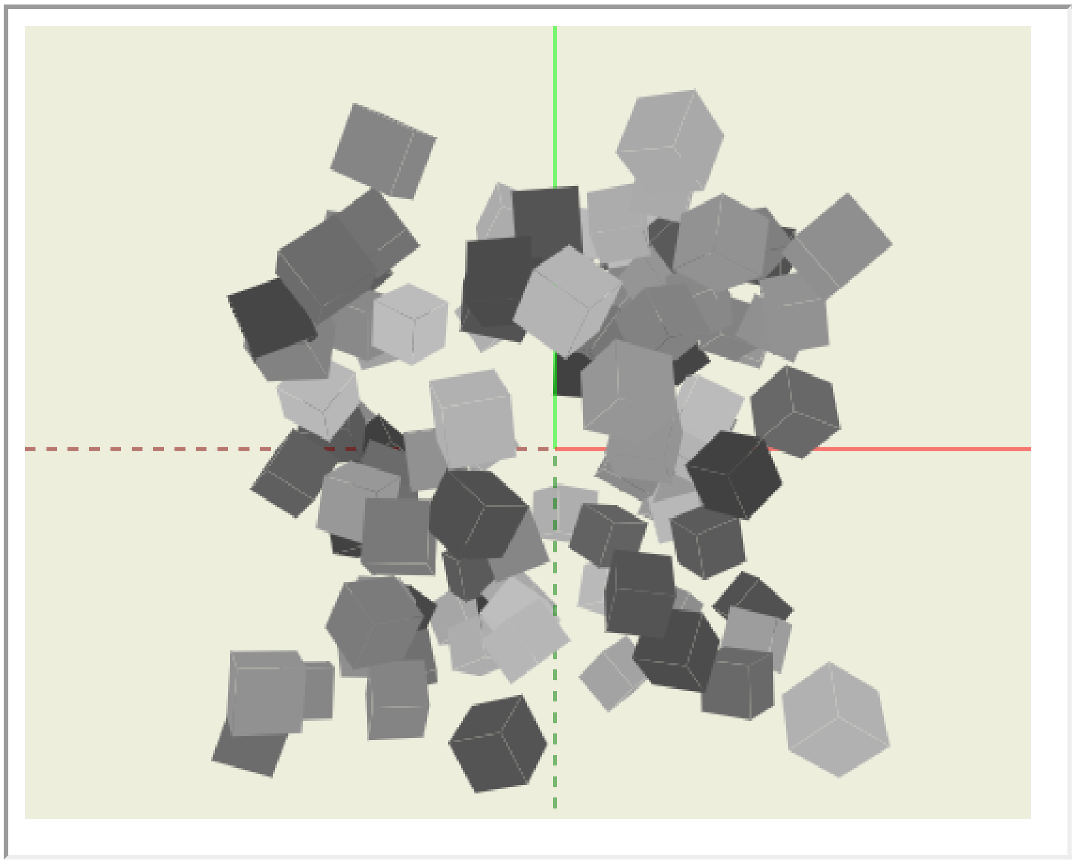
soledad penadés

repeat 4[fd 100 rt 90]

three.js tutorials →

Object picking

In this tutorial we'll learn what is **object picking**, how **projection** and **unprojection** work, and how to use those with three.js to build this scene where gray cubes turn reddish when the mouse hovers them– the closer to the camera they are, the redder they turn!



Get the source code for this tutorial if you want to dive straight into the code!

Here are a couple more examples:

- interactive cubes, where you can click on cubes and a black spot will appear in the exact point where you clicked, and…
- interactive voxel painter, which allows you to add voxels and remove them too

Did you play with them? Notice a common pattern? We are detecting objects in 3D space using our 2D coordinate world (the screen). That's what object picking is about!

How does it work?

Before we write a single line of code, it'll be really helpful to understand how computer 3D graphics work, even if in a very rough manner. How do we go from an abstract, perfect 3D scene to a 2D image in our screens?

When you use a *camera* to render a scene, a whole bunch of math machinery starts munching and processing your 3D scene, in order to produce a 2D representation of the fragment of your scene that can be seen from the camera. There are many steps involved but the one that interests us is the **projection**, which is what turns abstract 3D stuff into somewhat convincing 2D entities in your screen.

And that is great, but… *how do we go backwards?*

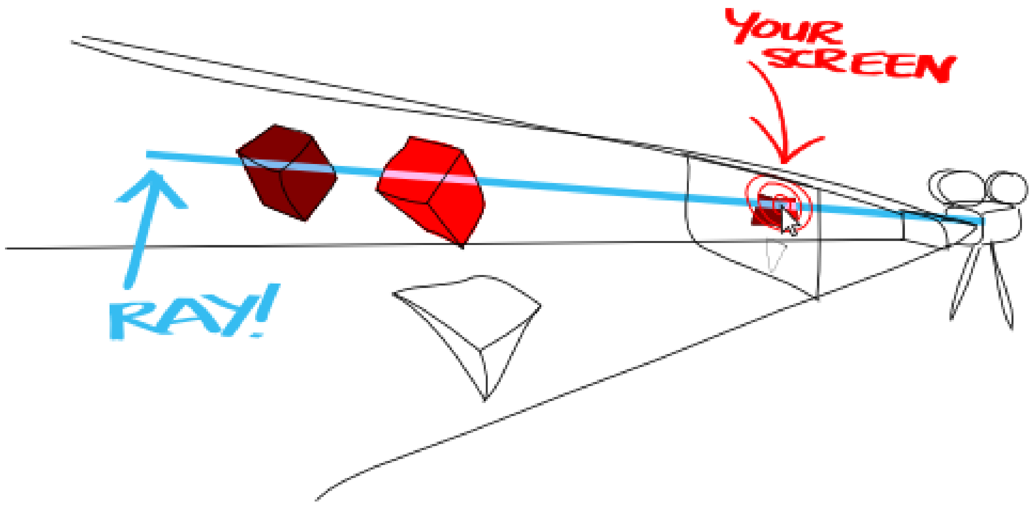
Why would we want to go backwards?, you might be asking. Well, if you want to find out which object is underneath your mouse pointer, you need to bring those 2D coordinates back into the 3D world before you try to find out if there's something there. That is **unprojecting**!

See? Now everything fits:

- **Going from 3D to 2D?** PROJECTION!
- **Going from 2D to 3D?** UNPROJECTION!

There's still a missing piece: how do we find the intersections once we're back in 3D land? It always sounds funny to me, but here's the answer: **we cast rays**! Yes, I'm being serious! We cast a ray from the position of our mouse in 3D space, following the current direction of the camera, and see if we hit any object(s) on the way. If we do, then we've found what we were looking for. If we don't, then there's nothing underneath the mouse.

I know it can be confusing, so please look at this drawing:



There's an abstract 3D scene on the left, with two cubes and a pyramid. Then there's a representation of our screen –i.e. how we see the 3D scene–. You can also see the mouse cursor, over the cubes. On the right, the camera from where we're looking, and in blue, the ray we're using to pick objects.

And now that you know how object picking works in theory, let's see how we do that with three.js!

Object picking in action

Thankfully, this is very easy to implement with three.js. We'll start with the basics: setting up the renderer, a scene and a camera.

```
var container = document.getElementById( 'container' ),
    containerWidth, containerHeight,
    renderer,
    scene,
    camera;

containerWidth = container.clientWidth;
containerHeight = container.clientHeight;

renderer = new THREE.CanvasRenderer();
renderer.setSize( containerWidth, containerHeight );
container.appendChild( renderer.domElement );

renderer.setClearColorHex( 0xeeeedd, 1.0 );

scene = new THREE.Scene();

camera = new THREE.PerspectiveCamera( 45, containerWidth /
containerHeight, 1, 10000 );
camera.position.set( 0, 0, range * 2 );
camera.lookAt( new THREE.Vector3( 0, 0, 0 ) );
```

Nothing specially surprising!

Now let's add some objects. We'll create the gray cubes, and place them randomly in the scene, but they will be children of a *cubes* object. NOTE: if you want to learn more about adding objects to other objects, read this tutorial :-)

```
geom = new THREE.CubeGeometry( 5, 5, 5 );

cubes = new THREE.Object3D();
scene.add( cubes );

for(var i = 0; i < 100; i++ ) {
        var grayness = Math.random() * 0.5 + 0.25,
                mat = new THREE.MeshBasicMaterial(),
                cube = new THREE.Mesh( geom, mat );
        mat.color.setRGB( grayness, grayness, grayness );
        cube.position.set( range * (0.5 - Math.random()), range * (0.5
- Math.random()), range * (0.5 - Math.random()) );
        cube.rotation.set( Math.random(), Math.random(), Math.random()
).multiplyScalar( 2 * Math.PI );
        cube.grayness = grayness; // *** NOTE THIS
        cubes.add( cube );
}
```

It's the same geometry for all the cubes, but the materials have to be different in order to have a different colour per cube. Notice also the *grayness* property that we add to each cube. We'll use it later!

With that done, we'll prepare two key objects for picking: the *projector* and the *mouse* vector:

```
projector = new THREE.Projector();
mouseVector = new THREE.Vector3();
```

Since we want to pick objects when we the mouse is moved, we'll need to add a listener:

```
window.addEventListener( 'mousemove', onMouseMove, false );
```

And then all the interesting stuff will happen in *onMouseMove*!

Pay close attention as we walk through its code. You need to be *very* careful with the order and the details, or else you won't get ray picking working *at all* (and I speak from experience!)

```
mouseVector.x = 2 * (e.clientX / containerWidth) - 1;
mouseVector.y = 1 - 2 * ( e.clientY / containerHeight );
```

You **must** get these two first lines right. They convert the mouse coordinates, which go from *0* to *containerWidth*, and from *0* to *containerHeight*, to *(-1, 1)* in both axes.

Did you notice that the calculations for the y coordinate are negated? That's because in the classic DOM coordinate system the Y axis grows *from top to bottom* (i.e. top is 0), whereas in 3D it grows *from bottom to top* (i.e. bottom is 0). I can't tell you how many times I've forgotten about this and thus failed to get picking to work!

So ensure you get this right… because this is actually the most complicated part of the function!

Next up is casting a ray and using it to find objects in its path:

```
var raycaster = projector.pickingRay( mouseVector.clone(), camera );
```

Super easy!

In previous versions of THREE (probably in r54 or less) you had to perform all the calculations that the *pickingRay* function now does for you, and there are still many tutorials out there doing the calculations manually. It's OK if you know what you're doing, or want to do something specific that *pickingRay* doesn't do, but I find it's generally safer and way less error prone to use *pickingRay*.

Also note that we're cloning *mouseVector*, instead of just sending it to *pickingRay*. That's because the function modifies the values of *mouseVector*. You can look at Projector.js to see what I mean, but it's not really something to worry about as long as you make sure to clone the vector before calling *pickingRay*.

And to find intersections of the ray with objects:

```
var intersects = raycaster.intersectObjects( cubes.children );
```

This will return an Array with all the ray intersections with the children of *cubes*, ordered by distance (the nearest object goes first). Each intersection is an object with these properties:

- **distance:** how far from the camera the intersection happened
- **point:** the exact point in the object where the ray intersects it
- **face:** the intersected face.
- **object:** which object was intersected

We'll use these data to paint the intersected cubes redder as they get closer to the camera. First we set them all back to gray again, using the *grayness* property that we set up earlier:

```
cubes.children.forEach(function( cube ) {
    cube.material.color.setRGB( cube.grayness, cube.grayness,
cube.grayness );
});
```

Then we iterate through the *intersects* array. For each intersection, we change the object's colour to a hue of red:

```
for( var i = 0; i < intersects.length; i++ ) {
    var intersection = intersects[ i ],
        obj = intersection.object;

    obj.material.color.setRGB( 1.0 - i / intersects.length, 0, 0 );
}
```

And that's all for the *onMouseMove* function!

Resizing the window

It is always more user friendly to take into account window resizing. For example, the user might open the JavaScript console, and thus the size of the viewport will change.

OK, I was joking! A typical user probably won't open the console (unless by accident), but they might truly want to resize the window.

So let's listen and react to *resize* events:

```
window.addEventListener( 'resize', onWindowResize, false );

function onWindowResize( e ) {
        containerWidth = container.clientWidth;
        containerHeight = container.clientHeight;
        renderer.setSize( containerWidth, containerHeight );
        camera.aspect = containerWidth / containerHeight;
        camera.updateProjectionMatrix();
}
```

Each time this function is called, we'll update the *containerWidth* and *containerHeight*. This is so that picking keeps working properly. Also, the renderer's size needs to be updated, and same goes for the camera aspect –else we will get a distorted output. Try commenting out that line and see what happens!

Something that might look a bit unusual is that we also manually update the camera's projection matrix, but that's because we're using a CanvasRenderer. If we use a WebGLRenderer, we don't need to manually update that matrix.

Sources and further reading

Here's the source code for this tutorial.

Also, you should really study the source for the two examples I linked at the beginning: canvas interactive cubes and interactive voxel painter.

There are more examples with user interaction in three.js's **examples** folder. For example, the webgl interactive draggable cubes also demonstrates dragging an object in 3D space, and the webgl interactive cubes is similar to the similar canvas based example, but uses WebGL instead.