JsPhysics

Index

Setting up JsPhysics

Downloading the Library

By the GitHub Website

– Go to https://GitHub.com/MatheusTomazella/JsPhysics;

– Press the green button named “Clone or Download” and select “Download ZIP”;

– Go to where you downloaded (by default the folder “Download”);

– Right click on the ZIP folder and select “Extract Here”;

– Drag and drop the file “JsPhysics.js” to the root of your page directory.

By GitBash

– Go to your page directory and right click the blank space;

– Select “GitBash Here”;

– Type “Git clone https://github.com/matheustomazella/jsphysics.git” and press enter;

Important:

Make sure the file “JsPhysics.js” is on the root of your page’s directory.

Adding to the project

– Open your HTML file and add the library on the head tag by the code: “<script src=”JsPhysics.js”>”;

Initializing script

– On a new script tag, type “difineArea(canvasId, resolutionMultplier);

– Make sure to add the quotation marks on the Id if it isn’t a variable;

– Add the function “Loop()” and start programming!

Obs: the canvas has a default width and height (300x150px). The resolutionMultiplier parameter multiplies this default size.

Introduction

PhysicalObjects System

All the library is based in a proper object type named PhysicalObject. It’s a simple object created by the user to keep all the information about a body.

There are three types of objects, separated by the form: circular, square and custom. The basic structure for each one is:

Square:

```
name: {

    name: name,
    type: square,
    color: drawingColor,
    x: XaxisPosition,
    y: YaxisPosition,
    w: drawingWidth,
    h: drawingHeight,
    vx: velocityX,
    vy: velocityY,
    ax: accelerationX,
    ay: accelerationY,
    gravity: T/F,
    collision: T/F

}
```

Circular:

```
name: {

    name: name,
    type: circle,
    color: drawingColor,
    x: XaxisPosition,
    y: YaxisPosition,
    r: radius,
    vx: velocityX,
    vy: velocityY,
    ax: accelerationX,
    ay: accelerationY,
    gravity: T/F,
    collision: T/F

}
```

Custom:

name: {

name: name,
type: custom,
drawing: function,
x: XaxisPosition,
y: YaxisPosition,
collisionType: square/circle,
collisionX: collisionXpoint,
collisionY: collisionYpoint,
collisionW: squareCollisionWidth,
collisionH: squareCollisionHeight,
collisionR: circularCollisionRadius,
vx: velocityX,
vy: velocityY,
ax: accelerationX,
ay: accelerationY,
gravity: T/F,
collision: T/F

}

The required **function** of the propriety **drawing** needs to draw the form of the body and will be called when the object is going to be rendered.

Collision of Custom Type PhysicalObjects

There are algorithms on the library only to detect collision between circular and rectangular bodies. The custom bodies, because they have different shapes, need to adopt one of the two types of collision supported.

Environment constants

There is also another object that keeps the data related to the environment of the canvas:

```
globalConstants = {

    gravitationalacceleration: -0.2,
    backgroundcolor: "white",
    display: { h: 150, w: 300 }

}
```

All the object have their properties open to be changed by the user, so fell free to change, for example, the backgroundcolor propriety.

Velocity, acceleration and gravity

The canvas Y axis has higher values the lower the position, but the proprieties of velocity and acceleration use the more intuitive system, so a positive velocity makes the body go up.

Each PhysicalObject has proprieties related to velocity and acceleration. The acceleration proprieties increment their respective velocities each frame, and the velocities the position ones.

PhysicalObjects also have a property called gravity which keeps a boolean value. If it’s “true” the Y axis velocity is incremented by the value of gravity, saved on the globalConstants object.

Library functions

Physical Objects

Creating PhysicalObejcts

To create a PhysicalObject you have two options:

- **Creating manually**

Use physicalObjects[obejctName] = { propriety1: value, property2: value... }

Always make sure to use the default parameters, otherwise it may cause some problems. You still can add custom proprieties.

Example:

Square:

physicalObjects[“test”] = { name: ‘test’, type: ‘square’, color: ‘black’, x: 20, y: 75, w: 10, h: 10, gravity: true, collision: true };

Circular:

physicalObjects[“test”] = { name: ‘test’, type: ‘circle’, color: ‘black’, x: 20, y: 75, r: 5, gravity: true, collision: true };

Custom:

physicalObjects[“test”] = { name: ‘test’, type: ‘custom’, x: 20, y: 75, drawing: draw, gravity: true, collision: true };

Important: the collision of custom PhysicalObejcts isn’t defined on its creation. See **Collision – setCollision( )** to know how to set it up.

- **Creating with the functions**

Rectangular body

Use the function createPhysicalObject( ) and pass as parameters the following sequence: objectName, color, initialPosX, initialPosY, drawingWidth, drawingHeight, useGravity, useCollision.

Example:

```
createPhysicalObject( ‘test’, ‘black’, 20, 75, 10,  10, true, true );
```

Circular body

Use the function createCircularPhysicalObject( ) and pass as parameters the following sequence: objectName, color, initialPosX, initialPosY, circleRadius, useGravity, useCollision.

Example:

```
createPhysicalObject( ‘test’, ‘black’, 20, 75, 10, true, true );
```

Custom body

Use the function createCustomPhysicalObject( ) and pass as parameters the following sequence: objectName, drawingFunction, initialPosX, initialPosY, useGravity, useCollision.

Example:

```
function draw( ){
        display.fillStyle = ‘blue’;
        display.beginPath();
        display.arc(physicalObejct.test.x, physicalObject.test.y, 5,
                Math.PI, Math.PI*1.5);
        display.fill();
}

createPhysicalObject( ‘test’, draw, 20, 75, true, true );
```

Dealing with PhysicalObject’s proprieties

You can access all of the PhysicalObject’s properties by using:

physicalObjects.objectName.property = newValue

Or

physicalObjects[objectName].property = newValue

Example:

physicalObjects.test.vx = -20;

Or

physicalObejcts[“test”].vx = -20;

Deleting PhysicalObjects

To delete a PhysicalObject you can use:

delete physicalObjects.objectName

Or

deletePhysicalObject(objectName **)**

Example:

deletePhysicalObejct(‘test’);

Input

Adding inputs

To add an input you can use the function addInput( ). Inform the parameters keyCode(int), pressingFunction and releasingFunction(optional), identifier(optional).

Example:

```
function goUp( ){ }
function stopGoUp( ){ }

addInput( 38, goUp, stopGoUp, ‘playerJump’ );
```

All of the inputs are saved on an object called inputs and can be accessed.

The eventListener for the keyboard inputs are already defined.

Collision

Adding collisions

To add a collision use the function createCollisionHandler( ) using the parameters PhysicalObject1, PhysicalObject2 and collisionFunction.

Example:

```
function stopAll( ){ }

createCollisionHandler( physicalObjects.test1, physicalObjects.test2, stopAll }
```

The object collisionHandlers keeps all of the collision situations created and is open to changes.

Setting up CustomPhysicalObject’s collision

As told on the introduction about PhysicalObjects, custom shaped objects need to receive a circular or rectangular collision box to be understood by the algorithm. To add their boxes use the function setCustomCollision( ) with the parameters PhysicalObject, type(circle/square), posX, posY, width/radius(width to square type and radius to circle type), height(just used by circle type).

Example:

```
SetCustomCollision( physicalObjects.test, ‘square’,
physicalObjects.test.x, physicalObjects.test.y-10, 10, 20 );
```

Example2:

```
SetCustomCollision( physicalObjects.test, ‘circle’,
physicalObjects.test.x, physicalObjects.test.y-10, 10 );
```

Keep in mind that if you don’t use a position relative to the object the collision box will remain static.

Loop

Adding functions to Loop

Loop is the function that initiate the simulation and keeps calling itself, so every function that needs to be repeated each frame has to be added to it.

The function addToLoop( ) use as parameter a function and a priority which can be either ‘high’ or ‘low’. A high priority function will run first than the others, and a low priority will run after all of them. The default to priority is ‘low’.

Example:

```
function score( ){ }

addToLoop( score, ‘high’ );
```