SGRAI
Graphics Systems and Interaction

P.PORTO

isep Instituto Superior de Engenharia do Porto

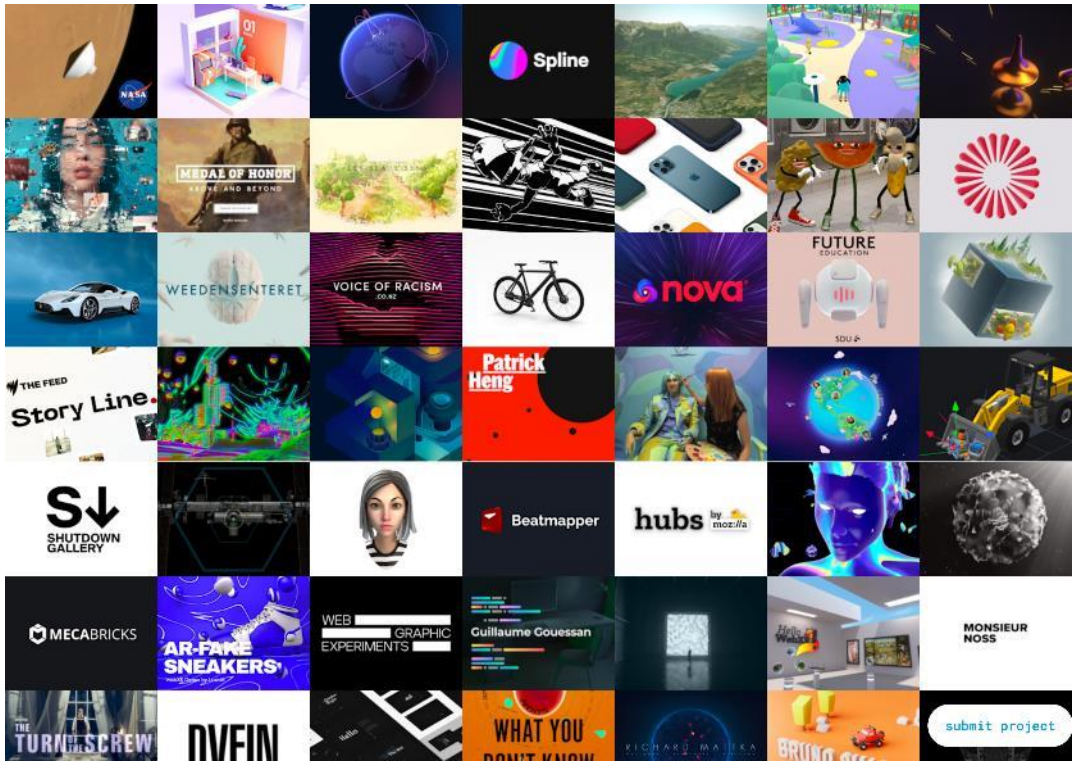
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Filipe Pacheco (ffp@isep.ipp.pt)

Three.js

Three.js is a JavaScript 3D library and application programming interface (API)

- Create and display animated 3D computer graphics in a web browser
- Cross-browser
- WebGL-based



Three.js concepts

What are we drawing

Scene

Mesh
Object3D
Light
…

What is the 3D view

Camera

Perspective / Orthographic
Position
Field Of View
…

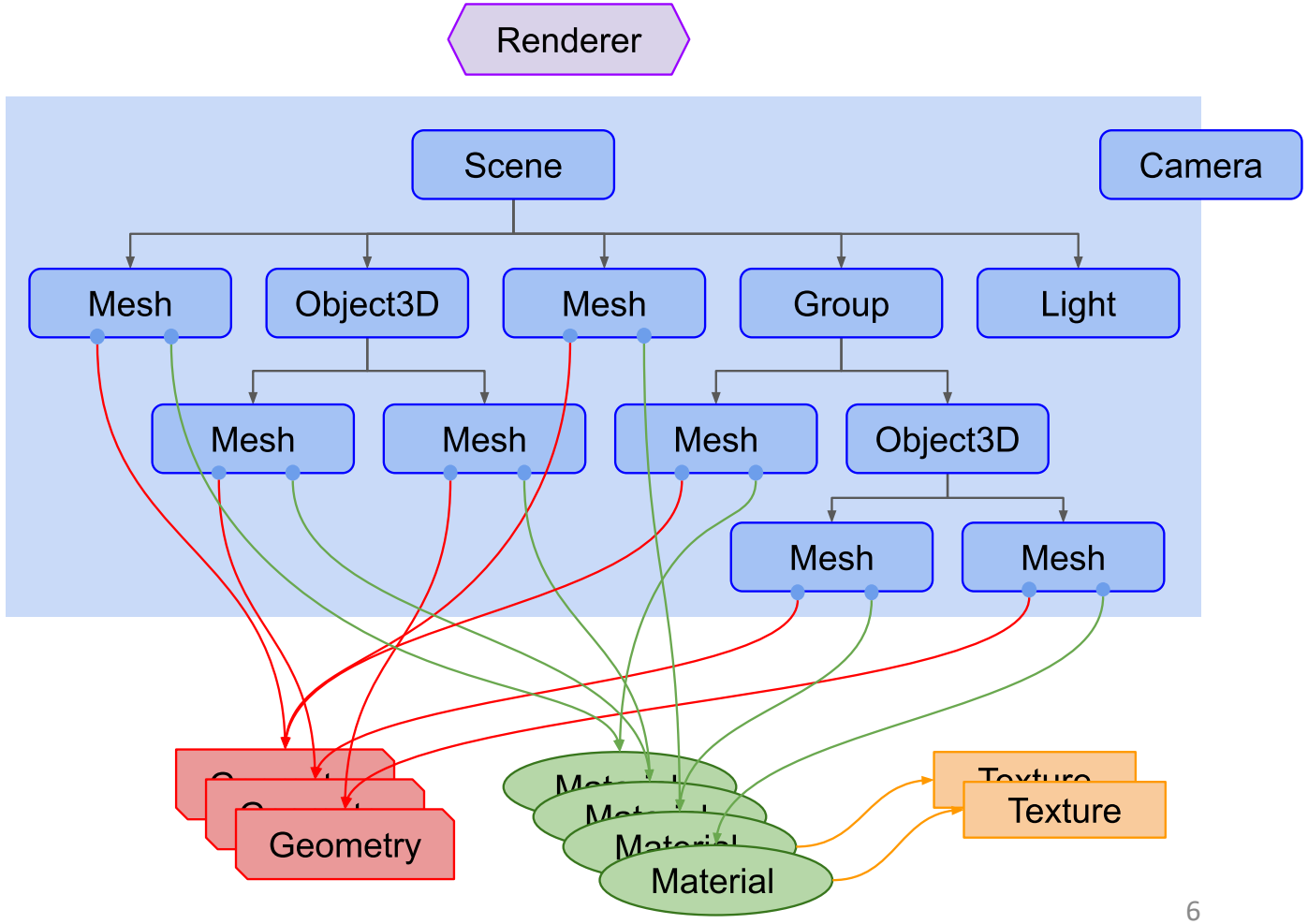
How are we drawing

Renderer

WebGL

Example of a three.js App

- A **Scene** can include multiple **Mesh**, **Light**, **Group**, **Object3D** and **Camera** objects
- **Mesh** is an **Object3D** that represents drawing a specific **Geometry** and a specific **Material**
- Both **Geometry** & **Material** objects can be shared between multiple **Mesh** objects



https://threejsfundamentals.org/threejs/lessons/threejs-fundamentals.html

Making our first scene

First, we must have and HTML file where three.js content will be displayed:

```html
<html>
<head>
    <meta charset="utf-8">
    <title>My first three.js app</title>
    <style> body { margin: 0; } </style>
</head>
<body>
<script type="importmap">
{
    "imports": {
        "three": "https://unpkg.com/three@0.156.0/build/three.module",
        "three/": "https://unpkg.com/three@0.156.0/",
        "dat.gui": "https://unpkg.com/dat.gui"
    }
}
</script>
<script type="module">
// Our JavaScript will go here.
</script>
</body>
</html>
```

You can also use a local copy of ThreeJS library

The bare basics

```
import * as THREE from 'three';
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(
        75, window.innerWidth / window.innerHeight, 0.1, 1000
);
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

Adds a <canvas> element where things are displayed

Cameras

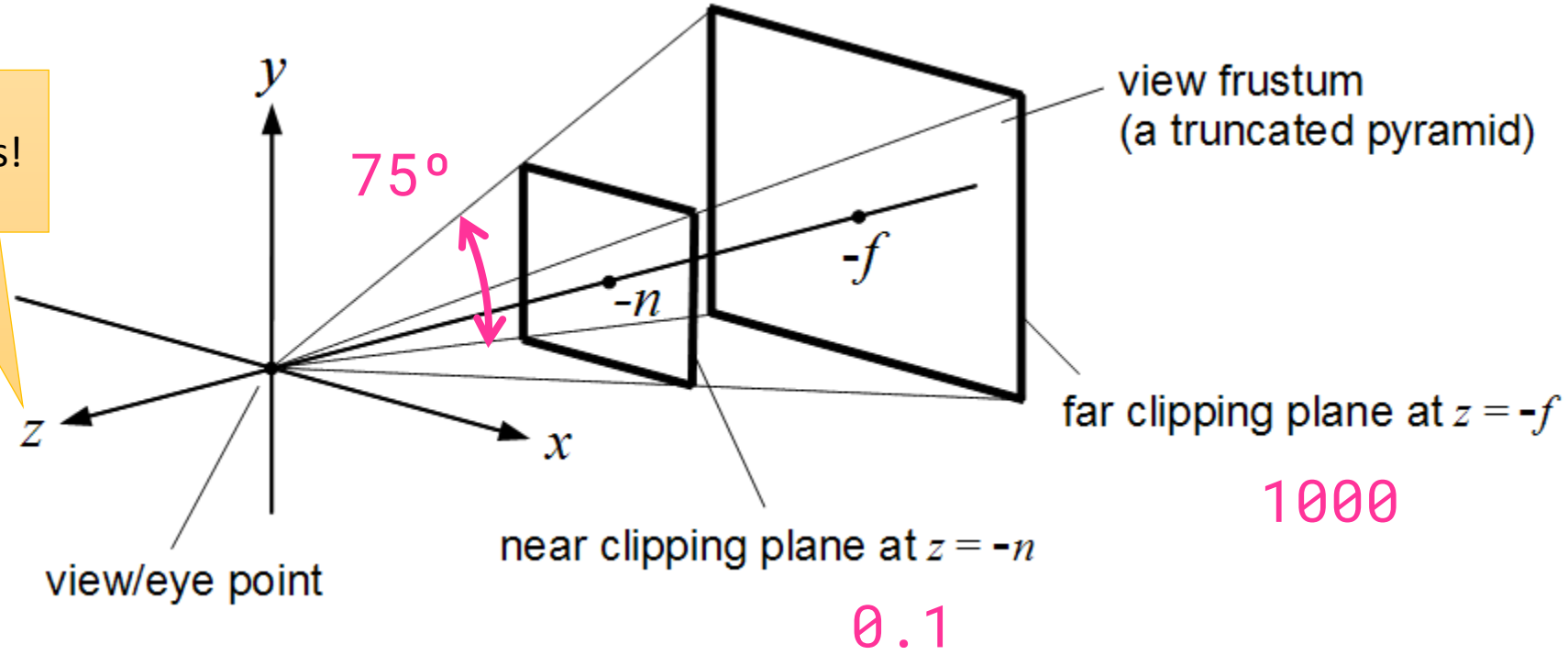
Aspect Ratio
W/H

- The perspective camera

```
PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1, 1000);
```

FoV
AspectRatio
Near (>0 !)
Far

Note the axis!

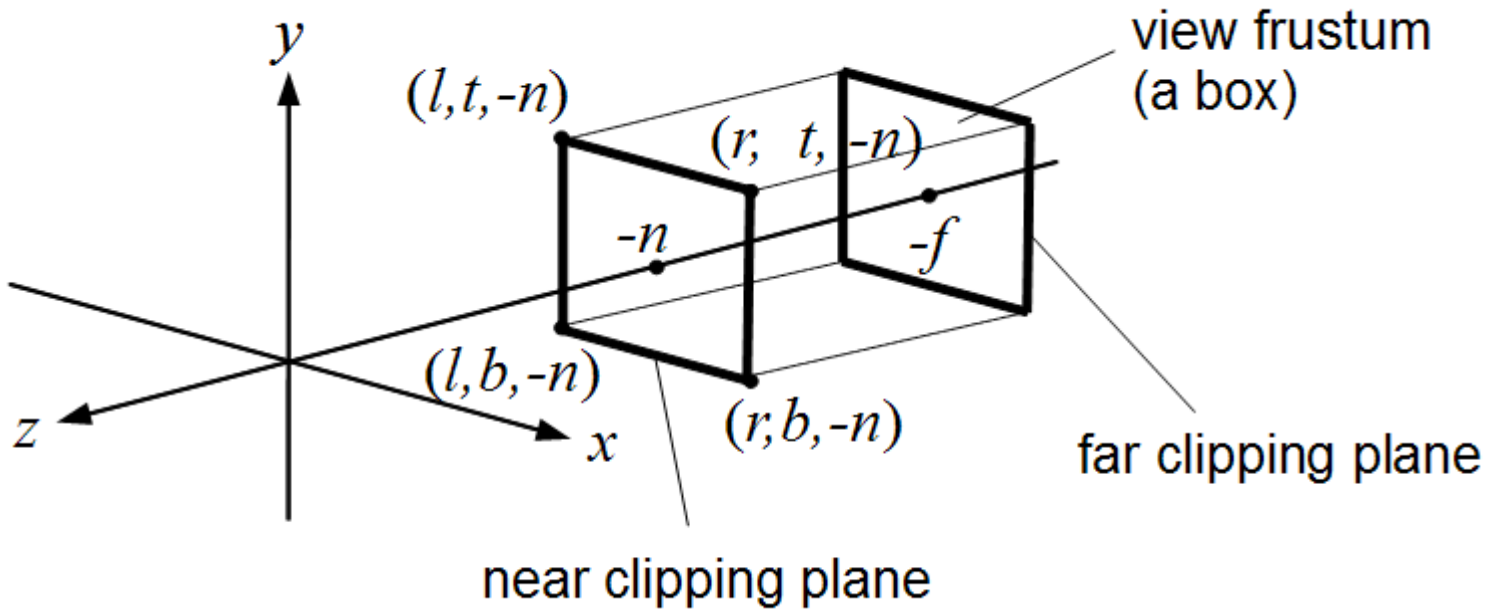


Cameras

- The orthographic camera

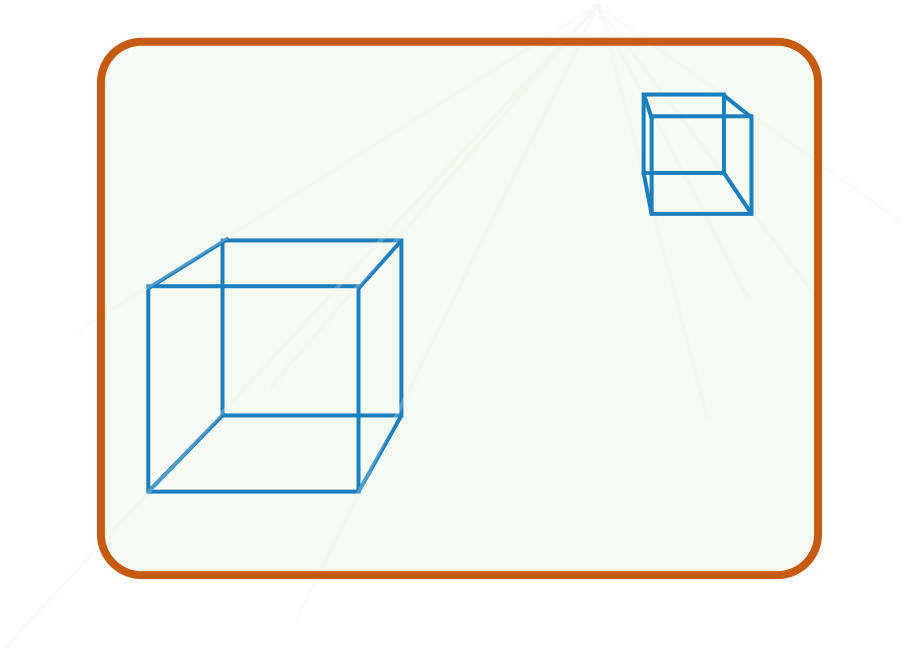
```
const aspectRatio = window.innerWidth / window.innerHeight;
camera = new THREE.OrthographicCamera
        (-aspectRatio, aspectRatio, 1.0, -1.0, 0.0, 1.0);
```

Left
Right
Top
Bottom
Near (>= 0 !)
Far

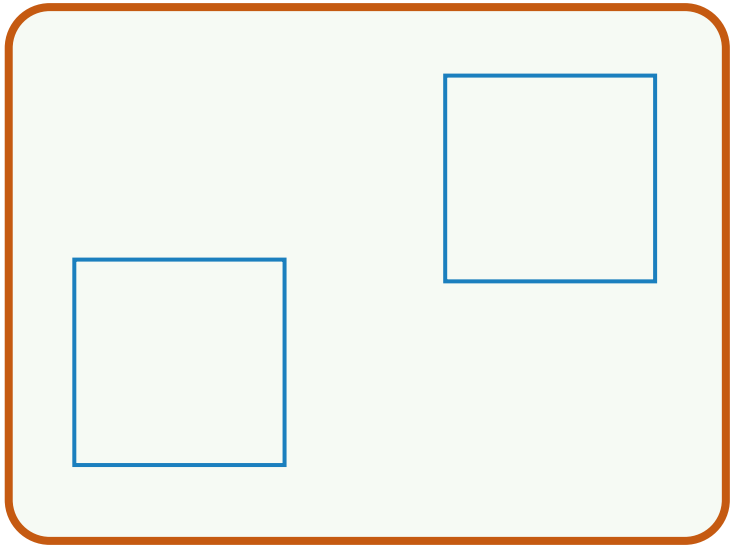


Cameras

Perspective

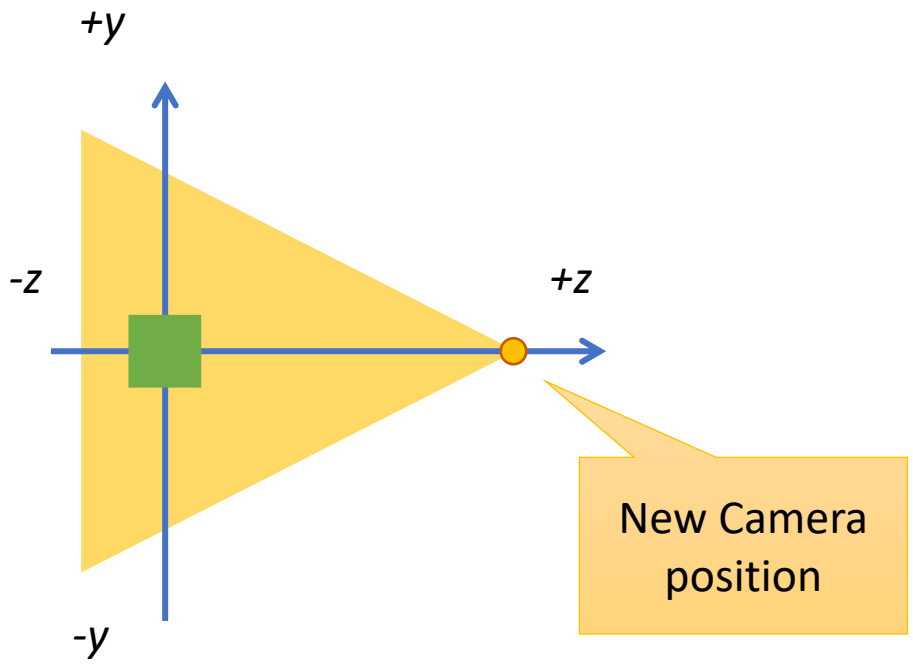


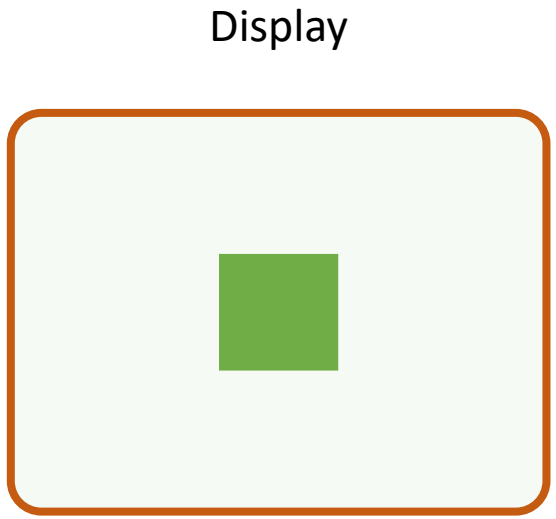
Orthographic



Moving the camera

- The default camera is in the origin, but our object is also in the origin… to display things properly we want to move the camera a little bit in the z axis





Add a Cube

Vertexes & faces of a cube
Center: origin
Side: 1 unit

Basic GREEN material
0xRRGGBB

```
const geometry = new THREE.BoxGeometry();
const material = new THREE.MeshBasicMaterial( { color: 0x00ff00 }
);
const cube = new THREE.Mesh( geometry, material );
scene.add( cube );

camera.position.z = 5;
```

Add our cube to the Scene

Create a 3D object from the vertexes/faces & material

Move the camera 5 units from the cube

Show things

```
function animate() {
    requestAnimationFrame( animate );
    renderer.render( scene, camera );
}
animate();
```

Register our function so it is called *next time* the screen is refreshed

Render (show) our Scene

First call to our function

Animate our cube

```
function animate() {
    requestAnimationFrame( animate );
    cube.rotation.x += 0.01;
    cube.rotation.y += 0.02;
    renderer.render( scene, camera );
}
animate();
```

Rotate our cube
in X and Y axis

Test

Check and edit the final code at:

https://jsfiddle.net/9ur6Lzgo/

You should also test the code in your local setup.

Drawing Lines

Let’s draw a line.

Startup code.

```
import * as THREE from 'three';
const renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
const camera = new THREE.PerspectiveCamera
       ( 45, window.innerWidth / window.innerHeight, 1, 500 );
camera.position.set( 0, 0, 100 );
camera.lookAt( 0, 0, 0 );
const scene = new THREE.Scene();
scene.background = new THREE.Color( 0xf0f0f0 );
```

Camera Position x,y,z

Where is the Camera pointing to

Light gray background

Line materials & geometry

```
//create a blue Line
const material = new THREE.LineBasicMaterial( { color: 0x0000ff } );

// define geometry
const points = [];
points.push( new THREE.Vector3( -10, 0, 0 ) );
points.push( new THREE.Vector3( 0, 10, 0 ) );
points.push( new THREE.Vector3( 10, 0, 0 ) );
const geometry = new THREE.BufferGeometry().setFromPoints( points );
```

Draw the line

Make a line connecting the 3 points

```
const line = new THREE.Line( geometry, material );
```

Render stuff…

```
scene.add( line );
renderer.render( scene, camera );
```

Result!

Test

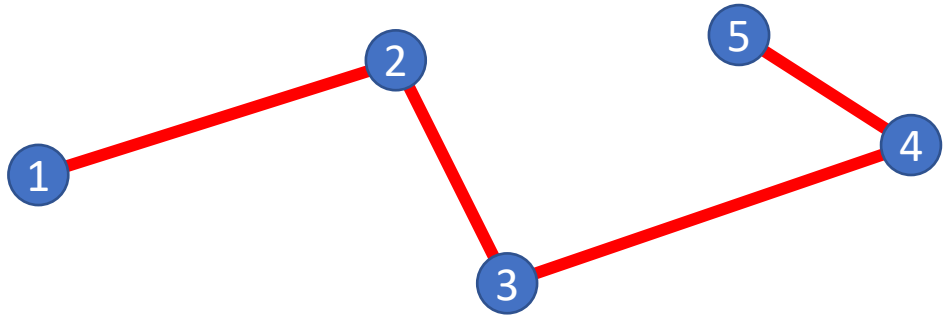
Check and edit the final code at:

https://jsfiddle.net/21wquath/

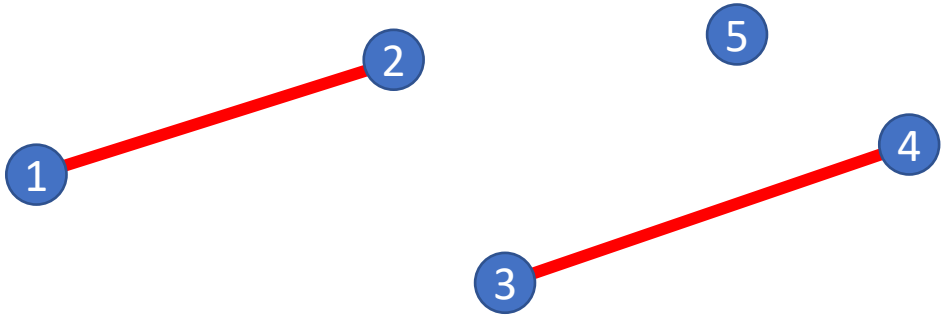
You should also test the code in your local setup.

Line & LineSegments

- https://threejs.org/docs/#api/en/objects/Line



- https://threejs.org/docs/#api/en/objects/LineSegments



LineDashedMaterial

- https://threejs.org/docs/#api/en/materials/LineDashedMaterial

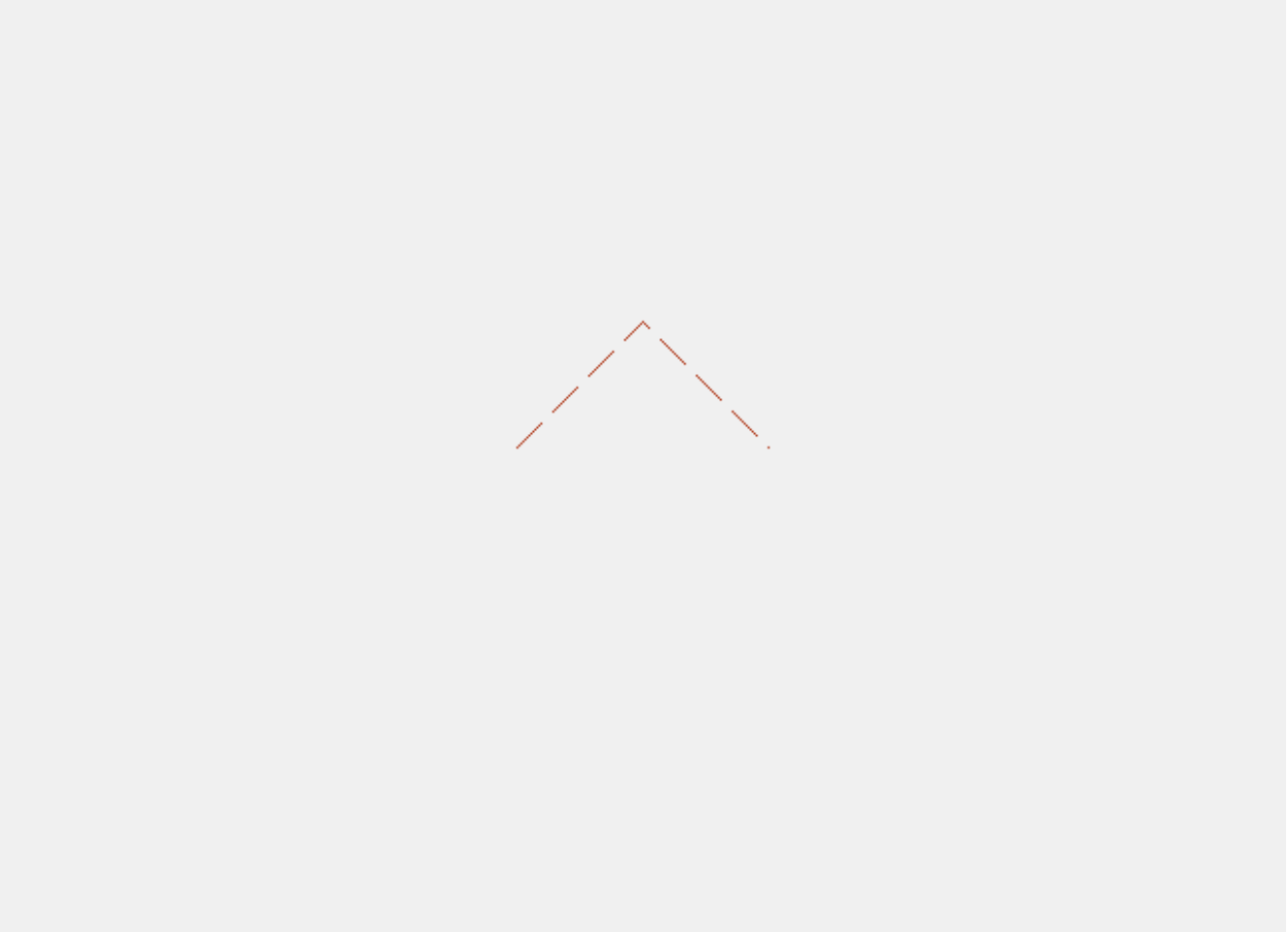
```
const material = new THREE.LineDashedMaterial( {
    color: 0xd00000,
    linewidth: 1,
    scale: 1,
    dashSize: 3,
    gapSize: 1,
} );
```

>1 ignored in most WebGL implementations

Important: add this after defining your line, when using LineDashedMaterial:

```
line.computeLineDistances();
```

LineDashedMaterial



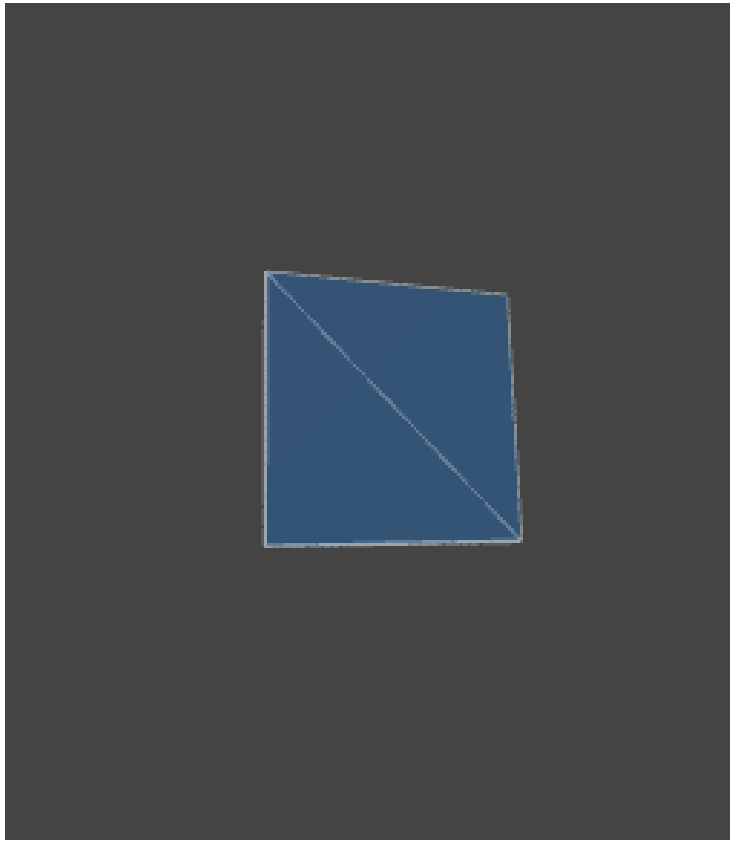
Drawing other things

PlaneGeometry → square or rectangle

https://threejs.org/docs/#api/en/geometries/PlaneGeometry

Width & Height (X, Y)

```
const geometry = new THREE.PlaneGeometry( 1, 1 );
const material = new THREE.MeshBasicMaterial(
     {color: 0xffff00, side: THREE.DoubleSide} );
const plane = new THREE.Mesh(geometry, material);

scene.add( plane );
```

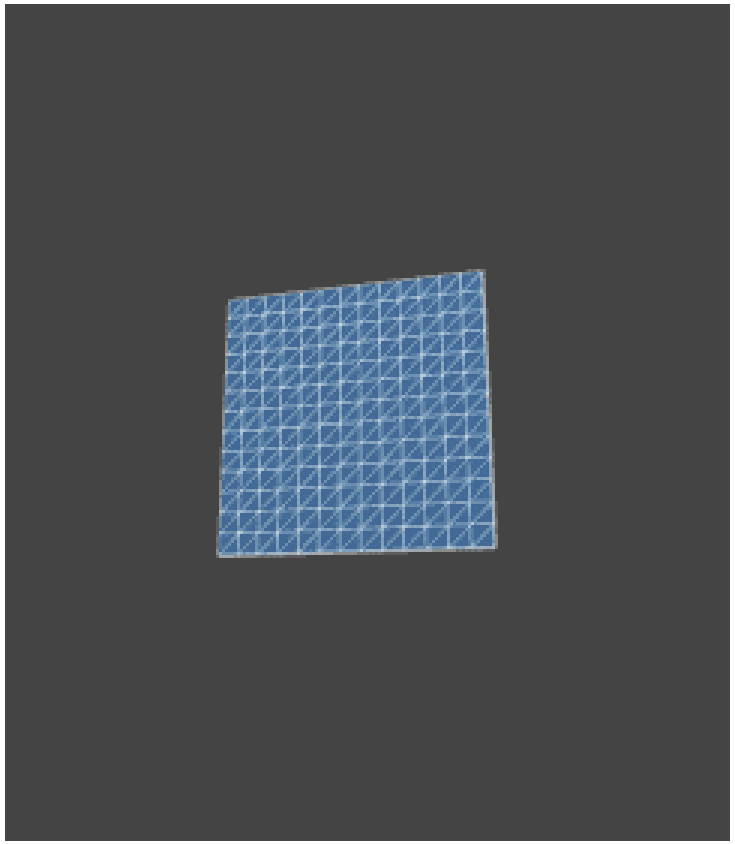


Drawing other things

- PlaneGeometry

```
PlaneGeometry(
    width : Float, height : Float,
    widthSegments : Integer,
    heightSegments : Integer
)
```

- We can add segments for several reasons including better light rendering (e.g. light in a wall)



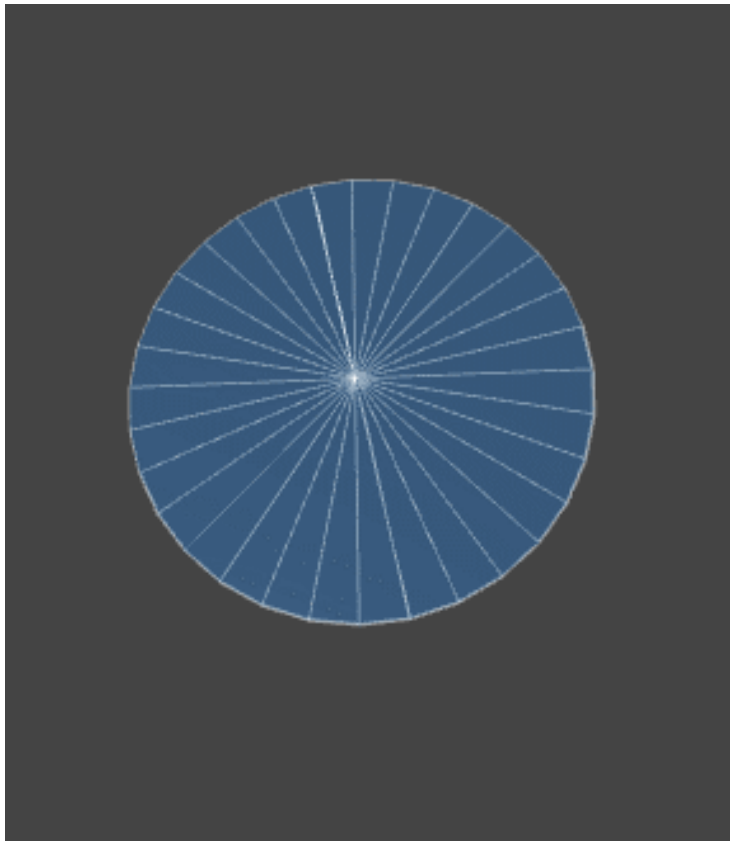
Drawing a circle

- CircleGeometry
  https://threejs.org/docs/#api/en/geometries/CircleGeometry

Radius

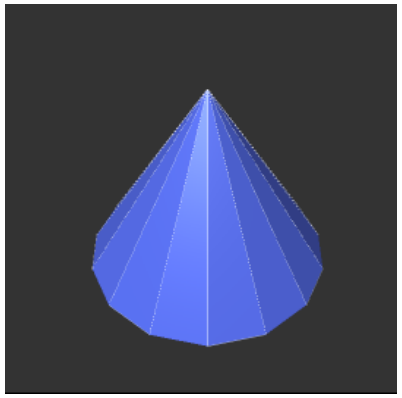
Segments

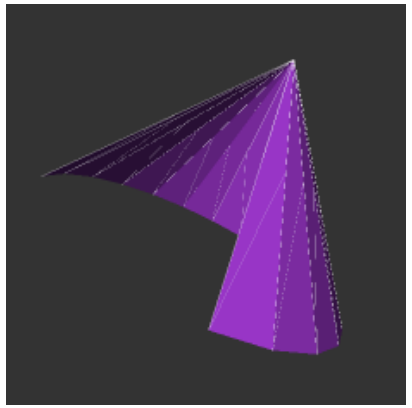
```
const geometry = new THREE.CircleGeometry( 5, 32 );
const material = new THREE.MeshBasicMaterial(
   { color: 0xffff00 } );
const circle = new THREE.Mesh( geometry, material );
scene.add( circle );
```



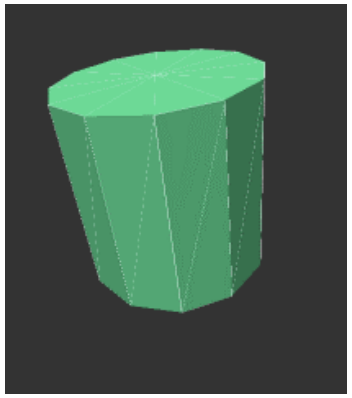
Other Basic Geometry

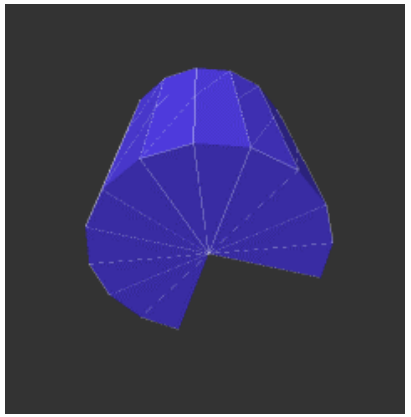
ConeGeometry



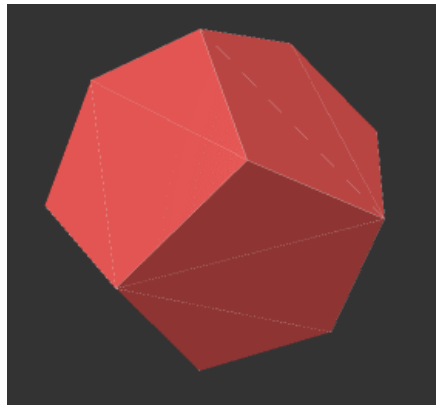


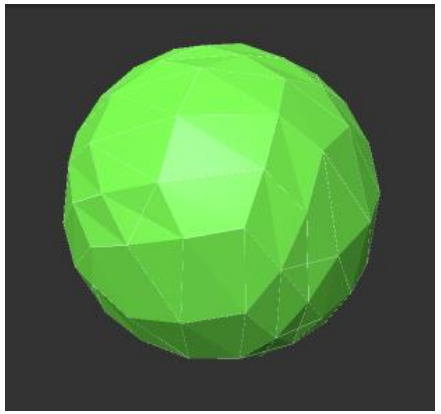
CylinderGeometry





DodecahedronGeometry

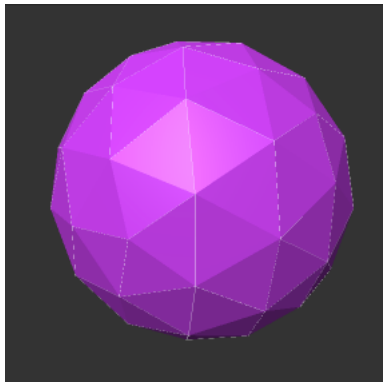




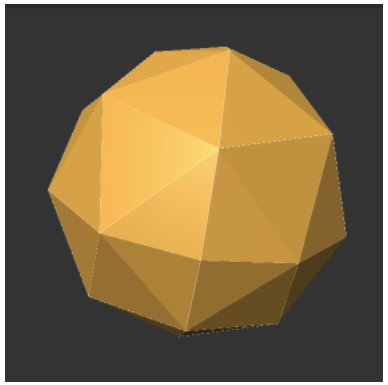
https://threejs.org/manual/#en/primitives

Other Basic Geometry

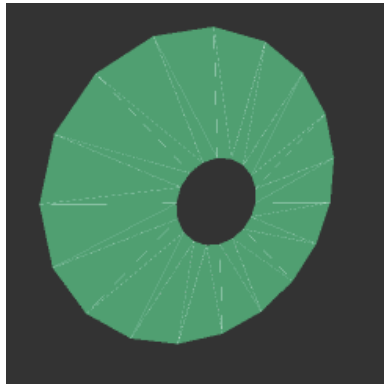
IcosahedronGeometry



OctahedronGeometry



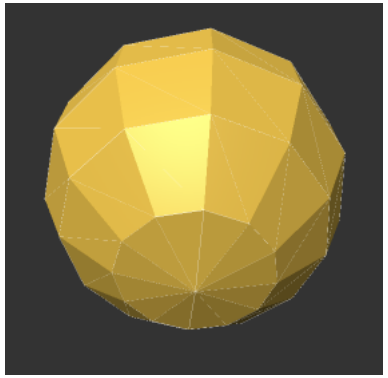
RingGeometry

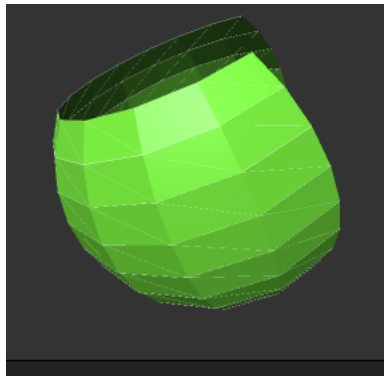


https://threejs.org/manual/#en/primitives

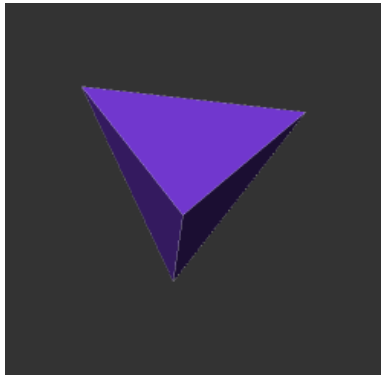
Other Basic Geometry

SphereGeometry

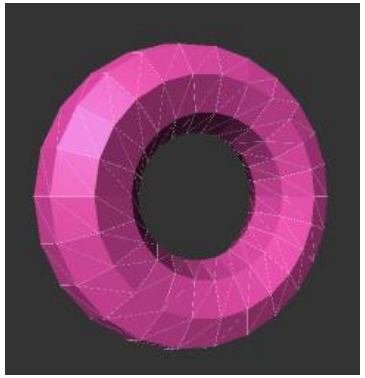


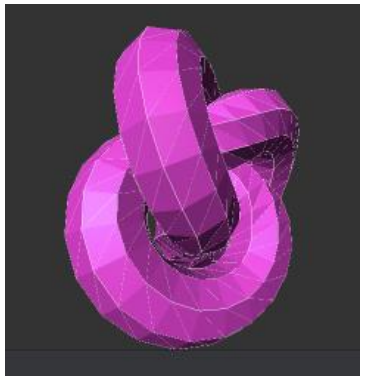


TetrahedronGeometry



TorusGeometry



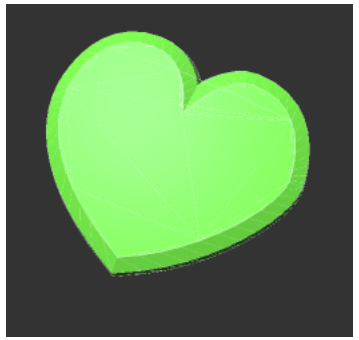


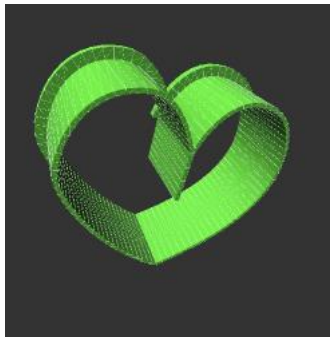
https://threejs.org/manual/#en/primitives

Other Derived Geometry

ExtrudeGeometry

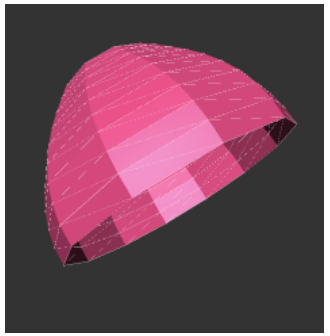
Using **Shape**

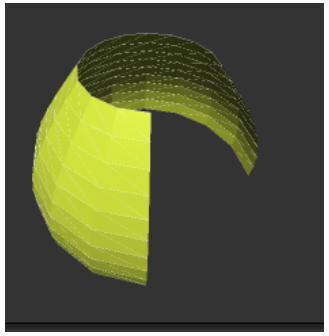




LatheGeometry

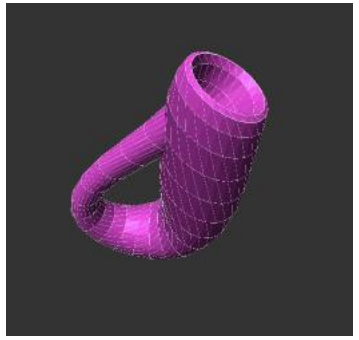
Using a point vector





ParametricGeometry

Using a function u,v → x,y,z



https://threejs.org/manual/#en/primitives

Other Derived Geometry

ShapeGeometry

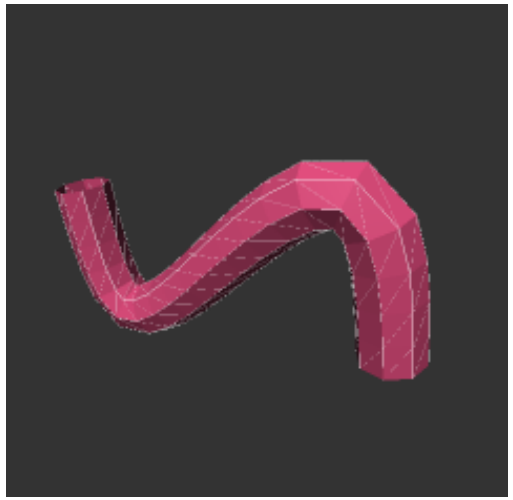
Using **Shape**

TextGeometry

From a 3D Font and string

TubeGeometry

A circle traced down a **Curve**



https://threejs.org/manual/#en/primitives

Drawing generic objects

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

A generic class (super-class of **PlaneGeometry** & **CircleGeometry**) used to store, in way that is *efficient for GPU*, geometry information of:

- Meshes with non-indexed faces
- Meshes with indexed faces
- Lines
- Indexed Lines
- And more…

Drawing generic objects

BufferGeometry has fixed sized TypedArrays (*binary data buffers* ready-to-use by the GPU), and includes data for:

- Vertexes
- Indexes
- Normals
- Colors
- Texture mappings (UV)

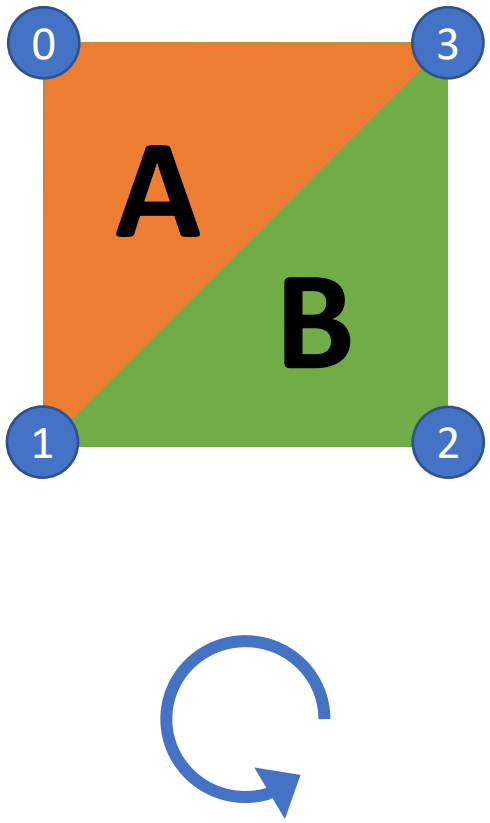
Data can be used in ranges and/or divided in groups.

Non-Indexed vs Indexed Faces/Lines

- Example: a simple square built using 2 triangles (A & B)
- Non-indexed version
  - Add **Vertices** coordinates for points 0, 1 and 3
  - Add **Vertices** coordinates for points 1, 2 and 3

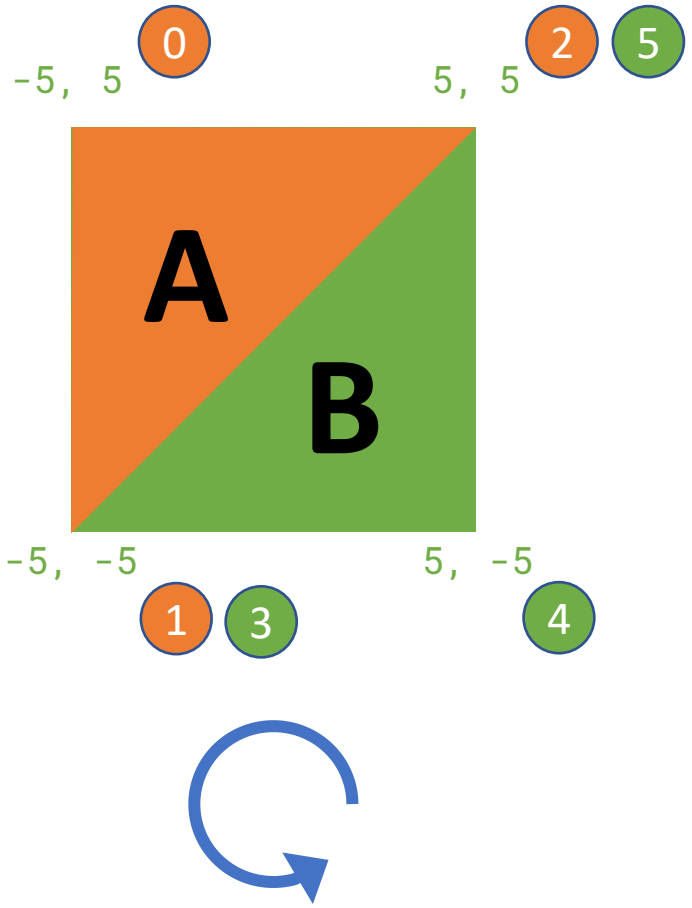
  → Vertex 2 is added twice
- Indexed version
  - Add **Vertices** coordinates for points 0 to 3
  - Add **Indexes** {0, 1, 3} and {1, 2, 3}

  → No duplicated vertex data



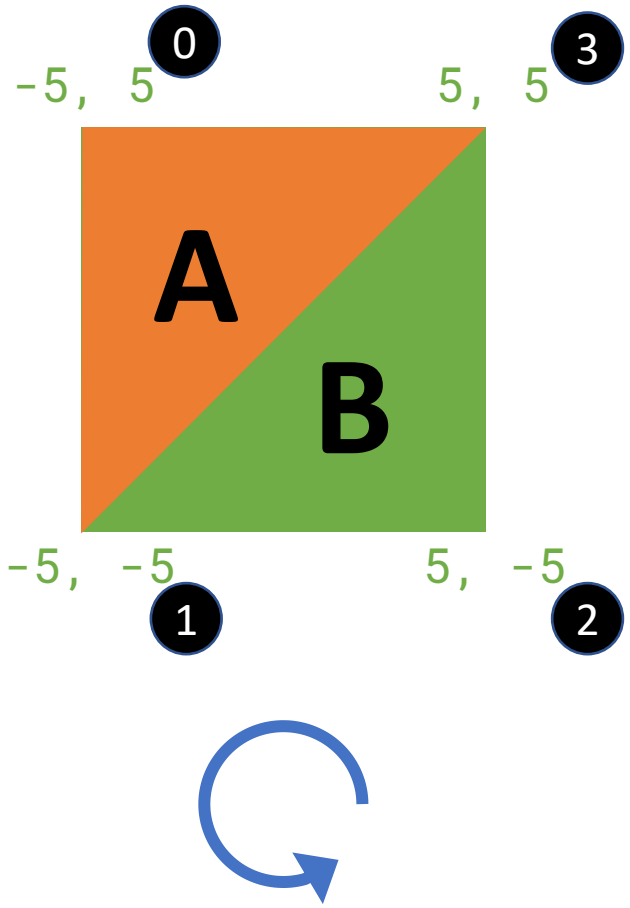
Non-Indexed Face Example

```
// define geometry
const points = [];
points.push( new THREE.Vector3( -5, 5, 0 ) );
points.push( new THREE.Vector3( -5, -5, 0 ) );
points.push( new THREE.Vector3( 5, 5, 0 ) );

points.push( new THREE.Vector3( -5, -5, 0 ) );
points.push( new THREE.Vector3( 5, -5, 0 ) );
points.push( new THREE.Vector3( 5, 5, 0 ) );

const geometry = new
    THREE.BufferGeometry().setFromPoints( points );
```



Indexed Face Example

```
// define geometry
const points = [];
points.push( new THREE.Vector3( -5, 5, 0 ) );   0
points.push( new THREE.Vector3( -5, -5, 0 ) );  1
points.push( new THREE.Vector3( 5, -5, 0 ) );   2
points.push( new THREE.Vector3( 5, 5, 0 ) );    3

const indices = [0, 1, 3,     1, 2, 3];  A  B

const geometry = new THREE.BufferGeometry()
        .setFromPoints( points );
geometry.setIndex(new THREE.BufferAttribute(
        new Uint16Array(indices), 1));
```



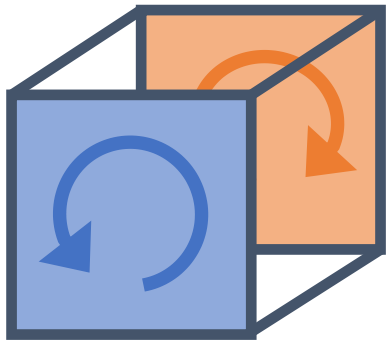
Single Sided vs Double Sided

By default, Three.js draws only faces where the vertexes appear in anti-clockwise order (on the screen).

This reduces the number of drawn faces in solids to about half (since back faces are not drawn).

If you want to draw both sides (e.g., an object without volume), you can setup the material:

```
const material = new THREE.MeshBasicMaterial(
  { color:  0x0000ff, side: THREE.DoubleSide }
);
```



Test

Check and edit the code at:

https://jsfiddle.net/59wvxz70/

You should also test the code in your local setup.

Object position/rotation/scale

All **Object3D** have the following properties related to 3D transformations:

.position:Vector3
.rotation:Euler Euler → x, y, z
.scale:Vector3

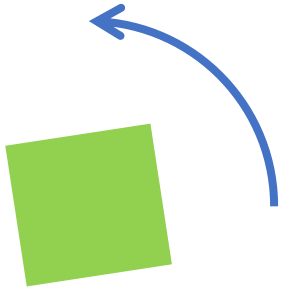
And the following methods:

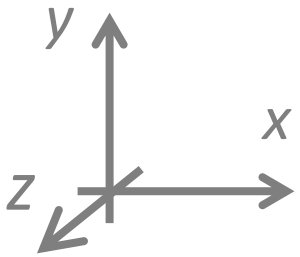
.rotateX(Float), **.rotateY**(Float) and **.rotateZ**(Float)
.translateX(Float), **.translateY**(Float) and **.translateZ**(Float)

Rotating an object

```
var animate = function () {
    requestAnimationFrame( animate );

    // cube.rotation.z += 0.01;
    cube.rotateZ(0.01);

    renderer.render( scene, camera );
};

animate();
```

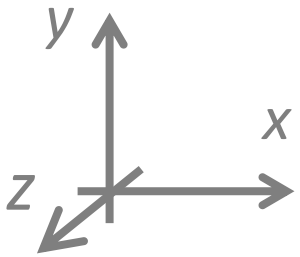
Same result





Moving an object

```
cube.position.x += 0.01;

cube.translateX(0.01);
```



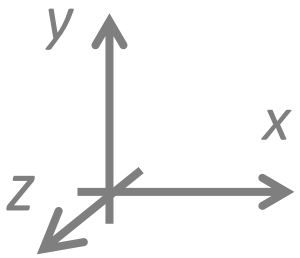
Change the scale

```
cube.scale.x*=1.01;
```

|scale| >1 → bigger

|scale| <1 → smaller

scale <0 → mirror effect



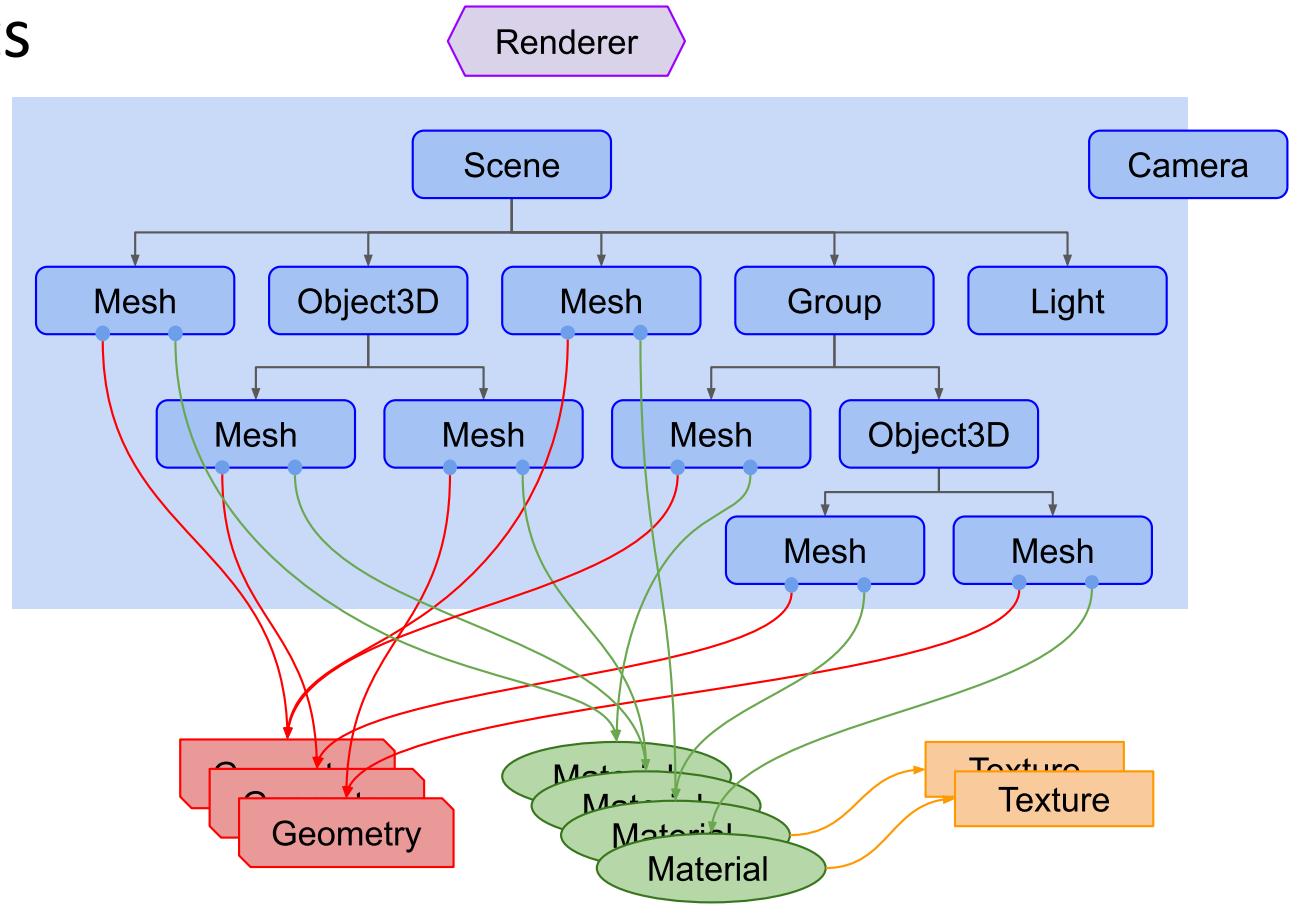
Hierarchy

Both **Object3D** and **Group** objects can have children (and grand-children, and so on…)

Note: **Mesh** is a super-class of **Object3D**

Group exists only to make hierarchy clearer when programming

When you do a transformation a parent, the transformations are applied to the children too

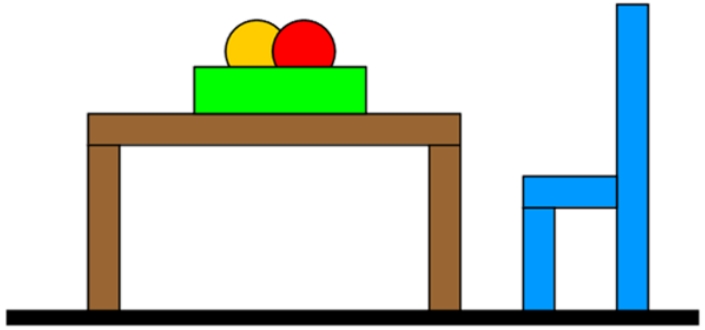


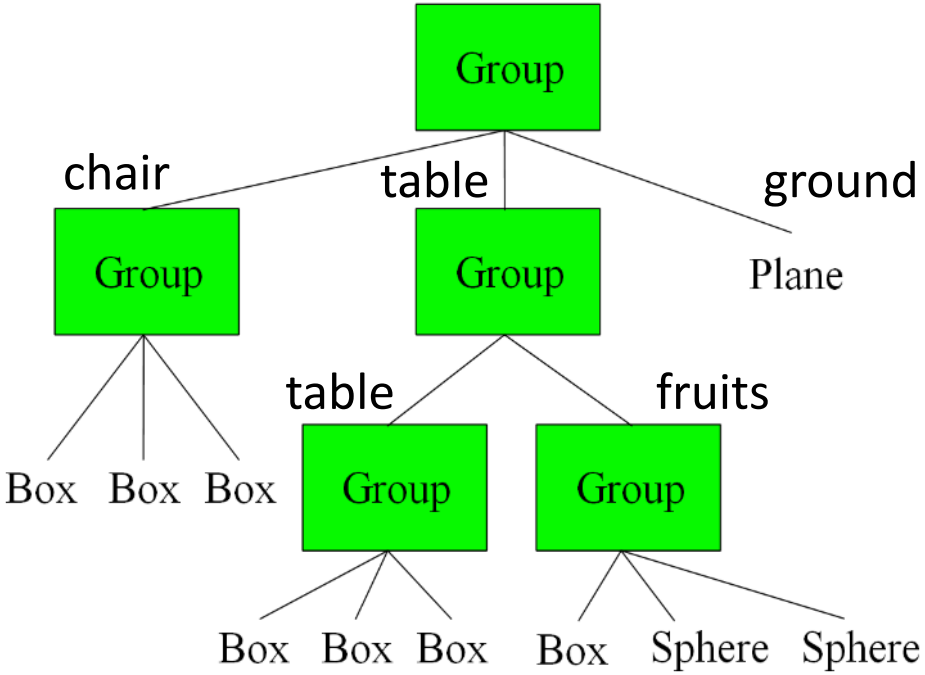
Hierarchy

- Tree structures to represent the model
- Each node is a transformation or object
- Nodes in the same branch represent accumulated transformations
- Nodes from different branches are independent transformations

Hierarchy

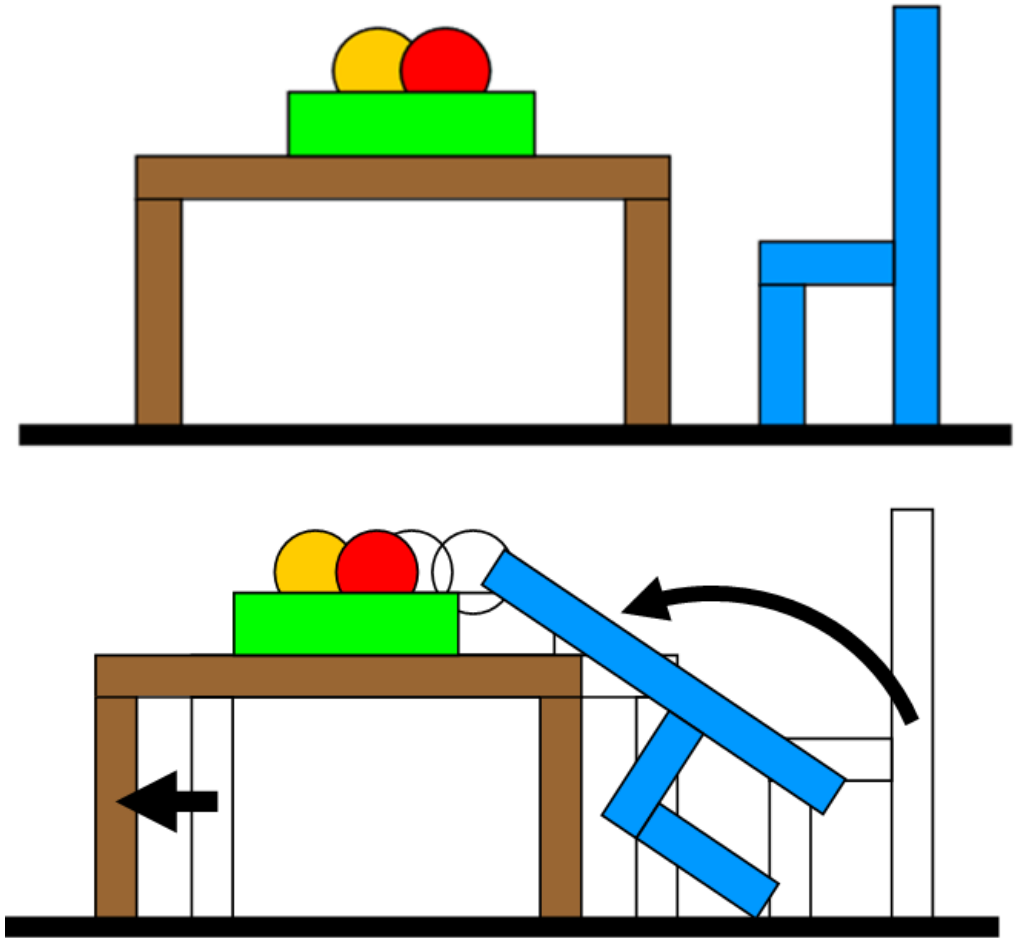
- Logical organization of a scene



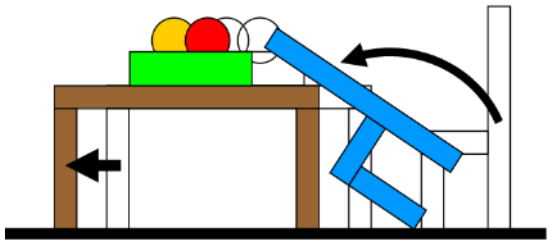


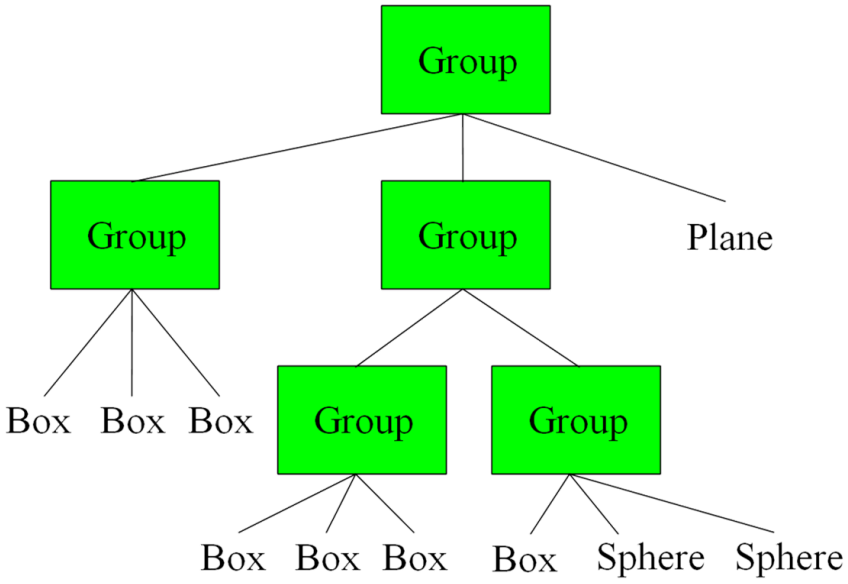
MIT Computer Graphics 6.837

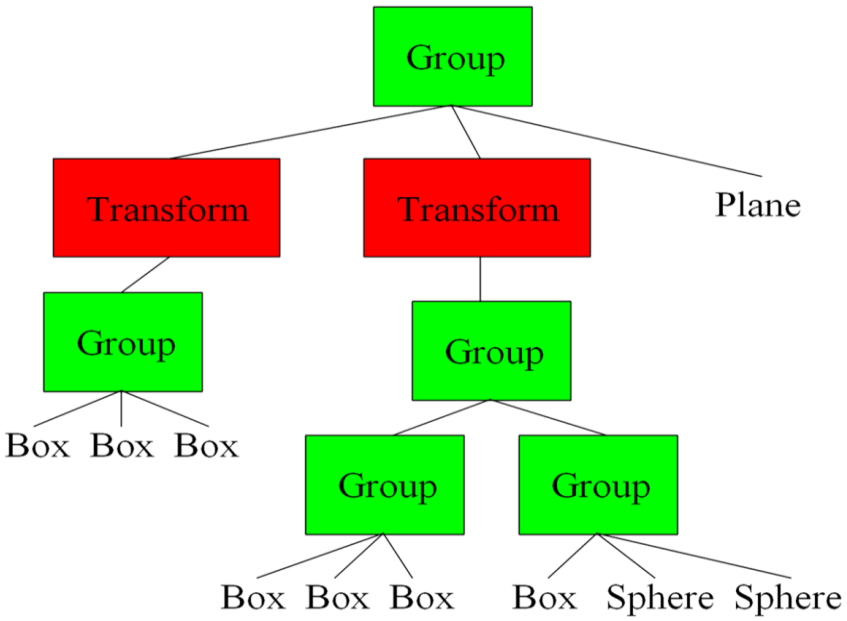
Applying Transformations



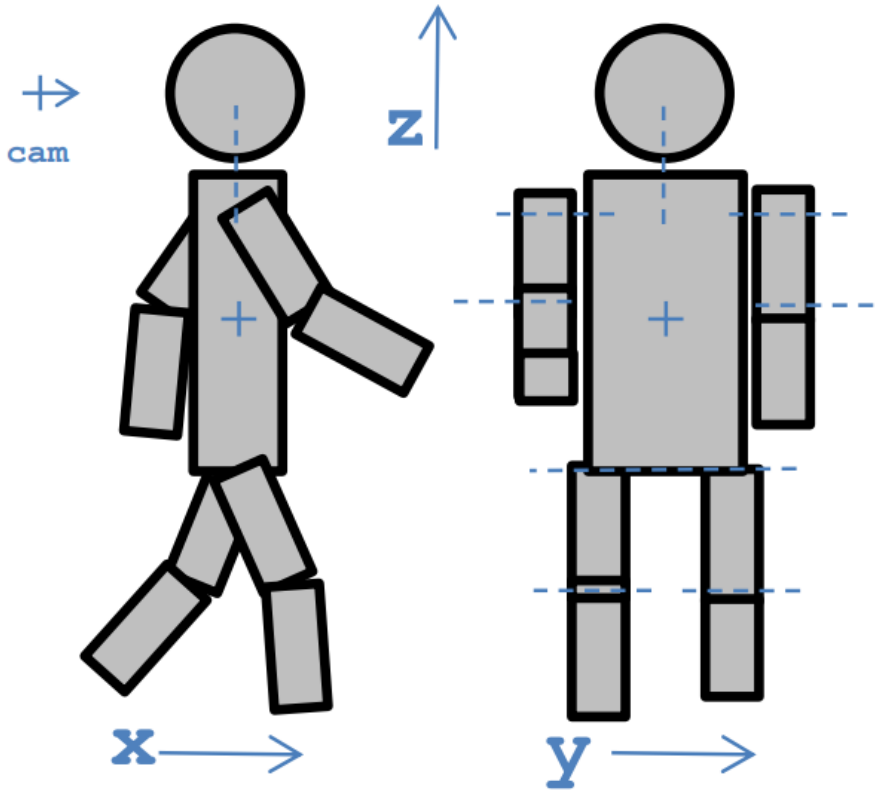
Transform the logical position of groups of objects within the scene



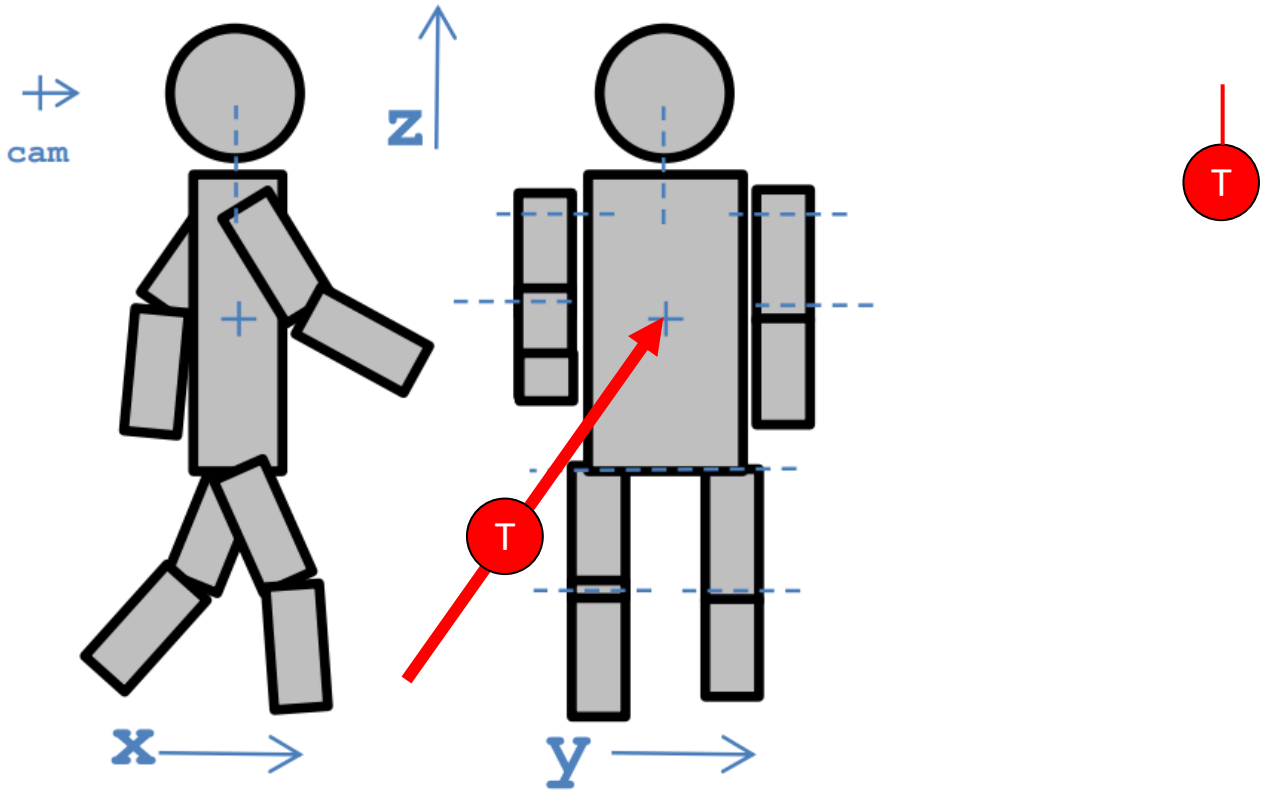




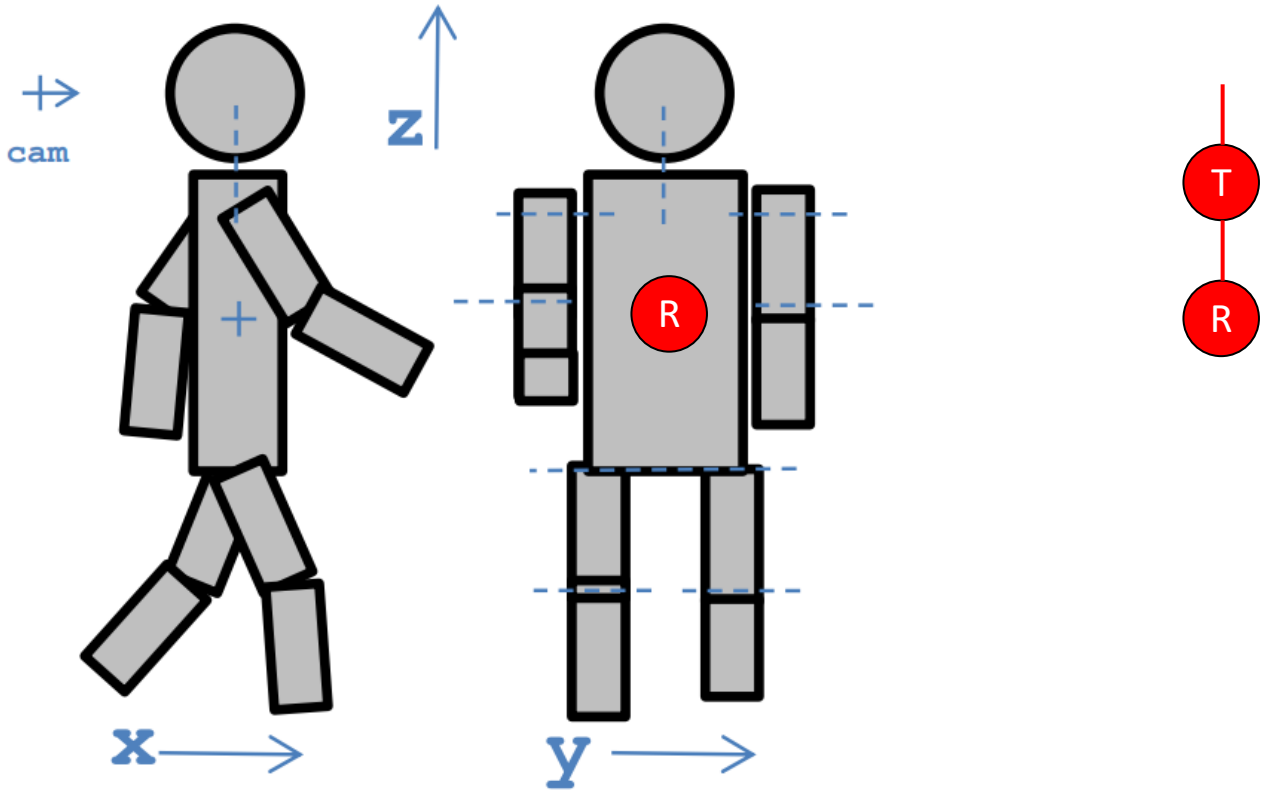
MIT EECS 6.837, Durand and Cutler



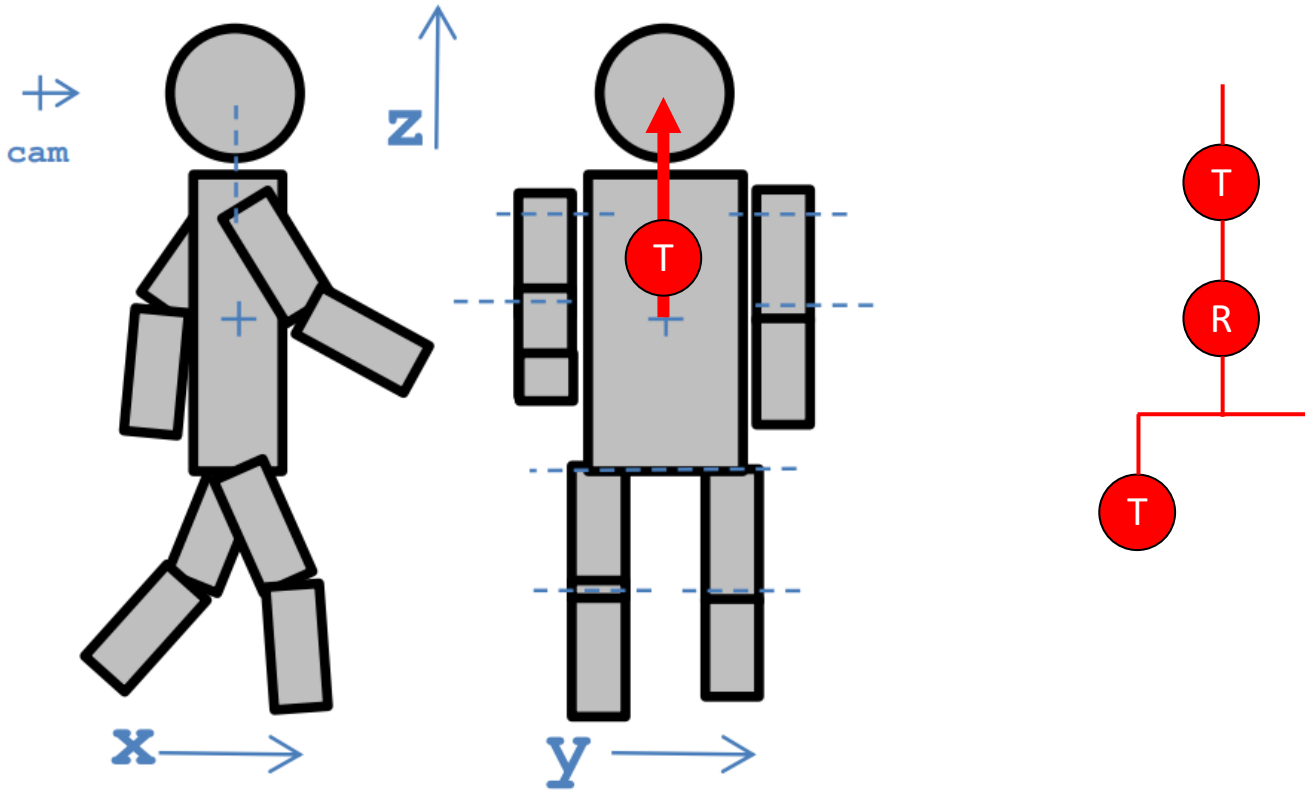
e. **[3.0]** Pretende-se implementar a árvore de desenho do boneco anterior, para o corpo, cabeça e **apenas um dos braços**. Use as funções `glutSolidCube(1.0)` para desenhar cada um dos paralelepípedos e `glutSolidSphere(modelo.corpo.raio_cabeca,10,10)` para desenhar a cabeça. Ambas as instruções desenham objetos centrados na origem atual. Construa a árvore de cena recorrendo a um ou mais dos blocos a seguir indicados:



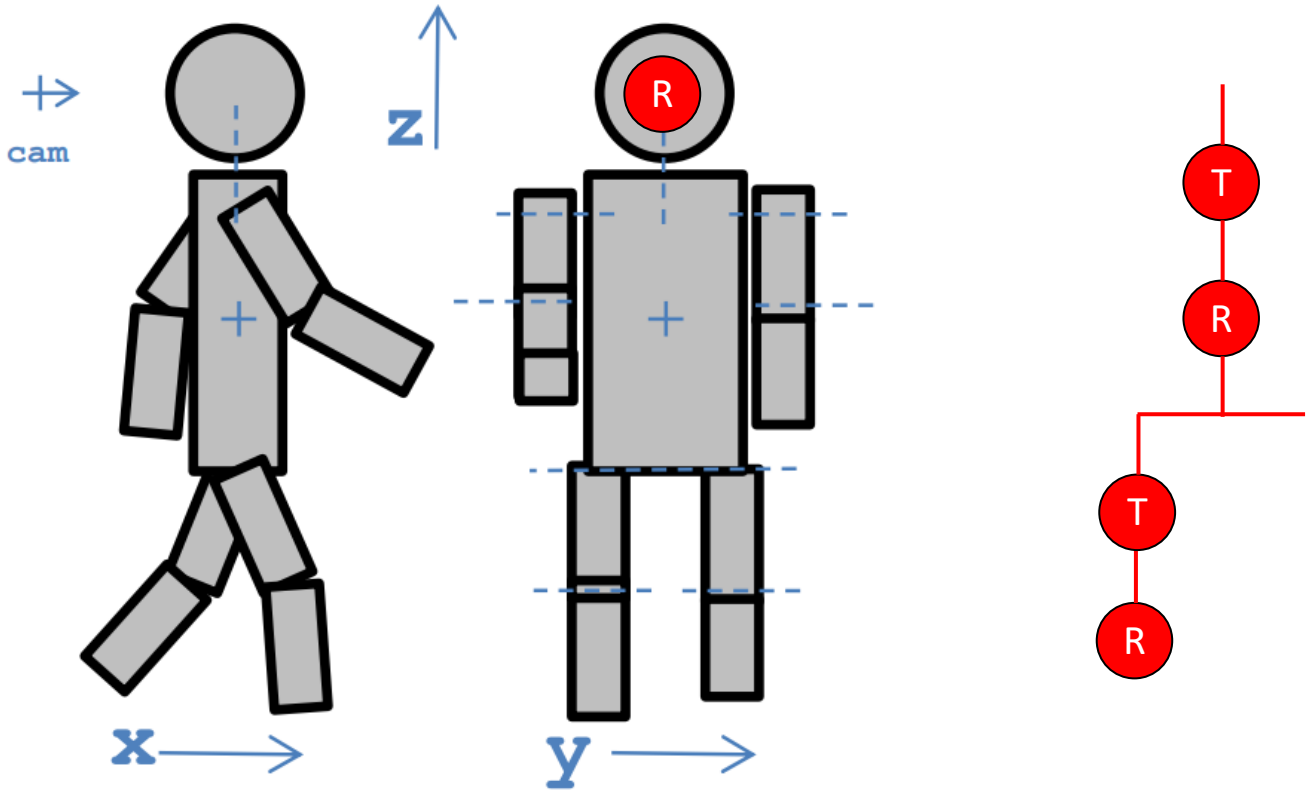
e. **[3.0]** Pretende-se implementar a árvore de desenho do boneco anterior, para o corpo, cabeça e **apenas um dos braços**. Use as funções `glutSolidCube(1.0)` para desenhar cada um dos paralelepípedos e `glutSolidSphere(modelo.corpo.raio_cabeca,10,10)` para desenhar a cabeça. Ambas as instruções desenham objetos centrados na origem atual. Construa a árvore de cena recorrendo a um ou mais dos blocos a seguir indicados:



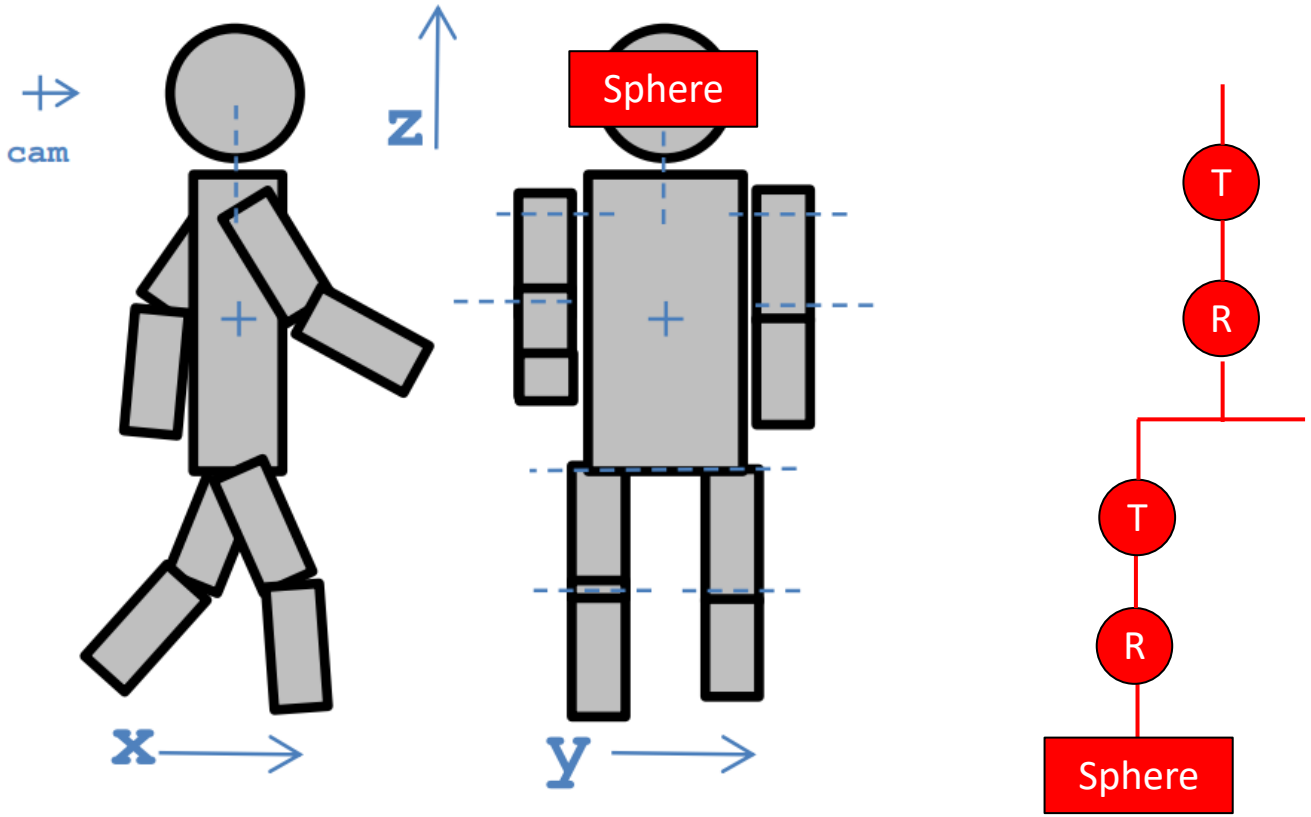
e. **[3.0]** Pretende-se implementar a árvore de desenho do boneco anterior, para o corpo, cabeça e **apenas um dos braços**. Use as funções `glutSolidCube(1.0)` para desenhar cada um dos paralelepípedos e `glutSolidSphere(modelo.corpo.raio_cabeca,10,10)` para desenhar a cabeça. Ambas as instruções desenham objetos centrados na origem atual. Construa a árvore de cena recorrendo a um ou mais dos blocos a seguir indicados:



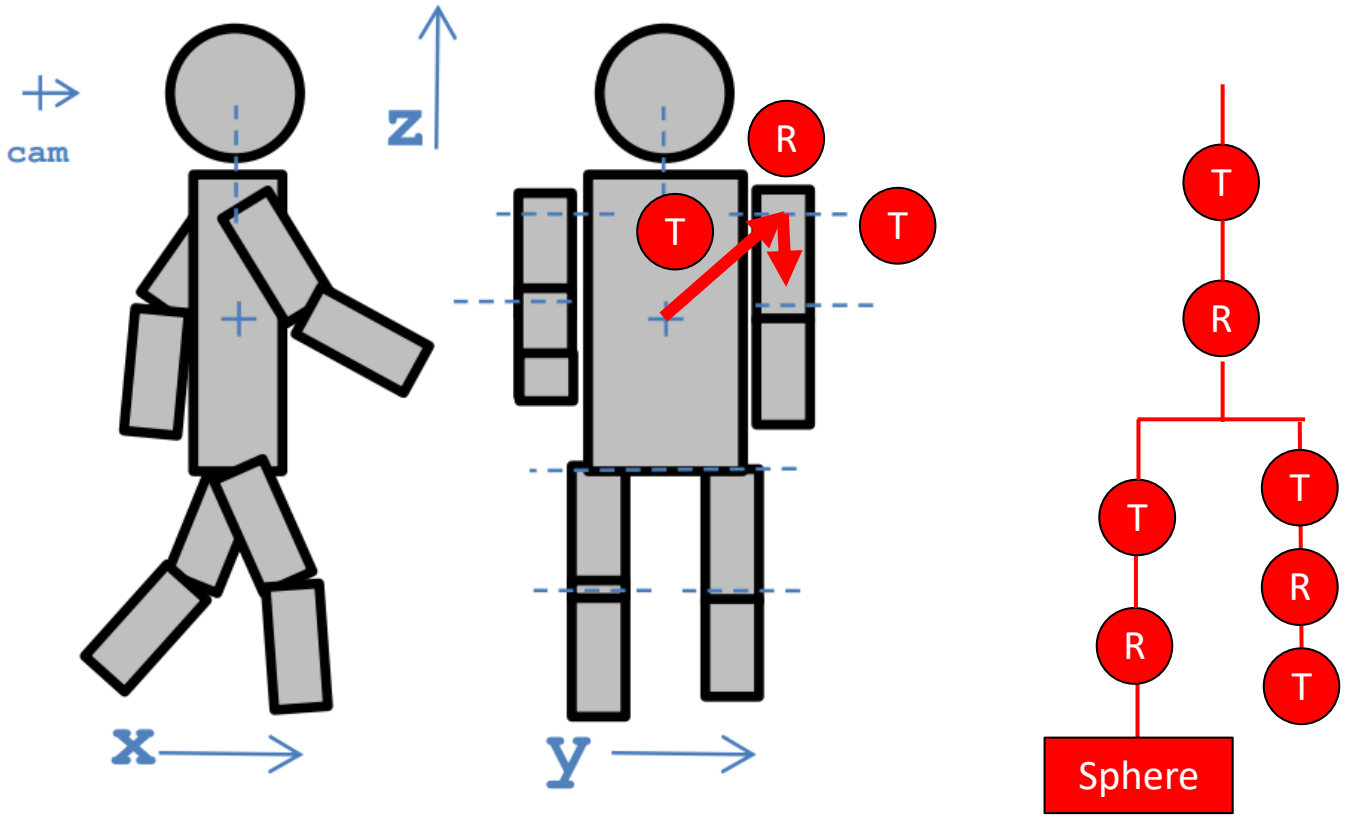
e. **[3.0]** Pretende-se implementar a árvore de desenho do boneco anterior, para o corpo, cabeça e **apenas um dos braços**. Use as funções `glutSolidCube(1.0)` para desenhar cada um dos paralelepípedos e `glutSolidSphere(modelo.corpo.raio_cabeca,10,10)` para desenhar a cabeça. Ambas as instruções desenham objetos centrados na origem atual. Construa a árvore de cena recorrendo a um ou mais dos blocos a seguir indicados:



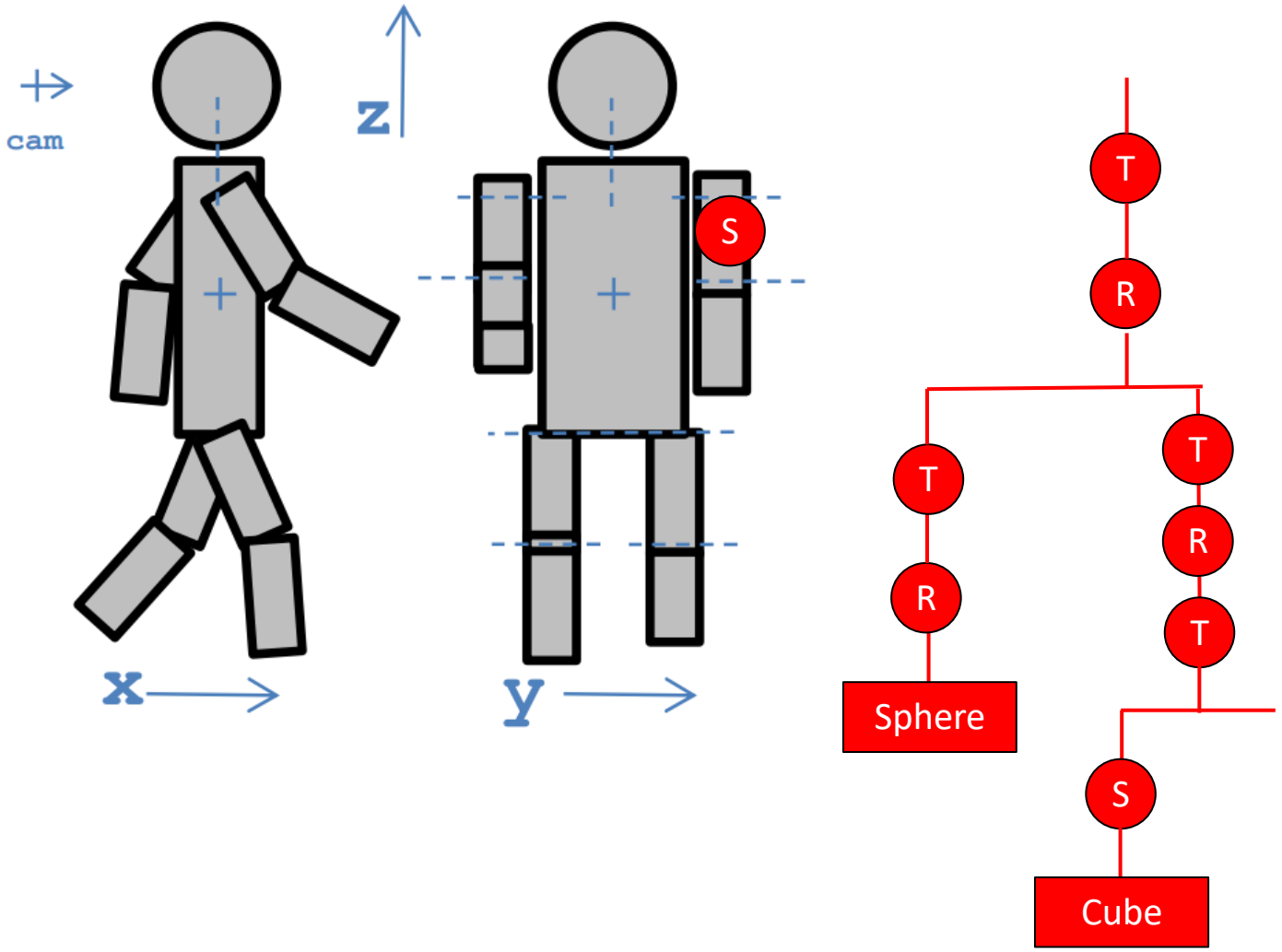
e. **[3.0]** Pretende-se implementar a árvore de desenho do boneco anterior, para o corpo, cabeça e **apenas um dos braços**. Use as funções `glutSolidCube(1.0)` para desenhar cada um dos paralelepípedos e `glutSolidSphere(modelo.corpo.raio_cabeca,10,10)` para desenhar a cabeça. Ambas as instruções desenham objetos centrados na origem atual. Construa a árvore de cena recorrendo a um ou mais dos blocos a seguir indicados:

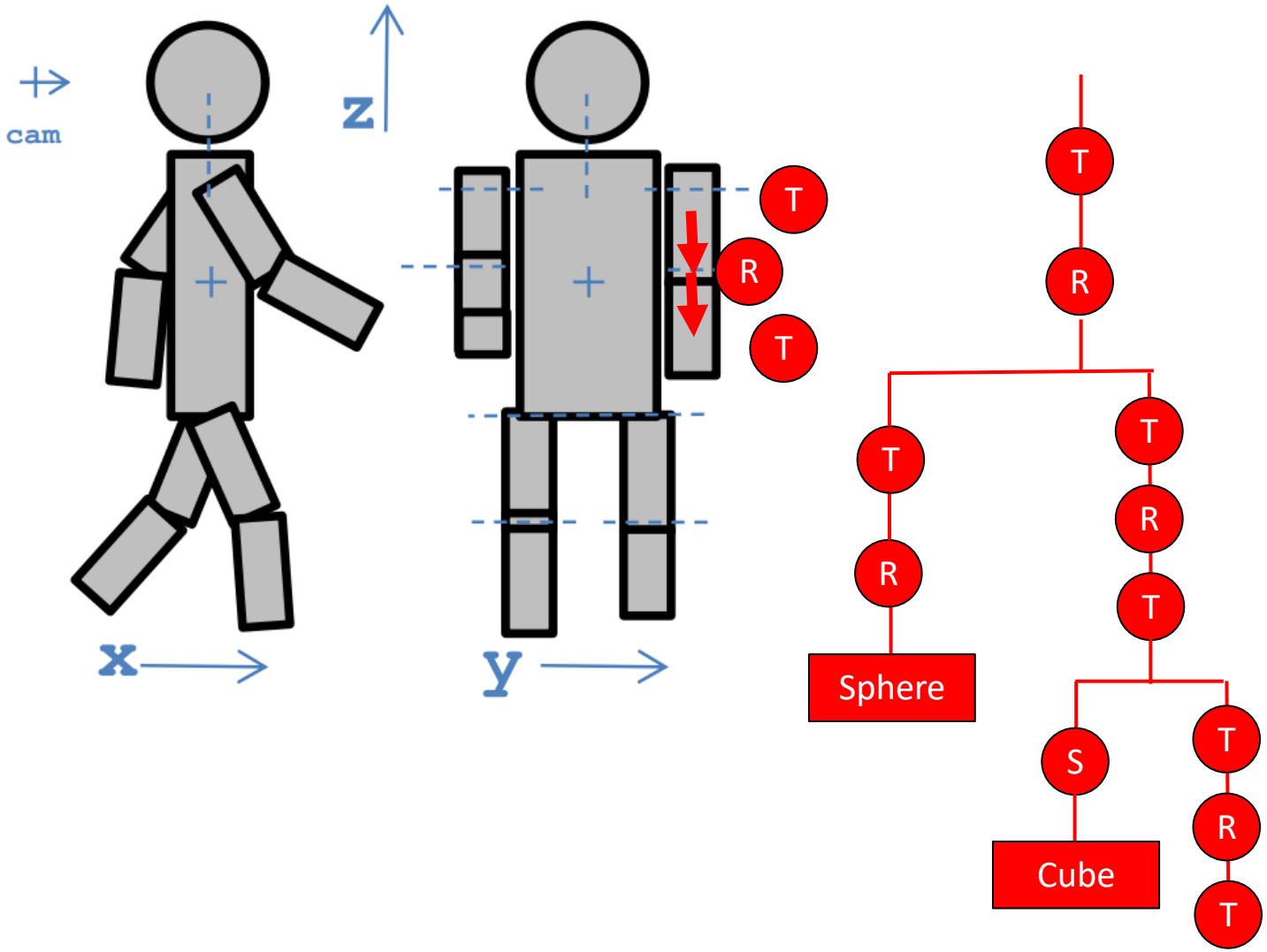


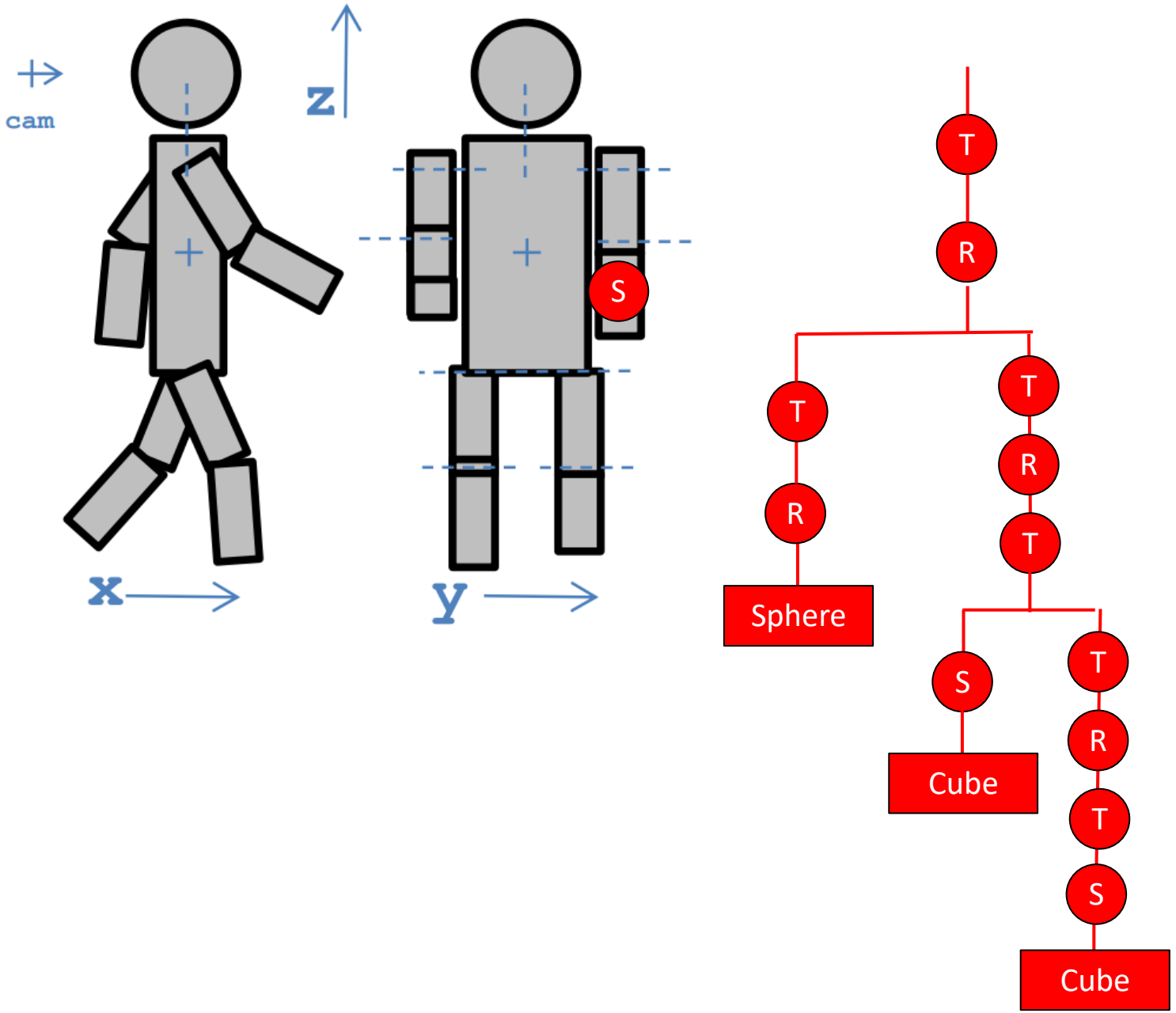
e. **[3.0]** Pretende-se implementar a árvore de desenho do boneco anterior, para o corpo, cabeça e **apenas um dos braços**. Use as funções `glutSolidCube(1.0)` para desenhar cada um dos paralelepípedos e `glutSolidSphere(modelo.corpo.raio_cabeca,10,10)` para desenhar a cabeça. Ambas as instruções desenham objetos centrados na origem atual. Construa a árvore de cena recorrendo a um ou mais dos blocos a seguir indicados:

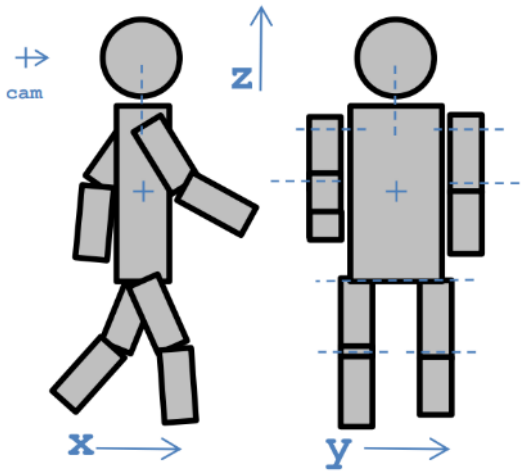


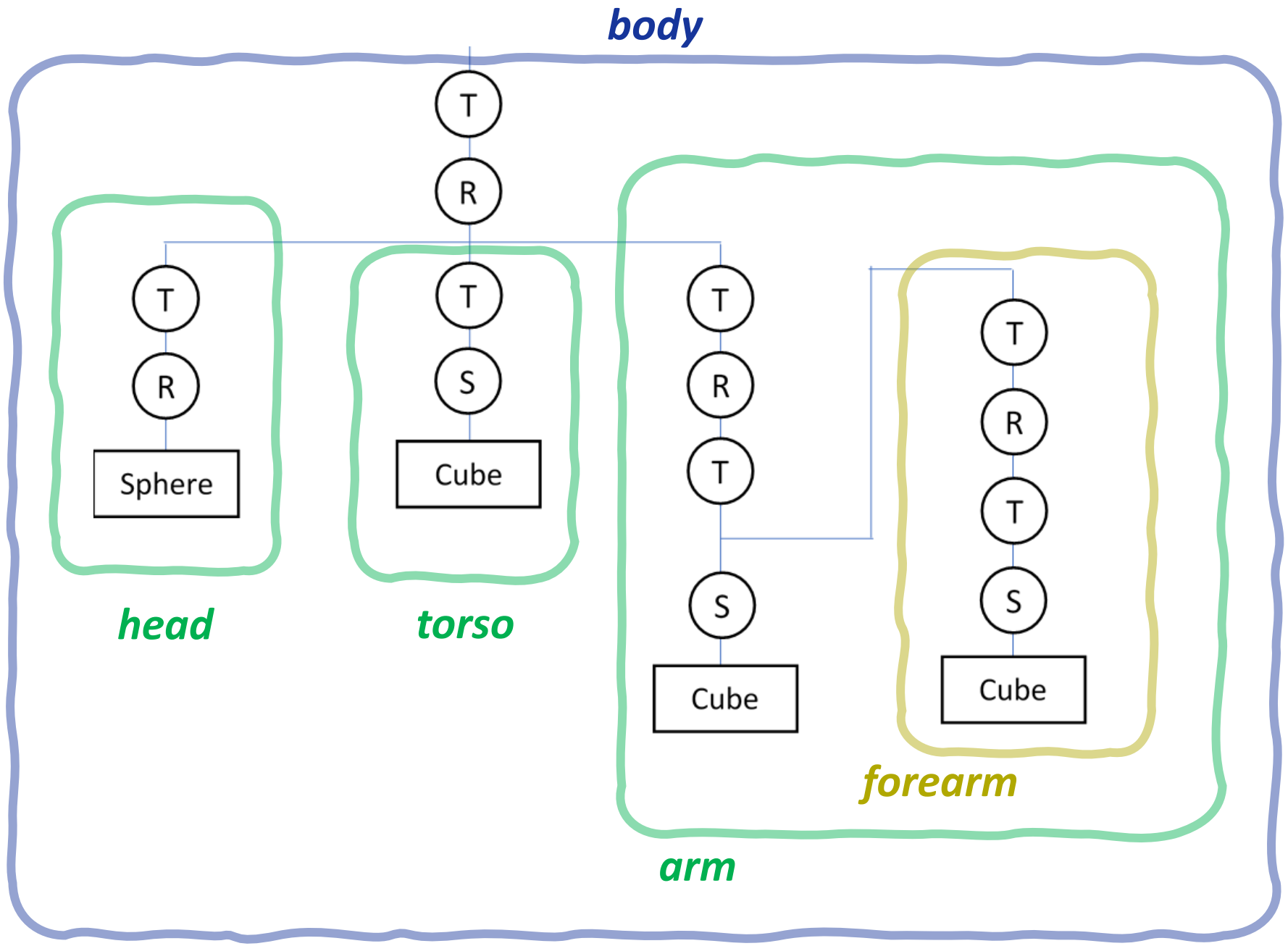
e. **[3.0]** Pretende-se implementar a árvore de desenho do boneco anterior, para o corpo, cabeça e **apenas um dos braços**. Use as funções `glutSolidCube(1.0)` para desenhar cada um dos paralelepípedos e `glutSolidSphere(modelo.corpo.raio_cabeca,10,10)` para desenhar a cabeça. Ambas as instruções desenham objetos centrados na origem atual. Construa a árvore de cena recorrendo a um ou mais dos blocos a seguir indicados:


cam
z
x
y
S
T
R
T
R
T
R
T
Sphere
S
Cube


cam
z
x
y
T
R
T
T
R
T
R
T
Sphere
S
T
R
T
Cube


cam
z
x
y
S
T
R
T
R
T
Sphere
T
R
T
S
Cube
T
R
T
S
Cube


cam
z
x
y


body
T
R
T
R
Sphere
head
T
S
Cube
torso
T
R
T
S
Cube
T
R
T
S
Cube
forearm
arm

Materials & Lights

On our previous examples the color of the material (**MeshBasicMaterial**) was used directly to display our object regardless of relative position of the surface or lights.

If we want to have a more realistic effect, we have to use a material that takes into account this details: **MeshPhongMaterial**

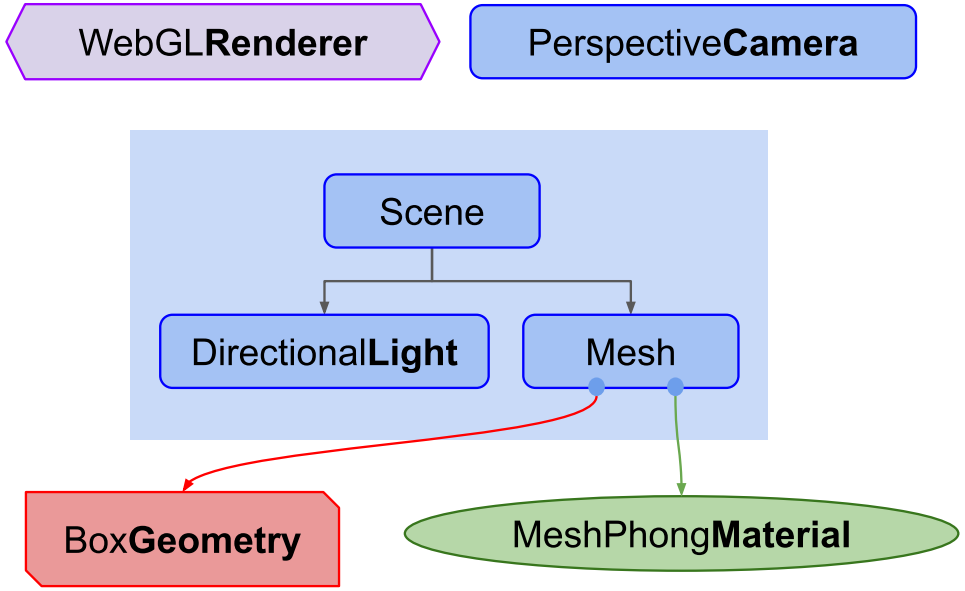
```
const material = new THREE.MeshPhongMaterial(
    {color: 0x44aa88}); // greenish blue
```

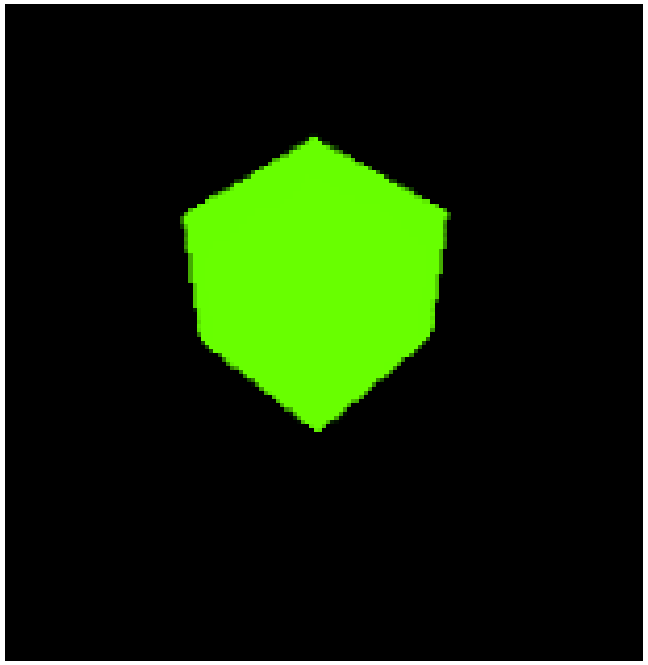
Materials & Lights

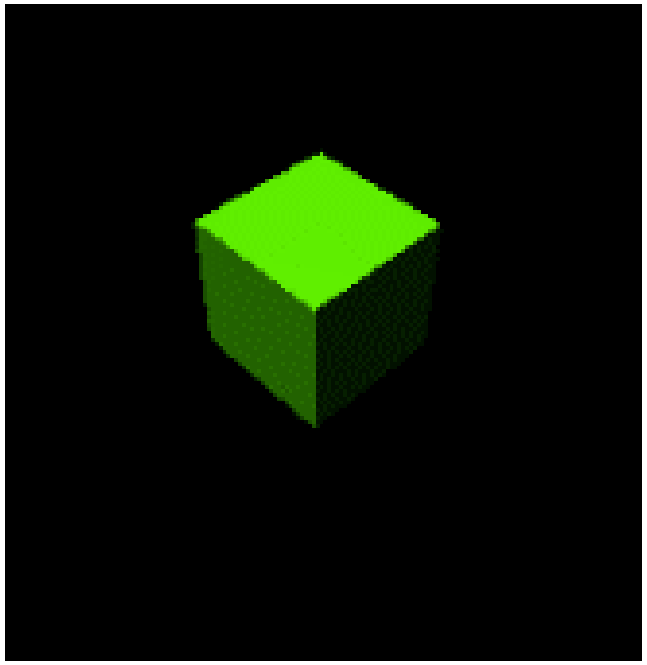
But since this material uses lights, we must add a new one, like this **DirectionalLight**:

```
const color = 0xFFFFFF;
const intensity = 1;
const light = new THREE.DirectionalLight(color, intensity);
light.position.set(-1, 2, 4);
scene.add(light);
```

Materials & Lights







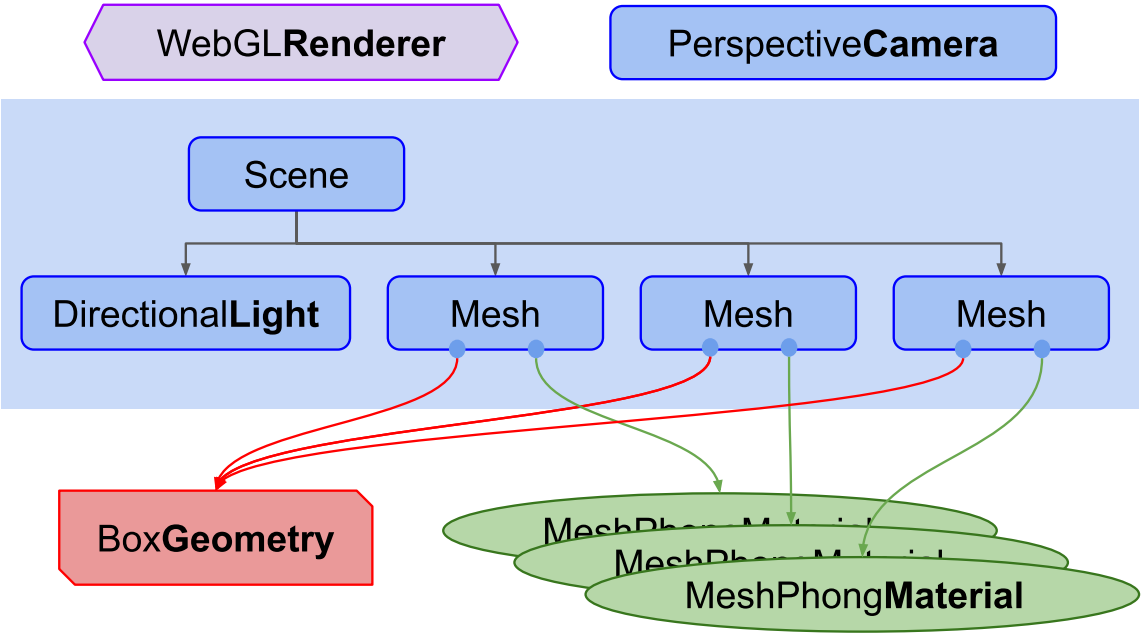
https://threejsfundamentals.org/threejs/lessons/threejs-fundamentals.html

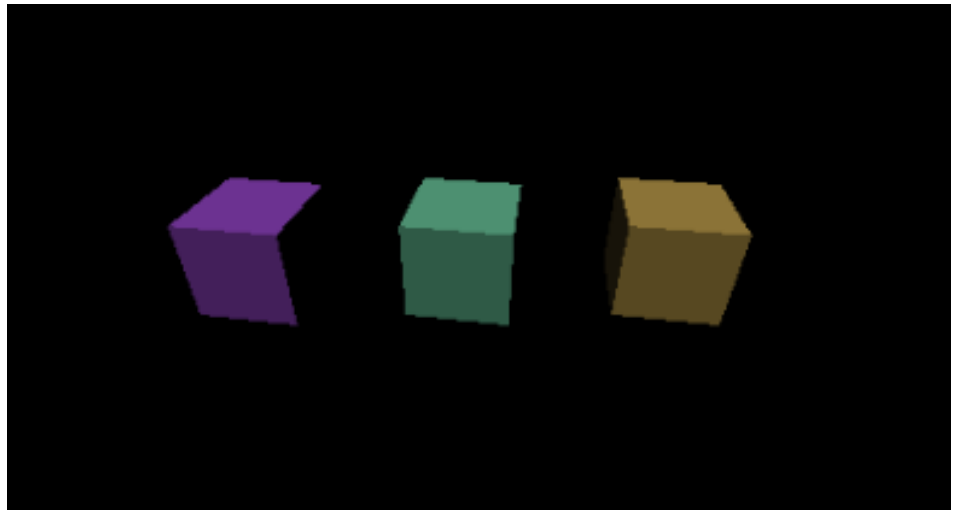
Share the geometry

```
function makeInstance(geometry, color, x) {
    const material = new THREE.MeshPhongMaterial({color: color});
    const cube = new THREE.Mesh(geometry, material);
    cube.position.x = x;
    cube.rotation.y = 10;
    cube.rotation.z = 30;
return cube;
}

scene.add(makeInstance(geometry, 0x44aa88, 0));
scene.add(makeInstance(geometry, 0x8844aa, -2));
scene.add(makeInstance(geometry, 0xaa8844, 2));
```

Note:
For this to work with the previous code, you must remove the **cube** references in the **animate** function.

Share the geometry





https://threejs.org/manual/#en/fundamentals

Quick Shadows

WebGL supports shadow maps for complex shadows.

Start enabling the shadow feature in the renderer:

```
const renderer = new THREE.WebGLRenderer({canvas});
renderer.shadowMap.enabled = true;
```

and for each relevant light:

```
const light = new THREE.DirectionalLight(color, intensity);
light.castShadow = true;
```

Shadows

We can now set each Mesh that we want to only *receive* shadows (e.g. the floor):

```
const mesh = new THREE.Mesh(planeGeo, planeMat);
mesh.receiveShadow = true;
```

or to both *receive and cast* shadows:

```
const mesh = new THREE.Mesh(cubeGeo, cubeMat);
mesh.castShadow = true;
mesh.receiveShadow = true;
```

Shadows

And that’s the basics for Shadows.

Some details that we didn’t look here (and will come back later):

- Shadows cast by a Directional Light have a frustum like Orthogonal Projection with left/right/near/far… if anything falls out of that volume a shadow is not cast.
- Shadows are textures with a maximum resolution, if the frustum is too large, or the camera is zooming in the shadow area, we will start seeing pixelization due to this limitation.

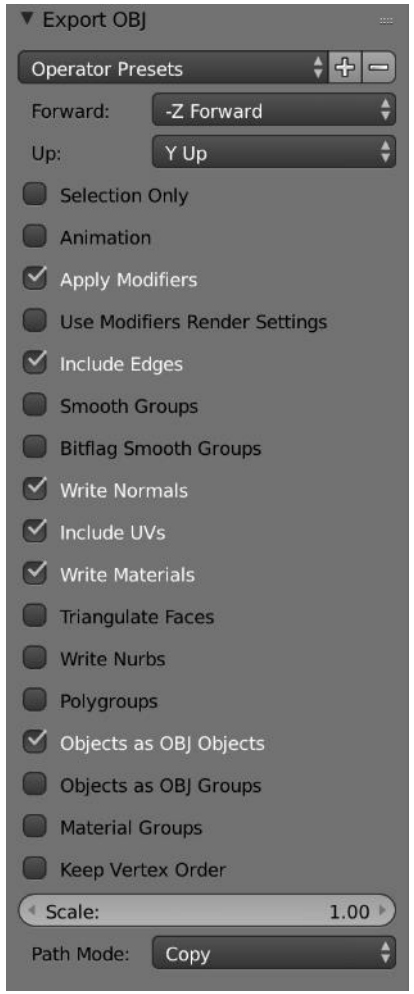
Load external 3D objects

Load an .OBJ

Starting with a blender file

Open the file in Blender and export as an .OBJ

(Blender is a free full-featured 3D editor)



https://threejs.org/manual/#en/load-obj

Load external 3D objects

First include the OBJLoader

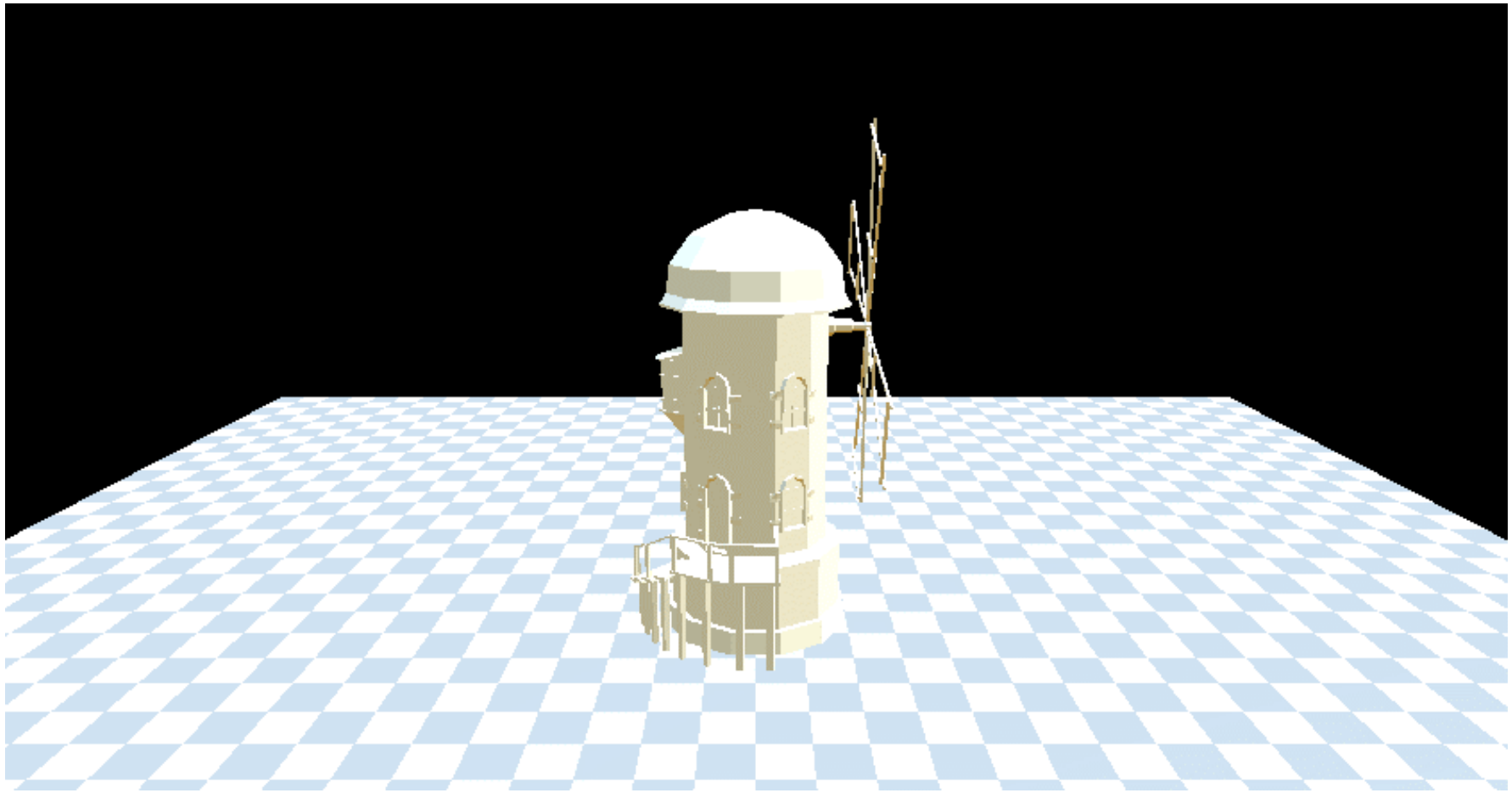
```
import {OBJLoader} from './three/examples/jsm/loaders/OBJLoader.js';
```

Then load and display the file

```
const objLoader = new OBJLoader();
objLoader.load('models/windmill/windmill.obj', (root) => {
    scene.add(root);
});
```

https://threejs.org/manual/#en/load-obj

Load external 3D objects



https://threejs.org/manual/#en/load-obj

Adding Materials & Textures

- Blender exported and .OBJ file and a .MTL file
- The .MTL file contains description of the materials and textures
- However… the texture files are not automatically exported by Blender
- To export the files in Blender go to:
  - File → External Data → Unpack All into Files
- And then choose
  - Write Files to Current Directory
- Now you should have the .jpg files used as textures in the materials
- Put these files in the same directory as the .OBJ and .MTL files

https://threejs.org/manual/#en/load-obj

Adding Materials & Textures

Include the MTLLoader

```
import * as THREE from './three/build/three.module.js';
import {OBJLoader} from
      './three/examples/jsm/loaders/OBJLoader.js';
import {MTLLoader} from
      './three/examples/jsm/loaders/MTLLoader.js';
```

https://threejs.org/manual/#en/load-obj

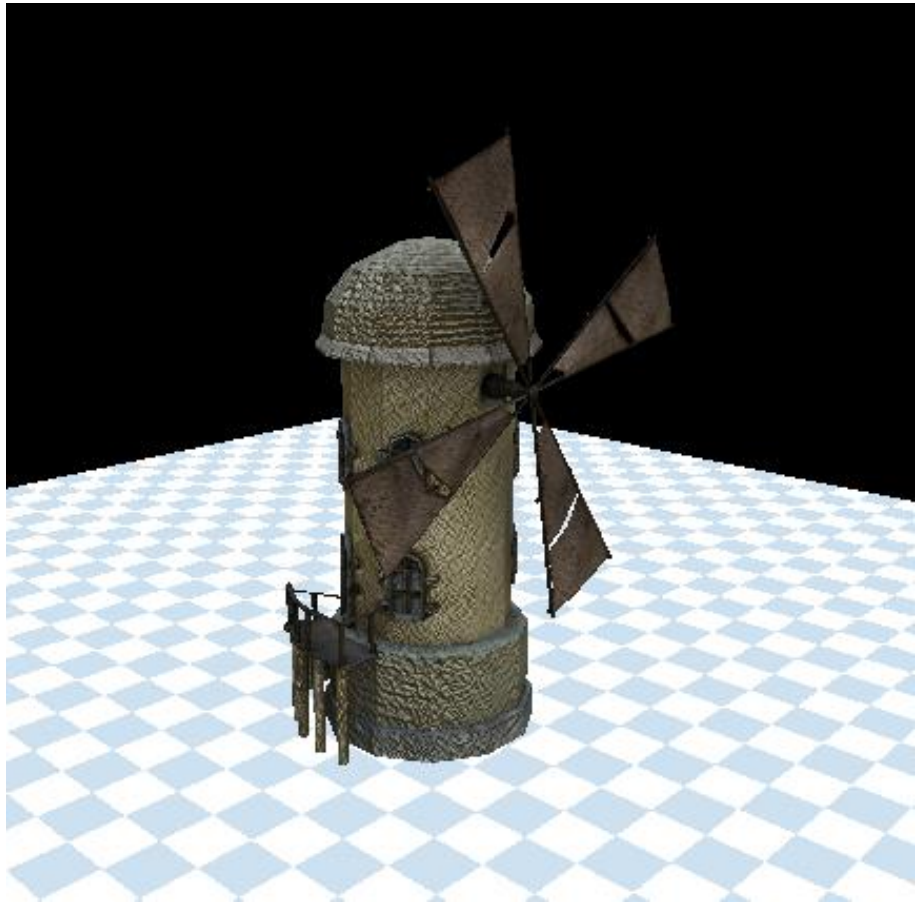
Adding Materials & Textures

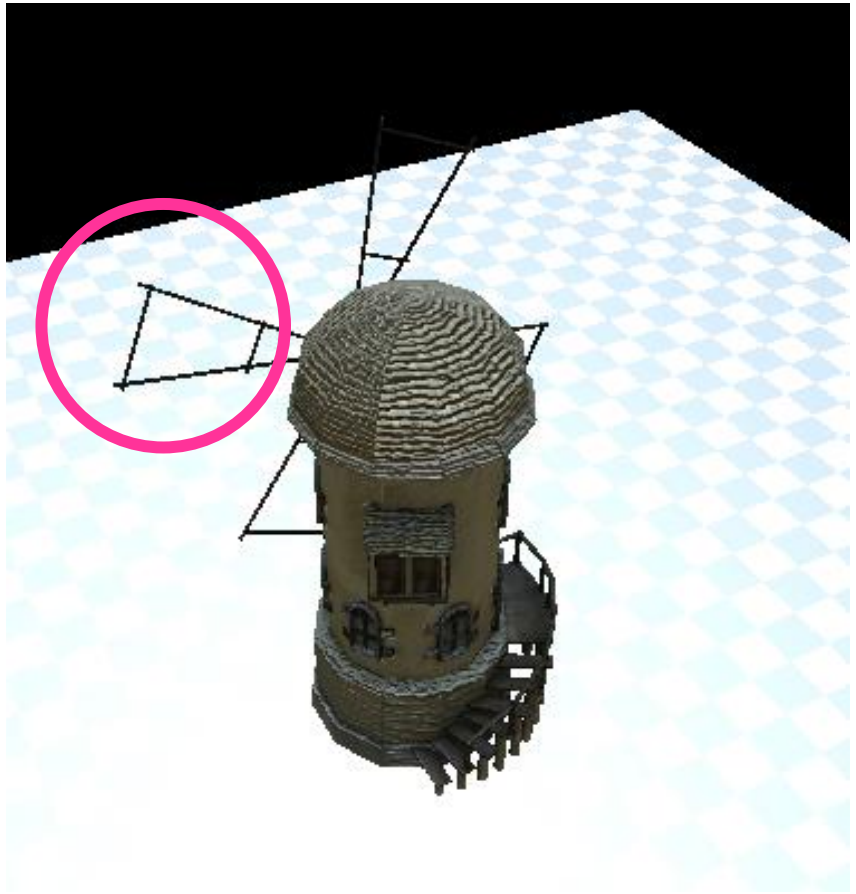
And load the MTL File

```
const mtlLoader = new MTLLoader();
mtlLoader.load('models/windmill/windmill.mtl', (mtl) => {
    mtl.preload();
    objLoader.setMaterials(mtl);
    objLoader.load('models/windmill/windmill.obj', (root) => {
        scene.add(root);
    });
});
```

https://threejs.org/manual/#en/load-obj

Adding Materials & Textures





https://threejs.org/manual/#en/load-obj

Adding Materials & Textures

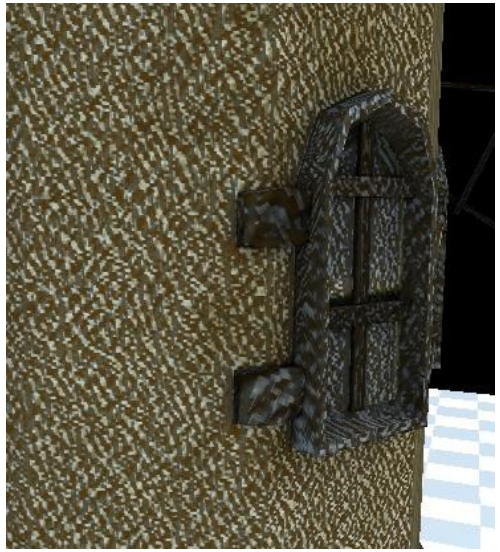
The problem here is that the material used is one-sided by default, we can correct that in the code, by example:

```
const mtlLoader = new MTLLoader();
mtlLoader.load('resources/models/windmill/windmill.mtl', (mtl) => {
    mtl.preload();
    mtl.materials.Material.side = THREE.DoubleSide;
    …
```

https://threejs.org/manual/#en/load-obj

Adding Materials & Textures

- Another issue is that the bitmap file used for normals is not being loaded correctly… MTLLoader expects that the file is referenced using keyword “norm”, but the .MTL file from blender uses “map_Bump”.
- Edit the .MTL file using a text editor to fix this



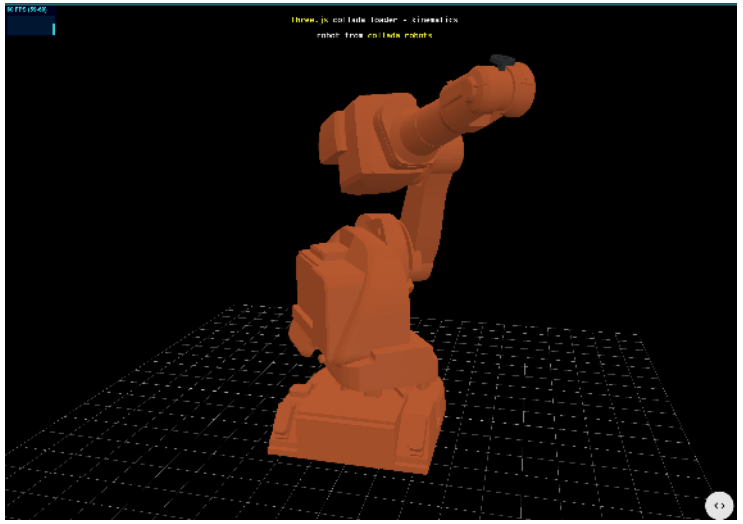
https://threejs.org/manual/#en/load-obj

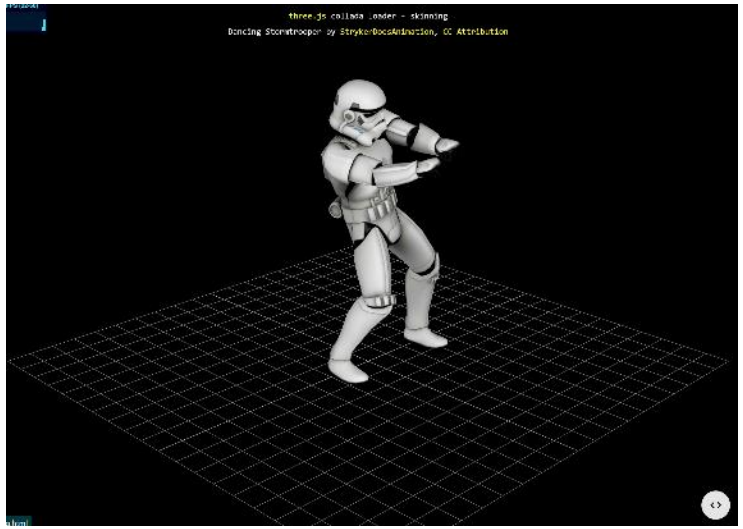
Loading a .GLTF File

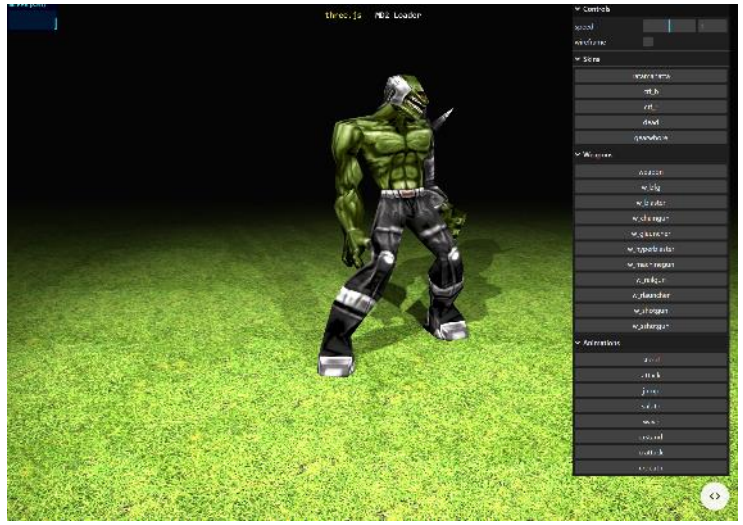
- gLTF files combine all the data for the 3D model in a standard way in a single file (you will have to sort out some things, either way).
- You can check out an example here: https://threejs.org/manual/#en/load-gltf

Examples of loaders

- There is support for much more loaders, including animations, and interaction, check the examples here: https://threejs.org/examples/?q=loader







Let there be Light

In order to use complex materials, we must have a light

Let’s use a simple **PointLight** for now.

```
const light = new THREE.PointLight( 0xff0000 );
light.position.set( -1, 2, 4 );
scene.add( light );
```

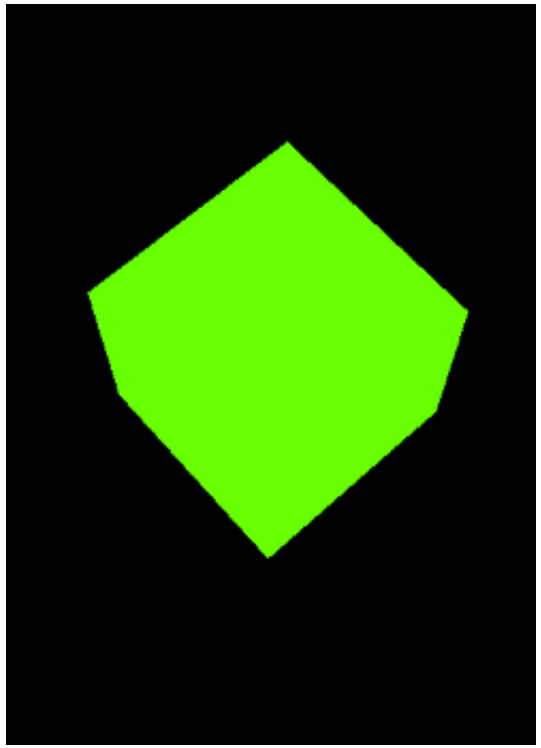
Basic Material

MeshBasicMaterial

A single solid color

Can be transparent

```
var material = new THREE.MeshBasicMaterial(
    { color: 0x00ff00  }
);
```



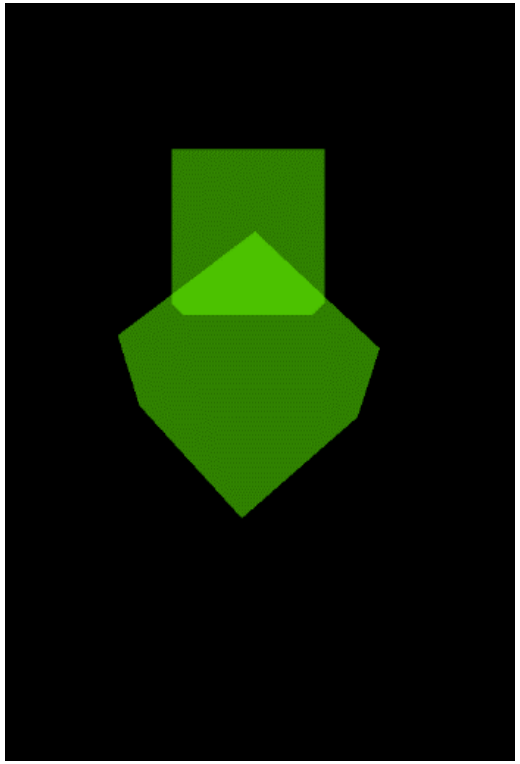
Basic Material

MeshBasicMaterial

A single solid color

Can be transparent

```
var material = new THREE.MeshBasicMaterial({
    color: 0x00ff00,
    transparent: true,
    opacity: 0.5
});
```



Lambert Material

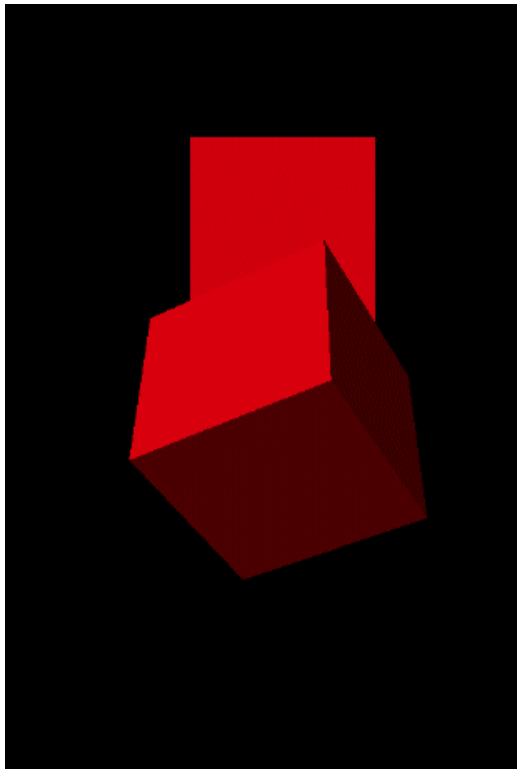
MeshLambertMaterial

Enables shading with a dull surface (must have lights!)

Can adjust color

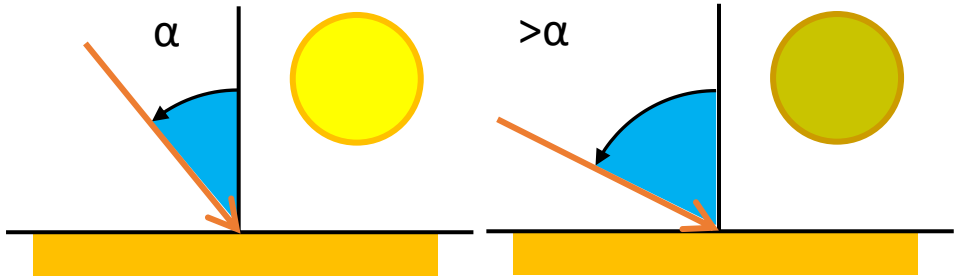
Can be emissive

```
var material = new THREE.MeshLambertMaterial({
    color: 0xff0000
});
```

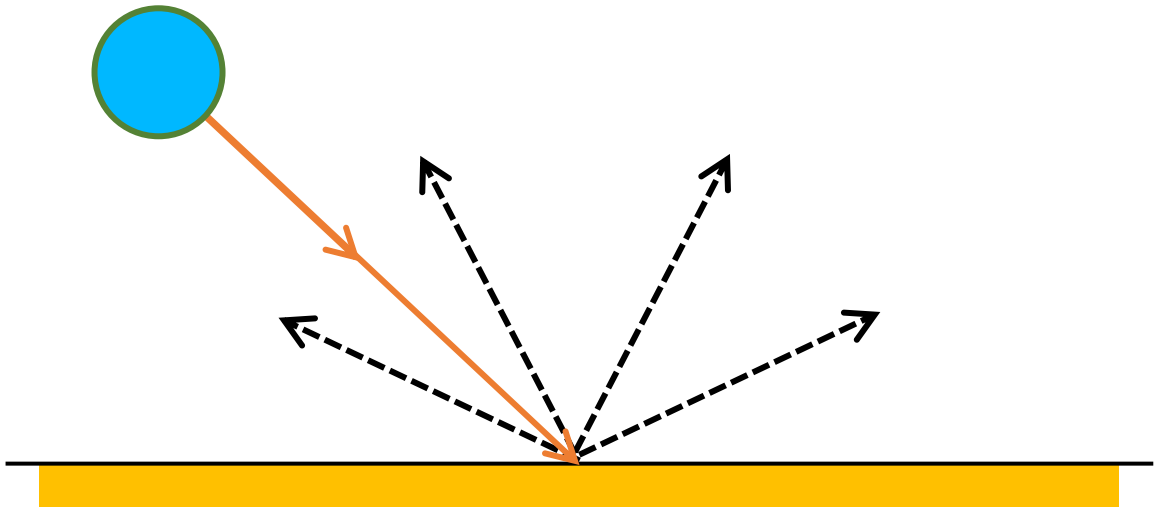


Lambert Material

Diffuse Light Effect



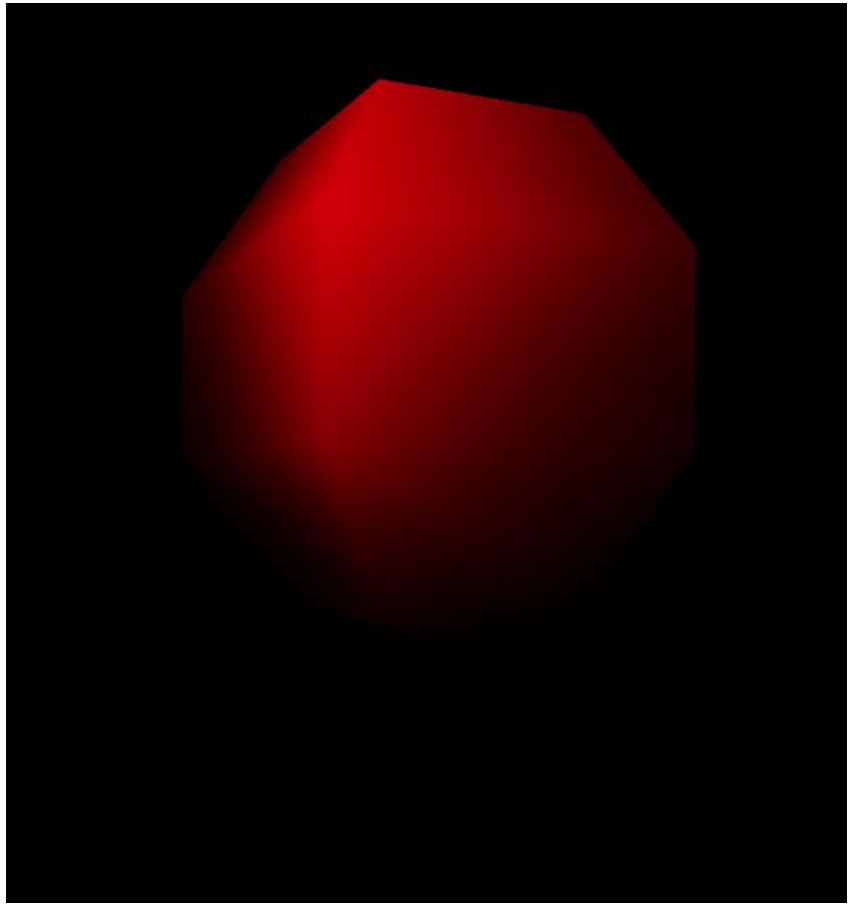
The result depends only on the **angle** between the light and the **normal**.



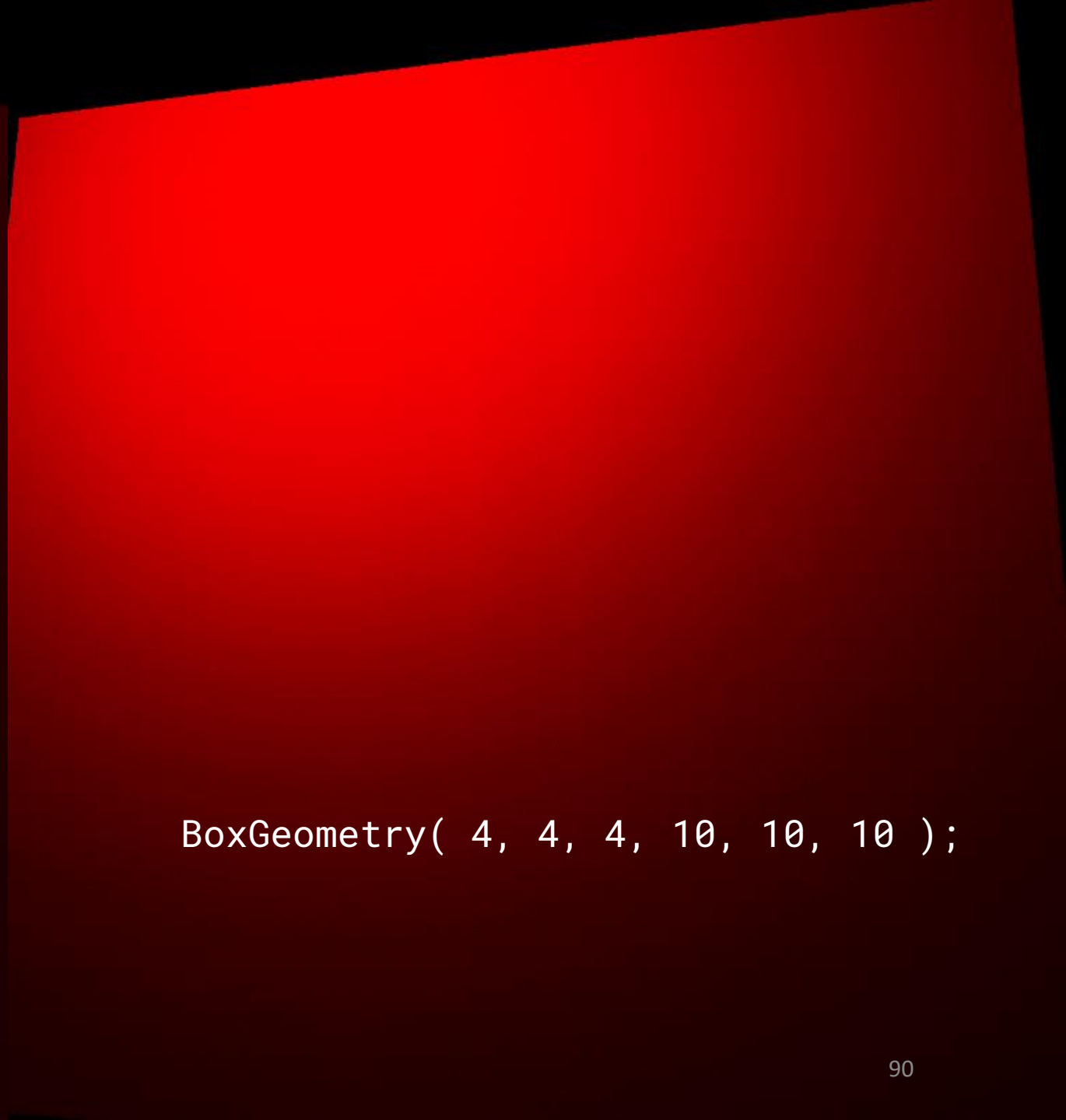
All observers see the same result regardless off their position.

Vertex Shading

- In previous versions of Three.JS (up to r143) the default shading in Lambert Material was calculated per vertex (in a *vertex shader*)
- The result was fast, but quality depends heavily on the position and number of vertexes in the surface, and the distance from the lights to the surface



```
BoxGeometry( 4, 4, 4, 1, 1, 1 );
```



```
BoxGeometry( 4, 4, 4, 10, 10, 10 );
```

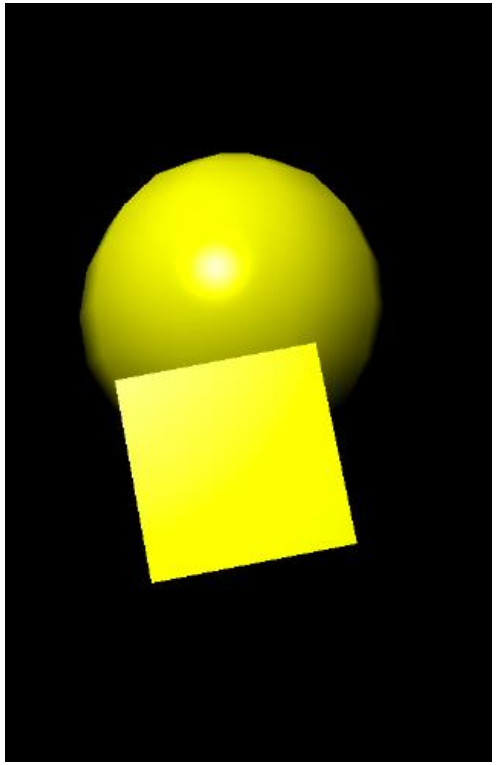
Phong Material

MeshPhongMaterial

Add “plastic” effect

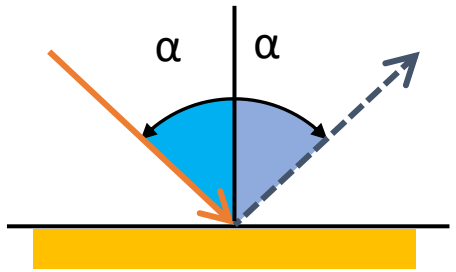
Adds specular color and shininess

```
var material = new THREE.MeshPhongMaterial({
    color: 0xffff00,
    shininess: 100
});
```

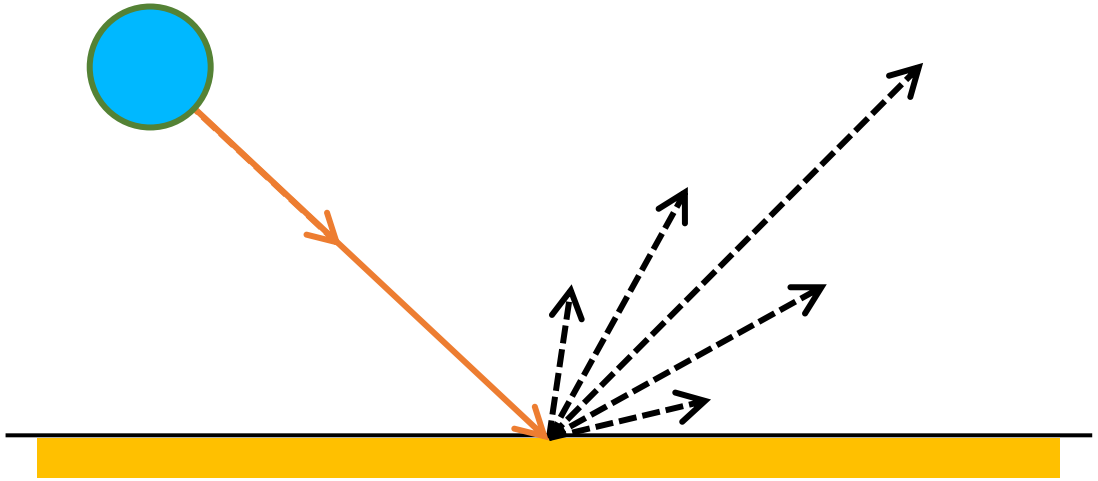


Phong Material

Specular Light Effect



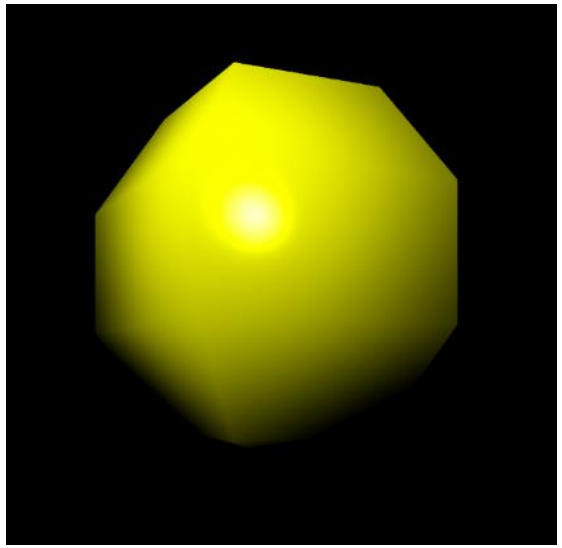
The result depends on the **angle** between the light and the **normal**.

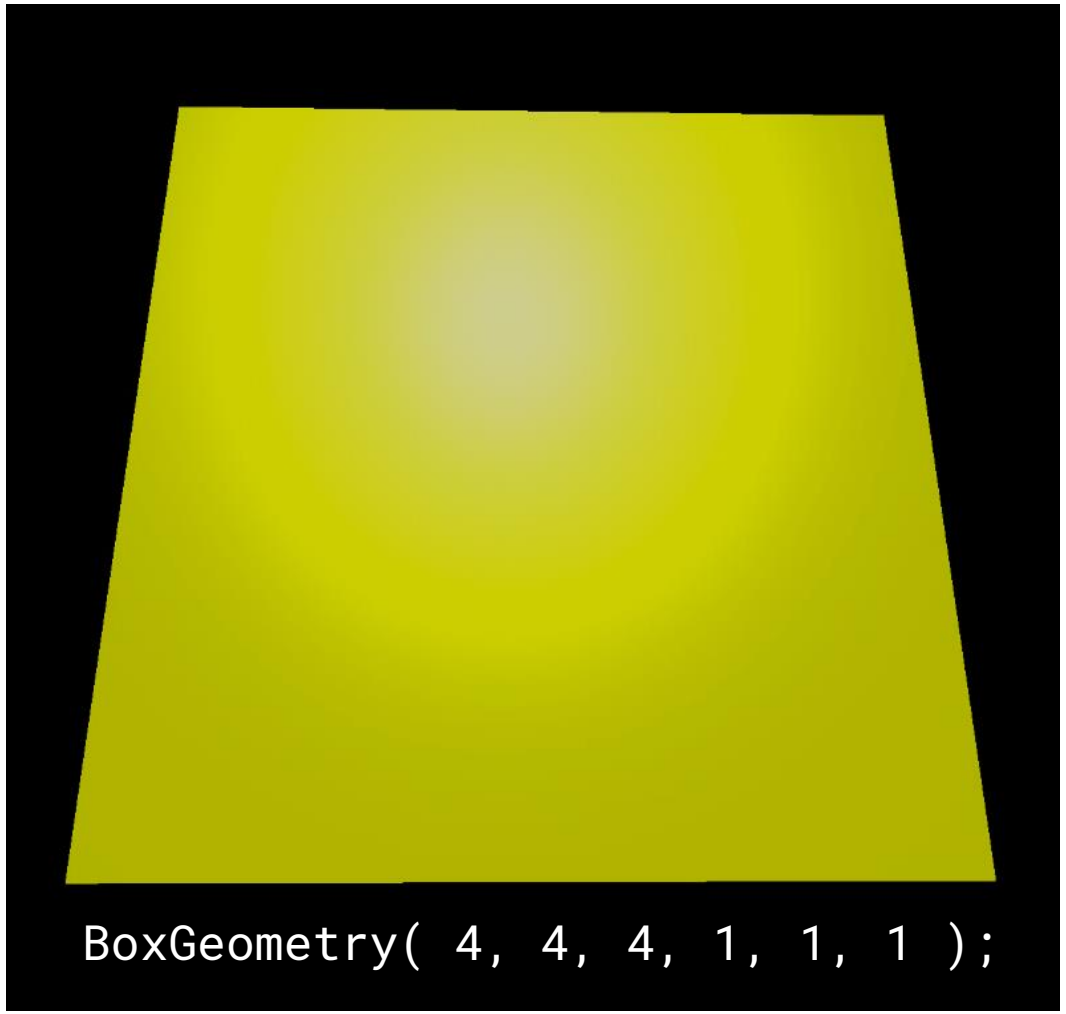


The result also depends on the position of the observer

Phong Material

- Shading is calculated per pixel (in a *fragment shader* a.k.a. *pixel shader*)
- This is more accurate and does not depend on having the surfaces sub-divided, but has a small cost on performance





Standard Material

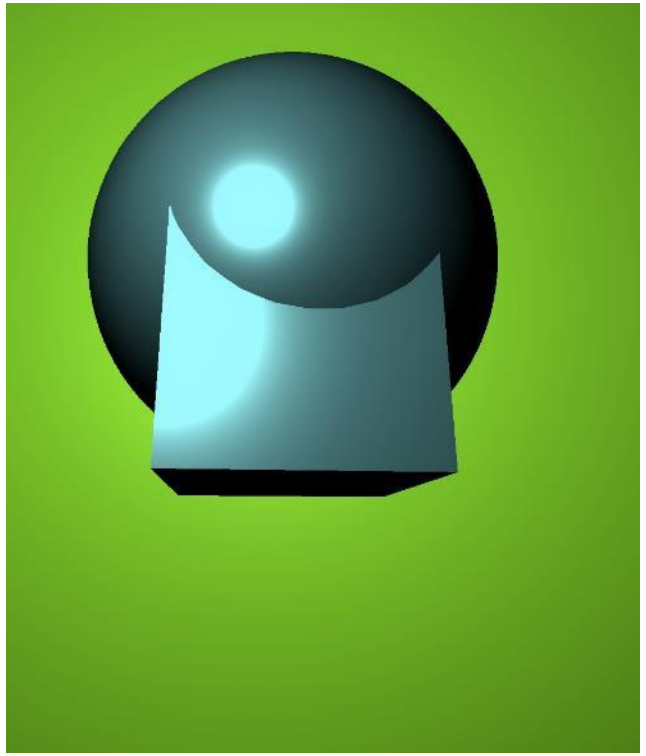
MeshStandardMaterial

A physical based rendering material

More computationally intensive than Phong and Lambert

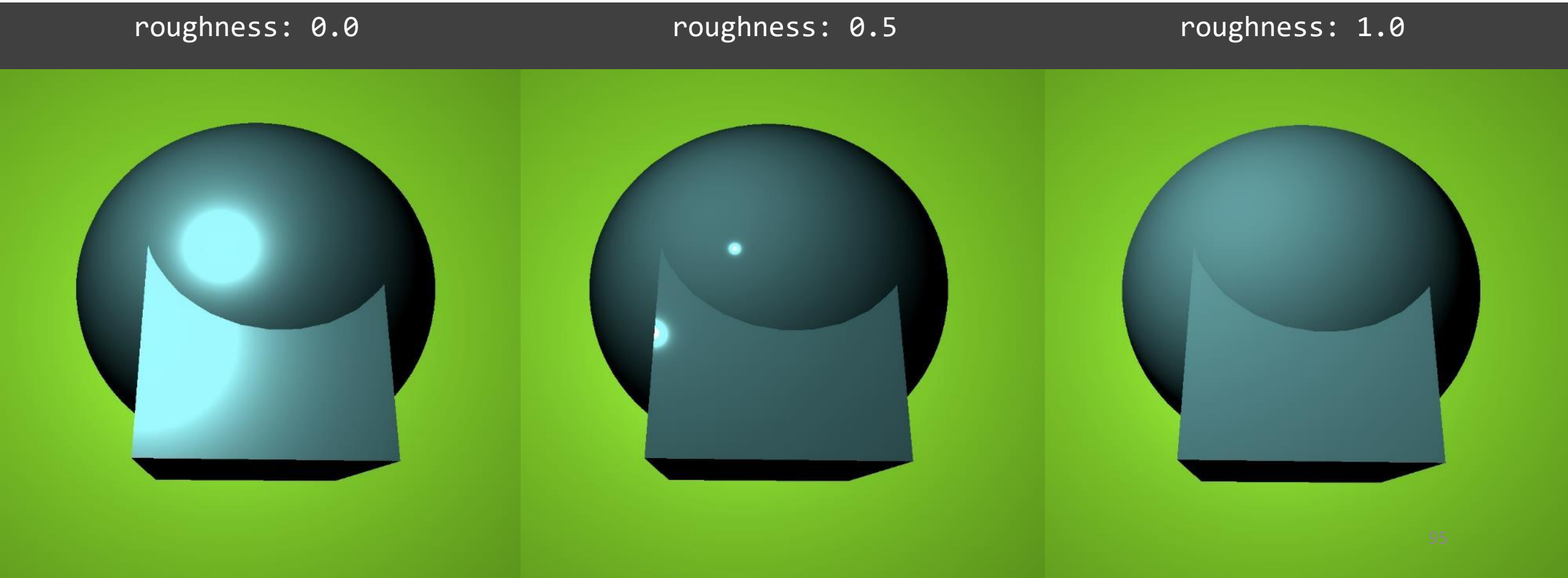
Adds **metalness** and **roughness** characteristics

```
var material = new THREE.MeshStandardMaterial({
    color: 0x00ffff,
    metalness: 0.5,
    roughness: 0.4
});
```



Standard Material

MeshStandardMaterial



Standard Material

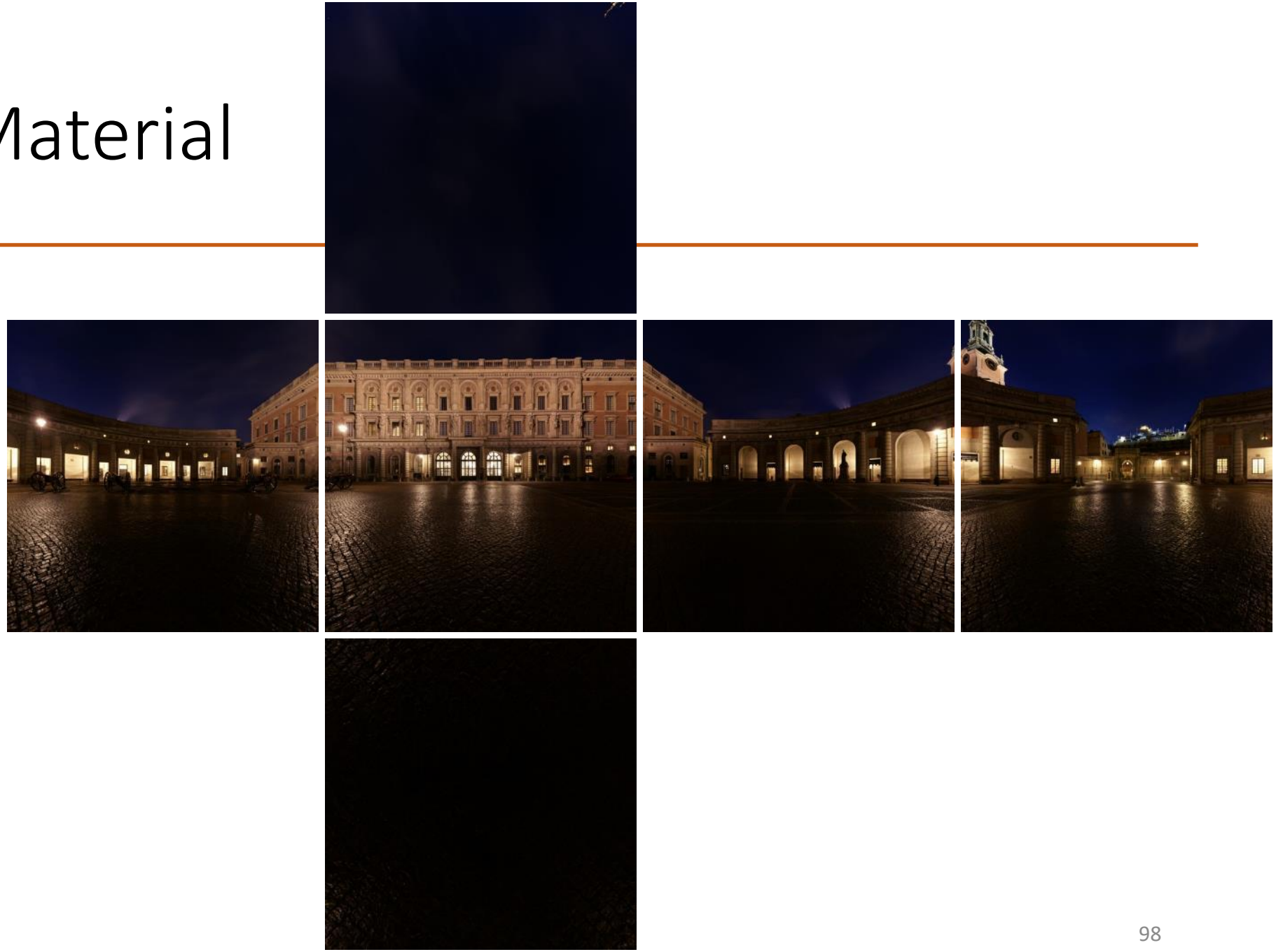
MeshStandardMaterial

Standard Material

- Add an **Environment Map**
- **Environment mapping**, or **reflection mapping**, is an efficient image-based lighting technique for approximating the appearance of a reflective surface by means of a precomputed texture. The texture is used to store the image of the distant environment surrounding the rendered object.
- The most used is **Cube Mapping** where the environment is projected in the 6 faces of a cube

Standard Material

- **Cube Mapping**



Standard Material

Limitations of an Environment Map

- All radiance incident on the object comes from an infinite distance.
  - Parallax effect will not work on the reflection, and this will be noticeable for nearer objects
- The object is convex, so that It does not contain any self-interreflections.
  - The object will not appear in the reflection, only the environment does

Standard Material

Environment Map

```
const path = "";
const format = '.jpg';
const urls = [
    path + 'px' + format, path + 'nx' + format,
    path + 'py' + format, path + 'ny' + format,
    path + 'pz' + format, path + 'nz' + format
];

const reflectionCube = new THREE.CubeTextureLoader().load( urls );
reflectionCube.mapping = THREE.CubeRefractionMapping;
```

Standard Material

Environment Map

```
scene.background = reflectionCube; //optional…

var material = new THREE.MeshStandardMaterial({
    color: 0x80ffff,
    metalness: 1, roughness: 0.0,
    envMapIntensity:1, envMap: reflectionCube
});
```

metalness: 0.9, roughness: 0.01

Ambient Light

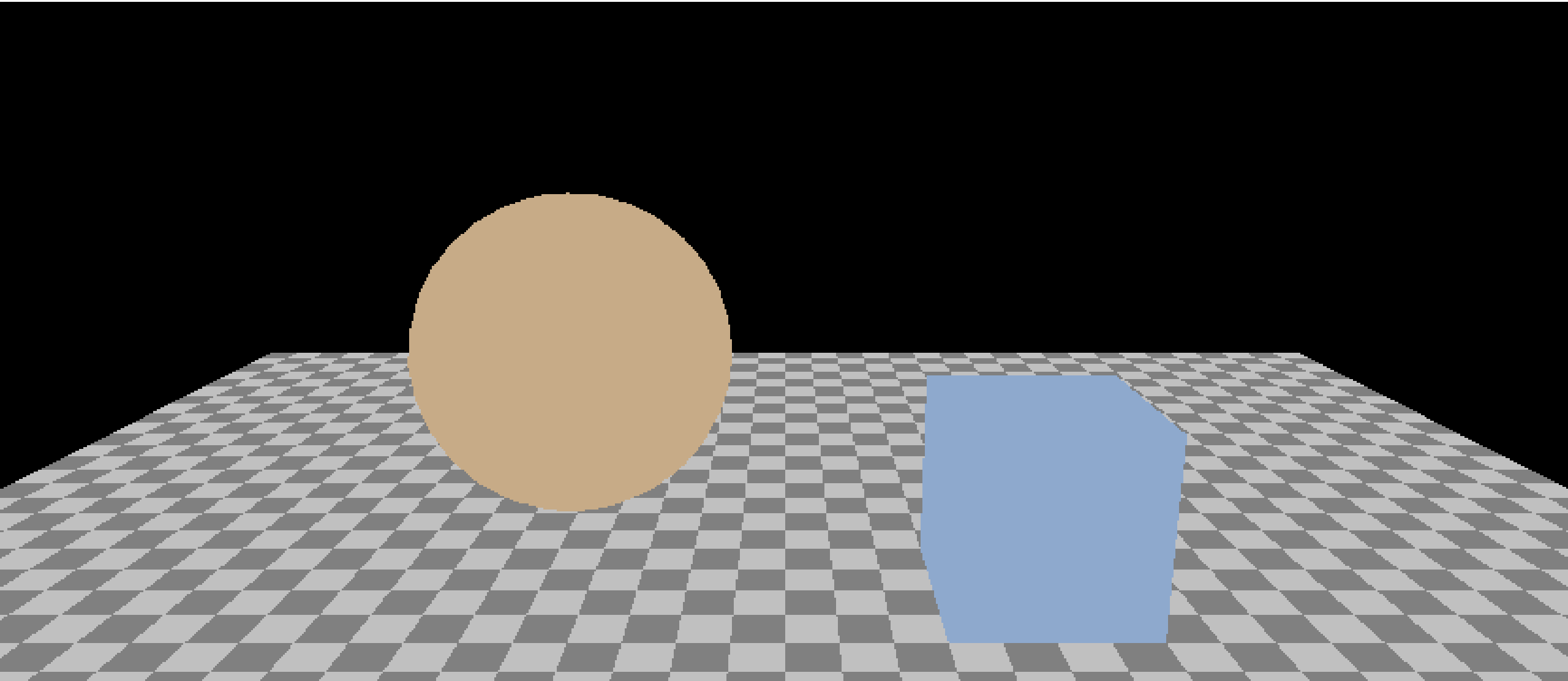
Adds environment light to everything regardless of position and direction.

It is used to avoid absolute black areas in the scene

```
const color = 0xFFFFFF;
const intensity = 1;
const light = new THREE.AmbientLight(
    color, intensity
);
scene.add(light);
```

The following examples can be found with interactive controls at:
https://threejs.org/manual/#en/lights

Ambient Light

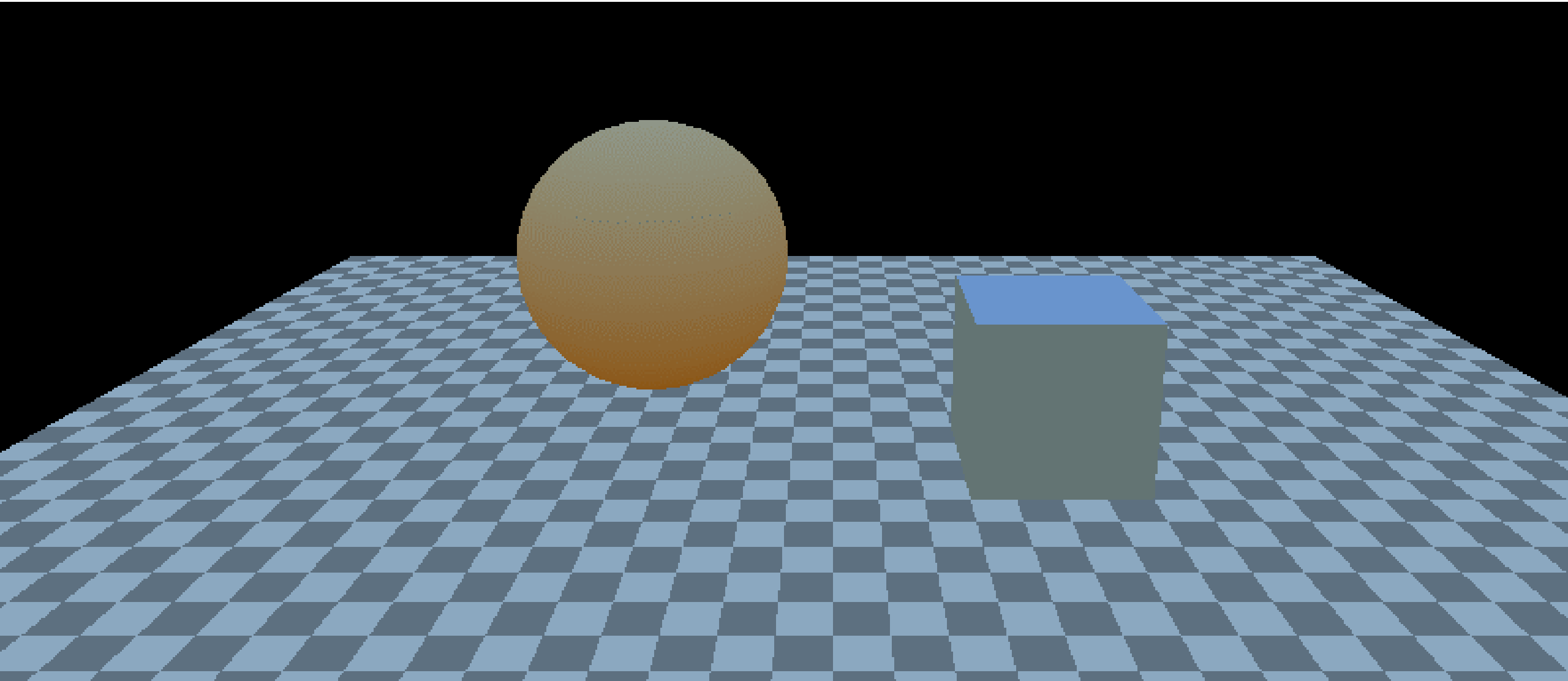


Hemishpere Light

Takes a sky color and a ground color and just multiplies the material's color between those 2 colors—the sky color if the surface of the object is pointing up and the ground color if the surface of the object is pointing down.

```
const skyColor = 0xB1E1FF;  // light blue
const groundColor = 0xB97A20;  // brownish orange
const intensity = 1;
const light = new THREE.HemisphereLight
      (skyColor, groundColor, intensity);
scene.add(light);
```

Hemisphere Light



Directional Light

A light that shots **infinite parallel light rays** with a direction defined by the position of the light and the position of a target.

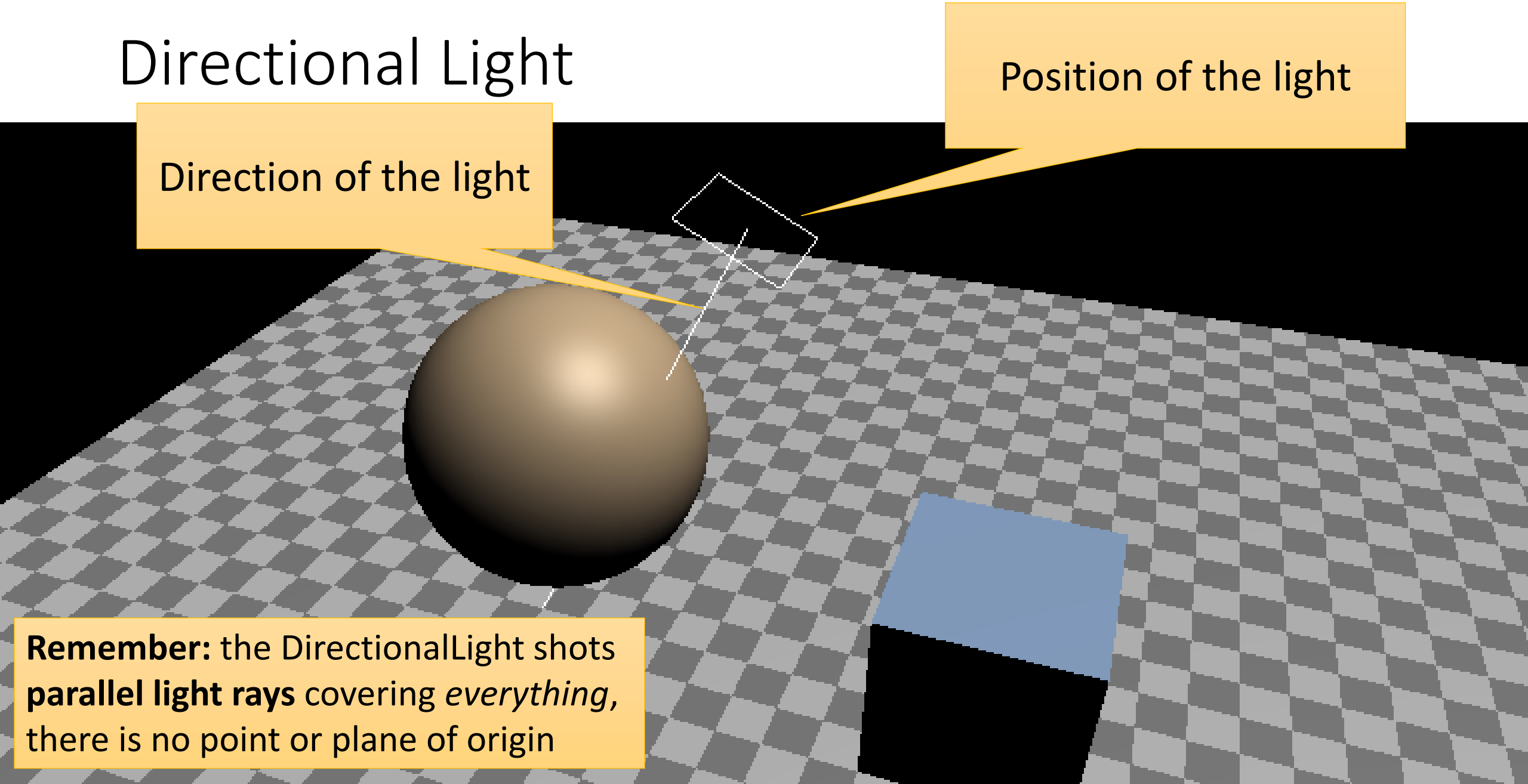
```
const color = 0xFFFFFF;
const intensity = 1;
const light = new THREE.DirectionalLight(color, intensity);
light.position.set(0, 10, 0);
light.target.position.set(-5, 0, 0);
scene.add(light);
scene.add(light.target);
```

Directional Light

Three.js has support to *helper objects* that can make things easier to visualize, like the **DirectionalLightHelper**.

```
const helper = new THREE.DirectionalLightHelper(light);
scene.add(helper);
```

Directional Light



Remember: the DirectionalLight shots **parallel light rays** covering *everything*, there is no point or plane of origin

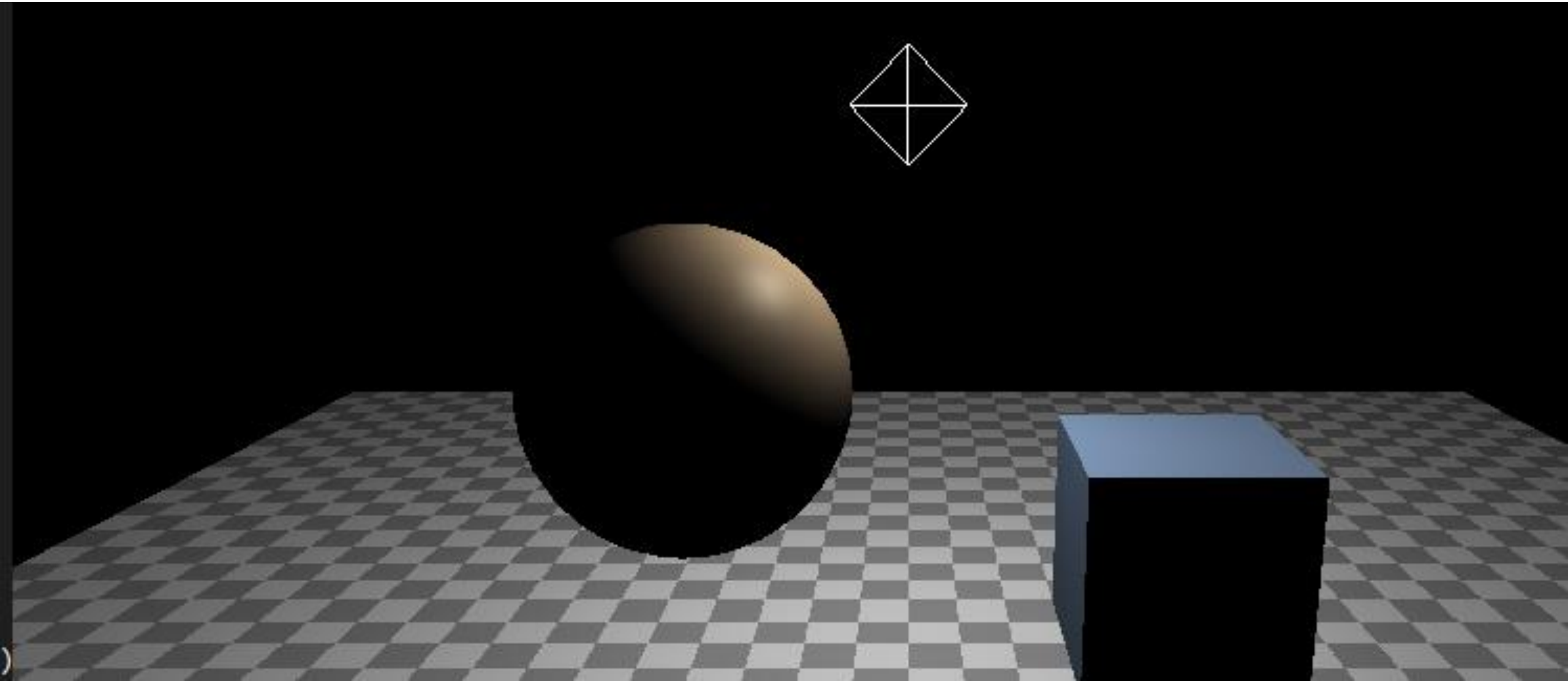
Point Light

A light that sits at a point and shoots light in all directions from that point.

We are also using a **PointLightHelper** in this example.

```
const color = 0xFFFFFF;
const intensity = 1;
const light = new THREE.PointLight(color, intensity);
light.position.set(0, 10, 0);
scene.add(light);
const helper = new THREE.PointLightHelper(light);
scene.add(helper);
```

Point Light



Spot Light

Spotlights are point lights with a cone attached where the light only shines inside the cone, in fact 2 cones: an outer cone and an inner cone.

Between the inner cone and the outer cone, the light fades from full intensity to zero.

```
const color = 0xFFFFFF;
const intensity = 1;
const light = new THREE.SpotLight(color, intensity);
light.position.set(0, 10, 0);
light.angle = THREE.MathUtils.degToRad(34);
light.penumbra = 0.4;
scene.add(light);
scene.add(light.target);
```

Spot Light

The **angle** property defines the outer cone.

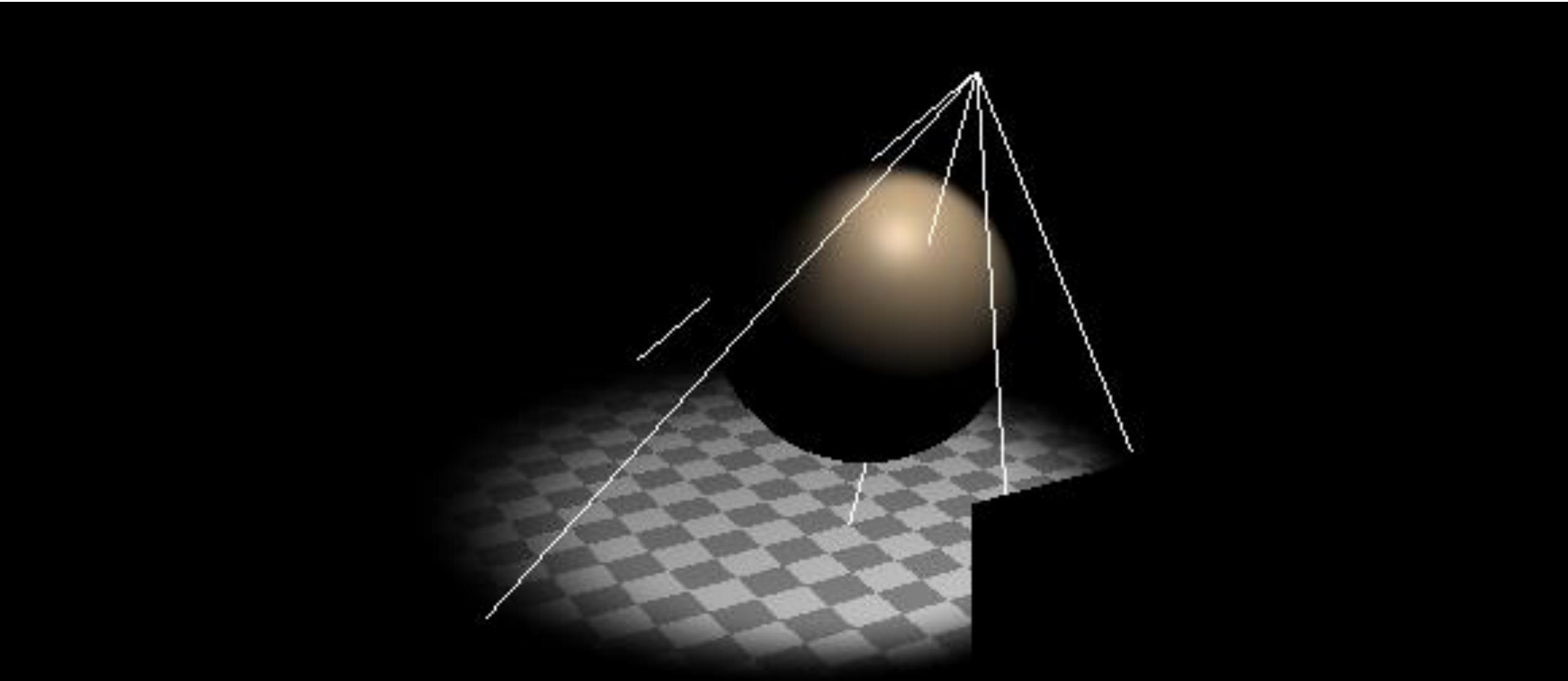
The **penumbra** property defines de inner cone in a percentage of ths outer cone angle.

Light will fade from full to zero between the two cones.

We can also use a **SpotLightHelper**.

```
const helper = new THREE.SpotLightHelper(light);
scene.add(helper);
```

Spot Light



Rect Area Light

A rectangular area of light like a long fluorescent light or a frosted sky light in a ceiling.

IMPORTANT: Only works with MeshStandardMaterial and the MeshPhysicalMaterial!

First, we must import the extra three.js modules for the Rect Area Light library.

```
import {RectAreaLightUniformsLib}
       from '../three/examples/jsm/lights/RectAreaLightUniformsLib.js';
import {RectAreaLightHelper}
       from '../three/examples/jsm/helpers/RectAreaLightHelper.js';
```

Rect Area Light

Then we need to call the initialization of the library, important so that things work as expected:

```
RectAreaLightUniformsLib.init();
```

We are now ready to create the RectAreaLight.

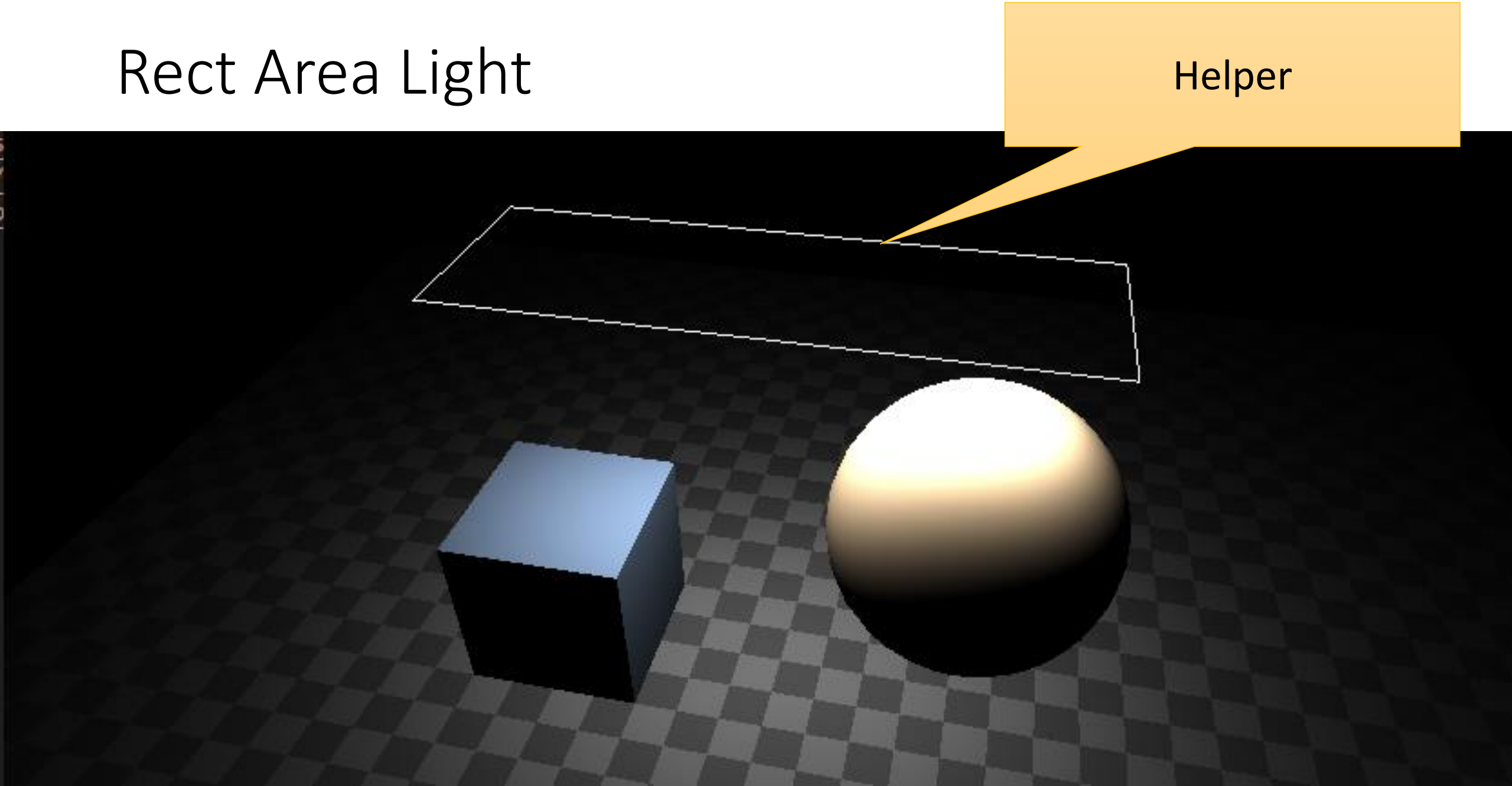
Note that the RectAreaLight uses **rotation** and not a target.

Also note that the helper is a **child** of the light, not of the scene.

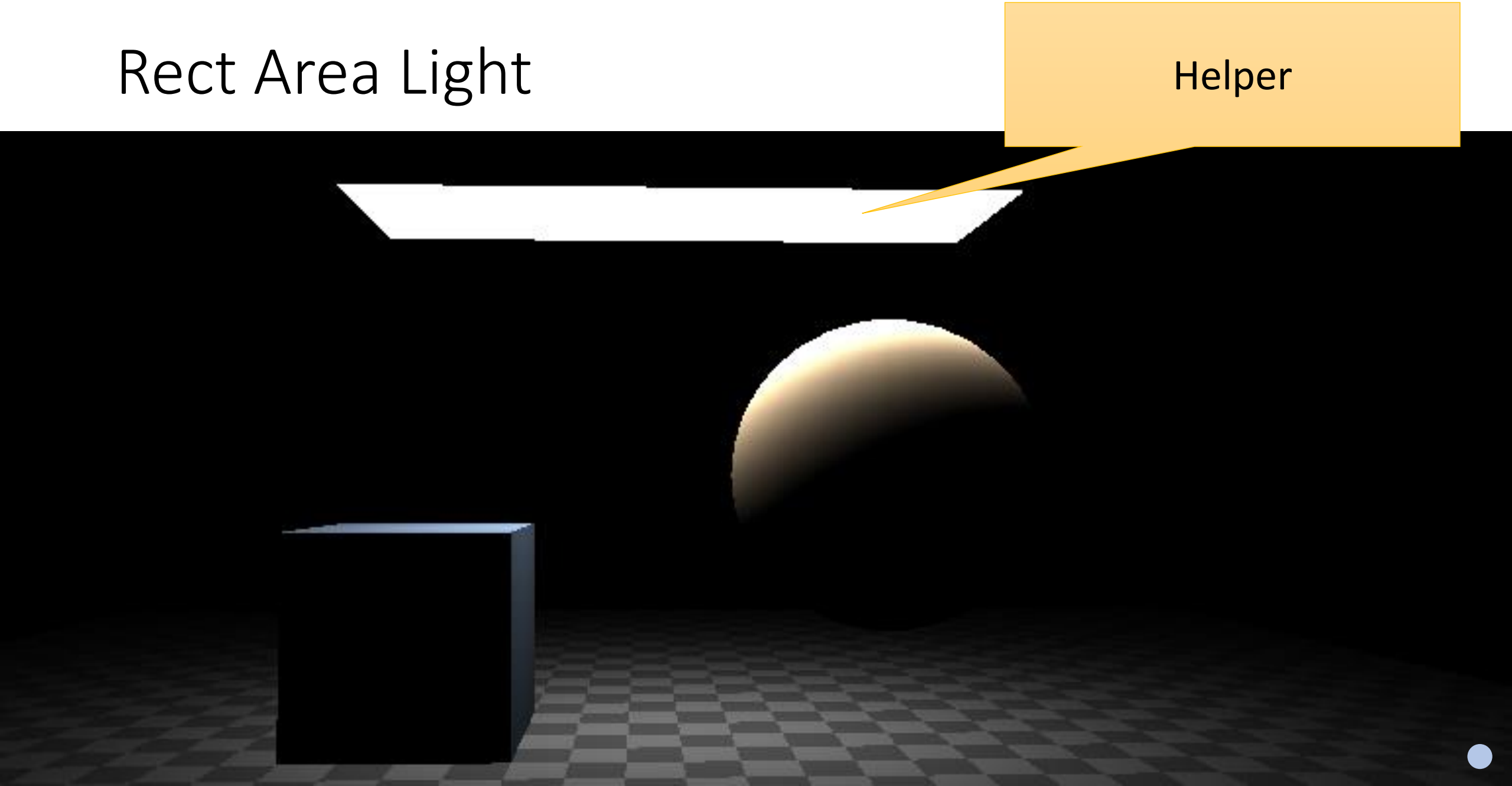
Rect Area Light

```
const color = 0xFFFFFF;
const intensity = 5;
const width = 12;
const height = 4;
const light = new THREE.RectAreaLight(color, intensity, width, height);
light.position.set(0, 10, 0);
light.rotation.x = THREE.MathUtils.degToRad(-90);
scene.add(light);

const helper = new RectAreaLightHelper(light);
light.add(helper);
```

Rect Area Light



Rect Area Light



Shadows

Three.js by default uses **shadow maps**.

The way a shadow map works is, for every light that casts shadows all objects marked to cast shadows are *rendered from the point of view of the light*.

So… if you have 5 lights and all objects are casting lights, the scene will be rendered 6 times: once for each light to cast the shadows, and the 6th time to render de final camera view with the shadows.

https://threejs.org/manual/#en/shadows

Shadows

You have to use **DirectionalLight**, **PointLight** or **SpotLight** to cast shadows.

First enable shadows on the renderer:

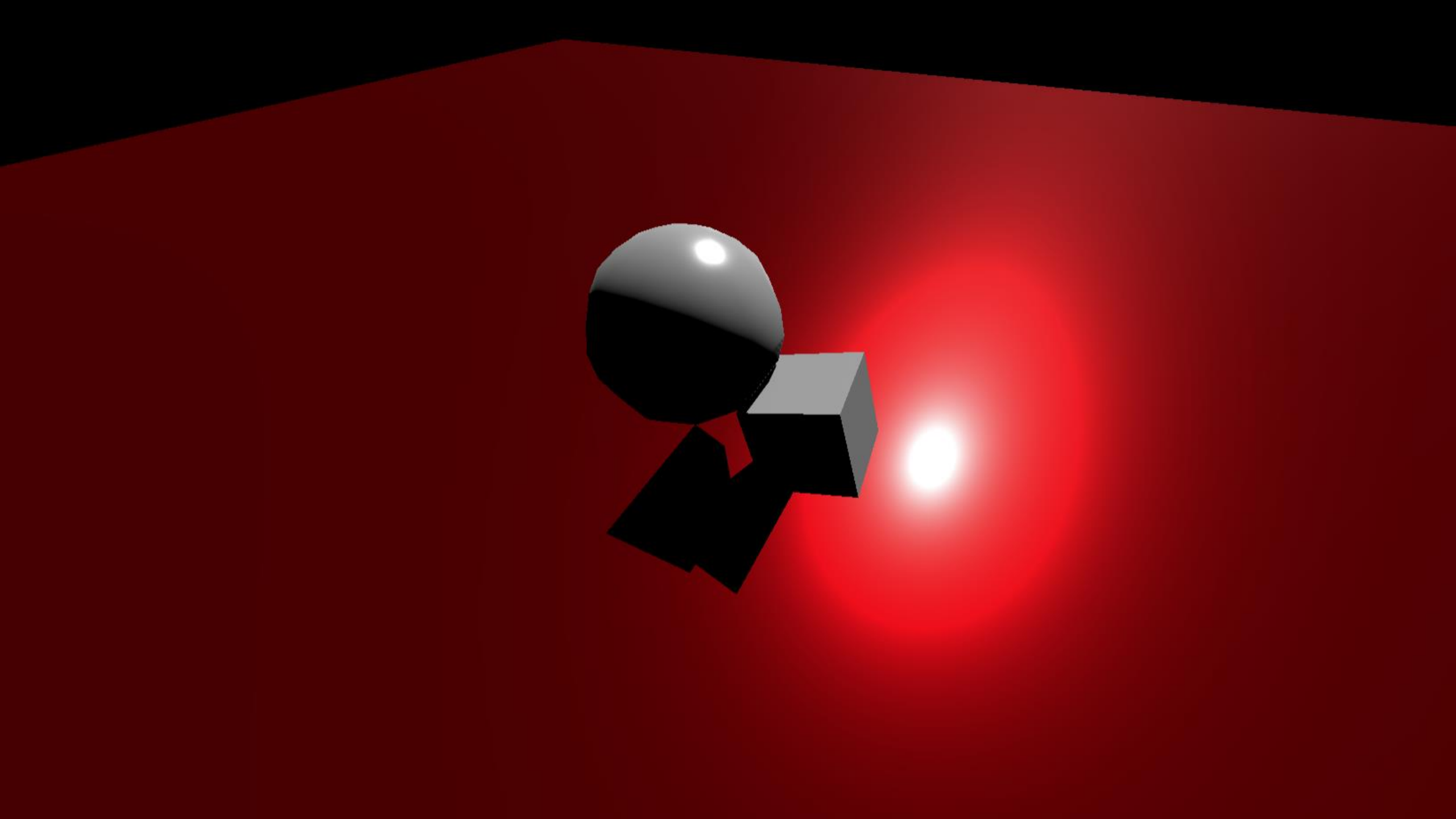
```
renderer.shadowMap.enabled = true;
```

Then enable the light to cast shadows:

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

And, finally, enable our meshes to receive and/or cast shadows:

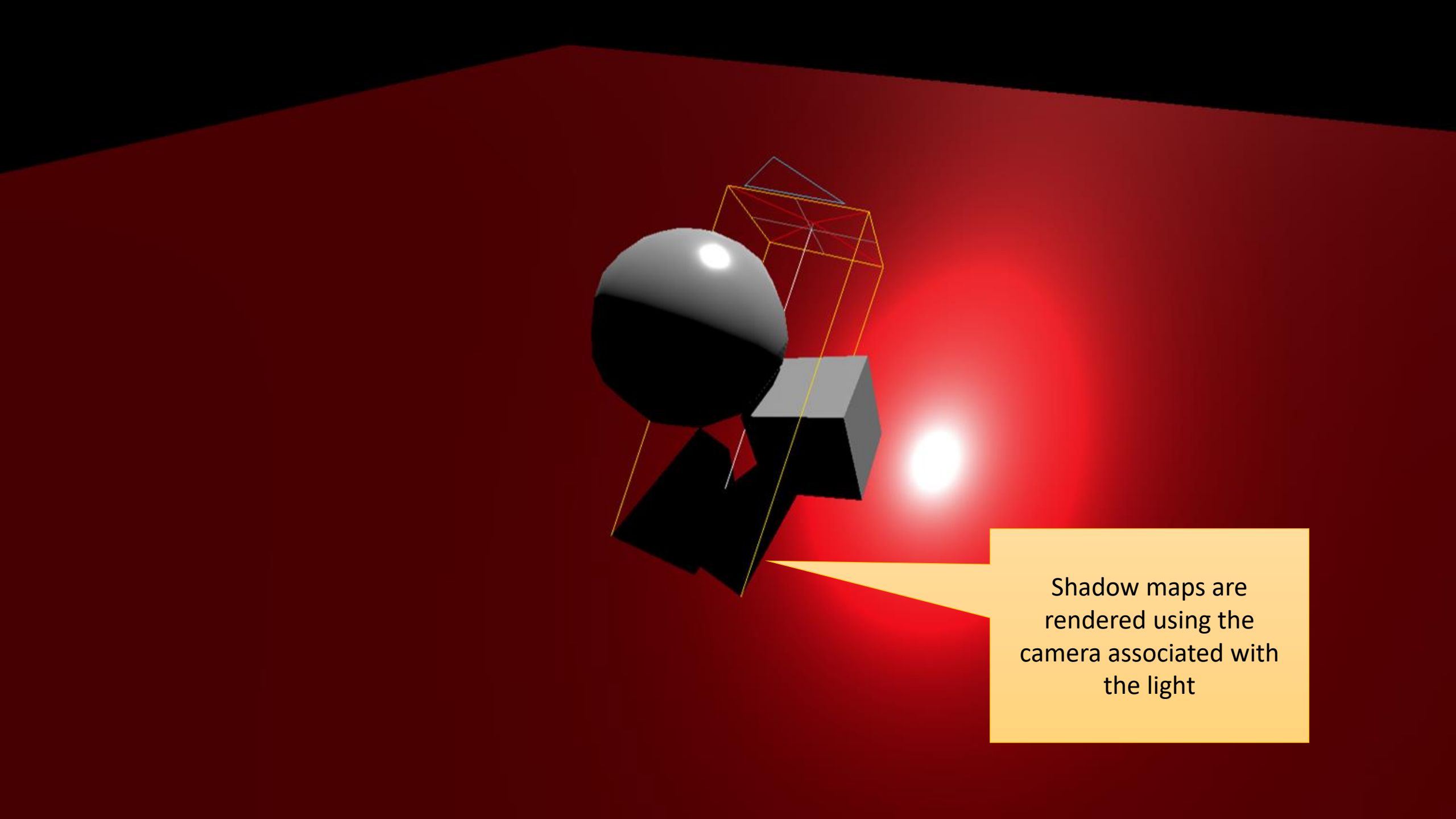
```
cube.castShadow = true;
cube.receiveShadow = true;
```



Shadows

There is something wrong with the shadow… let's see what is the *camera view* for this light:

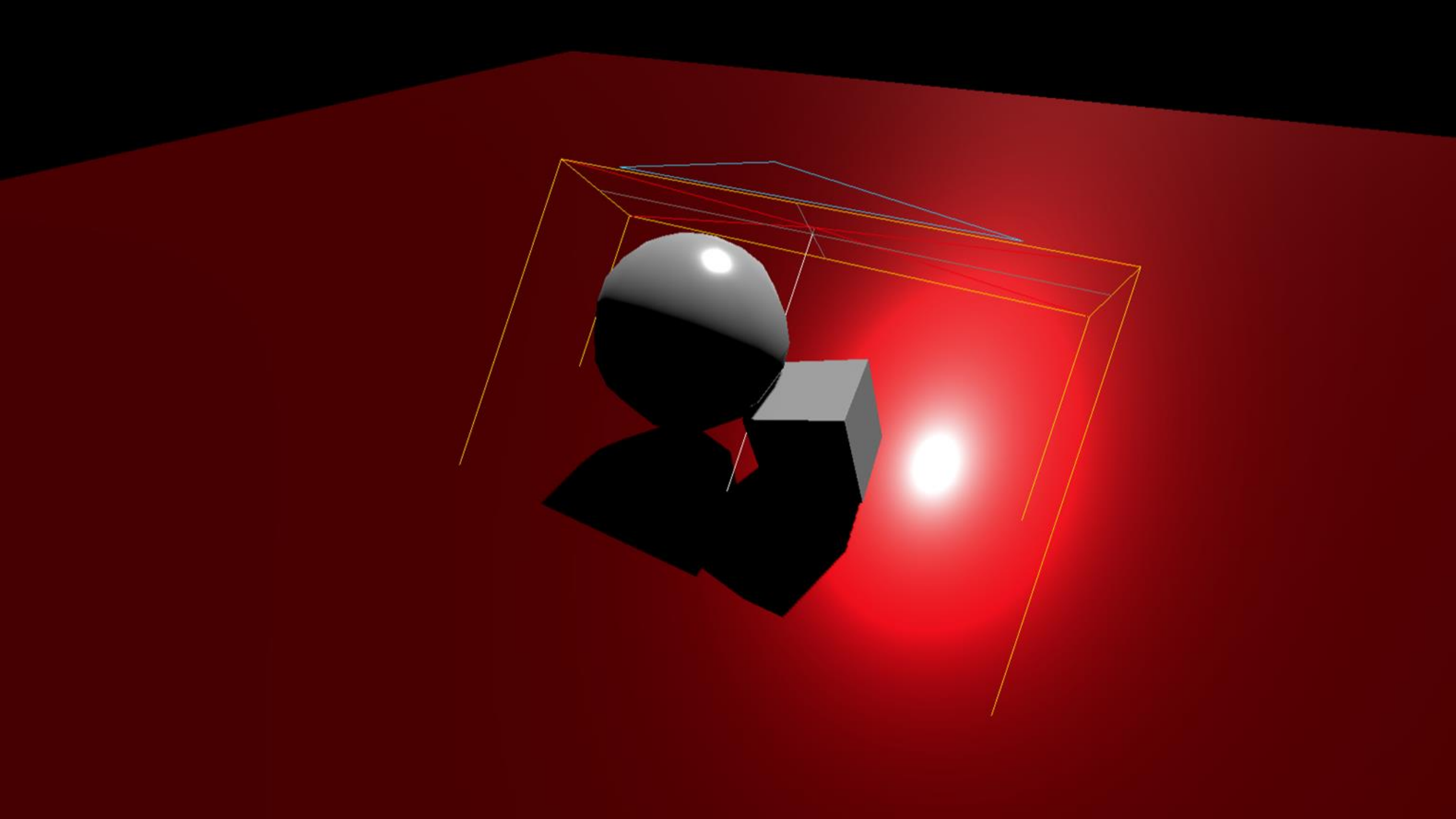
```
const cameraHelper =
   new THREE.CameraHelper(light.shadow.camera);

scene.add(cameraHelper);
```


Shadow maps are rendered using the camera associated with the light

Shadows

We can adjust the camera directly:

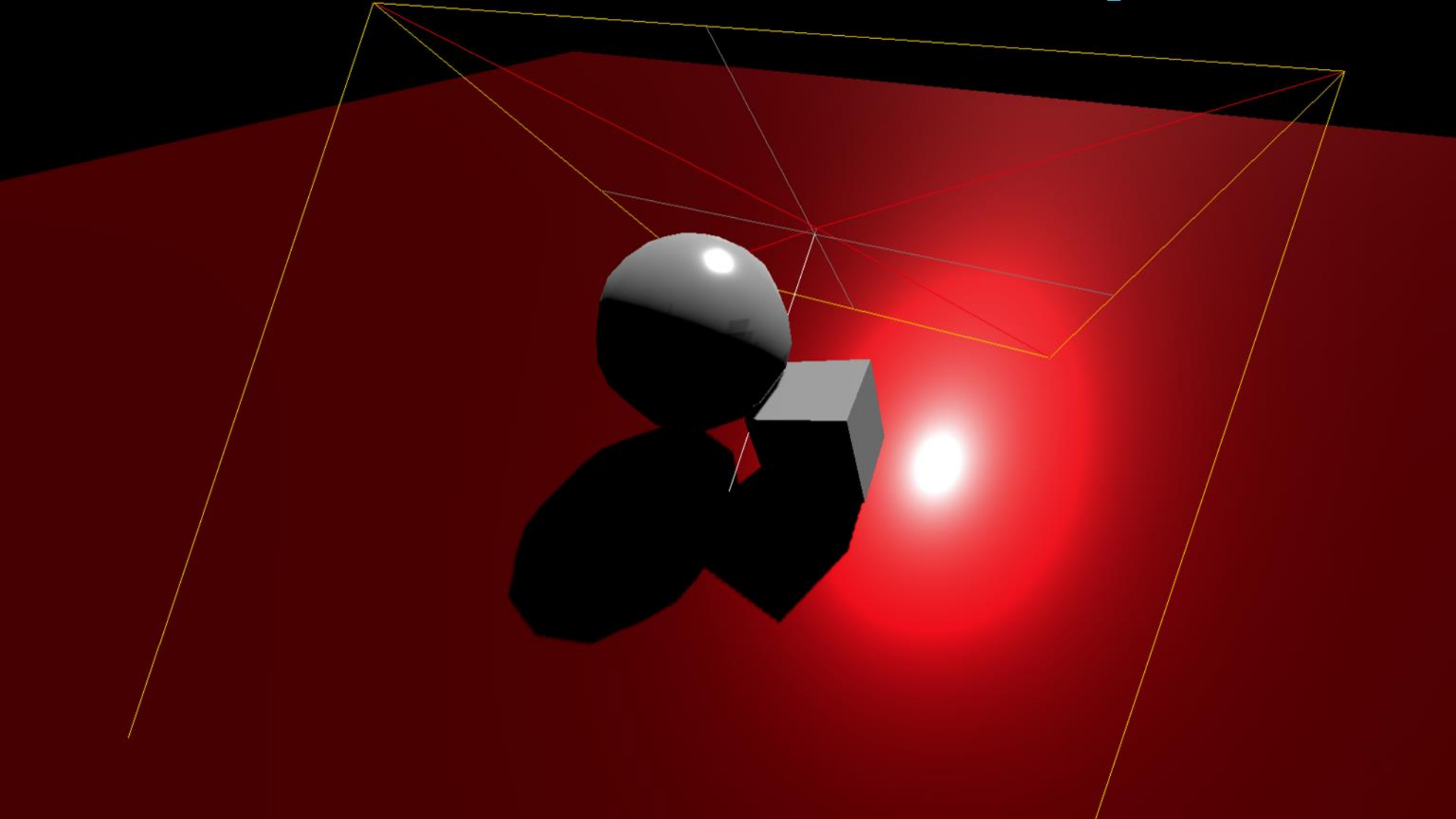
```
light.shadow.camera.left=-20;
light.shadow.camera.right=20;
```



Shadows

And finally:

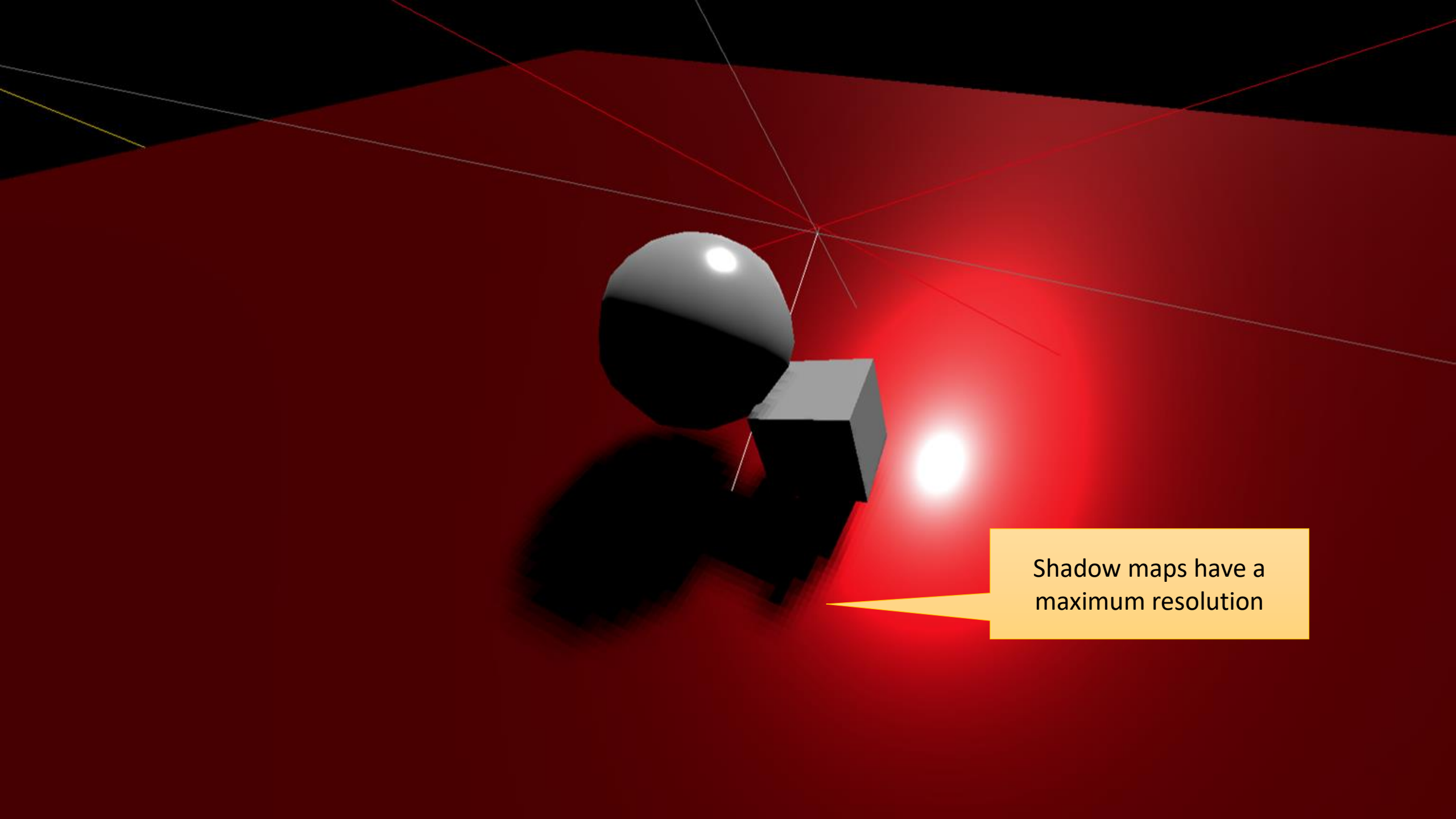
```
light.shadow.camera.top=20;
light.shadow.camera.bottom=-20;
```



Shadows

Then… why not use?

```
light.shadow.camera.left=-200;
light.shadow.camera.right=200;
light.shadow.camera.top=200;
light.shadow.camera.bottom=-200;
```


Shadow maps have a maximum resolution

Shadows

We can adjust the default 512x512 texture map size, e.g.

```
light.shadow.mapSize.width = 2000;
light.shadow.mapSize.height = 2000;
```

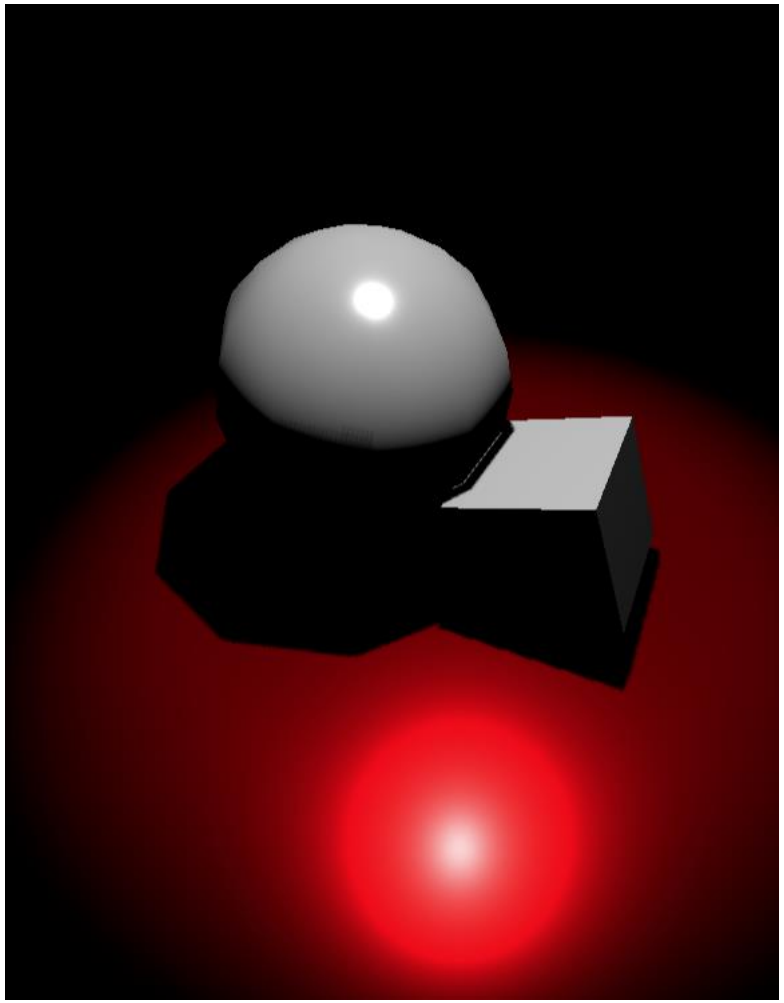
But remember that when you increase a texture map size, you are:

- Using more memory
- Rendering more pixels → slower performance

The idea is to have the texture maps as small as possible, but still getting results that are useful. Also note that each machine may have different limitations regarding both memory and texture size.

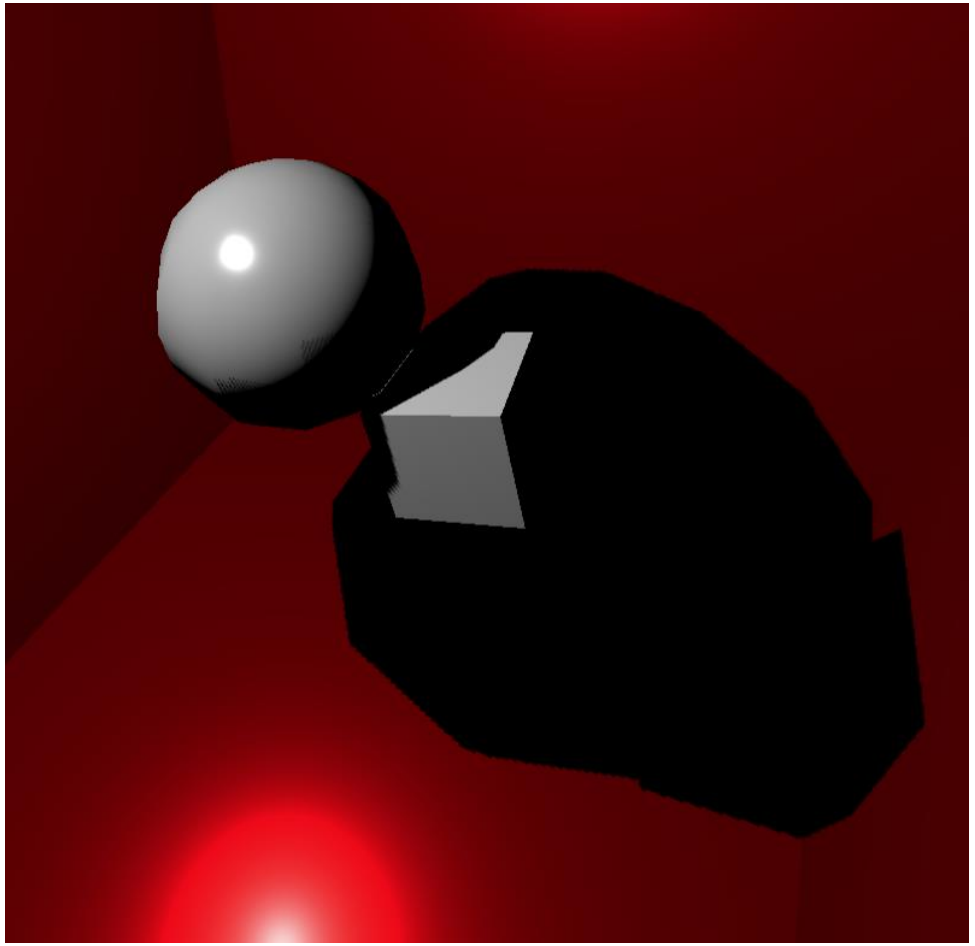
Shadows

- If you use a **SpotLight** it will work in a similar way for shadows but using a **PerspectiveCamera**
- But here the **fov** of the camera is controlled directly by the **angle** of the light



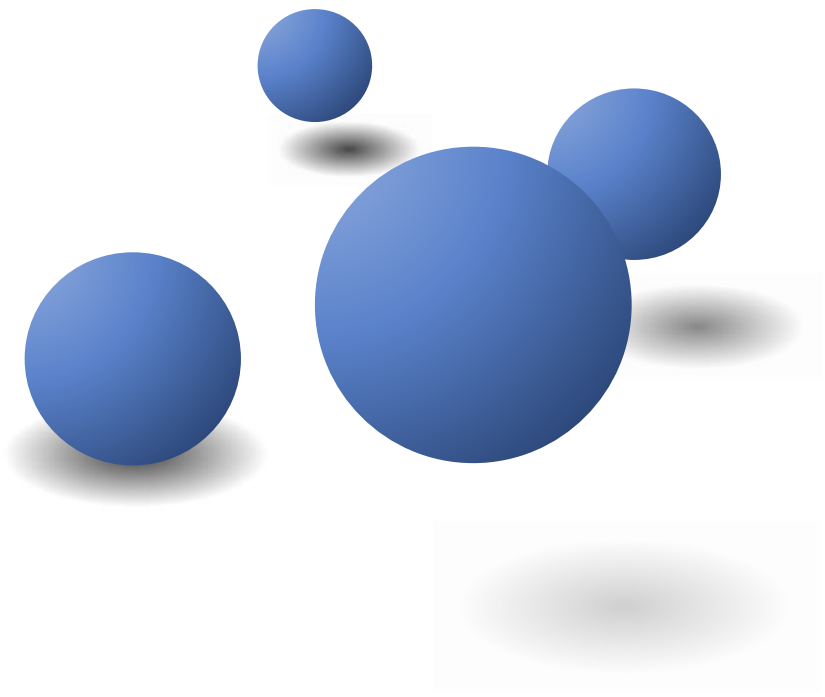
Shadows

- And we can also use a **PointLight** but here Three.js will use 6 cameras, one facing each of the faces of a cube centered in the light, and so this will be much slower than other lights!



Shadows

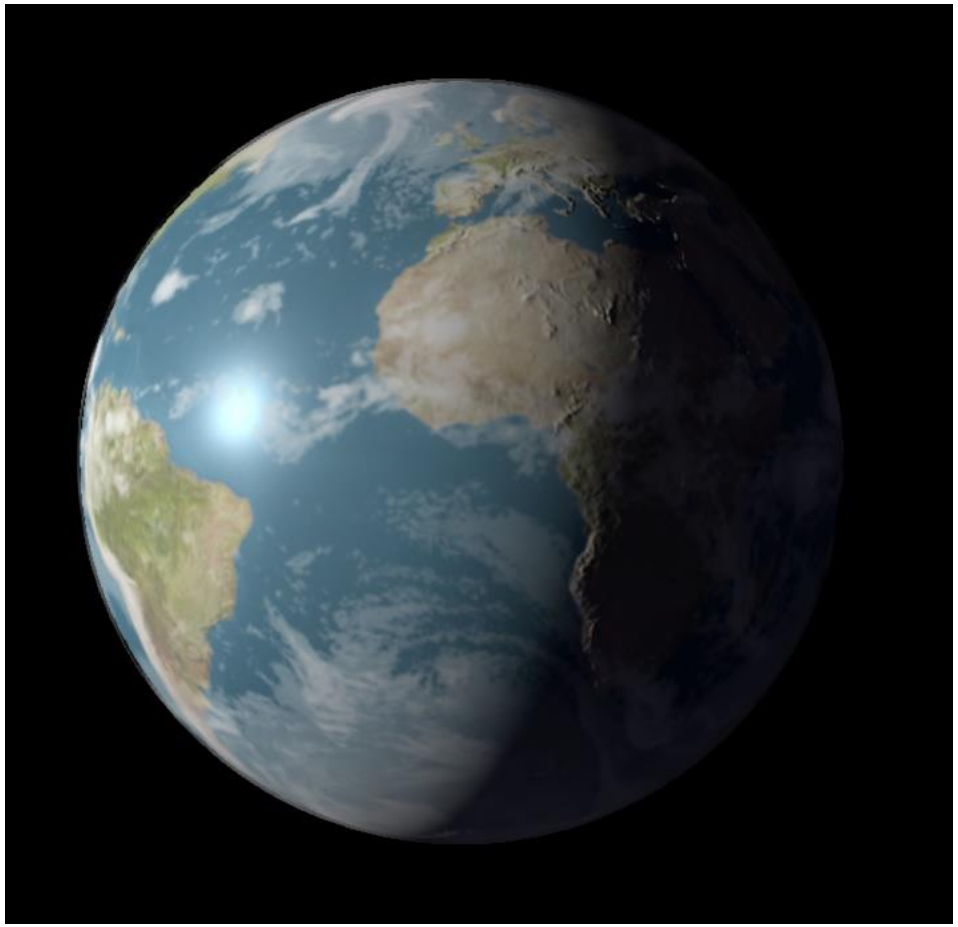
- Shadow maps can be very expensive in resources
- We can use fake shadows to ease the load, e.g. draw a simple oval shadow under the object
- See an example (using Textures) here: https://threejs.org/manual/#en/shadows



Textures

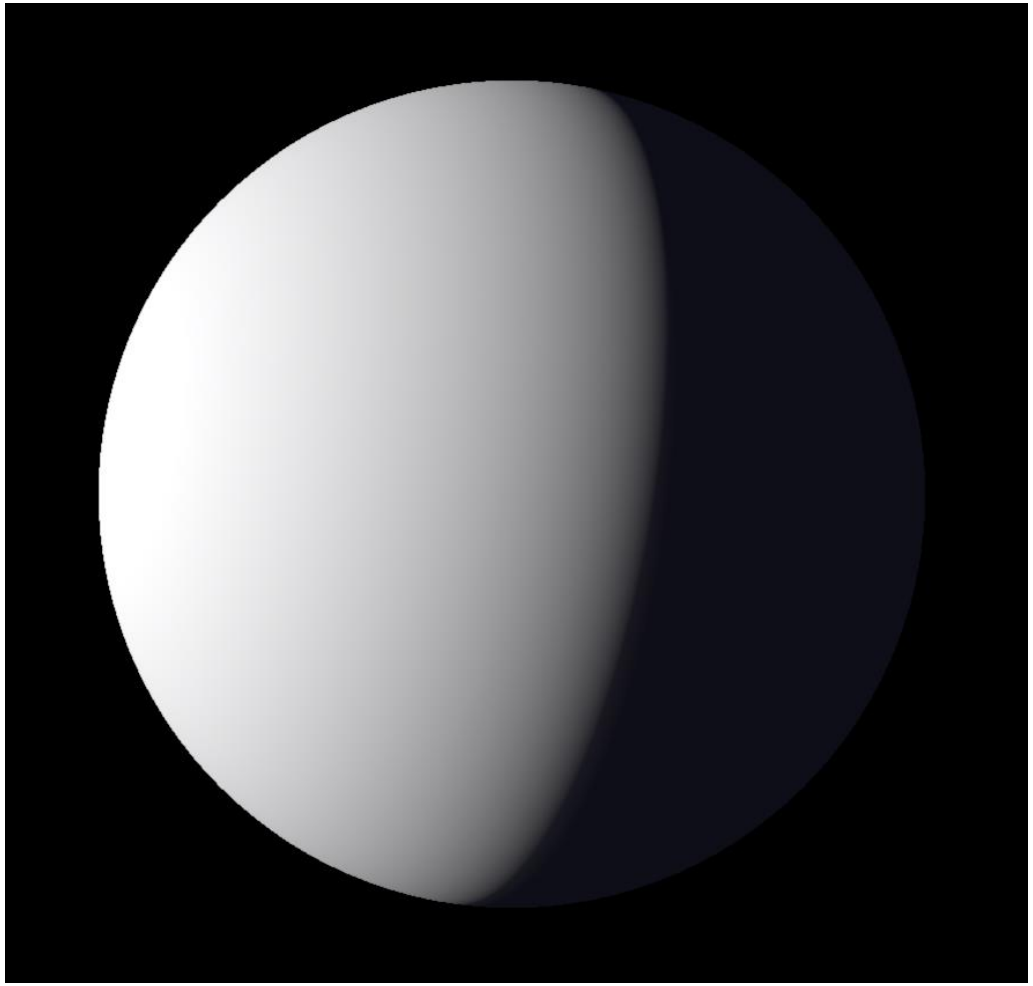
A simple example
(editable online)

https://jsfiddle.net/5mztorwy/



Creating the Sphere

- Create geometry for the sphere
- Create a material
- Create a 3D Object
- Add it to the scene
- Setup lights



Based on https://riptutorial.com/three-js/example/28900/creating-a-model-earth

Creating the Sphere

```
var geometry = new THREE.SphereGeometry( 20, 60, 60 );
var material = new THREE.MeshStandardMaterial
      ({color: 0xffffff, metalness: 0.1, roughness: 0.9});
var earthMesh = new THREE.Mesh( geometry, material );
scene.add( earthMesh );
```

Creating the Scene

```
const camera = new THREE.PerspectiveCamera(
    15, window.innerWidth/window.innerHeight, 0.1, 1000
    );
camera.position.set( 0, 0, 200 );
camera.lookAt( 0, 0, 0 );

const light_amb = new THREE.AmbientLight(0x8080ff, 0.01);
scene.add(light_amb);

const light = new THREE.DirectionalLight(0xFFFFFF, 1);
light.position.set(-30, 2, 10);
light.target.position.set(0, 0, 0);
scene.add(light);
```

Add a texture

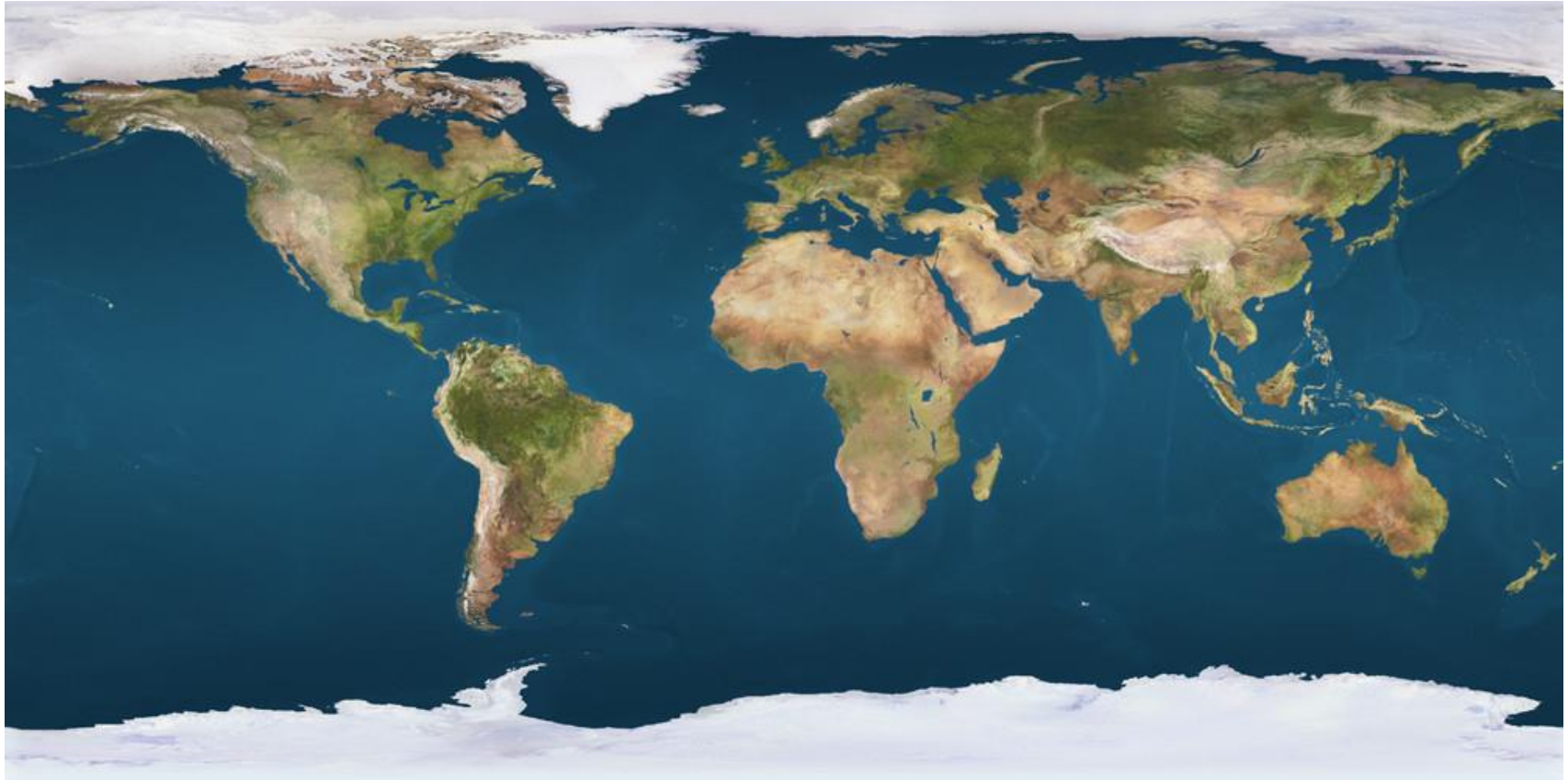
Let's add the main color texture

```
const texture = new THREE.TextureLoader()
    .load('earthmap1k.jpg');
material.map = texture;
```

And adjust the sphere rotation to have Europe in the view

```
earthMesh.rotation.y = Math.PI*1.5;
```

Add a texture



Setup OrbitControls

In order the enable simple camera control with mouse interaction add **OrbitControls**

In the start of the script after the import of THREE add:

```
import { OrbitControls } from
    '../three/examples/jsm/controls/OrbitControls.js';
```

After the Camera creation and setup add:

```
var cameraControls = new OrbitControls( camera, renderer.domElement );
cameraControls.target.set( 0, 0, 0 );
cameraControls.maxDistance = 400;
cameraControls.minDistance = 40;
cameraControls.update();
```

Add a Bump texture

Let’s add the “bump” texture

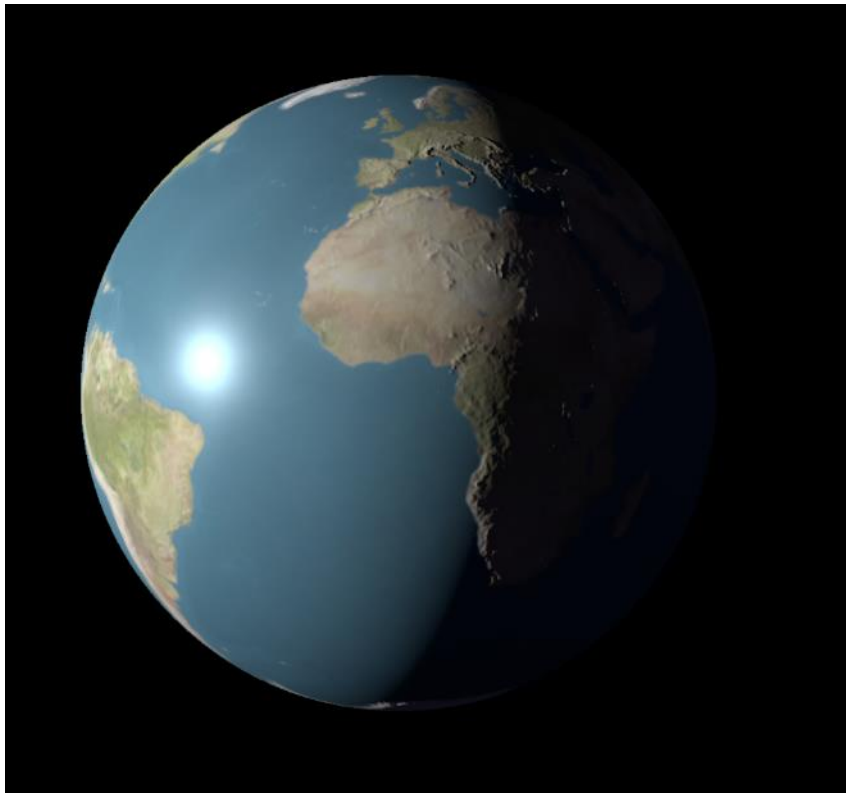
```
const textureBump = new THREE.TextureLoader()
    .load('earthbump1k.jpg');
material.bumpMap   = texture;
material.bumpScale = 0.3;
```

Add a Bump texture

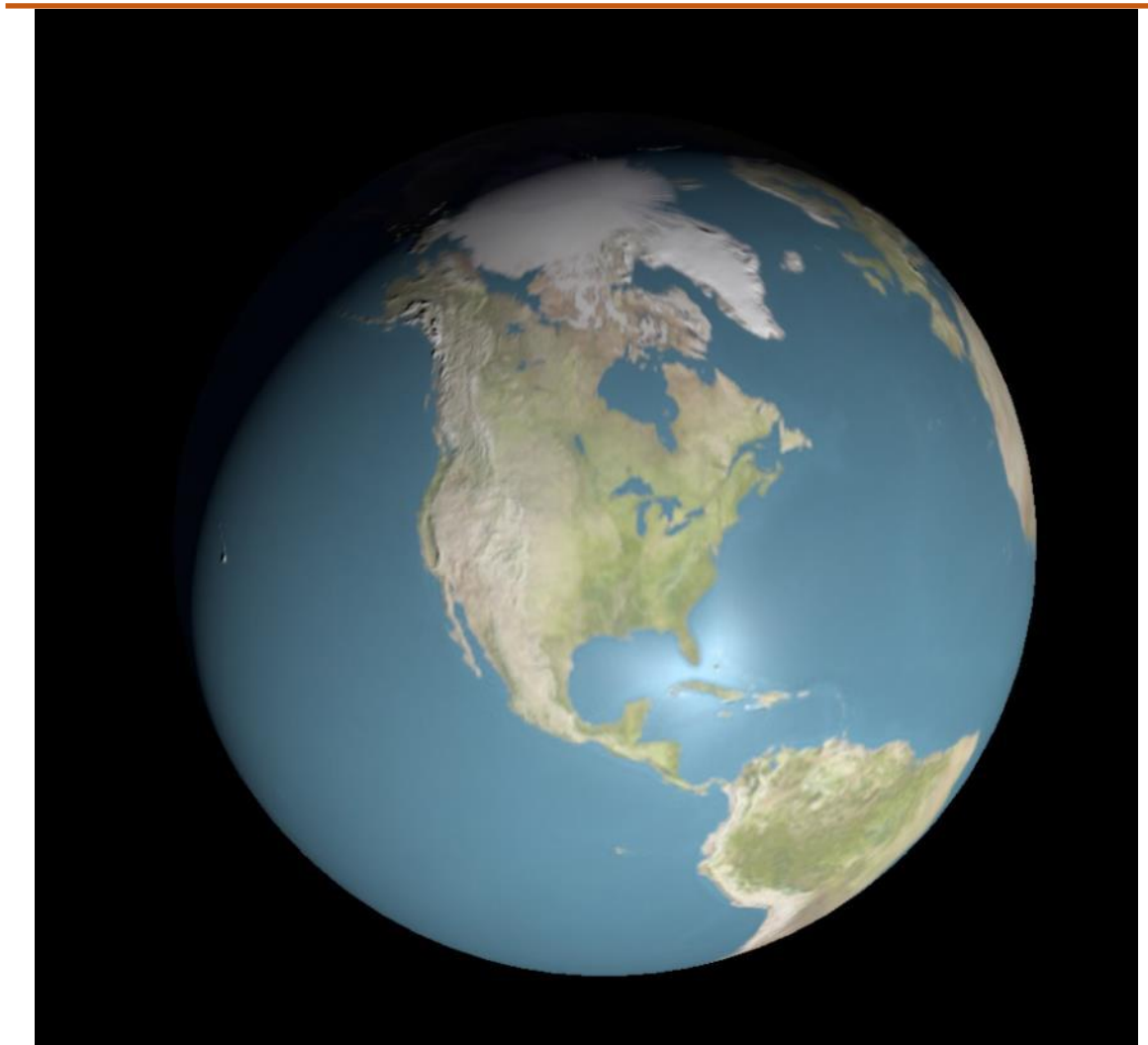
Add a Roughness texture

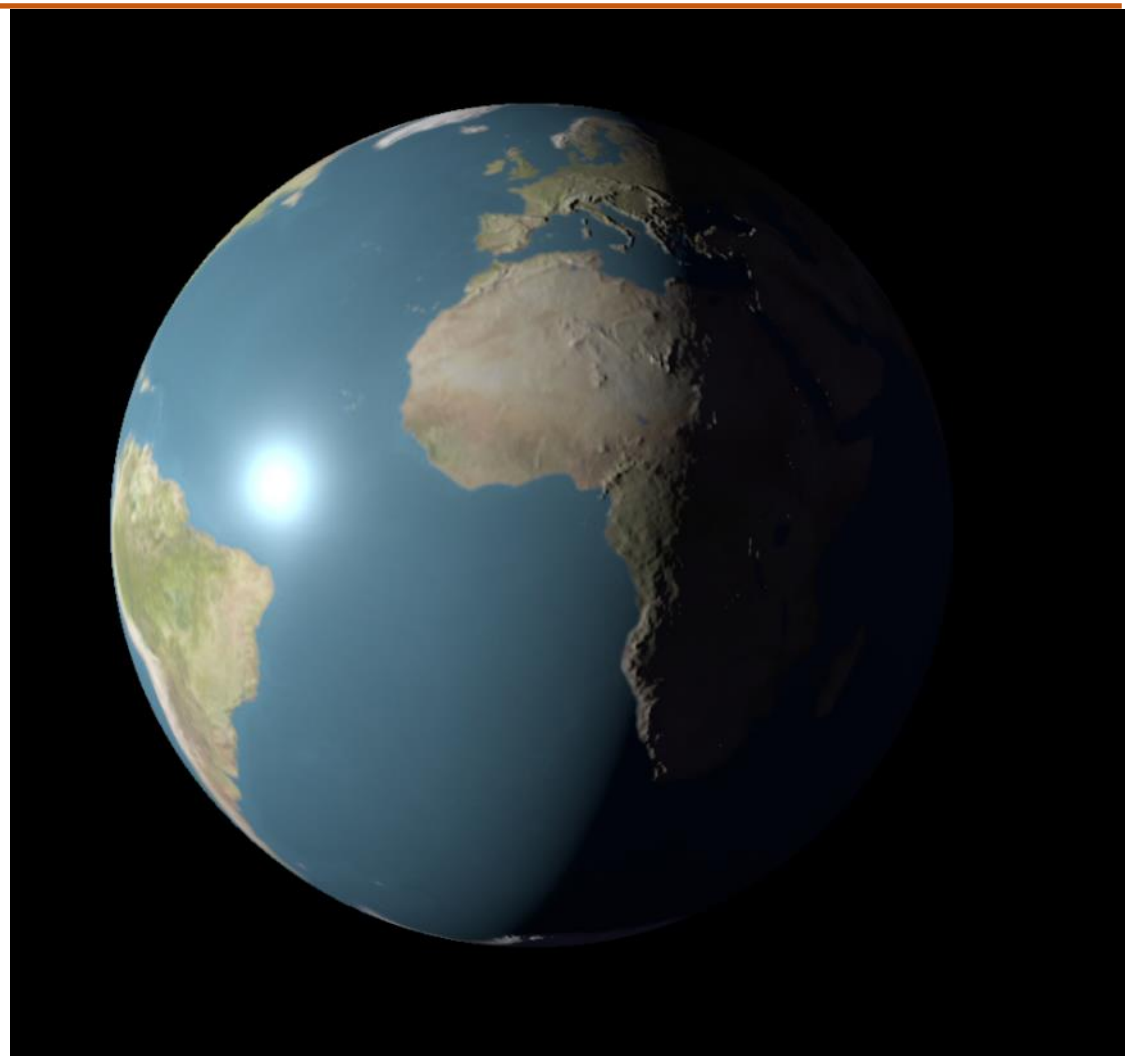
Let's add the "roughness" texture

```
const textureRoughness =
    new THREE.TextureLoader()
        .load('earthroughness1k.jpg');
material.roughnessMap = textureRoughness;
```

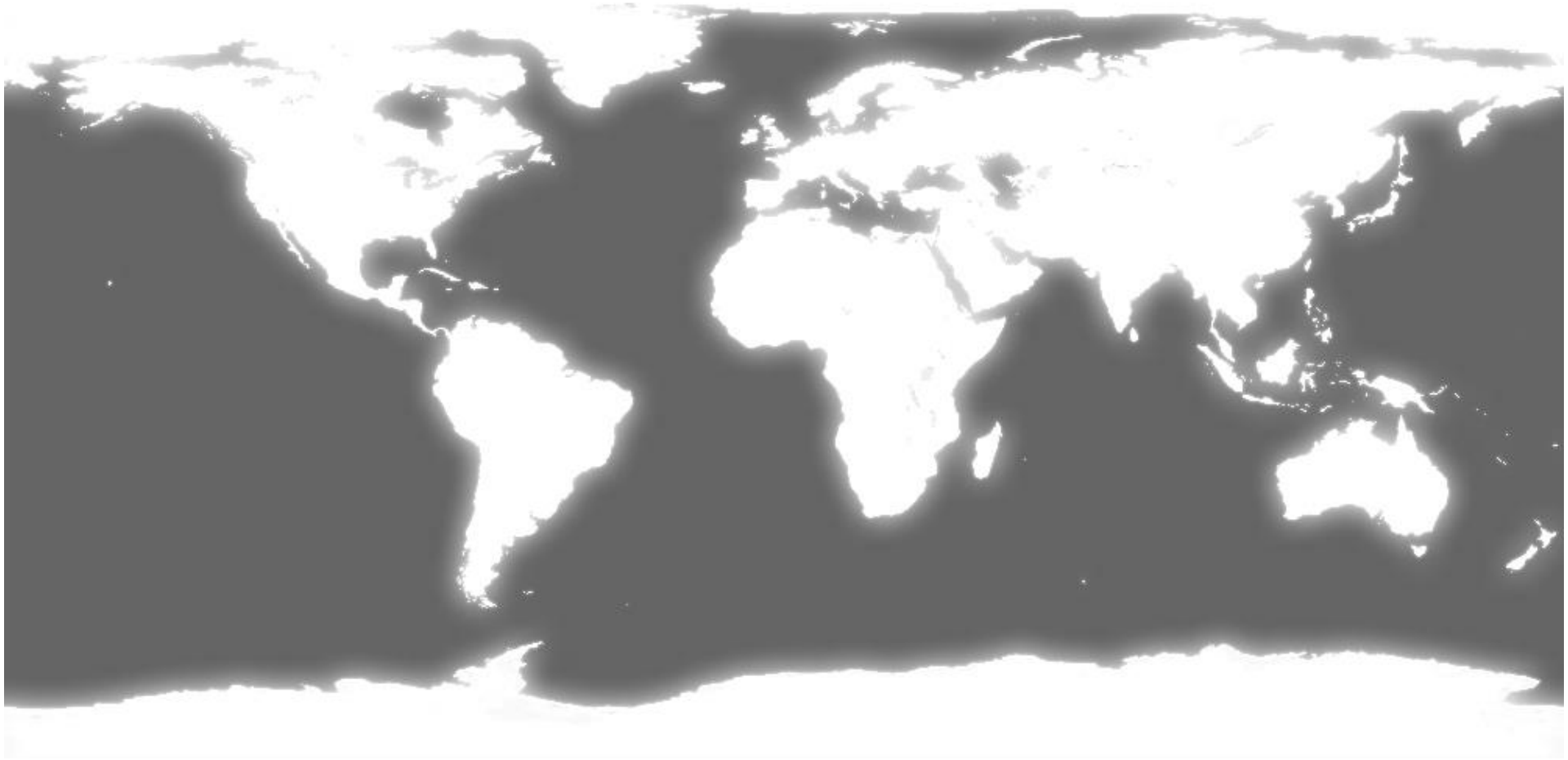


Add a Roughness texture



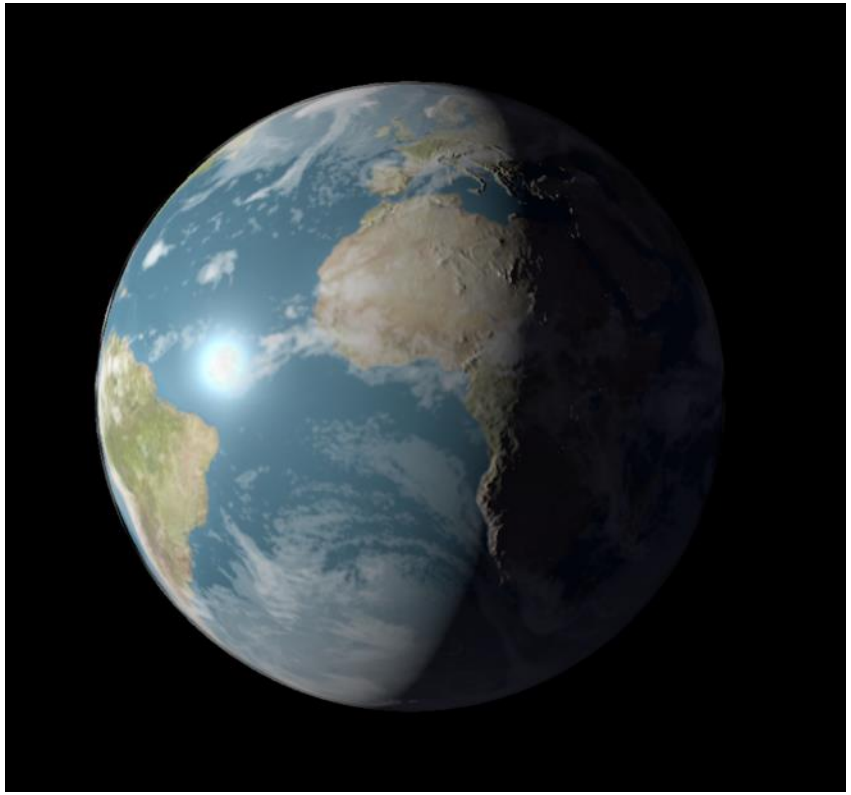


Add a Roughness texture

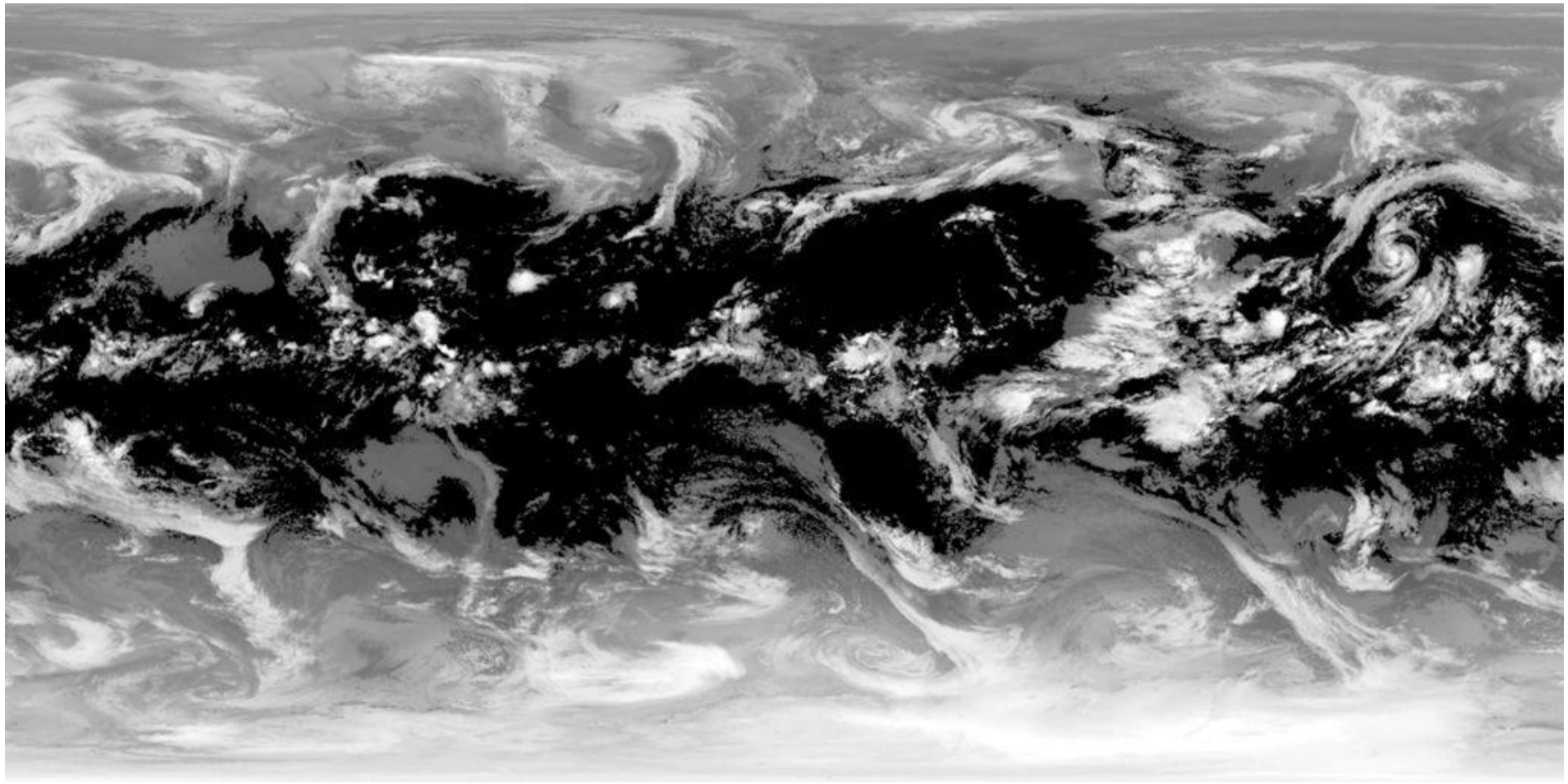


Add a Cloud layer with Alpha texture

```
var geometry   = new THREE.SphereGeometry(20.2, 41, 41)

const textureClouds = new THREE.TextureLoader()
    .load('earthcloudsa1k.jpg');

var material = new THREE.MeshStandardMaterial({
    color: 0xffffff, emissive: 0x010101,
    metalness: 0, roughness: 1,
    side: THREE.DoubleSide,
    opacity: 0.9, transparent : true,
    alphaMap: textureClouds
});

var cloudMesh = new THREE.Mesh(geometry, material)
earthMesh.add(cloudMesh)
```

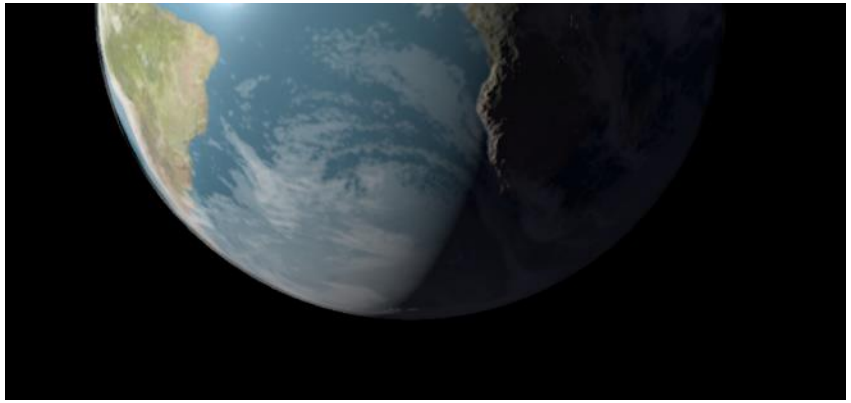


Add a Cloud layer with Alpha Texture



Let’s soften the night-day separation

```
const light = new THREE.DirectionalLight(0xFFFFFF,0.3);


for(var i=0;i<6;i++) {
 const light=new THREE.DirectionalLight(0xFFFFFF, 0.1);
 light.position.set(-30,
     5+Math.random()*6,
     10+Math.random()*6);
 light.target.position.set(0, 0, 0);
 scene.add(light);
}
```



Add a simple animation

```
var animate = function () {
    requestAnimationFrame( animate );

    cloudMesh.rotation.y += 0.001;
    earthMesh.rotation.y += 0.0021;

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

Texture Mapping

- Next class

Texture Mapping

Most of the objects in our everyday life are not made of uniform, single-color materials.

On the other hand, it is helpful to have a single mesh to represent an object despite the variations on the surface response, like in the previous globe example.

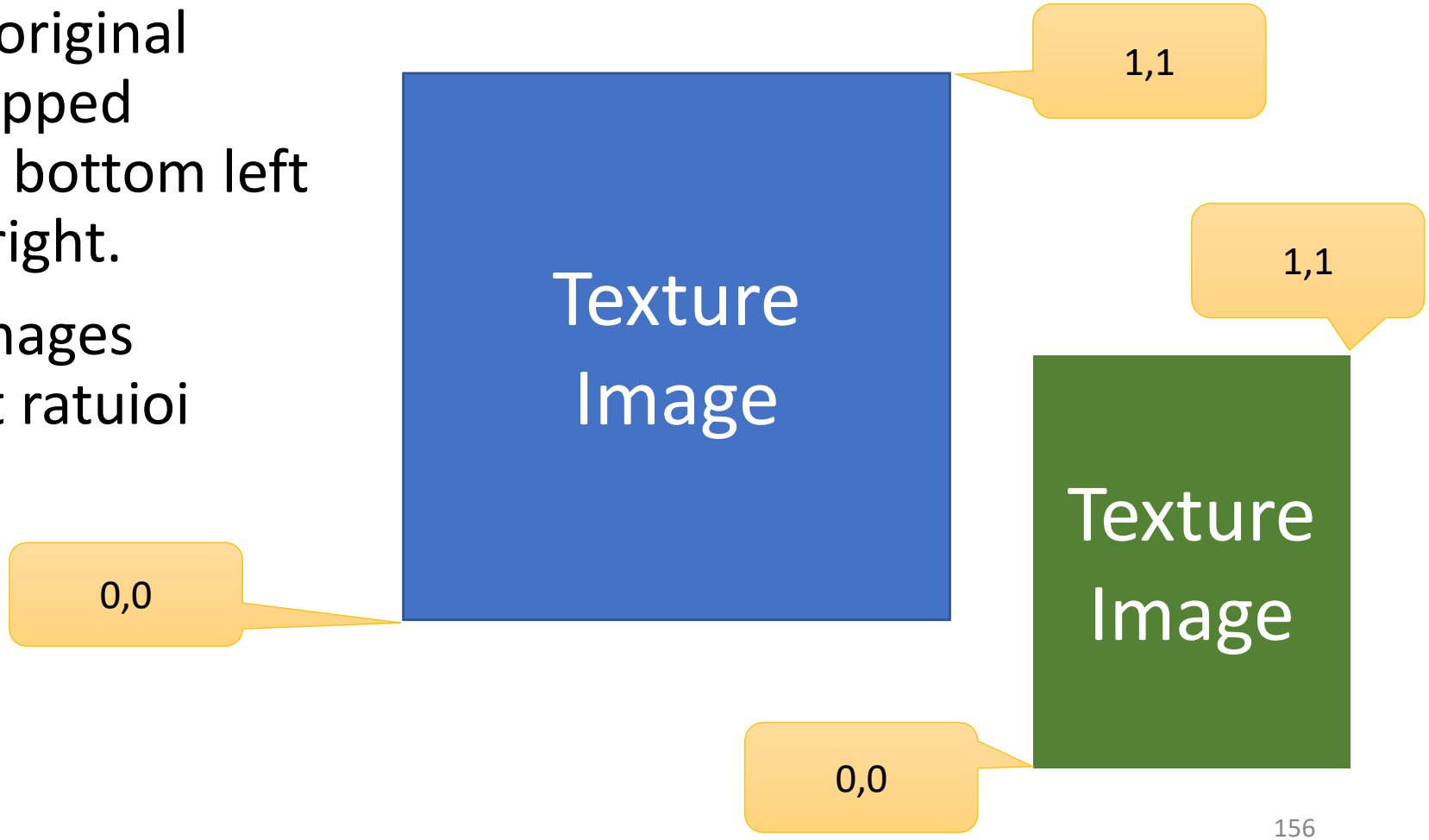
Textures are a very effective way to solve this, but there is a caveat:

how to map the texture images to the surface???

Based on: https://discoverthreejs.com/book/first-steps/textures-intro/

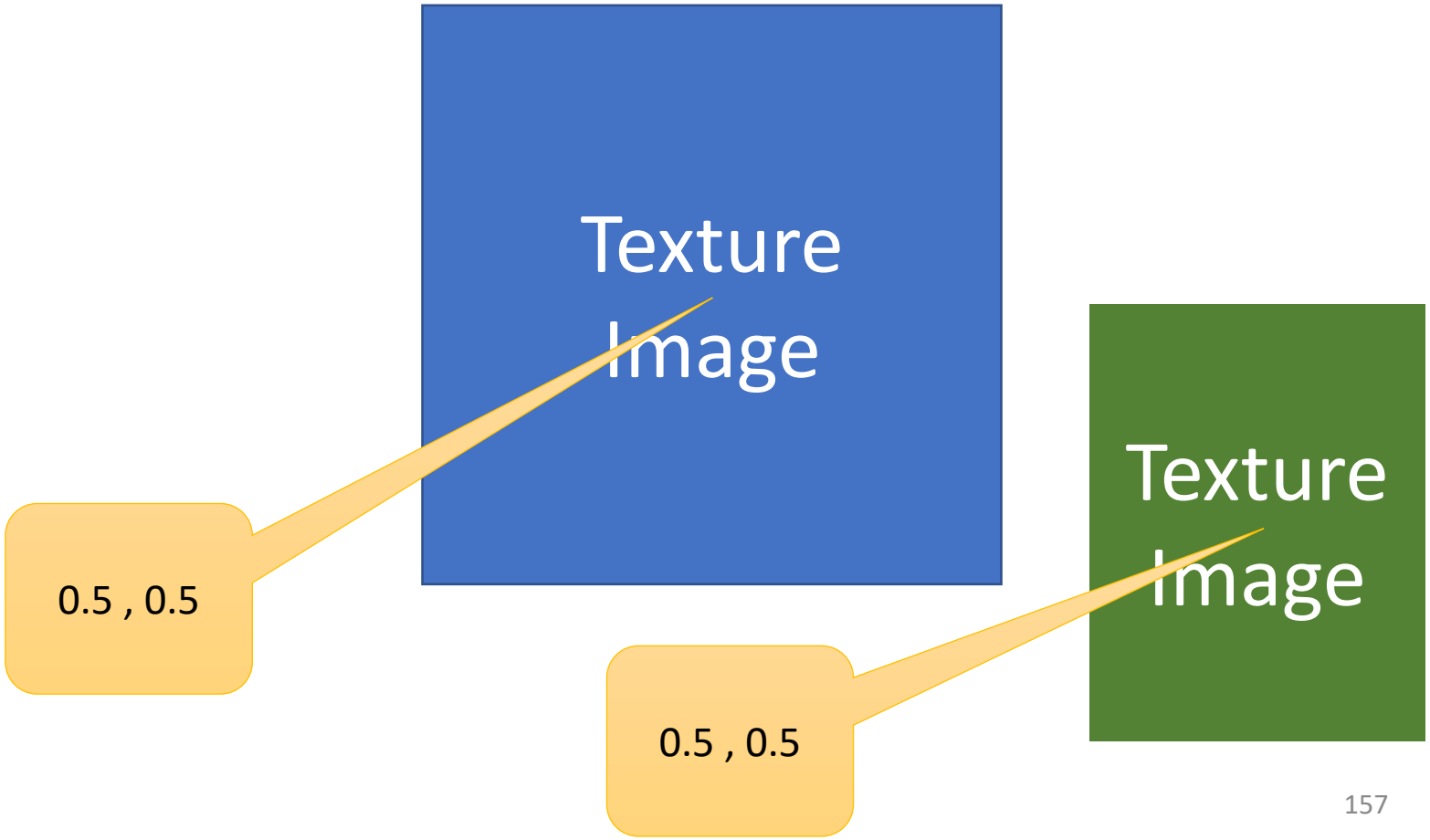
UV Mapping

- In UV Mapping the original texture image is mapped between 0,0 on the bottom left and 1,1 on the top right.
- This applies to all images regardless of aspect ratuioi



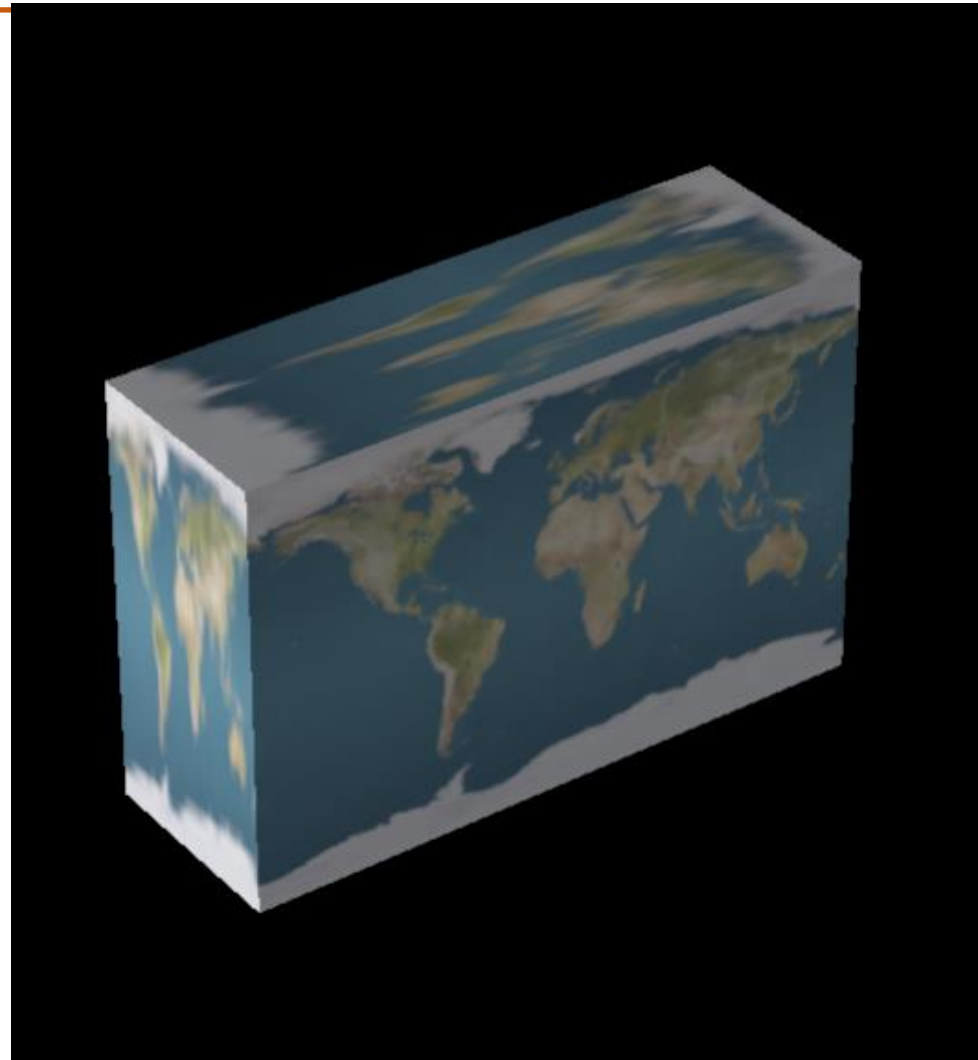
UV Mapping

- Center is 0.5 , 0.5



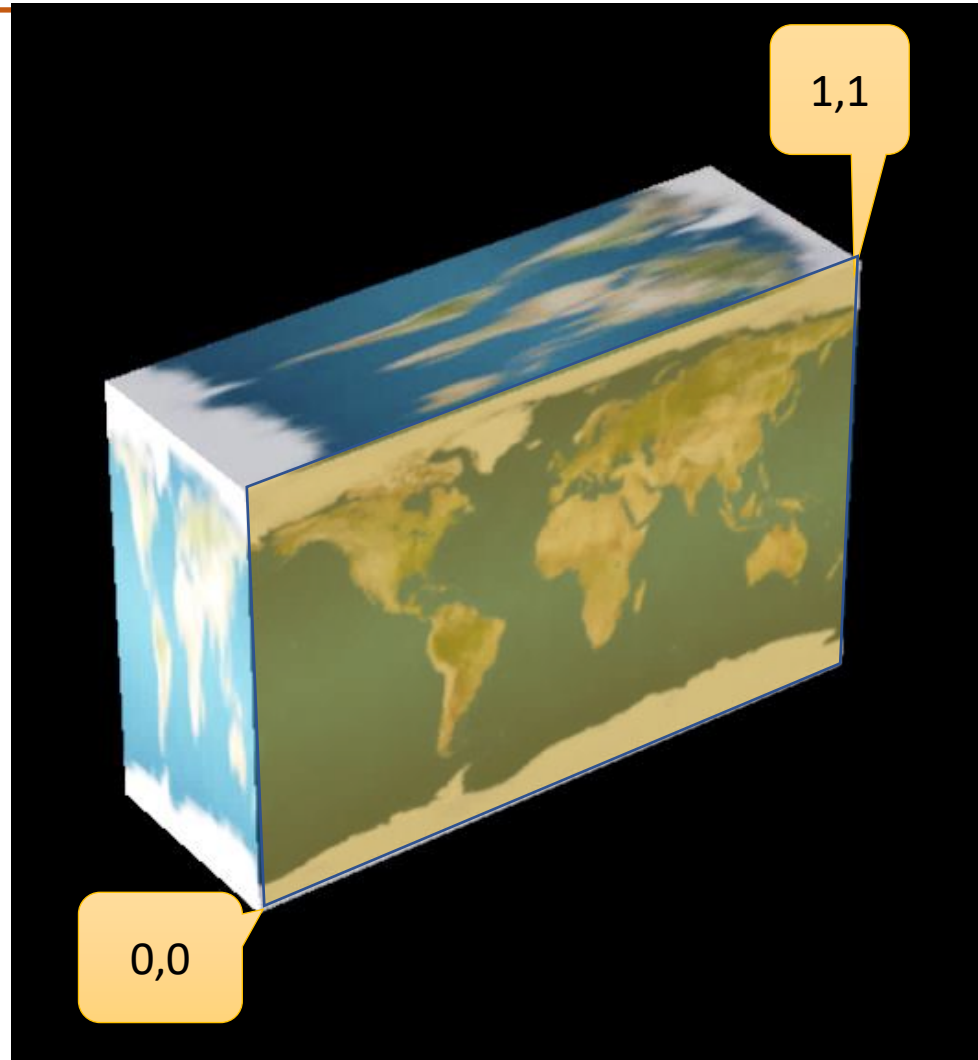
UV Mapping

- We can now use these coordinates to map our image to the 3D surface.
- We already saw **SphereGeometry** result
- **BoxGeometry** does this for us:



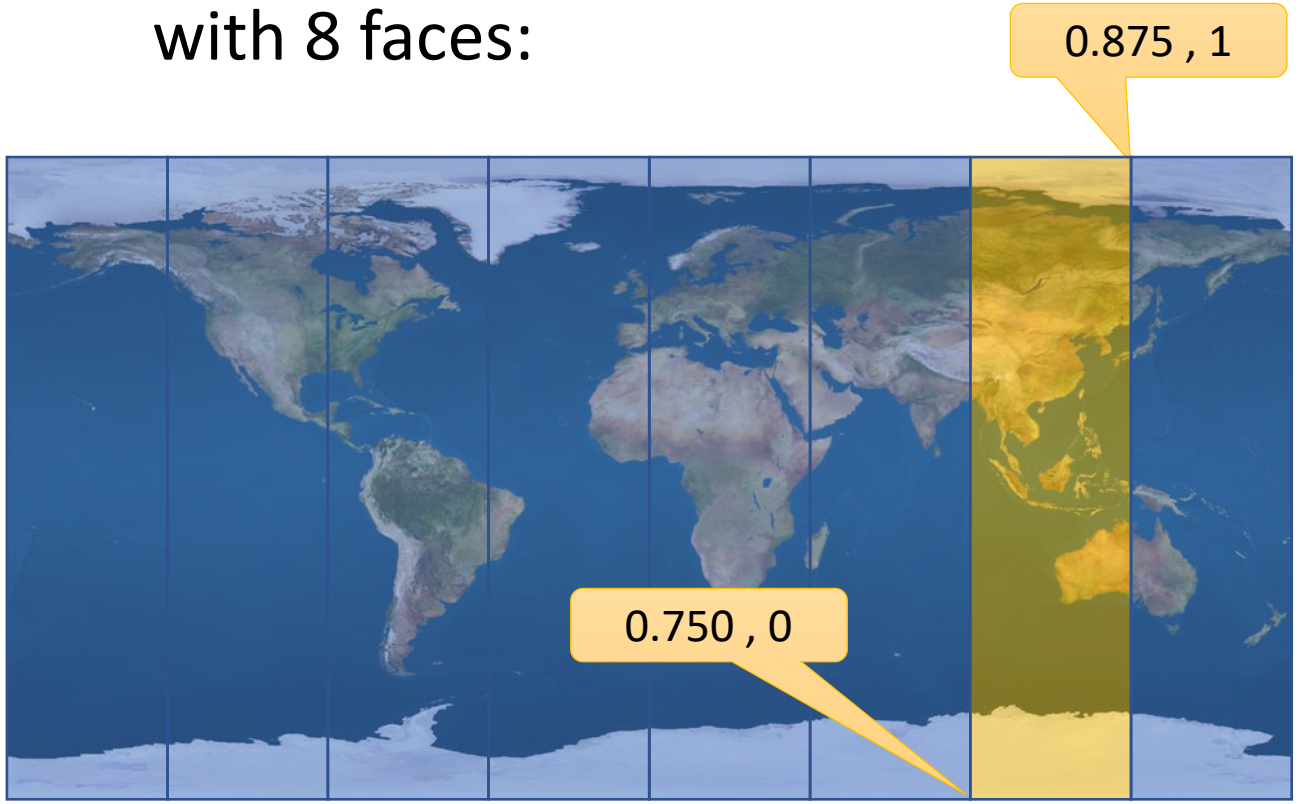
UV Mapping

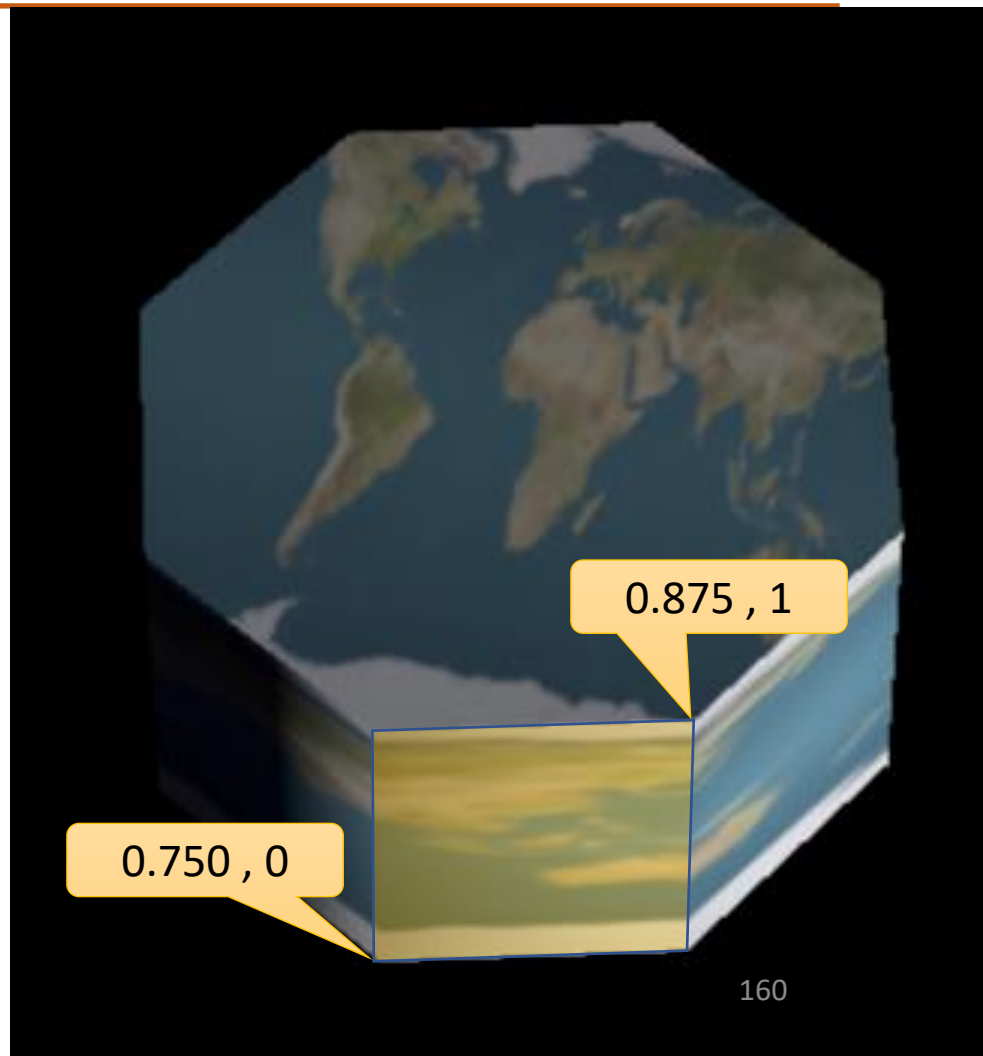
- We can now use these coordinates to map our image to the 3D surface.
- We already saw **SphereGeometry** result
- **BoxGeometry** does this for us:



UV Mapping

- And this is the result for **CylinderGeometry** with 8 faces:





UV Mapping

- 3D Modeling software handles complex UV Mapping

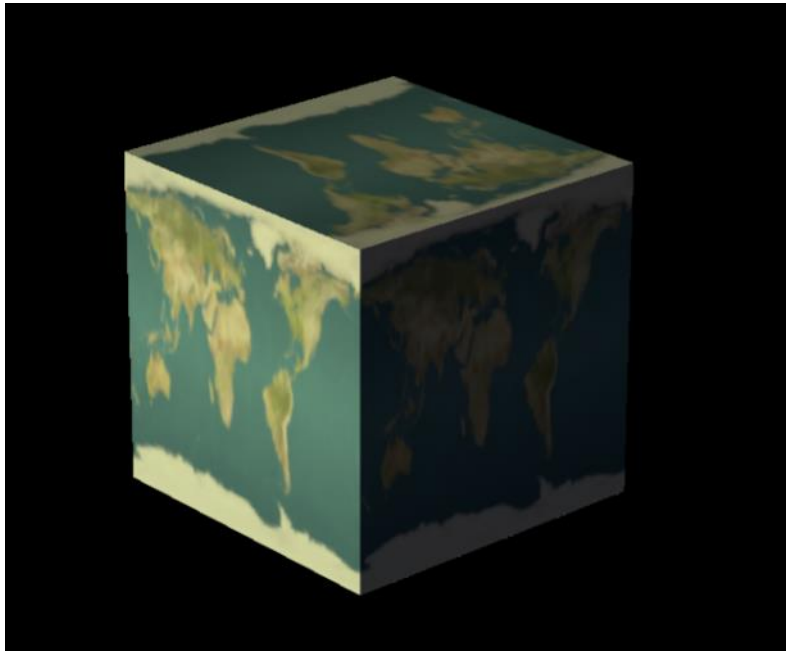
https://threejs.org/manual/#en/load-obj

UV Mapping

You can set your own UV Mapping at code using the '**uv**' attribute of a **BufferGeometry**.

```
const uvs = [
        //First triangle, of first face
        0.0, 1.0,
        0.0, 0.0,
        1.0, 1.0,
        //Second triangle, of first face
        0.0, 0.0,
        1.0, 0.0,
        1.0, 1.0,
        //…
]
geometry.setAttribute( 'uv',
    new THREE.BufferAttribute( new Float32Array(uvs), 2 ));
```

See the full example at:
https://jsfiddle.net/dx548euh/



Types of textures

- You can use **TextureLoader** to load a **Texture** object from an image that the browser supports like PNG, JPG, GIF and BMP.
- Speciliazed formats like HDR, EXR and TGA need specific loaders like **TGALoader**, and also return a **Texture** object
- Besides the 2D Textures, three.js supports other types of textures such as video textures, 3D textures, canvas textures, cube textures (that we already used in the environment map), compressed textures and more.

Texture Class

The Texture class is a wrapper around an HTML image element with extra settings specific for textures.

Normally we use TextureLoader to create new Texture objects from images

We then assign the Texture object to a Material

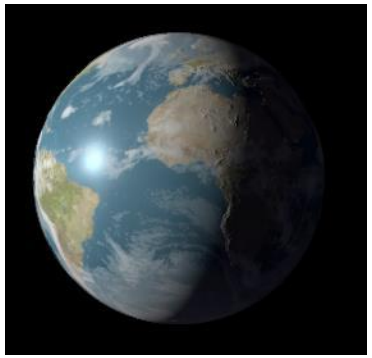
Note that TextureLoader may take some time to load the image, if the scene is rendered before the load is complete the Material will show as black

```
const textureRoughness =
    new THREE.TextureLoader()
        .load('earthroughness1k.jpg');
material.roughnessMap = textureRoughness;
```

Texture Class

- You can adjust several Texture parameters, including:
  - .offset
  - .repeat
  - .rotation
  - .center
  - .flipY
- Note that these (except .rotation and .flipY) are Vector2 so you can use .set(x,y) to adjust them
- You can try this at: https://jsfiddle.net/5mztorwy/

(tip: remove the clouds first…)

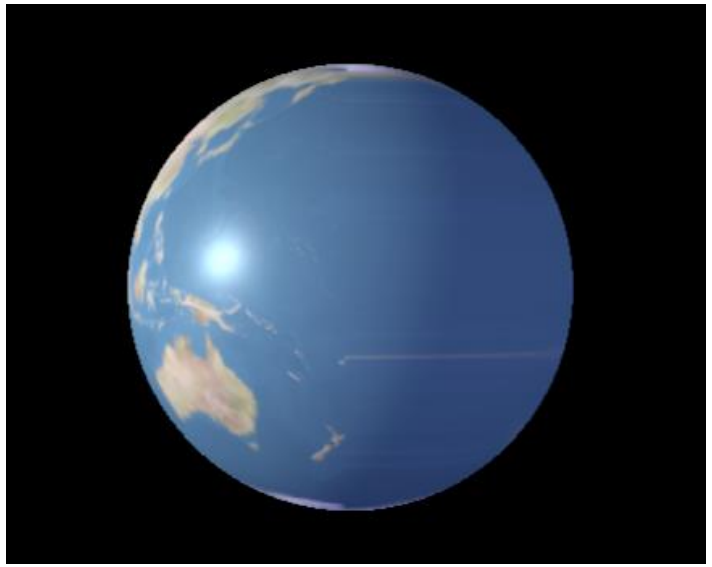


Texture Class

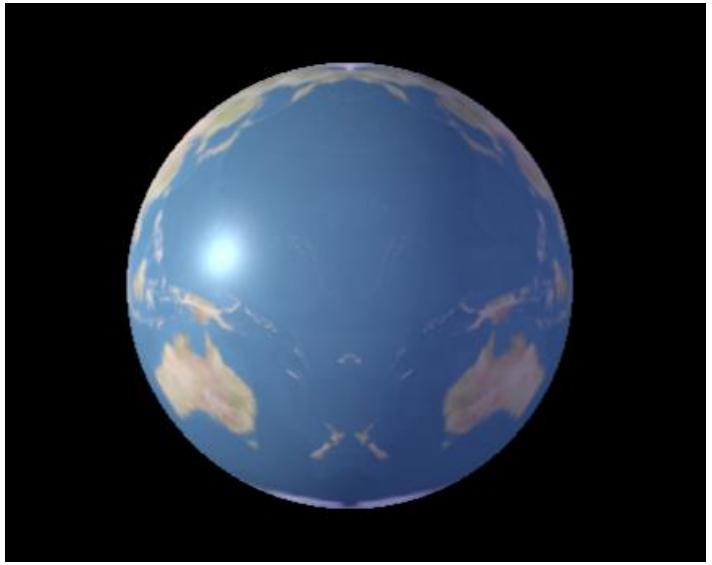
- Other important properties include:
  - **.wrapS** and **.wrapT** than define what happens when the mapping extends beyond the edge of the original image…

.wrapS examples (with texture.offset.set(0.5,0);):

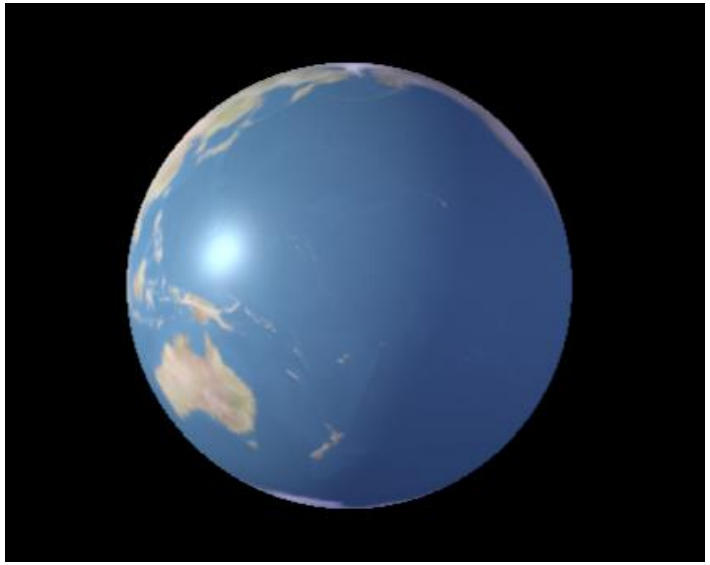
THREE.ClampToEdgeWrapping



THREE.MirroredRepeatWrapping



THREE.RepeatWrapping



Texture Class

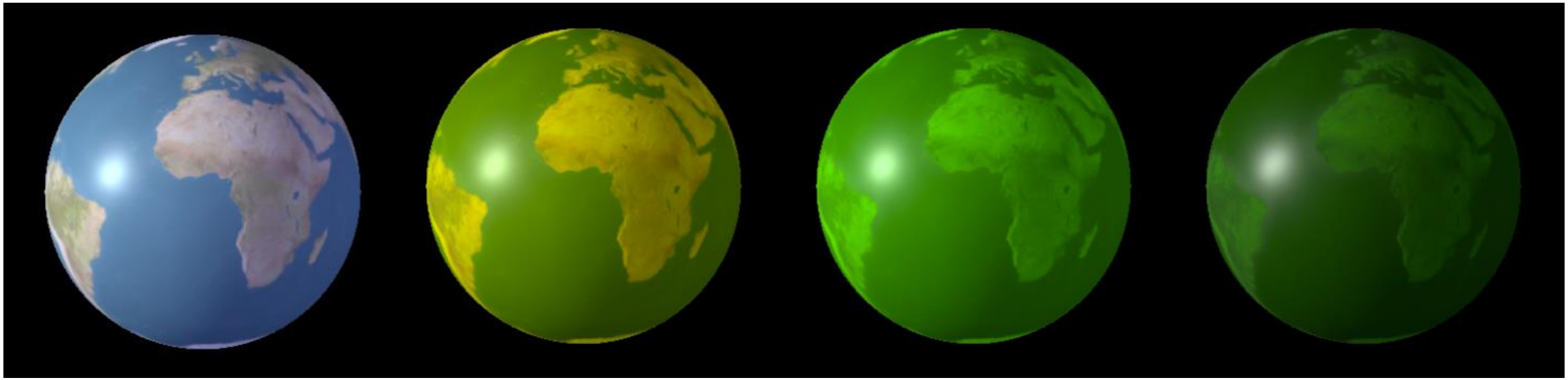
- Other important properties include:
  - **.minFilter** and **.magFilter** control how the texture is filtered when seen far away or up close.
  - The default is THREE.LinearFilter, which takes the four closest texels and bilinearly interpolates among them. The other option is THREE.NearestFilter, which uses the value of the closest texel (faster, but can look pixelated)

Texture & Material Color

- Color of the material modulates the texture. If the color is not 0xffffff (white), then it will change the result.

material.color

0xffffff 0xffff00 0x00ff00 0x004000



Where can I get textures?

- Three.js has its own collection here:
  - https://github.com/mrdoob/three.js/tree/dev/examples/textures
- And there are free sites like:
  - https://freepbr.com/
  - https://freestocktextures.com/
  - http://texturer.com/
  - https://3dtextures.me/
  - And others…
- Or sites with some free content like:
  - https://www.poliigon.com/textures/free

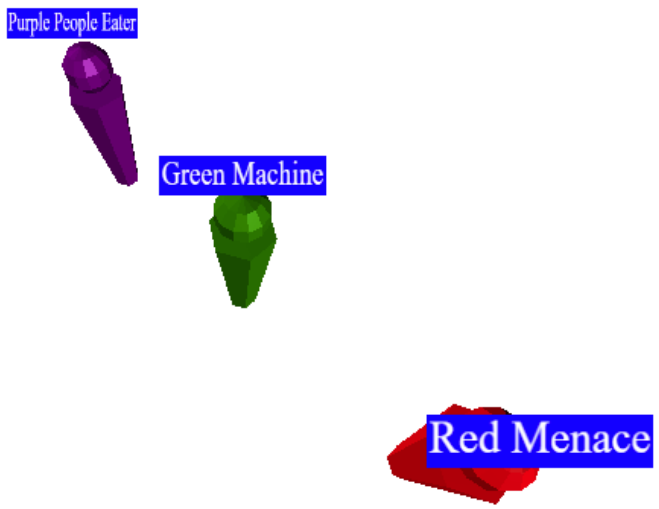
Sprites

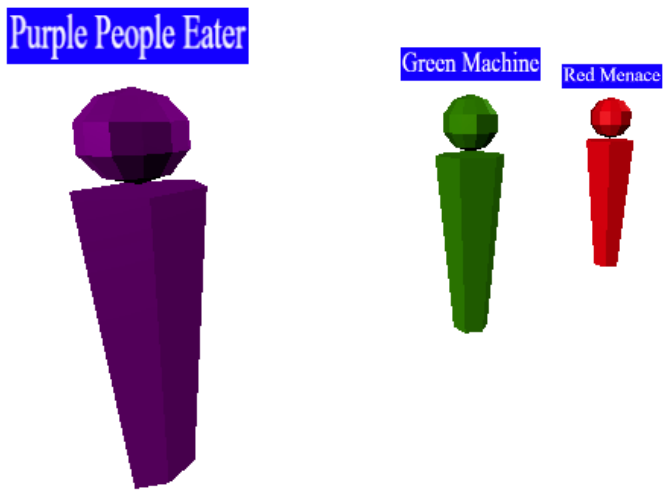
- In computer graphics a “Billboard” is some object that is always facing the camera
- In three.js this concept is implemented using **Sprite** and **SpriteMaterial**
- A **SpriteMaterial** is used like other three.js materials with **color**, **map**, **transparent** and so forth.
- A **Sprite** is a special **Mesh** object with only the **SpriteMaterial** and no **Geometry**.

More details at: https://threejs.org/manual/#en/billboards

Sprites

- A typical situation where you want to use a Sprite is for labeling objects

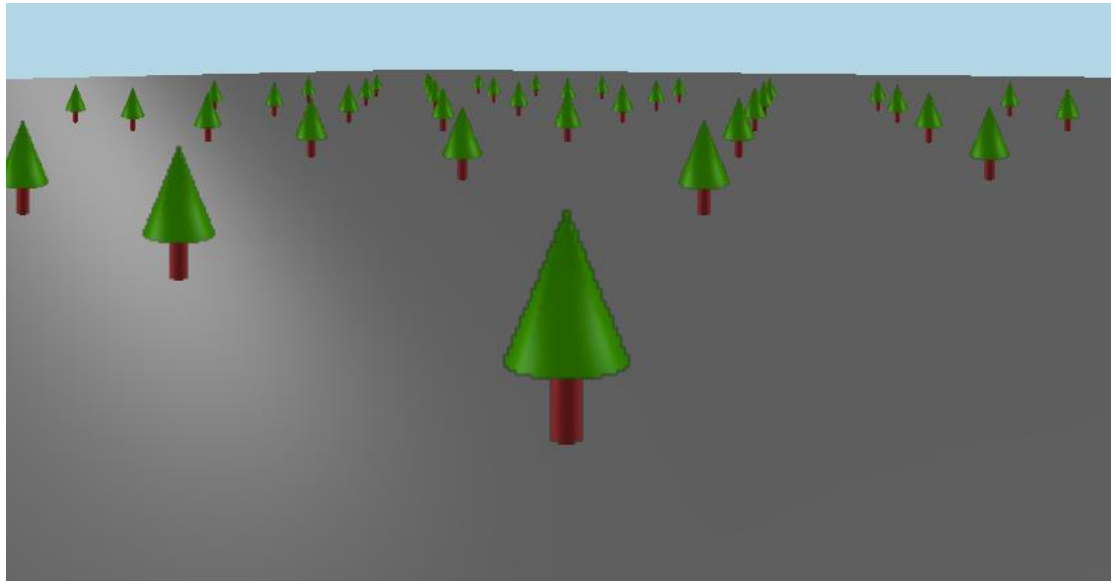


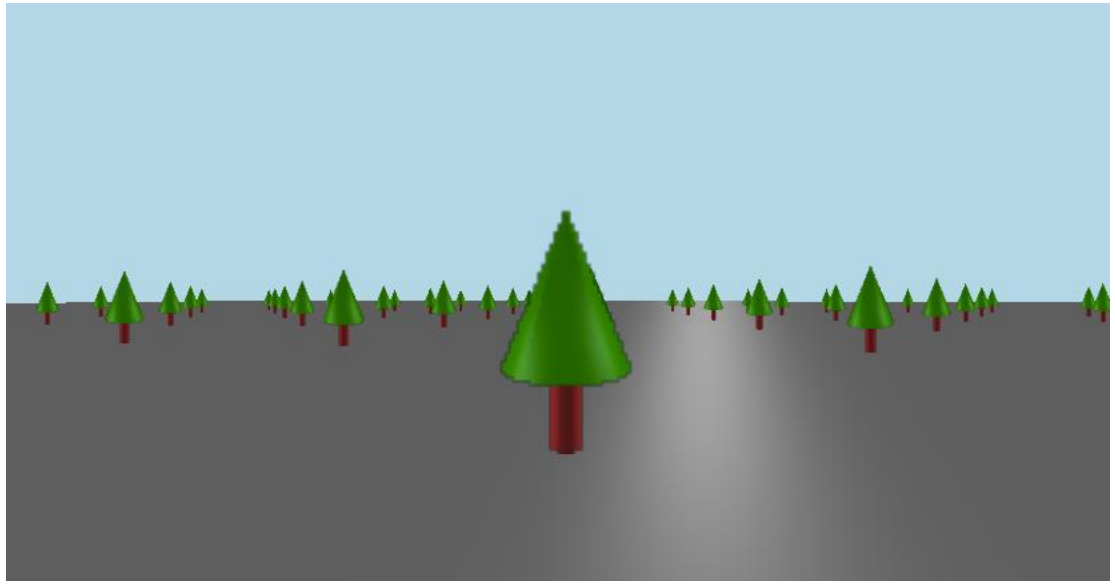


More details at: https://threejs.org/manual/#en/billboards

Sprites

- Another situation is where you want to use a pre-rendered image to “fake” 3D objects, when there are multiple of this objects in the scene. This can be much faster then rendering all the original objects.





More details at: https://threejs.org/manual/#en/billboards

UI: lil-gui

One of the most popular widget mechanisms for three.js was **dat.gui**, that as now been replaced by **lil-gui**

These are lightweight graphical user interface libraries. The original dat.gui was developed by the Google Data Arts Team but it is not maintained anymore and so the switch to lil-gui.

With just a few lines of code, one can create an interface that lets you modify Javascript variables and see the results on the fly, all without having to redeploy code.

The main advantage for three.js is that is allows to quickly add a simple user interface enabling simple interaction with the 3D scene and objects

https://lil-gui.georgealways.com/

UI: lil-gui

- This User Interface library is available at the default Three.js distributions, after r131
- Note that most dat.gui code will work directly with lil-gui

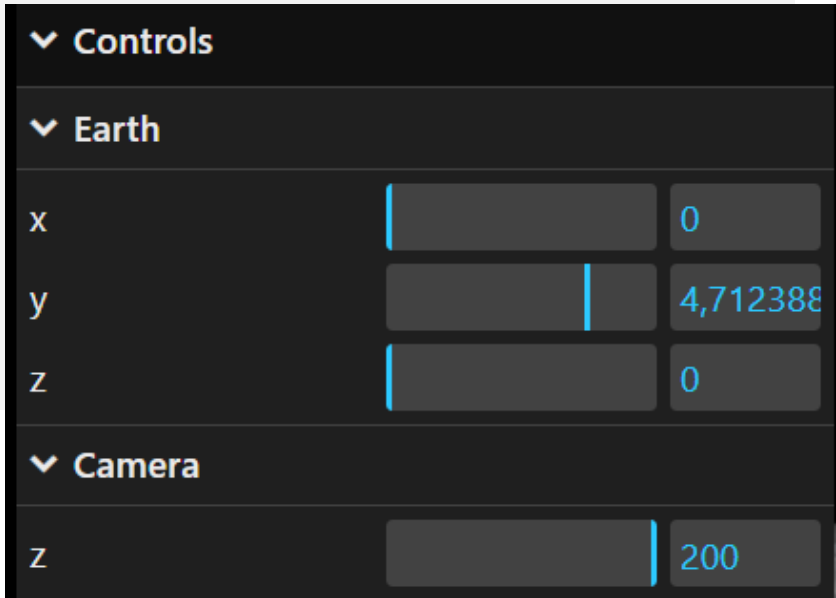
- First import the module:

```
import { GUI } from '../three/examples/jsm/libs/lil-gui.module.min.js';
```

UI: lil-gui

- Then you can simply add “folders” and object parameters directly:

```
const  gui = new GUI();
const cubeFolder = gui.addFolder('Earth')
cubeFolder.add(earthMesh.rotation, 'x', 0, Math.PI * 2)
cubeFolder.add(earthMesh.rotation, 'y', 0, Math.PI * 2)
cubeFolder.add(earthMesh.rotation, 'z', 0, Math.PI * 2)
cubeFolder.open()
const cameraFolder = gui.addFolder('Camera')
cameraFolder.add(camera.position, 'z', 100, 200)
cameraFolder.open()
```



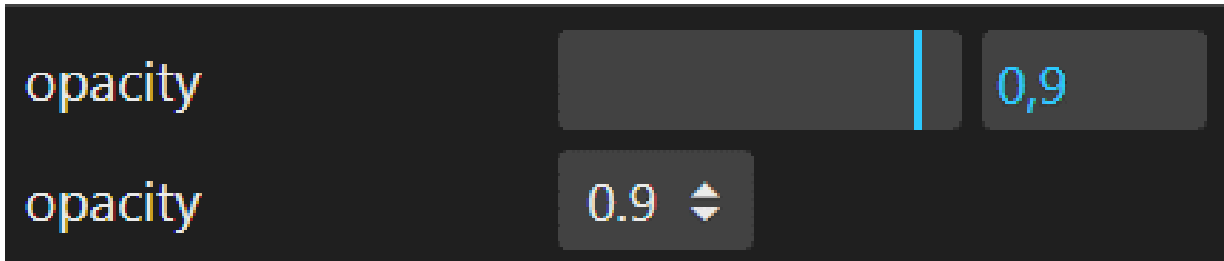
UI: lil-gui

- You can set min, max and step:

```
gui.add(cloudMaterial, 'opacity').min(0).max(1).step(0.25);
```

- And even choose between a set of named values… (dropdown)

```
gui.add(cloudMaterial, 'opacity', { None: 0, Std: 0.5, Full: 1 });
```



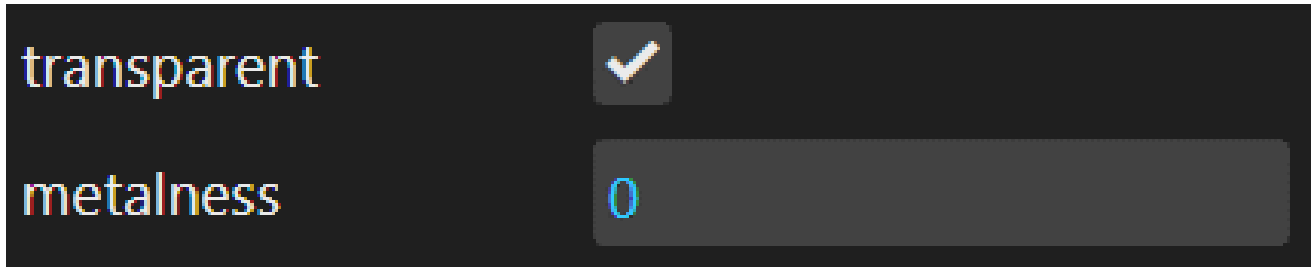
UI: lil-gui

- You can also add Boolean values

```
gui.add(cloudMaterial, 'transparent');
```

- And free values

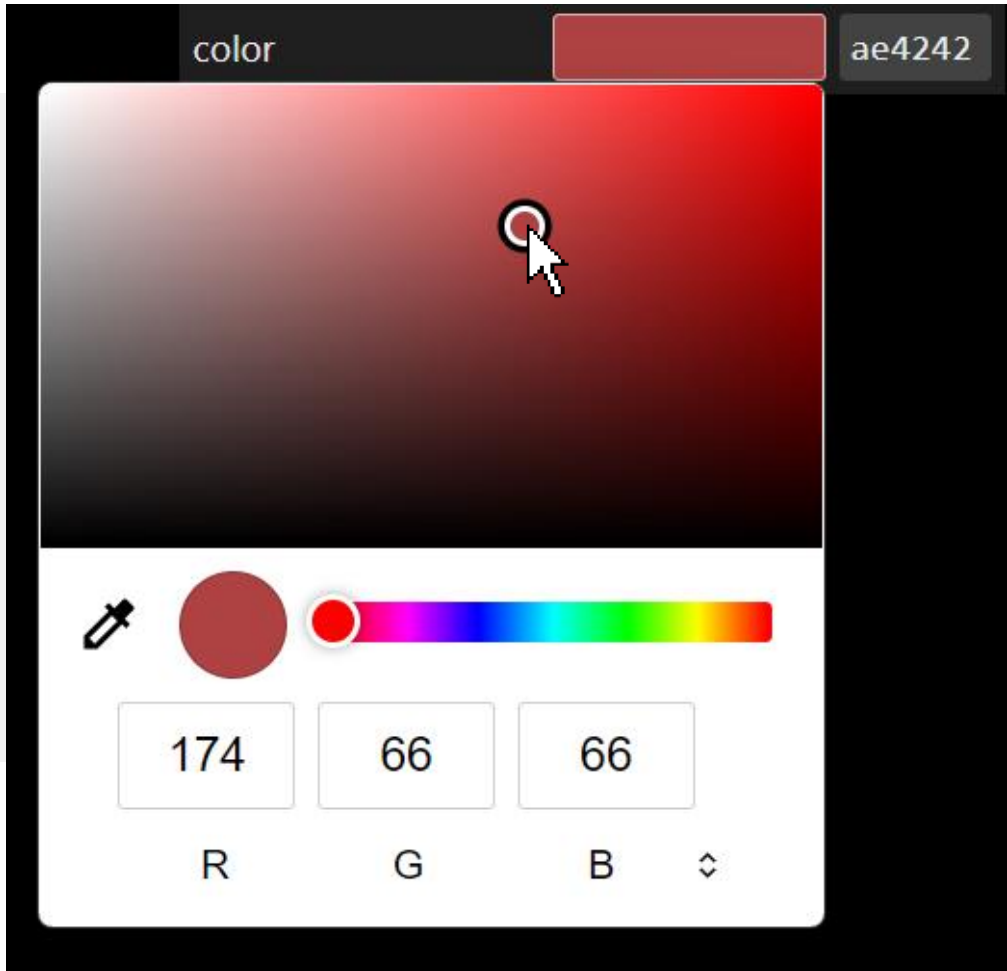
```
gui.add(cloudMaterial, 'metalness');
```



UI: lil-gui

- For Material colors you have to be a little bit more creative and use the **onChange** event.

```
var params = {
     color: 0xffffff
};
gui.addColor( params, 'color' )
      .onChange( function() {
          cloudMaterial.color.set(
params.color );
       }
);
```



See a GUI.dat example with multiple object color (and a RayCaster) at:
https://codepen.io/prisoner849/pen/KrmvMZ

UI: lil-gui

- You can also make sure the gui is updated to the latest values with **listen()**:

```
cubeFolder.add(earthMesh.rotation, 'y', 0, Math.PI * 2).listen();
```

- Now the slider with change with the rotation of the earth

UI: lil-gui

- You can add any object to the GUI

```
var animParams = {
    speedEarth: 1.0,
    speedClouds: 1.0
};
gui.add(animParams, 'speedEarth')
    .min(-2).max(2).step(0.1);
gui.add(animParams, 'speedClouds')
    .min(-2).max(2).step(0.1);
```

```
var animate = function () {
    requestAnimationFrame( animate );
    cloudMesh.rotation.y
        +=
0.001*animParams.speedClouds;
    earthMesh.rotation.y
        +=
0.0021*animParams.speedEarth;
    renderer.render( scene, camera );
};
```

Picking - RayCaster

Picking refers to the process of figuring out which object a user clicked on or touched.

Probably the most common way of picking is by doing **raycasting** which means to cast a ray from the mouse through the frustum of the scene and computing which objects that ray intersects.

First, we'd take the position of the mouse. We'd convert that into world space by applying the camera's projection and orientation. We'd compute a ray from the near plane of the camera's frustum to the far plane. Then, for every triangle of every object in the scene we'd check if that ray intersects that triangle.

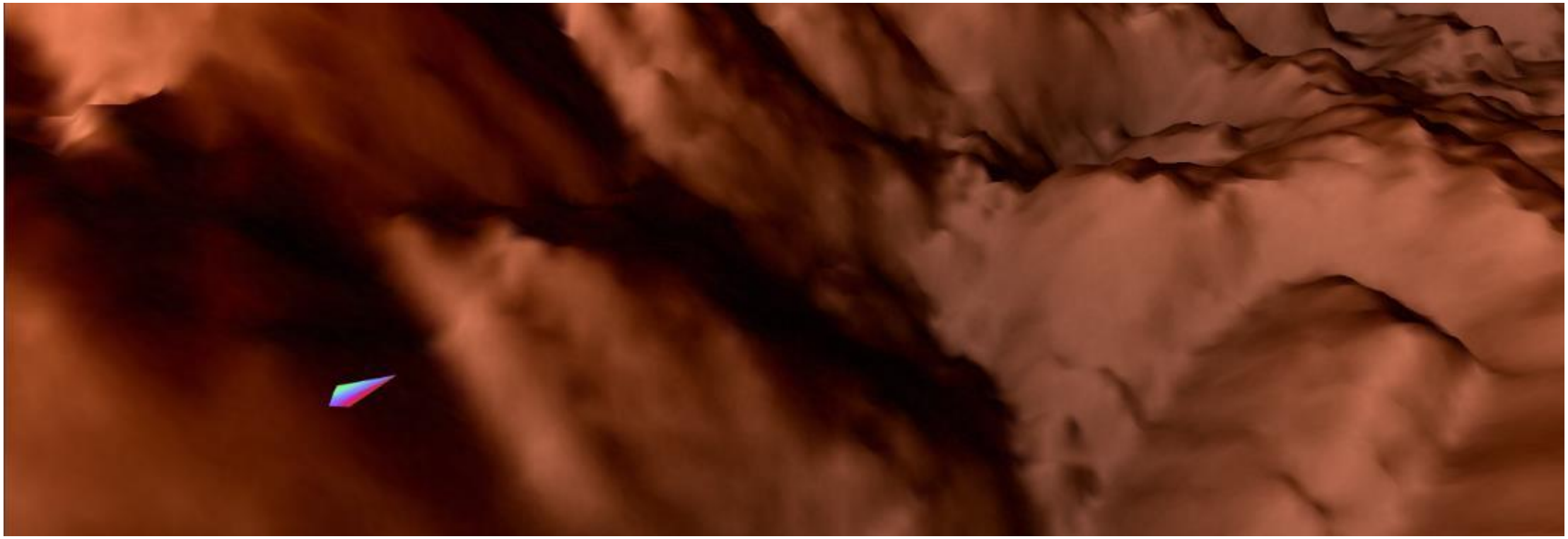
Based on: https://threejs.org/manual/#en/picking

Example 1

- https://threejs.org/manual/examples/picking-raycaster.html (+ code editor)

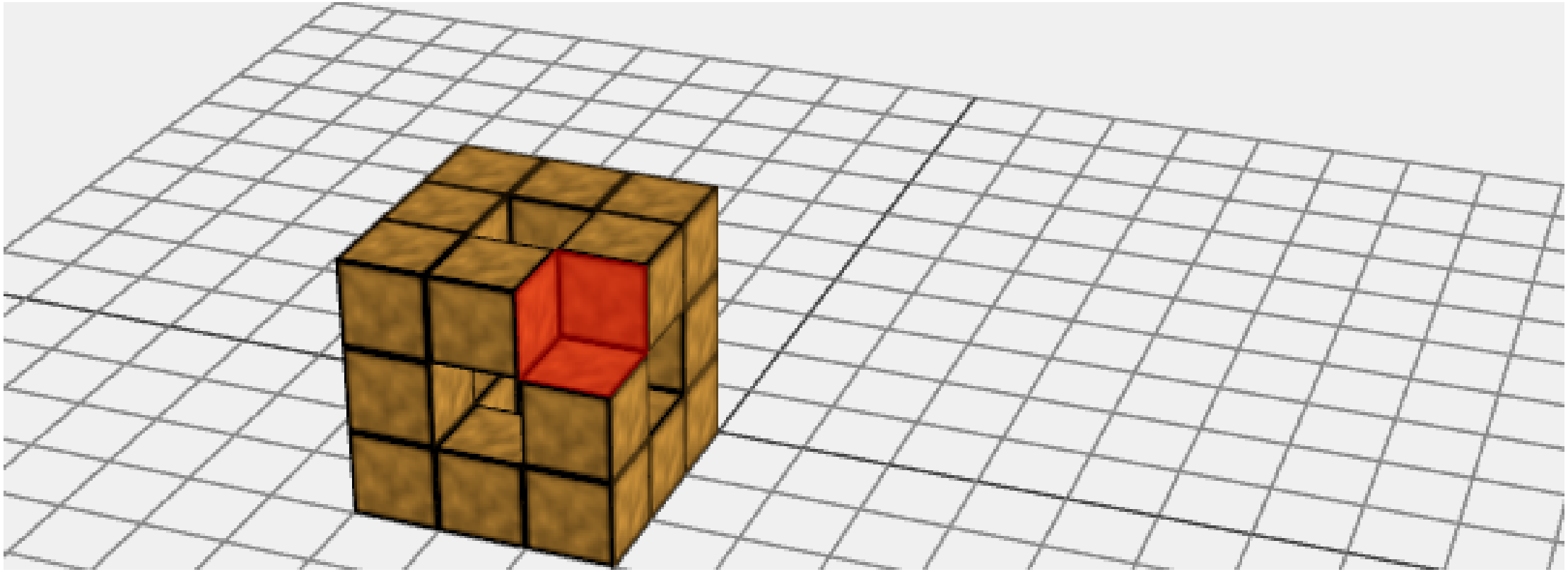
Example 2

- https://threejs.org/examples/#webgl_geometry_terrain_raycast



Example 3

- https://threejs.org/examples/?q=raycast#webgl_interactive_voxelpainter



Picking - RayCaster

- Three.js provides a **RayCaster** class that does this for us. Here is a usage example:

```
class PickHelper {
  constructor() {
    this.raycaster = new THREE.Raycaster();
    this.pickedObject = null;
    this.pickedObjectSavedColor = 0;
  }
  pick(normalizedPosition, scene, camera, time) {
    // restore the color if there is a picked object
    if (this.pickedObject) {
      this.pickedObject.material.emissive.setHex(
    this.pickedObjectSavedColor
      );
      this.pickedObject = undefined;
    }
    // cast a ray through the frustum
    this.raycaster.setFromCamera(normalizedPosition,
camera);
    // get the list of objects the ray intersected
    const intersectedObjects =
this.raycaster.intersectObjects(scene.children);
    if (intersectedObjects.length) {
      // pick the first object. It's the closest one
      this.pickedObject = intersectedObjects[0].object;
      // save its color
      this.pickedObjectSavedColor =
this.pickedObject.material.emissive.getHex();
      // set its emissive color to flashing red/yellow
      this.pickedObject.material.emissive.setHex(
              (time * 8) % 2 > 1 ? 0xFFFF00 : 0xFF0000
      );
}  }  }
```

Picking - RayCaster

- First, we define the class and base objects

```
class PickHelper {
  constructor() {
    this.raycaster = new THREE.Raycaster();
    this.pickedObject = null;
    this.pickedObjectSavedColor = 0;
  }
```

Picking - RayCaster

- Now we define the pick method, starting with reset the previous object color

```
pick(normalizedPosition, scene, camera, time) {
    // restore the color if there is a picked object
    if (this.pickedObject) {
      this.pickedObject.material.emissive.setHex(
    this.pickedObjectSavedColor
      );
      this.pickedObject = undefined;
    }
```

Picking - RayCaster

- … and then cast the ray and get the list of objects

```
    // cast a ray through the frustum
   this.raycaster.setFromCamera(normalizedPosition, camera);
    // get the list of objects the ray intersected
    const intersectedObjects =
this.raycaster.intersectObjects(scene.children);
```

Picking - RayCaster

- If there are objects, the first is the closest one and we can now change the color

```
      if (intersectedObjects.length) {
       // pick the first object. It's the closest one
       this.pickedObject = intersectedObjects[0].object;
       // save its color
       this.pickedObjectSavedColor = this.pickedObject.material.emissive.getHex();
       // set its emissive color to flashing red/yellow
       this.pickedObject.material.emissive.setHex(
               (time * 8) % 2 > 1 ? 0xFFFF00 : 0xFF0000
       );
}  }  }
```

Picking - RayCaster

- We can call this function only when the user presses the mouse **down**, but for this example we'll pick every frame whatever is under the mouse. To do this we first need to track where the mouse is…

Picking - RayCaster

- Here is the code for saving the mouse position

```
const pickPosition = {x: 0, y: 0};
clearPickPosition();
//...
function getCanvasRelativePosition(event) {
  const rect = canvas.getBoundingClientRect();
  return {
    x: (event.clientX - rect.left) * canvas.width  /
rect.width,
    y: (event.clientY - rect.top ) * canvas.height /
rect.height,
  };
}

function setPickPosition(event) {
  const pos = getCanvasRelativePosition(event);
  pickPosition.x = (pos.x / canvas.width ) *  2 - 1;
  pickPosition.y = (pos.y / canvas.height) * -2 + 1;  //
note we flip Y
}

function clearPickPosition() {
  // unlike the mouse which always has a position
  // if the user stops touching the screen we want
  // to stop picking. For now we just pick a value
  // unlikely to pick something
  pickPosition.x = -100000;
  pickPosition.y = -100000;
}

window.addEventListener('mousemove', setPickPosition);
window.addEventListener('mouseout', clearPickPosition);
window.addEventListener('mouseleave',
clearPickPosition);
```

Picking - RayCaster

- First, we must have a variable where to store the value, and we clear it:

```
const pickPosition = {x: 0, y: 0};
clearPickPosition();

//...
```

Picking - RayCaster

- We have an aux function to calculate the relative position -1 to 1 with the canvas area:

```
function getCanvasRelativePosition(event) {
  const rect = canvas.getBoundingClientRect();
  return {
    x: (event.clientX - rect.left) * canvas.width  / rect.width,
    y: (event.clientY - rect.top ) * canvas.height / rect.height,
  };
}
```

Picking - RayCaster

- That is used to store the pick position when there is a mouse event:

```
function setPickPosition(event) {
  const pos = getCanvasRelativePosition(event);
  pickPosition.x = (pos.x / canvas.width ) *  2 - 1;
  pickPosition.y = (pos.y / canvas.height) * -2 + 1;  // note we flip Y
}
```

Picking - RayCaster

- And an aux function to clear the pick position:

```
function clearPickPosition() {
  // unlike the mouse which always has a position
  // if the user stops touching the screen we want
  // to stop picking. For now we just pick a value
  // unlikely to pick something
  pickPosition.x = -100000;
  pickPosition.y = -100000;
}
```

Picking - RayCaster

- And, finally, add the event listeners

```
window.addEventListener('mousemove', setPickPosition);
window.addEventListener('mouseout', clearPickPosition);
window.addEventListener('mouseleave', clearPickPosition);
```

Picking - RayCaster

- Notice we're recording a **normalized mouse position**. Regardless of the size of the canvas we need a value that goes from -1 on the left to +1 on the right. Similarly, we need a value that goes from -1 on the bottom to +1 on the top.

- While we're at it lets support mobile (touch) as well

Picking - RayCaster

- Here is the code for saving the touch position

```
window.addEventListener('touchstart', (event) => {
  // prevent the window from scrolling
  event.preventDefault();
  setPickPosition(event.touches[0]);
}, {passive: false});

window.addEventListener('touchmove', (event) => {
  setPickPosition(event.touches[0]);
});

window.addEventListener('touchend', clearPickPosition);
```

Picking - RayCaster

- And, finally, the code in the render function where we call the pick method.

```
const pickHelper = new PickHelper();
function render(time) {
  time *= 0.001;  // convert to seconds;
  //...

  pickHelper.pick(pickPosition, scene, camera, time);
  renderer.render(scene, camera);
  //...
```

Next week…

- Fog
- Render Targets
- Post Processing

Fog

- Fog in a 3D engine is generally a way of fading to a specific color based on the distance from the camera. In three.js you add fog by creating **Fog** or **FogExp2** object and setting it on the scene's **fog** property.

- **Fog** grows linearly and lets you choose **near** and **far** settings which are distances from the camera. Anything closer than near is unaffected by fog. Anything further than far is completely hidden by the fog color. Parts between near and far fade from their material color to the fog color.

- On the other hand, **FogExp2** grows exponentially with distance from the camera, and the basic parameter is the fog **density**.

Based on: https://threejs.org/manual/#en/fog

Fog – Linear

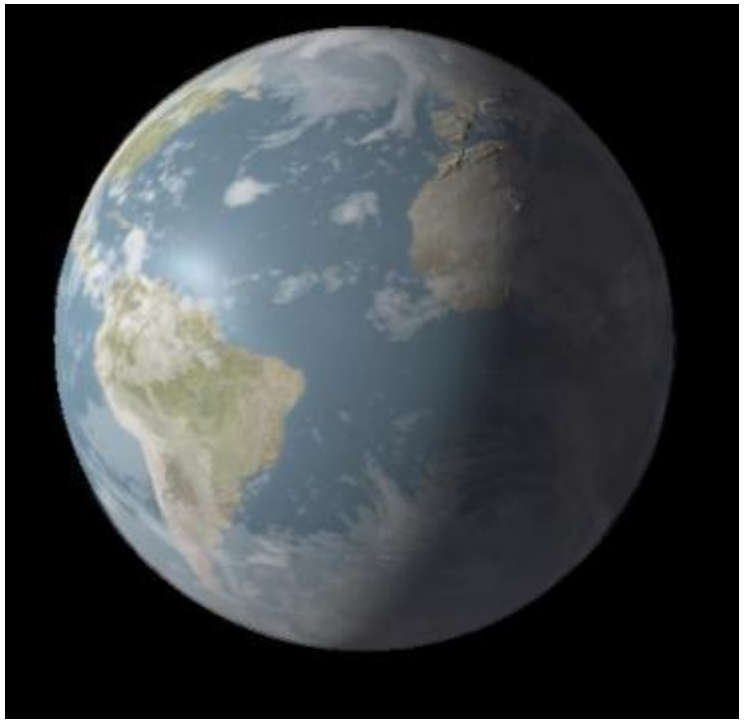
Adding fog into the scene is as simple as:

```
const color = 0xFFFFFF;  // white
const near = 180;
const far = 200;
scene.fog = new THREE.Fog(color, near, far);
```

Fog – Exponential

And these are the parameters for **FogExp2**:

```
const color = 0xFFFFFF;
const density = 0.002;
scene.fog = new THREE.FogExp2(color, density);
```



FogExp2 is more realistic, but can be tricky to get the results you want.

Fog – Background

You might have guessed by the earth example that fog only applies to **things that are rendered**, and not the background.

So, it makes sense to add:

```
scene.background = color;
```

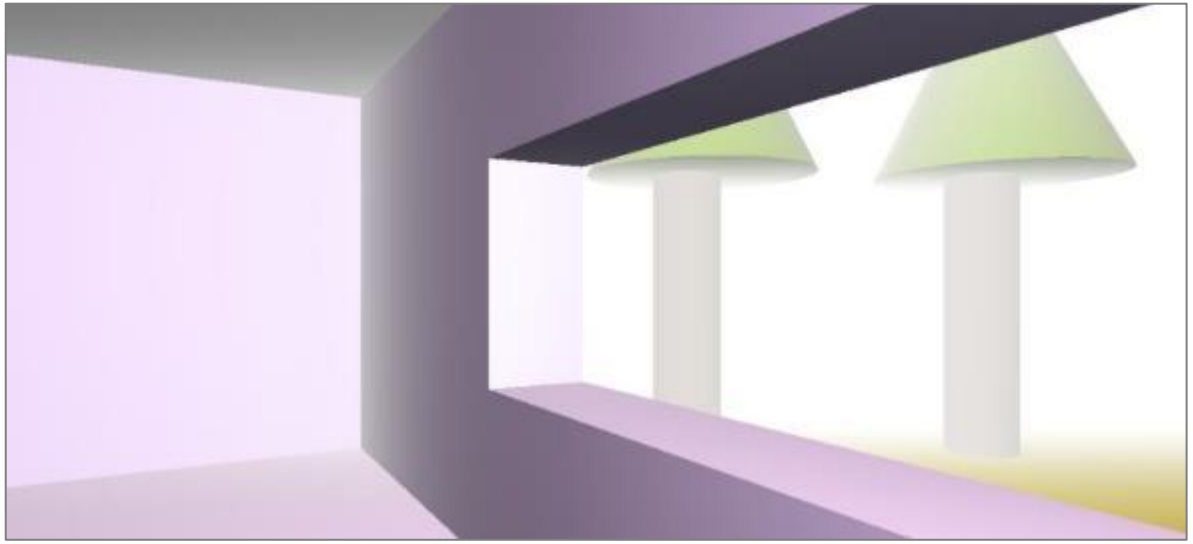
Check out the interactive example at:

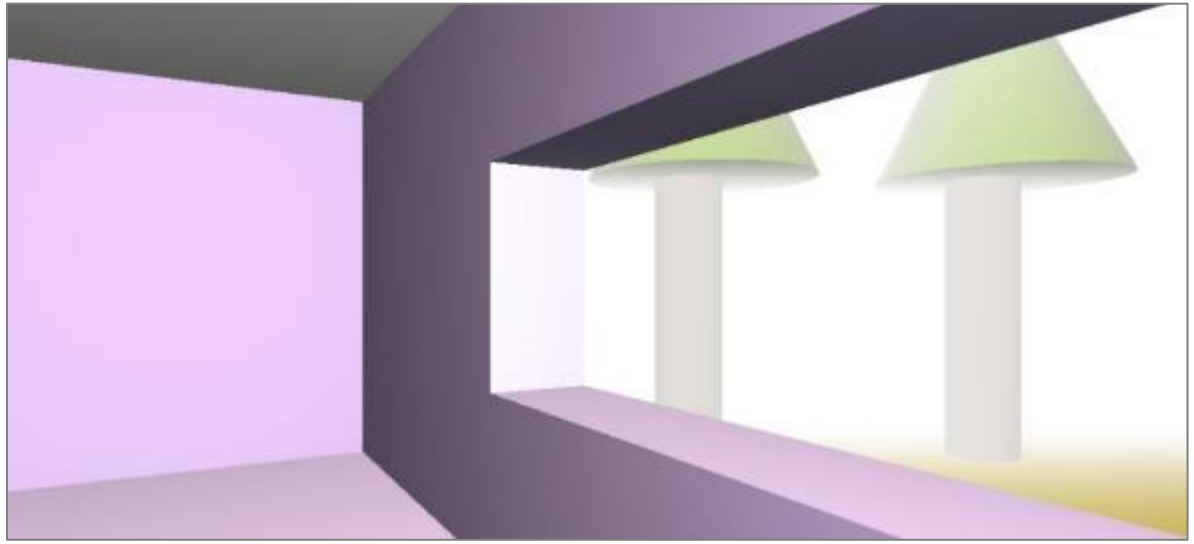
https://threejsfundamentals.org/threejs/threejs-fog-gui.html

Fog – Materials

Materials have a Boolean flag **fog**, that enables or disables the fog effect.

This is useful when you have a car or a house with fog outside, but don’t want fog inside the main object.





Render Targets

A render target in three.js is basically a texture you can render to. After you render to it you can use that texture like any other texture.

Rendering to a render target is almost exactly the same as normal rendering. First, we create a WebGLRenderTarget with the dimensions that we want for our texture, in this case our final target is a cube, so we will use 512 for both dimensions.

```
const rt_width = 512;
const rt_height = 512;
const renderTarget = new THREE.WebGLRenderTarget(rt_width, rt_height);
```

Render Target – the Scene

Then we create a **Scene** (with Fog and all!)

```
var rt_scene = new THREE.Scene();

const rt_color = new THREE.Color('white');
const rt_near = 180;
const rt_far = 200;

rt_scene.fog = new THREE.Fog(rt_color, rt_near, rt_far);
rt_scene.background = rt_color;
```

Render Target - Camera

Then a **Camera**

Note that the aspect ratio is based on the dimensions of the renderTarget

```
const rt_aspect = rt_width / rt_height;
const rt_camera = new THREE.PerspectiveCamera( 15, rt_aspect, 0.1,
1000 );
rt_camera.position.set( 0, 0, 200 );
rt_camera.lookAt( 0, 0, 0 );
```

Render Target - Objects

Now we just add our objects to the **Render Target scene**…

```
// ambient light for the "dark" side of the world
const rt_light_amb = new THREE.AmbientLight(0x8080ff, 0.01);
rt_scene.add(rt_light_amb);
// etc…
```

Render Target – Main Scene

We now draw the **main scene** as usual…

```
// Setup scene
var scene = new THREE.Scene();
const renderer = new THREE.WebGLRenderer();
renderer.setPixelRatio( window.devicePixelRatio );
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
// Setup camera, etc…
// Draw main objects…
```

Render Target – Main Scene Texture

Except where we want to use our **renderTarget.texture** …

```
// Set material texture map
const texture = renderTarget.texture;
material.map = texture;
```

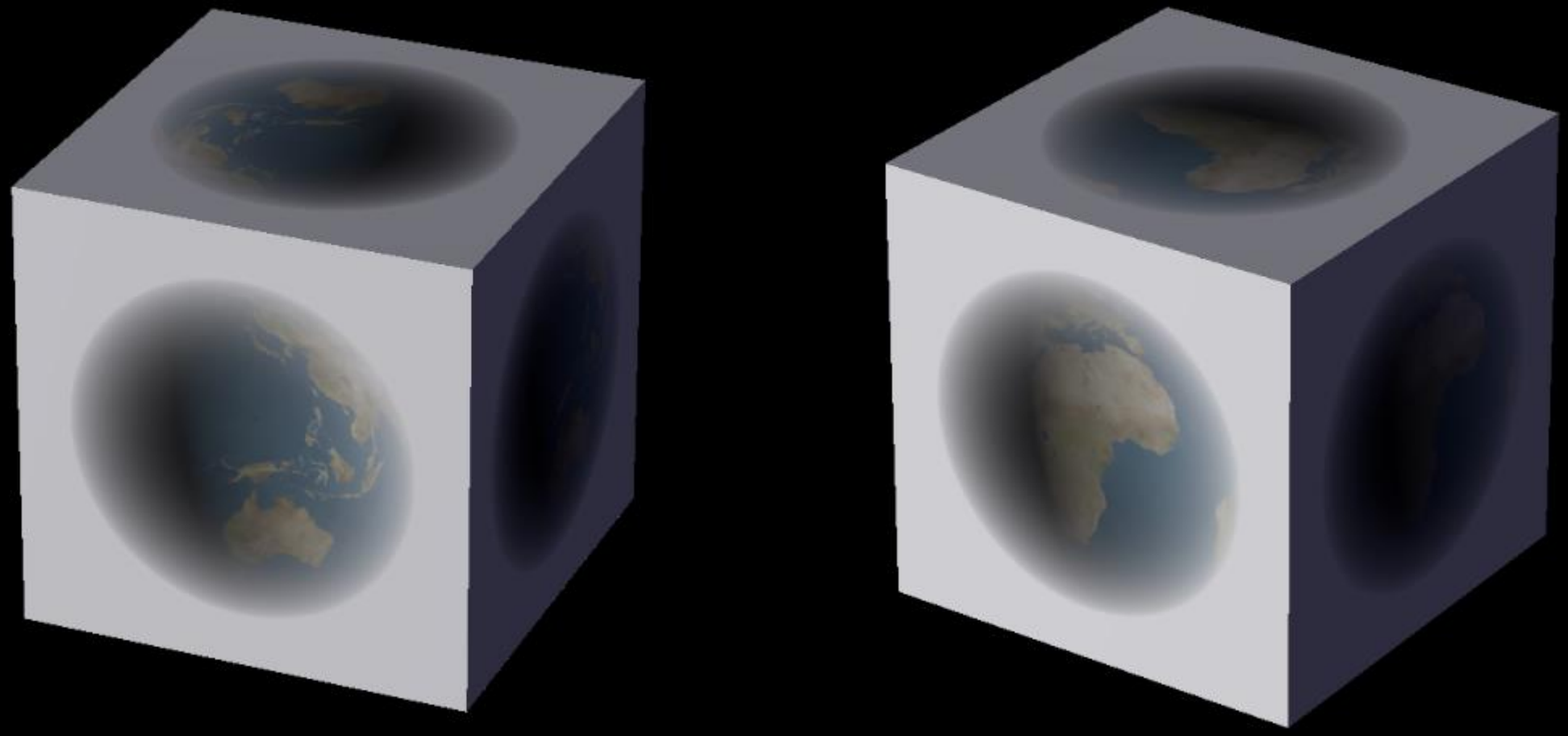
Render Target – Call the Renderer!

And, finally, we must execute both animation and render commands

```
var animate = function () {
    requestAnimationFrame( animate );

    rt_earthMesh.rotation.y += 0.021;
    renderer.setRenderTarget( renderTarget );
    renderer.render( rt_scene, rt_camera );
    renderer.setRenderTarget( null );

    cube.rotation.y += 0.01;
    renderer.render( scene, camera );
};
```

Render

Animation



You can play with the result at: https://jsfiddle.net/97epqyfz/

Render Target – Single Call

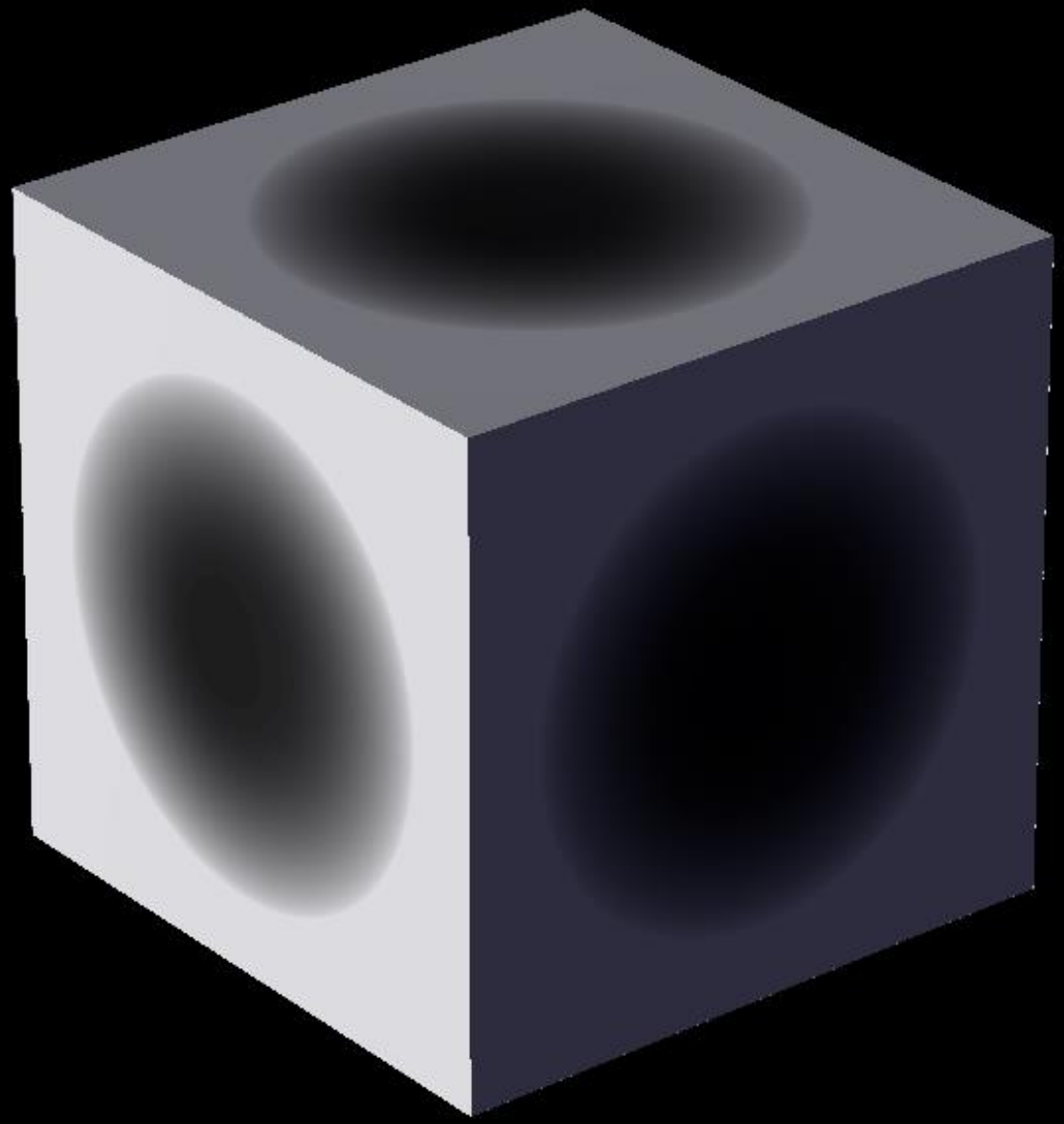
- In many situations you don’t need that texture to be changed periodically, and waste resources rendering it!
- We can just call the renderer once, and the render target texture is available

```
renderer.setRenderTarget(renderTarget);
renderer.render(rt_scene, rt_camera);
renderer.setRenderTarget(null);
```

Render Target – Single Call

- So… let’s add the code on the end of our code just before the animate() call.

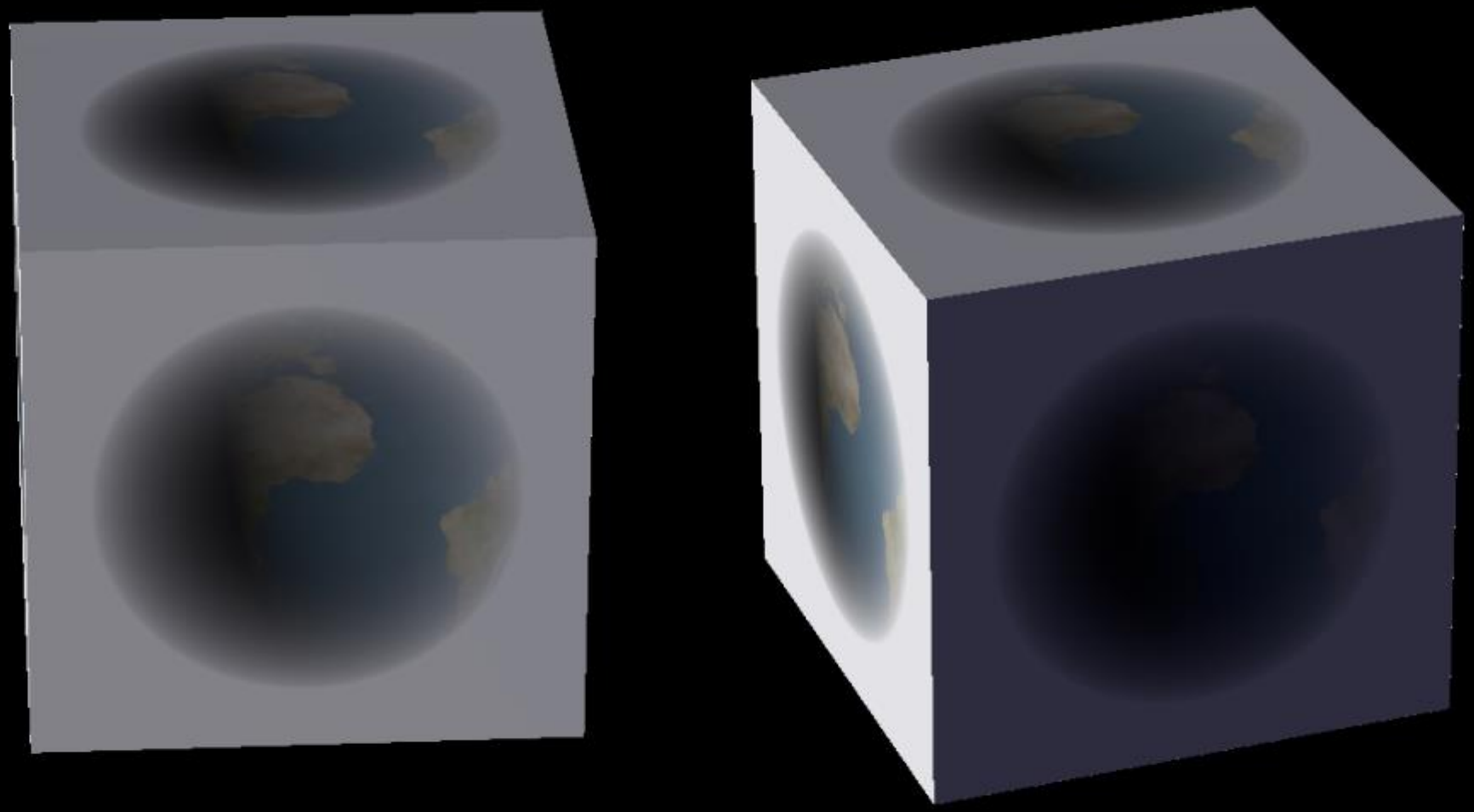
```
renderer.setRenderTarget(renderTarget);
renderer.render(rt_scene, rt_camera);
renderer.setRenderTarget(null);

animate();

</script>
</body></html>
```



Render Target – Single Call

- The problem is that we are calling the rendering process **before** the globe **texture is loaded**! This is easy to solve:

```
const texture = new THREE.TextureLoader().load('earthmap1k.jpg',
textureOnLoad);
function textureOnLoad(texture) {
    renderer.setRenderTarget(renderTarget);
    renderer.render(rt_scene, rt_camera);
    renderer.setRenderTarget(null);
}
material.map = texture;
```



Next week…

- Post Processing
- Reflection
- Physics

Post-processing

Many three.js applications render their 3D objects directly to the screen. Sometimes, however, you want to apply one or more graphical effects like Depth-Of-Field, Bloom, Film Grain or various types of Anti-aliasing. Post-processing is a widely used approach to implement such effects. First, the scene is rendered to a render target which represents a buffer in the video card's memory. In the next step one or more post-processing passes apply filters and effects to the image buffer before it is eventually rendered to the screen.

three.js provides a complete post-processing solution via **EffectComposer** to implement such a workflow.

Based on: https://threejs.org/docs/#manual/en/introduction/How-to-use-post-processing

Post-processing - Workflow

The first step in the process is to import all necessary files from the examples directory.

```
import { EffectComposer } from
'../three/examples/jsm/postprocessing/EffectComposer.js';
import { RenderPass } from
'../three/examples/jsm/postprocessing/RenderPass.js';
import { GlitchPass } from
'../three/examples/jsm/postprocessing/GlitchPass.js';
```

After all files are successfully imported, we can create our composer by passing in an instance of WebGLRenderer.

```
const composer = new EffectComposer( renderer );
```

Post-processing - Workflow

When using a composer, it's necessary to change the application's animation loop. Instead of calling the render method of **WebGLRenderer**, we now use the respective counterpart of **EffectComposer**.

```
requestAnimationFrame( animate );
//renderer.render( scene, camera);
composer.render();
```

Post-processing - Workflow

Our composer is now ready so it's possible to configure the chain of post-processing passes. These passes are responsible for creating the final visual output of the application. They are processed in order of their addition/insertion. In our example, the instance of **RenderPass** is executed first and then the instance of **GlitchPass**. The last enabled pass in the chain is automatically rendered to the screen. The setup of the passes looks like so:

```
const renderPass = new RenderPass( scene, camera );
composer.addPass( renderPass );

const glitchPass = new GlitchPass();
composer.addPass( glitchPass );
```

Post-processing - Workflow

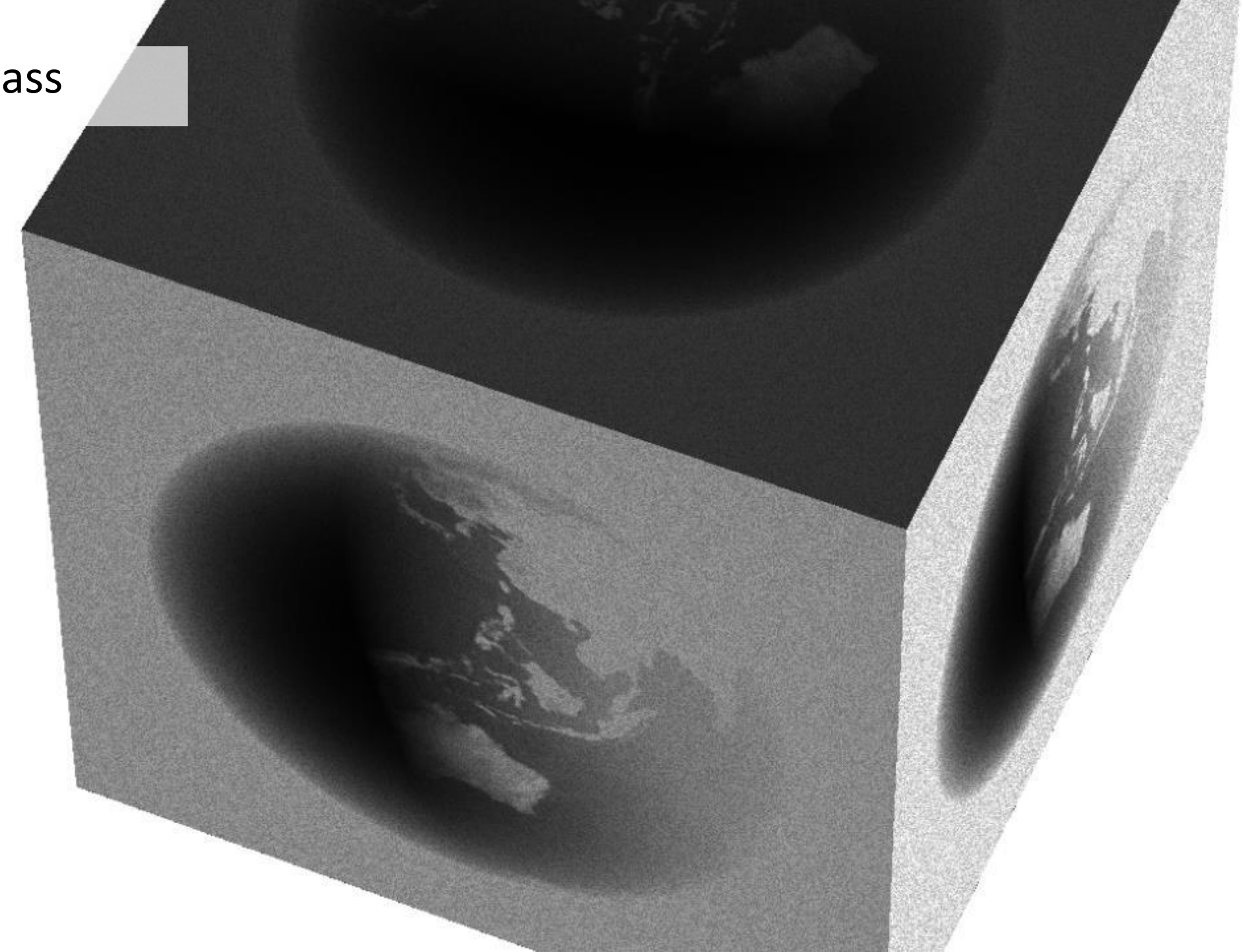
- **RenderPass** is normally placed at the *beginning of the chain* in order to provide the rendered scene as an input for the next post-processing step.
- In our case, **GlitchPass** is going to use these image data to apply a wild glitch effect.
- Check out this live example to see it in action:

https://threejs.org/examples/webgl_postprocessing_glitch

Post-Processing - Examples

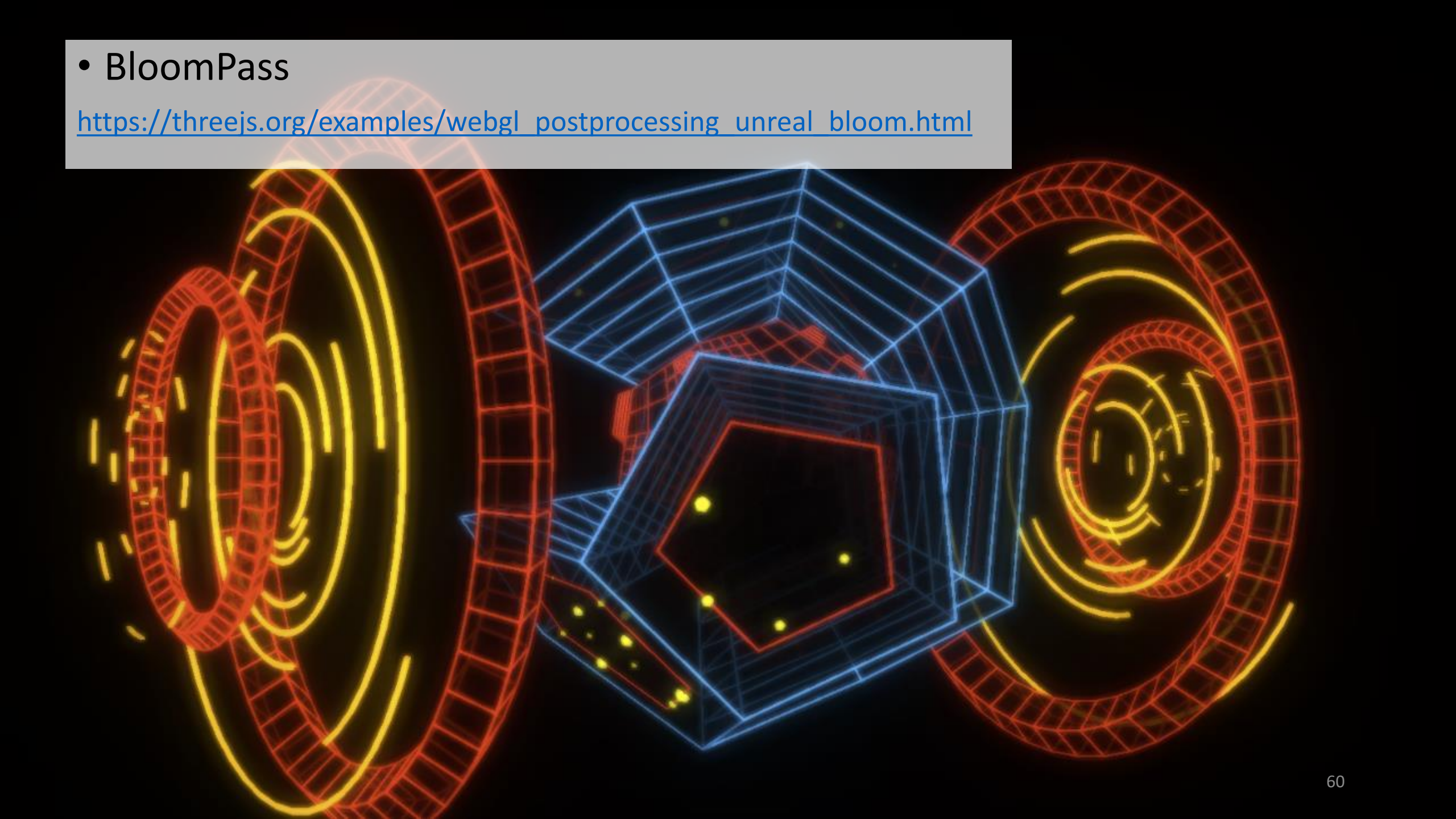
- Three.js includes several post-processing examples at examples/jsm/postprocessing
- https://github.com/mrdoob/three.js/tree/dev/examples/jsm/postprocessing

- FilmPass



- BloomPass

https://threejs.org/examples/webgl_postprocessing_unreal_bloom.html



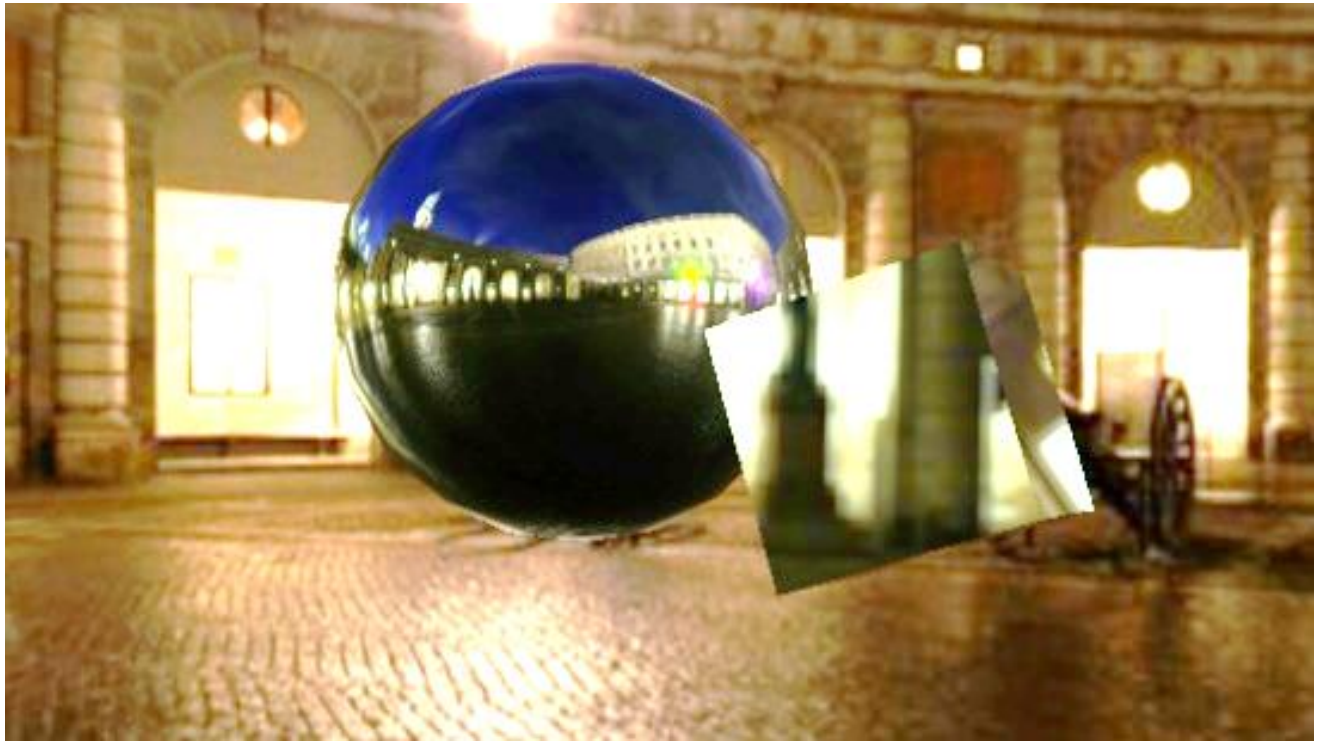
Post-Processing - Custom

- You can also use custom post-processing but that is out of the scope of SGRAI

Dynamic Reflections

We have already seen the use of **Cube Mapping** and **Environment Mapping** to simulate simple reflections of the environment.

But what if we want **real time reflections** on your objects?



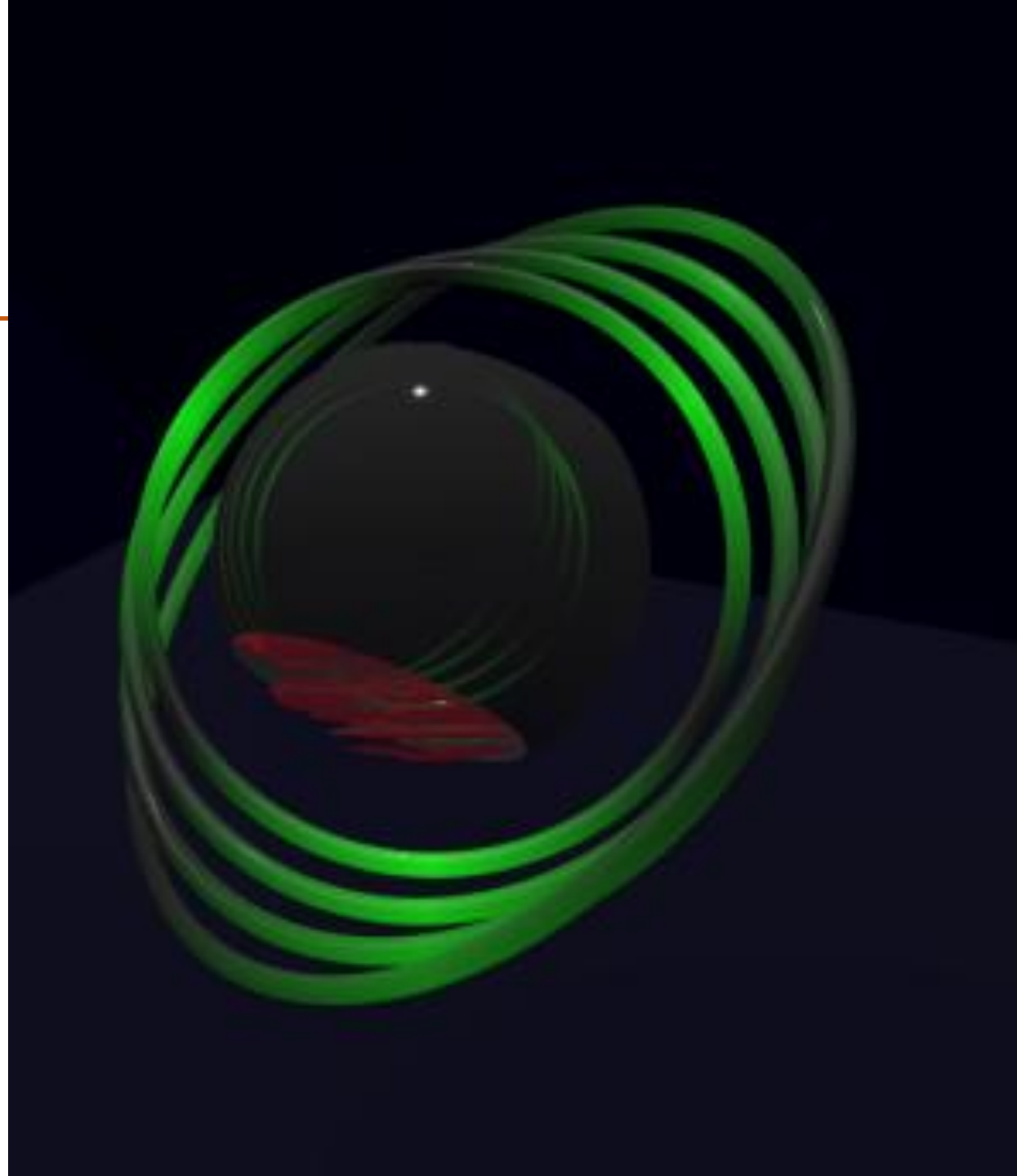
Based on https://pierfrancesco-soffritti.medium.com/dynamic-reflections-in-three-js-2d46f3378fc4

Dynamic Reflections

- Here is one example:

http://pierfrancescosoffritti.github.io/doodles/08_NeonsNStuff/index.html

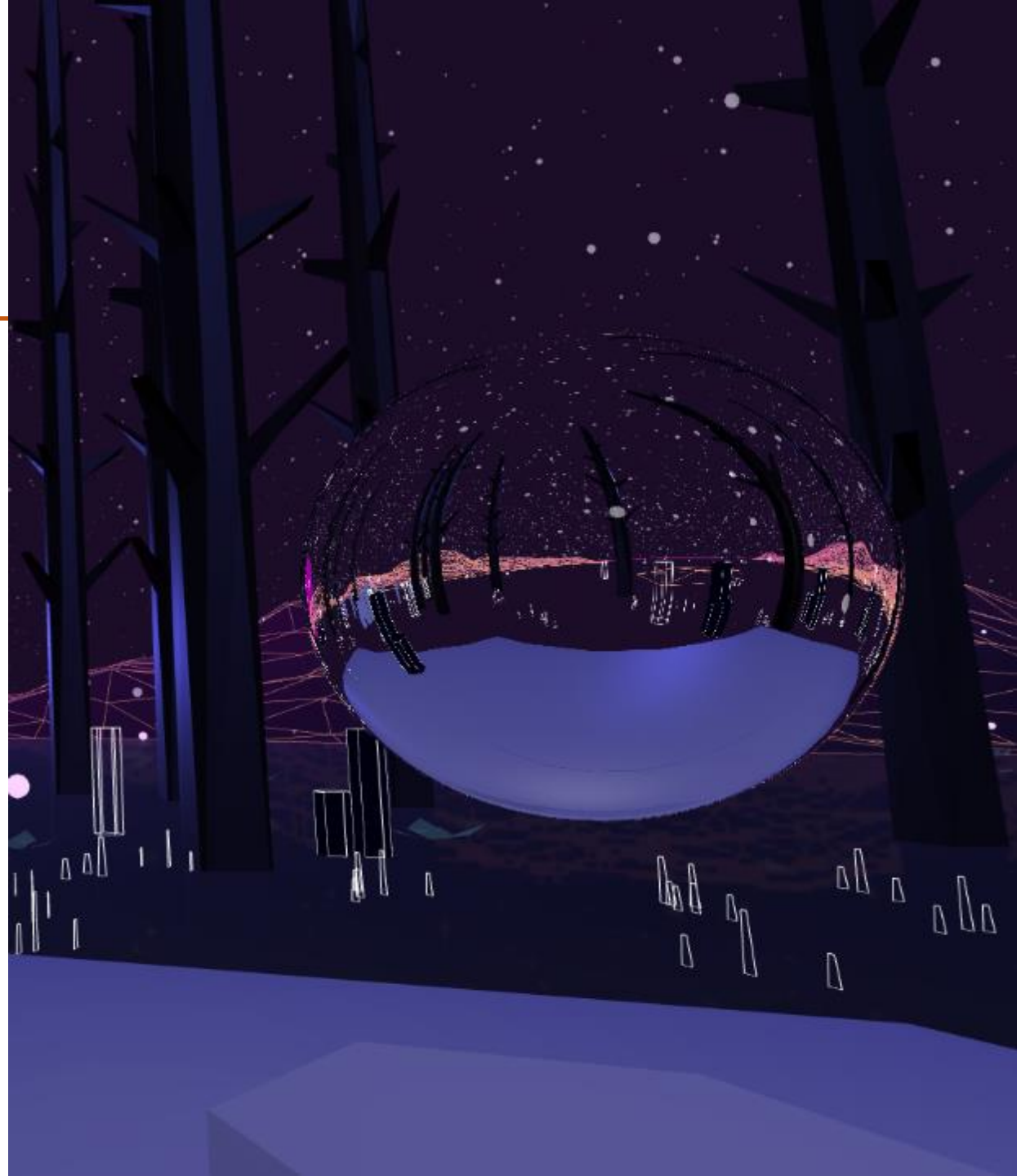
- Notice that the rings are reflected in the inner sphere in real time



Dynamic Reflections

- And another example:

http://pierfrancescosoffritti.github.io/doodles/13_Nowherelands/index.html



Dynamic Reflections

- These examples are on spheres, but you can apply the same technique to every other geometry.
- Real-time reflections can really improve the aesthetic of your work, even though they are computationally heavy.

Dynamic Reflections - CubeCamera

- It is easy to add this type of reflections to your scene, just use the **CubeCamera** object from Three.js.
- As the description says, this objects creates 6 cameras that render on a render target cube texture.
- Then you can use as a cube map (like we did with the environment map, but now with real-time view) on your objects in order to add reflections.

Dynamic Reflections - CubeRenderTarger

- First, create a new **CubeRenderTarget**

```
const cubeRenderTarget = new THREE.WebGLCubeRenderTarget( 512, {
      format: THREE.RGBFormat,
      generateMipmaps: true,
      minFilter: THREE.LinearMipmapLinearFilter
} );
```

- The 512 number basically defines the resolution of the image. The higher the resolution the heavier the rendering will be, so don’t abuse it. 512 should work for most cases.

Dynamic Reflections - CubeCamera

- And now we can create our **CubeCamera** and add it to the scene.

```
var near = 30;
var far = 1000;
const cubeCamera = new THREE.CubeCamera(near, far,
cubeRenderTarget);
scene.add(cubeCamera);
```

- Near and far are used to define how close and how far the camera is able to see. (Objects with a distance from the camera smaller or greater than near and far respectively won't be rendered).

Dynamic Reflections - CubeCamera

- Remember to update the CubeCamera at each frame.

```
cubeCamera.update(renderer, scene);
```

- At this point you only need to take the texture and apply it to your material as an environment map.

```
material.envMap = cubeRenderTarget.texture;
```

Dynamic Reflections - CubeCamera

- Obviously, you can play with the material's properties in order to change the aesthetic of the reflections. If you're feeling brave you can even do some post processing on it :)
- You should also set the position of your **CubeCamera**, based on what do you want to see reflected on your object.
- As the previous technique with a static texture, this one is straightforward to implement, yet graphically extremely effective.

Dynamic Reflections - CubeCamera

- In some situations, you will need to hide your object first when doing the CubeCamera rendering, e.g.

```
// Update the render target cube
car.visible = false;
cubeCamera.position.copy( car.position );
cubeCamera.update( renderer, scene );

// Render the scene
car.visible = true;
renderer.render( scene, camera );
```

Physics – Ammo.js

There are several Physics engines that can be used in Three.js

One very popular is **Ammo.js** and you can see a series of 3 tutorials with the basics starting here:

https://medium.com/@bluemagnificent/intro-to-javascript-3d-physics-using-ammo-js-and-three-js-dd48df81f591

And you should go there for the details…

the next slides are just a *very brief introduction*.

Physics – Rigid body & Collision shape

- A **Rigid Body** in Ammo.js is what moves, collides, has mass and can have forces (including gravity) and impulse applied to it, and we can set its velocity if we want to
- The **Collision Shape** is the “envelope” of the object that will interact with other objects, and it is used to calculate the inertia for example
- Some of the collision shapes supported in Ammo.js are primitive shapes (e.g box, sphere, cylinder, capsule, cone and multisphere) , compound shapes, triangle mesh, convex hull, static plane and others.
- A Rigid Body can also be static (unmovable)

Physics – Ammo.js / Three.js

The basic idea is that for each object that you create in three.js you will create a similar object in Ammo.js with the adequate characteristics, e.g.

- Three.js

```
let ball = new THREE.Mesh(new THREE.SphereBufferGeometry(radius),
material);
```

- Ammo.js

```
let colShape = new Ammo.btSphereShape( radius );
```

- And connect the two with three.js userData (body is from Ammo.js)

```
ball.userData.physicsBody = body;
```

Physics – Update Objects

- Then you can use the physics simulation

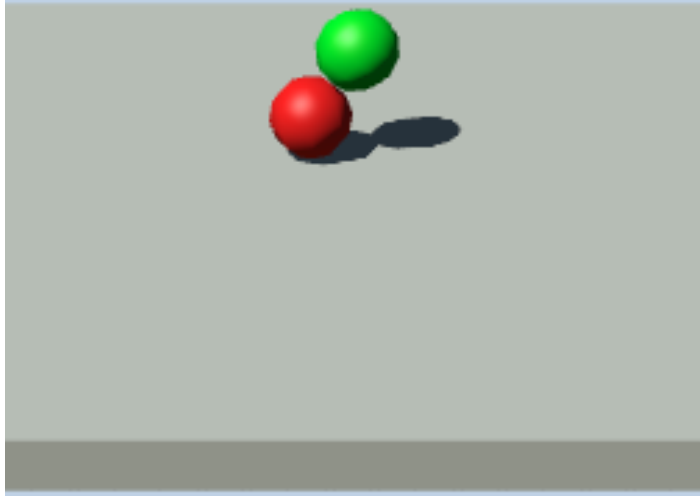
```
physicsWorld.stepSimulation( deltaTime, 10 );
```

- And update each of the Three.js bodies

```
let objAmmo = objThree.userData.physicsBody;
let ms = objAmmo.getMotionState();
if ( ms ) {
      ms.getWorldTransform( tmpTrans );
      let p = tmpTrans.getOrigin();
      let q = tmpTrans.getRotation();
      objThree.position.set( p.x(), p.y(), p.z() );
      objThree.quaternion.set( q.x(), q.y(), q.z(), q.w() );
}
```

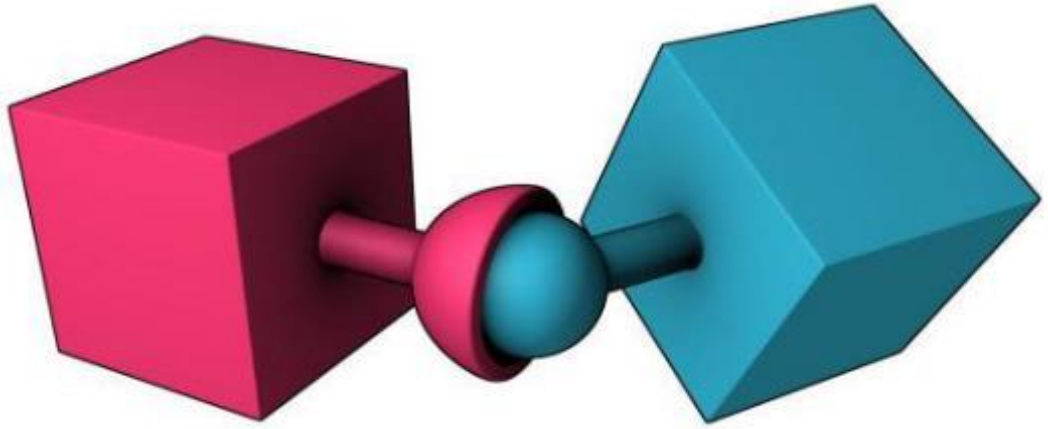
Physics - Collisions

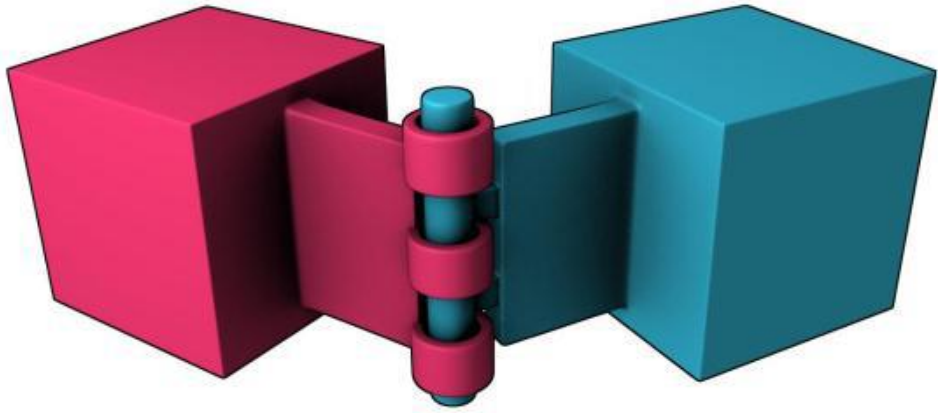
- With Ammo.js you can also set what objects will collide with other objects, e.g.
  - The green ball will collide with the red ball, but will fall through the gray block
- There are also methods to get information on these collisions, and this can be used in several situations like games (think grabbing a bonus object, detecting if something bad was touched, etc…)

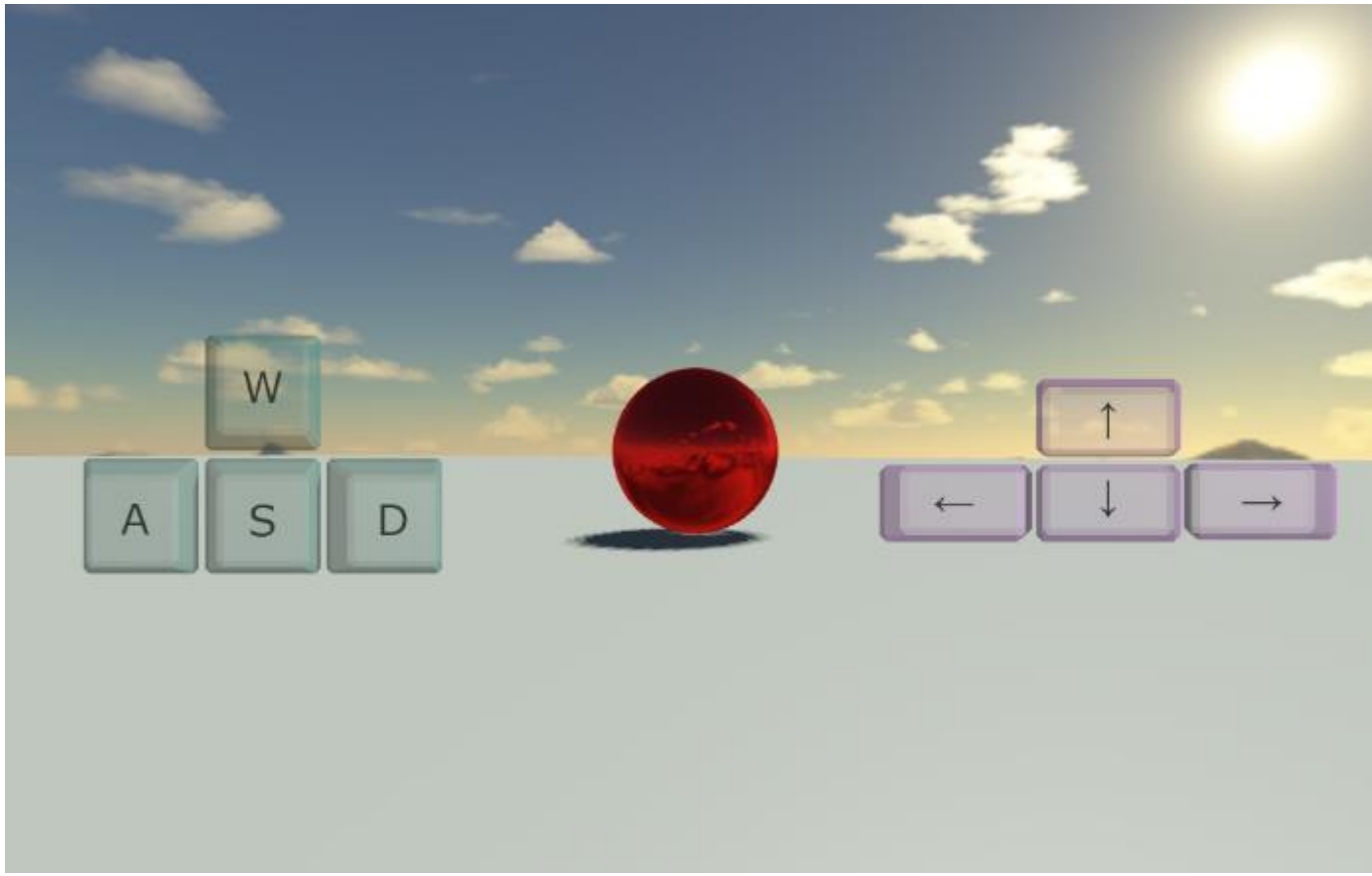


Physics - Constrains

- You can also have joints; a constraint component that connects two rigid bodies together or connects a rigid body to a static point in the world





Physics - Kinematic Rigid Bodies

- You can also have the "Kinematic" Rigid Bodies that affect non-kinematic objects but are not affected by other objects.
- This "one-way" interface is very useful to implement **user interaction** with dynamic objects