4-2021

Making the Easy Accessibility Package

Aaron G. Trudeau
Grand Valley State University

Making the Easy Accessibility Package

Aaron G. Trudeau

Grand Valley State University

Contents

Definitions

- *Assets* are code files, images, models, or any other files used to develop a game.
- *Classes* are programming constructs that bundle data and computer code together into a reusable format to simplify the implementation of abstract features in code.
- *Code libraries* (or just *libraries*) are collections of code that are usable within other programs or applications.
- *Control mappings* (or *mappings*) are programmed associations between inputs on physical devices (such as keys on keyboards or buttons on gamepads) and corresponding in-game actions (such as running or jumping). These may also change how the game interprets and applies physical inputs.
- *DLLs* (dynamically linked libraries) are special files that provide libraries that are usable by multiple applications at once.
- The *Input System* is a free package made by Unity designed to make retrieving player input more versatile than Unity's default input manager.
- *Game engines* are collections of programs and code libraries used to program video games.
- *Gamepads* are standard controllers (usually containing face buttons, shoulder buttons, and two joysticks) used to play a video game.
- *Makefiles* are special files containing compilation instructions that allow several code files to be compiled at once.
- *Packages* are bundles of assets game developers can import into their games to add functionality the developers would otherwise need to make from scratch.

- *Processors* are a special type of class in the *Input System* that receive an input from the player (such as a gamepad stick movement), modify it, and then apply it to the game.
- *Remappable controls* are video game settings that allow a player to change (or "remap") active control mappings from their default state.
- *Screen readers* are assistive software that give users of a computer auditory output in place of (or supplementary to) standard visual output.
- *Tab indexing* is a system which allows users to navigate between items on a user interface by pressing the Tab key on a keyboard.
- *Tolk* is an open-source screen reader code library that is used to output text to a variety of screen readers (based on which screen reader the user has active).
- *Unity* is a free game engine commonly used by beginner game developers.

Making the Easy Accessibility Package

The Easy Accessibility Package is a free, open-source package for Unity that aims to make accessible video game development as simple as possible. Common obstacles preventing game developers from implementing accessibility features are a lack of time to dedicate to the features and a lack of knowledge about accessible game design. This package helps overcome those obstacles by including both assets that simplify accessibility features and tutorials on how to use those assets. Through academic research and interviews with workers in the field of accessibility, I identified two accessibility features game developers often have trouble implementing (remappable controls and screen reader support) and used that knowledge to craft a Unity package that simplifies the work needed to include them in a game.

Research

The research component of this project was vital to understanding the issues and best practices surrounding accessibility. I came into this project without much previous experience in accessibility, so it was important to gain at least a basic understanding of the field before writing any code. My research into accessibility primarily came from three types of sources: scholarly literature, unscholarly accessibility websites or blogs, and interviews with workers in the field of accessibility.

Once I made it past the research phase of my project and began writing code, I referenced the Unity documentation[1] (as well as the documentation for specific Unity packages

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

I used)[2] often. I also checked online coding forums (such as Stack Overflow)[3] if I encountered any issues while developing my project, which helped direct me towards official documentation to find solutions.

Scholarly Literature

An Empirical Study of Issues and Barriers to Mainstream Video Game Accessibility

This paper was a great way to view firsthand accounts on accessibility in video games from both game developers and gamers with disabilities. In the paper, Porter and Kientz interviewed game developers to gather data on how the industry treats accessibility (2013). Then, Porter and Kientz surveyed a set of gamers with disabilities to synthesize statistics on the needs of the greater community, collecting data on items such as what percentage of the set had a certain disability, and what frustrations surveyed gamers experienced while playing video games (2013). The conclusions Porter and Kientz reached from interviewing game developers were helpful to my general understanding of accessibility in game development; the conclusions they reached when surveying gamers with disabilities significantly motivated the direction of my project.

The main lesson I took from the Porter and Kientz paper was that games not working with the accessibility technologies players already have installed is a major roadblock for gamers with disabilities (2013, p. 4). The only other screen reader package for Unity I'd found used its own text-to-speech system and wasn't directly compatible with major screen readers such as

---

[2] See https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/api/index.html.
[3] See https://stackoverflow.com/.

JAWS.[4] This meant that there was no way for most players using screen readers to get them working within Unity games, solidifying that a Unity package containing support for major screen readers was a necessity.

Improving Web Accessibility: A Study of Webmaster Perceptions

In this paper, Lazar et al. surveyed 175 webmasters to study how web developers view accessibility (2004). In the paper, they find several major roadblocks web developers face when trying to make a website accessible, including a lack of time and a lack of software tools (Lazar et al., 2004). While these findings were for the field of web development, they generalize to game development as well, as both fields deal with user interface design and user experience, which are both heavily affected by accessibility. The findings affirmed that a Unity package that simplified accessible design and saved developers' time would be helpful in eliminating barriers to accessibility in Unity game development.

Non-Scholarly Web Sources

Game Accessibility Guidelines

This website contains a basic, descriptive list of accessibility features, categorized by difficulty of implementation.[5] The extent of my use of this website was browsing in my free time to brainstorm project ideas. The website helped me directly when one of the sections I read led me to discover the page for the Unity screen reader package I mentioned previously.[6]

---

[4] The listing for this package is found at https://assetstore.unity.com/packages/tools/gui/ui-accessibility-plugin-uap-87935.

[5] See http://gameaccessibilityguidelines.com/.

[6] See http://gameaccessibilityguidelines.com/ensure-screenreader-support-for-mobile-devices/.

UX Collective – Designing Main Menu Screens for Visually Impaired Gamers

This article contains a basic overview of best practices to follow when designing user interfaces compatible with screen readers (Gantzer, 2020). Specifically, Gantzer covers the challenges players using screen readers face when playing games (2020). These challenges fall into four categories:

- knowing what menu screen the player is on;
- knowing what menu option the player has selected, and how to interact with it;
- knowing all the actions a player has on a given screen; and
- knowing if player input was registered in any way (Gantzer, 2020).

All these challenges are addressed within the screen reader feature in my project.

Interviews

Jeff Sykes – Assistive Technology Coordinator

For my first interview, I wanted to speak to someone at the Grand Valley State University (GVSU) Disability Support Resources (DSR) department to help me better understand accessible technology. I emailed the department, and they suggested I interview Jeff Sykes, an Assistive Technology Coordinator with the department (Disability Support Resources, personal communication, November 13, 2020).

To start our meeting, Jeff told me about what he did in his role with DSR. One responsibility he has is to set up hardware and software around the GVSU campus to assist those with visual and audio impairments (J. Sykes, personal communication, November 19, 2020). One of the technologies he mentioned working with were screen readers, which are provided on campus to aid people with visual impairments or people who are illiterate (J. Sykes,

personal communication, November 19, 2020). He specifically mentioned a free screen reader called NVDA, which ended up being crucial to the testing of my screen reader functionality in the project (J. Sykes, personal communication, November 19, 2020). Later in the interview, he recommended websites I could research to learn more about accessibility[7] and suggested features I could include in my project (J. Sykes, personal communication, November 19, 2020). Two of his suggestions (screen reader support and providing speech feedback for actions) ended up being implemented in the project (J. Sykes, personal communication, November 19, 2020).

Overall, my meeting with Jeff was incredibly helpful. He taught me a great deal about accessible technology and played a large part in my decision to include screen reader functionality within my project.

Clinton "halfcoordinated" Lexa – Freelance Gaming Accessibility Consultant

I was initially interested in meeting with Clinton because I was familiar with their accessibility work from their online presence as a video game live streamer. In our meeting, we discussed several aspects of how I could make my project as useful as possible, while covering specifics of how I should implement certain aspects of the two main features of the project.

For the remappable controls feature, Clinton informed me about several possible options I hadn't considered before, including adjustable mouse sensitivity, reorientable joystick controls, and the ability to map button inputs to joystick inputs (C. Lexa, personal

[7] Suggested websites included https://webaim.org/, https://docs.microsoft.com/en-us/windows/win32/winauto/microsoft-active-accessibility, and https://www.w3.org/WAI/.

communication, February 11, 2021). Clinton told me these options could be difficult to implement; this solidified that the options would be good to include in the package, since the point of the package is to make difficult tasks easier for game developers (C. Lexa, personal communication, February 11, 2021).

For the screen reader feature, Clinton gave me many resources which ended up drastically simplifying the feature's implementation. The most prominent resource they recommended was the *Tolk* library (C. Lexa, personal communication, February 11, 2021). Using *Tolk* significantly cut down on the amount of work needed to support screen readers and allowed me to dedicate more time to adding customizable options to the feature. These options were based on another resource Clinton directed me towards (being Gantzer's screen reader user interface tutorial I mentioned previously) (C. Lexa, personal communication, February 11, 2021; Gantzer, 2020).

Overall, my meeting with Clinton shaped the course of my project significantly, and it was the most valuable piece of research I conducted for this project.

Features

There are two main features included in the Easy Accessibility Package: remappable controls and screen reader support. To make the package more user-friendly for developers, I created tutorials on implementing both features and included them within the package. The development process for both features has been so enjoyable that I plan to continue working on it after I graduate. I already have ideas in mind for improving the main features after graduation, which I expand on below.

Remappable Controls

I began developing this feature by researching online for existing examples of remappable controls in Unity. I found a few good examples,[8] but they all involved workarounds to get around roadblocks stemming from Unity's default input manager (as there is no native way to remap controls during runtime using the default input manager). Using a workaround would've been unintuitive, so I kept searching for another way to implement remappable controls, leading to my discovery of Unity's *Input System*.

The *Input System* was the clear method of choice for implementing remappable controls both due to its general versatility, and because the package comes with a working example of remappable controls. Since the package's license clarifies that I can modify and distribute code from the package, I decided to use the included example as a base for the feature (Unity Technologies, 2019). This allowed me to dedicate my efforts towards features more difficult to implement than basic button remapping.

Adjusting Mouse Cursor Sensitivity

Mouse sensitivity is easily implemented through the *Input System* for controls such as player camera movement but adjusting sensitivity for an on-screen mouse cursor is a much more complex issue. If one tries to adjust the sensitivity of the system mouse, either the mouse location must be warped each frame (which looks quite jittery due to the operating system moving the location between frames), or the mouse speed must be changed at a much lower level using a system call. The latter option isn't ideal, as it would change the mouse sensitivity

[8] The most robust example can be found here: https://www.youtube.com/watch?v=iSxifRKQKAA&t=610s.

system-wide on Windows. I decided the best course would be to implement a software mouse cursor that takes input from the system mouse but displays in a different location than the system mouse cursor.

The *Input System* came packaged with a sample that demonstrated the use of a software mouse cursor (referred to as a "virtual mouse"). While the sample was only set up to be controlled by a gamepad, its features were so robust that I decided to modify it to be controlled by the system mouse as well. This task was easily achieved, but I encountered a bug after getting the positional controls working – each time I clicked the mouse, the virtual mouse would never exit the clicked state. The problem stumped me for a while, since it would only occur when controlling the virtual mouse with a system mouse, and not when controlling it with a gamepad. The root cause was my misunderstanding of the *Input System*; when I told the virtual mouse to accept input from the system mouse, the *Input System* gave it input from *any* mouse, including itself. This led to the virtual mouse causing itself to click every frame. After fixing this issue, the mouse worked as intended. I finished the feature when I added sensitivity to the virtual mouse's cursor movement by multiplying its change in position by a "sensitivity" percentage each frame.

Unfortunately, this feature goes unimplemented in the final version of the project due to a limitation of the example it was based on. The virtual mouse doesn't work on screen-scalable user interfaces, which makes it practically unusable in a real game. Still, working on this feature gave me a great deal of familiarity with the *Input System*, so I'm ultimately happy with the time I spent on it.

Saving and Loading Remapped Controls Between Play Sessions

I decided to include this feature to expand on the *Input System*'s existing remappable controls example and make it more suitable for a real game. To figure out what data I needed to store to make this feature happen, I needed to know more about how the *Input System* worked behind the scenes. This forced me to dig deeper into the *Input System* documentation, further solidifying my knowledge of the system.

My main difficulties while working on this feature stemmed from a misunderstanding of how the *Input System* treats remapping at runtime. I hit a wall while working on this feature because when querying the status of one of the controls that had been remapped at runtime, I was receiving the old value of the control instead of the new value. The *Input System* stores the *default* control options for a game separately from the *modified* control options created when a player remaps controls at runtime, but at first, I was only asking for the *default* options. After this realization, it was simple to retrieve the correct value, store it after a remapping occurred, and retrieve it when the game next loaded, allowing me to finish implementing the feature.

Remapping Stick Directions

This feature is made up of two key components: allowing the player to invert the X or Y axes of a gamepad joystick input (making pressing "left" go to the right, etc.), and allowing the player to change the orientation of the cardinal directions of a joystick. The latter feature involved changing where "up" on the joystick was (for example, the feature could make a "left" input register as "up") and changing the rest of the cardinal directions on the joystick accordingly. Both features were implemented using processors.

Processors make implementing axis inversion simple – all one needs to do is accept the two-dimensional vector input from a gamepad joystick and multiply the X or Y component of

the vector by -1 to invert the vector along an axis. By default, the *Input System* comes with a processor that does this, but to change the joystick orientation, I had to write my own processor to apply a vector rotation to a joystick input. I knew how to do this from the Linear Algebra course I took in Fall 2020, so I wrote the processor without issue.

From here, my only tasks left were implementing a basic user interface, figuring out how to apply the processors at runtime, and saving them between play sessions. The user interface was implemented easily, but the other two features were tougher to figure out. The main source of difficulty was that active processors on a specific input were stored as comma-delimited strings, and there were no helper methods to facilitate working with these strings. I was able to get around this by writing my own helper methods, which helped me apply processors at runtime. After this, I used the same save system as in previous tasks to save and load active processors between play sessions.

Mapping Stick Inputs to Button Actions

This feature was easy to implement, as it's implicitly included in the basic remappable controls example from the *Input System*. The *Input System* provides synthesized binary button controls that the joysticks on a controller trigger whenever they register a value in any of the four cardinal directions. However, there's a drastic limitation to this implementation: the inputs are triggered by the slightest positive or negative movement on both the X and Y axes. It's practically impossible to move a joystick on one axis without moving it on the other, so adjacent cardinal directions are guaranteed to register at the same time, which limits the feature's usefulness. Adding a joystick processor to make it so these inputs only trigger on significant joystick movements would make the functionality better, but that's not a possibility with this

type of input (since the “inputs” generated by the *Input System* are considered buttons, not joysticks). Overall, there’s a lot that could be done to make this feature better, but due to the short timeframe of this project, the basic implementation provided by the *Input System* was all that was possible.

Tutorial

The tutorial I built to teach developers to use my package’s remappable control features used a sample game included with the package. The game showed the player’s inputs on screen whenever the player used a gamepad or keyboard to trigger them. From this game screen, the player could navigate to a settings screen to remap their controls. The settings screen is empty at the beginning of the tutorial, and through reading the tutorial, the reader uses the pre-built assets I included in the project (some of which I repurposed from the *Input System* example) to add options allowing the player to remap the game’s controls. This teaches the reader how to implement all remappable control features included in the package except for mouse sensitivity.

Future Work

My main task immediately after graduation will be tweaking my implementation of saving and loading remapped controls. The class structure of my current implementation is built well, but it currently utilizes a pre-built Unity data saving and loading system that uses the Windows registry. In general, it’s not recommended to use the Windows registry for most purposes, so I will try to save the data in a different format (such as a JSON file) to avoid registry use. I designed the saving and loading system to function all from a single class, so that class will be the only one I need to change to implement this update.

Screen Reader Functionality Using *Tolk*

After learning about the *Tolk* library from Clinton, screen reader support seemed like it would be trivial to include within my project. However, there was a barrier that initially prevented me from using *Tolk*: compiled builds of the library (which were necessary to use it within Unity) were not included with the library's source code.[9] This meant I needed to compile the library myself.

Compiling Tolk

Tolk is written in the language C++, but the library comes with classes and libraries written in other languages (such as C# and Java) which allow those languages to interface with the main C++ code. This made compiling *Tolk* much harder, since I needed to have tools installed for each supported language to run *Tolk*'s Makefile. After installing all the necessary tools, I ran the Makefile, which failed halfway through compilation. By inspecting the error this failure generated, I found out this occurred because the necessary compilation tools for *Tolk*'s Java library weren't being properly linked from my computer's Java directory. I spent some time trying to resolve this, but I didn't need the Java library (as Unity uses C#), so I decided to remove its compilation instructions from the Makefile. This resolved the issue, which prompted me to test to see if the library worked by running an example C# application included with the library.

When I first ran the example application, it ended up crashing before passing any commands to the screen reader I had running. While reading the example application's error logs, I discovered the crash occurred because I'd compiled *Tolk* for 32-bit CPU architecture (x86), while the example application was being compiled for 64-bit CPU architecture (x86-64). Both

[9] See https://github.com/dkager/tolk.

architectures are used in modern computers, but x86-64 is newer than x86 (and faster, in some cases). Initially, I wasn't entirely sure why the x86/x86-64 mismatch caused the error, but I knew I could get around the error by compiling the example application for x86. This resolved the issue, allowing me to start working on integrating *Tolk* with Unity.

Integrating Tolk with Unity

There were a couple of main issues standing in my way of getting *Tolk* working in Unity. First, when *Tolk* is compiled, the main C++ library takes on the form of a C++ DLL. Additionally, a C# DLL (which is dependent on the main C++ DLL) is compiled, allowing *Tolk* to be used within C#. I'd never used DLLs before, so I researched online to learn how to integrate DLLs with Unity. I found a tutorial for this process[10], but after following the tutorial, another problem presented itself.

The new issue was that Unity games are built for x86-64 by default, but my *Tolk* DLLs were compiled for x86. This made Unity incompatible with my DLLs. This was the second time an architecture mismatch caused problems for me, so I did more research into which architecture I should've targeted when compiling the DLLs. I found that my computer, like Unity, targets x86-64 by default, which showed me that the *Tolk* DLLs being compiled in x86 was unusual. This prompted me to recompile the DLLs in x86-64. Doing so was simple, as the command line tool I used to compile *Tolk* had two separate versions: one for compiling in x86 (which I used accidentally), and one for compiling in x86-64. I compiled *Tolk* using the x86-64

[10] See https://ericeastwood.com/blog/17/unity-and-dlls-c-managed-and-c-unmanaged.

command line tool, and after importing the new DLLs into a Unity project, I made an example game that triggered speech from a screen reader when the game loaded.

Making Easily Usable Assets

From here, most of my work was spent making assets that would allow a game developer to add screen reader functionality into their games with little effort. I did this by reviewing the best practices listed in the Gantzer article and creating assets a developer could use to easily implement those best practices. The two main assets I made during this process were:

- Code that
    - initializes the *Tolk* library on load;
    - reads the purpose of the screen the player is on; and
    - adds a tab indexing system which allows players to cycle through user interface items to be read by the screen reader.
- Code that (a) makes game objects readable when the mouse cursor hovers over them; and (b) makes game objects focusable in the tab indexing system.

Tutorial

For the tutorial on this section, I included a sample game with the package. In the game, the player navigated between a main menu and a game screen. The gameplay only involved clicking a button to increment a counter. At the beginning of the tutorial, this game starts off without screen reader support, and over the course of the tutorial, the reader uses the pre-built assets I included in the project to make all the user interfaces within the game accessible by a screen reader. The goal of this tutorial is to teach the reader how to use the package to add

screen reader support to an existing game (as most developers would download my package to use on an existing project).

Future Work

When I continue working on this project after graduation, feedback from developers who are using the package will determine which aspects of the feature I work on. This feedback will come from support tickets opened through the package's website.[11] Additionally, I plan to use screen readers more often, which will help me identify potential improvements.

Conclusions

Working on the Easy Accessibility Package this semester has been the most valuable and enjoyable learning experience I've ever had. The research phase of this project taught me how to network with those in the field of accessibility, honing critical communication skills and helping me discover my passion for a field I hadn't ever engaged with before. This project nurtured my existing passions as well - game development has always interested me, and this project has allowed me to apply that interest to a real-world problem in a way no other experience has. Ultimately, my work on this project serves quite poetically as a culminating experience for my time in the Honors College, perfectly embodying all the lessons I've learned over the past four years. Service, equity, leadership, interdisciplinarity – all these themes have shaped my experience at GVSU, and I kept every one of them in mind while working on this project. As I graduate and move on to tackling more real-world problems, I'm confident the

[11] The section of the package's website that developers can file support tickets at is https://github.com/trudeaua21/EasyAccessibilityPackage/issues.

practice I've gained in applying these themes to my work will positively shape the work I do for the rest of my life.

References

Gantzer, T. (2020, January 12). *Designing main menu screens for visually impaired gamers.* UX Collective. https://uxdesign.cc/designing-main-menu-screens-for-visually-impaired-gamers-865a8bd76543

Lazar, J., Dudley-Sponaugle, A. & Greenidge, K. (2004). Improving web accessibility: a study of webmaster perceptions. *Computers in Human Behavior, 20(2),* 269-288. https://doi.org/10.1016/j.chb.2003.10.018

Porter, J. R. & Kientz, J. A. (2013, October). *An empirical study of issues and barriers to mainstream video game accessibility.* Paper presented at ASSETS '13: Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility, Bellevue, Washington. https://doi-org.ezproxy.gvsu.edu/10.1145/2513383.2513444

Unity Technologies (2019). *License.* *https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/license/LICENSE.html*