Web Browser Rendering and Interaction in Custom OpenGL Applications

Trevor Tracy
Computer Engineering
University of St. Thomas
St. Paul, MN 55105
trac7268@stthomas.edu

Thomas Marrinan
Computer and Information Sciences
University of St. Thomas
St. Paul, MN 55105
tmarrinan@stthomas.edu

Abstract

Web browsers have become a ubiquitous tool for accessing, manipulating, and interacting with data. This project aimed to create an embeddable web renderer that can easily be integrated into custom 2D and 3D applications using OpenGL. The success of this work would allow application developers to incorporate web content for a portion of their application while using custom techniques for other graphical elements without much effort. While embedded web content is already possible with existing solutions, there are many complexities that make this task non-trivial, especially for cross-platform compatibility. In this paper, we present our solution that abstracts away many of the complexities as well as providing three examples of how custom applications could use our embedded web renderer.

1 Introduction

Web technologies have been increasing in power and use. Modern browsers have the capability to leverage high-bandwidth networks [1], perform parallel computation [2], and render 3D scenes with hardware-accelerated graphics [3]. Due to their wide adoption and power, we aimed to create a simple web browser that can easily be embedded as a part of another application. This would allow application developers to render web content for a portion of their application, while using custom techniques for other graphical elements.

The Chromium Embedded Framework (CEF) was designed to accomplish such a goal [4]. However, there are numerous complexities that make it non-trivial to integrate into another application. For example, dropdown menus are rendered completely separately from the main page, and JavaScript dialogs and right-click menus are not rendered at all. Therefore, we abstracted on top of CEF to create a single unified web renderer that could easily be embedded into OpenGL applications. Our web renderer produces a single unified bitmap that incorporates dropdown menus, JavaScript dialogs, and right click menus. This solution makes it easy to use web content in a custom application by simply binding the resulting bitmap to an OpenGL texture.

The remainder of this paper will cover existing work surrounding integrating web technologies into custom applications, our methodology for overcoming the challenges that are present in current solutions, and sample use cases for integrating our web renderer. Finally, we will wrap up and share future directions for the use of our web renderer.

2 Related Work

A number of modern desktop applications leverage web technologies for some or all of its user interface. Visual Studio Code, Slack, and Skype all use Electron, a framework based on CEF for creating native applications with JavaScript, HTML, and CSS [5]. While these applications successfully integrate web rendering into custom applications, Electron is an all-JavaScript environment that uses Node.js [6], which makes it difficult for use in custom OpenGL applications.

A few existing applications have leveraged CEF for more performant rendering and data processing that incorporate web technologies. Eskilson [7], leveraged CEF for user interface design in the astrovisualization software OpenSpace [8]. Their goal was to enable easy cross-platform development for user interfaces. Similarly, S. Kumar et al. utilize CEF for a cross-platform user interface for the Molecular Evolutionary Genetics Analysis (MEGA) software [9]. MEGA was originally designed as an MS-DOS application and continued to be developed as a Windows application. The authors present their work to make MEGA cross-platform, terming their version MEGA X. In order to avoid the Operating System complexities for user interface design, the paper highlights the use of CEF for enabling their user interface to be designed using HTML, CSS, and

JavaScript. Both Eskilson and S. Kumar et al. use web technologies for custom user interfaces. Therefore, they only needed to address the keyboard input translation from the window management tool to CEF. JavaScript dialogs, dropdown menus, and right-click menus were not necessary for this work.

Additionally, C. Kumar et al. researched methods for integrating different input modalities into a web browser [10]. They demonstrate their concept through the use of eye-gaze navigation. The authors create a custom browser experience tailored to eye-gaze interaction by using CEF to extract various HTML elements and modifying their layout. They also implement click emulation and zooming on areas of interest. While this work represents an interesting and novel use of web technologies, it represents a specific use case rather than a general-purpose web renderer.

One example of a general-purpose embedded web renderer is from LiveCode, a cross-platform rapid application development runtime environment [11]. They allow developers to use web content as media within a custom application, and enable users to interact with the web page [12]. LiveCode is a commercial product that integrates web rendering as part of their wide array of libraries for building apps. This however, limits which applications can leverage their work.

3 Methods

At first glance, embedding web browser rendering and interaction into custom applications may appear straightforward. However, there are many intricate details that are fundamental to the application that are individually challenging. At the top of this list is user interaction. Without user interaction, the custom application is only a web page viewer rather than an interactive system. Also, some web content is not rendered in the normal HTML flow. Dropdown menus, JavaScript dialogs, and right-click menus are all implemented via the browser using native Operating System graphical elements. Finally, at the end of this section, we address how to incorporate multiple tabs/windows into one application and some additional features that we intend to develop as the project matures.

3.1 User Interaction

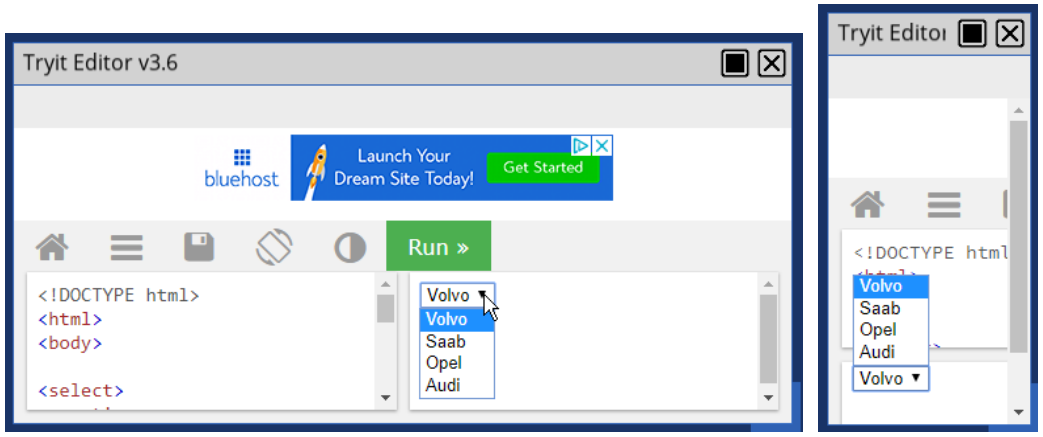
One of the challenges in creating an easily embedded web browser for OpenGL applications, was adding user interaction. When the user presses a key or moves the mouse, we can determine which button is pressed in addition to key modifiers (such as shift, alt, ctrl) thanks to OpenGL libraries. However, these key identifiers do not match CEF definitions for keys and therefore must be changed to values specific to CEF before relaying user input to the application. This process is Operating System dependent, which added an additional layer of complexity since we wanted our browser to be cross-platform.

As a result of the platform dependencies for keystrokes, two separate key definition sets had to be created and filtered upon. For example, if we detected the system as being a

Windows machine, we would ignore the Linux keystroke definitions and use the Windows set. On the other hand, mouse movement and clicks were not platform dependent which allowed for easier mouse navigation implementation.

3.2 Dropdown Menus

For the dropdown menus, we had to calculate the proper position, and instruct OpenGL to render the menu items on top of the main web content.



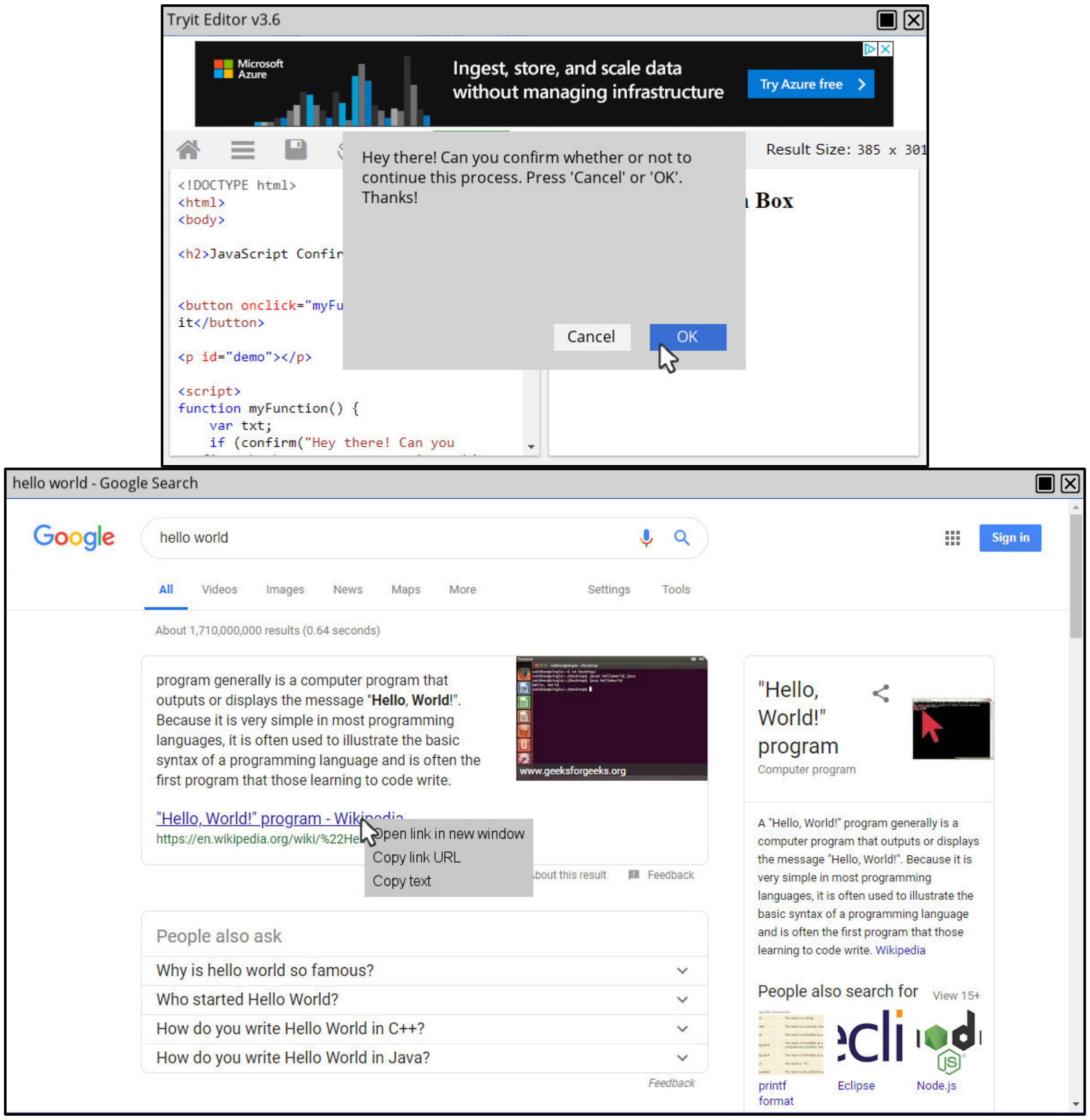
```
<!DOCTYPE html>
<html>
<body>

<select>
```

Figure 1: Dropdown menu example showing instances where the menu can drop below the selection box, and where it must pop above the selection box that is displayed as part of the main web content.

Figure 1 shows an example of the same web page where the dropdown menu must be drawn in two different ways. As you can see, depending upon the sizing of the web page and location of the selection box, the dropdown menu is positioned differently. In the left image, there is adequate room above and below so, by default, the menu is drawn below the selection box. In the right image however, there is a lack of room below for the dropdown menu, thus it is placed above the selection item.

3.3 JavaScript Dialogs and Right-click Menus

As mentioned earlier, JavaScript dialogs and right-click menus are not rendered at all using CEF. For JavaScript Dialogs, CEF simply provides a callback notifying the application what type of dialog is present (alert, confirm, or prompt) and what the text should be. For right-clicks, CEF provides a callback with an array of menu items. Therefore, we created custom OpenGL renderings to display the dialogs and menus in a manner similar to a typical web browser. Once rendered, these drawings, just like the dropdown menus, are overlaid on top of the main web content and rendered to a single unified texture that can be accessed by the end-developer's OpenGL application. Figure 2 illustrates this functionality.



Hey there! Can you confirm whether or not to continue this process. Press 'Cancel' or 'OK'. Thanks!

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Confir

<button onclick="myFu
it</button>

<p id="demo"></p>

<script>
function myFunction() {
    var txt;
    if (confirm("Hey there! Can you
```

About 1,710,000,000 results (0.64 seconds)

program generally is a computer program that outputs or displays the message "**Hello, World!**". Because it is very simple in most programming languages, it is often used to illustrate the basic syntax of a programming language and is often the first program that those learning to code write.

"Hello, World!" program - Wikipedia
https://en.wikipedia.org/wiki/%22He

"Hello, World!" program
Computer program

A "Hello, World!" program generally is a computer program that outputs or displays the message "Hello, World!". Because it is very simple in most programming languages, it is often used to illustrate the basic syntax of a programming language and is often the first program that those learning to code write. Wikipedia

People also ask
Why is hello world so famous?
Who started Hello World?
How do you write Hello World in C++?
How do you write Hello World in Java?

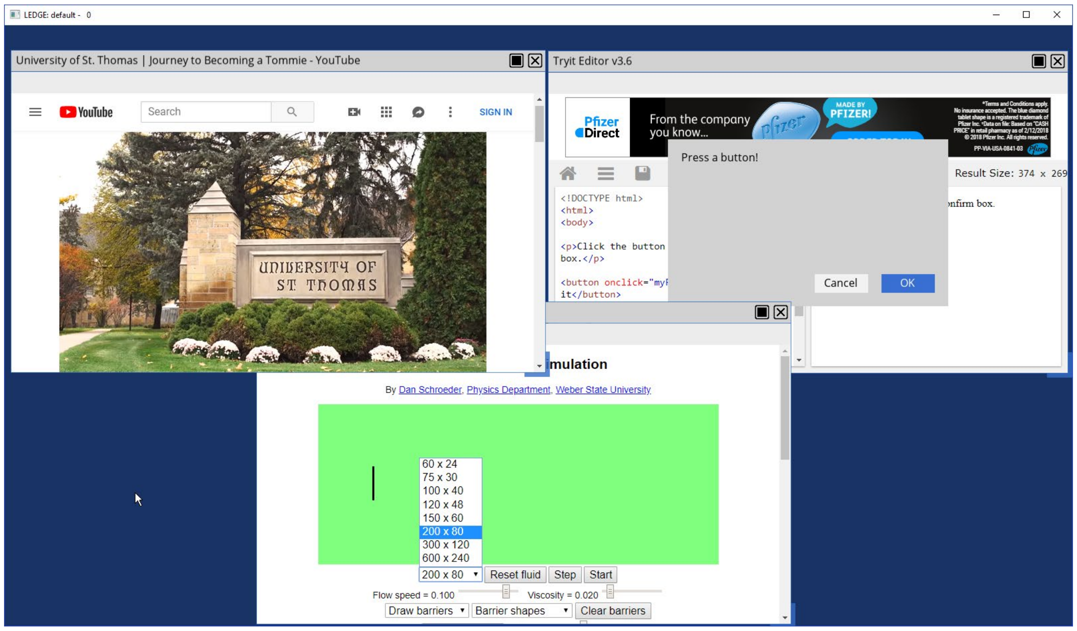
Figure 2. Custom user interfaces for non-HTML content. The top image shows the JavaScript dialog for a 'confirm' event. The bottom image shows a right-click menu for selecting a link.

3.4 Multiple Browsers in Single Application

After developing the easily embedded web browser, we created a sample application that can view multiple web content windows together. This gives the user a desktop-like environment within the application that allows for embedded browser windows to be repositioned and resized inside the OpenGL application. Embedded web content windows stack on top of each other when one browser window gets moved in front of the other. User interaction, such as mouse clicks and key presses also are directed to the appropriate browser window.

For resizing, we created a small extension of the bottom right corner of each browser window that the user can click on to change the window size. The benefit to this approach is that resizing is more universal between touch screen devices and mouse pointer interactions. Rather than developing two different solutions based on the user input method, we can have one approach that covers all inputs.

Figure 3 shows an example of three different web pages all active within our custom sample application. In the top-left, there is a web page with a YouTube video. This highlights our embedded web renderer's performance when dealing with rapid animation. On the top-right, there is a web page that opens a JavaScript 'confirm' dialog. This showcases our custom interface for dealing with web content that is outside of the normal HTML flow. Finally, in the bottom window, we have a web page on fluid simulation that contains dropdown menus.



```
<!DOCTYPE html>
<html>
<body>

<p>Click the button
box.</p>

<button onclick="myF
it</button>
```

Figure 3: Sample application rendering three separate web content pages simultaneously.

3.5 Features to Develop

We have successfully created a unified web renderer that can easily be integrated into custom OpenGL applications. However, there are some limitations to our work. For example, the right-click menu is currently only visual. Nothing will actually happen if an item is selected. This is due to the fact that the desired response for many items would be applications dependent. For example, what a given application would want to do if the user right-clicked a link and selected 'open in new tab' could vary. Therefore, we intend to build in hooks for application developers to get callbacks when such items are selected. This way application developers maintain their customizability for such actions.

We also hope to develop a distributed rendering framework so that web content can be leveraged for ultra-high-resolution data visualization on multi-monitor or even tiled display systems. This will also help with performance when rendering several tabs/windows since each node in a cluster would only be responsible for rendering a subset of all tabs/windows.

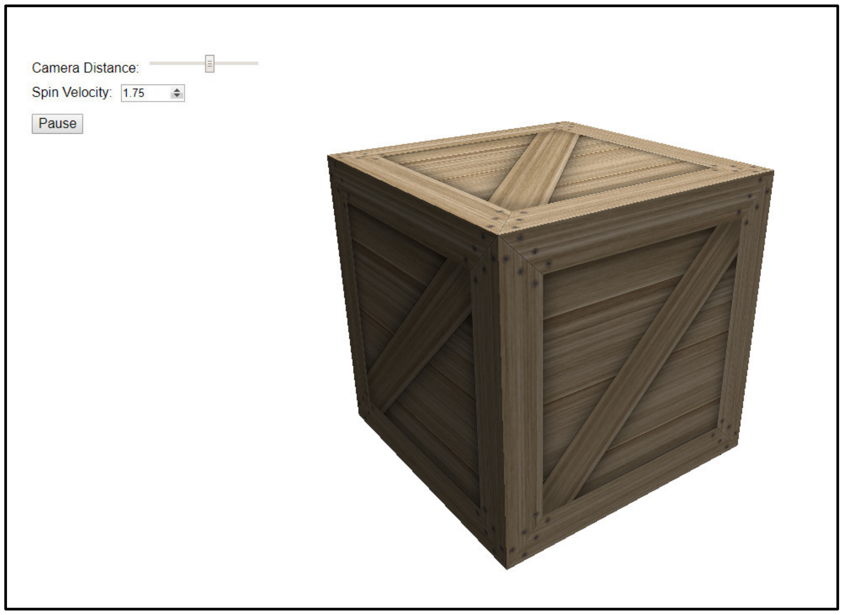
Finally, developing multiple user input functionality is of great interest. We intend to provide hooks for external devices to send messages over a network that will be transformed into key, mouse, and touch events on the embedded web page. This will greatly extend the use of standard browsers that can only allow set of input devices to interact at a time.

4 Results

One of the most interesting aspects of this project has been thinking about all the possible use cases this web renderer could apply to. In general, our web renderer could be used in the following ways: for 2D browser windows, as a 2D graphical user interface (GUI) overlay, and as a texture on a 3D object. With future development and the addition of features, such as distributed rendering, the use cases could continue to expand.

Our embedded web renderer gives developers the possibility to create 2D browser window(s) in their custom application. We have created a sample application that accomplishes this, as was highlighted in Figure 3. Creating a custom browser is especially useful for developers wishing to extend the input modalities offered by standard browsers. This need has already been highlighted in the Related Work section with for integrating eye-gaze input. Also, we have particular interest in enabling multiple users to interact with web content using their own personal devices.

Another use for embedding web content is to design cross-platform user interfaces that can overlay custom content. In this instance, a developer could write their graphical user interface in HTML, CSS, and JavaScript, then render it with a transparent background using our web renderer. The resulting texture would be transparent wherever GUI elements are not drawn, and therefore could be rendered on top of custom graphics. We show an example in Figure 4 of a simple application that renders a spinning cube, with a GUI added to control the camera distance, spin velocity, and a play/pause button for starting and stopping the animation.


Camera Distance:

Spin Velocity: 1.75

Figure 4. OpenGL application with web rendered GUI.

Finally, embedding web content is useful for applying a web page as a texture on a 3D object in a custom OpenGL application. This would enable developers to create an immersive environment for viewing web content. For example, a video game may have a character sit at a computer desk and browse the web. Figure 5 shows a web page rendered as a 3D texture that is drawn as the screen on a model of a computer monitor.

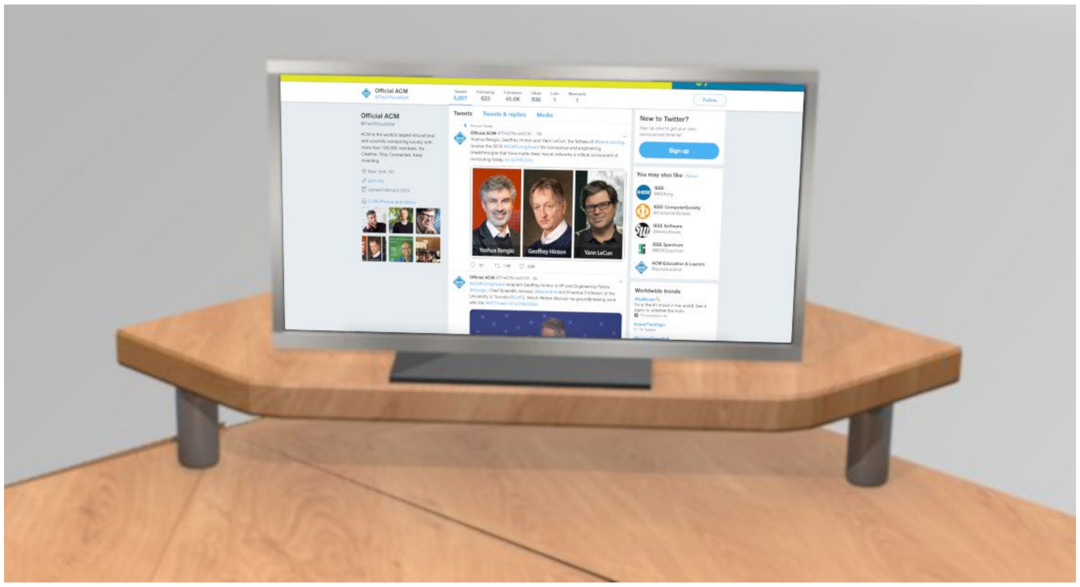


Figure 5: ACM Twitter page drawn using our web renderer embedded in 3D scene using "Desk with Computer 3D Model" [13].

5 Conclusion

Overall, web technologies are growing in both their power and use. As a result of this expanding technology, we aimed to create a simple web renderer that is universal and malleable in its structure. This way the web renderer could be easily embedded as part of custom applications in order for a variety of projects and developers to use and modify to fit their needs.

At its core, our web renderer leverages the Chromium Embedded Framework. This open-source framework gave us a foundation for our web renderer, but also contained numerous complexities. Our work creates a single unified texture for all content on a web page. We overcame challenges to correctly handle things such as user interaction, dropdown menus, and JavaScript dialogs / right-click menus. This greatly improves the ease in which developers can integrate web content into their custom OpenGL applications.

Finally, we highlighted three major use cases that benefit from the work we've done thus far. The web renderer can be used for 2D browser windows, a 2D GUI overlay, and as a texture on a 3D object. As this project continues to be developed and fine-tuned, it is possible that newer, more intriguing use cases emerge as well. Nevertheless, web applications are continuing to grow and show no signs of slowing down. This is very promising and exciting for any projects using and expanding upon the current web technologies.

References

[1] "The WebSockets API," 2019. Available: https://www.w3.org/TR/2009/WD-websockets-20091222/

[2] "Web Workers," 2019. Available: https://www.w3.org/TR/workers/

[3] "WebGL Overview," 2019. Available: https://www.khronos.org/webgl/

[4] "Chromium Embedded Framework," 2019. Available: https://bitbucket.org/chromiumembedded/cef

[5] "Electron," 2019. Available: https://electronjs.org/

[6] "Node.js," 2019. Available: https://nodejs.org/en

[7] Klas Eskilson. 2017. "Creating User Interfaces Using Web-based Technologies to Support Rapid Prototyping in a Desktop Astrovisualization Software." In *Thesis at Linköping University*.

[8] Alexander Bock, Emil Axelsson, Karl Bladin, Jonathas Costa, Gene Payne, Matthew Territo, Joakim Kilby, Masha Kuznetsova, Carter Emmart, Anders Ynnerman. 2017. "OpenSpace: An Open-source Astrovisualization Framework." In *The Journal of Open Source Software*.

[9] Sudhir Kumar, Glen Stecher, Michael Li, Christina Knyaz, and Koichiro Tamura. 2018. "MEGA X: Molecular Evolutionary Genetics Analysis across Computing

Platforms." In *Molecular Biology and Evolution*, volume 35, issue 6, June 2018, pp 1547–1549.

[10] Chandan Kumar, Raphael Menges, Daniel Müller, and Steffen Staab. 2017. "Chromium based Framework to Include Gaze Interaction in Web Browser." In *Proceedings of the 26th International Conference on World Wide Web Companion* (WWW '17 Companion).

[11] "LiveCode," 2019. Available: https://livecode.com/

[12] "How to Add a Web Browser to Your App," 2019. Available: https://livecode.com/how-to-add-a-web-browser-to-your-app/

[13] "Desk with Computer 3D Model," 2019. https://www.turbosquid.com/3d-models/desk-computer-table-monitor-3d-model-1269951