Animation Loop

If we want to show animation, we have to redraw the screen periodically.

The browser automatically refreshes its presentation few times in a second, so we can "ask" the browser to let us know whenever it refreshes.

To do so we use the browser's *requestAnimationFrame* function. The function gets a call back function as a parameter, which will be activated in every refresh.

```
let mainLoop = function() {

    requestAnimationFrame(mainLoop);
};
```

Ask the browser to let us know when it refreshes itself.

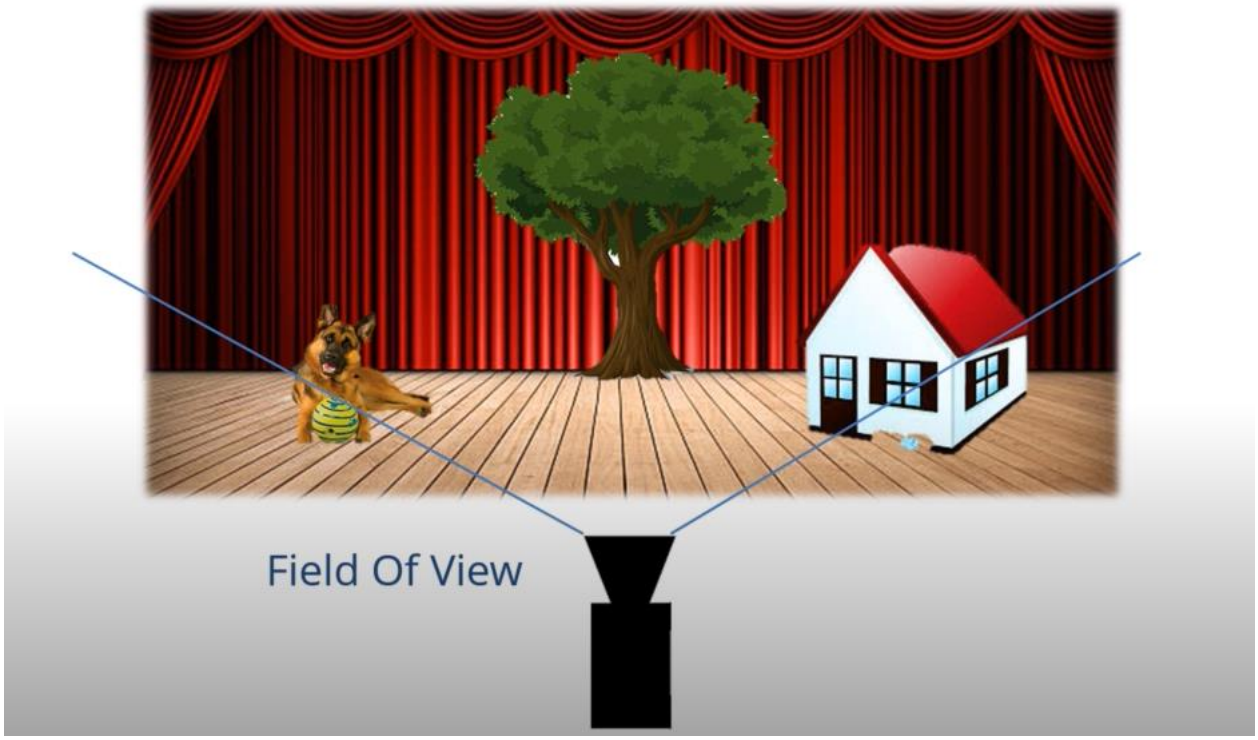
In each refresh the *mainLoop* function would be called again, creating an animation loop this way.

The Scene

- The first thing we have to do in a 3d application is to set up the scene.
- The scene is the place where all the action happens.

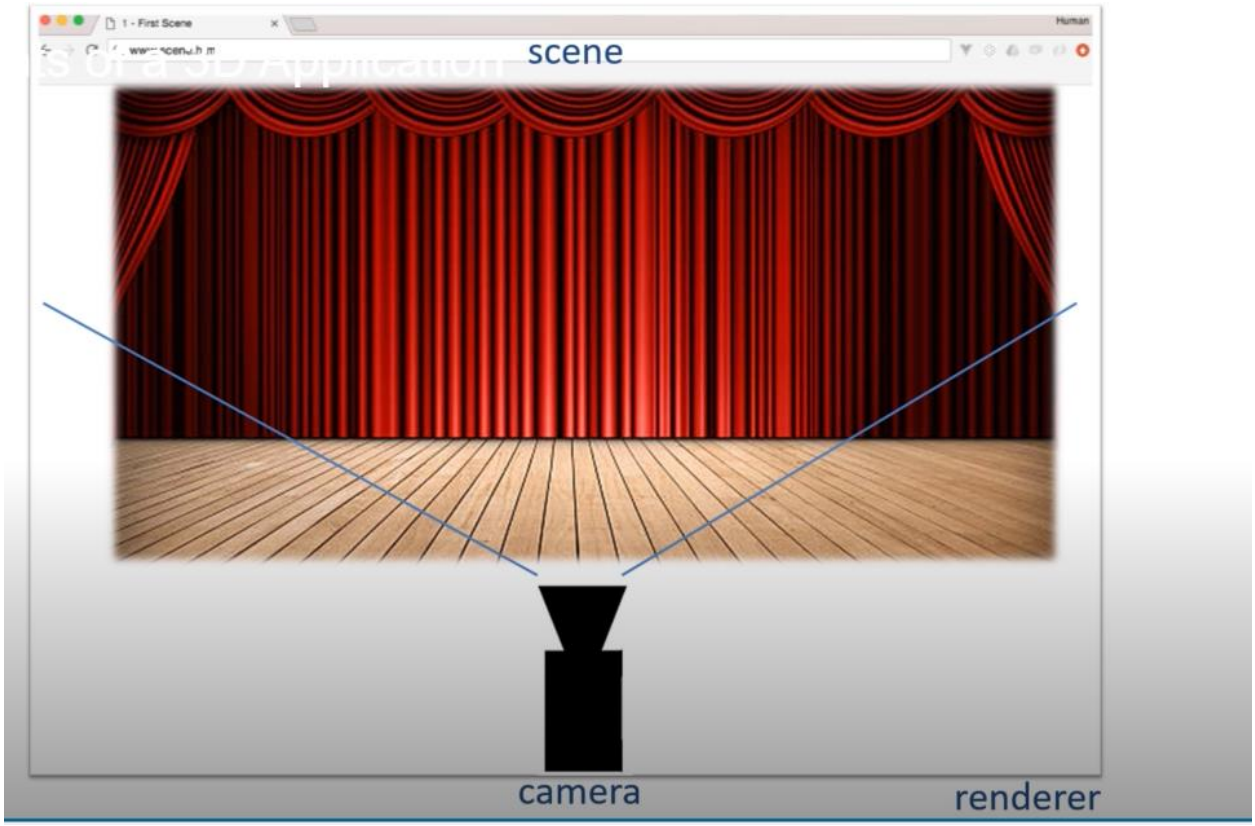
The Camera

- The scene contains all elements in the application.
- But we won't see the scene if we don't watch it through a camera.
- The camera is like a watching mechanism that allows us to see the scene. It determines which part of the scene we'll see, and how will we see it.


Field Of View

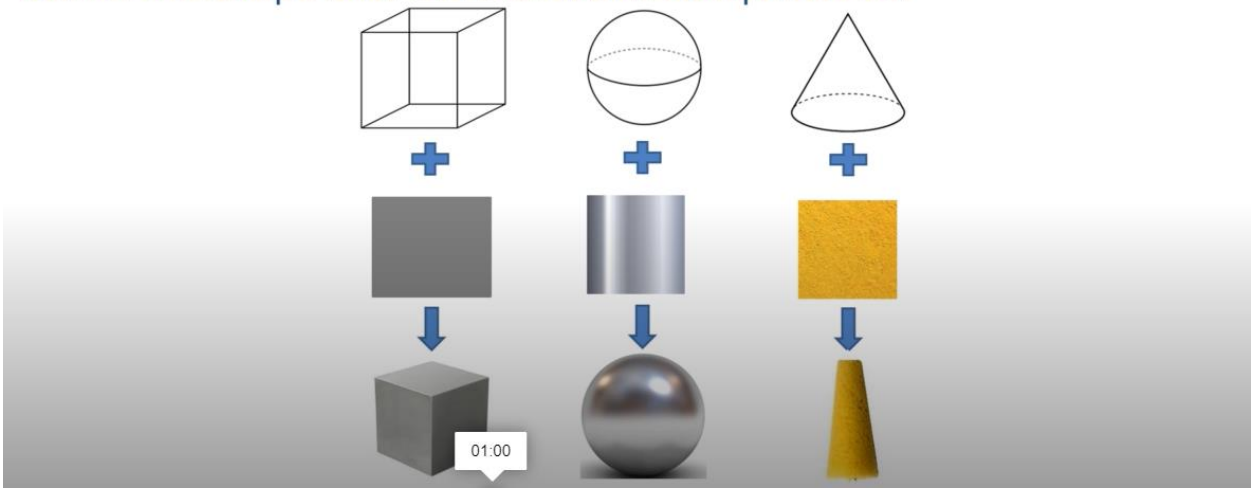
The Renderer

- The final thing is to take the scene that shows the application, and the camera that sets the point of view, and locate them both on the html.
- This task is made by the renderer object.
- This object interacts with the html and enables us to see the application we've made on the browser.



Working with Geometry

When we want to locate a geometric shape on the scene we have to define the shape and the material it composed of:



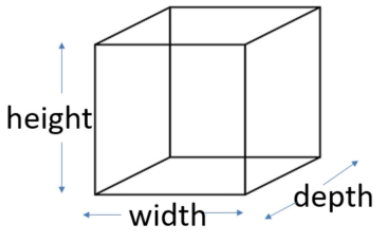
The action of taking a geometric shape and wrap it with a specific material is done via an object called “mesh”.

The mesh object gets as parameters the geometry and the material, and creates a visible 3d shape out of it.

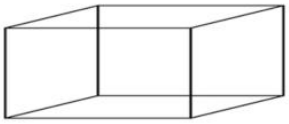
The BoxGeometry represents a 3d box.

```
let geometry = new THREE.BoxGeometry(1, 1, 1);
```

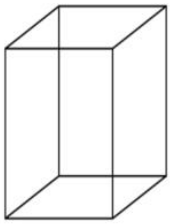
The constructor gets the box’s width, height and depth as parameters (there are three more parameters we don’t use now)



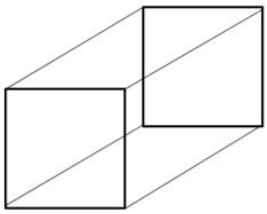
```
let geometry = new THREE.BoxGeometry(3, 1, 1);
```



```
let geometry = new THREE.BoxGeometry(1, 3, 1);
```



```
let geometry = new THREE.BoxGeometry(1, 1, 3);
```



```
let cube;

let createCube = function() {
    let geometry = new THREE.BoxGeometry(1, 1, 1);
    let material = new THREE.MeshBasicMaterial({color : 0x00a1cb});
    cube = new THREE.Mesh( geometry, material );
    scene.add(cube);
};
```

Sphere

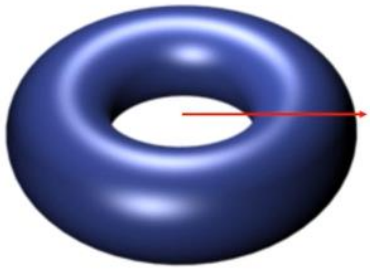
```
let geometry = new THREE.SphereGeometry(5, 30, 40);
```

The radius

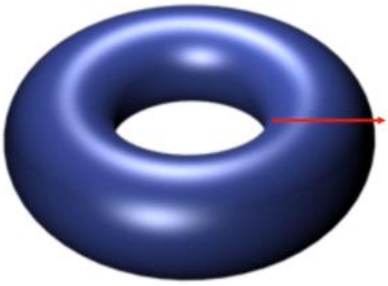
```
let geometry = new THREE.SphereGeometry(5, 30, 40);
```

Number of horizontal and vertical segments

The sphere is actually built from segments on its surface. The more segments there are, the more “Spheric” it would be.

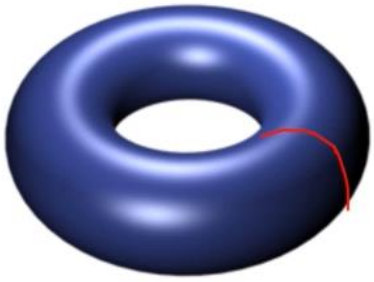


```
let geometry = new THREE.TorusGeometry(5, 1, 20, 20, 2 * Math.PI)
```



```
let geometry = new THREE.TorusGeometry(5, 1, 20, 20, 2 * Math.PI)
```

More higher – more round. Decrease – more like a disk



```
let geometry = new THREE.TorusGeometry(5, 1, 20, 20, 2 * Math.PI)
```

Number of segments

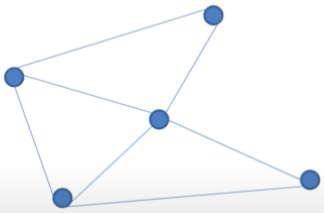
```
let geometry = new THREE.TorusGeometry(5, 1, 20, 20, 2 * Math.PI)
```

2*Math.PI – by default, can skip in constructor.

The object Geometry represents any kind of geometric shape in Three.js.

A geometric shape is made of vertices, each vertex is a 3d point:

Three vertices can make a face:

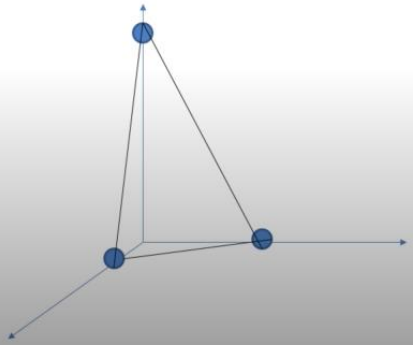


If we create a custom geometry object, we have to build manually its vertices and faces.

```
let geometry = new THREE.Geometry();
geometry.vertices.push(new THREE.Vector3(3, 0, 0));        // vertices[0]
geometry.vertices.push(new THREE.Vector3(0, 5, 0));        // vertices[1]
geometry.vertices.push(new THREE.Vector3(0, 0, 2));        // vertices[2]

geometry.faces.push(new THREE.Face3(0, 1, 2));

let material = new THREE.MeshBasicMaterial(
        {color: 0Xffffff,
         side: THREE.DoubleSide } );
let shape = new THREE.Mesh(geometry, material);
scene.add(shape);
```



MeshBasicMaterial

- The basic material gives a uniform color to each point on the geometry's surface:

```
let material = new THREE.MeshBasicMaterial({color: 0xffffff});
```

Normals

Every plane has a normal – a vector that is perpendicular to the plane (it has a 90 degree angle with the plane) :

- The Normal Material maps a normal vector to RGB color.
- For each face the material gives a color according to it's current normal.

MeshDepthMaterial

- The depth material gives a geometry gray scale colors based on the distance from the camera.
- Closer points are darker, farther away points become whiter.

Line Material

- The Line Material shows only the outline of the shape, some kind of a skeleton.
- To use this material we have to use a Line mesh instead of the usual Mesh object.

Directional Light

- To see the effect of the light-sensitive materials, we have to add light to our scene.
- A directional light is a light that comes from one direction (like the sun light).
- The object DirectionalLight represents this light, and its constructor accepts the light color.
- By default, this light radiates its rays to the (0, 0, 0) position, and comes from the top of the scene.

MeshLambertMaterial

- A material for non-shiny surfaces like stone. It gives the surface a dull look and with light it becomes brighter.
- Based on the Lambertian Model for modelling lights reflection.

Lambert Material

```
let material = new THREE.MeshLambertMaterial( {
                                        color:
                                        emissive:
                                        });
```

Emissive is like a glow from the material, that is unaffected by light. The emissive property is the emissive color of the material (default is null meaning that by default the material doesn't emit color).

```
let material = new THREE.MeshLambertMaterial( {
                                        color:
                                        emissive:
                                        emissiveIntensity:
                                        });
```

Defines how intense the emissive color will be. This is a number between 0 – no emissive, and 1 – full emissive. The default is 1.

Phong Material

```
let material = new THREE.MeshPhongMaterial( {
                                        color:
                                        emissive:
                                        emissiveIntensity:
                                        shininess:
                                        });
```

How much the material shines. Lower values mean less shining. The default is 30.

```
let material = new THREE.MeshPhongMaterial( {
                                        color:
                                        emissive:
                                        emissiveIntensity:
                                        shininess:
                                        specular:
                                        });
```

Defines the color of the shine. The default is 0x111111 (dark grey)

MeshStandartMaterial

- Based on a phisycal material model.
- This material uses a metalic-roughness model to simulate real materials.
- It combines both the lambert and sphong materials

Standard Material

```
let material = new THREE.MeshStandardMaterial( {
                                        color:
                                        emissive:
                                        emissiveIntensity:
                                        metalness:
                                        roughness:
                                        });
```

Defines how rough the material appears. 0 is for a smooth mirror reflection, 1 for fully diffuse.

Ambient Light

- An ambient light is a light that illuminates everywhere in the same level. It doesn't come from a specific direction.
- We usually use it to light the whole scene with a constant color so there will be no positions that have no light.

Ambient Light

```
let light = new THREE.AmbientLight(color, intensity);
```

color – the light's color
intensity – The light's intensity, the default is 1

Hemisphere Light

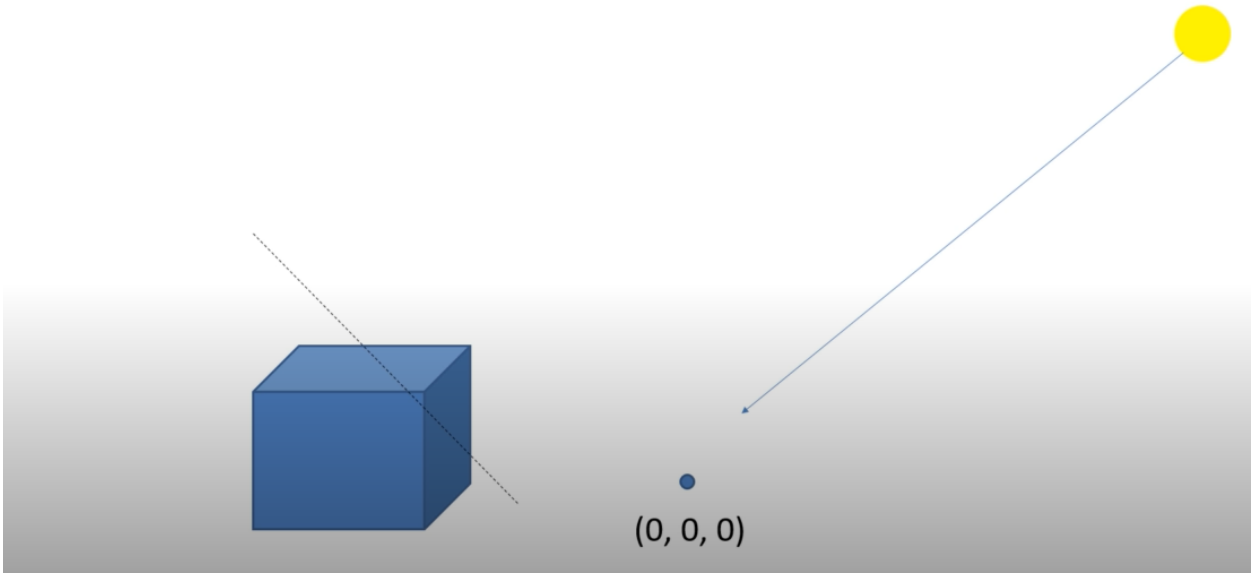
- The ambient light gave a uniform light to the whole scene, but because of that it looks a little unrealistic.
- The hemisphere light is similar to the ambient light in that it doesn't have a position or location.
- But the hemisphere light source fades as it goes from the top (sky) to the bottom (ground).

```
let light = new THREE.HemisphereLight(skyColor, groundColor, intensity);
```

skyColor – the color of the light that pours from the top of the scene.
groundColor – the color to which the sky would fade eventually.
intensity – The light's intensity, the default is 1

Directional Light

- The directional light is a light that pours from a light source and aims to a direction.
- It's like the sun light – since the sun is far away, we can practically assume that all light rays are parallel.

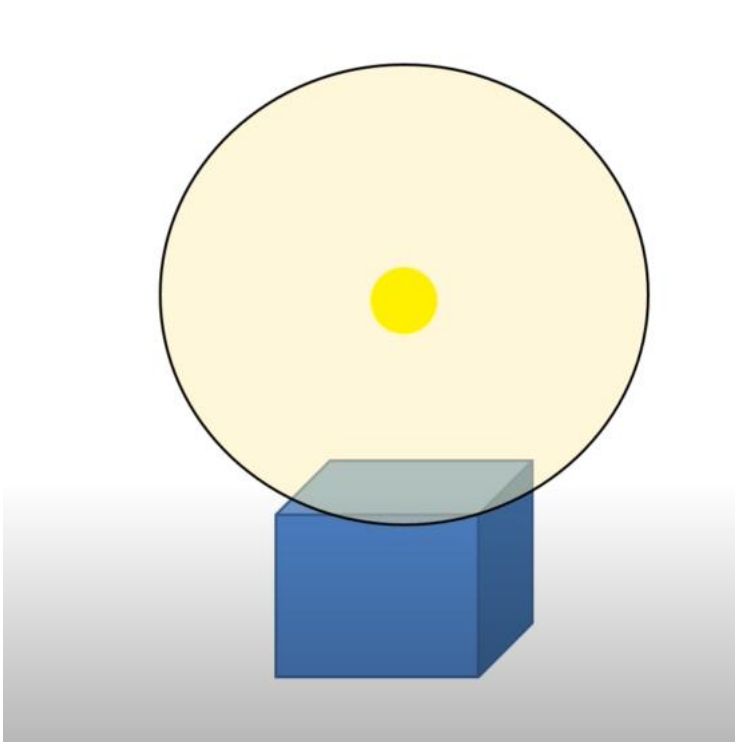


```
let light = new THREE.DirectionalLight(color, intensity);
```

color – the light's color
intensity – The light's intensity, the default is 1

Point Light

- Point light is a light that radiates from a single point to all directions.
- It's like a light bulb – the light fades as a function of the distance from the light source.



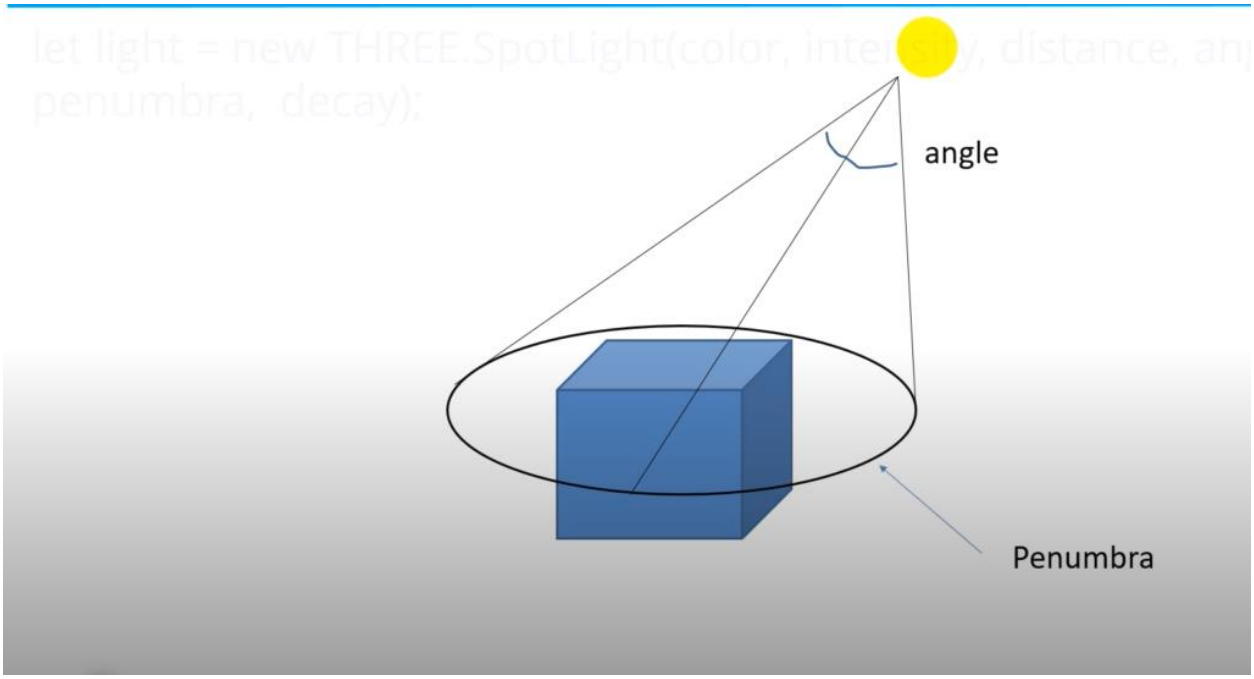
Point Light

```
let light = new THREE.PointLight(color, intensity, distance, decay);
```

color – the light's color
intensity – the light's intensity, the default is 1
distance – the distance from the light source where the intensity becomes 0. If it set to zero, the light never fades away. The default is zero.
decay – the amount the light fades along the distance. Default is 1.

Spot Light

- The spot light, as its name suggests, is a light that comes from a single point (like a spot) and it forms a cone shape.



let light = new THREE.SpotLight(color, intensity, distance, angle, penumbra, decay);

color – the light's color
intensity – the light's intensity, the default is 1
distance – the distance from the light source where the intensity becomes 0. If it set to zero, the light never fades away. The default is zero.
angle – the cone's angle in radians, the maximum can be PI / 2
penumbra – the percent of light that is shadowed. The values are from 0 (default) to 1.

Camera

- The camera enables us to see the scene and all it contains.
- The position of the camera and its type would determine how we would see the scene.
- We saw that the renderer object takes the camera as a parameter and uses it to render the scene.

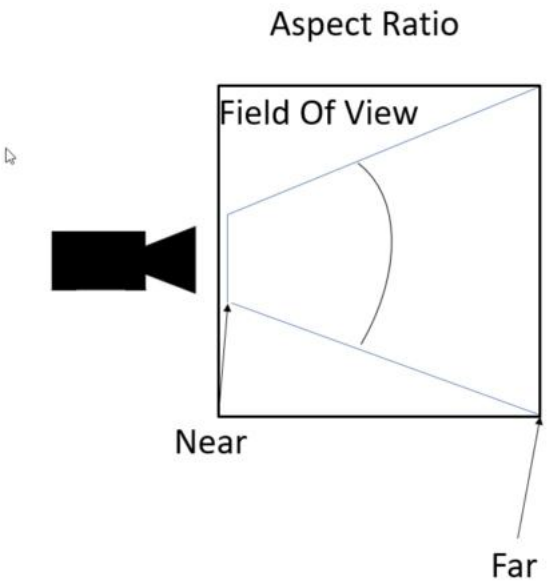
- The Three.js Camera object represents the basic camera.
- By itself it's useless, but Three.js uses it to define the main two camera types – the perspective and orthographic cameras.
- The camera object contains the projection matrix that contains the way the scene is projected.

Perspective Camera

- This camera designed to mimic how the human eye sees.
- It gives a perspective to the scene – closer objects would look bigger than farther objects.

Near – minimum distance the figure will be seen by camera. Far – maximum.

Frustum

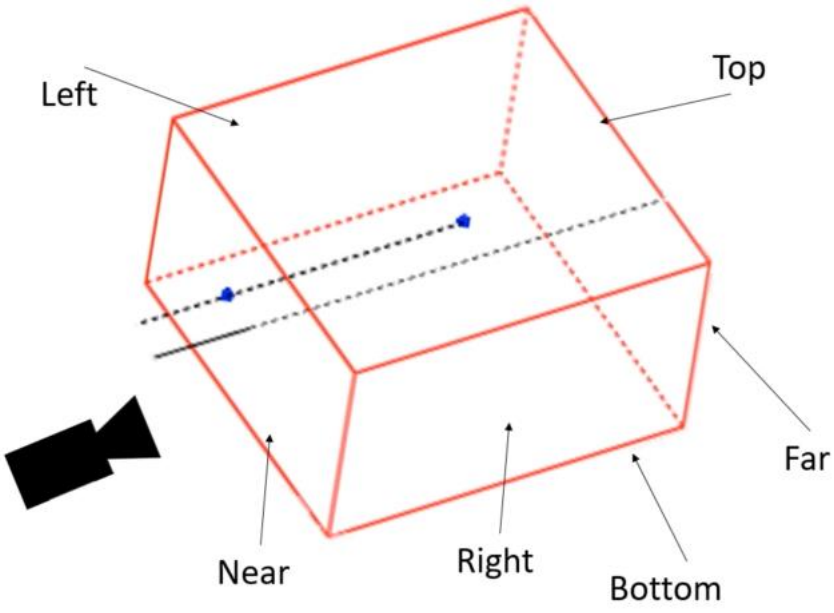



```
let camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
```

Orthographic Camera

- This camera doesn't give a perspective to the scene.
- Object's size stays the same, no matter what is the distance from the camera.

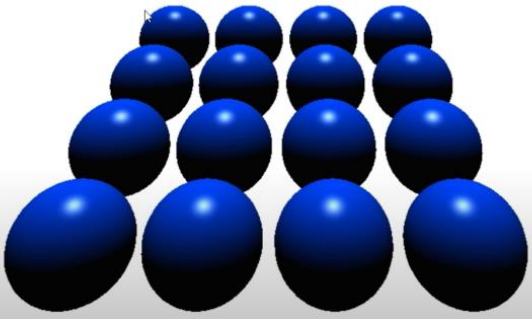
При создании такой камеры заботимся лишь о расстояниях, в пределах которых объект будет виден.




```
let camera = new THREE.OrthographicCamera(left, right, top, bottom, near, far);
```

Orthographic Camera

Perspective Camera



Orthographic Camera

Orthographic Camera

Cameras

- In this section we talked about the camera.
- The camera is the object that “sees” the scene, and it determines how will we see it on the screen.
- The perspective camera – gives a perspective to objects.
- The orthographic camera – doesn’t use perspective – distance is meaningless.

User Interaction

- We want our applications to react to user input.
  - Keyboard pressed
  - Mouse clicked
  - Mouse moved
  - Wheel rotated

Event Handling

- Every component fires an event whenever something interesting happens to it:
- A button fires a “click” event when it is pressed.
- A text box fires a “keyboard” event when a text is typed inside it.
- In order to react to these events we have to create **event handlers**.
- An event handler is a function that invokes when a specific event is called.

```
<html>
    <head>
        <title>18 - Events</title>
    </head>
    <body>
    </body>
</html>

<script>
    let onMouseClick = function(e) {
        console.log("Hello");
    };

    document.addEventListener("click", onMouseClick, false);
</script>
```

This function would perform the logic whenever the mouse was clicked in the window. We called it “onMouseClick” but we can name it whichever we want to. The function accepts one parameter which is the event object itself.

```
<html>
    <head>
        <title>18 - Events</title>
    </head>
    <body>
    </body>
</html>

<script>
    let onMouseClick = function(e) {
        console.log("Hello");
    };

    document.addEventListener("click", onMouseClick, false);
</script>
```

The addEventListener function attaches a function to a specific event of the document.

Here we attach the function for the “click” event. The function we want to perform whenever a click happens is “onMouseClick”

Common Events

click – when the mouse clicks the screen.
mousemove – when the user moves the mouse freely over the screen.
mouseenter – when the mouse pointer enters the screen area.
mouseleave – when the mouse pointer leaves the screen area.
keydown – when a key was pressed.

Keyboard Events

- The canvas would fire a keyboard event when a key pressed.
- The event is called “keydown”.

```
document.addEventListener(“keydown”, onKeyDown, false);
```

Keyboard Events

```
let onKeyDown = function(e) {
        console.log(e.keyCode);
}
```

The parameter e is the event object, which is passed to the function by the browser.

This object contains the key that was pressed. The "keycode" attribute contains the ASCII code of the key.

Some useful codes:

37 = left arrow
38 = up
39 = right
40 = down
32 = space
13 = enter

Mouse Events

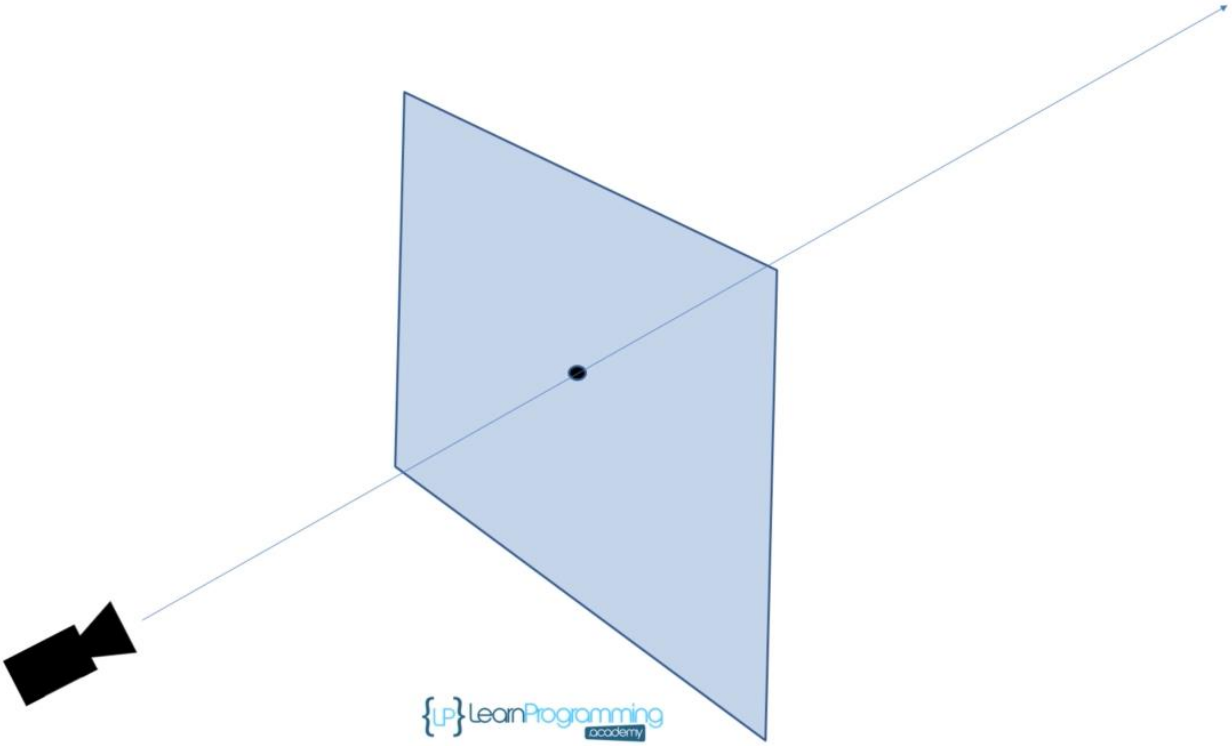
- The canvas would fire a mouse event whenever the user performs an action with the mouse.
- click
- mouse move
- mouse drag
- mouse enter
- mouse leave

Mouse Picking

- A lot of times we would like to use the coordinates of the pointer as input for our application.
- The coordinates are integer numbers between 0 and the canvas' dimensions.
- Those units are not coherent with the 3d world Three.js uses.
- In addition, the mouse coordinates represent a two dimensional point, while the application's world is three dimensional.

RayCaster

- We have to convert the mouse 2d point into a 3d world coordinates.
- We cast a virtual “ray” from the camera and to the mouse position on the screen.
- Those two points conceive a straight line that goes to infinity.
- We can now pick points along this lines that would represent the 3d application world.



The RayCaster is an object that cast a ray between the camera and the normalized mouse position.

After the ray was casted, we can check if it intersects with application's objects, and we can calculate points on the ray according to their distance from the camera.

```
let rayCast = new THREE.Raycaster();

rayCast.setFromCamera(mouse, camera);
```

A 2d point that represents the normalize mouse-position on screen

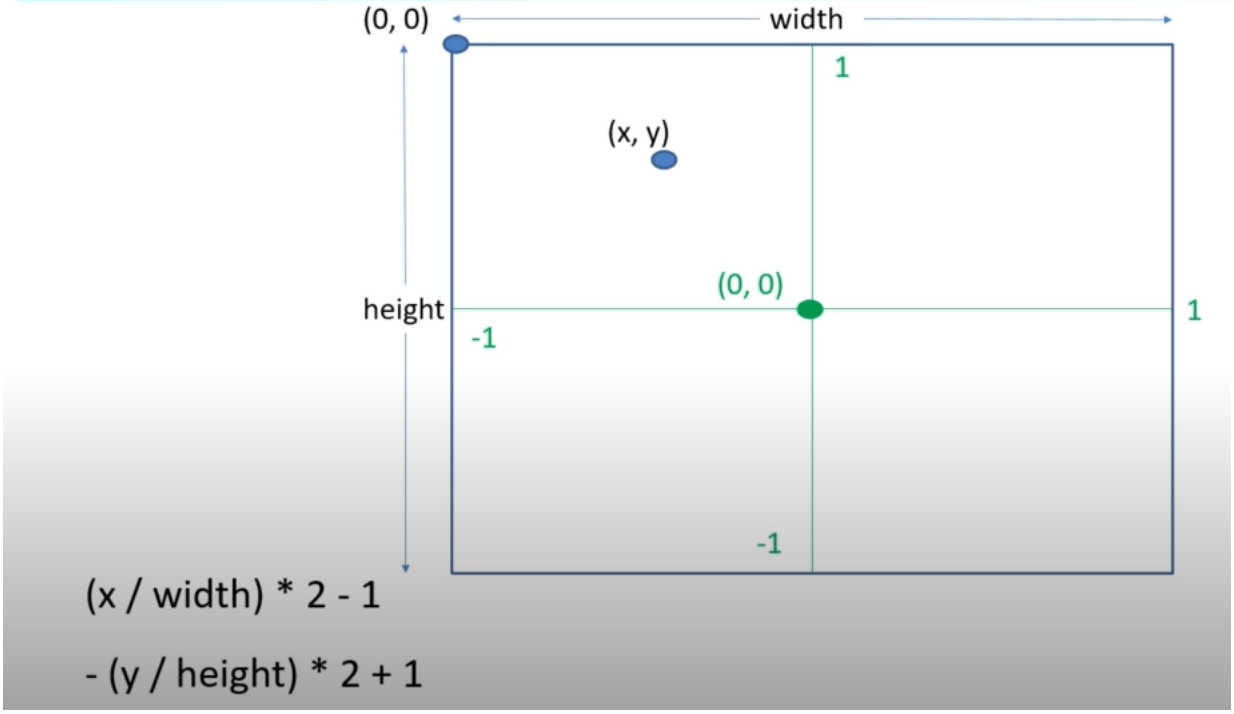
Mouse Events

The mouse position should be normalize when we pass it to the ray caster.

How do we normalize the mouse position?

The normalize point's coordinates should be real numbers between -1 and 1, while the screen coordinates are positive integers that represent the screen pixels.

Mouse Events



Mouse Events

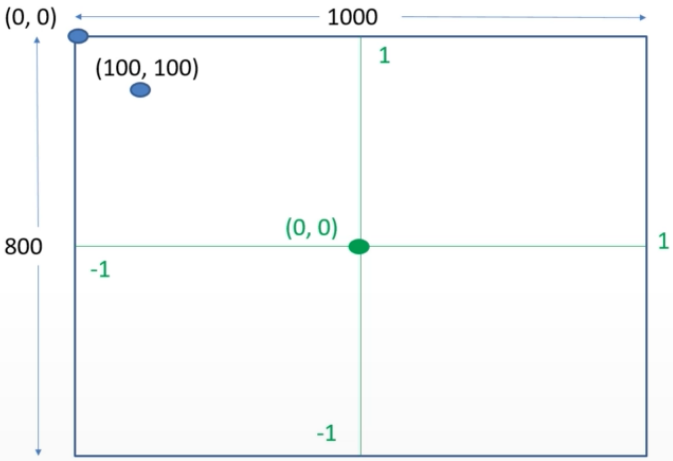
width = 1000, height = 800
x = 100, y = 100

(x / width) * 2 – 1 =

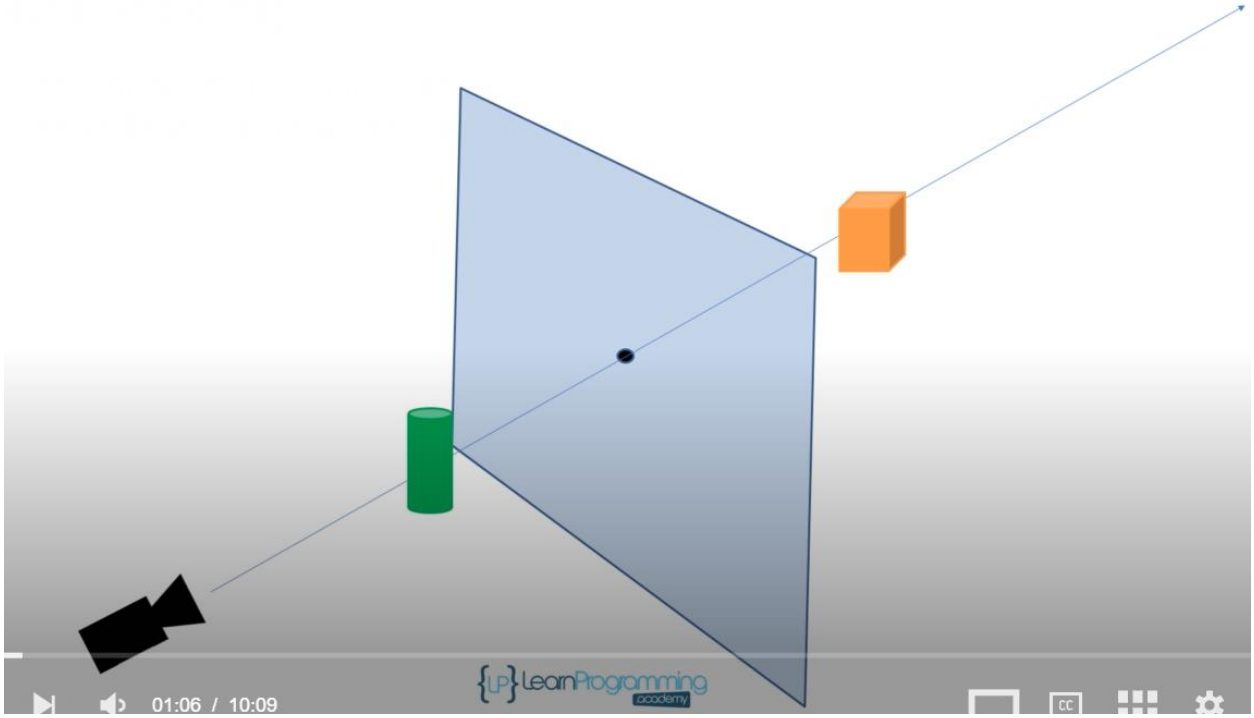
(100 / 1000) * 2 – 1 = -0.8

- (y / height) * 2 + 1 =

- (100 / 800) * 2 + 1 = 0.75



Cylinder and cube are objects found in RayCaster



Mouse Picking

```
rayCast = new THREE.Raycaster();
rayCast.setFromCamera(mouse, camera);
intersects = rayCast.intersectObjects(scene.children);
```

inersectObjects returns an array of objects that intersect with the current ray.

scene.children contains all the elements that were added to the scene.

Every object in the returned array contains an “object” attribute that is the Three.js geometric object.

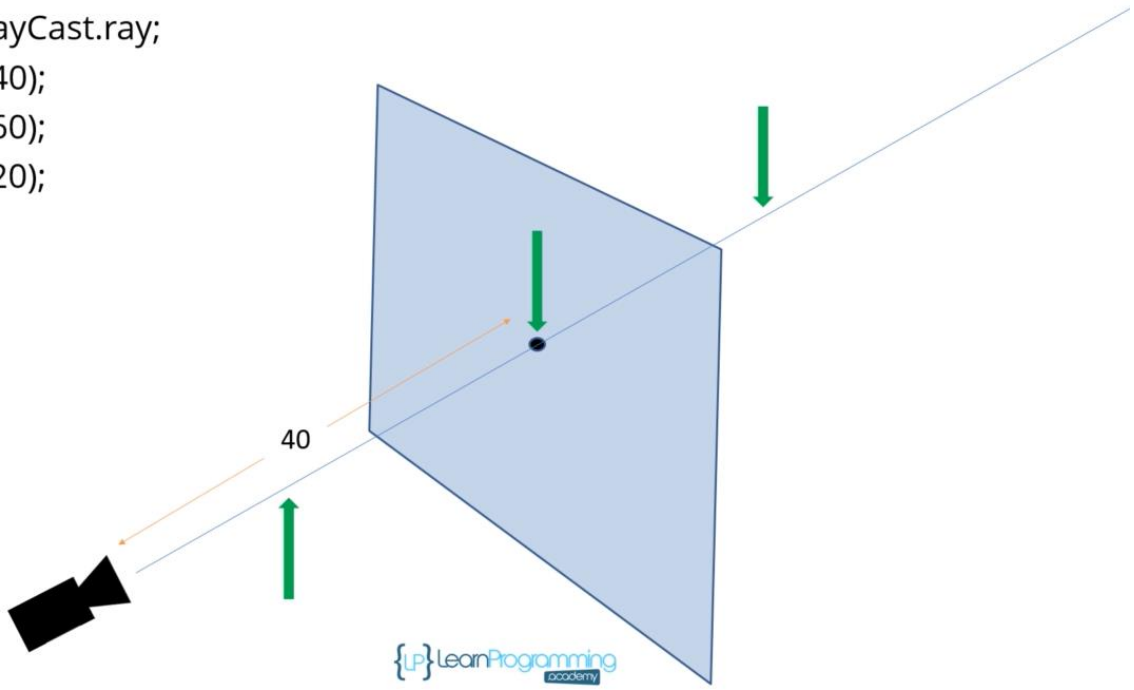
Ray Object

```
rayCast = new THREE.Raycaster();
rayCast.setFromCamera(mouse, camera);
rayCast.ray
```

The ray object represents the virtual ray that was casted. The ray origin is the camera position.

We can use the function “at” of the ray object. This function gets a distance from the origin, and returns a 3d point from this distance, on the ray.

```
ray = rayCast.ray;
ray.at(40);
ray.at(60);
ray.at(20);
```



Events Handling

- The user interacts with the application via the keyboard, mouse and so on.
- Every act is translated into an event in the browser.
- An event listener is a function that waits for a specific event to occur.

Keyboard Events Handling

- The keyboard event is called when a key is pressed.
- In this event we can extract the key code of the key that was pressed.

Mouse Events Handling

- The mouse events invoked from mouse interaction.
- We're usually interested in the mouse pointer's coordinates, which are given as positive integer values.
- In order to treat those coordinates as 3d position we had to convert them.
- The Raycaster object casts a virtual ray from the camera to the mouse position, and enables us to use the mouse coordinates in the application world.

What we've seen so far

- We know all the elements needed to set up a 3D application with Three.js:
- Scene and geometries.
- Materials.
- Lights.
- Cameras.

TextureLoader

- We used Three.js various materials to give all kind of looks to our objects.
- We can give even more interesting appearance to our objects by applying them a texture.
- A texture is an image that would wrap the object.

TextureLoader

- In order to apply a texture we must first load the texture.
- The texture is usually an image that is loaded into a Texture object.
- To load the image we use the TextureLoader object.

Textures

```
let texture = new THREE.TextureLoader().load(imagePath );
```

The TextureLoader object is used to load a texture from a file.

The load function accepts an image file path and returns a texture object, contains this image.

Loading Images

- Most web browsers don't allow Javascript code to see the local computer's files system.
- If we try to load an image from our computer we'll get an error:
- "Access to image has been blocked by CORS policy"
- CORS – Cross Origin Source – a policy that don't allowed files to be transferred from unknown origin.

Loading Images

- There are several solutions:
- We can remove the CORS protection from the web browser, but this operation may expose the computer to malicious software.
- We can create our own local server that would serve the image to the loader.
- We can load an image that is found on a remote server, via its url.

Textures

After the texture was loaded we can apply it to our material.

Every material object has a property called “map”. This property defines the texture of the material, and it should be a texture object.

```
let texture = new THREE.TextureLoader().load(imagePath );
let material = new THREE.MeshBasicMaterial({map: texture});
```

Panorama

- Panoramic view of a scene allow us to rotate the camera in 360 degrees and see a different view.
- To do so we should have an image that would wrap the entire view.

Texture

```
<script>
    let scene, camera, renderer, sphere, target, texture;
    let ADD = 0.005, theta = 0;

    let createGeometry = function() {
        texture = new THREE.TextureLoader().load(
'https://upload.wikimedia.org/wikipedia/commons/thumb/3/37/Kanchenjunga_India.jpg/1280px-Kanchenjunga_India.jpg');

        let material =  new THREE.MeshBasicMaterial({map: texture, side:
THREE.DoubleSide});

        let geometry = new THREE.SphereGeometry(5, 100, 100);

        sphere = new THREE.Mesh(geometry, material);

        scene.add(sphere);

    };
```

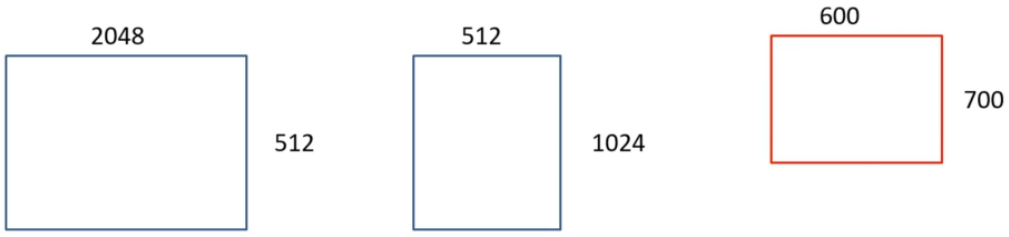
Power of 2 Rule

- Graphic engines usually process resources (like images) in chunks.
- The chunks should fit the physical limitations of the graphic engine and to its algorithm.
- Usually, if we want to render a two-dimensional image correctly we should use the power of 2 rule:

Power of 2 Rule

To ensure a 2D image conforms with the regulated dimensions, its width and height dimensions should be a number that is a whole power of two.

$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$



Shadow

- When a light source lights an object, we expect this object to cast a shadow on the opposite direction.
- Calculating the shadow is a complex task, so the default in Three.js is no shadows.
- We have to explicitly set the appropriate attributes in order to cast shadows.

Casting Shadow

Enabling the renderer

```
renderer.shadowMap.enabled = true;

renderer.shadowMap.type = THREE.PCFShadowMap;
```

Casting Shadow

Enabling the light

Only spot lights and directional lights can cast shadow

We have to explicitly set the light's castShadow property to true:

```
light.castShadow = true;
```

Casting Shadow

LightShadow object

The LightShadow object represents a shadow.

The shadow is viewed by a camera, so this object's constructor accepts a camera object

```
let shadow = new THREE.LightShadow(new THREE.PerspectiveCamera(50, 1, 10, 2500));
```

The LightShadow object has additional properties:

bias – helps to make object look more "grounded"

mapSize.width, mapSize.height – the shadow's dimensions.

Casting Shadow

Enabling the objects

After we've enabled the light to cast shadow, we must enable the objects on the screen to cast or receive shadow.

The geometry's property castShadow when set to true means that this object would cast a shadow

The geometry's property receiveShadow when set to true means that this object would get a shadow.

Note that objects which their properties are not set wouldn't cast or receive shadows.