Get The Treasure

Summary:

This document has been compiled so that the reader can educate themselves as to every aspect of the game’s development process, user testing and structure of the underlying software and game mechanics as well as some reasoning and empirical evidence for why the game is constructed the way it is.

Security Notes

***“Never, ever put a non-obfuscated, non-site locked version of the swf ANYWHERE online, It WILL get stolen.“***

Flash SWF’s are INCREDIBLY easy to steal. I use an obfuscation tool that has an added function of breaking the swf if it run anywhere on a domain I don’t want it to run. Never post a compiled swf online unless it has the site lock installed. There are teams of people in China who spend their whole lives ripping off swfs, decompiling them and distributing them on flash portals. This is a VERY real threat. If the swf is put anywhere online, people WILL steal it and distribute it and I will lose money.

Setting Up Flash Develop

Downloads:

* Download the **open source flex SDK** (required) from Adobe, <http://opensource.adobe.com/wiki/display/flexsdk/Downloads>
* Download the **“Debug Flash Player**” from Adobe (Special version of flash that allows you to see exception data and traces)
  + <https://www.adobe.com/support/flashplayer/downloads.html>
* Download and install **Flash Develop**
  + <http://www.flashdevelop.org/wikidocs/index.php?title=Main_Page>

Setup:

1. Open the project file located at the root of the SVN “Trunk” Folder
2. FlashDevelop will ask you where the open Source flex SDK is located
3. Go Tools -> Program Settings -> FlashViewer
   1. Set the “External Player” path to the path of your flash debug player
4. Go “Project” -> “OutPut” -> TestMovie and set it to “External Player” so it loads output in the external debugger
5. Try hitting run.
   1. Remember: Releasing in “Debug” mode will always produce a slower running SWF. Change the compiler option to “Release” to create a release version with better performance

# Folder Structure:

* **Root**:
  + **Box2D**: The Box2D physics engine folder
  + **Com**: The main game files
    - **Core: (**some fundamental objects relating to loading of xml and image assets)
      * **Factory**:Factories for loading levels from xml, sprite assets and sounds
      * **User:**  A user object which we use to save, load and store data between sessions such as “how many levels has this user unlocked”
      * **Util:** A utilities object that performs simple functions like finding the distance or angle between two objects or finding the centroid of a polygon
    - **Cupgame**: (unfinished mini-game that can be ignored)
    - **Danceoff**: (unfinished mini-game that can be ignored)
    - **Game** (everything game related)
      * **Blocks:** All the blocks (crates, monsters etc in the game)
      * **Fx:** special effects like particles and explosions
      * **Objects:** Badly named folder used for creating backgrounds and shadows for platforms
      * **Ship:** Holds everything relating to the pirate ship
        + **Weapon:** Holds all the cannonballs and weapons
      * **Ui:**  All the in-game ui controls such as pop ups, shot counters and so on.
    - **Giblet** (the level editor, can be ignored)
    - **Menu** (Everything menu and interface related)
      * **Controller:**  Nothing in here is really ever used, ignore it
      * **Factory:** Classes for dynamically loading menus from xml
      * **Menu:** menu objects and controls like buttons, achievements and so on
      * **Text:** a folder that holds classes for working with text, creating text
    - **Physics** (A set of objects which help make the Box2D game friendly )
      * **Blocks:** primitive physics blocks, platforms and objects from which the game is produced
      * **Controllers:** Special box2D controllers for handling magnetic effects, sticky effects, contacts between objects (E.g., how much damage a cannonball inflicts based on its force and so on)
      * **Factory:** The physics factory, we use this in every physics object to produce the underlying combination of the physics object and the graphics object as they go together
  + **Mochi:** Services relating to virtual goods, in-game store, leaderboards. Mochi API stuff.

# How The Game Works

## Loading Levels

Every level in the game is held in an xml file that looks like this:

* “Total shots” has been deprecated and no longer is used
* “Par” is how many shots can be taken before we lose
* “GridSize” is how many pixels by how many pixels each grid in the game is, everything gets locked to a grid because it was much easier to code that way in the level editor to make objects snap together and not overlap
* Platforms have two points….x1 and x2 and y1 and y2. The platform is drawn and positioned between those points
* The focus point is where the game’s camera starts out at the beginning

<level gridsize="50" par="10" totalshots="10">

<levelobject type="background"/>

<levelobject type="water"/>

<levelobject type="pirateship" x="0" y="9"/>

<levelobject type="treasure" x="7" y="8" width="1" height="1"/>

<levelobject type="objective" event="TreasureFound"/>

<levelobject type="focuspoint" x="13" y="8"/>

<levelobject type="starcrate" x="10" y="8"/>

<levelobject type="staticironplatform" x="7" y="9" x2="11" y2="9"/>

<levelobject type="staticironplatform" x="28" y="5" x2="31" y2="5"/>

<levelobject type="goldcoin" x="29" y="4"/>

</level>

**Important to notice:**

* Every treasure, every pirate, every enemy must have a corresponding “Objective” associated with it. On load, all the objectives are pushed into an array which is used to check whether the level is “over.” When the objectives array is length 0, we know the user has completed all the tasks they needed to complete

### How The Game Hooks Up With Box2D

Every object in the game has an invisible “physics object” which you can see if you enable debug draw and a “Sprite object” which is the visual manifestation of the object which the user sees. By default, debug draw is turned off.

Every object in the game is a meld between the physics object and the sprite object such that, every frame event (1/30), the position and rotation of the sprite object are matched to that of it’s physics representation.

**Some important notes on Box2D:**

* If you attempt to destroy a physics object at the wrong point in time, you will get an error. Physics objects must be destroyed / removed from the physics world immediately after the physics world performs a time-step. Prepare for horrible errors if you don’t do this.
* Platforms are generally extremely skinny. I had to do this so that it would be easy to draw exact shapes out of platforms without having them bump eachother out of the way. Also, I wanted to be able to have crates and other boxes squeeze in together. You can’t do that if you have fat platforms.

### The Physics Factory

The physics factory is where I create physics objects to match the incoming request for a game block.

### Important Notes On Flash Optimization And Performance

Performance in flash is fucking gay. The Garbage Collector never runs when it is supposed to and the only way to force the GC to run is to use a hack (calling LocalConnection() twice in a row). Objects with event listeners are kept in memory seemingly endlessly and you have to always remember to remove every single event listener and “=null” all objects when you are done using them. Even then, things still stick around and you have to use the garbage collector hack to get rid of them.

Memory is super fucking gay too. One thing I noticed is that the garbage collector performs differently in the local environment than the web environment. The garbage collector is much more aggressive when a swf is being run in a browser than it does in the local debugger. Use the flashdevelop memory profiler if you want…but I don’t think its highly accurate. IT is very useful for finding what kind of objects are lurking around and not getting disposed of but not for much else. It will show memory use piling up endlessly between level loads, even though that doesn’t seem to happen when the swf is published in a live environment.

## The Object Pool

In one of my last ditch attempts to make flash run at decent performance, I created an Object Pooling system that auto-loads a bunch of assets at start and then can be used to recycle them over and over again. It sort of works I think. I have no good way to measure its performance because the flash develop profiler doesn’t seem to work correctly. However, I did notice a performance improvement after I installed it.

## Loading and provisioning of Media Assets in Flash

All embeds of media assets occur in the “Asset Factory” class.

Here is how we embed a sound file:

[Embed(source = 'assets/library.swf', symbol = 'sfx\_seagull.mp3')]

private var \_seagull:Class;

Here is how we embed an xml file:

[Embed(source = 'assets/levels/level\_1.xml',

mimeType="application/octet-stream")]

private const \_level1:Class;

This is how we embed the library swf from where we get all the sound and graphics:

[Embed(source = 'assets/library.swf', symbol = 'Library')]

private static const \_library:Class;

Here is how we embed a custom font

[Embed(source="assets/fonts/treamd.ttf", fontFamily="TreasureMapFont")]

private var \_piratefont:String;

Because of the nature of online distribution, I have had to embed all media, sound, animations directly into the compiled swf. Flash game websites only accept a single .swf file, so it is impossible to be loading external assets in the real world…thus, the swf must bring with it everything it needs wrapped up.

Also, there are security issues involved with loading external assets. I don’t want people messing with the game levels to give themselves bonuses etc if I am attempting to sell those same bonuses as a virtual good.

All assets are embedded on compilation from a single source library.swf. When the project is compiled, all these assets are stuck in that swf along with it.

## Preloaders

The Preloader class shows a nice loading screen prior to the load of the swf. It works by some trick where it forces the Main() object into frame 2 of an animation and then stays on the embedded graphic until the frame 2 is loaded.

## Vector Graphics and Animations

For this game, I chose Vector Graphics because there are a lot of rotations going on. Also, vectors are much easier to work with in Flash than normal sprites. Sprites are ridiculously faster though . For a mobile device, we almost certainly need to use sprites or the performance is going to fail.

## Saving And Loading Data Between Sessions

Flash has a concept called “SharedObjects” which are impossible to delete cookies that people can only get rid of by using the Adobe website’s console tools usually.

I save and load user data between sessions using shared objects. More complex games may support multiple users on a single computer, but not my game. Shared objects are automatically created whenever you call = SharedObject.getLocal("data"); If that shared object already exists, then instead of creating a new one, we just wind up retrieving the data in that thing.

Shared objects look like this:

//Load the shared object

private static var so:SharedObject = SharedObject.getLocal("data");

so.data.UserData = data; //setting data in the shared object

so.flush(); //write the data to the shared object, saving it between sessions