

ON THE ACCESSIBLE GRAPHICS PROBLEM

By

William Allee

Bachelor of Science – Computer Science
University of Nevada, Las Vegas
2014

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science – Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2026

Copyright by William Allee, 2026

All Rights Reserved



Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

May 21, 2026

This thesis prepared by

William Allee

entitled

On the Accessible Graphics Problem

is approved in partial fulfillment of the requirements for the degree of

Master of Science – Computer Science
Department of Computer Science

Andreas Stefik, Ph.D.
Examination Committee Chair

Wolfgang Bein, Ph.D.
Examination Committee Member

Bryar Shareef, Ph.D.
Examination Committee Member

Sarah Harris, Ph.D.
Graduate College Faculty Representative

Alyssa Crittenden, Ph.D.
*Vice Provost for Graduate Education &
Dean of the Graduate College*

Abstract

Computer graphics has become essential across the breadth of computing, ranging across applications that are professional, educational, entertaining, and more. Games and game engines, 3D modeling, web pages, data science and data visualization, block programming, and many other applications all rely on graphical output. While advances in graphics have obvious benefits, one group has been consistently left behind: those with limited or no vision. This issue is known as the Accessible Graphics Problem. In this thesis, we make four core contributions toward this problem. First, we discuss the Accessible Graphics Problem in greater detail, outlining the underlying context and the difficulties in addressing it. Second, we provide a technical solution that we call the Quorum Accessible Semantics Layer, which facilitates communication between applications and accessibility clients by providing semantic models. Third, we discuss obstacles that can appear in what we call a “semantically accessible” application, as well as techniques to mitigate them. Finally, we present functional examples using the Quorum Accessible Semantics Layer and the discussed techniques, including applications for coding, visual scene design, data science, and games. The work we discuss here is general purpose, potentially allowing access to a wide variety of graphical applications and the corresponding professional and educational fields that use them.

Acknowledgements

I would like to thank my wife, Meagan, as well as Dr. Andreas Stefik. Without both of them, this thesis would never have been completed (for very different reasons).

WILLIAM ALLEE

University of Nevada, Las Vegas
August 2026

Table of Contents

Abstract	iii
Acknowledgements	iv
List of Figures	vii
Chapter 1 Introduction	1
Chapter 2 Related Work	3
2.1 Approaches for Accessible Graphics	3
2.2 Accessibility in Games	4
Chapter 3 The Need for Accessible Graphics	6
3.1 The Consequences of Inaccessible Graphics	6
3.2 Screen Readers and Other Accessibility Clients	7
3.3 Self-Accessible Solutions	8
Chapter 4 The Quorum Accessible Semantics Layer	11
4.1 The Quorum Graphics Engine	11
4.2 Providing Accessible Semantics	12
4.3 Parallel Structures	15
4.4 Platform-Specific APIs and Implementations	17
4.5 Shadow DOM and Accessible Semantics in Web Browsers	18
Chapter 5 Techniques for Accessible Semantics	20
5.1 Arbitrary Graphics	20
5.1.1 Custom Nodes	20
5.1.2 Arbitrary Messages and the Two-Dummy Approach	21
5.2 Accessible Starvation and Racing Screen Readers	22
5.2.1 Accessible Starvation	22
5.2.2 Racing the Screen Reader	23

Chapter 6	Examples of Semantic Accessibility: Code, Blocks, Scenes, Charts & Games	25
6.1	Quorum Studio	25
6.1.1	The Integrated Development Environment	25
6.1.2	Block Programming	26
6.1.3	The Visual Scene Editor	28
6.2	Charts and Data Visualization	31
6.3	Semantically Accessible Games	34
6.3.1	Ground Flight Tactics	34
6.3.2	Block Breaker	34
6.3.3	Melvin Quest	36
Chapter 7	Conclusion	38
	Bibliography	39
	Curriculum Vitae	46

List of Figures

1.1	Examples of accessible graphics, including: a 3D visual editor (top), an accessible 2D platformer (middle), and an interactive bar chart (bottom). All of these can be created by, and used by, people without seeing the screen.	2
4.1	A diagram showing the relationship between game and accessibility components in a Quorum program. The library detects the appropriate native accessibility implementation and provides it at runtime.	13
4.2	Quorum controls and their automatically assigned corresponding control type in Windows UIA.	14
4.3	A simple user interface application made with Quorum, with a small tree, a button, and a logo.	15
4.4	A diagram of graphical and semantic objects and their hierarchy from the application in Figure 4.3.	16
6.1	A screenshot of Quorum Studio. It showcases a simple program being edited as text in a traditional IDE.	26
6.2	A screenshot of Quorum Studio in block mode. The file on the right side is displayed as editable blocks, and the left side shows a palette with available blocks.	27
6.3	A screenshot of Quorum Studio's scene editor. This screenshot shows a 2D scene, where the user has created an island which is dotted with trees, treasure chests, and some characters. .	29
6.4	A screenshot of Quorum Studio's scene editor. This screenshot shows a 3D scene with a series of primitive objects placed into the level. The palette on the left shows available models which can be placed into the scene.	30
6.5	A screenshot of Quorum Studio's scene editor. This screenshot shows a 3D scene with many different models placed from the palette into the environment. The selection cursor is highlighting a ship in the middle of the scene, which is also selected in the list of scene elements in the tree on the left. A properties window in the bottom right allows for adjustments to the selected model.	31
6.6	A screenshot of an accessible, interactive bar chart made with Quorum.	32
6.7	A screenshot of an accessible, interactive scatter plot made with Quorum.	33

6.8	A screenshot of an accessible, interactive pie chart made with Quorum.	33
6.9	A screenshot of Ground Flight Tactics, an accessible tactics game made with the Quorum Studio 3D scene editor.	35
6.10	A screenshot of Block Breaker, an accessible brick-breaker style game incorporating 2D graphics and a 3D skybox.	36
6.11	A screenshot of Melvin Quest, an accessible exploration adventure game built with the Quorum Studio 2D scene editor.	37

Chapter 1

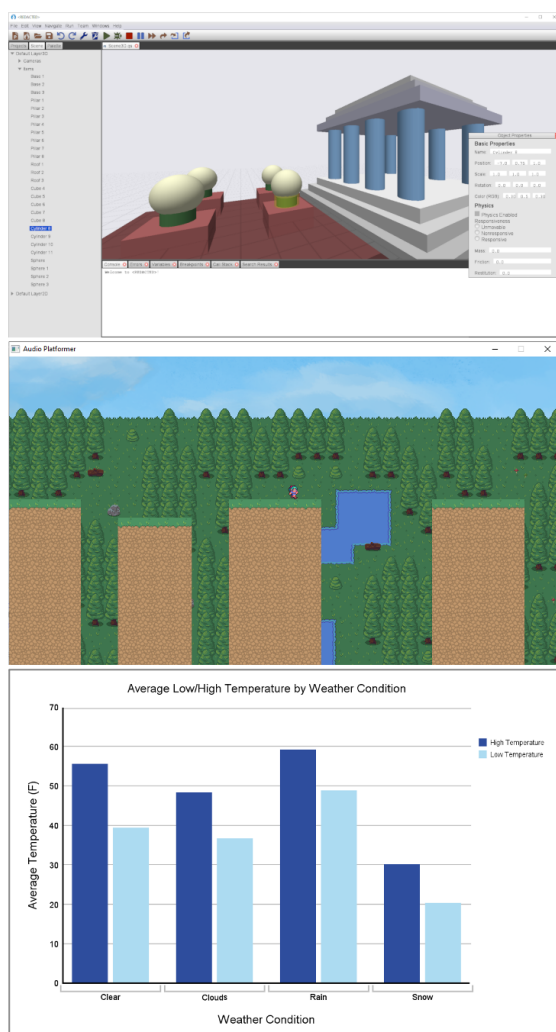
Introduction

Computer graphics are used ubiquitously across many sectors. This includes a wide-ranging array of professional tools across diverse fields, including tools for computer game design, like Unity3D [75] or Unreal Engine [33], for modeling, like Blender [17] or Maya [32], or engineering, like MATLAB [64], RStudio [65], or SolidWorks [8]. Computer graphics are also essential for data visualization, allowing complex information to be displayed through charts or other graphics. This again reaches across many applications, including scientific publications, corporate planning, news media, and school curriculum. Schools may also use computer graphics as part of other teaching tools – one common example in computer science curriculums is the use of block programming tools, such as Scratch [51] or Blockly [18].

In short, computer graphics are omnipresent today, and for good reason. For users that can see the screen, these graphics provide compelling ways to understand and interact with applications. For users that cannot, however, there is no practical way to participate. This means that people that are blind or visually impaired are cut off from entire professions and learning experiences. At a high-level, we call this issue the “Accessible Graphics Problem.”

Our work here has four primary contributions which we will discuss. First, we will review the nature of the Accessible Graphics Problem, why a solution is necessary, and what it means to make graphics “semantically accessible.” This important context informs the design of our second contribution, a technical solution to make graphical applications accessible via screen readers and other assistive technologies across different platforms, which we call the Quorum Accessible Semantics Layer. Third, we will discuss the design of semantically accessible applications, including hurdles that may appear and techniques to overcome them. Finally, we will discuss real, functional examples of accessible graphics that were created using the semantics layer and techniques previously discussed. Figure 1.1 shows just a few kinds of applications that can be made accessible. These examples serve to show that the Accessible Graphics Problem is not just narrowly solvable for one case, but that a general solution can be applied to solve it in many cases.

Figure 1.1: Examples of accessible graphics, including: a 3D visual editor (top), an accessible 2D platformer (middle), and an interactive bar chart (bottom). All of these can be created by, and used by, people without seeing the screen.



Chapter 2

Related Work

Accessibility is a broad and complex topic, encompassing both technical challenges and human factors. Research in this space frequently looks to make particular applications more accessible, but what it means to be accessible is at least partially subjective. In this chapter, we will review a wide number of methods and modalities which have been investigated. We focus our discussion around common approaches to accessibility and common technical considerations.

2.1 Approaches for Accessible Graphics

When considering how to make graphics broadly accessible, a variety of approaches have been examined. One common approach is sonification, the use of non-verbal audio to convey information. Among other domains, sonification has been particularly studied in relation to charts. Walker and Mauney studied similarities and differences in how sighted users expected graphs to be sonified compared to blind and visually impaired users [79], and Tomlinson et al. studied the use of the GNIE auditory graphing tool in a classroom over a period of 2 years [78]. In a broader application of sonification to graphics, Thapa et al. compared the use of image alt-text against “audemes” (essentially meaningful, non-verbal sound clips) in a pilot study [76]. Other studied applications of sonification include locating objects in a 3D virtual room [45], conveying colors [61], and chemical spectral analysis [56]. Work from this space has also impacted practical applications, such as the Graphics Accelerator by SAS, a sonification tool for charts and statistics [34].

It should be noted, however, that sonification is by no means a silver bullet. In fact, according to Neuhoff, “the widespread adoption of sonification to present complex data has largely failed to materialize” despite decades of research [55]. Neuhoff attributes this to a number of obstacles, among which is a simple problem: human hearing is less precise than vision in perceptualizing data [55]. Additionally, there is evidence that sonification is harder to understand and prompts slower reactions than speech or hybrid methods [54].

Aside from sound, the use of touch for accessible graphics has been studied extensively. A systematic literature review by Butler et al. identified 292 publications on touch-based accessible graphics between

2010 to 2020 [4]. Research in this area includes accessibility of non-digital graphics, especially through the use of tactile graphics or 3D printed models [26, 69]. Research into touch-based digital graphics includes the use of vibrotactile graphics, especially the use of vibration with touchscreens to convey graphics [22]. It also includes explorations using more specialized hardware, such as refreshable touch displays [58], haptic controllers [43], or haptic gloves [38]. Typical approaches often include the use of laser doppler vibrometers to estimate mathematical properties of vibration motors, which can be complex. Overall, a key takeaway from this research is that human perception through touch is powerful, but limited [22].

Another topic that has seen considerable research is the use of computer vision and artificial intelligence to contextualize graphics. For example, Jayant et al. showed that feedback from computer vision could improve the quality of photographs taken by blind individuals [35]. Meanwhile, a breadth of research has studied the use of computer vision and artificial intelligence for image captioning [28]. This includes specific applications such as news images [15], image tables [52], images on social media [82][83], and even memes [66]. Broadly, this research suggests that AI shows promise in improving accessibility for graphics.

Despite this, there are reasons to suspect that AI cannot solve the Accessible Graphics Problem alone. While the use of AI for accessibility is an area of continuing research, and its full potential is as of yet unknown, the current research suggests a number of possible limitations. Notably, current solutions are limited in scope. While more general models can be applied to provide image captioning, they appear to give poor results in some contexts. For example, when Mo et al. evaluated blind and low vision participants use of GPT-4V to describe data tables, participants considered its descriptions “too general”, and held a broader concern that “AI-generated information were often too vague or error-prone” [52]. In a study where blind participants were given a tool to generate descriptions sharing photos on Facebook, Zhao et al. encountered similar concerns – despite overall positive reception to the tool and the increased information available to them, “participants did not completely trust the computer-generated descriptions,” and most participants relied on assistance from sighted peers to verify the information [83]. The use of AI for accessibility in general is also complicated by a lack of existent or accessible training data specific to blind and visually impaired people, and adapting models to these specific conditions is difficult [80].

2.2 Accessibility in Games

As an obvious application of graphics, it is natural that we would consider digital games when thinking of accessible graphics as a whole. A considerable number of publications have investigated various facets of accessibility in games, with a systematic literature review by Hassan identifying 162 manuscripts on the topic between 2016 and 2020 alone [25]. It is important to note, however, that not all publications on accessible games are necessarily relevant to accessible graphics. Research in this space covers a wide array of disabilities, including visual, auditory, cognitive, and motor disabilities, and so this research may approach the topic from many different angles. For example, Roden and Parberry created a game engine which was capable of making audio only games, set in 3D environments [70], while Blind Adventure is a game engine

designed for blind game designers which is capable of making audio-based games [73]. These tools, while noteworthy in their own right, sidestep the issue of accessible graphics by simply removing the graphics from the equation.

For research that does touch on accessible graphics, this research commonly focuses on bespoke solutions for particular cases. In Hassan’s literature review, the most common research topic was the design and evaluation of “an accessibility related artefact, specifically, a full or partial game or game-based application” [25]. Just a few of these include a blind-accessible adaptation of an educational roller coaster simulator [9], crossword puzzles for the blind [67] and deaf [6], competitive 3D games for sighted and blind users via haptics, wind, and surround sound [23], a yoga exercise game for blind users incorporating body tracking with a Kinect [68], and an interface for blind users to play Minecraft through haptics and physical blocks [2].

Outside of academic research, some elements of the gaming industry have taken an interest in accessibility, with several recent high-profile games including significant accessibility features. *God of War Ragnarök* and *the Last of Us Part II* both have a range of options for players with low vision, deafness, motion sickness, or motor disabilities, with the latter also including magnification and text to speech [62][63]. Another notable example in recent years is *Diablo 4*. Among other accessibility features, it both contains a built-in screen reader and supports third-party screen readers [12]. Each of these is impressive in its own right, and demonstrates that even complex graphics can be made accessible. Unfortunately, all three are closed source, and to the best of our knowledge all three operate on custom game engines proprietary to their respective companies. As a result, the methods they have applied to make their games accessible appear to be bespoke, and any tools they have used are not available to other developers. Despite the advances made by these games, accessible games are rare, and as part of a systematic review of 45 studies, Aguado-Delgado et al. argue that video games being developed today are not truly accessible and that “none of the studied initiatives can guarantee universally accessible video game applications” [1].

Chapter 3

The Need for Accessible Graphics

In order to understand our contributions in this thesis, it is important to first understand why this work is necessary. The impacts of inaccessible graphics are not hypothetical and have real consequences for many people. In this chapter, we will discuss these impacts and the individuals they affect. We will also discuss how people with disabilities interact with graphical applications and the tools they use to do so.

3.1 The Consequences of Inaccessible Graphics

There are many people with disabilities across the globe, with the World Health Organization estimating that there are 1.3 billion people with significant disability – or in other words, 1 in 6 people globally [60]. What constitutes a disability is not obvious, with a number of competing definitions and models encompassing many different physical, mental, personal, and social factors [24][19]. Despite this, when we discuss disabilities in this thesis, we primarily talk about people who are blind or have low vision. That is not to say that graphics cannot be inaccessible to individuals with other disabilities – for example, charts can be inaccessible to individuals with cognitive impairments [81], and interactive graphics like games can be inaccessible to individuals with motor impairments [42]. With this in mind, we focus on people with blindness or low vision because they are the most obviously impacted population when applications rely purely on visual information, i.e. graphics – though, as we will discuss, the contributions in this thesis are helpful to people across a wide range of disabilities.

In the United States, approximately 6 million people have vision loss, and approximately 1 million people are blind [16]. This means that, without access to graphics, millions of people are automatically unable to participate in many learning or work experiences, limiting access to high-paying professions in data science related sectors like economics, health care, or finance [44]. Overall, those with disabilities do not participate in STEM related professions at the same rates as those without disabilities [77], and in the United States, the employment rate amongst blind and visually impaired individuals aged 21 to 64 is about 45% – nearly half the employment rate of individuals in the same range without a disability, at 80% [13].

Additionally, a number of laws in the United States guarantee people with disabilities some basic rights. These reach across multiple sectors, but are especially notable in the domain of education. The Individuals with Disabilities Education Act (IDEA) [10] and Section 504 of the Rehabilitation Act of 1973 [11] both guarantee the right to public education for people with disabilities under certain criteria. The Americans with Disabilities Act of 1990 applies more broadly, providing equal access to employment, state and local government services (once again including public education), and goods and services from many kinds of public-facing businesses [7]. Such rights are hardly exclusive to the United States, as other regions have also adopted similar laws, including the Accessible Canada Act [5] or the European Union’s EN 301 549 standard for digital accessibility [14]. The nuances of when and where these laws do and do not apply is complex, and a complete discussion of these and other laws is outside the scope of this work. The key takeaway is that people with disabilities have a right to be able to participate in key parts of society – and without accessible graphics, often they cannot.

3.2 Screen Readers and Other Accessibility Clients

To understand how we can solve the Accessible Graphics Problem, it is important to first understand how other digital content is made accessible. In other words, we must consider how blind and low vision people navigate, understand, and interact with content they cannot see. Typically, this is achieved by using some form of accessibility client. These tools convey content semantically through words, telling the user what is on the screen and how they can interact with it (typically using the keyboard). A simple example of this is the use of alt-text, where an image is accompanied by a brief description that can be read by an accessibility client. For a more complex example, consider how a blind person might navigate a web page. The page could contain headers, paragraphs, controls (like buttons or text fields), or graphical elements like our previously mentioned image with alt-text. The user can jump between elements (most likely with the keyboard), and the accessibility client will relay the semantic information relevant to each element as the user arrives at them. In other words, the individual pixels of the graphics are usually irrelevant – what matters is the context conveyed by them, and providing a method for accessibility clients to access that information.

While other technologies are used, the de facto standard for blind accessibility clients is screen reader software. Screen readers are so ubiquitous that every major operating system today comes with a screen reader natively available, including Narrator for Windows [48], TalkBack for Android [21], and VoiceOver for iOS and Mac OS X [29]. The operation of a screen reader is simple on its surface – when a user interacts with an element in an application (e.g., the element is focused with the keyboard), the screen reader uses synthesized speech to speak aloud any relevant semantic information attached to that element. Beneath the surface, things are more complicated, with screen readers often providing a vast array of hotkeys and options for more experienced users.

Still, screen readers provide a number of key advantages over other technologies. For example, compare them to refreshable braille displays. Braille displays are specialized pieces of hardware with pins arrayed

into rows. When an element is interacted with, the available semantic information is displayed as braille characters on the board by raising and lowering the pins. These devices are functional and used in practice, though to our knowledge there is no reliable measure of how many of these devices are used today. The American Printing House for the Blind estimates that 10,000 braille devices are sold worldwide annually, which suggests that the number of users is likely a small proportion of the blind population [27]. One possible reason for low adoption is the simple prerequisite that users must understand braille in order to use the device. The rates of braille literacy among the blind and visually impaired is often cited to be approximately 10%, though this number is contentious – an investigation by Sheffield et al. found that the “source of the often-cited ‘10%’ statistic” was from two sources: non-generalizable APH Federal Quota data, and a National Library Service report from 1979 [72]. A more concrete reason for low adoption, however, is that braille displays are notably expensive – the American Foundation for the Blind estimates that their costs “range from \$3,500 to \$15,000” [3], compared to screen readers which are often available for free.

With this in mind, we often frame the discussion in this thesis around screen reader usage. Still, what we really want is a system that supports both screen readers *and* braille displays – and ideally any other accessibility client, even ones we might not consider in advance. To do that, we will need our applications to be what we call “semantically accessible” – that is, our applications should provide the operating system with the necessary semantic information for accessibility clients to read. This is easier said than done, as each major operating system has its own semantic model which is incompatible with the others. As we will later discuss, these models tend to be complex and difficult to implement – but without them, we cannot relay information to accessibility clients.

3.3 Self-Accessible Solutions

Supporting the semantic accessibility models on various platforms requires significant effort. With this in mind, we may naturally be tempted to sidestep the issue by making our applications accessible without the use of outside tools or clients. In other words, rather than making our applications semantically accessible, we might consider making our applications “self-accessible.” While such solutions are possible, and at times necessary, they carry significant downsides.

A well known example of such self-accessible feature is self-voicing. A self-voicing application generates its own synthesized speech in a manner similar to a screen reader, but without relying on an external accessibility client. To be clear, this can be useful at times. For a trivial example, consider an installer for a screen reader – if the user is installing a screen reader for the first time, they reasonably might not have any other accessibility client available to them beforehand. Self-voicing can also be useful on platforms where no screen reader is reasonably available. For example, game consoles historically have not had screen readers – though there are signs this is changing, with consoles like the Nintendo Switch 2 including a built-in screen reader [57]. Still, self-voicing and other self-accessible solutions should be considered as supplementary or options of last resort. There are several reasons why.

First, self-accessible solutions are narrow in scope. Self-voicing, for example, can act as a substitute for a screen reader, but it cannot mimic input to a braille display. Similarly, self-voicing cannot provide assistance to the deaf or help users with motor impairments. Manually implementing different self-accessible solutions for each of these groups becomes quickly overwhelming. On the other hand, using the operating system’s underlying semantic model supports any compatible assistive technology. Put another way, accessible semantics means that an application can be designed for all users with disabilities, and graphics accessible to the blind or visually impaired are potentially useful outside of this group as well.

Second, users must relearn access to each self-accessible application they interact with. For applications that rely on a common semantic model, users can expect that such applications will reliably provide specific kinds of information, and their assistive technology of choice will respond to that information in a consistent way. This matters because different screen readers and braille displays have different user interfaces. For example, if a user interacts with a cell in a table, different screen readers might give different levels of detail regarding the element’s name, description, position, related row/column headers, etc. Even if two screen readers provide the same details, they might present those details in a different order. At best, a self-voicing application could replicate the exact feedback of one of them, but users familiar with the other would have to re-learn. Further, this also means self-accessible applications may ignore any customization users have made to their own tools, like voice options, text-to-speech speed, native language, or custom keyboard bindings.

Third, self-accessible applications are prone to redundant feedback, or “double speaking.” When screen readers or other technologies are presented with an incompatible application, they do not necessarily fall silent. For example, an application in C# that used common windows controls would be, perhaps without the developer knowing, sending information to the operating system’s native accessibility model. The information being sent in this case will almost certainly be unhelpful or nonsensical. This means the screen reader may speak this information and talk over any custom speech, because the speech systems screen readers use are different and very custom. Thus, if a user enters a self-voicing application while a screen reader is turned on, they may hear two different systems speaking over each other at the same time. In this case, someone who uses a screen reader to operate their computer would need to turn on the screen reader to navigate to the self-voicing application, turn it off while they are using the application, and turn it back on again whenever they are done – or worse, if they are bouncing between the application and other applications on their computer, they will need to toggle it continuously on and off. Since this user needs the screen reader to navigate their computer in the first place, this also means they will need to memorize how to turn on their assistive technology when one is not active and the computer gives no feedback.

Considering these limitations, accessibility through a semantic layer offers the potential for higher quality access to more people than self-accessible solutions. However, we might still consider using elements of self-accessibility as supplementary measures. For example, while we might rely on the semantic layer to do the bulk of the work, we could manually provide sonification or haptics for application-specific uses. That said, it is in the best interest of both developers and users to leverage the operating system’s semantic model as

much as possible.

Chapter 4

The Quorum Accessible Semantics Layer

On the surface, providing accessible semantics for a graphical application may sound straightforward. Modern operating systems have standard APIs that accessibility clients can interface with, so theoretically all an application needs to do is map graphics APIs to the accessibility ones. In practice, a number of subtle obstacles make this much harder than it may sound. In this section, we will discuss our technical solution as a whole, including a high-level consideration of how accessible semantics is implemented across different platforms.

4.1 The Quorum Graphics Engine

The Quorum Accessible Semantics Layer is integrated directly into the Quorum graphics engine. Quorum is an open source, evidence-oriented programming language which is heavily used in K-12 computer science settings [41]. The language’s standard libraries include a graphics engine that can perform hardware accelerated 2D and 3D graphics, which can be deployed across Windows, Mac, iPhone, and Android, or be embedded in web pages.

It is important to note that the graphics and accessibility pipelines are distinct. While Quorum uses OpenGL (and its sibling APIs, OpenGL ES and WebGL), this is not mandatory. The framework could be used alongside any graphics API and still provide accessible semantics. Theoretically, we could provide semantics to a window without any rendering at all – the pixels on the screen are irrelevant to the semantic model, after all.

The Quorum Accessible Semantics Layer is not a tool to make one application accessible. Instead, it is a tool for providing accessibility support to *any* application made with the Quorum graphics engine. It is designed to be a general purpose solution. Importantly, there is nothing inherently special about Quorum that makes this approach possible. This approach is also a proof of concept that accessible semantics can be adapted elsewhere. It is entirely possible that commercial tools like Unity3D, Unreal Engine, Blender, or many others could become compatible with screen readers and other accessibility clients using a similar

approach.

4.2 Providing Accessible Semantics

When users create applications with the Quorum graphics engine, they create a single program written with the Quorum programming language. At runtime, the engine determines which platform the application is running on and automatically provides a connection to that operating system’s accessibility APIs via a strategy pattern. On desktop platforms and Android, Quorum applications use Java byte code to run on the Java virtual machine, and native access to the platform is available via the Java Native Interface (JNI) [59]. For iOS platforms, Java byte code is translated to a native representation via cross-compilation using MobiVM [53]. Thus, we can say in broad terms that for desktop and mobile applications the system first passes through an intermediary layer, then enters into the platform-specific native layer to provide semantic information.

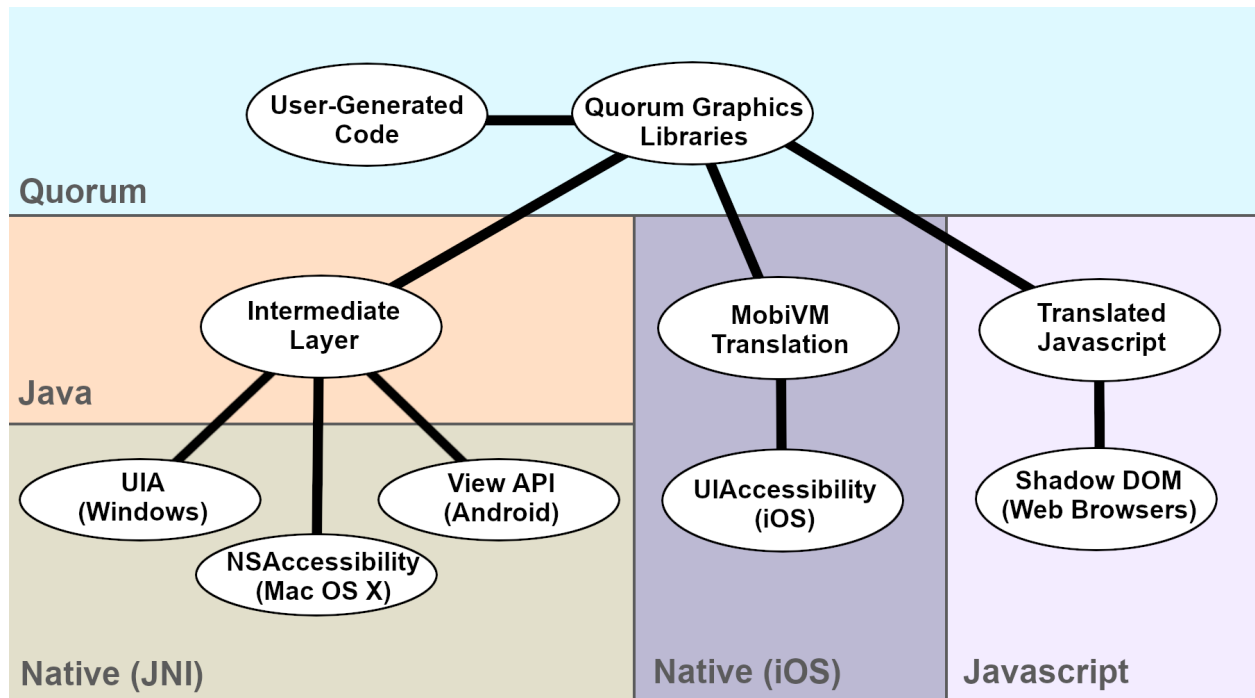
The exception is Quorum applications deployed to web browsers. Web browsers do not have a dedicated accessibility API – rather, the semantic model is inherent in the Document Object Model (DOM) underlying web pages. To leverage this, Quorum programs are cross-compiled to Javascript and provide semantic information using a technique known as shadow DOM [47]. Despite this, we will treat web browsers as a platform and the shadow DOM as an accessibility API for convenience of discussion (though we will note when our generalizations do not apply to the web). Figure 4.1 illustrates how accessible semantics is provided across different platforms.

Each platform has a different accessibility API and a different semantic model. These differences create a number of challenges to implementation, but the details of each are beyond the scope of this thesis. Instead, we will focus on the key similarities between them, which we can use to create a common foundation that will allow for cross-platform semantics.

First, each platform’s accessibility API is designed to facilitate user interfaces. Elements can be identified using common control types, such as buttons or text boxes. Whenever possible, we aim to associate Quorum elements with a respective control type available in the underlying accessibility API. In many cases, an obvious translation is available, e.g. a button in Quorum should be a button in the accessibility layer as well. (We discuss the situation where no convenient translation is possible in the next chapter.) These identifiers will inform screen readers of the type of content they are interacting with, which will influence what information they will request and how they speak to the user. The number of available control types differ between APIs, but generally a wide variety are supported. As an example, Figure 4.2 shows how Quorum controls are associated with available types in Windows’ accessibility API, User Interface Automation (UIA).

Second, each accessibility API represents an event-driven system of communication with outside processes. How much of the system falls within our control varies, but as a general rule our application will need to both send and respond to events. For example, we will need to indicate if text changes in a text box, or we must supply the name of an arbitrary control if a screen reader sends a request for it. To minimize response

Figure 4.1: A diagram showing the relationship between game and accessibility components in a Quorum program. The library detects the appropriate native accessibility implementation and provides it at runtime.



time, we aim to send events as soon as they occur. On the other hand, we generally try to process incoming events between frames of animation to prevent race conditions with ongoing game logic. As we will discuss later, this interplay of sending and receiving messages is delicate, and must be handled carefully.

Finally, each accessibility API organizes its semantic model into hierarchical structures. While the details of these again vary, in broad strokes we can consider our semantic models to be trees, with the application window as the root and the contents of the application as children of the window. This observation will give us a key approach to create our accessible applications.

Figure 4.2: Quorum controls and their automatically assigned corresponding control type in Windows UIA.

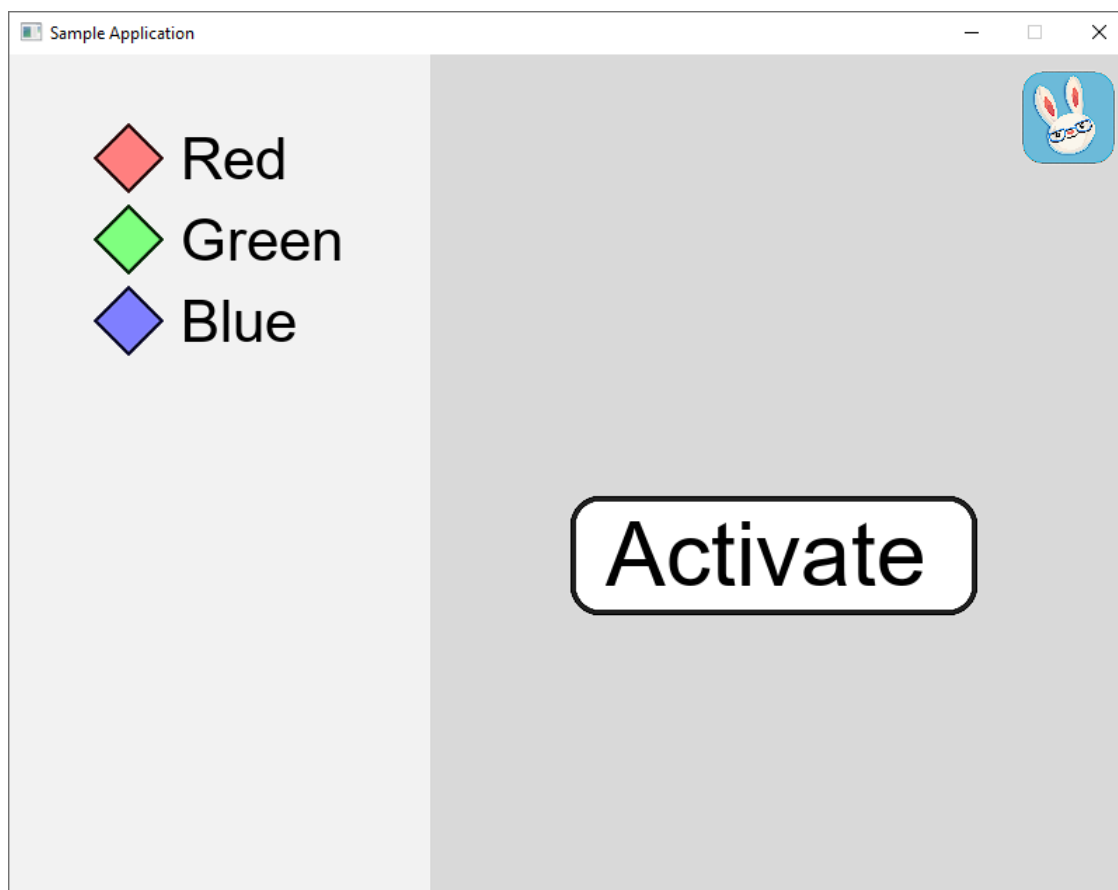
Quorum Control	UIA Control Type
Button	Button
ButtonGroup	Group
Cell	DataItem
Checkbox	CheckBox
Dialog	Window
Item	Custom
List	ListControl
ListItem	ListItem
MenuBar	MenuBar
MenuItem	MenuItem
PopupMenu	Menu
ProgressBar	ProgressBar
RadioButton	RadioButton
Spreadsheet	DataGrid
Tab	TabItem
TabPane	TabControl
TextBox	Document
TextField	Edit
ToggleButton	CheckBox
ToolBar	ToolBar
Tree	TreeControl
TreeItem	TreeItem
TreeTable	DataGrid
TreeTableCell	DataItem

4.3 Parallel Structures

When we consider a single accessible item in the Quorum engine, we can see it essentially has two parts: a Quorum element handling graphics and game logic, and a semantic component representing its accessible information. Items in the Quorum engine are arranged in a tree structure, where the game area acts as the root of the tree and Items are added onto it. Items can also be added onto each other to create hierarchical relationships. As we just discussed in the last section, the semantic model is also modeled as a tree, where the window is the root and elements are added hierarchically onto it.

This means that in order to add an element to the game, we essentially add two paired elements instead – the Quorum object representing the item’s graphics and game logic, and a record of the object in the semantic layer. The result is a heavy-weight application which handles graphics and game logic, which is visible on the screen, and a light-weight “reflection” in the semantic model, which is visible to accessibility clients. Crucially, inaccessible elements do not appear in the reflection – there is no need to store data for every blade of grass or cloud in the sky.

Figure 4.3: A simple user interface application made with Quorum, with a small tree, a button, and a logo.

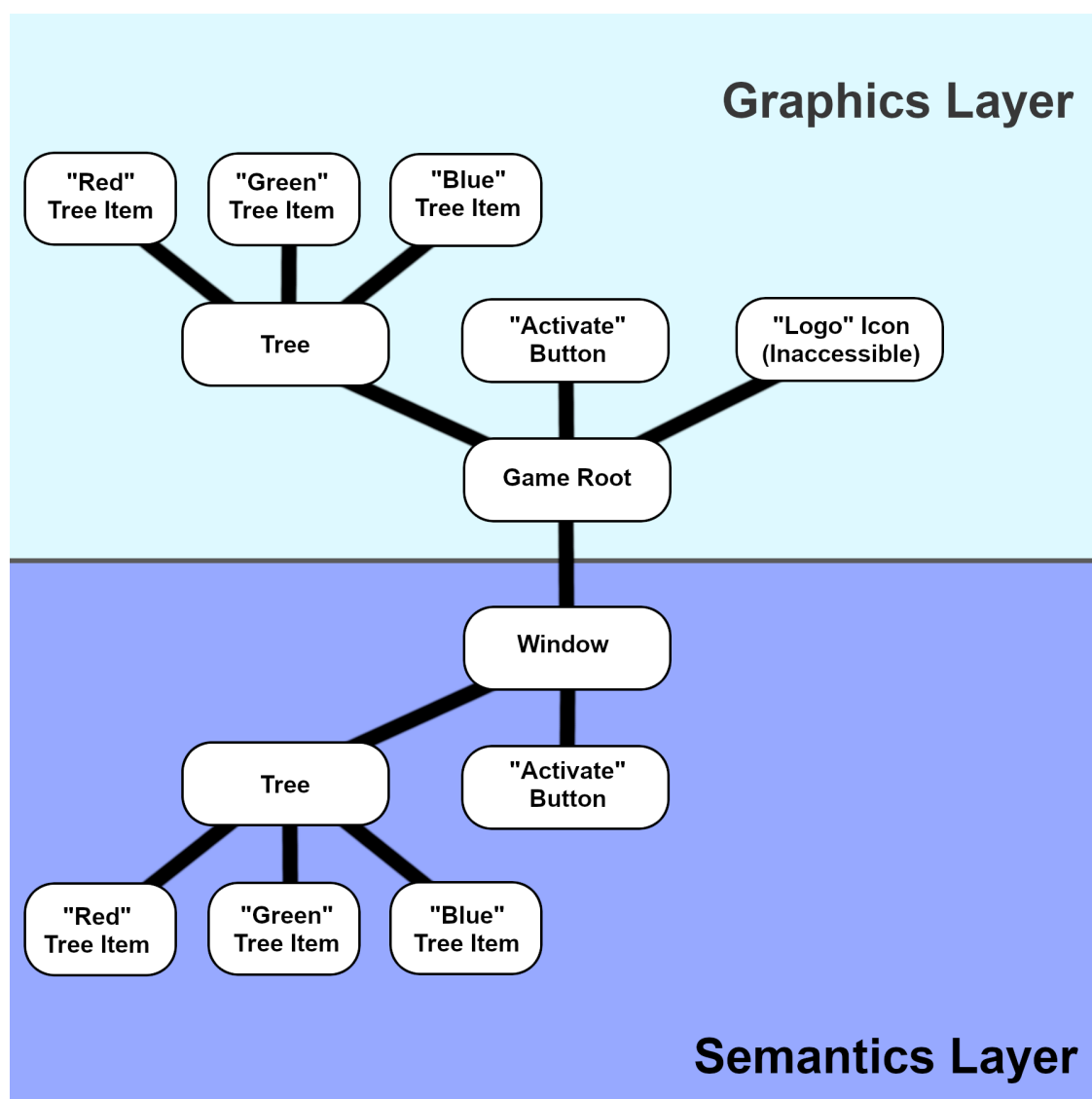


Consider an example. Figure 4.3 shows an accessible sample application created with Quorum. On the

left side, there is a tree with three top-level tree items, labeled “Red”, “Green”, and “Blue”, with colored diamond icons next to each one. The application also has a large button marked “Activate”, as well as a logo in the top-right of the screen. Semantically, there are two regions that can be interacted with: the tree items and the button. The logo is a purely decorative element which we will intentionally choose not to expose to accessibility clients.

When these elements are added to the application, they are added to the root of the game scene. At the same time, matching elements are added to the semantic layer. While each operating system has a different API, at an abstract level the semantic model will look the same across all supported systems. The result is a pair of parallel structures across the two halves of the system, as shown in Figure 4.4.

Figure 4.4: A diagram of graphical and semantic objects and their hierarchy from the application in Figure 4.3.



Inspecting the structure of the graphical and semantic nodes in this example, we see that they are identical with only one exception – the logo in our program appears in the graphics, but not in the semantic layer. This is intentional. Because the logo is neither interactive nor informative, it provides no benefit to a screen reader user, and so we explicitly choose not to convey information about it. In this sense, the semantic layer has no idea that the logo exists at all. Meanwhile, the tree, its contents, and the button are fully realized in the semantic layer, and so screen readers and other clients may find them and query for their information.

At a quick glance, it may seem as though we are essentially doubling the structure and complexity of our applications with this approach, but that is not the case. While the implementation of the semantics layer is dependent on the platform and native API, nodes in the semantic tree typically store only a few key accessibility related methods and values. Also, there are almost always dramatically more graphical nodes than semantic nodes in practice. In Figure 4.4, we describe the application in terms of its major UI elements, and as far as the semantic layer is concerned, those are indeed all that exists. This diagram arguably oversimplifies the graphics layer, however – UI elements are not truly a singular element, but a composite of many other graphical elements. The “Activate” button, for example, comprises a sprite for its background as well as another 8 sprites for its label (that is, one sprite for every character in “Activate”). In this way, we can say that the semantics layer consolidates 9 graphical nodes into just one semantic node for the button. The result is that our full-scale graphical application effectively casts a lean, accessible “shadow”.

4.4 Platform-Specific APIs and Implementations

So far in this chapter, we have discussed how the Quorum Accessible Semantics Layer works at a high level, and the common elements across platforms. In this section, we will briefly discuss the details of how the system works on each supported platform. This will not cover each native system in-depth, as a rigorous inspection of each platform’s API could easily be a paper on its own. Instead, we will highlight the APIs available on each system and comment on details of note. We will especially discuss the web browser implementation, which has a number of differences to the other systems.

On Windows, the system relies on the User Interface Automation (UIA) interface [49]. UIA has been the standard approach for providing screen reader access to applications since it was first introduced with Windows XP [50]. It is not the first such system, however. It replaced an earlier API called Microsoft Active Accessibility (MSAA), which was in use since 1997 on Windows 95 systems [71]. We mention this because, while UIA did succeed MSAA, it did not replace it entirely – even now, UIA implementations can report legacy values from MSAA, and in some cases screen readers may even support it better. For example, in our testing we found that JAWS 2020 does not properly report the UIA progress value for a progress bar, but will correctly report legacy values for one instead.

As we have discussed, other platforms have other APIs. To achieve screen reader support, the Quorum Accessible Semantics Layer uses NSAccessibility [30] on Mac OS X, and the newer UIAccessibility API [31] on iOS. Interestingly, Android does not have a distinct API for accessibility. Instead, it is integrated directly

into the View API [20], which is used to handle drawing and event handling on the screen. That is not to say that applications are automatically accessible on Android, as custom graphics must still be reflected with semantic components as we have described above.

Compared to the other systems, there are a number of quirks worth noting about the Quorum Accessible Semantics Layer’s web browser implementation. To understand these quirks, we will need to discuss an approach we have only discussed in name so far: shadow DOM.

4.5 Shadow DOM and Accessible Semantics in Web Browsers

Content for web pages is organized in a format known as the Document Object Model, or DOM. Elements in the DOM are both structural and semantically meaningful. For example, elements could represent landmarks like headers, text content like paragraphs, or user interface elements like buttons and text boxes. Importantly, elements in the DOM are hierarchically ordered, just like the semantic trees that are implicit to the accessibility APIs we map to on other platforms.

Content in the DOM is typically visual, as web browsers will parse the nodes to determine what should be displayed on the screen. Since the content is both semantically rich and exposed to clients, screen readers are designed to use the DOM to identify page structure, read information, and assist in page navigation. Shadow DOM, then, is a natural extension of an observation we noted earlier: the pixels on the screen are irrelevant to semantic meaning. In other words, it is possible to inject content into the DOM that is meaningful to screen readers, but will never be displayed visually. Using this, we can create shadow DOM elements that act as nodes in a conceptual semantic layer to reflect our graphical content.

There are some important caveats to this approach. When we custom render graphics in a web browser, we use a canvas element. As we have discussed, each element has semantic meaning – unfortunately, the semantic meaning for a canvas is essentially that there is a region where some unknown thing will be drawn, and that is fundamentally unhelpful to a screen reader. We suspect that is in part why shadow DOM content embedded within a canvas is not touched by accessibility clients, as they assume the element to be inaccessible. In our testing, we found that our shadow DOM semantic layer instead had to be placed next to the canvas in the hierarchy, which introduced some complexity under the hood in managing the web page’s focus to ensure clients received correct information while maintaining usability of the embedded application. Also, while DOM content is semantically rich, screen readers do not necessarily agree on all the details of how to interpret them – depending on browser, elements must be very carefully supplemented with additional Accessible Rich Internet Applications (ARIA) tags to produce intended results [46].

To further complicate matters, when considering our accessible application for web browsers, we must separately consider users on mobile devices. Such users will most likely not navigate using a keyboard, instead relying on touch and gesture input. This may require some additional thought, but is not inherently special to the web, since an application natively deployed to Android or iPhone would need to consider these inputs as well. Rather, the deeper problem is that our browser application might not even receive gesture

input. This issue, which Kluthe et. al (including the author of this thesis) discuss in deeper detail elsewhere [37], is known as “The Missing Gesture Problem.”

In short, when a user interacts with a web page while using a screen reader on a mobile device, they typically navigate using gestures (especially swiping). Unfortunately, these navigation gestures are intercepted by mobile device screen readers before our applications can ever receive them, which means we cannot react to the input or even know they have occurred. To resolve this, the shadow DOM structure must be further supplemented with injected hidden headers that act as landmarks for navigation. Compared to our other platforms, where the user engages with the Quorum logic and the semantic layer is updated to reflect it, it is somewhat backwards here – our mobile screen reader user on the web navigates to the nodes in the semantic model via gesture navigation, and as they interact with the nodes, changes must propagate upwards to the Quorum layer.

While this introduces additional complexity, the end result is a system which is capable of providing semantically rich information to accessibility clients across many different platforms. While this is a critical step, there are some additional considerations we will have to weigh in order to successfully solve the Accessible Graphics problem. In the next chapter, we will discuss some of these.

Chapter 5

Techniques for Accessible Semantics

In the previous section, we discussed the technical implementation of the Quorum Accessible Semantics Layer. The semantics layer provides a key contribution to accessible applications, allowing them to communicate with screen readers and other accessibility clients through different operating systems’ native accessibility APIs. These APIs have limitations, however, which require some additional considerations. In this section, we will discuss techniques to address two important problems: managing arbitrary graphics, and accessible starvation.

5.1 Arbitrary Graphics

In the previous chapter, we discussed how native accessibility APIs are designed to facilitate user interfaces. This means that mapping our graphical components to semantic ones is relatively simple, so long as our graphics reflect the supported control types of the API on a given operating system. This also implies a subtle but important detail: the mapping will be most straightforward if interactions with our graphical components match the expected interactions of components in the native accessibility API. That is to say, accessibility is easiest if our graphics can be interacted with like a user interface.

Of course, this is not always the case. For example, if we wanted to make a semantically accessible racing game, there is no common control type to represent a racecar. We also cannot reasonably expect that the game can be operated like a user interface – the user will likely need information dynamically, not just in relation to user interface events. Thus, to provide accessible semantics broadly, we need to consider two cases: providing semantics for arbitrary elements, and sending information to accessibility clients at arbitrary times.

5.1.1 Custom Nodes

The first problem is relatively simple to address. When there is no available control type that closely emulates our desired element, we instead opt for the simplest available type in the system, such as the “Custom” type

in UIA. While the actual names and details vary between APIs, for our discussion, we will consider semantic nodes using the smallest available type as “custom” semantic nodes. Across the supported APIs in the Quorum Accessible Semantics Layer, custom nodes universally provide the element’s name. Depending on the platform, they may also allow for a description. No further information is exposed in the semantic layer. This very light set of available information means the semantic layer cannot provide a great depth of information automatically, but it also means that the custom nodes are very flexible.

When interacting with custom semantic nodes (e.g., its related Quorum component has gained focus), screen readers reliably read the name of the element. Depending on the platform and screen reader, it may optionally read the description as well. Thus, to convey complicated elements, we can pack information into the custom node’s name. This could include the element’s identifying features, important details, help text or instructions, and more. For example, the name of our hypothetical racecar from earlier could be “racecar, 200 miles per hour, hold space bar to accelerate.”

5.1.2 Arbitrary Messages and the Two-Dummy Approach

Custom nodes also play an important role in solving our second problem. To return to our hypothetical racing game, we might want to inform the user if their car is approaching a wall or if the track turns ahead. This information is responsive to our game logic, but not to any underlying usage of user interface events that can be tracked in the semantic layer. In short, we need a way to send messages to accessibility clients on demand.

Some native accessibility APIs have built-in methods to achieve this. For example, on Windows the *Uia-RaiseNotificationEvent* command sends an event to accessibility clients to notify users. Unfortunately, this is not completely reliable. Support for this feature varies by screen reader, and in our testing, we found that operating system version also mattered – screen readers were much more likely to speak the event aloud on Windows 11 than on Windows 10. Even if we could confidently rely on this approach for Windows, it would not provide the means to send messages for other APIs that do not have a similar feature. With this in mind, a different approach is needed.

Our approach for sending arbitrary screen reader messages hinges on two key observations. First, screen readers will reliably read the names of custom nodes when they receive focus, as previously discussed. Second, what our semantic layer reports as being focused does not have to match what is actually focused in the graphical layer (if we have any concept of focus in that layer at all). In other words, we can “lie” to the accessibility layer in order to forward messages along.

To send an arbitrary message, we can create a custom node for a dummy element. We pack our message into the node’s name, and then we inform the semantics layer that this dummy has received focus. At our convenience, we may choose to let the dummy element receive the focus in the graphical layer as well, or leave the focus with a previous element. As far as the screen reader is concerned, only the semantic focus matters – what is on the screen is irrelevant to the semantics model, after all.

If we want to send sequential messages to the screen reader, we can create a second dummy element. We bounce the semantic focus back and forth between the two dummies, changing each dummy’s name property just before it receives focus. Using the “two-dummy” approach, we can use the native accessibility APIs’ user interface types for any elements which closely correspond to them, and then handle all other arbitrary screen reader output for the entire program with two dummy providers. Thus, we can use traditional controls wherever it is convenient, and use dummies wherever it is not.

5.2 Accessible Starvation and Racing Screen Readers

As we have discussed, the Quorum Accessible Semantics Layer uses an event-driven request/response model to facilitate messages between applications and accessibility clients. While the fine details vary across platforms, broadly an application receives messages from screen readers and other clients via the application window’s message pump. The pump also usually handles common events like input (e.g. mouse or keyboard events) and window events (e.g. resizing, moving, or requesting to close the window).

For a graphical application, there is a natural incentive to spend as little time as possible in the message pump. Time saved here can be better spent on the core logic and in rendering. Thus a typical graphical application will process the contents of the pump once per frame of animation. The timing of the message pump can be further constrained by other graphical goals. For example, using vSync to synchronize our rendering to the refresh rate of a monitor will also impact the amount of time that can be spent processing messages between frames.

Unfortunately, if we apply this common approach to handling the message pump, users may experience serious delays between interacting with the application and receiving a response from a screen reader. In our testing, we found that each keystroke could take approximately 10 seconds before the screen reader speaks in response. In especially severe circumstances, screen readers might stop responding entirely. In other words, the performance of the accessibility system becomes so poor that the application essentially ceases to be accessible, despite the fact that our interface to the native accessibility API is correctly implemented. This phenomenon, identified by Stefik et al. (including the author of this thesis), is known as “accessible starvation” [74].

5.2.1 Accessible Starvation

Accessible starvation occurs when the accessibility system has insufficient resources to perform effectively – in this case, insufficient access to the message pump. To understand why this occurs, it is important to consider the different expectations of the graphics and accessibility systems.

In a non-semantically accessible graphics application, we broadly expect that we must react to events, but we do not need to dialog with the source of the event. For example, if the user clicks the mouse, the application reacts to it, and the event is handled. There is no need to maintain a continuous line of

communication. Screen readers and other assistive technologies do not work this way.

Screen readers will frequently follow-up on requests to gather more information. For example, if a user changes the focus, the screen reader might send a request asking what was focused. If it identifies that the focused element is a tree, it could send further requests regarding the structure of the tree, the number of items, etc. For each individual tree item, it might also request further information. Thus, a single event might produce hundreds or thousands of asynchronous messages from the screen reader that our application must respond to.

Now, imagine in our tree example that between each request we rendered an entire frame of animation. This is what occurs in our typical graphical case: we process the messages that are present, allow no time for follow-up messages, and immediately return to the core logic and rendering of our application. Because each follow-up request is dependent on a previous one, this cascades into a delay where the screen reader must wait for dozens or hundreds of frames of animation before it has gathered everything it needs, which would then trigger one, and only one, speech request.

Fortunately, there is a simple method available to prevent this accessible starvation. While using the Quorum Accessible Semantics Layer, we poll the message pump for a short period of time between frames of animation. We use a configurable target frame rate to set the upper bound for the polling period, and set a minimum delay time per frame as our lower bound. This dedicated polling period allows for many requests, responses, and follow-ups to be processed between frames and can be turned on or off.

The downside to this approach is that with accessibility turned on, responding in a timely manner to clients requires capping the frames per second. What the cap should be depends on the overall performance of the application, but in our testing we found accessible starvation could be effectively mitigated with only minimal time dedication – by default, the system targets 70 frames per second, and enforces a minimum delay of 0.01 seconds. Importantly, this polling period can be disabled entirely for users that are not using assistive technologies. Thus, the solution would have no impact on users that can see, while blind users only have to set a frame cap to make it work.

5.2.2 Racing the Screen Reader

The message pump provides an additional complication for us. Screen readers and other assistive technologies send requests via the message pump on the main thread. This is the same place where we process input events from the operating system. In an ideal case, a screen reader would not request information pertaining to a keystroke until we were done processing it in our game logic, but this is not necessarily the case. Further, while we describe here a “main” thread, note that screen readers vary by manufacturer how they handle threads. For example, during testing we found that JAWS required requests to be handled on the main thread, but NVDA did not.

Screen readers often send context-sensitive requests based on operating system input. For example, if the screen reader knows a text box is focused and the right arrow is pressed, it may ask for information about

the text in the new cursor position, even if we have not informed it of a selection change yet. As a result, we need to be careful that our accessibility polling period does not result in waiting too long to process the input in the core application logic, or we risk giving the screen reader old information and reading the wrong information to the user.

To resolve this, we end our polling period early if we receive an input event. In this case, we process only the messages remaining in the queue and do not poll for additional events, instead moving back into the core logic and rendering stage of the application. While input events may seem common, they are not overwhelmingly so: we expect with typical usage, most polling periods will not be interrupted by input, and the overall impact on the accessible performance is negligible. As an additional side effect, this method means that users do not experience input delay due to the polling period.

Chapter 6

Examples of Semantic Accessibility: Code, Blocks, Scenes, Charts & Games

In the previous chapters, we have discussed at length the need for, and benefits of, semantic accessibility. We have discussed how it can be implemented, and discussed its use in largely hypothetical terms. In this chapter, we will demonstrate concretely that accessible graphics can be made with the Quorum Accessible Semantics Layer. We will discuss several examples of applications made with Quorum which are both fully visual and fully accessible to individuals with disabilities, ranging across development tools to data visualization to games.

6.1 Quorum Studio

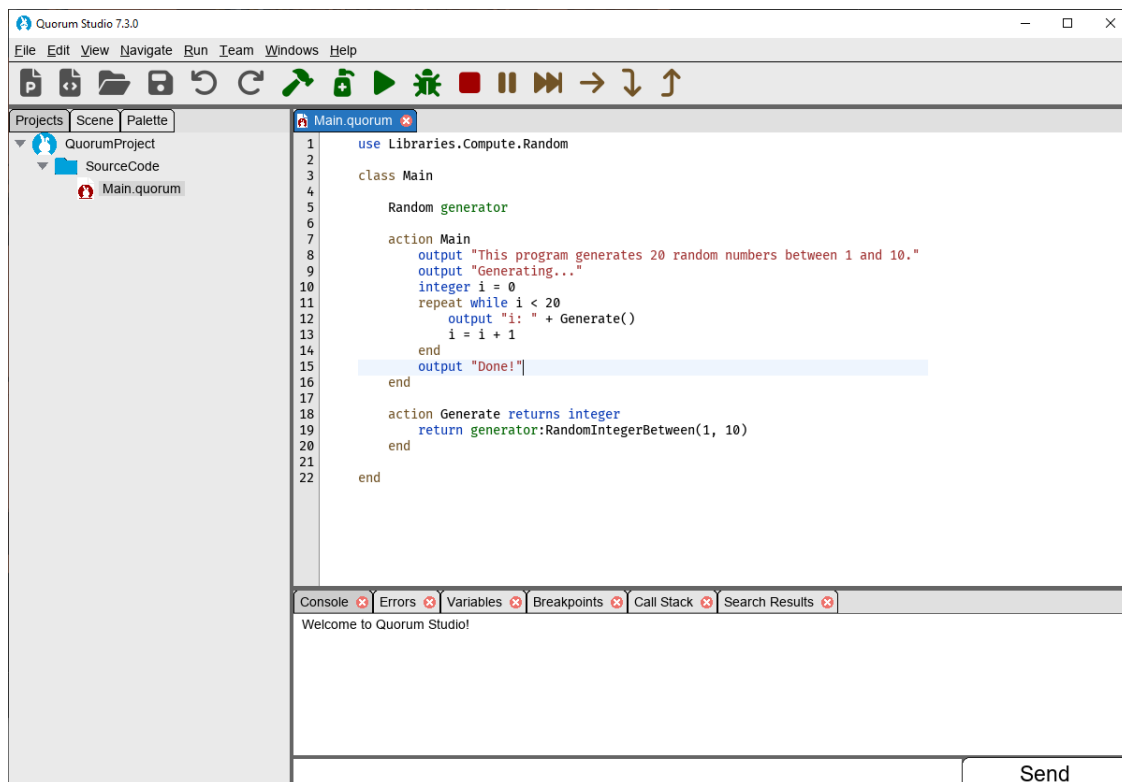
Quorum Studio is an open source integrated development environment for the Quorum programming language, which supports both traditional text programming as well as more visually focused block programming [39]. It also contains a scene editor for visual 2D and 3D game development, inspired in part by systems like Unity3D [75]. Quorum Studio is rendered from scratch using the Quorum game libraries, and provides semantics via the Quorum Accessible Semantics Layer. The result is a visually complete tool which is accessible to people with limited or no vision. Quorum Studio is used today, primarily in schools in the United States, and especially schools for the blind or visually impaired. Figures 6.1, 6.2, 6.3, 6.4, and 6.5 all showcase elements of Quorum Studio.

6.1.1 The Integrated Development Environment

The integrated development environment resembles a traditional user interface application, and uses traditional interface elements. Figure 6.1 shows a screenshot of the editor. It includes a menu, a toolbar with buttons, a project tree, a text editing region, and a region for console output or debugging information. All of these are actually custom rendered elements in the Quorum game engine, and each control has an auto-

matically generated semantic node to communicate with screen readers via the Quorum Accessible Semantics Layer. The various controls presented to users correspond closely to traditional control types, and overall the application does not feel remarkably different to use than any other accessible tool. This is intentional – the goal of semantic accessibility is to allow for natural interaction for users that need it, and to remain invisible to users that do not.

Figure 6.1: A screenshot of Quorum Studio. It showcases a simple program being edited as text in a traditional IDE.



One thing that is worth noting in this section, however, is the use of the two-dummy approach. While there is no need to use this technique for the IDE’s controls, it is useful to respond to certain events. For example, if a user loads a project, dummy nodes can be used to report when the project has finished loading.

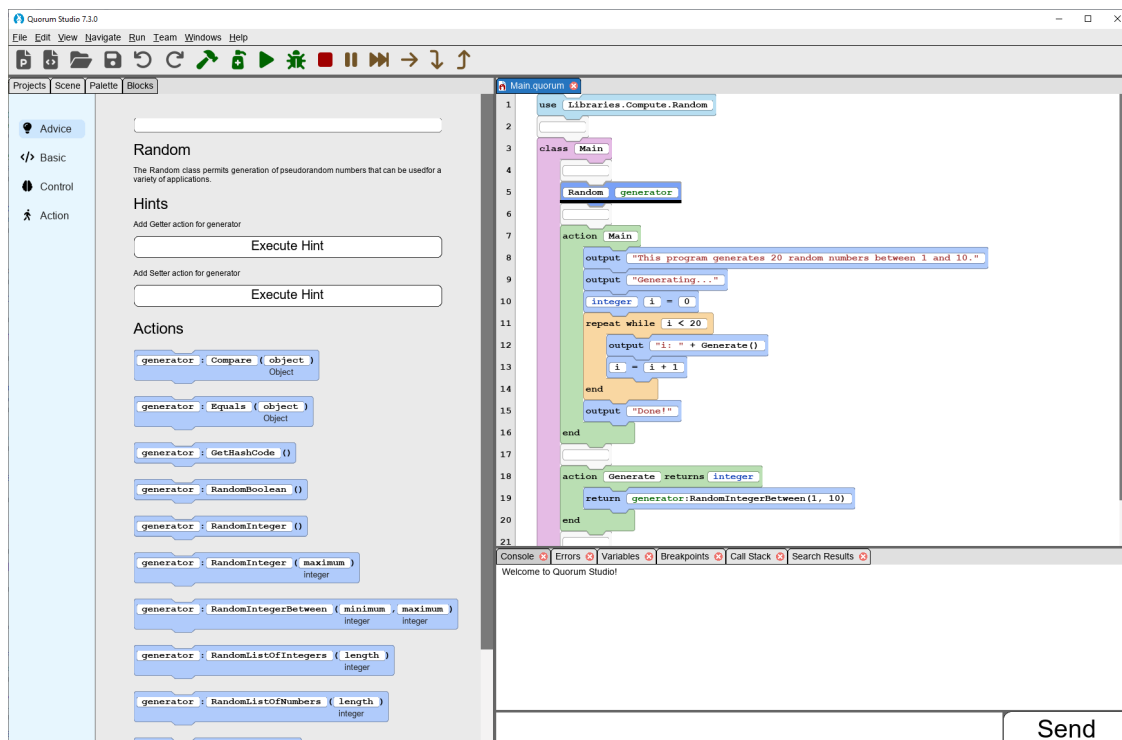
6.1.2 Block Programming

For users who prefer it, Quorum Studio also supports block programming – Figure 6.2 shows a screenshot of Quorum Studio in block editing mode. The same program shown previously in our text editing example is again on the right side of the screen, but this time the code is displayed as blocks rather than text. The palette on the left side has four categories of blocks to choose from, three of which contain common, novice-friendly blocks. The category showcased in Figure 6.2 is the “Advice” tab, which provides blocks related to

the currently selected block in the editor. It also includes a brief description of the selected block taken from standard library documentation, as well as available editor hints for that block.

Conceptually, block programming raises a number of challenges towards accessibility. Such systems are highly visual, relaying information like scope through visual presentation. Additionally, blocks are often custom-rendered shapes that do not closely align with common user interface elements, and so there are no common affordances for screen readers or other accessibility clients to interact with them. In many systems, blocks are primarily interacted via drag-and-drop using the mouse or touch, and do not offer alternative inputs like keyboard support. As a result, block programming can also be difficult to use for individuals with motor impairments.

Figure 6.2: A screenshot of Quorum Studio in block mode. The file on the right side is displayed as editable blocks, and the left side shows a palette with available blocks.



In Quorum Studio, blocks represent lines of text. Most blocks represent a single line, though scoping structures like loops and conditionals comprise multiple. The system supports both mouse and keyboard, with the mouse supporting typical drag-and-drop operations. While using the keyboard, the user operates on a single line at a time, moving the selection up and down to navigate the blocks. Hotkeys allow the user to jump into the editable regions of blocks, or to navigate by structures (e.g. navigating to next/previous scope or function). Hotkeys can also be used to turn a block into a line of raw text for direct modification. Overall, this approach allows individuals to act with the larger discrete blocks or operate as though using a

raw text editor at their convenience.

Importantly, blocks can communicate with accessibility clients via the Quorum Accessible Semantics Layer. Each block uses a custom semantic node, which contains the block’s line of text as the node’s name. Blocks may also contain other information when relevant, e.g. scope level for scoping structures or a message if a line contains an error. It is not necessary to use additional dummy providers in this case, as we do not need to provide messages at arbitrary times – we can simply allow screen readers to request the information when focus changes.

6.1.3 The Visual Scene Editor

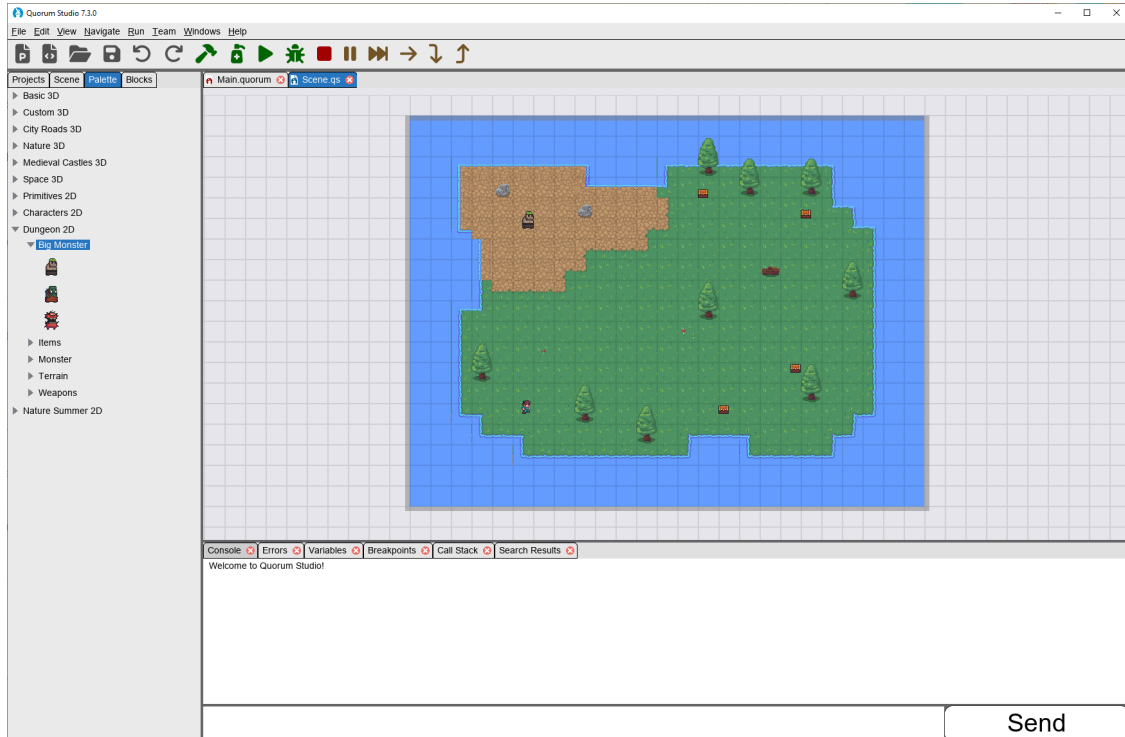
The scene editor presents a more complex set of problems. While blocks do not have a directly analogous UIA control type, we can still model their usage after commonly accessible patterns, like text editors or fillable forms. We use custom semantic nodes for blocks that emulate lines of text, and no custom nodes are needed at all for their sub-components – text fields are standard and accessible, after all. In some ways, we can consider the block editor to be a natural extension of existing accessible controls.

In contrast, the visual scene editor does not afford us the same luxury. There is no standardized way to interact with complex graphical scenes. For 3D graphics, there are no analogous control types in any of the native accessibility APIs the Quorum Accessible Semantics Layer implements. Even if we had a control type for each element in a vacuum, screen readers would not necessarily know how to convey useful information for our specific application. Imagine for a moment that we have a 3D model of a tank. This 3D model could be part of a first person shooter, or a turn-based strategy game, or a 3D modeling application. Each of these applications would need to provide very different information. Even within a single application, the differences could be vast – if a game features a dragon sitting next to a tree, those two models likely should not present the same kinds of information to a player.

To be clear, the purpose of this section is not to discuss what the ideal design for such systems should be. Investigating the human-computer interactions of accessible 3D graphics could fill whole series of papers. Rather, the focus here is to highlight that such problems exist, and to show a proof of concept that such systems are possible to make. For now, our goal is to demonstrate that even complicated interactive graphics can be made accessible. To that end, Figures 6.3, 6.4, and 6.5 all show examples of Quorum Studio’s scene editor.

The visual editor is explicitly designed to be fully navigable using both the mouse and keyboard. To facilitate understanding and navigation of the scene, the user is provided with a few tools. First, the project tree contains each item in the scene, and it can be used to jump to particular objects using first letter navigation (jumping to items with the keyboard, a common technique with blind users). To let the user search more intuitively, we also place a selection cursor into the scene, which the user can move, expand, and shrink with the keyboard. As the cursor changes, the screen reader speaks aloud its new position and what objects it is currently in contact with. The user can tab between the cursor and the objects it is touching to

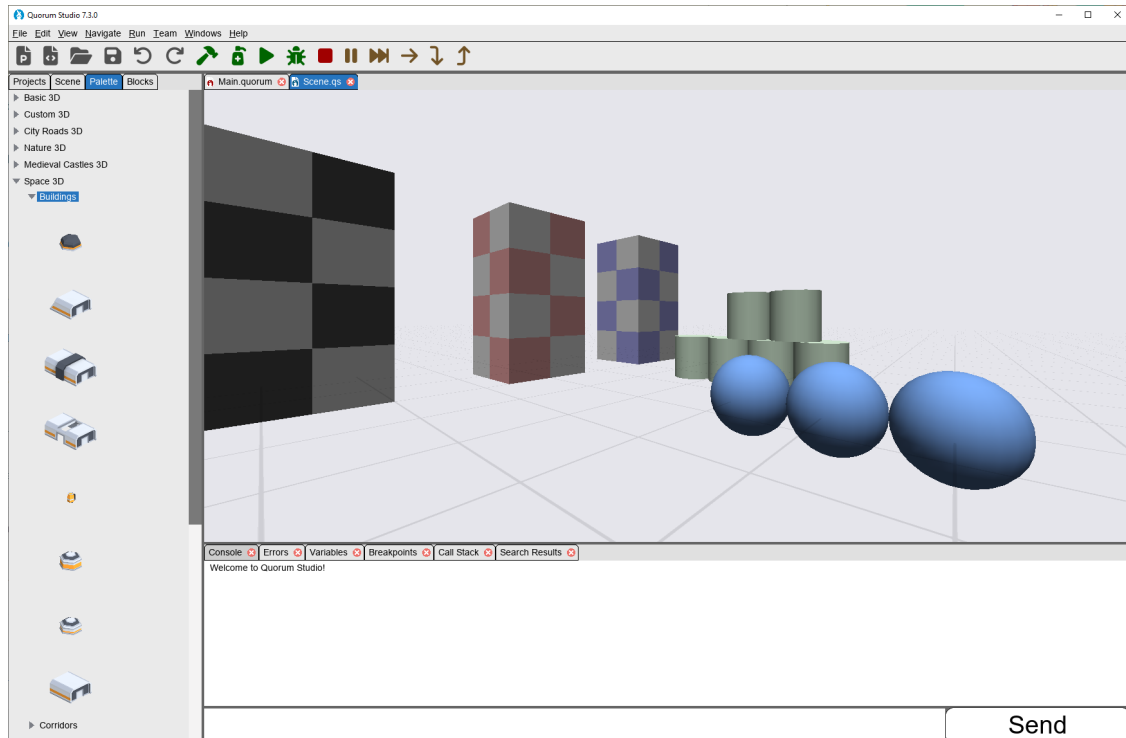
Figure 6.3: A screenshot of Quorum Studio’s scene editor. This screenshot shows a 2D scene, where the user has created an island which is dotted with trees, treasure chests, and some characters.



quickly check properties or make changes in the area. An item’s full properties can be viewed and adjusted in the properties window, which dynamically updates to match the focused item – this window can be seen in the bottom right of Figure 6.5. These properties are exposed through standard, accessible controls and are arranged like a form. Keyboard shortcuts in the editor also provide context-sensitive selections, e.g. select all items in view of the camera.

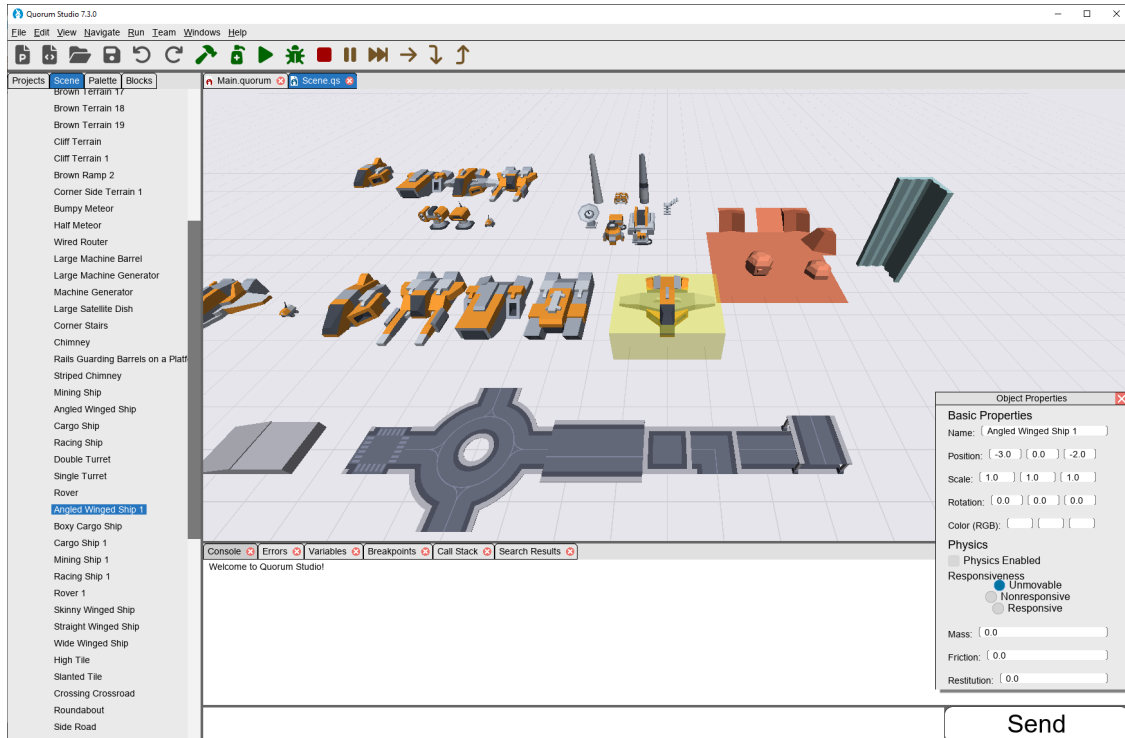
While the scene editor separates out 2D and 3D scenes as distinct applications, in practice both work fundamentally the same way. Because there are no standard control types for 2D or 3D scene objects, we must rely on custom semantic nodes to represent these elements. As for the cursor within the scene, we need to both represent a non-standard control, and we need to convey information at non-standard times. Thus, we report information to the screen reader using the two-dummy approach we outlined in the previous chapter. Using the Quorum Accessible Semantics Layer, users with screen readers can interact with the system in a meaningful way. Meanwhile, users without assistive technologies can use it visually with the mouse, likely without even being aware that a blind user can use it as well. Again, we do not claim that we have implemented “perfect” accessibility for a visual scene editor – there is undoubtedly further room for research. What we are claiming is that, through the tools and techniques discussed in this thesis, it is possible to create a 2D and 3D scene editing application that is blind accessible, fully supports screen readers

Figure 6.4: A screenshot of Quorum Studio's scene editor. This screenshot shows a 3D scene with a series of primitive objects placed into the level. The palette on the left shows available models which can be placed into the scene.



and other accessibility clients, and remains viable for sighted users.

Figure 6.5: A screenshot of Quorum Studio’s scene editor. This screenshot shows a 3D scene with many different models placed from the palette into the environment. The selection cursor is highlighting a ship in the middle of the scene, which is also selected in the list of scene elements in the tree on the left. A properties window in the bottom right allows for adjustments to the selected model.



6.2 Charts and Data Visualization

Independently of Quorum Studio, the Quorum Accessible Semantic Layer can be used to provide semantic accessibility to any Quorum graphical application. One particularly useful case of this is for charts and data visualization. Quorum can produce interactive charts that are automatically semantically accessible, providing much deeper information than would be possible with alt-text [36]. Figures 6.6, 6.7, and 6.8 show examples of semantically accessible charts using the tools and techniques described in this thesis.

While at a glance each chart would appear to be an image, in the semantic layer each one is structured as a tree instead. Screen reader users can use the keyboard to navigate between the top-level elements such as the main content region, the legend, or the axes, then drill down to find more specific information. Sighted users may also explore details of the data using the mouse. For example, in Figure 6.7, a single point has been focused, which reports the point’s coordinates to screen readers and also displays the information visually. Similarly, Figure 6.8 has a single wedge selected, which reports that information to the user semantically and visually.

For keyboard users, hotkeys are provided to navigate in multiple ways. For example, a user could navigate

the elements linearly in the order they appear, or instead navigate them from largest to smallest or vice versa. Importantly, all of the chart's interactivity, as well as its semantics, are generated automatically when the chart is created. No additional effort is necessary on the part of the chart's creator – accessibility is guaranteed simply for the visualization existing in the first place, with less effort than it would even take to write an alt-text description.

Figure 6.6: A screenshot of an accessible, interactive bar chart made with Quorum.

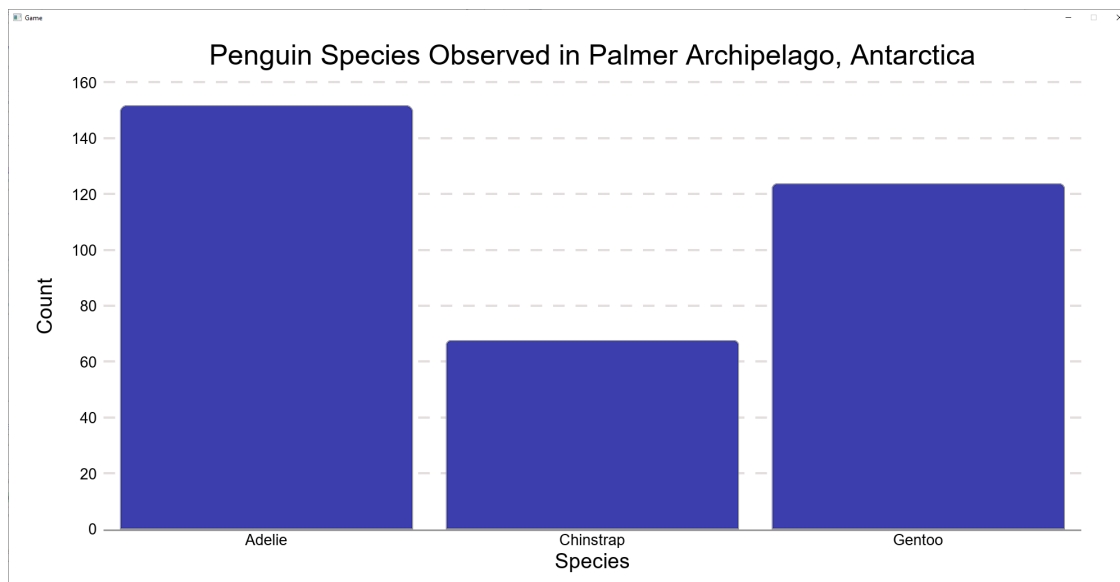


Figure 6.7: A screenshot of an accessible, interactive scatter plot made with Quorum.

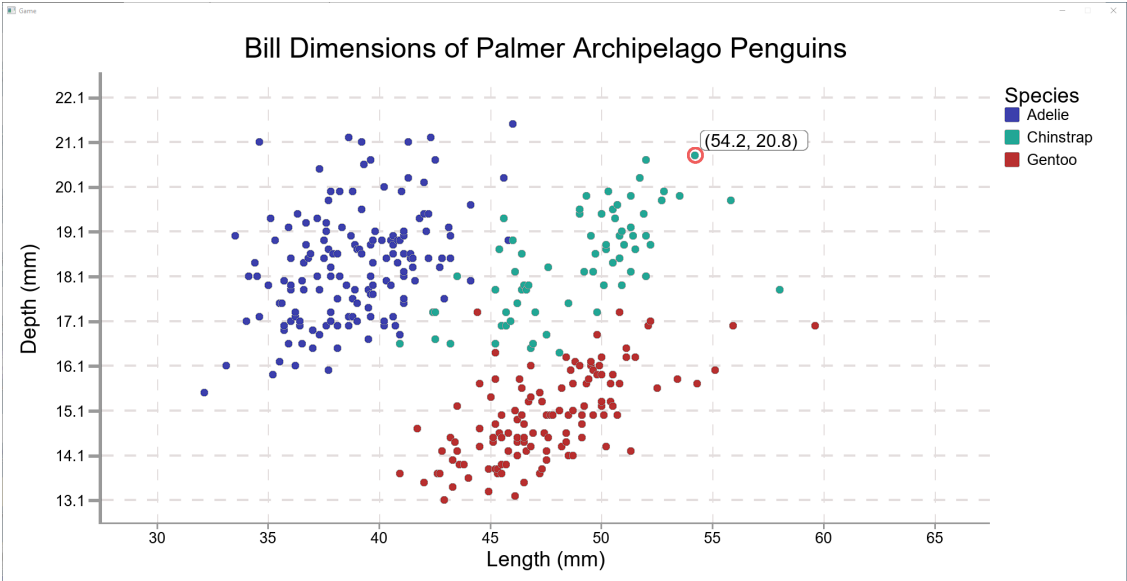
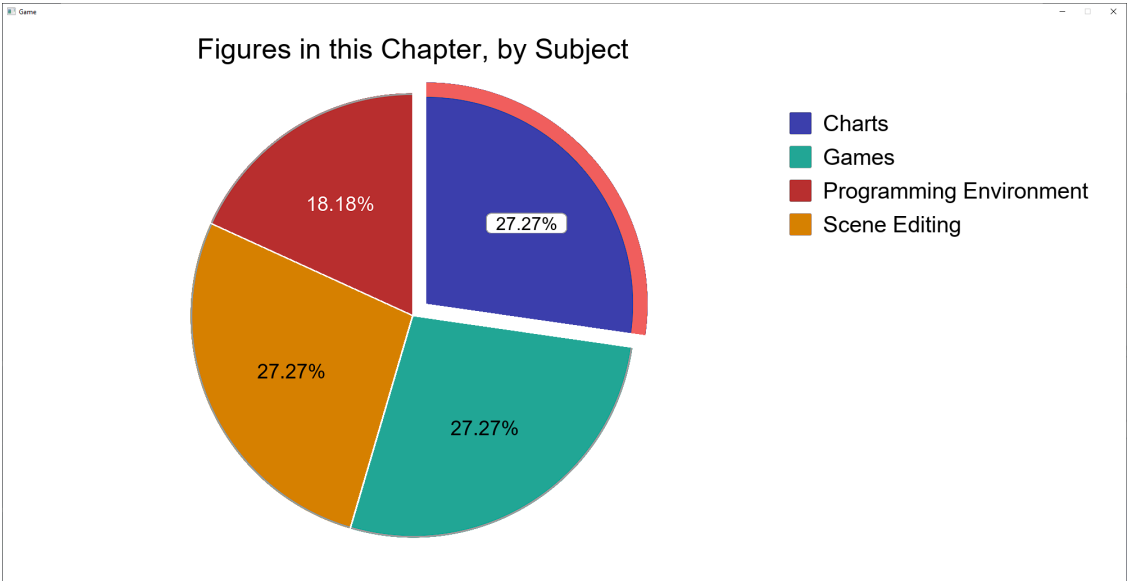


Figure 6.8: A screenshot of an accessible, interactive pie chart made with Quorum.



6.3 Semantically Accessible Games

Finally, here we will discuss a particularly complex example of accessible graphics: games. While a substantial amount of research around accessible gaming exists [25], to our knowledge the Quorum Accessible Semantics Layer is the first general-purpose system for providing screen reader and accessibility client compatibility to hardware accelerated graphical games. In this section, we will highlight three semantically accessible games made by the author of this thesis as part of the annual Experiencing Programming in Quorum (EPIQ) conference [40].

6.3.1 Ground Flight Tactics

Figure 6.9 showcases Ground Flight Tactics, a 3D turn-based tactics game. The game was created using Quorum Studio’s 3D scene editor, and takes place on a two-dimensional grid. The player controls a fighter plane which hovers along the ground, directing it to move along the grid spaces and to fire upon enemies. When the player ends their turn, the enemies move and attack. The game is made up of three stages, and to win the player must navigate to each level’s generators, activate them, and reach the escape portal – without being destroyed by enemy forces.

Ground Flight Tactics is complex, and requires a substantial amount of information available to the user in order to play. The game makes extensive use of custom semantic nodes to convey this information to screen readers. The player controls a cursor which can be navigated across the grid space using the arrow keys. As the cursor moves, the two-dummy approach is used to signal information about the grid space. This includes key information, like the presence of enemies or important landmarks, but it also includes thematic details, e.g. if the tile is dusty dirt or industrial concrete. These details provide contextual clues to help mentally map the space, as well as simply enriching the landscape – while we could claim the information is unnecessary to play, we could also argue that accessibility should not remove “fun” from the game.

In order to easily assess the state of the game grid, keyboard users can tab through key elements to quickly move the cursor to them. This allows players to quickly identify enemies and points of interest, as well as their location on the grid. Because the game is turn-based, users can spend as much or as little time as they would like to inspect the play area before committing to actions. Overall, the game’s design heavily leverages the use of the grid space, then provides semantics to access it.

6.3.2 Block Breaker

Figure 6.10 shows a screenshot of Block Breaker, a brick-breaker style game where the user must aim a ball to bounce and break up blocks. The game is predominantly 2D, though the space background is actually a 3D skybox which slowly rotates behind the main game area. The game is made up of five levels. Each level contains a number of “bug” blocks, which must be destroyed to complete the level, and “gold” blocks, which give the player coins. In between levels, the player can spend coins to gain various upgrades, including more

Figure 6.9: A screenshot of Ground Flight Tactics, an accessible tactics game made with the Quorum Studio 3D scene editor.

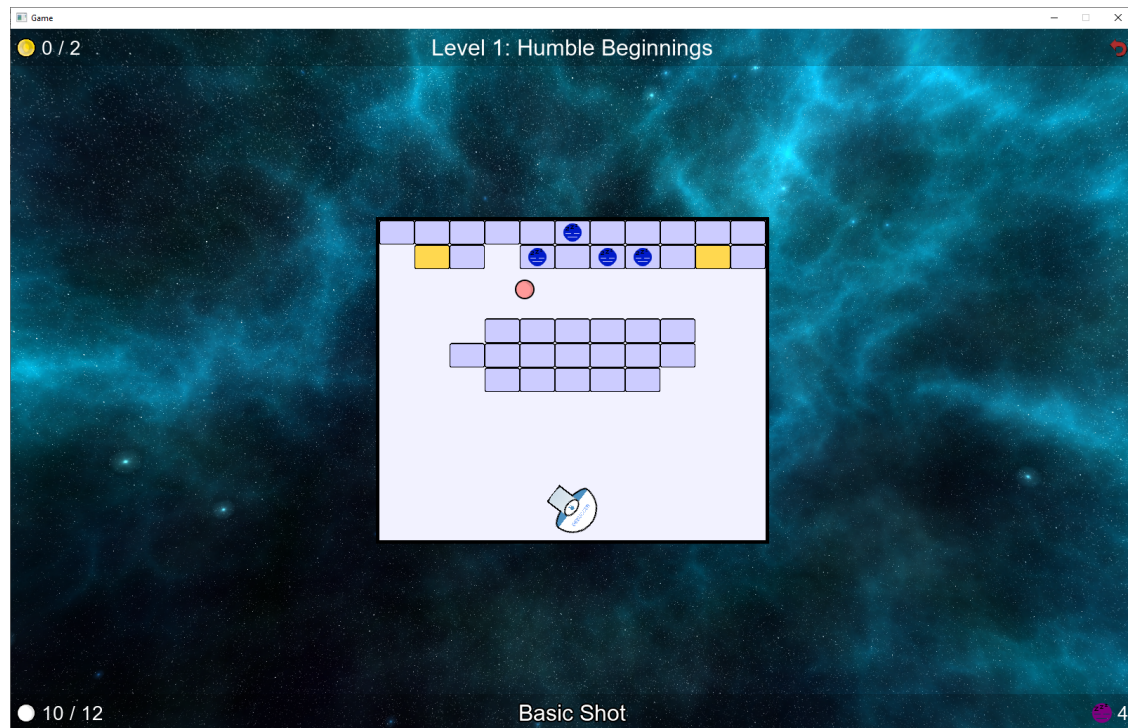


shots, more bounces, or a “shotgun” style cannon.

The player controls the cannon at the bottom of the screen, which is fixed in place. To aim, the cannon must be rotated. Semantically, when the cannon rotates, two-dummy is employed to inform the user of its new firing angle. Similar to Ground Flight Tactics, the game area leverages a grid to store semantic information. The grid can be navigated square-by-square by going into an inspect mode available through the game menu or via hotkey. Also like Ground Flight Tactics, hotkeys are available to jump to priority elements while in this inspect mode. In addition to information given at each cell in the grid, the cursor gives a brief summary of elements in the column. This makes it easier to scan the play area to determine the relative density of where high-priority blocks are located.

In addition to semantic information, this game relies on supplementary audio cues. When the user fires the ball, it makes a different sound for any surface it impacts – e.g., it sounds different if it hits the outer wall, a wall block, or a “bug” block. This gives a sense of what the ball is doing in real time without needing to send additional screen reader messages.

Figure 6.10: A screenshot of Block Breaker, an accessible brick-breaker style game incorporating 2D graphics and a 3D skybox.



6.3.3 Melvin Quest

Figure 6.11 showcases Melvin Quest, a 2D exploration adventure game. The game focuses on the titular Melvin, a wizard who has had a rough night out and needs help returning home. The game’s emphasis is on environment, navigation, and dialog. The environment is spread across four maps built with Quorum Studio’s 2D scene editor.

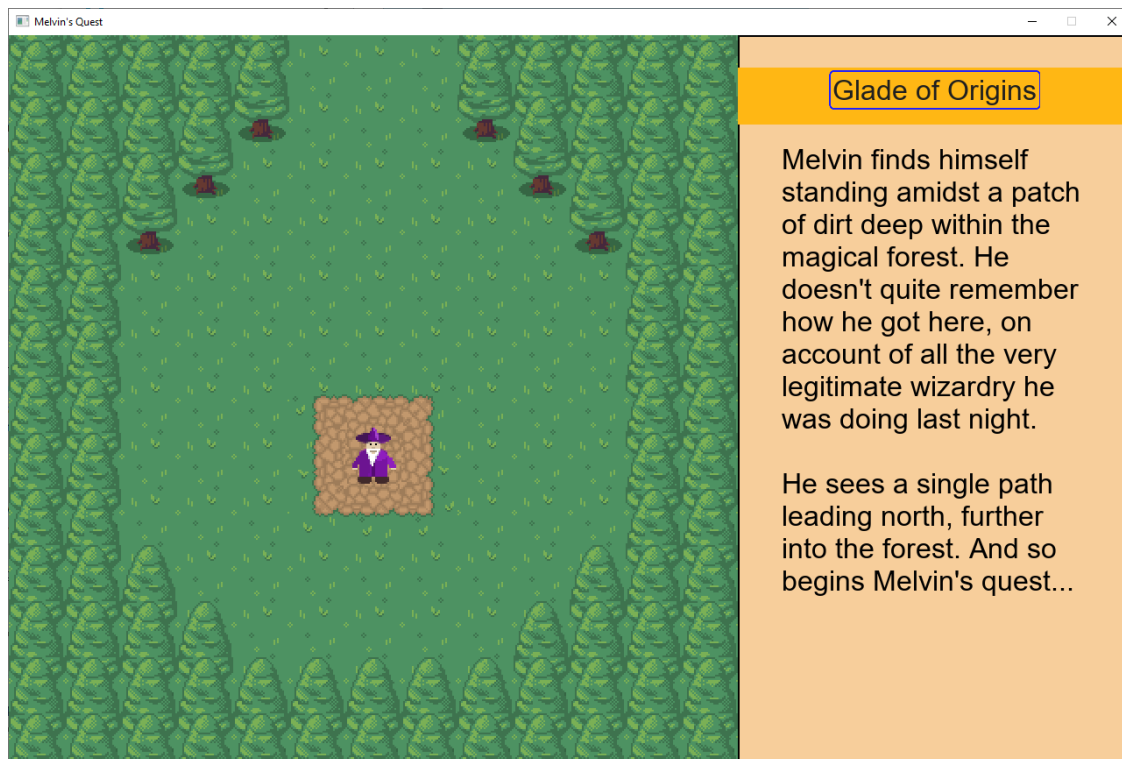
As the player navigates Melvin through the maps, the dialog pane on the right updates with new information, which is relayed to screen readers via semantic dummy nodes. This new information includes silly dialog and generally makes up the game’s humor, but it also includes important directions. For example, in Figure 6.11, the dialog indicates “a single path north”, indicating that the current map has a single exit at the top of the screen. While this is visually obvious, it provides important clues to users who cannot see the screen. These clues are embedded throughout the entire game to provide an accessible description of each map and the player’s next step as they progress.

Like Block Breaker, this game uses supplemental audio cues to provide additional context. Melvin makes sound as he moves, and the sound differs between the different surfaces he walks across (e.g., walking across grass or dirt). He also makes different noises when he walks into obstacles or interactive objects, such as signs he can read. Overall, the intent is to provide information through a combination of modalities to make

the game accessible broadly.

Notably, Melvin Quest – as well as Ground Flight Tactics and Block Breaker – were all created using Quorum Studio. In other words, not only are they accessible games, but they were made with accessible tools. This means that, using the contributions in this thesis, it is not only possible for people who are blind or visually impaired to use these graphical applications, but to create them as well.

Figure 6.11: A screenshot of Melvin Quest, an accessible exploration adventure game built with the Quorum Studio 2D scene editor.



Chapter 7

Conclusion

The Accessible Graphics Problem is complex and widespread. As we have discussed, those with limited or no vision are often excluded from using graphical applications – and by extension, excluded from professions, education, and experiences. Without standardized or general-purpose solutions, developers have little choice but to address this issue themselves with bespoke measures. Unfortunately, self-voicing and similar per-application custom solutions have severe drawbacks, especially that they cannot communicate with screen readers or other accessibility clients.

To address this problem, we have introduced the Quorum Accessible Semantics Layer, which allows applications to provide semantic models to the native operating system and communicate with accessibility clients. We discussed the implementation of the Quorum Accessible Semantics Layer, as well as additional techniques necessary for applications to provide reliable semantic accessibility. Finally, we have discussed several examples of accessible applications produced with the Quorum Accessible Semantics Layer, showcasing how many kinds of graphics can be made accessible.

Graphics are an ever-present part of our modern lives, facilitating and enriching countless applications across diverse fields. Through the work in this thesis, we hope to illustrate that such benefits do not need to come at the cost of those who cannot see. While there is no doubt that much research remains to be explored in this area, it is at least plausible that many graphical applications can be made accessible. Those with limited or no vision can be meaningfully included – and they should be.

Bibliography

- [1] Juan Aguado-Delgado et al. “Accessibility in video games: a systematic review”. In: *Universal Access in the Information Society* 19.1 (Mar. 2020), pp. 169–193.
- [2] David Bar-El et al. “Tangicraft: A Multimodal Interface for Minecraft”. In: *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS '18. Galway, Ireland: Association for Computing Machinery, 2018, pp. 456–458. ISBN: 9781450356503. DOI: 10.1145/3234695.3241031. URL: <https://doi.org/10.1145/3234695.3241031>.
- [3] American Foundation for the Blind. *Refreshable Braille Displays*. Accessed on March 31st, 2026 from <https://afb.org/node/16207/refreshable-braille-displays>. 2026.
- [4] Matthew Butler et al. “Technology Developments in Touch-Based Accessible Graphics: A Systematic Review of Research 2010-2020”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445207. URL: <https://doi.org/10.1145/3411764.3445207>.
- [5] Government of Canada. *Accessible Canada Act*. Accessed on April 22nd, 2026 from <https://laws-lois.justice.gc.ca/eng/acts/a-0.6/>. 2026.
- [6] Raouf Chebka and Fathi Essalmi. “A crosswords game for deaf”. In: *2015 5th International Conference on Information & Communication Technology and Accessibility (ICTA)*. 2015, pp. 1–6. DOI: 10.1109/ICTA.2015.7426880.
- [7] United States Code. *Americans with Disabilities Act of 1990*. Retrieved March 27th, 2026 from <https://www.ada.gov/law-and-reg/ada/>.
- [8] SolidWorks Corporation. *SolidWorks*. Accessed on April 1st, 2026 from <https://www.solidworks.com/>. 2026.
- [9] Laisa C.C. De Biase et al. “An Accessible Roller Coaster Simulator for Touchscreen Devices: An Educational Game for the Visually Impaired”. In: *2018 IEEE Games, Entertainment, Media Conference (GEM)*. 2018, pp. 101–105. DOI: 10.1109/GEM.2018.8516457.
- [10] U.S. Department of Education. *Individuals with Disabilities Education act*. Retrieved April 22nd, 2026 from <https://sites.ed.gov/idea/>.

- [11] U.S. Department of Education. *Section 504 of the Rehabilitation Act*. Retrieved April 22nd, 2026 from <https://www.ed.gov/laws-and-policy/individuals-disabilities/section-504>.
- [12] Blizzard Entertainment. *Combatting Demons with Accessibility in Diablo IV*. Accessed on February 26th, 2026 from <https://news.blizzard.com/en-us/article/23954932/combating-demons-with-accessibility-in-diablo-iv>. 2023.
- [13] W. Erickson, C. Lee, and S. von Schrader. *2018 Disability Status Report: United States*. Ithaca, NY: Cornell University Yang-Tan Institute on Employment and Disability (YTI)., Retrieved from www.disabilitystatistics.org. 2020.
- [14] ETSI. *EN 301 549*. Accessed on April 22nd, 2026 from https://www.etsi.org/deliver/etsi_en/301500_301599/301549/03.02.01_60/en_301549v030201p.pdf. 2021.
- [15] Yansong Feng and Mirella Lapata. “Automatic Caption Generation for News Images”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.4 (2013), pp. 797–812. DOI: 10.1109/TPAMI.2012.118.
- [16] AD Flaxman et al. “Prevalence of visual acuity loss or blindness in the US”. In: *JAMA Ophthalmology* (2021).
- [17] Blender Foundation. *Blender*. Accessed on April 1st, 2026 from <https://www.blender.org/>. 2026.
- [18] Raspberry Pi Foundation. *Blockly*. Accessed on April 1st, 2026 from <https://blockly.com/>. 2026.
- [19] Sara Goering. “Rethinking disability: the social model of disability and chronic disease”. In: *Current reviews in musculoskeletal medicine* (2015).
- [20] Google. *android.view.accessibility | Android Developers*. Accessed on January 22nd, 2022 from <https://developer.android.com/reference/android/view/accessibility/package-summary>. 2021.
- [21] Google. *Get started on Android with TalkBack*. Accessed on January 24th, 2022 from <https://support.google.com/accessibility/android/answer/6283677>. 2022.
- [22] Jenna L. Gorlewicz et al. “Design Guidelines and Recommendations for Multimodal, Touchscreen-Based Graphics”. In: *ACM Trans. Access. Comput.* 13.3 (Aug. 2020). ISSN: 1936-7228. DOI: 10.1145/3403933. URL: <https://doi.org/10.1145/3403933>.
- [23] Andreas Grabski et al. “Kinaptic - Techniques and insights for creating competitive accessible 3D games for sighted and visually impaired users”. In: *2016 IEEE Haptics Symposium (HAPTICS)*. 2016, pp. 325–331. DOI: 10.1109/HAPTICS.2016.7463198.
- [24] Alan Gutterman. *Definitions and Models of Disability*. July 2023.
- [25] Lobna Hassan. “Accessibility of games and game-based applications: A systematic literature review and mapping of future directions”. In: *New Media & Society* 26.4 (2024), pp. 2336–2384. DOI: 10.1177/14614448231204020. eprint: <https://doi.org/10.1177/14614448231204020>. URL: <https://doi.org/10.1177/14614448231204020>.

- [26] Leona Holloway, Kim Marriott, and Matthew Butler. “Accessible Maps for the Blind: Comparing 3D Printed Models with Tactile Graphics”. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–13. ISBN: 9781450356206. URL: <https://doi-org.ezproxy.library.unlv.edu/10.1145/3173574.3173772>.
- [27] Jared S. Hopkins. “New Touch Technologies Bring Screen Time to the Blind”. In: *The Wall Street Journal* (2024).
- [28] MD. Zakir Hossain et al. “A Comprehensive Survey of Deep Learning for Image Captioning”. In: *ACM Comput. Surv.* 51.6 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3295748. URL: <https://doi.org/10.1145/3295748>.
- [29] Apple Inc. *Accessibility - Vision - Apple*. Accessed on January 24th, 2022 from <https://www.apple.com/accessibility/vision/>. 2022.
- [30] Apple Inc. *Accessibility Programming Guide for OS X: The OS X Accessibility Model*. Accessed on January 22nd, 2022 from <https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXmodel.html>. 2018.
- [31] Apple Inc. *UI Accessibility | Apple Developer Documentation*. Accessed on January 22nd, 2022 from <https://developer.apple.com/documentation/objectivec/nsobject/uiaccessibility>. 2022.
- [32] Autodesk Inc. *Autodesk Maya*. Accessed on April 1st, 2026 from <https://www.autodesk.com/products/maya/overview>. 2026.
- [33] Epic Games Inc. *Unreal Engine*. Accessed on April 1st, 2026 from <https://www.unrealengine.com/en-US>. 2026.
- [34] SAS Institute Inc. *SAS Graphics Accelerator Customer Product Page*. Accessed on January 25th, 2022 from <https://support.sas.com/software/products/graphics-accelerator/>. 2022.
- [35] Chandrika Jayant et al. “Supporting Blind Photography”. In: *The Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS '11. Dundee, Scotland, UK: Association for Computing Machinery, 2011, pp. 203–210. ISBN: 9781450309202. DOI: 10.1145/2049536.2049573. URL: <https://doi.org/10.1145/2049536.2049573>.
- [36] T. Kluthe et al. “Facilitating independence through automated accessible chart creation”. In: *Assistive Technology Industry Association Assistive Technology Outcomes and Benefits* (in press).
- [37] Timothy Kluthe et al. “Toward Accessible Parsons Problems on Mobile Platforms”. In: *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1*. SIGCSE TS 2026. USA: Association for Computing Machinery, 2026, pp. 589–595. ISBN: 9798400722561. DOI: 10.1145/3770762.3772646. URL: <https://doi.org/10.1145/3770762.3772646>.

- [38] Julian Kreimeier, Pascal Karg, and Timo Götzelmann. “Tabletop Virtual Haptics: Feasibility Study for the Exploration of 2.5D Virtual Objects by Blind and Visually Impaired with Consumer Data Gloves”. In: *Proceedings of the 13th ACM International Conference on PErvasive Technologies Related to Assistive Environments*. PETRA ’20. Corfu, Greece: Association for Computing Machinery, 2020. ISBN: 9781450377737. DOI: 10.1145/3389189.3389194. URL: <https://doi.org/10.1145/3389189.3389194>.
- [39] The Quorum Programming Language. *Download*. Accessed on April 21st, 2026 from <https://quorumlanguage.com/download.html>. 2026.
- [40] The Quorum Programming Language. *Experience Programming in Quorum*. Accessed on April 21st, 2026 from <https://quorumlanguage.com/epiq.html>. 2026.
- [41] The Quorum Programming Language. *Quorum*. Accessed on April 21st, 2026 from <https://quorumlanguage.com/>. 2026.
- [42] Minju Lee et al. “Benefits, barriers, and accessibility in video games: a focus group study of college students with disabilities”. In: *Disability and Rehabilitation: Assistive Technology* 21.2 (2026). PMID: 41045296, pp. 759–774. DOI: 10.1080/17483107.2025.2568171. eprint: <https://doi.org/10.1080/17483107.2025.2568171>. URL: <https://doi.org/10.1080/17483107.2025.2568171>.
- [43] Sebastian Lieb et al. “Haptic and Auditive Mesh Inspection for Blind 3D Modelers”. In: *The 22nd International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS ’20. Virtual Event, Greece: Association for Computing Machinery, 2020. ISBN: 9781450371032. DOI: 10.1145/3373625.3417007. URL: <https://doi.org/10.1145/3373625.3417007>.
- [44] James Manyika et al. *Big data: The next frontier for innovation, competition, and productivity*. https://www.mckinsey.com/~/media/McKinsey/BusinessFunctions/McKinseyDigital/OurInsights/BigdataThenextfrontierforinnovation/MGI_big_data_full_report.ashx. 2011.
- [45] Keenan May et al. “Auditory Displays to Facilitate Object Targeting in 3D Space”. In: *International Conference on Auditory Display*. June 2019, pp. 155–162. DOI: 10.21785/icad2019.008.
- [46] MDN. *ARIA*. Accessed on March 31st, 2026 from <https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA>. 2025.
- [47] MDN. *Using shadow DOM*. Accessed on April 22nd, 2026 from https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM. 2026.
- [48] Microsoft. *Complete guide to Narrator*. Accessed on January 24th, 2022 from <https://support.microsoft.com/en-us/windows/complete-guide-to-narrator-e4397a0d-ef4f-b386-d8ae-c172f109bdb1>. 2020.
- [49] Microsoft. *UI Automation - Win32 apps | Microsoft Docs*. Accessed on January 22nd, 2022 from <https://docs.microsoft.com/en-us/windows/win32/winauto/entry-uiauto-win32>. 2020.
- [50] Microsoft. *UI Automation Overview - Win32 apps | Microsoft Docs*. Accessed on January 22nd, 2022 from <https://docs.microsoft.com/en-us/windows/win32/winauto/uiauto-uiautomationoverview>. 2021.

- [51] Scratch MIT. *Scratch*. Accessed on April 1st, 2026 from <https://scratch.mit.edu/>. 2026.
- [52] Ye Mo et al. “TableNarrator: Making Image Tables Accessible to Blind and Low Vision People”. In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. CHI ’25. Association for Computing Machinery, 2025. ISBN: 9798400713941. DOI: 10.1145/3706598.3714329. URL: <https://doi.org/10.1145/3706598.3714329>.
- [53] MobiVM. *MobiVM*. Accessed on April 22nd, 2026 from <https://mobivm.github.io/>. 2022.
- [54] Michael A. Nees and Eliana Liebman. “Auditory Icons, Earcons, Spearcons, and Speech: A Systematic Review and Meta-Analysis of Brief Audio Alerts in Human-Machine Interfaces”. In: *Auditory Perception & Cognition* 6.3-4 (2023), pp. 300–329. DOI: 10.1080/25742442.2023.2219201. eprint: <https://doi.org/10.1080/25742442.2023.2219201>. URL: <https://doi.org/10.1080/25742442.2023.2219201>.
- [55] John Neuhoff. “Is Sonification Doomed to Fail?” In: *International Conference on Auditory Display*. June 2019, pp. 327–330. DOI: 10.21785/icad2019.069.
- [56] Joseph Newbold, Andy Hunt, and Jude Brereton. “Chemical Spectral Analysis through Sonification”. In: *International Conference on Auditory Display*. June 2015.
- [57] Nintendo. *Using the Accessibility Features*. Accessed on March 31st, 2026 from <https://support.nintendo.com/ph/switch2/mastery/accessibility/index.html>. 2026.
- [58] Sile O’Modhrain et al. “Designing Media for Visually-Impaired Users of Refreshable Touch Displays: Possibilities and Pitfalls”. In: *IEEE Transactions on Haptics* 8.3 (2015), pp. 248–257. DOI: 10.1109/TOH.2015.2466231.
- [59] Oracle. *Java Native Interface Specification: Contents*. Accessed on April 22nd, 2026 from <https://docs.oracle.com/en/java/javase/25/docs/specs/jni/index.html>. 2025.
- [60] World Health Organization. *Global report on health equity for persons with disabilities*. 2022.
- [61] Dominik Osinski et al. “Designing Auditory Color Space for Color Sonification Systems”. In: *International Conference on Auditory Display*. June 2019. DOI: 10.21785/icad2019.076.
- [62] Playstation. *God of War Ragnarök Accessibility | Playstation (US)*. Accessed on February 19th, 2026 from <https://www.playstation.com/en-us/games/god-of-war-ragnarok/accessibility/>. 2026.
- [63] Playstation. *The Last of Us Part II - Accessibility (US)*. Accessed on February 19th, 2026 from <https://www.playstation.com/en-us/games/the-last-of-us-part-ii/accessibility/>. 2026.
- [64] PBC Posit Software. *RStudio Desktop*. Accessed on April 1st, 2026 from <https://posit.co/download/rstudio-desktop/>. 2026.
- [65] PBC Posit Software. *RStudio Desktop*. Accessed on April 1st, 2026 from <https://posit.co/download/rstudio-desktop/>. 2026.

- [66] K R Prajwal, C V Jawahar, and Ponnuram Kumaraguru. “Towards Increased Accessibility of Meme Images with the Help of Rich Face Emotion Captions”. In: *Proceedings of the 27th ACM International Conference on Multimedia*. MM ’19. Nice, France: Association for Computing Machinery, 2019, pp. 202–210. ISBN: 9781450368896. DOI: 10.1145/3343031.3350939. URL: <https://doi.org/10.1145/3343031.3350939>.
- [67] Hrishikesh V. Rao and Sile O’Modhrain. “2Across: A Comparison of Audio-Tactile and Screen-Reader based Representations of a Crossword Puzzle”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–12. ISBN: 9781450367080. DOI: 10.1145/3313831.3376207. URL: <https://doi.org/10.1145/3313831.3376207>.
- [68] Kyle Rector et al. “Design and Real-World Evaluation of Eyes-Free Yoga: An Exergame for Blind and Low-Vision Exercise”. In: *ACM Trans. Access. Comput.* 9.4 (Apr. 2017). ISSN: 1936-7228. DOI: 10.1145/3022729. URL: <https://doi.org/10.1145/3022729>.
- [69] Samuel Reinders. “Accessible Interactive 3D Models for Blind and Low-Vision People”. In: *SIGACCESS Access. Comput.* 129 (Mar. 2021). ISSN: 1558-2337. DOI: 10.1145/3458055.3458061. URL: <https://doi.org/10.1145/3458055.3458061>.
- [70] Timothy Roden and Ian Parberry. “Designing a Narrative-Based Audio Only 3D Game Engine”. In: *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*. ACE ’05. Valencia, Spain: Association for Computing Machinery, 2005, pp. 274–277. ISBN: 1595931104. DOI: 10.1145/1178477.1178525. URL: <https://doi.org/10.1145/1178477.1178525>.
- [71] Paul Schroeder. “A Brief History of Microsoft and Accessibility”. In: *AccessWorld®: Technology and People Who Are Blind or Visually Impaired* (July 2000).
- [72] Rebecca M. Sheffield et al. “How Many Braille Readers? Policy, Politics, and Perception”. In: *Journal of Visual Impairment & Blindness* 116.1 (2022), pp. 14–25. DOI: 10.1177/0145482X211071125. eprint: <https://doi.org/10.1177/0145482X211071125>. URL: <https://doi.org/10.1177/0145482X211071125>.
- [73] Viktor Stadler and Helmut Hlavacs. “Blind Adventure - A Game Engine for Blind Game Designers”. In: *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play*. CHI PLAY ’18. Melbourne, VIC, Australia: Association for Computing Machinery, 2018, pp. 503–509. ISBN: 9781450356244. DOI: 10.1145/3242671.3242703. URL: <https://doi.org/10.1145/3242671.3242703>.
- [74] Andreas Stefik et al. “Accessible to Whom? Bringing Accessibility to Blocks”. In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 1286–1292. ISBN: 9798400704239. DOI: 10.1145/3626252.3630770. URL: <https://doi.org/10.1145/3626252.3630770>.
- [75] Unity Technologies. *Unity*. Accessed on April 1st, 2026 from <https://www.unity.com>. 2025.

- [76] Ratan Bahadur Thapa, Mexhid Ferati, and G. Anthony Giannoumis. “Using Non-Speech Sounds to Increase Web Image Accessibility for Screen-Reader Users”. In: *Proceedings of the 35th ACM International Conference on the Design of Communication*. SIGDOC ’17. Halifax, Nova Scotia, Canada: Association for Computing Machinery, 2017. ISBN: 9781450351607. DOI: 10.1145/3121113.3121231. URL: <https://doi.org/10.1145/3121113.3121231>.
- [77] Thurston et al. *Postsecondary STEM Education for Students with Disabilities: Lessons Learned from a Decade of NSF Funding*. Nov. 2016. URL: <https://eric.ed.gov/?id=EJ1144615>.
- [78] Brianna J. Tomlinson et al. “Exploring Auditory Graphing Software in the Classroom: The Effect of Auditory Graphs on the Classroom Environment”. In: *ACM Trans. Access. Comput.* 9.1 (Nov. 2016). ISSN: 1936-7228. DOI: 10.1145/2994606. URL: <https://doi.org/10.1145/2994606>.
- [79] B. N. Walker and L. M. Mauney. “Universal Design of Auditory Graphs: A Comparison of Sonification Mappings for Visually Impaired and Sighted Listeners”. In: *ACM Trans. Access. Comput.* 2.3 (Mar. 2010). ISSN: 1936-7228. DOI: 10.1145/1714458.1714459. URL: <https://doi.org/10.1145/1714458.1714459>.
- [80] Hélène Walle et al. “A Survey on Recent Advances in AI and Vision-Based Methods for Helping and Guiding Visually Impaired People”. In: *Applied Sciences* 12.5 (2022). ISSN: 2076-3417. DOI: 10.3390/app12052308. URL: <https://www.mdpi.com/2076-3417/12/5/2308>.
- [81] Keke Wu et al. “Understanding Data Accessibility for People with Intellectual and Developmental Disabilities”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445743. URL: <https://doi.org/10.1145/3411764.3445743>.
- [82] Shaomei Wu et al. “Automatic Alt-Text: Computer-Generated Image Descriptions for Blind Users on a Social Network Service”. In: *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. CSCW ’17. Portland, Oregon, USA: Association for Computing Machinery, 2017, pp. 1180–1192. ISBN: 9781450343350. DOI: 10.1145/2998181.2998364. URL: <https://doi.org/10.1145/2998181.2998364>.
- [83] Yuhang Zhao et al. “The Effect of Computer-Generated Descriptions on Photo-Sharing Experiences of People with Visual Impairments”. In: *Proc. ACM Hum.-Comput. Interact.* 1.CSCW (Dec. 2017). DOI: 10.1145/3134756. URL: <https://doi.org/10.1145/3134756>.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

William Allee
william.allee@unlv.edu

Degrees:

Bachelor of Science – Computer Science 2014
University of Nevada, Las Vegas

Thesis Title: On the Accessible Graphics Problem

Thesis Examination Committee:

Chairperson, Dr. Andreas Stefik, Ph.D.
Committee Member, Dr. Wolfgang Bein, Ph.D.
Committee Member, Dr. Bryar Shareef, Ph.D.
Graduate College Representative, Dr. Sarah Harris, Ph.D.