BlockPusher
Game Engine and Web IDE

Adam Coggeshall
Alex Germann
Matthew Grant

Contents

1. Introduction

This document provides a high-level overview for the implementation of 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.

The BlockPusher web application stores games and assets. It supports saving game assets to the cloud as well as sharing games and source through unique URLs.

2. Architecture

2.1. Engine Architecture

The BlockPusher engine has a set of modular components including the world, rendering, input, time, physics/collisions, and sound. Client code is able to interface with these as necessary. Most of these components have a 'setup' method, which is called when the engine is started or reset, and an `update` method used to refresh the component each frame.

The engine communicates with the editor or player using the browser's message passing system. This is covered in more detail in the next section.

The engine source code is contained in '/engine/src' in our repository. The entry point is 'engine.ts'. It is built using 'npm run-script build' from the 'engine' directory.

2.1.1. Component Architecture

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

- gravityX: number, gravityY: number
  - Global gravity used by the default **GameObject** physics implementation.
- createObject(className: string, x: number, y: number): GameObject
  - Instantiates a GameObject of the given type in the world.
- createBlockType(name: string, imageFilename: string): void
  - Registers a block type with a given image. Should be called **once** in the World script. The name must stay consistent for level saves to work.
- getBlockTypeAt(x: number, y: number): string | null
  - Gets the block type at the given world coordinates. Returns the name of the block at the given position, or null if there is no block.
- setBlockTypeAt(x: number, y: number, blockType: string | null)

    - Sets the block type at the given world coordinates. Provide the block name, or null to clear the space.
- drawBackground(): void
    - User code can override this to draw things before anything else is drawn. This should normally clear the screen or draw some other kind of background.
- drawForeground(): void
    - User code can override this to draw things after everything else is drawn. This can be used to draw a HUD, some simple UI, or effects.
- update(): void
    - Called every frame while the game is running. Users can override and place global game logic in this method.
- render(): void
    - **Internal.** Renders the world and its blocks.
- setEditorPlacementObject(type: string, name:string)
    - **Internal.** Used to tell the world what kind of object the user wishes to place, if in edit mode.
- updateEdit(): void
    - **Internal.** Handles world editing.
- resize(sizeX: number, sizeY: number): void
    - **Internal.** Resizes the world. **Does not preserve block data.**
- save(): WorldSave
    - **Internal.** Saves the world to a javascript object which can be serialized.
- load(WorldSave): void
    - **Internal.** Loads the world from an object created by save().

The **Time** component tracks the current time It should provide the following:

- getTime(): number
    - Get the time since starting the game, in seconds.
- getDelta(): number
    - Get the time between the previous frame and the current frame, in seconds.
- update(): void
    - **Internal.** Updates the above times.

The **Input** component tracks keyboard events and provides an easy way to determine which buttons are currently being pressed by the user. Note that key names in this component are the browser's KeyboardEvent.key values, are not case-sensitive, and in most cases are the character produced by pressing the key on the keyboard. This component provides the following:

- isKeyDown(keyValue: string): boolean
    - Is the key with the given name currently being pressed?
- wasKeyPressed(keyValue: string): boolean
    - Was the key with the given name pressed this frame? (Did it move from up to down?)

- wasKeyReleased(keyValue: string): boolean
    - Was the key with the given name released this frame? (Did it move from down to up?)
- isMouseButtonDown(button: number): boolean
    - Is the given mouse button currently pressed?
- wasMouseButtonPressed(button: number): boolean
    - Was the given mouse button pressed this frame? (Did it move from up to down?)
- getCursorPos(): {x: number, y: number}
    - Retrieves the position of the mouse cursor in world coordinates.
- update(): void
    - **Internal.** Used to refresh the internal list of pressed keys and mouse buttons.

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

It has two modes of operation: The normal mode allows drawing objects using absolute pixel coordinates. This is mainly useful for drawing the game background and foreground (UI/HUD). The camera mode draws objects using world coordinates, and is offset by the current camera position.

The 'style' parameters can be any HTML color string.

This component provides the following:

- clear(style: string): void
    - Clears the screen with the given color. Should only be called from the normal rendering mode.
- drawRect(style: string, x: number, y: number, width: number, length: number): void
    - Draws a filled rectangle.
- drawRectOutline(style: string, x: number, y: number, width: number, height: number, lineWidth = 1): void
    - Draws an outline of a rectangle.
- drawCircle(style: string, x: number, y: number, radius: number): void
    - Draws a filled circle.
- drawCircleOutline(style: string, x: number, y: number, radius: number, lineWidth = 1): void
    - Draws an outline of a circle.
- drawLine(style: string, x1: number, y1: number, x2: number, y2: number, lineWidth = 1)
    - Draws a line between two points.
- drawImage(imageName: string, x: number, y: number, width?: number, height?: number): void

        - Draws an image with the given name at the given position and with the given size. The size defaults to the true image size if the renderer is in normal mode, otherwise it defaults to 1 tile by 1 tile.
- drawText(text: string, x: number, y: number, style = "black", font = "20px sans-serif"): void
    - Draws text at the given position. Mostly useful for debugging.
- setCameraPos(x: number, y: number): void
    - Used to set the camera's position. Note that the camera will not be used to render unless it is enabled.
- getCameraPos(): {x: number, y: number}
    - Used to retrieve the camera's current position.
- enableCamera(): void / disableCamera(): void
    - **Internal.** Enable/Disable camera rendering mode, described at the beginning of this subsection.
- registerImage(imageName: string, url: string): void
    - **Internal.** Used by the engine to load an image.
- findImage(imageName: string): Image
    - **Internal.** Retrieves the HTML Image object associated with the given filename.
- setup(canvasName: string): void
    - **Internal.** Sets up the renderer and the auto-resizing functionality of the game window.

The **Audio** component serves as a wrapper around the HTML audio controls. It provides the following:

- playSound(soundName: string): void
    - Plays a sound with the given filename.
- stopAll(): void
    - Stops all audio.
- registerSound(soundName: string, url: string): void
    - **Internal.** Used by the engine to load a sound.

The **Collision** component supplies the user with functionality for common physics tasks and collision detection via axis-aligned bounding box.

The ray and bounds casting functions return a CastResult structure, described by the following code:

```
interface CastResult {
   // Did the cast hit?
   hit: boolean;
   // Coords of the end of the ray
   x: number;
```

```
    y: number;
    // The length of the ray.
    distance: number;
    // Hit side. If set, will be one of the constants defined in
the Collision module.
    // Not set if no hit or if the ray started inside an object.
    side?: string;

    // Block type string, set if the cast hit a block.
    blockType?: string;
    // Coords of the block that was hit, set if the cast hits a
block.
    blockX?: number;
    blockY?: number;

    // The object that was hit, set if the cast hits an object.
    object?: GameObject;
}
```

The component provides the following:

- Top: string / Bottom: string / Left: string / Right: string
    - These are constants used by CastResult.side.
- checkPoint(x: number , y: number): GameObject[]
    - Returns a list of objects overlapping the given point.
- checkBounds(x: number , y: number, width: number, height: number): GameObject[]
    - Returns a list of objects overlapping the given bounding box.
- castRay(x: number, y: number, dx: number, dy: number, ignoreObject?: GameObject): CastResult
    - Casts a ray through the world that can hit both blocks and objects. Can optionally be set to ignore one object. See above for documentation on the return value.
- castBounds(x: number, y: number, width: number, height: number, dx: number, dy: number, ignoreObject?: GameObject): CastResult
    - Casts a bounding box through the world that can hit both blocks and objects. Can optionally be set to ignore one object. See above for documentation on the return value. The starting position and returned hit position are both the upper-left corner of the bounding box.
- update(): void

    - **Internal.** Updates the grid used for broad phase object collision detection.

2.1.2. GameObjects

GameObjects are more general objects in the game that can have their own state information. User code can create concrete **GameObjects** subclasses. **GameObjects** provide this functionality:

- x: number / y: number
    - The position of the object.
- velX: number / velY: number
    - The velocity of the object. Used by the default updatePhysics implementation.
- width: number / height: number
    - The size of the object's bounding box.
- image: string
    - The image used by the default render method. Also used as an icon in the editor.
- bounciness: number
    - Used by the default updatePhysics implementation to determine how much energy is retained when an object bounces. A value of 0 disables bouncing. A value of 1 means the object will retain all of its energy. Larger values than 1 are not recommended.
- friction: number
    - Used by the default updatePhysics implementation to determine how much energy is lost when sliding along a surface.
- setup(): void
    - This method should be overridden by user code to do any desired setup to a new instance.
- remove(): void
    - Deletes the object from the world.
- update(): void
    - This method should be overridden by user code for anything that should occur every frame.
- updatePhysics(): void
    - Called by the engine along with update. The default implementation applies gravity and velocity, and does simple bouncing and friction when a collision occurs. If this functionality is not desired, this method can be overridden.
- render(): void
    - Called by the engine to render the object. The default implementation renders the object's image over its bounding box, but it can be overridden by user code.

Active **GameObject** instances are managed by the **GameObjectManager** component, defined in the same file.

2.1.3. Client Code Management

The process of managing user code is controlled by an internal component called the **CodeManager**. For any script added to the game, the manager contains a setup function, the compiled version of the script, created via the javascript Function constructor. The manager also tracks the class/prototype added by the script. In order to create this class, the manager creates a subclass of GameObject, then passes it to the setup function which populates the class. The manager will update these two things whenever it is requested to do so by the editor. This component contains:

- World: World
  - The current world instance. Engine code should always use this to retrieve an up-to-date reference to the world.
- getGameObjectClass(name: string): GameObject
  - Retrieves the GameObject class of the given name.
- getGameObjectClassName(obj: GameObject)
  - Retrieves the name of the class associated with the given GameObject instance.
- runScript(name: string, code: string | null): void
  - Runs a script, creating or updating the associated setup function and class.
- setup(): void
  - Used to set up or reset the component. Should completely rebuild all classes from scratch.

2.2. IDE Architecture & Messaging

The BlockPusher IDE's layout is contained in 'website/BlockPusher/Views/Play/Edit.cshyml', and its code is contained in 'website/BlockPusher/Scripts/editor.js'.

Most of the IDE code is dedicated to managing lists of files and game objects.

In order to facilitate editing without constantly pushing updates to the server, the editor pulls all of a game's content from the web API and stores it in memory as blobs. It then passes them to the engine as data URLs. All modified files are committed as a batch using the web API when the user clicks the "Save" button.

The actual text editor is Monaco by Microsoft. Its documentation can be found here.

Since the engine is placed in a sandboxed iframe, the editor and player both communicate with the engine using window.postMessage. The messaging API is as follows:

Editor → Engine

- type = "setFile"

    - Used to send a file to the engine. The “file” field contains the filename, and the “url” field contains the URL of the file. In the player, this will be a normal HTTP URL, but in the editor the entire file is sent as a data URL.
- type = “setMode”
    - Enables or disables the engine’s level editor.
- type = “selectObject”
    - Used to tell the engine’s level editor what kind of object the user wishes to place.
- type = “requestLevel”
    - Used when the user wishes to save the game. The engine will respond with a “saveLevel” message. When this response message is received by the editor, it will send the level and all other modified files to the web API.

Engine → Editor

- type = “engineReady”
    - Sent when the engine is ready to receive messages. May be removed in the near future as it is not always received properly. When this is received, the editor will start the process of fetching content and sending it to the engine.
- type = “setObjectList”
    - Notifies the editor that it should update its list of block and object types. The list is contained in the “list” field of the message, and is used to update the visual list on the right side of the editor UI.
- type = “saveLevel”
    - Notifies the editor of changes to the level. The editor will update it’s in-memory version of “level.json” with the level data in the “data” field of the message.

2.3. Website Design

The BlockPusher website implements a simple IDE in the browser, allowing the user to run the game side-by-side with their code.

The website supports user registration and login.

The website handles storage, uploading, and downloading of game assets. These assets include the following: JavaScript scripts, PNG images, WAV and OGG sounds, and saved levels.

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

2.3.1. User Registration/Login

The website user login and register system is the default ASP.NET user system implementation. All user account methods are contained in the AccountController.cs. See ASP.NET documentation for more information on their user system.

2.3.2. File Management

File management is handled within ContentController.cs. Users are able to save game assets to the server in order to use them within their game. This file contains:

- UpdateGameFiles(int gameId, string[] deleted, IEnumerable<HttpPostedFileBase> updated):
    - Handles general file management. Deletes all files with names passed in with the 'deleted' array. Updates (saves) all files in the updated IEnumerable.
- CheckFileName(string fileName):
    - Checks to see if a given file name is sanitized correctly.
- GetFileNames(int gameId):
    - Gets the files names for the game currently being edited.
- CheckGameOwnership(int gameId):
    - Checks the SQL database to ensure that the game is owned by the current user.

2.3.3. Game Saving/Sharing

Games are saved to the SQL database with info: Author, Title, Description, and an auto-generated game ID. They are then displayed on the GameList page with a link to the Play page for that particular game. Games are saved using the ContentController:

- CopyGame(int gameId):
    - Creates a copy of a game, including a new SQL entry as well as the game file content.
- UpdateGameInfo(int gameId, string title, string description):
    - Updates a given games title and description in the SQL database.

2.3.4. Azure SQL Database

The game's database is currently hosted on Azure, using Azure SQL. Tables include the Games table with the information described earlier, as well as the ASP.NET auto-generated user identity tables.

3. User Interface

3.1. Game IDE

The IDE's user interface is as simple as possible.. Roughly half of the screen is devoted to the game view, the rest is 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.

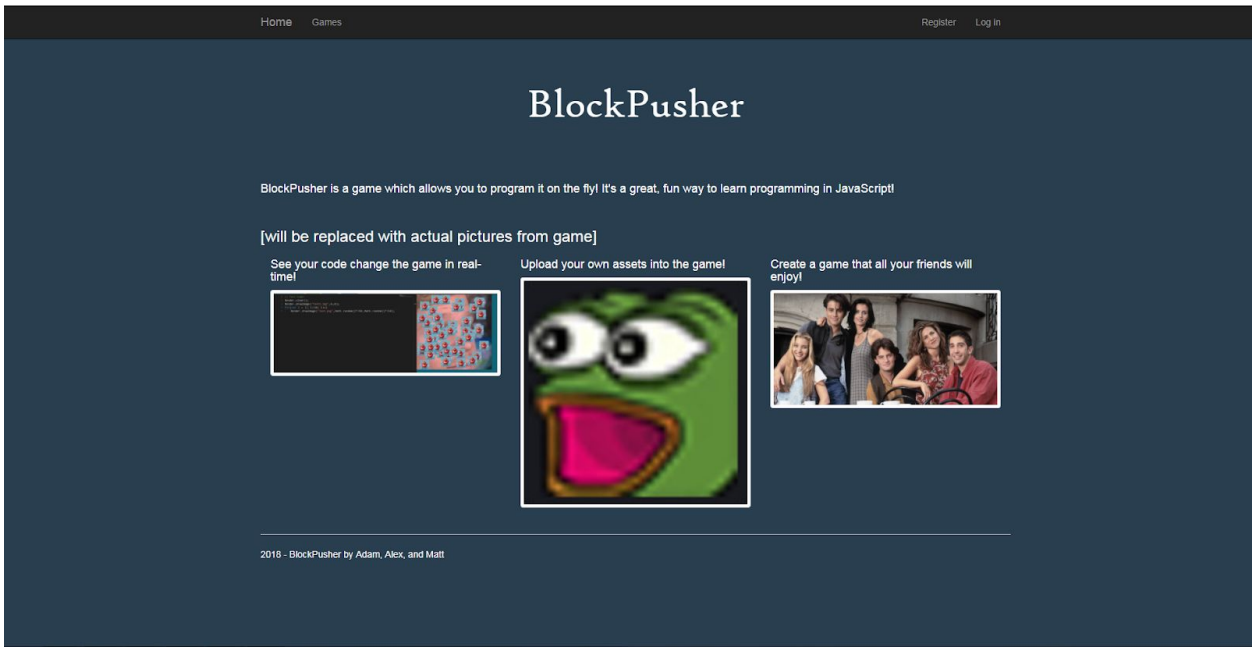
- **Navigation Bar -** A navigation bar used throughout the website. It should display a login/logout button and buttons for navigating the site.

The Editor Breakdown in the following section shows more detail on these elements.

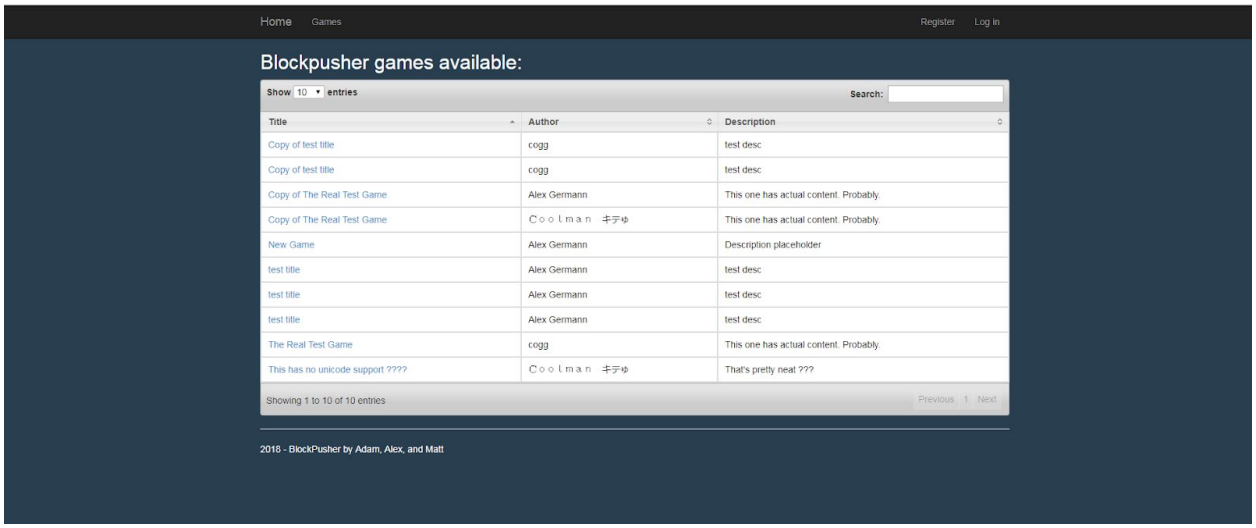
3.1.1. Screenshots

These images show the basic layout of the website, including the game explorer, the IDE and the home page.

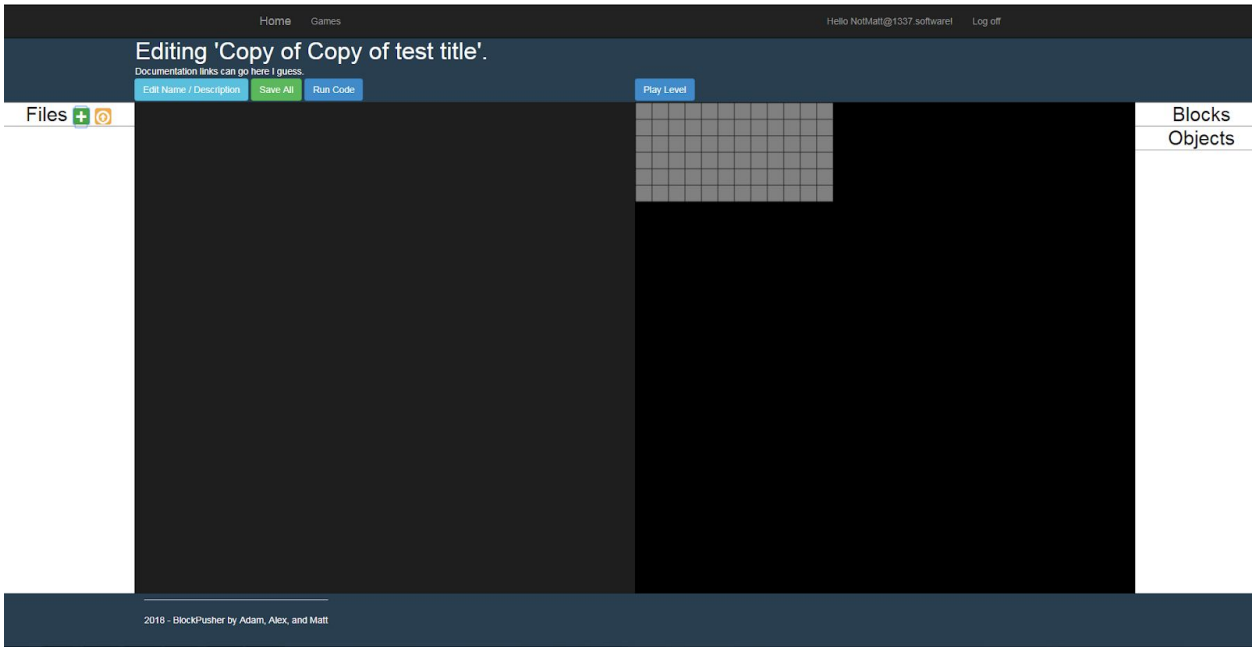
Home Page



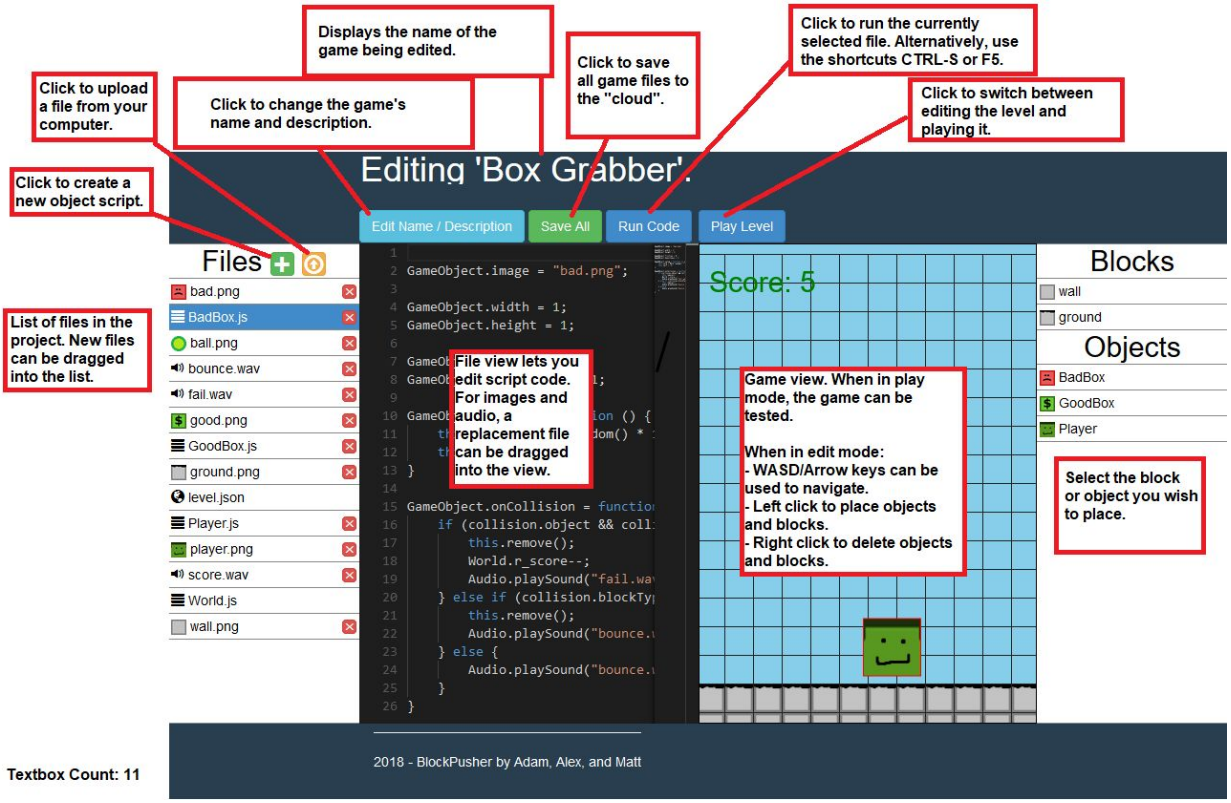
Game Explorer



Game Editor



Editor Breakdown



4. 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.

4.1. Cross Site Request Forgery / Cross Site Scripting

Security is always a concern when executing arbitrary code. In order to mitigate these risks, the game engine and user code is executed in a sandboxed iframe. The browser's message passing system is used to communicate with this context, as described in an earlier section.

In the future we may try to delete potentially harmful functions from the sandboxed environment.

4.2. 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 is currently not dealt with.

4.3. 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.

This is currently not dealt with.

5. Revision Log

Version 0.1.0, 11/25/17, Matt Grant

Version 0.2.0, 12/7/17, Adam Coggeshall

Version 0.2.1, 12/7/17, Matt Grant

Version 0.9.9 4/19/2018 Matt Grant

Version 1.0.0, 4/20/2018, Adam Coggeshall

Version 1.0.1, 4/21/2018, Alex Germann

Version 1.0.2, 4/24/2018, Adam Coggeshall