Technical Design Document

Defender's Light

By Isaiah Mann, Ranjini Das, Kaylynn Kosyka

(?) - *Represents a tentative feature or comment. May not be included in final version of the draft.*

Abstract

Glossary

Technology

Methodologies

Constraints

Internal Systems
- MannBehaviour
- DataController
- StatsController
- RewardController
- AnalyticsController
- AchievementController
- UpgradeController
- InputController
- AudioController
- EventController
- UIController
- WorldController
- MapController
- TowerController
- EnemyController
- EditorController
- UnitTestController
- TutorialController
- CheatsController
- AnimationController

Project Layout

Target Platforms

Plugins & SDKs

Task Management

Version Control

Code Conventions
- Cases
- Indent Style
- Specificity
- Misc
- Refactoring

Commenting Conventions

Code Review

Resources

Optional Resources

Revision History

Abstract

Defender's Light is a tower defense game where the player can only build structures in illuminated portions of the map. The backing system is created in Unity and is designed to be highly objected-oriented.

Glossary

1. *For design specific terms, see the Glossary in the Game Design Document*
2. **Cross-Platform**
    a. Expectation that the game should be exportable from Unity and able to function on a range of different platforms (minimum iOS and Android) without additional development or support.
3. **Internal System**
    a. Internal systems represent the major components of the Unity project
    b. They should be designed to be fully modular so that a piece of the system can be abstracted out, reused, or swapped out for a different implementation without affecting the integrity of the overall system
    c. The systems are intentionally given names in PascalCase

        i. Each system will have a singleton instance which provides oversight and control the other classes in the system
        ii. This singleton should be the only touchpoint of communication between other systems (though ideally system only communicate with each other by using the EventController as a mediator)
    d. The one exception of the above statements is the needlessly self indulgent "MannBehaviour" script
        i. This is a subclass of MonoBehaviour that all Unity components within the project are expected to inherit from
            1. i.e. No other class created for this project should inherit from MonoBehaviour directly
            2. Instead, they should inherit from MannBehaviour
            3. This allows for easier standardization of code and eliminates bugs that result from not adhering to best practices
4. **Value Types**
    a. C# has multiple value types:
    note: this list does not include primitives, e.g. integers
    b. Object
        i. The base class that all C# scripts inherit from
        ii. Can Contain variables and methods of any accessibility level that C# contains
    c. Interface
        i. Defines **only** public method and variables headers
        ii. Does **not** define any internal functionality for those methods
        iii. If a class **implements** a method, it must contain all methods that the interface does with matching headers
    d. Enum
        i. A numbered list of preset values
        ii. Can be cast to an integer and vice versa
        iii. Example of Use: Days of the week
    e. Struct
        i. Like a class, but can only store data
        ii. Has **no** methods
        iii. Example of Use: Coordinates of a triangle
5. **Capitalization Cases**
    a. Different types of capitalization (used to imply variable type)
    b. **PascalCase**: A type of capitalization where the first letter of each word is capitalized
        i. e.g. `NameOfVariable`
    c. **CamelCase**: A type of capitalization where the first letter of a variable is lowercase, and all following words in the variable are uppercase
        i. e.g. `nameOfVariable`

d. **Underscore Mixed Case:** A type of capitalization where the first character in a variable name is an underscore, following by an uncapitalized word, with any following words in the variable name capitalized
    i. e.g. `_nameOfVariable`
e. **Screaming Caps:** A type of capitalization where are all letters in the variable name are capitalized with the words separated by underscores:
    i. e.g. `NAME_OF_VARIABLE`
f. **Egyptian Brackets (1TBS)**: A convention for bracket spacing where the leading bracket is on the same line as its related statement:

```
i.  if (condition) {
    ...
    }
```

Technology

1. **Engine**: Unity 5.3.5f1
2. **Language**: C#
3. **Version Control:** GitHub
4. **Plugins/SDKs**: Mixpanel, TestFairy
5. **Additional Required Applications**
    a. Maya 2017
        i. Allows Unity to correctly handle .ma files (our 3D models)
    b. MonoDevelop or Visual Studio
        i. Supported IDE's for Unity)

Methodologies

1. Object-Oriented Programming
2. Observer Pattern
3. Test Driven Development
4. Singleton Pattern
5. Dependency Injection (?)

Constraints

1. The system must support cross platform builds:
    a. Android
    b. iOS
2. Preferably, the system will also support WebGL and Standalone builds
    a. However, mobile-specific SDK's will not be expected to function on these platforms

Internal Systems

Required

1. MannBehaviour
    a. DataController
        i. StatsController
        ii. RewardController
    b. InputController
    c. AudioController
    d. EventController
    e. UIController
    f. WorldController
        i. MapController
        ii. TowerController
        iii. EnemyController
    g. TutorialController
    h. CheatsController
    i. AnimationController

Stretch Goals

1. Submodules of DataController:
    a. AnalyticsController
    b. AchievementController
    c. UpgradeController
2. EditorController
    a. UnitTestController

MannBehaviour

A needlessly self indulgent play on words from MonoBehaviour. Subclasses MonoBehaviour (the base script used to create Unity components). Adds event based functionality and standardizes object initialization practices.

DataController

Responsible for storing, tracking, and serializing statistics. Tracks progress, game sessions, and unlocks. Handles all saving and loading functions.

StatsController

Subsystem of DataController, specifically focusing on player statistics.

RewardController

Subsystem of DataController, focuses on rewards to the player, from ranging from finishing levels to killing enemies to unlocking achievements.

AnalyticsController

Subsystem of DataController. Uses the Mixpanel API. Sends user data and events to the Mixpanel service for the developers to track player progress and engagement.

AchievementController

Subsystem of DataController. Integrated with Game Center and Google Play. Controls the achievements the user has earned the incentivized rewards the players earn from them. Achievements can be initialized from JSON.

UpgradeController

Subsystem of DataController. System for applying upgrades to towers and saving those upgrades in player data.

InputController

Handles all player input to the screen. Should abstract different methods of input: mouse vs. touch into a single system of events. This will enable easier Editor testing and cross-platform support.

AudioController

The system that handles all audio in the game. The source files are stored in the *Assets/Resources* directory so they're dynamically available at runtime. The AudioController should load them based on which events are called and handle playing them at the appropriate times.

EventController

A centralized system that all other systems message to report events. The EventController then broadcasts out events to any system subscribed to that event. This centralizes and standardizes the communication between different systems in the game.

Static calls can be made to the EventController using the following function:

```
EventController.Event("nameOfEvent");
```

These events can be received in any script that inherits from `MannBehaviour`:

```
protected override void HandleNamedEvent (string eventName) {
        if (eventName == "nameOfEvent") {
            // Run code to handle the corresponding event
        } else {
            base.HandleNamedEvent(eventName);
        }
}
```

UIController

Responsible for controlling the appearance and behavior of the user interface. Should abstract out behavior and appearance of elements so they can be swapped out arbitrarily.

WorldController

System responsible for handling all in-game objects. This extends to the environment, structures, and enemies. Should provided centralized control and oversight over the game world.

MapController

A subsystem of WorldController. Responsible for handling the display and behaviour of the game map. Contains the logic for placement and positioning of towers.

TowerController

Subsystem of WorldController. Handles the control and behavior of the towers within the game.

EnemyController

Subsystem of WorldController. Handles the control and behavior of the enemies in game, including their navigation logic.

EditorController

Developer only system. Custom tools for designing and debugging features of the game. Should promote test driven development and unit testing, as well as fully tunable game behaviour.

UnitTestController

Responsible for running unit tests on the system. Used in editor to verify that code is working as expected. Most ambitious version of this stretch goal would extend to functionality on mobile devices.

TutorialController

Responsible for running the logic and displays of the tutorial. Saves the state of the tutorials and reads them in from JSON files.

CheatsController

System for triggering specific game states and unlocks in order to test and debug more quickly. Cheats can be initialized from JSON.

AnimationController

Handles animations for the game, including units, UI, and particle effects. Can either attach modules to objects or be triggered via events.

Project Layout

1. Unity Project layout should adhere to the following structure (*note: structure only includes portions of the project files that are under version control)*
2. **Assets** is the folder in which Unity stores all scripts, art, audio, and other source files specific to the project.
    a. **Art** contains all visual assets (2d/3d) created for the game
    b. **Scenes** contains Unity scenes.
    c. **Scripts** contains all scripts written by the team.
    d. **Prefabs** contains saved Unity objects, often referred to as "prefabs." The purpose of these objects is to be reusable in a variety of cases.
    e. **Resources** is a folder containing assets that need to be accessible at runtime. All audio assets should be stored inside a subfolder: **Audio** so the event based audio system can load and play them dynamically.
3. **ProjectSettings** is a directory automatically generated by Unity. Should not have to modify these files directly. Only through the Unity Editor interface.
4. *Note:* At different directory levels throughout the project, there are folders denoted as "Main." These folders should comprise the central elements of the external directory. Ideally, directories should only contain sub-directories **or** files, **not** a combination of both. This makes for a clearer organization and hierarchy.

Target Platforms

Required

1. iOS

2. Android
3. Support for a range of screen sizes: 3.5" (*iPhone 4s*) to 12.5" (*iPad Pro*)
    a. Though the game may not support the internal specs, e.g. RAM of the *iPhone 4s*

Stretch Goals

1. WebGL
2. Standalone
    a. Mac
    b. PC
    c. Linux

Plugins & SDKs

1. No external API's are required for the MVP (Minimum Viable Product) version of the game
2. However, there are several to be considered for stretch goals:
    a. **Mixpanel**
        i. Player event tracking and analytics
        ii. http://mixpanel.com/
    b. **TestFairy (iOS)**
        i. TestFairy's SDK is implemented by default on Android when using TestFairy for AdHoc distribution
        ii. However, it must be manually added via Xcode for the iOS version
        iii. http://docs.testfairy.com/iOS_SDK/Integrating_iOS_SDK.html
    c. **Game Center (iOS)**
        i. Achievement tracking through Apple's integrated game service
        ii. https://developer.apple.com/game-center/
    d. **Google Play Game Services (Android)**
        i. Achievement tracking through Google Game Services
            1. *Note: Also technically supported on iOS, but likely cleaner to use Game Center*
        ii. https://developers.google.com/games/services/common/concepts/achievements

Task Management

Task management will be done through a lightweight version of agile: utilizing a Kanban board and daily standups. The lead programmer will provide direction and assign tasks, but all engineers are able to take on additional tasks, or request to be taken off tasks at their discretion.

Version Control

GitHub will be used for version control. Each engineer should maintain a personal branch with their name to submit hotfixes and small changes. For any large change, engineers should create a branch specific to that edit. Such branches should fall under one of the following categories:

1. Improvement
2. Feature
3. Bug

They should be named appropriately. **e.g.** feature/adding-score-tracking or bug/fixing-ui-flicker. Note the use of the slash after the category and the use of dashes/lowercase for the branch title.

Once work is complete on a specific task, or a significant amount of progress has been made, the engineer should open a pull request into the "master" branch. This pull request should be reviewed by at least one other engineer. The reviewing engineer(s) should checkout the branch locally, test it, and test merging it into master. Following this investigation, they should make any comments related to revisions, and once any necessary revisions have been made, approve the merge.

Code Conventions

Below is a list of conventions that all engineers should strive to adhere to in their code. While functionality and efficiency are the top considerations in reviewing other engineers' code, this list of conventions should also be taken into account when reviewing pull requests.

1. Cases

a. Different cases (methods of capitalization) should be used based on context:
b. **PascalCase**:
    i. All Class, Interface, Enum, and Struct Names
    ii. Public variables and methods
    iii. All enum types
c. **camelCase**:
    i. Protected and Private methods
d. _underscoreMixedCase
    i. Protected and Private variables
e. SCREAMING_CAPS
    i. const and final variables

2. Indent Style

a. The codebase universally uses Egyptian Brackets (1TBS)

3. Specificity

a. Variable, Method, and Class names should be as specific as possible
   i. This alleviates the need for comments
   ii. Length of variables names is not a concern because MonoDevelop has excellent auto-complete
   iii. Good Variable Declarations:

```
1. public int NumberOfPlayers;
2. const string ACHIEVEMENT_SAVING_KEY = "Achievements";
3. private string _nameOfPlayer = "Larry";
```

   iv. Good Method Headers

```
1. public void ProcessClickFromPosition (int x, int y);
2. protected int getNumberOfPlayers ();
3. string readTextFromFile (TextAsset document);
```

   v. Good Class Names:

```
1. UnlockableAchievement
2. InventoryItem
3. VisualTutorialScreen
```

   vi. Bad Variable Declarations

```
1. int _b = 5;
2. protected string _name = "Class 1";
3. public bool Test = false;
```

   vii. Bad Method Headers

```
1. string run ();
2. public void Test();
3. protected bool findStuff(string stuff2);
```

   viii. Bad Class Names

```
1. Stuff
2. Object2
3. Overwatch
```

4. Misc

a. If you do not add an access modifier (e.g. `public`, `protected`, `private`, `internal`), C# automatically assumes that variable or method is `private`.
b. It is acceptable and preferred to leave not specify an access modifier in the codebase when the modifier is intended to be `private`.

5. Refactoring

a. Refactoring is an excellent practice in keeping code clean and free of tech debt
b. Engineers should always feel free to refactor code, providing they adhere to the following conditions:

i. Only refactor code you're familiar with and understand
ii. Thoroughly test before pushing a major refactor
iii. Obtain permission from an engineers frequently working with an internal system before refactoring it
iv. Don't refactor an internal system in tandem with major work being done on it to add new features or fix bugs

Commenting Conventions

1. Use comments to explain things not made clear by the code
2. Do not use the comments to summarize things that the code already shows:
    a. e.g.
    ```
    // Gets a date
    public DateTime GetDate ();
    ```
3. Use comments at the top of each class file to summarize a class's functionality
    a. Include the names of any authors of that file in the this class header comment and any dependencies (classes this class relies on for its functionality)
    d.

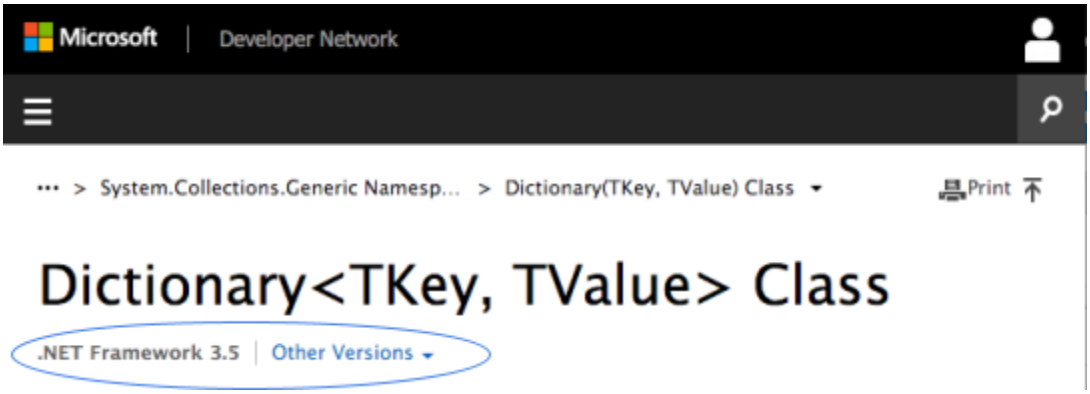
Code Review

1. Code review should primarily be conducted when doing pull requests
    a. This is the prime setting in which to give fellow engineers feedback on their work
    b. Also a well poised chance for them to improve
2. If the time allows, the engineering team will also attempt to schedule infrequent in-person code reviews. These reviews will be more formal, entailing more discussion of conventions and best practices.

Resources

1. **Unity API Documentation**: https://docs.unity3d.com/ScriptReference/
    a. For more information on the Unity libraries: UnityEngine, UnityEditor, UnityEngine.UI, etc.
2. **Unity Answers:** http://answers.unity3d.com/
    a. For specific queries and bugs found in Unity
3. **Stack Overflow**: http://stackoverflow.com/
    a. For literally any code-related question
4. **C# Documentation**: https://msdn.microsoft.com/en-us/library/kx37x362.aspx
    a. For documentation and more information about the C# language

b. Note: Some features here may not be implemented into Unity yet. Unity uses Version 3.5 of the .NET Framework, while some this documentation refers to Version 4 or newer. Be careful to check at the top of the page:



c.

5. **Git Documentation**: https://git-scm.com/documentation
    a. This is helpful if you're using GitHub from the command line
    b. If you prefer a GUI Application, the top recommendation is SourceTree
        i. It's reasonably intuitive and very powerful
6. **GitHub Documentation**: https://guides.github.com/
    a. For GitHub specific version control help
    b. Note: GitHub is built on Git, but is has certain custom features, such as "Pull Requests" that are extensions of built-in Git features (is this case "merging")
    c. It also provides a great free way to host Git repositories

Optional Resources

1. Windows Setup Tips
    a. Fixing line endings in Visual Studio 2015 for Windows
        i. When opening Visual Studio, you may see the message "The line endings in the following file are inconsistent. Do you want to normalize the line endings?" Pick "Yes" and "Do not show this message again".
        ii. Alternatively, you may see this message when you try to compile programs on Visual Studio. The message will say, "There are inconsistent line endings in this file. Some are for Mac OS X (UNIX) and some are for Windows. To fix this, follow these steps:
            1. Go to C:\Programs Files\Unity\Editor\Data\Resources\Script Templates\
            2. Open up each .txt file in the Script Templates folder in Visual Studio
            3. For each file in Visual Studio, choose File > Advanced Save Options > Line Endings. Change the option to Windows.
            4. Save each file.

5. You may have to change the Read/Write permissions for the Script Templates folder in order to save the .txt files. If you are unable to save files, follow this guide to change folder permissions.

Revision History

1. **10.14.16**
    a. Rearranged internal systems more cohesively
2. **8.13.16**
    a. Added indentation style
3. **8.03.16**
    a. Added AnimationController
4. **7.31.16**
    a. Added RewardController system
5. **7.30.16**
    a. Added note about “Main” folders to Project Structure section
    b. Specified when classes can be initialized from JSON
6. **7.25.16**
    a. Adding CheatsController to internal systems
7. **7.24.16**
    a. Adding TutorialController to internal systems
    b. Added Setting up Visual Studio 2015 for Windows under Windows Setup Tips to Optional Resources
8. **7.20.16**
    a. Rearranged sections by importance
    b. Added UnitTestController to internal systems
9. **7.16.16**
    a. Added initial content for the following sections
        i. Version Control
        ii. Code Conventions
        iii. Commenting Conventions
        iv. File Systems
        v. Project Layout
        vi. Code Review
        vii. Plugins & SDKs
        viii. Resources
    b. First draft complete
10. **7.10.16**
    a. Began to describe Internal Systems and added terms to Glossary