Booty5 HTML5 Game Maker Manual v1.0

Developed and maintained by Mat Hopwood of Booty5 (http://booty5.com)

This document is protected under copyright to Mat Hopwood @2015.

CREATE GAMES & INTERACTIVE CONTENT
CREATE FLASH STYLE ANIMATIONS
FOR DESKTOP AND MOBILE
FOR FREE!
BOOTY
5

Table of Contents

Booty5

What is Booty5?

Booty5 is a combination of WYSIWYG game editor and game engine that targets the HTML5 platform for desktops such as Windows, Mac and Linux and mobile devices such as iOS, Android, Windows Phone 8, Blackberry and more. Booty5 enables rapid game development and testing using an intuitive super quick WYSIWYG interface that can easily handle small to super massive projects.

The Booty5 engine is an evolving open source JavaScript SDK that enables developers to create games and apps using tried and tested industry standard game programming techniques that are widely used across the industry today.

Booty5 is currently maintained on Github at https://github.com/mrmop/Booty5

Support is provided on Facebook, Google+ and Twitter, see here for more details.

Booty5 game maker editor features include:

- Create and organise game levels / maps and app layouts into scenes and actors (smart sprites)
- Create and preview complex Flash style timeline based animations of entire scenes and their game objects
- Create game logic using action lists that are presented in human readable format
- Complete game and app creators integrated development environment (IDE)
- Export in multi-resolution friendly format, allowing exported data to be used on any sized display
- Assisted layout editing, including tools to enable easy layout / layout management, bookmarking, edge / vertex snapping, directional cloning and so on
- Full drag and drop support, drop entire folders of resources onto the editor and it automatically sorts them all for you
- Support for import of SVG, Texture Packer, audio and other formats
- Support for physics including materials, shapes, fixtures, joints and the ability to test physics
- Support for Javascript and other language editing, includes syntax highlighting, code folding and search / replace
- Create and edit shapes, trace bitmaps to optimised shapes and split concave polygons to convex
- Create and edit gradients then assign to game objects
- Interactive play mode that launches the game from the editor
- Create complete working / runnable projects right in the editor, no back-end services required
- Support for bitmap animation
- Support for user properties
- Exports JSON so output can be used by any game engine that can read JSON
- Export to Marmalade Web and CocoonJS, also tested with the Intel XDK

Booty5 game engine features include:

- Free and open source
- Its tiny and fast, under 120k! (under 90k without pre-defined action)
- Support for mobile and desktop
- Global and local resource management
- Scene and Actor (sprite game object) management / scene graph
- Particle systems
- Animation via Timeline and tweening
- Support for action lists
- Image, text and geometric shape rendering, including rounded rects
- Physics using Box2D via Box2DWeb including multiple fixtures, joints, materials and shapes
- Sprite atlas and frame based bitmap animation
- Game object docking to scene edges and other game objects
- Scene and game object clipping, including to circular and polygon shapes
- Scene and game object touch detection
- Scene cameras and touch panning
- Scene and actor local layering
- Image and gradient brushes, shadows, composite operations
- 3D sprite depth
- Touch event handlers
- Keyboard support
- 2D canvas
- Audio play back
- Support for automatic scaling / resizing to fit different sized gaming displays
- Support for cached rendering to speed up shape / gradient / font rendering etc..
- Support for Booty5 game Editor / IDE

Future planned features include:

- Tile map editor
- More HTML5 specific editing options
- Export to different game engines such as Phaser and Cocos2d-js

Installing Booty5 Game Maker

Firstly, download the latest version of the Booty5 Game Maker from http://booty5.com/download-booty5/ (Windows only)

Unzip the archive and run setup.exe to install.

You should also consider downloading the example projects that accompany the Booty5 game engine from https://github.com/mrmop/Booty5.

To open an example, run the Booty5 game maker. Click the open button then select one of the example projects project XML file and open. Hit the test button run.

Usage Rights and Warranties

Copyright @2015 Mat Hopwood. All rights reserved.

Booty5 game maker and Booty5 engine and all associated data and components (collectively known as Booty5) are provided “as is” and “without” any form of warranty. Yours, your employers, your companies, company employees, your clients use of Booty5 is completely at your own risk. We are not obligated to provide any kind of support in any shape or form.

You are free to use Booty5 in your projects in part or in whole as long as all copyright notifications in any files, applications or data remain in-tact.

You may not claim Booty5 or its documentation as your own work or package it up and include it in any kind of middleware product without express prior written notice from Mat Hopwood.

Please note that these licensing terms apply to all versions of Booty5.

Note that Booty5 can utilise Box2DWeb, so please take note of this license also.

Brief How To on Booty5

At this stage I think its important for you to have a basic understanding of how to use Booty5 as this will help you to better understand the rest of this material.

The basic purpose of Booty5 is to enable developers to get a game up and running quickly, with Booty5 taking care of all the basic and mundane tasks.

The easiest way to use Booty5 by far is to use the provided game editor as this will enable you to build much of your game using a point, click and drag interface, which in most cases is much faster than programming and organising / managing your own data for game levels and the like.

If you decide to bypass the game editor then to get up and running requires only a simple text editor and basic knowledge of HTML and JavaScript, although Notepad++ is a much better text editor. However I recommend you take a look at Jetbrains WebStorm.

To include and boot the Booty5 game engine simply make the following changes to your index.html, if you plan on using the Booty5 game maker exclusively then you can skip the rest of this section:

Include the Booty5 game engine into the head section:

```
<script src='lib/engine/booty5_min.js'></script>
```

Add an HTML5 canvas to the body section that will show the app:

```
<canvas id='gamecanvas' width='1024' height='768'>
</canvas>
```

Add a window onload handler to boot the engine when the page finishes loading:

```
window.onload = function()
{
    // Create the Booty5 app
    var app = new b5.App(document.getElementById('gamecanvas'));
    b5.app = app;
    app.debug = false;
    app.setCanvasScalingMethod(b5.App.FitBest);

    // Start game
    app.start();
};
```

Of course you will not see anything on the display because you do not currently have a scene or any game objects in there. Here is small example that shows how to create a scene and add a simple game object:

```
// Create a scene
var scene = new b5.Scene();
// Add scene to the app
app.addScene(scene);
```

```
// Create an arc actor
var actor = new b5.ArcActor();
actor.fill_style = "rgb(80,80,255)";
actor.x = 100;
actor.y = 100;
actor.vx = 100;
actor.radius = 100;

// Add the actor to the scene
scene.addActor(actor);
```

And to make the actor do something we can attach an onTick event handler:

```
// Attach an OnTick handler which gets called each frame
actor.onTick = function (dt) {
    if (this.x > 200) this.x = -200;
    this.y = Math.sin(this.frame_count / 20) * 100 + 100;
}
```

Booty5 Core Concepts

In order to facilitate easier explanation of the workings of Booty its essential to give an overview of various concepts that Booty5 utilises.

Canvas

The Canvas is the HTML5 canvas that Booty5 targets for rendering. Currently the Booty5 game engine targets HTML5 canvas only (no WebGL). Future versions of the Booty5 game maker will offer support for other game engines such as Phaser and Cocos2d-js

App, Scenes & Actors

Booty5 uses an hierarchical tree based scene graph to organise, update and display game objects.

The App is the top level container object that is responsible for keeping the game logic up to date, drawing objects to the canvas, monitoring device input, running and rendering scenes etc..

Scenes are containers for actors and are used to separate app functionality into parts. For example, your game could use a scene for each game level that it contains, as well as additional scenes for game HUD overlays, a main menu screen and so on. You can think of a Scene as a game level wide container

Actors represent the individual game objects that make up your game, for example a sprite based actor may represent the main game character, whilst other sprite based actors represent the baddies and the bullets etc.. Like scenes, actors also contain their own hierarchical tree, which can be used to manage sub actors.

When exporting your game from the game editor the canvas and app is automatically created for you. You can create scenes and actors from code or more intuitively using the game editor.

Animation

Animation is a huge part of Booty5 because animation is such a large part of making games. Booty5 supports animation in different ways including:

- Timelines – A timeline is a collection of animation key frames for the various properties of objects such as position, scale and rotation that change over time
- Bitmap animation – Using image atlases the bitmap that represents a game object can be changed to show frame by frame animation
- Physics - Using arcade style physics or full on physics simulation using Box2D, objects can be set up to travel / rotate and interact with the environment

You can create animations either in code or in the game editor. The game editor supports a Flash style timeline animation system based on key frames and tweening.

Actions

Action lists offer a way of adding game logic to scenes and actors using bolt together building block style actions. An action is a piece of functionality that can be added to an object to modify and extend its behaviour. An actions list is a collection of such actions that are executed sequentially / concurrently. Action lists can be added in code or in the game editor. The game editor supports the creation of action lists using human readable action commands, making it easy for none programmers to modify game object behaviour.

Events

Many of the objects defined in the Booty5 game engine use event handlers to react to the numerous events that can take place during a game. For example the main App responds to touch / mouse / keyboard input using events and game objects respond to being updated, touched, collided with etc.. Each event can have an event handler associated with it which will respond to the event taking place. For example, you may attach an OnTick event handler to an actor which will do something with the actor every time the actors logic loop is updated.

The Booty5 game editor makes responding to events very easy by exposing all available events for an object and allowing you to define what happens in response.

Resources

Booty5 supports a multitude of different types of resources from sounds and bitmaps to shapes and brushes. Resources can have local or global scope, resources that are global are accessible to all scenes and objects within the game and will remain in memory until the game is shut down or manually removed. Local resources are local to the scene that contains them and are generally only accessible to the scene and the actors that it contains. When a scene is loaded all of its local resources will be loaded, when a scene is destroyed all of its resources will also be destroyed. This type of resource management system is useful when dealing with games that run on resource constrained devices such as mobile phones.

Note that two resources of the same type should not share the same name within the same scene.

Namespace

All Booty5 engine classes / code are placed within the b5 namespace. For example a Scene is accessed via b5.Scene instead of Scene, for example:

```
var scene = new b5.Scene();
```

Examples and Tutorials

A large number of examples are available to download on Github. You can see these examples in action online at the Booty5 web site. A number of walk through style tutorials can also be found here.

Known Issues

- Sound resources cannot currently be preloaded on some versions of iOS

Booty5 Game Engine

The App – The Eye in the Sky

Introduction

The App object is basically the great eye-in-the-sky controller of the game engine that controls the game logic, rendering update and handling of various events such as input. The App takes care of many things including:

- Manages global resources
- Manages a collection of Scenes
- Manages a collection of Action Lists
- Handles touch input and keyboard
- Finds which Actor was touched when user touches screen
- Logic loop processing
- Rendering
- Manages global animation Timelines via a TimelineManager
- Controls rescaling of canvas to best fit to different display sizes
- Tracks time and measures frame rate

Creating the App

The first task that must be carried out when developing a game with Booty5 is to create the main App object. This object is the main game controller and takes care of processing the entire game. Below is a short piece of example code showing how to set up the app:

```
window.onload = function () {
    // Create the app
    var app = new b5.App(document.getElementById('gamecanvas'));
    b5.app = app; // Some classes expect the current app to be assigned to b5.app
    app.debug = false;
    app.target_frame_rate = 60;
    app.clear_canvas = false;
    app.setCanvasScalingMethod(b5.App.FitBest);

    // Start the app
    app.start();
};
```

We can set up various properties of the app before calling start() to modify how the app behaves. A complete list of those features are listed below:

- debug – If set to true then debug information will be output to the console
- target_frame_rate – The rate at which to update game logic, pass 0 for variable
- clear_canvas – If set to true then the canvas background will be cleared each frame before the next frame is drawn
- canvas_scale_method – Set via setCanvasScalingMethod() sets how the HTML5 canvas will be scaled to fit the browser window
- allow_touchables - If true then app will search to find Actors that were touched y the user, this can be disabled in apps that have no use for touch input
- adaptive_physics – If true then physics update will be ran more than once per frame if frame rate falls much below target frame rate
- loading_screen – An object that contains properties that affect how the loading screen appears

Game Logic and Rendering

Booty5 runs two main loops, the game logic loop and the game rendering loop. The game logic loop is ran on a timer at a rate determined by App.target_frame_rate, executing code that is ran in Scene / Actor update code. The game rendering loop is ran as fast as the browser window can update and executes code that is part of the App / Scene / Actor draw cycle.

Game logic is processed by b5.App.mainLogic() using b5.App.timer, whilst game rendering is processed by b5.App.mainDraw() using requestAnimationFrame().

The Main Canvas

Booty5 has to be able to render its content to a variety of different screen sizes, catering for a huge range of different sizes on a size by size basis is impossible. Instead, Booty5 renders to a virtual canvas of a specific size then scales the canvas to fit to the browser window. So for example, we can render to a fixed sized canvas of 1200x800 but have Booty5 scale that canvas to best fit the target device display. In Booty5 world the virtual canvas is the fixed size that the game will render to. This enables you to lay out your game levels and art work to fit a specific virtual resolution, saving heaps of time and money.

The following canvas fit options are available:

- b5.App.FitNone- No scaling or resizing of the HTML5 canvas will occur
- b5.App.FitX – The HTML5 canvas will be resized to the full size of the display and the virtual canvas will be scaled so that the entire x-axis is fit onto the display
- b5.App.FitY – The HTML5 canvas will be resized to the full size of the display and the virtual canvas will be scaled so that the entire y-axis is fit onto the display
- b5.App.FitBest – The HTML5 canvas will be resized to the full size of the display and the virtual canvas will be scaled either on the X or Y axis depending on which axis keeps the most of the information on the display
- b5.App.FitSize – The HTML5 canvas will be resized to the full size of the display and the virtual canvas will be set to the same size as the display (no scaling will be performed)

The main canvas will be cleared each time the game is ready to start rendering the game. If your game covers the entire area of the window then you can disable canvas clearing by setting b5.App.clear_canvas to false, saving a little rendering time.

Working with Resources

Resources that are stored within the main App's resource lists will persist throughout the lifetime of the app, these resources are known as global resources and are accessible to all other objects within the game.

Booty5 supports the following types of resources:

- Bitmaps – Bitmap images
- Brushes – ImageAtlas and Gradients that are used to render game objects
- Shapes - Shapes that can be used for paths, clipping, rendering and physics fixtures
- Materials – Physics materials
- Sounds – Sound effects

When a resource is created its best to add the resource to a resource manager to be managed. Lets take a look at an example that shows how to create a resource and add it to the apps global resource manager:

```
var material = new b5.Material("static_bounce");
material.restitution = 1;
b5.app.addResource(material, "Material");
```

In the above code we create a physics material then add it to the apps resource manager. Later we can search for this material by calling App.findResource():

```
var material = b5.app.findResource("static_bounce", "Material");
```

Note that as the App can store resources of different types we must specify the type of resource that we wish to find.

If we need to manually destroy a resource at some point in the future then we can either call App.destroyResource() or we can call destroy() on the resource itself, e.g.:

```
// If we do not have a reference to the resource then we can find and remove it
b5.app.destroyResource("static_bounce", "Material");

// If we already have reference to the material then we can destroy it through itself
material.destroy();
```

Note that the Booty5 game maker exports a JSON data format called XOML which the Booty5 game engine reads and converts to resources, scenes, actors etc..

Setting up a loading screen

As many HTML5 games are hosted on a server and assets are loaded asynchronously, its usually a good idea to delay game start until all resources have been loaded from the server. During this time it is customary to display a loading screen to provide feedback to the user so they know that the game is doing something and hasn't crashed.

Booty5 provides a resources loading screen feature out of the box. Calling b5.App.waitForResources() instead of b5.App.start() at the start of the app causes Booty5 to wait for all global and loaded scene resources to load before starting the game engine. In addition, a loading screen will be displayed that shows how far the game data is through being loaded. The loading screen can be customised to your product using the b5.App.loading_screen object. This contains the following properties:

- background_fill - Loading background fill style
- background_image - Loading background image
- bar_background_fill - Loading bar background fill style
- bar_fill - Loading bar fill style

Lets take a quick look at an example:

```
// Set up a loading screen object
b5.app.loading_screen = {
    background_fill: "#fffff",           // Loading background fill style
    background_image: "loading.png",     // Loading background image
    bar_background_fill: "#8080ff",      // Loading bar background fill style
    bar_fill: "#ffffff"                  // Loading bar fill style
};
```

Note that only resources that are marked to be preloaded will be included when the app determines which resources should be waited for. None preloaded resources will be loaded when they are first requested.

Working with Scenes

The main App object contains and manages a collection of scenes. All scenes that are currently loaded will be processed every game logic frame, but only ran if their state is set to active and rendered every render update if their visible state is visible. This system allows you to disable scenes that are not currently in view or not being used, cutting down on overall processing overhead.

Scenes are created and added to the App object using b5.App.addScene(), e.g.:

```
// Create a scene
var scene = new b5.Scene();
// Add scene to the app
app.addScene(scene);
```

Once the scene is added to the app the app will begin processing and rendering it. When we are done with a scene we can remove / destroy it by calling b5.App.removeScene(), e.g.:

```
// Remove scene from app
app.removeScene(scene);
```

You can also remove / destroy a scene by calling destroy() on the scene object:

```
// Remove scene from app
scene.destroy();
```

Note that the scene will not be removed until the end of the apps processing loop.

You can search for scenes by name using b5.App.findScene();

Handling App Events

The App takes care of receiving and passing on a variety of input events that it receives such as touch and keyboard events.

At any given time, only a single scene can have the primary focus (App.focus_scene), any events that are received are passed on to this scene and its contained actors. A secondary focus scene (App.focus_scene2) can also be specified which will receive touch events. Note that game objects within the secondary focus scene will only receive such events if they are "not" processed by an actor in the primary focus scene.

Note that you can change the focus and secondary focus scenes at any time by simply re-assigning them.

The App currently responds to the following events:

- touchstart / mousedown – Touch started
- touchmove / mousemove – Touch moved
- touchend / mouseup – Touch ended
- mouseout – Mouse moved out of control
- keypress – A key was pressed
- keydown – A key is down
- keyup – A key is up
- resize – Window resized, Booty5 will automatically resize the canvas to fit the new size

The App supports a number of properties which allow you to test for certain touch conditions at any time or modify the way the system works:

- touched – True if a touch has occurred in any scene
- touch_pos – The last screen position that was touched
- touch_drag_x and touch_drag_y – The last touch drag delta x and y
- touch_focus – The actor that currently has touch focus
- touch_supported – If touch is supported then this property is set to true
- allow_touchables – Setting this property to false will disable checks for touching actors globally

Working with Variable Frame Rates

Its essential when creating games that run across a variety of different speed devices and platforms that we take into account the speed of the device that the game is being played on. (in game development terms we usually refer to this measurement as frame rate and is measured in frames per second or fps for short) The app tracks how much time the last game logic frame took to update so we know how fast our next frame will run. Knowing this information we can adjust how fast things move to ensure that the game play seems consistent and fluid.

The app automatically tracks how long the last game frame took to render via b5.App.dt (delta time), this value, measured in seconds is also passed on to all scenes, actors, timelines and anything else that relies on time synchronisation. All movement and animation is scaling by this value to ensure that it plays back at a consistent speed regardless of frame rate.

The app also measures the average frame rate in fps via b5.App.avg_fps, measured every 60 frames. You can use this value to determine how fast the device is that you are running your game on. This value is also used with adaptive physics (b5.App.adaptive_physics) which when set to true will run the physics system multiple times during a single game frame to help ensure that constant time step physics behaves better at lower frame rates (it attempts to match the value set in b5.App.target_frame_rate).

App Public Properties

- scenes – An array of Scenes
- canvas – The HTML5 canvas
- allow_touchables – if true then app will search to find Actor that was touched
- target_frame_rate – Frame rate at which to update the game logic
- adaptive_physics – When true physics update will be ran more than once if frame rate falls below target
- focus_scene – Scene that has current input focus
- focus_scene2 – Scene that has secondary input focus
- clear_canvas – If true then canvas will be cleared each frame
- touch_focus – The Actor that has the current touch focus
- debug – Can be used to enable / disable debug trace info
- timelines – Global animation TimelineManager
- pixel_ratio – Device pixel ratio
- canvas_width – Virtual canvas width
- canvas_height -Virtual canvas height
- display_width – Width of display (client area)
- display_height – Height of display (client area)
- canvas_scale_method – Method of scaling used to scale virtual canvas to display
- touch_supported – True if touch is supported
- loading_screen – An object that contains the following properties that affect how the loading screen appears:
- background_fill – Loading screen background colour
- background_image – An image that will be displayed for the loading screen
- bar_background_fill – Loading bar background colour
- bar_fill – Loading bar colour

App Internal Properties

- removals – Array of scenes that were deleted last frame
- touched – true if the screen is being touched, false otherwise
- touch_pos – Position of last screen touch {x, y}
- touch_drag_x – Amount touch position was last dragged on x axis
- touch_drag_y – Amount touch position was last dragged on y axis
- last_time – Time of last frame update in milliseconds
- dt – Time period that has passed since the last update of the app in seconds
- avg_time – Total time since last average time measurement
- avg_fps – Frames per second of last average time measurement
- avg_frame – Counter used to take average time measurements
- canvas_fill_window – If set to true then canvas will be scaled to fit window
- canvas_scale – The amount that the virtual canvas is scaled to fit the screen
- canvas_cx – Canvas x-axis centre
- canvas_cy – Canvas y-axis centre
- total_loaded – Total resources that have been loaded so far (only resources that have been marked as preload)
- total_load_errors – Total number of resources that were not loaded due to errors
- order_changed – Set to true when scene order changes, causes a re-sort
- bitmaps – Array of Bitmap resources
- brushes – Array of ImageAtlas (other brush types will be added in future) resources

- shapes – Array of Shape resources
- materials – Array of Material resources
- sounds – Array of Sound resources

App Constants

- b5.App.FitNone- No scaling or resizing of the HTML5 canvas will occur
- b5.App.FitX – The HTML5 canvas will be resized to the full size of the display and the virtual canvas will be scaled so that the entire x-axis is fit onto the display
- b5.App.FitY – The HTML5 canvas will be resized to the full size of the display and the virtual canvas will be scaled so that the entire y-axis is fit onto the display
- b5.App.FitBest – The HTML5 canvas will be resized to the full size of the display and the virtual canvas will be scaled either on the X or Y axis depending on which axis keeps the most of the information on the display
- b5.App.FitSize – The HTML5 canvas will be resized to the full size of the display and the virtual canvas will be set to the same size as the display (no scaling will be performed)

App Events

- onResourcesLoaded(resource, error) – Called when a resource has been loaded

App Methods

- App(canvas) – Creates and instance of the app
  - canvas – The HTML5 canvas object that will be drawn to
- onTouchStart(e) – Event handler that is called by the system when the user begins touching the display (this includes mouse button down) (used internally)
  - e – A mouse touch event
- onTouchEnd(e) – Event handler that is called by the system when the user stops touching the display (this includes mouse button up) (used internally)
  - e – A mouse touch event
- onTouchMove(e) – Event handler that is called by the system when the user moves a touch around the display (this includes mouse move) (used internally)
  - e – A mouse touch event
- onKeyPress(e) – Event handler that is called by the system registers a key press (used internally)
  - e – A key event
- onKeyDown(e) – Event handler that is called by the system receives a key down event (used internally)
  - e – A key event
- onKeyUp(e) – Event handler that is called by the system receives a key up event (used internally)
  - e – A key event
- onResourceLoadedBase(resource, error) – Base event handler for onResourceLoaded. If you override onResourceLoaded then ensure that you call this base to ensure correct operation
  - resource – Resource that was loaded
  - error – true if an error occurred whilst loading
- addScene(scene) – Adds a scene to the app

  - scene – The scene to add
- removeScene(scene) – Removes the specified scene from the app destroying it. Note that the scene is not removed immediately, instead it is removed when the end of the frame is reached.
  - scene – The scene to remove
- cleanupDestroyedScenes() – Cleans up all destroyed scenes, this is called by the app to clean up any removed scenes at the end of its update cycle (used internally)
- findScene(name) – Searches the app for the named scene
  - name – Name of the scene to find
  - returns the found scene object or null for scene not found
- addResource(resource, type) – Adds a resource to the global app resource manager
  - resource – The resource to add
  - type – Type of resource to search for (brush, sound, shape, material, bitmap or geometry)
- removeResource(resource, type) – Removes a resource from the global app resource manager
  - resource – The resource to remove
  - type – Type of resource to remove (brush, sound, shape, material, bitmap or geometry)
- findResource(name, type) – Searches the global app resource manager for the named resource
  - resource – Name of resource to search for
  - type – Type of resource to search for (brush, sound, shape, material, bitmap or geometry)
  - returns the found resource or null if not found
- countResourcesNeedLoading(include_scenes) – Counts and returns how many resources that need to be loaded
  - include_scenes – if set to true then resources in all loaded scenes will also be counted
- findHitActor(position) – Searches all touchable actors in all app scenes to see if the supplied position hits them
  - position – The position to hit test
  - returns the actor that was hit or null for no hit
- draw() – Draws the app and all of its contained scenes
- update(dt) – Updates the app and all of its contained scenes
  - dt – The amount of time that has passed since this app was last updated
- mainLogic() – The apps main loop (used internally)
- mainDraw() – The apps draw loop (used internally)
- start() – Starts the app going
- dirty() – Dirties the scene and all child actors transforms
- setCanvasScalingMethod(method) – Sets the method of scaling the virtual canvas to the HTML5 canvas
  - method – The method of scaling to use, can be FitNone, FitX, FitY, FitBest or FitSize
- waitForResources() – Waits for all preload resources to load before starting the app, also displays a loading screen and loading bar. Use this in place of calling app.start() directly.
- parseAndSetScene(scene_name) – Parses the named scene and sets it as the current focus scene
  - scene-name - The name of the scene, note that the scene name and the exported name of the scene must match

Scenes – A Place for Actors to Play

Introduction

Booty5 is built around Scenes and Actors. A scene is a container for a collection of actors and other resources (such as shapes, materials, bitmaps etc..), whilst an actor is simply a game object that provides some kind of game functionality. A scene will manage the lifetime, processing and rendering of any game objects that are added to it. A scene will also manage the lifetime of any local resources (such as images, sounds etc..) as well as animation timelines (animation timelines are targeted at scenes and game objects to animate them) and actions lists.

It can be easier to think about Booty5 game development if we think in terms of the movie business, we all watch movies and programmes on the TV which makes it easy to relate to. A movie usually consists of a number of scenes that contain the environment, actors (provide the focus and entertainment of the movie) and a view into the scene (a camera).

Booty5 is based on these similar principles, where the scene acts as the game environment where the action takes place, actors represent individual game objects and the camera (which in this case is part of the scene) the view into the game.

A scene contains and manages the following:

- A hierarchy of actors (game objects)
- A collection of game resources (bitmaps, sounds, physics materials, shapes etc..) that are local to the scene
- A collection of local animation timelines
- A collection of action lists
- A camera to navigate the scene
- A clipping region which can be used to clip child game objects
- Physics world update

Creating Scenes

Scenes are created by creating an instance of the b5.Scene object which are then added to the main App object for processing using b5.App.addScene(), e.g.:

```
// Create a scene
var scene = new b5.Scene();
// Add scene to the app
b5.app.addScene(scene);
```

Once the scene has been created you can begin setting the various properties of the scene. Its best to always give your new scene a name by assigning the name to scene.name. This will allow you to find the scene at some point in the future by searching for it by name using b5.App.findScene(name).

Once the scene is added to the app the app will begin processing and rendering it. When we are done with a scene we can remove / destroy it by calling b5.App.removeScene(), e.g.:

```
// Remove scene from app
b5.app.removeScene(scene);
```

You can also remove / destroy a scene by calling destroy() on the scene object:

```
// Remove scene from app
scene.destroy();
```

Note that the scene will not be removed until the end of the apps processing loop.

Scene Events

The scene handles a number of events that you can tap into:

- onCreate() - When a scene that was exported from the Booty5 game editor is created it will call the this event in response to the scene being created as long as it was defined within the editor
- onDestroy() - When a scene is to be destroyed during tear down this event will be raised
- onTick(delta_time) - This event is raised every time the scene is about to be updated (every logic frame)
- onBeginTouch(touch_pos) – This events is raised when the user touches the scene
- onEndTouch(touch_pos) – This events is raised when the user stops touching the scene
- onMoveTouch(touch_pos) – This events is raised when the user moves a touch around the scene

Lets take a quick look at an example of using scene events:

```
scene.onBeginTouch = function(touch_pos) {
    console.log("Scene was touched");
}
```

Processing and Visibility

Scenes are processed when their active property is set to true, if a scene is marked as not active then the scene and all of its actors and timelines will not be updated. Its generally a good idea to deactivate scenes when they are not in use, especially when targeting resource constrained devices such as mobile devices.

Scenes are rendered when their visible property is set to true. If a scene is marked as not visible then the scene and all of its contained actors will not be rendered. Again, its a good idea to hide scenes that are hidden by other scenes or are currently off screen.

Note that each time the scene is updated (each game frame and as long as the scene is active) its onTick() method will be called. You can tap into this event to add scene specific functionality, e.g.:

```
scene.onTick = function (dt) {
    // Do something with the scene (dt is delta time passed since last updated)
};
```

Local Resources

Resources can be local or global. If a resource is global then it is available to all scenes and the objects which they contain. Global resources are also always present in memory and are not freed until the app exits or are removed manually. If a resource is local to a scene then it is usually only accessible to the scene and objects within that scene. In addition, when the scene is destroyed all resources that it contains are destroyed along with it.

Resources can be added to a scene using b5.Scene.addResource(), e.g.:

```
var material = new b5.Material("static_bounce");
material.restitution = 1;
scene.addResource(material, "Material");
```

In the above piece of code we create a physics material then add it to the scenes resources.

To later retrieve that resource we would search for it using:

```
scene.findResource("static_bounce", "Material");
```

Resources can also be located via their path. A path represents the hierarchical path to the object which consists of the names of the parent objects separated by dots. For example, if you want to locate a physics material called "material1" that is located in a scene named "scene1" then the path to that resources would be "scene1.material1". If the material was located in the Apps global resource space then the path would simply be "material1". To find the instance of the resource from the path you can call b5.Utils.findResourceFromPath(path, type), e.g.:

```
var material = b5.Utils.findResourceFromPath("scene1.material1", "material");
```

Later if we need to remove and destroy the resource then we can either call:

```
// If we do not have a reference to the resource then we can tell the scene to find and
remove it
scene.destroyResource("static_bounce", "Material");

// If we already have reference to the material then we can destroy it through the object
itself
material.destroy();
```

Working with Actors

Scenes are designed by their very nature to contain and manage game objects (actors). An actors life time is determined by the lifetime of a scene, once a scene is killed off so are all of the game objects that it contains. When an actor is first created it needs to be added to a scene in order for it to be processed and rendered.

To add an actor to a scene use b5.Scene.addActor(actor), to later find an actor simply call b5.Scene.findActor(actor_name), you can pass an additional parameter to this method that will force the search to search all child actors also.

Once an actor is part of a scene it can later be removed by calling b5.Scene.removeActor(actor) or by calling b5.Actor.destroy() on the actor object.

Actors can be categorised using a tag name that is specified by the actors tag property. Game objects can be removed en-mass by tag using b5.Scene.removeActorsByTag(tag_name);, this method will remove all actors within the scene that have the specified tag; tags enable you to mark groups of actors and later remove them all in one go.

Whilst layers can be used to visually order the order in which visible scenes are overlaid, where layers are not used you can bring actors to the front of the visual stack or send them to the back using b5.Scene.bringToFront() and b5.Scene.sendToBack().

The Scene Camera

The camera is the view into the scene (game world). The scene can be moved around by changing Scene.camera_x and Scene.camera_y or by changing the cameras velocities Scene.camera_vx and Scene.camera_vy.

A scene has a single camera with a variety of built in functions:

- Touch panning
- Target tracking
- Constrained movement
- Velocity damping

Touch panning

Touch panning is a feature that allows the user to pan around the game world (a scene) using their finger on touch devices or the mouse on desktop devices. When the user holds down the mouse button or touches the screen and drags, the camera will follow the players finger. This feature is not only great during testing but also very useful for many different types of games that have game worlds that are larger than the screen as it gives the user the opportunity to take a look around the world.

Touch panning can be enabled for each separate axis by setting b5.Scene.touch_pan_x and / or b5.Scene.touch_pan_y to true.

Its often useful to be able to tell if the user is currently touch panning the camera, this can be done by checking the value of b5.Scene.panning, if true then the user is currently panning around the scene. During the development of Leapo I bumped into an issue on mobile devices. I allow the user to pan around the game world, but also when the user taps the screen it causes the frog to jump. On some mobile devices, very small touch pans were causing the game to ignore frog jumps. To fix this I included a tolerance in b5.Scene.min_panning, which sets a minimum amount of panning movement that should be considered a pan (note that this value is the squared distance of the minimum, for example if you want a minimum value of 2 pixels in either direction then set this value to 2*2 + 2*2 = 8).

Target tracking

Target tracking (or camera follow) is a feature that enables the scenes camera to follow a specified actor or pair of actors. The camera can track game objects on separate axis, so for example you could have the camera track a player character on the x-axis, whilst also simultaneously tracking an evil alien on the y-axis.

To enable target tracking simply set b5.Scene.target_x and / or b5.Scene.target_y to the actor that you wish the camera to follow (in the game editor this option is available the the scenes properties), e.g.:

```
// Camera tracking two actors
scene.target_x = b5.Utils.resolveObject("scene1.actor1");
scene.target_y = b5.Utils.resolveObject("scene1.actor2");
```

The rate at which the camera follows the targets can be set via b5.Scene.follow_speed_x and b5.Scene.follow_speed_y, higher values will cause the camera to catch up with the target more quickly,

Constrained Movement

Scenes have a rectangular boundary (extents) that can be used to constrain the camera and its actors. A scenes extents can be set via b5.Scene.extents which is an array of 4 values that represents the top, left, width, height of the constrained area. (In the game editor the scenes extents is represented by a green rectangle). When the camera hits the edges of the scene extents it will stop, e.g.:

```
// Limit movement of camera between -200, -200 and 200, 200
scene.extents = [-200, -200, 400, 400];
```

Coordinate System

A scenes coordinate system is based at its centre, if the scene is located in the middle of the screen then the scenes world origin will be at the centre of the screen. The coordinate system is designed this way to make it easier to design outwards. Designing outwards is incredibly useful when designing games that can be played across a variety of different sized displays as the main focus of the game is at its centre and not at a potentially clipped edge.

Position and Opacity

A scene has position and size (b5.Scene.x, b5.Scene.y, b5.Scene.w, b5.Scene.h) and can be moved around the screen, enabling effects such as scrolling scenes on and off the screen. When a scene is moved all of its contained actors are also moved.

Scenes have an opacity level (b5.Scene.opacity) between 0 and 1 which determines how opaque / transparent the scene and its contained game objects are This values can be used for effects such as fades where the entire scene is faded up and down

Note that within the game editor the opacity values ranges from 0 to 255, instead of 0 to 1 and is the 4th parameter of the scenes Colour property.

Child Clipping

A scene can clip its child actors by enabling Scene.clip_children. By default the scene will clip children against its dimensions (defined by Scene.w and Scene.h), but this can be changed by assigning a Shape to Scene.clip_shape which causes scene objects to be clipped against that shape, e.g.:

```
// Set clipping shape that scene will use to clip children
scene.clip_shape = b5.Utils.resolveResource("shape1", "shape");
```

Scene Layering

During game development you will find times when you need to display more than one scene at the same time. For example in the game Leapo there is the main game scene which contains the game world and the heads up display (HUD) scene which contains the players lives left. time left, level and pause button. Each scene can have a layer number (b5.Scene.layer) which determines the visual order in which scenes are drawn to the display, scenes on higher layers will appear above scenes on lower layers.

Note that you should always set the scenes layer via the b5.Scene._layer property setter to ensure that the scene layers get re-sorted, e.g.:

```
scene1._layer = 1; // This scene will appear below layer 2
scene2._layer = 2; // This scene will appear above layer 1
```

Scene Physics

Scenes can support a Box2D physics world, by default Box2D physics is disabled and has to be enabled. To enable physics you need to include the Box2D JavaScript library into your index.html file:

```
<script src='lib/Box2dWeb-2.1.a.3.min.js'></script>
```

Booty5 uses the Box2DWeb JavaScript library, you can see reference for this library here.

Scenes can selectively use Box2D physics, to enable physics in a scene you need to initialise the scenes physics world by calling b5.Scene.initWorld(), e.g:

```
scene.initWorld(gravity_x, gravity_y, do_sleep);
```

In the call to initWorld() we pass the x-axis and y-axis gravity (0, 10 as default) and a flag that determines if the physics system should allow objects to sleep. I recommend that you do enable sleeping as it provides a good speed boost on mobile devices.

Once physics has been enabled for the scene you can set a number of other properties that affect how the physics system behaves:

- time_step – This is the amount of time that has elapsed between each call to update physics in seconds. By default this value is set to 0, which causes the scene to pass a variable time step to the physics system, this however is not ideal as physics will not behave consistently across different speed devices. Its generally better to pass a constant value and run the physics system multiple times when the frame rate drops. When exporting from the game editor a default value of 0.033 is passed (30 fps). Note that when adaptive physics is enabled in the App, the physics system will automatically be ran multiple times if the frame rate drops.
- world_scale – This value affects how the physics world maps to the visual world (default value is 20)

If a scene has any objects that use the Box2D physics system then the scene must have physics enabled for it to work.

When physics is enabled in a scene the physics simulation will be updated during the scenes game logic update.

Animation Timelines

A scene manages a collection of animation timelines via its TimelineManager (b5.Scene.timelines). An animation timeline is basically a collection of animations that are played back asynchronously. Usually objects within the scene or even the scene itself will add its animation to the scene for processing. Timelines will only play whilst the scene is active and destroying the scene will also destroy all contained animation timelines.

To add an animation timeline to a scene you can call b5.Scene.timelines.add(my_timeline), e.g.:

```
// Create a timeline that scaled actor1
var timeline = new b5.Timeline(actor1, "_scale", [1, 1.5, 1], [0, 0.5, 1], 1,
[b5.Ease.quartin, b5.Ease.quartout]);

// Add timeline to the scene for processing
scene.timelines.add(timeline);
```

Actions Lists

A scene manages a collection of actions lists via the ActionsListManager (b5.Scene.actions). An actions list is a collection of actions that are executed consecutively. An action is a single unit of functionality such as tween a collection of properties over time, start a sound effect or animation etc..

Usually objects within the scene or even the scene itself will add its action lists to the scene for processing. Action Lists will only play whilst the scene is active and destroying the scene will also destroy all contained action lists.

To add an actions list to a scene you can call b5.Scene.actions.add(actions_list), e.g.:

```
// Create actions list
var actions_list = new b5.ActionsList("turn", 0);

// Add an action to actions list
actions_list.add(new b5.A_SetProps(actor, "vr", 2 / 5));

// Add actions list to the scenes actions list manager
scene.actions.add(actions_list);
```

Input Events

Scenes can receive input events if they are set as a focus or secondary focus scene in the App. When a touch / mouse event occurs it is sent from the app to the focus scene and then the secondary focus scene. In order to act upon these events you need to assign the following event handlers:

- onTapped(touch_pos) – This event is raised when the user taps the screen (a tap is defined as when the user touches the screen then lets go)
- onBeginTouch(touch_pos) – This event is raised when the user touches / clicks the screen
- onEndTouch(touch_pos) – This event is raised when the user stops touching / clicking the screen
- onMoveTouch(touch_pos) – This event is raised when the user moves their finger / mouse around the screen whilst touching it

A scene also receives keyboard input events, but only the main focus scene can receive them and not the secondary focus scene. These events include:

- onKeyDown(e) – This event is raised when the user presses a key, the raised event e contains the keys details with e.keyCode containing the key code
- onKeyUp(e) – This event is raised when the stops pressing a key, the raised event e contains the keys details with e.keyCode containing the key code

Note that within the game editor, the code entered into the event handlers will be called each time the event is raised.

A number of useful properties of the scene can be used to determine if the user is touching or moving a touch:

- b5.Scene.touching – Set to true when the user is touching the screen
- b5.Scene.touchmove – Set to true when the user is moving a touch

Detecting when Resources have Loaded

Booty5 can automatically take care of waiting for resources to be loaded, including loading the resources that need to be loaded for all scenes that are loaded at the start of the app. However, you may find that you need to load scenes at a later date that contain resources.

Its possible to check when a scenes resources have finished loading using b5.Scene.areResourcesLoaded(), which will return true when all resources have finished loading. You could do this using a timer then update the UI to display a loading bar, e.g.:

```
// Wait for scene resources to finish loading
var resource_check_interval = setInterval(function () {
    if (scene.areResourcesLoaded()) {
        // Resources have finished loading
    }
    clearInterval(resource_check_interval);
}, 500);
```

You can also find out how many resources need to be loaded in the scene by calling b5.Scene.countResourcesNeedLoading().

Scene Public Properties

- name – Name of the scene (used to find actors in the scene)
- tag – Tag (used to find groups of actors in the scene)
- active – Active state, inactive scenes will not be updated
- visible – Visible state, invisible scenes will not be drawn
- layer – The visible layer that this object sits on
- x – Scene x axis position
- y – Scene y axis position
- w – Scene canvas width
- h – Scene canvas height
- clip_children – If set to true then actors will be clipped against extents of this scene
- clip_shape – If none null and clipping is enabled then children will be clipped against shape (clip origin is at centre of canvas)
- camera_x – Camera x position
- camera_y – Camera y position
- camera_vx – Camera x velocity
- camera_vy – Camera y velocity
- vx_damping – Damping to apply to camera_vx
- vy_damping – Damping to apply to camera_vy
- panning – Set to true when user is panning the scene
- min_panning – A minimum distance that the user must pan for panning to be set to true (this value is the squared distance)
- follow_speed_x – Camera target follow speed x axis
- follow_speed_y – Camera target follow speed y axis
- target_x – Camera actor target on x axis
- target_y – Camera actor target on y axis
- touch_pan_x – If true then scene will be touch panned on x axis
- touch_pan_y – If true then scene will be touch panned on y axis
- world_scale – Scaling from graphical world to Box2D world
- time_step – Physics time step in milliseconds (1/60 for 60 fps), setting to 0 will run physics at a variable rate based on frame update speed
- extents – Scene camera extents [left, top, width, height]
- opacity – Scene opacity

Scene Internal Properties

- app – Parent container app
- actors – Array of actors
- removals – Array of actors that were deleted last frame
- frame_count – Number of frames that this scene has been running
- world – Box2D world
- touching – Set to true when user touching in the scene
- touchmove – Set to true when touch is moving in this scene
- timelines – Scene local animation TimelineManager
- actions – Scene local ActionsListManager
- bitmaps – Array of Bitmap resources
- brushes – Array of ImageAtlas (other brush types will be added in future) resources
- shapes – Array of Shape resources
- materials – Array of Material resources

- sounds – Array of Sound resources
- order_changed – When layer changes this property is marked as true to let the system know to resort all objects on the layer

Scene Setters

- _layer – Changes the scenes layer, note that visual ordering of scenes does not take place until the end of the game frame
- _focus_scene – Sets the current focus scene, can use path to scene or instance of scene
- _focus_scene2 – Sets the current secondary focus scene, can use path to scene or instance of scene
- _clip_shape – Sets the clip shape, can use path to scene or instance of scene

Scene Events

- onCreate() – Called just after the scene has been created
- onDestroy() – Called just before the scene is destroyed
- onTick(delta_time) – Called each time the scene is updated (every frame)
- onBeginTouch(touch_pos) – Called when the scene is touched
- onEndTouch(touch_pos) – Called when the scene has top being touched
- onMoveTouch(touch_pos) – Called when a touch is moved over the scene
- onKeyPress(e) – Called when a key is pressed, e is key event
- onKeyDown(e) – Called when a key is down, e is key event
- onKeyUp(e) – Called when a key is up, e is key event

Scene Methods

- Scene() – Creates an instance of a Scene object. Once a scene has been created it should be added to the main app object to be processed and drawn
- release() – Releases the scene, this is called by the app system when an scene has been destroyed
- destroy() – Begins the destruction of a scene and its content, note that the scene will not actually be destroyed until the end of the apps processing
- addActor(actor) – Adds the specified actor to the scene, returns the added actor
- removeActor(actor) – Removes the specified actor from the scene, destroying it
- removeActorsWithTag(tag) – Removes all actors from the scene that are tagged with the specified tag
- cleanupDestroyedActors() – Cleans up all destroyed scenes, this is called by the app to clean up any removed scenes at the end of its update cycle
- findActor(name, recursive) – Searches this scenes child list for the named actor.
  - name: The name of the actor
  - recursive – If true then the complete child hierarchy will be searched
  - returns the found actor or null if not found
- bringToFront() – Moves the scene to the end of the scenes list, bringing it to the front of all other scenes in the app.
- sendToBack() – Moves the scene to the start of the scenes list, pushing it to the back of all other scenes in the app.
- initWorld(gravity_x, gravity_y, allow_sleep) – Initialises a Box2D world and attaches it to the scene. Note that all scenes that contain Box2D physics objects

must also contain a Box2D world
  - gravity_x – X axis gravity
  - gravity_y – Y axis gravity
  - allow_sleep – If set to true then actors with physics attached will be allowed to sleep
- draw() – Draws this scene and its children, this method can be overridden by derived scenes
- baseUpdate(dt) – Base update function (used internally) that should be called by all derived scenes that wish to use base scene functionality, this is usually called from your update() method.
  - dt – The amount of time that has passed since this scene was last updated
- update(dt) – Updates the scene (used internally), this method can be overridden by derived scenes
  - dt – The amount of time that has passed since this scene was last updated
  - returns true if active
- updateCamera(dt) – Updates the scenes camera
  - dt – The amount of time that has passed since this scene was last updated
  - returns true if active
- findHitActor(position) – Searches all touchable actors in the scene to see if the supplied position hits them
  - position – The position to hit test
  - returns the actor that was hit or null for no hit
- addResource(resource, type) – Adds a resource to the scenes resource manager
  - resource – The resource to add
  - type – Type of resource to add (brush, sound, shape, material, bitmap or geometry)
- removeResource(resource, type) – Removes a resource from the scenes resource manager
  - resource – The resource to remove
  - type – Type of resource to remove (brush, sound, shape, material, bitmap or geometry)
- findResource(name, type) – Searches the scenes resource manager for the named resource, if the resource is not found in this scene then the apps global resources will be searched
  - resource – Name of resource to search for
  - type – Type of resource to search for (brush, sound, shape, material, bitmap or geometry)
  - returns the found resource or null if not found
- areResourcesLoaded() – Returns true if all scene resources have been loaded, otherwise false

Scene Examples

Creating a scene

```
var scene = new b5.Scene();      // Create instance of a scene
scene.name = "my_scene";         // Name the scene
b5.app.addScene(scene);          // Add the scene to the app for processing
b5.app.focus_scene = scene;      // Set our scene as the focus scene
```

Adding clipping to a scene

```
var clipper = new b5.Shape();       // Create a circle shape
clipper.type = b5.Shape.TypeCircle;
clipper.width = 100;
scene.clip_shape = clipper;         // Assign the shape as the scenes clip shape
```

Adding touch panning to a scene

```
scene.touch_pan_x = true;        // Enable touch panning on x-axis
scene.touch_pan_y = true;        // Enable touch panning on x-axis
```

Adding a physics world to a scene

```
scene.initWorld(0, 10, true);    // Initialise physics world (gravity_x, gravity_y,
do_sleep)
```

Adding an onTick event handler to a scene

```
scene.onTick = function(dt) {
    this.x++;
};
```

Adding touch event handlers to a scene

```
scene.onBeginTouch = function(touch_pos) {
    console.log("Scene touch begin");
};
scene.onEndTouch = function(touch_pos) {
    console.log("Scene touch end");
};
scene.onMoveTouch = function(touch_pos) {
    console.log("Scene touch move");
};
```

Make the scene camera follow a target actor

```
scene.target_x = my_actor;
scene.target_y = my_actor;
```

Scroll scene off to right using timeline animation

```
var timeline = new b5.Timeline(scene, "x", [0, 1024], [0, 2], 1, [Ease.quartin]);
scene.timelines.add(timeline); // Add to scenes timeline manager to be processed
```

Add an actions list

```
// Create an actions list
var actions_list = new b5.ActionsList("fade", 1);
// Add an action that moves the camera
actions_list.add(new b5.A_CamMoveTo(scene, 100, 100, 2, b5.Ease.linear, b5.Ease.linear));
// Add an action that fades the scene
actions_list.add(new b5.A_TweenProps(scene, ["opacity"], [1], [0], 2, [b5.Ease.linear]));
// Add actions list to scene and set it playing
scene.actions.add(actions_list).play();
```

Actors – Sprites with Brains

Introduction

Going back to our comparison in the scenes introduction section, actors play a pivotal role in our scenes, each actor having its own unique role and visual appearance. Actors are the building block of the game, they provide the unique functionality and visuals that make up the game as a whole, rather like each actor plays his / her role in a movie.

Actors can provide any type of functionality from a simple bullet fleeting across the screen to something as complex as a dynamic machine that modifies its behaviour and appearance based upon data streamed from a web server.

An Actor object represents a game object that can be added to Scenes for processing and display. You can add logic to the actor via its update() method and or by attaching an onTick event handler.

The base Actor has the following features:

- Position, size, scale, rotation
- Absolute (pixel coordinate) and relative (based on visible size) origins
- Layering
- Support for cached rendering
- 3D depth (allows easy parallax scrolling)
- Angular, linear and depth velocity
- Box2D physics support (including multiple fixtures and joints)
- Bitmap frame animation
- Timeline animation manager
- Actions list manager
- Sprite atlas support
- Child hierarchy
- Angular gradient fills
- Shadows
- Composite operations
- Begin, end and move touch events (when touchable is true), also supports event bubbling
- Canvas edge docking with dock margins
- Can move in relation to camera or be locked in place
- Can be made to wrap with scene extents on x and y axis
- Clip children against the extents of the parent with margins and shapes
- Supports opacity
- Can be represented visually by arcs, rectangles, polygons, bitmaps and labels
- Support for a virtual canvas that can scroll content around

Other Actor types are derived from b5.Actor, including:

- b5.ArcActor - Circular based game objects
- b5.LabelActor - Text based game objects
- b5.ParticleActor - Particle system based game objects

- b5.PolygonActor - Polygon based game objects
- b5.RectActor - Rectangular based game objects

All shape and text based actors can be rendered filled or unfilled by changing the actors filled property.

Creating Actors

Actors are created by creating an instance of a b5.Actor object or any of the actor types mentioned above then adding that instance to a Scene or another Actor. Lets take a quick look at an example:

```
var actor = new b5.Actor(); // Create instance of Actor
scene.addActor(actor);      // Add actor to scene to be processed and drawn
```

Of course this actor will do absolutely nothing as it has no visual component assigned. Lets take a look at how to create actors of different types.

Creating an Arc Actor

An arc actor represents a circular shaped game object, an example showing how to create one is shown below:

```
var actor = new b5.ArcActor(); // Create instance
actor.x = 100;                 // Set x axis position
actor.y = 0;                   // Set x axis position
actor.fill_style = "#00ffff";  // Set fill style
actor.start_angle = 0;         // Set start angle
actor.end_angle = 2 * Math.PI; // Set end angle
actor.radius = 50;             // Set radius
actor.filled = true;           // Set filled
scene.addActor(actor);         // Add actor to scene to be processed and drawn
```

In the above code we first of all create an instance of ArcActor, set some properties of the actor such as position, size, start and end angle etc.. then we add it to the scene to be processed.

Creating a Rectangular Actor

A rectangular actor represents a rectangular shaped game object, an example showing how to create one is shown below:

```
var actor = new b5.RectActor(); // Create instance
actor.fill_style = "#40ff4f";   // Set fill style
actor.filled = true;            // Set filled
actor.w = 100;                  // Set drawn width
actor.h = 100;                  // Set drawn height
actor.corner_radius = 10;       // Set corner radius (this is a rounded ract)
scene.addActor(actor);          // Add actor to scene for processing and drawing
```

Creating a Label Actor

A label actor represents a game object that displays a string of text, an example showing how to create one is shown below:

```
var actor = new b5.LabelActor(); // Create instance of actor
actor.font = "16pt Calibri";     // Set font
actor.text_align = "center";     // Set horizontal alignment
actor.text_baseline = "middle";  // Set vertical alignment
actor.fill_style = "#ffffff";    // Set fill style
actor.text = "Hello World";      // Set some text
scene.addActor(actor);           // Add to scene for processing and drawing
```

Creating a Polygon Actor

A polygon actor represents a game object that displays a shape, an example showing how to create one is shown below:

```
var actor = new b5.PolygonActor();           // Create instance of actor
actor.name = "polygon1";                     // Set actors name
actor.points = [0, -50, 50, 50, -50, 50];    // Set shape
actor.fill_style = "#804fff";                // Set fill style
actor.filled = true;                         // Set filled
scene.addActor(actor);                       // Add to scene for processing and drawing
```

Creating a Bitmap Actor

Lets take a look at creating a slightly more complex actor, one with a bitmap attached:

```
// Create an actor
var actor = new b5.Actor();
actor.x = 100;
actor.y = 100;
actor.w = 200;
actor.h = 200;
scene.addActor(actor); // Add to scene

// Create and attach a bitmap
act.bitmap = new b5.Bitmap("background", "images/background.jpg", true);
```

Whilst in the above example we create a bitmap and assign it to the actor directly, its better practice to create the bitmap then add it to either the scenes resources or apps global resources so that it may be managed and re-used.

Actors can be destroyed by calling their b5.Actor.destroy() method, this will remove them from their scene and destroy all physics joints, fixtures and so forth that are attached to them. You can also destroy and actor by calling Scene.removeActor().

Note that when an actor is created its onCreate() method will be called to allow any post creation tasks to be carried out (Booty5 game editor only). When an actor is destroyed its onDestroy() method will be called so that any pre destruction tasks can be carried out.

You can see an example of the creation of different types of actors in the shapes demo.

Processing and Visibility

Actors are processed when they are part of a scene or some other actor and their active property is set to true, if marked as not active then it and all of its child actors will not be updated. It can be a good idea to deactivate actors when they are not in use, especially if you have a large number of them within a scene.

Actors are rendered when their visible property is set to true. If marked as not visible then the actor and all of its children will not be rendered.

Note that each time the actor is updated (each game frame and as long as it is active) its onTick() method will be called, e.g.:

```
actor.onTick = function (dt) {
    // Do something (dt is time passed since last updated)
};
```

Its generally here where you will add your actor logic to give your actor its unique functionality, unless of course you plan on deriving your own actor type from one of the existing types and implementing the update() method.

Transforms

All actor types have various properties that can be modified to change the position, rotation and scale of the actor and all of its children. These properties are listed below:

- x – X position in scene
- y – Y position in scene
- ox – X origin
- oy – Y origin
- absolute_origin – If true then ox and oy are taken as absolute coordinates, if false then ox and oy are taken as percentage of width and height
- rotation – Rotation in radians
- scale_x – X scale
- scale_y – Y scale
- depth – Z depth (3D depth), 0 represents no depth

An actors internal visual transform will only be re-calculated when properties such as position, scale and rotation changes, this is an optimisation to cut down on unnecessary processing. Changing these properties after actor creation should be done via the associated propertiy setters such as _x, _y (see Actor Setters for a full list).to ensure that the internal transform is updated. Alternatively you can set b5.Actor.transform_dirty to true to force the internal transform to be updated. Note that when an actors transform is dirtied, all of its children's transforms will also be dirtied.

Note that when setting an actors origin, there are two ways in which the origin properties can be interpreted. If absolute_origin is set to true then the origin will be taken as absolute coordinates. if however it is set to false then the coordinates will be taken as percentage of actor width and height values, so for example an ox value of 0.5 and an actor width of 100 will result in a final x origin value of 50.

Child Hierarchies

Its often very useful when creating game objects to build them out of parts, for example, in the Leapo game, the frog player is built up out of multiple parts. We have the frogs main body, which has a head attached, the head also has two eyes attached to it. This enables me to change the head independent of the body and the eyes independent of the head. However, if the head rotates then I want the eyes to rotate with it. The hierarchical system ensures that changes to the parent are propagated to its children, ensuring children obey the same transforms and opacity settings as their parent.

Each actor maintains its own list of children. To make an actor a child of another, simply call parent_actor.addActor(child). This will add the child to the parents hierarchy, e.g.:

```
var actor1 = new b5.Actor();
var actor2 = new b5.Actor();
actor1.addActor(actor2);
scene.addActor(actor1);
```

To remove a child from the parent simply call child.parent.removeActor(child).

To search an hierarchy for an actor call parent.findActor("actors name", recursive). Note that if recursive is true then the entire actor hierarchy will be searched for the named actor.

Child and Self Clipping

An actor can clip its children as well as itself against its extents or a supplied clip shape. To enable clipping of children set b5.Actor.clip_children to true. To enable clipping against itself set b5.Actor.self_clip to true. Without a clip shape defined the actor will be clipped against the natural shape of the actor, for example if the actor is an ArcActor then the clipping region will be the circle area that is generated by the actor. For rectangular clipping regions a clip margin (defined by b5.Actor.clip_margin which is an array of left,top,right, bottom) can be used to shrink or expand the clipping region. If a clipping shape is defined (b5.Actor.clip_shape) then the supplied shape will be used to clip. Note that clipping can be slow on some devices, so you use it sparingly and should test thoroughly.

Lets take a look at how to set up an actor that clips its children:

```
// Create an actor that will clip its children
var actor = new b5.ArcActor();
actor.fill_style = "#000000";
actor.radius = 50;
actor.filled = true;
actor.clip_children = true;
scene.addActor(actor);

// Create a child actor that will be clipped by its parent
var label = new b5.LabelActor();
label.font = "30pt Calibri";
label.fill_style = "#ffffff";
label.text = "Hello World";
actor.addActor(label);
```

In the above example we create a circle actor that contains text reading "Hello World". Note how the text is clipped against the circle extents of the parent actor.

Now lets take a look at a self clipping example:

```
// Create a clip shape
var shape = new b5.Shape();
shape.type = b5.Shape.TypePolygon;
shape.vertices = [0, -20, 20, 20, -20, 20];

// Create an actor that will clip against the clip shape
var actor = new b5.RectActor();
actor.fill_style = "#000000";
actor.w = 100;
actor.h = 100;
actor.filled = true;
actor.self_clip = true;
actor.clip_shape = shape;
scene.addActor(actor);
```

In the above example we create a triangular polygon shape then create a rectangular actor and assign it is the clipping shape, we also mark the actor as self clipping which clips the actor against its own clip shape.

A good example of self clipping can be seen in the self clipping demo.

Layering

As with scenes, actors can also be visually sorted using layers. Each game object has its own layer number, actors on the same layer will be rendered in the order in which they were created. Actors are sorted local to their parent, so all actors that have the scene as a parent are sorted against each other, whilst all actors in a child tree will be sorted against other actors on the same tree level.

Note that you should always set the layer via the Actor._layer property setter to ensure that layers get re-sorted. A good example of dynamically sorted actors can be seen in the planets demo.

Opacity

Actors have an opacity level (b5.Actor.opacity) that ranges from 0 and 1 which determines how opaque / transparent it its child actors are. Child actors can opt to not be affected by their parents opacity by setting b5.Actor.use_parent_opacity to false.

Note that within the game editor the opacity values ranges from 0 to 255, instead of 0 to 1 and is the 4th parameter of the actors Colour and Selected Colour property.

Filling and Outlines

All shape and text based actors can be rendered as filled or outlined. You can change the filled state of an actor by setting its filled property to true for filled or false for outlined.

Filled actors will use the fill_style property to decide how to render the fill style. Outlined actors will use the stroke_style property to decide how to render the outline. A stoke_thickness can also be specified that specifies how thick to render the outline.

Lets take a quick look at a couple of examples:

```
// Create filled circle
var actor1 = new b5.ArcActor();
actor1.x = -100;
actor1.radius = 100;
actor1.fill_style = "#ff00ff";
actor1.filled = true;
scene.addActor(actor1);

// Create outlined circle
var actor2 = new b5.ArcActor();
actor2.x = 100;
actor2.radius = 100;
actor2.stroke_style = "#ffff00";
actor2.stroke_thickness = 5;
actor2.filled = false;
scene.addActor(actor2);
```

We can also draw an actor filled with a gradient:

```
// Create gradient
var gradient = new b5.Gradient();
gradient.addColourStop("#ff0000", 0);
gradient.addColourStop("#ff00ff", 1);

// Create filled circle
var actor1 = new b5.ArcActor();
actor1.x = -100;
actor1.w = 100;
actor1.radius = 100;
actor1.filled = true;
actor1.fill_style = gradient.createStyle(actor1.w, actor1.w, { x: 0, y: 0 }, { x: 1, y:
1 });
scene.addActor(actor1);
```

A good example of filled and outlined shapes can be seen in the shapes demo.

Docking

Its often useful to be able to arrange parts of a games user interface around the edges of a scene to ensure that they are positioned consistently across different sized displays. Booty5 enables this using docking.

Docking allows you to dock actors to the edges of a scene so that regardless of how the scene is scaled, docked actors will remain where you expect them to be.

You can dock an actor on the x and y axis by setting its b5.Actor.dock_x and b5.Actor.dock_y property to one of the following:

- b5.Actor.Dock_None
- b5.Actor.Dock_Top
- b5.Actor.Dock_Bottom
- b5.Actor.Dock_Left
- b5.Actor.Dock_Right

When an actor is docked it can be adjusted using a margin. b5.Actor.margin specifies an array of margin values (left, right, top, bottom).

In addition to docking against the scene, actors can be docked against the edges of parent actors that are marked as using a virtual canvas. Note that the virtual actor must be a child of the parent for it to dock.

Lets take a quick look at an example that shows how to dock an actor to the top of the scene:

```
// Create an actor that is docked against top of scene
var actor = new b5.RectActor();
actor.fill_style = "#0000ff";
actor.w = 100;
actor.h = 100;
actor.filled = true;
actor.dock_y = 1;
actor.margin = [0, 0, 20, 0];
scene.addActor(actor);
```

Bitmap Animation

Actors can display animating bitmaps out of the box. Animation is achieved by displaying different rectangular areas of a larger image, this image is often called an image atlas (see ImageAtlas). An ImageAtlas is basically a sprite atlas (or sprite sheet) which is a large image that contains multiple sub images. Each sub image can be identified using a rectangular area to specify where it is located in the larger main image. Bitmap animation is achieved by displaying different areas of the main sprite atlas over time, changing the area of the bitmap that is shown.

Lets take a look at how to create a game object that shows a bitmap animation:

```
// Create an actor
var actor = new b5.Actor();
actor.x = 100;
actor.y = 100;
actor.w = 200;
actor.h = 200;
scene.addActor(actor); // Add to scene

// Create an image atlas from a bitmap image then assign as the actors atlas
actor.atlas = new b5.ImageAtlas("sheep", new Bitmap("sheep", "images/sheep.png", true));
actor.atlas.addFrame(0, 0, 86, 89);     // Add frame 1 to the atlas
actor.atlas.addFrame(86, 0, 86, 89);    // Add frame 2 to the atlas
actor.current_frame = 0;                // Set initial animation frame
actor.frame_speed = 1;                  // Set animation playback speed
```

In the above code we create an ImageAtlas and assign it to the actor. We then set the actors frame_speed property to 1 causing the bitmap animation to be played back at speed of 1 frame per second.

The order in which bitmap animations are played can be changed by assigning an array of frame indices to b5.Actor.anim_frames. This array will override the atlases current frame order, e.g.:

actor.anim_frames = [1,0,0,0,1,0,1];

Note that you can set frame_speed to 0 and control the animation manually by changing current_frame.

Adding Physics

All actors of all types can be easily turned into physics objects. To turn an actor into one that supports physics you can need to initialise the actors Box2D physics body then add a fixture to allow collision, e.g.:

```
// Initialise physics body
actor.initBody("dynamic", false, false);

// Add a physics fixture
actor.addFixture({ type: b5.Shape.TypeBox, width: actor.w, height: actor.h });
```

When we call initBody() we pass 3 parameters, the first is the type of body, in this case "dynamic" which means the object can move around the physics world. The second and third tell the physics engine if the body uses fixed rotation and can move very fast respectively, in both cases we opt to turn those features off.

Once the actor has a physics body we can add a fixture. A fixture is a shape that describes the physical shape of the actor and how it reacts to other actors in the simulation. In this case we use a simple box shape that is the same size as the actor. We could pass in additional options to addFixture() such as density, friction, restitution etc..

Note that an actor should not be added to a scene that does not support physics.

Its possible to detect if an actor is under control of physics by checking b5.Actor.body, if not null then the actor is under control of Box2D physics.

Physics materials

Physics materials are represented by the Material class and are used to store physical material parameters for the Box2D engine. A Material can be supplied as an option to b5.addFixture() when creating fixtures, e.g.:

```
actor.addFixture({ material: my_material, shape: my_shape, is_bullet: false });
```

Physics shapes

Physics shapes are represented by the Shape class and are used to store physical dimensions for the Box2D engine. A Shape can be supplied as an option to b5.addFixture() when creating fixtures, e.g.:

```
actor.addFixture({ material: my_material, shape: my_shape, is_bullet: false });
```

Physics joints

Physics joints enable you to connect physical bodies together in a variety of ways. The following types of physical joints are currently supported:

- Weld joint – This is a joint that simply attaches two objects together, when one object moves the other is dragged with it
- Distance joint – A distance joint limits the distance of two bodies and attempts to keep them the same distance apart, damping can also be applied
- Revolute joint – A revolute joint forces two bodies to share a common anchor point. It has a single degree of freedom and the angle between the two bodies can be limited. In addition a motor can also be applied to the joint
- Prismatic joint – A prismatic joint limits movement between the two bodies by translation (rotation is prevented). The translational distance between the two joints can be limited. In addition a motor can also be applied to the joint
- Pulley joint – A pulley joint can be used to create a pulley system between two bodies so that when one body rises the other will fall.
- Wheel joint – A wheel joint restricts one body to the line on another body and can be used to create suspension springs. A motor can also be applied to the joint
- Mouse joint – The mouse joitn can be used to move physical bodies towards a location

Joints can be created by calling b5.Actor.addJoint(options), where options includes information about the specific joint that should be added.

For more details regarding initBody(), addFixture() and addJoint() see the Actor Method reference.

Lets take a look at a quick example of creating a joint that joins two actors:

```
// Create floor
var floor = new b5.RectActor();
floor.fill_style = "#00ff00";
floor.x = 0;
floor.y = 300;
floor.rotation = 0;
floor.w = 500;
floor.h = 100;
floor.filled = true;
scene.addActor(floor);
floor.initBody("static", false, false);
floor.addFixture({ type: b5.Shape.TypeBox, width: floor.w, height: floor.h, restitution: 
0.9, density: 1.0, friction: 0.1 });

// Create actor 1
var actor1 = new b5.RectActor();
actor1.fill_style = "#0000ff";
actor1.x = -50;
actor1.y = -200;
actor1.w = 100;
actor1.h = 100;
actor1.rotation = -1;
actor1.filled = true;
scene.addActor(actor1);
actor1.initBody("dynamic", false, false);
actor1.addFixture({ type: b5.Shape.TypeBox, width: actor1.w, height: actor1.h, restitution: 
```

```
0.9, density: 1.0, friction: 0.1 });

// Create actor 2
var actor2 = new b5.RectActor();
actor2.fill_style = "#ff0000";
actor2.x = 50;
actor2.y = -200;
actor2.w = 100;
actor2.h = 100;
actor2.rotation = 1;
actor2.filled = true;
scene.addActor(actor2);
actor2.initBody("dynamic", false, false);
actor2.addFixture({ type: b5.Shape.TypeBox, width: actor2.w, height: actor2.h, restitution:
0.9, density: 1.0, friction: 0.1 });

// Create joint
actor1.addJoint({ type: "weld", actor_b: actor2, anchor_a: { x: 0, y: 0 }, anchor_b: { x: 0,
y: 0 }, self_collide: true });
```

Physics collision

When two actors that are under the control of the physics system start to collide or stop colliding then their onCollisionStart(contact) and onCollisionEnd(contact) event handlers are called (if specified). This gives you the opportunity to determine when, where and how collisions took place. Lets take a look at a quick example that shows two actors interacting during collision:

```
// Create floor
var floor = new b5.RectActor();
floor.name = "floor";
floor.fill_style = "#00ff00";
floor.x = 0;
floor.y = 300;
floor.rotation = 0;
floor.w = 500;
floor.h = 100;
floor.filled = true;
scene.addActor(floor);
floor.initBody("static", false, false);
floor.addFixture({ type: b5.Shape.TypeBox, width: floor.w, height: floor.h, restitution:
0.9, density: 1.0, friction: 0.1 });

// Create actor 1
var actor1 = new b5.RectActor();
actor1.name = "actor1";
actor1.fill_style = "#0000ff";
actor1.x = -50;
actor1.y = -200;
actor1.w = 100;
actor1.h = 100;
actor1.rotation = -1;
actor1.filled = true;
scene.addActor(actor1);
actor1.initBody("dynamic", false, false);
actor1.addFixture({ type: b5.Shape.TypeBox, width: actor1.w, height: actor1.h, restitution:
0.9, density: 1.0, friction: 0.1 });
actor1.onCollisionStart = function (contact) {
    var actor1 = contact.GetFixtureA().GetBody().GetUserData();
    var actor2 = contact.GetFixtureB().GetBody().GetUserData();
    console.log(actor1.name + " hit " + actor2.name);
};
```

In the above example we create a floor actor and dynamic actor, we assign the onCollisionStart() event handler which will be called when actor1 hits something. You can find out the two colliding actors by checking the user data assigned to the body, which happens to the the actor that contains it.

Applying force and torque

To apply force to a body you need to call ApplyForce() on the actors body, e.g.:

```
var b2Vec2 = Box2D.Common.Math.b2Vec2;
actor.body.ApplyForce(new b2Vec2(fx, fy), pos);
```

Where fx, fy represents the force to apply and pos the physics body position in Box2D world coordinates to apply the force at. You can get the world position of a body as follows:

```
var pos = actor.body.GetWorldPoint(new b2Vec2(0,0));
```

If you want to calculate a position that is offset from the body then you can pass the offset to GetWorldPoint(), e.g.:

```
var ws = actor.scene.world_scale;
var pos = actor.body.GetWorldPoint(new b2Vec2(dx / ws, dy / ws));
```

Note that we scale the offset dx,dy by the physics world scale allowing us to specify the offset in scene coordinates

To apply torque (turning force) to an actors body we can call ApplyTorque() on the actors body, e.g.:

```
actor.body.ApplyTorque(torque);
```

Changing velocity and applying impulses

To set an actors velocity directly you can call SetLinearVelocity() and SetAngularVelocity() on the actors physics body, e.g.:

```
var b2Vec2 = Box2D.Common.Math.b2Vec2;
actor.body.SetLinearVelocity(new b2Vec2(vx, vy));
actor.body.SetAngularVelocity(vr);
```

Here we set the linear velocity to vx,vy and the angular velocity to vr directly.

Note that setting the velocity directly will overwrite the current velocity, if instead you want to apply a new velocity then you should use impulse instead.

To apply impulse to an actors body you should call ApplyImpulse() on the actors body, e.g.:

```
var b2Vec2 = Box2D.Common.Math.b2Vec2;
actor.body.ApplyImpulse(new b2Vec2(fx, fy), pos);
```

As with ApplyForce the impulse can be applied at a specific point on the body.

None Box2D physics

When actors are not under control of the Box2D physics system they have their own linear / angular / depth velocity (vx, vy, vr, vd) and linear / angular / depth velocity damping (vx_damping, vy_damping, vr_damping, vd_damping). This enables you to move objects around under velocity without the need for a full physics simulation, however collision and collision response will not automatically be taken care of for you, you will need to do this yourself by checking for overlap between actors.

Note that when an actor is under control of the Box2D physics system and you find that you need to apply the actors current velocities to the physics system then you can call b5.Actor.updateToPhysics() to force the changes to be written back to the physics system, this will also awaken the actors body if asleep.

Scene extents

When an actor has b5.Actor.wrap_position enabled, if it goes beyond the parent scenes extents then it will wrap back around at the opposite edge.

Actor Animation

Actors can be animated using timelines, a timeline is a collection of Animations that change the properties of objects over time. A timeline can target just about any property of an actor. Actors can carry multiple animations that can be played back at the same time, as well as paused and restarted. The actors timeline manager b5.Actor.timelines is a TimelineManager which allows you to add, remove and find existing animation timelines.

Lets take a look at a quick example of creating an animation timeline that animates a property of an actor:

```
// Create a timeline that targets the x property of my_object with 4 key frames spaced out
every 5 seconds and using QuarticIn easing to ease between each frame
var timeline = new b5.Timeline(actor, "_x", [0, 100, 300, 400], [0, 5, 10, 15], 0,
[b5.Ease.quartin, b5.Ease.quartin, b5.Ease.quartin]);
my_object.timelines.add(timeline); // Add to timeline manager to be processed
```

In the above code we create a timeline with an animation that animates the x coordinate of the actor object. We add the timeline to the actors timeline manager to ensure that it is processed each frame.

Besides bitmap animation and timeline animations, its possible to use velocity to set off simple animations. For example, by setting the b5.Actor vr (rotation velocity) you can set it spinning at a specific speed.

Note that when an animation is finished playing it will automatically be destroyed and cleaned up unless its destroy property is set to false.

Input Events

All actors that are defined as touchable will receive touch input events as long as the scene that contains them is the focus or secondary focus scene. An actor is defined as touchable when the Actor.touchable property is set to true. When the app looks to determine which actor has potentially been touched the fewer actors it has to check the better, so this is a simple optimisation which enables only those actors that need to be checked for touch to be checked, whilst the rest can be comfortably ignored.

Actors can receive the following touch events:

- onTapped(touch_pos) – Called when actor tapped / clicked
- onBeginTouch(touch_pos) – Called when user is touching actor
- onEndTouch(touch_pos) – Called when user stops touching the actor
- onLostTouchFocus(touch_pos) – Called when actor loses touch focus, that is when the user drags their finger off the object but does not lift their finger from the screen
- onMoveTouch(touch_pos) – Called when a touch is moved over the actor

Lets take a look at a quick example:

```
var actor = new b5.ArcActor();
actor.name = "actor1";
actor.fill_style = "#0000ff";
actor.radius = 100;
actor.filled = true;
actor.touchable = true;
scene.addActor(actor);
actor.onBeginTouch = function (touch_pos) {
    console.log("Started touching actor " + this.name);
};
actor.onEndTouch = function (touch_pos) {
    console.log("Stopped touching actor " + this.name);
};
```

Note that in order for a child actor to receive input events its parent must also be touchable, otherwise touch events will not be passed down the chain.

In the case of overlapping actors, only the top most actor will receive the touch event. You can change this behaviour by allowing touch events from child actors to be passed back up to their parents by setting b5.Actor.bubbling to true, the default behaviour is to not pass events up to parents. Event bubbling is useful especially when creating user interface components. For example, in the list menu demo, the buttons that are contained within the scrolling menu actor are marked as bubbling to enable touch events to be passed to the container which in turn allows the user to scroll the list around whilst still touching child buttons.

Lets take a quick look at an example of event bubbling:

```
var actor = new b5.ArcActor();
actor.name = "actor1";
actor.fill_style = "#0000ff";
actor.radius = 100;
actor.filled = true;
actor.touchable = true;
scene.addActor(actor);
actor.onBeginTouch = function (touch_pos) {
    console.log("Started touching actor " + this.name);
};
actor.onEndTouch = function (touch_pos) {
    console.log("Stopped touching actor " + this.name);
};

var child1 = new b5.ArcActor();
child1.name = "child1";
child1.fill_style = "#00ffff";
child1.radius = 50;
child1.filled = true;
child1.touchable = true;
child1.bubbling = true;
actor.addActor(child1);
child1.onBeginTouch = function (touch_pos) {
    console.log("Started touching child actor " + this.name);
};
child1.onEndTouch = function (touch_pos) {
    console.log("Stopped touching child actor " + this.name);
};
```

You will notice that if you click the cyan coloured circle both touch event handlers for actor and child1 will be called.

Depth

Actors can have 3D depth using the b5.Actor.depth property. For values of greater than 0, the actor will be projected into the screen using the centre of the scene as the projection origin. This is ideal for creating effects such as parallax scrolling where objects in the background move more slowly and appear smaller than objects in the foreground. The parallax scrolling and planets demos show good examples of using depth.

Orphans

An orphan is an actor that sits inside another actors child hierarchy but does not obey its visual transform. This is useful in areas of game play where you need to generate game objects that are attached to the actor but do not follow it. To turn an actor into an orphaned actor simple set its b5.Actor.orphaned property to true.

Virtual Canvas

Actors can be made to act like a scrollable container for their children. This can be done by converting the actor into a virtual canvas actor, which gives the actor a virtual area that the user can scroll the children around in. This type of functionality is perfect for various types of user interface elements such as grids and lists.

When an actor has been made virtual using b5.Actor.makeVirtual(), or by setting true for virtual to the Actor constructor the following properties become available:

- scroll_pos_x – Canvas scroll X position
- scroll_pos_y – Canvas scroll Y position
- scroll_vx – Canvas scroll X velocity
- scroll_vy – Canvas scroll Y velocity
- scroll_range – Scrollable range of canvas (left, top, width, height)
- prev_scroll_pos_x – Previous canvas scroll X position
- prev_scroll_pos_y – Previous canvas scroll Y position

Lets take a quick look at a virtual canvas actor example:

```
var actor = new b5.RectActor();
actor.name = "actor1";
actor.fill_style = "#0000ff";
actor.w = 400;
actor.h = 400;
actor.filled = true;
actor.touchable = true;
scene.addActor(actor);
actor.makeVirtual();
actor.scroll_range = [-150, -150, 300, 300];

var child1 = new b5.ArcActor();
child1.name = "child1";
child1.fill_style = "#00ffff";
child1.radius = 50;
child1.filled = true;
child1.touchable = true;
child1.bubbling = true;
actor.addActor(child1);
```

In the above example actor acts as a parent for child1, actor has been converted to a virtual canvas actor which can now scroll its content around. Note that we set the scroll_range after the actor has been made virtual and that the child actor uses bubbling to ensure that touch events are passed up to the actor parent.

In addition to the virtual scrolling area, virtual actors also allow child actors to be docked around their edges using b5.Actor.dock_x and b5.Actor.dock_y.

You can see an demo of a virtual actor in the list menu demo.

Shadows and Composite Operations

Actors can be made to render a drop shadow around their edges by setting b5.Actor.shadow to true. When shadowing is enabled the following properties can be used to adjust how the shadow appears:

- shadow_x, shadow_y – Shadow position offset
- shadow_blur – Amount to blur shadow
- shadow_colour- Colour of shadow (e.g. #ffffff)

Lets take a quick look at an example of how to use shadows:

```
var actor = new b5.RectActor();
actor.fill_style = "#0000ff";
actor.w = 200;
actor.h = 200;
actor.filled = true;
actor.shadow = true;
actor.shadow_x = 20;
actor.shadow_y = 20;
actor.shadow_blur = 2;
actor.shadow_colour = "#000000";
scene.addActor(actor);
```

Shadows should be used sparingly as they can be quite slow to render on some devices.

Actors can be rendered using a variety of different composite operations by setting b5.Actor.composite_op to one of the following:

- source-over- displays the source image over the destination image (default)
- source-atop – displays the source image on top of the destination image. The part of the source image that is outside the destination image is not shown
- source-in – displays the source image in to the destination image. Only the part of the source image that is INSIDE the destination image is shown, and the destination image is transparent
- source-out – displays the source image out of the destination image. Only the part of the source image that is OUTSIDE the destination image is shown, and the destination image is transparent
- destination-over – displays the destination image over the source image
- destination-atop – displays the destination image on top of the source image. The part of the destination image that is outside the source image is not shown
- destination-in – displays the destination image in to the source image. Only the part of the destination image that is INSIDE the source image is shown, and the source image is transparent
- destination-out – displays the destination image out of the source image. Only the part of the destination image that is OUTSIDE the source image is shown, and the source image is transparent
- lighter – displays the source image + the destination image
- copy – displays the source image. The destination image is ignored
- xor – the source image is combined by using an exclusive OR with the destination image

Lets take a quick look at an example of how to use composite operations:

```
var actor1 = new b5.RectActor();
actor1.fill_style = "#0000ff";
actor1.w = 200;
actor1.h = 200;
actor1.filled = true;
scene.addActor(actor1);

var actor2 = new b5.RectActor();
actor2.fill_style = "#ff0000";
actor2.x = 100;
actor2.y = 100;
actor2.w = 200;
actor2.h = 200;
actor2.filled = true;
actor2.composite_op = "lighter";
scene.addActor(actor2);
```

Caching Rendering for Speed

Whilst HTML5 has come along leaps and bounds, it still has speed issues on some mobile devices. I have found that most of these bottle necks are due to rendering shapes, gradient fills and text. To combat this problem caching is built into the actor system. When an actor is marked as cached it is rendered to an off screen HTML5 canvas. When the actor is drawn at a later stage the pre-rendered canvas is drawn instead. This means that gradient fills and shape renders are only done a single time. To mark an actor as cached simply set b5.Actor.cache to true, the next time it is drawn it will be cached. The only downside to this is that once cached you lose some control over the actor, for example when you cache a label you can no longer change the text without firstly destroying the old cache and causing the actor to be re-rendered to the canvas.

Lets take a quick look at an example of how to cache an actor:

```
var actor = new b5.LabelActor();
actor.w = 300;
actor.h = 100;
actor.font = "30pt Calibri";
actor.fill_style = "#000000";
actor.text = "Hello World";
actor.cache = true;
scene.addActor(actor);
```

In the above example, setting the actors cache to true has caused the actor to be cached. Note however that if you do not set the width and height of the actor to a large enough size then the actors content will be clipped. For example, if you change the width of the above actor to 100 you will see that only the central portion of the text will be displayed.

We can take caching one step further using merged caching. Merged caching is the process of merging multiple actors into the same cache, saving on extra memory required for all those off screen canvases. Child actors can be marked as merged via b5.Actor.merge_cache, this causes them to be rendered into a parent cache if one is available. However, when a child actor is merged into its parents cache you can no longer change its position, scale, rotation or opacity.

Tagging

Its sometimes useful to be able to group actors together using some kind of common tag. All actors contain the tag property which is a string that can be used to tag a group of actors that are somehow related. For example, all actors are bullets.

The scene class contains the b5.Scene.removeActorsByTag(tag_name) method which enables you to remove a collection of actors that all have the same tag. This is very useful for removing a complete group of game objects from the scene. For example, if you want to clear up all bullets in a space shooter, tag them all with “bullet” then call scene. removeActorsByTag(“bullet”) to remove them all at the same time.

Ignoring the Camera

By default all game objects will move in relation to the scenes camera. Whilst this is a nice feature, its not always what we want. For example, if you have a number of game objects that are part of the HUD, you will want them to stay still when the camera moves. By setting b5.Actor.ignore_camera to true these actors will not move with the camera.

Particle Systems

A particle system is a special sort of actor that can handle the generation / re-generation of particle actors. To use a particle system actor you create an instance of b5.ParticleActor then create and add individual actor particles, specifying a life span (the amount of time the particle exists), a spawn delay (the amount of time to wait before spawning the particle) and the total number of times the particle can be reborn. When a particle system has no particles left alive it will be destroyed. A ParticleActor is derived from an Actor and inherits all of its properties, functions and so forth from its parent, so it can be moved around, rotated, placed under the control of physics etc..

Lets take a quick look at an example that shows how to create a particle actor:

```
// Create particles actor
var particles = new b5.ParticleActor();
scene.addActor(particles);

// Create and add 50 particles
for (var t = 0; t < 50; t++)
{
    var particle = new b5.ArcActor();
    particle.atlas = scene.findResource("lives", "brush");
    particle.radius = 50;
    particle.fill_style = "#ffff00";
    particle.vx = Math.random() * 100 - 50;
    particle.vy = -200;
    particle.vo = -2;
    particle.vsx = -1;
    particle.vsy = -1;
    particles.addParticle(particle, 1, 1, t * 0.05);
}
```

A number of utility methods within the ParticleActor class are available for creating different types of effects easier:

- b5.ParticleActor generateExplosion() - Generates an explosion particles actor
- b5.ParticleActor generatePlume - Generates a a smoke / fire plume particles actor
- b5.ParticleActor generateRain - Generates a rain / snow particles actor

Lets take a quick look at an example that shows how to create an explosion one of the utility methods:

```
// Create explosion
var particles = new b5.ParticleActor();
scene.addActor(particles);
particles.generateExplosion(10, b5.ArcActor, 1, 200, 1, 10, 1, {
    fill_style: "#ffa0e0",
    radius: 30,
    vsx: 1.5,
    vsy: 1.5,
});
```

The last property passed to generateExplosion() is a properties list that will be transferred to all generated particles.

Miscellaneous

Testing for actor overlap

You can test if two actors overlap by calling b5.Actor.overlaps(other_actor). Note that this method currently does a simple test and does not take into account shape, orientation, origin or scale.

Transforming a point by an actors transform

You can transform an x,y point by an actors current transform using b5.Actor.transformPoint(x, y)

Test a point for actor hit

You can test to see if an x,y point hits an actor by calling b5.Actor.hitTest(position), where position is {x, y}

Actor Public Properties

- name – Name of actor (used to find game objects in the scene)
- tag – Tag (used to find groups of game objects in the scene)
- id – User defined ID
- active – Active state, will not be updated when inactive (update method will not be called)
- visible – Visible state, will not be draw if invisible (draw method will not be called)
- touchable – Touchable state, true if can be touched / clicked and will receive touch events
- layer – Visible sorting layer
- x – X position in scene
- y – Y position in scene
- w – Visual width
- h – Visual height
- ox – X origin
- oy – Y origin
- absolute_origin – If true then ox and oy are taken as absolute coordinates, if false then ox and oy are taken as percentage of width and height
- rotation – Rotation in radians
- scale_x – X scale
- scale_y – Y scale
- depth – Z depth (3D depth), 0 represents no depth
- opacity – Opacity (between 0 and 1)
- use_parent_opacity – Scale opacity by parent opacity fi true
- current_frame – Current bitmap animation frame
- frame_speed – Bitmap animation playback speed in seconds
- anim_frames – An array of frame indices, if set then current frame will be read from this array
- bitmap – Bitmap (used if no atlas defined)
- atlas – Image atlas
- vr – Rotational velocity (when no physics body attached)
- vx – X velocity (when no physics body attached)
- vy – Y velocity (when no physics body attached)
- vd – Depth velocity, rate at which depth changes
- vr_damping – Rotational damping (when no physics body attached)
- vx_damping – X velocity damping (when no physics body attached)
- vy_damping – Y velocity damping (when no physics body attached)
- vd_damping – Depth velocity damping
- ignore_camera – If set to true them then this game object will not use camera translation
- wrap_position – If true then game object will wrap at extents of scene
- dock_x – X-axis docking (0 = none, 1 = left, 2 = right)
- dock_y – Y-axis docking (0 = none, 1 = top, 2 = bottom)
- margin – Margin to leave around game object when it has been docked [left, right, top, bottom]
- bubbling – If true then touch events will be allowed to bubble up to parents
- clip_children – If set to true then children will be clipped against extents of this one
- clip_margin – Margin to leave around clipping [left, top, right, bottom]
- clip_shape – If set then this shape will be used to clip, otherwise the usual extents / shape of this game object will be used to clip

- self_clip – If set to true then this game object will be clipped against its own clipping
- orphaned – If true then will ignore parent transforms
- virtual – Set to true if supports a virtual canvas
- anim_frames – Array of animation frame indices, if not set then frames will be used in order specified in the atlas
- shadow – When set to true this object will show a shadow
- shadow_x, shadow_y – Shadow offset
- shadow_blur – Amount to blur shadow
- shadow_colour – Colour of shadow (e.g #ffffff)
- composite_op – Composite operation to use when drawing this object (e.g source-over)
- cache – When set to true this actors rendering will be cached (does not apply down the entire child hierarchy)
- merge_cache – When set to true, will try to render this object into a parent cache, if one is available

Actor Internal Properties

- type – Actor type
- scene – Parent Scene
- parent – Parent actor (If null then this game object does not belong to an hierarchy)
- actors – Array of child actors
- removals – Array of actors that should be deleted at end of frame
- joints – Array of physics joints that are attached
- timelines – A timeline manager that contains and manages animations local to this game object
- actions – An actions list manager that contains and manages actions local to this game object
- frame_count – Number of frames that this object has been running
- accum_scale_x – Accumulated X scale
- accum_scale_y – Accumulated Y scale
- accum_opacity – Accumulative opacity
- body – Box2D body
- transform – Current transform
- transform_dirty – If set to true then transforms will be rebuilt next update
- touching – Set to true when user is touching
- touchmove – Set to true when touch is moving on this game object
- layer – The visible layer that this object sits on
- order_changed – When layer changes this property is marked as true to let the system know to resort all objects on the layer
- cache_canvas – A canvas that contains cached drawing for this actor

Virtual Actor Properties

These properties are only available if created as a virtual canvas or is made to support a virtual canvas using makeVirtual()

- scroll_pos_x – Canvas scroll X position
- scroll_pos_y – Canvas scroll Y position
- scroll_vx – Canvas scroll X velocity
- scroll_vy – Canvas scroll Y velocity
- scroll_range – Scrollable range of canvas (left, top, width, height)
- prev_scroll_pos_x – Previous canvas scroll X position
- prev_scroll_pos_y – Previous canvas scroll Y position

Actor Setters

- _x – Sets the x property
- _y – Sets the y property
- _ox – Sets the ox property
- _oy – Sets the oy property
- _rotation – Sets the rotation property
- _scale – Sets both x and y scale property to the same value
- _scale_x – Sets the scale_x property
- _scale_y – Sets the scale_y property
- _depth – Sets the depth property
- _layer – Sets visible layer
- _atlas – Sets the ImageAtlas from a path to or an instance of an ImageAtlas
- _bitmap – Sets the Bitmap from a path to or an instance of a Bitmap
- _clip_shape – Sets the slipping shape from a path to or an instance of a Shape

Note: Use these setters instead of the likes of x, y, ox, oy etc.. to ensure that this game objects internal transform gets updated.

Actor Constants

- Actor.Dock_None – No docking
- Actor.Dock_Left – Docks to the left edge
- Actor.Dock_Right – Docks to the right edge
- Actor.Dock_Top – Docks to the top edge
- Actor.Dock_Bottom – Docks to the bottom edge

Actor Events

- onCreate() – Called just after creation
- onDestroy() – Called just before destruction
- onTick(delta_time) – Called each update (every frame)
- onTapped(touch_pos) – Called when tapped / clicked
- onBeginTouch(touch_pos) – Called when user is touching
- onEndTouch(touch_pos) – Called when user stops being touching
- onLostTouchFocus(touch_pos) – Called when object loses touch focus
- onMoveTouch(touch_pos) – Called when a touch is moved over the game object
- onCollisionStart(contact) – Called when this game object starts colliding with another
- onCollisionEnd(contact) – Called when this game object stops colliding with another

Actor Methods

- Actor(virtual) – Creates instance of an Actor object
  - virtual – If set to true then the actor will have a virtual canvas attached that can scroll / dock its child actors
- setPosition(x, y) – Sets the scene position (causes internal transform to be recalculated if values differ to existing values).
  - x – X axis position
  - y – Y axis position
- setPositionPhysics(x, y) – deprecated, use set setPosition instead
- setOrigin(x, y) – Sets the origin (causes internal transform to be recalculated if values differ to existing values).
  - x – X axis origin
  - y – Y axis origin
- setScale(x, y) – Sets the scale (causes internal transform to be recalculated if values differ to existing values).
  - x – X axis scaling factor
  - y – Y axis scaling factor
- setRotation(angle) – Sets the rotation angle (causes internal transform to be recalculated if values differ to existing values).
  - angle – Rotation in radians
- setRotationPhysics(angle) – Deprecated, use setRotation instead
- setDepth(depth) – Sets the actors 3D depth (causes internal transform to be recalculated if values differ to existing values). Actors with a depth value of anything other than 0 and 1 will undergo perspective transformation, the affect of this is that the actor will be scaled in size depending upon distance, the actors position will also be adjusted to take into account its depth.
  - depth – Depth value, 0 is the same as 1.0, greater values will project the actor further into the distance
- release() – Releases the actor, this is called by the scene or actor systems when an actor has been destroyed.
- destroy() – Begins the destruction of this game object, note that it will not actually be destroyed until the end of the scene or parent actors processing.
- changeParent(parent) – Changes the actors parent to the new parent
  - parent – The new parent
- addActor(actor) – Adds an actor to this actors list of children

  - actor – The actor to add
- removeActor(actor) – Removes the specified actor from this actors list of children
  - actor – The actor to remove
- removeActorsWithTag(tag) – Removes all child actors that have the specified tag.
  - tag – A string tag
- cleanupDestroyedActors() – Cleans up all destroyed actors, this is called by the actor to clean up any removed actors at the end of its update cycle.
- findActor(name, recursive) – Searches child list for the named actor.
  - name: The name of the actor
  - recursive – If true then the complete child hierarchy will be searched
  - returns the found actor or null if not found
- findFirstParent() – Travels up the child hierarchy to return the very first Actor parent
- updateParenTransforms() – Travels up the child hierarchy updating all parent transforms
- bringToFront() – Moves the actor to the end of the child list, bringing it to the front of all others in the parent.
- sendToBack() – Moves the actor to the start of the child list, pushing it to the back of all others in the parent.
- releaseBody() – Releases the attached physics body.
- releaseJoints() – Releases all attached physics joints.
- initBody(body_type, fixed_rotation, is_bullet) – Creates and attached a physics body to this actor, putting it under control of the Box2D physics system.
  - body_type – Type of body, can be either static, dynamic or kinematic
  - fixed_rotation – true if you want to fix rotation so that it cannot change
  - is_bullet – true if this is a very fast moving game object
- addFixture(options) – Creates a new fixture and attaches it to the physics body.
  - options – An object describing the fixtures properties
    - type – Type of fixture (Shape.TypeBox, Shape.TypeCircle or Shape.TypePolygon)
    - density – Fixture density
    - friction – Fixture friction
    - restitution – Fixture restitution
    - is_sensor – true if this fixture is a sensor
    - width / height – Width and height of box (if type is box), or width is radius for circle type
    - vertices – An array of vertices of form [x1, y1, x2, y2 etc] that define a shape (if type is polygon)
    - material – A Material resource, if passed then density, friction, restitution will be taken from the material resource
    - shape – A Shape resource, if passed then width, height, type and vertices will be taken from the shape resource
  - returns the created fixture or in the case of a multi-fixture shape an array of fixtures
- addJoint(options) – Creates a new joint and attaches it to the physics body.
  - options
    - type – Type of joint to create (weld, distance, revolute, prismatic, pulley, wheel, mouse)
    - actor_b – The other actor that this joint attaches to
    - anchor_a – The joints anchor point on this body
    - anchor_b – The joints anchor point on actor_b's body
    - self_collide – If set to true then actors that are connected via the joint will

    collide with each other
        - frequency – Oscillation frequency in Hertz (distance joint)
        - damping – Oscillation damping ratio (distance and wheel joints)
        - limit_joint – If true then joint limits will be applied (revolute, prismatic, wheel joints)
        - lower_limit – Lower limit of joint (revolute, prismatic, wheel joints)
        - upper_limit – Upper limit of joint (revolute, prismatic, wheel joints)
        - motor_enabled – if true then the joints motor will be enabled (revolute, prismatic, wheel joints)
        - motor_speed – Motor speed (revolute, prismatic, wheel joints)
        - max_motor_torque – Motors maximum torque (revolute joints)
        - max_motor_force – Motors maximum force (prismatic, wheel, mouse joints)
        - axis {x,y} – Movement x,y axis (prismatic, wheel joints)
        - ground_a {x,y} – Ground offset for this actor (pulley joints)
        - ground_b {x,y} – Ground offset for actor_b (pulley joints)
    - returns the created joint
- removeJoint(joint) – Removes the specified joint from the joints list and destroys the joint.
    - joint – The joint to remove and destroy
- updateTransform() – Forces the internal visual transform to update, call internally when the transform is dirtied.
- draw() – Draws this actor and its children, this method can be overriden by derived actors
- drawToCache() – Draws this actor to a cached canvas, subsequent calls to draw() will cause this cached canvas to be drawn instead
- preDraw() – Pre drawing shared functionality
- postDraw() – Post drawing shared functionality
- baseUpdate(dt) – Base update function that should be called by all derived actors that wish to use base actor functionality, this is usually called from your update() method.
    - dt – The amount of time that has passed since last updated
    - returns true if active
- update(dt) – Updates the actor, this method can be overriden by derived actors
    - dt – The amount of time that has passed since the last update
    - returns true if active
- updateToPhysics() – Copies actor velocities to the physics system
- dirty() – Dirties the game object and all child transforms
- hitTest(position) – Performs a test of the specified position against the boundaries of the actor returning true if a hit occurs
    - position – The 2D position to test
- transformPoint(x, y) – Transforms the point into the actors coordinate system
    - x,y – The 2D position to test
    - returns the transformed point as a point object {x, y}
- overlaps(other) – Tests if two actors overlap
    - other – The other actor to test for overlap
    - returns true if both overlap
- makeVirtual() – Attaches a virtual canvas
- setClipping(context, x, y) – Used internally to set the clipping shape for this actor
    - context – Display context
    - x – x coordinate of clip
    - y – y coordinate of clip

Actor Examples

Basic actor creation

```
var actor = new b5.Actor();
actor.x = 100;
actor.y = 100;
actor.w = 200;
actor.h = 200;
scene.addActor(actor); // Add to scene
```

Adding a bitmap image to an actor

```
actor.bitmap = new b5.Bitmap("background", "images/background.jpg", true);
```

Adding a bitmap image from the scene resources to an actor

```
actor.bitmap = scene.findResource("background", "bitmap");
```

Adding a bitmap image from the scene resources to an actor using a path

```
actor._bitmap = "scene1.bitmap";
```

Adding basic physics to an actor

```
actor.initBody("dynamic", false, false);    // Initialise physics body
actor.addFixture({type: b5.Shape.TypeBox, width: actor.w, height: actor.h}); // Add a
physics fixture
```

Adding a physics joint

```
actor.addJoint({ type: "weld", actor_b: actor2, anchor_a: { x: 0, y: 0 }, anchor_b: { x: 0,
y: 0 }, self_collide: true });
```

Adding bitmap animation to an actor

```
// Create an image atlas from a bitmap image
actor.atlas = new b5.ImageAtlas("sheep", new b5.Bitmap("sheep", "images/sheep.png", true));
actor.atlas.addFrame(0,0,86,89);     // Add frame 1 to the atlas
actor.atlas.addFrame(86,0,86,89);    // Add frame 2 to the atlas
actor.frame = 0;                     // Set initial animation frame
actor.frame_speed = 0.5;             // Set animation playback speed
```

Add a child actor

```
var child_actor = new b5.Actor();    // Create child actor
actor.addActor(child_actor);      // Add as child actor
```

Adding an onTick event handler to an actor

```
actor.onTick = function(dt) {
    this.x++;
};
```

Adding touch event handlers to an actor

```
actor.touchable = true ;          // Allow actor to be tested for touches
actor.onTapped = function(touch_pos) {
    console.log("Tapped");
};
actor.onBeginTouch = function(touch_pos) {
    console.log("Touch begin");
};
actor.onEndTouch = function(touch_pos) {
    console.log("Touch end");
};
actor.onMoveTouch = function(touch_pos) {
    console.log("Touch move");
};
```

Docking an actor to the edges of the scene

```
actor.dock_x = b5.Actor.Dock_Left;
actor.dock_y = b5.Actor.Dock_Top;
actor.ignore_camera = true;
```

Adding an actions list

```
var actions_list = new b5.ActionsList("moveme1", 0);
actions_list.add(new b5.A_MoveWithSpeed(actor,100,2,b5.Ease.linear));
actor.actions.add(actions_list).play();
```

Create a particle explosion

```
var particles = new b5.ParticleActor();
scene.addActor(particles);
particles.generateExplosion(10, b5.ArcActor, 1, 200, 1, 10, 1, {
    fill_style: "#ffa0e0",
    radius: 30,
    vsx: 1.5,
    vsy: 1.5,
});
```

ArcActor Properties

- fill_style – Style used to fill the arc
- stroke_style – Stroke used to draw none filled arc
- stroke_thickness – Thickness of line stroke
- radius – Radius of arc
- start_angle – Start angle of arc in radians
- end_angle – End angle of arc in radians
- filled – if true then arc interior will filled otherwise empty

ArcActor Methods

- ArcActor() – Creates an instance of an ArcActor object
- draw() – Draws an arc actor and its children, this method can be overridden by derived actors
- drawToCache() – Draws this actor to a cached canvas, subsequent calls to draw() will cause this cached canvas to be drawn instead
- update(dt) – Updates the actor, this method can be overridden by derived actors
  - dt – The amount of time that has passed since this actor was last updated
- hitTest(position) – Performs a test of the specified position against the circular boundaries of the actor returning true if a hit occurs
  - position – The 2D position to test

LabelActor Properties

- text – The text to display
- font – The font to display the text in
- text_align – Text horizontal alignment
- text_baseline – Text vertical alignment
- fill_style – Fill style for filled text
- stroke_style – Stoke style for none filled text
- stroke_thickness – Thickness of line stroke
- filled – When true filled text will be drawn

LabelActor Methods

- LabelActor() – Creates an instance of a LabelActor
- draw() – Draws an arc actor and its children, this method can be overridden by derived actors
- drawToCache() – Draws this actor to a cached canvas, subsequent calls to draw() will cause this cached canvas to be drawn instead
- update(dt) – Updates the actor, this method can be overridden by derived actors
  - dt – The amount of time that has passed since this actor was last updated
- draw() – Draws the actor, this version is specific to drawing text instead of images (called internally)

ParticleActor Properties

gravity – The amount of gravity to apply to particles within the system
onParticlesEnd() – This callback function is called when the particle actor runs out of particle
onParticleLost(particle) – This callback function is called each time a particle is about to be destroyed. Returning false will prevent the particle from being destroyed allowing it to be re-used

Additional particle properties

- vo – Opacity velocity
- vsx – X axis scale velocity
- vsy – Y axis scale velocity

ParticleActor Methods

- ParticleActor() – Creates an instance of a ParticleActor
- resetParticle(actor) – Resets the particle to its initial spawn state (used internally)
- addParticle(actor, life_span, num_lives, spawn_delay) – Adds an actor to the particle system as a particle
  - actor – The actor to add as a particle
  - life_span – The life span of the particle in seconds
  - num_lives – The number of times a particle will re-spawn before it is destroyed (0 for infinite)
  - spawn_delay – The number of seconds to wait before spawning this particle
  - Returns the created particle actor
- update(dt) – Updates the actor, this method can be overridden by derived actors
  - dt – The amount of time that has passed since this actor was last updated
- generateExplosion(count, type, duration, speed, spin_speed, rate, damping, properties) – Utility method that automatically generates an explosion type particle system
  - count – Total number of particles to create
  - type – The actor type of each particle created, for example b5.ArcActor or “ArcActor”
  - duration – The total duration of the particle system in seconds
  - speed – The speed at which the particles blow apart
  - spin_speed – The speed at which particles spin
  - rate – rate at which particles are created
  - damping – A factor to reduce velocity of particles each frame, values greater than 1 will increase velocities
  - properties – A collection of actor specific properties that will be assigned to each created particle
- generatePlume(count, type, duration, speed, spin_speed, rate, damping, properties) - Utility method that automatically generates a smoke plume type particle system
  - count – Total number of particles to create
  - type – The actor type of each particle created, for example b5.ArcActor or “ArcActor”
  - duration – The total duration of the particle system in seconds

  - speed – The speed at which the particles rise
  - spin_speed – The speed at which particles spin
  - rate – rate at which particles are created
  - damping – A factor to reduce velocity of particles each frame, values greater than 1 will increase velocities
  - properties – A collection of actor specific properties that will be assigned to each created particle
- generateRain(count, type, duration, speed, spin_speed, rate, damping, width, properties) – Utility method that automatically generates weather effects such as rain
  - count – Total number of particles to create
  - type – The actor type of each particle created, for example b5.ArcActor or "ArcActor"
  - duration – The total duration of the particle system in seconds
  - speed – The speed at which the particles fall
  - spin_speed – The speed at which particles spin
  - rate – rate at which particles are created
  - damping – A factor to reduce velocity of particles each frame, values greater than 1 will increase velocities
  - width – The width of the area over which to generate particles
  - properties – A collection of actor specific properties that will be assigned to each created particle

PolygonActor Properties

- fill_style – Style used to fill the arc
- stroke_style – Stroke used to draw none filled arc
- stroke_thickness – Thickness of line stroke
- filled – if true then arc interior will filled otherwise empty
- points – An array of points that describe the shape of the actor in the form [x1,y1,x2,y2,....]

PolygonActor Methods

- PolygonActor() – Creates an instance of a PolygonActor object
- draw() – Draws a polygon actor and its children, this method can be overridden by derived actors
- drawToCache() – Draws this actor to a cached canvas, subsequent calls to draw() will cause this cached canvas to be drawn instead
- update(dt) – Updates the actor, this method can be overridden by derived actors
  - dt – The amount of time that has passed since this actor was last updated

RectActor Properties

- fill_style – Style used to fill the rect
- stroke_style – Stroke used to draw none filled rect
- stroke_thickness – Thickness of line stroke
- filled – if true then rect interior will filled otherwise empty
- corner_radius – Radius of corner for rounded rects

RectActor Methods

- RectActor() – Creates an instance of a RectActor object
- draw() – Draws a rect actor and its children, this method can be overridden by derived actors
- drawToCache() – Draws this actor to a cached canvas, subsequent calls to draw() will cause this cached canvas to be drawn instead
- update(dt) – Updates the actor, this method can be overridden by derived actors
  - dt – The amount of time that has passed since this actor was last updated

Resources – The Stuff that Games are Made of

Introduction

One of the central features of Booty5 is the management of resources. Resources are assets that the game needs to help it function such as bitmaps and sound effects. Booty5 supports a number of different types of resources including:

- Bitmaps – Bitmaps are represented by bitmap files
- Sounds – Sounds are represented by audio files
- Brushes – Brushes represent something that can draw a game object, such as an ImageAtlas or a Gradient
- Shape – Shapes are geometric shapes that can be used for clipping, rendering, physics fixtures and paths
- Materials – Materials represent physical materials that can affect how physics objects behave and interact

Resources can be stored and managed locally by scenes or globally by the main App object. Local resources are loaded with the scene and destroyed when the scene is destroyed, whilst global resources are loaded and managed by the global App object and will remain in memory for the duration of the application or until removed manually. This type of resource management system is useful when dealing with games that run on resource constrained devices such as mobile phones.

Each resource has its own name which should be unique to its category of resource that can be used to find the resource at a later date. For example, you should not have two images called "player", although you can have an image called player and a shape called "player".

You can add a resource to a scene or the main App by calling addResource(), e.g.:

```
// Create a bitmap resource
var bitmap = new b5.Bitmap("background", "images/background.jpg", true);

// Add to a scene to be managed
scene.addResource(bitmap, "bitmap");
```

Resources can be manually removed from a scene or the main App by calling removeResource(), e.g.:

```
scene.removeResource(bitmap, "bitmap");
```

You can also call destroy() on the resource to remove and destroy it, e.g.:

```
bitmap.destroy();
```

Once a resource has been added to a resource manager it can later be searched for using findResource(), e.g.:

```
var bitmap = scene.findResource("background", "bitmap");
```

Note that if you call the scene version of findResource() and the resource is not found in the scene then it will automatically check the global App's resource list.

Resource Paths

All resources in Booty5 can be located using their child hierarchy path. A child hierarchy path is a string that consists of the names of the parents separated by dots. For example, if you want to locate a physics material called "material1" that is located in a scene named "scene1" then the path to that resources would be "scene1.material1". If the material was located in the Apps global resource space then the path would simply be "material1". To find the instance of the resource from the path you can call b5.Utils.findResourceFromPath(path, type), e.g.:

```
var material = b5.Utils.findResourceFromPath("scene1.material1", "material");
```

This method of locating a resource is so convenient that many parts of the Booty5 engine utilise it. For example, all resources that are passed to actions accept either an instance of a resource or a path to the resource. Many object properties also accept paths, such as b5.Actor._clip_shape and b5.Actor.bitmap etc...

Scenes, actors, timelines and actions lists can also be searched for using paths using the b5.Utils.findObjectFromPath(path, type), where type can be timeline, actions or null for scene / actor.

Bitmap Resources

Bitmap resources are resources that hold image data that is loaded from a file. Bitmaps are commonly used as the visual representation for sprites (Actors). To create a bitmap you create an instance of a b5.Bitmap class, for which the prototype is:

b5.Bitmap(name, location, preload, onload)

Where name is the name of the Bitmap object, location is the location of the bitmap, preload is a flag that tells Booty5 to load the image immediately and onload is an optional callback function which will be called when the bitmap has finished loading.

Lets take a quick look at an example:

```
var bitmap = new b5.Bitmap("player", "images/player_anims.png", true);
```

Lets take a quick look at an example of how to create a Bitmap that notifies us when it has finished loading:

```
var bitmap = new b5.Bitmap("player", "Textures.png", true, function (b) {
    console.log("bitmap loaded " + b.name);
});
```

You can check to see if a Bitmap has finished loading by checking its loaded property.

You can tell a Bitmap to load itself by calling b5.Bitmap.load().

Bitmaps are usually assigned to an Actor via its bitmap property or _bitmap property setter. Remember that a property setter can take an instance of a resource or a path to a resource.

If you intend to manage the Bitmap yourself then you do not need to add the Bitmap to a scene or the main App objects resource managers.

Bitmaps can also be assigned to an ImageAtlas brush if the bitmap contains multiple sub-images. More on this later.

Sound Resources

Sound resources are resources that hold audio data that is loaded from a file. Sounds are commonly used to play back music and sound effects. To create a sound you create an instance of a b5.Sound class, for which the prototype is:

b5.Sound(name, location, reuse)

Where name is the name of the Sound object, location is the location of the sound file, reuse is a flag that tells Booty5 that it should be created only once and re-used.

Lets take a quick look at an example:

```
var sound = new b5.Sound("explosion", "sounds/explosion.mp3", true);
```

Sounds are usually added to a scene or the main App object for management

Because not all sound file types are supported across all platforms, Booty5 supports a fallback option when loading sound effects by assigning a second sound effect to the Sound object, if the first sound fails to load then Booty5 will attempt to load the second,e.g.:

```
var sound = new b5.Sound("explosion", "sounds/explosion.ogg", true);
sound.location2 = "sounds/explosion.mp3";
```

In the above example, if the ogg sound file fails to load then Booty5 will attempt to load the mp3 instead.

You can tell a Sound to load itself by calling b5.Sound.load().

You can play, stop and pause a playing sound by calling b5.Sound.play(), b5.Sound.stop() and b5.Sound.pause(). When you call play() an instance of the created HTML5 Audio object (or equivalent) will be returned. You can also get access to this instance via b5.Sound.snd property. Note that in the case of playing multiple copies of the same sound, a different instance of Audio will be returned for each one if the sound is not set as reuse.

The sound can be told to loop by setting b5.Sound.loop to true.

Shape Resources

Shape resources are resources that hold information about a geometric shape or path. Unlike bitmaps and sounds, shapes are not stored in separate files. Shapes are commonly used as the geometric visual representation for sprites, as fixtures for physics bodies, as clipping regions for actors and scenes or as paths for objects to follow. To create a Shape you create an instance of a b5.Shape class, for which the prototype is:

b5.Shape(name)

Where name is the name of the Shape object. However, this only creates an instance of a Shape object, we still need to fill in the rest of the information.

Shapes can be of 3 different kinds:

- b5.Shape.TypeBox – A rectangular box with width and height
- b5.Shape.TypeCircle – A circular shape with a radius
- b5.Shape.TypePolygon – A polygonal shape made from vertices

All 3 types can be used for rendering, physics fixtures and as clippers. However, only polygon shapes can be used as paths.

Lets take a quick look at an example that shows how to create a shape of each type:

```
// Create a box shape
var box = new b5.Shape();
box.type = b5.Shape.TypeBox;
box.width = 100;
box.height = 50;

// Create a circle shape
var circle = new b5.Shape();
circle.type = b5.Shape.TypeCircle;
circle.width = 100;

// Create a polygon shape
var polygon = new b5.Shape();
polygon.type = b5.Shape.TypePolygon;
polygon.vertices = [0, -100, 100, 100, -100, 100];
```

Material Resources

Material resources are resources that hold information about physical properties of an object. Like Shapes, Materials are not stored in separate files. To create a Material you create an instance of a b5.Material class, for which the prototype is:

b5.Material(name)

Where name is the name of the Material object. However, this only creates an instance of a Material object, we still need to fill in the rest of the information.

Materials carry the following pieces of information:

- type – Type of material (can be static, dynamic or kinematic), default is static
- density – Material density, higher values make for heavier objects, default is 1
- friction – Material friction, lower values make objects more slippery, default is 0.1
- restitution – Material restitution, higher values make the object more bouncy, default is 0.1
- gravity_scale – Gravity scale, lower values lessen the affects of gravity on the object, default is 1
- fixed_rotation – Set to true to prevent objects from rotating, default is false

You can use material properties when you add physics fixtures to an Actor, e.g.:

```
// Add a physics fixture from a shape and material
actor.addFixture({ shape: circle_shape1, material: material1 });
```

Brush Resources

Brush resources are resources that are used to fill shapes and create bitmap animations. Booty5 currently supports two kinds of brushes:

- Gradient – A gradient brush is used to create fill styles that draw gradients on shape and text based actors (ArcActor, RectActor, PolygonActor and LabelActor)
- ImageAtlas – An ImageAtlas brush is used to draw different bitmap animations frames on an Actor

Note that all brush types use the resource type "brush", so when searching for a Gradient or ImageAtlas resource ensure that you search for the "brush" type.

Gradient

The Gradient is a an object that stores gradient information. A gradient is made up of a number of colour stops which define a colour and distance along the gradient at which the colour should appear. For example, if I specify a colour at distance 0 then this colour will appear at the start of the gradient. If I specify a distance of 1 then the colour will appear at the end of the gradient. All colour in between will be smoothly interpolated to create the gradient. Lets take a quick look at an example that shows how to create a gradient:

```
var gradient = new b5.Gradient();
gradient.addColourStop("#ff0000", 0);
gradient.addColourStop("#00ff00", 0.5);
gradient.addColourStop("#0000ff", 1);
```

In the above example, we create a gradient with 3 colour stops:

- A gradient stop with colour red at distance 0 (start of gradient)
- A gradient stop with colour green at distance 0.5 (half way through the gradient)
- A gradient stop with colour blue at distance 1 (end of gradient)

Once a Gradient has been created you should add it to a Scene or App resource manager so that it can be managed.

Once we have a gradient brush we can create a fill style from it and assign that to an Actor to be rendered, e.g.:

```
// Create fill style from gradient
var fill_style = gradient.createStyle(actor1.w, actor1.w, { x: 0, y: 0 }, { x: 1, y: 1 });
```

In the above example we call the b5.Gradient.createStyle() method to generate a fill style. createStyle takes 4 parameters:

- w – Width of the gradient area (usually width of fill area)
- h – Height of the gradient area (usually height of fill area)
- start – Start point of gradient {x, y}
- end – End point of gradient {x, y}

To see the gradient we set it to either the fill_style or stoke_style of the actor.

```
// Create an actor and assign the gradient fill style
var actor = new b5.ArcActor();
actor.x = -100;
actor.w = 100;
actor.radius = 100;
actor.filled = true;
actor.fill_style = fill_style;
scene.addActor(actor);
```

ImageAtlas

Booty5 creates bitmap animations by displaying one image after another, or more accurately one area of an image after another. The ImageAtlas brush is used to store bitmap frame information as rectangular areas as well as a sprite sheet based bitmap image. Each frame of animation has an x,y location in the bitmap and a frame width and height. Animation is achieved by displaying those different areas over time.

Lets take a quick look at how to create an ImageAtlas:

```
var atlas = new b5.ImageAtlas("sheep", sheep_bitmap);
atlas.addFrame(0, 0, 86, 89);     // Add frame 1 to the atlas
atlas.addFrame(86, 0, 86, 89);    // Add frame 2 to the atlas
```

In the above example we create an ImageAtlas object called sheep using the bitmap sheep_bitmap. We then add two frames, the first is located at x=0,y=0 in the bitmap and the second is located at x=86, y=0, both frames are the same size 86x89 pixels.

Once an ImageAtlas has been created you should add it to Scene or App resource manager so that it can be managed.

We can now assign this atlas brush to an actor to make it animate. Lets take a quick look at an example:

```
var actor = new b5.Actor();
actor.w = 86;
actor.h = 89;
scene.addActor(actor);
actor.atlas = atlas;
actor.current_frame = 0;                // Set initial animation frame
actor.frame_speed = 1;                  // Set animation playback speed
```

In the above example we assign the atlas to the atlas property then set the frame_speed to begin the playback of the bitmap animation.

The b5.ImageAtlas.generate() method has been added to ImageAtlas to enable easy automatic generation of frames. Lets take a quick look at an example of how to use it:

```
var atlas = new b5.ImageAtlas("car_anim", car_bitmap);
atlas.generate(0, 0, 64, 32, 10, 20);
```

The above example will generate 20 animation frames each 64x32 pixels in size, starting at x=0,y=0 on the bitmap, picking 10 frames across the bitmap then moving down 32 pixels then picking another 10 frames.

Bitmap Properties

- name – The name of the bitmap, this name is used when searching for bitmap resources in a Scene’s resources or the the App’s global resources
- location – The location of the bitmap file that is used to create the image
- parent – The parent container
- onload – Callback function which will be called when the bitmap has been loaded
- loaded – Set to true when the bitmap has been loaded
- Private properties
- image – An image object that represents the bitmap

Bitmap Methods

- Bitmap(name, location, preload, onload) – Creates an instance of a Bitmap object.
- name – Name of the bitmap
- location – The bitmap file location
- preload – If set to true then the bitmap will be loaded when created
- onload – Callback function that will be called when the image has been loaded
- load() – Loads the bitmap, used in cases where the bitmap is not preloaded
- destroy() – Removes the bitmap from the scene / app and destroys it

Sound Properties

- name – The name of the sound, this name is used when searching for sound resources in a Scene’s resources or the the App’s global resources
- parent – Parent container scene or app
- location – The location of the sound file that is used to create the audio object
- location2 – The location of the sound file that is used as a fall back if sound at location does not load
- reuse – When set to false the generated sound Audio will be re-used, this can prevent sounds that are currently playing being replayed whilst currently being played but can help resolve audio playback issues
- loop – If set to true then sound will be looped
- preload – if set to true then this sound will be preloaded
- loaded – If set to true then this resources has finished loading
- snd – Local audio object

Sound Methods

- Sound(name, location, reuse) – Creates an instance of a Sound object.
  - name – Name of the sound
  - location – The sound file location
  - reuse – Mark the sound to be re-used
- load() – Loads the sound
- play() – Plays the sound
  - returns an audio object representing the playing sound, this values is also set to the snd property of the Sound object
- stop() – Stops playback of the sound
- pause() – Pauses playback of the sound

- destroy() – Removes the sound from the scene / app and destroys it

Shape Properties

- name – The shapes name
- parent – Parent container scene or app
- type – Type of shape
- width – Width of shape (or radius if circle)
- height -Height of shape
- vertices – Array of vertices for a polygon type shape in the form [x1,y1,x2,y2,....]
- convexVertices – If shape is concave then contains an array of array of vertices that make up convex shapes, if empty then shape is convex and vertices should be used

Shape Constants

- b5.Shape.TypeBox – A box shape type
- b5.Shape.TypeCircle – A circle shape type
- b5.Shape.TypePolygon – A polygon shape type

Shape Methods

- Shape(name) – Creates an instance of a Shape object
  - name - The name of the geometry, this name is used when searching for resources in a Scene's resources or the the app's global resources
- typeToConst(type_name) – Converts a string based shape type name to a Shape type constant
  - type – Name of the shape type (box, circle or polygon)
- destroy() – Removes the shape from the scene / app and destroys it

Material Properties

- name – The name of the material, this name is used when searching for bitmap resources in a Scene's resources or the the App's global resources
- type – Type of material (can be static, dynamic or kinematic)
- density – Material density, higher values make for heavier objects
- friction – Material friction, lower values make objects more slippery
- restitution – Material restitution, higher values make the object more bouncy
- gravity_scale – Gravity scale, lower values lessen the affects of gravity on the object
- fixed_rotation – Set to true to prevent objects from rotating

Material Methods

- Material(name) – Creates an instance of a Material object.
  - name – Name of the bitmap
- destroy() – Removes the material from the scene / app and destroys it

Gradient Properties

- name – The name of this object
- parent – The parent container scene or app
- stops – An array of colour stops

Gradient Methods

- Gradient(name, colour_stops) – Creates an instance of a Gradient object
  - name – The name of the gradient
  - colour_stops – An array of colour stops
- addColourStop(colour, offset) – Adds a gradient stop to the gradient
  - colour – Colour of gradient stop
  - offset – Offset of gradient stop
- getColourStop(index) – Returns the colour stop at the specified index
- getMaxStops() – Returns the total number of colour stops in the gradient
- destroy() – Removes the gradient from the scene / app and destroys it
- createStyle(w, h, start, end) – Creates a style from this gradient that can be used by fills and strokes
  - w – Width of the gradient area (usually width of fill area)
  - h – Height of the gradient area (usually height of fill area)
  - start – Start point of gradient {x, y}
  - end – End point of gradient {x, y}

ImageAtlas Properties

- name – The name of the ImageAtlas, this name is used when searching for brush resources in a Scene's resources or the the App's global resources
- parent – The parent container
- bitmap – The bitmap object that images will be used as a source for sub images
- Internal properties
- frames – Array of atlas frame objects in the form {x, y, w, h}

ImageAtlas Methods

- ImageAtlas(name, bitmap, x, y, w, h) – Creates an instance of an ImageAtlas
  - name - The name of the ImageAtlas, this name is used when searching for brush resources in a Scene's resources or the the App's global resources
  - bitmap – The source bitmap object
  - x, y, w, h – The top-left position and width / height of the initial atlas frame (optional)
- addFrame(sx, sy, sw, sh) – Adds a new frame to the atlas
  - sx, sy – Top-left hand position of sub image
  - sw. sh, Width and height of the sub image
- getFrame(index) – Returns the frame at the specified index
  - index – The index of the requested frame
  - returns the specified frame object
- getMaxFrames() – Returns total number of available frames in this atlas

- generate(start_x, start_y, frame_w, frame_h, count_x, count_y, total) – Generates multiple atlas frames, working from left to right, top to bottom
  - start_x – Start x position
  - start_y – Start y position
  - frame_w – The width of each frame
  - frame_h – The height of each frame
  - count_x – Total frames to generate across the image
  - count_y – Total frames to generate down the image
  - total – Optional parameter that can be used to limit total number of generated frames
- destroy() – Removes the atlas from the scene / app and destroys it

Animation – Lets Dance

Introduction

One of Booty5's biggest and best features is its animation editor. The Booty5 timeline animation editor enables rapid production of Flash style animations. Booty5 exports animations that utilise the Booty5 engines timeline animation system. The animation system is split over a number of classes:

- b5.Animation – An animation is a collection of animation frames
- b5.Timeline – A timeline is a collection of animations
- b5.TimelineManager – A collection of timelines

Each Scene, Actor has its own TimelineManager that generally manages its local timelines. In addition, the global App object has its own TimelineManager that handles global animations.

To see what type of animations Booty5 can create take a look at the basic animation demo and world animation demo.

Working with Animations

Animations are created by creating instances of b5.Animation objects that target specific objects and their properties. Created animations are added to a Timeline (a container for multiple animations), usually all of the animations that target a specific object are added to the same Timeline although this is not a restiction. Timelines are then added to the TimelineManager to be processed each frame.

An Animation consists of a target, a property and a collection of key frames and key times. A key frame is the value of a property at a specific point in time stipulated by the key times. For example, the target could be an actor called player, the property could be the x position of the actor and the key frames could be an array of numbers that specify the values of the actors x position at specific points in time. e.g.:

- At time 0 seconds objects x position is 0
- At time 2 seconds objects x position is 200

Whilst the animation is playing it will interpolate the values of the objects property over time to create a smooth transition from one value to the next, so for example at time 1 second the value will be 100 (half way between 0 and 200). You can modify how this interpolation (tweening) is applied by using easing functions. Easing functions affect how the value is tweened from one value to the next, the default is Linear easing which simply smoothly tweens from one value to the next. You can specify which tweening function to apply to each individual frame by passing in an optional array to the animation creation function that specifies which tweening functions to use for each frame. The current list of available tweening functions includes:

- b5.Ease.linear
- b5.Ease.quadin
- b5.Ease.quadout
- b5.Ease.cubicin
- b5.Ease.cubicout
- b5.Ease.quartin
- b5.Ease.quartout
- b5.Ease.sin

Lets take a quick look at an example that shows how to create an Animation:

```
var anim = new b5.Animation(null, actor1, "_x", [0, 200], [0, 2], 0, [b5.Ease.quartin]);
```

In the above example we create an animation that targets the _x property of actor1, it changes this property from 0 to 200 over 2 seconds, using the quartic-in easing function. Note that we target the _x property setter instead of the x property of an actor because internally the actor system has to rebuild its visual transform, which the property setter does.

Simply creating the animation is not enough, it must be added to a Timeline and then that Timeline must be added to a TimelineManager, e.g.:

```
// Create a timeline
var timeline = new b5.Timeline();

// Add the animation
timeline.add(anim);

// Add the timeline to the actors timeline manager
actor1.timelines.add(timeline);

// Start the timeline playing
timeline.play();
```

Once an animation has been created and added to a Timeline, you can later locate it by calling b5.Timeline.find(animation_name). You can also locate a timeline using a path, e.g.:

```
b5.Utils.resolveObject("scene1.actor1.timeline1", "timeline");
```

Animations can repeat playback multiple times by passing in the number of times to repeat when creating the Animation object, passing a value of 0 will repeat the animation forever.

Once an animation reaches the end of its duration it will automatically be destroyed, unless you set the b5.Animation.destroy property to false. You can also remove an animation from a Timeline by calling b5.Timelime.remove(animation).

Working with Timelines

Timelines represent collections of animations with each animation potentially targeting a different property and a different object. Timelines are a way of creating a complete set of animations that represent a complete sequence of animation. For example, a timeline could contain all of the animations that represent a cut scene sequence during a game, or all of the animations that represent an action that takes place when a certain event occurs, such as a pick-up animation when the player picks up certain objects.

To create a Timeline we create an instance of a b5.Timeline object, e.g.:

```
var timeline = new b5.Timeline();
```

Once a Timeline is created we can begin adding animations to it using b5.Timeline.add(), which has two flavours:

- b5.Timeline.add(anim) – Adds the supplied Animation instance "anim" to the timeline
- b5.Timeline.add(target, property, frames, times, repeat, easing) – Creates an animation with the specified properties then adds it to the timeline. This is a convenience function that allows you to add animations without having to create the animation up front. Internally an Animation object will be created for you

Once a Timeline has been created it should be added to a TimelineManager in order to be processed each game frame, e.g.:

```
scene.timelines.add(timeline);
```

Once a timeline has been created and added to a TimelineManager, you can later locate it by calling b5.TimelineManager.find(timeline_name).

The main App, Scenes and Actors all have their own TimelineManager's (e.g. scene.timelines) that you can add created timelines to. However, you can create and manage your own as long as you call b5.TimelineManager.update() on a regular basis.

To later remove a Timeline from its manager you can call b5.TimelineManager.remove(timeline). Be aware that the timeline may no longer be present as timelines clean themselves up when all animations within the Timeline have been destroyed; an animation object will destroy itself when it reaches the end of its playback, unless its destroy property is set to false.

A number of methods are available which can affect the timeline:

- b5.Timeline.play() - Plays all animations in the timeline, resumes play back it is paused
- b5.Timeline.pause() - Pauses playback of all animations in the timeline
- b5.Timeline.restart() - Restarts all animations from their starts, also resets the total number of times to repeat each animation to their original values

These methods are also available within the TimelineManager and operate across all Timelines that the manager contains:

- b5.TimelineManager.play() - Plays all timeline in the timeline manager, resumes play back it is paused
- b5.TimelineManager.pause() - Pauses playback of all timelines in the timeline manager
- b5.TimelineManager.restart() - Restarts all timelines in the timeline manager from their starts

Animation Events

Animations can fire off various events based on the status of the animation. The following events are currently supported:

- onEnd – Called when the animation finishes playing
- onRepeat – Called when the animation repeats
- Frame hit – Called when a specific animation frame is hit

The first two events are simple to to set up and use, lets take a quick look at an example:

```
// Create animation
var anim = new b5.Animation(null, actor1, "_x", [0, 200], [0, 2], 2, [b5.Ease.linear]);
anim.name = "anim1";

// Set callbacks
anim.onEnd = function (e) {
    console.log("Animation ended " + e.name);
};
anim.onRepeat = function (e) {
    console.log("Animation repeated " + e.name);
};

// Create a timeline
var timeline = new b5.Timeline();

// Add the animation
timeline.add(anim);

// Add the timeline to the actors timeline manager and play
actor1.timelines.add(timeline).play();
```

The animation system can also fire events when each individual frame is hit by setting action functions (not to be confused with action lists) with b5.Animation.setAction(index, action_function). Lets take a quick look at an example:

```
var timeline = new b5.Timeline();
var anim = timeline.add(this, "x", [0, 100, 300, 400], [0, 5, 10, 15], 0, [b5.Ease.quartin,
b5.Ease.quartin, b5.Ease.quartin]);
anim.setAction(0,function() { console.log("Hit frame 0"); });
anim.setAction(1,function() { console.log("Hit frame 1"); });
anim.setAction(2,function() { console.log("Hit frame 2"); });
```

In the above example, we create an animation with 4 key frames then assign an action to the first 3 frames, note that we do not assign an action to the last frame as that frame will call either onEnd or onRepeat. When the animation reaches each of the frames the corresponding action function will be called.

Responding to animation events are great for setting off other animations, updating game logic, playing sound effects and so on.

Animation Playback Speed

You can change the speed at which an animation is played back by setting the b5.Animation.time_scale property. Setting it to a value less than 1 will play the animation back at a slower speed, whilst setting it to a value of greater than 1 will play back the animation at an higher speed. For example, setting time_scale to 2 will play back the animation at double its intended speed, whilst setting it to 0.5 will play back the animation at half the intended speed.

Adjusting the time_scale of animations can be used to create temporal distortion effects.

Tween or not to Tween

Each animation has a tween property which when set to true will cause animation key frames to be smoothly interpolated. This is however not appropriate for all types of object properties. For example, if we create an animation that changes the text of a label over time, tweening is not possible as we want the text to change to a discrete value. In this case tweening can be turned off by setting the tween property of the animation to false. Lets take a quick look at a none tweened animation:

```
var timeline = new b5.Timeline();
var anim = timeline.add(actor, "text", ["Hello", "World", ""], [0, 1, 2], 0);
anim.tween = false;
actor.timelines.add(timeline);
```

Animation properties

- name – Name of the animation
- target – Target object to tween
- property – Property to tween
- frames – Key frame data (array of key frame values)
- times – Key frame times (array of key frame times)
- easing – Key frame easing functions (array of easing functions, see Ease reference)
- repeat – Total number of times to repeat animation (0 for forever)
- destroy – If true then animation will be destroyed when it finishes playing (default is true)
- actions – Array of action functions (called when each frame is reached)
- time_scale – Amount to scale time (default is 1.0)
- tween – When set to true, key frames will be smoothly interpolated, if set to false then frames will not be interpolated
- timeline – Parent timeline
- state – State of playback (AS_playing or AS_paused)
- time – Current time
- repeats_left – Number of repeats left to play
- index – Optimisation to prevent searching through all frames

Animation Constants

- b5.Animation.AS_playing – Animation is playing
- b5.Animation.AS_paused – Animation is paused

Animation Events

- onEnd() – Called when the animation ends
- onRepeat() – Called when the animation repeats
- actions – Each frame can be assigned a function that is called when that start of that frame is reached

Animation Methods

- Animation(timeline, target, property, frames, times, repeat, easing) – Creates an instance of an Animation object
  - timeline – The parent Timeline that will manage this animation
  - target – The target object that will have its properties animated
  - property – The name of the property that will be animated
  - frames – An array of key frame values that represent the value of the property at each time slot
  - times – An array of time values that represent the time at which each key frame should be used
  - repeat – the total number of times that the animation should repeat (0 for forever)
  - easing – An array of easing values (optional), see Ease for reference
- pause() – Pauses the playback of the animation
- play() – Plays / resumes the animation
- restart() – Restarts the animation from its beginning
- update(dt) – Updates the animation (called internally)
  - dt – The amount of time that has passed since this animation was last updated
- setTime(time) – Sets the current playback time value of the animation
  - time – Playback time of animation in seconds, a negative value can be used to delay the start of the animation
- setAction(index, action_function) – Sets an action function that will be called when the animation reaches the frame specified by index
  - index – The frame at which to set the action function
  - action_function – function to call when the specified animation frame is reached

Timeline Properties

- anims – An array of Animations
- manager – The parent timeline manager that manages this timeline
- name – The name of this timeline

Timeline Methods

- Timeline(target, property, frames, times, repeat, easing) – Creates an instance of a

Timeline object
  - target – The target object that will have its properties animated
  - property – The name of the property that will be animated
  - frames – An array of key frame values that represent the value of the property at each time slot
  - times – An array of time values that represent the time at which each key frame should be used
  - repeat – the total number of times that the animation should repeat (0 for forever)
  - easing – An array of easing values (optional), see Ease for reference
- add(animation) – Adds an Animation object to this timeline
  - animation – Instance of animation to add
- add(target, property, frames, times, repeat, easing) – Adds an Animation object to this timeline
  - target – The target object that will have its properties animated
  - property – The name of the property that will be animated
  - frames – An array of key frame values that represent the value of the property at each time slot
  - times – An array of time values that represent the time at which each key frame should be used
  - repeat – the total number of times that the animation should repeat (0 for forever)
  - easing – An array of easing values (optional), see Ease for reference
- remove(animation) – Removes the specified animation from the timeline
- find(name) – Finds and returns the named animation or null if not found
- pause() – Pauses all Animations in the timeline
- play() – Plays all Animations in the timeline
- restart() – Restarts all Animations in the timeline
- print() - Debug method which prints out a list of animations in the timeline
- update(dt) – Updates all Animations in the timeline
  - dt – The amount of time that has passed since this timeline was last updated

TimelineManager Properties

- timelines – Array of Timelines

TimelineManager Methods

- TimelineManager() – Create an instance of a TimelineManager object
- add(timeline) – Adds a timeline to this TimelineManager
  - timeline – The timeline to add to this manager
- remove(timeline) – Removes the specified timeline from the TimelineManager
  - timeline – The timeline to remove from the manager
- find(name) – Finds and returns the named timeline, or null if not found
- pause() – Pauses all Timelines in the manager
- play() – Plays all Timelines in the manager
- restart() – Restarts all Timelines in the manager
- print() - Debug method which prints out a list of Timelines in the manager
- update(dt) – Updates all Timelines in the manager
  - dt – The amount of time that has passed since this manager was last updated

Actions – Building with Blocks

Introduction

Actions are pieces of pre-defined logic that can be added to Actors and Scenes to extend their functionality. There is no particular base class for an action, instead actions conform to a strict frame work which consists of the following:

- A constructor which creates an instance of the action
- An onInit() method which is called when the action is first executed or when it is executed again after it has finished executing and ran again
- An onTick() method which is called each time the action is updated

Actions once created are added to an actions list, each action in an actions list is executed consecutively, which means that the next action will not execute until the current action has finished executing. For example:

- Move object to position 100,100 over 5 seconds
- Move object to position 0,0 over 2 seconds
- Play a sound effects

The 2nd action will not execute until the object reaches position 100,100 after 5 seconds. The 3rd action will not execute until the object reaches position 0,0 after a further 2 seconds, so the sound effect will not play until 7 seconds have passed.

Action lists can repeat any number of times or can play forever, you can specify the number of repeats when creating an instance of b5.ActionList.

By building up a library of actions, game functionality can be bolted together by creating lists of those actions and assigning them to game objects.

Actors and Scenes both contain an ActionsListManager which allows the object to run multiple action lists simultaneously

Lets take a quick look at an example that shows how to add some of the pre-defined actions that ship with Booty5 to an actor object to modify its behaviour:

```
// Create actions list that repeats forever
var actions_list = new b5.ActionsList("moveme1", 0);

// Add an action that will move the object to position 100,100 over 5 seconds
actions_list.add(new b5.A_MoveTo(actor, 100, 100, 5));

// Add an action that will move the object to position 0,0 over 2 seconds
actions_list.add(new b5.A_MoveTo(actor, 0, 0, 2));

// Add the actions list to the actor and start it playing
actor.actions.add(actions_list).play();
```

In the above example, we create an actions list called "moveme1", we add two actions to the actions list that moves the object then we add the actions list to the actors actiosn list manager for processing and set if off playing.

Actions are added to the ActionsList using b5.ActionsList.add(), they can later be removed by calling b5.ActionsList.remove(). Once an ActionsList has been added to the manager you can later find it by calling b5.ActionsList.find(actions_list_name). You can also find an actions list from a path, e.g.:

```
b5.Utils.resolveObject("scene1.actor1.actionslist1", "actions");
```

An ActionsList has a number of methods that can affect its behaviour:

- b5.ActionsList.play() - Starts the actions list playing or resumes if paused
- b5.ActionsList.pause() - Pauses playback of the the actions list
- b5.ActionsList.restart() - Restarts the actions list from the beginning, also resets the number of repeats

Creating Custom Actions

Whilst a large collection of pre-defined actions have been created for you, you can easily create and register actions of your own that fit your specific game.

To create an action you need to create an actions class then optionally register it with the actions register ActionsRegister. An action class consists of a constructor, an onInit() method and an onTick() method:

- constructor – The constructor creates an instance of the action remembering any passed parameters
- onInit() – This method sets up the initial state of the action
- onTick() – This method is called every time the action is executed (every frame)

Each time an action is encountered in the actions list it is initialised, this includes when the actions list repeats. Once initialised the action will call its onTick() method until false is returned from the onTick() method. Note that if no onTick() method is supplied then the action will exit immediately.

Lets take a quick look at an example that shows how to create our own action:

```
b5.A_StopMove = function(target, stop_vx, stop_vy, stop_vr, duration)
{
    this.target = target;
    this.stop_vx = stop_vx;
    this.stop_vy = stop_vy;
    this.stop_vr = stop_vr;
    this.duration = duration;
};
b5.A_StopMove.prototype.onInit = function()
{
    this.target = b5.Utils.resolveObject(this.target);
    this.time = Date.now();
};
b5.A_StopMove.prototype.onTick = function()
{
    if (!((Date.now() - this.time) &lt; (this.duration * 1000)))
    {
        var target = this.target;
        if (this.stop_vx)
            target.vx = 0;
        if (this.stop_vy)
            target.vy = 0;
        if (this.stop_vr)
            target.vr = 0;
        return false;
    }
    return true;
};
b5.ActionsRegister.register("StopMove", function(p) { return new
b5.A_StopMove(p[1],p[2],p[3],p[4],p[5]); });
```

In the above code we declare a new action called A_StopMove which accepts 5 parameters. We store these parameters in the instance of the action because we will need them later.

We have also declared onInit() and onTick() methods. Notice how we return true from onTick() until the action has ran for its duration, at which point we return false to let the actions system know that it should move onto the next action in the list.

Finally, notice how we register the action with the actions register. We pass in the name of the action "StopMove" along with a function that creates an instance of the actions class. Registering your class with the actions register is only required if you want to make the action callable as a custom action from the Booty5 game maker editor.

Predefined Actions

Booty5 comes with a large range of pre-defined actions out of the box. A complete list of these is shown below:

Action List Actions

The A_ChangeActions action changes the state of an actions list then exits

- actions – Path to actions list (e.g. scene1.actor1.actionslist1)
- action – Action to perform on the actions list (play, pause or restart)

Actor Actions

The A_CreateExplosion action creates a particle system actor then exits

- container – Path to scene or actor that will contain the generated particle actor (e.g. scene1.actor1)
- count – Total number of particles to create
- type – The actor type of each particle created, for example ArcActor
- duration – The total duration of the particle system in seconds
- speed – The speed at which the particles blow apart
- spin_speed – The speed at which particles spin
- rate – Rate at which particles are created
- damping – A factor to reduce velocity of particles each frame, values greater than 1 will increase velocities
- properties – A collection of actor specific properties that will be assigned to each created particle (e.g. vx=10,vy=10)
- actor – If provided then the generated particle actor will be placed at the same position and orientation as this actor, actor is path to an actor

The A_CreatePlume action creates a particle system actor then exits

- container – Path to scene or actor that will contain the generated particle actor (e.g. scene1.actor1)
- count – Total number of particles to create
- type – The actor type of each particle created, for example ArcActor
- duration – The total duration of the particle system in seconds
- speed – The speed at which the particles rise
- spin_speed – The speed at which particles spin
- rate – Rate at which particles are created
- damping – A factor to reduce velocity of particles each frame, values greater than 1 will increase velocities
- properties – A collection of actor specific properties that will be assigned to each created particle (e.g. vx=10,vy=10)
- actor – If provided then the generated particle actor will be placed at the same position and orientation as this actor, actor is path to an actor

Animation Actions

The A_ChangeTimeline action changes the state of an animation timeline then exits

- timeline – Path to timeline (e.g. scene1.actor1.timeline1)
- action – Action to perform on the timeline (play, pause or restart)

Attractor Actions

The A_AttractX action pulls objects towards or repels objects away on the x-axis that are within a specific range, does not exit

- target – Path to actor object that will attract other objects (e.g. scene1.actor1)
- container – Path to object that contains the actors (actors that can be attracted have attract property set to true)
- min_x – Minimum x-axis attraction range
- max_x – Maximum x-axis attraction range
- min_y – Minimum y-axis inclusion range
- max_y – Maximum y-axis inclusion range
- strength – Strength of attraction, negative for repulsion
- stop – If set to true then attracted objects will stop when they hit the minimum distance range
- bounce – If set to true then objects when stopped at the minimum distance range will bounce

The A_AttractY action pulls objects towards or repels objects away on the y-axis that are within a specific range, does not exit

- target – Path to actor object that will attract other objects (e.g. scene1.actor1)
- container – Path to object that contains the actors (actors that can be attracted have attract property set to true)
- min_y – Minimum y-axis attraction range
- max_y – Maximum y-axis attraction range
- min_x – Minimum x-axis inclusion range
- max_x – Maximum x-axis inclusion range
- strength – Strength of attraction, negative for repulsion
- stop – If set to true then attracted objects will stop when they hit the minimum distance range
- bounce – If set to true then objects when stopped at the minimum distance range will bounce

The A_Attract action pulls objects towards or repels objects away on the x and y-axis that are within a specific range, does not exit

- target – Path to actor object that will attract other objects (e.g. scene1.actor1)
- container – Path to object that contains the actors (actors that can be attracted have attract property set to true)
- min_dist – Minimum attraction range
- max_dist – Maximum attraction range
- strength – Strength of attraction, negative for repulsion
- stop – If set to true then attracted objects will stop when they hit the minimum distance range

- bounce – If set to true then objects when stopped at the minimum distance range will bounce

Audio Actions

The A_Sound action plays, pauses or stops a sound then exits

- name – Path to sound resource (e.g. scene1.sound1)
- action – Action to perform on sound (play, pause or stop)

General Actions

The A_Wait action waits for a specified time then exits

- duration – Amount of time to wait

The A_SetProps action sets a group of properties of an object to specified values then exits

- target – Path to target object (e.g. scebe1.actor1)
- properties – Property / value pairs that will be set (e.g. vx=0,vy=0)

The A_AddProps action adds the specified values onto the specified properties then exits

- target – Path to target object (e.g. scebe1.actor1)
- properties – Property / value pairs that will be updated (e.g. vx=0,vy=0)

The A_TweenProps action tweens the specified property values over time then exits

- target – Path to or instance of target object (e.g. scene1.actor1)
- properties – Property / value pairs that will be tweened (e.g. vx=0,vy=0)
- start – Array of start values (e.g. 0,0)
- end – Array of end values (e.g. 100,500)
- duration – Amount of time to tween over
- ease – Array of easing functions to apply to tweens

The A_Call action calls a function then exits

- func – Function to call
- params – Parameter to pass to function (passed to function as an array)

The A_Create action creates an object from a xoml template then exits

- objects – Collection of objects in XOML JSON format (as exported from Booty5 editor) that contains the template (e.g. gamescene)
- scene – Path to scene that contains the template and its resources (e.g. gamescene)
- template – The name of the object template (e.g. actor1)
- type – The type of object (e.g. icon, label, scene etc)
- properties – Property / value pairs that will be set to created object (e.g. vx=0,vy=0)

The A_Destroy action destroys an object then exits

- target – Path to object to destroy (e.g. scene1.actor1)

The A_FocusScene action sets the current focus scene then exits

- target – Path to scene to set as focus
- focus2 – Set as secondary focus instead

The A_Custom action is used to call your own custom actions (Booty5 editor only)

- name – Name of custom action
- target – Path to actor or scene
- params – Parameter to pass to function (passed to function as an array)

Movement Actions

The A_StopMove action stops an object from moving then exits

- target – Path to target object (e.g. scene1.actor1)
- stop_vx – Stops x velocity
- stop_vy – Stops y velocity
- stop_vr – Stops rotational velocity
- duration – Amount of time to wait before stopping

The A_Gravity action applies gravity to an object, does not exit

- target – Path to target object (e.g. scene1.actor1)
- gravity_x – Gravity strength on x-axis
- gravity_y – Gravity strength on y-axis

The A_Move action moves an object dx, dy units over the specified time then exits

- target – Path to target object (e.g. scene1.actor1)
- dx, dy – Distances to move on x and y axis
- duration – Amount of time to move over

The A_MoveTo action moves an object to a specific coordinate over the specified time then exits

- target – Path to target object (e.g. scene1.actor1)
- x, y – Target position to move to
- duration – Amount of time to move over
- ease_x – Easing function to use on x-axis
- ease_y – Easing function to use on y-axis

The A_MoveWithSpeed action moves an object at specific speed in its current direction over the specified time then exits

- target – Path to target object (e.g. scene1.actor1)
- speed – Speed at which to move
- duration – Amount of time to move over

- ease – Easing function used to increase to target speed

The A_Follow action follows a target, does not exit

- source – Path to target object (e.g. scene1.actor1)
- target – Path to target object (e.g. scene1.actor2)
- speed – Speed at which to follow, larger values will catch up with target slower
- distance – Minimum distance allowed between source and target (squared)

The A_LookAt action turns an object to face an object, does not exit

- source – Path to target object (e.g. scene1.actor1)
- target – Path to target object (e.g. scene1.actor2)
- lower – Lower limit angle
- upper – Upper limit angle

The A_FollowPath action follows a path, does not exit

- target – Path to target object (e.g. scene1.actor2)
- path – Path to shape to follow (e.g. ecene1.shape1)
- start – Distance to start along the path
- speed – Speed at which to travel down the path
- angle – If set to true then angle will adjust to path direction

The A_FollowPathVel action follows a path using velocity, does not exit

- target – Path to target object (e.g. scene1.actor2)
- path – Path to shape to follow (e.g. ecene1.shape1)
- start – Distance to start along the path
- speed – Speed at which to travel down the path
- catchup_speed – Speed at which object catches up with path target modes
- angle – If set to true then angle will adjust to path direction

The A_LimitMove action limits movement of object to within a rectangular area, does not exit

- target – Path to target object (e.g. scene1.actor2)
- area – Rectangular area limit [x,y,w,h] (e.g. -100,-100,100,100)
- hit – Action to perform when object oversteps boundary (bounce, wrap, stop)
- bounce – Bounce factor

Camera Actions

The A_CamStopMove action stops a scene camera from moving then exits

- target – Path to target scene that contains camera (e.g. scene1)
- stop_vx – Stops x velocity
- stop_vy – Stops y velocity
- duration – Amount of time wait before stopping

The A_CamGravity action applies gravity ti a scene camera, does not exit

- target – Path to target scene that contains camera (e.g. scene1)
- gravity_x – Gravity strength on x-axis
- gravity_y – Gravity strength on y-axis

The A_CamMove action moves a scene camera dx, dy units over the specified time then exits

- target – Path to target scene that contains camera (e.g. scene1)
- dx, dy – Distances to move on x and y axis
- duration – Amount of time to move over

The A_CamMoveTo action moves a scene camera to a specific coordinate over the specified time then exits

- target – Path to target scene that contains camera (e.g. scene1)
- x, y – Target position to move to
- duration – Amount of time to move over
- ease_x – Easing function to use on x-axis
- ease_y – Easing function to use on y-axis

The A_CamFollow action causes scene camera to follow a target, does not exit

- source – Path to scene that contains camera that will follow the target (e.g. scene1)
- target – Path to target object to follow
- speed – Speed at which to follow, larger values will catch up with target slower
- distance – Minimum distance allowed between source and target (squared)

The A_CamFollowPath action causes scene camera to follows a path, does not exit

- target – Path to target scene that contains camera (e.g. scene1)
- path – Path to shape to follow
- start – Distance to start along the path
- speed – Speed at which to travel down the path

The A_CamFollowPathVel action causes scene camera to follow a path using velocity, does not exit

- target – Path to target scene that contains camera (e.g. scene1)
- path – Path to shape to follow (e.g. ecene1.shape1)
- start – Distance to start along the path
- speed – Speed at which to travel down the path
- catchup_speed – Speed at which object catches up with path target modes

The A_CamLimitMove action limits movement of scene camera to within a rectangular area, does not exit

- target – Path to target scene that contains camera (e.g. scene1)
- area – Rectangular area limit [x,y,w,h] (e.g. -100,-100,100,100)
- hit – Action to perform when object oversteps boundary (bounce, wrap, stop)
- bounce – Bounce factor

Physics Actions

The A_SetLinearVelocity action sets the linear velocity of an actors body for a duration then exits

- target – Path to actor that will be changed
- vx – x-axis velocity
- vy – y-axis velocity
- duration – Duration of action

The A_SetAngularVelocity action sets the angular velocity of an actors body for a duration then exits

- target – Path to actor that will be changed
- vr – Angular velocity
- duration – Duration of action

The A_ApplyForce action applies force to an object over a period of time then exits

- target – Path to actor that will be changed
- fx – x-axis force
- fy – y-axis force
- dx – x-axis offset to apply force
- dy – y-axis offset to apply force
- duration – Duration of action

The A_ApplyImpulse action applies an impulse to an object then exits

- target – Path to actor that will be changed
- ix – x-axis impulse
- iy – y-axis impulse
- dx – x-axis offset to apply force
- dy – y-axis offset to apply force

The A_ApplyTorque action applies torque to an object over a period of time then exits

- target – Path to actor that will be changed
- torque – Torque
- duration – Duration of action

ActionsList Properties

- manager – The ActionsListManager that manages this actions list
- name – The name of this actions list
- repeat – The total number of times to repeat this actions list before exiting, 0 is forever
- actions – A collection of actions
- current – The index of the currently active action
- destroy – When set to true this actions list will be destroyed when it finishes executing
- playing – The playback state of this actions list

ActionsList Methods

- ActionsList(name, repeat) – Creates an instance of an ActionsList
    - name- The name of the actions list
    - repeat – Number of times to repeat playback of the actions list, 0 is forever
- add(action) – Adds the specified action to the list
    - action – The action to add to the list
- remove(action) – Removes the specified action from the actions list
    - action – The action to remove from the list
- execute() – Executes this actions list
- pause() – Pauses execution of the actions list
- play() – Starts or resumes execution of the actions list
- restarts() – Restarts execution of the actions list from the beginning

ActionsListManager Properties

- actions – An array of action lists

ActionsListManager Methods

- ActionsListManager() - creates an instance of an ActionsListManager
- add(actionlist) – Adds the specified actions list to the manager
    - actionlist – The actions list to add to the manager
- remove(action) – Removes the specified actions list from the manager
    - actionlist – The actions list to remove from the manager
- find(name) – Returns the named actions list or null if not found
    - name – The name of the actions list

XOML – Booty5 Editor Data

Introduction

The Booty5 game editor exports data for your game in a JSON format called XOML. The Xoml class reads this data and converts it into scenes, game objects and resources etc..

Booty5 will export a JSON file for each of your scenes as well as globals,js that contains information about all of your global app resources.

To parse a XOML JSON file, you need to create an instance of the Xoml class then call Xoml.parseResources() to parse the file. Lets take a look at a quick example:

Below is a small example of an exported XOML file:

```
window.globals = [
    {
        "ResourceType":"Shape",
        "Name":"circle1",
        "ShapeType":"Circle",
        "Width":40,
        "Height":40,
        "Absolute":false
    },
    {
        "ResourceType": "Shape",
        "Name": "rect3761",
        "ShapeType": "Polygon",
        "Width": 0,
        "Height": 0,
        "Absolute": false,
        "Vertices": [-126.336, -106.399, -42.018, -90.992, 153.614, 46.411, -66.468, 106.4,
-153.614, -7.28]
    }
];
```

Now lets take a look at how we parse the data to generate Booty5 compatible objects:

```
// Create XOML loader
var xoml = new b5.Xoml(app);

// Parse and create global resources placing them into the app
xoml.parseResources(app, xoml_globals);
```

Post Loading Scenes

In the Booty5 game editor you can mark scenes as “do not load” which means they will not be parsed when the app begins. At a later date you will need to load this exported scene XOML, the example below shows how to do this:

```
new b5.Xoml(app).parseResources(app, xoml_data);
```

In the above example we create an instance of the Xoml class then call parseResources(), passing in the app as the place to put global resources and the object that contains the XOML JSON data. Note that all exported scenes data is placed in the window object, so accessing for example a scene called level1 you would pass [window["level1"]] as xoml_data.

Creating Resources from XOML templates

You can create a copy of any object / resource that exists in the XOML JSON data, you can think of the XOML data as a collection of templates. Lets take a quick look at an example:

```
var app = b5.app;

// This scene will receive a copy of ball object
var game_scene = app.findScene("gamescene");

// Search Xoml gamescene for ball icon actor resource
var ball_template = b5.Xoml.findResource(window.gamescene, "ball", "icon");

// Create ball from the Xoml template and add it to game_scene
var xoml = new b5.Xoml(app);
xoml.current_scene = game_scene;  // Xoml system needs to know current scene so it knows
where to look for dependent resources
var ball = xoml.parseResource(game_scene, ball_template);
ball.setPosition(0, -350);
ball.vx = 4;
ball.fill_style = "rgb(" + ((Math.random() * 255) << 0) + "," + ((Math.random() * 255) << 0)
+ "," + ((Math.random() * 255) << 0) + ")";
```

In the above example we find the XOML template for the ball actor. We create an instance of the Xoml class that parses this template and creates a ball Actor object from it. We pass in the scene that contains resources that the ball object is dependent upon, such as bitmaps, timelines etc... Finally we modify the generated Actor object to add our own changes, such as give it a random fill style, a position and a velocity.

Booty5 Game Editor

Coming soon.....