EGAD: The Electron Grid

of Aligned Documents

A Major Qualifying Project

Submitted to the Faculty of

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree in Bachelor of Science

in

Interactive Media and Game Development

by

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Bailey Sostek

Date: 25 April 2019

Project Advisor:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Brian Moriarty, IMGD Professor of Practice

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see <http://www.wpi.edu/Academics/Projects>.

# **Abstract**

This report documents the design and build process for the Electron Grid of Aligned Documents (EGAD). This framework is an open source project which allows any developer to quickly and efficiently make grid-based applications linking webpages to one another. Modules called widgets can be inserted into any grid cell to implement additional functionality that a developer may want to incorporate into their application. Several commonly-needed widgets are provided, as well as a thoroughly-documented API supporting the development of custom widgets.

# **Acknowledgements**

This paper would not have been possible without the expert guidance of my mentor and close friend, Professor Brian Moriarty. His knowledge of JavaScript ES6, the resources he provided, and his design background were invaluable through the development of this framework. Next, I would like to thank Bill Chamberlain for his guidance when I was learning to program. I would not have pursued an education in computer science were it not for the free-form learning environment of his classroom. The frequent puzzles which illustrated fundamental computer science concepts proved to be exceptionally helpful when furthering my education. Next, I would like to thank Joe Rose for his personal guidance and mentorship throughout my early high school years. Without the persistent guidance of Rachel Palleschi, my writing skills would not have developed to the point where writing a paper of this magnitude was possible. This paper is a direct result of the impact she made on my life. David Medvitz provided me with excellent guidance through the later years of high school. His connection with me allowed me to pursue advance computer science concepts while still in high school, and he heavily encouraged me to attend WPI. Were it not for his advice I would have never have found a field I love as much as this one. Finally, I would like to thank my parents and grandparents. The encouragement from my parents and feedback they have given me through growing up has really shaped me into who I am. They have given me so much support and enabled me to pursue a dream of mine. Without the direct help of my parents and grandparents I would not have been able to attend the schools that shaped me so much, for that opportunity I am incredibly grateful.

Thank you.

Contents

[**Abstract** 1](#_Toc7080843)

[**Acknowledgements** 2](#_Toc7080844)

[**Grid** 5](#_Toc7080845)

[**Save System** 8](#_Toc7080846)

[**Widget** 9](#_Toc7080847)

[**Webview** 13](#_Toc7080848)

[**File Tree** 14](#_Toc7080849)

[**Code Editor** 16](#_Toc7080850)

[**Development Console** 18](#_Toc7080851)

[**Tab Bar** 19](#_Toc7080852)

[**Canvas** 20](#_Toc7080853)

[**Process Spawner** 22](#_Toc7080854)

[**Development Customizable Menu System** 24](#_Toc7080855)

[**Real world Applications** 25](#_Toc7080856)

[**Perlenspiel IDE** 26](#_Toc7080857)

[**FraudTek IDE** 28](#_Toc7080858)

[**3D Viewer** 30](#_Toc7080859)

[**GitHub** 31](#_Toc7080860)

[**Works cited** 32](#_Toc7080861)

[**Appendix** 33](#_Toc7080862)

[**Documentation** 33](#_Toc7080863)

## Development

Initially, the Electron Grid of Aligned Documents (EGAD) was conceived as an integrated development environment (IDE) for the Perlenspiel game engine. However, it soon became evident that, with the addition of abstraction layers, a more flexible and powerful framework could be created. The project evolved into a generalized tool supporting the creation of applications consisting of an arbitrary grid of web pages, combined with common tools useful to developers when working with Electron.

The inspiration for this project came from Visual Studio Code[[1]](#footnote-1), an Integrated Development Environment made with the Electron framework. EGAD is similar to Visual Studio Code in that it is based on Electron, and allows users to dock and move panels around. EGAD differs from Visual Studio Code in that it assumes nothing about the type of application that is being developed, except that its UI will be grid-based. EGAD handles both the initialization of grid panels and a pipeline for communicating state changes between panels. It can be used to build any Web-based application requiring a flexible panel layout.

Any EGAD panel can be occupied with prebuilt utilities called widgets. EGAD provides a small library of commonly-needed widgets, including a webpage viewer, file browser, code editor, development console, and a tab bar. The framework also documents how custom widgets can be created, giving developers the flexibility to extend EGAD for use in a wide variety of to applications. One example would be a tool that allows users to post to multiple social media sites at the same time. This application could have three webpages open at the same time, as well as a text input widget, and a post button widget. When the button was pressed, messages entered into the input widget would be posted to all three social media sites concurrently.

## Grid

EGAD’s essential functionality is its ability to create and manipulate a grid of functional application modules. The grid is made up of horizontally resizable **columns,** each containing child **rows** that can be individually resized vertically. Locations in the grid are are referred to as **cells**. For example, the grid presented in Figure 1 is comprised of five columns, each containing five rows. EGAD’s API function grid.getWidget(column, row) can be called to reference a specific cell in the grid. This returns

the widget stored in cell (column, row). If there is no widget at the location specified, the value *null* is returned. Similarly the grid.setWidget(column, row, widget) function can be used to set or replace the widget residing in a grid cell.

The grid.addWidget(column, row, widget) function is used to add an entirely new cell to the grid at the specified location. This allows for grids to have independently-formatted rows or columns. Figure 2 depicts a grid with one cell in column one, and three cells in column two. Structuring rows in this free-form manner enables a variety of application designs,

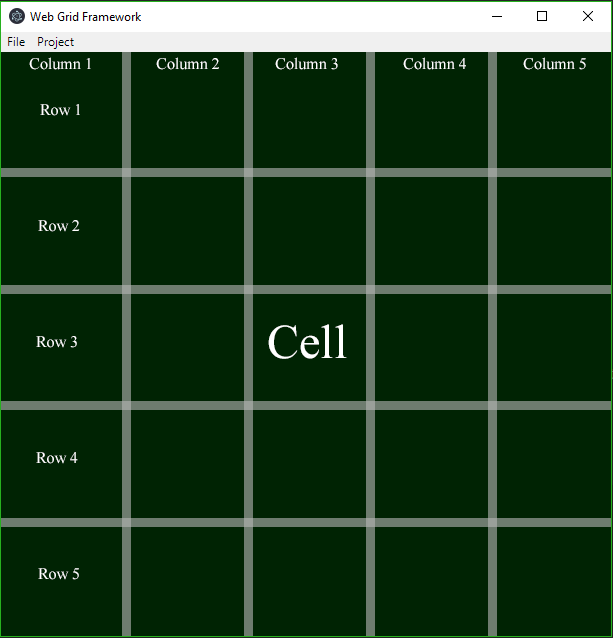


Figure 1. 5 x 5 grid of cells.

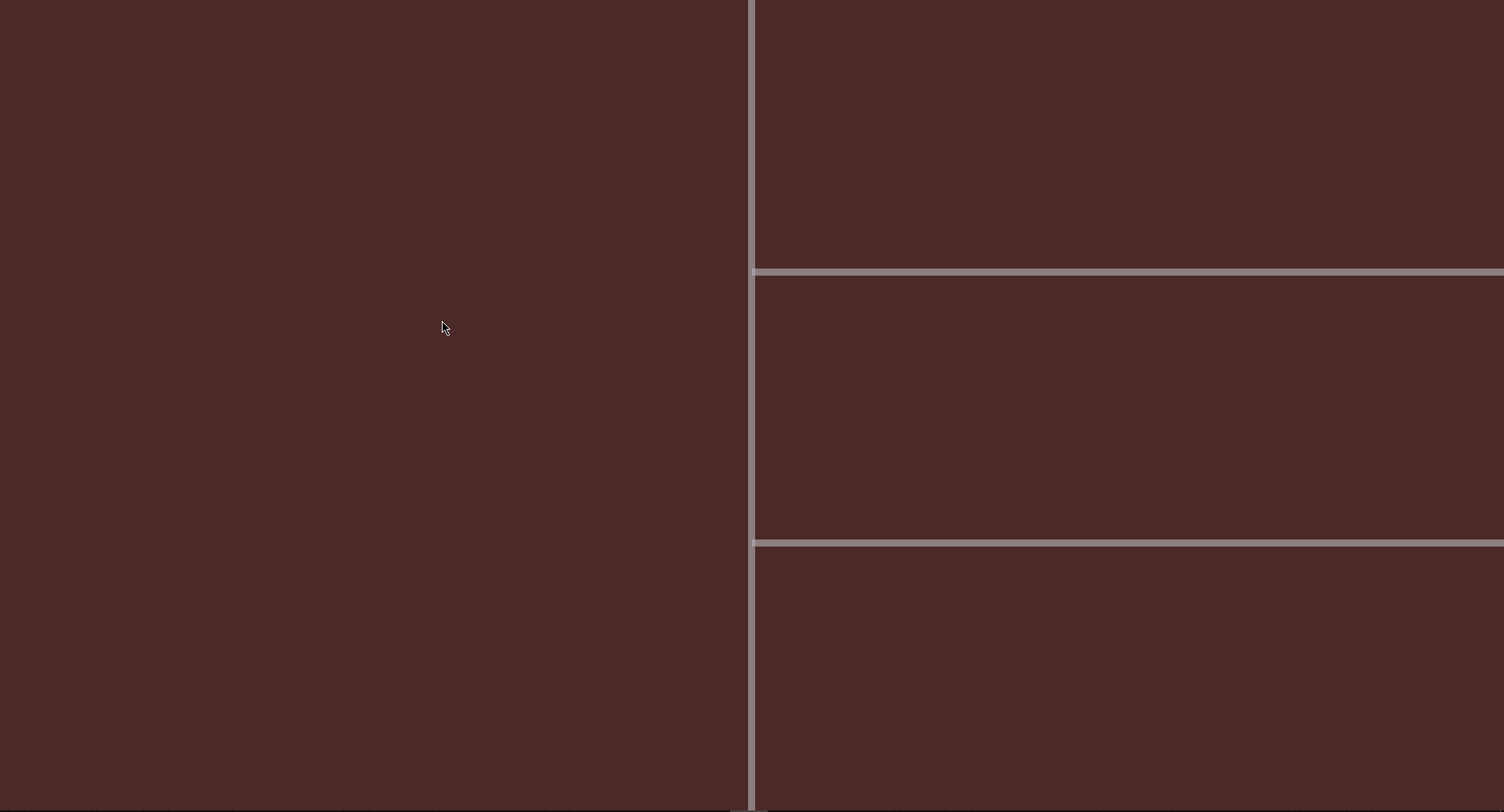


Figure . Independent column format.

thus increasing the use case flexibility of the EGAD framework.

The network of white bars distinguishing cell boundaries in the grid are referred to as **drags.** All cells in the grid detect what drags represent their boundaries, and can be resized dynamically when any of these drags are moved. To move a drag, a user simply clicks on the white boundary and drags it to a desired location. All the drags that move horizontally span the entire height of the window. This ensures that all cells within a column will have a consistent size. Rows are not consistent throughout the grid. Every column can contain an arbitrary number of rows. The only guarantee about a column is that it will have at least one row. Figure 3 illustrates how moving a drag horizontally will resize all cells in a column, however resizing a drag vertically will only change the size of cells in that column.

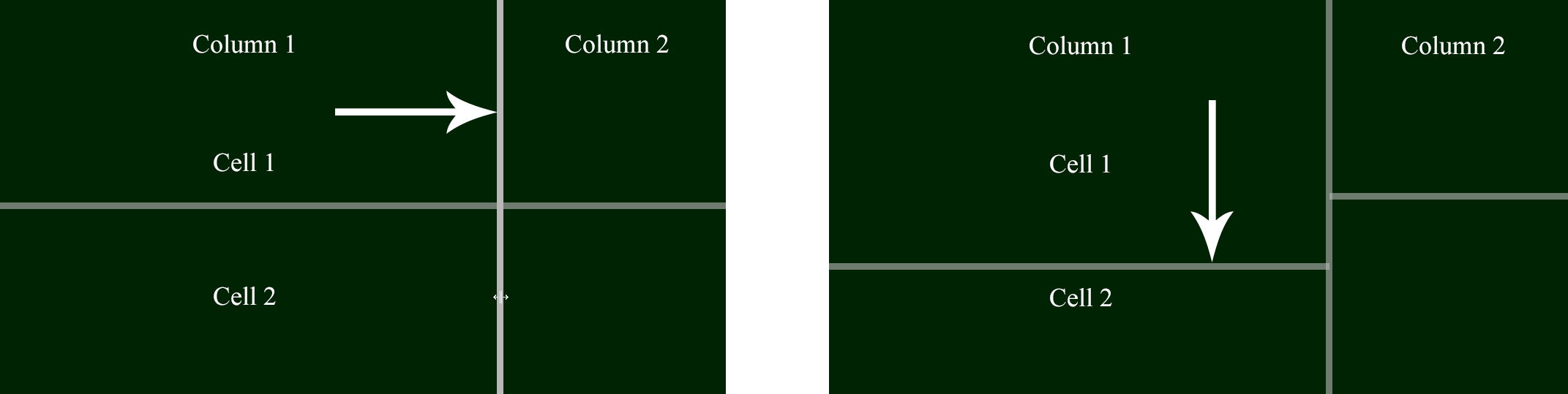


Figure 3. Demonstration of horizontal and vertical resizing.

Every cell in the grid can contain a single **widget**. Widgets are containers for displaying and manipulating data. When a grid is initialized, an array of widgets is specified. These widgets are synchronously initialized, which means that the second widget will not be initialized until the first widget is done initializing. The parent widget class is abstract; therefore, independent developers can design their own widgets to add functionality needed for specific applications. EGAD offers a library of built in widgets which provide commonly-needed functionality, allowing developers to quickly start developing an application. Figure 4 shows a 5x5 grid initialized with three instances of the built-in file tree widget. Each of these instances are completely independent. They can point to different file locations on the disk, have different stylistic themes, and can configure properties of their specific file tree without affecting the appearance or functionality of other widgets.

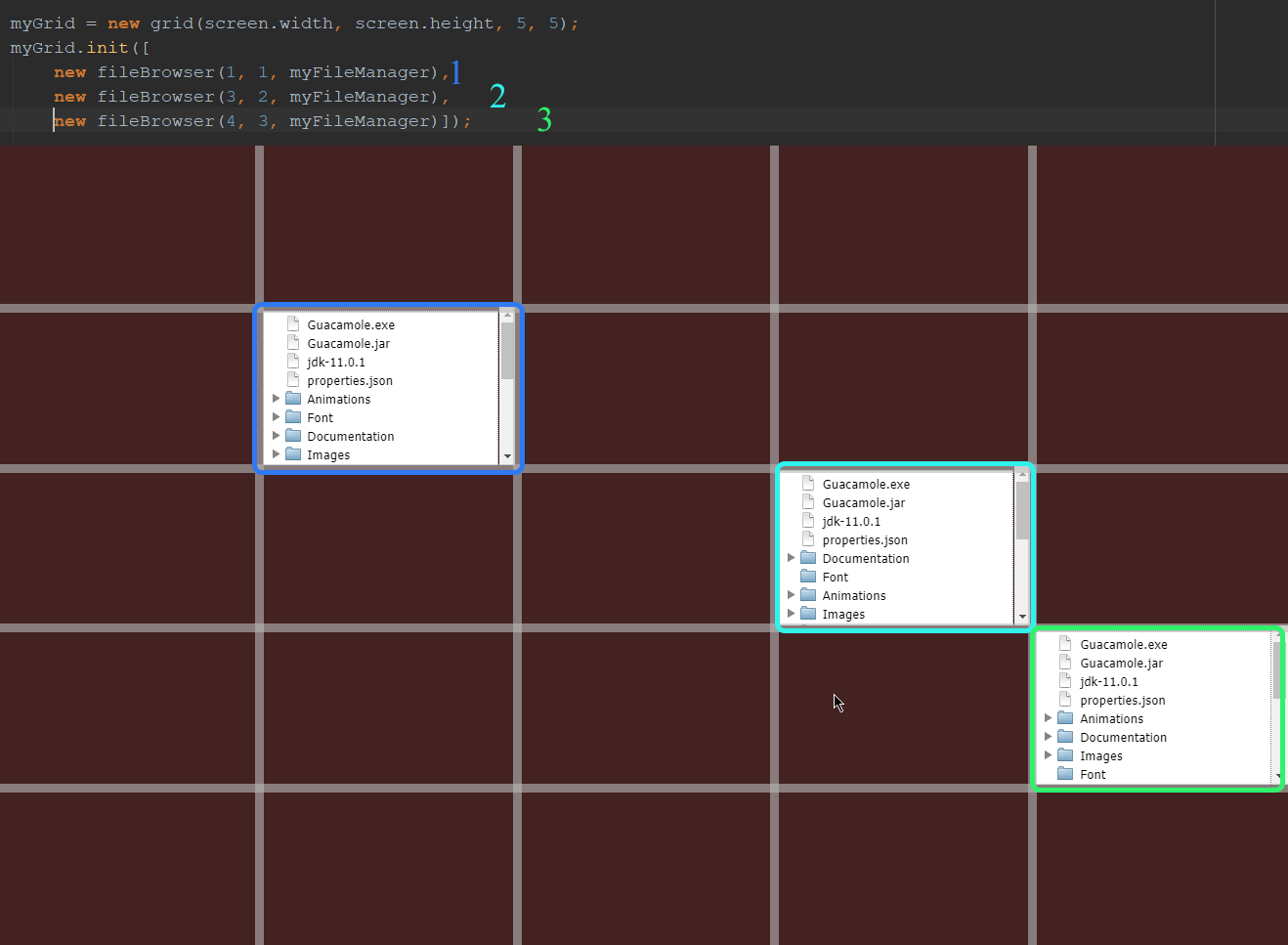


Figure . 5 x 5 grid with three file tree widgets.

The number of cells in a given column is not fixed. At any time, an EGAD application can modify how many cells there are in a column by calling the grid.addWidget() function. Widgets can be removed from the grid by calling grid.removeWidget(). These functions allow the layout of an application to be modified at any time. A possible use case for this functionality is to show a tooltip widget when the user is interacting with a certain piece of data. When the tooltip is no longer needed, the widget can be removed.

## Save System

Whenever an application built with EGAD is closed, a custom save object is generated representing the exact state of the grid. To achieve this preservation of the grid state, all columns are iterated through and their widths are recorded. Then the height of each cell in each column is recorded. This forms a two-dimensional array of width and height data that can be used to recreate the dimensions of the grid when re-opened.

Cells containing widgets need to have a way to persist data as well. The parent widget class has an abstract **Save** function that is overridden by child widget classes. The implementations of this method in child classes return JSON objects containing all persistent data needed for that specific widget. For example, the file tree widget generates a custom save object with a “path” attribute which keeps track of the directory to which the file tree is pointing. These custom JSON objects are inserted into the object containing the width and height data of the associated cell. This JSON object is then written to a configuration file specified by the developer. When the application is re-opened, the save file is loaded, and the JSON data inside is parsed back into a save object. As the grid is initialized, each widget reads its relative save information from the save object to return to the state existing when the application was closed.   
 If additional save information is needed, the developer can integrate with the fileManager class, to read and write files. This class acts as a wrapper for the default node.js file system module fs,[[2]](#footnote-2) and enables a developer to easily read, write and manage external files and integrate them with their application.

Widget Widgets are the fundamental display container used by EGAD. The parent widget class is an abstract class which defines many useful helper functions and data management tools. This parent class also defines many fields that every widget needs to support.

Widgets must contain a name describing itself, a column (x), and a row (y) position to be directly parented to in the grid, and a reference to a DOM element that the widget can internally modify. This DOM element is what is inserted into the application when a widget is parent to an (x, y) cell. Widgets also maintain a list of references to other widgets with which they may need to exchange data. This is how widgets communicate between one another.

Widgets also generate a JSON object, and store any configuration data they need inside that object as attributes. All configuration data that a widget needs when it is initialized is read from this object, which EGAD saves the object’s state when the application is closed.

The final field that every widget has is an isLoaded Boolean that returns true once a widget has successfully been initialized. This field is used when widgets want to communicate between one another. If the widget that is trying to be accessed isLoaded is false, there is no guarantee that any field within that widget will be defined. If isLoaded is true, then the widgets constructor has been called, insuring that all fields on the widget are initialized and safe to reference.

In order for widgets to rely on one another, it is paramount that the initialization order of all widgets can be controlled. If widget *A* relies on widget *B,* and widget *A*’s constructor references fields in widget *B*, if widget *B* has not finished initializing, some or all of its fields may be undefined. One would think that this solution could be solved by simply assuring that widget *B* will have been initialized by widget A’s constructor. However, it is not that simple. Assume that we create widget *B* immediately before widget *A* as is seen in Figure 5.

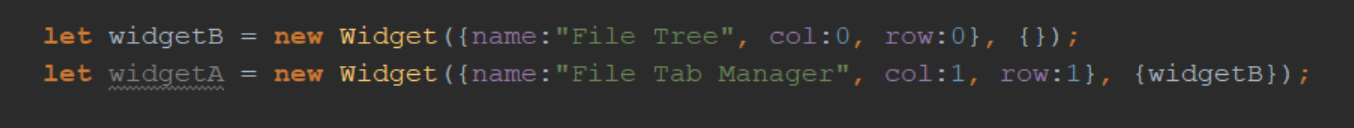


Figure . Importance of initialization order.

Widget *B* could spawn asynchronous processes in its constructor. This would cause widget *B* to be in a state where its constructor has terminated yet not be fully initialized. Since its constructor has terminated, the next line of the program would execute to create widget *A*. If widget *A* goes on to reference widget *B* in its own constructor, there is no guarantee that the fields being accessed will be defined. Now widget *A* has finished executing its constructor with references to undefined in places that should reference fields of widget *B*. By this time the asynchronous calls made by widget *B* have finished executing, and widget *B* updates its DOM element to use the data retrieved from the asynchronous calls. This will put the grid in a state where widget *B* looks as if it were initialized first, however widget *A* really finished initializing first, and contains references to invalid data taken from widget *B*.

The solution to this problem is to assert that any widget created must return a promise from its initialization call. Any asynchronous calls made within this constructor must be handled in such a way that the promise does not resolve until all asynchronous calls have terminated. With these rules in place, an array of widgets can be initialized within a locking loop, as illustrated in Figure 6.

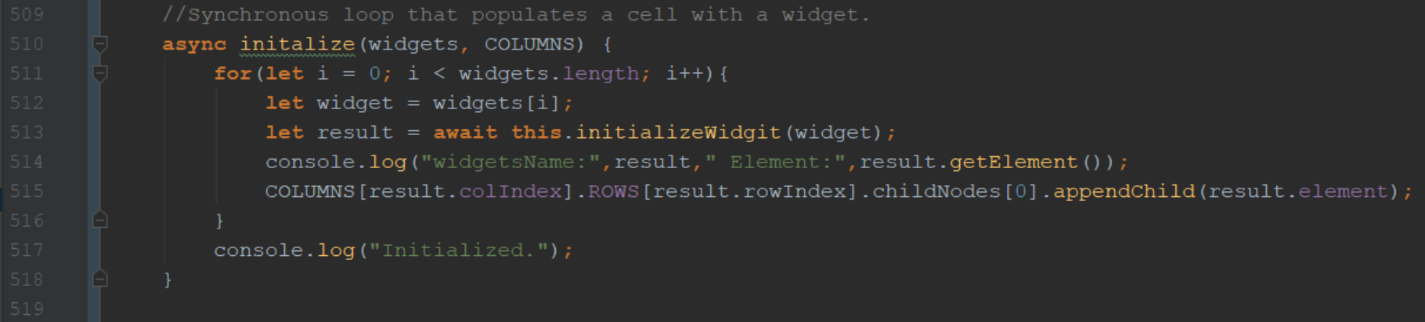


Figure . Implementation of synchronous loop structure.

This loop uses a new feature of JavaScript ES6, async await[[3]](#footnote-3). The keyword async can preface any function within JavaScript. This keyword asserts that within the body of the function some asynchronous function will be executed. This keyword also enables the use of a second keyword, await. The keyword await must preface an asynchronous function call. When the code in Figure 6on line 513 is interpreted, the execution of line 514 will halt until the asynchronous call spawned from line 513 terminates. The code being executed on line 513 is evaluating the promise returned from a widget’s initialize function. This promise was designed to not return until all asynchronous calls in a widget’s initialize function have terminated. Therefore, Figure 6 depicts a blocking chain that will wait to initialize the next widget in the chain until the previous link has finished initializing. This design ensures that widget *B* will be initialized by the time it is passed to widget *A*.

One of the design goals for widgets was for them to encapsulate all of their functionality within their class. The reason for this is that it enables any widget developed for an EGAD application to be deployed in any other EGAD project. In the future, users will be able to search the web for an EGAD widget that they would like to include in their project, and hopefully find one that works as-is, or which can edited to fit their needs. The open-source EGAD project will allow users to develop applications quickly through open-source widgets developed by other community members.

EGAD’s library of pre-defined widgets are intended to help developers implement common functionality quickly, and to demonstrate how flexible widgets can be. The included widgets are a webpage viewer, a file browser, a code editor, a tab bar, and a console. These widgets are especially useful for developing Integrated Development Environments. Initially this project was targeted at making IDE’s, however with changes like the abstract widget class, many more applications can be developed with this framework. Rather than making an exhaustive set of widgets for all possible applications, the abstract widget structure was created to allow developers to build their own new widgets easily.

Figure 7 depicts a custom widget built for the EGAD framework. The application that this custom widget was developed to allow users to manipulate models in 3D space. A helpful tool for this kind of manipulation is a transform viewer that displays the rotation and scale of a 3D model. The widget depicted in Figure 7 uses three sliders and an array of text cells to show the transform of the 3D model’s current position in space. The widget can be placed into any cell and will automatically display itself. The widget also records its current sliders values inside of its save object. This preserves the state of all sliders between the application closing and opening. This widget was quickly implemented, relying heavily on the built-in properties of widgets to manage persistent data and display the widget on the screen.

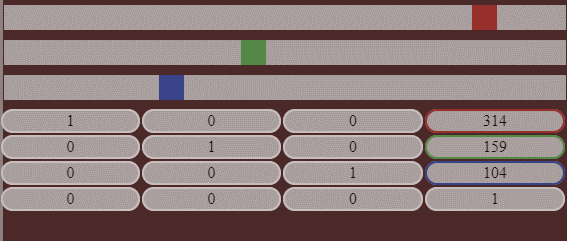


Figure 7. Custom transform widget.

### **Webview**

The goal behind the Webview widget is to wrap the Electron Webview[[4]](#footnote-4) DOM element within an EGAD widget. Webviews act as more powerful IFrames, in that they allow a developer to tap into the output stream of a webpage. Webviews also provide access to a V8[[5]](#footnote-5) instance interpreting any JavaScript in the webpage. The ability to access these streams is wrapped to internal functions that allow a developer to execute a callback whenever data is written to the page’s output. Other calls exist to run JavaScript within the Webview. This provides developers with immense power and control over the pages that they display within Webview widgets. Developers can use JavaScript to interact with all aspects of the webpage and listen to responses from the webpage through the output stream.

Developers can embed any webpage into their application by simply specifying the URL of the desired page. The URL can either point to an external webpage hosted on the Internet, or it can specify a local path to an .html file. The ability to communicate directly with these web pages

allows developers to have web pages interact with one another. Figure 8 depicts a 3x3 grid with four Webviews inside of it. The centermost Webview is pointing to an HTML file included in the project, where the other thee Webviws are pointing to popular social media sites.

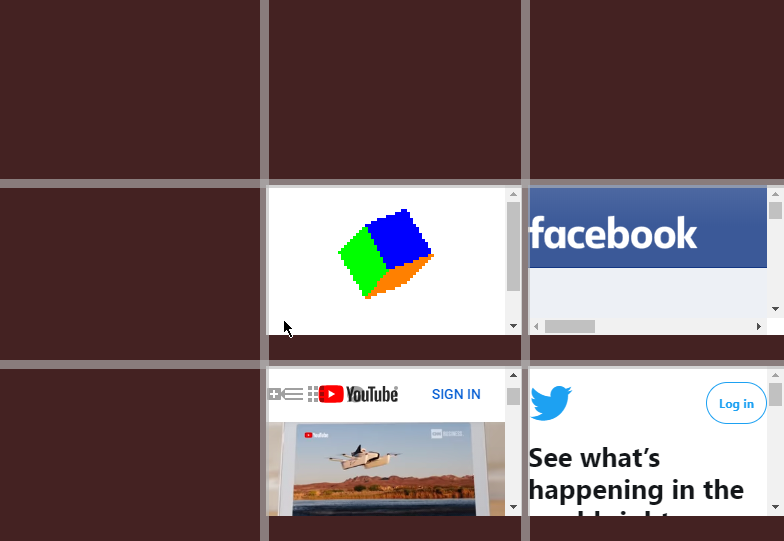


Figure . 3 x 3 grid of Webviews.

### **File Tree**

One feature which many projects rely on is the ability to traverse a file structure. To implement this within EGAD, the FancyTree[[6]](#footnote-6) library was selected and wrapped within a widget. This library requires the developer to pass a JSON object representing a file directory structure to the fancy tree class constructor. In order to generate this JSON object, the node.js file system module is used. This module allows a computer’s local storage to be accessed through JavaScript using EGAD’s getProjectFiles() function, which allows a user to access all files and subdirectories within a specific directory. This method also utilizes an ignore object to omit certain files from the JSON object returned. Much like a .gitignore[[7]](#footnote-7), this object is a blacklist of files to exclude from the returned object. If the developer wanted to ignore all .html files in all directories, they would use the wildcard ‘\*’ character. This would appear as “\*.html” inside of their ignore object. If only a specific file should be ignored, it can be excluded by simply typing its name (such as “index.html”) inside the ignore object. If all files of a specific name with different file extensions should be ignored, the wildcard character can be used. The following would ignore all files named example with any file extension: “example.\*”.

Figure 9 depicts the output of the getProjectFiles()method, and shows the resulting File Tree widget that is generated from this data. The desired outcome of this widget is to allow the developer to simply specify a directory that they want the user to have access to, and have this directory and all sub directories represented visually in an interactable tree. Much like the Webview widget, the File Browser widget requires an additional URL parameter to be passed in through the widgets configData object. This URL can point to any location on the host computer.

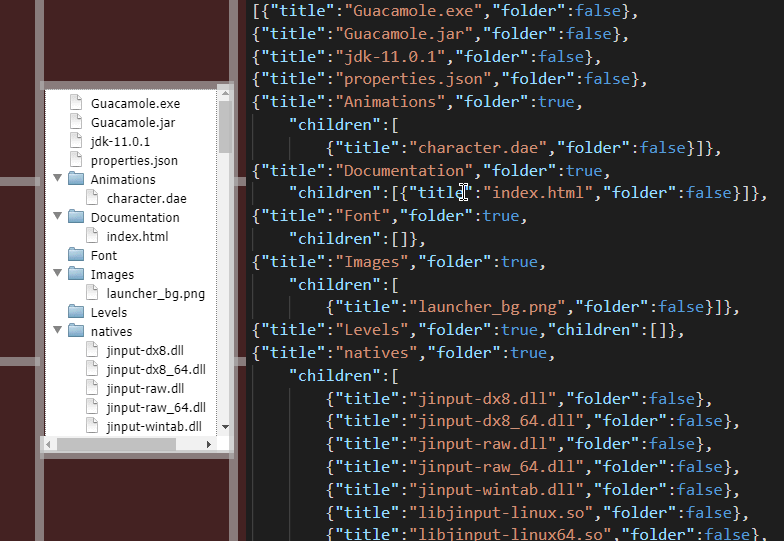


Figure . Sample output of getProjectFiles() method.

In future versions of EGAD, this module will be improved to support remote file server access through the FTP[[8]](#footnote-8) protocol. This will enable far more complex applications by allowing developers to create an FTP widget which could connect to a remotely-hosted server or database and display the resulting files in a tree structure.

### **Code Editor**

The Code Editor widget is a language-agnostic editor which allows a user to edit any programming language specified by the developer. This widget also contains a custom Language Parser which suggests commands that a user could be trying to type. Code Mirror[[9]](#footnote-9) is the base editor tool that the widget is designed and built around. Code Mirror is open source, widely used, incredibly flexible, and well-documented making it an excellent choice for a language-agnostic editor. The widget takes in an additional parameter calledlanguageMap which associates file extensions with languages. This widget has configurable hotkeys for saving the code being edited to a file, as well as facilities for copying, pasting, and commenting code, and registering function callbacks to be executed whenever a hotkey is triggered.

The language parser works by loading a JSON object which describes the language. Information such as variable keywords, scope declaration characters, comment headers and footer as well as any function names are included in this object. When a file is loaded, the file extension is checked against a map of known associations stored within the Code Editor widget. For example, if a file was opened with the extension .js, the JavaScript configuration file would be loaded. Then all scopes within the file are detected and a tree structure is generated. This tree has information about the line start and end of every scope in the file. This information allows the Language Parser to know exact which scope any line of the file is within. Additionally, whenever a new line is added or removed from the file, all scopes below that point are offset such that the line start and end values of all scopes in the tree hold true throughout editing the file. After all scopes have been established, the file is iterated through line by line and broken up into smaller tokens. These tokens are compared to the list of known variable keywords to try and determine what the user is trying to type. When variables are created, they are inserted into the scope that the cursor currently. Figure 10 shows an example of a JavaScript file with scopes and variables highlighted. Whenever a new character is added to the file, the changed line is tokenized. The last token is compared to all known functions that exist in the language, as well as any variables that are accessible in that scope. The parser then generates a set of predictions as to what variable or function names the user could be trying to type. The user can then confirm one of the suggestions to autocomplete what they were typing without typing the entire expression.

A real-world use case for this “IntelliSense” functionality is suggesting library functions for the Perlenspiel game engine. In Figure 10, the IntelliSense being displayed is referencing a JSON object listing all library functions of Perlenspiel. This presents programmers with possible functions that they could be trying to reference and inserts the exact spelling of the library functions into the code editor.

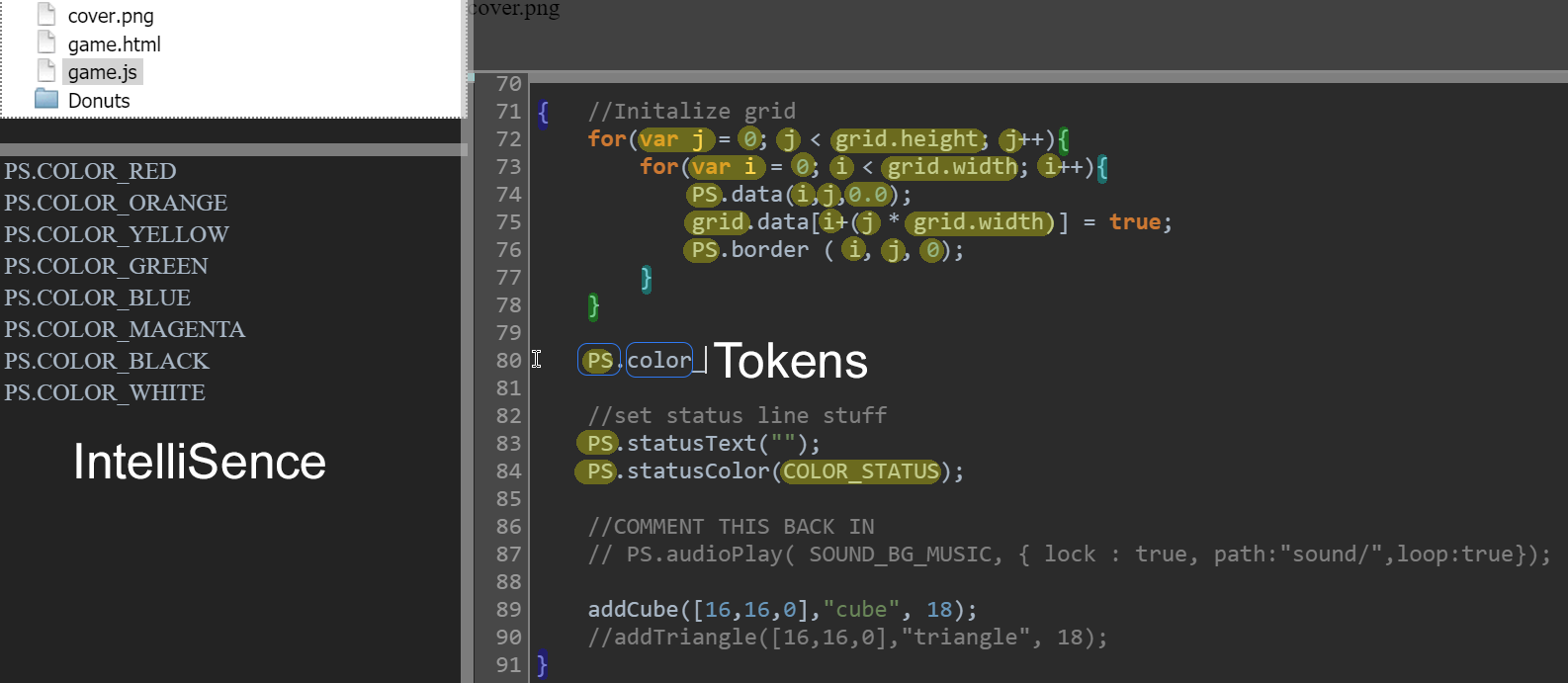


Figure . Example of Code Editor widget.

**Development Console**

The Console widget acts as a wrapper to interface with standard input and output streams. This widget was developed to integrate with Webview widgets or processes spawned from EGAD. Figure 11 depicts the output of a Webview widget wrapped to an instance of the development console widget. Whenever the Webview writes to its output stream through a JavaScript call to console.log, the stream text is also passed into the development console widget. The development console widget simply wraps this output to its own output text area. The development console also has an input text field. The arrows in Figure 11 depict a command being sent to the stdin stream of the WebView. The command that is being interpreted tells the Webview to write “Test” to its console. This write to console is then sent to the Webview’s output stream which causes “Test” to show up in the Development Console output area.

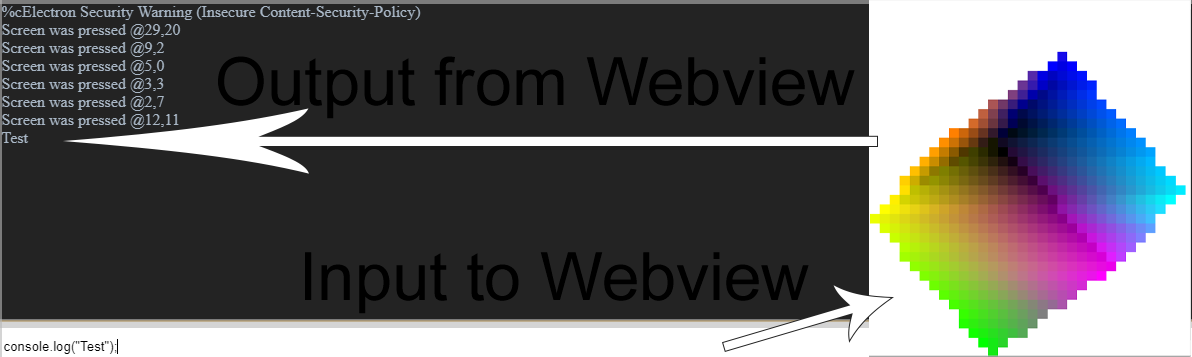


Figure . Illustration of stream flow.

Developers are also able to link Development Console widgets to other processes spawned by EGAD. Whenever EGAD spawns a process, references to the input and output streams of that process are stored for later use. These streams can easily be mapped to a Development Console to view the print statements generated from a running process.

**Tab Bar**

The Tab Bar widget allows a user to switch between multiple files open at the same time. This widget requires a reference to a File Browser widget. The Tab Bar widget listens for the file tree to signal that a file has been double clicked on. When this event is sensed, the file that was clicked on is passed to the Tab Bar widget, which looks at the file extension and executes the callback function registered for that type of file. It is the developer’s responsibility to provide callback functionality for every file type that they want the ability to open. Figure 12 shows the result of a callback function which adds an image div to the DOM when a “.png” file is clicked. Whenever a new file is clicked on, the Tab Bar will create a new tab for that specific file. These tabs can be clicked on to run the callback function for that file.

Currently, EGAD tabs are not able to be closed or moved around. This limited functionality is due to the time invested in this project so far. The Tab Bar widget was not a priority for the sample applications being developed, so only basic functionality has been integrated so far. In future versions of EGAD, polishing up the visuals of the tab bar and enabling users to drag tabs around will be priority features.



Figure . Callback Function on Tab Bar widget.

**Canvas**

The Canvas widget was designed to provide an easy way for developers to integrate with WebGL[[10]](#footnote-10). WebGL is a web implementation of the Open Graphics Library (OpenGL) [[11]](#footnote-11) which provides hardware acceleration for graphics applications on web pages. Since each cell of the EGAD grid is its own web page, WebGL can be used to provide a hardware-accelerated canvas to the developer. WebGL canvases are often used as the basis for HTML5 games. If a developer were to make a game engine based off the EGAD framework, a cell could be populated with a WebGL canvas and other cells could be used for debug information and world editing utilities. Also, since EGAD is written on top of Electron, the project can be compiled to native executables and distributed across systems easily.

The Canvas widget itself preserves no information between runs of the application, and simply acts as a display which other JavaScript files can subscribe to. If a developer writes a piece of code to integrate with WebGL, such as a render function that draws an element, they can send this element to the canvas by calling **canvas.subscribeToDraw(<drawFunction>).** This function adds the specified function to an array of callbacks to be executed whenever the canvas redraws. This way developers can add draw functionality to the canvas class without modifying the canvasWidget class itself. These callback functions are executed at a fixed interval; EGAD assumes that the fastest the canvas can be refreshed is 255 times per second. Every additional draw call added to this function increases the total time it takes to render a frame of the game, possibly decreasing the overall performance. Since most games target 60 fps, there is a lot of leeway for adding additional draw calls. If a developer wants to set the canvas to render at a fixed frame rate, they can simply call **canvas.setFrameRate(int frames)** to limit the refresh rate to ‘frames’ per second.

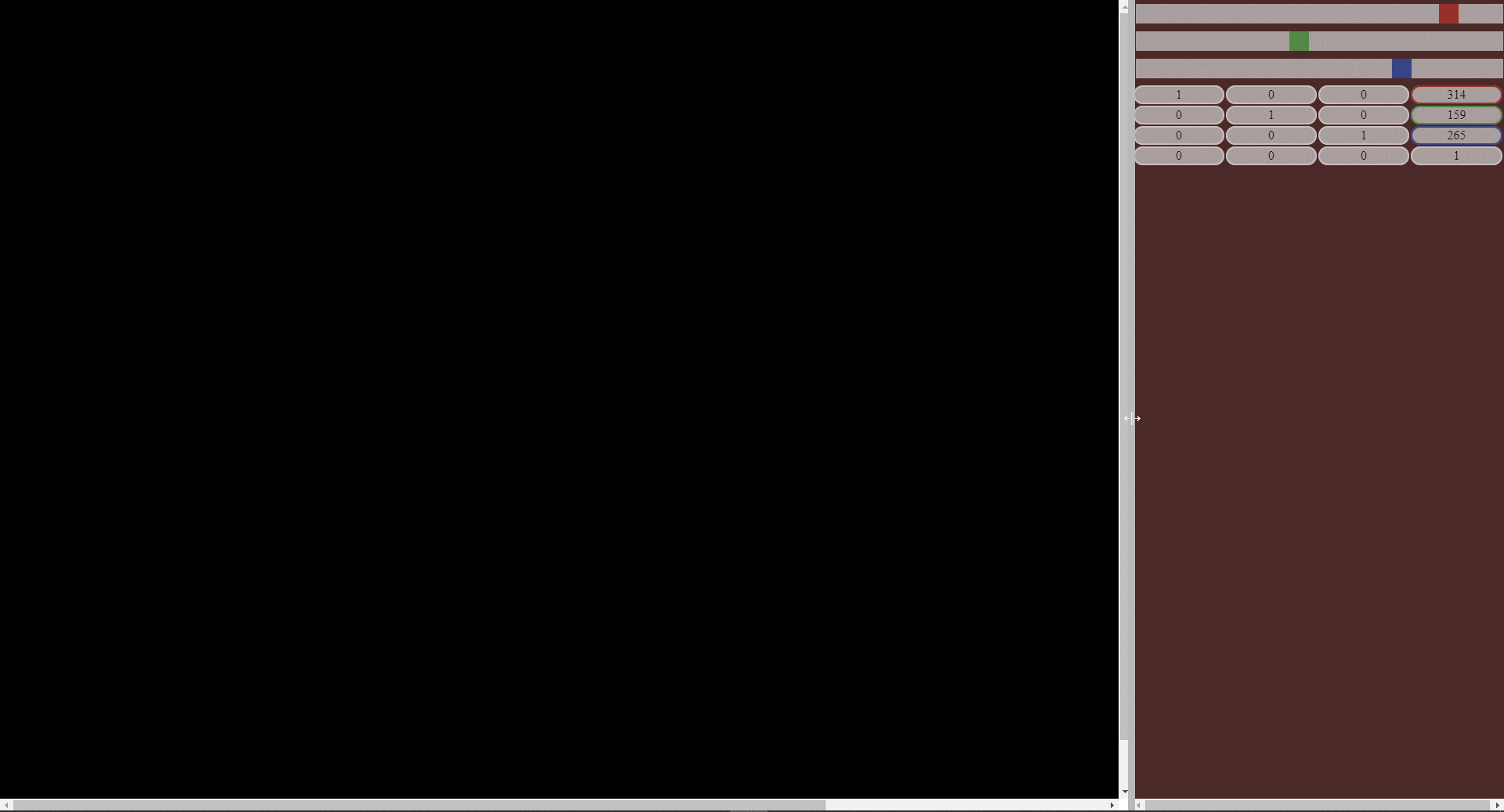


Figure . Blank WebGL canvas.

Figure 13 depicts a blank Canvas widget adjacent to a Transform widget. There is nothing displayed in the canvas window, because WebGL implementation is the application developer’s responsibility. There are many ways to implement a WebGL renderer, rather than one best way to implement things. Libraries such as pixi.js and three.js[[12]](#footnote-12) provide excellent implementations of many commonly desired WebGL use cases. The canvas being blank allows it to easily be used as a render target for libraries such as three.js.

## Process Spawner

One of the most powerful features of node.js[[13]](#footnote-13) is its ability to start a native process and capture the streams going into and out of that process. EGAD encapsulates this functionality inside a utility class called processSpawner**.** This class allows the developer to spawn native processes and pass callback functions to the input and output streams of those processes. Any time stdin detects input, it will trigger the callback function passed into the processSpawner and forward the input data to the callback function. Similarly, any time stdout detects that data has been written, the callback function for stdout will be triggered with the output data passed to the function. The processSpawner class also allows a developer to register interest in a process terminating through an additional callback function. This function is triggered whenever the spawned process terminates, and the exit code of the process is forwarded to the onClose callback function.

Figure 14 shows the output stream of a spawned process being mapped to the V8 developer console. The spawned process is a game engine which sends all of its logging data to standard out. These prints are caught and logged to the console. The console also has an input field where a user can type. Anything that is typed into this input field is sent to the game engine. The game engine sends the text from this input stream directly into its scripting engine and assumes that the sent code is valid JavaScript. The engine will then interpret the script and compute the result. This is useful for debugging and gives developers access to variables inside the game engine at runtime

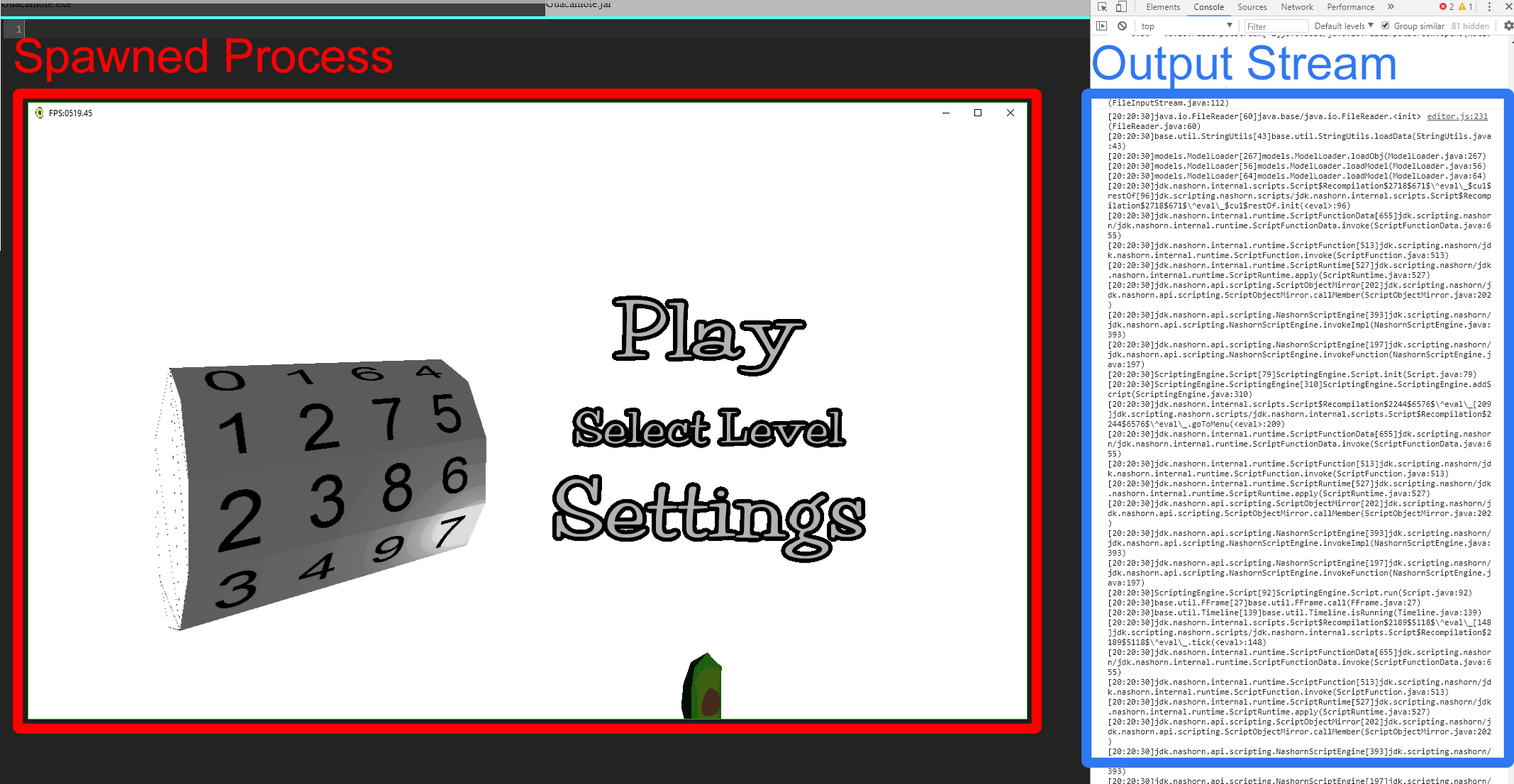


Figure . Depiction of spawned process output stream.

One possible application for the processSpawner is to compile and then run code written inside a Code Editor widget. Another possible application for the processSpawner is to run test cases on a running program. If internal variables of the process can be queried through stdin, and then printed to stdout, a simple automation test application could be written to check that objects are in certain states at certain times. This would allow users to create automated test cases, and regression tests to help maintain functionality in an application throughout the development lifecycle. Developers would even be able to detect the program closing prematurely by registering a callback function when the process terminates. They would then be able to compare the exit code of the process against a known return value to ensure that the process ran to completion.

## Development Customizable Menu System

Applications have drastically different menu layouts. In order to support as many layout options as possible, nothing can be assumed about the menu requirements for an application. If a Fullscreen application or game were to be developed a menu may not be wanted at all. In a photo editing application, there could be extensive menus. All customization and callback functions need to be pushed onto the developer’s hands. In EGAD, this is implemented through a straightforward interface which allows the developer to map callback functions to hotkey commands. This mapping can then be parented to a menu tab the developer creates such as “file” or “edit” or “preferences”. This menu functionality is encapsulated within a helper class. New menu tabs can be defined by simply calling the function **menuBuilder.addMenuDropDown(<name>).** This function will create a new tab called ‘name’ to the menu. To add functions to tabs, a developer simply needs to call **menuBuilder.registerFunctionCallback(<tab>, <name>, <key>, <function>).** This function adds a new option to <tab> with the name <name>. For instance, a developer may add the ‘save’ option to the ‘file’ tab. <key> indicates the hot key which triggers this menu option. In the case of ‘save’ this key would most likely be ‘S’ so when ‘ctrl + S’ is pressed the ‘save’ function will be called. The <function> parameter is the string name of a function within editor.js to execute when this menu item is triggered. Figure 15 shows a menu next to the code required to build that menu.



//---------------------------------------------------------------------------  
// Build Menu here  
//---------------------------------------------------------------------------  
**let** file\_dd = menu.addMenuDropDown("File");  
menu.registerAppCallback(file\_dd, 'Quit', 'Q', 'quit');  
menu.registerFunctionCallback(file\_dd, 'Save', 'S', 'save');  
menu.registerFunctionCallback(file\_dd, 'New', '=', 'newWidget');  
menu.registerFunctionCallback(file\_dd, 'Remove', '-', 'removeWidget');  
menu.registerWindowCallback(file\_dd, 'Developer Console', 'I', 'toggleDevTools');  
**let** help\_dd = menu.addMenuDropDown("Help");  
**let** test\_dd = menu.addMenuDropDown("Test");  
**let** project\_dd = menu.addMenuDropDown("Project");

Figure . Code needed to build a menu.

# **Real world Applications**

To accompany the release of EGAD Version 1.0, three example applications were developed to provide a starting point for different kinds of applications. The first project that was developed was a language-specific Integrated Development Environment called the Perlenspiel IDE. Perlenspiel is the custom library that this IDE was developed to edit. The language file that Perlenspiel uses is included and well documented, so users can adapt this application to fit their language-specific needs.

The second application which was developed is a tool that spawns a process and then maps the input and output streams of this process to the V\* developer console. The purpose of this application is to show users how easily native processes can be manipulated through this framework. This application could be modified to fit a wide variety of needs. The final application that was developed is a 3D Model viewer which uses an OpenGL enabled Canvas widget, as well as a custom Transform widget. This application allows users to view 3D .ply files and modify their transforms in space. The reason this application was developed is to show users how simple it is to create new widgets for the EGAD framework. The Transform widget is a well-documented custom widget that will work in any EGAD project. These three applications serve as jumping-off points for users who want to develop their own applications in EGAD.

## Perlenspiel IDE

Professor Brian Moriarty developed the Perlenspiel game engine for use in his digital game design courses at WPI. In these courses, students use JetBrains WebStorm[[14]](#footnote-14) to develop web games in Perlenspiel. WebStorm is an integrated development environment that focuses on developing websites. This tool works very well for developing Perlenspiel games; however, there are several missing features that would allow students to develop Perlenspiel games faster. A tool with features such as, the ability to view the output game in real time, IntelliSense recognizing and suggesting Perlenspiel library functions, and the ability to view and change variable values in real time, would allow students to debug their games in new visual ways not possible with traditional WebStorm.

EGAD would be an excellent choice for developing the application described above. The editor itself would be comprised of a grid utilizing many of EGAD’s built-in widgets. The gird would be comprised of three columns. The left column would contain a file tree, and a custom documentation widget. The center column would contain a Tab Bar widget, a Code Editor widget with a custom language file, and a Console widget. The right column would contain a Webview widget pointing to the Perlenspiel game being developed as well as a Webview widget displaying the documentation for Perlenspiel. This design provides exactly enough functionality for Perlenspiel specifically and only requires the development of one custom widget.

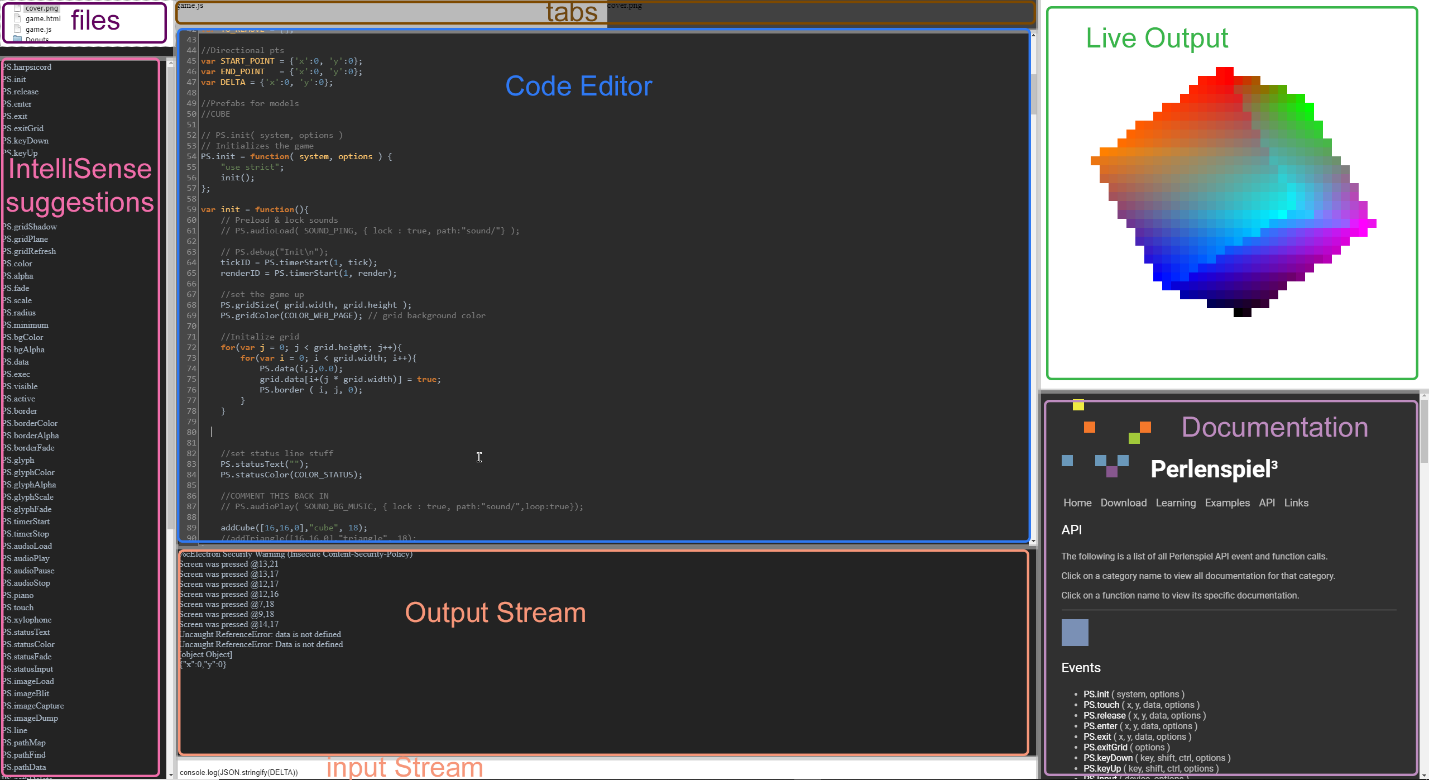


Figure . Perlenspiel Integrated Development Environment.

Figure 16 shows version 1.0 of the Perlenspiel IDE Application. Whenever code is typed in the code editor, the language interpreter generates IntelliSense suggestions which appear in the leftmost column. Every time code is saved, the live output is updated. Any console logs from the live output are wrapped to the output stream. Code can be injected to the live output through the input field under the output stream. This input field is evaluated by the live output’s V8 instance, which allows for any variable to be changed in real time.

This application allows for easier development and testing of games made with the Perlenspiel game engine. Rather than saving a Perlenspiel webpage and replaying the game to the state where changes have been made, developers can evaluate and change the values of variables throughout the development process in real time.

## FraudTek IDE

The FraudTek game engine is written with a Java backend which interprets JavaScript game code at runtime. The engine loads a custom library into its JavaScript interpreter in order to register many additional commands and primitive types. These additions are unique to FraudTek and unknown to all JavaScript IDE’s IntelliSense. This makes it difficult to write FraudTek scripts because spelling errors and referencing unknown library calls are not caught by a traditional IDE. The editor simply does not know if a reference is spelled correctly or not because it has no knowledge of these library specific references. Another problem with script creation in traditional IDE’s is that there is no way to directly execute the FraudTek script interpreter to test the newly created scripts. The script interpreter is a java executable that wraps errors to stdout and allows running scripts to be modified in real time through stdin.

The EGAD framework would allow a developer to easily create a IDE to create FraudTek scripts. A 2x2 grid could be created containing a FileTree widget, a custom IntelliSense widget, a Code Editor widget, and a Console widget. The FileTree would point to the active FraudTek project directory. All custom language functions would be added to a custom language file which integrate with the custom IntelliSense widget to provide suggestions about the FraudTek Script that is currently being written. The Code Editor widget would be formatted as if it were plain JavaScript, however, have knowledge of the custom FraudTek Functions through the custom language file. The Console widget would be wrapped to an instance of FraudTek spawned through the process spawner. Overall this application could be created with minimal custom development and provide an incredibly more intuitive editing experience when creating FraudTek files.

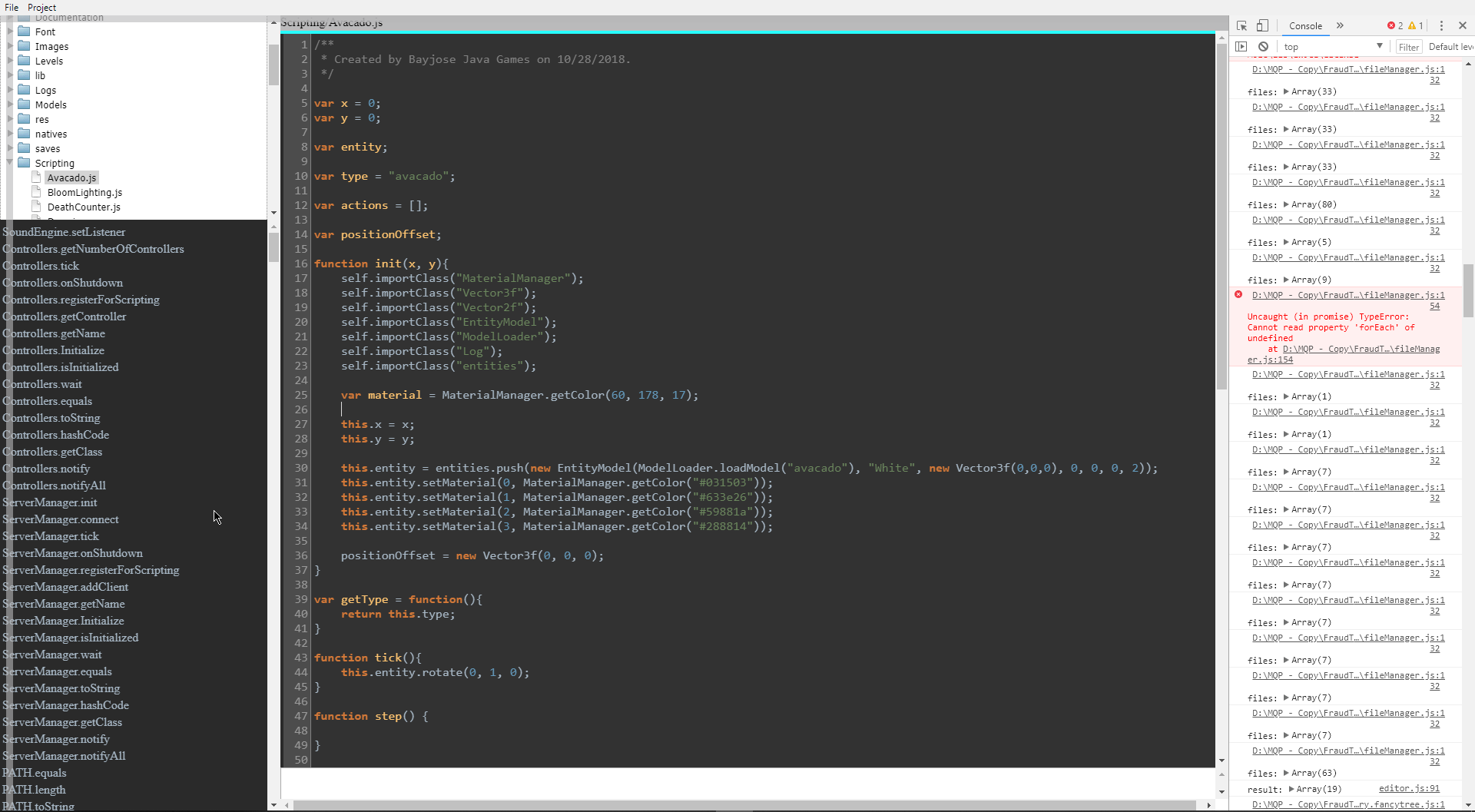


Figure . FraudTek Editor.

Figure 17 depicts the FraudTek Editor. A File Tree shows all files in the project in the top left corner. Below that, a IntelliSense shows all possible functions that can be called from the current line the cursor is on inside of the Editor. The Editor shows a script file which can be edited and interpreted at any time. Below the Editor a Console widget allows the developer to query a running instance of FraudTek to monitor variables or modify a running script. The Right column is the V8 output console. This output is wrapped to the FraudTek process returned from the process spawner and is updated whenever a new instance of FraudTek is run. Overall this application took little time to build and provides immense utility to a FraudTek developer.

## 3D Viewer

The 3D viewer is an application to demonstrate the 3D performance capabilities of the EGAD library. The application uses a Canvas widget and a Transform widget to allow the user to modify the rotation of a model in 3D space. Figure 18 shows a WebGL enabled canvas on the left and a Transform widget on the right. The canvas widget can interact with the WebGL canvas to modify the rotation of various objects in the scene. In the future, this application will be further developed into a 3D game engine, where users can click on elements on the WebGL canvas, and see that entities properties in the right panel.

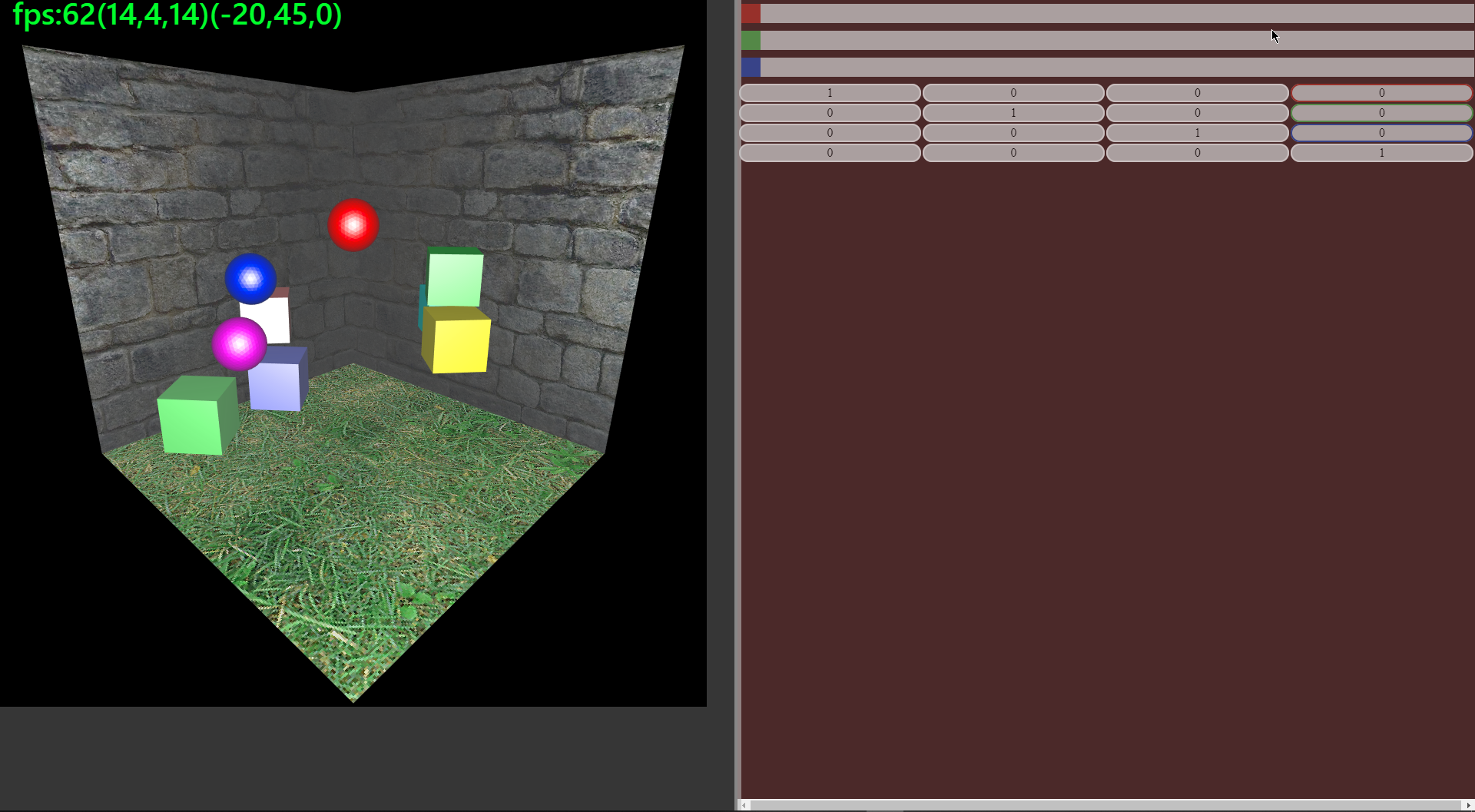


Figure | WebGL enabled Canvas with Transform

# **GitHub**

EGAD is an open source framework available on GitHub which includes everything needed for a developer to get started. Anyone can view the entire source of the EGAD Framework directly on GitHub, or fork the repository to make their own improvements. If a drastic oversight was made while developing the framework, anyone can come up with a fix and submit a pull request to have their change added to the official EGAD repository. The documentation for EGAD is also included in the repository to encourage developers to learn how to use the built-in functionality of EGAD, as well as extend that functionality. The project can simply be cloned to a developer’s computer. This project is open source to facilitate adoption and encourage adaptation.

The repository can be viewed at <https://github.com/bhsostek/EGAD>

# **Works cited**

Electron  
<https://github.com/electron/electron>

Code Mirror  
<https://codemirror.net>

Light Table  
<http://lighttable.com>

Atom Editor  
<https://github.com/atom>

Node JS  
<https://nodejs.org/en/>

vs Code using Electron as a backend  
<https://arstechnica.com/information-technology/2015/04/microsofts-new-code-editor-is-built-on-googles-chromium/>

ES6 async await:  
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function>

Fancy Tree:  
<http://wwwendt.de/tech/fancytree/demo/sample-api.html>  
<https://wwwendt.de/tech/fancytree/doc/jsdoc/FancytreeNode.html>

Gitignore  
<https://git-scm.com/docs/gitignore>  
  
FTP  
<https://tools.ietf.org/html/rfc959>

# **Appendix**

## Documentation

1. <https://code.visualstudio.com> [↑](#footnote-ref-1)
2. <https://code-maven.com/reading-a-file-with-nodejs> [↑](#footnote-ref-2)
3. <https://www.ecma-international.org/ecma-262/8.0/#sec-async-function-definitions> [↑](#footnote-ref-3)
4. <https://electronjs.org/docs/api/webview-tag> [↑](#footnote-ref-4)
5. <https://v8.dev> [↑](#footnote-ref-5)
6. <https://github.com/mar10/fancytree/wiki> [↑](#footnote-ref-6)
7. <https://git-scm.com/docs/gitignore> [↑](#footnote-ref-7)
8. <https://tools.ietf.org/html/rfc959> [↑](#footnote-ref-8)
9. <https://codemirror.net> [↑](#footnote-ref-9)
10. <https://www.khronos.org/webgl/> [↑](#footnote-ref-10)
11. <https://www.khronos.org/opengl/> [↑](#footnote-ref-11)
12. <http://www.pixijs.com/>; <https://threejs.org> [↑](#footnote-ref-12)
13. <https://nodejs.org/en/> [↑](#footnote-ref-13)
14. <https://www.jetbrains.com/webstorm/> [↑](#footnote-ref-14)