JavaScript Physics Library Documentation

You can see the library in action in my Wordpress theme Physical, which i have created for my portfolio www.christophpacher.com. Most of the code was written by me or by following some physics articles and papers. The resolution of contacts and collisions in real-time in a system of stacking objects is very complex and an area of on-going academic research. Thanks to Erin Catto of box2D for his notes and code samples about his sequential impulses algorithm. Credits for the line drawing algorithm go to Mathieu Henri from p01.org, who had the same Idea like I did, just seven years before me. I modified his implementation.

Why should you use this library and not the JavaScript port box2D? I think it is a little less complex and ready to use with graphical HTML objects. More over there is not the overhead of calculating the angular momentum, as it still will take several years until most of the web users have a browser that can rotate HTML objects. Then there is the possibility to use scriptaculous effects on the objects of the physics engine. The scale effect even works on all children HTML elements that are inside the parent HTML element. If you want to draw connection lines between your particles to form a mesh, this works out of the box.

Features and Notes:

- Axis oriented bounding box collision detection (AABB) and a sweep version of the algorithm for collision detection between fast objects (change this.collAABB(p1, p2) to this.collAABBsweep(p1, p2) in PhysicsWorld.js in line 97 to switch between the two)
- Second order Runge Kutta numerical integration for smaller integration errors and more stable spring forces
- Boxes can take energy out of a collision if mRestitutionN > 0. If this value is larger than 0.0 boxes will have a hard time to come to rest on top of each other.
- The Boxes are influenced by air drag. The drag coefficient mDragN is computed unscientifically (proportionally to box width and height) by updateDrag() after every scale update of the box. You can adapt this to your liking.
- The Boxes are influenced by contact friction proportionally to the collision velocity and mFrictionN
- It is possible to apply an impulse on a particle by using particle.mExternalImpulsesA.push([x,y,z]);
- To use scriptaculous with the physics engine I needed to adapt the core effects scale and move, so they interact with the engine. I extended them to MoveP and ScaleP, you find them in EffectsParticle.js. ScaleP can scale only around the centre of the particle. All combined effects can now easily be adapted to the physics engine, by using this two extended core effects, see ShakeP for an example.
- MoveP ensures that the particle ends up at the coordinates as defined in the options of the effect. MoveImP applies impulses. The vector defined by the options, divided by the change variable of the move effect, to be exact. So there is an ease in and out in the strength of the impulses applied over the time of the effect. The particle is still influenced by forces and collisions.

- Boxes can be dragged and dropped. Mouse screen coordinates can be transformed back into 3D, so drag and drop works with the 2D and 3D renderer. See the demo implementation and the code snippets in the getting started tutorial, for more information.
- Particles and/ or their collisions can be killed with world.killParticle(pP) and particle.killCollisions()
- I optimized the engine by using simple arrays for vectors and performing vector math directly with the array contents. Every function and class call is costly in JavaScript, when there are several thousand per second. By this, I could reduce the step time of the simulation of a 6 story pyramid from 90ms to 17ms on my Core2 Duo 2.4 GHz laptop in Firefox.
- The projection and drawing of the boxes is encapsulated in the renderer classes. R2D has no perspective projection and the origin is by default in the lower left corner of the viewport. R3D has perspective projection and the origin is in the middle of the view port. In both renderers the camera can be moved. There is no rotation or rolling implemented. In the demo app the camera can be moved with the keys cursor up, down, left right and page up and down.
- There are several types of additional forces, which can be applied to selected particles. Every force can be turned on and off. There are general Omi Forces like wind and Omni forces that are mass independent like gravity. One instance of these can affect several particles. A particle can always be removed from or added to an Omni force. Moreover there are attraction, repel and spring forces, one instance for every force between two particles. In the demo i only implemented the possibility to add spring/repel forces between particle p20 and all other particles.
- Boxes fall asleep to save CPU cycles. The sleep algorithm was specifically adapted to work with all types for forces as well as drag and drop. Collisions still need to be calculated. (the box colour is changed from red to green in the renderers draw function. If you want to disable the colouring you have to edit the drawing function inside the render)
- I adapted Scriptaculous with 3 lines of code, so that no frames are skipped when the browser lags behind. Frame skips in animations would be seen by the physics engine as a larger impulse, as the engine runs by fixed time steps and does not skip frames, to prevent the simulation from exploding. Large Impulses would result in larger velocities, which would be different from the intended animation. See timePos = this.MyOldPos + 15; in effects.js line 278 for the changes. You need to adapt every new Scriptaculous version if you want to upgrade, or you just ignore it if it does not bother you.
- You can draw a line between two particles with myWorld.addParticleLine(particle1,particle2);

Getting Started

The following guide describes the code of the demo app.

Includes:

You have to put the following includes in the head of your html file. Be sure that the includes point to the position where you placed the folders with the JavaScript files.

```
<script type='text/javascript'
src='http://ajax.googleapis.com/ajax/libs/prototype/1.6.1/prototype.js'></script>
 <script type='text/javascript'
src='http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js'></script>
```

I decided to pull jQuery and prototype from the google online storage

```
<script type='text/javascript'>try{jQuery.noConflict();}catch(e){};</script>
```

Unfortunately while adapting the physics engine to work with Wordpress I mixed in some jQuery code. Set jQuery to noConflict mode to go along with prototype.

```
 <script src="jQuery/jquery.batchImageLoad.js" type="text/javascript"></script>
```

The batch image loader preloads the images for the line drawing feature.

```
<script src="scriptaculous/scriptaculous.js" type="text/javascript"></script>
<script src="scriptaculous/effects.js" type="text/javascript"></script>
```

This is the Scriptaculous file that was edited by me.

```
 <script src="physics/Vector.js" type="text/javascript"></script>
 <script src="physics/PhysicsWorld.js" type="text/javascript"></script>
 <script src="physics/PhysicsForce.js" type="text/javascript"></script>
 <script src="physics/PhysicsParticle.js" type="text/javascript"></script>
 <script src="physics/PhysicsCollision.js" type="text/javascript"></script>
 <script src="physics/EffectsParticle.js" type="text/javascript"></script>
 <script src="physics/Renderer.js" type="text/javascript"></script>
 <script src="MyUtilities.js" type="text/javascript"></script>

 <link rel="stylesheet" type="text/css" media="screen" href="style/style.css" />
```

Don’t forget to include your style sheet. My code searches in all style sheet files for all the style values that need to be adapted when one of the elements is scaled.

Viewport and Renderer

First get the dimensions of the browser window and define the viewport dimension and position:

```
var windowDim = MyUtilities.getWindowDimensions();
viewportPadding = [50, 50];
viewportOrigin = [viewportPadding[0], viewportPadding[1]];
viewportDim = [windowDim.width - 2*viewportPadding[0], windowDim.height -
2*viewportPadding[1]];
```

Create a new Renderer (R3D or R2D), pass the window dimensions and define the viewport inside the window:

```
var myRenderer = new Renderer.R3D(windowDim, viewportOrigin, viewportDim);
```

viewportOrigin defines the upper left corner of the viewport in browser coordinates (in px, origin is the upper left of the browser window, y+ points down).

You can set and animate the position of the camera by setting its position vector:

```
myRenderer.mCameraPosV3D = [viewportDim[0] / 2 * myRenderer.px2m, viewportDim[1] / 2
* myRenderer.px2m, 0];
```

World

It is time to create your world and pass the renderer you just created:

```
var myWorld = new Physics.World3D(myRenderer);
```

Then define the bounds of the world. Particles collide with these. Default is min[0,0,0], max [100,100,100]

```
myWorld.mBoundsMinV3D = [0, 0, 0];
myWorld.mBoundsMaxV3D =  [viewportDim[0] * myRenderer.px2m, viewportDim[1] * 2 *
myRenderer.px2m, 10];
```

You can start and stop the simulation with:

```
myWorld.start();
myWorld.stop();
myWorld.toggleStartStop();
```

But you have not created any particles yet.

Particles/boxes

A box can be an HTML element like a or <img>. This element can already exist in your HTML document. You then just need the ID of the element or you can create the element on the fly, like i do it in the demo.

```
var element = document.getElementById("myID");
var element = document.createElement('div');
```

You can write content inside the element by writing to the innerHtml node:

```
element.innerHTML = ' p' + myWorld.mParticlesA.length;
```

If the DOM Element we use as our Particle Sprite has css dimensions other than width and height (eg. fontsize, padding, margin etc.) and/or sub elements (eg. images, divs) that need to be scaled too, we need to mark the element and save the start dimensions at 100% scale.

```
element.scaleContent = true;
```

```
myRenderer.saveStartDimensions(element, 0, element);
```

Now add the element to the document body and use the id of the element to create a new particle in the engine, that uses the element as its sprite. The Html element also needs a reference to the particle in the engine:

document.body.appendChild(element);

```
//position in meters, box extensions from postion in m, mass in kg, start velocity in m/s2, the representing DOM element
element.particle = myWorld.addParticle([posx, posy, posz], [BoxWidth * myRenderer.px2m /2, BoxWidth * myRenderer.px2m /2, BoxWidth * myRenderer.px2m /2], 1, [0,0,0], element);
```

Set the restitution and friction of the box to your liking:

```
element.particle.mRestitutionN = 0.0;
element.particle.mFrictionN = 0.5;
```

If you want to manually set a new scale of the particle do it by setting:

```
element.particle.mNewScaleN = 0.5;
```

The particle will then be scaled by the engine and updateDrag() is automatically called. You can kill a particle by using:

```
myWorld.killParticle(p);
```

All its references to collisions, forces and lines will be deleted too.

Forces

Let's start with the general forces like wind and gravity that affect several particles all in the same way. There is only one difference between them, as the acceleration of gravity is independent from the mass of the particle. These two forces are implemented as Omni and OmniMassInd:

```
// parameters: id, world, [affectedParticleN, .....], acceleration vector
var wind = new Physics.Omni('wind', myWorld, MyUtilities.copyArray(myWorld.mParticlesA), [2, 0, 0]);
var grav = new Physics.OmniMassInd('grav', myWorld, MyUtilities.copyArray(myWorld.mParticlesA), [0, -9.8, 0]
```

After the creation of an Omni force you can remove or add a particle by using:

```
wind.removeParticle(p);
wind.addParticle(p);
```

Then there are more specific forces that exist between two particles.

The first one is Attraction, which can repel too, if its' strength is set negative. You have to make sure that the particles are awake for the force to work instantly:

```
// parameters:
// the id string of the force
// the world
// the affected particle
// the source particle
// is the source attracted to the other particle
// the strength of the attraction, repells when negative
// the min
// and max distance (m) between the particles for the force to work in
new Physics.Attraction('attrP20', myWorld, p, myWorld.mParticlesA[20], true, -10, 0.5, 3);
p.setAwake(true);
myWorld.mParticlesA[20].setAwake(true);
```

Then there are Spring forces, that push and pull the connected particles, until the rest lenght of the spring is reached:

```
// parameters:
// the id string of the force
// the world
// the affected particle
// the source particle
// is the source pulled to the other particle by the spring
// the spring constant, the larger, the stiffer
// the restlenght of the spring in m
// the damping, how much kinetic energy is lost because of stretching and compressing of the
spring, controls how fast the particles come to rest
new Physics.Spring('sprP20', myWorld, p, myWorld.mParticlesA[20], true, 7, 2, 0.2);
p.setAwake(true);
myWorld.mParticlesA[20].setAwake(true);
```

You can set all forces on and off and kill forces by using e.g.:

```
wind.setOn(false);
wind.setOn(true);
myWorld.killForce(wind);
```

You can give every particle an impulse, an instant acceleration, by pushing an acceleration vector on the mExternalImpulsesA array. The vector will be taken from the stack in the next simulation step. You have to make sure, that the particle is awake, otherwise the impulse will not affect the particle:

```
p.setAwake(true);
p.mExternalImpulsesA.push([0, 0, 0.5]);
```

Effects

As I already mentioned in the notes, all Scriptaculous effects need to be adapted before they can be used with the physics engine. I already did this, for Scale, Move and Shake. All combined effects that use Move and Scale can be easily adapted. I will go through the process on the example of the effect Puff.

First I copied over the effect declaration of Puff from Scriptaculous/effects.js to physics/EffectsParticle.js and renamed the class to Effect.PuffP. The effect creates internally two effects that run in sync: Effect.Scale and Effect.Opacity. All that has to be done is to rename Effect.Scale to Effect.ScaleP and we are finished, as Effect.Opacity did not need to be adapted to the engine. The result looks like this:

```
Effect.PuffP = function(element) {
 element = $(element);
 var oldStyle = {
  opacity: element.getInlineOpacity(),
  position: element.getStyle('position'),
  top:  element.style.top,
  left: element.style.left,
  width: element.style.width,
  height: element.style.height
 };
 return new Effect.Parallel(
 // changes: replaces Scale by ScaleP
  [ new Effect.ScaleP(element, 200,
    { sync: true, scaleFromCenter: true, scaleContent: true, restoreAfterFinish: true }),
   new Effect.Opacity(element, { sync: true, to: 0.0 } ) ],
   Object.extend({ duration: 1.0,
    beforeSetupInternal: function(effect) {
     Position.absolutize(effect.effects[0].element)
    },
    afterFinishInternal: function(effect) {
      effect.effects[0].element.hide().setStyle(oldStyle); }
   }, arguments[1] || { })
  );
};
```

You can now for instance use the effect before you kill a particle, like i do it in the demo:

```
new Effect.PuffP(elem, {duration:0.7, fps: 20, afterFinish: function(effect) {
     effect.effects[0].mParticleP.mWorldW3D.killParticle(effect.effects[0].mParticleP);
}});
```

Check the demo app for more examples how to use the already adapted effects. You should then have no problems creating your own effects.

For more information about the handling of effects, like their options or how to queu or sync them, check the scriptaculous help.

Drag and Drop

Drag and Drop in combination with normal clicks on particles and a working wakeup and sleeping system, is a little tricky. It is best you take a long look at the implementation of the demo app. I only describe some key points of my implementation. The rest should be obvious from my code. First you need two different moveHandler functions that get the mouse coordinates, one for IE and one for all the other browsers. Most of the magic happens in the mouseUpdater function, which is called periodically. There I check if the mouse button has been held down on a particle for longer than 2 engine steps. Only then I set the flag, that the particle is dragged. If it is dragged, I activate all flags, so that the engine no more computes forces, collision impulses, does not integrate the velocity and position and that it is controlled externally (by the user). Then there is a special case, if the particle has spring or attraction forces. Then all the connected particles need to be woke up periodically, because the code that would wake them up otherwise, located in the force calculation during integration is not called when the particle is dragged. From then on you just have to position the particle according to the mouse coordinates. The myRenderer.screen2world() function nicely projects the mouse screen coordinates back into the 3D simulation space