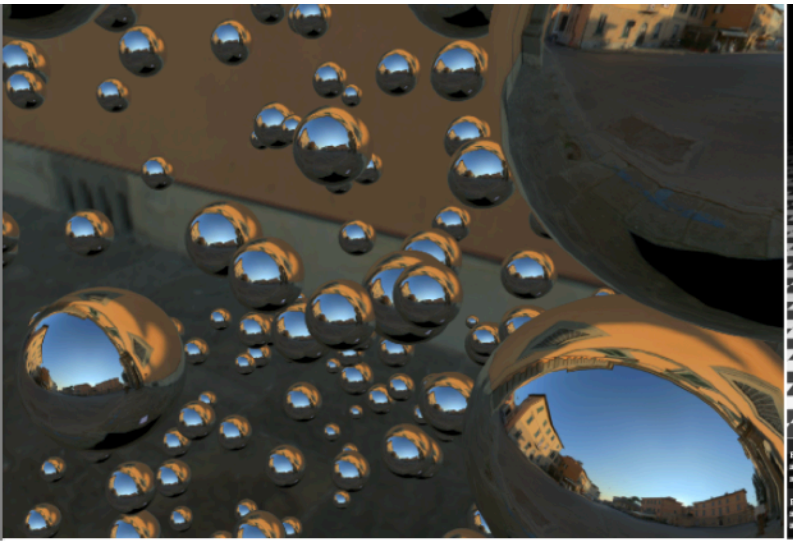
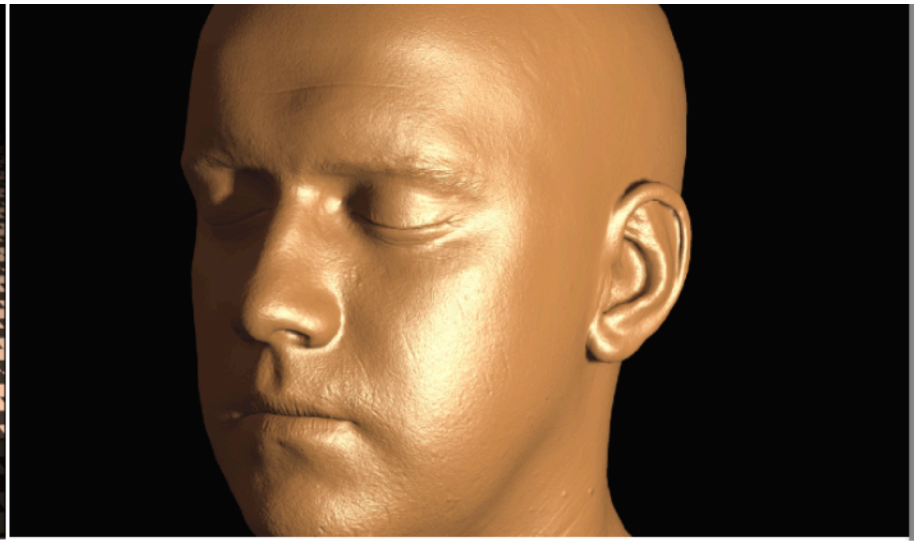
Visual Computing I:

Interactive Computer Graphics and Vision





Recap Lecture Computer Graphics

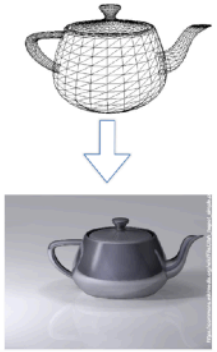
Stefanie Zollmann and Tobias Langlotz

Revision

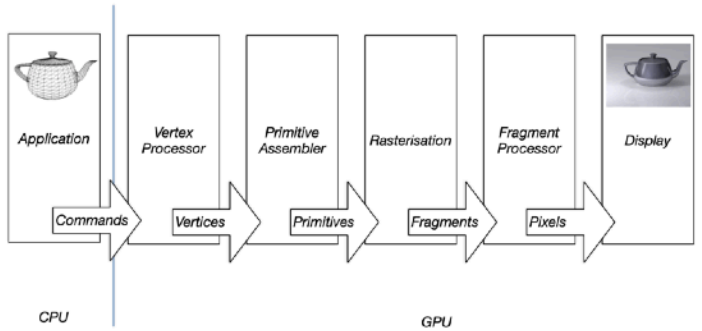
OpenGL Introduction

Graphics Pipeline

- Input
  - Geometric model
  - Vertices, normals, texture coordinates
  - Lighting/shading model
  - Light positions
  - View point and virtual camera configuration
- Output
- Colour (and depth) per pixel on a screen

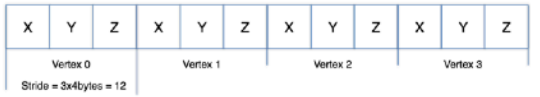


Graphics Pipeline

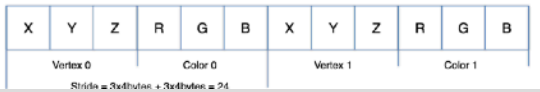


Stride

- Specifies the byte offset between consecutive generic vertex attributes (if stride equals 0 -> means tightly packed)
  - Tightly packed:

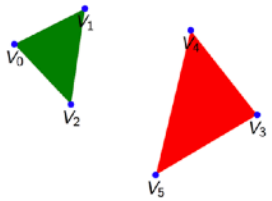


  - Interleaved:

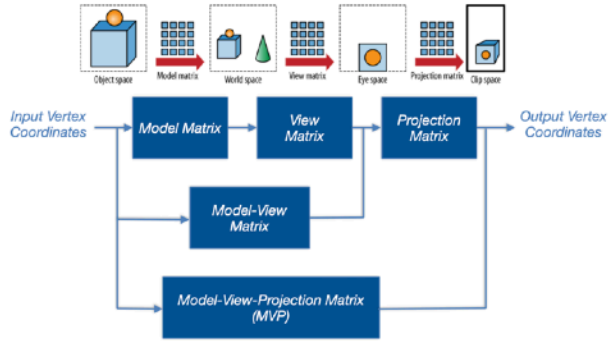


Primitive Types

- Points
- Lines
- Triangles
- Line strips
- Lineloops
- Triangle strips
- Triangle fans

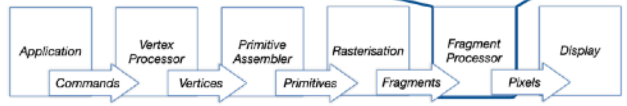


Vertex Shader: Transformations



Fragment Processing

- Output of the rasterisation stage are fragments
- Fragments are processed by a fragment shader
- Fragment shader have control over the color and depth values



Shader Programming

Shader

- Programs implementing the programmable parts of the pipeline
- Parts of pipeline
  - Vertex Shader
  - Fragment Shader
  - Others (e.g. Geometry Shader, Tessellation Shader)
- Name originates from small programs used to calculate the shading of a surface

Shader Usage

1. Create Shader

```
GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);

GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
```

2. Load shader program into object

```
char const * shaderSource = someCodeString.c_str();
glShaderSource(VertexShaderID, 1, &shaderSource , NULL);
```

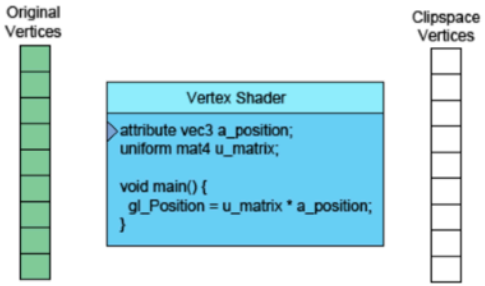
3. Compile Shader program

```
glCompileShader(VertexShaderID);
```

4. Use Shader program (bind)

```
glUseProgram(VertexShaderID);
```

Vertex Shader



Passing parameters to Shader

For passing uniforms to shaders we need to create the location (glGetUniformLocation) and specify the value (glProgramUniformXX)

Example:

- Add a colour value using a 4-dimensional vector:

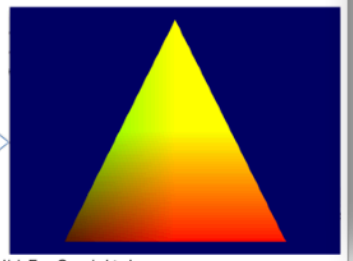
```
// add color parameter to shader
GLint colorID = glGetUniformLocation(programID, "colorValue");
glm::vec4 color = glm::vec4(1.0,1.0,1.0,1.0);
glProgramUniform4fv(programID,colorID,1, &color[0]);
```

Example code for ColourShader.cpp

Fragment Shader

Simple Fragment Shader outputting gl_FragCoord:

```
#version 330 core

// Ouput data
out vec3 color;

void main()
{
 // Output color = screen coord
        color =  vec3(gl_FragCoord.r/1024,
        gl_FragCoord.g/768, 0.0);

}
```



https://registry.khronos.org/OpenGL-Refpages/gl4/html/gl_FragCoord.xhtml

Datatypes GLSL

- Basic Types
  - int, uint, float, bool: scalar numeric and logical types
  - ivec2, ivec3, ivec4: integer vectors
  - uvec2, uvec3, uvec4: unsigned integer vectors
  - bvec2, bvec3, bvec4: boolean vectors
- Floating-Point Vectors
  - vec2, vec3, vec4: 2D/3D/4D float vectors
  - Commonly used for positions, colors, and directions
- Matrices
  - mat2, mat3, mat4: 2×2, 3×3, 4×4 float matrices
  - Mixed forms also exist: mat2x3, mat3x4, etc.
  - Used for transforms (model, view, projection)

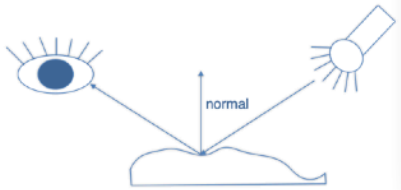
Illumination

Why Illumination?

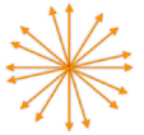
- Illumination is important for perception and understanding of 3D scenes
- Has visual cues for humans
- Provides information about
- Positioning of light sources
- Characteristics of light sources
  - Materials
  - Viewpoint

Local Illumination Model

- OpenGL cannot render full global illumination
- We need a simplified approximation
- Local illumination model
- Does not consider light reaching after bouncing off other objects
- Function of:
  - Viewer position
  - Light source
  - Surface material properties
  - Geometry



Light Sources

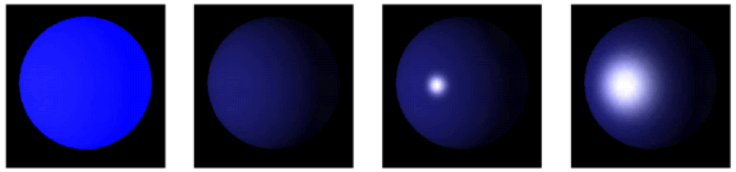


Point Light

Spot Light

Directional Light

Reflection model



Ambient Diffuse Specular Combined

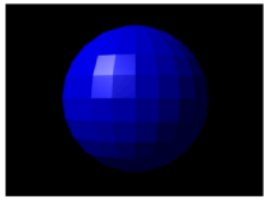
Phong Reflection Model

Illumination = $I_{ambient} = I_a k_a$ $I_{diffuse} = I_d k_d (\hat{N} \cdot \hat{L})$ $I_{specular} = I_s k_s (\hat{R} \cdot \hat{V})^{n_s}$

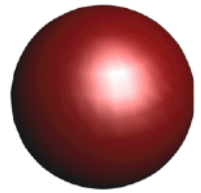
Illumination = $I_a k_a + I_d k_d (\hat{N} \cdot \hat{L}) + I_s k_s (\hat{R} \cdot \hat{V})^{n_s}$

Shading Models

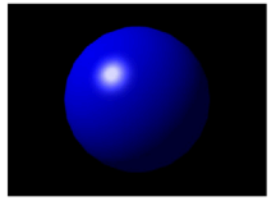
- Shading model determines on which shader stage and with which quality lighting for triangles is calculated



Flat Shading



Gouraud Shading



Phong Shading

Texture Mapping

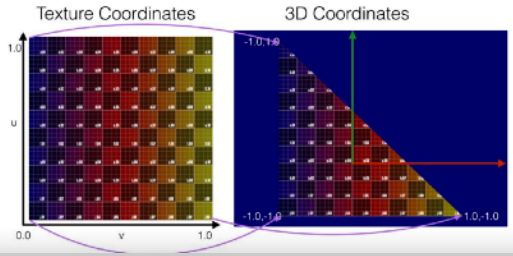
Texturing

Simulation of different material properties:

- Color
- Reflection
- Gloss
- Transparency
- Bumpiness

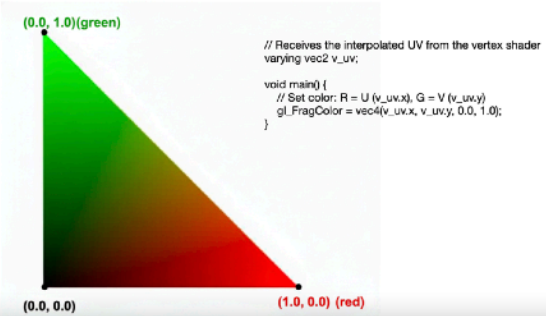
Texture Mapping

- Process of finding u,v coordinates for each vertex
- Texture is defined in a normalised space (e.g. for 2D textures: (u,v)∈[0...1,0...1]



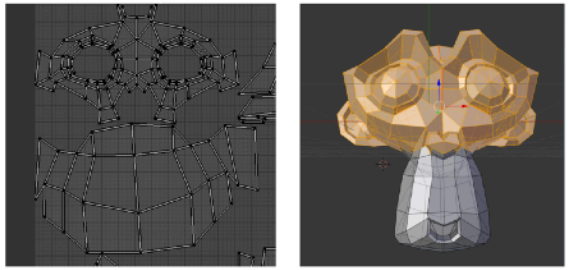
Texture Coordinates

- Texture coordinates linearly interpolated over primitive



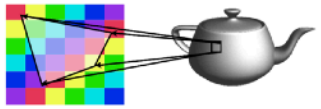
Parametrisation

- Manual mapping
  - Unwrap object (e.g. in Blender)

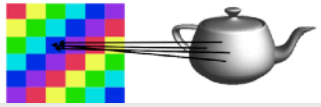


Challenges

- Undersampling: one fragment maps to an area covering many texel (Minification)



- Oversampling: many fragments map to an area contained by only one texel (Magnification)



Texture Filtering

- Interpolate texel value from neighbours

Nearest Neighbour (lower quality)

Linear interpolate the neighbours (better quality, slower)

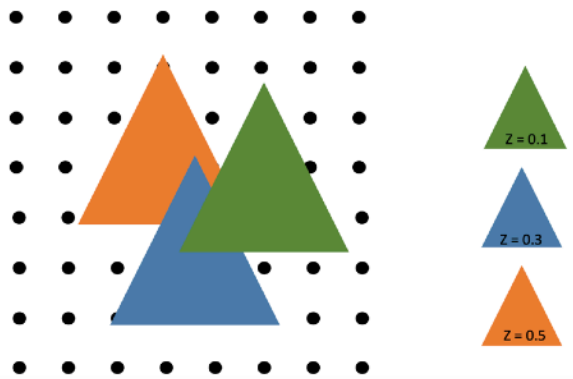
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

Advanced Rendering with Buffers

Buffers

- Buffer is simply a contiguous block of memory
- Different types:
  - Framebuffer: The conceptual container for all screen-related buffers
  - Colour Buffer: Visible output, handled with double buffering for smooth animation
  - Depth Buffer: Essential for correct 3D spatial relationships and occlusion
  - Stencil Buffer: Mask for advanced rendering effects
  - Data Buffers (Vertex Buffer Objects, Index Buffer Objects): Efficiently store and transfer geometric and uniform data to the GPU

Which triangle should be rendered?



Depth Buffer

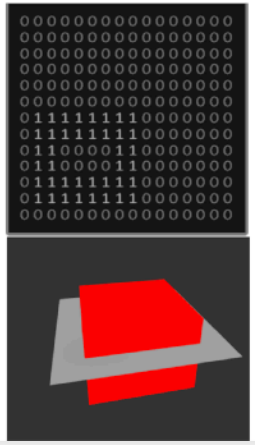
- 2D array storing a floating-point depth value for each pixel (often 0.0 to 1.0)
- In addition to RGB(A)
- Floating point value representing the depth at this pixel
- Called Depth Buffer or Z-Buffer
- Solves the occlusion problem -> correctly determines which objects are in front of others

Alpha Blending

- Blends the new fragment's colour with the colour already in the colour buffer, based on its alpha
- Use Case: Glass, water, smoke, UI elements
- How (OpenGL Setup): Blending must be enabled and blend function needs to be setup

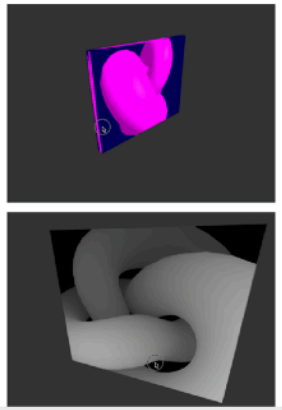
Stencil Testing

- 2D array storing integer values (e.g., 8-bit, 0-255) for each pixel
- Purpose:
  - Acts as a per-pixel mask or filter
  - It allows to control when and where rendering operations can occur
- How it Works (Stencil Test):
- "Draw" a pattern into the stencil buffer first.
- Configure OpenGL to only draw subsequent objects where the stencil buffer meets certain conditions (e.g., "stencil value equals 1")



Render to Texture

- Concept:
  - Instead of drawing just once to the screen, scene is rendered in multiple stages
  - Key is to render the first pass into an off-screen Framebuffer Object (FBO), which saves the result as a texture
- Pipeline:
  - Pass 1 (Render-to-Texture): Render scene (e.g., all 3D objects) into the FBO -> colour buffer is now a texture
  - Pass 2 (Render-to-Screen):
    - Draw a simple full-screen quad
    - Apply Texture: Apply the texture you just created in Pass 1 to this quad



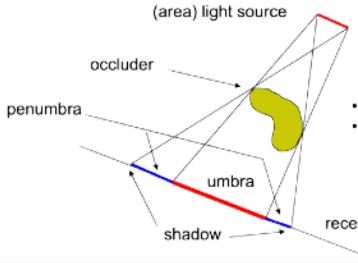
Shadow Mapping

Shadows

- Important depth cue
- Spatial Relationship between objects
- Realism
- Provides information about scene lighting

Terminology

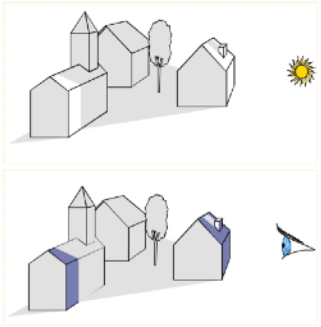
- Point lights have hard edges
- Area lights have soft edges



- **umbra** – fully shadowed region
- **penumbra** – partially shadowed region

Shadow/View Duality

- A point is in shadow if it not visible from the perspective of the light source
- Use the view from the light source to compute shadows



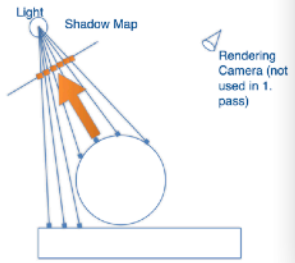
Shadow Mapping

- Use shadow/view duality
- Two rendering passes
  - 1st Pass: Rendering shadow map representing the depth from light source
  - 2nd Pass: Render final image from camera view and check shadow map to see if points are in shadow

First Pass: Render Shadow Map

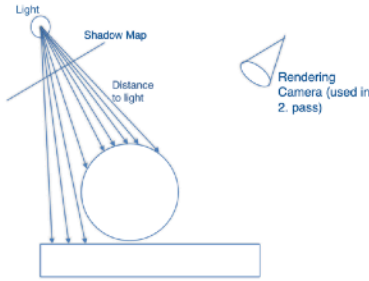
Create a map of depth values as seen from the light's point of view

- Geometry is rendered into a depth buffer from the point of view of the light
- Transform the geometry into light-view space



Second Pass: Render Scene

- Render final output from camera view and check shadow map to see if points are in shadow
- Compare distance between point and light source and value in shadow map
- If distance larger than value in shadow map point is in shadow



Advanced Rendering: Deferred Shading

Deferred Shading

- "Defer" (delay) all the expensive lighting work until second rendering pass
- Changes our pipeline:
  - Pass 1 (Geometry Pass): Render all objects, but instead of a final colour, we output their data (Position, Normal, Colour) into a set of textures
  - Pass 2 (Lighting Pass): Use those textures to compute all the lighting at once -> Decoupling lighting complexity from geometry complexity

Geometry Pass/G-Buffer

- Geometry Pass does not render to the screen:
  - Renders to a set of textures called the G-Buffer
- Use Multiple Render Targets (MRTs). In OpenGL: glDrawBuffers
- What's in the G-Buffer?
  - Position (World-space)
  - Normals (World-space)
  - Albedo (Diffuse Colour)
  - Material Properties (e.g., Specular, Roughness, Metallic)

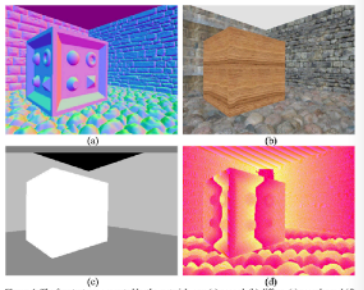


Figure 4: The four textures generated by the material pass: (a) normal, (b) diffuse, (c) specular and (d) encoded depth as color.

Policarpo et al.

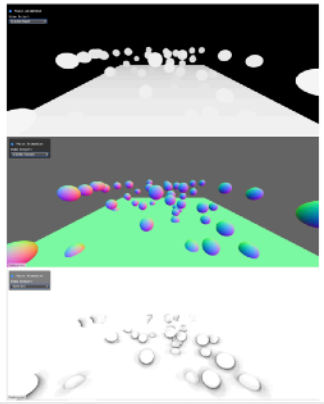
Ambient Occlusion

- Coarse approximation of global illumination
- Often referred to as "sky light"
- Global method: Illumination at each point is a function of other geometry in the scene
- Soft, realistic shadows that appear in corners, creases, and where objects get close to each other

Nvidia: Screen Space Ambient Occlusion

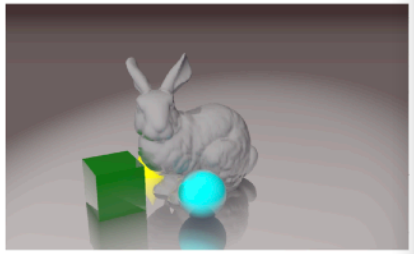
Screen-Space Ambient Occlusion (SSAO)

- Post-processing effect that approximates ambient occlusion
- Runs as another pass after the G-Pass
- Render a full-screen quad
- For every pixel, the fragment shader:
  - Reads its Position and Normal from G-Buffer
  - Takes several random samples from the Position texture in a small radius around the pixel
  - Checks how many of those samples are behind the original pixel's surface (i.e., "occluding" it).
- The more samples that are occluded, the darker that pixel becomes
- This creates a grayscale "occlusion map," which we then multiply with our final lit scene to apply the soft shadows.
- This is fast: We do it all in screen-space using the G-Buffer data we already have



Screen-Space Reflections (SSR)

- Use the G-Buffer to find reflections
- How it Works:
- For each pixel on a reflective surface (like a wet floor or a mirror)
  - Calculate the reflection vector (just like in Blinn-Phong)
  - Instead of sampling a cubemap, we "march" a ray from that pixel in the reflection direction in screen-space
  - Use the Position/Depth Buffer to see if that ray "hits" any other pixel on the screen
  - If it does, we sample the colour of the pixel it hit and use that as the reflection
- Advantage: Gives dynamic, real-time reflections of other objects in the scene for a very low cost
- Disadvantage: It can only reflect what is already visible on the screen -> If reflected object is off-screen or hidden, the reflection disappears



The Final Polish: HDR, Tone Mapping & Bloom

- How to create cinematic look?
- Problem (HDR):
  - Real world is "High Dynamic Range" (HDR) -> But displays can only show "Low Dynamic Range" (LDR)
- Solution (Tone Mapping):
  - Render entire scene to an HDR texture (framebuffer with a format like GL_RGB16F)
  - Then, draw a final full-screen quad -> Fragment shader for this quad runs a Tone Mapping algorithm
  - Algorithm intelligently "squashes" the HDR values back into the 0-1 range, preserving detail in both bright and dark areas (e.g., you can still see the bright sky and the dark shadows)

Standard (Low) Dynamic Range

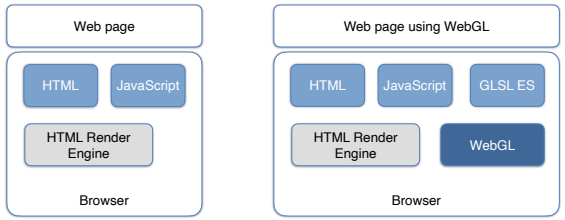
High Dynamic Range

WebGL/Three.JS

WebGL

- Inside an HTML <canvas> element.
- Development is likely to involve HTML, CSS, JavaScript, GLSL, in addition to WebGL's OpenGL-style naming.

WebGL Application Structure



WebGL Shaders

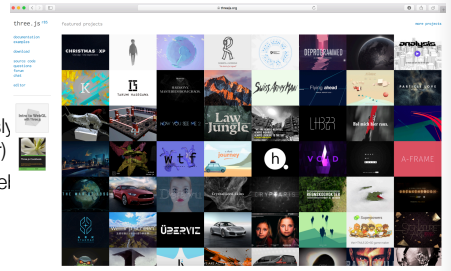
```
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '}\n';

// Fragment shader program
var FSHADER_SOURCE =
  'void main() {\n' +
  '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
  '}\n';
```

JAVASCRIPT: HELLOTRIANGLE.JS

Example from WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL by Matsuda and Lea

THREE.JS

- First released April 2010
- 3D Javascript Library
- Under MIT license
- Uses WebGL for rendering (previously also CanvasRenderer, SVGRenderer)
- Runs in all browsers that support Wel
- Website: threejs.org
- Library is in single javascript file



Interactive: Object Picking

- Raycaster computes the intersection of a ray and a set of scene objects
- Raycaster used for mouse picking
- Given a mouse coordinate computes which objects in the 3d space are hit
- Returns an array of intersected objects

```
var raycaster = new THREE.Raycaster();
raycaster.setFromCamera( mouse, camera );
var intersects = raycaster.intersectObjects( scene.children );
```


WEBXR

- API is for accessing virtual reality (VR) and augmented reality (AR) devices on the Web
- Includes sensors and head-mounted displays
- Successor of WebVR API
- Often used in combination with WebGL frameworks

	Headset Devices	Handheld Device e.g. Phone
VR	VR Devices, previously handled by WebVR	Magic Window Behaviour
AR	Mixed Reality Headsets	Phone AR

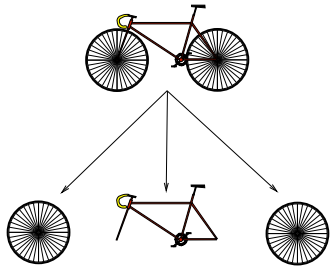
https://github.com/immersive-web/webxr

https://immersive-web.github.io/webxr/

Scene Graphs/Game Engines

Scene Graph

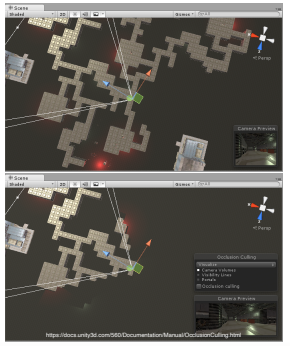
- Data structure
- Represented by a tree
  - Directed acyclic graph
  - Root on top
  - Leaves representing geometries
  - Objects are placed relative to each other
- Represents logical and spatial relationship between objects in a scene



15

Advantages

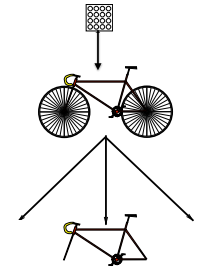
- Reuse geometry
- Reuse materials
- Optimisation based on view settings
- Optimisation based on occlusions
- Optimisation based on distance



16

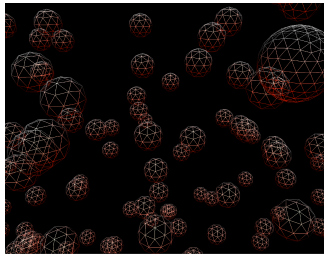
Scene Graph: Transformations

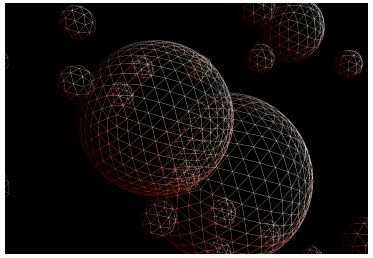
- Object has a local transformation relative to its parent
- If bike root is transformed the frame and all child nodes would be transformed as well
- Root of the graph corresponds to “world coordinates”
- Each geometry is defined in local coordinates



20

Level of Detail



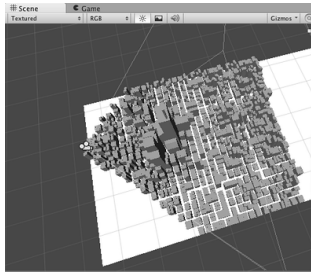


- Rendering Optimisation
- Change level of detail depending on distance to the camera

28

Culling

- Scene graph structure supports efficient culling
  - Bounding volume is calculated at each node in the scene graph
  - Includes geometry of node and child nodes
  - When traversing the scene graph, bounding volumes are evaluated against view frustum
  - If bounding volume does not intersect with view frustum -> culled
  - Culled nodes and their children are not sent to the graphic card

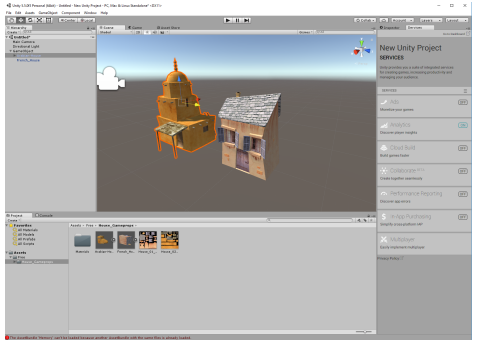


https://docs.unity3d.com/460/Documentation/Manual/OcclusionCulling.html

33

Render/Game Engines

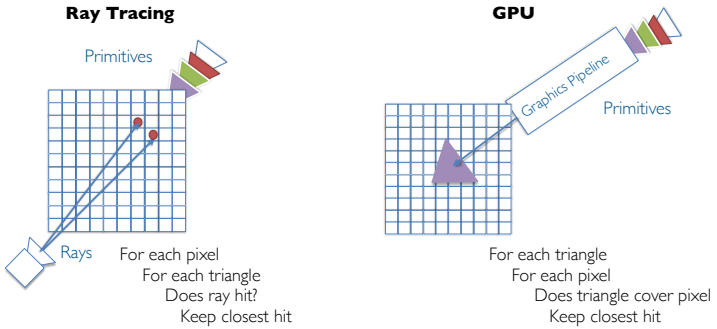
- Unity
- Unreal
- CryEngine
- Etc.



45

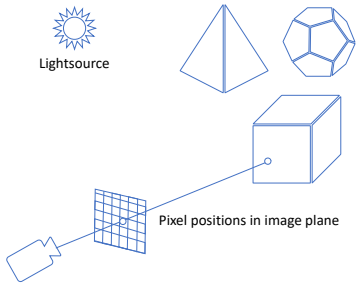
Raytracing

Raytracing vs Rasterised Graphics



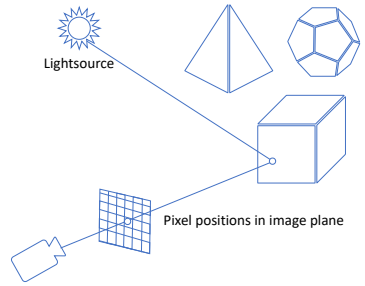
Raytracing Process

- Create an image plane and viewpoint
- For each pixel trace a ‘ray’ from the eye through a corresponding point in the image plane
- For each ray, return the colour of the object at the hit point (closest to the camera)



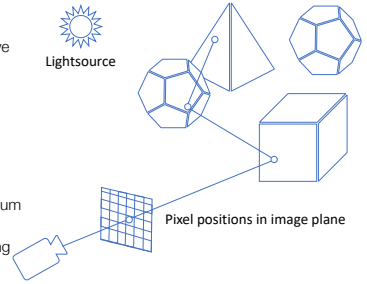
Raytracing Process - Shadows

- Shadows:
  - Ray is casted towards light source
  - If an object is located between hit point (surface point) and light source then the hit point is in shadow
  - If not, there is no shadow
  - Point lights: hard binary shadows
  - Area lights: sample positions on the emitter to produce soft penumbrae



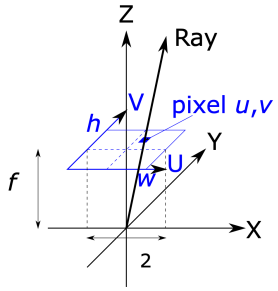
Raytracing Process - Reflection

- Method:
  - If ray is reflected on the surface (reflective material), a new (secondary) ray is created and cast in the reflective direction
  - Compute reflected direction using the law of reflection $\mathbf{r} = \mathbf{v} - 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n}$
  - Spawn secondary ray from hit point
  - Modulate the reflected contribution by the material’s reflectivity ρ_r
  - Trace the reflected ray recursively, respecting a maximum depth or probabilistic termination
  - Finally: Add returned reflected radiance to local shading $L = L_{\text{local}} + \rho_r L_{\text{reflected}}$



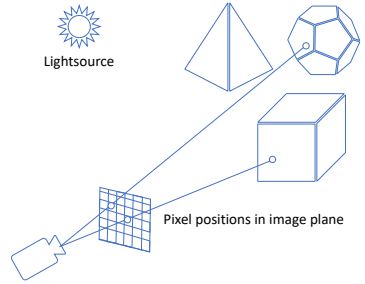
Primary Ray and Camera Model

- Primary Ray
  - Defined by:
    - Origin at the camera center
    - Direction through a pixel on the image plane
    - Intersection tests determine visible surfaces
- Camera models:
  - Pinhole (standard for educational ray tracers)
  - Thin-lens (focus, depth of field)



Geometric Intersection Tests

- Ray–sphere: closed-form quadratic equation
- Ray–box: testing each side
- Intersection cost dominates rendering -> acceleration required



Q&A

The end!