



# Bark of the byte

## (<http://barkofthebyte.azurewebsites.net/>)

Running with bytes and other wild creatures



(syndication.axd)

### Three.js projecting mouse clicks to a 3D scene - how to do it and how it works (</post/2014/05/05/three-js-projecting-mouse-clicks-to-a-3d-scene-how-to-do-it-and-how-it-works>)

📅 5. May 2014   👤 acarlon (<http://barkofthebyte.azurewebsites.net/author/acarlon>)   📁 Web development  
(/category/Web-development), HTML5 (/category/HTML5)

💬 (0) (</post/2014/05/05/three-js-projecting-mouse-clicks-to-a-3d-scene-how-to-do-it-and-how-it-works#comment>)

👍 Like 8   Share



## Introduction

In Three.js, it is quite common to detect when a user clicks an object in 3D space. I have come across a few posts and answers describing how to do this, but I found that I needed to go a bit deeper under the surface. Without some knowledge of what is happening it can be tricky to work outside of the scope of the sample code (such as having a canvas that does not fill the screen or having additional effects). In this blog post, we will still only scratch the surface, but we should hopefully cover enough to deviate from the samples with some confidence. Note that in this blog post we will use PerspectiveCamera, not OrthographicCamera.

## How to do it

Lets start with a simple scenario where an object changes colour when clicked. The code for the mouse down event would be something like:

```
function onDocumentMouseDown( event ) {  
    var mouse3D = new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 - 1  
                                      -( event.clientY / window.innerHeight ) * 2 +  
                                      0.5 );  
  
    projector.unprojectVector( mouse3D, camera );  
    mouse3D.sub( camera.position );  
    mouse3D.normalize();  
    var raycaster = new THREE.Raycaster( camera.position, mouse3D );  
    var intersects = raycaster.intersectObjects( objects );  
    // Change color if hit block  
    if ( intersects.length > 0 ) {  
        intersects[ 0 ].object.material.color.setHex( Math.random() * 0xffffff );  
    }  
}
```

With the more recent three.js releases (around r55 and later), you can use `pickingRay` which simplifies things even further so that the internals of the function are:

```
var mouse3D = new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 - 1  
                                -( event.clientY / window.innerHeight ) * 2 +  
                                0.5 );  
  
var raycaster = projector.pickingRay( mouse3D.clone(), camera );  
var intersects = raycaster.intersectObjects( objects );  
// Change color if hit block  
if ( intersects.length > 0 ) {  
    intersects[ 0 ].object.material.color.setHex( Math.random() * 0xffffff );  
}
```

For this blog post we will stick with the old approach as it gives more insight into what is happening under the hood. You can see this working here (</demos/projspikesfssimple.htm>), simply click on the cube to change its colour.

## How it works

Now, let's look at what is happening step by step:

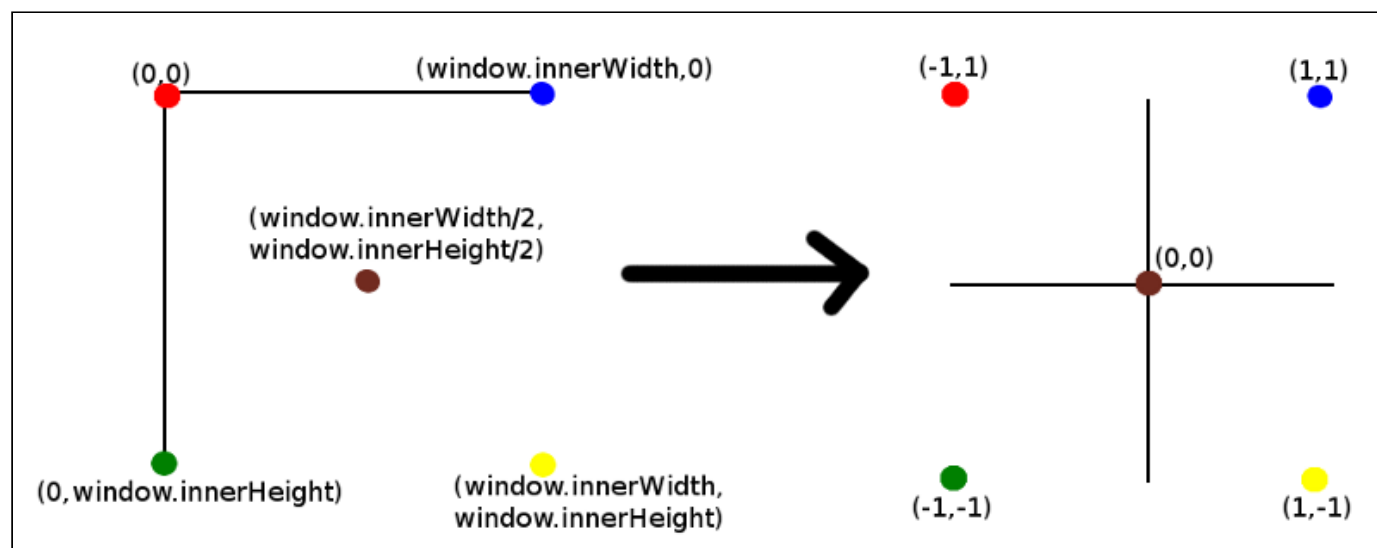
```
var mouse3D = new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 - 1
                                  -( event.clientY / window.innerHeight ) * 2 +
                                  0.5 );
```

I have seen a number of blogs that describe this step as 'some kind of magic', however this is actually a step that we can quite easily understand. `event.clientX` is the x coordinate of the click position. Dividing by `window.innerWidth` gives the position of the click in proportion of the full window width. The tables below show the corresponding result for the location of the mouse click:

<code>event.clientX</code>	<code>event.clientX / window.innerWidth*2-1</code>
0	-1
<code>window.innerWidth/2</code>	0
<code>window.innerWidth</code>	1

<code>event.clientY</code>	<code>-event.clientY / window.innerHeight * 2 + 1</code>
0	1
<code>window.innerHeight/2</code>	0
<code>window.innerHeight</code>	-1

Basically, this is translating from screen coordinates that start at (0,0) at the top left through to (`window.innerWidth`,`window.innerHeight`) at the bottom right, to the cartesian coordinates with center (0,0) and ranging from (-1,-1) to (1,1) as shown below:



Note that  $z$  has a value of 0.5. I won't go into too much detail about the  $z$  value at this point except to say that this is the depth of the point away from the camera that we are projecting into 3D space along the  $z$  axis. More on this later.

OK, so we now understand the position of where the mouse was clicked in terms of the graph on the right (mouse3D contains this position). Next:

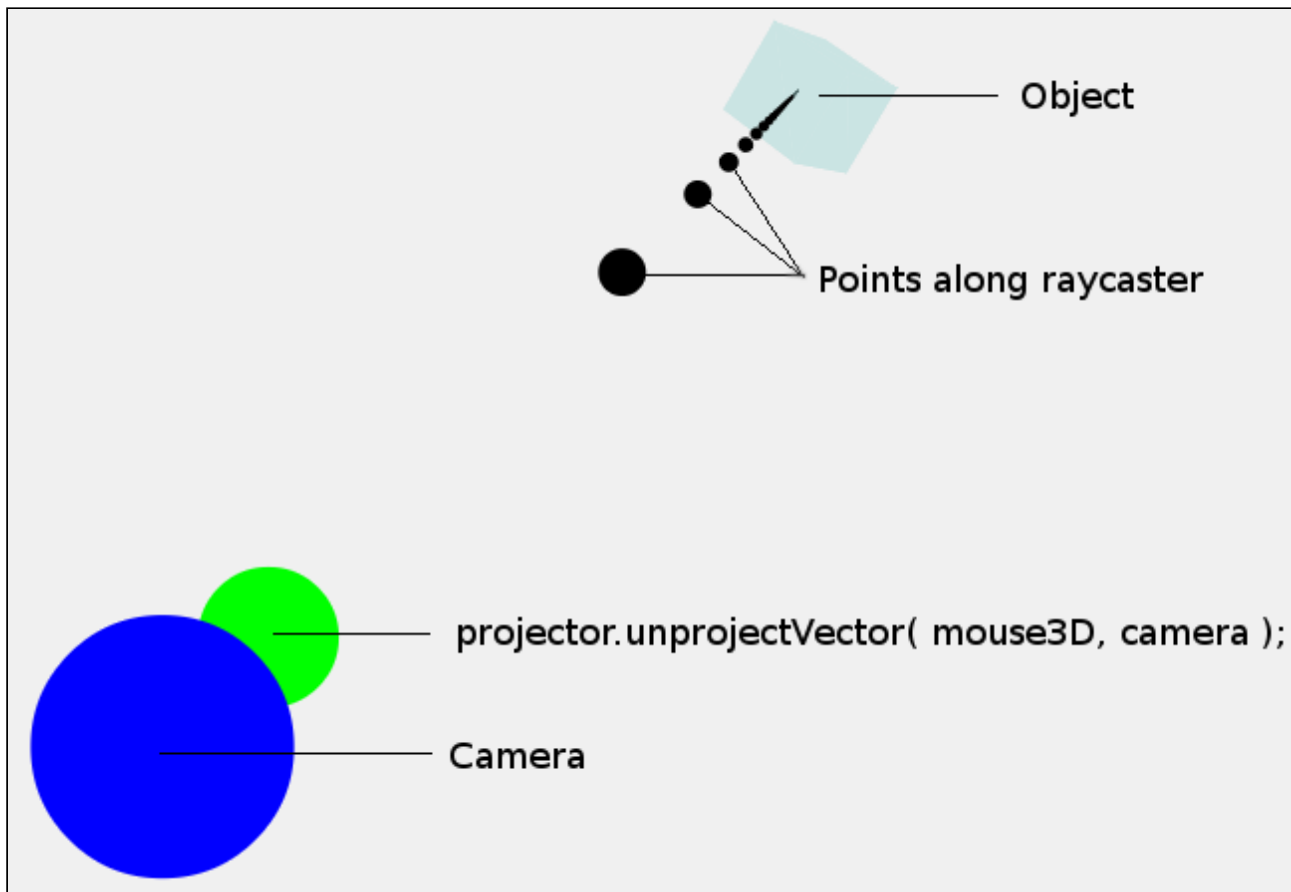
```
projector.unprojectVector( mouse3D, camera );
```

This is actually the step where magic happens. If you look at the three.js code you will see that the camera projection onto the 3D world is applied to the vector. The way that I understand it is as follows:

- Forgetting about projecting from 2D to 3D, let's first think about the inverse - how the 3D world is projected onto your 2D screen. In order to get from 3D world coordinates to a projection on the screen, the 3D world needs to be projected onto the 2D surface of the camera (which is what you see on your screen).
- So, to get from 2D coordinates on the screen to 3D space the opposite needs to be done.
- Start with the camera screen as a 2D coordinate system with 0,0 at the center. Positioning mouse3D in this space will give you a position on this surface. Now, (0,0) might not be at the center so there is some additional translation to convert to the actual position. This means that you can move the camera around and have it look at different points in 3D space and the projection still works. Which is great.

The workings behind the above bullet points are beyond the scope of this blog post, but hopefully they give some insight into what is happening behind the scenes. Note that mouse3D will now contain this unprojected value. This is the position of a point in 3D space along the ray/trajectory that we are interested in. The exact point depends on the  $z$  value (we will see this later).

At this point, it may be useful to have a look at the following image:



The point that we have just calculated (mouse3D) is shown by the green dot. Note that the size of the dots are purely illustrative, they have no bearing on the size of the camera or mouse3D point. We are more interested in the coordinates at the center of the dots. Also note that **the points shown along the ray are just arbitrary points**, the ray is a direction from the camera, not a set of points.

Now, we don't just want a single point in 3D space, but instead we want a ray/trajectory (shown by the black dots) so that we can determine whether an object is positioned along this ray/trajectory. Fortunately, because we have a point and we know that the trajectory must pass from the camera to this point, we can determine the direction of the ray. Therefore, the next step is to subtract the camera position from the mouse3D position, this will give a directional vector rather than just a single point:

```
mouse3D.sub( camera.position );  
mouse3D.normalize();
```

We now have a direction from the camera to this point in 3D space (mouse3D now contains this direction). This is then turned into a unit vector by normalizing it. You can read about unit vectors online (e.g. Wikipedia ([http://en.wikipedia.org/wiki/Unit\\_vector](http://en.wikipedia.org/wiki/Unit_vector))), for our purpose it is only really important to know that the unit vector

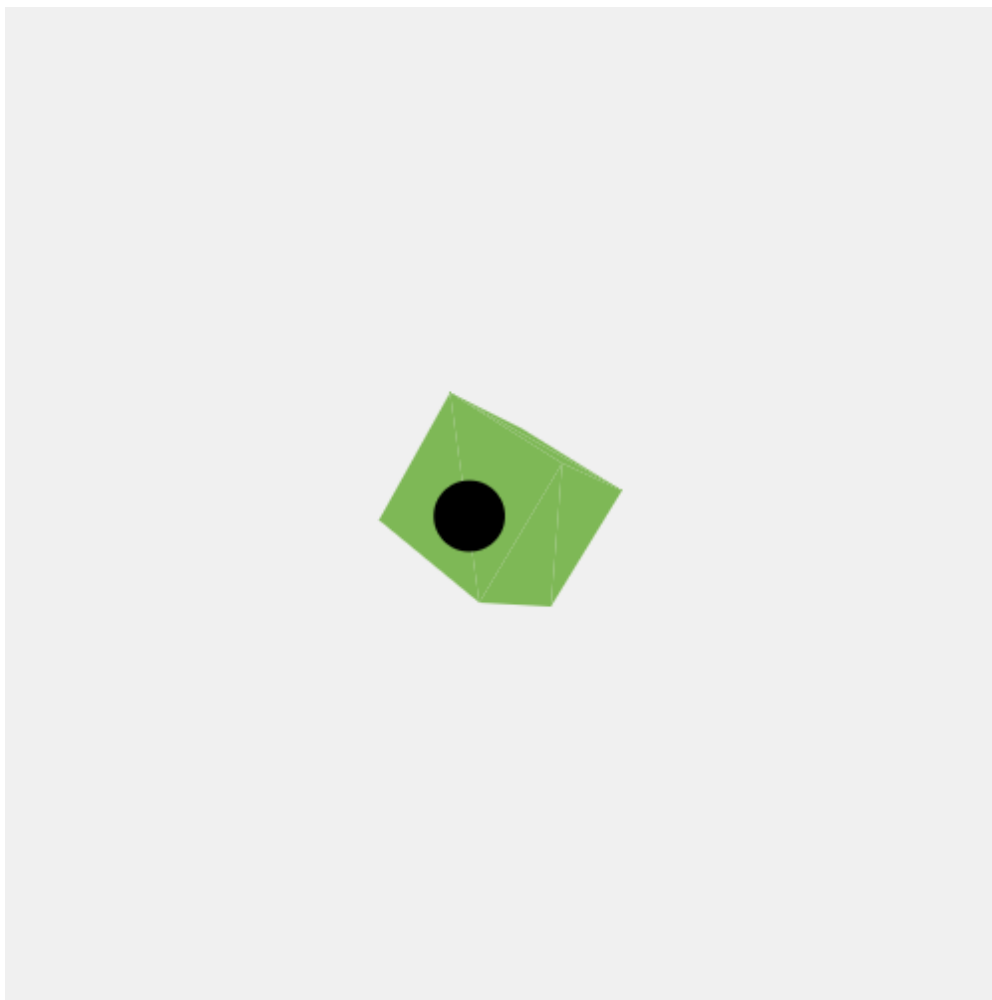
has the same direction as the original). The next step is to create a ray (Raycaster) starting from the camera position and using the direction (mouse3D) to cast the ray.

```
var raycaster = new THREE.Raycaster( camera.position, mouse3D );
```

The rest of the code determines whether the objects in 3D space are intersected by the ray or not. This can be done by determining for each object whether the ray is within the x, y and z bounds of the object. Happily it is all taken care of us behind the scenes using intersectsObjects.

## The Demo

OK, so let's look at a demo that shows these rays being cast in 3D space. When you click anywhere, the camera rotates around the object to show you how the ray is cast. Note that when the camera returns to its original position, you only see a single dot. This is because all the other dots are along the line of the projection and therefore blocked from view by the front dot. This is similar to when you look down the line of an arrow pointing directly away from you - all that you see is the base. Of course, the same applies when looking down the line of an arrow that is travelling directly towards you (you only see the head), which is generally a bad situation to be in. OK, the demo:



The lines that you see are drawn by my function `drawRayLine(rayCaster)` that draws dots along the ray.

## The z coordinate

As promised, let's take another look at that z coordinate. Refer to this demo (</demos/projspikesfs.htm>) as you read through this section and experiment with different values for z.

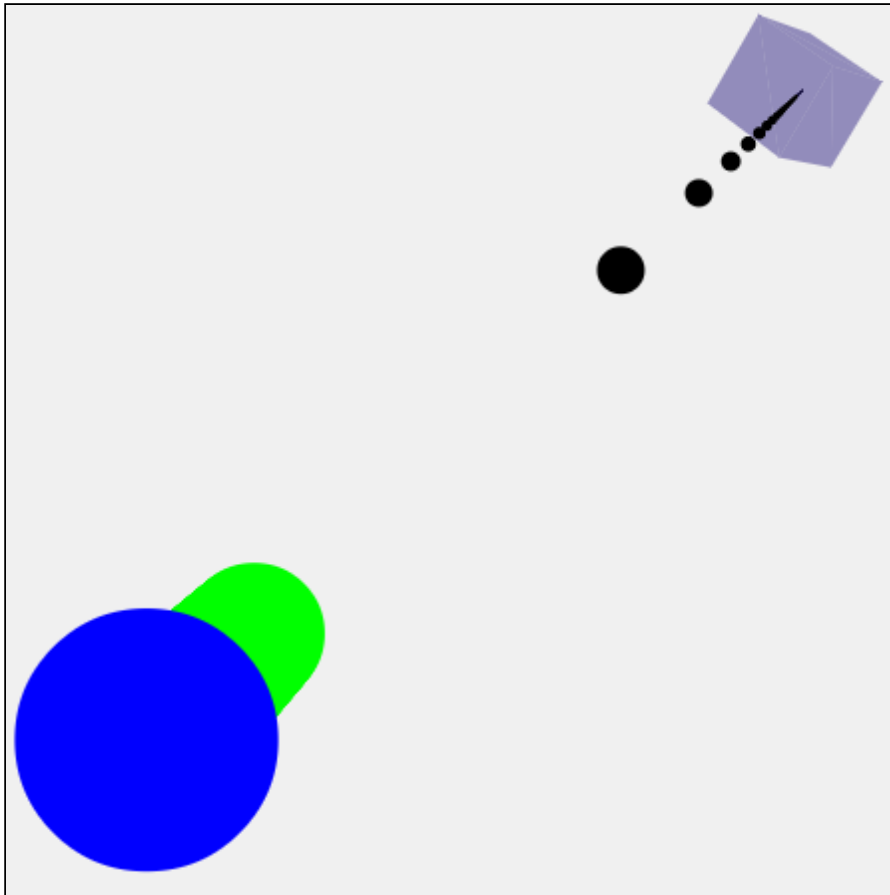
OK, let's take another look at this function:

```
var mouse3D = new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 - 1  
                                  -( event.clientY / window.innerHeight ) * 2 +  
                                  0.5 );
```

We chose 0.5 as the value. I mentioned earlier that the z coordinate dictates the depth of the projection into 3D. So, let's have a look at different values of z to see what effect it has. To do this, I have placed a blue dot where

the camera is, and a line of green dots from the camera to the unprojected position. Then, after the intersections have been calculated, I move the camera back and to the side to show the ray. Best seen with a few examples.

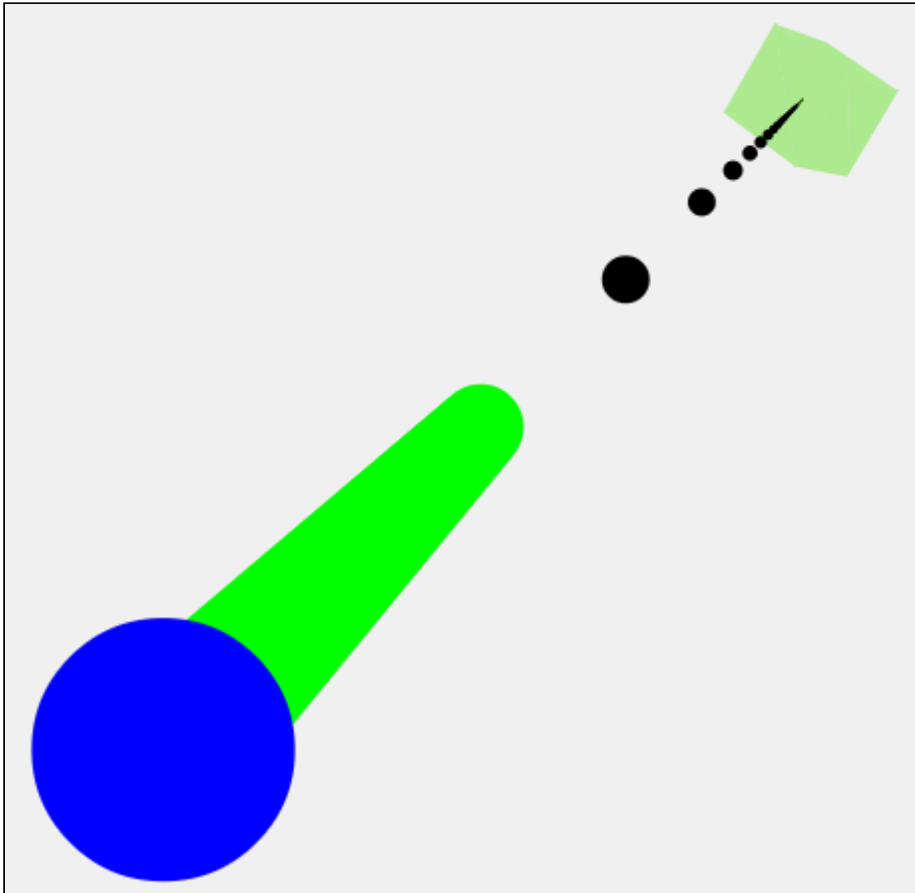
First, a z value of 0.5:



Note the green line of dots from the camera (blue dot) to the unprojected value (the coordinate in 3D space). This is like the barrel of a gun, pointing in the direction that the ray should be cast. The green line essentially represents the direction that is calculated before being normalised.

OK, let's try a value of 0.9:





As you can see, the green line has now extended further into 3D space. 0.99 extends even further:

I do not know if there is any importance as to how big the value of  $z$  is. It seems that a bigger value would be more precise (like a longer gun barrel), but since we are calculating the direction, even a short distance should be pretty accurate. The examples that I have seen use 0.5, so that is what I will stick with unless told otherwise.

## Projection when the canvas is not full screen

Now that we know a bit more about what is going on, we can figure out what the values should be when the canvas does not fill the window and is positioned on the page. Say, for example, that: -

- the div containing the three.js canvas is `offsetX` from the left and `offsetY` from the top of the screen.
- the canvas has a width equal to `viewWidth` and height equal to `viewHeight`.

For full-window we had:

```
var mouse3D = new THREE.Vector3( ( event.clientX / window.innerWidth ) * 2 - 1
                                   -( event.clientY / window.innerHeight ) * 2 +
                                   0.5 );
```

This will now become:

```
var mouse3D = new THREE.Vector3( ( event.clientX - offsetX ) / viewWidth * 2 -  
                                  -( event.clientY - offsetY ) / viewHeight * 2  
                                  0.5 );
```

Basically, what we are doing is calculating the position of the mouse click relative to the canvas (for x: `event.clientX - offsetX`). Then we determine proportionally where the click occurred (for x: `/ viewWidth`) similar to when the canvas filled the window.

## Done

That is it. We had a look at what is going on when mouse clicks magically select objects in 3D space using Three.js. I think the next steps for digging deeper would be to have a look at what `projector.unprojectVector()` does in more detail and especially have a look at how the camera projection matrix works. But, that is for another time.

Tags : three.js (<http://barkofthebyte.azurewebsites.net/?tag=/three.js>), html5 (<http://barkofthebyte.azurewebsites.net/?tag=/html5>), projection (<http://barkofthebyte.azurewebsites.net/?tag=/projection>)

## Related posts

Three.js projecting mouse clicks to a 3D scene - how to do it and how it works (</post/2014/05/05/three-js-projecting-mouse-clicks-to-a-3d-scene-how-to-do-it-and-how-it-works>)

Projecting mouse clicks into a three.js 3D world

Can the HTML5 Canvas save us from cross-browser woe? (</post/2014/03/13/Can-the-HTML5-Canvas-save-us-from-cross-browser-woe>)

Can the HTML5 Canvas be the solution to cross-browser woe?

Comment Preview

Save comment

--

Setting up BlogEngine.NET 2.9  
(/post/2014/03/03/Setting-up-BlogEngineNET-29) (1)  
Oliver Richmond wrote: I am trying to set up BlogEngine.NET 2.9. And I am... [More]  
(/post/2014/03/03/Setting-up-

The Solar System Clock - what is in a day? (/post/2016/01/17/the-solar-system-clock-what-is-in-a-day)


Comments: 0

---

The Solar System Clock  
(/post/2016/01/16/The-Solar-System-Clock)

BlogEngineNET-29#id\_86012ede-94d3-4c7d-8f10-c7a1c2067a21)

---

Comment RSS  (/syndication.axd?comments=true)

Comments: 0

---

Three.js projecting mouse clicks to a 3D scene - how to do it and how it works (/post/2014/05/05/three-js-projecting-mouse-clicks-to-a-3d-scene-how-to-do-it-and-how-it-works)

Comments: 0

---

Part 3: Locked

(/post/2014/04/08/part-3-locked)

Comments: 0

---

Part 2: Clobbered

(/post/2014/04/05/Part-2-Clobbered)

Comments: 0

---

Part 0.2 Concurrent versus parallel mind games (/post/2014/04/03/a-few-mind-games)

Comments: 0

---

Part 0.1 Defining concurrent vs parallel

(/post/2014/04/03/Concurrent-programming-and-friends-Part-01-Defining-concurrent-vs-parallel)

Comments: 0

---

Part 0: Concurrent versus parallel (/post/2014/04/02/Concurrent-programming-and-friends-Part-0-Concurrent-versus-parallel)

Comments: 0

---

Part 1: The opposite of concurrent (/post/2014/03/27/Concurrent-programming-and-friends-Part-1-The-opposite-of-concurrent)

Comments: 0

---

Can the HTML5 Canvas save us  
from cross-browser woe?  
(/post/2014/03/13/Can-the-HTML5-  
Canvas-save-us-from-cross-browser-  
woe)  
Comments: 0

Newsletter

Get notified when a new post is  
published.

Enter your e-mail

Notify me




Enter search term or APML Search

Calendar

<<		January 2018					>>	
Mon	Tue	Wed	Thu	Fri	Sat	Sun		
25	26	27	28	29	30	31		
1	2	3	4	5	6	7		
8	9	10	11	12	13	14		
15	16	17	18	19	20	21		
22	23	24	25	26	27	28		
29	30	31	1	2	3	4		

View posts in large calendar  
(http://barkofthebyte.azurewebsites.net/calendar/default.aspx)

Category list

-  (/category/feed/Agile) Agile (1)  
(/category/Agile)
-  (/category/feed/How-to) How to (1)  
(/category/How-to)
-  (/category/feed/How-to-blogs) How to blogs (1)

Tag cloud

agile software development burn down (/?  
tag=/agile-software-development-burn-down)  
blogging blogengine.net favicons social  
media (/?tag=/blogging-blogengine.net-  
favicons-social-media)

Page List

(/category/How-to-blogs)

---



(/category/feed/Programming) Programming  
(5) (/category/Programming)

---

(/category/feed/Concurrent-parallel-multithreaded) Concurrent, parallel, multithreaded (5)  
(/category/Concurrent-parallel-multithreaded)

---

(/category/feed/Space) Space (2)  
(/category/Space)

---

(/category/feed/Web-development) Web development (2)  
(/category/Web-development)

---

(/category/feed/HTML5) HTML5  
(2) (/category/HTML5)

---

Canvas (/?tag=/Canvas)

concurrent (/?  
tag=/concurrent)

html (/?tag=/html)    html5 (/?tag=/html5)

locks (/?tag=/locks)

mindgames (/?tag=/mindgames)

multithreaded (/?tag=/multithreaded)

mutexes (/?tag=/mutexes)

parallel (/?tag=/parallel)

programming (/?tag=/programming)

projection (/?tag=/projection)

sequential (/?tag=/sequential)

synchronization (/?tag=/synchronization)

threads (/?tag=/threads)

three.js (/?tag=/threejs)

universe (/?tag=/universe)

COPYRIGHT © 2018 BARK OF THE BYTE (HTTP://BARKOFTHEBYTE.AZUREWEBSITES.NET/) - POWERED BY BLOGENGINE.NET

(HTTP://DOTNETBLOGENGINE.NET) 2.9.1.0 - DESIGN BY FS (HTTP://SEYFOLAH.NET/)