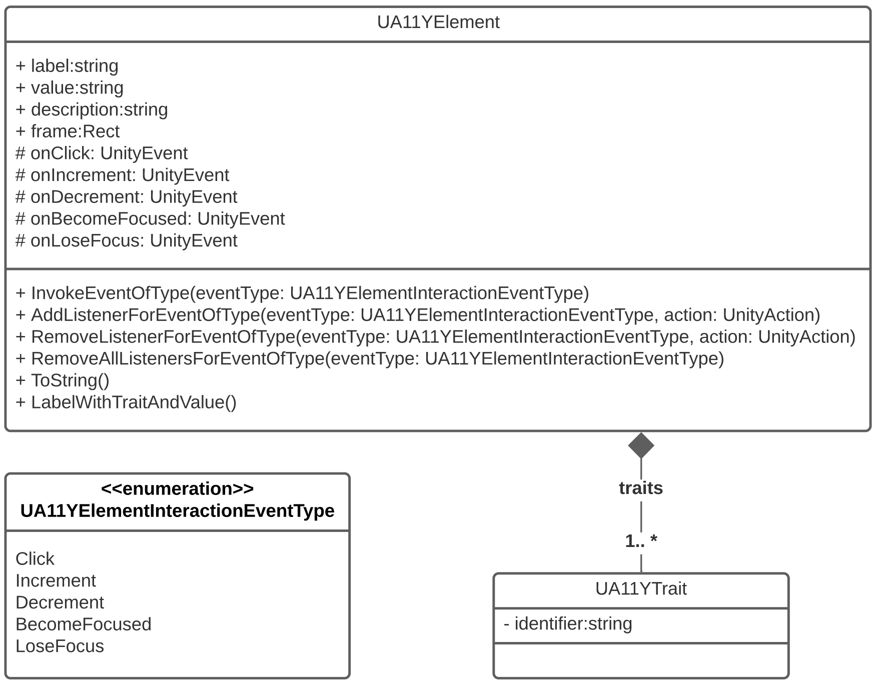

UA11YElement
+ label:string
+ value:string
+ description:string
+ frame:Rect
# onClick: UnityEvent
# onIncrement: UnityEvent
# onDecrement: UnityEvent
# onBecomeFocused: UnityEvent
# onLoseFocus: UnityEvent
+ InvokeEventOfType(eventType: UA11YElementInteractionEventType)
+ AddListenerForEventOfType(eventType: UA11YElementInteractionEventType, action: UnityAction)
+ RemoveListenerForEventOfType(eventType: UA11YElementInteractionEventType, action: UnityAction)
+ RemoveAllListenersForEventOfType(eventType: UA11YElementInteractionEventType)
+ ToString()
+ LabelWithTraitAndValue()
<<enumeration>>
UA11YElementInteractionEventType
Click
Increment
Decrement
BecomeFocused
LoseFocus
traits
1.. *
UA11YTrait
- identifier:string

Figure 3.6: Simplified class diagram of the `UA11YElement`

the signifier part of the `UA11YElement`, but doesn't cover reacting to interaction. For this, we have implemented a system based on the observer pattern (Gamma et al., 1995). The developer can register as a listener to the `UA11YElement` for any specified `UA11YElementInteractionEventType`. When the event type is invoked through `InvokeEventOfType(eventType)`, the callback to all the listeners will be called. Triggering the interaction itself is still up to the accessibility technology and other game specific parts.

From a developers perspective, the `UA11YElement` comes with the downside that both rerouting the interaction from accessibility technologies to the game and filling in the accessibility properties has to be done manually. While this cannot be eased for most parts of the game, we can ease it for the menus and other UI parts. For this, we provide multiple subclasses of the `UA11YElement Component` for common Unity UI elements. All of them can be seen in Figure 3.7. These subclasses automatically fill in accessibility information and handle certain interactions. For example, when the `UA11YButton Component` is attached to a `GameObject` that has a `Button Component`, it automatically sets the `traits` to `[.Button]`, uses the string value of its `Label Component` as the accessibility `label` and forwards the `Click UA11YElementInteractionEventType` to Unitys' event system. This tremendously eases supporting accessibility technologies for standard UIs, because developers only have to attach the correct `Component` to the UI elements and the rest will work automatically.

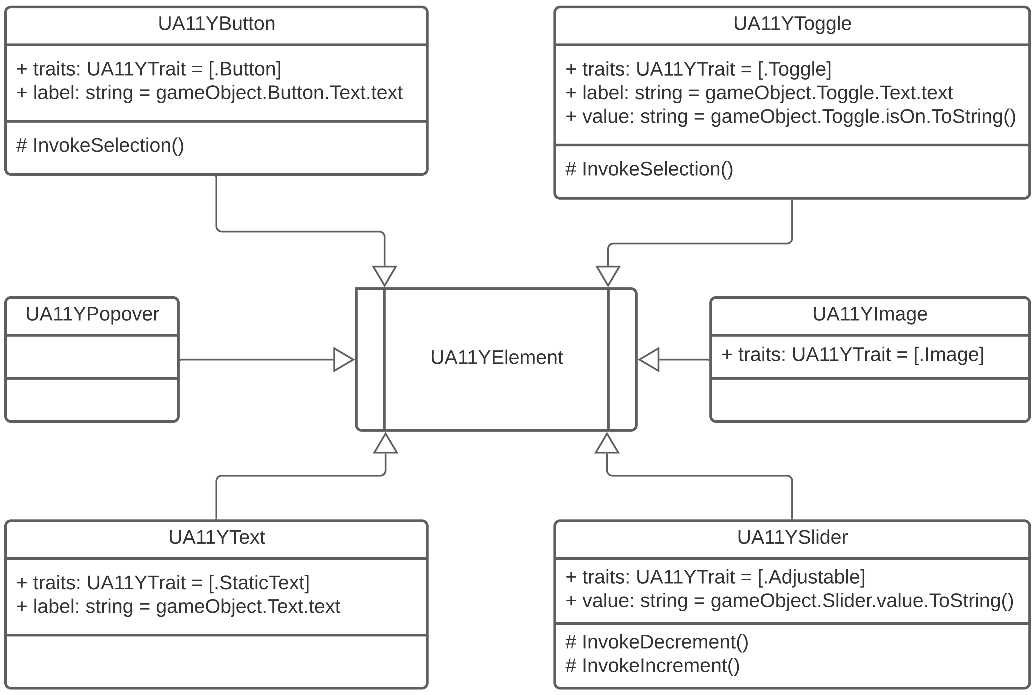

UA11YButton
+ traits: UA11YTrait = [.Button]
+ label: string = gameObject.Button.Text.text
# InvokeSelection()
UA11YToggle
+ traits: UA11YTrait = [.Toggle]
+ label: string = gameObject.Toggle.Text.text
+ value: string = gameObject.Toggle.isOn.ToString()
# InvokeSelection()
UA11YPopover
UA11YElement
UA11YImage
+ traits: UA11YTrait = [.Image]
UA11YText
+ traits: UA11YTrait = [.StaticText]
+ label: string = gameObject.Text.text
UA11YSlider
+ traits: UA11YTrait = [.Adjustable]
+ value: string = gameObject.Slider.value.ToString()
# InvokeDecrement()
# InvokeIncrement()

Figure 3.7: Simplified class diagram of the `UA11YElement` sub-classes we provide to ease supporting accessibility technologies in Unity UIs

The accessibility technology that is most likely to be used when interacting with Unity UIs is the screen reader. So in the next part, we will take a close look at its implementation and features.

3.3 Implementation of the Screen Reader

We've already discussed in Section 3.1.1 that our screen reader technology consists of multiple interconnected parts, both in Unity and on the system side. The main component on the Unity side is the screen reader manager, which we're gonna describe in the following. Note that the implementation on the system side was only done fully for iOS and partially for macOS and Windows. Supporting more platforms than these would go beyond the scope of the thesis. We nevertheless made sure that the interface from Unity was built in a way that it is easily extendable for other platforms. Extending the platform support should be covered in potential future work.

Without further ado, let's dive into the implementation of the screen reader manager.

3.3.1 Screen Reader Manager Implementation

The screen reader manager is the central part of our screen reading technology and was implemented in Unity in form of the `UA11YScreenReader-Manager Component`. We provide a Unity prefab[1], which is a pre-configurated and reusable Unity `GameObject`, with this `Component` attached to it. A simplified class diagram of the manager can be seen in Figure 3.8.

When the developer adds the `UA11YScreenReaderManager` prefab to a Unity scene, the manager will first determine what screen reading technology is available. If the currently available platform has a screen reader and we've implemented the required parts on the system side to communicate with this screen reader, the manager will create a `UA11YNativeScreenReader`. Otherwise, a `UA11YCustomScreenReader` is created. After that, the manager will fetch all `GameObjects` from the scene that have a `UA11YElement Component` attached to them and filter out those who are not visible on the screen. Finally, these accessibility elements are forwarded to the screen reader that was created earlier.

[1]https://docs.unity3d.com/Manual/Prefabs.html

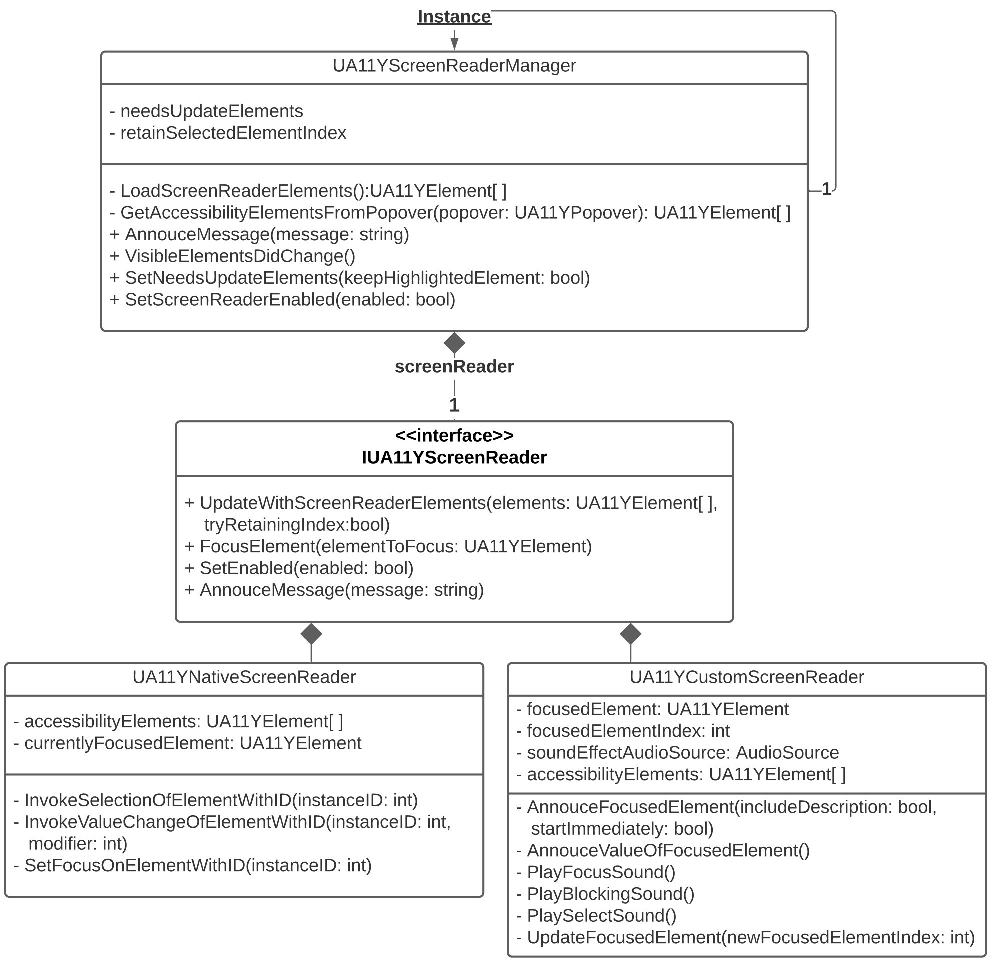

Instance
1
UA11YScreenReaderManager
- needsUpdateElements
- retainSelectedElementIndex
- LoadScreenReaderElements():UA11YElement[ ]
- GetAccessibilityElementsFromPopover(popover: UA11YPopover): UA11YElement[ ]
+ AnnouceMessage(message: string)
+ VisibleElementsDidChange()
+ SetNeedsUpdateElements(keepHighlightedElement: bool)
+ SetScreenReaderEnabled(enabled: bool)
screenReader
1
<<interface>>
IUA11YScreenReader
+ UpdateWithScreenReaderElements(elements: UA11YElement[ ], tryRetainingIndex:bool)
+ FocusElement(elementToFocus: UA11YElement)
+ SetEnabled(enabled: bool)
+ AnnouceMessage(message: string)
UA11YNativeScreenReader
- accessibilityElements: UA11YElement[ ]
- currentlyFocusedElement: UA11YElement
- InvokeSelectionOfElementWithID(instanceID: int)
- InvokeValueChangeOfElementWithID(instanceID: int, modifier: int)
- SetFocusOnElementWithID(instanceID: int)
UA11YCustomScreenReader
- focusedElement: UA11YElement
- focusedElementIndex: int
- soundEffectAudioSource: AudioSource
- accessibilityElements: UA11YElement[ ]
- AnnouceFocusedElement(includeDescription: bool, startImmediately: bool)
- AnnouceValueOfFocusedElement()
- PlayFocusSound()
- PlayBlockingSound()
- PlaySelectSound()
- UpdateFocusedElement(newFocusedElementIndex: int)

Figure 3.8: Simplified class diagram of the `UA11YScreenReaderManager`

After this initial fetching of the accessible elements, it is up to the game itself to ensure that the screen reader elements stay up to date. Whenever appropriate, the `VisibleElementsDidChange()` method of the `UA11YScreenReaderManager` can be called. This will trigger a refetch of all visible `GameObjects` with an `UA11YElement Component` attached and the screen reader update with these objects. This operation can be pretty resourceful, especially when dealing with system screen readers. So we advise that these refetches are scheduled thoughtfully.

The previously described sequence of events is shown graphically as a sequence diagram in Figure 3.9. This diagram intentionally left out the inner workings of the `UA11YNativeScreenReader` and the `UA11YCustomScreen-Reader`. These are gonna be the topics of the following two sections.

3.3.2 Custom Screen Reader Implementation

The `UA11YCustomScreenReader` is a screen reader built from the ground up for Unity. Figure 3.12 shows the class diagram of the main parts of this screen reader. On the input side, we have an `UA11YInput` to retrieve actions from the gamer. On the output side, there is the `UA11YSpeechSynthesizer` to convert the properties of `UA11YElements` to an auditory form, and the `UA11YCustomScreenReaderVisualizer` to draw a frame around the currently focused element.

To understand how the `UA11YCustomScreenReader` works, we again turn to Figure 3.9 and to its extension in Figure 3.11. When no native screen reader is accessible, the `UA11YScreenReaderManager` creates the `UA11YCustomScreen-Reader` and forwards the initial `GameObjects` with an `UA11YElement Component` attached to this screen reader by calling `UpdateWithScreenReaderElements(e)`. Before being stored as the `accessibilityElements` property, the `GameObjects` will be sorted based on their screen location, as we've discussed in Section 3.1.1. The Unity Engine will then call the `Start` method of the `UA11Y-CustomScreenReader`. Here, the screen reader will create the `UA11YSpeech-Synthesizer`, the `UA11YCustomScreenReaderVisualizer` and a fitting `UA11Y-Input`. On mobile platforms with a touch screen, a `UA11YMobileInput` will

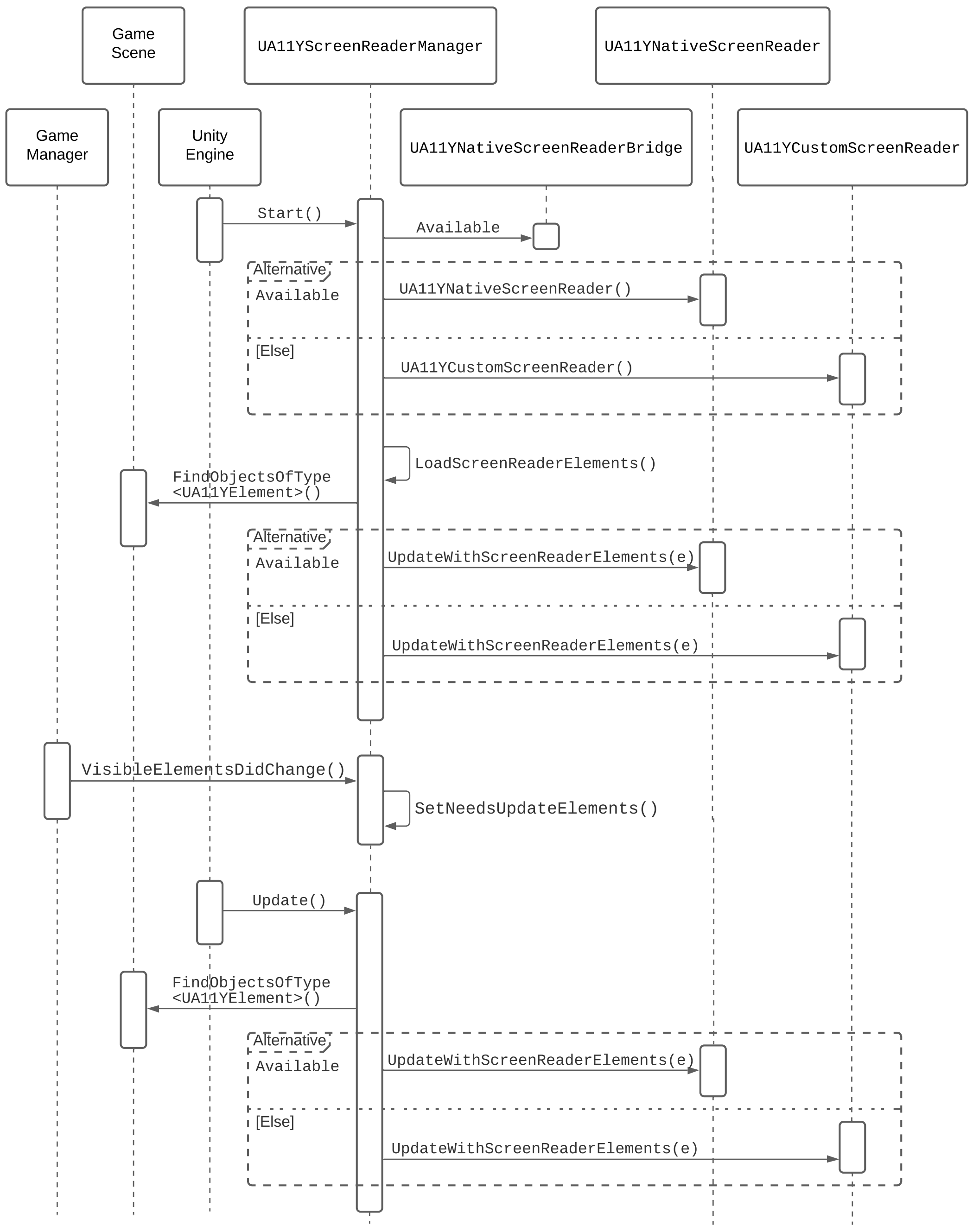

Game Scene
UA11YScreenReaderManager
UA11YNativeScreenReader
Game Manager
Unity Engine
UA11YNativeScreenReaderBridge
UA11YCustomScreenReader
Start()
Available
Alternative
Available
UA11YNativeScreenReader()
[Else]
UA11YCustomScreenReader()
LoadScreenReaderElements()
FindObjectsOfType <UA11YElement>()
Alternative
Available
UpdateWithScreenReaderElements(e)
[Else]
UpdateWithScreenReaderElements(e)
VisibleElementsDidChange()
SetNeedsUpdateElements()
Update()
FindObjectsOfType <UA11YElement>()
Alternative
Available
UpdateWithScreenReaderElements(e)
[Else]
UpdateWithScreenReaderElements(e)

Figure 3.9: Sequence diagram of the screen reader manager. The *Game Manger* is a placeholder for a arbitrary element of a game that will trigger updating the screen reader.

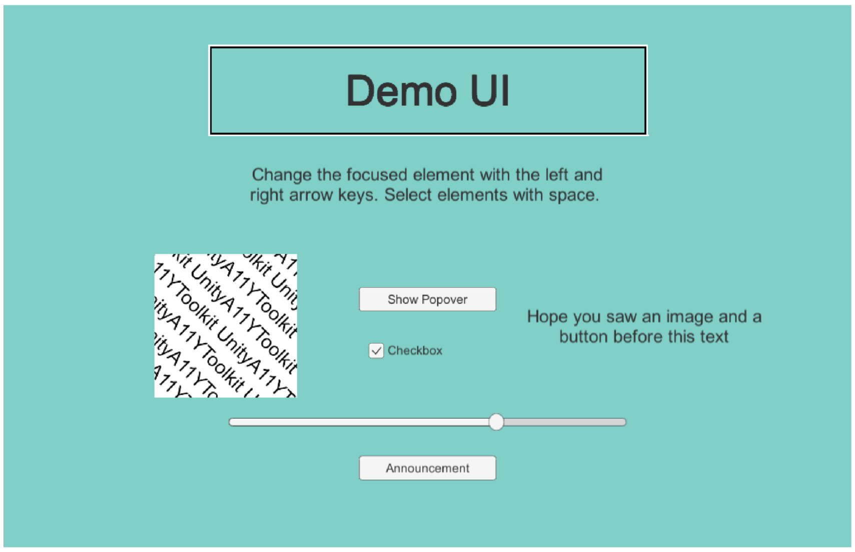

Demo UI
Change the focused element with the left and right arrow keys. Select elements with space.
Show Popover
Checkbox
Hope you saw an image and a button before this text
Announcement

Figure 3.10: A demo UI with the `UA11YCustomScreenReader` activated. The `UA11YCustomScreenReaderVisualizer` draws a high constrat frame around the currently focused element.

be created. The `UA11YDesktopInput` will be used on devices with mouse and keyboard attached.

After that, the screen reader will update itself for the first time. This means that it will set the focus on the first object of the `accessibilityElements` array and announce the different properties of the `UA11YElement Component` attached to it, i.e. `label`, `value`, `traits` and `description`. Synthesizing the text is done by the system. So the `UA11YSpeechSynthesizer` itself only acts as a bridge to the system speech synthesizer.

Besides announcing its values, we also want to give a visual indication for the focused element as a visual hint for people with low vision. So we draw a rectangle around the focused element every time the Unity Engine redraws the UI, via the `OnGUI()` method. This frame consists of a thin black and a thin white line to be well easily visible on every background. Figure 3.10 shows how this frame looks like in a demo UI.

This last step concludes the initial phase of the screen reader. After it, the screen reader sits there idle until the `UA11YInput` recognizes some input from the player. If a focus change is trigged via `FocusNextElement()`, `FocusPreviousElement()` or `FocusElementAtPosition(p)`, and the update was successful, the screen reader will play a sound to indicate the change, announces the values of the newly focused element and draws the rect around it when the next `OnGUI()` call happens. In case the focus could not be successfully updated, like when the user tries to focus the previous element when the very first element is focused, another sound will be played to indicate that the focus didn't change.

For interactions recognized by the `UA11YInput`, like selecting the focused element, the screen reader simply forwards this event to the `UA11YElement Component`. Likewise, the `UA11YElement` will forward the event to the listeners that have previously subscribed to it, as described in Section 3.2. Additionally, the screen reader will play a fitting sound to indicate the success of the interaction and, in case the `value` of the `UA11YElement` changed, announce just the new `value` via the `UA11YSpeechSynthesizer`.

This is all there is to say about the implementation of the `UA11YCustom-ScreenReader` A simplified class diagram of the previously described parts can be found in Figure 3.12. As with the rest of our description, this diagram only covers the Unity side of the `UA11YSpeechSynthesizer`. We left out the system side because it looks virtually identical to the Unity side. The system part is going to be more interesting for our next topic, the `UA11YNativeScreenReader`.

3.3.3 Native Screen Reader Implementation

The native screen reader allows visually impaired players to interact with Unity using the system screen reader. So most parts of its implementation happen on the system side via a Unity native plug-in[2] and have to be tailored to each platform and its screen reader. We've limited this implementation to the iOS platform, as multi-platform support would go beyond the scope of this thesis.

[2]https://docs.unity3d.com/Manual/NativePlugins.html

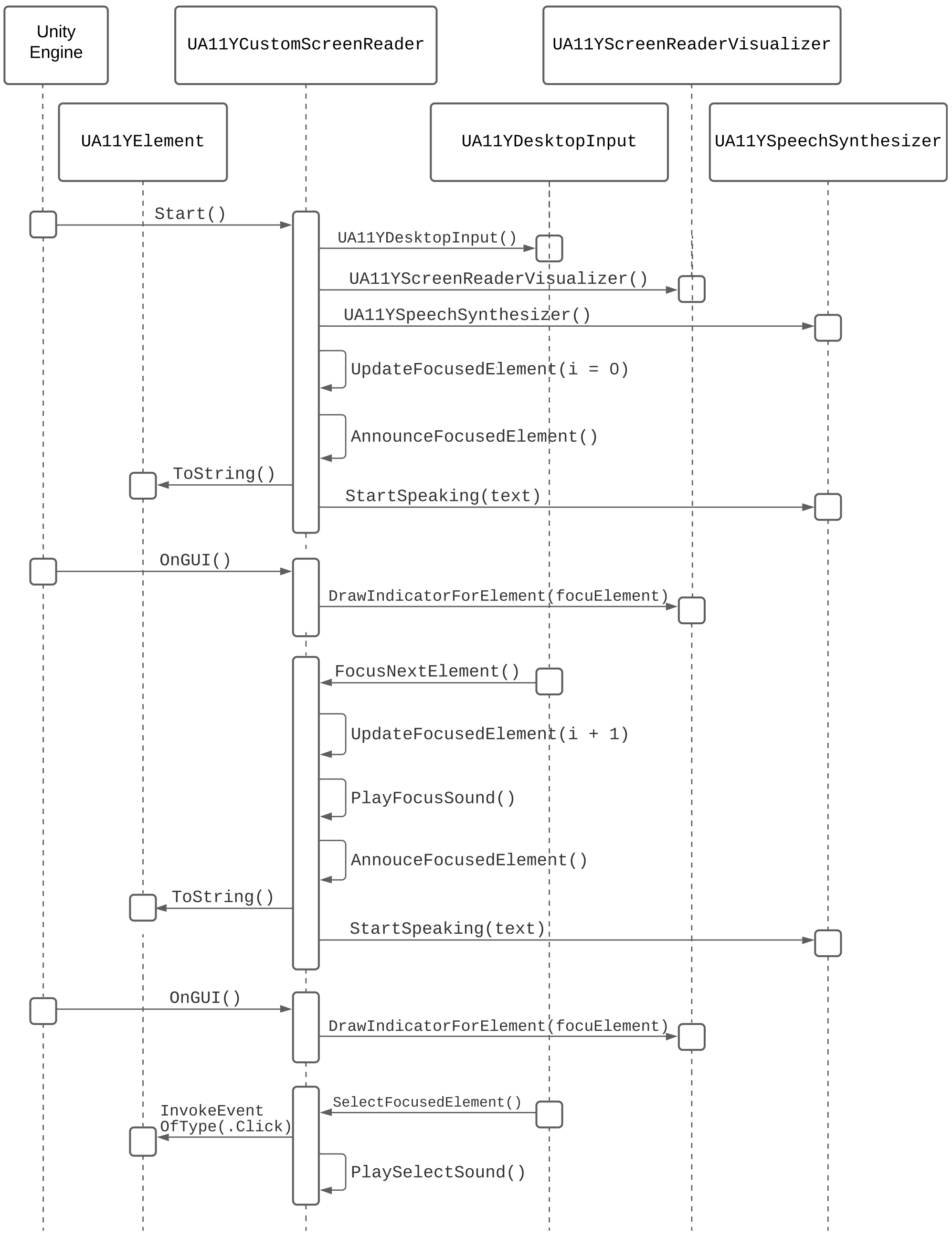
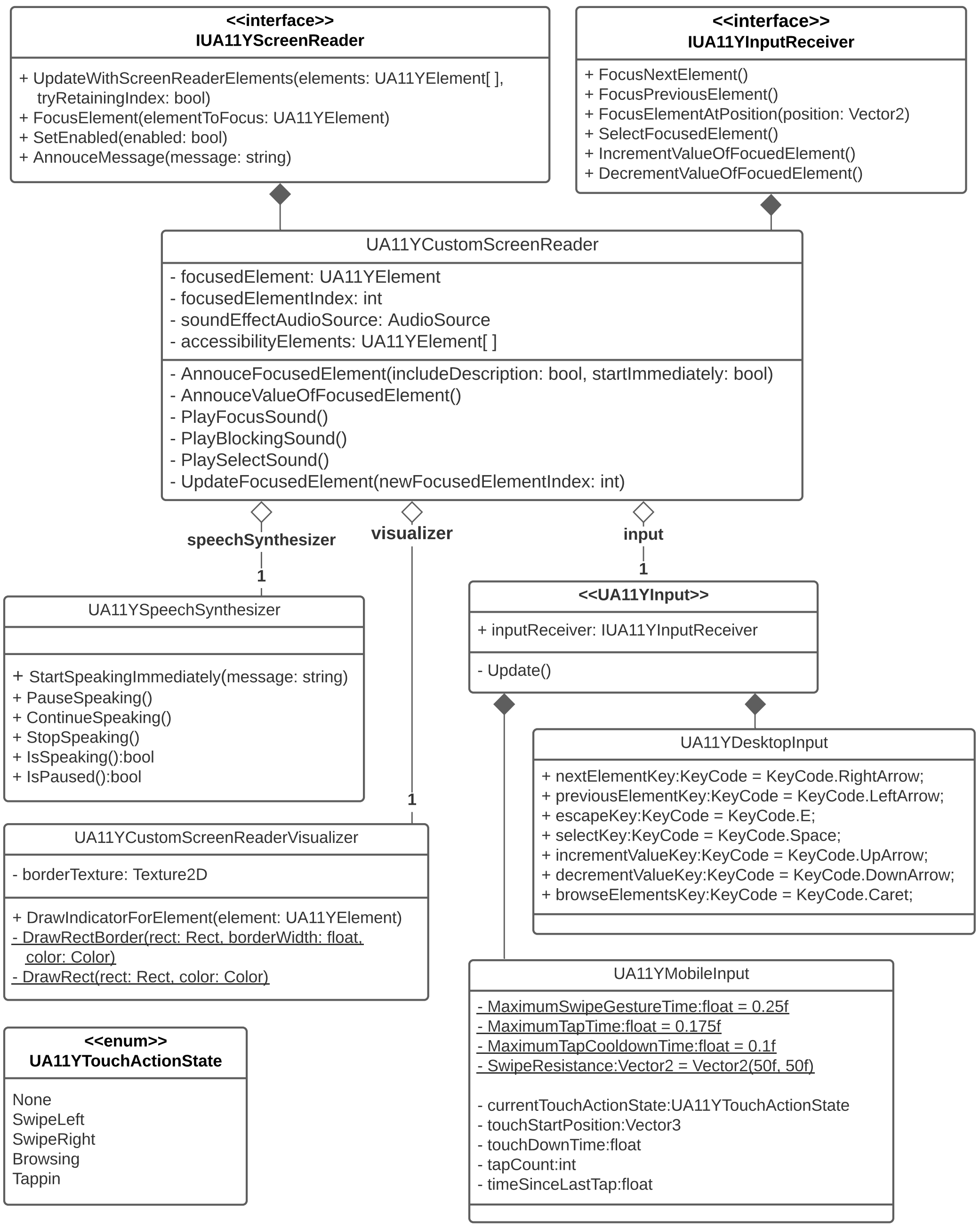


Figure 3.11: Sequence diagram of the `UA11YCustomScreenReader` with an `UA11YDesktopInput`. It covers the initialization, changing the focused `GameObject` and selecting the focused `GameObject`. The `UA11YElement` in the diagram is a placeholder for the dynamically changing focused `GameObject`.



<<interface>>
IUA11YScreenReader
+ UpdateWithScreenReaderElements(elements: UA11YElement[ ], tryRetainingIndex: bool)
+ FocusElement(elementToFocus: UA11YElement)
+ SetEnabled(enabled: bool)
+ AnnouceMessage(message: string)
<<interface>>
IUA11YInputReceiver
+ FocusNextElement()
+ FocusPreviousElement()
+ FocusElementAtPosition(position: Vector2)
+ SelectFocusedElement()
+ IncrementValueOfFocuedElement()
+ DecrementValueOfFocuedElement()
UA11YCustomScreenReader
- focusedElement: UA11YElement
- focusedElementIndex: int
- soundEffectAudioSource: AudioSource
- accessibilityElements: UA11YElement[ ]
- AnnouceFocusedElement(includeDescription: bool, startImmediately: bool)
- AnnouceValueOfFocusedElement()
- PlayFocusSound()
- PlayBlockingSound()
- PlaySelectSound()
- UpdateFocusedElement(newFocusedElementIndex: int)
speechSynthesizer
1
visualizer
1
input
1
UA11YSpeechSynthesizer
+ StartSpeakingImmediately(message: string)
+ PauseSpeaking()
+ ContinueSpeaking()
+ StopSpeaking()
+ IsSpeaking():bool
+ IsPaused():bool
<<UA11YInput>>
+ inputReceiver: IUA11YInputReceiver
- Update()
UA11YDesktopInput
+ nextElementKey:KeyCode = KeyCode.RightArrow;
+ previousElementKey:KeyCode = KeyCode.LeftArrow;
+ escapeKey:KeyCode = KeyCode.E;
+ selectKey:KeyCode = KeyCode.Space;
+ incrementValueKey:KeyCode = KeyCode.UpArrow;
+ decrementValueKey:KeyCode = KeyCode.DownArrow;
+ browseElementsKey:KeyCode = KeyCode.Caret;
UA11YCustomScreenReaderVisualizer
- borderTexture: Texture2D
+ DrawIndicatorForElement(element: UA11YElement)
- DrawRectBorder(rect: Rect, borderWidth: float, color: Color)
- DrawRect(rect: Rect, color: Color)
UA11YMobileInput
- MaximumSwipeGestureTime:float = 0.25f
- MaximumTapTime:float = 0.175f
- MaximumTapCooldownTime:float = 0.1f
- SwipeResistance:Vector2 = Vector2(50f, 50f)
- currentTouchActionState:UA11YTouchActionState
- touchStartPosition:Vector3
- touchDownTime:float
- tapCount:int
- timeSinceLastTap:float
<<enum>>
UA11YTouchActionState
None
SwipeLeft
SwipeRight
Browsing
Tappin

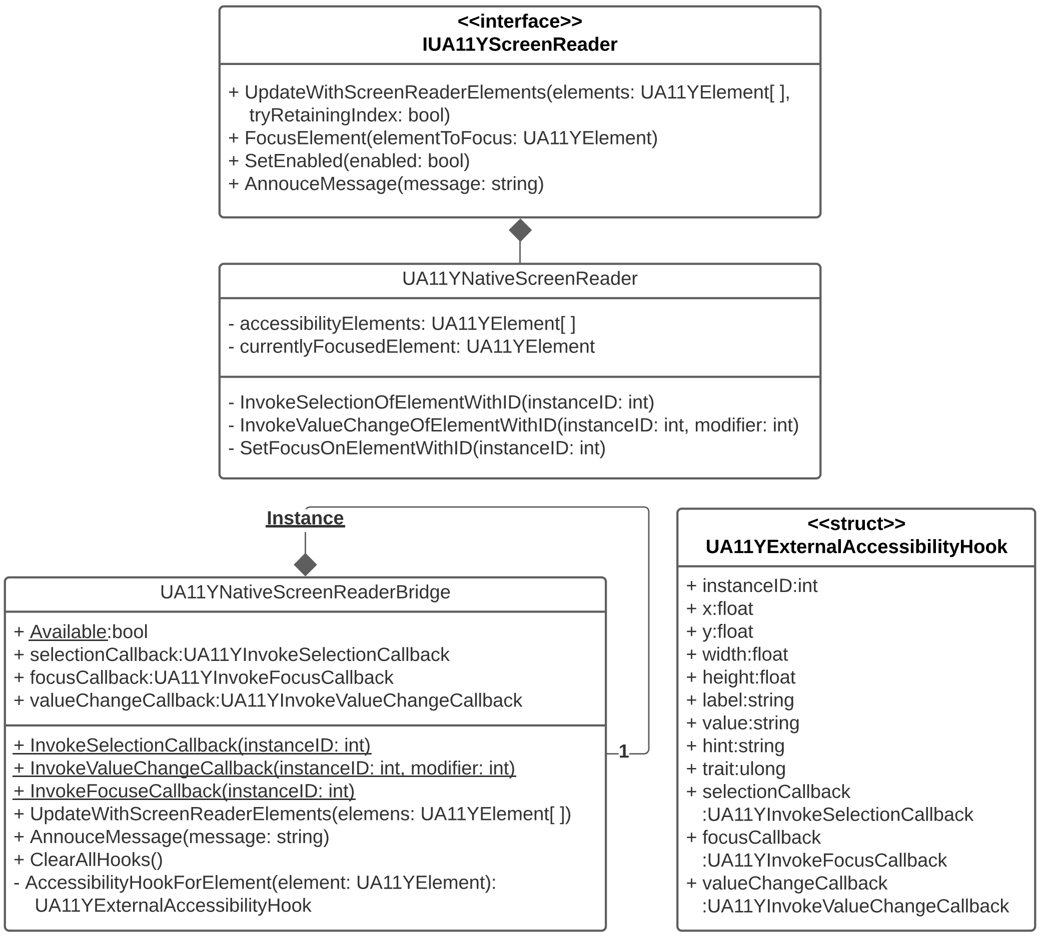
Figure 3.12: Simplified class diagram of the Unity side of the `UA11YCustomScreenReader`

Lets start with the Unity side of the implementation. The only two classes needed there are the `UA11YNativeScreenReader` and the `UA11YNativeScreen-ReaderBridge`, both can be seen in Figure 3.13. The `UA11YNativeScreen-Reader` handles the communication with the `UA11YScreenReaderManager` and the `UA11YNativeScreenReaderBridge`, but also stores the current `acces-sibilityElements` and forwards events to them. The `UA11YNativeScreen-ReaderBridge` is a singleton that forwards information about the accessible elements on the screen to the system and reroutes callbacks from the system. These callbacks are the reason why the `UA11YNativeScreenReaderBridge` has to be a singleton, because they have to be static methods within Unity.

On the iOS side of the implementation, we have the `UA11YVoiceOverPipe`. This C-interface is accessible from Unity, so the `UA11YNativeScreenReader-Bridge` calls its methods to send information about the accessible elements to iOS. The `UA11YVoiceOverPipe` converts the information to the correct format and forwards it, with a few intermediate steps, to the `UA11YVoiceOverHookOverlaySKScene`. All classes involved can be seen in Figure 3.14.

The `UA11YVoiceOverHookOverlaySKScene` is actually responsible to translate the information about each accessible elements in Unity into an iOS view, called the `UA11YVoiceOverHookSKNode`. These views draw an invisible background so that no game content is visually blocked. While this makes them hidden to the eye, they are clearly visible to the VoiceOver screen reader. The user can step through these views as if the game was built with native elements. Interactions triggered from VoiceOver are routed back from the `UA11YVoiceOverHookSKNode` through multiple intermediate steps to the `UA11YNativeScreenReader`, where it forwards the events to the `UA11YElement Component` in the same fashion as we've discussed it for the `UA11YCustomScreenReader`.

This whole translation from Unity content into screen reader perceivable elements is, as we've previously mentioned, resource-heavy. It should be triggered as sparsely as possible, otherwise, it might affect the game's performance and thus, the user enjoyment of the game. But if implemented correctly by the developer, we believe that this screen reader gives the best possible experience to the user.



<<interface>>
IUA11YScreenReader
+ UpdateWithScreenReaderElements(elements: UA11YElement[ ], tryRetainingIndex: bool)
+ FocusElement(elementToFocus: UA11YElement)
+ SetEnabled(enabled: bool)
+ AnnouceMessage(message: string)
UA11YNativeScreenReader
- accessibilityElements: UA11YElement[ ]
- currentlyFocusedElement: UA11YElement
- InvokeSelectionOfElementWithID(instanceID: int)
- InvokeValueChangeOfElementWithID(instanceID: int, modifier: int)
- SetFocusOnElementWithID(instanceID: int)
Instance
1
UA11YNativeScreenReaderBridge
+ Available:bool
+ selectionCallback:UA11YInvokeSelectionCallback
+ focusCallback:UA11YInvokeFocusCallback
+ valueChangeCallback:UA11YInvokeValueChangeCallback
+ InvokeSelectionCallback(instanceID: int)
+ InvokeValueChangeCallback(instanceID: int, modifier: int)
+ InvokeFocuseCallback(instanceID: int)
+ UpdateWithScreenReaderElements(elemens: UA11YElement[ ])
+ AnnouceMessage(message: string)
+ ClearAllHooks()
- AccessibilityHookForElement(element: UA11YElement): UA11YExternalAccessibilityHook
<<struct>>
UA11YExternalAccessibilityHook
+ instanceID:int
+ x:float
+ y:float
+ width:float
+ height:float
+ label:string
+ value:string
+ hint:string
+ trait:ulong
+ selectionCallback :UA11YInvokeSelectionCallback
+ focusCallback :UA11YInvokeFocusCallback
+ valueChangeCallback :UA11YInvokeValueChangeCallback

Figure 3.13: Simplified Class Diagram of the Unity side of the *Native Screen Reader*

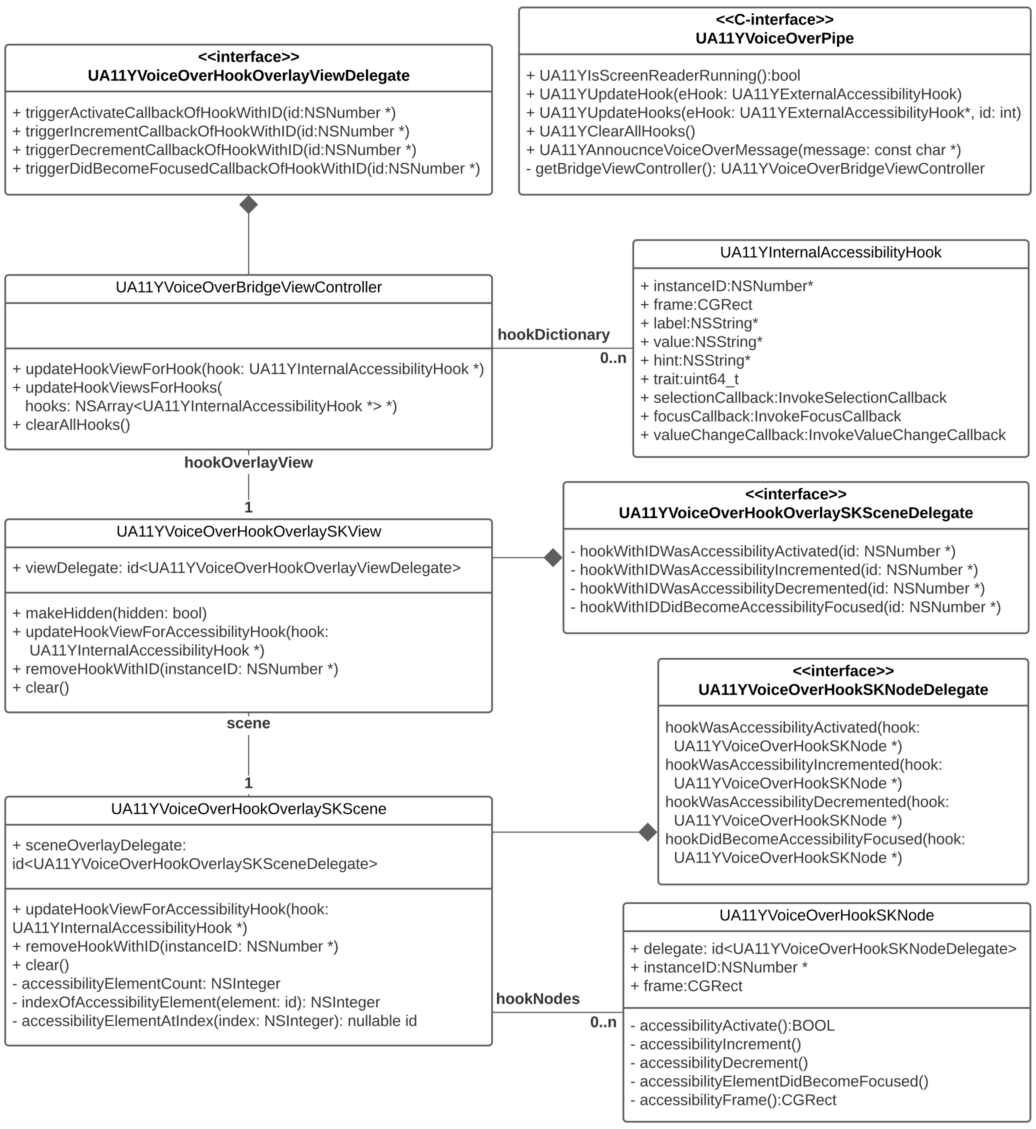

<<interface>>
UA11YVoiceOverHookOverlayViewDelegate
+ triggerActivateCallbackOfHookWithID(id:NSNumber *)
+ triggerIncrementCallbackOfHookWithID(id:NSNumber *)
+ triggerDecrementCallbackOfHookWithID(id:NSNumber *)
+ triggerDidBecomeFocusedCallbackOfHookWithID(id:NSNumber *)
<<C-interface>>
UA11YVoiceOverPipe
+ UA11YIsScreenReaderRunning():bool
+ UA11YUpdateHook(eHook: UA11YExternalAccessibilityHook)
+ UA11YUpdateHooks(eHook: UA11YExternalAccessibilityHook*, id: int)
+ UA11YClearAllHooks()
+ UA11YAnnoucnceVoiceOverMessage(message: const char *)
- getBridgeViewController(): UA11YVoiceOverBridgeViewController
UA11YInternalAccessibilityHook
+ instanceID:NSNumber*
+ frame:CGRect
+ label:NSString*
+ value:NSString*
+ hint:NSString*
+ trait:uint64_t
+ selectionCallback:InvokeSelectionCallback
+ focusCallback:InvokeFocusCallback
+ valueChangeCallback:InvokeValueChangeCallback
UA11YVoiceOverBridgeViewController
+ updateHookViewForHook(hook: UA11YInternalAccessibilityHook *)
+ updateHookViewsForHooks(
hooks: NSArray<UA11YInternalAccessibilityHook *> *)
+ clearAllHooks()
hookDictionary
0..n
hookOverlayView
1
UA11YVoiceOverHookOverlaySKView
+ viewDelegate: id<UA11YVoiceOverHookOverlayViewDelegate>
+ makeHidden(hidden: bool)
+ updateHookViewForAccessibilityHook(hook:
UA11YInternalAccessibilityHook *)
+ removeHookWithID(instanceID: NSNumber *)
+ clear()
<<interface>>
UA11YVoiceOverHookOverlaySKSceneDelegate
- hookWithIDWasAccessibilityActivated(id: NSNumber *)
- hookWithIDWasAccessibilityIncremented(id: NSNumber *)
- hookWithIDWasAccessibilityDecremented(id: NSNumber *)
- hookWithIDDidBecomeAccessibilityFocused(id: NSNumber *)
scene
1
<<interface>>
UA11YVoiceOverHookSKNodeDelegate
hookWasAccessibilityActivated(hook:
UA11YVoiceOverHookSKNode *)
hookWasAccessibilityIncremented(hook:
UA11YVoiceOverHookSKNode *)
hookWasAccessibilityDecremented(hook:
UA11YVoiceOverHookSKNode *)
hookDidBecomeAccessibilityFocused(hook:
UA11YVoiceOverHookSKNode *)
UA11YVoiceOverHookOverlaySKScene
+ sceneOverlayDelegate:
id<UA11YVoiceOverHookOverlaySKSceneDelegate>
+ updateHookViewForAccessibilityHook(hook:
UA11YInternalAccessibilityHook *)
+ removeHookWithID(instanceID: NSNumber *)
+ clear()
- accessibilityElementCount: NSInteger
- indexOfAccessibilityElement(element: id): NSInteger
- accessibilityElementAtIndex(index: NSInteger): nullable id
hookNodes
0..n
UA11YVoiceOverHookSKNode
+ delegate: id<UA11YVoiceOverHookSKNodeDelegate>
+ instanceID:NSNumber *
+ frame:CGRect
- accessibilityActivate():BOOL
- accessibilityIncrement()
- accessibilityDecrement()
- accessibilityElementDidBecomeFocused()
- accessibilityFrame():CGRect

Figure 3.14: Simplified Class Diagram of the iOS side of the *Native Screen Reader*

What a screen reader cannot do well is to assist with continuous navigation. For this purpose, we've created the navigation agent. Its design will be the topic of the next section.

3.4 Implementation of the Navigation Agent

The purpose of the navigation agent is to help users with vision impairments find certain points of interest in a game world in which they can move freely. As we've discussed in the design part of this chapter, we help with this navigation by calculating a path from the point of interest to the user and guide the user along this path with an agent that emits earcons.

For the implementation of this agent, we take advantage of the *Navigation and Pathfinding*[3] system of Unity. This system allows developers to create non-player characters that intelligently moves around the games environment. The system first needs a walkable area, which is a space in a Unity scene that a character can walk on (Unity Technologies, 2020c). This area is calculated from the environment of the game and stored as a data structure called *NavMesh*. *NavMesh Baking* is the name of the process to create such a data structure. For this process to work, the developer has to mark all static elements, like floors, walls, or other static obstacles in a scene as *Navigation Static* through the navigation window (Unity Technologies, 2020a). After that, the developer can navigate to the ake-tab in the same window and click on *Bake* to calculate the NavMesh.

Having such a NavMesh setup is the first precondition for our navigation agent. The second is about the actual player character. For our navigation agent to work, the developer needs to attach an `AudioListener Component` to the player object. Only this can ensure that the spatial audio emitted from the navigation agent will be retrieved relative to the players' position. Additionally, the player object also needs a `Collider` and a `Rigidbody Component` so that we can detect when the player has reached the agent.

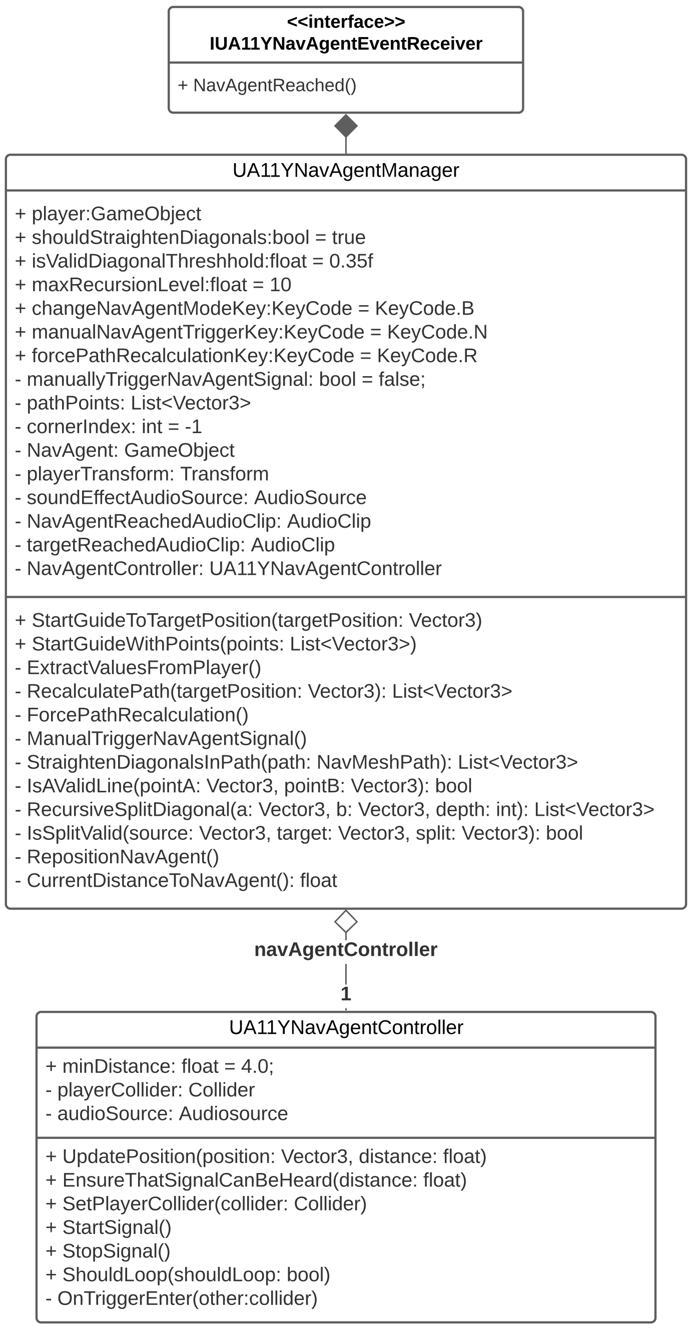
With all of this set up, we can talk about the actual navigation agent. The two classes involved here are the `UA11YNavigationAgentManager` and

[3]https://docs.unity3d.com/Manual/Navigation.html

UA11YNavigationAgentController, both of which can be seen in the class diagram in Figure 3.15. The UA11YNavigationAgentManager is responsible for triggering the calculation of the path to a destination location and for redepositing the actual navigation agent. The UA11YNavigationAgentController is the Component attached to the navigation agent object, the NavAgent, and responsible for emitting the earcons. Similar to the UA11YScreenReaderManager, we also provide a prefab of an GameObject with the UA11YNavigationAgentManager attached, which the developer can simply drag into the scene. But no matter how a UA11YNavigationAgentManager is added to the scene, the developer must in any case set its player property before requesting a guide. The procedure of guiding a player to a requested location can be seen in the sequence diagram in Figure 3.16. A textual description is given in the following.

When the player wants to be guided to a certain point of interest, the developer has to call the StartGuideToTargetPosition(p) method, where p is a Vector3 of a reachable position within the NavMesh. This will trigger the internal RecalculatePath, which will use the NavMesh.CalculatePath() method to create a NavMeshPath from the player position to the given point. The NavMeshPath includes orthogonal and diagonal subsections. Since we want to make the guidance as easy as possible, we try to straighten the diagonals by calling the StraightenDiagonalsInPath method. This method recursively splits a diagonal into orthogonal lines that fully lie in the walkable area. Listing 1 shows the pseudo code of this algorithm.

After we have removed the diagonals from the NavMeshPath, we move the NavAgent to the first point of the path, set it to active and start the audio signal by calling StartSignal. Depending on the mode, this will either start a continuous audio signal or allow the user to manually trigger the earcon by hitting the manualNavAgentTriggerKey key. Since the audio is spatial, the user can now find the NavAgent by listening to the direction and the volume of the audio signal.



<<interface>>
IUA11YNavAgentEventReceiver
+ NavAgentReached()
UA11YNavAgentManager
+ player:GameObject
+ shouldStraightenDiagonals:bool = true
+ isValidDiagonalThreshhold:float = 0.35f
+ maxRecursionLevel:float = 10
+ changeNavAgentModeKey:KeyCode = KeyCode.B
+ manualNavAgentTriggerKey:KeyCode = KeyCode.N
+ forcePathRecalculationKey:KeyCode = KeyCode.R
- manuallyTriggerNavAgentSignal: bool = false;
- pathPoints: List<Vector3>
- cornerIndex: int = -1
- NavAgent: GameObject
- playerTransform: Transform
- soundEffectAudioSource: AudioSource
- NavAgentReachedAudioClip: AudioClip
- targetReachedAudioClip: AudioClip
- NavAgentController: UA11YNavAgentController
+ StartGuideToTargetPosition(targetPosition: Vector3)
+ StartGuideWithPoints(points: List<Vector3>)
- ExtractValuesFromPlayer()
- RecalculatePath(targetPosition: Vector3): List<Vector3>
- ForcePathRecalculation()
- ManualTriggerNavAgentSignal()
- StraightenDiagonalsInPath(path: NavMeshPath): List<Vector3>
- IsAValidLine(pointA: Vector3, pointB: Vector3): bool
- RecursiveSplitDiagonal(a: Vector3, b: Vector3, depth: int): List<Vector3>
- IsSplitValid(source: Vector3, target: Vector3, split: Vector3): bool
- RepositionNavAgent()
- CurrentDistanceToNavAgent(): float
navAgentController
1
UA11YNavAgentController
+ minDistance: float = 4.0;
- playerCollider: Collider
- audioSource: Audiosource
+ UpdatePosition(position: Vector3, distance: float)
+ EnsureThatSignalCanBeHeard(distance: float)
+ SetPlayerCollider(collider: Collider)
+ StartSignal()
+ StopSignal()
+ ShouldLoop(shouldLoop: bool)
- OnTriggerEnter(other:collider)

Figure 3.15: Simplified class diagram of the classes involved in our navigation agent

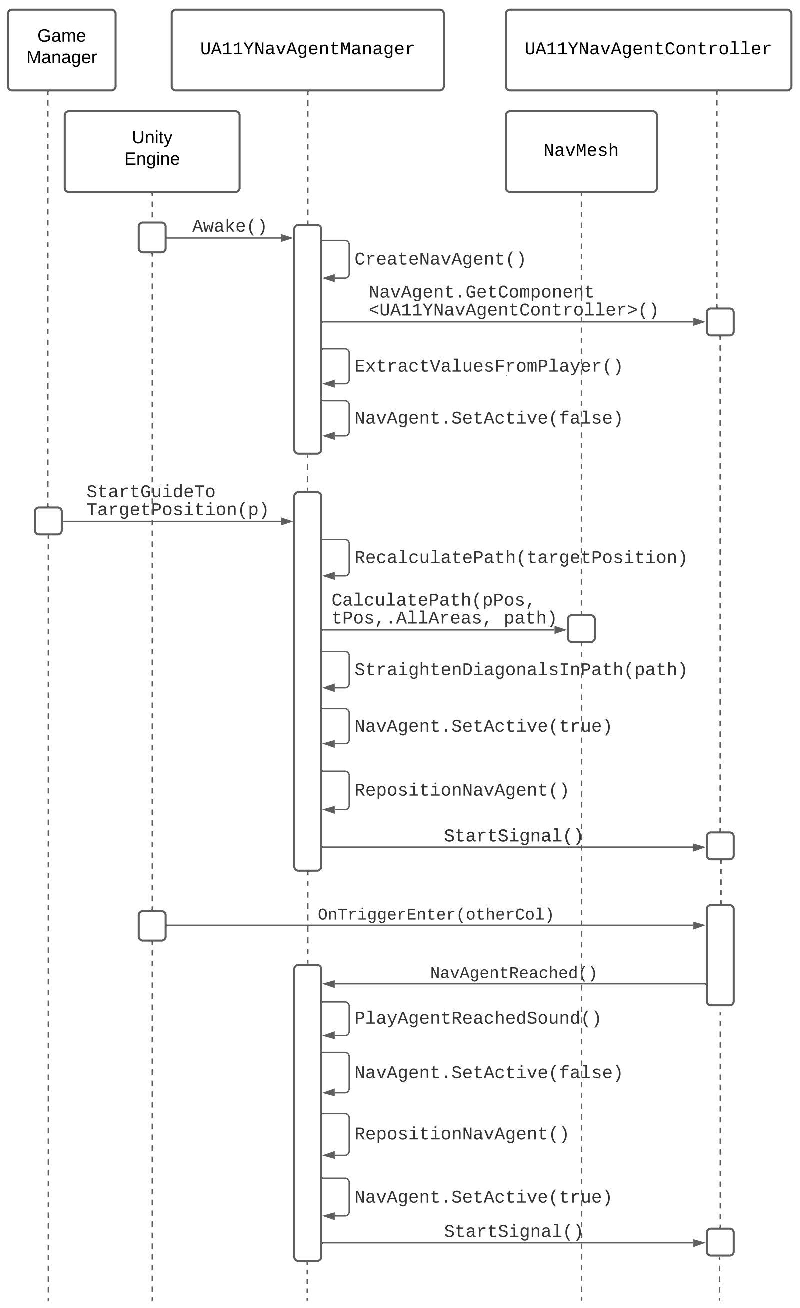


Figure 3.16: Sequence diagram of the navigation agent. The *Game Manger* is a placeholder for a arbitrary element of a game that will request the guidance to a certain point.

```
Result: Array of points that can be connected with orthogonal lines
Function SplitDiagonal(Point pointA, Point pointB, int depth) is
    points = [];
    if IsOrthogonalLine(pointA, pointB) then
        pointAxBy = Point(pointA.x, pointB.y) ;
        pointBxAy = Point(pointB.x, pointA.y) ;
        if NoObstaclesBetweenLines(pointA, pointAxBy, pointB) then
            points += pointAxBy;
        else if NoObstaclesBetweenLines(pointA, pointBxAy, pointB) then
            points += pointBxAy;
        else if depth ¡ maximumRecurrsion then
            center = CenterOfDiagonal(pointA, pointB) ;
            points += SplitDiagonal(pointA, center, depth + 1) ;
            points += center ;
            points += SplitDiagonal(center, pointB, depth + 1) ;
    end
    return points;
end
```

Listing 1: Pseudo code of the `SplitDiagonal` algorithm

When the `NavAgent` is actually reached, the `UA11YNavigationAgentManager` gets a callback from the `UA11YNavigationAgentController`. This will reposition the `NavAgent` to the next point and emit an earcon to indicate this event. The procedure of searching for the `NavAgent` and repositioning it when it has been found repeats until the point of interest has been reached. When this happens, the `NavAgent` will be deactivated, and another earcon will be played.

How the navigation agent can look in a game is shown in figure 3.17. The game seen there is a very simple labyrinth based on a sample project by Brackeys[4].

As we've mentioned in the design section of this chapter, the user or developer can also manually trigger a recalculation of the path by hitting the `forcePathRecalculationKey`. Through this, we can ensure that the user can

[4]https://github.com/Brackeys/NavMesh-Tutorial

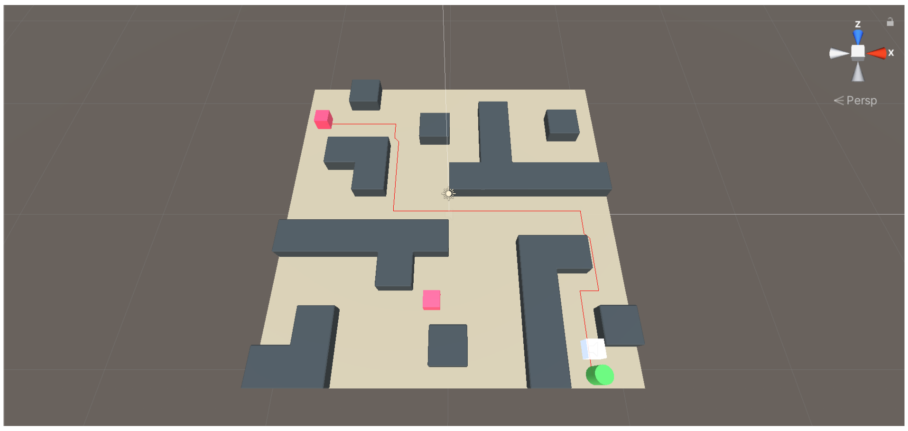


Figure 3.17: Navigation agent in a test game

always be guided to the destination location, even if they unintentionally move away from the agent.

3.5 Possible Applications of the Toolkit

At the beginning of this chapter, we've mentioned that we want to concentrate on making four different areas accessible: menu navigation, environment observation, continuous navigation and discrete navigation of a game world. To show how our toolkit can help in each of these areas, we will present some concrete examples in the following.

3.5.1 Menu Navigation

Menu navigation is about making game menus accessible through a screen reader. As a basis for this demonstration, we've used the FPS Microgame[5] project provided by Unity as a learning example. To make menus like the Pause/Options menu of this game accessible, we had to add the

[5]https://assetstore.unity.com/packages/templates/fps-microgame-156015

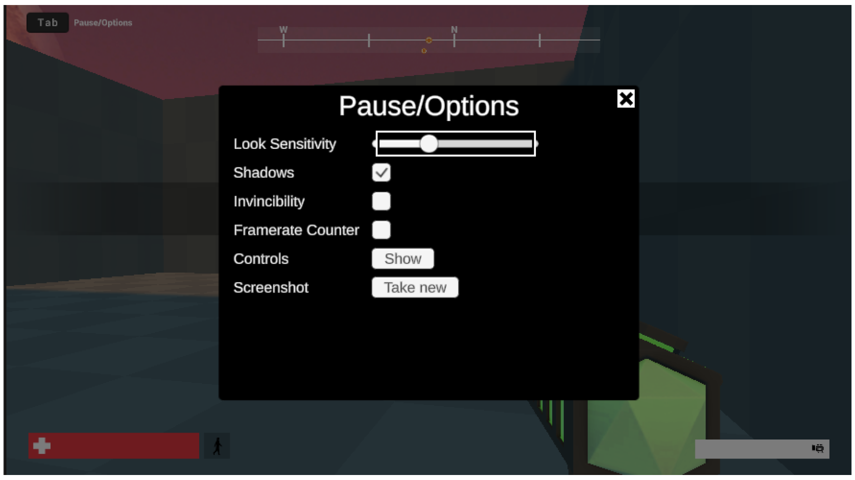


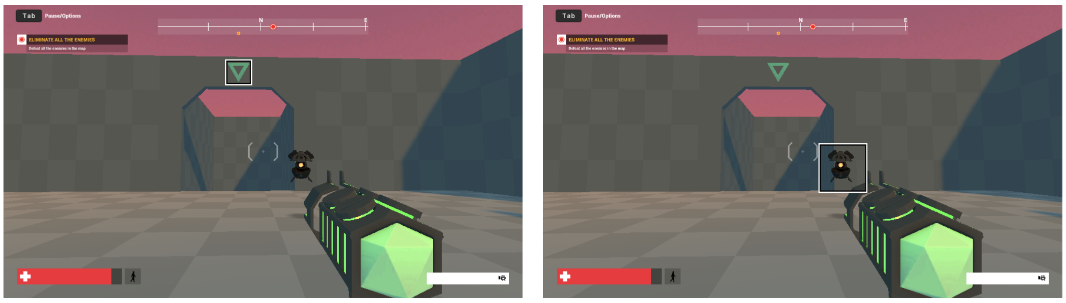
Figure 3.18: Menu navigation in the Pause/Options menu of the FPS Microgame. The highlighted element is the Look Sensitivity slider.

`UA11YSlider`, `UA11YToggle`, `UA11YButton`, `UA11YText` and `UA11YPopover Components` to the corresponding Unity UI elements. We also needed to set the `label` property for all of them, with the exception of the `UA11YText`, manually, because an understandable label couldn't be inferred from the `GameObject`. We also had to add the `UA11YScreenReaderManager` prefab to the scene, which is responsible for creating the screen reader, finding the accessible objects in the scene, and handing them to the screen reader.

Now when the menu visibility changes, we had to activate or deactivate the `UA11YScreenReaderManager` and call its `VisibleElementsDidChange()` method. After that, we were able to use the screen reader to step through the UI, get an auditory description of the highlighted element and interact with each element. Figure 3.18 shows how this menu with the activated screen reader looks like.

3.5.2 Environment Observation

The idea of environment observation is to let the player pause the game and observe their surroundings via the screen reader by either stepping through the visible objects or browsing the elements with the mouse cursor. To



(a) Door indicator (b) Enemy robot

Figure 3.19: Getting information about the visible objects via environment observation.

demonstrate this, we again used the FPS Microgame from Unity as a basis. For this to work, we had to add the `UA11YElement Component` to important objects, like enemies or door markers, label them and made sure that every element had an `Collider Component` attached. The latter was necessary so that we can exclude objects that are not visible to the player through ray casting[6]. Finally, we needed to implement a way to pause the game through a key press, activate the `UA11YScreenReaderManager` and call its `VisibleElementsDidChange()`. After that, we could step through the visible accessible elements on the screen, as can be seen in Figure 3.19.

What we've just described is only a trivial implementation of the environment observation. In a real game, developers could include information about the current on-screen location of each object in their accessibility description, allow to lock the shooting cursor on an object, or start a navigation guide to it. The latter could make use of our navigation agent, which we will describe more in the next subsection.

3.5.3 Continuous Navigation

With continuous navigation we mean navigating in a 2D or 3D space without any constraints. An example would be to find a certain location or non-player character within an open-world game. To make this kind of navigation accessible, we've created the navigation agent. See Section 3.4 for more

[6]https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

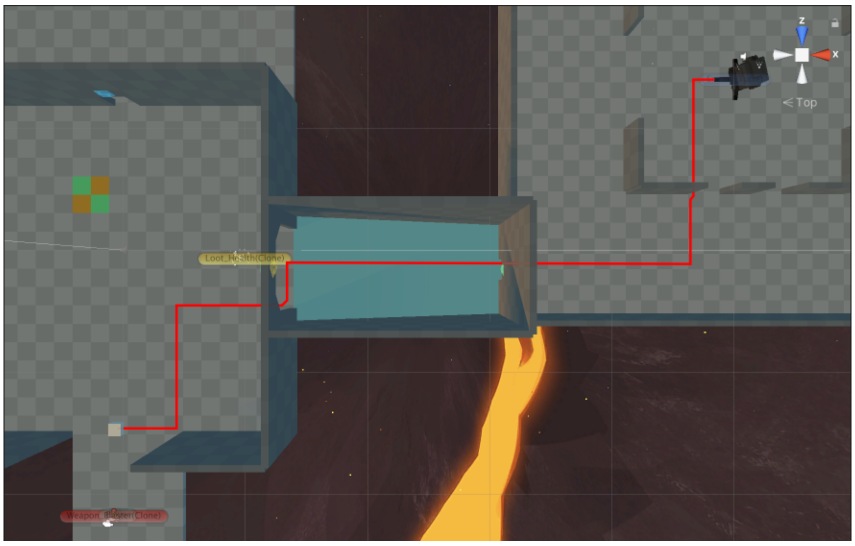


Figure 3.20: The navigation path calculated by our navigation agent, going from the players' position to the turret enemy.

details. To demonstrate the agent in practice, we turned one last time to the FPS Microgame.

This game features a main enemy, the turret, and we wanted to get an auditory guide to this enemy via the navigation agent. All we had to do was to add the `UA11YNavAgentManager` to the scene and on a keypress, retrieve the enemy turret `GameObject` and start the navigation guide by calling `UA11YNavAgentManager.Instance.StartGuideToTargetPosition()`. This triggered a calculation of a navigation path from the players' position to the turret and starts the audio navigation along this path, as described in Section 3.4. The path can be seen in Figure 3.20.

3.5.4 Discrete Navigation

The last area we wanted to address is discrete navigation, which is navigating on a fixed grid with the screen reader. Since the FPS Microgame features Continuous Navigation, we had to use another sample project to

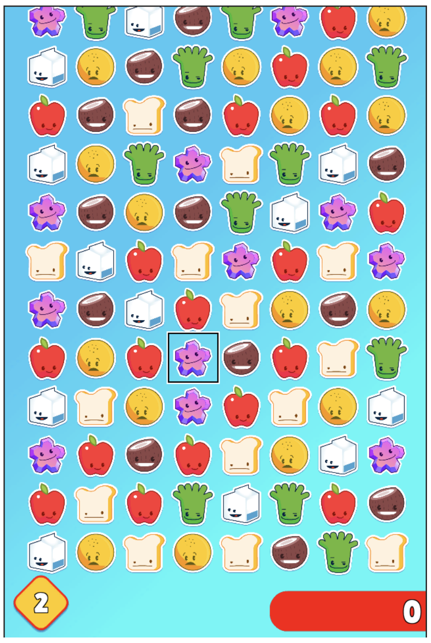


Figure 3.21: Navigating the match-3 puzzle game with out screen reader.

demonstrate this. The game we chose comes from a RayWenderlich tutorial[7] and is a match-3 puzzle game, similar to Bejewled (PopCap Games, 2001).

To make the game playable to someone with a vision impairment, we had to attach a `UA11YElement Component` to each game tile and label them with their type. We also included the location of each item in the grid to make it playable with sequential navigation, though browsing is a much quicker and effective way to play this game. We also had to subscribe to the `Click UA11YElementInteractionEventType` to trigger the internal method for selecting a tile. Lastly, the screen reader needed to be kept up to date by calling `UA11YScreenReaderManager VisibleElementsDidChange()` every time a match has been made. With this last addition, the game board was fully observable and interactive through the screen reader, as can be seen in Figure 3.21.

[7]https://www.raywenderlich.com/673-how-to-make-a-match-3-game-in-unity

3.6 Summary

In this chapter, we went over the design and implementation of the Unity Accessibility Toolkit, a Unity asset package that helps making accessible games for visually impaired gamers. The package comes with two major parts: a screen reader and a navigation agent. As a basis for both of them, we introduced an accessibility signifier. This signifier marks an element as accessible for our accessibility systems and holds information like the label or the value of the element. We then discussed the implementation of this signifier as the `UA11YElement Component`, which can be added to any Unity `GameObject`. Other classes can register themselves to a certain `UA11YElement` to get callbacks for different events, like when the `value` of the element changes or when a selection is invoked.

After discussing the signifier, we continued by sketching out the inner workings of our screen reader. The core of this system is the screen reader manager, implemented as the `UA11YScreenReaderManager Component` in Unity. It is responsible for fetching the accessible elements within a Unity scene and forwarding it either the native or the custom screen reader.

The custom screen reader is a screen reader built from the ground up in Unity as the `UA11YCustomScreenReader Component`. We've shown that this screen reader gives access to the accessible elements in a serial manner and how the user can change the currently focused element. We've also discussed the connection from Unity to the system that is used to convert the properties of the `UA11YElement` to a speech when the focused element changes. Far stronger tied to the system is the native screen reader, which is just a bridge to a system screen reader. We've discussed its implementation for iOS to let the user interact with the Unity game through the VoiceOver screen reader.

We then discussed the navigation agent, a tool to guide a person to any reachable point of interest through audio cues. We went over its implementation and showed that the path calculation takes advantage of the navigation and pathfinding system of Unity, which is already commonly used in many games.

Finally, we did present concrete examples of how the toolkit can be used to make games menu navigation, environment observation, continuous navigation, and discrete navigation accessible in games.

These concrete examples gave a glimpse at how easy it is to use the toolkit is if one is familiar with it, but they are not a good measure for how easy it is to use for someone who has no experience with the toolkit. To find out how newcomers cope with the toolkit, we made an evaluation. This evaluation will be presented in the next chapter.

4 Evaluation

The Unity Accessibility Toolkit was built to make adding vision accessibility features easier to any Unity game. To find out how effective the toolkit is, we wanted to evaluate its usability. In the following, we will present this evaluation and its results.

4.1 Procedure

The focus of the evaluation was set on the screen reader because it is both a familiar tool to visually impaired users and can be used in many different game scenarios. So we asked developers familiar with the Unity game engine to make a simple match-3 puzzle game accessible with the screen reader of our toolkit.

The evaluation was conducted remotely in two phases: An implementation phase (I) and a post-questionnaire (II). As preparation, participants were asked to install Unity version 2020.1.10f1 and a screen recording tool of their choosing. No hard time limit was set, but the participants were told to expect it to take at least 2 hours. Each of them was compensated with a 10€steam gift card for their participation.

We connected with the participants via Discord and used it to sent them the project and a task description. Participants were requested to check if the project did open correctly and afterward, start the screen recording. This marks the start of the implementation phase, where the participants were advised to read the task description and solve the task. To simulate a somewhat realistic scenario, we asked participants to solve the task without our intervention. We were still reachable during the whole phase in case some serious problem would occur. To conclude the first phase, participants were asked to contact us when they were done and send us their implementation and screen recording. Being done was determined by the participant playing the game with closed eyes.

We need to point out here that just being able to play a game with closed eyes as a sighted participant is not a sufficient way to determine vision accessibility. The sighted participant already has a mental model of the game and thus will navigate and interact with the game differently from a visually impaired person unfamiliar with the game. So we as domain experts compared the participants' solution with a reference implementation.

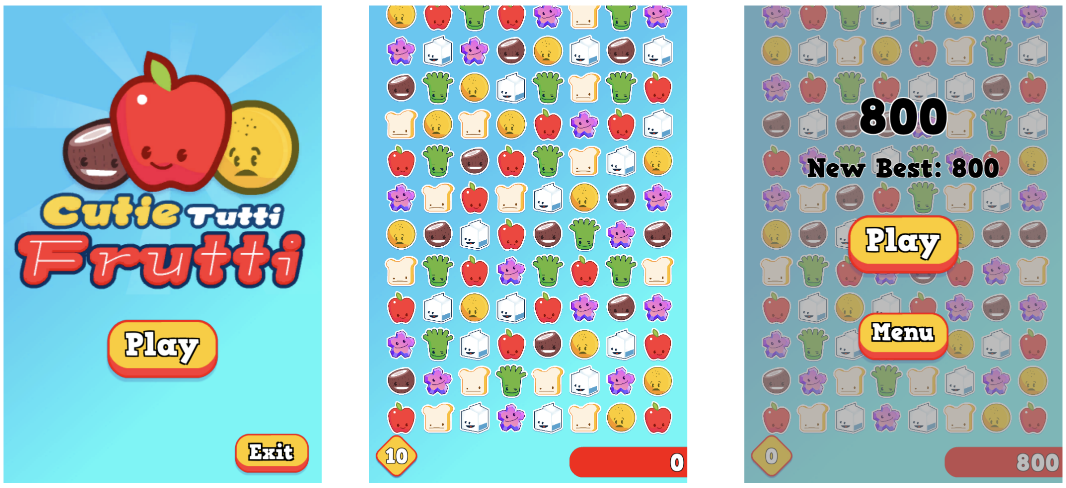
In the second phase, each participant was given a short post-questionnaire to assess their experience with the toolkit. We also asked them about their experience with accessibility tools before the evaluation.

4.2 Participants

9 developers (8 male) participated in the study, with most of them being between the ages 21-30. Table 4.1 gives an overview of all the participants. As can be seen there, every participant was familiar with the Unity game engine and had multiple years of programming experience. One of the participants uses Unity professionally.

Participant	Age Group	Programming Experience	Unity Experience	Uses Unity Professional
P1	21-30	5-6 years	less than 1 year	no
P2	21-30	5-6 years	3-4 years	no
P3	21-30	3-4 years	1-2 years	no
P4	21-30	5-6 years	1-2 years	no
P5	21-30	6+ years	3-4 years	no
P6	21-30	6+ years	1-2 years	no
P7	31-40	6+ years	5-6 years	yes
P8	21-30	6+ years	1-2 years	no
P9	31-40	1-2 years	3-4 years	no

Table 4.1: Demography of the evaluation participants



(a) Start screen (b) Game screen (c) Result screen

Figure 4.1: The three screens of the match-3 sample game.

In addition to these 9 developers, we also had one (male) pilot tester. His results are not included here because we did alter the evaluation process and post-questionnaire slightly after his participation.

4.3 Materials

For the evaluation, we used the same sample game as for demonstrating discrete navigation in Section 3.5. This project was taken from a RayWenderlich tutorial[1] and consists of three screens. These screens can be seen in Figure 4.1. The project sent to the participants was a slightly modified version of this Unity game project, which included the UA11Y toolkit and added a convenience method to retrieve the name of a game tile. The exact project is included in the materials appendix on the CD.

Additionally to the Unity project, each participant also received a task description. It included an overview of the screen reader part of the UA11Y toolkit, the exact task split into smaller subtasks, and a list of tips. These tips

[1]https://www.raywenderlich.com/673-how-to-make-a-match-3-game-in-unity

did not give away the solution to the task but helped the users navigate the project more efficiently to keep the evaluation in a reasonable time frame.

The questionnaire that was given to the participants after the implementation was finished consisted of three parts. The first part included 5 demographic questions, the second 10 questions about making the sample project accessible and interacting with a screen reader, and the last part 11 questions about vision accessibility in general. Finally, there was a possibility to give closing remarks.

4.4 Results

The results presented in the following both consist of our assessment of the participants' solutions and the answers given by the participants in the post-questionnaire. The latter are a combination of multiple choice answers, single choice answers, answers on a Likert scale between 1 (strongly agree) and 5 (strongly disagree), and open-ended and answers.

4.4.1 Accessibility in General

The questionnaire about accessibility in general, shows that most of the participants lack experience with adding accessibility support to software products. Only 3 of the 9 participants worked on the implementation of accessibility features prior to this evaluation, namely on color blindness post-processing filters and button remapping for a Unity game. None of the participants implemented screen reader support for any software before. The reasons for this vary for both participants and for projects they worked on. They range from just not thinking of adding screen reader support (6/9), to believing that the target group for the screen reader support was too small (3/9) and wanting to do adding support but not having the right tools provided by the engine (2/9). Only one participant noted that someone else in the project was responsible for adding support.

We asked participants about their general knowledge of accessibility. The majority (7/9) informed themselves about game accessibility in the past.

These 7 participants did it by watching videos (4/7), attending talks (6/7), reading articles (6/7), and learning about it during a lecture (1/7).

Around half of the participants (4/9) also used accessibility features themselves while playing games. Almost all participants (8/9) already knew what a screen reader is before the evaluation. 5 had used a screen reader before and everyone who did stated that using the custom screen reader of our toolkit felt natural.

4.4.2 Making the Game Accessible

As expected, most of the participants (6/9) needed 90-120min to complete the task. Only one was done earlier, after 60 - 90min, and two participants were only finished after 150-180min. All participants stated that the task was easy to understand (AVG: 4.44 STD: 0.53). They also could find everything they needed easily in the package (AVG: 4.78 STD: 0.44) and, on average, did think that implementing the screen reader support was easy (AVG: 4.00 STD: 1.00). The majority (AVG: 4.67 STD: 0.71) thought that it is important to offer accessibility features in a game. They would also consider making their next game accessible with a screen reader (AVG: 4.33 STD: 0.50), and use our toolkit for this task (AVG: 4.67 STD: 0.50), if applicable.

When asking the participants about what they found particularly easy to implement, everyone pointed to making texts, buttons, and other Unity UI elements accessible. For example, one participant stated that *"All Ui elements and text were basically only drag and drop in Unity, which was nice"*. This fact is also visible in our assessment of their solutions. The UI elements of the start screen, result screen, and game screen were visible and interactive with the screen reader in every solution.

Our assessment also shows where people did struggle the most, namely with making the tiles and actual gameplay accessible. As one participant put it: *"The tile was the only thing where no finished script was provided, hence this took the most effort"*. But before talking about the solutions of the participants, we need to take a step back and talk about the reference implementation here. To make the game tiles perfectly discoverable with both sequential navigation and browsing, the accessibility description of each tile in our reference implementation included its type, the location of the item (row and column), and a hint if it was selected. The selection of an item in this reference implementation could be toggled with the usual selection mechanism of the screen reader.

Looking at the solutions that the participants provided, almost all differ from the reference implementation. Only one participant (P8) did match our reference implementation in both interaction and description of the tiles. One participant (P3) missed updating the description of the tiles after interacting with the game board. This made the game state diverge from what was presented to the user via the screen reader, making the implementation of P3 unplayable to someone with a visual impairment. All other 7 solutions that we got diverged from the reference solution, but were all still playable with closed eyes, as requested from the participants.

One issue was that majority of solutions (7/9) didn't include the location of a tile, making it hard to play with sequential navigation. Another problem with most solutions (6/9) was the selection state. After selecting a tile, it was not possible to determine if the tile is still selected by simply focusing it with the screen reader. The selection state can only be determined by trying to select the tile and listen to the audio cue. Speaking of selection, 4/9 solutions implemented selecting the tiles via a mouse click instead of using the screen reader for interaction like in the reference implementation.

Where exactly the solution of each participant differed from the reference can be seen in Tables 4.3 and 4.2.

4.5 Discussion

We were very pleased with the general reception of the toolkit. *"It was an interesting first contact with integrating accessibility features for visually impaired people in a video game. Looking forward to doing this more often!"*. *"Please keep working on this package and publish it to the asset store. Once you get the hang of it becomes really easy to use and in the end providing screen reader support can make a huge difference for people with vision impairment."*. The participants

Solution	Visible to Screen Reader	Description Includes Type	Description Includes Location	Description Includes Selection	Description Updates
Reference	yes	yes	yes	yes	yes
P1	yes	yes	no	no	yes
P2	yes	yes	no	yes	yes
P3	yes	yes	yes	no	no
P4	yes	yes	no	no	yes
P5	yes	yes	no	yes	yes
P6	yes	yes	no	no	yes
P7	yes	yes	no	no	yes
P8	yes	yes	yes	yes	yes
P9	yes	yes	no	no	yes

Table 4.2: Visibility and accessibility description of the tiles in the different solutions.

overall liked the toolkit and seemed to be inspired to make their games more accessible in the future.

The results of the assessment and the feedback from the post-questionnaire showed that there were no major issues with the package itself. We think that the problems that did arise during the implementation can be at least partially traced back to the lack of experience the participants had. None of them worked on implementing a screen reader before, so it should be no surprise that only one solution matched the reference implementation, even if that result is not ideal. The problems that made the solutions differ from the reference, which was (I) a not detailed enough accessibility description, (II) missing updates of the accessibility description, and (III) interaction through the mouse instead of the screen reader, could have been avoided through a simple feedback loop. So we should try to include a feedback and a refinement step in future evaluations of this package, ideally including a person with visual impairments in this process.

Solution	Interactive through the Screen Reader	Interactive through Mouse	Overall Playable
Reference	yes	no	yes
P1	no	yes	yes
P2	yes	no	yes
P3	no	yes	no
P4	no	yes	yes
P5	no	yes	yes
P6	yes	no	yes
P7	yes	no	yes
P8	yes	no	yes
P9	yes	no	yes

Table 4.3: How interaction was solved in the different solutions and if they are overall playable.

5 Lessons Learned

The following chapter will discuss what we've learned during the theoretical, implementation, and evaluation parts of this thesis. Parts of it will already be a peek into potential future work, which will be discussed in Chapter 6.

5.1 Theory

Game accessibility for people with visual impairments has been a topic in the scientific community for several years now, but most work focuses on experimental solutions and games that do not exist beyond the studies they've been created for. So it was helpful to get insight from other developers on this topic and to have several people in the blind gaming community speak out online, in documentaries, and conference talks about their gaming experiences. These insights showed us that people with visual impairment can play far more games than we initially thought. But most of these games can only be played through exhausting workarounds, like repeatedly listening to audio cues to learn the layout of a racing track, and not thanks to extensive accessibility features. This made it clear to us both how important our work is and that a lot more research into mainstream gaming with visual impairment needs to be done, especially in the light of games like The Last of Us Part II (Naughty Dog, 2020).

5.2 Design & Implementation

During the implementation of the toolkit, we found it particularly useful to have access to different kinds of screen readers. This let us explore what features and properties could work well in the context of game interaction and use that knowledge to design and implement our basic accessibility building block.

For translating this building block to a perceptible form, we've used a speech synthesizer. Going into the project, we imagined searching for some library to do this or build a crude speech synthesizer ourselves. So it was very helpful to find a way to access the systems speech synthesizer on Windows, macOS, and iOS and use it both for the navigation agent and the screen reader. This simplified our work while also providing a familiar voice to the user.

Implementing the native plugin to access the systems speech synthesizer also opened the door for the native screen reader. This allows the user to interact with the Unity game via the VoiceOver screen reader on iOS. Having access to the systems screen reader is something we didn't think was feasible when we started this thesis.

5.3 Evaluation

The evaluation showed us that developers seem to be interested in game accessibility in general, but lack experience with building accessible games and software. So it is important to give them clear documentation and easy-to-use tools to help them get started. In future evaluations, it would also be useful to either have a feedback loop during the evaluation or provide other sample projects to account for their lack of experience. This would probably have prevented non-detailed accessibility descriptions and other smaller problems. But even without experience, users were able to make Unitys' UI elements accessible without any problems, which showed that our simple system for this worked well.

What we didn't see coming were problems with the sequential navigation. Participants did expect to be able to also navigate the game board vertically with the arrow keys. Because of this limitation, some participants only used the browsing strategy to interact with the game. This might be an explanation why some participants used the mouse instead of screen reader interaction to toggle the tile selection. The restriction to sequential navigation was intended because we think there is a better solution to the problem of quickly navigating a large number of elements with a screen reader, which will be discussed in the next chapter. While this reason was clear to us, it wasn't clear to the participants. So we should have at least provided them with some kind of explanation.

6 Suggestions for Future Work

We know that building the toolkit as presented in this thesis was only the first step in making Unity games more accessible. Continuing to extend its features and platform support is crucial to increase both adoption from developers and the impact for visually impaired gamers. We have already shown several areas for future work in Section 2.5.2, but we want to use this chapter to discuss the ones where we see the most potential.

6.1 Improvements for Existing Tools

This first part will concern itself with extending the capabilities of tools that have already been created for this thesis.

6.1.1 Native Screen Reader Support

With our work to support the VoiceOver screen reader on iOS, we did already show that bridging to a native screen reader is possible. The logical next step would be to do the same for other platforms and screen readers. The easiest is probably supporting for VoiceOver on macOS since it could potentially reuse some of the code from the iOS implementation.

Another good pick would be looking into support for the Android screen reader TalkBack. Android is the leading mobile operating system, powering 76 percent of all mobile devices (Statista, 2019). Since Android and iOS together cover 98 percent of the mobile market, supporting these two screen readers makes the adoption of our package for cross-platform mobile games more likely.

Other interesting choices for future support would be the aforementioned screen readers Microsoft Narrator, NVDA, and JAWS.

6.1.2 Voice Configuration and Language Support

Our toolkit provides access to the platform speech synthesizer on Windows, iOS, and macOS, which is mainly used to power text to speech conversion for our custom screen reader. To make the implementation easier, we've used a fixed configuration for the speech synthesizer. This is far from ideal and should be enhanced in future iterations of the toolkit. Users should be able to adjust pace, gender, and most importantly, the language of the voice. This last point also ties into another feature that should be considered for further enhancement. All the properties of our accessibility signifier, like label or description, are only localized in a single language. A future version of the toolkit should give a possibility to provide localizations of these values for different languages and use the most appropriate language for the users' system settings.

6.1.3 Custom Rotors

We've previously discussed that our screen reader built for Unity gives access to the different elements in either a serial manner or by browsing the screen with a finger or a mouse cursor. Accessing elements in a serial manner only allows moving to the next or previous element. The position of the element in the series is based on their location on the screen. This can make it hard and overly complicated to find a particular element if there are many accessible elements on the screen. To enhance this experience, we could turn again to the VoiceOver screen reader for inspiration. This screen reader has a feature called *Custom Rotors*, which allows the developer to limit the series of accessible elements to certain kinds of elements. For example, this allows to cycle through all available links on a website (Apple Inc., 2021).

In the context of games, such a feature can be quite useful. In a chess game, for example, there could be a Rotor to give quicker access to all the games'

UI elements and another Rotor that only cycles through pieces that can be moved.

6.1.4 Evaluating and Extending the Navigation Agent

The previously mentioned extensions all focused on the screen reader, but a second major part of the toolkit is our navigation agent. Doing a proper evaluation of it exceeded the scope of this thesis. So another topic for future work would be to do such an evaluation both with developers and visually impaired players, preferably by using existing projects of developers. The results of such an evaluation then could be used to enhance the current version of the navigation agent.

6.2 New Tools

We've selected the screen reader and navigation agent for this thesis because we think that our work in this area can have a big impact. But there are additional tools that could ease the development of visually accessible games. We want to explore two more in the following.

6.2.1 Dynamic Type System & Fonts

Many modern operating systems offer some kind of system-wide setting that lets the user dynamically change the font size. Depending on the platform, applications can sometimes automatically adjust to these changes, or retrieve such a setting and manually adapt to it. The idea would be to build native plugins for different platforms to expose this dynamic type setting in Unity and make user interface components adapt to it.

In combination with that, future contributors could research which typefaces are more easily legible for people with low vision and either offer a selection of such fonts to developers or create a new font for this purpose.

6.2.2 Increased Contrast

Another area that concerns legibility is contrast. Since video games are an interactive medium, a player can almost always cause a situation where two elements are hard to distinguish because they have low contrast to each other. A common example where this happens is with the so-called heads-up display (HUD). The HUD is an overlay to show crucial information to players while they move around in the gaming world. For many games the HUD does not visually adapt to the background it is displayed on and thus, can be hardly legible in certain situations. So a useful tool for Unity would be to have a system that calculates a color with high contrast to the image or section of the screen. This color could then be used to dynamically recolor or add a border around these elements and thus, make them easier to see. Such a dynamic recoloring would not work for all games though, since colors are often used to transport information and because of aesthetic reasons. For these cases, we can imagine a variation of the proposed system which could detect areas of low contrast and make the developer aware of them during development.

Besides offering tools to increase the HUDs contrast, future versions of this toolkit could add an outline shader or some other means to increase contrast in the whole game, similar to the ones we've seen in Section 2.3.4 when discussing the accessibility tools of The Last of Us Part II (Naughty Dog, 2020).

6.3 Increasing Appeal

Having a broad number of very capable vision accessibility tools is important to increase the appeal of the toolkit, but there are other things that can be done to make it more relevant to developers.

6.3.1 Support for other Native Assistive Technologies

Different assistive technologies are often built upon the same basic building blocks. So while a screen reader and a voice interface are two completely different technologies that often serve different people, they need to access the same accessibility information of UI elements, like their position or a label. This basic fact implies that our toolkit could already support other assistive technologies on the iOS platform where we have a plugin to access the native screen reader.

The two technologies that come to mind are Switch Control[1] and Voice Control[2]. Both of these technologies support people with motor disabilities to use their devices. So one area for future work would be testing out the support for these assistive technologies on iOS, making potential adaptions to enhance the support, and trying to do the same for other platforms and technologies.

6.3.2 Automatic Testing

Something that we've learned while implementing the toolkit is that screen reader accessibility support is sometimes used for automated testing, like in UIKit using the `UIAccessibilityIdentification` [3] protocol. We can image building something similar for Unity using our accessibility toolkit. Having such a system in place would help both with discoverability of the toolkit and adoption, since making the game accessible can then also help increase the general quality of the software through testing.

[1]https://support.apple.com/en-us/HT201370
[2]https://support.apple.com/en-us/HT210417
[3]https://developer.apple.com/documentation/uikit/uiaccessibilityidentification

7 Summary

Video games are intended to be a medium for everyone, and yet people with visual impairments are often excluded from them. Most games that can be played by this demographic are either created specifically for them or require a tiresome trial-and-error process to learn the basic structure of the game. Mainstream games, the ones that everyone else plays, only rarely support accessibility tools like screen readers or navigation agents. The main reason for this lack of support is the way that games are built, using cross-platform game engines. Unity, the most popular game engine, does not include any advanced tools to make visually accessible games. So making even simple Unity games accessible is often not feasible for developers, since they have to build all the tools themselves.

In this thesis, we've presented the design and implementation of the Unity Accessibility Toolkit, which helps developers build visually accessible games with the Unity game engine. We showed how an accessibility signifier, the UA11Y Element, can be used as the basic building block for different accessibility tools. The first of these tools was a Navigation Agent. This agent can guide a player to a selected location in an environment using auditory signals. The second tool is a screen reader, which can be used in a range of different scenarios, from menu navigation to environment observation. In addition to this custom screen reader, we've also managed to implement a native plugin for iOS that allows gamers to interact with Unity using the VoiceOver screen reader.

We also conducted a first evaluation of the screen reader that is included in the toolkit. Developers familiar with Unity were given a sample project of a match-3 puzzle game that is not accessible at all. Their task was to make the game accessible using our toolkit and fill out a post-questionnaire about their experience with the toolkit and with accessibility technologies in general.

The evaluation showed that the toolkit is easy to use even for developers without any prior experience with implementing screen reader support. We also discovered that a feedback loop would have been useful during the evaluation to prevent simple errors likely caused by that lack of experience. This is something we should consider for future evaluations. But all participants have shown a high interest in such a tool, empathizing the importance of work in this field.

Appendix

Bibliography

Abdolrahmani, Ali, Ravi Kuber, and Stacy M. Branham (2018). “”Siri Talks at You”: An Empirical Investigation of Voice-Activated Personal Assistant (VAPA) Usage by Individuals Who Are Blind.” In: *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS ’18. Galway, Ireland: ACM, pp. 249–258. ISBN: 978-1-4503-5650-3. DOI: 10.1145/3234695.3236344. URL: http://doi.acm.org/10.1145/3234695.3236344 (cit. on pp. 18, 19).

AbleGamers (2020). *The AbleGamers Charity: Our Stories*. AbleGamers. URL: https://ablegamers.org/pages/our-stories/ (visited on 01/15/2021) (cit. on p. 1).

Allman, Troy et al. (2009). “Rock Vibe: Rock Band® computer games for people with no or limited vision.” In: pp. 51–58. DOI: 10.1145/1639642.1639653 (cit. on p. 26).

Andrade, Ronny et al. (2019). “Playing Blind: Revealing the World of Gamers with Visual Impairment.” In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI ’19. Glasgow, Scotland Uk: ACM, 116:1–116:14. ISBN: 978-1-4503-5970-2. DOI: 10.1145/3290605.3300346. URL: http://doi.acm.org/10.1145/3290605.3300346 (cit. on pp. 1, 24, 27, 28, 31, 32).

Apple Inc. (2019a). *Accessibility for iOS and tvOS*. URL: https://developer.apple.com/documentation/uikit/accessibility_for_ios_and_tvos (visited on 10/17/2019) (cit. on p. 39).

Apple Inc. (2020a). *Accessibility Traits*. URL: https://developer.apple.com/documentation/objectivec/nsobject/uiaccessibility/accessibility_traits?language=objc (visited on 11/05/2020) (cit. on p. 51).

Apple Inc. (2019b). *SceneKit*. URL: https://developer.apple.com/documentation/scenekit (visited on 07/16/2019) (cit. on p. 39).

Apple Inc. (2019c). *Speech Synthesis*. URL: https://developer.apple.com/documentation/avfoundation/speech_synthesis (visited on 07/16/2019) (cit. on p. 39).

Apple Inc. (2019d). *SpriteKit*. URL: https://developer.apple.com/documentation/spritekit (visited on 07/16/2019) (cit. on p. 39).

Apple Inc. (2020b). *UIAccessibility*. URL: https://developer.apple.com/documentation/objectivec/nsobject/uiaccessibility?language=objc (visited on 10/30/2020) (cit. on p. 50).

Apple Inc. (2021). *UIAccessibilityCustomRotor*. URL: https://developer.apple.com/documentation/uikit/uiaccessibilitycustomrotor (visited on 01/15/2021) (cit. on p. 94).

Apple Inc. (2020c). *Vision Accessibility - iPhone*. URL: httpshttps://www.apple.com/accessibility/iphone/vision/ (visited on 10/02/2020) (cit. on p. 22).

Apple Inc. (2020d). *Apps for Accessibility*. URL: https://apps.apple.com/story/id1266441335 (visited on 10/08/2020) (cit. on p. 23).

Araújo, Maria C. C. et al. (2017). "Mobile Audio Games Accessibility Evaluation for Users Who Are Blind." In: *Universal Access in Human–Computer Interaction. Designing Novel Interactions*. Ed. by Margherita Antona and Constantine Stephanidis. Cham: Springer International Publishing, pp. 242–259. ISBN: 978-3-319-58703-5 (cit. on pp. 28, 32).

Arielle M. Silverman, Jason D. Gwinn and Leaf Van Boven (2014). "Stumbling in Their Shoes: Disability Simulations Reduce Judged Capabilities of Disabled People." In: *Social Psychological and Personality Science* 6.4, pp. 464–471. DOI: 10.1177/1948550614559650 (cit. on p. 32).

Atkinson, Matthew T. et al. (2006). "Making the Mainstream Accessible: Redefining the Game." In: Sandbox '06. Boston, Massachusetts: Association for Computing Machinery, pp. 21–28. ISBN: 1595933867. DOI: 10.1145/1183316.1183321. URL: https://doi.org/10.1145/1183316.1183321 (cit. on p. 1).

Baber, C. (1993). "Interactive Speech Technology." In: ed. by Christopher Baber and Janet M. Noyes. Bristol, PA, USA: Taylor & Francis, Inc. Chap. Developing Interactive Speech Technology, pp. 13–18. ISBN: 0-7484-0127-X. URL: http://dl.acm.org/citation.cfm?id=210140.210143 (cit. on p. 18).

Bailey, Ian L. and Jan E. Lovie-Kitchin (2013). "Visual acuity testing. From the laboratory to the clinic." In: *Vision Research* 90. Testing Vision: From

Laboratory Psychophysical Tests to Clinical Evaluation, pp. 2–9. ISSN: 0042-6989. DOI: https://doi.org/10.1016/j.visres.2013.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0042698913001259 (cit. on p. 8).

Blattner, Meera M., Denise A. Sumikawa, and Robert M. Greenberg (1989). "Earcons and Icons: Their Structure and Common Design Principles (Abstract Only)." In: *SIGCHI Bull.* 21.1, pp. 123–124. ISSN: 0736-6906. DOI: 10.1145/67880.1046599. URL: http://doi.acm.org/10.1145/67880.1046599 (cit. on p. 29).

Borodin, Yevgen et al. (2010). "More Than Meets the Eye: A Survey of Screen-reader Browsing Strategies." In: *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*. W4A '10. Raleigh, North Carolina: ACM, 13:1–13:10. ISBN: 978-1-4503-0045-2. DOI: 10.1145/1805986.1806005. URL: http://doi.acm.org/10.1145/1805986.1806005 (cit. on pp. 16, 18, 19).

Calleja, Gordon (2010). "Digital Games and Escapism." In: *Games and Culture* 5.4, pp. 335–353. DOI: 10.1177/1555412009360412. eprint: https://doi.org/10.1177/1555412009360412. URL: https://doi.org/10.1177/1555412009360412 (cit. on p. 1).

Christopoulou, Eleftheria and Stelios Xinogalos (2017). "Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices." In: *International Journal of Serious Games* 4, pp. 21–36. DOI: 10.17083/ijsg.v4i4.194 (cit. on p. 36).

Cowan, Benjamin R. et al. (2017). ""What Can I Help You with?": Infrequent Users' Experiences of Intelligent Personal Assistants." In: *Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services*. MobileHCI '17. Vienna, Austria: ACM, 43:1–43:12. ISBN: 978-1-4503-5075-4. DOI: 10.1145/3098279.3098539. URL: http://doi.acm.org/10.1145/3098279.3098539 (cit. on p. 18).

Darilek, Nolan (2018). *Add accessible_description property to Control*. URL: https://github.com/godotengine/godot/pull/20254 (visited on 10/16/2020) (cit. on p. 39).

Dix et al. (2003). *Human-Computer Interaction (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0130461091 (cit. on pp. 10, 11, 14, 16).

Dring, Christopher (2019). *AAA game downloads on the brink of overtaking physical in Europe*. URL: https://www.gamesindustry.biz/articles/

2019-06-07-aaa-game-downloads-on-the-brink-of-overtaking-physical-in-europe (visited on 10/02/2020) (cit. on p. 22).

Electronic Arts Inc. (2019). *EA Sports UFC 3 Guide for the Blind and Visually Impaired*. URL: https://www.ea.com/able/resources/ufc/ufc-3/ps4/guides (visited on 06/13/2019) (cit. on pp. 24, 25).

Ellis, Gerry (2016). "Impairment and Disability: Challenging Concepts of 'Normality'." In: *Researching Audio Description. New Approaches*. Ed. by Anna Matamala. Ed. by Pilar Orero. First. Palgrave Macmillan UK. Chap. 3, pp. 35–45. ISBN: 978-1-137-56916-5 (cit. on p. 4).

Entertainment Software Association (2020). *2020 Essential Facts About the Video Game Industry*. Tech. rep. URL: https://www.theesa.com/wp-content/uploads/2020/07/Final-Edited-2020-ESA_Essential_facts.pdf (visited on 10/02/2020) (cit. on p. 20).

Epic Games, Inc (2020a). *Set Color Vision Deficiency Type*. URL: https://docs.unrealengine.com/en-US/BlueprintAPI/Widget/Accessibility/SetColorVisionDeficiencyType/index.html (visited on 10/16/2020) (cit. on pp. 38, 41).

Epic Games, Inc (2020b). *Supporting Screen Readers*. URL: https://docs.unrealengine.com/en-US/Engine/UMG/UserGuide/ScreenReader/index.html (visited on 10/16/2020) (cit. on p. 38).

Epic Games, Inc (2020c). *Unreal Engine 4.23 Release Notes*. URL: https://docs.unrealengine.com/en-US/Support/Builds/ReleaseNotes/4_23/index.html (visited on 10/16/2020) (cit. on p. 38).

Friberg, Johnny and Dan Gärdenfors (2004). "Audio Games: New Perspectives on Game Audio." In: *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*. ACE '04. Singapore: ACM, pp. 148–154. ISBN: 1-58113-882-2. DOI: 10.1145/1067343.1067361. URL: http://doi.acm.org/10.1145/1067343.1067361 (cit. on p. 28).

Game Accessibility Guidelines (2019a). *Ensure screenreader support, including menus & installers. Game Accessibility Guidelines*. URL: http://game%5C-accessibility%5C-guidelines.com/ensure-screenreader-support-including-menus-installers/ (visited on 06/22/2019) (cit. on p. 24).

Game Accessibility Guidelines (2019b). *Include assist modes such as auto-aim and assisted steering. Game Accessibility Guidelines*. URL: http://gameaccessibilityguidelines.com/include-assist-modes-such-as-auto-aim-and-assisted-steering/ (visited on 06/13/2019) (cit. on p. 28).

Gamma, Erich et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201633612 (cit. on p. 57).

Gaver, William W. (1986). "Auditory Icons: Using Sound in Computer Interfaces." In: *Hum.-Comput. Interact.* 2.2, pp. 167–177. ISSN: 0737-0024. DOI: 10.1207/s15327051hci0202_3. URL: https://doi.org/10.1207/s15327051hci0202_3 (cit. on p. 29).

ghost (2017). *Add Accessibility for Blind Developers Who Use Screenreaders*. URL: https://github.com/godotengine/godot/issues/14011 (visited on 07/16/2019) (cit. on p. 38).

Gregory, Jason (2014). *Game Engine Architecture, Second Edition*. 2nd. Natick, MA, USA: A. K. Peters, Ltd. ISBN: 1466560010, 9781466560017 (cit. on p. 36).

Hamilton, Ian (2013). *Screenreaders and game engines*. URL: http://ian-hamilton.com/screenreaders-and-game-engines/ (visited on 06/21/2019) (cit. on p. 24).

Higgins, Tom (2010). *Unity Turns 5, Happy Birthday!* Unity Technologies. URL: https://blogs.unity3d.com/2010/06/07/unity-turns-5-happy-birthday/ (visited on 04/29/2021) (cit. on p. 37).

Hiliges et al. (2017). "Grasping Virtual Objects in Augmented Reality." US 9,552,673 B2. Microsoft Corporation (cit. on p. 10).

Huang, Jia-Bin et al. (2007). "Information Preserving Color Transformation for Protanopia and Deuteranopia." In: *Signal Processing Letters, IEEE* 14, pp. 711–714. DOI: 10.1109/LSP.2007.898333 (cit. on p. 8).

HumanWare (2019). *Brailliant BI 14 braille display*. URL: https://store.humanware.com/hus/brailliant-bi14-braille-display.html (visited on 07/12/2019) (cit. on pp. 13, 14).

Hussain, Badrul et al. (2006). "Changing from Snellen to LogMAR: debate or delay?" In: *Clinical & Experimental Ophthalmology* 34.1, pp. 6–8. DOI: 10.1111/j.1442-9071.2006.01135.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1442-9071.2006.01135.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1442-9071.2006.01135.x (cit. on p. 6).

Huston, Lainie (2019). *New Garage project bakes accessibility into game development via responsive spatial audio*. Microsoft Cooperation. URL: https:

//www.microsoft.com/en-us/garage/blog/2019/04/new-garage-project-bakes-accessibility-into-game-development-via-responsive-spatial-audio/ (visited on 07/17/2019) (cit. on p. 42).

Hyvärinen, Lea, Pentti Laurinen, and Jyrki Romvmo (1983). "Contrast sensitivity in evaluation of visual impairment due to diabetes." In: *Acta Ophthalmologica* 61.1, pp. 94–101. DOI: 10.1111/j.1755-3768.1983.tb01399.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1755-3768.1983.tb01399.x (cit. on p. 12).

Icel, Berk (2017). *Gaming Through New Eyes*. Youtube. URL: https://www.youtube.com/watch?v=P7n9s7yBlGw (visited on 06/13/2019) (cit. on p. 25).

IllegallySighted (2017). *Steam, Steam Website, & Steam IOS App Accessibility*. Youtube. URL: https://www.youtube.com/watch?v=DIHyg8-BrMk (visited on 06/22/2019) (cit. on p. 24).

Irvine et al. (2014). "Tablet and Smartphone Accessibility Features in the Low Vision Rehabilitation." In: *Neuro-Ophthalmology* 38.2, pp. 53–59. DOI: 10.3109/01658107.2013.874448. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5123149/ (cit. on p. 13).

Juan Linietsky, Ariel Manzur (2019). *Godot License*. Godot. URL: https://godotengine.org/license (visited on 07/16/2019) (cit. on p. 38).

Kovács, Péter et al. (2015). "Application of immersive technologies for education: State of the art." In: pp. 283–288. DOI: 10.1109/IMCTL.2015.7359604 (cit. on p. 31).

Kruger, Rynhardt and Lynette van Zijl (2014). "Rendering Virtual Worlds in Audio and Text." In: *Proceedings of International Workshop on Massively Multiuser Virtual Environments*. MMVE '14. Singapore, Singapore: ACM, 5:1–5:2. ISBN: 978-1-4503-2708-4. DOI: 10.1145/2577387.2577390. URL: http://doi.acm.org/10.1145/2577387.2577390 (cit. on p. 33).

Linietsky, Juan (2014). *Godot Engine Reaches 1.0, First Stable Release*. Godot. URL: https://godotengine.org/article/godot-engine-reaches-1-0 (visited on 07/16/2019) (cit. on p. 38).

Lokki, Tapio and Matti Grohn (2005). "Navigation with Auditory Cues in a Virtual Environment." In: *IEEE MultiMedia* 12.2, pp. 80–86. ISSN: 1070-986X. DOI: 10.1109/MMUL.2005.33. URL: http://dx.doi.org/10.1109/MMUL.2005.33 (cit. on p. 33).

Luger, Ewa and Abigail Sellen (2016). ""Like Having a Really Bad PA": The Gulf Between User Expectation and Experience of Conversational

Agents." In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI '16. San Jose, California, USA: ACM, pp. 5286–5297. ISBN: 978-1-4503-3362-7. DOI: 10.1145/2858036.2858288. URL: http://doi.acm.org/10.1145/2858036.2858288 (cit. on p. 18).

Molloy, David and Paul Carter (2020). *Last of Us Part II: Is this the most accessible game ever?* BBC. URL: https://www.bbc.com/news/technology-53093613 (visited on 10/08/2020) (cit. on p. 33).

National Eye Institute (2019). *Priority eye diseases: Refractive errors and low vision*. URL: https://www.who.int/blindness/causes/priority/en/index4.html (visited on 07/15/2019) (cit. on p. 6).

Norman, Donald A. (2002). *The Design of Everyday Things: Revised & Expanded Edition*. USA: Basic Books, Inc. ISBN: 9780465050659 (cit. on p. 49).

Oliveira, João et al. (2011). "BrailleType: Unleashing Braille over Touch Screen Mobile Phones." In: *Human-Computer Interaction – INTERACT 2011*. Ed. by Pedro Campos et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 100–107. ISBN: 978-3-642-23774-4 (cit. on p. 12).

Pivec Labs (2020). *Visual Aids*. URL: https://docs.piveclabs.com/game-creator/accessibility-module-for-game-creator/visual-aids (visited on 10/21/2020) (cit. on p. 42).

Porter, John R. (2014). "Understanding and Addressing Real-World Accessibility Issues in Mainstream Video Games." In: *SIGACCESS Access. Comput.* 108, pp. 42–45. ISSN: 1558-2337. DOI: 10.1145/2591357.2591364. URL: https://doi.org/10.1145/2591357.2591364 (cit. on p. 1).

Pradhan, Alisha, Kanika Mehta, and Leah Findlater (2018). ""Accessibility Came by Accident": Use of Voice-Controlled Intelligent Personal Assistants by People with Disabilities." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 459:1–459:13. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3174033. URL: http://doi.acm.org/10.1145/3173574.3174033 (cit. on pp. 18, 19).

Preece, Aaron (2013). *The World of Audio Games: A Crash Course*. American Foundation for the Blind. URL: https://www.afb.org/aw/14/11/15738 (visited on 06/13/2019) (cit. on p. 28).

RARECSM (2019). *Sea of Thieves: Configuring 'Let Games Read to Me' Game Transcription*. Microsoft Corporation. URL: https://support.seaofthieves.com/hc/en-gb/articles/360022122074--Configuring-Let-Games-

Read-to-Me-Game-Transcription (visited on 10/02/2020) (cit. on p. 24).

Saylor, Steve (2020). *The Last Of Us Part II - MOST ACCESSIBLE GAME EVER! - Accessibility Impressions*. Youtube. URL: https://www.youtube.com/watch?v=PWJhxsZb81U&t=689s (visited on 06/13/2020) (cit. on p. 34).

Schaller, Dave (2006). *Unity and Accessibility*. URL: https://forum.unity.com/threads/unity-and-accessibility.8606/ (visited on 07/16/2019) (cit. on p. 37).

Schmidt et al. (2002). "Refreshable Braille Display System." US 6,354,839 B1 (cit. on p. 12).

Schneider, Joel (2002). *Classic Eye Chart*. URL: http://www.i-see.org/block_letter_eye_chart.pdf (visited on 10/29/2020) (cit. on p. 7).

Simunovic, M P (2010). "Colour vision deficiency." In: *Eye* 24.5, pp. 747–755. DOI: 10.1038/eye.2009.251. URL: https://doi.org/10.1038/eye.2009.251 (cit. on p. 8).

Smith, Brian A. and Shree K. Nayar (2018). "The RAD: Making Racing Games Equivalently Accessible to People Who Are Blind." In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 516:1–516:12. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3174090. URL: http://doi.acm.org/10.1145/3173574.3174090 (cit. on p. 32).

Sony Interactive Entertainment Europe (2020). *The Last of Us Part II sells more than 4 million copies*. URL: https://blog.playstation.com/2020/06/26/the-last-of-us-part-ii-sells-more-than-4-million-copies/ (visited on 10/09/2020) (cit. on p. 33).

Sony Interactive Entertainment LLC (2020). *Accessibility options for The Last of Us Part II*. URL: https://www.playstation.com/en-us/games/the-last-of-us-part-ii-ps4/accessibility/ (visited on 10/09/2020) (cit. on pp. 33–35).

Statista (2019). *Mobile operating systems' market share worldwide from January 2012 to July 2019*. URL: httpshttps://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (visited on 11/03/2019) (cit. on p. 93).

Stevens, Karen (2018). *AAA Gaming While Blind*. Youtube. URL: https://www.youtube.com/watch?v=YaDR4hJkskc (visited on 06/26/2019) (cit. on pp. 24–28).

Taylor, Paul (2009). *Text-to-Speech Synthesis*. Cambridge University Press. DOI: 10.1017/CBO9780511816338 (cit. on p. 14).

Thatcher, J. (1994). "Screen Reader/2: Access to OS/2 and the Graphical User Interface." In: *Proceedings of the First Annual ACM Conference on Assistive Technologies*. Assets '94. Marina Del Rey, California, USA: ACM, pp. 39–46. ISBN: 0-89791-649-2. DOI: 10.1145/191028.191039. URL: http://doi.acm.org/10.1145/191028.191039 (cit. on p. 15).

Thomsen, Mike (2012). *History of the Unreal Engine*. IGN. URL: https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine (visited on 07/16/2019) (cit. on p. 38).

Torrente, Javier et al. (2014). "Development of a Game Engine for Accessible Web-Based Games." In: *Games and Learning Alliance*. Ed. by Alessandro De Gloria. Cham: Springer International Publishing, pp. 107–115. ISBN: 978-3-319-12157-4 (cit. on p. 37).

Two-Tone (2018). *Have in editor options to simulate common forms of color vision deficiency (color blindness)*. URL: https://github.com/godotengine/godot/issues/21304 (visited on 07/16/2019) (cit. on p. 39).

UN General Assembly (2007). *Convention on the Rights of Persons with Disabilities*. URL: https://www.refworld.org/docid/45f973632.html (visited on 04/24/2019) (cit. on p. 5).

Unity R&D UX (2019). *Accessibility and inclusion*. URL: https://forum.unity.com/threads/accessibility-and-inclusion.694477/ (visited on 07/16/2019) (cit. on p. 37).

Unity Technologies (2019a). *Asset packages*. URL: https://docs.unity3d.com/Manual/AssetPackages.html (visited on 07/16/2019) (cit. on p. 40).

Unity Technologies (2020a). *Building a NavMesh*. URL: https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html (visited on 12/03/2020) (cit. on p. 70).

Unity Technologies (2020b). *Built-in Controls*. URL: https://docs.unity3d.com/Manual/UIE-Controls.html (visited on 11/05/2020) (cit. on p. 51).

Unity Technologies (2019b). *Game engines — how do they work?* URL: https://unity3d.com/what-is-a-game-engine (visited on 07/16/2019) (cit. on p. 36).

Unity Technologies (2019c). *How to make a video game in Unity without any coding experience*. URL: https://unity3d.com/make-a-game-in-unity-without-programming (visited on 07/16/2019) (cit. on p. 36).

Unity Technologies (2020c). *Inner Workings of the Navigation System*. URL: https://docs.unity3d.com/Manual/nav-InnerWorkings.html (visited on 12/03/2020) (cit. on p. 70).

Unity Technologies (2020d). *Multiplatform*. URL: httpshttps://unity.com/features/multiplatform (visited on 10/16/2020) (cit. on p. 37).

Unity Technologies (2019d). *Public Relations*. URL: https://unity3d.com/public-relations (visited on 07/16/2019) (cit. on p. 37).

Unity Technologies (2019e). *VisionUtility.GetColorBlindSafePalette*. URL: https://docs.unity3d.com/ScriptReference/Accessibility.VisionUtility.GetColorBlindSafePalette.html (visited on 07/16/2019) (cit. on p. 37).

Vice News (2019). *This Is How To Play Video Games If You're Totally Blind*. Youtube. URL: https://www.youtube.com/watch?v=aX0oPwQPo9A (visited on 06/13/2019) (cit. on p. 27).

Vtyurina, Alexandra et al. (2019). "Bridging Screen Readers and Voice Assistants for Enhanced Eyes-Free Web Search." In: *The World Wide Web Conference*. WWW '19. San Francisco, CA, USA: ACM, pp. 3590–3594. ISBN: 978-1-4503-6674-8. DOI: 10.1145/3308558.3314136. URL: http://doi.acm.org/10.1145/3308558.3314136 (cit. on p. 18).

White, Gareth R., Geraldine Fitzpatrick, and Graham McAllister (2008). "Toward Accessible 3D Virtual Environments for the Blind and Visually Impaired." In: *Proceedings of the 3rd International Conference on Digital Interactive Media in Entertainment and Arts*. DIMEA '08. Athens, Greece: ACM, pp. 134–141. ISBN: 978-1-60558-248-1. DOI: 10.1145/1413634.1413663. URL: http://doi.acm.org/10.1145/1413634.1413663 (cit. on pp. 29, 34).

WHO Programme for the Prevention of Blindness and Deafness (2003). *Consultation on Development of Standard for Characterization of Vision Loss and Visual Functioning*. URL: https://apps.who.int/iris/handle/10665/68601 (visited on 07/15/2019) (cit. on p. 7).

World Health Organization (2001). *International Classification of Functioning, Disability and Health*. ISBN: 92 4 154542 9 (cit. on p. 5).

World Health Organization (2010). *Gobal data on visual impairment 2010*. URL: https://www.who.int/blindness/publications/globaldata/en/ (visited on 01/15/2021) (cit. on p. 1).

World Health Organization (2018). *International Classification of Diseases*. Version 11. URL: https://icd.who.int/browse11/l-m/en (visited on 07/15/2019) (cit. on pp. 6, 7).

Yuan, Bei and eelke folmer eelke (2008). "Blind Hero: Enabling Guitar Hero for the Visually Impaired." In: pp. 169–176. DOI: 10.1145/1414471.1414503 (cit. on p. 26).

Yuan, Bei, Eelke Folmer, and Frederick C. Harris (2011). "Game accessibility: a survey." In: *Universal Access in the Information Society* 10.1, pp. 81–100. ISSN: 1615-5297. DOI: 10.1007/s10209-010-0189-5. URL: https://doi.org/10.1007/s10209-010-0189-5 (cit. on p. 33).

Zhao, Yuhang et al. (2019). "SeeingVR: A Set of Tools to Make Virtual Reality More Accessible to People with Low Vision." In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: ACM, 111:1–111:14. ISBN: 978-1-4503-5970-2. DOI: 10.1145/3290605.3300341. URL: http://doi.acm.org/10.1145/3290605.3300341 (cit. on p. 10).

Ludography

DOWINO (2016). *A Blind Legend*. (Visited on 12/18/2020) (cit. on pp. 30, 31).

EA Tiburon (2020). *Madden NFL 20*. (Visited on 12/19/2020) (cit. on pp. 26, 27).

Epic Games (2017). *Fortnite*. (Visited on 12/18/2020) (cit. on p. 38).

Epic MegaGames Digital Extremes, Legend Entertainment (1998). *Unreal* (cit. on p. 38).

Harmonix (2007a). *Guitar Hero* (cit. on p. 26).

Harmonix (2007b). *Rock Band* (cit. on p. 26).

Mojang Studios, Double Eleven (2020). *Minecraft Dungeons*. (Visited on 12/18/2020) (cit. on p. 38).

Naughty Dog (2020). *The Last of Us Part II*. (Visited on 12/18/2020) (cit. on pp. 33–35, 46, 90, 96).

NetherRealm Studios (2019). *Mortal Kombat 11*. (Visited on 12/19/2020) (cit. on p. 27).

Nintendo (2017). *Mario Kart 8 Deluxe*. (Visited on 12/18/2020) (cit. on pp. 27, 28).

PopCap Games (2001). *Bejewled*. (Visited on 12/19/2020) (cit. on pp. 29, 79).

Schultz, Marty (2014). *Blindfold Sudoku*. (Visited on 12/18/2020) (cit. on pp. 29, 30).

Schultz, Marty (2016). *Blindfold Pinball*. (Visited on 12/18/2020) (cit. on pp. 29, 30).

Six to Start (2012). *Zombies, Run!* (Visited on 12/19/2020) (cit. on p. 32).

Skizzix (2009). *Shredder Chess*. (Visited on 12/18/2020) (cit. on p. 33).

Strasser, Klemens (2018). *Subwords*. (Visited on 12/18/2020) (cit. on p. 33).