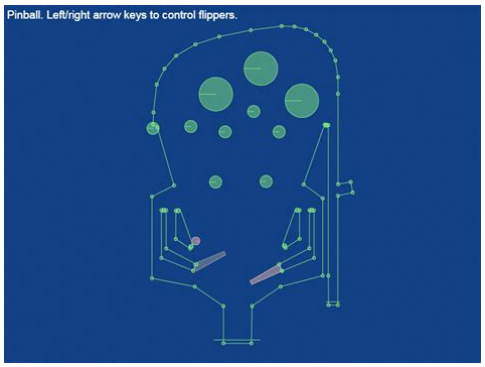
box2d javascript manual



File Name: box2d javascript manual.pdf
Size: 4457 KB
Type: PDF, ePub, eBook
Category: Book
Uploaded: 1 May 2019, 16:11 PM
Rating: 4.6/5 from 717 votes.

Status: AVAILABLE

Last checked: 17 Minutes ago!

In order to read or download box2d javascript manual ebook, you need to create a FREE account.

Download Now!

eBook includes PDF, ePub and Kindle version

Book Descriptions:

We have made it easy for you to find a PDF Ebooks without any digging. And by having access to our ebooks online or by storing it on your computer, you have convenient answers with box2d javascript manual . To get started finding box2d javascript manual , you are right to find our website which has a comprehensive collection of manuals listed.
Our library is the biggest of these that have literally hundreds of thousands of different products represented.

Book Descriptions:

box2d javascript manual

Uncaught TypeError Box2D.JSDraw is not a constructorLooking at the Box2D variable, it seemed to be a Promise. Tested on latest vesions of Chrome, Edge and Firefox The source code is translated directly to JavaScript, without human rewriting, so functionality should be identical to the original Box2D. For general notes on using the bindings, see the ammo.js project a port of Bullet to JavaScript using Emscripten, many of the details of wrapping classes and so forth are identical. Look in the tests folder and find the template.js file. CreateBody new b2BodyDef . CreateBody bodyDef ; CreateJoint jointDef , b2WheelJoint ; JSDraw ; Here are the two functions mentioned above, as an example of how you would wrap the passed b2Color and b2Vec2 parameters and use them in your drawing. This example is to draw on a HTML5 canvas GetFixtureA . GetFixtureB ; Here is the callback used in the testbed to find the fixture that the mouse cursor is over when the left button is clicked Reload to refresh your session. Reload to refresh your session. Over the years it has been ported to a number of different languages. Today there are two version in JavaScript, Box2dweb and Box2djs. The latter one is sadly no longer supported so my suggestion is using the first. It is a direct port from Box2DFlash 2.1a and is the one that this tutorial is using. This could be a bit weird but no worries. Read more here. It is however possible to put together multiple element to create concaves. This means that we have to paint the ourselves see the Loop section. Notice Box2D however provides a debugging feature where it can draw the objects itself, search for debug in the demo source to see how to use it. Then it calls box2d.addToWorld that adds the shape to Box2D. Next section explains how that function works. Usually you see it like bodies are containers which have a position and then contains fixtures that have a shape, density, friction and collision control.

- **box2d javascript tutorial, box2d javascript reference, box2d javascript manual.**

In this demo all fixtures uses the sam default fixture definition. In this demo it has three steps. Read more about it here. Both Circle and Box have their own.draw method. I will not explain these in detail, but they basically get the position from the world object and then uses it to paint itself on the canvas with help of the ctx. Here it the.draw method from Box. Programmers can use it in their games to make objects move in realistic ways and make the game world more interactive. From the game engines point of view, a physics engine is just a system for procedural animation. Most of the types defined in the engine begin with the b2 prefix. Hopefully this is sufficient to avoid name clashing with your game engine. If not, please first consult Google search and Wikipedia. You can get these tutorials from the download section of box2d.org. You should be comfortable with compiling, linking, and debugging. However, not every aspect is covered. Please look at the testbed included with Box2D to learn more. The latest version of Box2D may be out of sync with this manual. A testbed example that reproduces the problem is ideal. You can read about the testbed later in this document. We briefly define these objects here and more details are given later in this document. They are hard like a diamond. In the following discussion we use body interchangeably with rigid body. A fixture puts a shape into the collision system broadphase so that it can collide with other shapes. A 2D body has 3 degrees of freedom two translation coordinates and one rotation coordinate. If we take a body and pin it to the wall like a pendulum we have constrained the body to the wall. At this point the body can only rotate about the pin, so the constraint has removed 2 degrees of freedom. You do not create contact constraints; they are created automatically by Box2D. Box2D supports several joint types revolute, prismatic, distance, and more. Some joints may have limits and motors.

For example, the human elbow only allows a certain range of angles. For example, you can use a motor to drive the rotation of an elbow. Box2D supports the creation of multiple worlds, but this is usually not necessary or desirable. The Box2D solver is a high performance iterative solver that operates in order N time, where N is the number of constraints. Without intervention this can lead to tunneling. First, the collision algorithms can interpolate the motion of two bodies to find the first time of impact TOI. Second, there is a substepping solver that moves bodies to their first time of impact and then resolves the collision. The Common module has code for allocation, math, and settings. Finally the Dynamics module provides the simulation world, bodies, fixtures, and joints. These tolerances have been tuned to work well with meterskilogramsecond MKS units. In particular, Box2D has been tuned to work well with moving shapes between 0.1 and 10 meters. So this means objects between soup cans and buses in size should work well. Static shapes may be up to 50 meters long without trouble. Unfortunately this will lead to a poor simulation and possibly weird behavior. An object of length 200 pixels would be seen by Box2D as the size of a 45 story building. Keep the size of moving objects roughly between 0.1 and 10 meters. Youll need to use some scaling system when you render your environment and actors. The Box2D testbed does this by using an OpenGL viewport transform.The billboard may move in a unit system of meters, but you can convert that to pixel coordinates with a simple scaling factor. You can then use those pixel coordinates to place your sprites, etc. You can also account for flipped coordinate axes. If your world units become larger than 2 kilometers or so, then the lost precision can affect stability. Use b2WorldShiftOrigin to support larger worlds. I recommend to use grid lines along with some hysteresis for triggering calls to ShiftOrigin.

This call should be made infrequently because it is has CPU cost. You may need to store a physics offset when translating between game units and Box2D units. The body rotation is stored in radians and may grow unbounded. Consider normalizing the angle of your bodies if the magnitude of the angle becomes too large use b2BodySetAngle. So when you create a b2Body or a b2Joint, you need to call the factory functions on b2World. You should never try to allocate these types in another manner. These definitions contain all the information needed to build the body or joint. By using this approach we can prevent construction errors, keep the number of function parameters small, provide sensible defaults, and reduce the number of accessors. So you can create definitions on the stack and keep them in temporary resources. These are created via b2WorldCreateBody. It has been used in many games, including Crayon Physics Deluxe, winner of the 2008 Independant Game Festival Grand Prize. In this tutorial we shall use the Box2d library to write a simple Hello World program. It used version Box2d ver. 2.1 at the time of writing this article. Try clicking anywhere in the white area and a solid object would be created at the point and fall down. The fall would be realistic. And whatever is falling and staying fixed, are objects. The gravity parameter is a vector which indicates the force magnitude and direction. If you are developing a super mario kind of game then you need a world with normal gravity, so that things fall downwards by default. If you are writing a car racing game where car move in any direction, then you need to remove the gravity. Can be circle or polygon set of vertices. 3. Body Defines the point where the object is and what is its type dynamic, static or kinetic Like a wall or the floorLike the game characters. Dynamic bodies can collide with all other kinds of bodies. Things should be between 0.1 10 metre for proper simulation.

http://ninethreefox.com/?q=node/10810

When drawing on screen, a certain scale is needed to convert from metre to pixels. Over here I have used a scale of 100, which means, an item 1 metre wide would be drawn as 100 pixels wide. This is achieved by calling the Step method of the world object. As the world moves the position, velocity etc of all bodies change according to the laws of physics. It draw the bodies according to their shapes and also marks the radius of circles and joints if any. The scale is the conversion factor that converts box2d units to pixel. A scale of 50 means that 1 metre in box2d would be draw as 50 pixels

on the canvas. There are a few more properties to set the details of which can be found in the documentation. Most important is the change of coordinate system. Box2d works in cartesian coordinate system which is like a graph with x axis increasing positively rightwards and y axis increasing positively upwards. And the origin is lower left corner. In the canvas the x axis increases to right as usual, but the y axis increases downwards. Means upwards it is negative. And the origin is the top left corner. That could be somewhere near 0,500 in canvas coordinates. The scale changes the direction of the y axis to its opposite, that is, it starts increasing upwards. Now the canvas is more like a cartesian coordinate graph and objects draw would appear correct. It is very easy to write a complete game using just box2d and a few graphic sprites. If you wish to build an html5 game with realworld like physics then give box2d a try. Writes about Computer hardware, Linux and Open Source software and coding in Python, Php and Javascript. It is not currently accepting answers.Update the question so its ontopic for Stack Overflow.Which one is the best For example, Matter.js and p2.js both seem to be pretty solid. Check out Matter.js and p2.js What I found with the version difference is that some code from iOS works doesnt work in the JS version and the dev halt.

https://johannstraussensemble.at/images/casio-ctk-480-keyboard-user-manual.pdf

After that, I used half a week to change all the code to box2dweb and every thing work fine as same as my iOS box2d game. So the version difference matters a lot. As of August 2013 its more uptodate than the other ports Ive found, and the demos seem to work. It has all the features of Box2D and liquid physics features. Its ported using emscripten, so performance is decent. Are there any tutorial out there especially how to actually get the library into my javascript. Documentation seems a bit scarce Im going down this path. If you need the API documentation for Box2DWeb, check out Box2DFlash. So the API is the same. Browse other questions tagged javascript box2d or ask your own question. Like fretless. This is often sufficient for JumpnRun or RPG games, but sometimes you need some more accurate and realistic physics.So lets use that in Impact! All Box2D Classes are now members of the Box2D. object. So for instance to refer to the b2BodyDef class you can use Box2D.Dynamics.b2BodyDef. I also wrapped the source into a module, to be easily loadable by Impact. For this, we just need to define a gravity vector and create an instance of b2World This is different from Impact, where the position of an entity is set for the upper left corner of the entity. Also, the SetAsBox method takes halfwidth and halfheight parameters. Thus box we created is 10 by 10 units in size and because its center is at 100,100, it will stretch out from 95,95 to 105,105. Your objects should be roughly between 0.1 and 10 meters in size for most games this means that simply taking the pixel sizes as foundation for the physics could introduce some artifacts. Therefore, we use a global scaling factor to convert from our pixels to meters Box2D.SCALE. By default it is set to 0.1 that means that 1 pixel equals 0.1 meters. For this, I wrote the ig.Box2DGame and ig.Box2DEntity classes, that do all the dirty work for you. See the source code of the physics example game.

http://chamabusinesscenter.com/images/casio-ctk-4000-keyboard-manual.pdf

However, instead of just creating one physics object for each tile, it tries to find larger rectangle regions of tiles to combine into one object. In my tests, this worked quite nice even for larger game worlds. The performance of Box2D seems to be mostly bound by the number of collisions and not the number of objects in th world. Its update method takes the position of the body from the physics simulation and converts it back to Impacts unit system. This update method completely bypasses the one the base ig.Entity that normally moves the entity. The answer is Box2D and here we show how to get started with it in JavaScript.a Ideal for the next blockbuster game in a browser. This includes realistic details like gravity, friction, elasticity, jointed bodies, collisions and more. There are also a few successful games that make use of it, including Angry Birds. In this article we are going to use a JavaScript port, box2dweb. There are other JavaScript ports and they work in roughly the same way only the details differ. Indeed the same is true of the different language versions of Box2D and what this means is once you have mastered it in one language you should be able to use it in another of

course you would have to master the other language but this is a small matter. It is also assumed that you have a favourite JavaScript IDE to work with. Im not going to go into details of how to create a web page or how to edit and debug JavaScript. You dont have to know much and as long as you have a rough idea of how the world works the following explanations should be enough. It works in the SI system of units which means distances are measured in meters, angles in radians and forces in Newtons. In fact we need to go a step further here. Box2D isnt about graphics; it computes the movement of real world objects using world coordinates i.e. meters. You then add physical bodies whose motion is simulated on request.

There are a number of different variations on a body that you can add to a World, but for this introduction we will just look at a small subset of the possibilities. To give it a shape and other properties you have to add a fixture to it. You can think of this as the body carrying the fixture around as if it was a cutout shape attached to it. So to add, say, a circle to the World you would first have to create a body which sets the position and velocity and then create a fixture with the shape of a circle and attach this to the body. All it is going to do is create a circle and let if fall under gravity later we will make it bounce. If you dont want to do this you dont have to, but it is fairly standard. It handles the interrelation of various solid objects. It includes gravity, forces, rotations, collisions and more. It does not include the actual drawing of the objects.I found a good build of box2djs at github. It is a 1file version based on the most recent version of box2d. It also does not have a requirement for any external javascript libraries. The other popular port is based on an older version and requires prototype.js. I added this code to the start of the box2d.js, and it works like a charm in IE9 and IE10. Ive had horrible experience with the canvas simulation experience in older versions of IE, so Ill wait until later to worry about that. For now, I just have it added to the full version of the source. However, a minification should not be much of a problem.A simple animation consists of the following componentsThe one caveat is that on a canvas the upper left hand corner is 0,0, while with box2d, the lower left hand corner is 0. I simply inverted the value in the drawings. However, there are plenty of other cleaner methods for handling this. It has the same userfriendly approach to its API that the rest of Phaser does,View them online. Run them, play with them and edit the code to get a feel for how it all works.

If all you need is to calculate when a couple of rectangles overlap, it’s not a big deal, but for games that involve lots of irregularly shaped sprites moving and reacting to each other realistically, you’ll want to use an industrial strength tool to get the job done. For an overview of a number of different engines, take a look at Chandler Prall’s JavaScript Physics Engine Comparison. One reason for this is that it was used as the simulation engine behind megahit Angry birds, whose physicsbased gameplay required a bulletproof physics implementation. Box2D is the 2D physics engine of choice for many games, and thus was ported wholesale over to Flash and ActionScript a while back. To get a JavaScript version, some ambitious folks have taken advantage of the similarities between ActionScript and JavaScript and created a converter that converted the ActionScript to JavaScript. So that you don’t go insane with all the namespacing, it’s recommended that you pull out some of the mostused classes into easytoaccess variables, as shown below This example will wrap many of the more verbose pieces of Box2D in wrapper classes to simplify the API you need to expose to your game. It take two parameters, a gravity vector and whether or not it should skip simulating inactive bodies. 2D Vectors in Box2DWeb are created using the b2Vec2 object, whose constructor takes in x and y components. Creating a world with earth gravity that puts inactive objects to sleep would look like This can, however, lead to situations where you need to be careful to progamatically wake up objects when you manually move them to a state where they should be simulated like when you pick up a box with the mouse cursor and leave it hanging in mid air. Let’s settle on a scale of 30 pixels

per meter for now, but you can adjust this as you like. Step takes three parameters a time step, the number of velocity iterations.

Box2D works best when this is a fixed amount of time and not a variable amount of time dependant on the framerate. This means that you'll need to do some house keeping to make sure that the physics simulation runs at a constant frame rate, even when being driven by requestAnimationFrame. This will be used for the first few examples and a helper method to activate it is shown below The first is that since requestAnimationFrame stops firing when your game's tab is unfocused, if you run your simulation code from a requestAnimationFrame loop you'll want to put in an upper time limit on the size of dt to prevent your game from fastforwarding when the user refocuses the tab. If you're building a multiplayer game, it's probably not a good plan and using setTimeout or setInterval might be better. In order to give it some life, we'll need to add some Box2D bodies to it. Bodies are what Box2D calls the objects that it simulates. Static and kinematic bodies are used for walls and platforms that don't need to react to collisions, while dynamic bodies are used for everything that's needs to be simulated normally. Both static and kinematic bodies are treated as if they have infinite mass and can be moved manually by the developer. Kinematic bodies can also have a velocity. Static and kinematic bodies don't collide with other static and kinematic bodies, only with dynamic bodies. The body definition defines the initial attributes of the entire body, such as it's position, velocity, angular velocity, damping the inverse of bounciness and boolean attributes such as whether to start out active, if the body is allowed to sleep. You can also decide whether the body should be treated as bullet, which garners it more processor intensive but accurate collision detection to prevent small, fast moving objects from passing through other objects entirely. You can reuse body definitions if you like. This userData is often used to tie the physics object back to a graphical representation.

A body can have one or more fixtures. All fixtures are rigid inside of a single body, which means that the individual fixtures don't move relative to each other but rather combine to create a single rigid body. Fixtures also contain additional information such as density, friction, restitution, and collision flags that control with other fixtures to collide with. You can also flag fixtures as sensors to prevent a collision response but still receive notification of a collision. Fixtures are created by calling b2Body.CreateFixture and passing in a fixture definition. Box2D supports two different types of shapes. The first type, b2CircleShape represent, not surprisingly, shapes that are perfect circles. The second type b2PolygonShape are used for shapes that can be represented with a series of vertices. The vertices of b2PolygonShape 's must define a convex polygon in clockwise order if your vertices are counterclockwise, your shape won't collide with anything. You must add multiple fixtures to your body instead. Once thing to remember is that your collision shape does not need to match your game sprite exactly, but can instead be a simplified representation of that shape. So even if your video game character running around with a big bazooka doesn't look like a convex shape, modeling them as one is probably ok. Concave polygons tend to get stuck on things, so even if you could realistically model Bazooka guy, he might be better off as concave anyway so his Bazooka doesn't get caught on ledges and leave him dangling off cliffs unrealistically as a multifixture rigidbody. This class will take in the physics object from the previous section and go through the steps necessary to add a new body to the world The first is to keep a list of bodies separate from Box2D and loop over those after each step and render them. The second is to use the world.GetBodyList to return the first body and then call body.GetNext to return the next element.

This works because Box2D keeps a linked list of bodies to iterate over with. You can pull the element you've associated with the body out by calling GetUserData. The easiest way to handle this is simply to add in a draw method to our simple Body wrapper class that looks at its type and then renders itself by drawing the appropriate shape. To make this more applicable to normal games as well, bodies will also support drawing an image. The global scale transform can be handled outside of the

main loop, while the latter two transformations both need to be applied per object. This method will save the current transformation matrix, translate and then rotate the matrix so that the object can be drawn centered at the origin and then will draw the object based on the shape. In this example, the shape will be drawn if the object has a color property set. Polygon shapes draw a path using the same points as the shape definition. Blocks just use the fillRect method to draw a rectangle. In order to achieve this you must first be able to determine the object at a specific pixel position on your canvas. Given the returned body you could then loop over each of the bodies fixtures and match the position against the exact shapes. Luckily Box2DWeb has wrapped this functionality up in a single method call QueryPoint. With Box2D there are three ways you can get an dynamic object moving set its velocity, apply a force or apply an impulse. One thing you should not do is just try to set a dynamic object's position, as this effectively breaks the simulation and can cause unexpected effects with objects behaving badly. If you really need you move an object to a position, you can call body.SetTransform, but this method is undocumented in the Box2DFlash documentation, which should give you an idea of whether it's a good practice or not. Forces are good to use when you want consistent speed over a number of frames, while impulses are great as you can just set and forget them.

You might use the former to keep a player running at a consistent speed while the latter works well for things like explosions. Both ApplyForce and ApplyImpulse take a vector as the first argument that controls direction and strength. They also both take a second argument that defines the point where the force is applied. If you don't want the object to torque i.e. start to rotate, you can apply the force to an object's center using body.GetWorldCenter. A lot of times in games, however, you don't necessarily want your player to have to slow down to a stop before changing direction. You can defy the laws of physics by directly setting the velocity on an object as opposed to applying forces and impulses. If you set the velocity each frame, you can get defacto consistent speed that is useful in things like space shooters and platformers. The only thing to watch out for is that dynamic objecttoobject interactions will be slightly outofwhack when two objects of different masses that both have their velocity set interact. While you might be tempted to try to keep reseting the angular velocity to try to prevent a object from rotating, this won't work well and you're better off setting the fixedRotation property on the body definition or calling SetFixedRotationtrue. If you want to visually see the differences between ApplyImpulse, ApplyForce and SetLinearVelocity, try swapping the call to ApplyImpulse out with the others to see the effect. This is because the Box2D assumes objects have a uniform density controlled by the density property in the fixture definition and so larger objects weigh more and need more force to move. RayCast is particularly useful for doing things like visibility detection can enemy A see the player and determining what bullets and shrapnel will hit. The first BeginContact occurs when contact is initiated between two objects. The second EndContact is triggered when two objects stop touching.

The third callback PreSolve, is called on every collision, but before the collision is sent to the solver to resolve. All three of these only provide details on what two objects are in contact and normal details with what's known as the contact manifold, but don't give us a strengthofimpact impulse. As bodies are crashing into each other all the time, PostSolve will get called very frequently, so it's important not to do too much processing each time the callback is triggered. Two bodies coming into contact with each other could each only get a single BeginContact and EndContact callback, but will mostly likely get a number of PreSolve and PostSolve callbacks. If you need to make changes to any aspects of the physics simulation as a result of one of these four callbacks, you should queue up what needs to be done and then make those changes after the step call. Secondly, the data structure for contact information is reused, so if you do queue up steps to change afterwards, you should make sure to copy the individual pieces of that structure out to separate objects. BeginContact and EndContact both only take the contact object as a parameter. You can use this callback to determine when objects have received enough damage to be destroyed or explode. The contact callback has

been added to the box that is originally blue, and the addanimpulse with a click behavior from Example 3 is still in there so you can see how collisions trigger. Joints allow you to constrain bodies in one or more dimensions to another body or another point You can constrain this joint to limit the angles between the two objects as well as turn on a motor to drive rotation. You can limit constrain the joint to limit the minimum and maximum distance as well as turn on a motor to drive translation. As one body goes up, the other goes down. You can control the ratio of rotation or translation. Primarily used for “soft” drag and drop of objects. Finally world.

CreateJoint is called to create an actual joint from the definition. To add in touch support, you could add in touchstart, touchmove and touchend events. Adding support for multiple touches would allow you to drag multiple objects around the page at once see Mobile Game Primer for how to do multitouch Box2D by default creates an empty body with no fixtures in it that it calls the Ground Body. In this case, the MouseJoint attaches the first body to the Ground body, but that body isn’t actually used for anything. You don’t need to use the ground body, you could use any other static body, but the ground body is nice because it’s always there and has no fixtures so you don’t need to worry about collisions. If we find one, the object is saved for future use. Then, the first time the mouse is moved, it creates a new mouse joint. On that and every subsequent mousemove event, we update the target of the joint to be the new location of the mouse, which will move the object towards the mouse if possible the object will not move through static or dynamic objects, and will exert a force only upto the maxForce parameter. Finally, when the mouse is released, the joint is destroyed. The joint doesn’t, however, limit rotation in any way. You can hook up any types of bodies, but the joint will only move dynamic bodies. These allow you to turn distance joints into soft springs by lowering the damping ratio and upping the response frequency. See the second distance example to watch this in action. If you connect two dynamic bodies together, it will act as if there is a hinge between those bodies at that point. If you connect to a static body, the dynamic body will rotate around that point. See the Flash Documentation for the additional options on the b2RevoluteJointDef class. Creating one is similar to creating a distance joint you specify two bodies and then two anchor points, as shown below See the Box2DWeb Documentation for more details on the options that enable motor movement.