Lecture 9

https://threejs.org/examples/#webgl_lights_hemisphere

Recap

- HTML Canvas
  - Comparison to SVG browser graphics
  - Drawing shapes
  - Basic Trigonometry
  - Basic collision detection
  - Animation and User Interaction
    - Examples
  - Advanced collision handling
  - Linear Algebra and Transformations
- SVG
- D3

3D Computer Graphics

3D Computer Graphics

- Graphics that use a three-dimensional representation of geometric data (often Cartesian) that is stored in the computer for the purposes of performing calculations and rendering images.
- Such images may be stored for viewing later or displayed in real-time.
- 3D computer graphics rely on many of the same algorithms as 2D vector computer graphics E.g.
  - Wire-frame models
  - Raster graphics for the final rendered display.

3D Models vs. 3D Graphics

- 3D computer graphics are often referred to as 3D models.
- The model is contained within the graphical data file.
  - A 3D model is the mathematical representation of any three-dimensional object.
- A model is not technically a graphic until it is displayed.
- A model can be displayed visually as a two-dimensional image through a process called 3D rendering or used in non-graphical computer simulations and calculations.
  - E.g. Medical imaging diagnosis
- A hologram is a three-dimensional image formed by the interference of light beams from a laser or other coherent light source.
- With 3D printing, 3D models are similarly rendered into a 3D physical representation of the model

3D Computer Graphics

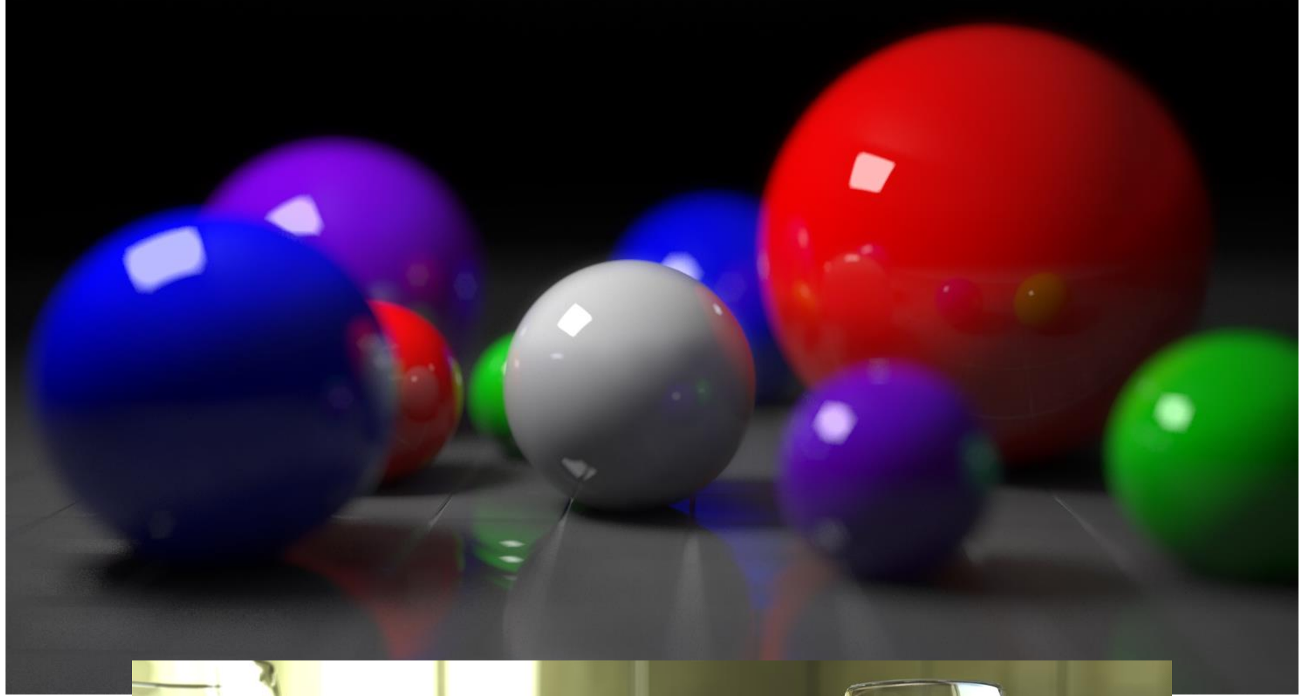
- 3 Basic Phases
  - **3D modelling:** – the process of forming a computer model of an object's shape.
    - The two most common sources of 3D models are 1) those that an artist or engineer creates on computer with some kind of 3D modeling tool, and 2) models scanned into a computer from real-world objects. Models can also be produced procedurally via simulation. A 3D model is formed from polygons. A polygon is an area formed from at least three vertices.
  - **Layout and animation:** – the motion and placement of objects within a scene
    - Before rendering into an image, objects must be laid out in a scene, defining spatial relationships including location and size. Animation refers to the temporal description of an object i.e., how it moves and deforms over time.
  - **3D rendering:** – the computer calculations that, based on light placement, surface types, and other qualities, generate the image
    - Rendering converts a model into an image by simulating light transport to get photo-realistic images.
    - The two basic operations in realistic rendering are transport (how much light gets from one place to another) and scattering (how surfaces interact with light). This step is usually performed using a 3D graphics API.
    - 3D projection (onto a receiver) displays a three-dimensional image in two dimensions.

Ray Tracing

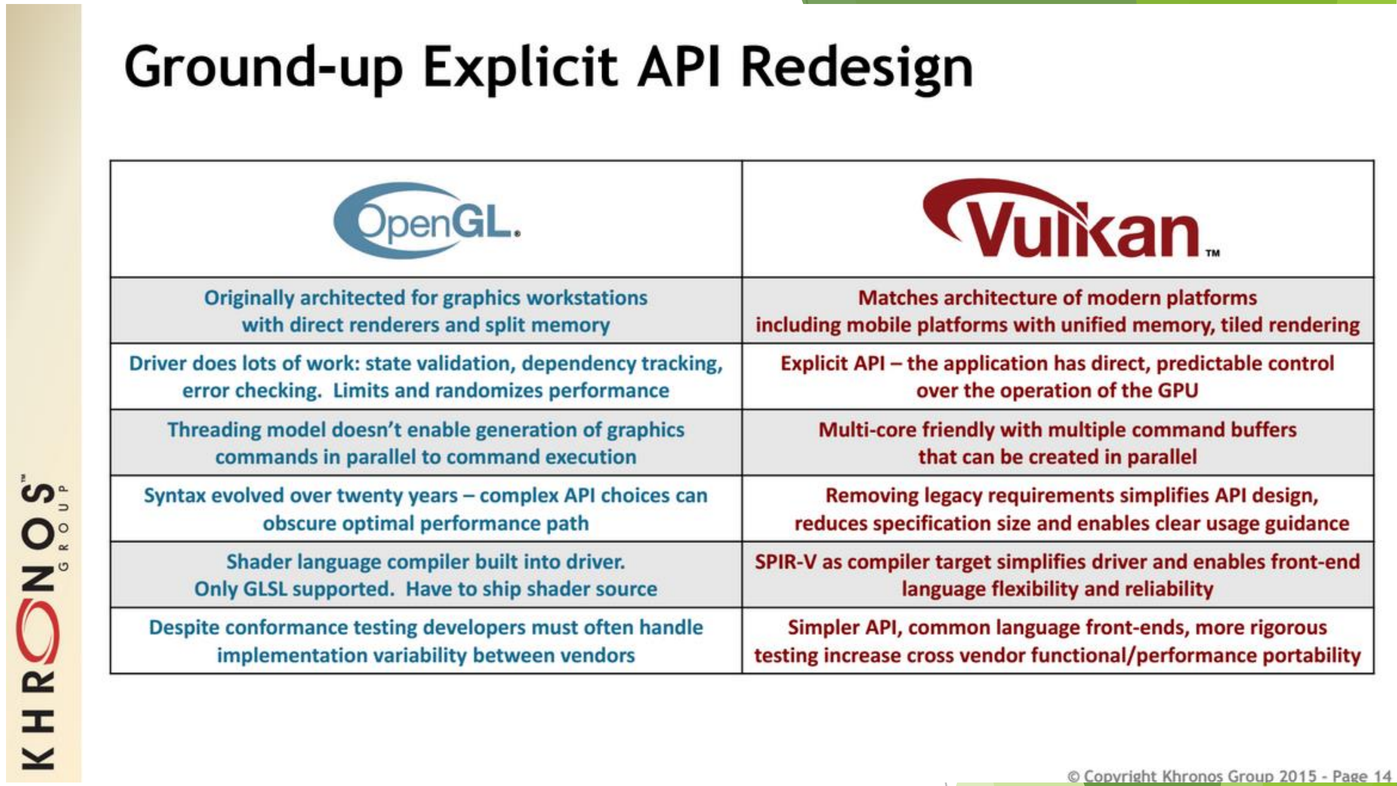
- Ray Tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects.
- The technique is capable of producing a very high degree of visual realism, usually higher than that of other typical rendering methods, but at a greater computational cost.
- This makes ray tracing best suited for applications where the image can be rendered slowly ahead of time, such as in still images and film and television visual effects, and more poorly suited for real-time applications like video games where speed is critical.
- Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering, and dispersion phenomena (such as chromatic aberration).
- Modern graphics cards can perform ray-tracing in real-time

Ray Tracing vs. Rasterisation

- For true photo-realism:
- Cannot compute color or shade of each object independently
  - Some objects are blocked from light
  - Light can reflect from object to object
  - Some objects might be translucent
- Can rasterization produce global lighting effects?
- Can ray tracing?
- The big advantage of rasterization is…?



OpenGL



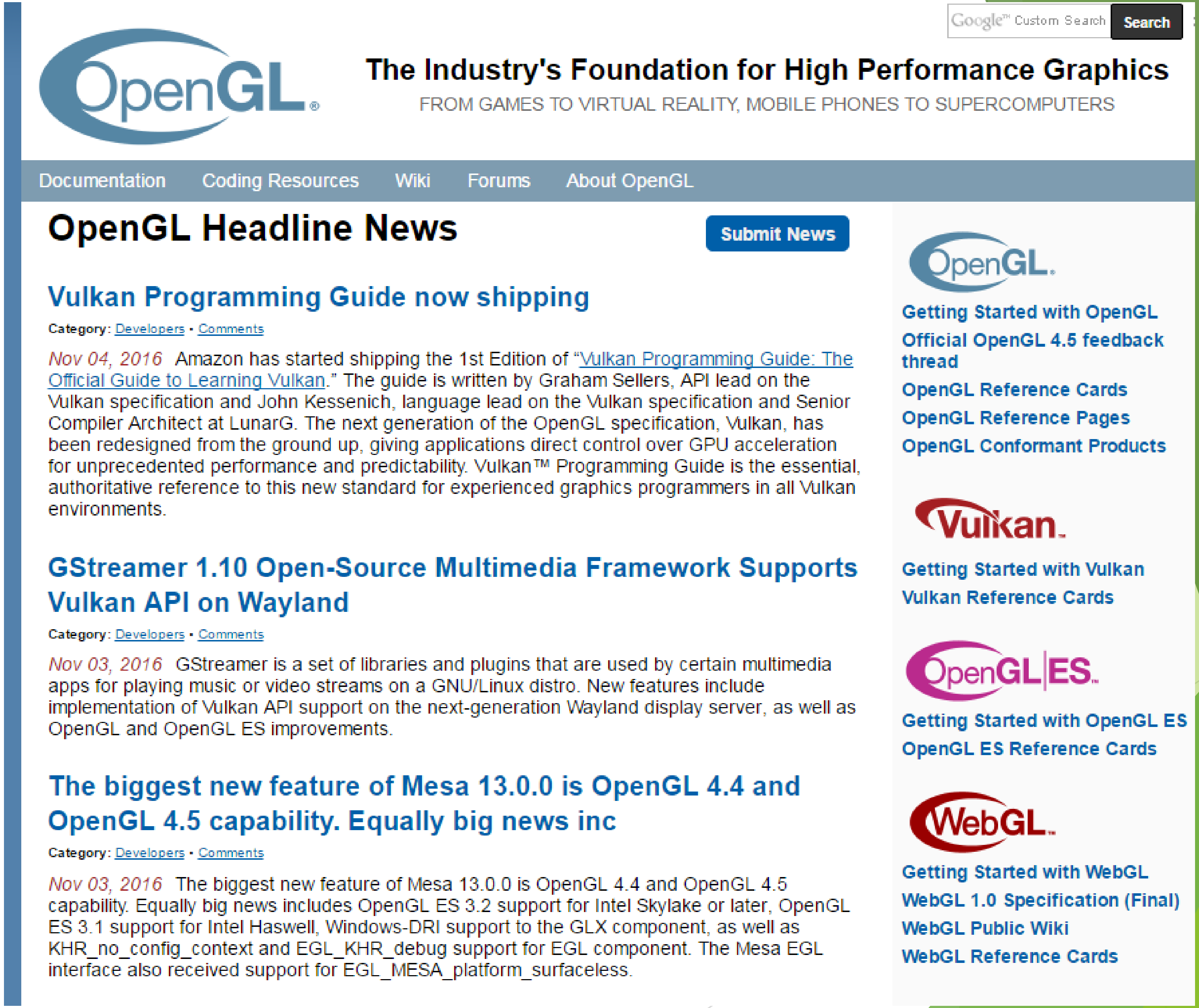
KHRONOS GROUP

Ground-up Explicit API Redesign

OpenGL.	Vulkan™
Originally architected for graphics workstations with direct renderers and split memory	Matches architecture of modern platforms including mobile platforms with unified memory, tiled rendering
Driver does lots of work: state validation, dependency tracking, error checking. Limits and randomizes performance	Explicit API – the application has direct, predictable control over the operation of the GPU
Threading model doesn't enable generation of graphics commands in parallel to command execution	Multi-core friendly with multiple command buffers that can be created in parallel
Syntax evolved over twenty years – complex API choices can obscure optimal performance path	Removing legacy requirements simplifies API design, reduces specification size and enables clear usage guidance
Shader language compiler built into driver. Only GLSL supported. Have to ship shader source	SPIR-V as compiler target simplifies driver and enables front-end language flexibility and reliability
Despite conformance testing developers must often handle implementation variability between vendors	Simpler API, common language front-ends, more rigorous testing increase cross vendor functional/performance portability

- Platform-neutral competitor to DirectX.
- Open standard for graphics programming
  - Developed by Silicon Graphics in 1992
  - Available on most platform
  - Bindings available for lots of languages…
  - It's low level
  - Supported by all graphics cards (different versions)
- Designers made decision to do *only rendering*, no input, audio, or windowing.
  - "Windowing" typically requires another library
    - e.g. GLUT (GL Utility Toolkit)
- Vulkan API announced as the successor technology (2015)

OpenGL



The Industry's Foundation for High Performance Graphics

FROM GAMES TO VIRTUAL REALITY, MOBILE PHONES TO SUPERCOMPUTERS

OpenGL Headline News

Vulkan Programming Guide now shipping

Category: Developers • Comments

Nov 04, 2016 Amazon has started shipping the 1st Edition of "Vulkan Programming Guide: The Official Guide to Learning Vulkan." The guide is written by Graham Sellers, API lead on the Vulkan specification and John Kessenich, language lead on the Vulkan specification and Senior Compiler Architect at LunarG. The next generation of the OpenGL specification, Vulkan, has been redesigned from the ground up, giving applications direct control over GPU acceleration for unprecedented performance and predictability. Vulkan™ Programming Guide is the essential, authoritative reference to this new standard for experienced graphics programmers in all Vulkan environments.

GStreamer 1.10 Open-Source Multimedia Framework Supports Vulkan API on Wayland

Category: Developers • Comments

Nov 03, 2016 GStreamer is a set of libraries and plugins that are used by certain multimedia apps for playing music or video streams on a GNU/Linux distro. New features include implementation of Vulkan API support on the next-generation Wayland display server, as well as OpenGL and OpenGL ES improvements.

The biggest new feature of Mesa 13.0.0 is OpenGL 4.4 and OpenGL 4.5 capability. Equally big news inc

Category: Developers • Comments

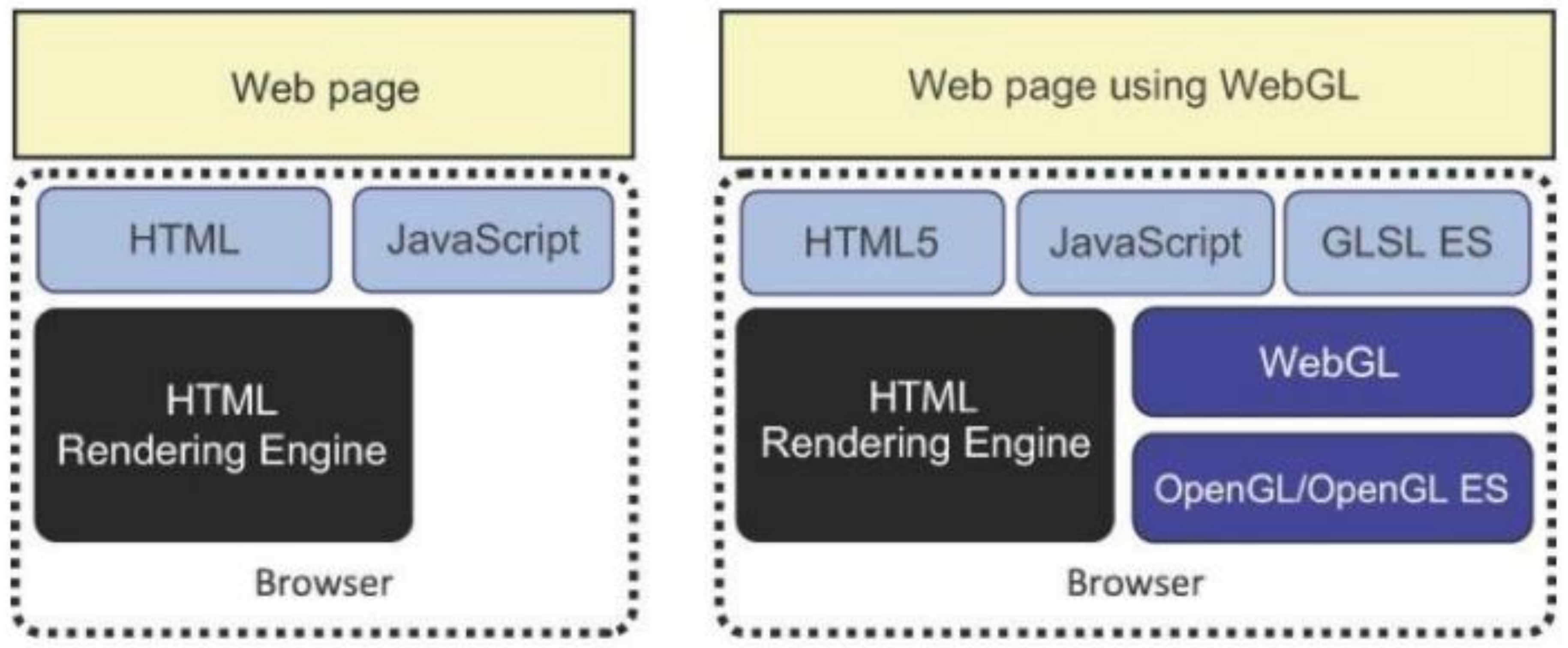
Nov 03, 2016 The biggest new feature of Mesa 13.0.0 is OpenGL 4.4 and OpenGL 4.5 capability. Equally big news includes OpenGL ES 3.2 support for Intel Skylake or later, OpenGL ES 3.1 support for Intel Haswell, Windows-DRI support to the GLX component, as well as KHR_no_config_context and EGL_KHR_debug support for EGL component. The Mesa EGL interface also received support for EGL_MESA_platform_surfaceless.

OpenGL
Getting Started with OpenGL
Official OpenGL 4.5 feedback thread
OpenGL Reference Cards
OpenGL Reference Pages
OpenGL Conformant Products
Vulkan
Getting Started with Vulkan
Vulkan Reference Cards
OpenGL ES
Getting Started with OpenGL ES
OpenGL ES Reference Cards
WebGL
Getting Started with WebGL
WebGL 1.0 Specification (Final)
WebGL Public Wiki
WebGL Reference Cards

WebGL

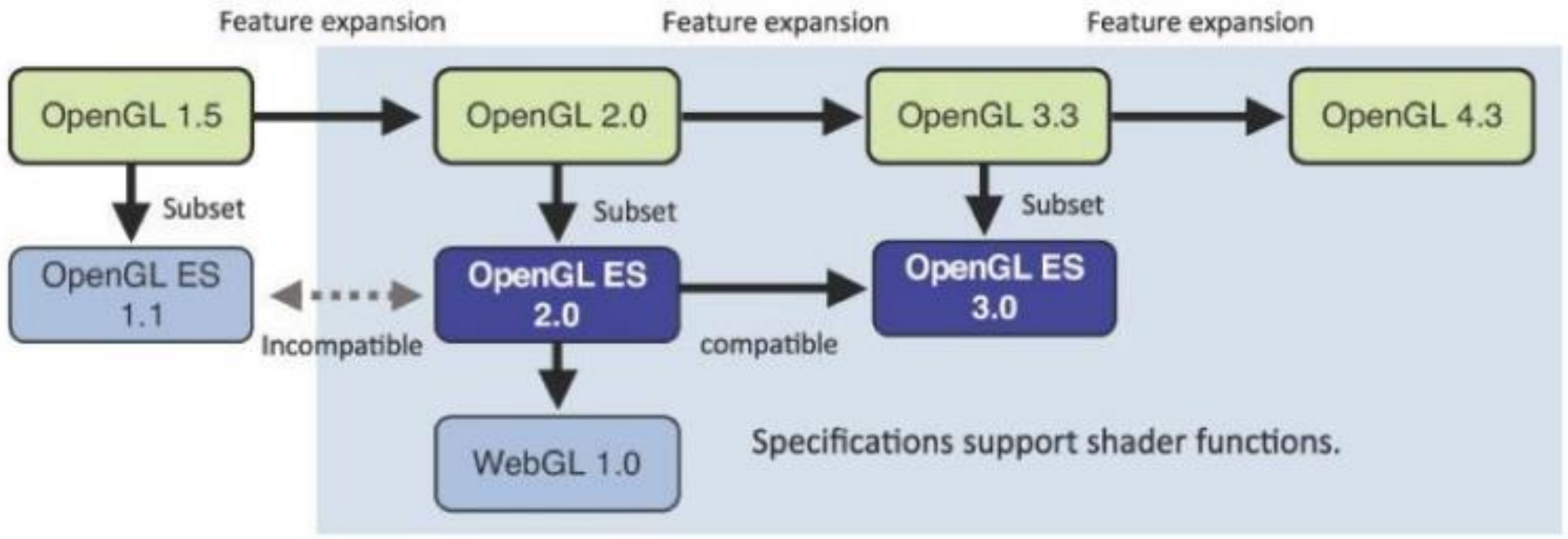
- WebGL relatively new (2011) 3D graphics support for web
- WebGL advantages
  - Runs in browser
  - Naturally cross-platform
  - Don’t need to obtain/build other libraries
  - Gives you “windowing” for free
  - Easy to publish/share your stuff
- Disadvantages
  - Depends on how you feel about JavaScript
  - Performance can be tricky

WebGL Application Structure



WebGL is not exactly OpenGL

- ES stands for Embedded Systems



WebGL is not exactly OpenGL

- WebGL is not exactly OpenGL
  - It is derived from OpenGL ES 2.0 which is a subset of OpenGL 2.0
  - Therefore WebGL only supports a subset of shaders that OpenGL offers
  - WebGL does not support 3D textures and vertex array objects
  - WebGL natively supports windowing
- OpenGL library is written in C++
- WebGL is a Javascript interface

WebGL Programming

- With JavaScript APIs like WebGL, modern browsers are fully capable of rendering advanced 2D and 3D graphics without help from third-party plugins.
- By leveraging the horsepower of dedicated graphics processors, WebGL gives our web pages access to dynamic shading and realistic physics.
- Such powerful APIs typically come with a drawback.
  - WebGL is certainly no exception and its downside comes in the form of complexity.

3d Javascript Libraries

- JavaScript 3D game engines are a hot topic right now with everyone building browser based 3D games using JavaScript, HTML5 and WebGL technology.
  - The best thing about browser based games is platform independence (run on iOS, Android, Windows or any other platform).
- Many libraries on offer:
  - Babylon.js
  - Three.js
  - Turbulenz
  - Famo.us
  - PlayCanvas.js
  - Blend4Web

Three.js vs Babylon

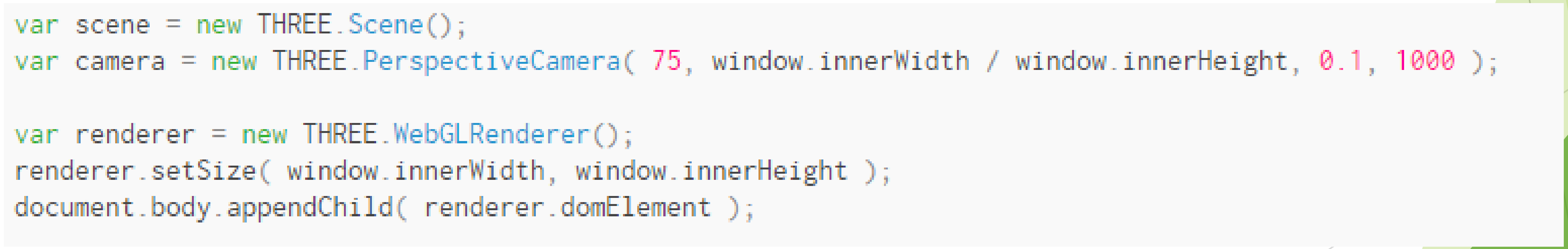
- The ever popular Three.js along with the newer Babylon.js offer web developers an abstract foundation for crafting feature rich WebGL creations ranging from animated logos to fully interactive 3D games.
- Three.js started in April of 2009 and was originally written in ActionScript before being translated to JavaScript. Having been created before the introduction of WebGL, Three.js has the unique convenience of a modular rendering interface allowing it to be used with SVG and HTML5's canvas element in addition to WebGL.
- Babylon.js, being the relative newcomer, arrived in 2013. Brought to you by Microsoft, Babylon.js was introduced alongside Internet Explorer 11's first official support for the WebGL API.
  - Despite originating from Redmond's labs, Babylon.js (as well as Three.js) maintains an open source license.

Three.js vs Babylon

- The main difference between the two lies in their intended use. It may be true that either of these frameworks can be shoehorned into creating the same 3D animation, it's important to know what is each's intended purpose.
- Three.js was created to take advantage of browser-based renderers for creating GPU enhanced 3D graphics and animations.
  - As such, this framework employs a very broad approach to web graphics without focusing on any single animation niche.
  - This flexible design makes Three.js a great tool for general purpose web animations like logos or modeling applications
- Where Three.js attempts to bring a wide range of animation features to the WebGL table, Babylon.js takes a more targeted approach.
  - Originally designed as a Silverlight game engine, Babylon.js is targeted at web based game development with features like collision detection and antialiasing. As previously stated, Babylon.js is also fully capable of general web graphics and animations

Three.js - Intro

- Creating the scene:
  - To be able to display anything with Three.js, we need three things:
    - A scene
    - A camera
    - A renderer - so we can render the scene with the camera.



```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight, 0.1, 1000 );

var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
```

Three.js - Setting up

- We now have the scene, our camera and the renderer.
- The scene is instantiated without any parameters:

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

- There are a few different cameras in Three.js. For this course, we'll focus on a PerspectiveCamera.

```
var camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight, 0.1, 1000 );
```

  - The first attribute is the field of view.
  - The second is the aspect ratio. You almost always want to use the width of the element divided by the height, or you'll get the same result as when you play old movies on a widescreen TV - the image looks squished.
  - The next two attributes are the near and far clipping plane. What that means, is that objects further away from the camera than the value of far or closer than near won't be rendered.

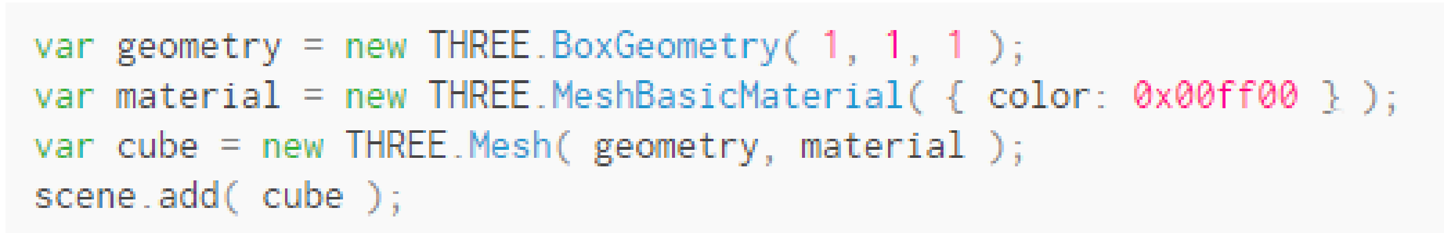
Three.js – Setting up

- Next up is the renderer. This is where the magic happens. In addition to the WebGLRenderer demonstrated here, Three.js comes with a few others, often used as fallbacks for older browsers that don't have WebGL support.
- In addition to creating the renderer instance, we also need to set the size at which we want it to render our app. It's a good idea to use the width and height of the area we want to fill with our app. For performance intensive apps, you can also give setSize smaller values, like window.innerWidth/2 and window.innerHeight/2, which will make the app render at half size.
- Finally, we add the renderer element to our HTML document. This is a <canvas> element the renderer uses to display the scene to us.

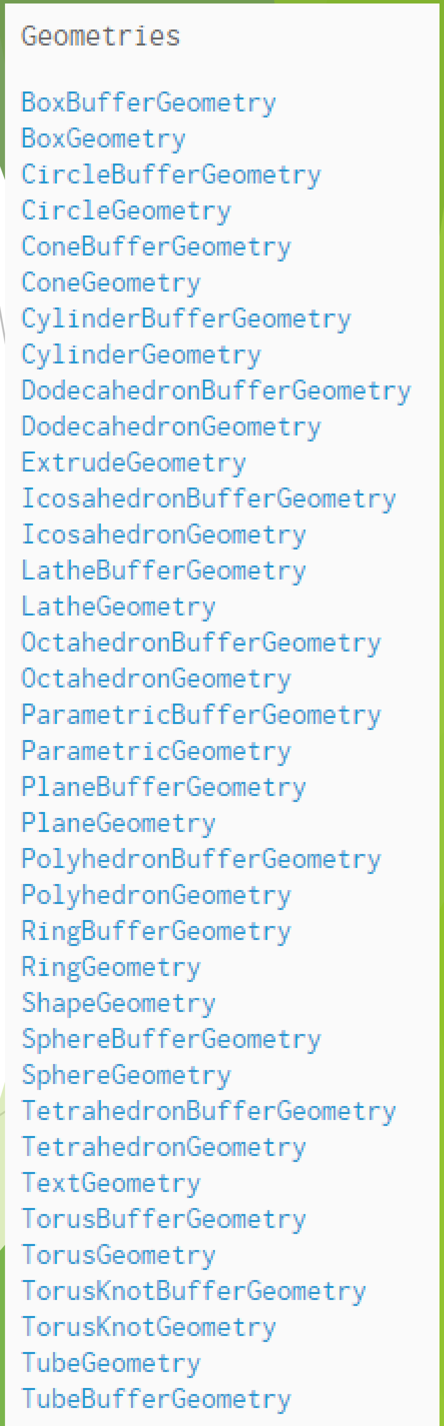
```
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
```

Three.js - Geometries

- Adding Geometries:
  - Three.js supports a wide range of geometries.
  - Can be created using a constructor and added to scene
- Example Box geometry



```
var geometry = new THREE.BoxGeometry( 1, 1, 1 );
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
var cube = new THREE.Mesh( geometry, material );
scene.add( cube );
```



Geometries

BoxBufferGeometry
BoxGeometry
CircleBufferGeometry
CircleGeometry
ConeBufferGeometry
ConeGeometry
CylinderBufferGeometry
CylinderGeometry
DodecahedronBufferGeometry
DodecahedronGeometry
ExtrudeGeometry
IcosahedronBufferGeometry
IcosahedronGeometry
LatheBufferGeometry
LatheGeometry
OctahedronBufferGeometry
OctahedronGeometry
ParametricBufferGeometry
ParametricGeometry
PlaneBufferGeometry
PlaneGeometry
PolyhedronBufferGeometry
PolyhedronGeometry
RingBufferGeometry
RingGeometry
ShapeGeometry
SphereBufferGeometry
SphereGeometry
TetrahedronBufferGeometry
TetrahedronGeometry
TextGeometry
TorusBufferGeometry
TorusGeometry
TorusKnotBufferGeometry
TorusKnotGeometry
TubeGeometry
TubeBufferGeometry

Three.js - Geometries

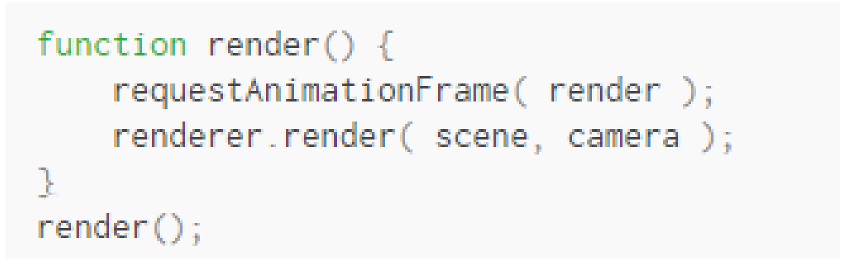
```
var geometry = new THREE.BoxGeometry( 1, 1, 1 );
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
var cube = new THREE.Mesh( geometry, material );
scene.add( cube );
```

- In addition to the geometry, we need a material to color it. Three.js comes with several materials, but we'll stick to the MeshBasicMaterial for now. All materials take an object of properties which will be applied to them. To keep things very simple, we only supply a colour attribute of 0x00ff00, which is green.
- The third thing we need is a Mesh. A mesh is an object that takes a geometry, and applies a material to it
- By default, when we call scene.add(), the thing we add will be added to the coordinates (0,0,0). This would cause both the camera and the cube to be inside each other. To avoid this, we simply move the camera out a bit.

```
camera.position.z = 5;
```

Rendering the scene

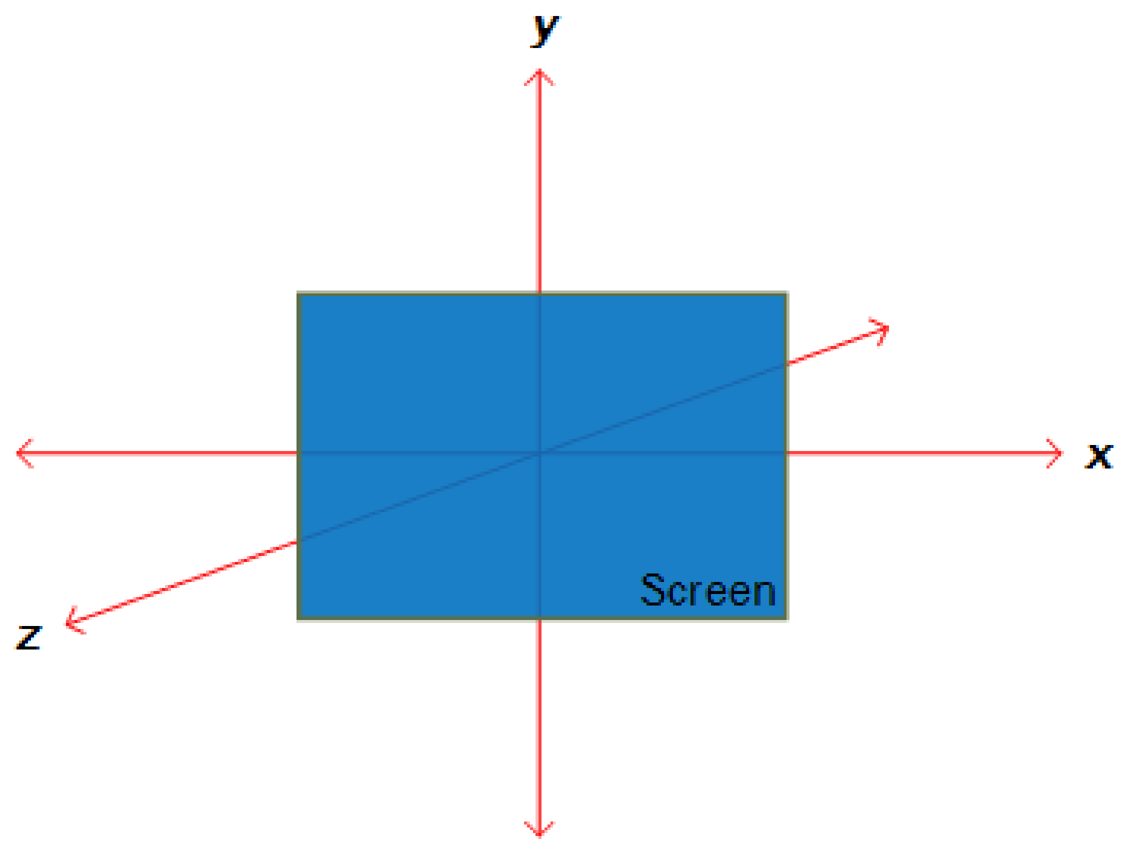
- If you copied the previous code into a HTML file, you wouldn't see anything.
  - This is because we're not actually rendering anything yet. For that, we need what's called a render loop.



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

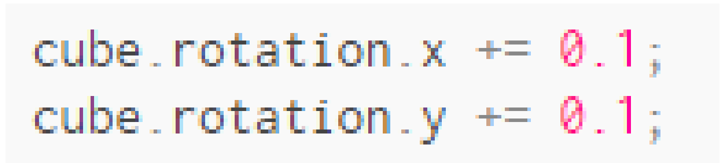
- This will create a loop that causes the renderer to draw the scene 60 times per second. If you're new to writing games in the browser, you might say "why don't we just create a setInterval?
- We could, but requestAnimationFrame has a number of advantages. Perhaps the most important one is that it pauses when the user navigates to another browser tab, hence not wasting their precious processing power and battery life.
- http://creativejs.com/resources/requestanimationframe/

Three.js coordinate system



Animation

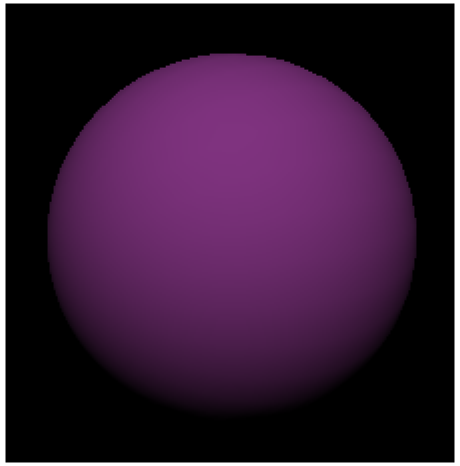
- Within requestAnimationFrame, can rotate the cube:

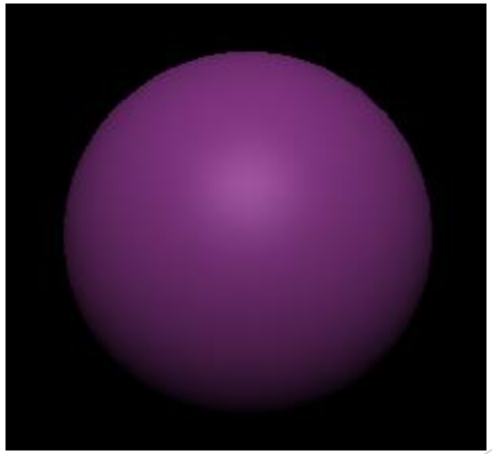


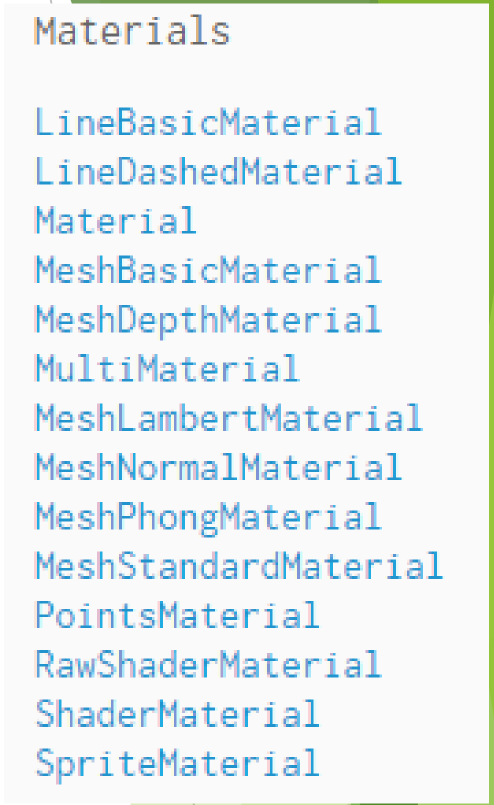
```
cube.rotation.x += 0.1;
cube.rotation.y += 0.1;
```

Materials

- We’ve seen MeshBasicMaterial – doesn’t support lighting or shadows
- Wide range of other materials E.g:
  - Lambert material defines an ideal "matte" or diffusely reflecting surface. The apparent brightness of a Lambertian surface to an observer is the same regardless of the observer's angle of view.
  - Phong shading



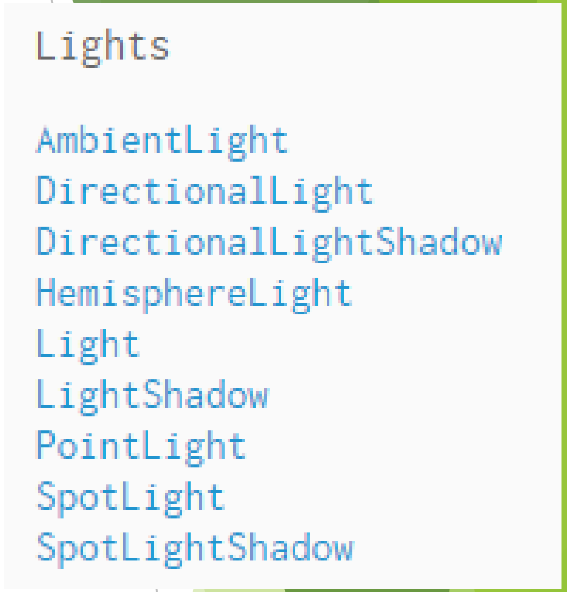




Materials

LineBasicMaterial
LineDashedMaterial
Material
MeshBasicMaterial
MeshDepthMaterial
MultiMaterial
MeshLambertMaterial
MeshNormalMaterial
MeshPhongMaterial
MeshStandardMaterial
PointsMaterial
RawShaderMaterial
ShaderMaterial
SpriteMaterial

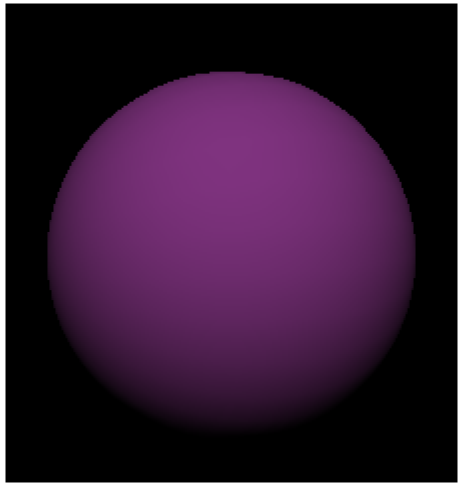
Light sources

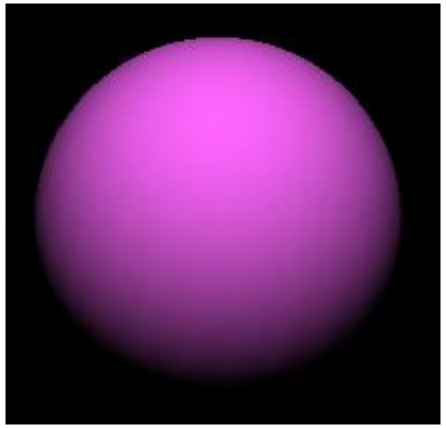


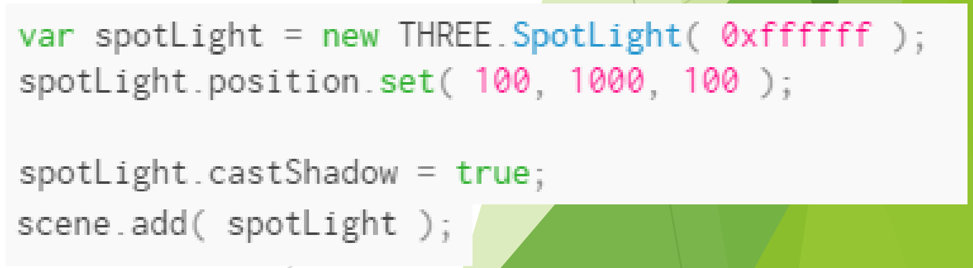
Lights

- AmbientLight
- DirectionalLight
- DirectionalLightShadow
- HemisphereLight
- Light
- LightShadow
- PointLight
- SpotLight
- SpotLightShadow

- When using more sophisticated materials, a light source needs to be added so the objects can be illuminated
- Wide variety of light sources:
  - E.g. SpotLight

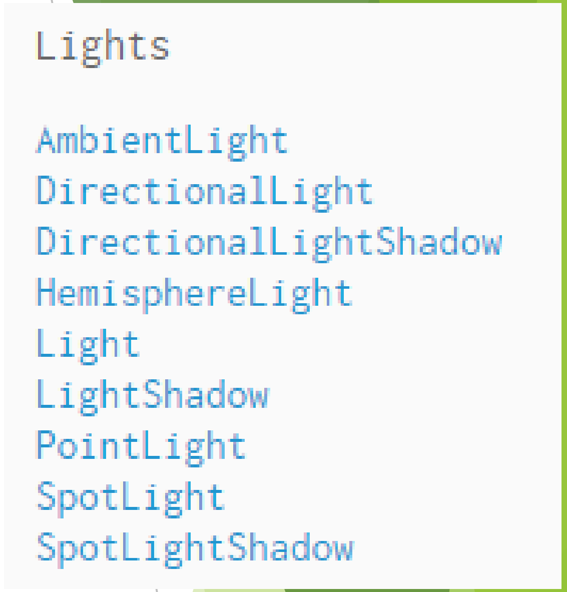






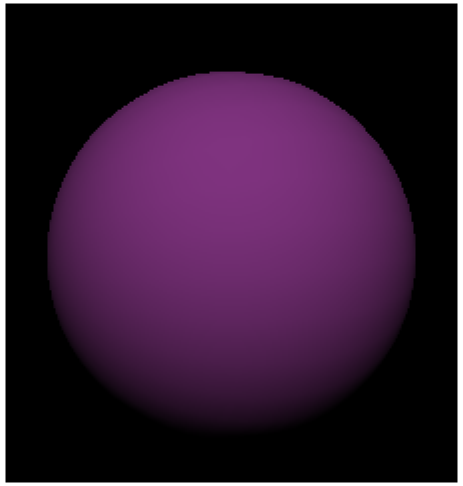
```
var spotLight = new THREE.SpotLight( 0xffffff );
spotLight.position.set( 100, 1000, 100 );

spotLight.castShadow = true;
scene.add( spotLight );
```

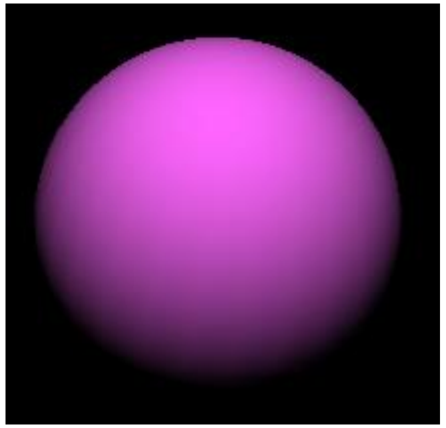
Light sources

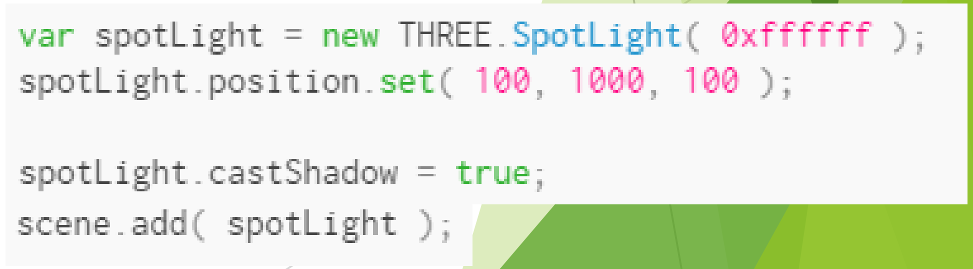

Lights

AmbientLight
DirectionalLight
DirectionalLightShadow
HemisphereLight
Light
LightShadow
PointLight
SpotLight
SpotLightShadow

- A directional light is when light rays are parallel. A bit like when you look at the sun rays. It mostly behaves like a light source very far from us.
- A spot light is when light rays seems to originate from a single point, and spreads outward in a coned direction






```
var spotLight = new THREE.SpotLight( 0xffffff );
spotLight.position.set( 100, 1000, 100 );

spotLight.castShadow = true;
scene.add( spotLight );
```

Shadows

```
renderer.shadowMap.enabled = true;
renderer.shadowMap.type = THREE.PCFSoftShadowMap;   // to antialias the shadow
document.body.appendChild(renderer.domElement);
```

- https://threejs.org/examples/#webgl_shadowmap_viewer
- Add shadow to renderer
- Enable objects to cast/receive shadows:
  - sphere.castShadow = true;
  - plane.receiveShadow = true;
- Enable light to cast shadow:
  - directionalLight.castShadow = true;
- Create camera helper to illustrate source of light
- Make sure to use correct (not basic) materials

```
var helper = new THREE.CameraHelper( directionalLight.shadow.camera );
scene.add( helper );
```