P.PORTO

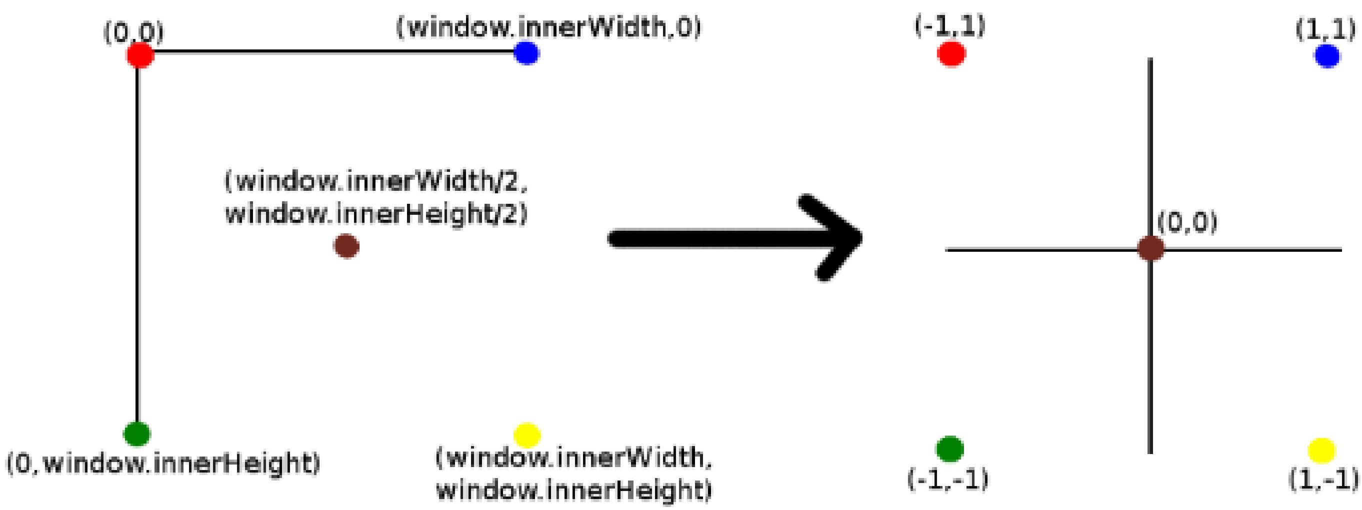
POLITÉCNICO
DO PORTO
ESMAD

COMPUTAÇÃO GRÁFICA
TSIW

Three.js - Picking

- **Picking**: object selection by user interaction
- How it works? Normally the selection is performed using the **mouse cursor**, so first it is necessary to normalize the cursor coordinates (from window coordinates to [-1,1] coordinates)

```
let mouse = new THREE.Vector2();
mouse.x =   ( event.clientX / window.innerWidth ) * 2 – 1;
mouse.y = - ( event.clientY / window.innerHeight ) * 2 + 1);
```



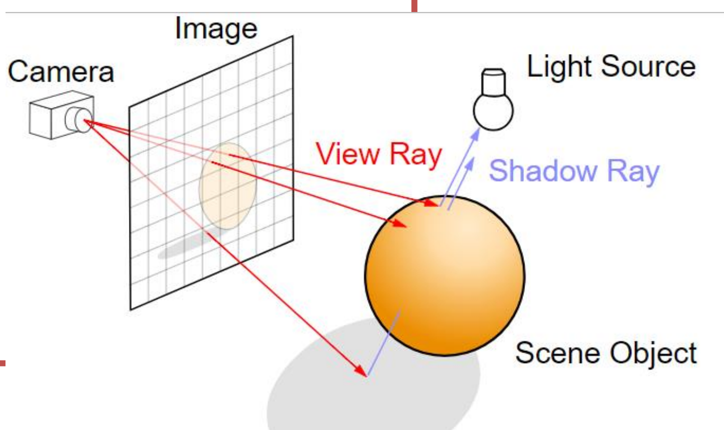
Three.js - Picking

- **RayCaster**: Three.js class that determines intersections between a ray and the objects present in the scene
  - For picking purposes using the mouse, the ray must begin in the **center of the camera**, and its direction is set from the **mouse cursor normalized coordinates**
  - The collision determination is performed by **ray distance** (closest ones first)
  - See examples in [Three.js RayCaster documentation]()

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

// update the picking ray with the camera and mouse position
raycaster.setFromCamera( mouse, camera );

// calculate objects intersecting the picking ray
let intersects = raycaster.intersectObjects(
                            scene.children );
if (intersects.length > 0) {
         … //do something
}
```



Three.js - Picking

- **RayCaster**: object intersection methods

  `[]=intersectObject(`*`object`*`,`*`recursive`*`)`: checks intersection between the ray and the object

  `[]=intersectObjects(`*`[objects]`*`,`*`recursive`*`)`: checks all intersections between the ray and the provided objects' array

  – `recursive`: if true, it also checks intersections with all the object(s) descendants (default: false)

  – `[]`: intersections are returned sorted by distance, closest first; this array contain the following information for each item:

    `distance`: distance between the origin of the ray and the intersection
    `point`: point of intersection, in world coordinates
    `face`: intersected face
    `faceIndex`: index of the intersected face
    `object`: the intersected object
    …

Three.js - Dragging

- **Dragging**: Drag-and-drop functionality that occurs after an object has been selected (usually by the mouse pointer)
  - An **auxiliary plane** is used to help determine the new object position
  - This plane could be invisible
  - The plane defines the orientation to where the object can be repositioned

```
// invisible helper plane (big enough)
// for example, aligned with the XY-plane (Z=0)
plane = new THREE.Mesh(
                   new THREE.PlaneGeometry(1000, 1000, 10, 10),
                   new THREE.MeshBasicMaterial({
                            opacity: 0.0,
                            transparent: true,
                            visible: false
                   })
);
scene.add(plane);
```

Three.js - Dragging

- **Dragging**:
  - When the **mouse selects an object**, the offset between the object and the intersection point of the `Raycaster` is calculated with the plane:

    OFFSET = Plane intersection point – Object center
  - Selected object must also be saved

```
function onMouseDown() {
   (…)
   // check if ray intersects any of the objects' array
   let intersects = raycaster.intersectObjects(objects);
   // if any…
   if (intersects.length > 0) {
         // gets closest intersected object (must be a global variable)
         selectedObject = intersects[0].object;
         // gets ray intersection with the helper plane
         let intersectsPlane = raycaster.intersectObject(plane);
         // calculates the offset (also a global variable):
         //       plane ray intersection – intersected object center
         offset.copy(intersectsPlane[0].point).sub(selectedObject.position);
   }
}
```

Three.js - Dragging

- **Dragging**:
  - When the **mouse moves across the screen**, and if an object is selected, it must be positioned according to the determined offset and the new point of intersection with the auxiliary plane:

    New object position = New plane intersection point – OFFSET

  - You may (or may not) need to reposition the helper plane

```
function onMouseMove() {
   (…)
   if (selectedObject) {
         // gets (again) the new ray intersection with the helper plane
         let intersects = raycaster.intersectObject(plane);
         // drag the intersect object around
         selectedObject.position.copy(intersects[0].point.sub(offset));
   }
}
```

Three.js - Dragging

- **Dragging**:
  - When user releases the mouse, indicate that there are no more objects to move

```
function onMouseUp(event) {
	// finish drag & drop
	selectedObject = null;
}
```

Three.js - Dragging

- **Dragging**:
  - **Sometimes** you may want to reposition the helper plane when the camera moves, so that the plane is ALWAYS camera oriented

```
function onMouseMove() {
   (…)
   if (selectedObject) {
         //drag an object around if we've already clicked on one
         let intersects = raycaster.intersectObject(plane);
         selectedObject.position.copy(intersects[0].point.sub(offset));
   }
   //OPTIONAL: reposition the plane to the center of the object to be dragged
   //on the next call of the mouse move event
   else {
         let intersects = raycaster.intersectObjects(objects);
         if (intersects.length > 0)
                  plane.position.copy(intersects[0].object.position);
                  // OPTIONAL helper plane reorientation
                  plane.lookAt(camera.position);
         }
}
```

Exercises

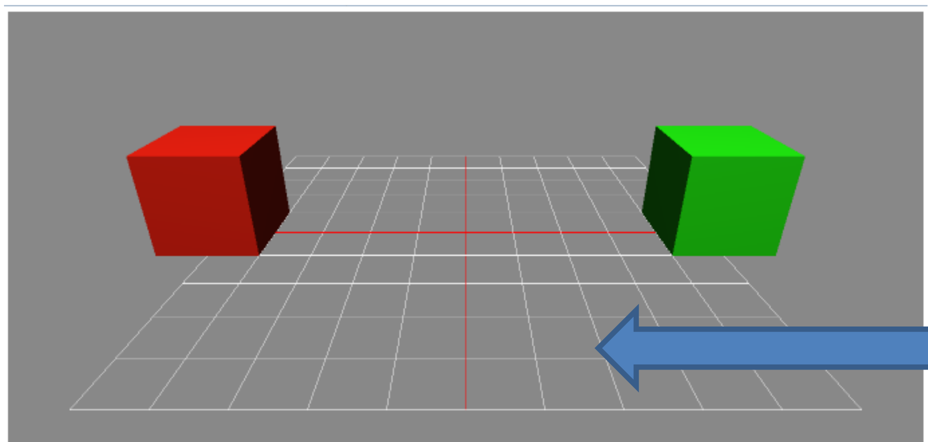
1) Create the following scene:

– One GridHelper (size = 100, divisions = 10) and two cubes (20x20x20)

– When you click on one of the squares, spin it at a constant speed of 0.1 radians per frame, stopping when it completes a full revolution

– Open the browser console and check the value of the array returned by the RayCaster intersection method

Hint: assign different names to the cubes and use them to verify which cube was intersected by the RayCaster



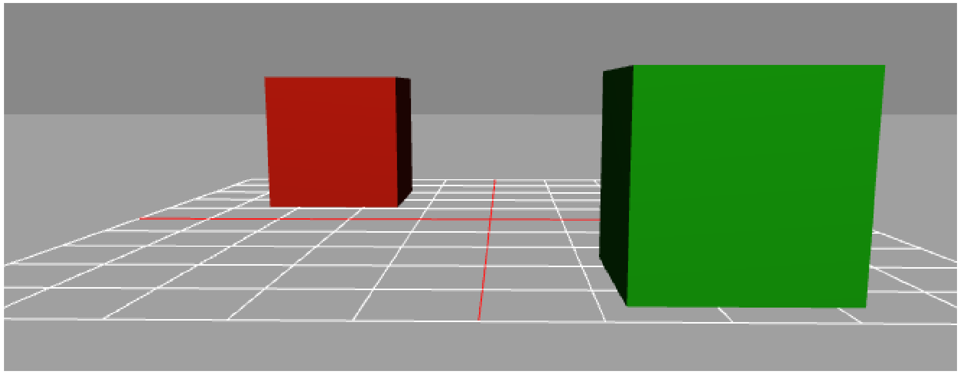
THREE.GridHelper

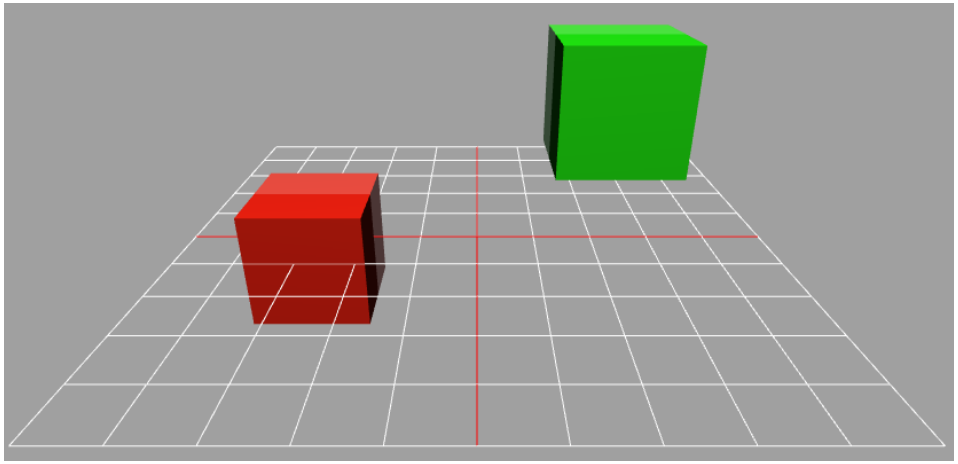
Exercises

2) Alter previous exercise, and implement a Drag-and-Drop movement:

a) Move the cubes horizontally (along the GridHelper)

b) Move the cubes vertically (on a plane that passes on the selected cube center), making sure the <u>plane is always facing the camera</u>

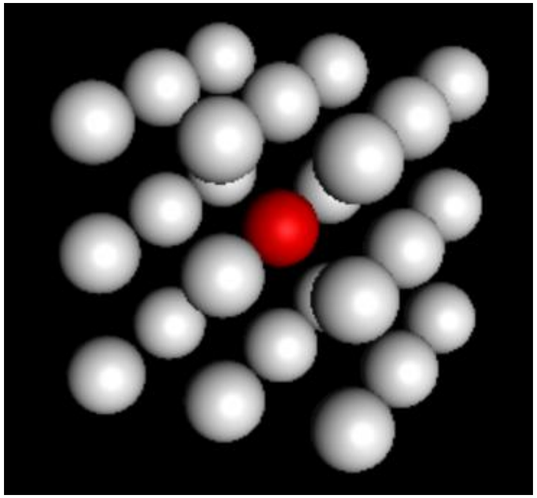


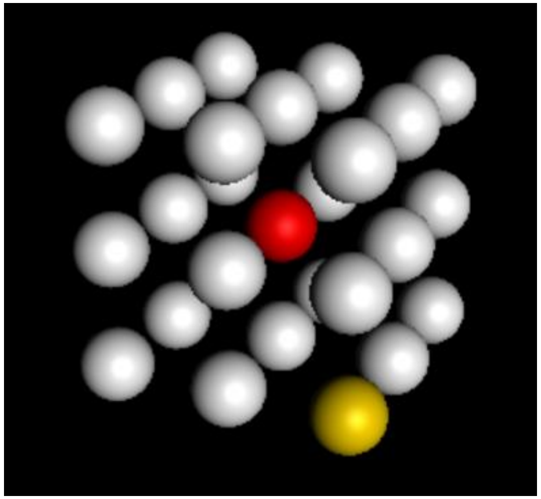


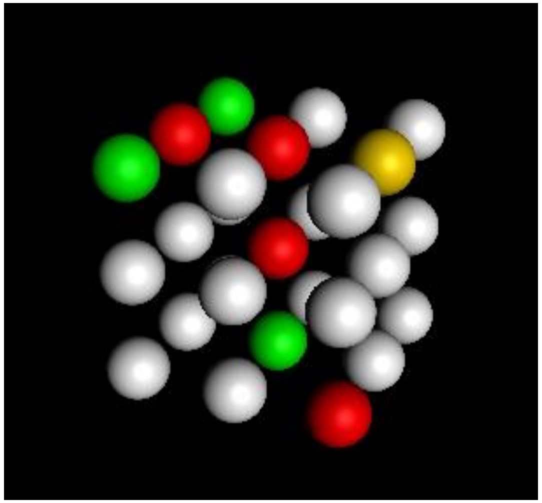
Exercises

3) TIC-TAC-TOE (3D):

- Build a structure with 27 (3x3x3) spheres
- When user passes the mouse pointer on a sphere, turn the sphere yellow
- When user clicks on a sphere, turn the sphere red or green, alternately



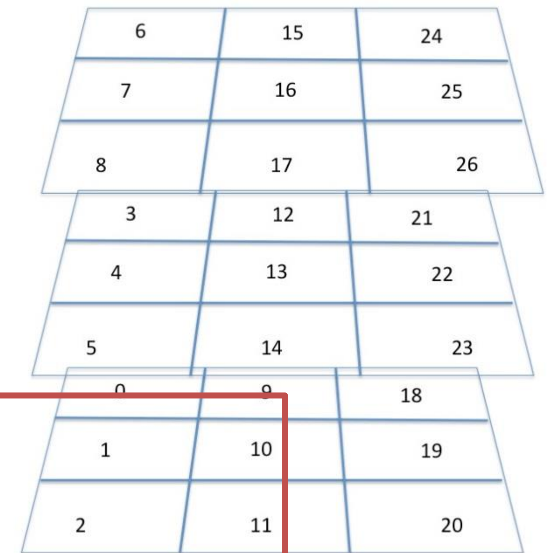




Exercises

3) TIC-TAC-TOE (3D):

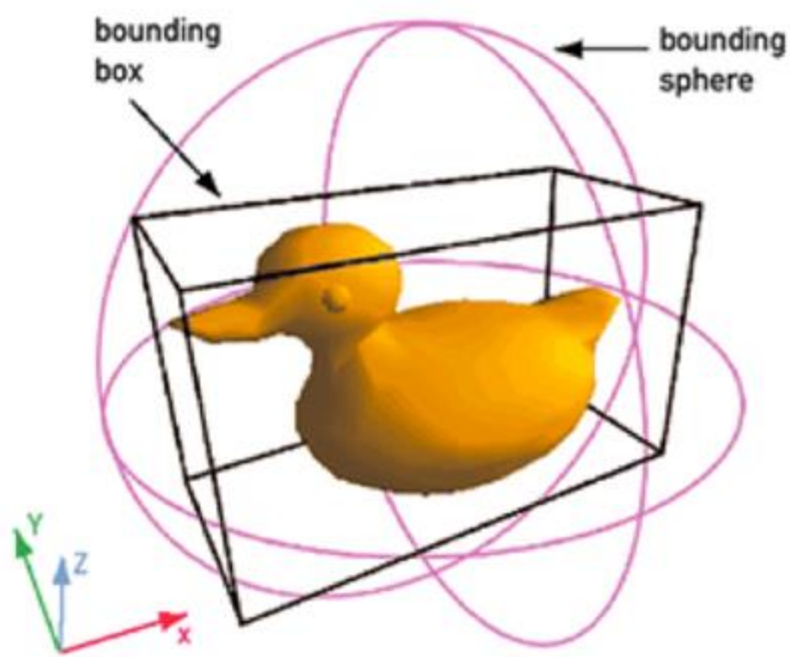
- If you number the spheres as shown in the figure, the winning plays are the following combinations:




```
// combinations of possible win sphere selections
const wins =
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11], [12, 13, 14],
[15, 16, 17], [18, 19, 20], [21, 22, 23], [24, 25, 26],
[6, 15, 24],  [7, 16, 25], [8, 17, 26], [3, 12, 21], [4, 13, 22],
[5, 14, 23], [0, 9, 18], [1, 10, 19], [2, 11, 20], [18, 21, 24],
[19, 22, 25], [20, 23, 26], [9, 12, 25], [10, 13, 16],
[11, 14, 17], [0, 3, 6], [1, 4, 7], [2, 5, 8], [6, 16, 26],
[8, 16, 24],  [3, 13, 23], [5, 13, 21], [0, 10, 20], [2, 10, 18],
[18, 22, 26], [20, 22, 24], [2, 14, 26], [8, 10, 20], [2, 4, 6],
[0, 4, 8], [0, 12, 24], [6, 12, 18], [2, 13, 24], [6, 13, 20],
[0, 13, 26], [8, 13, 18], [11, 13, 15], [9, 13, 17],
[1, 13, 25], [7, 13, 19] ];]
```

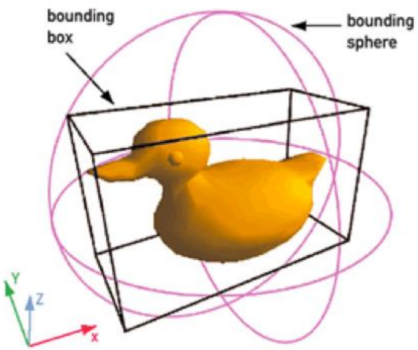
Three.js - Collisions

- Three.js does not provide a system for collision detection or colliders
- Two options:
  - implement collision detection with some math and coarse bounding volumes like THREE.Sphere or THREE.Box3
  - integrate a physics engine (e.g. Physi.js or Ammo.js)
- For most apps, a real physics engine is an overkill
  - Check the code for this example, were in the render function it is ensured the balls are kept inside the room and collide against each other using simple math logic



Three.js - Collisions

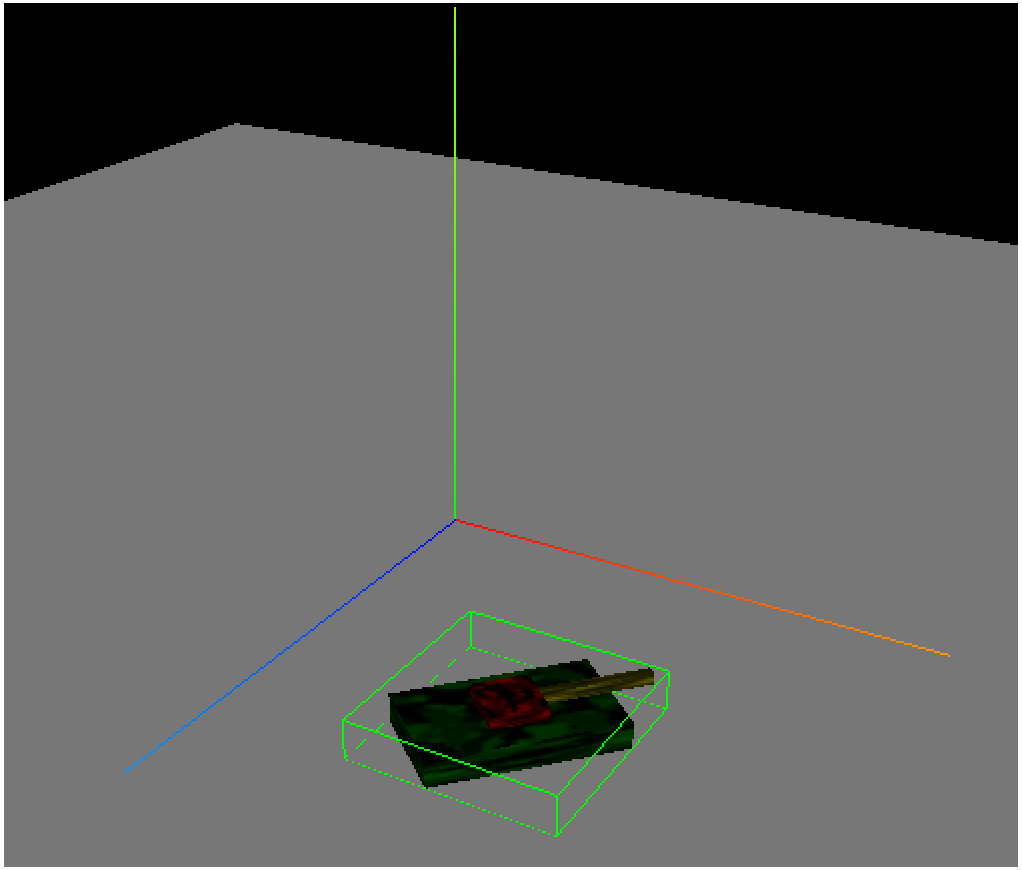
- Fastest way to detect collisions between objects - ***Bounding Volumes***
  - Most common volumes: boxes and spheres



- Other techniques: using the **raycasting** technique
  - https://stemkoski.github.io/Three.js/Collision-Detection.html
  - Sets a **ray** using the moving **object position** and a **direction** (in the example, determines a ray passing by each of the red cube vertices)
  - Checks if they intersect any mesh in the array of target meshes (for increased collision accuracy, one can add more vertices to the cube)
  - **HOWEVER**: when the origin of the ray is within the target mesh, collisions do not occur

Three.js - Collisions

- **Three.js bounding box**: the box definition surrounding an object is of type AABB - ***Axis-Aligned*** *Bounding Box*



Three.js - Collisions

P.PORTO ESCOLA SUPERIOR DE MEDIA ARTES E DESIGN

- Three.js:

  1. **Build the geometry *Bounding Volume*:**

`mesh.geometry.computeBoundingBox()` → computes a THREE.Box3, the object's Minimum Axis-Aligned Bounding Box
`mesh.geometry.computeBoundingSphere()` → computes a THREE.Sphere, the object's Minimum Bounding Sphere

BE AWARE: those methods take as reference the geometry (not the mesh and its transformations), so they **ignore** any transformation that is applied to the mesh!

!

So, you must use:

```
let BBox = new THREE.Box3().setFromObject(mesh);
```

(the Bounding Sphere doesn't have the same method)

```
let BSphere = new THREE.Sphere().setFromPoints(mesh.vertices);
Bsphere.applyMatrix4(mesh.matrixWorld);
```

Three.js - Collisions

- Three.js:

  2. Intersection methods - return **true** or **false** whether or not the given volume intersects another given geometry:
     `boundingVolume.containsPoint(point)`
     `boundingVolume.intersectsBox(box)`
     `boundingVolume.intersectsSphere(sphere)`
     `boundingVolume.intersectsPlane(plane)`

  3. If an object is in motion, it is necessary to determine the updated bounding volume, detect collisions (intersections) and only change the position if there is no intersection

     ```
     boundingVolume.applyMatrix4(mesh.matrixWorld);
     ```

Three.js - Collisions

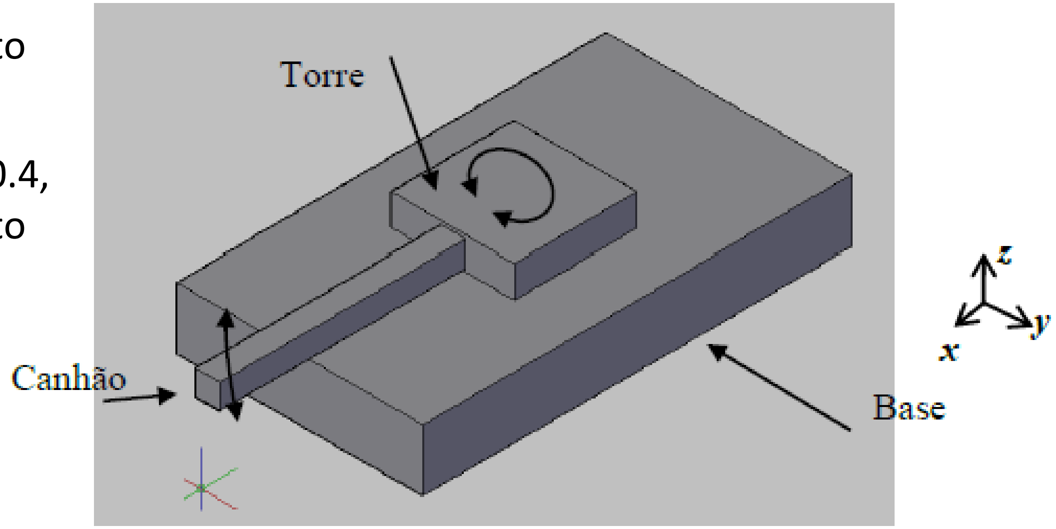
- Three.js:

```
// object definition
…
// helper object to show the world-axis-aligned bounding box around an object
mesh.geometry.computeBoundingBox();
bbHelper = new THREE.BoxHelper(mesh, 0x00FFFF);
scene.add(bbHelper); // adds AABB to the scene
…
// animation function
…
bbHelper.update(); // updates helper object
…
let BBox = new THREE.Box3().setFromObject(mesh);
let BBox2 = new THREE.Box3().setFromObject(othermesh);
let collision = BBox.intersectsBox(BBox2); // checks collision between mesh and othermesh
…
```

! Not mandatory, only for debugging purposes

Cameras - exercises

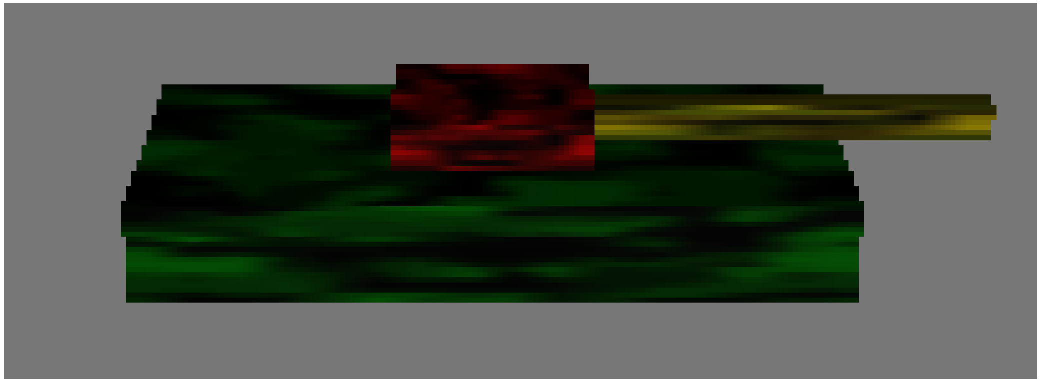
- *TANK*
  - **Camera**: perspective camera: fov = 45º, position = (0, 50, 70)
  - **DirectionalLight** : color = white, intensity = 1.5, position = (0.3, 0.6, 1)
  - **Plane**: horizontal, size = 100x100, color = 0x777777
  - **Tank**: 3 objects of type *BoxGeometry*
    - **Base**: length = 7, width = 4, height = 1 / color = 0x008800 / center in (0,0,0)
    - **Tower**: length = 2, width = 2, height = 0.5 / use the image to position it / color = 0xff0000
    - **Cannon**: length = 4, width = 0.4, height = 0.4 / use the image to position it / color = 0xffff00



Cameras - exercises

P.PORTO ESCOLA SUPERIOR DE MEDIA ARTES E DESIGN

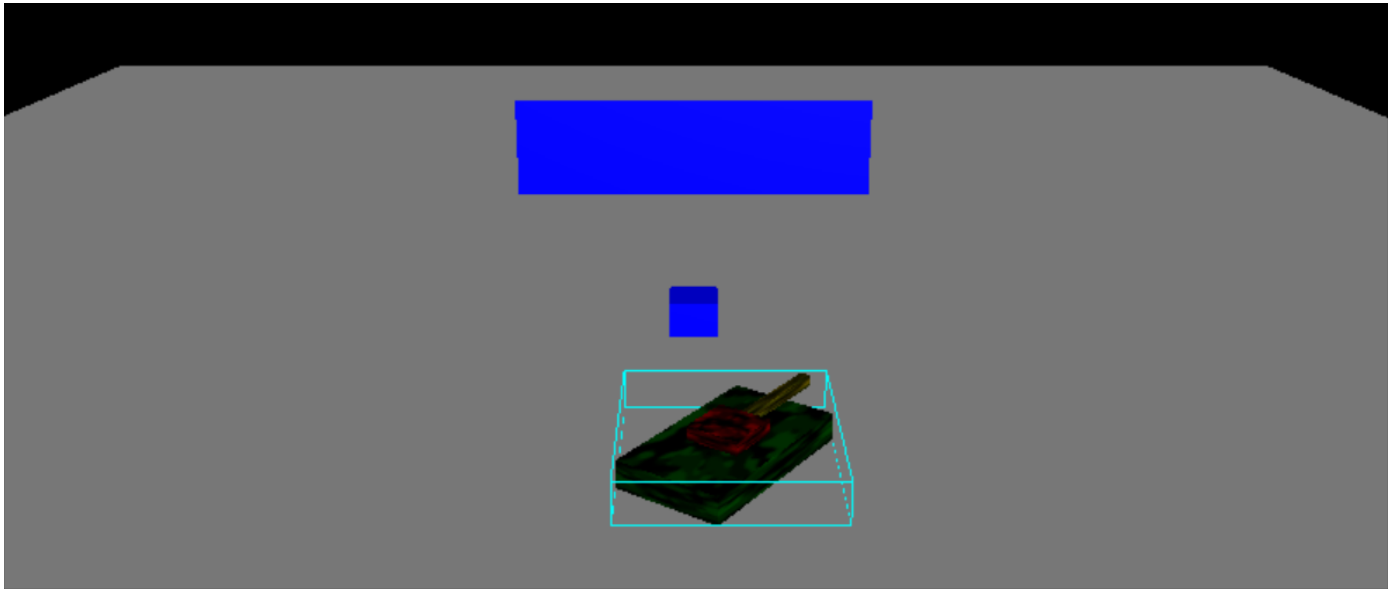
- *TANK*



  - Add texture using image *camouflage.png*
  - Add movement:
    Keys W and S: tank moving forward or backward, respectively (constant velocity of 0.1 units per frame)
    Keys D and A: tank moving right or left (increase or decrease base rotation by 0.01 radians per frame)
    Keys Z and X: rotate tower / Keys V and B: rotate cannon
    Key C: shift between static camera and “third person view” (offset to the base center: Y=5, Z=-15)

Exercise - Collisions

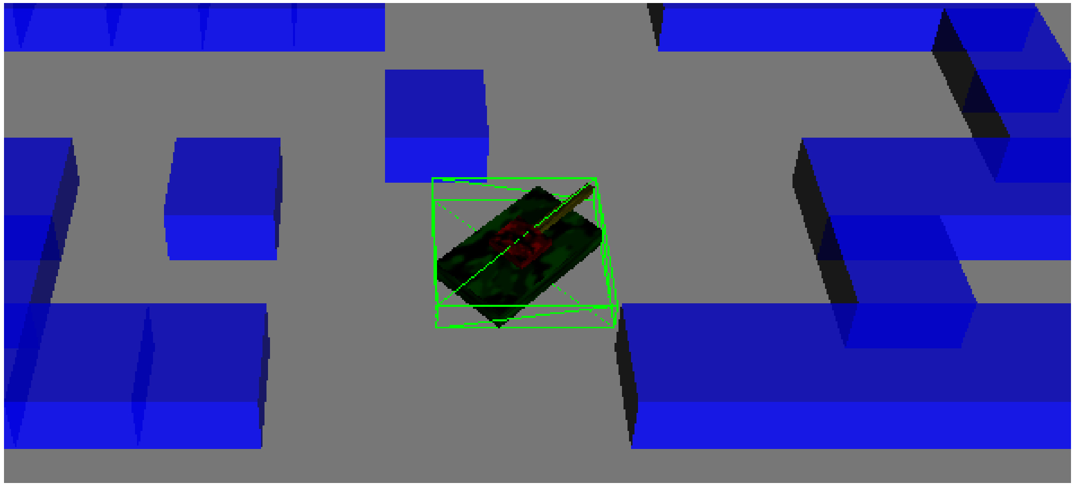
- Add some obstacles (rectangles) to the scene
- Compute the tank bounding volume and visualize it using a BoxHelper



- Move the tank only if there is no collision between it and any of the obstacles

Exercise - Collisions

- You may find that the *Bounding Box* may not be the ideal solution for all cases!



  - **HINT**: determine the collision using the tank **vertices** and use the `RayCaster` to set a ray from the tanks' center and all its child meshes vertices to determine if the ray intersects any face of the labyrinth cubes
    ! for increased collision accuracy, add more vertices to the mesh geometry

Exercise - Collisions

- Collisions using RayCasting

```
function checkCollisions() { // COLLISION BETWEEN ONE MESH AND A SET OF OBSTACLES (COLLIDABLE MESHES)

      const ray = new THREE.Raycaster();
      let originPoint = new THREE.Vector3;
      mesh.getWorldPosition(originPoint);

      for (let i = 0; i < mesh.geometry.attributes.position.count; i++) {
          // get X, Y and X values for the vertex coordinates at the given index i
          let x = mesh.geometry.attributes.position.getX(i);
          let y = mesh.geometry.attributes.position.getY(i);
          let z = mesh.geometry.attributes.position.getZ(i);
          let localVertex = new THREE.Vector3(x, y, z);
          // convert from local to world coordinates
          let globalVertex = localVertex.applyMatrix4(mesh.matrixWorld);

          // calculate direction from mesh center to vertex
          let directionVector = globalVertex.sub(originPoint);

          // updates the ray with a new origin and direction
          ray.set(originPoint, directionVector.clone().normalize());
          let collisionResults = ray.intersectObjects(collidableMeshes);
          if (collisionResults.length > 0 && collisionResults[0].distance < directionVector.length())
                return true;

          }
      return false;
}
```

Means that a collidable mesh is INSIDE the mesh