Coding with 3.js

UWVR Winter 2021

what is 3.js?

WebGL – Rasterization API; client side; lines, points, and triangles

some advantages of 3.js

- Abstracts away difficulties of WebGL and easy to learn
- No need for plug-ins!
- Large community
- Good documentation and examples
- Integrates well into existing web code
- Great for interactive and innovative websites
- Support for most of the standard industry file formats - easy to author assets in your favorite modelling software and import them for use them in three.js.

and more...

some disadvantages of 3.js

- It's not a game engine - lacks features like particle physics, scripting, asset management...
- Little support for deferred rendering
- Has an editor but it's not too comprehensive (https://threejs.org/editor/)

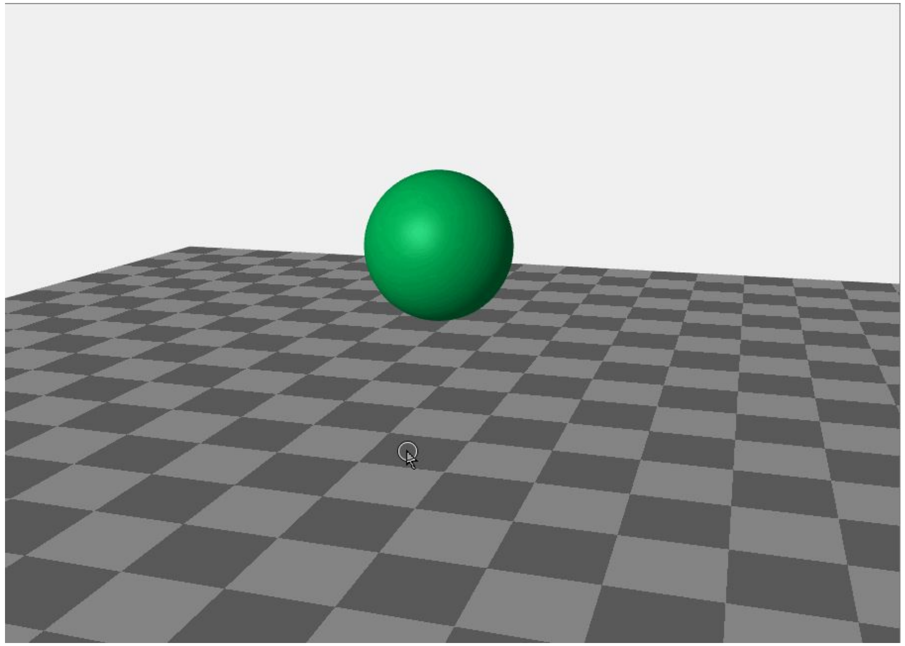
and more...

what will we be doing today?

concepts

end goal

- ❏ Scenegraph
- ❏ Renderer
- ❏ Camera
- ❏ Scene
- ❏ Lights: Directional and Ambient
- ❏ Geometry: Sphere
- ❏ Basic Material and Phong Material
- ❏ Mesh
- ❏ Texture
- ❏ Animation loop
- ❏ Orbit controls

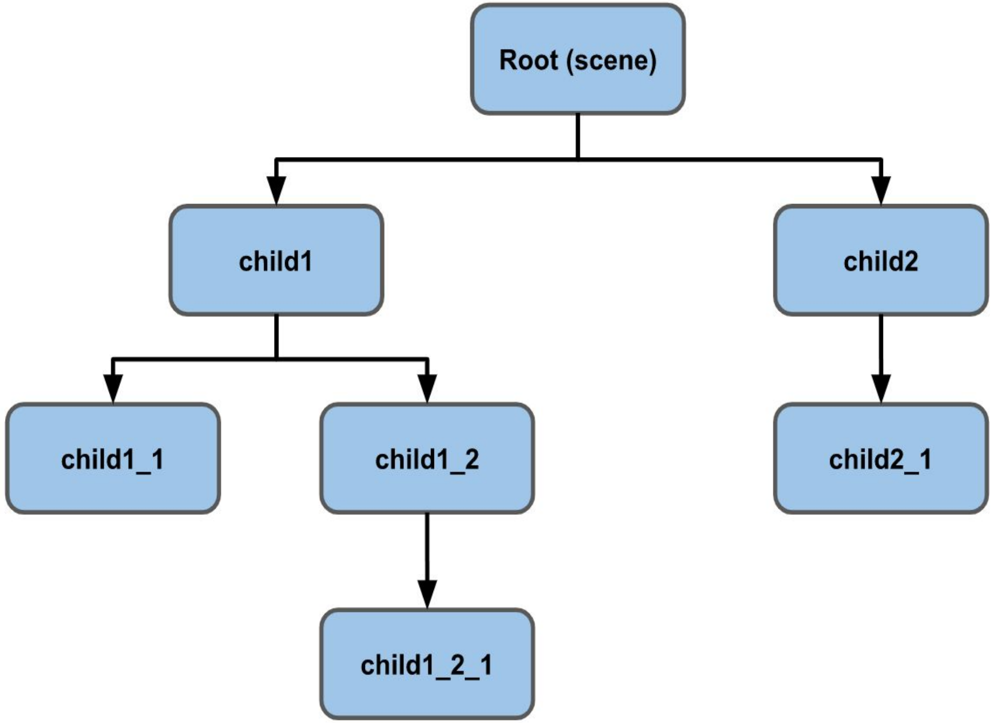


the Scenegraph

The scenegraph is a hierarchical parent/child tree like structure. Each node in the scenegraph represents an object, its location, and orientation in the local space.

Children are positioned and oriented relative to their parent.

The root of a scenegraph is the Scene.



Before we start...

We need to import the three.js module!

```
import * as THREE from "https://cdn.jsdelivr.net/npm/three@v0.108.0/build/three.module.js";
```

We are importing the module or library through a Content Delivery Network (CDN). A CDN is a highly-distributed platform of servers that helps minimize delays in loading web page content by reducing the physical distance between the server and the user.

the Renderer

We need to create a WebGLRenderer as ultimately it is what is responsible for rendering all the data and created objects onto the canvas.

The antialias option is false by default. Antialiasing makes your graphic and textures smoother at the expense of computing power.

Include this statement at the end of your program:

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

```
const canvas = document.querySelector('#b');

const renderer = new THREE.WebGLRenderer({canvas,
antialias: true});

renderer.setPixelRatio(window.devicePixelRatio);
```

the Camera

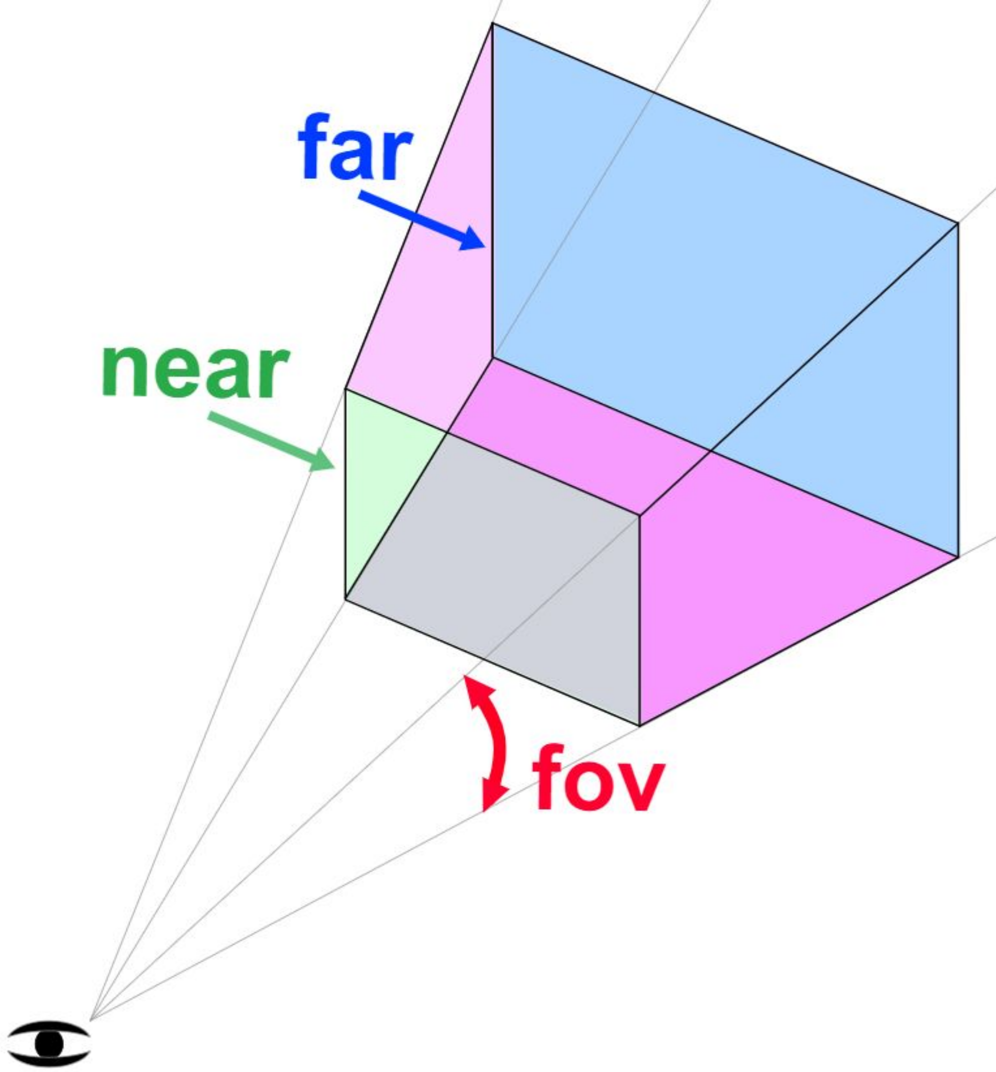
- Unlike the other objects, a Camera does not have to be in the scenegraph.

- A camera has four important attributes that define its frustum:
  - Field of view (in degrees, not radians)
  - Aspect
  - Near
  - Far

- The camera defaults to looking down the -Z.

- Only stuff inside the defined frustum will be rendered.

- Recall that the aspect ratio of an image is the ratio of its width to its height.



the Camera

The dimensions of the canvas we are drawing on is 720px by 540px.

The aspect ratio is then set to 720/540 ~ 1.4

NOTE also that:

The height of the near and far planes are determined by the field of view.

The width of both planes is determined by the field of view and the aspect.

View an example frustum at
https://www.geogebra.org/m/yrgvmktn

```
const fov = 45;
const aspect = 1.4;
const near = 0.1;
const far = 100;
const camera = new THREE.PerspectiveCamera(fov,
aspect, near, far);
camera.position.z = 5;
camera.position.y = 2;
camera.position.x = 0;
```

the Scene

As mentioned, the scene is the root of the scenegraph. Anything you want to draw must be added to the scene.

```
const scene = new THREE.Scene();
// change background color to gray
scene.background = new THREE.Color(0xf0f0f0);
```

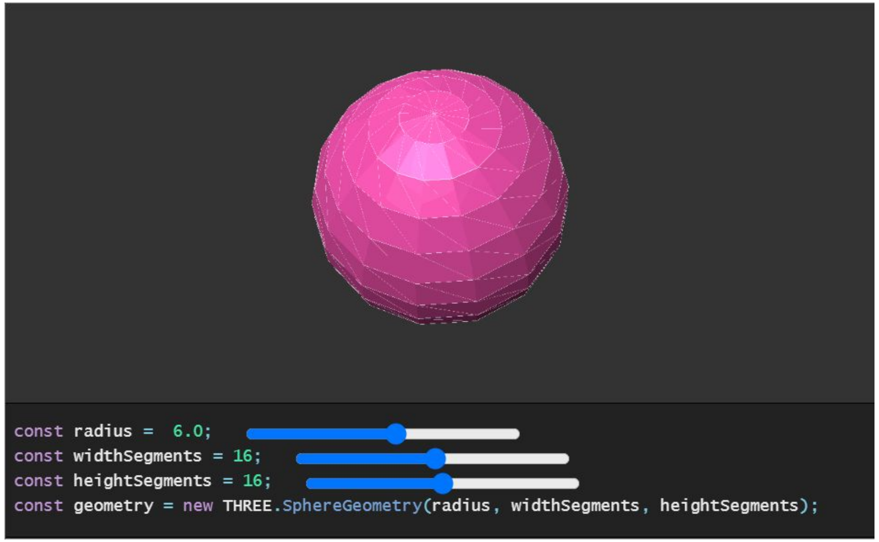
scenegraph

WebGLRenderer **PerspectiveCamera**

Scene

Sphere Geometry

- Geometry specifies the shape of the object.
- SphereBufferGeomtery is a primitive geometry type.
- What are the segments all about?



```
const radius =  6.0;
const widthSegments = 16;
const heightSegments = 16;
const geometry = new THREE.SphereGeometry(radius, widthSegments, heightSegments);
```

```
const radius = 1;
const widthSegments = 32;
const heightSegments = 32;
const ballGeometry = new
THREE.SphereBufferGeometry(radius,
widthSegments, heightSegments);
```

Basic material

Material objects represent the surface properties used to draw geometry including things like the color to use and how shiny it is.

How do we “bind” this material to the sphere we created?

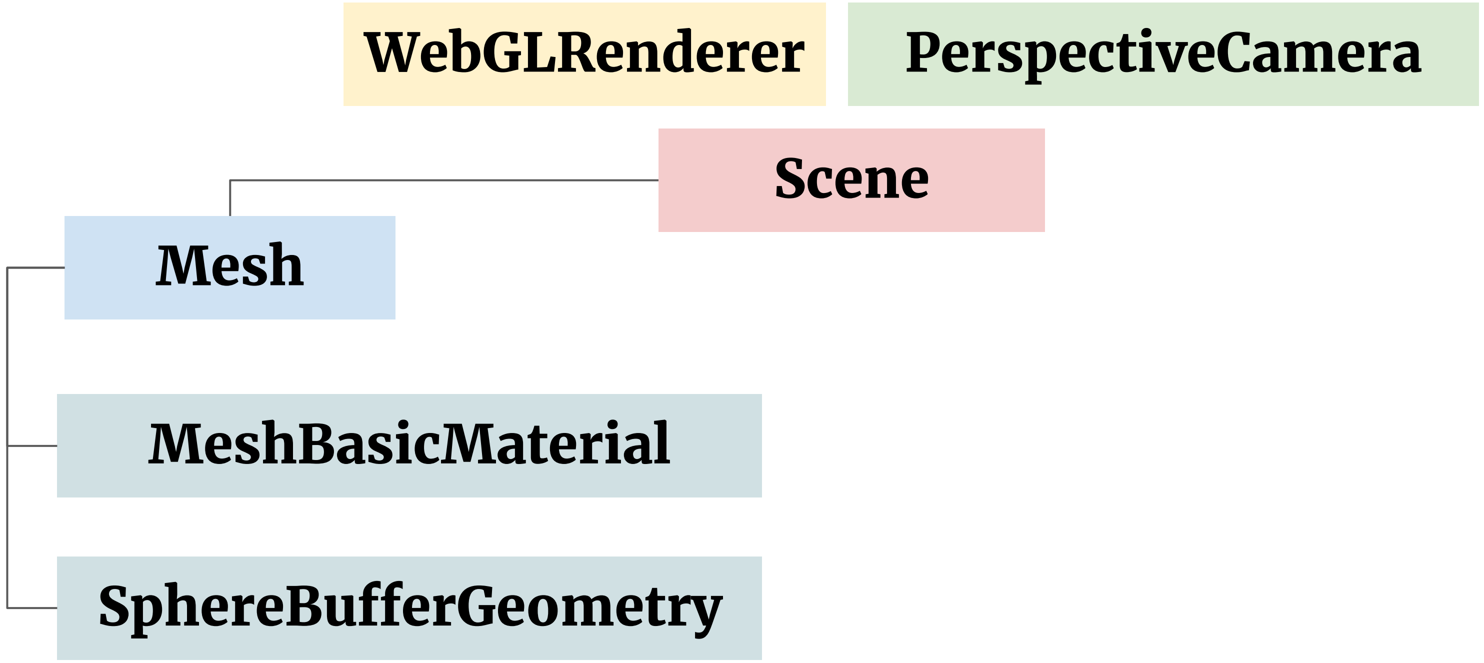
```
const ballMaterial = new
THREE.MeshBasicMaterial({ color: 0x44aa88 });
// greenish blue
```

Mesh

- Mesh objects represent drawing a specific geometry with a specific Material.
- Both Material objects and Geometry objects can be used by multiple Mesh objects.
- Both Mesh objects could reference the same Geometry object and the same Material object.

```
const ball = new THREE.Mesh(ballGeometry,
ballMaterial);
scene.add(ball);
```

scenegraph

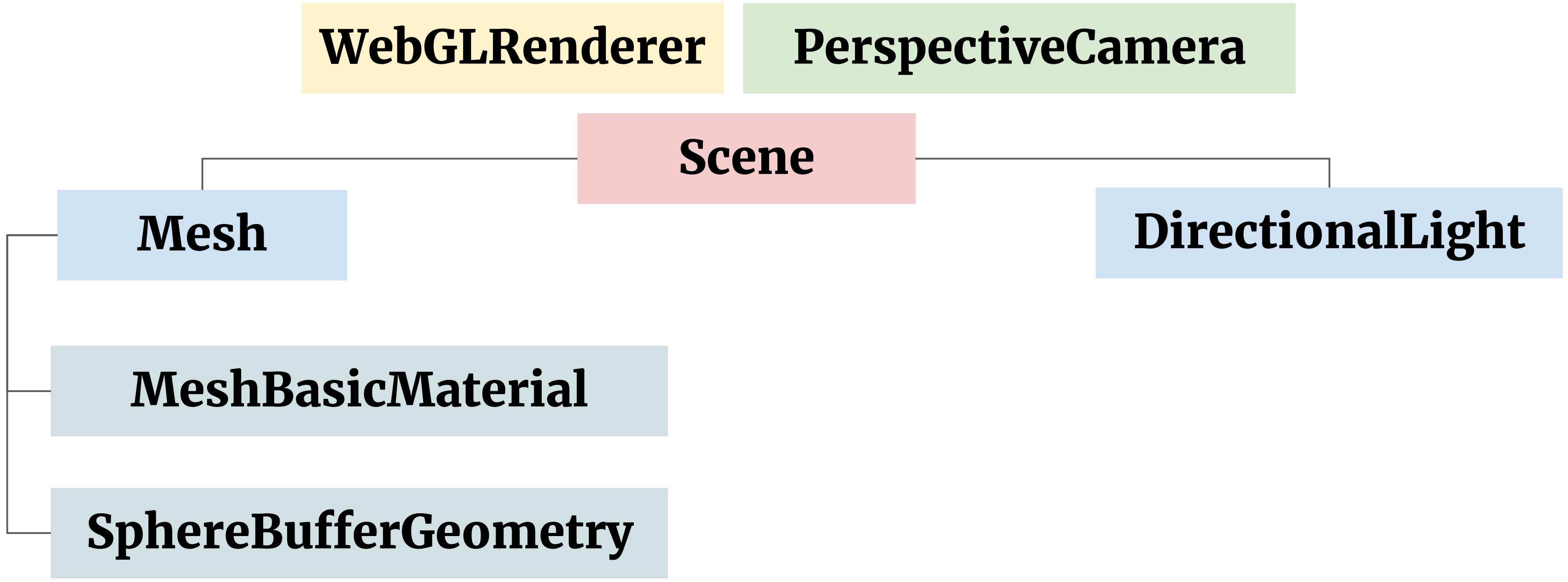

WebGLRenderer
PerspectiveCamera
Scene
Mesh
MeshBasicMaterial
SphereBufferGeometry

Directional light

There are many different types of lights in three.js. We will be using the DirectionalLight. Later, we will use an ambient light to brighten the entire scene.

```
{
    const color = 0xFFFFFF;
    const intensity = 0.7;
    const directionalLight = new
THREE.DirectionalLight(color, intensity);
    directionalLight.position.set(-1, 2, 4);
    scene.add(directionalLight);
}
```

scenegraph


WebGLRenderer
PerspectiveCamera
Scene
Mesh
DirectionalLight
MeshBasicMaterial
SphereBufferGeometry

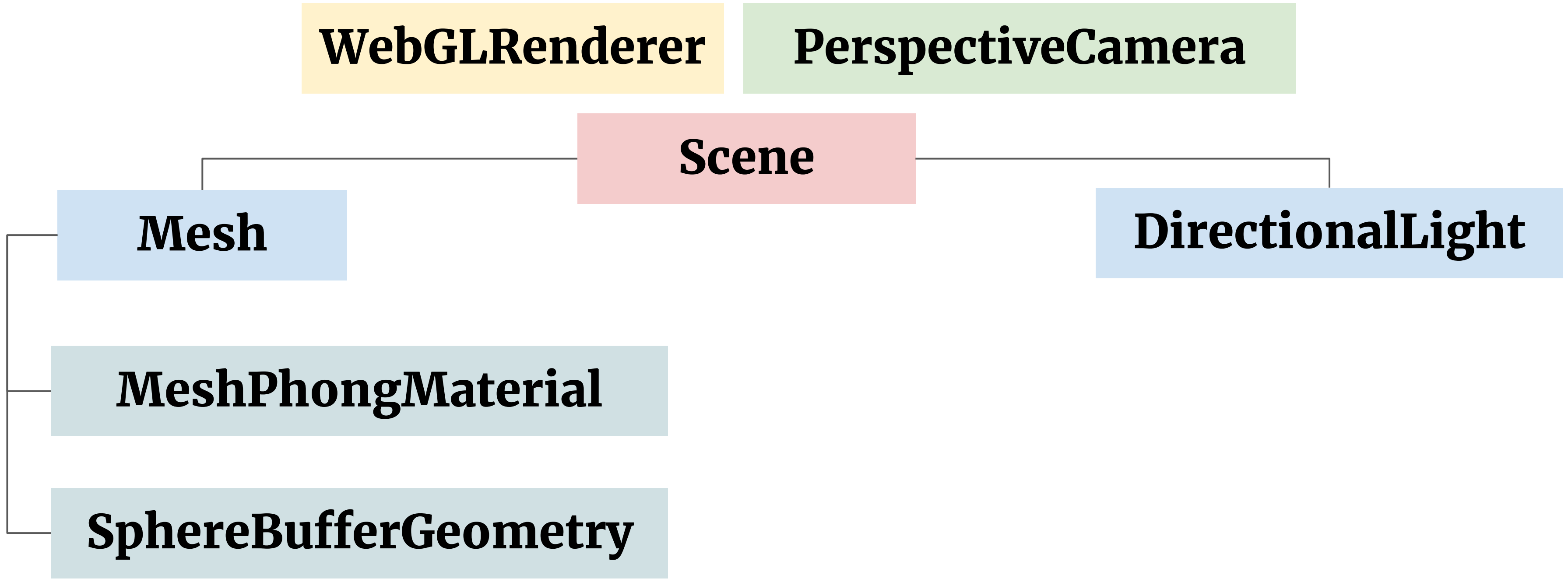
Phong material

The MeshBasicMaterial is not affected by lights. The MeshPhongMaterial computes lighting at every pixel, which allows reflections. The MeshPhongMaterial also supports specular highlights.

Replace MeshBasicMaterial with MeshPhongMaterial

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

scenegraph


WebGLRenderer
PerspectiveCamera
Scene
Mesh
DirectionalLight
MeshPhongMaterial
SphereBufferGeometry

Texture

- Texture objects generally represent images either loaded from image files, generated from a canvas or rendered from another scene.

- To use a texture, we need a texture loader.

Replace all of the MeshPhongMaterial code we just wrote with the following:

```
const loader = new THREE.TextureLoader();
const texture =
loader.load('assets/schcbgfp_2K_Albedo.jpg');
texture.repeat.set(1, 1);
const ballMaterial = new
THREE.MeshPhongMaterial({map: texture});
const ball = new THREE.Mesh(ballGeometry,
ballMaterial);
scene.add(ball);
```

the Plane

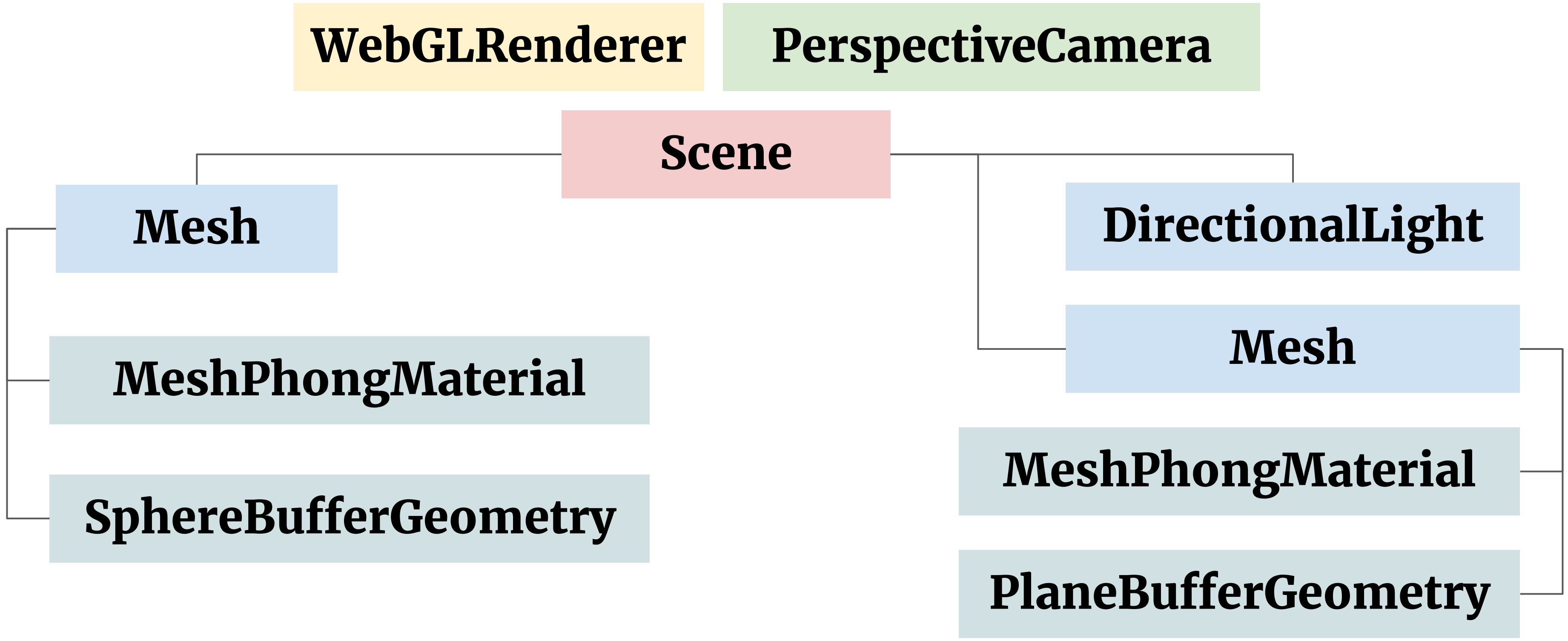
We will add a plane to our scene to give us a better sense of space.

You can read more about the extra details at https://threejsfundamentals.org/threejs/lessons/threejs-lights.html

```
const planeSize = 20;
const planeTexture =
loader.load('https://threejsfundamentals.org/threejs/resources/images/checker.png');

planeTexture.wrapS = THREE.RepeatWrapping;
planeTexture.wrapT = THREE.RepeatWrapping;
planeTexture.magFilter = THREE.NearestFilter;
const repeats = planeSize / 2;
planeTexture.repeat.set(repeats, repeats);
const planeGeo = new
THREE.PlaneBufferGeometry(planeSize, planeSize);
const planeMat = new THREE.MeshPhongMaterial({
        map: planeTexture,
        side: THREE.DoubleSide,
    });
const mesh = new THREE.Mesh(planeGeo,planeMat);
mesh.rotation.x = Math.PI * -.5;
scene.add(mesh);
```

scenegraph


WebGLRenderer
PerspectiveCamera
Scene
Mesh
MeshPhongMaterial
SphereBufferGeometry
DirectionalLight
Mesh
MeshPhongMaterial
PlaneBufferGeometry

Animation loop

We need an animation loop in order to update or change one or more of the attributes of an object frame by frame or as a function of time.

In our case, we want the ball to bounce up and down so we need to update the y coordinate or vertical position of the ball object every frame.

```
// Animation loop
function render(time) {
    time *= 0.001;  // convert time to seconds
    ball.position.y = Math.sin(4*time)+2;
    controls.update();
    renderer.render(scene, camera);
    requestAnimationFrame(render);
    }

    requestAnimationFrame(render);
}
```

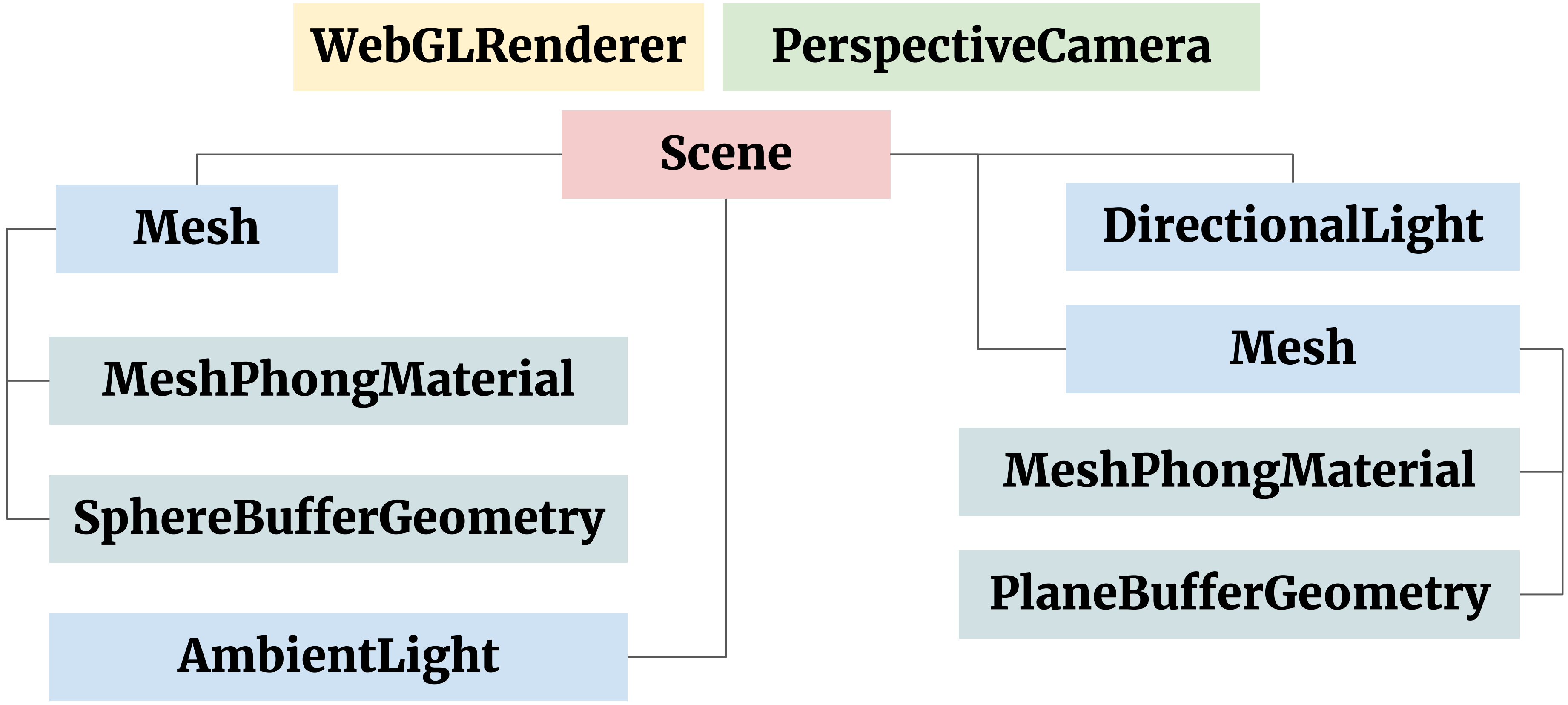
Ambient light

In addition to a directional light, we also need AmbientLight to illuminate the entire scene, not just in a particular direction

Add the following code right beneath the code for directional light:

```
const light = new THREE.AmbientLight(0x636363);
// soft white light
scene.add(light);
```

scenegraph


WebGLRenderer
PerspectiveCamera
Scene
Mesh
MeshPhongMaterial
SphereBufferGeometry
AmbientLight
DirectionalLight
Mesh
MeshPhongMaterial
PlaneBufferGeometry

Orbit controls

Tools like OrbitControl are made available for use in the examples directory of the 3.js module.

We will also use a CDN to import the OrbitControls.

Then, we initialize the control object but we will not be adding it as a child of scene (not part of scenegraph).

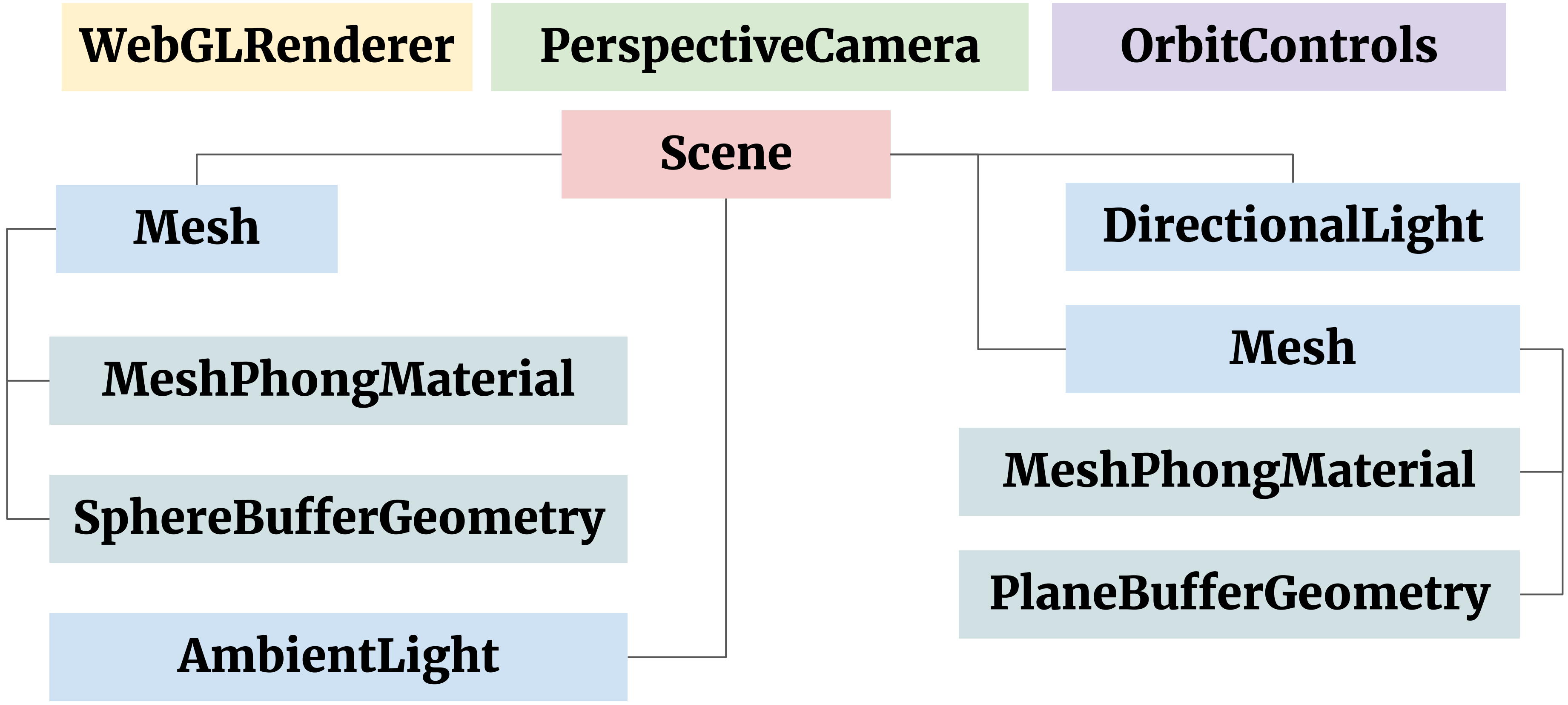
Finally, use the update method defined for the controls in our animation/render loop so that the view can be updated based on the input from the user.

```
import {OrbitControls} from
"https://cdn.jsdelivr.net/npm/three@v0.108.0/exa
mples/jsm/controls/OrbitControls.js";


const controls = new OrbitControls(camera,
renderer.domElement);


controls.update(); // in render function
```

scenegraph


WebGLRenderer
PerspectiveCamera
OrbitControls
Scene
Mesh
MeshPhongMaterial
SphereBufferGeometry
AmbientLight
DirectionalLight
Mesh
MeshPhongMaterial
PlaneBufferGeometry

Responsive design

To avoid distortion, we need to set the aspect of the camera to the aspect of the canvas's display size in our render/animation loop. We can do that by looking at the canvas's clientWidth and clientHeight properties.

To fix the blockiness and the pixelated edges, the resolution of canvas must be updated to the size of the canvas.

Canvas elements have 2 sizes:

- size the canvas is displayed on the page (set with CSS)
- the number of pixels in the canvas itself.

A canvas's internal size, its resolution, is often called its drawingbuffer size. We can set the canvas's drawingbuffer size by calling renderer.setSize to get "the same size the canvas is displayed".

```
// seperate function in main
function resizeRendererToDisplaySize(renderer) {
    const canvas = renderer.domElement;
    const width = canvas.clientWidth;
    const height = canvas.clientHeight;
    const needResize = canvas.width !== width ||
canvas.height !== height;
    if (needResize) {
        renderer.setSize(width, height, false);
    }
    return needResize;
}

// in the render function
if (resizeRendererToDisplaySize(renderer)) {
   const canvas = renderer.domElement;
   camera.aspect = canvas.clientWidth /
   canvas.clientHeight;
   camera.updateProjectionMatrix();
}
```

Thank you for coming!

All resources and code will be posted later.