myPhysic2.0 documentation

made by Wei-Chang

Introduction

This module is for someone who wants to easily build 2D physic world in their web projects.

The module is written by JavaScript, a programing language mostly used in web projects. You can insert it into your html file.

If you got any interest in it, read the next paragraph, and build a 2D physic world yourself.

Get start

Installation

There are 3 ways for you to install the module into your project.

1. **Download from GitHub and insert them into your html file:**

Step1: download:

The file is at <https://github.com/weichang0307/myModules>

You can use git to clone one into your project, navigate in your Terminal to your project and run this command.

git clone <https://github.com/weichang0307/myModules>

and you will get a folder named **myModules**with three JavaScript file inside.

Step2: insert them into your html file:

What you need to do is to insert them into your html file in the sequence of

**myMatrix.js** 🡪 **myVector.js** 🡪 **myPhysic.js** like this:

    <script src="./myModules/myMatrix.js"></script>

    <script src="./myModules/myVector.js"></script>

    <script src="./myModules/myPhysic.js"></script>

And you can use the module in any JavaScript file inserted after these three files.

1. **Insert the whole module with url:**

You can also insert the whole module into html file with this url: <https://29ykl1.deta.dev/myPackage>

    <script src="https://29ykl1.deta.dev/myPackage"></script>

The url will return a JavaScript file which include **myMatrix**, **myVector** and **myPhysic**.

So you can use the module in any JavaScript file inserted after this.

1. **Insert files separately with urls:**

If you want, you can also insert three JavaScript files separately like this:

  <script src="https://29ykl1.deta.dev/myMatrix"></script>    //myMatrix.js

    <script src="https://29ykl1.deta.dev/myVector"></script>    //myVector.js

    <script src="https://29ykl1.deta.dev/myPhysic"></script>    //myPhysic.js

If you need, you can also use them individually. But remember this:

1. myVector.js need myMartix.js inserted before it to work well.
2. myPhysic.js need myMatrix.js and myVector.js inserted before it to work well.

Build first 2D Physic world

Example:

Index.html

<!DOCTYPE html>

<html>

<head>

    <title>my first physic world</title>

</head>

<body>

    <canvas id="canvas"></canvas>

    <script src="https://29ykl1.deta.dev/myPackage"></script>

    <script src="./main.js"></script>

</body>

</html>

main.js

const canvas=document.getElementById('canvas')

canvas.width=500

canvas.height=500

const ctx=canvas.getContext('2d')

let fps=50

let world\_=new world(0,100,10)

let b1=new ball(200,100,20,10)

world\_.add(b1)

setInterval(()=>{

    world\_.update(1/fps)

    ctx.fillStyle='black'

    ctx.fillRect(0,0,500,500)

    world.draw\_helper(b1,'red')

},1000/fps)

Open the index.html, and you will see a dropping red ball.

Explanation:

* about [world](#world)

[world](#world) is a class that can include objects and simulate their behaviors in real physic world, when  [world\_.update(1/fps)](#world_update) is called, world\_ will calculate all objects added into the world\_ , and change objects’ properties to where they should be after 1/fps second if they are in real physic world. In other words, [world\_.update(1/fps)](#world_update) can push the whole world forward for 1/fps second.

Create a new [world](#world) like this:

let world\_=new world(0,100,10)

first two parameters are about the gravity in the world, the third one is about how many times the world should iterate when [world.update](#world_update) is called. More iteration means more accuracy.

Use [world.add(b1)](#world_add) to add b1 into world:

world\_.add(b1)

* about [ball](#ball)

[ball](#ball) is a physic object that can be added into [world](#world), the first two parameters are about its position, and the third is about its radius, the fourth is about its mass.

let b1=new ball(200,100,20,10)

* about drawing

The drawing tools provided by [world](#world) class can help you draw physic object on [HTML5 canvas](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API).

So if you want to use it, you must add canvas in your html file.

    <canvas id="canvas"></canvas>

And introduce it into your js file.

const canvas=document.getElementById('canvas')

canvas.width=500

canvas.height=500

const ctx=canvas.getContext('2d')

canvas context (ctx in this case) would be used when you want to draw something on canvas.

ctx.fillStyle='black'

ctx.fillRect(0,0,500,500)

canvas context is also needed in [world drawing tools](#world_drawhelper), but there is a default that your canvas context named ctx , so if your canvas context named ctx, you don’t need to add any parameter after color, or you should input your canvas context as a parameter like this:

    world.draw\_helper(b1,'red', ctx)

* Remember that in canvas, the direction of y axis is the downside of the web page. And the zero point is at the left top of the page.

The process of using this module

Part 1: initialize

1. Create a [world](#world)

Use let world\_=new world( gravity on x axis, gravity on y axis, iteration) to create a world

1. Create your physic objects

Create objects, [polygon](#polygon) or [ball](#ball)

1. Add objects into the world

use world\_.add(obj) to [add objects to the world](#world_add)

Part 2: update

1. Use [setInterval](https://developer.mozilla.org/zh-TW/docs/Web/API/setInterval) to set a loop:

setInterval(update,20)

1. Call [world.update()](#world_update) in the update function:

function update(){

world\_.update(20/1000)

}

Part 3: draw

1. Draw background in draw function
2. [Draw objects](#world_drawhelper) in draw function
3. Use [requestAnimationFrame](https://developer.mozilla.org/zh-TW/docs/Web/API/window/requestAnimationFrame) to call back in draw function
4. Call draw function

function draw(){

    //draw background

    ctx.fillStyle='black'

    ctx.fillRect(0,0,500,500)

    //draw objects

    world.draw\_helper(r1,'yellow')

    world.draw\_helper(r2,'white')

    //requestAnimationFrame

    requestAnimationFrame(draw)

}

draw()

world

constructor

constructor(gravityx,gravityy,iteration=50)

* gravityx: the gravity on x axis
* gravityy: the gravity on y axis
* iteration: the times world iterate when world.update is called, more iteration means more accurate simulation

properties

this.gravity=[gravityx,gravityy]

this.objs=[]

this.springs=[]

this.coefficients=[]

this.iteration=iteration

this.accuracy=1

* gravity: the gravity of the world (vector2)
* objs: an array of all the objects added into the world
* springs: an array of all the [springs](#ss_spring) added to the world
* coefficient: an array of all the [coefficients](#ms_coefficient) added to the world
* iteration: the times world iterate when [world.update](#world_update) is called, more iteration means more accurate simulation
* accuracy: an parameter for correction of position, if the accuracy=0 the world won’t make any correction of position

add()

add(obj)

add obj into the world

* obj: a physic object, a [polygon](#polygon) or a [ball](#ball)

delete()

delete(obj)

remove obj from the world

* obj: a physic object which has been added to the world, a [polygon](#polygon) or a [ball](#ball)

update()

update(time)

push the world’s time forward for <time> second

* time: the duration (second) you want to push the world’s time forward for

addSpring()

addSpring(a,b,origin\_dis,count,ap=[0,0],bp=[0,0],stretch=true,compress=true)

add a [spring](#ss_spring) to the world and return id of the spring, which you will need when [removing](#world_removeSpring) or [getting spring](#world_getSpring)

* a: an object connected to one end of the spring
* b: an object connected to the other end of the spring
* origin\_dis: origin length of the spring
* count: the elastic coefficient of the spring
* ap: a fixed point (vector2) of a, which one end of spring attached to
* bp: a fixed point (vector2) of b, which the other end of spring attached to
* compress: whether the spring work when compressed (Boolean)
* stretch: whether the spring work when stretched (Boolean)

removeSpring()

removeSpring(id)

remove a [spring](#ss_spring) which has been added to the world with id

* id: the id of the spring you want to remove from the world (you can get id when [adding the spring to the world](#world_addSpring))

getSpring()

getSpring(id)

get a [spring](#ss_spring) which has been added to the world with id

* id: the id of the spring you want to get (you can get id when [adding the spring to the world](#world_addSpring))

setCoefficient()

setCoefficient(material1,material2,friction=0,restitution=1)

set [coefficient](#ms_coefficient) between two [materials](#ms_material) in the world

* material1: one of materials’ name
* material2: the other material’s name
* friction: the coefficient of friction between two materials
* restitution: the coefficient of restitution between two materials

getCoefficient()

getCoefficient(material1,material2)

get the [coefficient](#ms_coefficient) between two [materials](#ms_material) in the world

* material1: one of materials’ name
* material2: the other material’s name

draw\_helper() static\*

static draw\_helper(obj,color,fill=true,through=1,ctx\_=ctx)

* it is a static method means that you can call this method with class itself without create any object

draw\_helper help you draw [ball](#ball) or [polygon](#polygon) with canvas

* obj: the object you want to draw (ball or polygon)
* color: the color of the shape
* fill: if true, the shape will be filled
* through: the transparent extent of the shape, 0 is totally transparent and 1 is not transparent
* ctx\_: the [canvas](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API) context you want to draw with

spring\_drawer() static\*

static spring\_drawer(spring,line\_width=3,discolor\_range=20,compress\_color='blue',stretch\_color='red',ctx\_=ctx)

* it is a static method means that you can call this method with class itself without create any object

spring\_drawer help you draw [spring](#ss_spring) with canvas

* spring: the spring you want to draw
* line\_width: the width of the spring
* discolor\_range: the range that spring discolor from compress\_color to stretch\_color
* compress\_color: the color of the spring when compressed
* stretch\_clolor: the color of the spring when stretched
* ctx\_: the [canvas](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API) context you want to draw with

physic\_object

* it is just a class for [ball](#ball) and [polygon](#polygon) to extend from, don’t new it and add it into [world](#world)

constructor

    constructor(x,y,mass=0,material='default',resistance=0,angular\_resistance=0)

* x: physic\_object’s position of x axis
* y: physic\_object’s position of y axis
* mass: mass of physic\_object
* material: the name of physic\_object’s [material](#ms_material) (string)
* resistance: physic\_object’s velocity will scale for (1-resistance) every second
* angular\_resistance: physic\_object’s omega will scale for (1- angular\_resistance) every second

properties

    this.world=null

    this.position=[x,y]

    this.velocity=[0,0]

  this.angle=0

    this.omega=[0,0,0]

    this.forces=[]

    this.mass=mass

    this.resistance=resistance

    this.angular\_resistance=angular\_resistance

    this.isgravity=true

    this.iscollision=true

    this.material=material

    this.collision=function(e){}

* world: it will become the world physic\_object added to when you add the physic\_object to a [world](#world)
* position: the position of the physic\_object (vector2)
* velocity: the velocity of the physic\_object (vector2)
* angle: the angular position of the physic\_object
* omega: the angular velocity of the physic\_object, which is a vector3 parallel to z axis
* forces: an array of all the forces added to the physic\_object
* mass: mass of the physic\_object
* resistance: physic\_object’s velocity will scale for (1-resistance) every second
* angular\_resistance: physic\_object’s omega will scale for (1- angular\_resistance) every second
* isgravity: whether the physic\_object affected by gravity (Boolean)
* iscollision: whether the physic\_object collide with other objects (Boolean)
* collision: a function which will be called when the physic\_object collide with other object, if return false that collision won’t cause any impact

add\_force()

add\_force(id,point=[0,0],force=[0,0])

add a [force](#fs_force) to the physic\_object

* id: the id of the force you want to add
* point: the position of the force point (vector2) relative to the center to the object (when angle=0)
* force: the force (vector2)

remove\_force()

    remove\_force(id)

remove the [force](#fs_force) which has been added to the physic\_object by id

* id: the id of the force you want to remove

get\_force()

get\_force(id)

get the [force](#fs_force) which has been added to the physic\_object by id

* id: the id of the force you want to get

polygon

* polygon extends from [physic\_object](#physicobject), which means it has all the properties and method physic\_object has

constructor

constructor(x,y,points=[],mass=0,material='default',resistance=0,angular\_resistance=0,

locate\_at\_center=true,get\_inertia=true)

* x: polygon’s position of x axis
* y: polygon’s position of y axis
* points: an array of all the vertex position (vector2) relative to the center of the polygon
* mass: mass of polygon
* material: the name of polygon’s [material](#ms_material) (string)
* resistance: polygon’s velocity will scale for (1-resistance) every second
* angular\_resistance: polygon’s omega will scale for (1- angular\_resistance) every second
* locate\_at\_center: if true, adjust the points and let the center of geometry at the center of the polygon
* get\_inertia: if true, set inertia as the moment of inertia calculated by points and mass (as the mass is uniform in polygon)

properties

    this.type='polygon'

    this.points=points

    this.area=this.get\_area()

    this.rmax=0

    this.inertia=0

    this.sides=[]

* type: the type of object (polygon)
* points: an array of all the vertex position (vector2) relative to the center of the polygon
* area: the area of the polygon
* rmax: the maximum distance from the center of the polygon to any point in the shape
* inertia: the moment of inertia of the polygon
* sides: an array of all the sides (vector2) of the polygon

get\_area()

get\_area()

return the area of the polygon

set\_mass()

set\_mass(mass,get\_inertia=true)

set the polygon’s mass as mass

* mass: the mass you want to set for polygon
* get\_inertia: if true, set inertia as the moment of inertia calculated by points and mass (as the mass is uniform in polygon)

set\_mass\_by\_density()

set\_mass\_by\_density(density,get\_inertia=true)

set the polygon’s mass as polygon’s area multiply by density

* density: density you want to set for polygon
* get\_inertia: if true, set inertia as the moment of inertia calculated by points and mass (as the mass is uniform in polygon)

ball

* ball extends from [physic\_object](#physicobject), which means it has all the properties and method physic\_object has

constructor

constructor(x,y,radius,mass=0,material='default',resistance=0,angular\_resistance=0,get\_inertia=true)

* x: ball’s position of x axis
* y: ball’s position of y axis
* radius: radius of the ball
* mass: mass of the ball
* material: the name of ball’s [material](#ms_material) (string)
* resistance: ball’s velocity will scale for (1-resistance) every second
* angular\_resistance: ball’s omega will scale for (1- angular\_resistance) every second
* get\_inertia: if true, set inertia as the moment of inertia calculated by radius and mass (as the mass is uniform in ball)

properties

    this.type='ball'

    this.radius=radius

    this.rmax=this.radius

    this.area=Math.PI\*this.radius\*\*2

    this.inertia=0

* type: the type of object (ball)
* radius: the radius of the ball
* rmax: the maximum distance from the center of the ball to any point in the shape
* area: the area of the ball
* inertia: the moment of inertia of the ball

set\_mass()

set\_mass(mass,get\_inertia=true)

set the ball’s mass as mass

* mass: the mass you want to set for ball
* get\_inertia: if true, set inertia as the moment of inertia calculated by radius and mass (as the mass is uniform in ball)

set\_mass\_by\_density()

set\_mass\_by\_density(density,get\_inertia=true)

set the ball’s mass as ball’s area multiply by density

* density: density you want to set for ball
* get\_inertia: if true, set inertia as the moment of inertia calculated by radius and mass (as the mass is uniform in ball)

force system

force system provide a way to add force to object, this might be useful when mimicking engine or some power system

force

every force is an object with three properties

* id: id of a force help you find it
* point: a fixed point of an object, which is a vector2 of the position relative to the center of the object (angle=0)
* force: a vector2 of the force (impulse per second)

add force

use [physic\_object’s add\_force()](#physicobject_addforce) method to add a force on fixed point of an object

    world\_=new world(0,0,10)

    r1=new polygon(100,100,[[0,0],[50,0],[50,50],[0,50]],10)

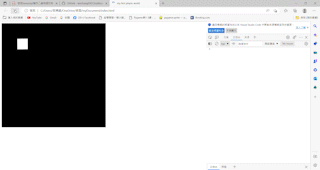
    r1.resistance=0.5

    r1.angular\_resistance=0.5

    world\_.add(r1)

    r1.add\_force('test',[25,25],[1000,0])

in this case, r1 will continuously force for [1000,0] on the fixed point of [25,25] from its center (the right bottom of the square), and it will look like this:



remove force

use [physic\_object’s remove\_force()](#physicobject_removeforce) method to remove force:

r1.remove\_force('test')

set force

use [physic\_object’s get\_force()](#physicobject_getforce) method to get force and change its properties

r1.get\_force('test').force=[-1000,0]

spring system

spring

every spring is an object with several properties

* id: id of a spring help you find it
* a: an object connected to one end of the spring
* b: an object connected to the other end of the spring
* origin\_dis: origin length of the spring
* count: the elastic coefficient of the spring
* ap: a fixed point (vector2) of a, which one end of spring attached to
* bp: a fixed point (vector2) of b, which the other end of spring attached to
* compress: whether the spring work when compressed (Boolean)
* stretch: whether the spring work when stretched (Boolean)

add spring

use [world’s addSpring()](#world_addSpring) method to add a spring

    world\_=new world(0,0,10)

    r1=new polygon(100,100,[[0,0],[50,0],[50,50],[0,50]],Infinity)

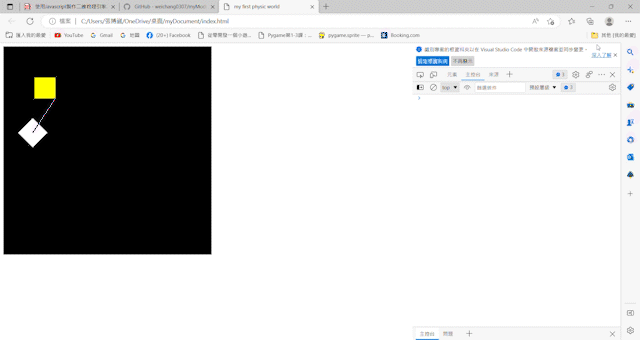
    r2=new polygon(300,100,[[0,0],[50,0],[50,50],[0,50]],10)

    world\_.add(r1)

    world\_.add(r2)

    spring1\_id=world\_.addSpring(r1,r2,100,100,[25,25],[0,0])

in this case it will add spring between r1 and r2, and it will look like this:



draw spring

You can see there is a line which represent the spring (red means stretch and blue means compress), and that is because I call [world’s spring\_drawer()](#world_springdrawer) method when drawing

world.spring\_drawer(world\_.getSpring(spring1\_id))

you can use this function by input a single [spring](#ss_spring), or you can also draw out all the spring in the [world](#world) like this:

    for(let i of world\_.springs){

        world.spring\_drawer(i)

    }

remove spring

use [world’s removeSpring()](#world_removeSpring) method to remove spring:

world\_.removeSpring(spring1\_id)

set spring

use [world’s getSpring()](#world_getSpring) method to get spring and change its properties

world\_.getSpring(spring1\_id).count=50

material system

when two object collide, different material causes different impact, and that is because there are different coefficient between different material.

material system help you set those coefficients easily.

material

every [physic\_object](#physicobject) has [material property](#physicobject_properties), you can set it as the name of the object’s material, and set the material’s coefficient in the world.

the material’s name must be a string, like this:

r1.material='material1'

if a [physic\_object’s](#physicobject) material hasn’t been set, it will be ‘default’:

material='default'

coefficient

those coefficients like coefficient of friction or coefficient of restitution are decided by both objects’ material anticipate in the collision

so coefficient should be set in the [world](#world), with two [material](#ms_material) as one of its properties

each coefficient has three properties

* object: an array which includes two material’s name
* friction: the coefficient of friction between two materials in object
* restitution: the coefficient of restitution between two materials in object

set coefficient

use [world’s setCoefficient()](#world_setCoefficient) method to set coefficient

setCoefficient(material1,material2,friction=0,restitution=1)

* the order of two materials doesn’t matter, it will all set as the coefficient between those two materials

here is an example:

world\_.setCoefficient('default','default',1,0.5)

get coefficient

use [world’s getCoefficient()](#world_getCoefficient) method to get coefficient

getCoefficient(material1,material2)

if the coefficient hasn’t been set, it will return this:

{object:'default',friction:0,restitution:1}