[Company name]

Electron Grid of Aligned Documents

A Framework for Creating Gird Based Applications

Bailey Sostek
1-9-2019

Abstract

This paper documents the design and build process for the Electron Grid of Aligned Documents. 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 and add additional functionality that a developer may want to incorporate into their application. There are several built in modules which developers can include in their application right away, as well as a documented library stating how new widgets can be developed quickly.

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 grand parents 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

Development

Initially the Electron Grid of Aligned Documents (EGAD) was designed to be an integrated development environment for the Perlenspiel game engine, however it became evident that with several abstractions a much more powerful framework could be created. The project evolved into creating a grid of web pages and common tools useful to a developer when working with Electron. The goal of this framework is to provide developers with a subset of prebuilt utilities which enable developers to create grid-based applications with ease. The inspiration for this project came from Visual Studio Code[1], an Integrated Development Environment made with the Electron framework. The Electron Grid of Aligned Documents is similar to Visual Studio Code in that it is based on Electron and allows users to dock and move panels around. The Electron Grid of Aligned Documents is unique in that it assumes nothing about the type of application that is being developed besides that the application will be grid based. With the Electron Grid of Aligned Documents developers are not limited to creating Integrated Development Environments or code editing tools like is possible with Visual Studio Code, they can use the Electron Grid of Aligned Documents to create something new.

The framework itself handles initialization of the grid of panels and provide ways to communicate between panels. These panels can also be populated with prebuilt utilities called widgets. The prebuilt widgets include a webpage viewer, file browser, code editor, development console, and a tab bar. The framework also documents how new widgets can be created by future developers. Development environments such as the Perlenspiel IDE which this project was initially designed for is a good example of what is possible with this framework however, giving developers the ability to design their own widgets makes this framework flexible enough to be

1

used for limitless 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 3 webpages open at the same time, as well as a text input cell, and a post button. When the button was pressed a post could be made to all three social media sites concurrently.

Grid

The most important feature that EGAD is built around is the ability to create and manipulate a grid. The grid is made up of horizontally resizable **columns** each containing child **rows** that can be individually resized vertically. Locations in the grid can be addressed and are referred to as **cells**. For example, the grid presented in figure 1 is comprised of five columns each containing five rows. This means that there are 25 total cells in the grid. The 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 null is returned. Similarly the grid.setWidget(column, row, widget) call can be used to override the widget residing in a current cell of the grid. 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 inconsistent row sizes. Figure 2 depicts a grid with one cell in column one, and three cells in column two. Structuring rows in this free-form manor enables many more application designs, thus increasing the use case flexibility of the EGAD framework.

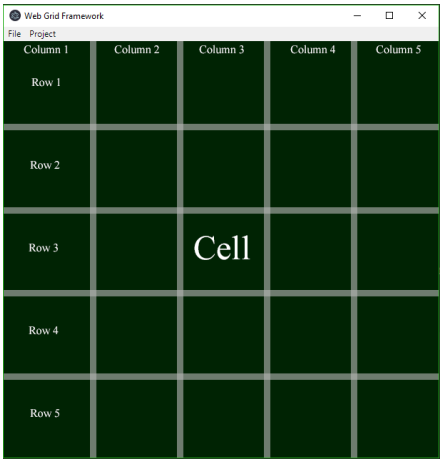


Figure 1|5 x 5 Gird of Cells

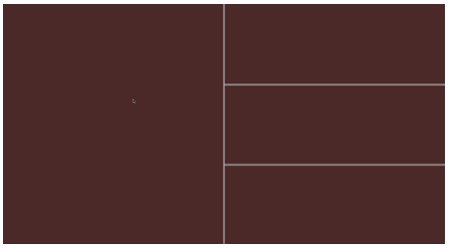


Figure 2|Mismatched Columns

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 are resized dynamically when any of these drags are moved. To move a drag, a user can simply click on the white boundary and drag it to a desired destination. 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.

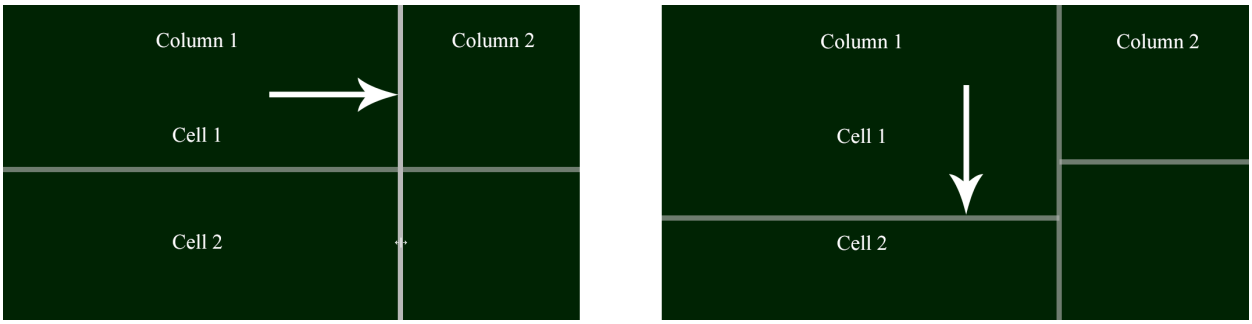


Figure 3|Demonstration of Horizontal and Vertical Resizing

Every cell in the grid can contain a single 'widget'. Widgets are interactable containers for displaying data. When a grid is initialized an array of widgets is passed in. These widgets are synchronously initialized which means that the second widget cannot start to initialize until the first widget is completely done initializing. The parent widget class is abstract; therefore, independent developers can design their own widgets to add functionality needed by their specific application. There are many built in widgets which provide common features that a developer may want to use. This helps developers 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 unique in that they can point to different file locations on the disk, have different

stylistic themes, and can configure properties of their specific file tree without impacting the other widgets.

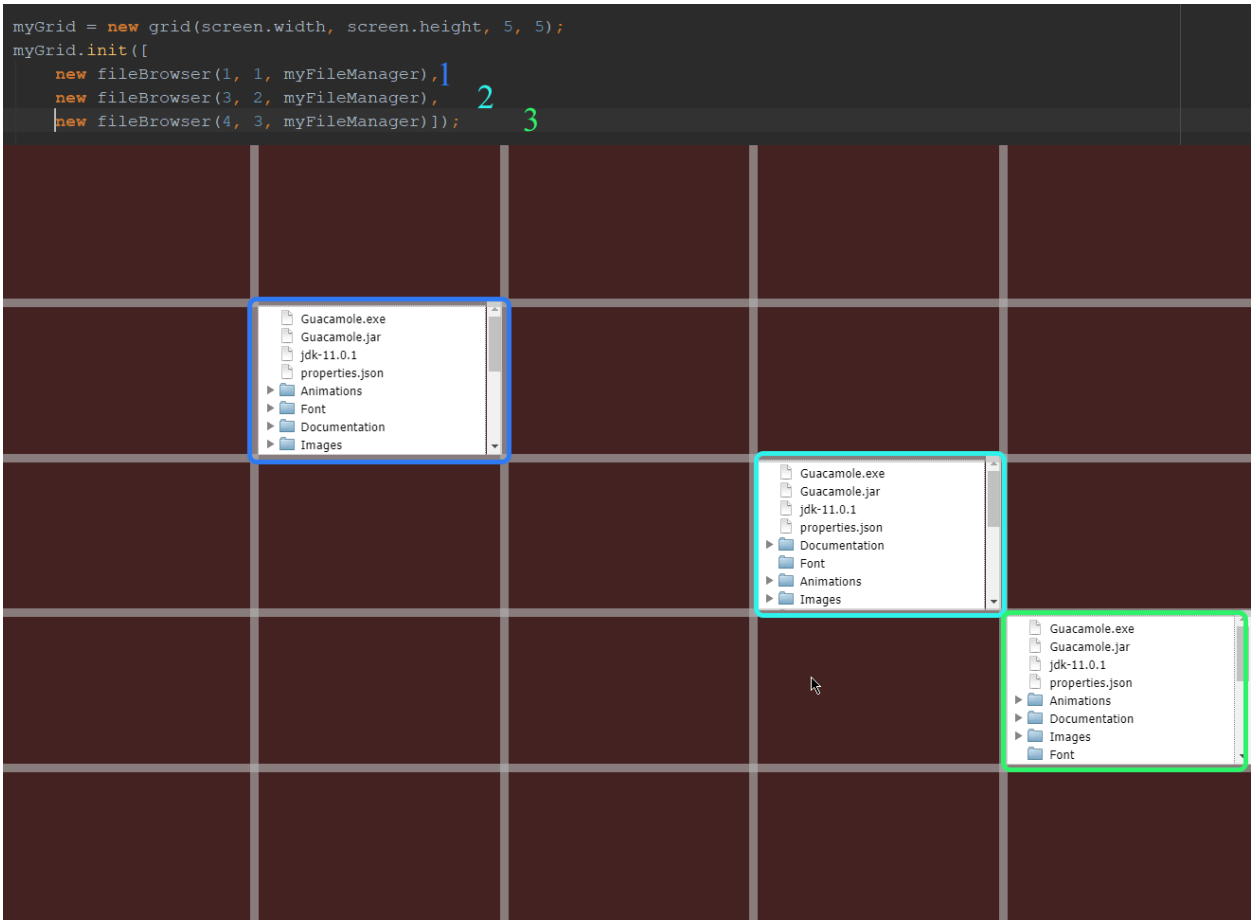


Figure 4 | 5 x 5 Grid with Three File Tree Widgets

The number of cells in a given column is not constant. At any time an application can modify how many cells there are in a column by calling the grid.addWidget() function. Widgets can also be removed from the grid in a similar fashion, by simply calling grid.removeWidget(). This is useful because it allows the entire layout of an application to be switched out 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 this tooltip is no longer needed the widget can be removed from the grid.

Save System

Whenever an application built with EGAD is closed, a custom save object is generated representing the exact state of the grid at that moment. To achieve this complete 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 read to know the exact size of all cells within the grid. 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 where the file tree is open to. These custom JSON objects are then inserted into the object containing the width and height data. This JSON object is then written to a configuration file which the developer specifies. When the application is 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 which application was closed. This achieves complete preservation of the grid state.

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 NodeJS's default file system module 'fs'[2] and enables a developer to easily read and write files relative to where the application. This functionality helps developers to quickly and efficiently manage external files and integrate them with their application.

[2] https://code-maven.com/reading-a-file-with-nodejs

Widget

Widgets are the fundamental display container used by the Electron Grid of Aligned Documents. The parent widget class is an abstract class which defines many useful helper functions and data management tools for widgets. This parent class also defines many fields that every widget needs to have. Widgets must contain a name describing the widget 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 hold a list of references to other widgets that they may need to take data from. 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 and when the grid is saved this object is written to. 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 and all fields on the widget should be defined making the widget 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 inside of *Widget A*'s constructor it references fields on *Widget B*, if *Widget B* has not finished initializing many of its fields will be undefined. One would think that this solution could be solved by simply assuring that *Widget B* will have been initialized by the widget A's constructor is called. It is not that

simple however; Assume that we create *Widget B* immediately before *Widget A* as is seen in figure 5.

```
let widgetB = new Widget({name:"File Tree", col:0, row:0}, {});
let widgetA = new Widget({name:"File Tab Manager", col:1, row:1}, {widgetB});
```

Figure 5| Illustration 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 it is still initializing therefore some of its fields may be currently undefined. Since its constructor has terminated the next line of the program would execute to create *Widget A*. When *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 undefined 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 illustrated in figure 6.

```
509     //Synchronous loop that populates a cell with a widget.
510     async initalize(widgets, COLUMNS) {
511         for(let i = 0; i < widgets.length; i++){
512             let widget = widgets[i];
513             let result = await this.initializeWidgit(widget);
514             console.log("widgetsName:",result," Element:",result.getElement());
515             COLUMNS[result.colIndex].ROWS[result.rowIndex].childNodes[0].appendChild(result.element);
516         }
517         console.log("Initialized.");
518     }
519
```

Figure 6| Depiction of Synchronous Loop Structure

This loop uses a new feature of JavaScript ES6, "async await"[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 the second keyword "await". The keyword "await" must preface an asynchronous function call. When the code in figure 6 on 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 any application to be include and used in any 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 find one that works. Or edit an existing widget to fit their needs. This open source project will allow users to develop applications quickly through open source widgets developed by other community members.

3

The Electron Grid of Aligned Documents has a small set of pre-defined widgets which help developers implement common functionality quickly without needing to develop their own widgets. These built in widgets were developed to show how flexible widgets can be and provide developers with tools that are commonly desired in applications. The included widgets are a webpage viewer, a file browser, a code editor, a tab bar, and a console. The widgets contained in this subset 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 to show 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 saves its 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.

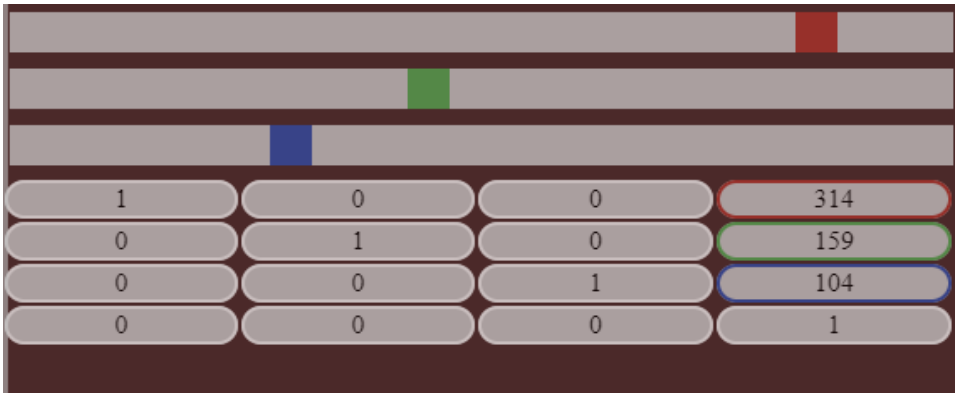


Figure 7|Transform Widget

Webview

The goal behind the Webview widget is to wrap the Electron Webview[4] DOM element within an EGAD widget. Webviews act as more powerful Iframes[5] in that they allow a developer to tap into the output stream of a webpage/ Webviews also provide access to exact V8[6] instance interpreting 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 webpage's output. Other calls exist to run JavaScript within the Webview. This provides developers with immense power and control over the webpages 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 out on the internet or 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 out on the internet.

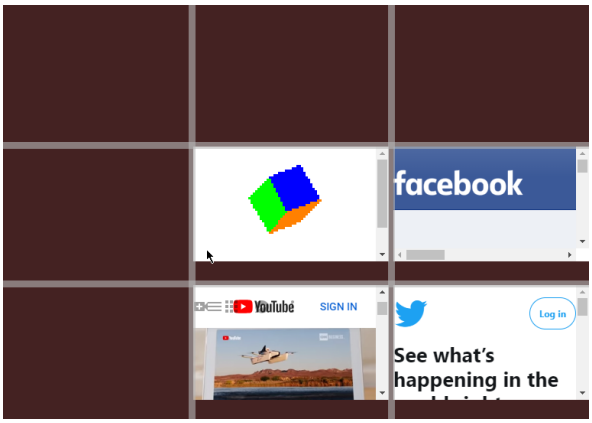


Figure 8| 3 x 3 Gird of Webviews

[4] https://electronjs.org/docs/api/webview-tag

[5]

[6]

File Tree

One feature which many projects rely on is the ability to traverse a file structure. To implement this inside EGAD the FancyTree[7] 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 NodeJS file system module was used. This module allows a computers disk to be looked through within JavaScript. This functionality was abstracted within EGAD to the **getProjectFiles()** which allows a user to get all files and subdirectories of a specific directory. This method also utilizes an ignore object to omit certain files from the JSON object returned. Much like a .gitignore[8], 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 that file can be excluded by simply typing its name "index.html" inside the ignore object. If all files of a specific name with different file extensions should be ignored, the wildcard character could be used again. 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 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 file 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.

[7] https://github.com/mar10/fancytree/wiki
[8] https://git-scm.com/docs/gitignore

Figure 9| Depiction of FileTree Widget

In future versions of EGAD this module will be improved to support remote file server access through the FTP[9] protocell, however this is out of scope of the first version of the project. This would enable developers to create far more complex applications. Connecting to a remote server may be out of scope for this version of EGAD but that does not mean that it cannot be implemented in the framework currently. An independent developer would be able to create their own FTP widget which could connect to a database and display the resulting files in a tree structure.

[9] https://tools.ietf.org/html/rfc959

Code Editor

The Code Editor widget is a language agnostic editor which allows a user to edit an arbitrary 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[10] 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 called "languageMap" which associates file extensions with languages. This widget has configurable hotkeys for saving the code being edited to a file as well as, copying, pasting, and commenting code. The widget provides a way to register function callbacks to be executed whenever one of these hotkeys 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

10

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 Language 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 name out.

A real-world use case for this IntelliSense 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.

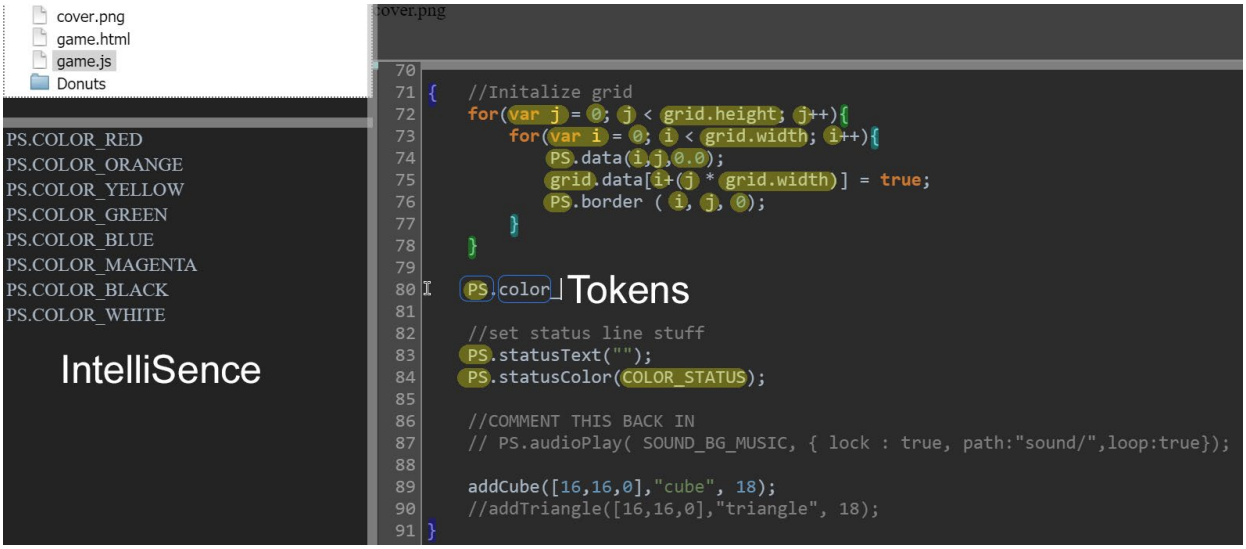


Figure 10| Depiction of Code Editor Widget

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, this print 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 depicts a command being sent to the stdin 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.

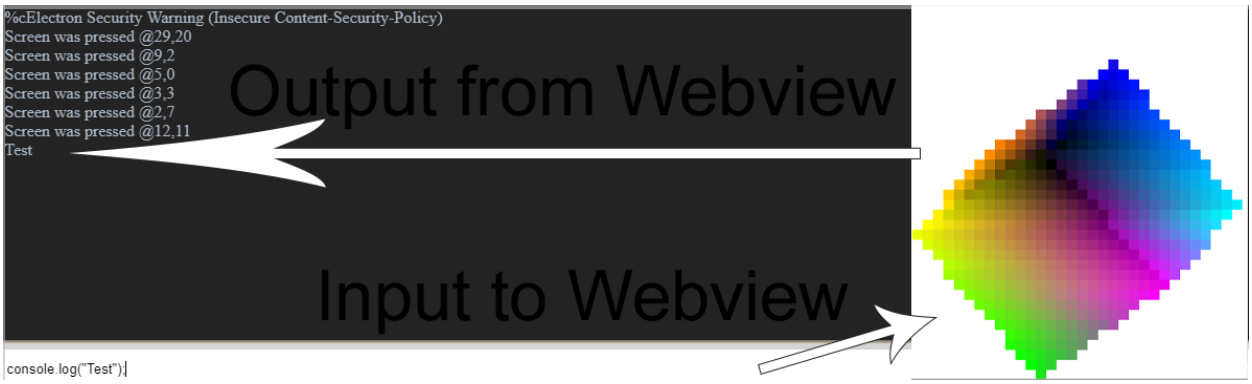


Figure 11 | Illustration of Stream Flow

Developers are also able to link Development Console widgets to processes spawned by the Electron Grid of Aligned Documents. 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 exists to allow 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. The Tab Bar Widget then 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 in 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.

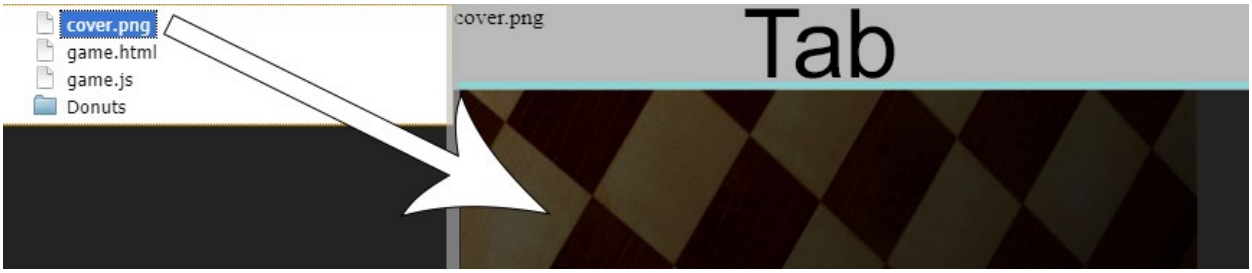


Figure 12|Callback Function on Tab Bar Widget

Canvas

The Canvas widget was designed to provide an easy way for developers to integrate with webGL[11]. WebGL is a web implementation of the Open Graphics Library[12] (openGL) 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 passed 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, the fastest the canvas can be refreshed is 250 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 60fps, there is lots of leeway to add additional

11

12

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.

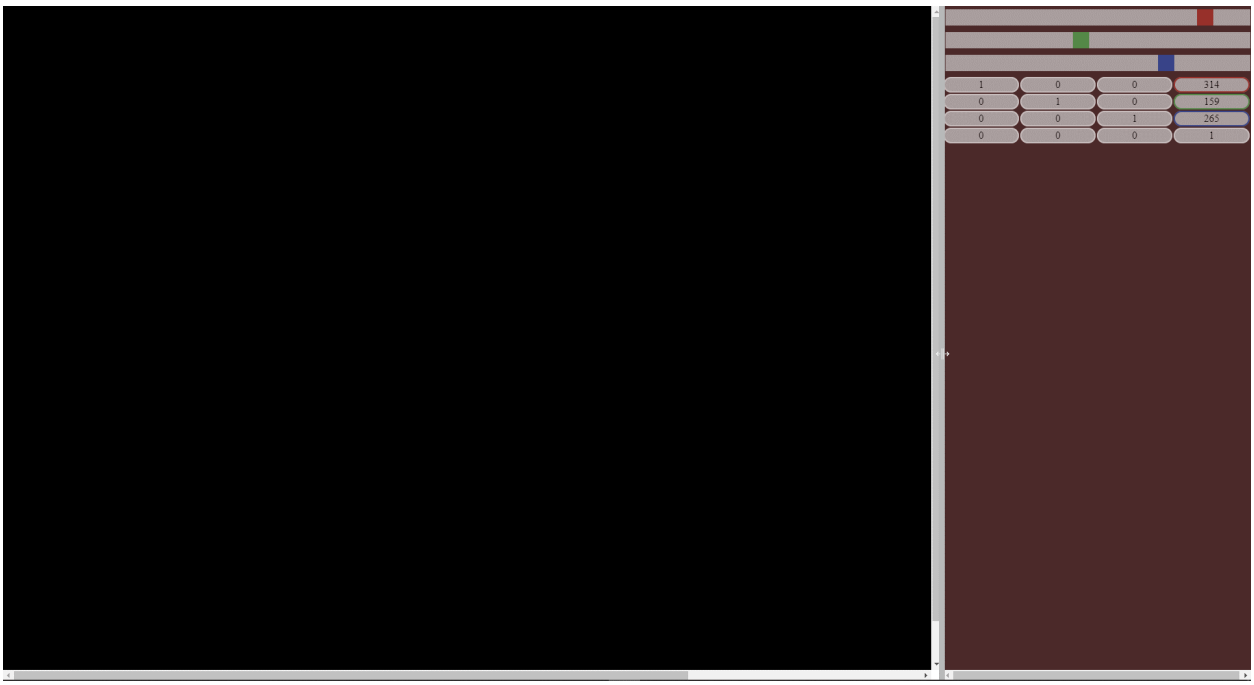


Figure 13 | Blank WebGL Canvas

Figure 12 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 three.js[13] 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.

[13] https://threejs.org

Process Spawner

One of the most powerful abilities of NodeJS[14] is the 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 process spawner 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 13 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

14

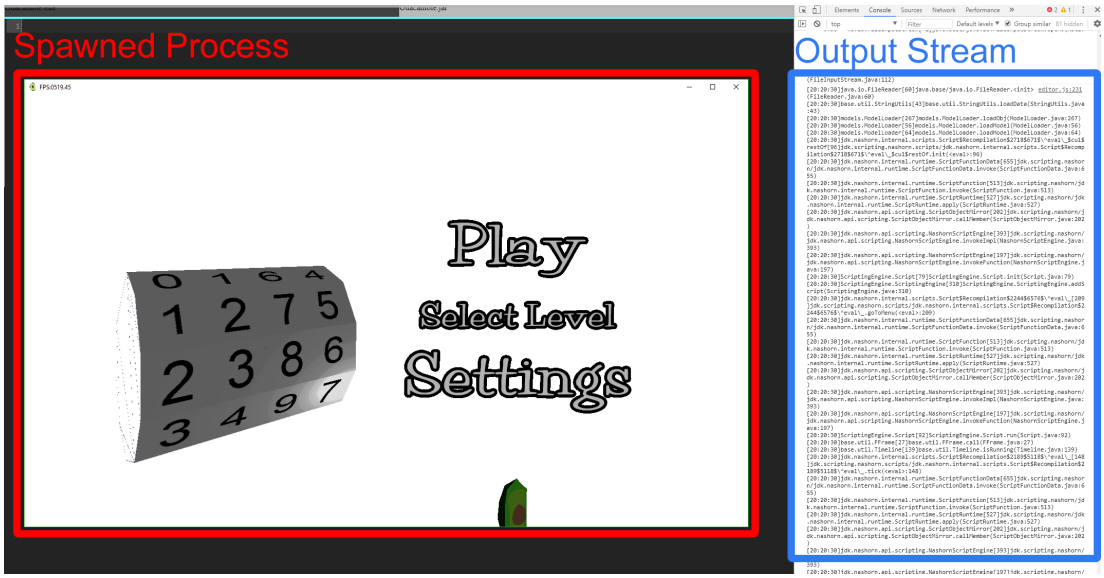
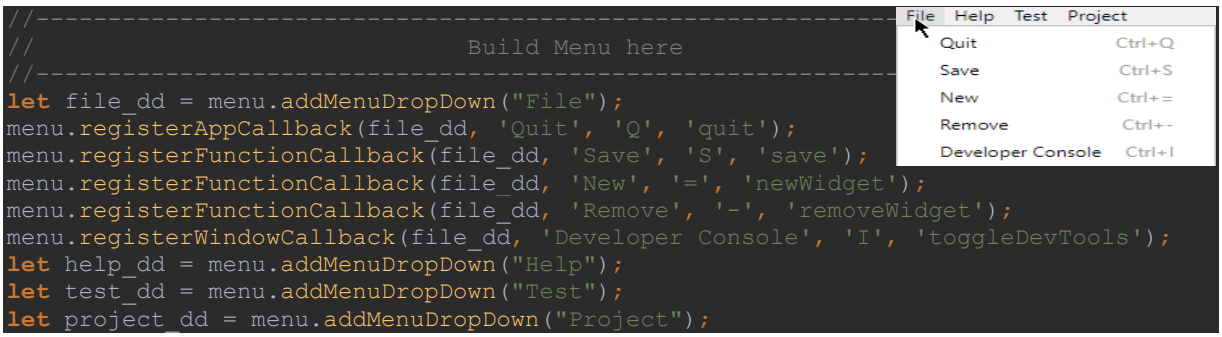


Figure 14| Depiction of Spawned Process Output Stream

One possible application for the process spawner is to compile and then run code written inside a Code Editor Widget. Another possible application for the Process Spawner 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 the Electron Grid of Aligned Documents 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 use the 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 14 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 15 | Code Needed to Build the 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 the digital game design courses[15]. In these courses, students use JetBrains, WebStorm[16] application 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.

The Electron Grid of Aligned Documents is an excellent choice for developing the application described above. The editor itself would be comprised of a grid utilizing many of the built-in widgets of the Electron Grid of Aligned Documents. 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.

15

16

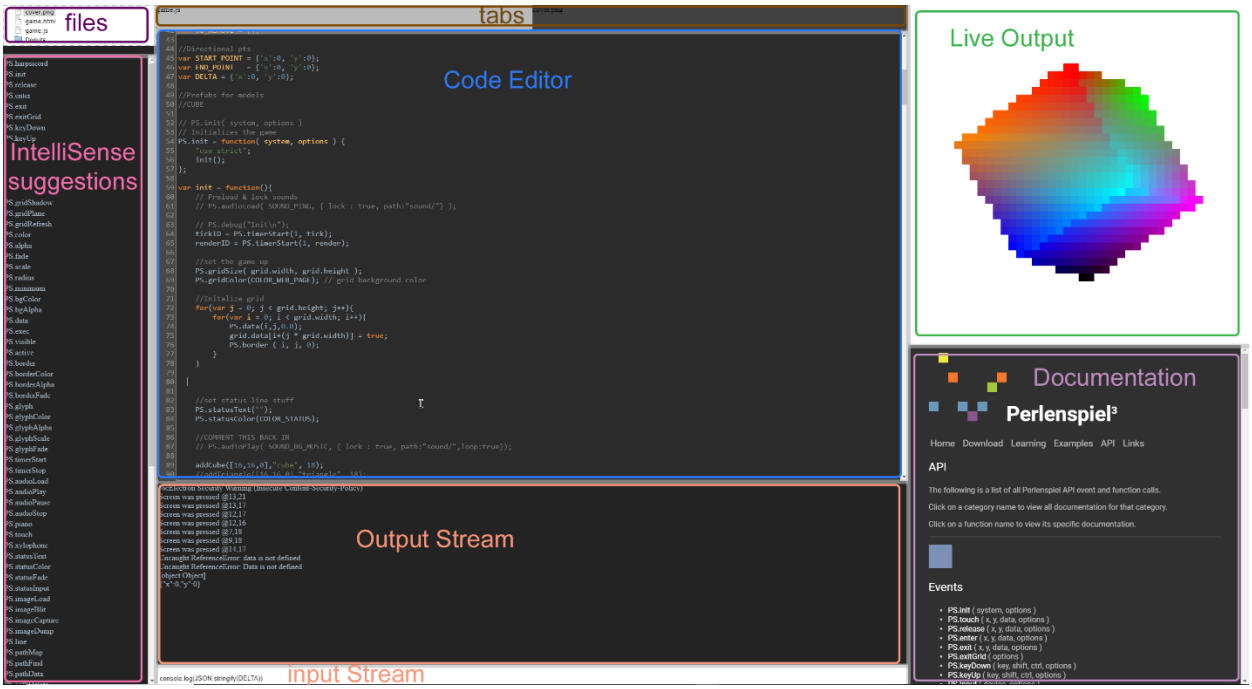


Figure 16| Perlenspiel Integrated Development Environment

Figure 15 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.

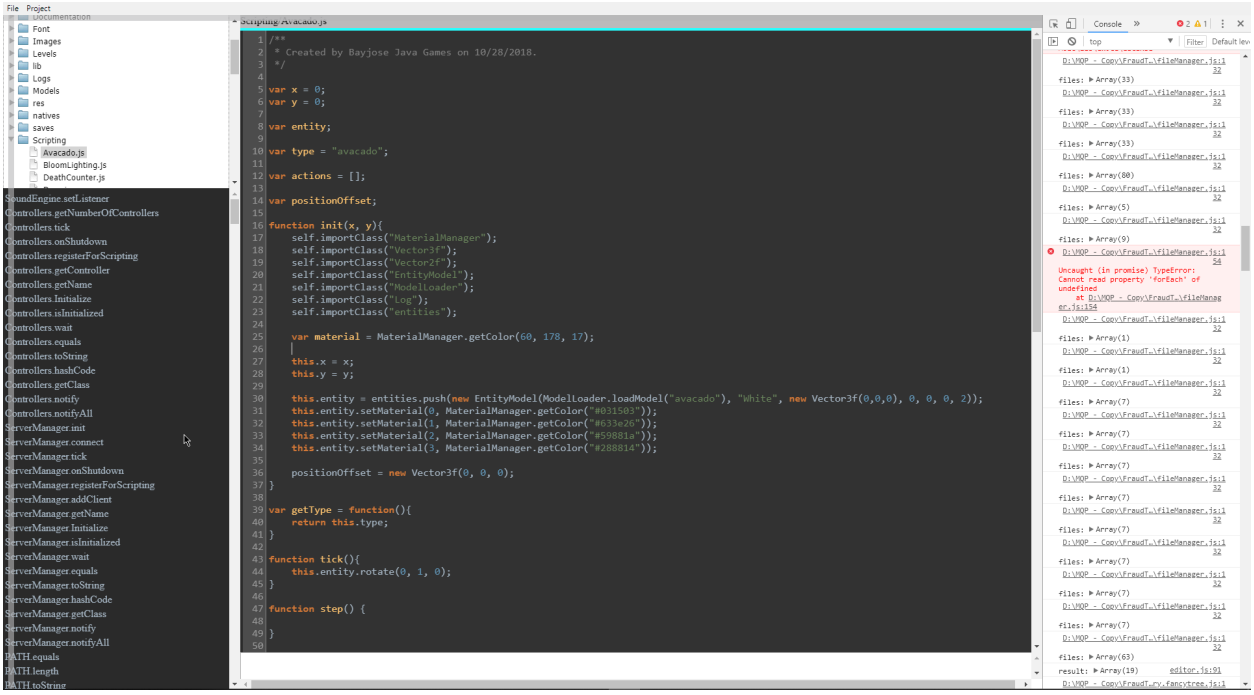


Figure 17 | FraudTek Editor

Figure 16 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.

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, as well as 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. If a developer chooses to, they could easily develop new Widgets for EGAD and submit them as a pull request to the repository. This project is open source to facilitate adoption and encourage adaptation.

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

Works cited

-IRB
-JavaDoc

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

Code Mirror
https://codemirror.net

Light Table
http://lighttable.com

Atom Editor
https://github.com/atom

vsCode
https://code.visualstudio.com

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 awit:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
https://www.ecma-international.org/ecma-262/8.0/#sec-async-function-definitions

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

Grid
Widget
Webview
File Tree
Code Editor
Console
Tab bar
Canvas
Menu
Process Spawner

Source Code