BlockPusher
Game Engine and Web IDE

Adam Coggeshall
Alex Germann
Matthew Grant

Contents

1. Introduction

This document provides a high-level design overview for the BlockPusher engine and web based IDE.

The BlockPusher engine provides a simple interface into the HTML5 media features to allow game development in a concise manner. It features a set of components that handle common tasks in game development including rendering, physics, collision detection, input and sound.

The BlockPusher web-based Integrated Development Environment (IDE) allows for quick prototyping of games using a simple web interface. It will support saving game assets to the cloud as well as sharing games and source through unique URLs.

2. Design Considerations

2.1. Assumptions and Dependencies

BlockPusher should be operating system independent because it will be running on the browser. It is assumed that the product will be used on a modern web browser with support for modern JavaScript. Current dependencies include the TypeScript compiler, HTML5 APIs provided by the browser, and the C# .NET framework.

It may also depend on various JavaScript libraries.

It is assumed the end user is familiar with both Javascript and API navigation.

2.2. General Constraints

BlockPusher will primarily be designed to support two dimensional, single-player games with tile-based maps. The engine will be modular, so other types of games could be supported in the future, but the primary focus is to get this simple format working with the IDE.

Although modern web browsers tend to be compatible, care should still be taken to test on the following major browsers: Firefox, Chrome, Safari, and Edge. The engine should also fail gracefully with a warning on non-supported browsers.

Security is a large concern whenever the execution of client code is concerned and must be considered during development. The safety of both the BlockPusher server as well as unsuspecting clients must be held in high regard.

Storage of game assets is another concern that must be addressed. Large games with many images and audio files can use up a large volume of storage space as well as system memory during execution. The product will need to provide the tools to deal with this problem. For instance, the site administrator should be able to view resource usage and set quotas on a per-game or per-user basis. Downloading these assets from the server is also a concern, so the engine should either wait for assets to load, or implement a lazy-loading scheme that fetches assets on demand.

2.3. Goals and Guidelines

The IDE should support the ability to refresh code as the game is running. This possible in a scripting language like JavaScript, but care must be taken to maintain the original states of objects. This will be achieved by modifying the prototype of the object in question, which will then updated or re-applied to object instances.

Error messages should be displayed to the user. If possible, the line where the error occurred should be shown in the IDE. Care must also be taken to have the engine generate error messages that make sense. For instance, if the engine API function is passed a null value, it should not complain about not being able to index null from some internal engine function but should check the parameter for the correct type at the beginning of the API function.

The end product should emphasize usability and simplicity of interface above all else.

3. Architecture

3.1. Engine Architecture

The BlockPusher engine will have a set of modular components including the base, rendering, physics/collisions and sound. Client code will include each component as necessary.

3.1.1. Component Architecture

The **Base** component will handle the most basic of necessities for a game including the game loop and execution of client code. For the most part it will not be accessible by the client.

The **Game** component is a special component which users can edit to add global game logic, as well as register blocks to be used by the world.

The **World** component will allow client code to interact with the map of blocks and define new types of blocks. Blocks themselves will not contain any state information, and can internally be represented by a two dimensional array of integers. It should should provide the following:

- A method to define a block type. This method should be passed a string which identifies the block type, as well as a JavaScript object which defines the image(s) to use, how the block handles collisions, and any other metadata useful to the client.
- Methods to retrieve and modify blocks at specific positions on the map.
- Methods to get the world's size. The world size will only be modifiable through the editor.

The **Time** component will track the current time and schedule events. It should provide the following:

- A method to get the current game time, in seconds.
- A method to get the time between the previous frame and the current frame, in seconds.

- A method to schedule events to be run in the future. This method should take a delay and a callback as parameters.

The **Input** component will track keyboard events and provide an easy way to determine which buttons are currently being pressed by the user. Mouse input is not currently planned since the mouse will be used to modify the game. It should provide the following:

- A method to check if a keyboard button is currently being pressed.
- A method to check if a keyboard button was pressed in the last frame.

The **Render** component will function as a wrapper around the HTML canvas. Its purpose is to simplify client interaction with the canvas. It should provide the following:

- Methods to control the viewport position by adding an offset to all rendering functions
- A method to draw images to the screen.
- Methods to draw colored primitive shapes.

The **Physics** component will supply the user with functionality for common physics tasks and collision detection via axis-aligned bounding box. It should provide the following:

- A method to run a raycast which returns the position, normal, and entity/block that was intersected.
- A method to check an axis-aligned bounding box for intersecting blocks and entities.
- A method to cast an axis-aligned bounding box along a ray which returns the position, normal, and entity/block that was intersected.
- Methods to get and modify the current gravity.

The **Sound** component will serve as a wrapper around the HTML audio controls. It should provide the following:

- A method to play a sound file.

The **Snapshot** component will allow the game's state to be saved and reloaded at a later time. This component will be used both for save games and to store level information. It should provide the following:

- Methods to save and load game states.
- Methods to serialize and deserialize game states, which will be used when loading or saving to files.

3.1.2. Client Code

Users will be able to add two types of script files to the project. The client can add a single **Game** component and any number of entity scripts. Entities are general objects in the game that can have their own state information. Entities should provide the following functionality:

- Properties that define the size of the entity's axis-aligned bounding box.
- A property for the default image used when rendering the entity.

- A property that defines the position of the object instance.
- A property that for the velocity of the object instance.
- An overridable method that is called on the entity when the world is updated.
- An overridable method that is called on the entity when physics are updated. This is separated from the above method because the default will provide simple physics that will work for most cases. This default implementation will update the entity's velocity based on the global gravity defined by the **Physics** component, then attempt to update the position using this velocity, checking for collisions using the **Physics** component's tracing methods.
- An overridable method that is called when the entity is rendered. The default implementation will simply draw the image specified above at the entity's position.

3.2. Website Design

The BlockPusher website will implement a simple IDE in the browser, allowing the user to run the game side-by-side with their code. It will be implemented in the typical MVC style.

The website must support a user registration and login system.

Published games will appear on the homepage. This list should include an image chosen by the creator of the game, a title, and the name of the publisher.

The website must handle storage, uploading, and downloading of game assets. These assets will include the following: JavaScript scripts, PNG images, WAV and OGG sounds, and saved levels. Along with these files, the server should also be able to serve a manifest that lists the files needing to be downloaded for a given game.

Functionality will be implemented to allow users to save games to the server as well as share those games through unique URLs.

4. User Interface

4.1. Game IDE

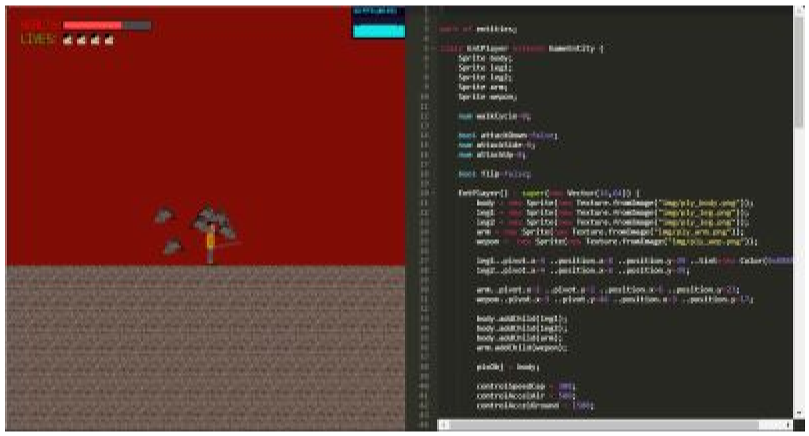
The IDE's user interface should be as simple as possible, but it must still provide the functionality required to edit the game. Roughly half of the screen will be devoted to the game view, the rest will be used for the code editor and navigation.

- **Game View -** This will allow the user to play the game being edited. When it is in focus, the keyboard can be used as input the game. It will also allow the user to move, add, and remove entities and blocks using the mouse. It should provide buttons to pause, resume, and restart the game from a saved state.
- **Editor -** Used to edit the currently selected file. Should provide syntax highlighting. Shortcuts commonly used to save and run code, such as CTRL-S and F5, should save the current file and refresh the code in the game respectively.

- **File Browser** - This will list the directory structure of the project and allow users to modify, upload, and delete files from the project. Script files will be modifiable using the editor, while other assets must be edited externally. The file browser will contain another tab which allows the user to select entities and blocks to be placed in the game view.
- **Messages** - A small console that displays compile and runtime errors. To save space and make the interface less complex, it will be hidden in the absence of output.
- **Navigation Bar** - A navigation bar used throughout the website. It should display a login/logout button and buttons for navigating the site.

4.1.1. Mockup

This mockup shows the basic layout of the IDE, including the game view and editor. The file browser, navigation bar, and messages area are not pictured.



5. Security Considerations

There are a number of likely attack vectors on the product. Because it is running untrusted code on users' machines, extra caution must be taken. This section outlines some of the most likely issues and suggested countermeasures. It is important that whatever security features are used are carefully audited.

5.1. Cross Site Request Forgery / Cross Site Scripting

Ideally, the product would completely sandbox code to prevent malicious use of game scripts. The web worker API could be used to do this, but it might introduce a significant amount of latency, since data would need to be passed between the worker context and the parent context.

Placing the game view in a frame and setting a strict Content Security Policy should mitigate CSRF attacks, but other JavaScript APIs not affected by the CSP could still be dangerous. Simply removing dangerous APIs from the global environment may prove effective, but it is dependant on the Javascript engine being used and may not be reliable.

5.2. Denial of Service -- Server

Since the application is storing files, a user might attempt to upload large amounts of data to waste space on the server. This could be mitigated using a per-user file quota.

5.3. Denial of Service -- Client

An attacker could add an infinite loop to game code which would crash the browser tab running the game. A crashed browser tab is not a major security risk, but it does negatively affect the user's experience.

Ideally, the product should check for long running loops and exit them. A library (loop-protect) exists for this purpose, although it does not handle recursive loops, so modifications may be necessary.

A similar tactic to the one described in the previous section could be used to have a client download large amounts of useless files disguised as game assets, wasting the client's memory and potentially crashing the browser tab. A per-game file quote could mitigate this.

6. Revision Log

Version 0.1, 11/25/17, Matt Grant

Version 0.2, 12/7/17, Adam Coggeshall

Version 0.21, 12/7/17, Matt Grant