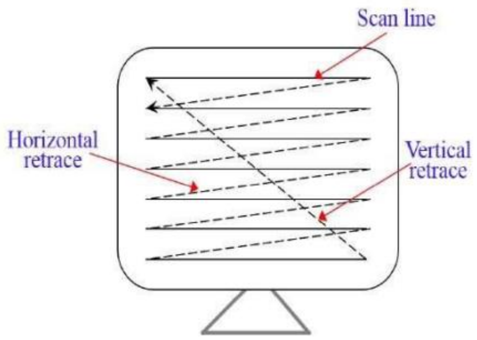
Brayan Sebastián Becerra Beltrán (1202052)

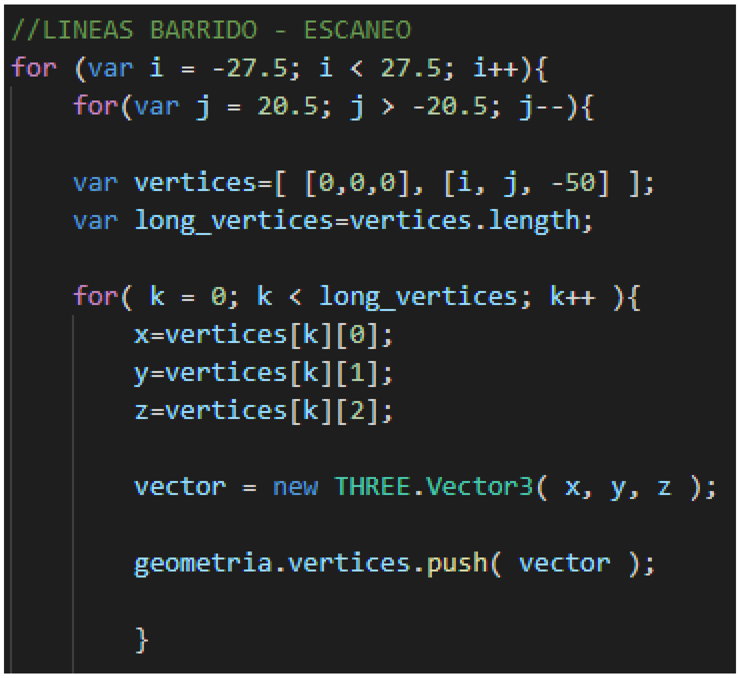
COMPUTACIÓN GRÁFICA:

Raycaster y líneas de barrido

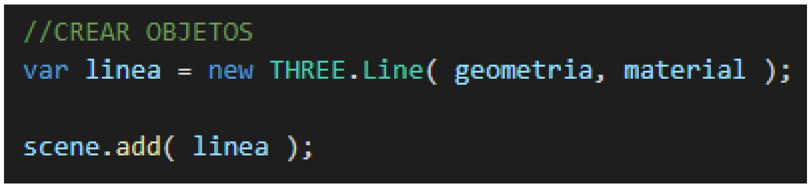
ESCANEO DE LA ESCENA (LINEAS DE BARRIDO)

Para hacer el método de barrido el cual consiste en un escaneo de toda la escena, para identificar la intersección entre un vector y una superficie.





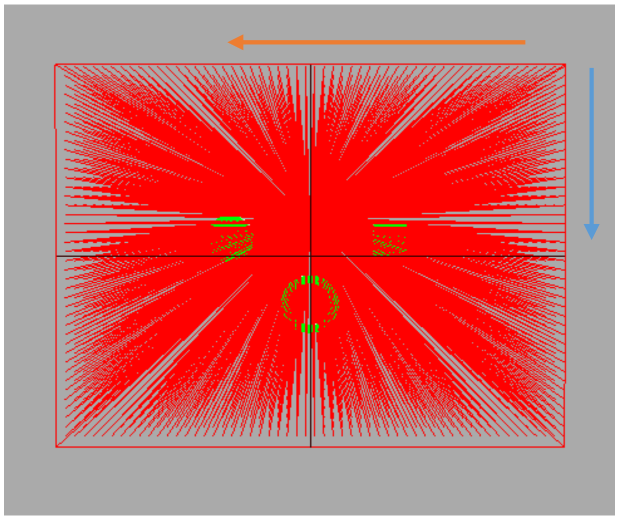
```
//LINEAS BARRIDO - ESCANEO
for (var i = -27.5; i < 27.5; i++){
    for(var j = 20.5; j > -20.5; j--){

    var vertices=[ [0,0,0], [i, j, -50] ];
    var long_vertices=vertices.length;

    for( k = 0; k < long_vertices; k++ ){
        x=vertices[k][0];
        y=vertices[k][1];
        z=vertices[k][2];

        vector = new THREE.Vector3( x, y, z );

        geometria.vertices.push( vector );

        }
```



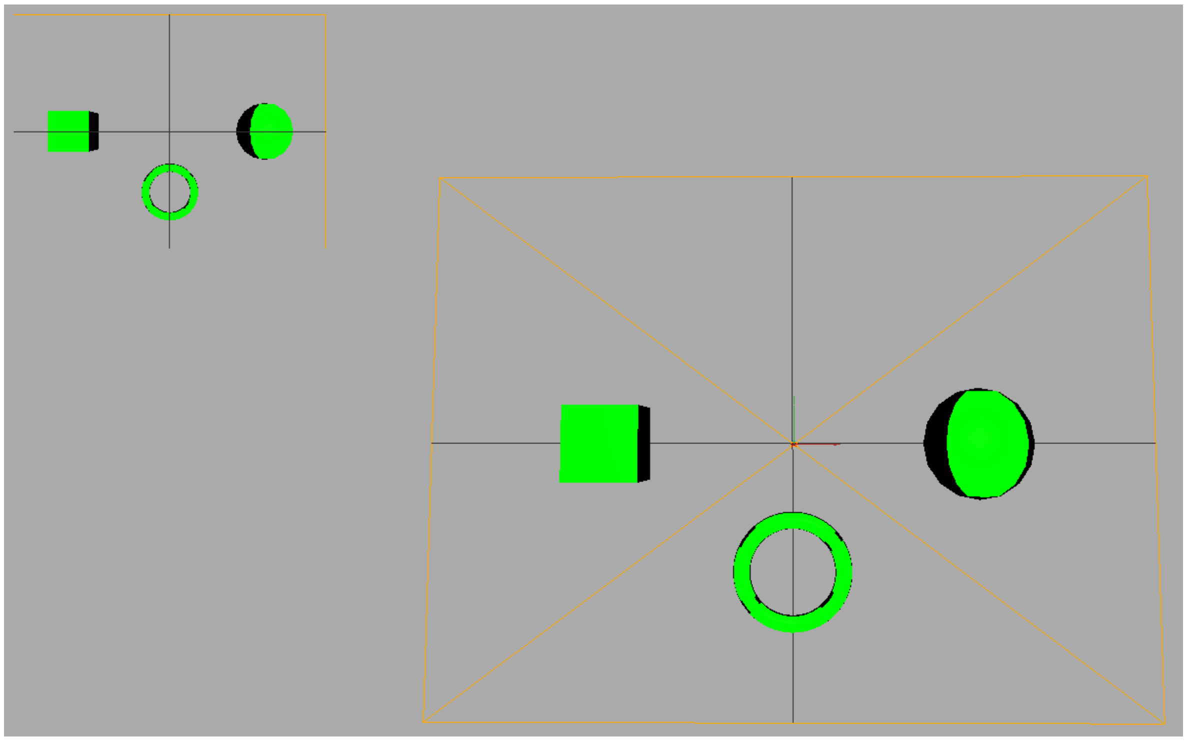
```
//CREAR OBJETOS
var linea = new THREE.Line( geometria, material );

scene.add( linea );
```

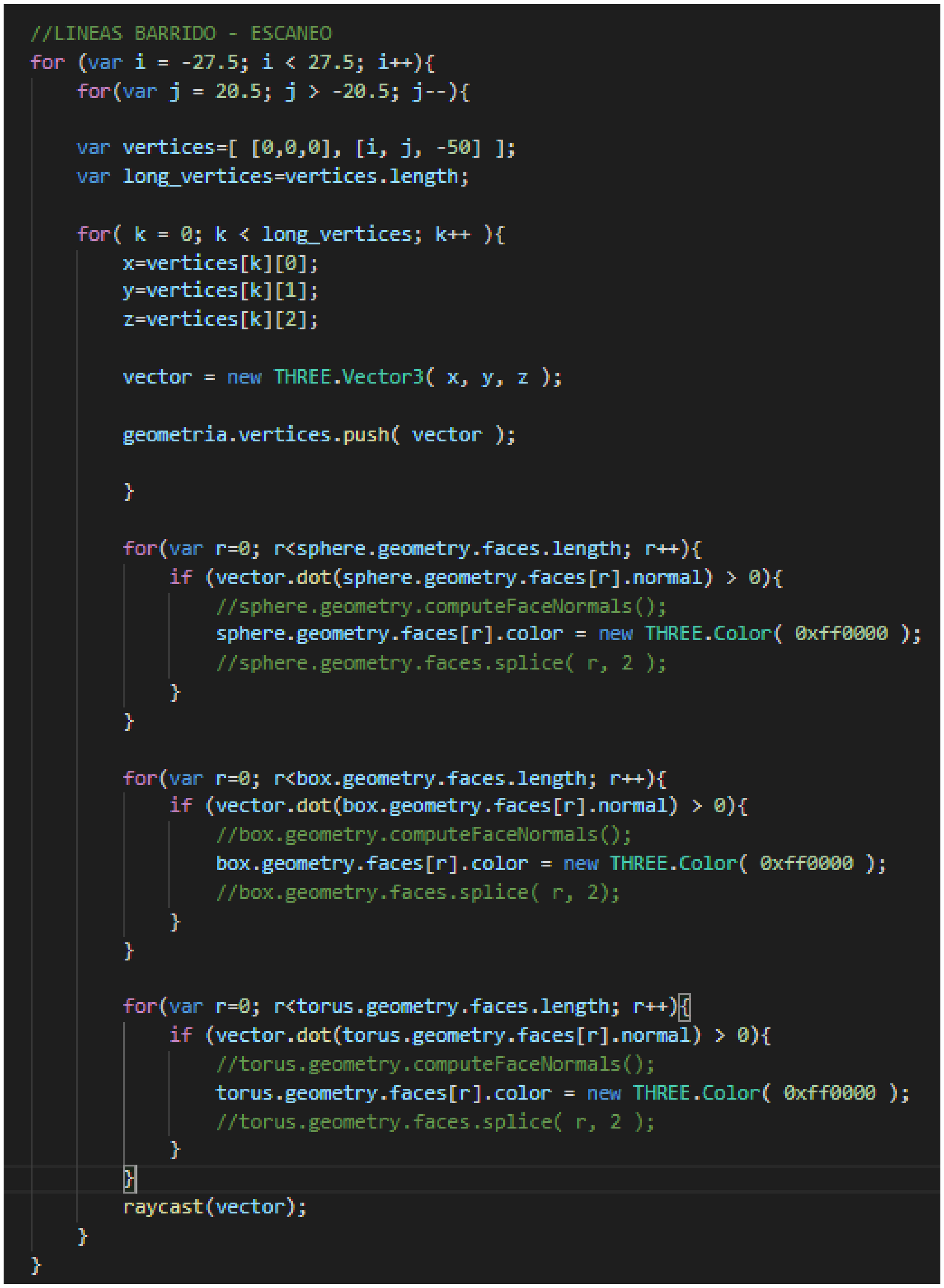
El escaneo se realiza de izquierda a derecha con una disminución en y cada vez que x llega a su tope, los valores están relacionados con las dimensiones de “camera2”, el valor de la profundidad se va a mantener constante.

X y **Y:**



Para poder visualizar las figuras, se tiene en cuenta el número de caras y la dirección del vector normal que se genera desde la cámara.





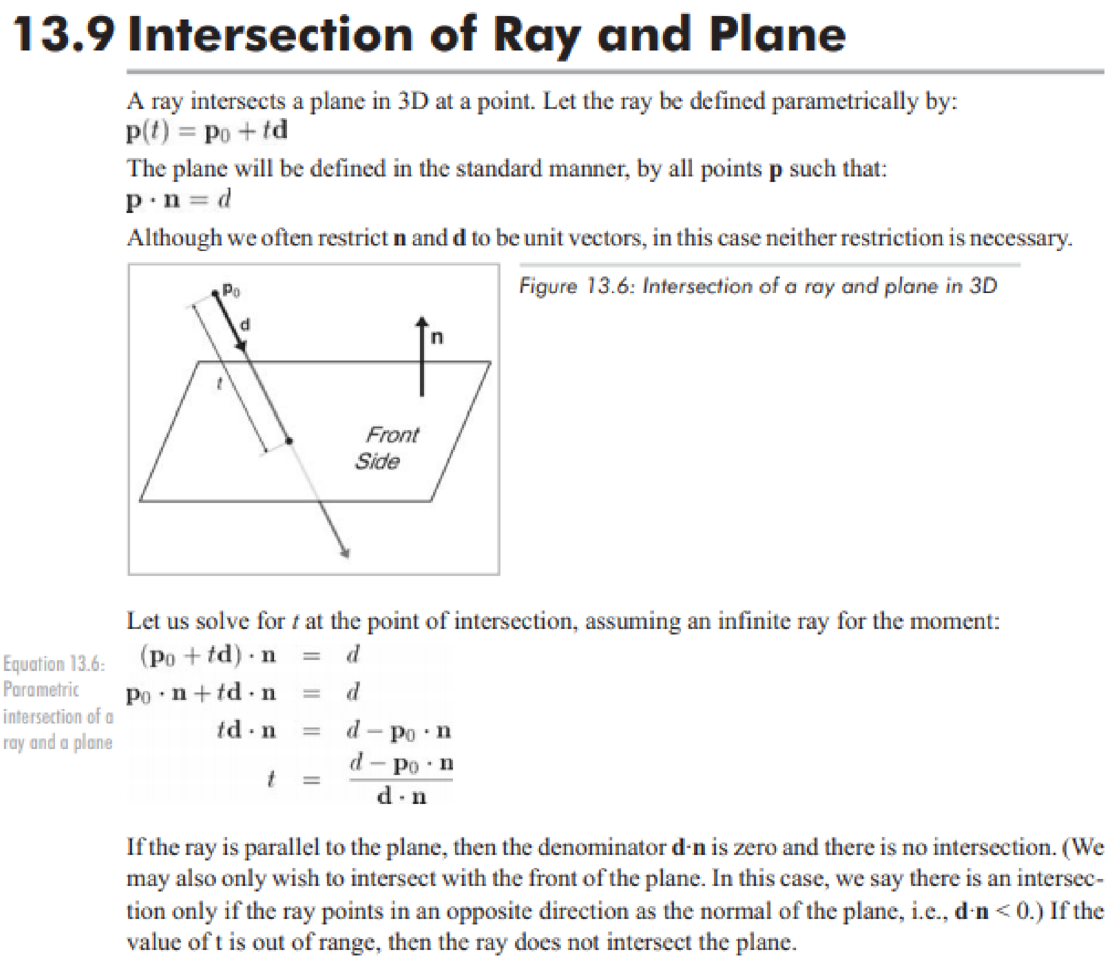
```
//LINEAS BARRIDO - ESCANEO
for (var i = -27.5; i < 27.5; i++){
    for(var j = 20.5; j > -20.5; j--){

    var vertices=[ [0,0,0], [i, j, -50] ];
    var long_vertices=vertices.length;

    for( k = 0; k < long_vertices; k++ ){
        x=vertices[k][0];
        y=vertices[k][1];
        z=vertices[k][2];

        vector = new THREE.Vector3( x, y, z );

        geometria.vertices.push( vector );

        }

        for(var r=0; r<sphere.geometry.faces.length; r++){
            if (vector.dot(sphere.geometry.faces[r].normal) > 0){
                //sphere.geometry.computeFaceNormals();
                sphere.geometry.faces[r].color = new THREE.Color( 0xff0000 );
                //sphere.geometry.faces.splice( r, 2 );
            }
        }

        for(var r=0; r<box.geometry.faces.length; r++){
            if (vector.dot(box.geometry.faces[r].normal) > 0){
                //box.geometry.computeFaceNormals();
                box.geometry.faces[r].color = new THREE.Color( 0xff0000 );
                //box.geometry.faces.splice( r, 2);
            }
        }

        for(var r=0; r<torus.geometry.faces.length; r++){
            if (vector.dot(torus.geometry.faces[r].normal) > 0){
                //torus.geometry.computeFaceNormals();
                torus.geometry.faces[r].color = new THREE.Color( 0xff0000 );
                //torus.geometry.faces.splice( r, 2 );
            }
        }
        raycast(vector);
    }
}
```

DETECCION DE PROFUNDIDAD

La detección de profundidad entre el vector y la maya de algún objeto no pude realizarlo, investigue a cerca de esto.

Brayan Sebastián Becerra Beltrán (1202052)

En el libro Math Primer for Graphics and Game Development, capitulo 13.9 (Intersection of Ray and Plane) y 13.16 (Intersection of Ray and Triangule) encontré información relacionada acerca de este inconveniente que tenía sobre la intersección entre un rayo y una superficie.



13.9 Intersection of Ray and Plane

A ray intersects a plane in 3D at a point. Let the ray be defined parametrically by:
$\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{d}$

The plane will be defined in the standard manner, by all points **p** such that:
$\mathbf{p} \cdot \mathbf{n} = d$

Although we often restrict **n** and **d** to be unit vectors, in this case neither restriction is necessary.

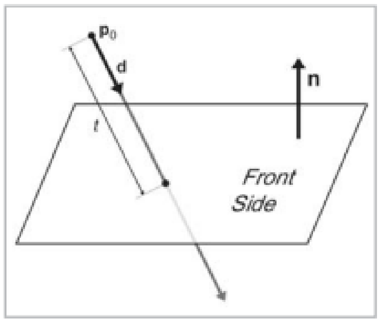


Figure 13.6: Intersection of a ray and plane in 3D

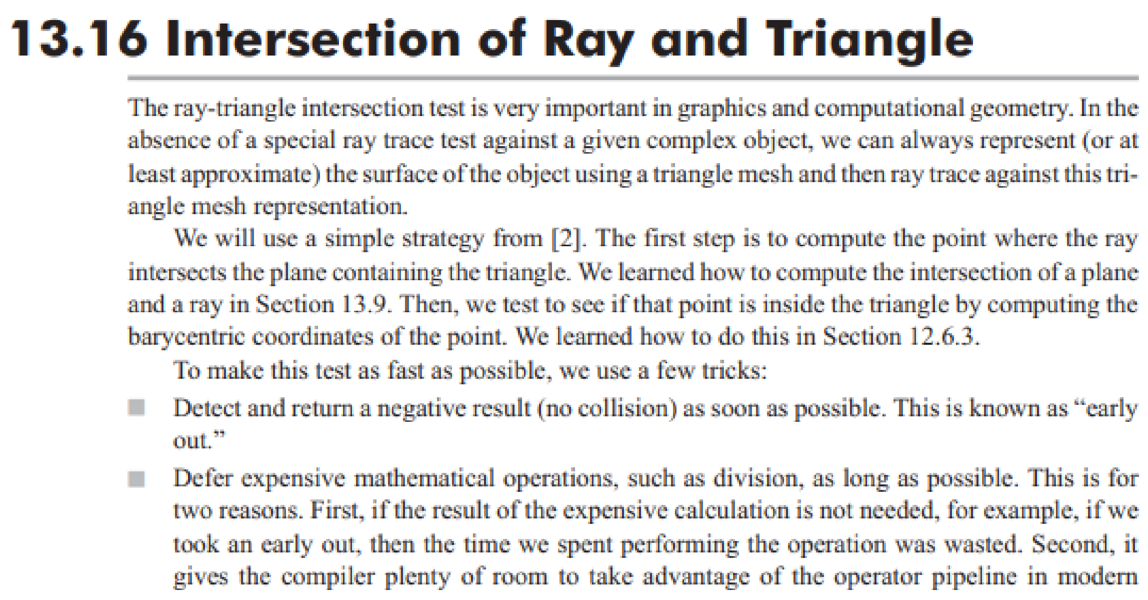
Let us solve for t at the point of intersection, assuming an infinite ray for the moment:

Equation 13.6: Parametric intersection of a ray and a plane

$$\begin{aligned}
(\mathbf{p}_0 + t\mathbf{d}) \cdot \mathbf{n} &= d \\
\mathbf{p}_0 \cdot \mathbf{n} + t\mathbf{d} \cdot \mathbf{n} &= d \\
t\mathbf{d} \cdot \mathbf{n} &= d - \mathbf{p}_0 \cdot \mathbf{n} \\
t &= \frac{d - \mathbf{p}_0 \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}
\end{aligned}$$

If the ray is parallel to the plane, then the denominator **d**·**n** is zero and there is no intersection. (We may also only wish to intersect with the front of the plane. In this case, we say there is an intersection only if the ray points in an opposite direction as the normal of the plane, i.e., $\mathbf{d}\cdot\mathbf{n} < 0$.) If the value of t is out of range, then the ray does not intersect the plane.

Brayan Sebastián Becerra Beltrán (1202052)



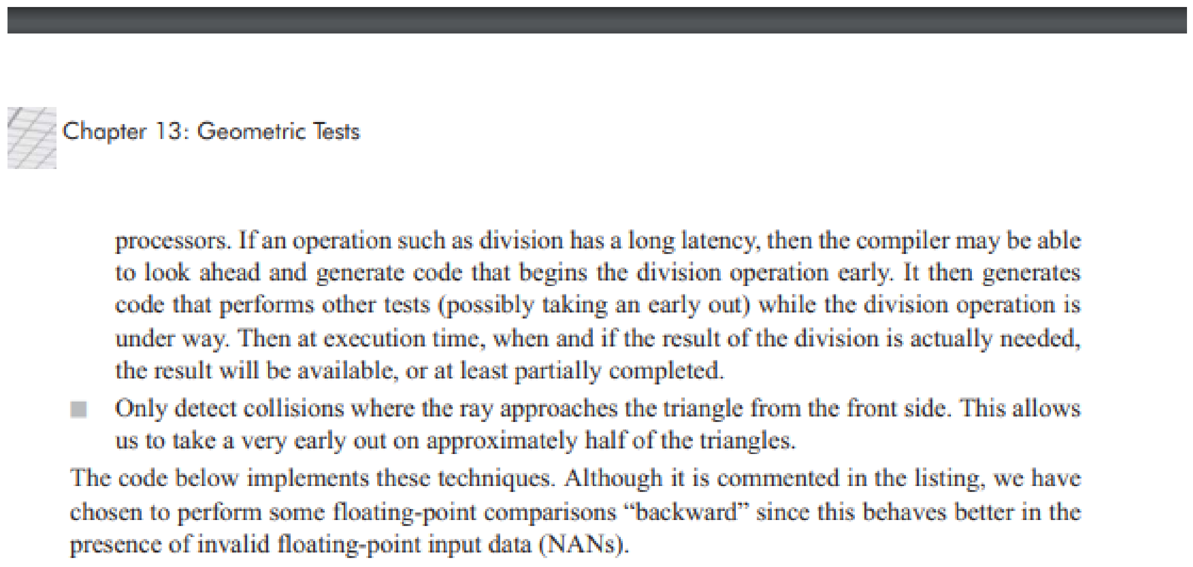
13.16 Intersection of Ray and Triangle

The ray-triangle intersection test is very important in graphics and computational geometry. In the absence of a special ray trace test against a given complex object, we can always represent (or at least approximate) the surface of the object using a triangle mesh and then ray trace against this triangle mesh representation.

We will use a simple strategy from [2]. The first step is to compute the point where the ray intersects the plane containing the triangle. We learned how to compute the intersection of a plane and a ray in Section 13.9. Then, we test to see if that point is inside the triangle by computing the barycentric coordinates of the point. We learned how to do this in Section 12.6.3.

To make this test as fast as possible, we use a few tricks:

- Detect and return a negative result (no collision) as soon as possible. This is known as "early out."
- Defer expensive mathematical operations, such as division, as long as possible. This is for two reasons. First, if the result of the expensive calculation is not needed, for example, if we took an early out, then the time we spent performing the operation was wasted. Second, it gives the compiler plenty of room to take advantage of the operator pipeline in modern

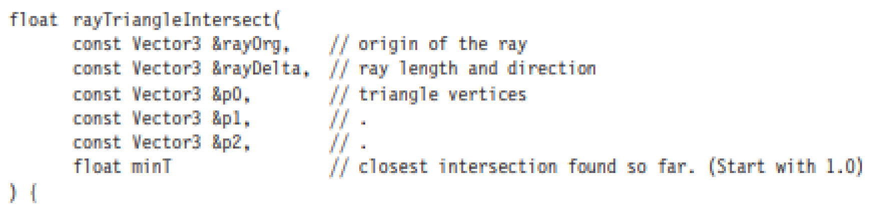


processors. If an operation such as division has a long latency, then the compiler may be able to look ahead and generate code that begins the division operation early. It then generates code that performs other tests (possibly taking an early out) while the division operation is under way. Then at execution time, when and if the result of the division is actually needed, the result will be available, or at least partially completed.

- Only detect collisions where the ray approaches the triangle from the front side. This allows us to take a very early out on approximately half of the triangles.

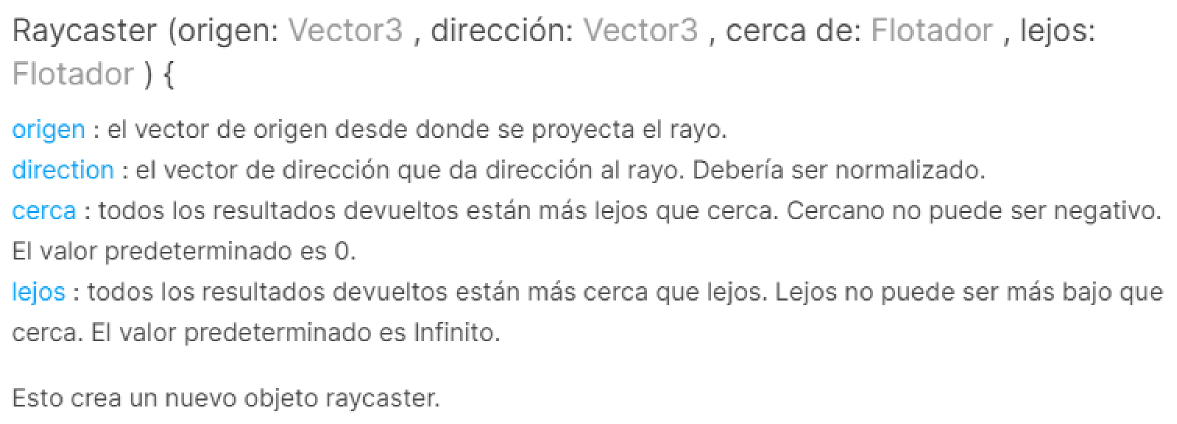
The code below implements these techniques. Although it is commented in the listing, we have chosen to perform some floating-point comparisons "backward" since this behaves better in the presence of invalid floating-point input data (NANs).

En este último capítulo nos dan un ejemplo por medio de un código, el cual es bastante parecido al Raycaster que aparece en la documentación de Three js, respecto a las constantes de origen del vector y su dirección y la distancia mínima.



```
float rayTriangleIntersect(
      const Vector3 &rayOrg,    // origin of the ray
      const Vector3 &rayDelta,  // ray length and direction
      const Vector3 &p0,        // triangle vertices
      const Vector3 &p1,        // .
      const Vector3 &p2,        // .
      float minT                // closest intersection found so far. (Start with 1.0)
) {
```

Brayan Sebastián Becerra Beltrán (1202052)



Raycaster (origen: Vector3 , dirección: Vector3 , cerca de: Flotador , lejos: Flotador) {

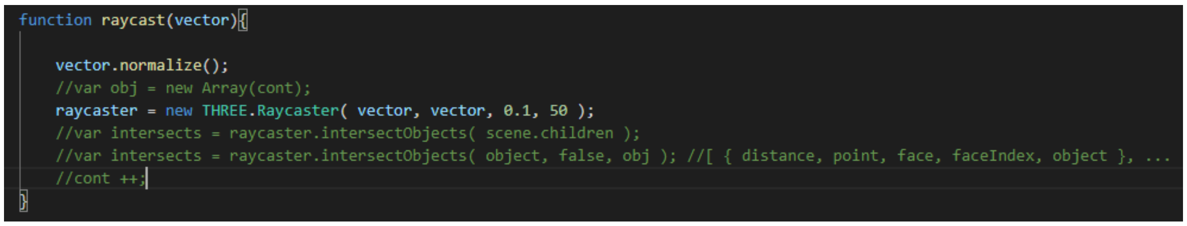
origen : el vector de origen desde donde se proyecta el rayo.
direction : el vector de dirección que da dirección al rayo. Debería ser normalizado.
cerca : todos los resultados devueltos están más lejos que cerca. Cercano no puede ser negativo. El valor predeterminado es 0.
lejos : todos los resultados devueltos están más cerca que lejos. Lejos no puede ser más bajo que cerca. El valor predeterminado es Infinito.

Esto crea un nuevo objeto raycaster.

RAYCASTER

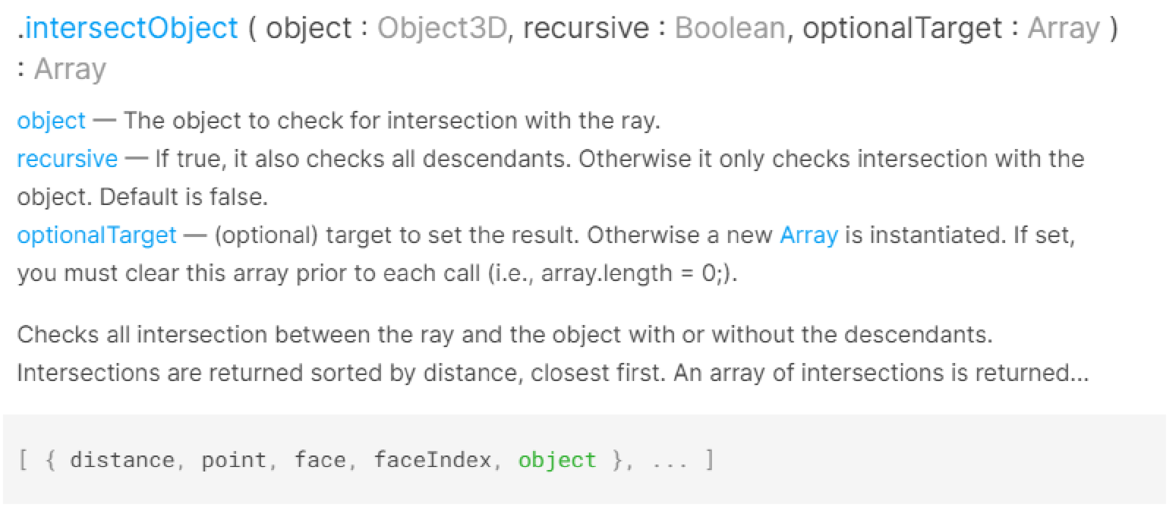
- Siguiendo los instructivos de los documentos de three js, trate de implementar el Raycaster.

- Normalizando el vector de escaneo, ya que es un requisito para el Raycaster y su dirección.
  Estableciendo las variables “far” y “near”, teniendo en cuenta las dimensiones de la cámara.



```
function raycast(vector){

    vector.normalize();
    //var obj = new Array(cont);
    raycaster = new THREE.Raycaster( vector, vector, 0.1, 50 );
    //var intersects = raycaster.intersectObjects( scene.children );
    //var intersects = raycaster.intersectObjects( object, false, obj ); //[ { distance, point, face, faceIndex, object }, ...
    //cont ++;
}
```

Implementando en el render las actualizaciones de la posición y cámara.

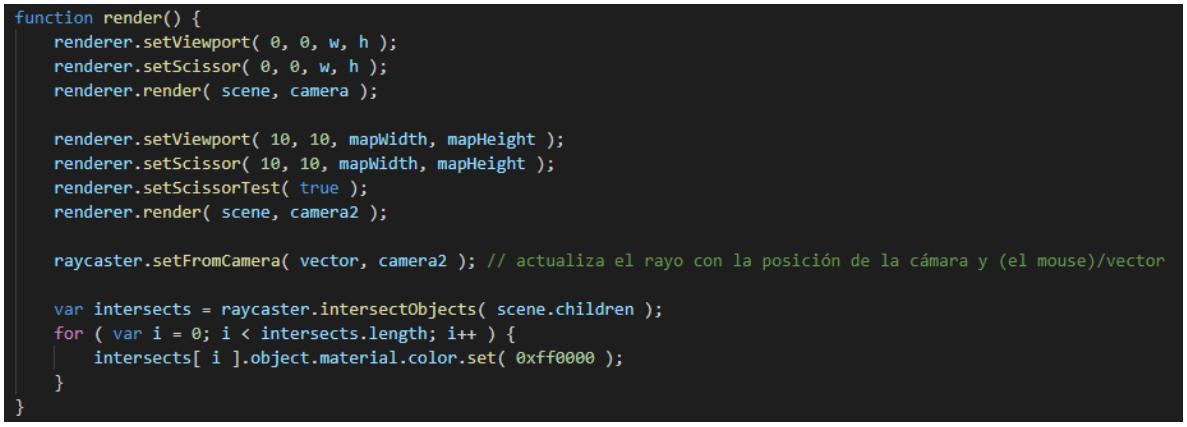


.intersectObject (object : Object3D, recursive : Boolean, optionalTarget : Array) : Array

object — The object to check for intersection with the ray.
recursive — If true, it also checks all descendants. Otherwise it only checks intersection with the object. Default is false.
optionalTarget — (optional) target to set the result. Otherwise a new Array is instantiated. If set, you must clear this array prior to each call (i.e., array.length = 0;).

Checks all intersection between the ray and the object with or without the descendants. Intersections are returned sorted by distance, closest first. An array of intersections is returned...

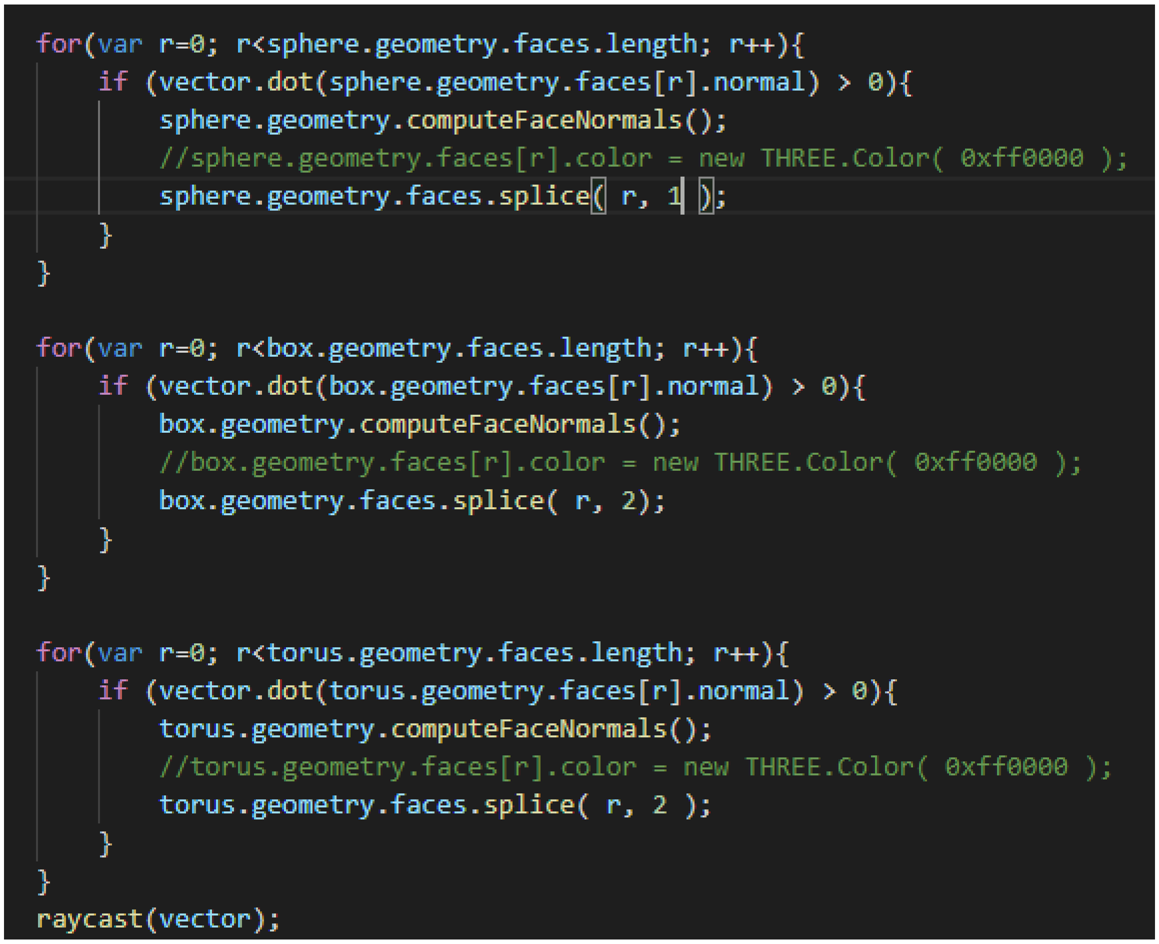
```
[ { distance, point, face, faceIndex, object }, ... ]
```



```
function render() {
    renderer.setViewport( 0, 0, w, h );
    renderer.setScissor( 0, 0, w, h );
    renderer.render( scene, camera );

    renderer.setViewport( 10, 10, mapWidth, mapHeight );
    renderer.setScissor( 10, 10, mapWidth, mapHeight );
    renderer.setScissorTest( true );
    renderer.render( scene, camera2 );

    raycaster.setFromCamera( vector, camera2 ); // actualiza el rayo con la posición de la cámara y (el mouse)/vector

    var intersects = raycaster.intersectObjects( scene.children );
    for ( var i = 0; i < intersects.length; i++ ) {
        intersects[ i ].object.material.color.set( 0xff0000 );
    }
}
```

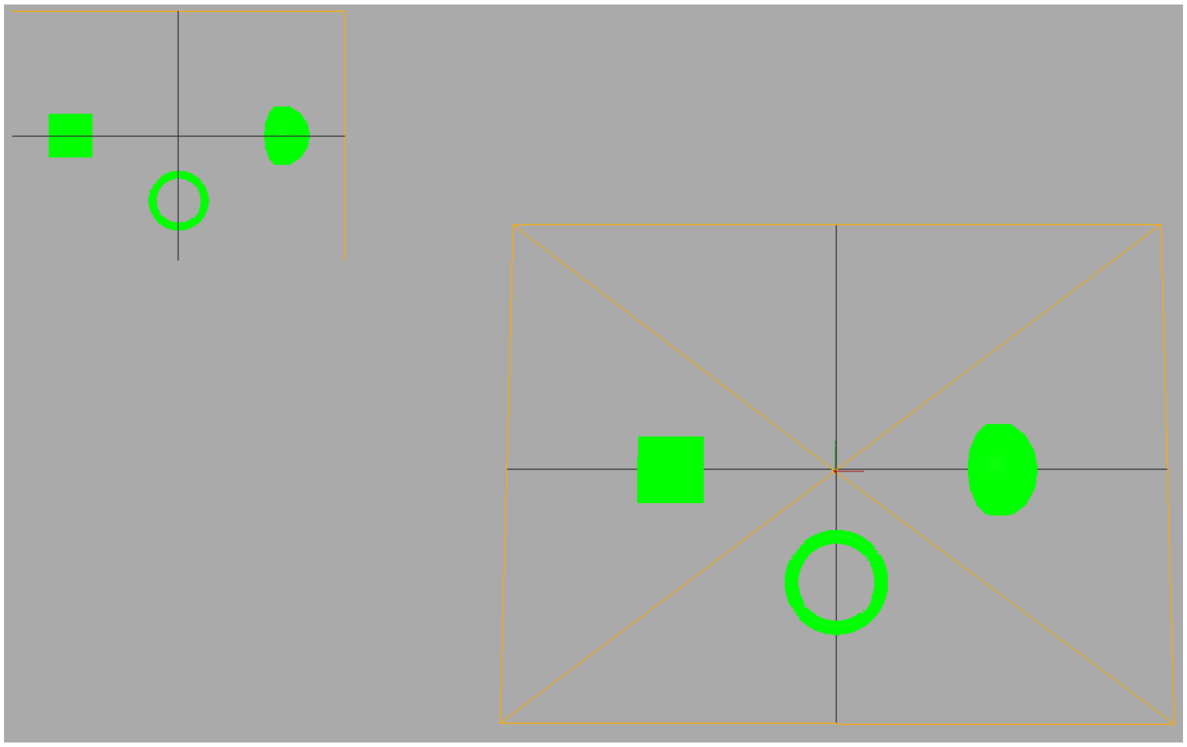
SPLICE

Con ayuda de Splice se eliminaron las caras ocultas que no se desean renderizar.



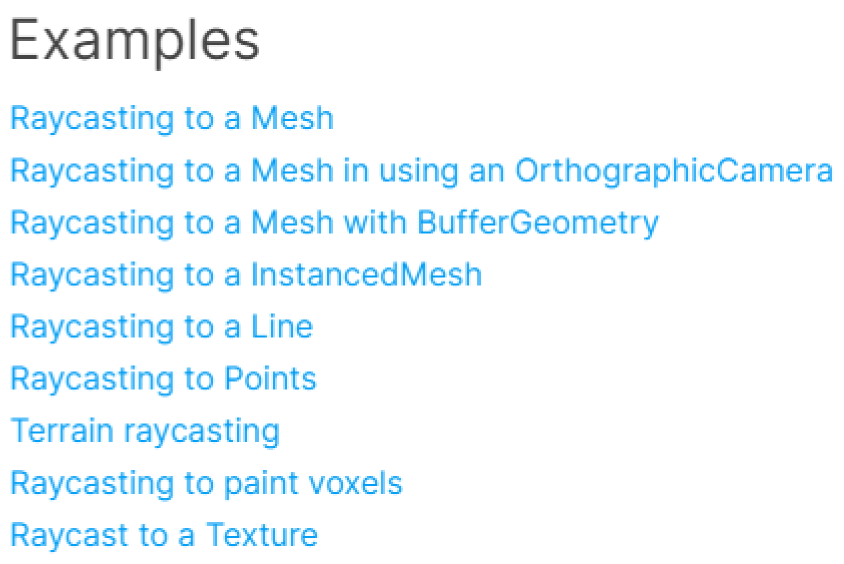
```
for(var r=0; r<sphere.geometry.faces.length; r++){
    if (vector.dot(sphere.geometry.faces[r].normal) > 0){
        sphere.geometry.computeFaceNormals();
        //sphere.geometry.faces[r].color = new THREE.Color( 0xff0000 );
        sphere.geometry.faces.splice( r, 1 );
    }
}

for(var r=0; r<box.geometry.faces.length; r++){
    if (vector.dot(box.geometry.faces[r].normal) > 0){
        box.geometry.computeFaceNormals();
        //box.geometry.faces[r].color = new THREE.Color( 0xff0000 );
        box.geometry.faces.splice( r, 2);
    }
}

for(var r=0; r<torus.geometry.faces.length; r++){
    if (vector.dot(torus.geometry.faces[r].normal) > 0){
        torus.geometry.computeFaceNormals();
        //torus.geometry.faces[r].color = new THREE.Color( 0xff0000 );
        torus.geometry.faces.splice( r, 2 );
    }
}
raycast(vector);
```

Brayan Sebastián Becerra Beltrán (1202052)



EJEMPLOS

Algunos ejemplos que investigue fueron los siguientes:



Examples

Raycasting to a Mesh

Raycasting to a Mesh in using an OrthographicCamera

Raycasting to a Mesh with BufferGeometry

Raycasting to a InstancedMesh

Raycasting to a Line

Raycasting to Points

Terrain raycasting

Raycasting to paint voxels

Raycast to a Texture

https://tfetimes.com/wp-content/uploads/2015/04/F.Dunn-I.Parberry-3D-Math-Primer-for-Graphics-and-Game-Development.pdf

https://threejs.org/docs/#api/en/core/Raycaster

Brayan Sebastián Becerra Beltrán (1202052)

https://iguagofernando.wordpress.com/2018/03/04/entendiendo-el-buffer-de-profundidad/

https://www.tutorialspoint.com/computer_graphics/visible_surface_detection.htm

https://riptutorial.com/es/three-js/example/17088/recoleccion-de-objetos---raycasting

https://living-sun.com/es/javascript/401149-get-object-position-in-a-threejs-scene-dynamically-javascript-threejs.html