P.PORTO

POLITÉCNICO
DO PORTO
ESMAD

COMPUTAÇÃO GRÁFICA
TSIW

P.PORTO

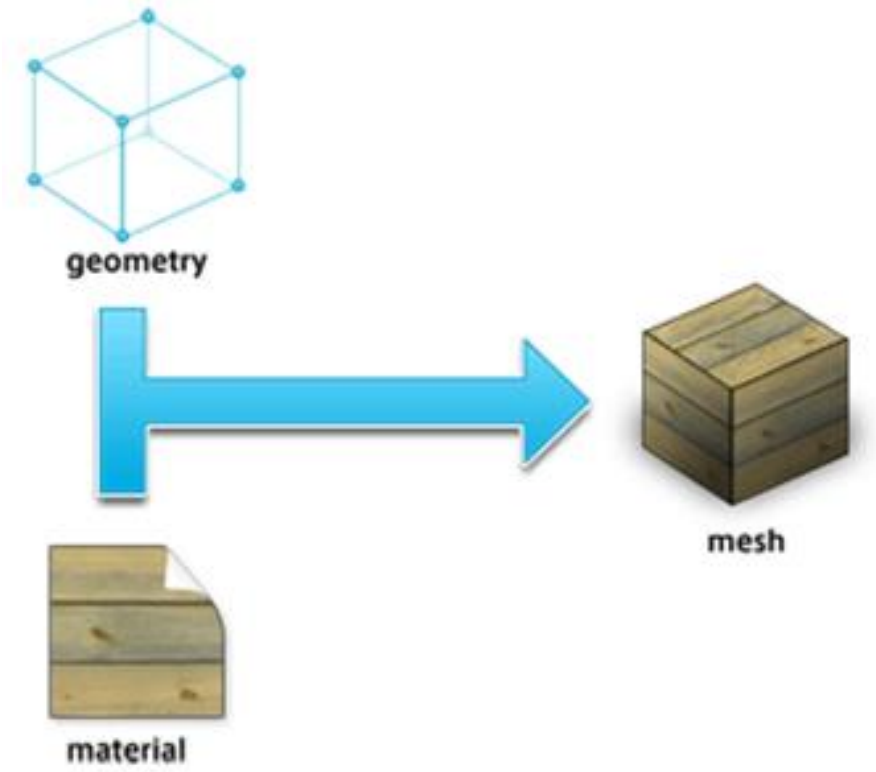
ESCOLA
SUPERIOR
DE MEDIA ARTES
E DESIGN

Syllabus

- Materials
- Textures
- Cameras
- Lights
- Shadows

Materials

- MESH = GEOMETRY + **MATERIAL**
- Acts like a “skin” of the geometry
- Defines the outer aspect of a geometry
  - Metallic?
  - Opaque?
  - Wireframe?
  - …



Materials - Three.js

- Basic properties:

https://threejs.org/docs/#api/materials/Material

  - *color*
  - *opacity*: defines transparency [0-1]
  - *transparent*: if *true*, the object has the transparency defined in ***opacity***
  - *wireframe*: (for mesh materials) renders material as wireframe
  - *side*: defines which side of faces will be rendered - front, back or both; default is `THREE.FrontSide`; other options are `THREE.BackSide` and `THREE.DoubleSide`
  - *flatShading*: defines if the shading effects are per face (*true*) or not (default: *false*)

```
// create a wireframe material
let material = new THREE.MeshBasicMaterial(
                                { color: 0x009e60, wireframe: true });
```

Materials - Three.js

- Set up properties:
  1. At creation time

```
// create a wireframe material
let material = new THREE.MeshBasicMaterial(
                                { color: 0x009e60, wireframe: true });
```

  2. After creation, by accessing to the `Material` class properties

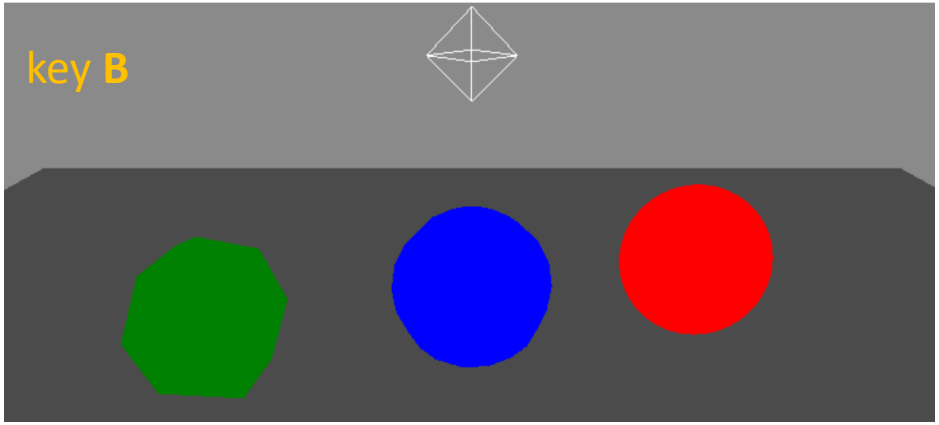
```
// create a wireframe material
let material = new THREE.MeshBasicMaterial();
material.color.setRGB(0, 158, 96);
material.wireframe = true;
```

- Properties of type `THREE.Color` can be set in different ways
  (see examples in https://threejs.org/docs/#api/en/math/Color)

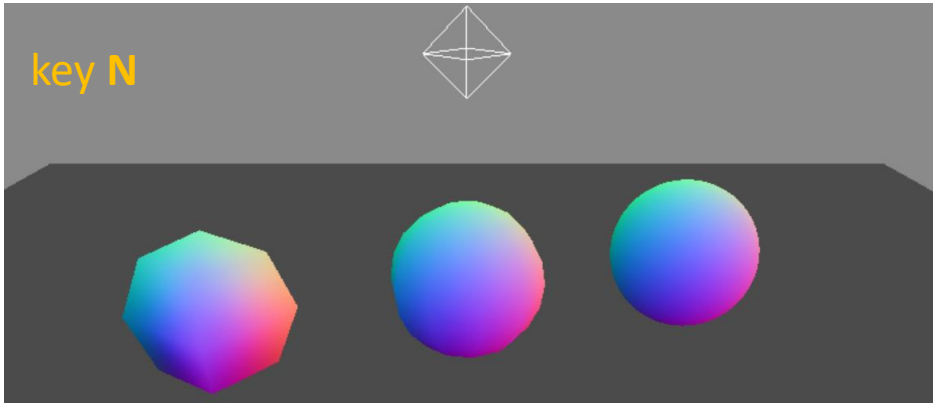
Materials - Three.js

Open the Materials.html example in Moodle:

- **THREE.MeshBasicMaterial**
  - **Not affected by lights** or shadows
  - Objects will have a solid, static color

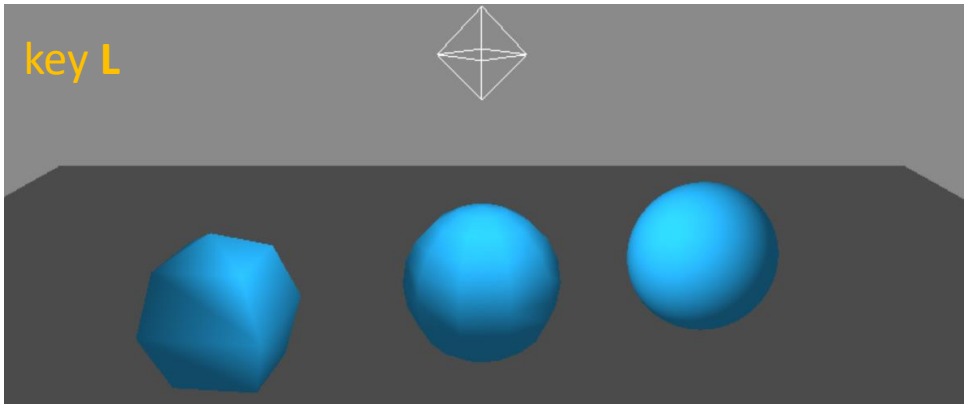


- **THREE.MeshNormalMaterial**
  - **Not affected by lights** or shadows
  - Assigns colors to faces according to their **normal vector**
  - Dynamic colors (change with position)



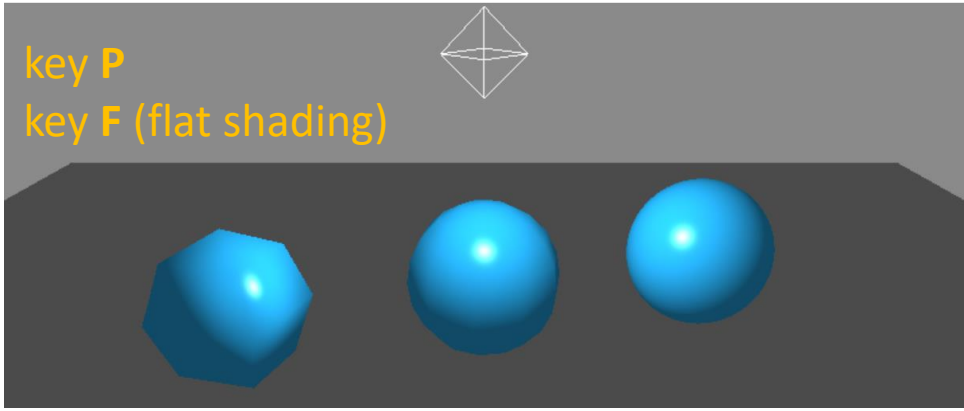
Materials - Three.js

- **THREE.MeshLambertMaterial**
  - It is **affected by lights**
  - Non-shiny surfaces, without specular reflections
  - **Shading calculated by vertex**, using a Gouraud shading model (3 normals per face on each vertex, using interpolation to compute the color via gradients)
  - Properties:
    - *color* - color of the material (color that is shown when it is affected by a light source); default is white
    - *emissive* - solid color emitted by the material (regardless if there is any light or not); default is black



Materials - Three.js

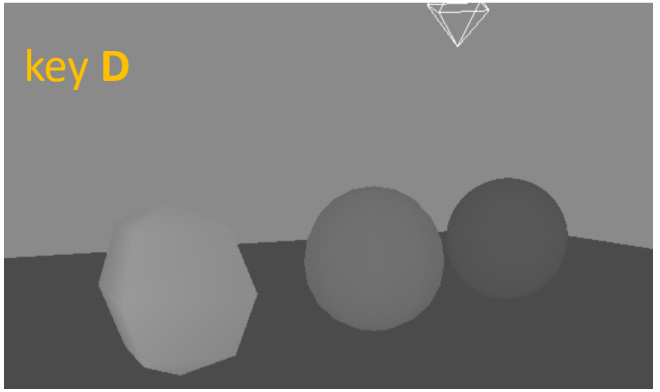
- **THREE. MeshPhongMaterial**
  - It is **affected by lights**
  - Shiny surfaces, with specular highlights
  - **Shading calculated by pixel**, using a Phong model (gives more accurate results than the Gouraud model at the cost of some performance)



  - Properties:
    - *color* - color of the material (color that is shown when it is affected by a light source); default is white
    - *emissive* - solid color emitted by the material (regardless if there is any light or not); default is black
    - *specular* - highlight color; default is 0x111111 (very dark grey)
    - *shininess* - defines how shiny the specular highlight is (default = 30)

Materials - Three.js

- **THREE.MeshDepthMaterial**
  - **Not affected by lights**
  - Color varies from white to black depending on how close the object is to the camera
  - Used for fading effects



- **THREE.MeshStandardMaterial / THREE.MeshPhysicalMaterial**
  - Use a physically correct model for the way in which light interacts with a surface
  - Physically based rendering (PBR) materials: material that reacts 'correctly' under all lighting scenarios
  - Use much more complex math to come close to what happens in the real world

key **S** (standard)

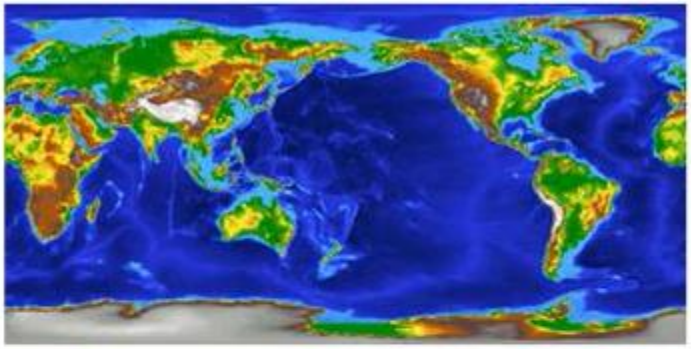
Textures - Three.js

- Fast and easy way to alter the color or overall appearance of an object, using **images**
- Pixel color is obtained from na image file and translated into the object's material **`map`** property
- It is possible also to add other effects,by changing the map to where the image is translated:
  - opacity (**`alphaMap`**)
  - depth (**`displacementMap`**)
  - wrinkles (**`bumpMap`**/**`normalMap`**)
  - glow/reflexes (**`emissiveMap`**/**`lightMap`**)
  - specular effects (**`specularMap`**)
  - …

Textures - Three.js

Object

+



Texture

=

Texture Mapped Object

Textures - Three.js

- Example @ Moodle
  - 3 cube meshes
  - Leftmost cube: color given by **color map** texture
  - Center cube: color + wrinkles simulated by **normal map** texture
  - Rightmost cube: color + wrinkles + vertex displacement given by **displacement map** texture

Textures - Three.js

- Textures are built from **images**
  - `THREE.TextureLoader().load(image_file)`: image upload
  - https://threejs.org/docs/#api/loaders/TextureLoader
  - The texture obtained from the image can be attributed to one of the texture maps of a material, where the color map (most used) is given by property **`map`**

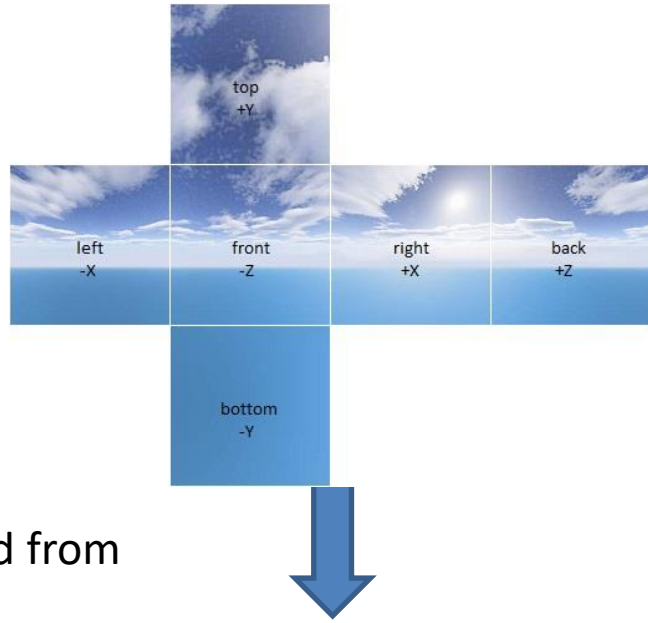
```
// upload image for texture
let texture = new THREE.TextureLoader().load ('texture.png');
// create a textured PHONG material
let material = new THREE.MeshPhongMaterial({ map: texture });
```

- **Asynchronous** method: if there is no animation cycle, it is wise to use a callback function that waits for image upload and texture creation

```
// wait for texture to be loaded
let texture = new THREE.TextureLoader().load ('texture.png',
                function () {  callback();  });
```

Textures - Three.js

- CubeTexture: creates a cube texture made up of six images, one for each cube face
  - Can be used to implement CubeMapping, a method for environment mapping
  - Imagine a large enough cube to involve the 3D scene, **mapped inside** with one image per face
  - The images define a **background** that can be viewed from any point of view (the order is important)




```
let scene = new THREE.Scene();
let textureCube = new THREE.CubeTextureLoader()
                .setPath( 'texturesPath/' )
                .load( [
                        'posX.png','negX.png',
                        'posY.png', 'negY.png',
                        'posZ.png', 'negZ.png'
                ] );
scene.background = textureCube;
```

Textures - Three.js

- Texture mapping properties

Wrapping: **`wrapS`** and **`wrapT`** - define the texture's horizontal and vertical wrapping around the object:
`THREE.ClampToEdgeWrapping`: (default) the last pixel of the texture stretches to the edge of the mesh
`THREE.RepeatWrapping`: texture will simply repeat by the number of times defined in `repeat` property – tiling effect

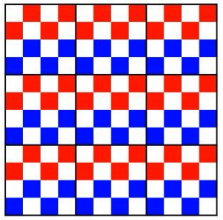
```
let texture = new THREE.TextureLoader().load ('texture.png');
// the texture will repeat itself 3x horizontally and 3x vertically
texture.wrapS = THREE.RepeatWrapping;
texture.wrapT = THREE.RepeatWrapping;
texture.repeat.set( 3, 3 );
// create textured material
let material = new THREE.MeshPhongMaterial({ map: texture });
```

`THREE.MirroredRepeatWrapping`: texture repeats, by the number of times defined in `repeat` property, mirroring on each repeat

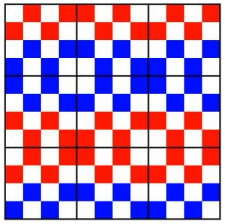
Textures - Three.js

- Texture mapping properties
  - Wrapping:
    `THREE.ClampToEdgeWrapping`

`THREE.RepeatWrapping`



`THREE.MirroredRepeatWrapping`



Exercises - Textures

Exercise 1

- Camera: perspective (FOV = 45º),
  Z position = 3
- Sunlight: directional, position (5, 5, 5)
- Earth:
  - Sphere geometry (radius = 1)
  - Phong material (no color, with color texture – property **map** – given by the image `no_clouds_4k.jpg` available in Moodle)
  - Animation: Y rotation of +0.005 radians per frame

Exercises - Textures

- The ***bump mapping*** technique simulates (https://en.wikipedia.org/wiki/Bump_mapping) wrinkles and imperfections; apply it to simulate the 3D effect of the Earth's surface
  - Material property **`bumpMap`** (https://threejs.org/docs/#api/materials/MeshPhongMaterial.bumpMap)
  - Image `elev_bump_4k.jpg` (black and white image, whose intensity indicates the terrain altitude)
  - increase the **`bumpScale`** property of the material to visualize the bump mapping effect (default value = 1)

Exercises - Textures

- The clouds are missing on our planet!
- Create another sphere, with a radius slightly higher than the previous one (e.g. 1.003)
- Use image `fair_clouds_4k.png` to map its texture
- Turn on the *`transparent`* property of the material (since the image has transparent background, **only** the clouds will appear)
- Do not forget to also rotate the cloud mesh!

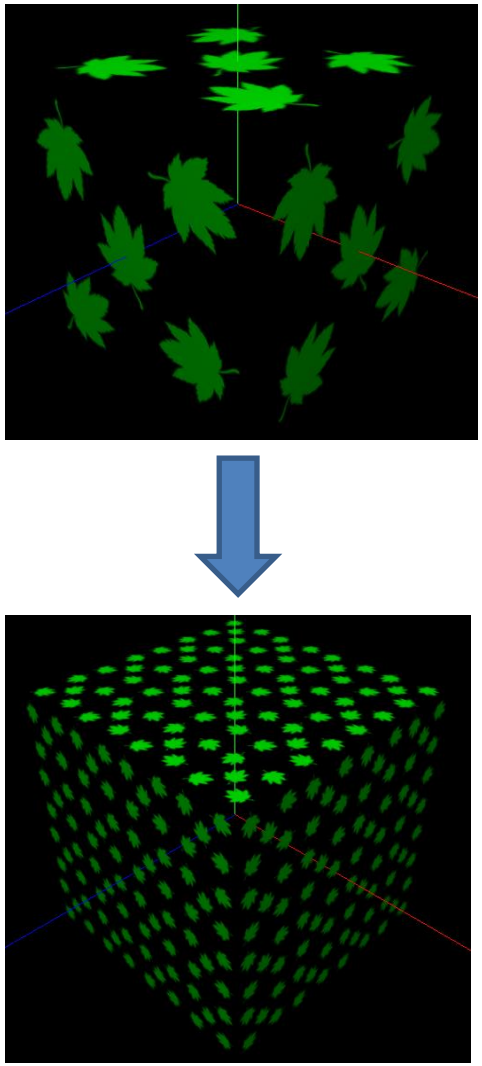
Exercises - Textures

- Now all you need is the stars!
- Add a third sphere, with a radius much much greater than the previous ones (e.g. 1000)
- Use image *galaxy_starfield.png* to map its texture into a basic material (no color)
- Alter material property `side` to `THREE.BackSide` so that the texture is mapped into the **inside** of the sphere
- Do not rotate the starfield!

Exercises - Textures

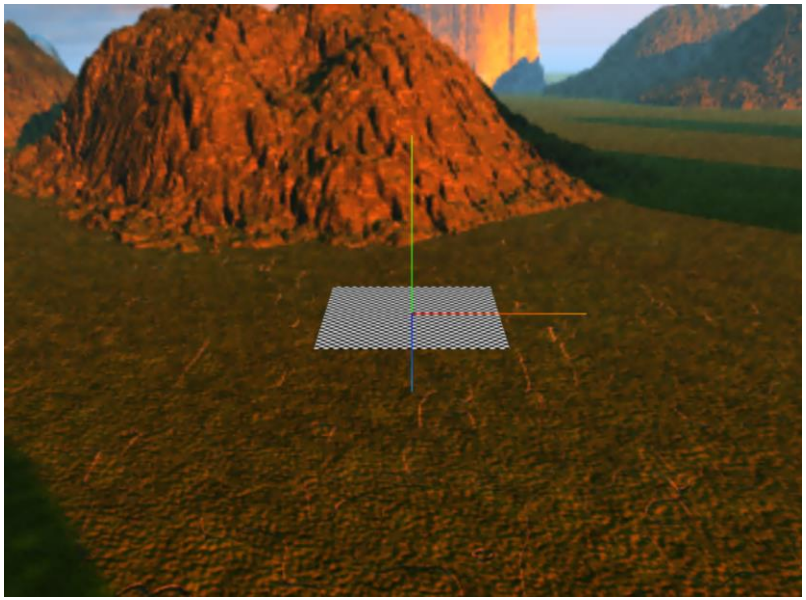
Exercise 2

- Camera: perspective (FOV = 45º), position (15,15,15) lookAt(0,0,0) – can you figure out why?
- Lighten the scene with a white `THREE.SpotLight` at position (25,50,30)
- Create a cube (size 10) with a transparent Phong material and green color
- Use image `partial-transparency.png` for cube's texture; repeat the texture 4 times, both horizontally and vertically
- **Animate** the texture by increasing its horizontal and vertical `offset` property values (0.01 units/frame), both horizontally and vertically (`offset` is a variable of type `THREE.Vector2D`)



Exercises - Textures

Exercise 3



- Place the camera at (0, 100, 200), and make it look down to the scene center position (perspective, FOV = 75º)
- Create a 100x100 plane with a texture given by the repetition of image `checkerboard.jpg` (10 times in X and Y)
- Use images *`dawnmountain-….jpg`* to create a Cube Texture and assign it to the scene background
- Render the scene and use the OrbitControls to move around the camera

P.PORTO

ESCOLA SUPERIOR DE MEDIA ARTES E DESIGN

Cameras - Three.js

`THREE.Camera`: abstract base class for cameras

`THREE.PerspectiveCamera`

– uses **perspective projection**, designed to mimic the way the human eye sees; it is the most common projection mode used for rendering a 3D scene
https://threejs.org/docs/#api/cameras/PerspectiveCamera

`THREE.OrthographicCamera`

– uses **orthographic projection**, where an object's size in the rendered image stays constant regardless of its distance from the camera
https://threejs.org/docs/#api/cameras/OrthographicCamera

`THREE.CubeCamera`

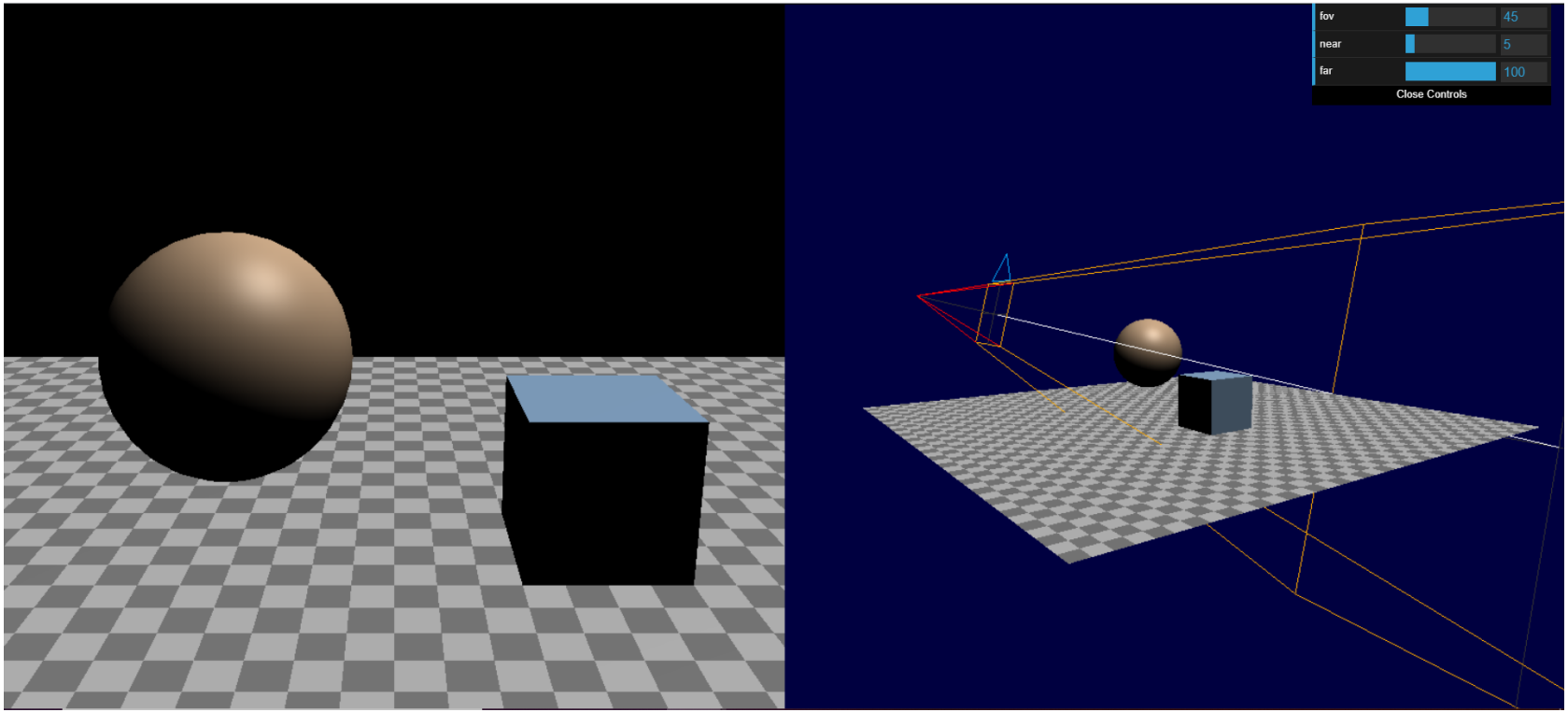
– creates **6 perspective cameras that render to a target cube** (used for reflections and refractions)
https://threejs.org/docs/#api/cameras/CubeCamera

P.PORTO

ESCOLA SUPERIOR DE MEDIA ARTES E DESIGN

Cameras - Three.js

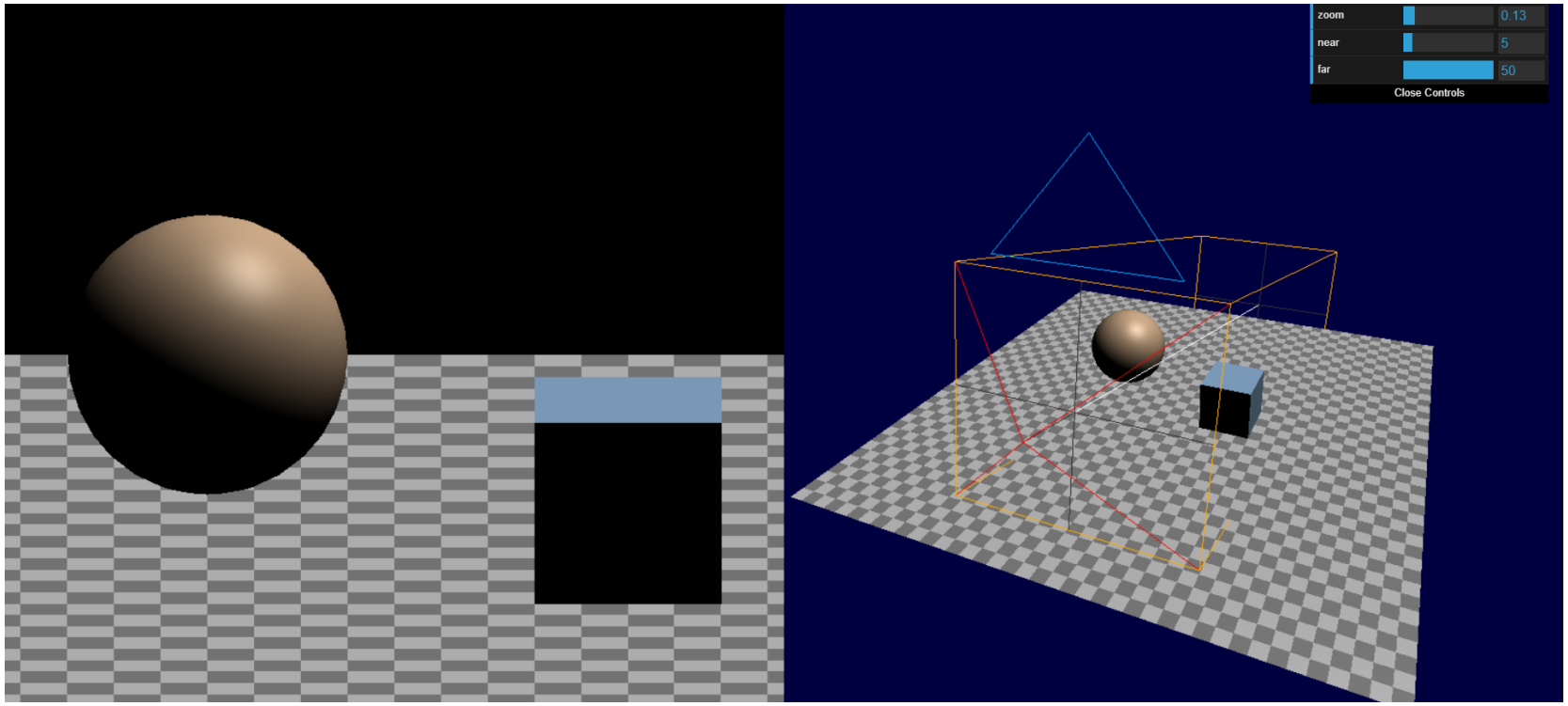
`THREE.PerspectiveCamera`



Example: https://threejsfundamentals.org/threejs/threejs-cameras-perspective-2-scenes.html

Cameras - Three.js

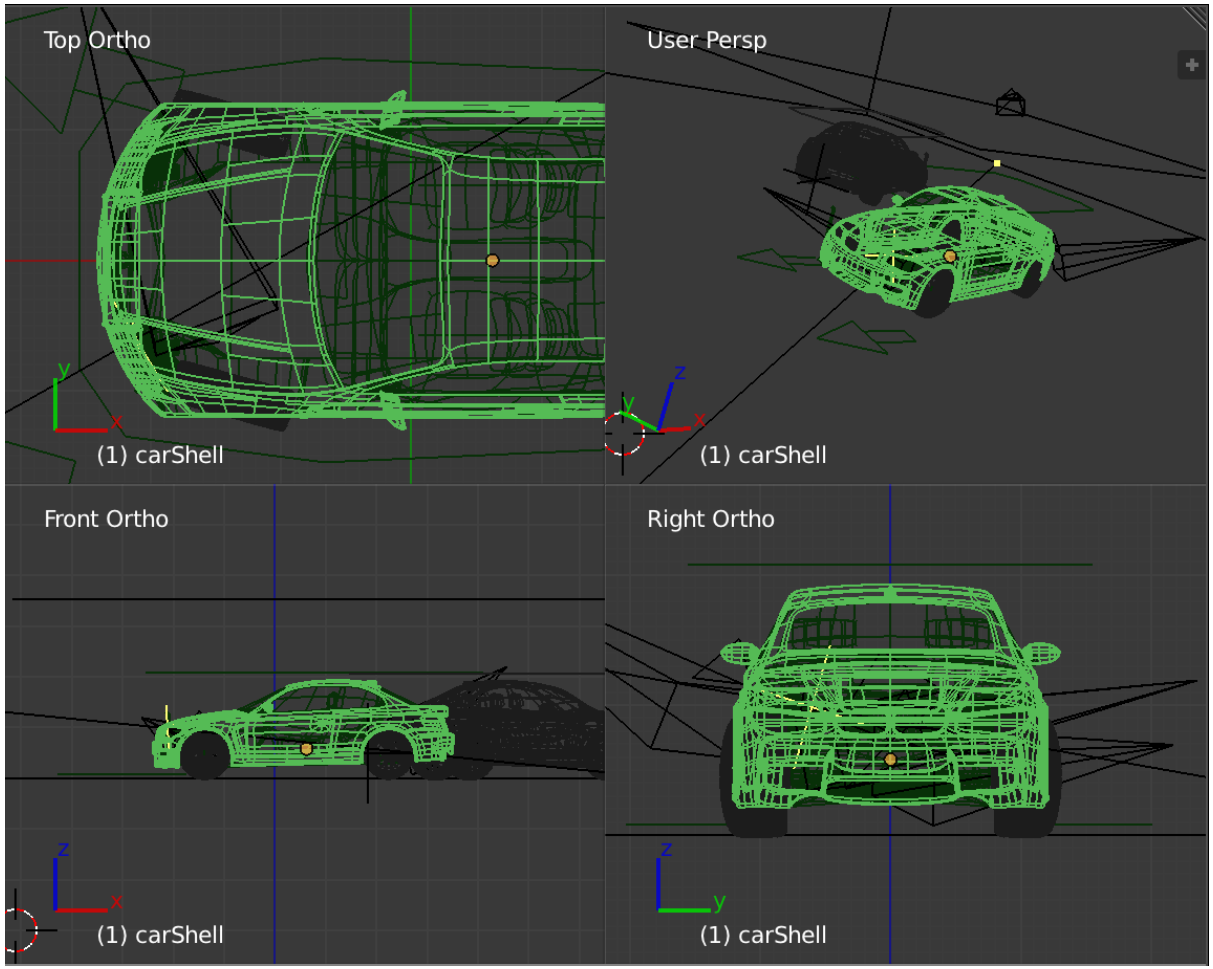
`THREE.OrthographicCamera`



Example: https://threejsfundamentals.org/threejs/threejs-cameras-orthographic-2-scenes.html

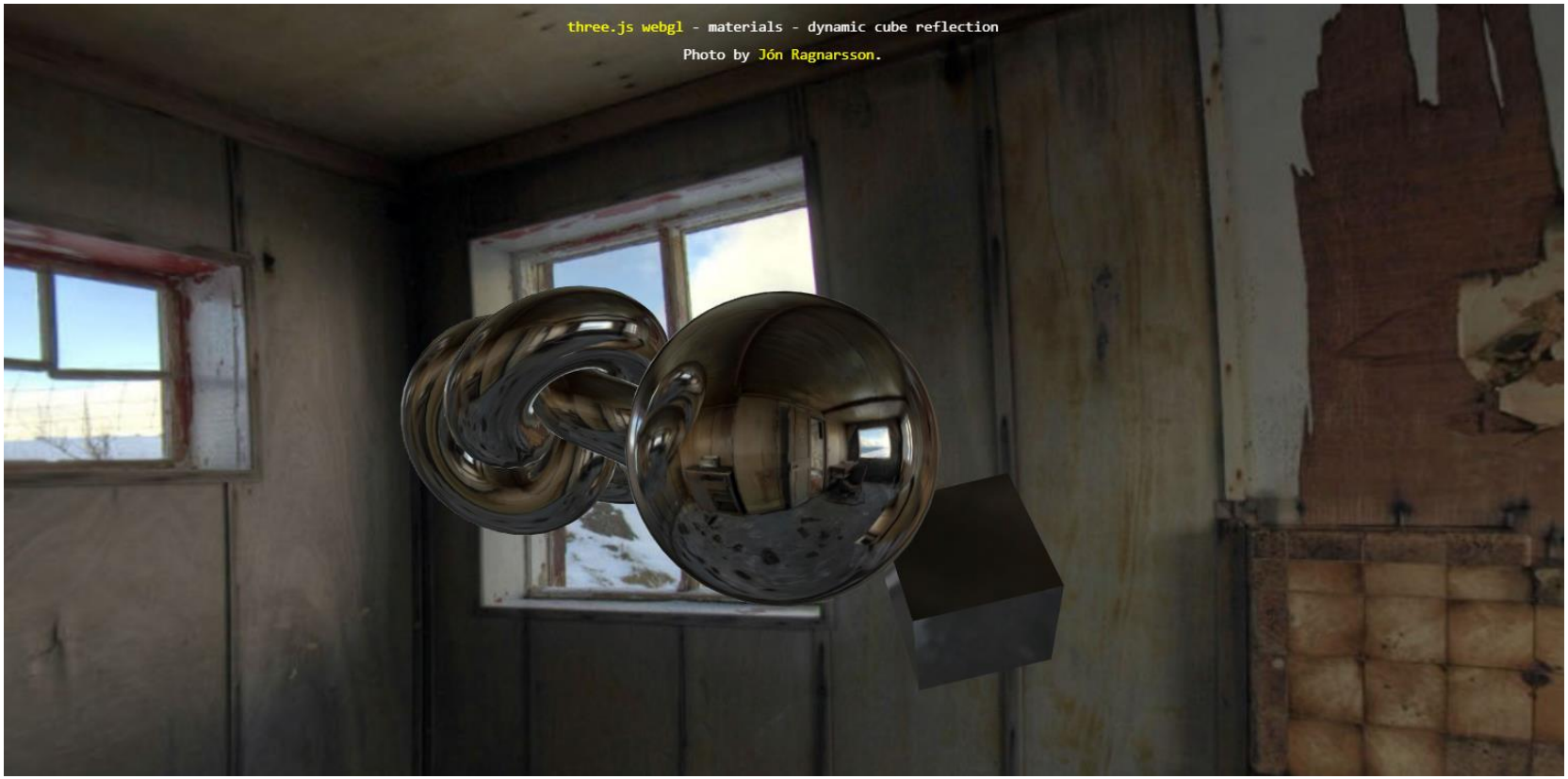
Cameras - Three.js

Perspective vs Orthographic



Cameras - Three.js

- `THREE.CubeCamera`: used for environment mapping (cheap way to create reflections on curved surfaces)



Example: https://threejs.org/examples/#webgl_materials_cubemap_dynamic

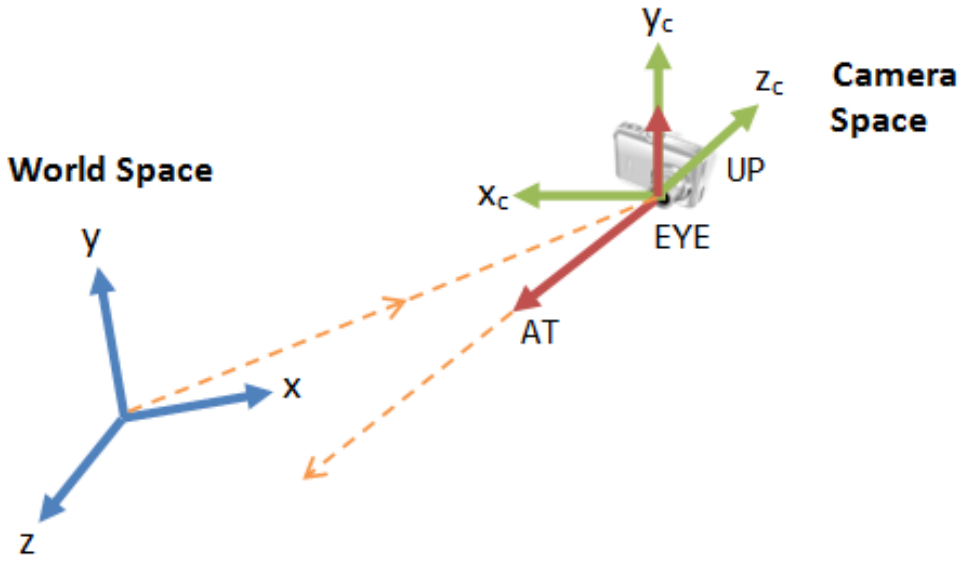
Cameras - Three.js

- **Camera parameters**
  – constructor sets the **projective transformation** parameters
  – the **view transformation** is set up using:

`position(Vector3)` sets the **camera's eye**; default: `(0,0,0)`

`lookAt(Vector3)` rotates the camera **to face a point** in world space; default: `(0,0,0)`

`up(Vector3)` determines the **upward orientation** of the camera; default: `(0,1,0)`



Cameras - Three.js

- **Camera helpers**
  - https://threejs.org/docs/?q=helper#api/en/helpers/CameraHelper
  - helps with visualizing the camera visualization volume (frustum)
  - works with both perspective and orthographic cameras

```
const camera = new THREE.PerspectiveCamera(
                 75, window.innerWidth / window.innerHeight, 0.1, 1000 );
const helper = new THREE.CameraHelper( camera );
scene.add( helper );
```

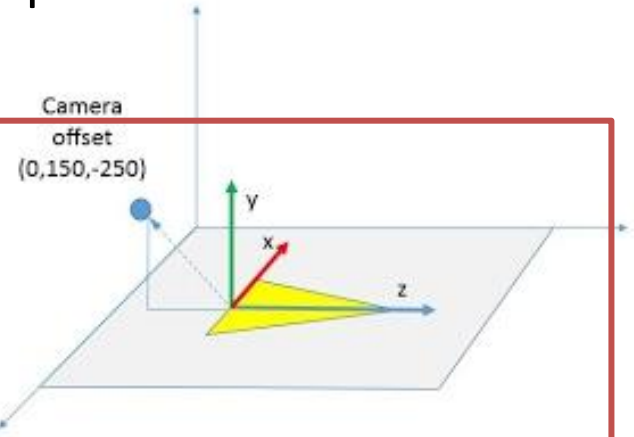
Cameras – advanced techniques

- **Camera following an object** – use the `lookAt` method to update the camera orientation (camera's position is not altered)

```
// some object and camera
let object, camera;

(…)

// animation loop
function render(){
        // camera looks at the object's position
        camera.lookAt(object.position);

        renderer.render(scene, camera);
    }
```

Cameras – advanced techniques

- "*Chase cam*" or "**Third person view**" – camera placed behind the object (always following him)




```
// some object and camera
let object, camera;
(…)
// animation loop
function render(){
        // camera TO object relative offset
        let relativeOffset = new THREE.Vector3(X, Y, Z);
        // updates the offset with the object's global transformation matrix
        let cameraOffset = relativeOffset.applyMatrix4(object.matrixWorld);
        // updates the camera position with the new offset
        camera.position.copy(cameraOffset);
        // camera looks at the object's position
        camera.lookAt(object.position);

        renderer.render(scene, camera);
}
```

CLICK FOR EXAMPLE

Cameras – advanced techniques

- **Multiple cameras** – add several cameras to the scene; camera choice is performed **on rendering**

```
// two different cameras
let camera1, camera2;
// some boolean flag
let camera1isActive = true;

(…)

// animation loop
function render(){
        if (camera1isActive)
                renderer.render(scene, camera1);
        else
                 renderer.render(scene, camera2);
}
```

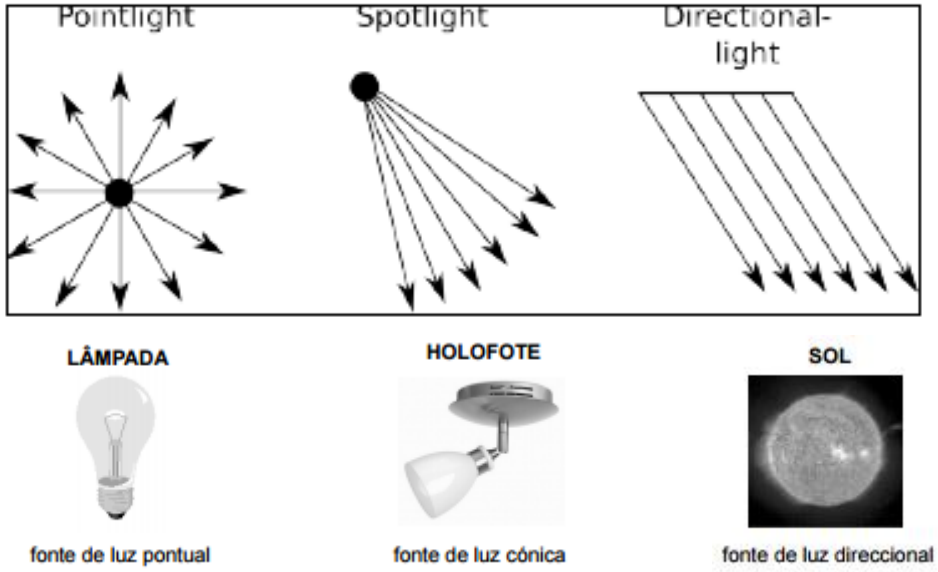
CLICK FOR EXAMPLE

Lights

- Lights:
  - Without lighting it may not be possible to see a scene (depends on the **object's material**)
  - More realistic animations
  - Helps on depth perception
- How an object looks depends on:
  - Properties of the **light source**, such as: color, distance between light source and object, direction defined by light source and object, intensity of light source
  - Characteristics of the object's **surface**, such as: color and reflection properties
  - Observer's location
- Light that strikes the surface of an object can be reflected, absorbed, transmitted or a combination of them

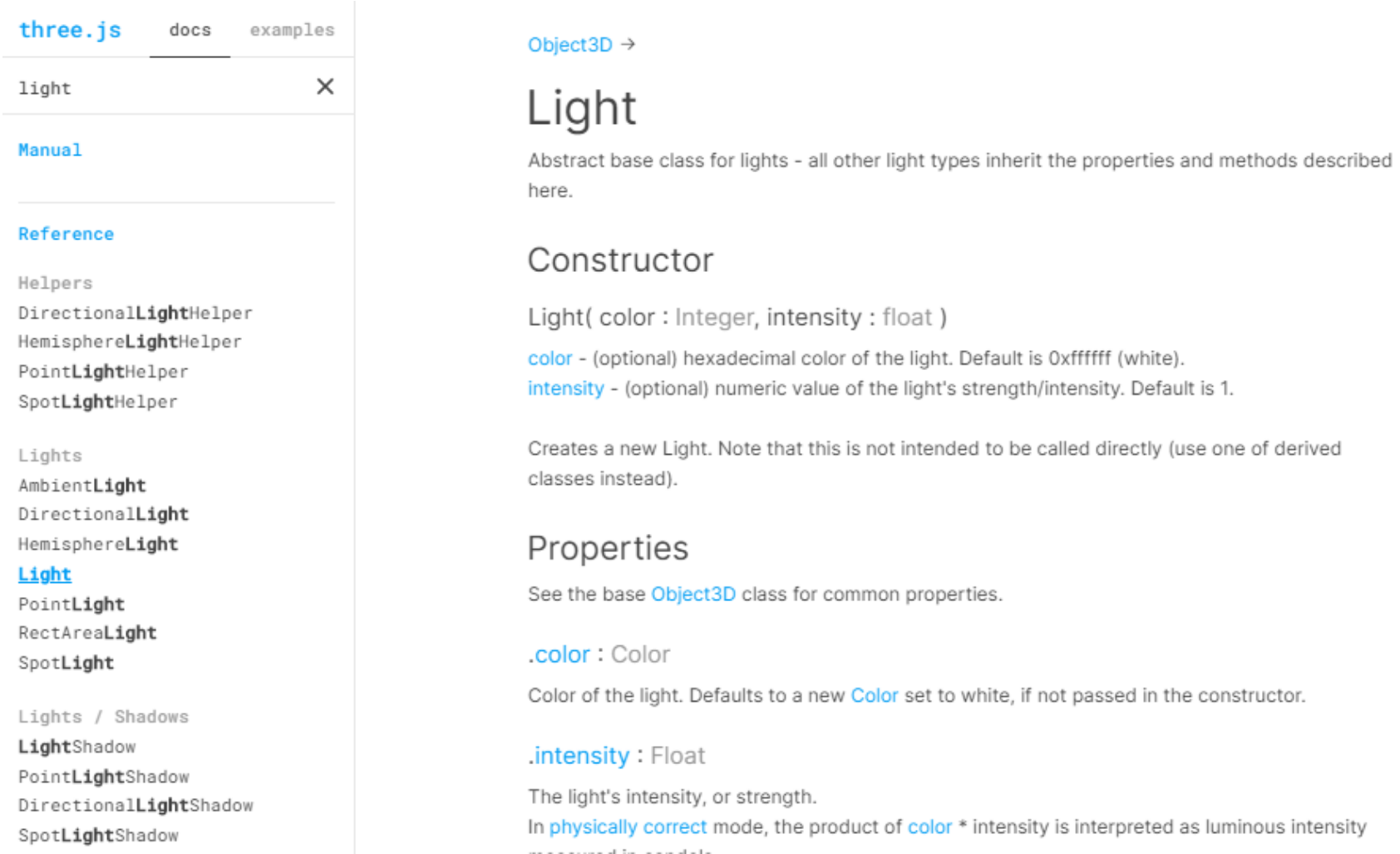
Lights

- In WebGL, the lighting of a scene requires its configuration in the shaders!
- Three.js has a number of light features that let you illuminate the scene and its objects
- Three.js **basic lighting types**:
  - `THREE.AmbientLight`
  - `THREE.PointLight`
  - `THREE.SpotLight`
  - `THREE.DirectionalLight`



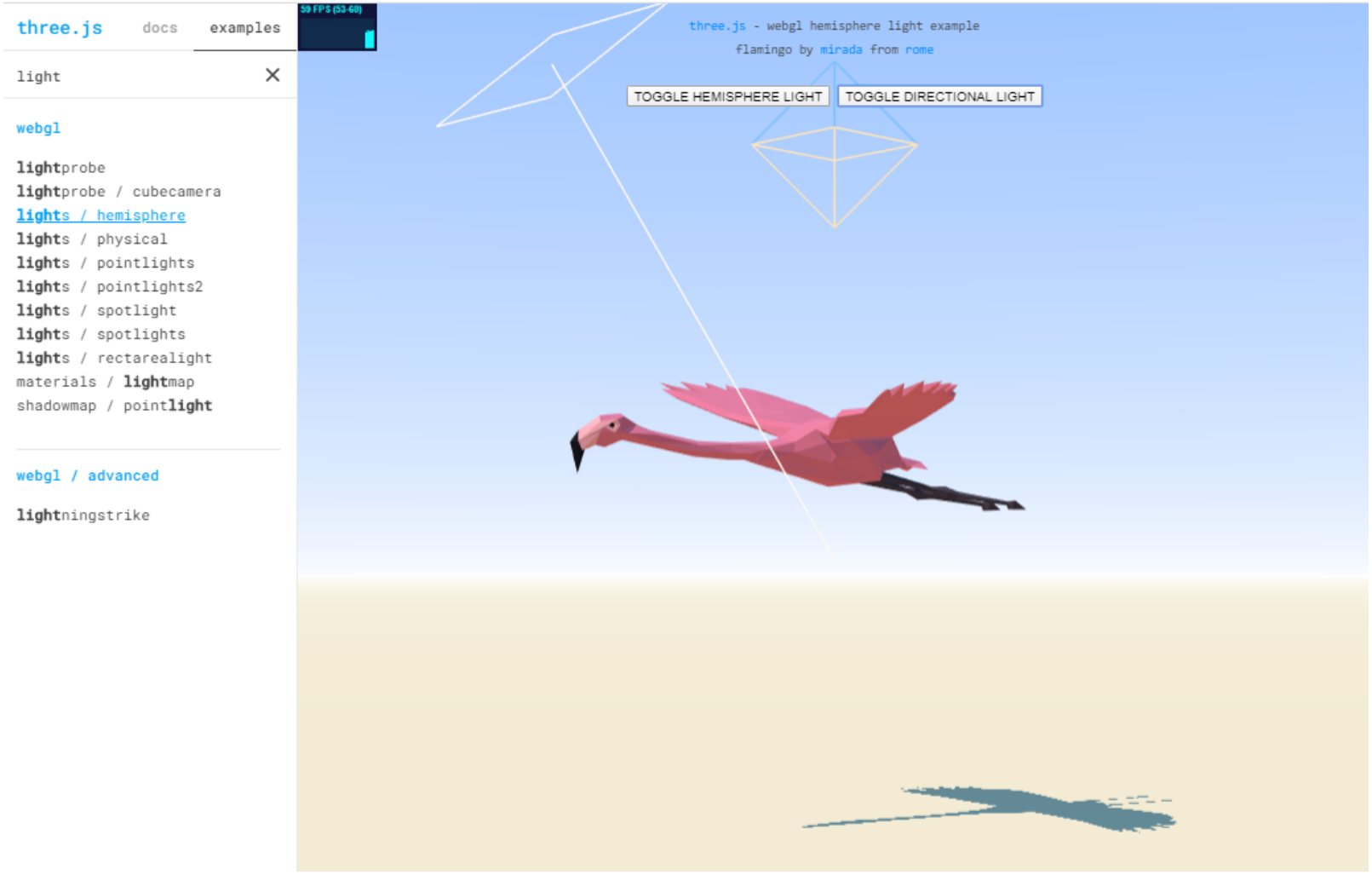
Lights

- Three.js documentation



Lights

- Three.js examples



Lights

- Three.js Object3D **LIGHT**:
  - https://threejs.org/docs/index.html#api/lights/Light
  - Abstract base class for lights (all other light types inherit the properties and methods described here)
- Constructor `THREE.Light([color],[intensity])`:
  - `color`: hexadecimal color of the light (default: `0xffffff` - white) - optional
  - `intensity`: numeric value of the light's strength/intensity (default: `1.0`) – optional
- Like all 3D objects, must be added to the scene

```
scene.add(light);
```

Lights - *AmbientLight*

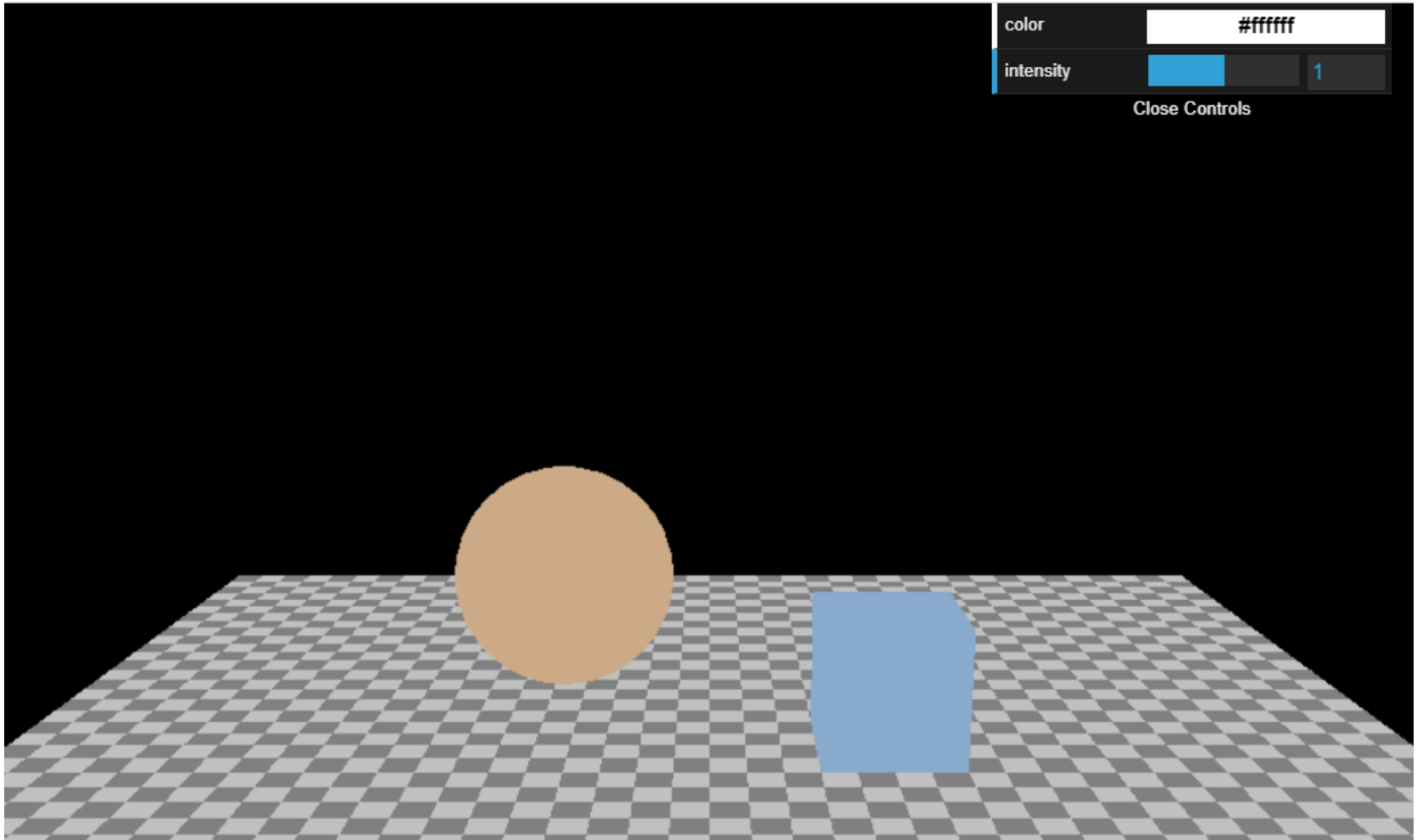
- It does not come from any particular light source
- Globally illuminates all objects in the scene equally
- Does not have a direction
- **Cannot be used to cast shadows**
- `THREE.AmbientLight([color],[intensity])`:

```
// soft white light
let light = new THREE.AmbientLight( 0x404040 );

// add light to the scene
scene.add( light );
```

- Documentation: https://threejs.org/docs/#api/lights/AmbientLight

Lights - *AmbientLight*

- Example

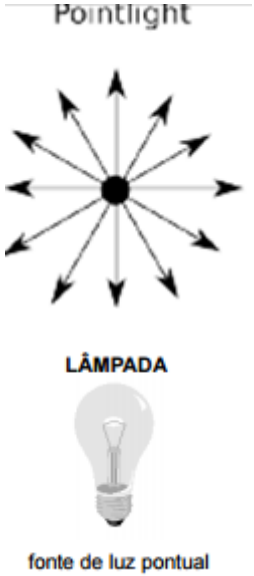


Lights - *AmbientLight*

- Notice in the previous example that there is no shape definition and the light as no direction
  - Ambient light effectively just multiplies the material's color by the light's color times the intensity
  - color = materialColor * light.color * light.intensity
- This lighting is often **used in conjunction with others** to soften shadows or add an additional color to the scene!

Lights - *PointLight*

- `THREE.PointLight`: light that gets emitted from a single point in a radial form in all directions
  - Light intensity attenuates with the distance to the light source
  - **Can cast shadows**
  - Documentation: https://threejs.org/docs/index.html#api/lights/PointLight



- Constructor parameters
  - `color`
  - `intensity`
  - `distance`: maximum range of the light (default: 0.0 = infinite light – no limit)
  - `decay`: the amount the light dims along the distance of the light (default: 1.0 / for physically correct light falloff: 2.0)

Lights - *PointLight*

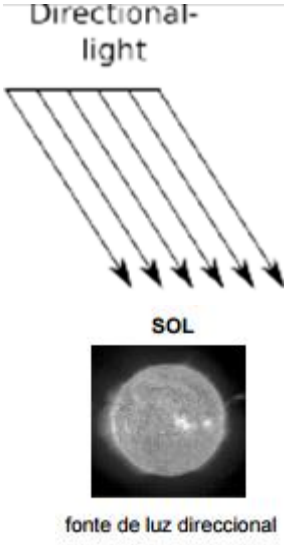
- Properties (inherited from class Object3D)
  - `position`: light source position on the scene
  - `visible`: allows to turn on (*true*) or turn off the light (*false*)

```
// light  blue
let pointLight = new THREE.PointLight("#ccffcc");
pointLight.position.set(10,10,10);

// add light to the scene
scene.add(pointLight);
```

- **Helper**: `THREE.PointLightHelper(light,size)`:
  - helper object consisting of a diamond mesh for visualizing a PointLight position
- Example (alter the parameters – including the distance – and observe)

Lights - *DirectionalLight*

- `THREE.DirectionalLight`: models a point light source at infinity (ex: the sun) - the rays produced from it are all parallel
  - Light intensity does not attenuates with the distance to the light source
  - **Can cast shadows**
  - All objects receive the same amount of light
  - Common use: simulate daylight
  - Documentation: https://threejs.org/docs/index.html#api/lights/DirectionalLight



fonte de luz direccional

- Constructor parameters
  - `color`
  - `intensity`

Lights - *DirectionalLight*

- Important properties
  - `position`: light source position on the scene
  - `visible`: allows to turn on (*true*) or turn off the light (*false*)
  - `target`: object where the light source points to (`target.position` default is `(0,0,0)`)

```
light.target.position.set(X,Y,Z);
```

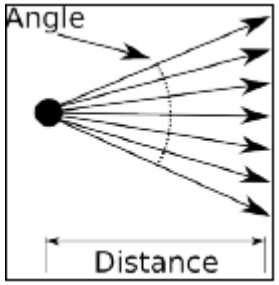
OR

```
light.target = targetObject;
```

- Light ray direction is calculated as pointing from the light's position to the target's position
- **Helper**: `THREE.DirectionalLightHelper(light,size,color)`
  - helper object consisting of a plane and a line representing the light's position and direction, respectively
- Example (alter the parameters and observe)

Lights - *SpotLight*

- `THREE.SpotLight`: light emitted in a cone shape
  - Light intensity attenuates with the distance to the light source
  - **Can cast shadows**
  - Documentation: https://threejs.org/docs/index.html#api/lights/SpotLight
- Constructor parameters (most important)
  - `color`
  - `intensity`
  - `distance`: maximum distance from origin where light will shine (when zero, there is no limit)
  - `angle`: maximum angle of light dispersion from its direction (upper bound is Math.PI/2)
  - `penumbra`: % of the cone that is attenuated (default 0: no attenuation)



Lights - *SpotLight*

- Important properties
  - `position`: light source position on the scene
  - `visible`: allows to turn on (*true*) or turn off the light (*false*)
  - `target`: object where the light source points to (`target.position` default is `(0,0,0)`)

```
light.target.position.set(X,Y,Z);
```

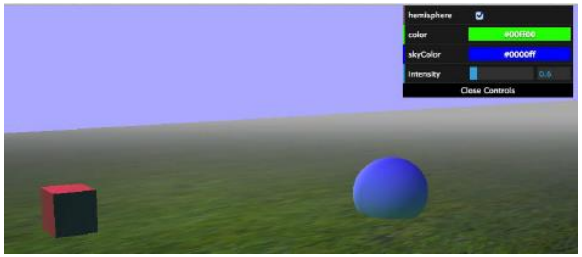
OR

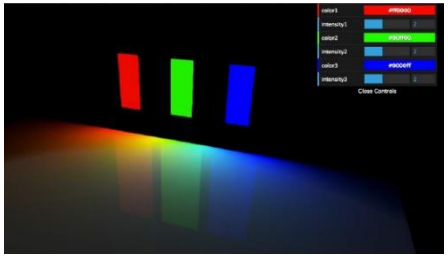
```
light.target = targetObject;
```

- Light ray direction is calculated as pointing from the light's position to the target's position
- **Helper**: `THREE.SpotLightHelper(light,color)`:
  - helper object consisting of a cone shaped helper object
- Example

Other lights

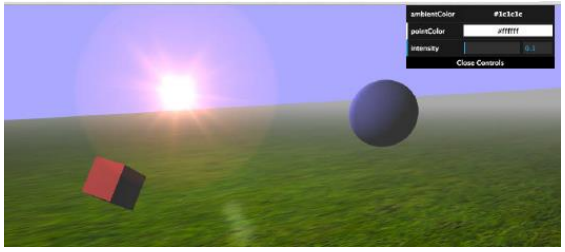
- Three.js implements other more complex illuminations:
  - **`THREE.HemisphereLight`**: allows the recreation of natural light in outdoor scenes
    Consists of a light source positioned directly above the scene, with color fading from the sky color to the ground color
    https://threejs.org/examples/#webgl_shaders_ocean





  - **`THREE.RectAreaLight`**: creates rectangular areas that emit light (instead of a single point of light)
    https://threejs.org/examples/#webgl_lights_rectarealight

  - **`LensFlare`**: simulates lens light reflections
    https://threejs.org/examples/#webgl_lensflares



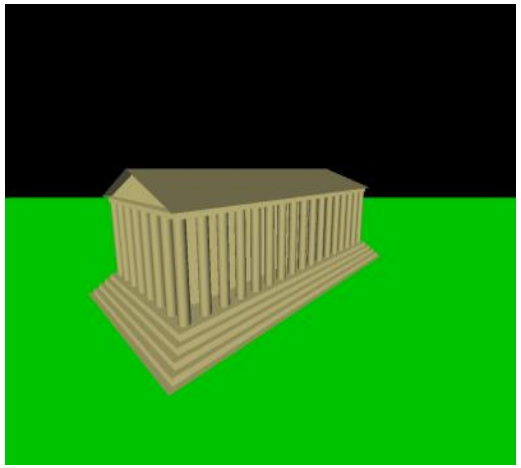
Lights – advanced techniques

- Moving lights:
  - **A. Alter the source light position**
  - change property `position` of the light source
  - **B. Light follows an object**
  - change property `target` of the light source
  - **C. Moving the light source along a path**
  1. create a path, e.g. of type `THREE.Curve` (`THREE.ArcCurve`, `THREE.EllipseCurve`, `THREE.SplineCurve`,…)
  2. equals property `position` of the light source to the points along the path (use method `getPointAt` of paths) https://threejs.org/docs/#api/en/extras/core/Curve.getPointAt

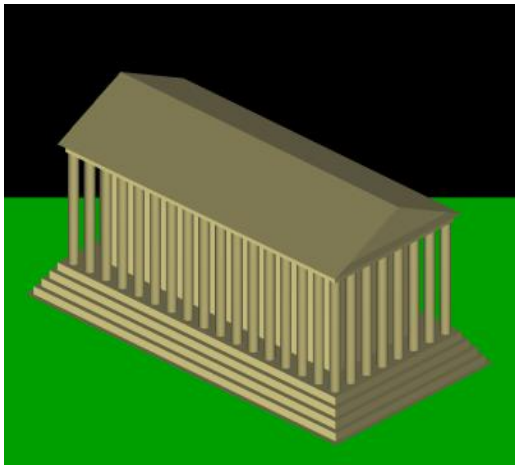
Shadows

- Some lights project **shadows in the scene**, which change depending on the position of the light source
- By default, shadows are disabled because they consume enough computational resources
  - Three.js uses **shadow maps** (see next slides): for every light that cast shadows, all objects marked to cast shadows are rendered from the point of view of the light
  - Meaning: if you have 20 objects casting shadows, and 5 lights casting shadows, then your entire scene will be drawn 6 times: all 20 objects will be drawn for light #1, …, light #5, and finally the actual scene will be drawn using data from the first 5 renders
  - It gets worse, if you have a point light casting shadows the scene must be drawn 6 times just for that light (3 axis x 2 directions each)!
- Usually, only one of the lights in a scene generates shadows

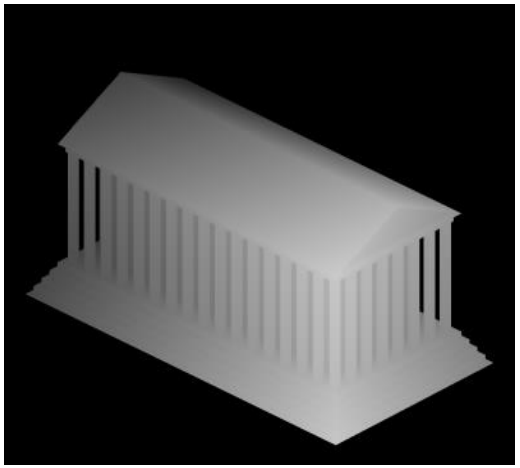
Shadow mapping



Scene with no shadows

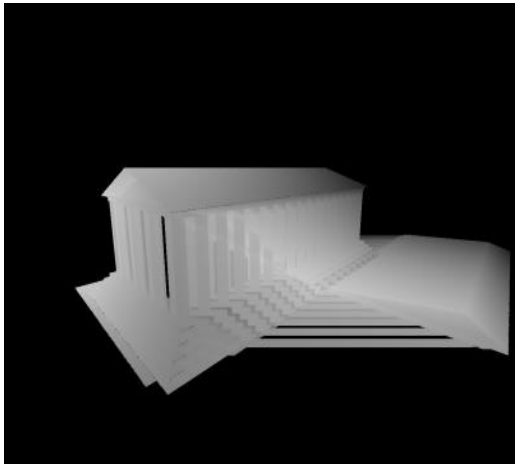


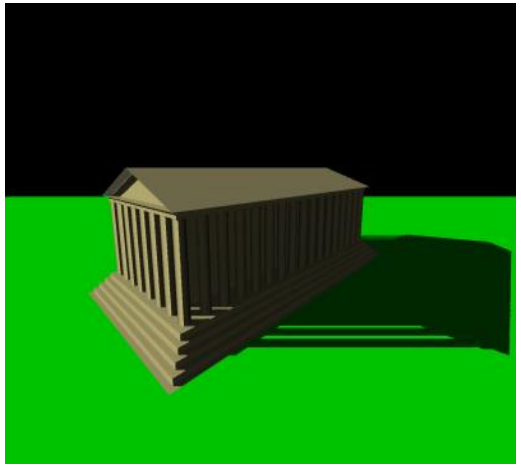
Scene rendered from the light source



Depth map from the light view

Depth map projected onto the scene

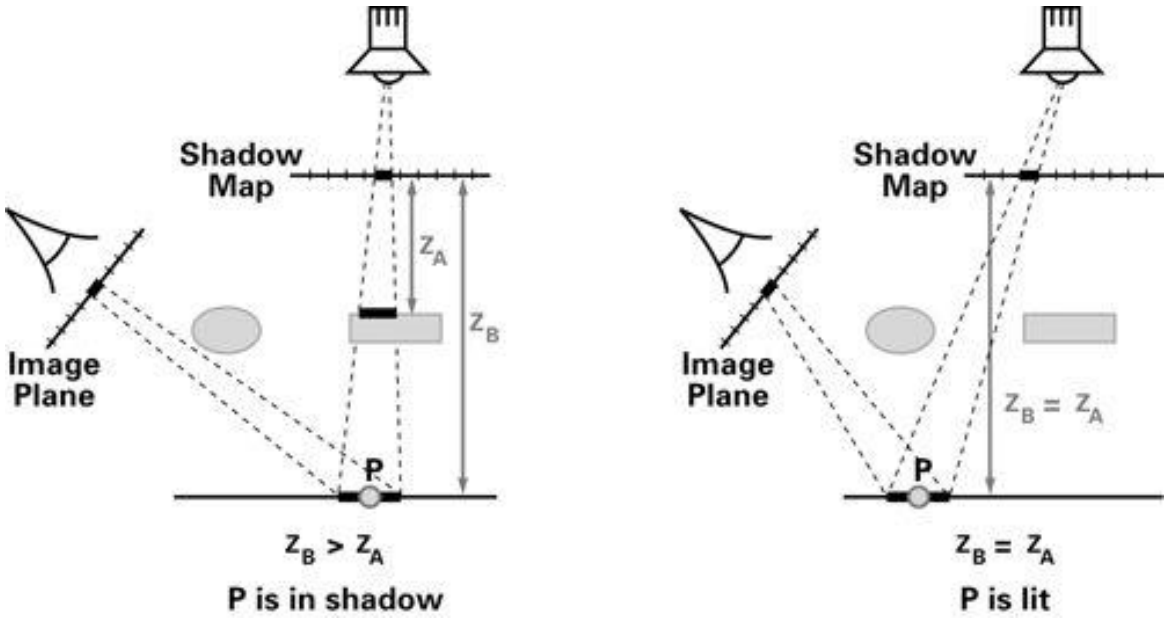




Scene with shadow mapping

Shadow mapping

- Shadow map (only applicable for spot and directional light – x6 for point light)



  - Shadows are created by testing when a pixel is visible from a light point of view, and comparing with the z-buffer or depth map, saved as a texture; this texture, unlike an image that saves colour, it stores pixel distance to the light source

Shadows

- How to setup shadows in THREE.js:

1. Set up the RENDERER:
   - Enable property `renderer.shadowMap.enabled = true`
   - (optionally) Smooth produced shadows `renderer.shadowMap.type = THREE.PCFSoftShadowMap`
2. Tune your LIGHTS:
   - Enable property `light.castShadow = true` of at least one light that can cast shadows
   - If we want the light source to only create shadows and not add any lighting to the scene, turn on the property `light.onlyShadow = true`

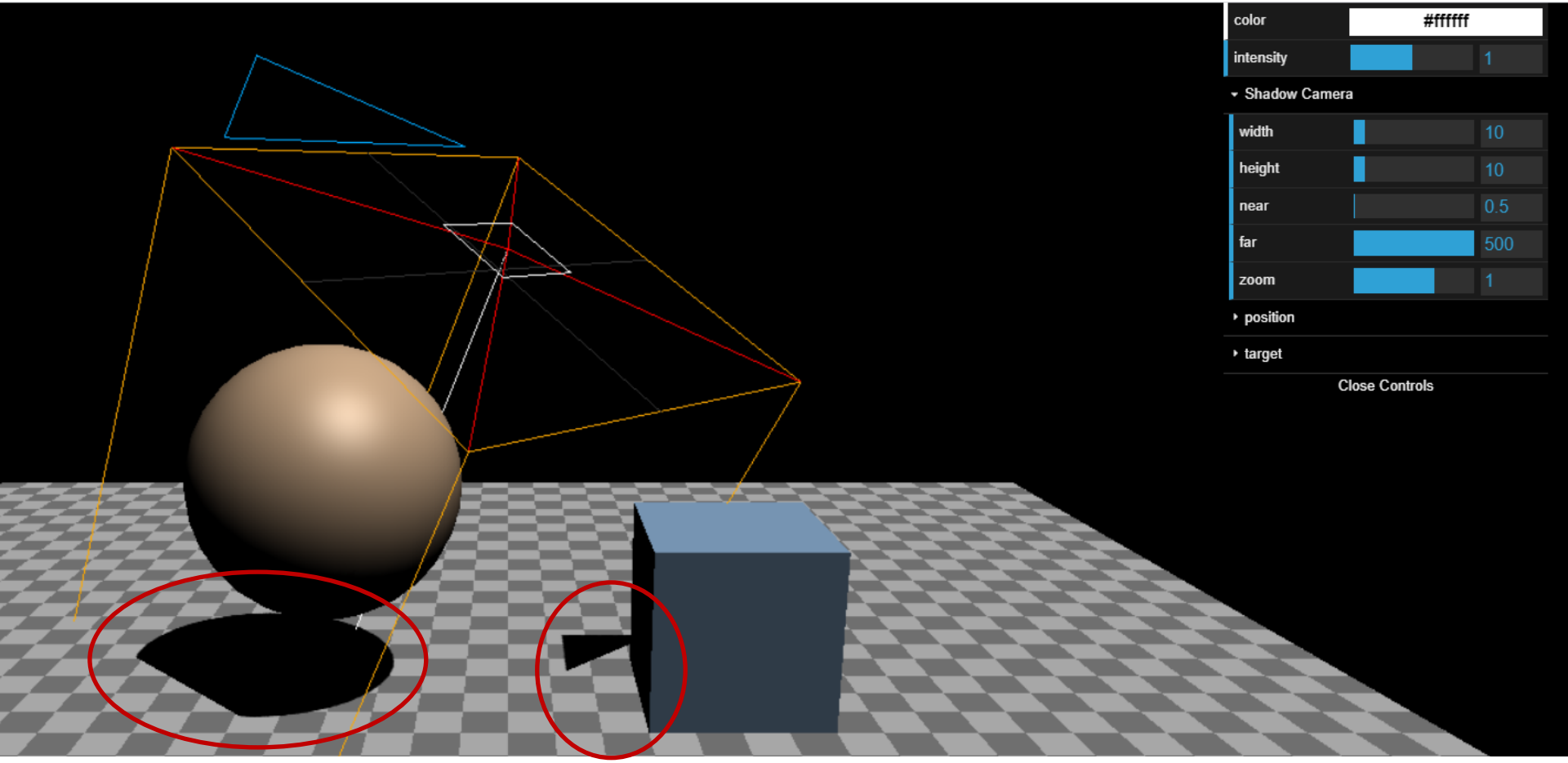
Shadows

3. Configure your OBJECTS of type MESH:
   - Set `mesh.castShadow = true` for the object to **cast** shadows on other objects
   - Set `mesh.receiveShadow = true` for the object to **receive** shadows from other objects

- If shadows have a “pixel effect”, it means that the shadow map has lower resolution
  - Increase property values `light.shadow.mapSize.width` and `light.shadow.mapSize.height` of the light source (those values must be powers of 2)
- Lights have a **camera** property used to generate the depth map of the scene; objects outside the `light.shadow.camera` frustum will not be affected by shadows
  - Directional lights create shadows with orthographic cameras and spotlights create shadows with perspective cameras, pointlights create shadows using 6 perspective cameras

Shadows

- Directional light example (light shadow camera is orthographic)

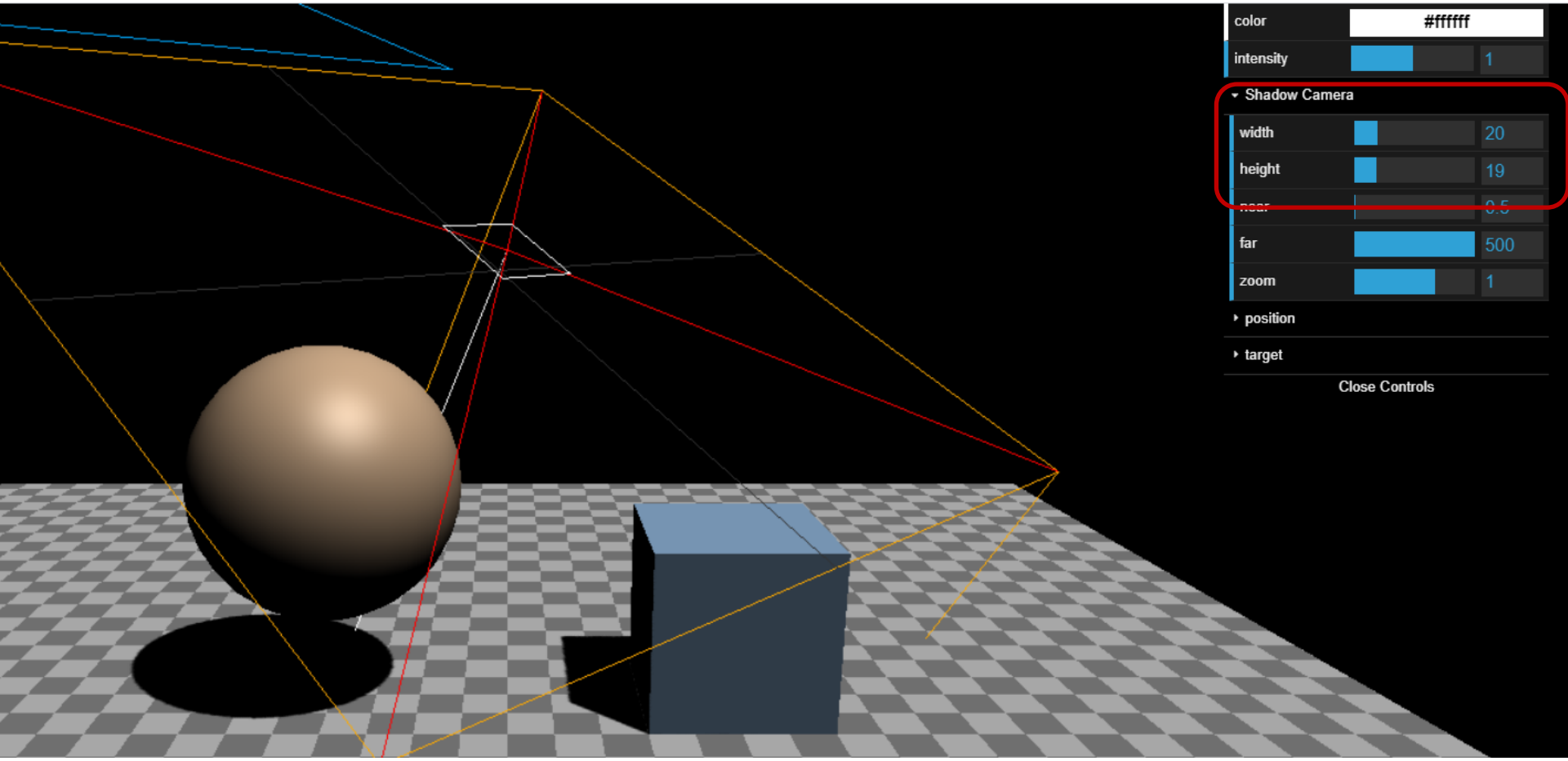


P.PORTO

ESCOLA SUPERIOR DE MEDIA ARTES E DESIGN

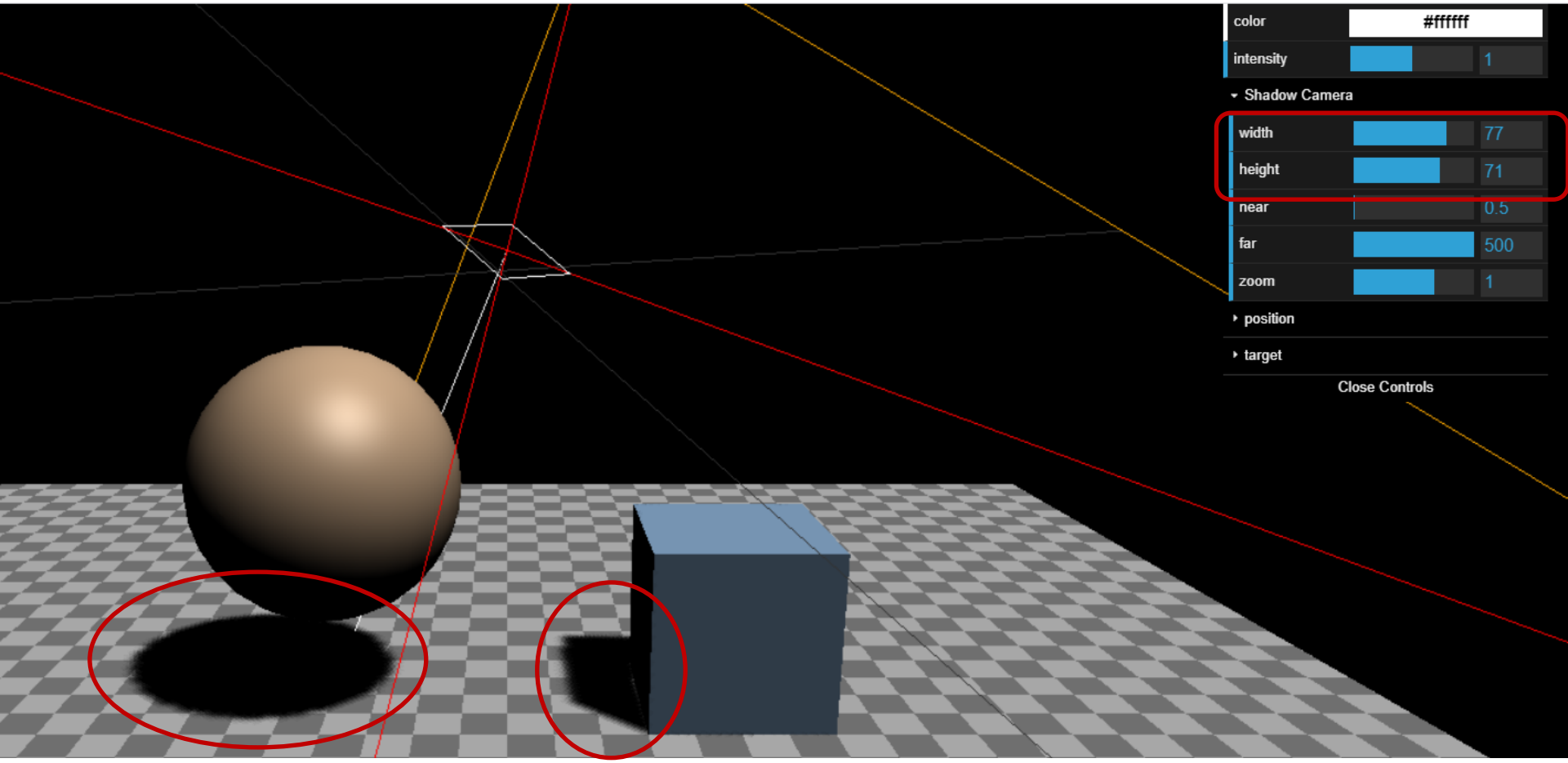
Shadows

- Directional light example (light shadow camera is orthographic)



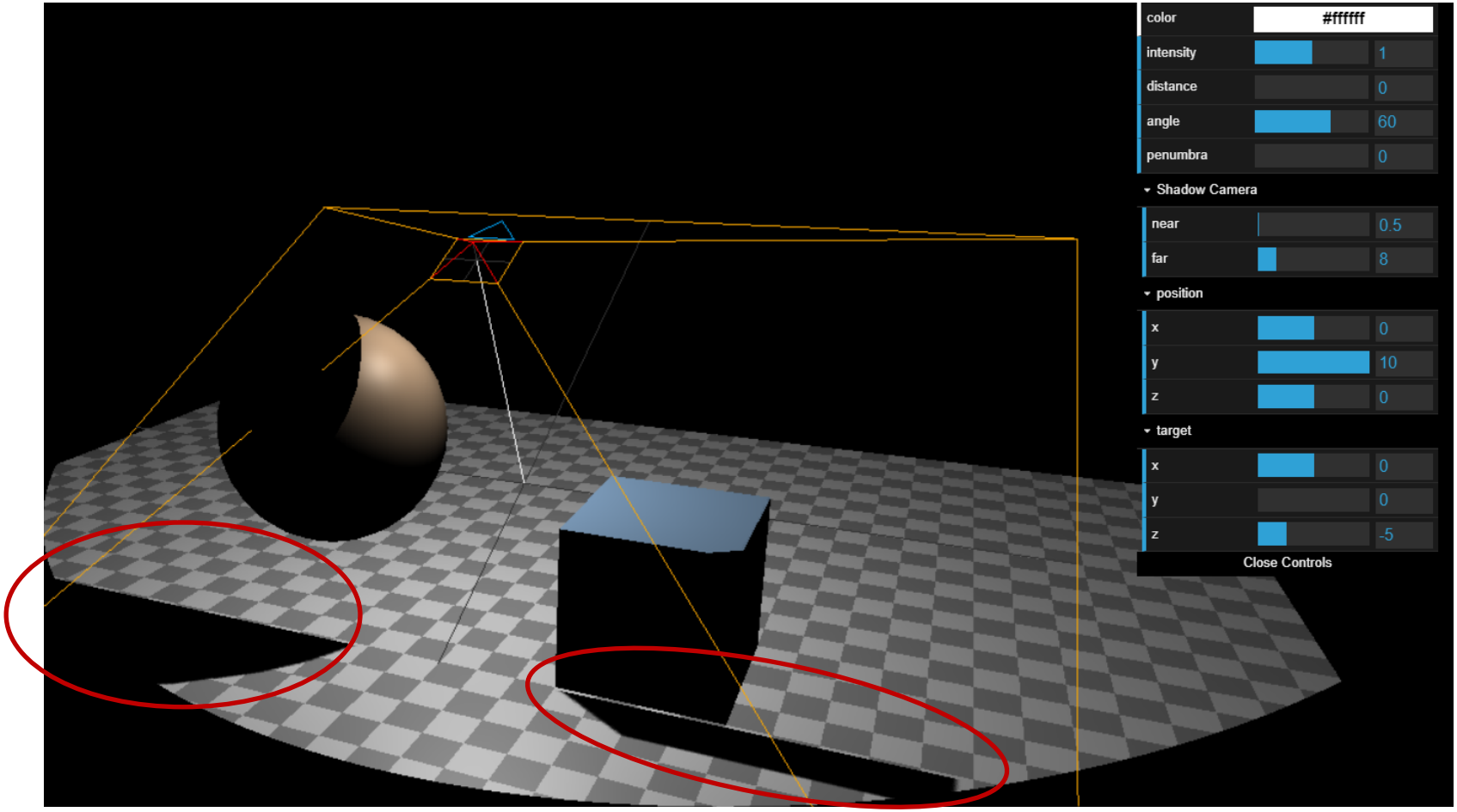
Shadows

- Directional light example (light shadow camera is orthographic)



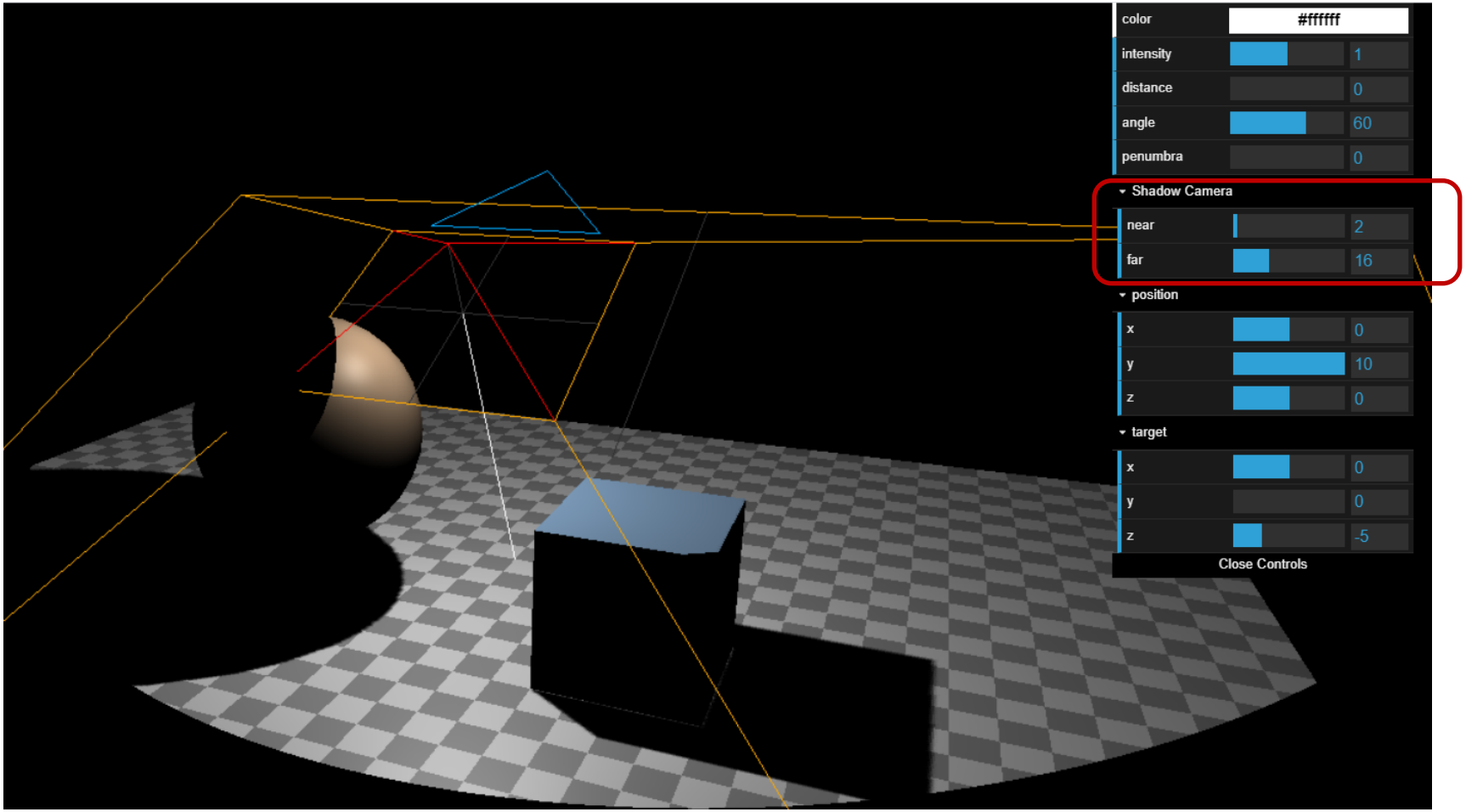
Shadows

- Spotlight example (light shadow camera is perspective)



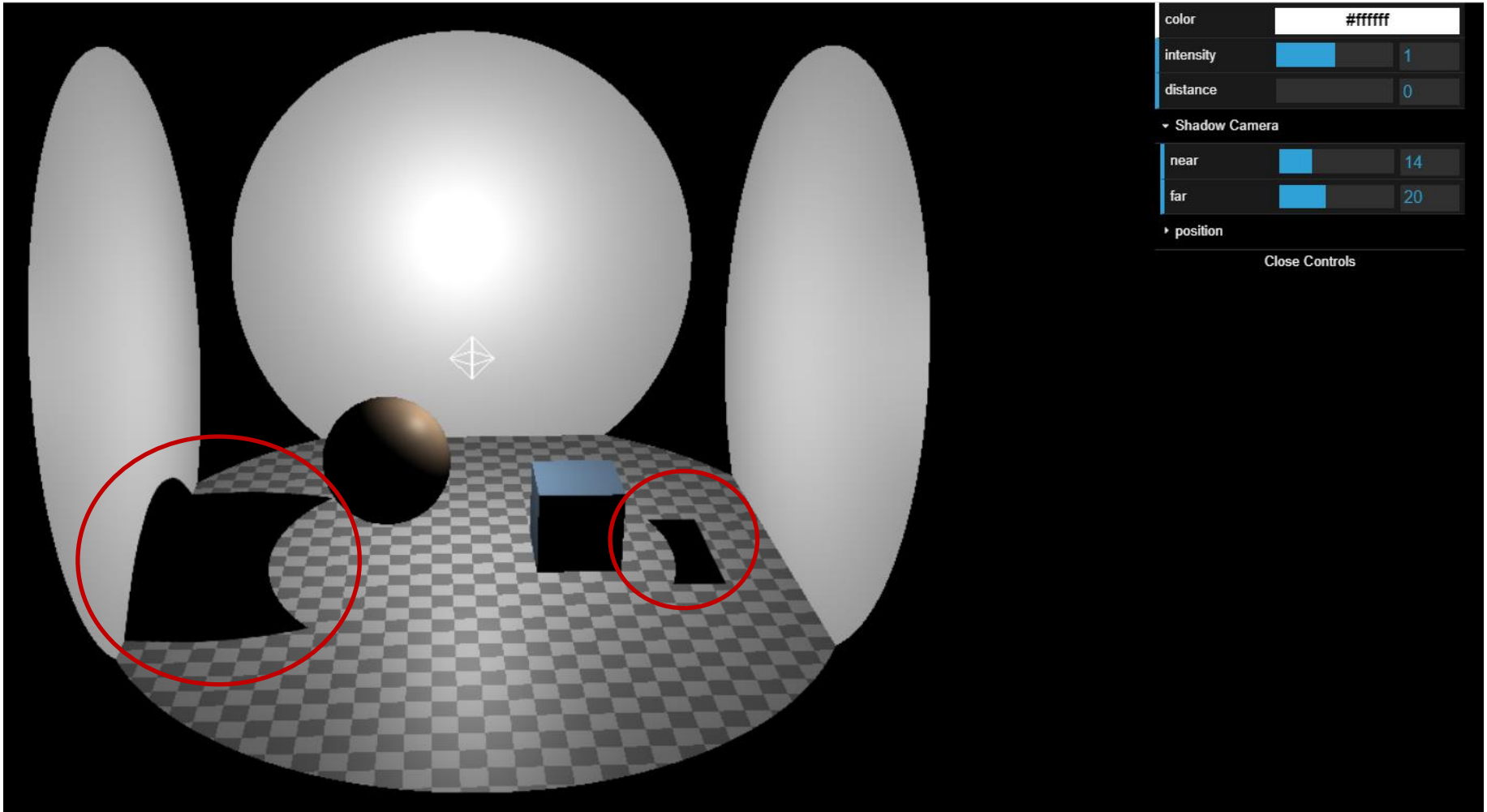
Shadows

- Spotlight example (light shadow camera is perspective)



Shadows

- Pointlight example (6 perspective light shadow cameras)



Shadows

- Pointlight example (6 perspective light shadow cameras)