Pulse Getting Started Guide

This guide will aid the developer through setting up and utilizing Pulse.

Contents of the release package:

LICENSE.txt	**The license that applies to using the Pulse engine.**
RELEASE_NOTES.txt	**The release notes for the current release of pulse. Including any api changes to be aware of.**
README.pdf	**This document.**
/bin/pulse.min.js	**Minified version of the pulse library.**
/bin/pulse.js	**The full source of the pulse library. Useful for debugging.**
bin/modules/pulse.debug.min.js	**Minified version of the debug panel module. For more information on the debug panel module, check out the docs.**
bin/modules/pulse.debug.js	**The full source of the debug panel module.**
bin/modules/pulse.physics.min.js	**Minified version of the physics module. For more information on the debug panel, check out the docs.**
bin/modules/pulse.physics.js	**The full source of the physics module.**
/doc/pulse/index.html	**Auto-generated pulse source code documentation.**
/doc/modules	**Auto-generated included modules source code documentation.**

/example	Contains the source for an example game.

Installing

Include either the minified version (pulse.min.js) or the full version (pulse.js) in the head of an HTML demo file. For example:

```
<html>
  <head>
    <script type=”text/javascript” src=”pulse/build/bin/pulse.js”></script>
  </head>
</html>
```

Using Pulse

pulse Namespace

Every object within Pulse resides in the “pulse” namespace (pulse.Sprite, pulse.Layer, etc).

Ready Callback

Pulse will raise a ready callback when the DOM and Pulse are both ready for use. The game’s javascript file should use this callback as the starting point for implementation.

```
pulse.ready(function() {
  //TODO: start building a game
});
```

The Engine

The Pulse Engine object is the root class for a given game. Each Engine object is responsible for maintaining the visual state of a single game window. Websites can run multiple games simultaneously - each one will have a separate Engine object. The constructor for the Engine object requires a DOM object (or id) that will be used as the hosting element for the game. These are typically DIV elements, but can be other types if needed.

```
pulse.ready(function() {
  var engine = new pulse.Engine( { gameWindow: “myDivElementId” } );
});
```

Starting the Engine

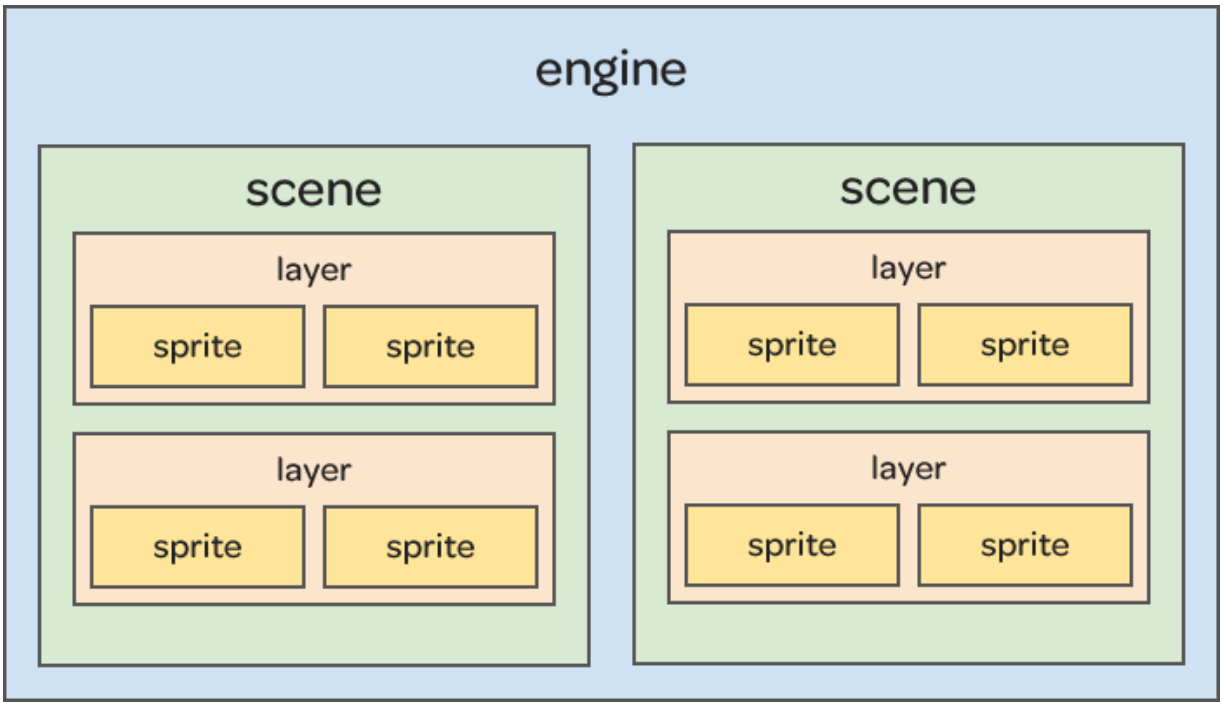
After the game has been configured and is ready to begin, call go on the Engine class. The only required argument is the frames per second the game will run at. For example:

```
pulse.ready(function() {
  //The game is all setup here.
  …
  //Run the game at 33 milliseconds between frames
  engine.go(33);
});
```

The go function also accepts an optional callback that will be invoked on each update loop. This callback can be used to control game logic, animations, etc. For sufficiently advanced games, however, it is recommended to extend the base Pulse classes.

Visual Hierarchy

Visual elements (nodes) in Pulse are organized into Layers, Scenes, and an Engine. Each game can have only one Engine. Each Engine can include multiple Scenes, however only one scene can be active at any given time. Each scene can contain multiple layers and each layer can include multiple nodes (sprites, text, etc).



The minimum amount of code required to display something to the screen is:

```
pulse.ready(function() {
  var engine = new pulse.Engine( { gameWindow: "myDivElementId" } );
  var scene = new pulse.Scene();

  engine.scenes.addScene(scene);
  engine.scenes.activateScene(scene);
```

```
  var layer = new pulse.Layer( { x : 320, y : 240 } );
  scene.addLayer(layer);

  var diamond = new pulse.Sprite({ src: 'img/diamond.png' });
  diamond.position = { x: 320, y: 240 };
  layer.addNode(diamond);

  engine.go(33);
});
```

Extending Pulse Classes

All Pulse classes are implemented using a classical approach to inheritance, which makes extending them very easy. To extend the update logic of sprite, you would use the following code:

```
var MyCustomSprite = pulse.Sprite.extend({
  //The update function is the most likely function to override.
  update: function(elapsed) {
    //Custom update logic - change the position, etc.

    //Remember to always call the base version.
    this._super(elapsed);
  }
});
```

Refer to the game demos in the pulse-demos project for more complex and practical uses of object inheritance.

The Render Loop

Each update cycle is split into two parts - update and render. The update loop is used to set the visual state of the object before it is rendered to the screen. This is where an object's new position, color, texture, etc should be assigned. The render loop will then draw the object's visual state to the screen.
Pulse includes optimizations to ensure objects are not redrawn if their visual states have not been altered.

Events

Pulse supports many types of mouse and keyboard events. Each object has an instance of EventManager ([object].events) that can be used to bind events. For example, I can bind to a keydown event using the following code:

```
scene.events.bind('keydown', function(e) {
  if(e.keyCode == 37) {
    arrowLeft = true;
  }
  if(e.keyCode == 39) {
```

```
    arrowRight = true;
  }
  if(e.keyCode == 38) {
    arrowUp = true;
  }
  if(e.keyCode == 40) {
    arrowDown = true;
  }
});
```

More Information

You can find full documentation at http://www.withpulse.com/docs/.
If, in the course of using Pulse, you find bugs or would like to see a new feature, please send specific details to support@withpulse.com.