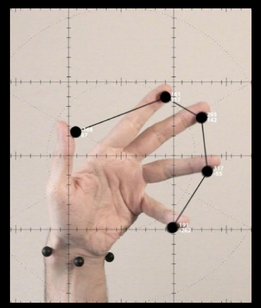
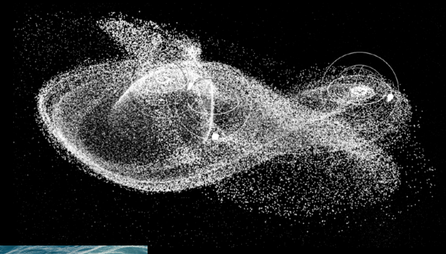
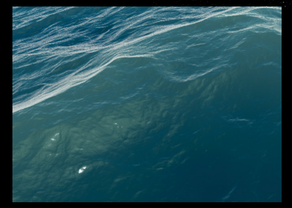
Three.js as a platform for interaction, storytelling, and *experimental graphics*







Susie Li, Dec 2, 2025

Interaction

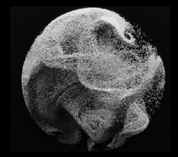
Three.js turns the browser into a canvas for interactive computation — not just rendering. It plays well with real-time inputs like mouse, touch, sensors, audio, and computer vision models such as MediaPipe. Instead of static scenes, we can build dynamic environments that respond to gestures, motion, and behavior. This shifts the mindset from *graphics as display* to *graphics as interface*, enabling playful, embodied, spatial interaction.

Storytelling

Three.js lets us build narrative experiences — guiding users through spatial scenes, transitions, and cinematic motion. The web becomes a medium for immersive explorable explanations: where data, environment, and narrative work together to communicate ideas

Experimental graphics

Three.js is lightweight, modular, and fast to iterate with — perfect for prototyping creative ideas without rebuilding WebGL boilerplate each time. Custom shaders, procedural geometry, particles, and physics can be combined rapidly, letting us experiment with form, movement, and materiality. It's a sandbox for speculative graphics, generative art, and research explorations where the goal is to push boundaries and test concepts quickly.



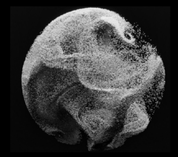
Outline

Intro:
How it works

Demos:
What we can do with it

Try it yourself?

Contents



Outline

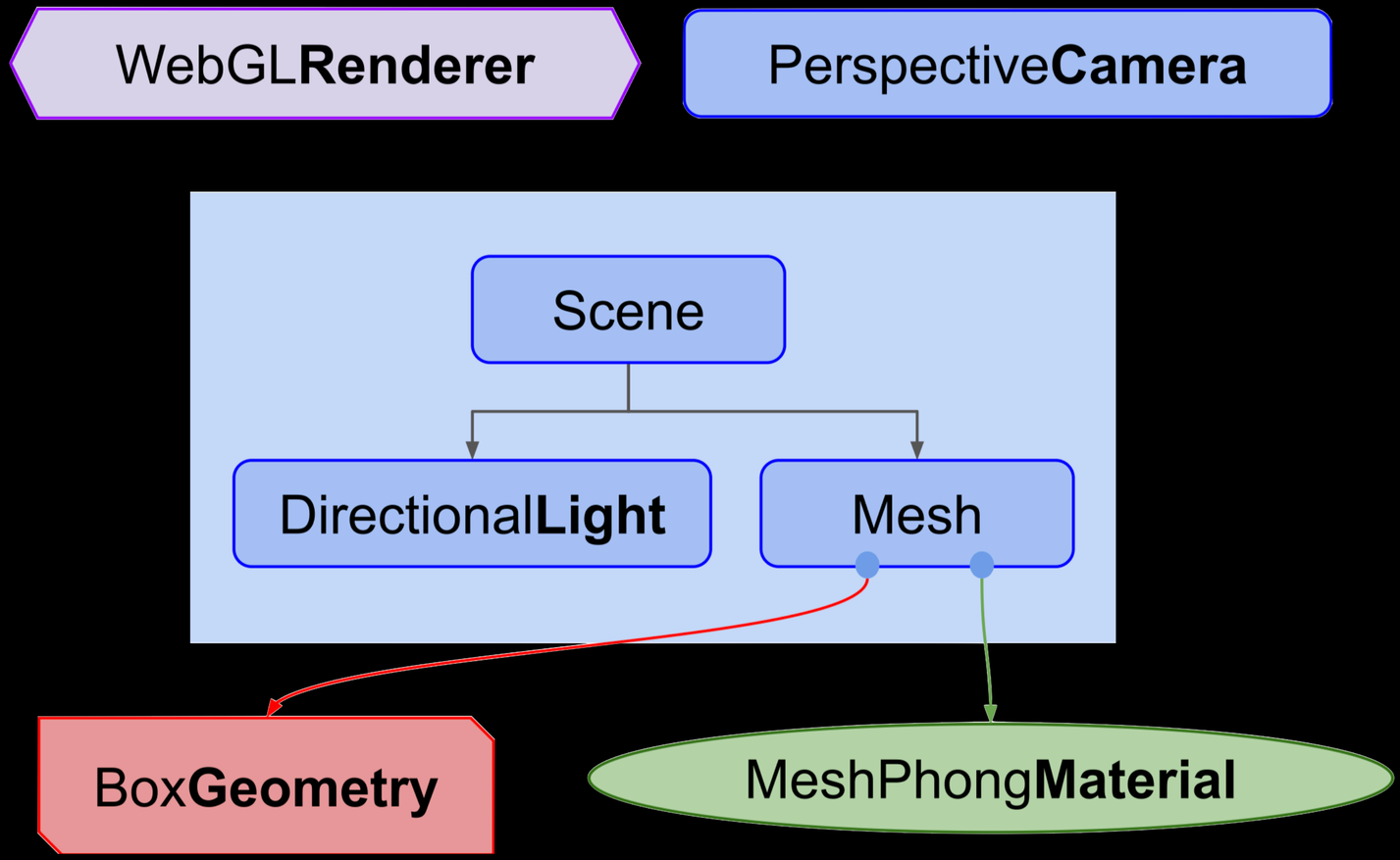
Intro:
How it works

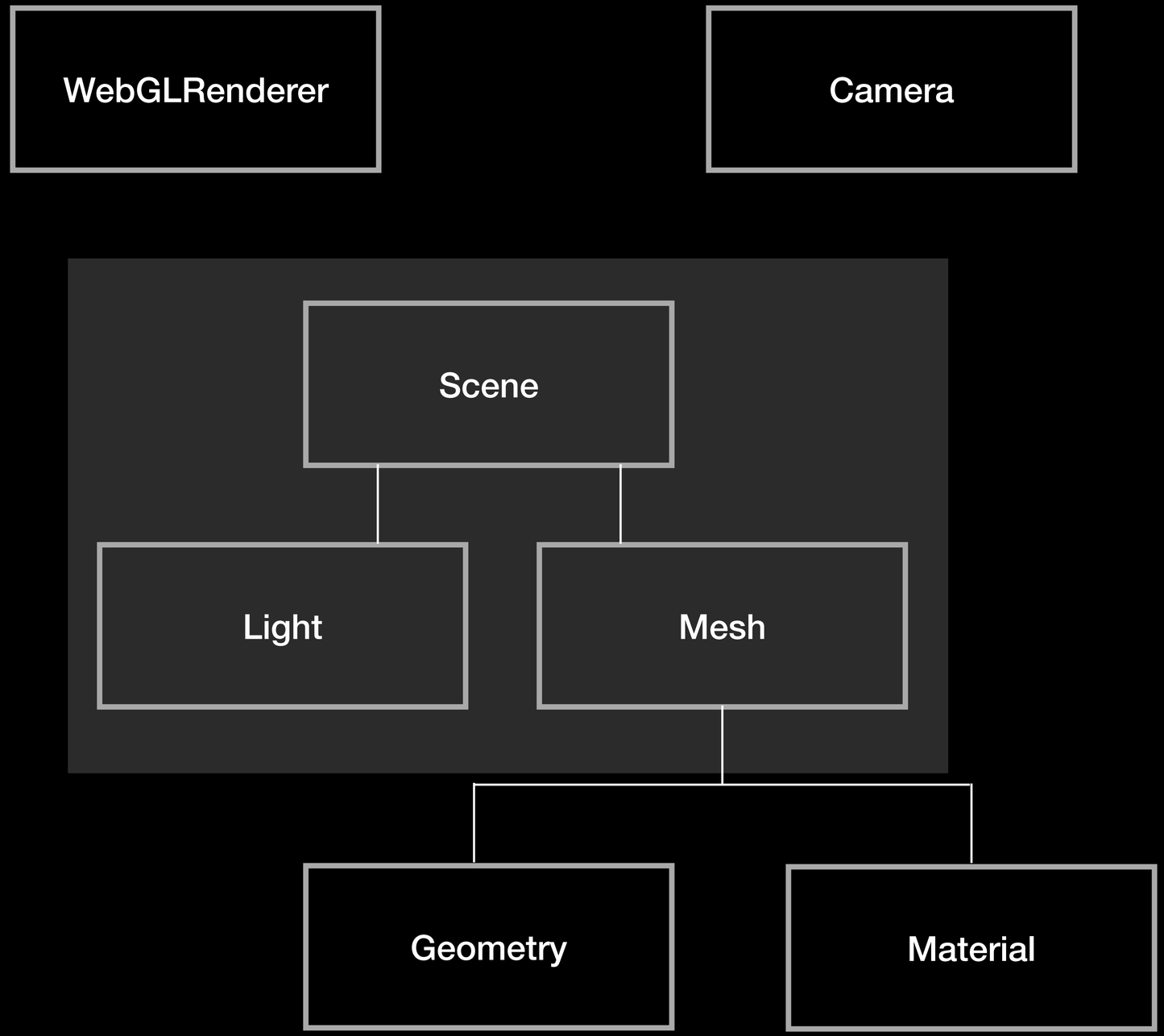
Demos:
What we can do with it

Try it yourself?

Contents

Basic architecture to render a scene in three.js




WebGLRenderer
Camera
Scene
Light
Mesh
Geometry
Material

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Demo: scene, camera, renderer in three.js

Geometries in Three.js

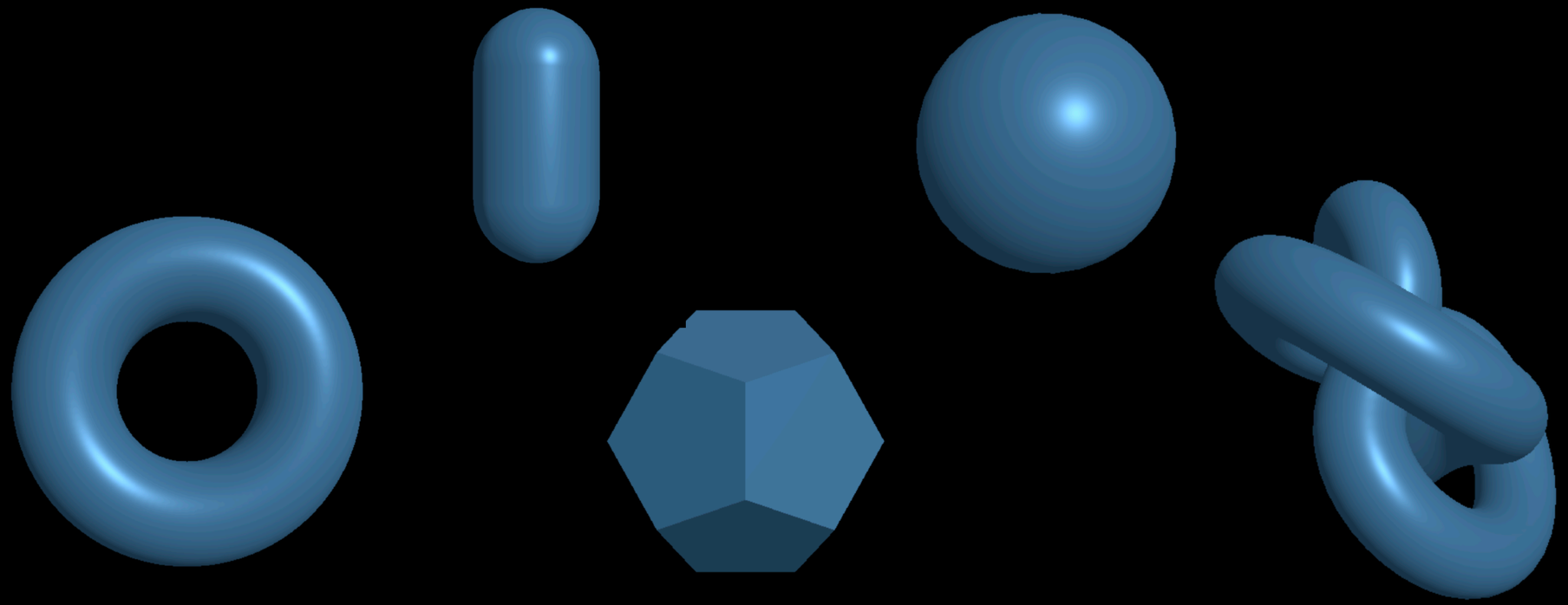
Geometries

- BoxGeometry
- CapsuleGeometry
- CircleGeometry
- ConeGeometry
- CylinderGeometry
- DodecahedronGeometry
- EdgesGeometry
- ExtrudeGeometry
- IcosahedronGeometry
- LatheGeometry
- OctahedronGeometry
- PlaneGeometry
- PolyhedronGeometry
- RingGeometry
- ShapeGeometry
- SphereGeometry
- TetrahedronGeometry
- TorusGeometry
- TorusKnotGeometry
- TubeGeometry
- WireframeGeometry

Under the hood, Three.js geometries are still vertices and faces: all geometries are composed of vertices (3D points) and faces (triangles).

Three.js handles geometry by using BufferGeometry to store vertex data in flat arrays. It manages attributes like vertex positions, normals (for lighting calculations), and UV coordinates (for texture mapping) within buffers.

BufferAttribute: BufferGeometry uses BufferAttribute objects to wrap these flat arrays (e.g., position, normal, color data). Each BufferAttribute holds the data for a specific attribute of the vertices.



Materials in Three.js

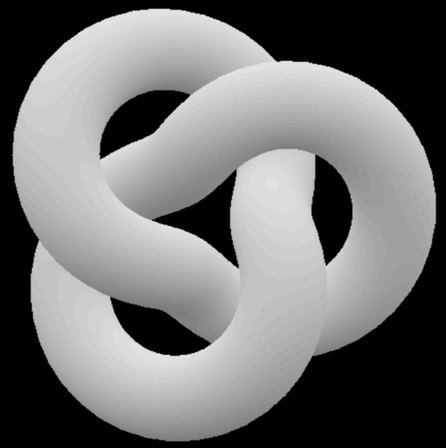
Materials

- Line2NodeMaterial
- LineBasicMaterial
- LineBasicNodeMaterial
- LineDashedMaterial
- LineDashedNodeMaterial
- Material
- MeshBasicMaterial
- MeshBasicNodeMaterial
- MeshDepthMaterial
- MeshDistanceMaterial
- MeshLambertMaterial
- MeshLambertNodeMaterial
- MeshMatcapMaterial
- MeshMatcapNodeMaterial
- MeshNormalMaterial
- MeshNormalNodeMaterial
- MeshPhongMaterial
- MeshPhongNodeMaterial
- MeshPhysicalMaterial
- MeshPhysicalNodeMaterial
- MeshSSSNodeMaterial
- MeshStandardMaterial
- MeshStandardNodeMaterial

Under the hood, Three.js handles materials by choosing and building the correct shader program based on the user's specified material mode (e.g. list on the left).

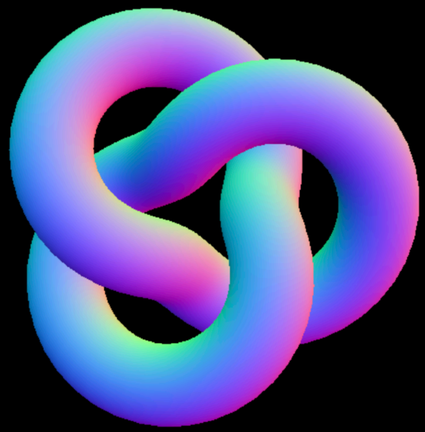
Materials in Three.js (MeshStandardMaterial, MeshBasicMaterial, etc.) are essentially **descriptions of rendering behavior**, not shaders themselves. They store settings like: color, metalness, roughness

When a mesh with this material is rendered, Three.js dynamically composes a shader out of reusable GLSL snippets based on info such as whether the material use normalMap / envMap, which light types to use (directional/spot/point/etc?), and whether shadows are enabled.



MeshMatcapMaterial

MeshToonMaterial



MeshNormalMaterial

Materials in Three.js

Materials

- Line2NodeMaterial
- LineBasicMaterial
- LineBasicNodeMaterial
- LineDashedMaterial
- LineDashedNodeMaterial
- Material
- MeshBasicMaterial
- MeshBasicNodeMaterial
- MeshDepthMaterial
- MeshDistanceMaterial
- MeshLambertMaterial
- MeshLambertNodeMaterial
- MeshMatcapMaterial
- MeshMatcapNodeMaterial
- MeshNormalMaterial
- MeshNormalNodeMaterial
- MeshPhongMaterial
- MeshPhongNodeMaterial
- MeshPhysicalMaterial
- MeshPhysicalNodeMaterial
- MeshSSSNodeMaterial
- MeshStandardMaterial
- MeshStandardNodeMaterial

Internally, three.js does the following:

gl.createShader

gl.shaderSource

gl.compileShader

gl.createProgram

gl.attachShader

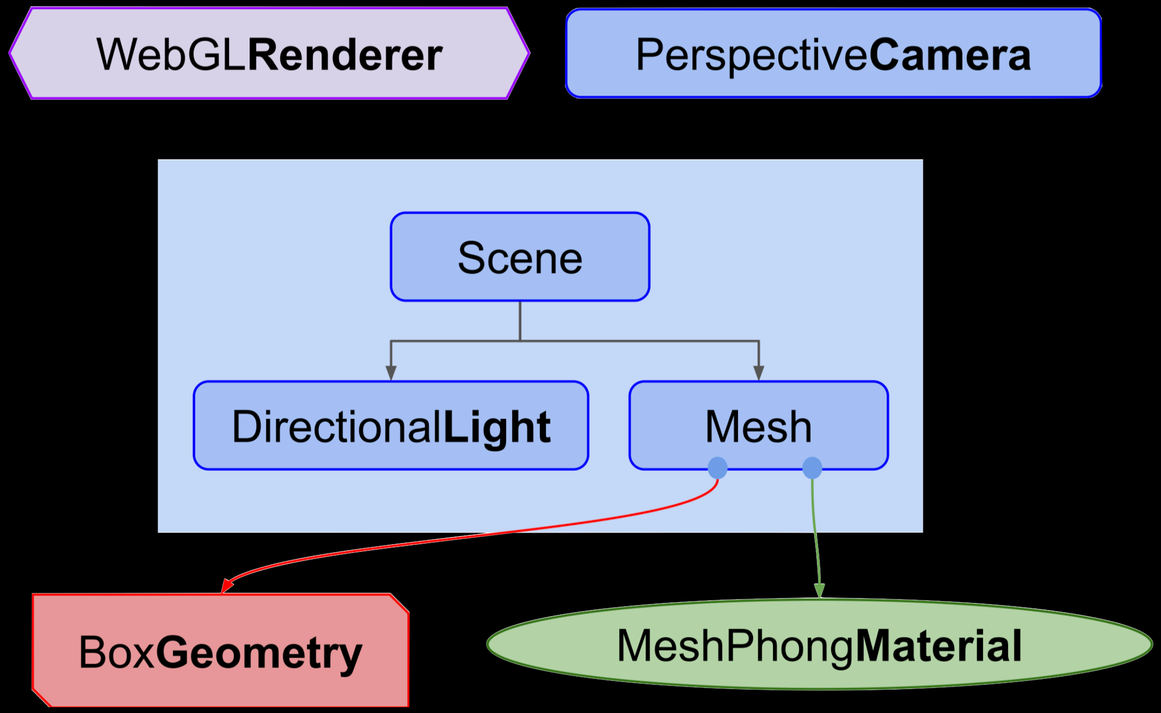
gl.linkProgram

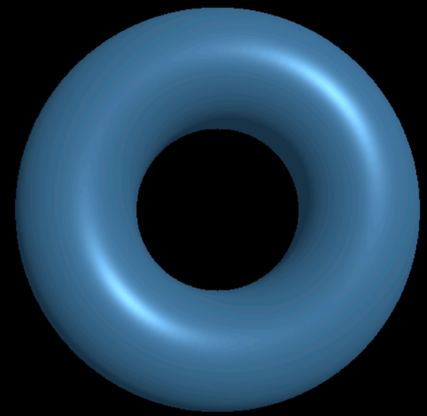
To improve performance, Three.js caches the compiled shader so if another object uses the same features, it reuses the program rather than recompiling.

Uniforms & attributes are auto-managed. Once a shader program exists, Three.js uploads the required uniforms:

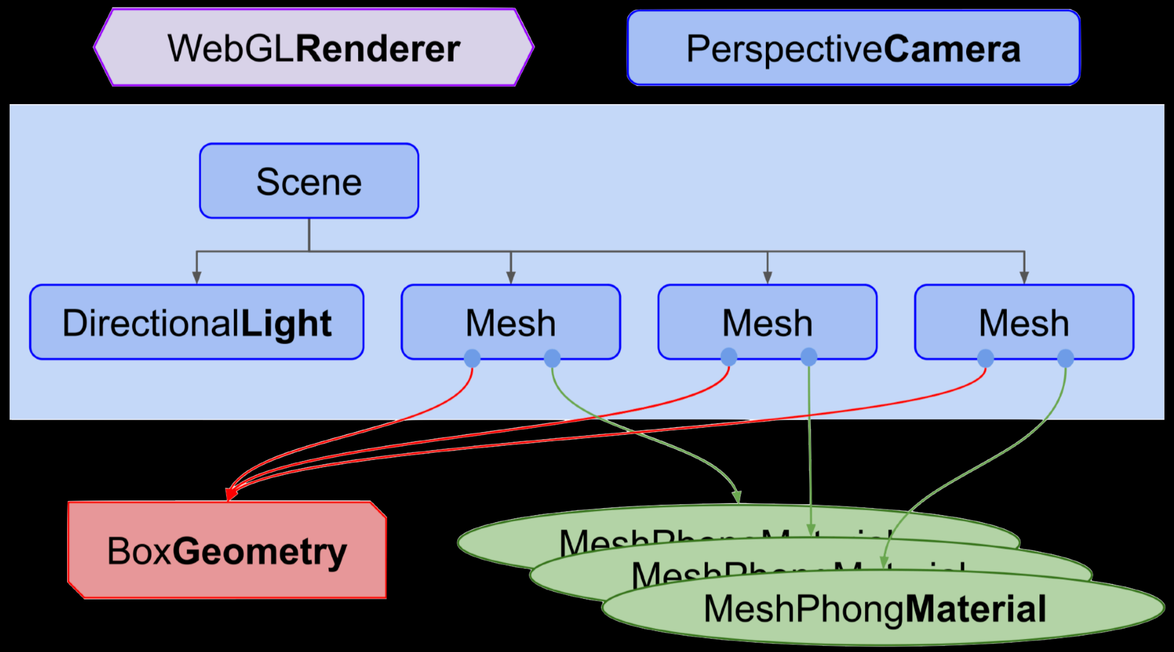
- transformation matrices (modelViewMatrix, projectionMatrix, normalMatrix)
- light uniforms
- texture samplers
- material properties

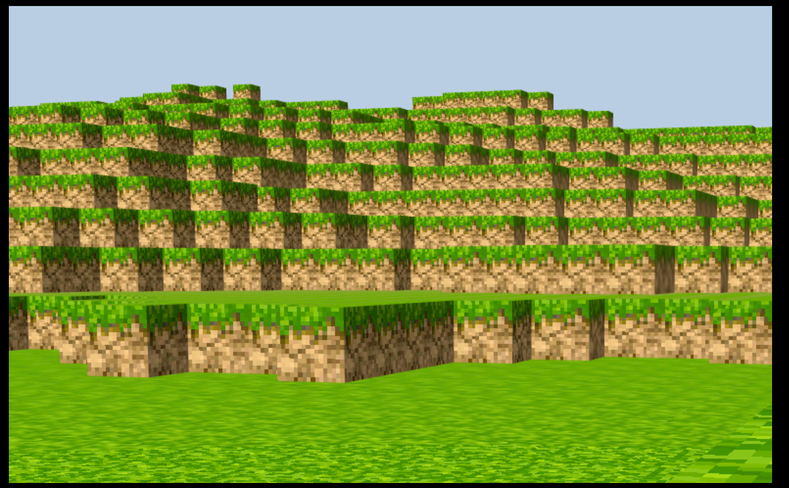
Basic architecture to render a scene in three.js



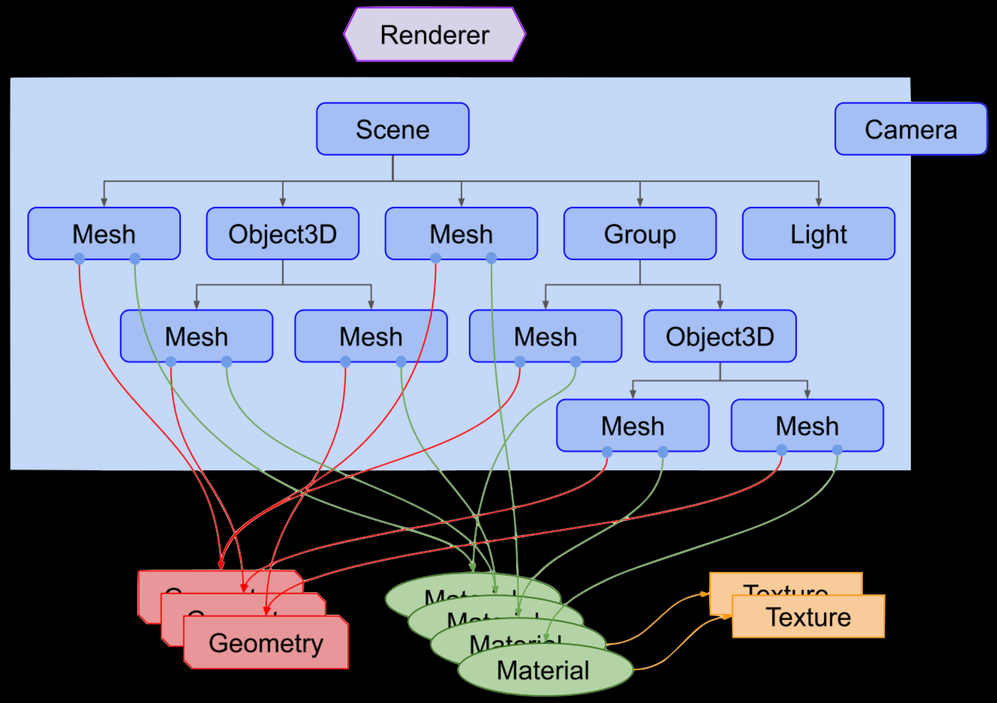


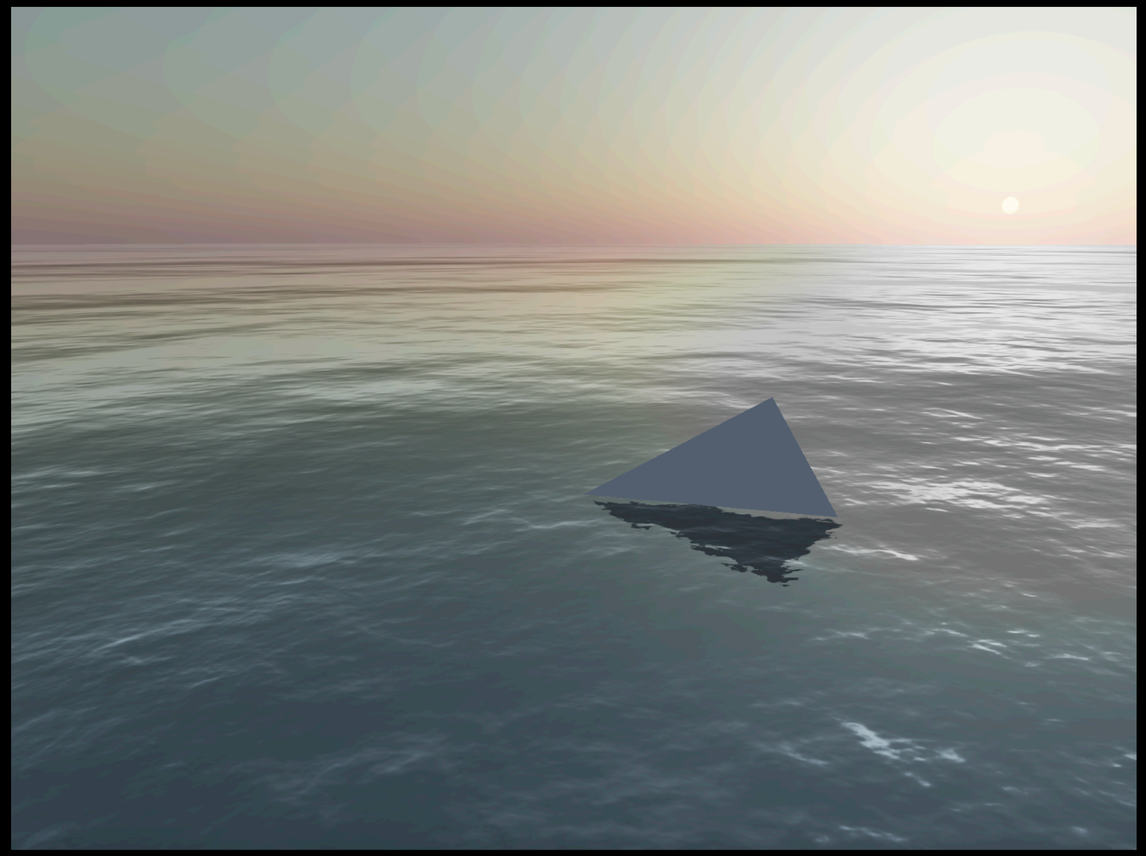
Slightly more objects and textures, but underlying architecture is the same

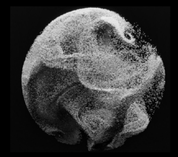




Even more complex scene - more shapes, geometries, 3D models/assets, but the architecture is still exactly the same!







Outline

Intro:
How it works

Demos:
What we can do with it

Try it yourself?

Contents

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/ shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/ shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/ shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/ shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

+ Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

+ Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

+ Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/ shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

+ Loaders (assets) pipeline

Model, texture, environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Key Abstractions in Three.js (vs Raw WebGL) + Tradeoffs

Camera/scene/renderer

Three.js gives a high-level scene graph and a renderer that manages the GPU pipeline. Instead of manually creating buffers, matrices, and projection transforms every frame, you declare objects and the system handles draw calls and transforms.

Tradeoffs:

Gain:
- Rapid scene construction, intuitive mental model, fewer lines of code
- Easy camera control (PerspectiveCamera, OrbitControls, etc.)

Lose:
- Fine-grained control over the draw pipeline
- More hidden work = harder to optimize extreme edge cases

Raw WebGL equivalent: manually build MVP matrices, manage VAOs/VBOs, handle render loops.

Loading geometry/mesh

Typed geometry attributes (position, normal, UVs, custom attributes) managed automatically rather than manually populating buffers.

Tradeoffs:

Gain:
- Easier to construct and modify geometry
- Instancing (InstancedMesh) and loaders for common formats (GLTFLoader, OBJLoader)

Lose:
- Little support for deep custom vertex formats or advanced GPGPU geometry pipelines

Shader/texture/GLSL

Manual shader compilation, linking, and binding
Managing uniform and attribute locations
Dozens of shader combinations for lights/ shadows

Tradeoffs:

Gain:
- Plug-and-play materials “presets”
- Access to structured uniforms, shader chunks, and physically-based rendering

Lose:
- Less control over pipeline stages (e.g., MRT, custom blending, compute-style passes)
- Harder to modify internals beyond shader chunks

Loaders (assets) pipeline

Model, texture, HDR environment, animation, and compression loaders.

Tradeoffs:

Gain:
- A few lines of code to import animated character/animated scene
- Ecosystem of tools (Blender → GLTF → scene)

Lose:
- File format expectations (GLTF strongly preferred)
- Harder to integrate non-standard or experimental formats

Custom Shaders in Three.js

If we want to write raw GLSL but keep Three.js features:

```
new THREE.ShaderMaterial({ vertexShader, fragmentShader, uniforms });
```

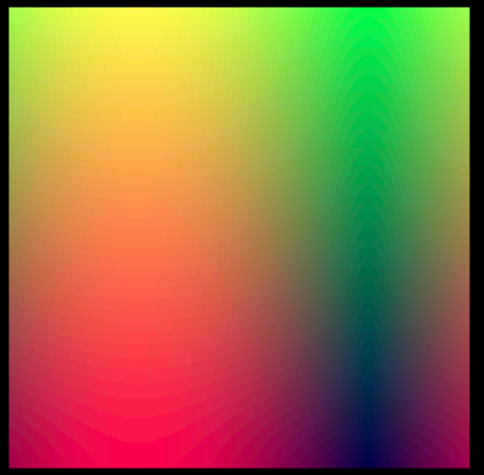
glsl strings, same as in WebGL

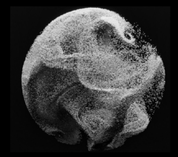
e.g. color, time, etc

Demo: custom shader with a time uniform

```
export const vertexShader = `
  varying vec2 vUv;

  void main() {
    vUv = uv; // pass UV coords to the fragment shader
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
  }
`;

// ——— Fragment Shader ———
export const fragmentShader = `
  uniform float uTime;
  varying vec2 vUv;

  void main() {
    float r = abs(sin(uTime + vUv.x * 3.14));
    float g = vUv.y;
    float b = 0.3;

    gl_FragColor = vec4(r, g, b, 1.0);
  }
```

```
// Shader material with our own GLSL + a uniform (time)
const material = new THREE.ShaderMaterial({
  vertexShader,
  fragmentShader,
  uniforms: {
    uTime: { value: 0.0 },
  },
});

const mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);
```





Outline

Intro:
How it works

Demos:
What we can do with it

Try it yourself?

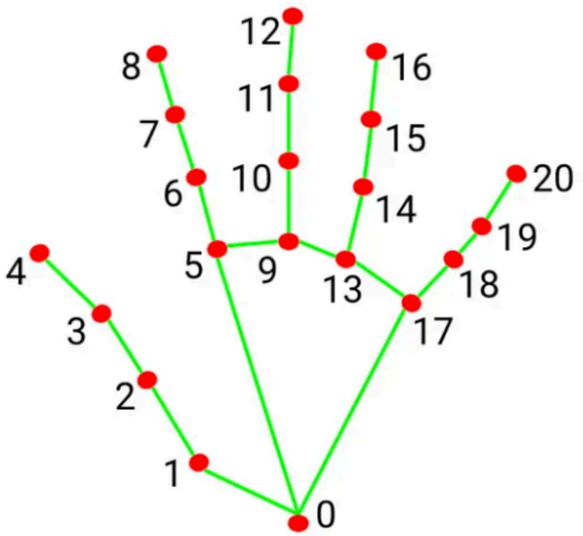
Contents

Three.js + interaction modes

Demo: https://susiesyli.com/neural-interfaces/cube/index.html

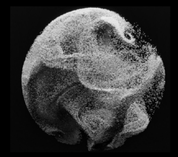
MediaPipe

21 “landmarks” aka tracking points:



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

- each point is (x, y, z)
- x and y are normalized into [0, 1]
- (0,0) = top-left of the webcam frame
- (1,1) = bottom-right
- z = depth from the camera



Outline

Intro:
How it works

Demos:
What we can do with it

Try it yourself?

Contents

Animation

Three.js animation is basically 3 pieces working together:

AnimationClip – what should happen (the “script”)

At 0 seconds the tower is short
At 1 second the tower gets taller
At 2 seconds the tower shrinks again

An AnimationClip is a container for one animation, like:

- “Walk”
- “Run”
- “OpenDoor”
- “BounceUpAndDown”

Each clip has:

- name: e.g. "Bounce"
- duration: in seconds
- tracks: an array of KeyframeTracks

KeyframeTrack – how a single property changes over time

Inside the script are tracks that say how one property changes

E.g. a single KeyframeTrack for just the scale.y property:

Time	Tower Y scale
0s	1.0
1s	1.3
2s	1.8
3s	1.2

There may be many tracks in one clip:

- One for rotation of a windmill
- One for position of a car
- One for light intensity… etc.

So Clip = collection of KeyframeTracks.

AnimationMixer – execution of script: advances time, blends clips, and writes values into the scene

- Keeps track of current time inside the animation
- Looks up what value each track should have at that time
- Interpolates smoothly between keyframes
- Writes those values into the 3D objects

Every frame we call:

```
mixer.update(deltaTime);
```

and the mixer moves the animation forward a step.

Demo: 3D cityscape (GLB)

Demo: live-tweaking animation by editing the **mixer**

Side note: in this animation, we import an ‘asset’ - a 3D model file that contains the objects we see, as well as the ‘script’. In this case, the keyframes and clip are baked into the model asset.

*GLB = GL Transmission Format Binary file

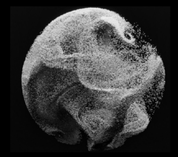
At a high level, every frame:

```
const delta = clock.getDelta();   // seconds since last frame
mixer.update(delta);              // advance all animations
renderer.render(scene, camera);
```

Inside mixer.update(delta) three.js:

1. Advances an internal time cursor for each playing AnimationAction.
2. For each AnimationClip on each action, looks at all its KeyframeTracks.
3. For each track, samples the value at the current time (interpolating between keyframes).
4. Blends contributions if multiple actions affect the same property.
5. Writes the result into your scene graph (mesh.position, bone.quaternion, material uniforms, etc.).

You almost never touch that inner loop yourself; you just build clips, tracks, and a mixer, then call update() each frame.



Outline

Intro:
How it works

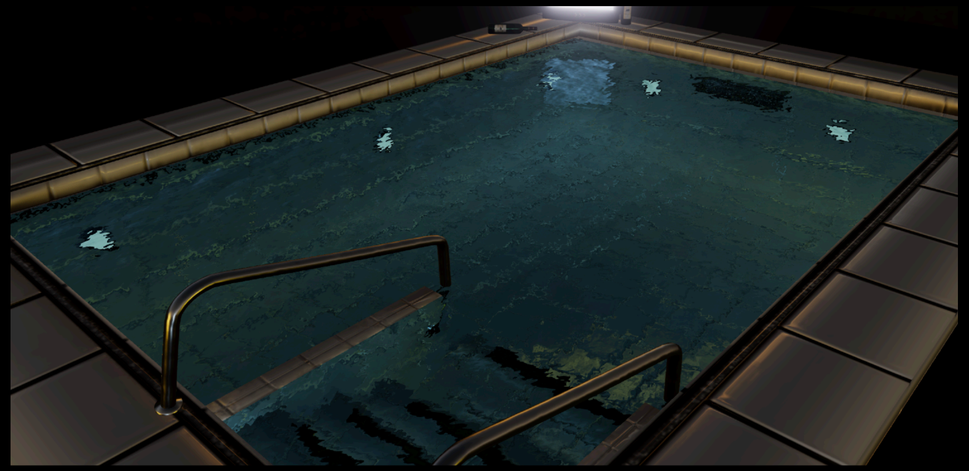
Demos:
What we can do with it

Try it yourself?

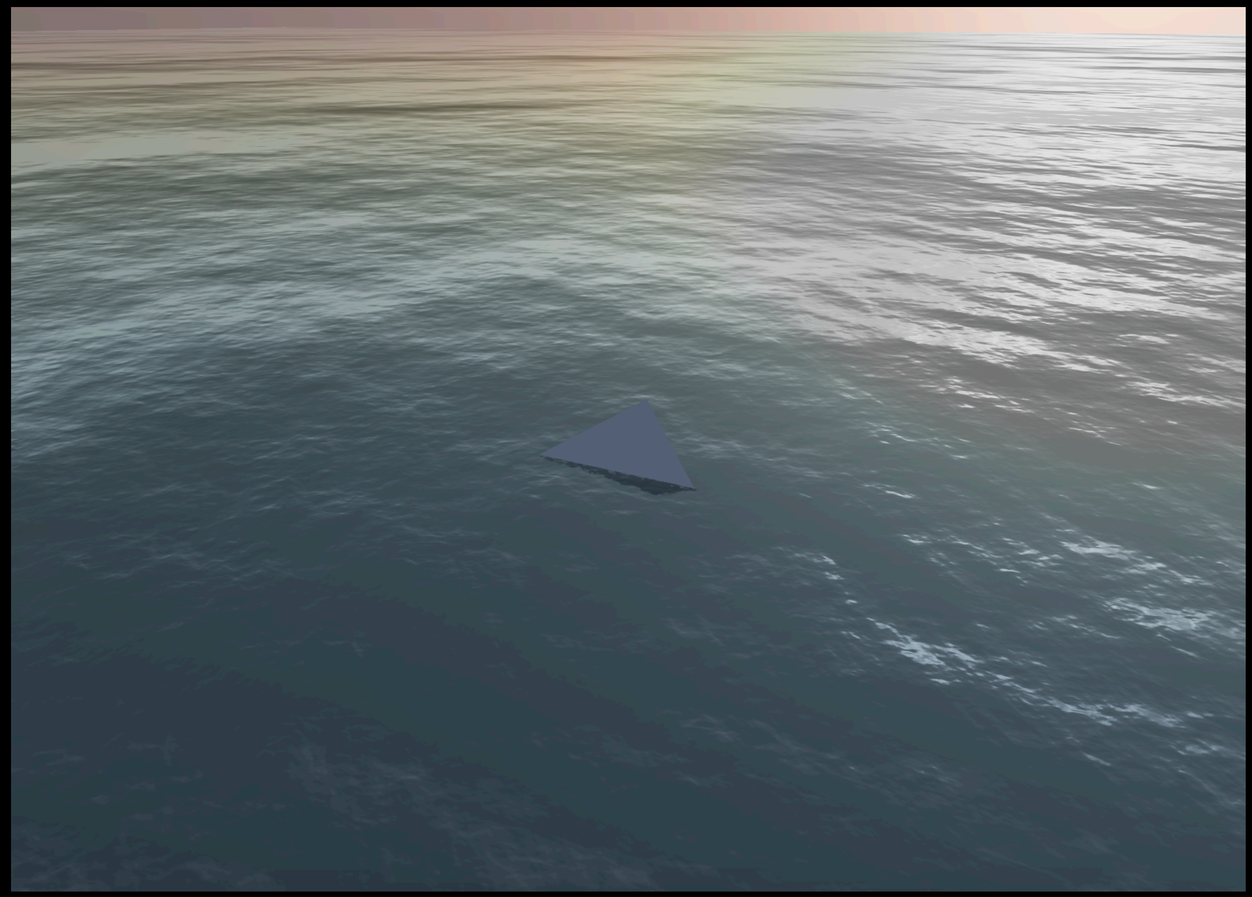
Contents

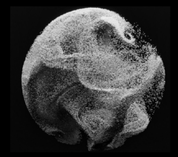
Physical simulation

https://threejs.org/examples/?q=water#webgl_gpgpu_water



https://threejs.org/examples/?q=water#webgl_shaders_ocean





Outline

Intro:
How it works

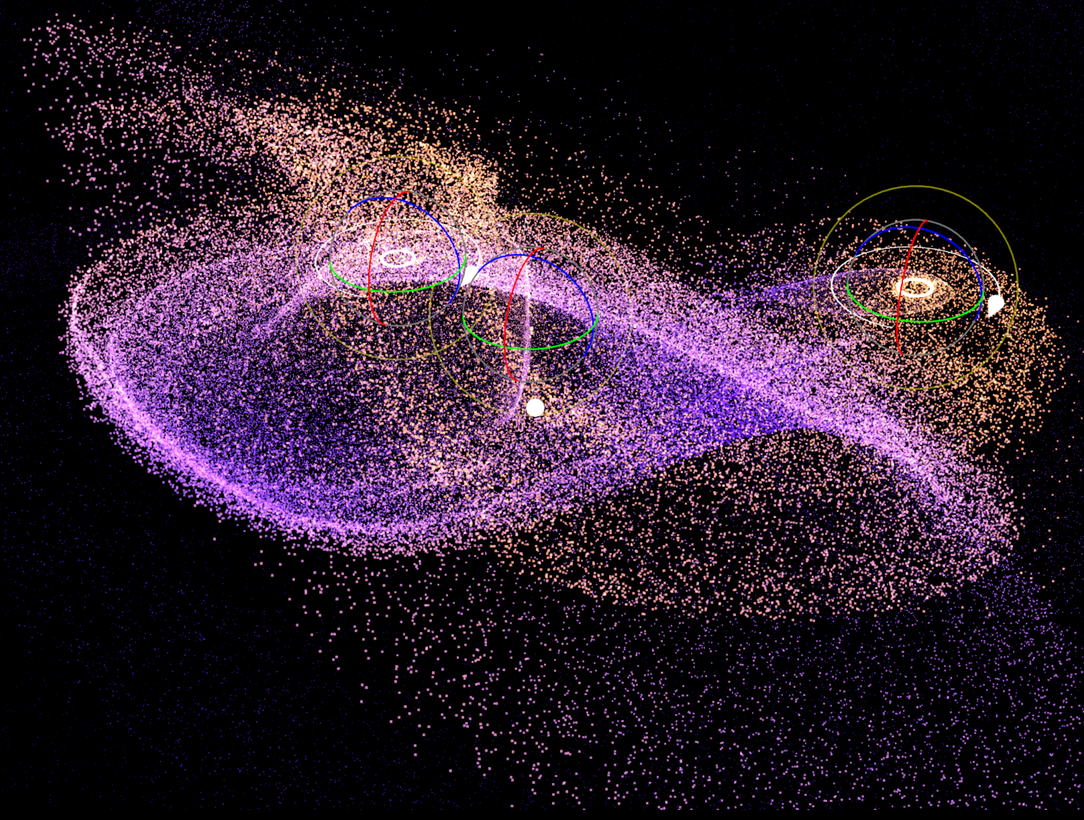
Demos:
What we can do with it

Try it yourself?

Contents

Particle systems w/ motion

https://threejs.org/examples/?q=webgpu#webgpu_tsl_compute_attractors_particles



ASCII shader

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

```
import { AsciiEffect } from 'three/examples/jsm/effects/AsciiEffect.js';

const effect = new AsciiEffect(renderer, ' .:-=+*#%@', { invert: true });
```

More resources:

Physics

- Oimo.js
- enable3d
- ammo.js
- cannon-es
- rapier
- Jolt

Postprocessing

- postprocessing

3D text & layout

Raycast performance

https://github.com/AxiomeCG/awesome-threejs

Web wrappers & frameworks

Particle systems

- three.quarks
- three-nebula

Inverse Kinematics

Questions? +
Project check-ins