AtariArcade SDK Quick-Start Guide

Introduction

This document provides an overview of the development mechanics, architectural concepts, and other information pertaining to developing games in HTML5 for Atari Arcade.

The first steps into development include:

1. Download the SDK, which includes code, documentation which outlines the SDK classes, and how to use them. http://atari.com/arcade/developers/downloads-documentation/
2. Read the articles on the developer center. They are chalk-full of examples and code snippets, techniques, and information on building games. A great example is the Building HTML5 Games with the Atari Arcade, which provides some helpful information, code snippets, and examples.
3. Check out CreateJS.com. Clone the GIT repositories, and dive into the examples. EaselJS contains some starter tutorials too!

4. Jump over to the forums to discuss the games and the SDK (coming soon)
5. Download Windows 8 and Internet Explorer 10 to start writing cross-browser, standards-based code.
   http://windows.microsoft.com
   http://windows.microsoft.com/ie
6. Read the Developer Submission Guide, included in GIT.

Code Overview

The Atari Arcade consists of a wrapper website, which handles a lot of the nitty-gritty of setting up and kicking off a game. The code library that comes with the SDK is used both by the website, and the game framework.

1. Framework Code

All code for the framework is contained in the **Atari** namespace (with some minor exceptions):

```
(function(scope) {
      function ClassName() {}
      ClassName.staticMethod = function() {}
      ClassName.prototype.instanceMethod = function() {}
      scope.ClassName = ClassName;
}(window.Atari))
```

Framework code can then be accessed using:

```
Atari.ClassName.staticMethod();
```

Or via an instance:

```
var instance = new Atari.ClassName();
instance.instanceMethod();
```

It's not necessary to interact with most of the framework code, but is good to get a sense of how it's written, how it works, and what's available. Here's some important ones:

- Atari.proxy: Since JavaScript has no method closure, callbacks and event handlers will often have the wrong *scope* when they fire. The proxy is a function wrapper to maintain scope. It makes your life a lot easier.

  ```
  tween.onComplete = Atari.proxy(this.handler, this);
  function handler() {
      console.log("I am in the right scope now");
  }
  ```
- Atari.developerMode: It's a good idea to remove testing code and cheats, but the developerMode flag provides a safety net for stuff that gets left in.

```
      if (Atari.developerMode) {
         this.currentLevel = 10; // Skip levels
      }
```

2. Game Libraries

These are libraries for game development, used both by the framework, and by developers making games. They reside in the **GameLibs** namespace.

```
(function(scope) {
      function LibraryName() {}
      scope.LibraryName = LibraryName;
}(window.GameLibs))
```

Game Libraries can be accessed from games or the framework using:
`GameLibs.LibraryName.method();`

It's a good idea to get familiar with the capabilities of the existing GameLibs - there's a lot of necessary and useful functionality that has been stress-tested already.

3. Game Code

Code for each game resides in the `currentGame` namespace, which is defined as a property of the **Atari** namespace.

```
(function(scope) {
      function Game() {}
      var libs = GameLibs; // Store a library reference

      Game.prototype = {
            property: null,
            method: function() {},
            initialize: function(stage, assets) {}
            callbackMethod: null
      }
      scope.Game = Game;

}(window.Atari.currentGame))
```

Game code can access all other game classes for the game, by explicitly defining the scope. For example:
`var explosion = new scope.Explosion();`

The game code is where all user-defined code for the game lives. This provides an easy way to reference and cleanup the code when the game is over, while ensuring it doesn't collide with other similarly named classes.

CreateJS Classes

CreateJS classes can be used anywhere in your game, and reside on a **createjs** namespace.

```
function drawPlayer() {
      var ss = new createjs.SpriteSheet(jsonData);
      var player = new createjs.BitmapAnimation(ss);
}
```

Defining a Game

Creating a game is pretty straightforward. Here is an overview of the steps, and some guidelines to get you started.

Game Manifests

Each game has its own JSON-formatted manifest file that provides information necessary to load and start the game, including:

- The game title
- A unique id
- The class that gets instantiated when the game starts
- Loader screen to display during preloading
- Dependencies manifest for preloading scripts. Note that scripts MUST be defined in order to be included, and must be put in the correct order to prevent dependency problems.
- Asset manifest for preloading images, audio, and data. Note that assets MUST be defined here, as it helps with preloading, and providing proper paths. The `assets` property passed to the game contains the preloaded assets, and should be used to access the assets, instead of using the asset path.
- The target framerate (fps).
- Additional ancillary data, such as if the game can be continued.
- Control and Mode information.
- Multiplayer info (not available yet, but stay tuned!).

Game Methods

Let's get into the guts of a game!

The main script for a game gets instantiated as soon as everything is loaded. The framework requires a number of methods be defined, which allow the framework to communicate with,

and control, the game code. The framework will display messaging in the console for missing methods and callbacks.

An initialize function is called once preloading has completed, and some pertinent data is passed in as arguments:

- `stage`: a reference to the EaselJS stage where all visual assets will live.
- `assets`: the assets that were preloaded (audio, images, JSON, XML, etc). Use this object to reference the actual loaded elements, rather than loading them again.
- `gameInfo`: Information about the game, with some important system information, such as if multitouch is enabled, giving you the opportunity to present touch controls to the user. Down the road, this will also contain multi-player information.

A gameStart method is called when the user clicks “start” in the arcade. This kicks off the action. Similarly, once a game is over, the user can continue or restart the game, and the website calls the `continueGame()` and `restart()` methods respectively.

As the game plays, a `tick` method is called each “frame”, which is usually when general updates to game should happen. Move some sprites, check collisions, handle user input, and run game logic. The stage is automatically updated for you afterwards. When the user pauses the game, this tick is stopped for you.

Additional methods include

- `pause(paused)`: The game has been paused or resumed - update the game appropriately. Note that the game tick and global audio is automatically paused, so this is mainly to update the visuals if required.
- `destroy()`: Clean up the game.
- `getScore()`: Provide the current level, score, and lives to the framework. A GameDetails class is provided to contain this information. This is usually called when the game ends.

Multiplayer methods are provided in the SDK, but are not currently available for third-party created games. Support is planned for the next public release of the SDK. Want a sneak peek?

- `sync(gamePacket)`: Synchronize the whole game using an update from the host.
- `updatePlayers(framePackets)`: Update all players using the packets available each tick.
- `getGamePacket()`: Get a snapshot of the entire game to synchronize other players.
- `getFramePacket()`: Get a snapshot of the current frame to be shared with other players.

Check out the `Game` class in the documentation for more info.

Callbacks/Events

A number of callbacks are supported that communicate back to the game framework. This helps externalize all the common functionality like continues, dialogs, settings, achievements, etc. The game should define a null callback, and then call it at the appropriate time.

- `onGameOver`: The player has lost.
- `onGameComplete`: The player has won. This only needs to be provided if the game is winnable.
- `onError`: An error has occured. This will provide a framework dialog.

When a game completes, it is important to stop audio. The tick will be stopped, and one additional stage update will be made.

Some additional callbacks are also available. Check out the `Game` class in the documentation for more information.

Testing

Local testing requires a few steps, but is fairly straightforward.

1. Test environment
Due to limitations in loading content locally, it is recommended that a local webserver is used for all local testing. Check out MAMP on OSX, and XAMPP on Windows and Linux. Simply point to a directory holding the site content, and load it up! Gain an understanding of what errors can be ignored when testing locally, and which are legitimate defects in code that will be thrown when hosted online.

2. Site shim
We have provided a site shim, which handles most of the steps involved with bootstrapping a game. This simulates launching your game from the Atari Arcade website. Update the game manifest to include your game, and it will preload and play with minimal effort.

3. Sample game
A sample game, “Falling Stuff” has been added to the GitHub repository, and is a great example of how a game should be structured.

The SDK Framework

The framework is defined to handle all common functionality, and let each game focus on the gameplay mechanics. This means some core concepts are provided for free:

- Start screens, including instructions and pre-roll ads.
- Preroll ads, which play at pre-defined intervals before gameplay starts.
- Asset preloading
- Pause controls and screens (including pausing the actual game tick)

- Game Over / Continue screens
- Audio muting
- Game mode selection screens (see Super Breakout as an example)

GameShell
The `GameShell` is responsible for creating the container which populates the game. Once the game manifests are loaded, an IFRAME is created. The template.html is injected, which contains all the relevant library imports, as well as a stub loader function which starts the `BootStrapper`. GameShell only lives in the parent window.

GameBootStrap
The `GameBootStrap` is responsible for preloading and initializing the game. It uses the game's manifest.xml (above) to preload all the content, instantiates the game, and then initializes it. This is initialized automagically.

GameMediator
The `GameMediator` manages communication between the game instance and the framework. The `GameMediator` has a **`command()`** method, which is mapped to functions in the game, and callbacks are injected into the game instance, which are routed back to the game shell. Games do not need to know about the `GameMediator`, only that they have to define the required methods, and call the required callbacks.

Game Libraries

It is good to familiarize yourself with the available game libraries. Here is a short overview. The API documentation includes more information about each class.

Game Libraries

- GameDetails: A wrapper for game score, levels, and lives with serialization for socket communication. An instance should be returned when the framework calls getScore.
- GameLibs: A namespace for all game libraries.
- GameUI: An interface for controlling the game UI/chrome from the game.
- Math2: Additional math functions, including some collision detection.
- Parallax: Create and control an HTML parallax effect (not in Canvas) for dynamic backgrounds.
- ParticleEmitter: Create decaying particles, emitted from a specific point.
- PerformanceMonitor: Watches the framerate, and issues a notification if it drops below a threshold. Framerate is automatically set to 30fps for low quality mode.
- Point2: Extended point APIs.
- ScoreManager: Provides score utilities like formatting, tweening, and ties into Achievements.
- StringUtils: General-use String functions.

- SpriteSheetWrapper: Utility for spritesheets.
- TexturePackerUtils: Utility for using TexturePacker with EaselJS.
- GamePad: An interface for user input from the current player. Input controls such as Joystick all pass input into the GamePad, which is translated into consistent output.

Note that the GameInfo class (listed below in multiplayer) is extremely important to get information about the current game, and will probably be moved out of multi-player in future versions.

Components
All input components shuttle user input (mouse/keyboard/touch) through the GamePad which normalizes the input, and is accessed by the games. Additionally, multiplayer commands will come through the GamePad as well.

- FPSMeter: Measure and display game framerates.
- TouchBar: Convert touch, drag, and mouse input to more general controls.
- Joystick: Convert 9-way joystick input to general controls, including on/off directional states, angles, and distance ratios.
- Throttle: A two-directional (horizontal or vertical) joystick.
- ArcadeButton: Convert on-screen button presses to more general input controls.

Multiplayer

Multiplayer libraries are largely unavailable at present, however they are used in the existing games, and some of the classes are required to run the framework. It is good to familiarize yourself with them for the impending availability of 3rd-party multiplayer games in the near future.

The site will provide the groundwork for starting games, it will be up to the game to send and receive game and frame packets.

Already-Available Multiplayer Libraries

- MultiPlayerGame: Facilitate multi-user games and communication between the socket and the GameMediator. It also handles game timers and room events.
- GameInfo: Provides an API/properties to get information about the current game.
- Player: A player instance. Includes player name, and information on if the player is the current player, a friend, or neither.
- FramePacket: Pass frame-specific data (player position, shots fired, etc.) to other players in a multi-player game.
- GamePacket: Pass an entire game state to other players in a multiplayer game. This is handled by a host game only.

Submitting a Game

Once your game is complete, read through the Game Submission Guide, provided in the GIT repository, in the docs folder. This will lead you through the final steps to submit a game to the arcade.

1. Testing
2. Preparing arcade art assets
3. Packaging and submitting

Conclusion

The AtariArcade SDK provides you with all the peripheral functionality required to build amazing games in HTML5 for the Atari Arcade. It enables developers to focus on the most enjoyable aspect of game development: the gameplay. We can't wait to see what you create!

Resources

CreateJS

- Learn more about CreateJS
- Community Forum and Feedback
- CreateJS on GitHub

More HTML5 and Flash goodness on gskinner.com

Internet Explorer 10 Guide for Developers
HTML5 Labs
W3C
Removing legacy IE Markup with Compat Inspector
Feature Detection for HTML5
IE Blog
MSDN IE DevCenter
IETestDrive
Windows 8 / Internet Explorer 10 : Touch Friendly
Windows 8 / Internet Explorer 10 : Plug-in Free
Windows 8 / Internet Explorer 10 : Pinning Sites