Bemuse Documentation

Release 37.0.1

The Bemuse team

Mar 25, 2017

Contents

Bemuse is a BMS player with a gameplay similar to Lunatic Rave 2.

Bemuse is web-based, which means that you don't have to download any special software to play this game; simply use your web browser.

Bemuse is built on new, cutting-edge HTML5 technologies. It does not require any plugin (like Flash or Unity Web Player). This game can be played on browsers that implement these technologies (such as Firefox, Google Chrome, and Safari).

Part I

User Documentation

CHAPTER 1

Gameplay

How do I adjust the speed?

To set the speed in-game, use the Up and Down buttons on your keyboard. This adjusts the speed by 0.5x. If you hold down Alt, the speed is adjusted by 0.1x instead.

There is a huge audio latency! Can it be fixed?

As a web-based game, it does not have low-level access to the audio driver. There are many factors that affect the audio latency in this game (and web-based audio applications in general):

- The browser's implementation of the Web Audio technology
- Your sound card driver and settings on your operating system

Bemuse has a mechanism for delay compensation. Simply open the options screen and enter your system's audio+input latency. You can click on the "Calibrate" button to find out your system's audio+input latency.

From my experience, Safari has the lowest latency so far.

CHAPTER 2

Scoring and Judgment

The scoring system and judgment system in Bemuse focuses on both accuracy and combo.

Judgment

When hitting the note, the accuracy of your button press will be judged according to this table:

Name	Maximum Offset (ms)	Accuracy Score
METICULOUS!	20	100%
PRECISE!	50	80%
GOOD!	100	50%
OFFBEAT!	200	0%
MISSED!	–	0%

Source code reference

- Defined at src/game/judgments.js:7-14 (judgment timegate)
- Defined at src/game/judgments.js:51-58 (judgment weight)

Scoring

The player's score is calculated from this formula:

$$\text{score} = 500000 \times \text{accuracy} + 55555 \times \text{combo bonus}$$

$$\text{accuracy} = \frac{\sum \text{accuracy score}}{\sum \text{total combos}}$$

$$\text{combo bonus} = \frac{\sum_{c \in \text{combos}} \text{combo level}(c)}{\sum_{i=1}^{\text{total combos}} \text{combo level}(i)}$$

$$\text{combo level}(c) = \begin{cases} 0 & c = 0 \\ 1 & 1 \le c \le 22 \\ 2 & 23 \le c \le 50 \\ 3 & 51 \le c \le 91 \\ 4 & 92 \le c \le 160 \\ 6 & 161 \le c \end{cases}$$

Here's how the combo level formula comes from. Let's assume, for simplicity, a player with 99% hit rate, regardless of difficulty. The probability that the player will attain c combos is 0.99^c.

Now we have 6 combo levels. The probability that the player will attain that level gradually decreases. Therefore, the minimum combo is $\lceil \log_{0.99} p \rceil$.

Combo Level	Max. Probability	Min. Combo
1	100%	1
2	80%	23
3	60%	51
4	40%	92
5	20%	161

Source code reference

- Defined at src/game/state/player-stats.js:27-29 (score) [outdated: 17be8477]
- Defined at src/game/state/player-stats.js:99-106 (combo)

Grading

After playing the game, the grade is calculated according to this table:

Grade	Minimum Score
F	0
D	300000
C	350000
B	400000
A	450000
S	500000

Source code reference

- Defined at src/rules/grade.js:2-12 (grade)

CHAPTER 3

Running Your Own Music Server

> **Warning:** This section is under construction.

Bemuse comes with a default music server to help new players get started. This default music server contains a selection of songs that I think are really nice. You can also run your own music server and play it in Bemuse. This page describes how you can do it. This page assumes some knowledge about using the command line and web hosting.

A Music Server

A music server is simply a web server that hosts the files in a specific structure, which allows Bemuse to find the list of songs and the song data. A Bemuse music server has the following structure:

- (root of the server)
  - index.json
  - song1/
    - bms1.bme
    - bms2.bml
    - assets/
      - metadata.json
      - (something).bemuse

index.json

This file holds the list of all available songs and charts in this server. It also includes some metadata information.

When entering the game, Bemuse will download this file to create the song list that you see in the music selection screen.

Song directory

Besides the `index.json` file is a song directory. This directory contains the BMS files and the assets folder, a **Bemuse assets package**.

This file is generated using the Bemuse tools, which we will cover in the next section.

Bemuse assets package

Usually, a BMS package will come with hundreds (or even thousands) of sound files (the keysounds). It is not suitable for serving over the web. Sometimes, they are `.wav` files and usually, they are `.ogg` files. Wave files are too large, and not all browsers can play OGG files.

A Bemuse asset package contains the keysounds in OGG and M4A format, because most browsers can play these file formats. These sound files are grouped together into multiple parts. Each part is approximately 1.4 MB large.

The Bemuse assets package is also generated using the Bemuse tools, which we will cover in the next section.

Prerequisites

Bemuse tools work on Windows and Mac OS X. Before that, you need to install:

Node.js

Bemuse tools is written in JavaScript for the Node.js platform and distributed through npm. First, you need to install Node.js, which also includes npm.

SoX

SoX is a tool to process and convert audio files. Bemuse tools uses it to convert the keysounds into a web-friendly format.

Windows users should list SoX in their System PATH by going to Control Panel>System and Security>System>Advanced System Setitngs>Environment Variables or by setting the PATH variable in CMD. Please see TechNet documentation for more info: https://technet.microsoft.com/en-us/library/bb490963.aspx?f=255&MSPPError=-2147217396

QuickTime and qaac (Windows only)

Mac OS X already comes with QuickTime and a command-line M4A encoder. Windows, on the other hand, does not. Therefore, Windows users needs to install qaac. It is a command to convert audio files into MP4 audio (AAC) for browsers that doesn't support OGG. Currently, Bemuse Tools looks for the 32-bit executable for qaac; don't just install the 64-bit executable.

Windows users should list qaac in their System PATH by going to Control Panel>System and Security>System>Advanced System Setitngs>Environment Variables or by setting the PATH variable in CMD. Please see TechNet documentation for more info: https://technet.microsoft.com/en-us/library/bb490963.aspx?f=255&MSPPError=-2147217396

Installing Bemuse Tools

Bemuse tools is distributed via npm. To install Bemuse tools, use the following command:

```
npm install -g bemuse-tools
```

If you get a permission error on Windows, try running the above command in an Administrator command prompt.

If you get a permission error on Mac OS X, try:

```
sudo npm install -g bemuse-tools
```

Using Bemuse Tools

First, you will want to create a folder to hold your music server.

TODO: More detailed instructions

Generating Bemuse assets package

Use this command:

```
bemuse-tools pack songdir
```

Generating index file

Use this command:

```
bemuse-tools index
```

Hosting

On Dropbox

TODO

On a Web Server

Upload the files to the web server. Also set up CORS on the server to allow Bemuse to access.

Direct players to your server with the following url: http://bemuse.ninja/?server={[}SERVER URL] **TODO**: More detailed instructions

CHAPTER 4

BMS Support in Bemuse

Bemuse supports BMS, BME and BML, but some features are not supported. They are noted here.

BMS

- BGA is not supported yet.
- Free-zone not supported.
- Invisible objects not yet supported.

BML

- Supports #LNTYPE 1 (RDM; loose BML) and #LNOBJ (RDM type 2; strict BML)
- Long notes are judged both at the start and at the end (2 judgments per long note, similar to O2Jam).
- Player must release the button at the end of long notes. Otherwise, the end of long note will be missed.
  - **Exception:** Player doesn't have to stop spinning the turntable at the end of SCRATCH notes. If the player keeps spinning, player will get METICULOUS (Perfect) judgment.

PLAYER

As Bemuse is currently a single-player game, only #PLAYER 1 is supported.

RANK

- Bemuse does not support #RANK.
- Judgment timegate is described at **'scoring and judgment'_** section.
- Bemuse judge notes per unit time.

TOTAL

Bemuse does not support #TOTAL.

DIFFICULTY

- BMS charts with #DIFFICULTY 1-4 are treated as the same (non-insane).
- BMS charts with #DIFFICULTY 5 are treated as BMS (INSANE chart). In music selection screen, it is displayed in different color.

TITLE and SUBTITLE

- Supports implicit subtitles.
- Supports multiple subtitles.

ARTIST and SUBARTIST

- Supports multiple subartists.

Landmine

Not supported (in the future, it will display as fake note).

WAV

- Supports OGG, M4A, MP3, WAV samples.
- Polyphony of #WAV is 1, consistent with major BMS implementations.
- However, it does not apply to sounds played when player hits the button without note (freestyle sound).
  - Sound of the nearby note will play with no polyphony limit.
- If player gets OFFBEAT (bad) judgment, the sound will play in wrong pitch.

RANDOM

- Only #RANDOM, #IF, #ENDIF in original specification are supported. However, they can be nested (nesting level can be ended with #ENDRANDOM).

CHAPTER 5

Bemuse's BMS Extensions

Base36 Channels

Most BMS players support Base10 (00-99) and Base16 (00-FF) channels. I don't want to deal with number clashes in the future, so I took the liberty to extend the support for channel numbers into Base36 (00-ZZ).

This allows more meaningful channel names, such as `#xxxSC` for scrolling factor and `#xxxSP` for spacing factor.

Speed and Scroll Segments

```
#SCROLL01 1
#SCROLL02 0.5
#SPEED01 1
#SPEED02 0.5
#001SC:01020102
#001SP:01020102
```

History

Speed and scroll segments are introduced in StepMania 5 to dynamically change notes spacing and notes scrolling speed. This allows scrolling effects to be created without relying on BPM changes or STOPs.

Extension Lines

```
#EXT #xxxyy:....
```

Part II

Developer Documentation

CHAPTER 6

Getting Started

This page describes how to setup the project on your computer for local development.

Prerequisites

- Git
- Node.js v0.12
- Text Editor with EditorConfig support.

Setting Up the Project

Note: For Windows users, there (will be) a dedicated quick-start guide.

First, you should create a folder for Bemuse development:

```
mkdir Bemuse
cd Bemuse
```

Then, clone the Bemuse repository and music repository:

```
git clone git@github.com:bemusic/bemuse.git
git clone git@github.com:bemusic/music.git
```

After these repository has been cloned, `cd` into the Bemuse repository:

```
cd bemuse
```

Install the project's dependencies:

```
npm install
```

When it finishes installing, start the development server:

```
npm start
```

The game should be accessible at `http://localhost:8080/?mode=app`.

To run unit tests, go to `http://localhost:8080/?mode=test`.

Coverage Mode

We measure the code coverage to make sure that most part of our code is covered by some test. This helps us be more confident in modifying our code.

To turn on the coverage mode, start the server with the `BEMUSE_COV` environment variable set to `true`:

```
BEMUSE_COV=true npm start
```

Then run the unit tests. After the unit tests are run, the coverage report will be generated. They can be viewed at `http://localhost:8080/coverage/`.

Building

To build the source code into a static web application, run:

```
npm run build
```

The built files will reside in the `build` directory.

Running Tests from Command Line

You can run tests from the command line by running:

```
npm test
```

This will effectively

1. build Bemuse with coverage mode turned on,

2. start a web server,
3. start a web browser and navigate to the test page, effectively running the tests,
4. collect the results and code coverage and write reports.

CHAPTER 7

Getting Started (Windows)

This setup guide is based on Windows 8.1. This guide assumes some command-line knowledge.

Installing Prerequisites

GitHub for Windows

GitHub for Windows makes it easy to get started using Git and GitHub on Windows. Download GitHub for Windows from https://windows.github.com/. Install it and sign in using your GitHub account.

Chocolatey

Chocolatey lets you install applications from the Command Prompt. To install Chocolatey, visit https://chocolatey.org/.

Node.js

Node.js is a JavaScript runtime outside the browser. We use it to perform build tasks (such as compiling the source codes, running tests, static analysis, and performing deployment tasks). Open Command Prompt as Administrator and install Node.js using:

```
choco install nodejs.install
```

Install Git for Windows

GitHub for Windows comes with PowerShell-based Git Shell, only available on Windows.

However, Bemuse is developed on multiple platforms, so we prefer to use a shell that can be used on all platforms. One of it is the "bash" shell. The easiest way to install a bash shell is to install Git for Windows:

```
choco install git.install
```

Install Google Chrome

Google Chrome is used for automated testing, so make sure you have it installed:

```
choco install google-chrome-x64
```

Restart Your Computer

These tools modify your system `PATH` variable. They need to be reloaded. One of the easiest and most reliable way to do it is to restart your computer.

Setting Up the Project

Create an empty folder for the Bemuse project, then clone the following repositories into that folder using GitHub for Windows:

bemusic/bemuse The game repository.

bemusic/music The music repository. This repository is huge, so it will take a while.

After cloning them, right click the `bemuse` project folder and select "Git Bash here."

Install `npm`

Even though npm is already installed with Node.js, the installer is buggy, causing npm not to run. To fix, use `npm` to install itself:

```
npm install npm -g
```

Install Project Dependencies

The Bemuse project depends on hundreds of other free software projects. To install them, use the `npm` command line tool to install:

```
npm install
```

Start the Development Server

To start the development server, type in:

```
npm start
```

CHAPTER 8

Project Architecture

This section describes the architecture of the project.

Directory Structure

bin Useful scripts for routine work. Examples include setting up Git commit hooks and releasing a new version.

config Configuration code for webpack and other things.

docs This documentation.

public Files that will be deployed verbatim to the server, **except for** `index.html`, where the boot script will be inlined. These include skin files.

spec Contains the unit tests. Its directory structure resembles the `src` directory.

src Contains the production code. Code is split into *modules* for different parts of the application.

tasks Gulp tasks to run test server, build, test the application.

Important Modules

These modules live in the `src` directory. There may be an arbitrary number of modules. Therefore, this section only lists the significant modules.

boot This module is the entry point to Bemuse. It reads the `?mode=` parameter and determines the name of the main module to load. It then displays a loading indicator and loads the main module asynchronously. After the main module is downloaded, finally, it is executed. Main modules include `app`, the game, and `test`, the unit tests. Upon building, the boot script will be inlined into `index.html`.

Rationale: No one likes blank white page. We want the user to see the application starting up as soon as possible, even though it is simply a loading indicator. To make this *blazingly fast*, we keep the compiled size of the `boot` very small, and inline that compiled code directly into the HTML file. So, no round-trip HTML requests! If they can load the HTML, they *will* see the loading bar.

app This is the main module of the game's application flow. Executing this module will present the game's main menu.

test This is the main module for unit tests. Executing this module will setup the environment for testing, load the unit tests in `spec` directory, and then execute them. After the test is run, the results and coverage data (if available) will be sent back to the server for further processing.

game This module contains the actual game part. For example, the logic for judging notes, calculating score, and rendering the scene.

Related Projects

Apart from the `bemuse` project, we also maintain other closely-related projects in a separate repository.

bemusic/bms-js This project is a BMS parser written in JavaScript. It is written in plain ES5 for maximum portability.

bemusic/bmspec This project is an executable specification of the BMS file format. It is used to make sure that bms-js can properly parse BMS file format, especially the edge cases.

bemusic/pack This repository contains the code needed to convert a BMS package into a Bemuse package. Traditional BMS packages are optimized for offline playing. They are distributed as a large `.zip` file with `.wav`, `.mpg`, and `.bms` files. This is not suitable for web consumption. See bemusic/pack for more information.

CHAPTER 9

Documentation

To keep the codebase and documentation in sync, we keep the documentation in 2 places:

1. Inside the `docs` folder.
2. Inside the code.

Documentation System

We use Sphinx to generate the project's documentation. Documentation building and hosting is provided by Read the Docs.

Generating Documentations Locally

Install Sphinx using:

```
pip install sphinx
```

Inside the `docs` folder, type in:

```
make html
```

This will generate the HTML documentations in `_build/html`.

Writing Documentation Pages

To create a new documentation page, create an `.rst` file in the appropriate place, and then reference the `.rst` file inside `README.rst`'s toctree.

Writing Documentation In Code

This project is written in ES6, and we don't have an adequate documentation tool yet. Some tools don't work with ES6 syntax, or some tools requires us to be too verbose and repetitive (making documentation a boring and error-prone process).

Therefore, this project invents its own code documentation tool which integrates nicely with Sphinx.

Source code files that matches the pattern `src/**/*.js` or `docs/**/*.py` will be processed to find documentation comments.

To write a block of documentation inside code (called codedoc), start a block of inline comment with `>> name`:

```
// >> game/input
// Documentation text goes here.
```

To include that codedoc, use the `codedoc` directive:

```
.. codedoc:: game/input
```

The previous codedoc can be added later by starting a block of inline comment with `>>`:

```
// >> game/input
// Some documentation...

... /* code */ ...

// >>
// Some more documentation...
```

Automatically-Generated Module Documentation

To facilitate documenting the game internals, documentation for each ES6 module is generated.

If a block of inline comment starts with `:doc:`, then the comment will be added into the module's documentation.

If a method definition syntax is found directly after a block of comment, then it will be used for documenting that method:

```
// Reports the progress.
report(current, total, extra) {
```

If a getter definition syntax or an assignment to `this` is found after a comment block, then the comment block documents that attribute:

```
constructor() {
  // ``true`` if the game is started, ``false`` otherwise.
  this.started = false
}

// The song duration in seconds.
get duration() {
```

If `export let` is found after a comment block, then the comment block documents that module export:

```
// The global SceneManager instance.
export let instance = new SceneManager()
```

If `export function` is found after a comment block, then the comment block documents that module export:

```
// The global SceneManager instance.
export function download() {
```

How It Works

I think from the outside, this system looks pretty cool, but from the inside, this is a dirty hack.

The code documentation and module documentations are generated by a Python script, `generate.py`, which scans the source files for comment blocks followed by the code that matches the regular expression.

This is simple and works very well, but the code must be written according to the coding convention.

CHAPTER 10

The Game Loop

Turn-Based Update Cycle

At each iteration of the game loop, each game component takes turn and update itself. Each game component involved in this game loop should have a `update(...)` method, which takes care of updating itself. This is the only time the component will be mutable.

Outside of the `update` method, a component should behave like an immutable object. This allows us to have some sense of immutability without having to create new objects. See the case for immutability.

At each cycle, the following happens:

- the Clock is updated to get the high-accuracy time
- the Timer is updated to get the in-game time
- the Input is updated to get button presses
- the State is updated to react to button presses – judging notes and updating scores
- the Audio is updated to emit sound based on the updated state
- the Display is updated to render the game display based on the updated state

CHAPTER 11

Skinning

To make it easy to adjust the gameplay screen's appearance, the *skin* is not built into the game's source code.

Location

The skin is located at the public/skins/default folder. Inside, you will see several files.

`skin.xml` This is the skin file that will be loaded by Bemuse. **Do not edit this file,** since this file has been generated from the skin template.

`skin_template.jade` This is the skin template that is used to generate `skin.xml`. It is written in Jade language.

`skin_data.yml` A YAML file describing global variables available to the skin template.

`gulpfile.js` A Gulp script to compile the template into `skin.xml`.

`*/*.png` Image assets.

Skin Development

Make sure you have already set up the project and started the local development server.

Install Gulp globally, so that you can invoke it directly from the command line:

```
npm install -g gulp
```

In another terminal window, **cd** to the skin's directory:

```
cd public/skins/default
```

Then run Gulp:

```
gulp
```

This will compile the skin into `skin.xml`. If you change `skin_template.jade` or `skin_data.yml`, the skin is recompiled. Now, you can refresh the browser to see the changes.

Skin Elements

TODO: generate documentation

CHAPTER 12

Game Options and Configuration

This page lists various options for configuring game objects.

Options for Loading Game

bms A `Resource` object for the BMS file to play.

assets A `Resources` object that contains the BMS resources needed by the BMS file: keysounds, images, and movies, for example.

metadata An `Object` containing *song metadata* (page 37).

options An `Object` containing *game options* (page 38).

Song Metadata

title A `String` representing the song title.

artist A `String` representing the song's artist.

genre A `String` representing the song's genre.

subtitles A `Array` of `String` representing the song's subtitles.

subartists A `Array` of `String` representing the song's sub-artists. This may include BGA maker and note charter.

volwav The playback volume of the BMS. This metadata overrides the BMS #VOLWAV header.

Game Options

players An `array` of *player options* (page 38).

Player Options

speed The speed effector.

CHAPTER 13

Modules Index

boot

Bemuse's **main** entry point. We need this file to load as soon as possible, therefore, we minimize the amount of third-party dependencies.

The Booting Process

After the `boot` script has been loaded, the main script is scanned from the `mode` query parameter.

The main script is then loaded and imported into the environment, and its `main()` method is invoked.

Available Modes

The available modes are specified in boot/loader.js.

app The main game application. This will bring up the Title Screen.

music The music collection viewer which shows all the songs.

test The unit tests.

comingSoon Displays the "coming soon" text.

sync Displays a simple UI for determining your computer's audio+input latency.

game Runs the game shell.

playground Various playgrounds...

bootstrap

Bootstraps the environment with:

- 'babel-polyfill'_
- debug
- Bluebird (with extended Promise API)

game/clock

class `Clock()`
The game clock provides a high-accuracy time source for the game.

We want our timing to be based on audio. If the audio lags, the time clock should lag with the audio as well, in order to keep the game time in sync with the background audio.

But there is a problem on some browsers (Android): `AudioContext.currentTime` is not precise. Therefore, we take the average of the offset between system time (which is more precise) and the audio time to compute a high-precision-and-accuracy time.

`time`
The clock's time value, in seconds.

game/game

class `Game()`
The Game model holds the game's options and the player objects. This class represents all the data needed to start a game. However, it is immutable. See `GameState()` for mutable stuff.

`options`
The Game's options

`players`
The Game's players

game/game-controller

class `GameController()`
The GameController takes care of communications between each game component, and takes care of the Game loop.

game/game-timer

class `GameTimer()`
The game timer keeps track of song progression in-game. This class should be tied to the AudioContext.

time
The time, in seconds, since the start of the game.

readyFraction
The number in range 0...1 representing the progression in this second. When player presses the "Start" button, the game will wait for the next second before actually starting the game. Since the clock is synchronized globally, if player presses start in the same second, the game will start at the same time.

game/player

class `Player()`
The object representing the player's information, notechart and options.

game/state/player-state

class `PlayerState()`
The PlayerState class holds a single player's state, including the stats (score, current combo, maximum combo).

stats
The PlayerStats object.

notifications
The notifications from the previous update.

speed
The current note scrolling speed.

finished
`true` if finished playing, `false` otherwise.

omni-input

class `OmniInput()`
Public: OmniInput is a poll-based class that handles the key-pressed state of multiple inputs.

Each call to `update()` returns a mapping of active inputs. This object may be reused and mutated by OmniInput for performance reasons, and thus the caller SHOULD NOT hold on to or mutate it.

Keys

The key is a string that identifies some key on each input. It follows the following convention:

- `65` A keyboard keycode.
- `gamepad.0.button.3` A gamepad button.
- `gamepad.0.axis.3.positive` A gamepad axis (positive value).

- `gamepad.0.axis.3.negative` A gamepad axis (negative value).
- `midi.[id].[channel].note.[note]` A MIDI note.
- `midi.[id].[channel].sustain` Sustain pedal.
- `midi.[id].[channel].mod` Modulation level.
- `midi.[id].[channel].pitch.up` Pitch bend (up).
- `midi.[id].[channel].pitch.down` Pitch bend (down).

progress

class `Progress`()

The Progress class represents the progress of an asynchronous operation. It is inspired by C#'s IProgress interface.

current

The current progress (out of `Progress#total`) as a Number.

total

The maximum value of this Progress as a Number.

extra

Some arbitrary information associated with this Progress.

formatter

The formatter of this progress. This formatter will be used to compute the text representation of this progress (`Progress#toString`).

progress/formatters

BYTES_FORMATTER

Formats the `Progress` as bytes (e.g. 1.23mb of 1.31mb).

PERCENTAGE_FORMATTER

Formats the `Progress` as percentage.

EXTRA_FORMATTER

Formats the `Progress` simply by using the value of `Progress#extra`.

sampling-master

class `SamplingMaster`()

The sampling master is a wrapper class around Web Audio API that takes care of:

- Decoding audio from an ArrayBuffer or Blob (resulting in a "Sample").
- Playing the `Sample` and managing its lifecycle.

class `SoundGroup`()

Sound group

class `Sample`()
The Sample is created by and belongs to the `SamplingMaster`.

You don't invoke this constructor directly; it is invoked by `SamplingMaster#create`.

class `PlayInstance`()
When a `Sample` is played, a PlayInstance is created. A PlayInstance may not be reused; after the sound finishes playing, you have to invoke `Sample#play` again.

You don't invoke this constructor directly; it is invoked by `Sample#play`.

scene-manager

class `SceneManager`()
The SceneManager takes care of managing the scenes in this game. Only a single scene may be displayed at any given time, but a scene may contain any number of UI elements.

A scene is a React component that renders a <Scene>:

```
export default React.createClass({
  render() {
    return <Scene>contents</Scene>
  }
})
```

Behind the scene, are using some advanced techniques which requires the scene class to be declared using `React.createClass` (not by extending `React.Component`)

To use the SceneManager, get the instance and use it:

```
import SCENE_MANAGER from 'bemuse/scene-manager'
import TitleScene from './TitleScene'
SCENE_MANAGER.display(<TitleScene />)
```

Non-React Scene You can also use a scene that is not React-based. Define your scene as a function that accepts an HTMLElement and returns an object with a `teardown()` method.

instance
The shared SceneManager instance.

utils/callbacks

class `Callbacks`()
A utility class to hold a collection of callbacks.

utils/now

High-accuracy timer, optionally synchronized globally.

utils/query

This module exports the query string in browser.

Example:

```
import query from 'bemuse/utils/query'
alert(query.mode)
```

Index