Investigation into GUI Support for a Web-Based Game Engine

Dexter Hu

A project paper

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington Bothell

2016

Reading Committee:

Kelvin Sung, Chair

Kelvin Sung

Jason Pace

Mark Kochanski

Program Authorized to Offer Degree:

Computer Science and Software Engineering

**Abstract**

Investigation into GUI Support for a Web-Based Game Engine

Dexter Hu

Chair of the Supervisory Committee:  
Dr. Kelvin Sung  
Computing and Software Systems

A game engine is an Application Programming Interface (API) designed to support game development. Typically, advanced programming skills and knowledge of the back-end API structure are required to build games with these. A well-designed graphical user interface (GUI) to a game engine would define an interactive editor for the game developers, remove the need for in-depth API knowledge, and allow developers to build games easier than with the game engine alone. In a web-based environment, such an editor would allow developers to build and play games from anywhere on the internet. This project, based on the existing web-based GTCS game engine, investigates the technical requirements and a number of other factors affecting usability of a GUI frontend, but does not propose a final GUI design implementation. Our study analyzed GUI modules of existing, popular game engine editors, identified web-based technologies suitable for building the modules, developed prototypes, and integrated the prototypes into a functional web-based GUI frontend for the GTCS game engine. Our results demonstrated feasibility and are the pathway for building a user-friendly game editor of a web-based game engine.

**Table of Contents**

List of Figures

Chapter 1. Introduction

Chapter 2. Literature Review

2.1 Purpose

2.2 Summary of Literatures

2.3 Design Guidelines from Literatures

Chapter 3. Qualities, Architecture, and Design

3.1 Extent of Qualities

3.2 Architecture

3.3 Lifecycle and Processes

3.4 Implementation

3.5 Game Build

3.6 Rationale for Major Design Decisions

3.7 Supporting Software

Chapter 4. Results

4.1 Extent and Metrics of Success

4.2 Implementation Issues

Chapter 5. Conclusion

Bibliography

**List of Figures**

Figure 3.1 Architecture diagram (MVC).

Figure 3.2 View of the “sandbox” application, containing a few widgets and panels.

Figure 3.3 Two paper prototype snapshots.

Figure 3.4 View of the editor. It contains a central GL canvas and three sections of panels, located in the left, right, and bottom. The left (A) and bottom panels (B) display (and allow the creation of) items required for a game. The right panel (C) is automatically updated with (and allows the editing of) the details of the selected item. Finally, run, file open, and file save can be performed from the top menu bar (D).

Figure 3.5. Employing reading gravity.

Figure 3.6. Reducing cognitive load and grouping related items. The arrows point to the sections of tabbed interfaces.

Figure 3.7. Using a single automatically-updating panel to simplify the editor.

Figure 3.8. Alternative instantiation method, devised to reduce work.

Figure 3.9. Familiar terminology, consistent with many other applications.

Figure 3.10. Managing attention and avoiding distraction/confusion.

Figure 3.11. Appealing to visual fields. The supporting images are on the left (red) and the text is on the right (blue).

Figure 3.12. Another example of appealing to visual fields. Again, the supporting images are on the left (red) and the text is on the right (blue).

Figure 3.13. Three scenes of a simple breakout game, created using the editor.

Figure 3.14. Keyboard input.

Figure 3.15. Object instantiation.

Figure 3.16. Autonomous behaviors, object behavior switching, and collision detection.

Figure 3.17. Use of multiple cameras.

Figure 3.18. Use of multiple scenes. Scene can be changed based on an in-game condition.

# 

# 

# 

# Introduction

This project explores the technical and usability requirements for the development of an initial graphical user interface (GUI) for a given web-based game engine and the building of simple games with the resulting system (a game editor). Functionally and architecturally, a web-based game engine with a GUI frontend differs from applications like Unity3D or Unreal Engine,[[1]](#footnote-0) which are designed for the desktop. Web applications, in contrast, are highly accessible with just the internet, and do not require a download.

The major stakeholders of this project are game developers and game engine developers. Our editor is designed to reduce the technical understanding needed to create games (when compared to using just the backend game engine), simplifying the game development process. A simpler development process would let game developers focus less on technical aspects and more on creative development. To represent game engine developers, we investigate and demonstrate the feasibility of an editor that is accessible anywhere via the internet.

This project investigates the support for creativity in game development and is part of Professor Sung’s Game-Themed Research Group’s investigations. We have implemented the initial web-based GUI frontend (forming an editor), which will guide and simplify the game development process for its particular game engine and allow games to be created more efficiently.

Key terms and concepts used are: API, GUI, functionality, and usability.

# Literature Review

## Purpose

The purpose of this literature review is to gain an understanding in the design of applications with intensive interactions. This is needed in order to guide the appearance of the initial GUI and the placement of its components, in order to make the overall editor more user-friendly.

In Section 2.2, we have surveyed and summarized relevant literatures, including studies and experiments. We have also made observations based the designs of Unity3D and Unreal Engine. Through this, we have derived and compiled a list of design guidelines in Section 2.3.

## Summary of Literatures

George J. Pangalos’s “Designing the User Interface” is a classical survey article on the usage of consistency and simplicity [5]. In his report, he first discusses consistency, separating it to three levels: semantic, syntactic, and physical.

Semantic consistency refers to the extent that a user interface (UI) behaves based on its perceived understanding. Syntactic consistency refers to using the same terms across UIs, allowing users to develop early understanding of the functions. Finally, physical consistency refers to the ability of repeating a process to reproduce a result.

In terms of simplicity, Pangalos urges for the ability to directly manipulate items in the interface, so little user training is needed. One example is with the display of the currently manipulated object. He suggests that such a display should continuously update itself, so only one display is needed for manipulating objects. In his conclusion, Pangalos acknowledges a rapid growth of technologies and that UIs may change greatly. As such, the design principles he has discovered and reported were not reviewed in great detail.

Understanding the types of metrics used in usability experiments of similar GUIs can guide the design of our GUI. In “Intelligent Graphical User Interface Design Utilizing Multiple Fuzzy Agents,” Agah and Tanie explore the use of fuzzy agents in a GUI application to assist users in completing simple tasks [1]. The authors then conducted an experiment to determine how efficiently users could complete the tasks via the following metrics, all of which adhere to the major design guidelines from their research: running time, number of mouse moves, and number of operations. They stated that reducing the values of these metrics would reduce the amount of work a user would need to do. However, the authors also identify a scalability concern, concluding that more metrics are needed to evaluate more complex systems, as users would be completing more complex tasks.

Aside from reducing the amount of work a user needs to do, reducing the time it takes a user to process information is also important. Patricia A. Chalmers’s research in “The Role of Cognitive Theory in Human-Computer Interface” searches to extend cognitive theories from traditional learning to computer-assisted learning [2]. In terms of screen design, she argues that a well-designed screen allows for “automatic processing” in users and removes the need of an auxiliary manual. To achieve this, Chalmers suggests that developers pay attention to layout, consistency, color, and spatial display. In terms of layout, it is important to consider the placement and grouping (e.g. vertical, branching trees, etc.) of objects. Consistency is a vaguer concept, which some find to have no impact on design, while others suggest being consistent in grouping together similar information. The next topic, color, relates to balancing interest and distraction to attract or sway a user’s eyes. Chalmers recommends using familiar colors, as well as single colors (to prevent distraction), and advocates against using colors that might cause issues for users with color-blindness (such as red and green). Finally, when describing spatial display, the author raises concept of “unity”: objects that are related should be in close proximity, relative to other objects, as well as the edges of their container. Chalmers ends stating that these theories still need validation and should be treated only as suggestions for design.

Hillstrom and Chai in “Factors that Guide or Disrupt Attentive Visual Processing” further discuss means of managing a user’s attention, by reporting their research findings [3]. The authors most notably suggest that in order to attract a user’s attention, an object needs to be distinct and expectable (in terms of location and existence), which follows the concept of schema matching. Hillstrom and Chai conclude that this should only be applied only to traditional single-user applications, rather than groupware or future applications (relative to 2006).

In “SOCIALSENSE: Graphical User Interface Design Considerations for Social Network Experiment Software,” Stupak et al. conduct an experiment based on a social network, but more relevantly, research factors that make GUIs usable [4]. Their research confirms the previous schema-matching concept, indicating that it causes users to report very positive experiences and thus, should be used in implementation. Additionally, Stupak et al. articulate the importance of employing reading gravity, minimizing cognitive load, and using colors selectively to attract attention. In the Gutenberg diagram for reading patterns, they find that Western readers have a tendency to focus on the top-left corner (moving to the bottom-right), suggesting selective placement of information. However, this will be actually be ineffective if the user is subject to high cognitive load, or when their working memory is overwhelmed. But, by minimizing the “disorientation” of users, or the time it takes to complete tasks, this can be overcome. Many factors, found through experimentation, can minimize disorientation, such as using text-based interfaces (with icons, if well-known) or tabbed interfaces when there are too many windows. Finally, Stupak et al. indicates that color selection could be used to attract attention, noting that a limited color palette reduces distraction. The authors conclude that implementing these four principles caused an increase in user-friendliness all-around, increasing the validity of their procedure and thus, the validity of previous literatures with similar findings.

Yee et al. dive deeper into the topic of memory in “GUI Design Based on Cognitive Psychology: Theoretical, Empirical, and Practical Approaches,” and conduct experiments to prove the legitimacy of various related theories [6]. In one experiment, they confirmed the Left-to-Right theory, finding that two-thirds of the subjects prefer images in their left visual field and text in their right, as the theory states. However, another experiment contradicted this: when data was placed in various parts of a screen, a majority of subjects paid attention to the data in the center. The experimenters also performed various other tests, finding a strong preference (93.33% agreement) that subjects preferred backgrounds to be soft-colored and text to be black-colored, both chosen over bright-colored. These results make sense: the authors explain that bright colors cause pupil contraction, causing discomfort and muscular tiredness. In conclusion of the experiments, Yee et al. make a declaration that acknowledging cognitive psychology is a requirement for a user-friendly GUI. The authors urge for the simplicity of GUI designs, the use of familiar icons, and the careful selection of color.

## Design Guidelines from Literatures

**Applying Reading Gravity.** The concept of reading gravity is one such guideline, based off the Left-to-Right theory and the tendency for Western readers to focus their eyes on the top-left corner [4], [6]. However, as Yee et al. indicated, this is not always the case: given an screen containing data, users are more likely to focus their eyes on the center [6].

**Reducing Cognitive Load.** Excess information may overwhelm the user’s working memory (i.e. high cognitive load). Stupak et al. offers several ideas to minimize this effect: in a GUI with many windows, excess information can be hidden by using a tabbed interface. This may also allow a larger font size to be used, further reducing cognitive load [3], [4].

**Maintaining Simplicity.** A simpler GUI can reduce cognitive load and the amount of training needed to operate it [5], [6]. One way to build for simplicity is by using a single, self-updating panel: it would lessen the amount of information displayed at once [5].[[2]](#footnote-1)

**Grouping and Listing Related Items.** It is also beneficial to store related items into lists. In Unity3D, this is done by type (such object, camera, and texture). Menus should also categorize items by meaning: this may allow a user to find what they need faster, due to schema-matching [2], [3]. For example, in a file menu, “open” and “save” are commonly placed next to each other, since they both relate to file input and output (IO): should a user desire a file IO operation, they will benefit from seeing the available file IO functions next to each other.

**Using Consistent Naming.** Consistency is also important, especially across applications. One notable aspect is through naming: using the same operation names as other applications fosters early understanding of the operations [2].

**Reducing Work if Applicable.** In terms of metrics, it is valuable to design GUIs to reduce the running time, number of mouse clicks, and amount of mouse movement. This, however, is only applicable to simple tasks. Tasks where users may be required to come up with their own solutions cannot possibly be measured in terms of these metrics alone [1].

**Appealing to Visual Fields.** The careful placement of images and selection of color is highly advocated. In accordance to the visual fields, images should be places on the left and text on the right; matching this reduces cognitive load [6]. This can be applied to icons as well. Stupak et al. found out that text is usually preferred over icons, unless the icon is well-known [4]. However, using both is helpful as well.[[3]](#footnote-2) When accompanying text with an icon, it may also be beneficial to place the icon on the left side of the text, to match the visual fields.

**Managing Attention.** Color can be used as an effective tool to attract a user’s attention. Yee et al. determined that softer-colored features were more preferred in attracting attention; bright or contrasting colors caused pupil contraction, and therefore, user discomfort [6]. This was strongly true for both background colors and distinct, attention-grabbing features [3], [6].

**Avoiding Unwanted Distraction or Confusion.** Finally, caution should be taken when using many colors: having too many is distracting [2]. Thus, it may be better to stick with one color, or a few familiar colors [2], [4]. Additionally, it is confusing to use different-colored foregrounds or backgrounds unless the items are different [6]. This is due to semantic meanings that may be associated with the colors, e.g. using “red” for headings [4]. Lastly, it is important to not use color combinations that would negatively impact or confuse color-blind users (such as red and green) [2].

# Architecture and Design

This project focuses on building a prototype GUI for a given web-based game engine and creating simple games through the resulting editor. The project process was separated into two parts. First, we investigated the technologies required to deliver the desired functionalities, by building relevant prototypes. Second, we integrated the technology prototypes into a proof-of-concept system, demonstrating feasibility in building such an editor.

## Extent of Qualities

In this project, the main quality attribute measured is functionality. To meet this quality, a game developer must be able to use our editor to create a simple game. We have determined a list of features necessary for building such games:

* Scenes: The editor should support multiple scenes, as well as the ability to switch scenes. This is a fundamental aspect of games as it allows them to transition to other levels, title/win/lose scenes, etc.
* Cameras: The editor should support multiple, modifiable cameras. This allows games created through the editor to have additional views (e.g. minimaps, split-screens, etc.).
* Objects: The editor should support multiple objects, as well as the ability to instantiate and destroy/remove those objects.
* Input: The editor should support tracking keyboard and mouse input, particularly for objects. This allows games created through the editor to be interactive.
* Autonomous Behaviors: The editor should support the ability for objects to behave on their own. This supports the existence of roaming sentries and various other autonomously-acting objects in games created through the editor.
* Object Behaviors: The editor should support the ability for objects to engage in and conditionally switch between behaviors.
* Collision Detection: The editor should also support box-to-box and circle-to-circle collision detection to allow objects to be collidable. It is a fundamental for objects to be able to switch to a new behavior in the event of a collision.

With the above features, our editor will have sufficient functionality to create a game like Super Breakout. The features in Super Breakout serve as a proof-of-concept for many classical videogames, including: Space Invaders, Pac-Man, and Asteroids.[[4]](#footnote-3) In this way, the support of the listed functionality demonstrates that the system is able to build an entire class of games.

## Architecture

The architecture chosen was Model-View-Controller (MVC) with event-based messaging (see Figure 3.1). In this case, the model is the game that a game developer would be building (stored as lists in GameCore and as global variables), which is updated according to data inputted by the game developer. The view is the GL canvas that displays the state of the model graphically as a scene of the game. The controller is the GUI, which contains components that trigger events based on a game developer’s actions, and, depending on the component, triggers updates to the model.

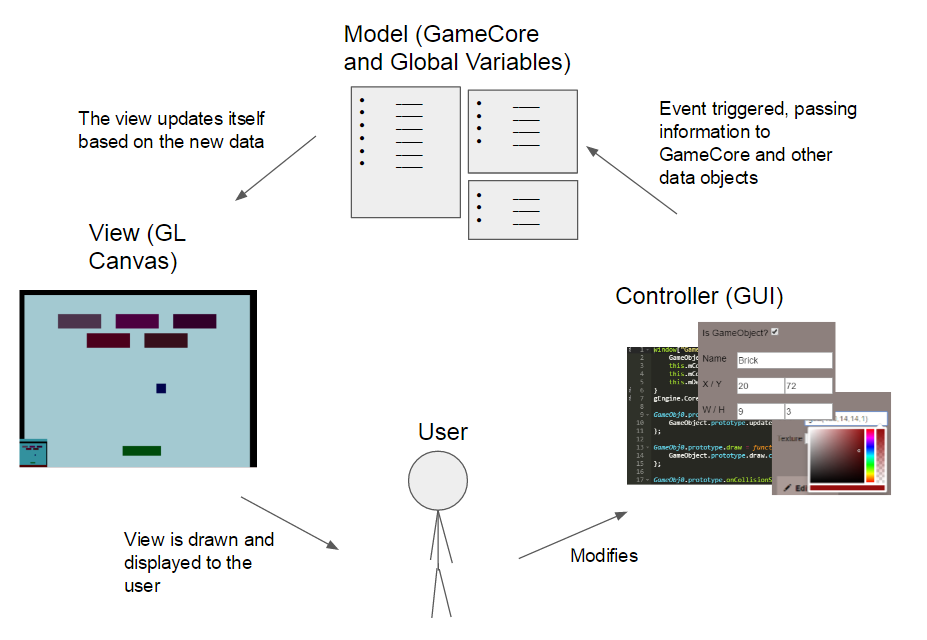


Figure 3.1. Architecture diagram (MVC).

## Lifecycle and Processes

While our original plan was to follow an agile lifecycle, the waterfall lifecycle was adopted instead because the required functionality is similar to many existing game engines and is well-understood. This allowed us for a focused investigation and system development.

The requirement was to develop an initial GUI for a web-based game engine and then create a simple game using the resulting editor. The design stage followed the initial requirements, and began with a technology (e.g. jQuery, CSS, HTML) investigation. All functionalities explored during this time were built into a “sandbox” application (originally made by a student from Professor Sung’s research group; see Figure 3.2) to demonstrate feasibility. Lastly, paper prototypes were created to rehearse the design of the initial GUI (see Figure 3.3).

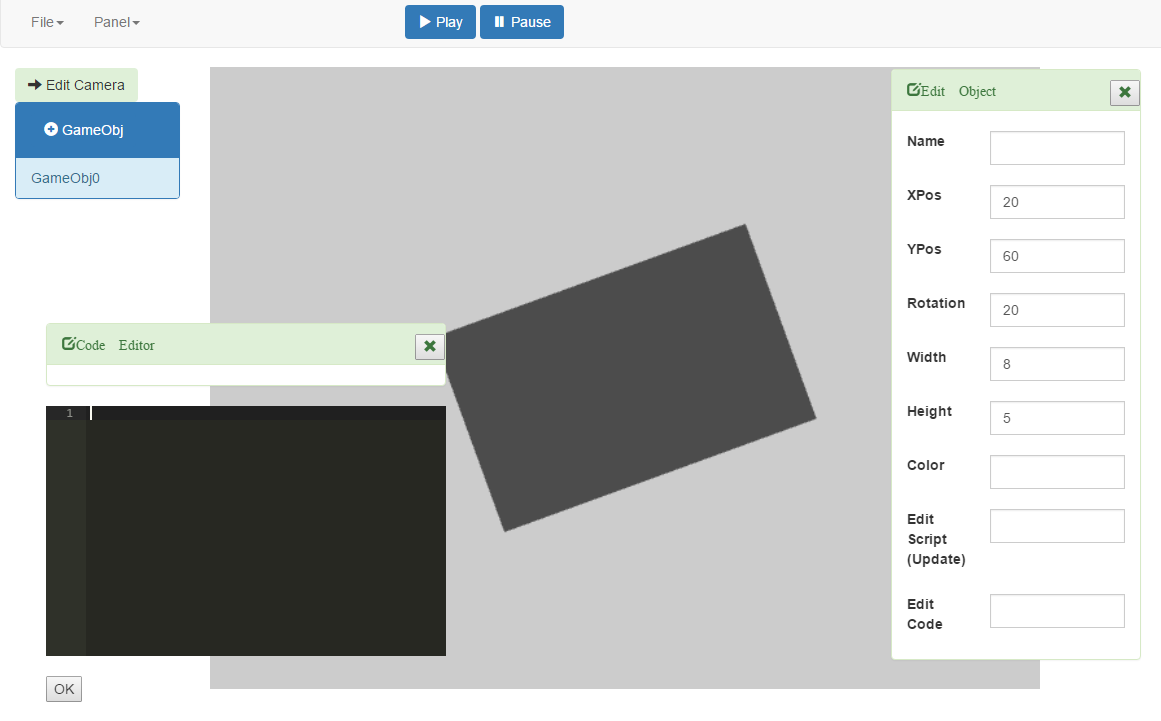


Figure 3.2. View of the “sandbox” application, containing a few widgets and panels.

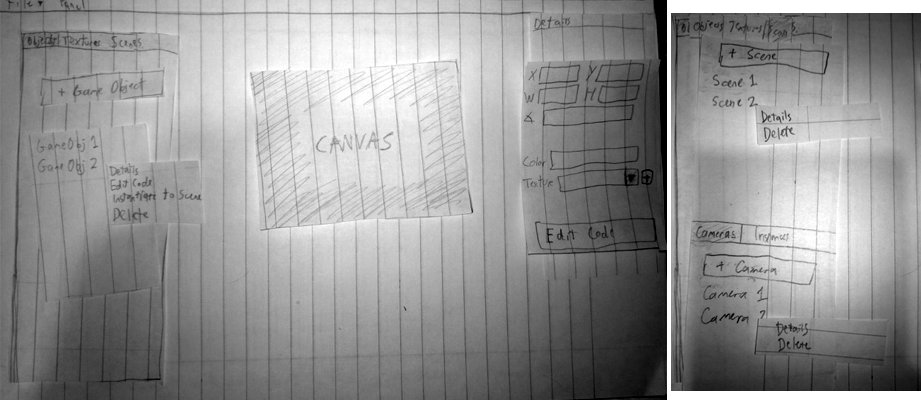


Figure 3.3. Two paper prototype snapshots.

## Implementation

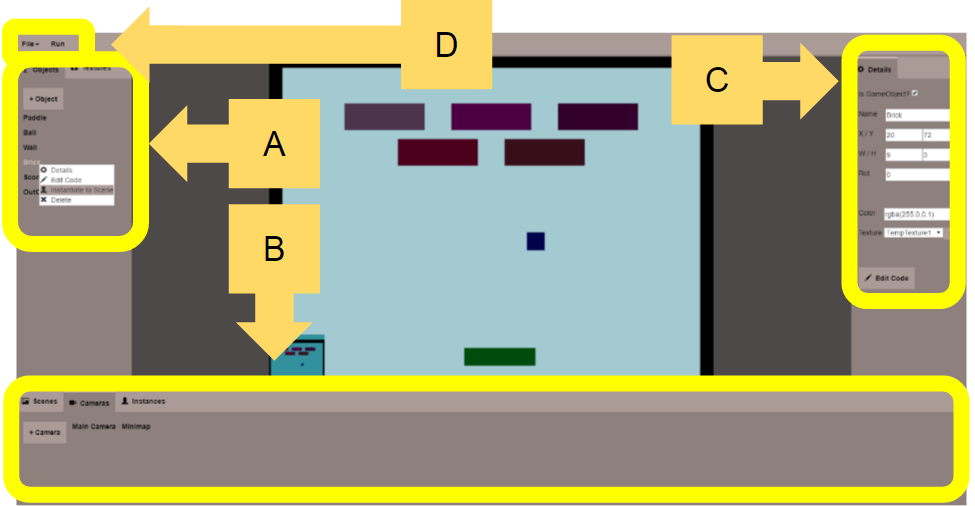


Figure 3.4. View of the editor. It contains a central GL canvas and three sections of panels, located in the left, right, and bottom. The left (A) and bottom panels (B) display (and allow the creation of) items required for a game. The right panel (C) is automatically updated with (and allows the editing of) the details of the selected item. Finally, run, file open, and file save can be performed from the top menu bar (D).

The implementation stage began from the sandbox experimentation results. Development was shifted to the initial GUI (see Figure 3.4), based on the paper prototype in Figure 3.3 and the design guidelines from Section 2.3.

In A of Figure 3.4, tabs (representing lists) are provided for objects and textures. Objects are the classes of the game, and can have their initial appearance and code modified based on the needs of game developers. Textures were not implemented due to time constraints, however.

The panel represented by B contains tabs for scenes, cameras, and instances. Instances, as introduced earlier, are the physical actors in the game, and each instance is based on a previously-defined object. The editor allows these to be added during development or runtime, and modified further, as needed. Cameras are the tools used to draw the game world: they draw a viewport containing all the actors of a scene, and can have their location, size, and background color modified. Scenes are the “levels” of a game, and contain cameras and instances. Additionally, scenes can also be changed at runtime as needed.

The panel represented by C contains the details of the selected item. As the selected item (i.e. object, camera, scene, or instance) may change, the contents of this panel will change as well, in order to make the editable contents applicable to the selected item.

Finally, we address the features of the top menu bar, as represented by D. The file menu allows game developers to start a new game project, export their current game project as a .zip file, and import an existing game project. The run button to the side simply runs the game, temporarily saving the game data before running and re-loading it after, in order to maintain the state of the game from possible alterations that may occur during runtime.

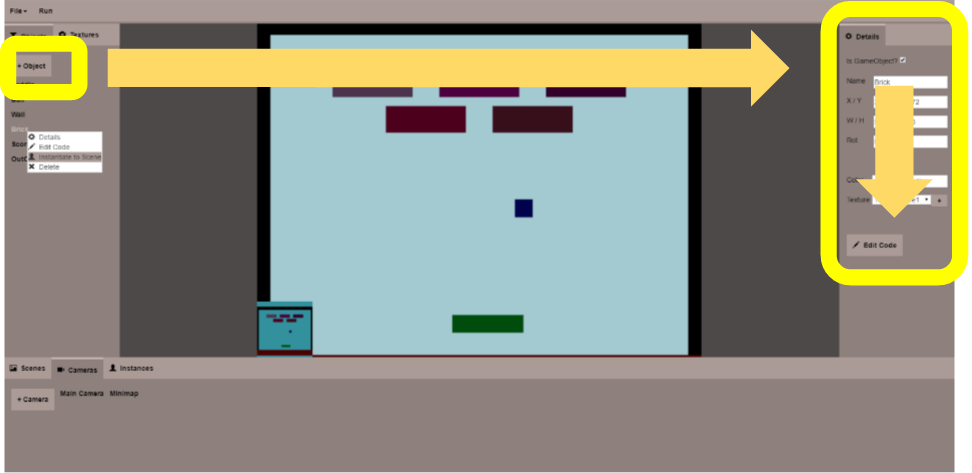


Figure 3.5. Employing reading gravity.

Our initial GUI was also designed based on previously determined design guidelines. The first guideline addresses reading gravity: this is employed in the suggested workflow (see Figure 3.5). When a game developer creates a new game, it will have no point of interest in the center of the screen (i.e. a blank canvas, unlike Figure 3.5, which actually contains a game). Thus, a game developer may direct their attention to the top-left corner instead (left box in Figure 3.5), where they can create the first object. In doing so, the details panel (right box in Figure 3.5) populates. Then, a game developer can fill out the details top-to-bottom, as is comfortable for Western readers.

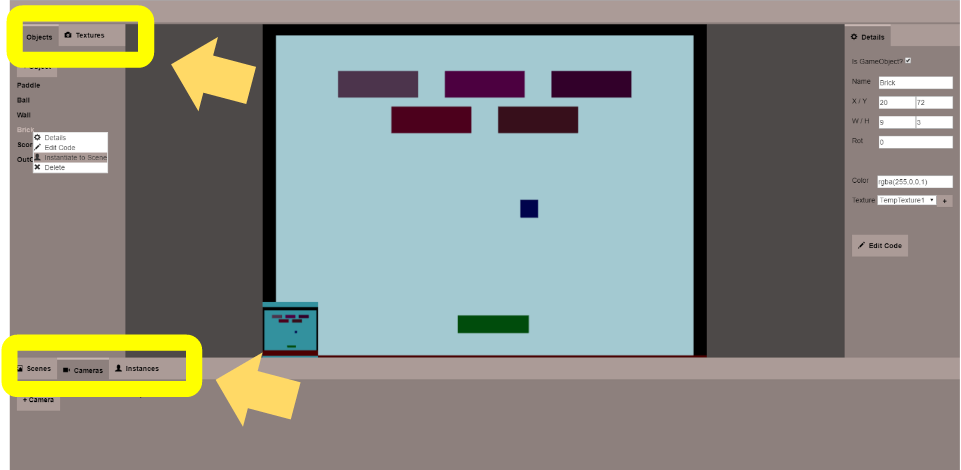


Figure 3.6. Reducing cognitive load and grouping related items. The arrows point to the sections of tabbed interfaces.

In order to reduce cognitive load, excess information is hidden via tabbed panels (see Figure 3.6). The items that fill the screen (e.g. cameras, instances, etc.) are sorted into appropriate lists, with each tab (located near the arrows in Figure 3.6) representing one of the lists. This allows a game developer to easily locate items. Cognitive load is further reduced by the semantic grouping of related panels: all global items are stored in the left (upper arrow in Figure 3.6), and all items specific to scenes are stored in the bottom (lower arrow in Figure 3.6).

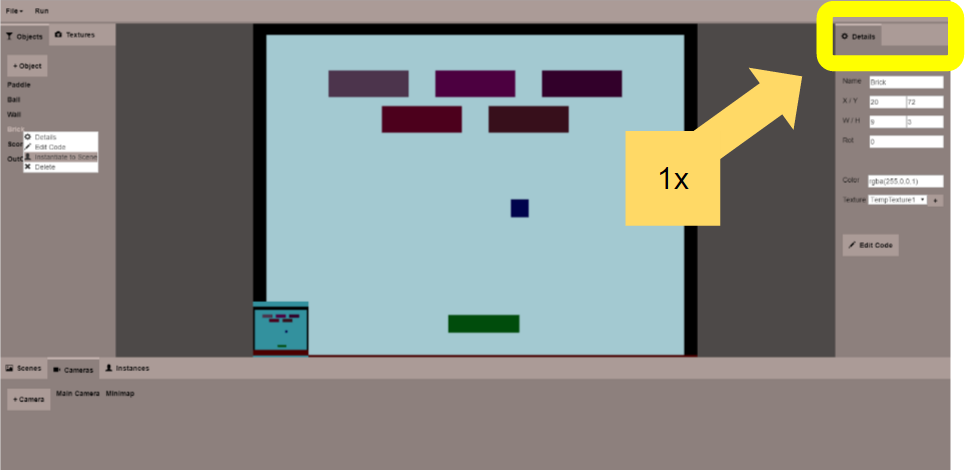


Figure 3.7. Using a single automatically-updating panel to simplify the editor.

The right panel (as indicated in Figure 3.7) is also automatically updated with information on about the currently selected item, removing the need for multiple detail panels. This simplifies the editor, also reducing cognitive load.



Figure 3.8. Alternative instantiation method, devised to reduce work.

Effort is also made to simplify processes, reducing the amount of work a game developer would need to do. When performing instantiation via an object’s right-click menu, a game developer would need to make 2 clicks for each instantiation: a right-click to select the object, followed by a left-click. This sequence becomes repetitive and annoying when a large number of instantiations are required. Thus, an alternate method was devised. A dropdown menu is provided in the instances panel (as opened in Figure 3.8), which takes 2 clicks to navigate, plus 1 click per instantiation thereafter. This makes it more efficient when a game developer needs to perform more than 2 instantiations.



Figure 3.9. Familiar terminology, consistent with many other applications.

Consistent naming was also employed. Notably, the file menu (as opened in Figure 3.9) is easily perceived due to the familiar naming of its options (new, open, and save): those names are consistent with other applications.



Figure 3.10. Managing attention and avoiding distraction/confusion.

Consistency was also considered in the use of colors. To avoid potential confusion, a soft brown was used for all panels, and only for panels. Similarly, a light beige was used for all distinct features (as indicated with the “selected” text in Figure 3.10). These two colors -- brown and beige -- are soft and not highly contrasting, which make the initial GUI more appealing because the effect of pupil contraction on game developers is reduced, when compared to more common color combinations like black and white. Additionally, as these colors are of similar hues, they would not trouble color-blind game developers.

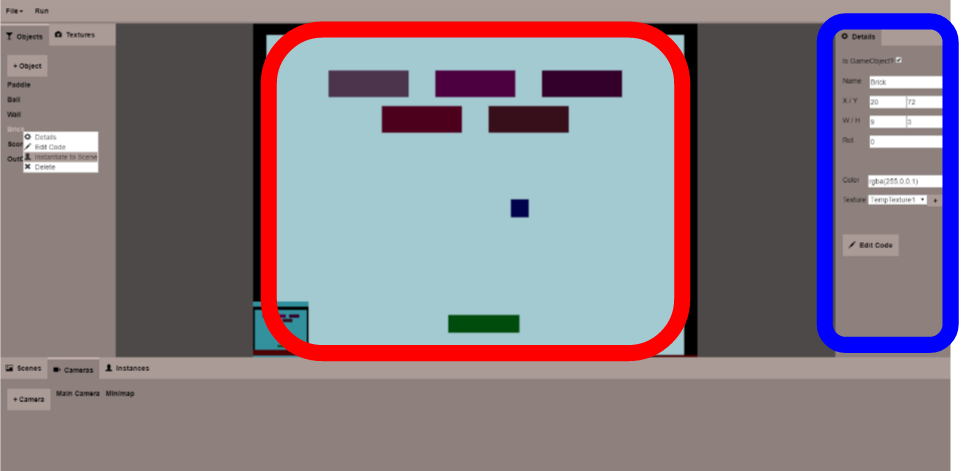


Figure 3.11. Appealing to visual fields. The supporting images are on the left (red) and the text is on the right (blue).

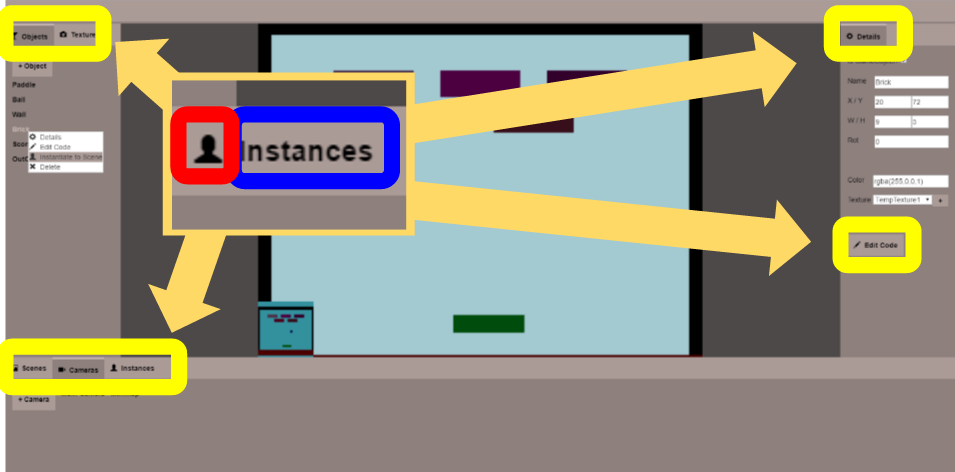


Figure 3.12. Another example of appealing to visual fields. Again, the supporting images are on the left (red) and the text is on the right (blue).

Finally, the concept of visual field-matching is also integrated: labels (as well as the GL canvas itself) are always placed to the left of editable information (see Figure 3.11). The same applies to icons: these are always placed on the left of text (see Figure 3.12).

Figure 3.13. Three scenes of a simple breakout game, created using the editor.

## Game Build

To verify the implemented functionality, we use this editor to create a simple game: Super Breakout (see Figure 3.13). This game contains all of the features outlined previously, in Section 3.1.



Figure 3.14. Keyboard input.

The green rectangle in the far-left scene is a keyboard-controlled paddle (as indicated in Figure 3.14), demonstrating the ability to accept keyboard input. Through this, the paddle can move left and right, allowing interactivity.



Figure 3.15. Object instantiation.

The ability to instantiate objects is also evident in that scene, which evidences this with multiple instances of balls (instantiable during runtime by pressing the space bar), bricks, and rectangles that form the walls. Figure 3.15 indicates instantiated bricks, as an example. Each brick instance originates from the same brick object, but vary in position and color (in this case).



Figure 3.16. Autonomous behaviors, object behavior switching, and collision detection.

Also, the balls in that scene move autonomously and can change behaviors (in this case, moving in a different direction) based on the situation (see Figure 3.16). Here, the catalyst for changing behaviors is in detecting a collision (notably a wall, brick, or paddle).



Figure 3.17. Use of multiple cameras.

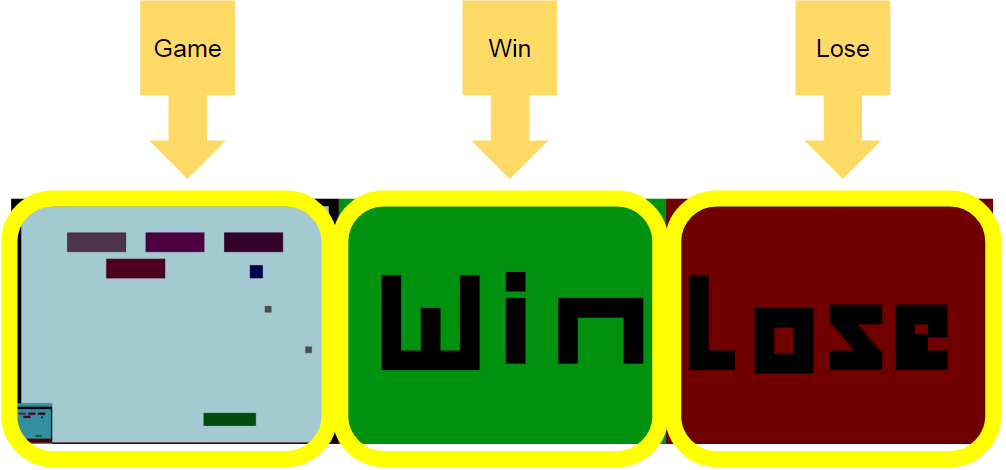


Figure 3.18. Use of multiple scenes. Scene can be changed based on an in-game condition.

Lastly, the game shows support for multiple cameras (as indicated in Figure 3.17), multiple scenes (as indicated in Figure 3.18), and the ability to switch between scenes based on an in-game condition (destroying all the bricks, or the ball falling out of bounds).

Due to the logistically complex issues involved in the dynamic creation of new object types that subclass from an existing superclass, the implementation took longer than initially scheduled. For this reason, we were unable to conduct user testing and refine our initial GUI design.

## Rationale for Major Design Decisions

Since a game developer will need to interact with the editor to create a game, event-based messaging is chosen as the means to send information: this works well because jQuery offers this functionality through HTML components. MVC is chosen as the architectural style because it best supports the requirements and workflow: a game developer will need to see the current scene of their game, so a view is needed. A central storage will be used to hold game data for the entire application, so a model is also needed. Lastly, because messaging is event-based and events are triggered from HTML components, the GUI serves as the controller; thus, MVC is a logical choice of architecture.

The client-server architecture has been considered where the backend game engine could be hosted on a remote server machine. However, this was rejected due to server-sided storage requirements and potential security concerns (e.g. potential of a game developer injecting Javascript code to the server). In a client-server model, game developers would need to create accounts to save their data. This lets them access their data from any computer with the proper login details. However, accessibility is reduced due to the barrier of requiring an account. If game developers were to have accounts based on IP address instead, access is further restricted by the particular computer. As for security, a client-server model risks the injection of malicious code into the server.

## Supporting Software

External libraries and supporting software were used to implement some of the features within the editor:

* jQuery is used to provide functionality to HTML components.
* Cascading Style Sheets are used to stylize those components.
* The Twitter bootstrap colorpicker is used to provide a colorpicker widget so game developers would not need to meticulously define numerical values for colors.
* Ace is used as an embeddable, syntax-highlighting Javascript code editor, giving the code editor panels the appearance of a standard code editor.
* JSZip is used for creating and loading the project’s file system in the form of a .zip folder.
* FileSaver is used for downloading the .zip folder.
* Bootstrap 3 Glyphicons are used to provide symbols to accompany text, improving the understanding of the editor’s various operations.

# Results

## Extent and Metrics of Success

As demonstrated, the playable game created (Super Breakout in Figure 3.5) supports common features present in an entire class of classical videogames. In this way, it serves as a feasibility proof-of-concept, showing that our deliverable -- a prototype editor -- satisfies the functionality requirement for this project and is successful.

There is a caveat, however. In the current implementation, the dynamic creation of objects during the game (e.g. spawning another ball) can be complicated. It requires the understanding of the existing GameObject construction and the overall usage of the Transform object. Additionally, game developers would need to know several utility functions that are built inside the game engine, such as for switching scenes. As is the case for all editors for game development, some knowledge of the existing game engine is required to define customized behaviors.

## Implementation Issues

There were several unexpected issues encountered throughout the investigation process, affecting the progress of the project. The most notable was in determining how to support the ability to create and instantiate objects with customized behaviors. Our initial solution was to use a UI GameObject (UIGO) that contained instances via has-a relationships. However, this solution suffers from inefficiency: when the UIGO is changed, it must update all of its instances manually.

Our eventual solution was to dynamically subclass GameObjects into new data types. Dynamic subclassing allows new data types to be defined and all instances to be created. Instances of a defined data type (related as an is-a relationship) should automatically receive modifications (as they occur), and thus, performance should vastly improve. However, our investigations revealed that any modification to an existing class definition results in the creation of a new data type, where existing instances will still belong to the old, unmodified data type. For this reason, we had to manually update all existing instances of a data type when class definitions are changed. Ultimately, this was no more asymptotically efficient than the initial solution and the lengthy investigation significantly impacted the project schedule.

# Conclusion

We accomplished the goals of our project: implementing an initial GUI for a given web-based game engine, and creating a simple game with the resulting editor as a proof-of-concept for completeness of functionality in building a class of classical videogames.

Originally, the project schedule included user testing. However, this has become unrealistic due to unforeseen issues relating to the redefinition of classes.

Additionally, the delivered editor, as in all existing editors for game development, requires specific API knowledge for customizing object behaviors. This API knowledge does not cover the entire game engine. Rather, only usage information about GameObject, Transform, and additional utility functions are required.

Several functions from the game engine, as they were not mandatory for building a simple game, were not implemented into the editor due to time constraints. These include, but are not limited to: sounds, textures, sprite-based animation, lights, illumination, the use of rigid bodies, and particles. However, as our editor supports many features leading to the creation of our proof-of-concept game, the ability to add the other features should be implementable in the future without extensive technical difficulties.

While our work in this project does not expose every function present in the web-based game engine, sufficient functionality is supported such that game developers can create simple games through the editor. In this way the project still meets its initial goal. On a micro scale, this builds a foundation for a web application and paves the path for the future, in which our GUI’s design can be tested and augmented to offer an easier, more user-friendly experience. On a macro scale, this project is innovative in that it allows game development in a web browser. This expands the means in which game designers can develop games and allows games to be created from anywhere over the internet.

**Bibliography**

[1] Agah, A., and K. Tanie. “Intelligent Graphical User Interface Design Utilizing Multiple

Fuzzy Agents.” Interacting with Computers 12.5 (2000): 529-42. Oxford Journals Oxford

UP]. Web. 14 Feb. 2016.

[2] Chalmers, Patricia A. “The Role of Cognitive Theory in Human–computer Interface.”

Computers in Human Behavior 19.5 (2003): 593-607. ScienceDirect. Web. 4 Mar. 2016.

[3] Hillstrom, Anne P., and Yu-Chin Chai. “Factors That Guide or Disrupt Attentive Visual

Processing.” Computers in Human Behavior 22.4 (2006): 648-56. ScienceDirect. Web. 4 Mar. 2016.

[4] Stupak, Noah, et al. “SOCIALSENSE: Graphical User Interface Design Considerations for

Social Network Experiment Software.” Computers in Human Behavior 26.3 (2010):

365-70. Elsevier ScienceDirect. Web. 15 Feb. 2016.

[5] Pangalos, George J. “Designing the User Interface.” Computers in Industry 22.2 (1993):

193-200. ScienceDirect. Web. 12 Mar. 2016.

[6] Yee, Chee Kit, et al. “GUI Design Based on Cognitive Psychology: Theoretical, Empirical

and Practical Approaches.” Computing Technology and Information Management 2

(2012): 836-41. IEEE. Web. 10 Mar. 2016.

1. Unity3D: [https://unity3d.com](https://unity3d.com/),Unreal Engine: <https://www.unrealengine.com>. These are two of the most widely-available game engines, used both commercially and by hobbyists. [↑](#footnote-ref-0)
2. Observations based on Unity3D [↑](#footnote-ref-1)
3. Observations based on Unreal Engine [↑](#footnote-ref-2)
4. <https://en.wikipedia.org/wiki/Golden_age_of_arcade_video_games> [↑](#footnote-ref-3)