Bump - A 2D Collision library for Pixi (v3.0.11)

Bump is a lightweight suite of easy-to-use 2D collision methods for use with the Pixi rendering engine. You can use these methods to make almost any kind of 2D action or arcade game.

(Important! this library targets Pixi v3.0.11, which is the most stable version of Pixi, and is the only version I can recommend using. This library will eventually be upgraded for Pixi v4 when the v4 branch matures.)

Installing and setting up Bump

Use a `<script>` tag to link the `bump.js` file to your HTML document.

```
<script src="bump.js"></script>
```

(If you prefer, you can load `bump.js` using any JavaScript module system you might be familiar working with: ES6 modules, SystemJS, AMD or CommonJS.) The Bump source files contain both ES5 and ES6 versions of the `bump.js` code. You should probably use the ES5 version for production code.

Next, create a new instance of Bump at the beginning of your program. Supply the renderer you want to use in the constructor (the defautl is `PIXI`) like this:

```
b = new Bump(PIXI);
```

The variable `b` (for "bump", of course!) now represents the running instance of Bump that you can use to access all of Bump’s collision methods.

Using Bump's collision methods

Using Bump is just a matter of choosing the collision method you want to use, and applying it based on the examples ahead. Just append `b` (or whatever variable name you chose for the Bump instance) to the beginning of each method, like this:

```
b.hitTestRectangle(spriteOne, spriteTwo);
```

Here are all the collision methods you can use:

###`hit`

`hit` is a universal collision function. It automatically detects the kinds of sprites that are being used in the collision and chooses the appropriate collision function for you. This means that instead of having to remember which of the many collision functions in Bump's library to use, you only need remember one: `hit`.

In its simplest form, you can use `hit` like this:

```
hit(spriteOne, spriteTwo)
```

It will return `true` if the sprites are touching and `false` if they aren't. And that’s all you need to know to start making some truly captivating games!

The sprites can be circles or rectangles. Bump's methods assumes they're rectangular by default. But if you want the methods to interpret a sprite as circular, give it a `circular` property and set it to `true`.

```
anySprite.circular = true;
```

If you want the sprites to react to the collision, so that they don’t intersect, set the third argument to `true`.

```
hit(spriteOne, spriteTwo, true)
```

This prevents them from overlapping, so it's useful for making walls, floors, or any other kind of solid boundary.

If you want the sprites to bounce apart, set the fourth argument to `true`.

```
hit(spriteOne, spriteTwo, true, true)
```

Now your sprites will bounce!

Setting the fifth argument to `true` makes the `hit` method use the sprites’ **global coordinates**.

```
hit(spriteOne, spriteTwo, true, true, true)
```

The global coordinates are the sprites positions relative to the canvas's top left corner, instead of the top left corner of their parent container (their local coordinates.)

If you want to check a point object for a collision against a sprite, use the point as the first argument, like this:

```
hit({x: 145, y:65}, sprite)
```

A point object is just any object with two properties, `x` and `y`, that define the point's position.

The `hit` method also lets you check for a collision between a sprite and an array of sprites. Just include the array as the second argument. In this example the array is called `bricks.children`:

```
hit(ball, bricks.children, true, true, true);
```

You'll see that `hit` automatically loops through all the sprites in the array for you and checks them against the first sprite. This means you don't have to write your own `for` loop or `forEach` loop.

The `hit` function also returns a collision object, with a return value that matches the kinds of sprites you're checking. For example, if both sprites are rectangles, you could find the side on which the collision occurred like this:

```
let collision = hit(rectangleOne, rectangleTwo, true);
message.text = "collision side: " + collision;
```

`collision` will always be `undefined` if there's no collision.

A final feature is that you can use an optional callback function as the sixth argument. This lets you inject some extra code that should run when the collision occurs. This is especially useful for checking a collision between a single sprite and an array of sprites. If there's a collision, the callback will run, and you can access both the collision return value and the sprite involved in the collision. Here's how you could use this feature to check for a collision between a sprite called `player` and an array of sprites in an array called `world.platforms`:

```
let playerVsPlatforms = hit(
  player,
  world.platforms,
  true, false, false,
  function(collision, platform){

    //`collision` tells you the side on player that the collision occurred on.
    //`platform` is the sprite from the `world.platforms` array
    //that the player is colliding with
  }
);
```

This is a compact way of doing complex collision checks that gives you a lot of information and low-level control but saves you from having to manually loop through all the sprites in the array.

However, the `hit` method is just a high-level wrapper for Bump's many lower level collision methods. If you prefer to use the lower-level methods, they're all listed next.

###`hitTestPoint`

The most basic collision test is to check whether a point is intersecting a sprite. `hitTestPoint` will help you to figure this out. `hitTestPoint` takes two arguments: a point object with `x` and `y` properties, and a sprite:

```
hitTestPoint(
  {x: 128, y: 128},  //An object with `x` and `y` properties
  anySprite          //A sprite
)
```

`hitTestPoint` will return `true` if the point intersects the sprite, and `false` if it doesn't.

The `hitTestPoint` method works equally well with rectangular and circular sprites. If the sprite has a `radius` property, `hitTestPoint` assumes that the sprite is circular and applies a point collision detection algorithm for circles. If the sprite doesn't have a `radius` property, the method assumes it is a square. You can give any sprite a `radius` property. An easy way to do this is to give the sprite a `circular` property and set it to `true`.

```
anySprite.circular = true;
```

The sprite will now be interpreted as circular and have a new `radius` property which is equal to half the sprite's width.

###`hitTestCircle`

If you want to check for a collision between two circular sprites, use the `hitTestCircle` method:

```
hitTestCircle(sprite1, sprite2)
```

Use it with any sprite that has a `radius` property. It returns `true` if the circles are touching, so you can use it with an `if` statement to check for a collision, like this:

```
if (hitTestCircle(sprite1, sprite2)) {
  //The circles are touching
}
```

###`circleCollision`

If a moving circle hits a non-moving circle, you can create a collision reaction using the `circleCollision` method:

```
circleCollision(circle1, circle2, true);
```

This stops the circles from intersecting.

The first argument is the moving ball, and the second is the non-moving ball. The third argument is an optional Boolean that determines whether the first circle should bounce off the second. (The Boolean will default to `false` if you leave it out, so if you want the circles to bounce, set it to `true`. You probably want to do this!)

There's an optional fourth argument that, if you set it to `true`, makes the method use the sprites' global coordinates. This is important if you want to check for collisions between sprites that have different parent containers.

###`movingCircleCollision`

You can create a collision reaction between two moving circles using the `movingCircleCollision` method. Supply two circular sprites as arguments:

```
movingCircleCollision(circle1, circle2)
```

If the circles have a `mass` property, that value will be used to help figure out the force with which the circles should bounce off each other. The `movingCircleCollision` method makes the sprites bounce apart by default. An important feature of this method is that when two moving circles collide, they transfer their velocities to each other in a way that makes them bounce apart very realistically.

Checking for multiple collisions

If you have a bunch of moving circles, like marbles, you need to check for a collision between each of them on every frame of your game loop. To do this you need to make sure that no pair of marbles is checked for collisions with each other more than once. The key to making this work is to use a nested `for` loop and to start the counter of the inner loop by one greater than the outer loop. Here's some example code that using the `movingCircleCollision` method to make an array of marbles bounce off each other (the marble sprites are in an array called `marbles.children`):

```
for (let i = 0; i < marbles.children.length; i++) {

  //The first marble to use in the collision check
  var c1 = marbles.children[i];
  for (let j = i + 1; j < marbles.children.length; j++) {

    //The second marble to use in the collision check
    let c2 = marbles.children[j];
    //Check for a collision and bounce the marbles apart if they collide
    movingCircleCollision(c1, c2);
  }
}
```

You can see that the inner loop starts at a number that’s one greater than the outer loop:

```
let j = i + 1
```

This prevents any pair of objects from being checked for collisions more than once.

The Bump library also has a convenience method called `multipleCircleCollision` that automates this entire nested for loop for you. You can use it in a game loop to check all the sprites in an array with all the other sprites in the same array, without duplication. Use it like this:

```
multipleCircleCollision(marbles.children)
```

It will automatically call `movingCircleCollision` on each pair of sprites to make them bounce off one another. You now know most of the important techniques you need to make a wide range of games using circular sprites.

###`hitTestRectangle`

To find out whether two rectangular sprites are overlapping, use a function called `hitTestRectangle`:

```
hitTestRectangle(rectangle1, rectangle2)
```

`hitTestRectangle` returns `true` if the rectangles overlap, and `false` if they don't. You can use it in a simple collision test like this:

```
if (hitTestRectangle(rectangle1, rectangle2)) {
  //The rectangles are touching
} else {
  //They're not touching
}
```

###`rectangleCollision`

`rectangleCollision` make the rectangles behave as though they have solid mass. It prevents any of the rectangular sprites in its first two arguments from overlapping:

```
rectangleCollision(rectangle1, rectangle2)
```

`rectangleCollision` also returns a string, whose value may be "left", "right", "top", or "bottom", that tells you which side of the first rectangle touched the second rectangle. You can assign the return value to a variable and use the information in your game. Here’s how:

```
let collision = rectangleCollision(rectangle1, rectangle2);

//On which side of rectangle1 is the collision occuring?
switch (collision) {
  case "left":
    message.text = "Collision on left";
    break;
  case "right":
    message.text = "Collision on right";
    break;
  case "top":
    message.text = "Collision on top";
    break;
  case "bottom":
    message.text = "Collision on bottom";
    break;
  default:
    message.text = "No collision...";
}
```

`collision` has a default value of `undefined`. This example code will prevent the rectangles from overlapping and displays the collision side in a text sprite called `message`.

The `rectangleCollision` method has a very useful side effect. The second sprite in the argument has the ability to push the first sprite out of the way. If you need to add a block-pushing or tile-sliding feature to a game, use this feature.

`rectangleCollision` has a third, optional Boolean argument, `bounce`:

```
rectangleCollision(rectangle1, rectangle2, true)
```

If `bounce` is `true`, it makes the first sprite bounce off the second sprite when they collide. Its default value is `false`. (As with the all the other collision methods, you should set the final optional argument, `global`, to `true` if you want to use the sprites’ global coordinates.)

###`hitTestCircleRectangle`

`hitTestCircleRectangle` checks for a collision between a circular and rectangular sprite. The first argument is the circular sprite, and the second is the rectangular sprite:

```
let collision = hitTestCircleRectangle(ball, box);
```

If they're touching, the return value (collision) will tell you where the circle is hitting the rectangle. It can have the value "topLeft", "topMiddle", "topRight", "leftMiddle", "rightMiddle", "bottomLeft", "bottomMiddle", or "bottomRight". If there's no collision it will be `undefined`.

###`circleRectangleCollision`

Use `circleRectangleCollision` to make a circle bounce off a square's sides or corners:

```
circleRectangleCollision(ball, box, true);
```

Setting the optional third argument to `true` makes the sprites bounce apart, and setting the fourth argument to `true` tells the method to use the sprites' global coordinates.

###`contain`

`contain` can be used to contain a sprite with `x` and `y` properties inside a rectangular area.

The `contain` function takes four arguments: a sprite with `x` and `y` properties, an object literal with `x`, `y`, `width` and `height` properties. The third argument is a Boolean (true/false) value that determines if the sprite should bounce when it hits the edge of the container. The fourth argument is an extra user-defined callback function that you can call when the sprite hits the container. (Only the first two arguments are required.)

```
contain(anySprite, {x: 0, y: 0, width: 512, height: 512}, true, callbackFunction);
```

The code above will contain the sprite's position inside the 512 by 512 pixel area defined by the object. If the sprite hits the edges of the container, it will bounce. The `callBackFunction` will run if there's a collision.

An additional feature of the `contain` method is that if the sprite has a `mass` property, that value will be used to dampen the sprite's bounce in a natural looking way.

If the sprite bumps into any of the containing object's boundaries, the `contain` function will return a Set of string values that tells you which side the sprite bumped into: "left", "top", "right" or "bottom". Here's how you could keep the sprite contained and also find out which boundary it hit:

```
//Contain the sprite and find the collision value
let collision = contain(anySprite, {x: 0, y: 0, width: 512, height: 512});

//If there's a collision, display the boundary that the collision happened on
if(collision) {
  if collision.has("left") console.log("The sprite hit the left");
  if collision.has("top") console.log("The sprite hit the top");
  if collision.has("right") console.log("The sprite hit the right");
  if collision.has("bottom") console.log("The sprite hit the bottom");
}
```

If the sprite doesn't hit a boundary, the value of `collision` will be `undefined`.