WHAT IS RENDERING?

Generating image from a 3D scene

Let's think HOW.

SCENE
• A coordinate frame
• 3D objects
• Their materials
• Lights
• Cameras

RENDERING

FRAME BUFFER
• Portion of RAM on videocard (GPU)
• What we see on the screen
• Rendering destination

SCREEN
• Displays what's in frame buffer
• Terminology:
  • Pixel: basic element on device
  • Resolution: number of rows & columns in device
    • Measured in:
      • Absolute units (in, cm)
      • Density units (dpi, ppi)

Scene

Coordinate Frame
Materials
Lights
Cameras

Framebuffer

final image

Delay: 2016-02-01

Slide credits:
Mikhail Bessmeltsev
**SINGLE OBJECT**

- How to describe a single piece of geometry?
- So far geometry has been constructed for you.

**SHAPES: TRIANGLE MESHES**

- Triangle = 3 vertices
- Mesh = [vertices, triangles]
- Example

**SCENE**

- How to describe a scene?
- Local Transformations

**SKETCH OF A RENDERING PIPELINE**

- Scene
  - Coordinate frame
  - 3D models
  - Properties (color, material)
  - Lights
  - Camera
- Camera View
  - 3D position of shapes
  - Normals
- Image
  - Shape pixels
  - Their color
  - Which pixel is visible

**OPENGL/WEBGL**

- Open Graphics Library
- One of the most popular libraries for 2D/3D rendering
- A software interface to communicate with graphics hardware
- Cross-language API

**OPENGL RENDERING PIPELINE**

- Vertices and indices
- Vertex Shader
- Fragment Shader
- Pixel Shader
- Color Operations
- Framebuffer
- Final Image
**OpenGL Rendering Pipeline**

- **Scene**
- **Camera Coords**
- **Device Coords**

  - Vertices are stored in vertex buffer
  - Each one is processed by vertex shader
  - Outputs 2D position
  - May compute per-vertex variables (normal, etc.)

---

**Geometry**

- Triangle meshes
  - Set of vertices
  - Triangle defines as [vertex_index1, vertex_index2, vertex_index3]

---

**Vertex Shader**

- Outputs 2D position
- May compute per-vertex variables (normal, etc.)

---

**Three.js**

- High-level library for Javascript
- Uses WebGL for rendering
  - Has Scene, Mesh, Camera objects
  - Scene is hierarchical
  - Mesh has geometry and material properties
  - Camera is used for rendering

---

**GLSL**

- OpenGL shading language
- Used for Fragment and Vertex shaders
  - Lots of useful stuff:
    - vec3, vec4, dvec4, mat4, sampler2D
    - mat*vec, mat*mat
    - Reflect, refract
    - vec3(v3, v4, st)

---

**Open GL Rendering Pipeline**

- Javascript + Three.js
  - Uses WebGL for rendering
  - Has Scene, Mesh, Camera objects
  - Scene is hierarchical
  - Mesh has geometry and material properties
  - Camera is used for rendering
**VERTEX SHADER**

- VS is run for each vertex SEPARATELY
- By default doesn’t know connectivity
- Input: vertex coordinates in Object Coordinate System
- Its main goal is to set \texttt{gl\_Position}

Object coordinates \rightarrow WORLD coordinates \rightarrow VIEW coordinates

**OPENGL RENDERING PIPELINE**

**CAMERA VIEW**

**RASTERIZATION**
**Rasterization**

- Rasterization

**OpenGL Rendering Pipeline**

- Javascript + Three.js
- GLSL
- VertexShader
- Interpolation
- FragmentShader
- Post-Interpolation
- Framebuffer

**Fragment Shader**

- Fragment = data for drawing a pixel
- Has gl_FragCoord = 2D window coords
- May set color!

- Common Tasks
  - texture mapping
  - per-pixel lighting and shading
  - Synonymous with Pixel Shader
MINIMAL VERTEX SHADER

```c
void main()
{
// Transforming The Vertex
gl_Position = modelViewMatrix * position;
}
```

MINIMAL FRAGMENT SHADER

```c
void main()
{
// Setting Each Pixel To Red
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

CONCEPTS

- **uniform**: same for all vertices
- **varying**: computed per vertex, automatically interpolated for fragments
- **attribute**: some values per vertex, available only in Vertex Shader

FRAGMENT SHADER

```c
uniform float uVertexScale;
attribute vec3 vColor;
varying vec3 fColor;

void main()
{
    gl_Position = vec4(position.x * uVertexScale, position.y, 0.0, 1.0);
    fColor = vColor;
}
```

ATTACHING SHADERS

```javascript
var remoteMaterial = new THREE.ShaderMaterial(
    {
      uniforms: {
        remotePosition: remotePosition,
      },
    });
//here goes loading shader files into shaders[]
remoteMaterial.vertexShader = shaders[0]('remote.vs.glsl');
remoteMaterial.fragmentShader = shaders[1]('remote.fs.glsl');
var remoteGeometry = new THREE.SphereGeometry(1, 32, 32);
var remote = new THREE.Mesh(remoteGeometry, remoteMaterial);
scene.add(remote);
```

PIPELINE: MORE DETAILS

```javascript
var remoteMaterial = new THREE.ShaderMaterial(
    {
      uniforms: {
        remotePosition: remotePosition,
      },
    });
//here goes loading shader files into shaders[]
remoteMaterial.vertexShader = shaders[0]('remote.vs.glsl');
remoteMaterial.fragmentShader = shaders[1]('remote.fs.glsl');
var remoteGeometry = new THREE.SphereGeometry(1, 32, 32);
var remote = new THREE.Mesh(remoteGeometry, remoteMaterial);
scene.add(remote);
```
**MODELING AND VIEWING TRANSFORMATIONS**

- Placing objects - Modeling transformations
  - Map points from object coordinate system to world coordinate system
- Looking from the camera - Viewing transformation
  - Map points from world coordinate system to camera (or eye) coordinate system

---

**MODELING & VIEWING TRANSFORMATIONS**

- **Modeling Transformations:**
  - Object Placement
  - E.g. Scaling (non-uniform)

- **Viewing Transformation:**
  - Looking from a Camera

---

**MODELING & VIEWING TRANSFORMATIONS**

- Linear transformations
  - Rotations, scaling, shearing
  - Can be expressed as 3x3 matrix
  - E.g. scaling (non-uniform):
    
    \[
    \begin{bmatrix}
    x' \\
    y' \\
    z'
    \end{bmatrix} =
    \begin{bmatrix}
    2 & 0 & 0 \\
    0 & 3 & 0 \\
    0 & 0 & 1
    \end{bmatrix}
    \begin{bmatrix}
    x \\
    y \\
    z
    \end{bmatrix}
    \]

- Other transformations (not handled by rendering pipeline):
  - Features: deformation
**MODELING & VIEWING TRANSFORMATION**

- Affine transformations
  - Linear transformations + translations
  - Can be expressed as 3x3 matrix + 3 vector
  - E.g. scale + translation
- Another representation: 4x4 homogeneous matrix

**MATRICES**

- Object coordinates -> World coordinates
  - Model Matrix
    - One per object
- World coordinates -> Camera coordinates
  - View Matrix
    - One per camera

**PERSPECTIVE TRANSFORMATION**

- Purpose: Project 3D geometry to 2D image plane
  - Simulate a camera
- Camera model:
  - Pinhole camera (single viewpoint)
  - More complex camera models exist, but are less common in CG

**PERSPECTIVE PROJECTION**

- In computer graphics:
  - Image plane conceptually in front of center of projection
  - Perspective transformation is one of projective transformations
  - Linear & affine transformations also belong to this class
  - All projective transformations can be expressed as 4x4 matrix operations

**CLIPPING**

- Removing invisible geometry
  - Geometry outside viewing frustum
  - Plus too far or too near one
- Optimization

**PIPELINE: MORE DETAILS**

- Model/View Transformation
  - Vertices transformed
- Vertex/Fragment Shaders
  - Per-vertex functions: lighting, shading
SCAN CONVERSION/RASTERIZATION

- Convert continuous 2D geometry to discrete
- Raster display - discrete grid of elements
- Terminology
  - Screen Space: Discrete 2D Cartesian coordinate system of the screen pixels

SCAN CONVERSION

- Problem:
  - Line is infinitely thin, but image has finite resolution
  - Results in steps rather than a smooth line
  - Jaggies
  - One of the fundamental problems in computer graphics

COLOR INTERPOLATION

- Example:

PIPELINE: MORE DETAILS

TEXTURING
TEXTURING

• Images:
  • Computing 3D/2D map (low distortion)
  • How to map pixel from texture (texels) to screen pixels
  • Texture can appear widely deformed rendering
• Shrinkage / stretch and distortion
• Filtering of textures
• Preventing aliasing (anti-aliasing)

TEXTURE MAPPING

• PIPELINE: MORE DETAILS
  - Preprocess
  - Transformations
  - Vertex/fragments
  - Pixel operations
  - Framebuffer

DISPLACEMENT MAPPING

LIGHTING

• PIPELINE: MORE DETAILS
  - Preprocess
  - Transformations
  - Vertex/fragments
  - Pixel operations
  - Framebuffer

COMPLEX LIGHTING AND SHADING

• PIPELINE: MORE DETAILS
  - Preprocess
  - Transformations
  - Vertex/fragments
  - Pixel operations
  - Framebuffer

WITHOUT HIDDEN LINE REMOVAL
**Hidden Line Removal**

- Remove invisible geometry
- Parts that are hidden behind other geometry

**Possible Implementations:**
- Pixel level decision
- Depth buffer
- Object space decision
- E.g. intersection order in ray tracing

**Hidden Surface Removal**

- Depth test / hidden surface removal

**Depth Test / Hidden Surface Removal Pipeline:**

- Blending:
  - Fragments -> Pixels
  - Draw from farthest to nearest
  - No blending - replace previous color
  - Blending: combine new & old values with some arithmetic operations
  - Frame Buffer: video memory on graphics board that holds resulting image & used to display it

**Blending Pipeline:**

- VertexShader
- ModelViewTransform
- ViewProjTransform

- FragmentShader
- Texturing
- Lighting/Shading

**Reflection/Shadows**

- Pipeline: More Details

- Blending:
  - Fragments -> Pixels
  - Draw from farthest to nearest
  - No blending - replace previous color
  - Blending: combine new & old values with some arithmetic operations
  - Frame Buffer: video memory on graphics board that holds resulting image & used to display it