CPSC 427
Video Game Programming

Rendering Pipeline and OpenGL

Helge Rhodin
Recap: Rendering – Rasterization

Approximate objects with triangles

1. Project each corner/vertex
   • projection of triangle stays a triangle
     \[
     \begin{bmatrix}
     u \\
     v 
     \end{bmatrix} = \frac{1}{z} \begin{bmatrix}
     x \\
     y
     \end{bmatrix}
     \]
   • $O(n)$ for $n$ vertices

2. Fill pixels enclosed by triangle
   • e.g., scan-line algorithm
Recap: Rasterizing a Triangle

- **Determine pixels enclosed by the triangle**
- **Interpolate vertex properties linearly**

Vertices

Fragments
(for every pixel; color or attributes to compute color: texture coordinate, direction, …)
Self study: Interpolation with barycentric coordinates

- linear combination of vertex properties
  - e.g., color, texture coordinate, surface normal/direction

- weights are proportional to the areas spanned by the sides to query point $P$
Graphics processing unit (GPU)

Specialized hardware designed for rendering
- highly parallel architecture
- dedicated instructions
- hardware pipeline (parts are not programmable)

Proved useful for high-performance computing
- machine learning
- bitcoin mining
- …
OpenGL Rendering Pipeline

**Input:**
- 3D vertex position
- Optional vertex attributes: color, texture coordinates,…

**Output:**
- **Frame Buffer**: GPU video memory, holds image for display
- RGBA pixel color (Red, Green, Blue, Alpha / opacity)
Programming languages

Traditionally, the entire pipeline was fixed (until ~2004)

• vertex and fragment shaders now programmable with GLSL

C++
OpenGL

Vertices and attributes

GLSL

Vertex Shader → Vertex Post-Processing → Rasterization

Fragment Shader → Per-Sample Operations

still fixed pipeline

GLSL

Framebuffer
OpenGL Rendering Pipeline (simplified)

1. **Vertex shader**: geometric transformations
2. **Fragment shader**: pixel-wise color computation

World Coordinates → Pixel-wise attributes → RGBA image

- Vertices and attributes
- Vertex Shader
- Fragment Shader
- Framebuffer

Shader: Programmable functions to define object appearance locally (vertex wise or fragment wise)
Vertex shader examples

Object motion & transformation
- translation
- rotation
- scaling

Projection
- Orthographic
  - *simple, without perspective effects*
- Perspective
  - *pinhole projection model*
The OpenGL Shading Language (GLSL)

- Syntax similar to the C programming language
- Build-in vector operations
  - functionality as the GLM library our assignment template uses

```c
void main()
{
    // Transforming The Vertex
    vec3 out_pos = projection * transform * vec3(in_pos.xy, 1.0);
    gl_Position = vec4(out_pos.xy, in_pos.z, 1.0);
}
```

- `vec3` and `vec4` floats
  - x and y coordinates of a vec2, vec3 or vec4
  - vector of 3 (vec3) and 4 (vec4) floats
  - float (32 bit)
From local object to camera coordinates

- **World coordinates**
  - Object coordinates (local)
  - transform

- **World coordinates**
  - Camera coordinates
  - projection

- **Camera coordinates**
  - Object coordinates
  - projection * transform
Camera types

Perspective projection (P)  Orthographic projection (O)

full 3D

used in our 2D games!
Fragment shader examples

- simulates materials and lights
- can read from textures

Diffuse
Specular
Directional
Fragment shader overview

(e.g., light direction)

(e.g., surface direction/normal)
GLSL fragment shader examples

**Minimal:**

```glsl
out vec4 out_color;

void main()
{
    // Setting Each Pixel To ???
    out_color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Specify color output

Red, Green, Blue, Alpha
Shader demo

• go to https://www.shadertoy.com/view/ttKcWR

• lets play together
Coordinate transformations

World Coordinates  Camera Coordinates  Window Coordinates  Pixel-wise attributes*

Vertices and attributes → Vertex Shader → Vertex Post-Processing → Rasterization → Fragment Shader → Per-Sample Operations → Framebuffer

*usually multiple fragments for every pixel (fragment != pixel)
OpenGL Rendering Pipeline (detailed)

- **Vertices and attributes**
  - Vertex Shader:
    - Modelview transform
    - Per-vertex attributes
  - Rasterization:
    - Scan conversion
    - Interpolation
  - Per-Sample Operations:
    - Depth test
    - Blending

- **Vertex Post-Processing**
  - Viewport transform
  - Clipping

- **Fragment Shader**
  - Texturing/...
  - Lighting/shading

→ **Framebuffer**
The OpenGL library

• Low-level graphics API
• C Interface accessed from C++

• How to
  • set shaders
  • set shader inputs
  • start rendering
Loading and compiling shaders

CREATING SHADER OBJECTS

```c
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, sourceCode, sourceCodeLength);
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, sourceCode, sourceCodeLength);
```

COMPILING

```c
glCompileShader(vertexShader);
glCompileShader(fragmentShader);
```
Linking vertex and fragment shaders together

LINKING

program = glCreateProgram();

glAttachShader(program, vertexShader);

glAttachShader(program, fragmentShader);

glLinkProgram(program);
GEOMETRY

Triangle meshes

• Set of vertices
• Connectivity defined by indices
  • `uint16_t indices[] = {vertex_index1, vertex_index2, vertex_index3, ...}`

OpenGL resources

Create

```
Gluint vbo;
glGenBuffers(vbo);
```

```
Gluint ibo;
glGenBuffers(ibo);
```

three indices make one triangle
Programmatic geometry definition

```cpp
vec3 vertices[150];
vertices[0].position = { -0.54, +1.34, -0.01 };
vertices[1].position = { +0.75, +1.21, -0.01 };
...
vertices[150].position = { -1.22, +3.59, -0.01 };

uint16_t indices[] = { 0, 3, 1,.. , 152, 150 };
GLuint vbo;
glGenBuffers(vbo);
glBindBuffer(vbo);
glBufferData(vbo, vertices);
GLuint ibo;
glGenBuffers(ibo);
glBindBuffer(ibo);
glBufferData(ibo, indices);
```
Vertex Shader

- VS is run for each vertex SEPARATELY
  - doesn’t know connectivity (by default)
- Input:
  - vertex coordinates in Object Coordinate System
  - vertex attributes: color, normal, …
  - uniform/global variables
- It’s primary role is to transform
  Object coordinates
  -> WORLD coordinates
  -> VIEW coordinates
Recap: GLSL Vertex shader

The OpenGL Shading Language (GLSL)
• Syntax similar to the C programming language
• Build-in vector operations
  • functionality as the GLM library our assignment template uses

```cpp
uniform mat3 transform;
uniform mat3 projection;

void main() {
  // Transforming The Vertex
  vec3 out_pos = projection * transform * vec3(in_pos.xy, 1.0);
  gl_Position = vec4(out_pos.xy, in_pos.z, 1.0);
}
```

world -> camera
object -> world
Recap: From local object to camera coordinates

1. **World coordinates**
   - object -> world: transform

2. **World coordinates**
   - world -> camera: projection

3. **Camera coordinates**
   - object -> camera: projection * transform
Setting (Vertex) Shader Variables

**Uniform variable**

```c
mat3 projection_2D{ { sx, 0.f, 0.f },{ 0.f, sy, 0.f },{ tx, ty, 1.f } }; // affine transformation as introduced in the prev. lecture
GLint projection_uloc = glGetUniformLocation(texmesh.effect.program, "projection");
glUniformMatrix3fv(projection_uloc, 1, GL_FALSE, (float*)&projection);
```

**In variable (attribute for every vertex)**

// assuming vbo contains vertex position information already
```c
GLint vpositionLoc = glGetAttribLocation(program, "in_position");
glEnableVertexAttribArray(vpositionLoc);
glVertexAttribPointer(vpositionLoc, 3, GL_FLOAT, GL_FALSE, sizeof(vec3), (void*)0);
```
Variable Types

**Uniform**
- same for all vertices

**Out/In (varying)**
- computed per vertex, automatically interpolated for fragments

**In (attribute)**
- values per vertex
- available only in Vertex Shader
Salmon Vertex shader

```glsl
#version 330

// Input attributes
in vec3 in_position;
in vec3 in_color;

out vec3 vcolor;
out vec2 vpos;

// Application data
uniform mat3 transform;
uniform mat3 projection;

void main() {
    vpos = in_position.xy; // local coordinated before transform
    vcolor = in_color;
    vec3 pos = projection * transform * vec3(in_position.xy, 1.0);
    gl_Position = vec4(pos.xy, in_position.z, 1.0);
}
```

pass on color and position in object coordinates as before
Recap: Fragment shader examples

- simulates materials and lights
- can read from textures

Diffuse  Specular  Directional
#version 330
// From Vertex Shader
in vec3 vcolor;
in vec2 vpos; // Distance from local origin

// Application data
uniform vec3 fcolor;
uniform int light_up;

// Output color
layout(location = 0) out vec4 color;

void main() {
    color = vec4(fcolor * vcolor, 1.0);

    // Salmon mesh is contained in a 1x1 square
    float radius = distance(vec2(0.0), vpos);
    if (light_up == 1 && radius < 0.3) {
        // 0.8 is just to make it not too strong
        color.xyz += (0.3 - radius) * 0.8 * vec3(1.0, 1.0, 0.0);
    }
}
(Hidden) Vertex Post-Processing

• Viewport transform: camera coordinates to screen/window coordinates
  • set with `glViewport(θ, θ, w, h);`

• Clipping: Removing invisible geometry (outside view frame)
SPRITES: Faking 2D Geometry

- Creating geometry is hard
- Creating texture is “easy”
- In 2D it is hard to see the difference

- SPRITE:
  - *Use basic geometry (rectangle = 2 triangles)*
  - *Texture the geometry (transparent background)*
  - *Use blending (more later) for color effects*
Sprite basics

A textured quad looks like fine-grained 2D geometry

Transparent with alpha = 0
e.g., color_RGBA = \{1,1,1,0\}

Proper occlusion despite simple geometry
Create Quad Vertex Buffer

```
vec3 vertices[] = { v0, v1, v2, v3 };

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, vertices_size, vertices, GL_STATIC_DRAW);
```

OpenGL initialization (once):
SPRITES: Creation

OpenGL initialization (once):

Create Quad Index Buffer

```c
uint16_t indices[] = { 0, 1, 2, 1, 3, 2 };  
GLuint ibo;  
glGenBuffers(1, &ibo);  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices_size, indices, GL_STATIC_DRAW);
```

Load Texture

```c
GLuint tex_id;  
glGenTextures(1, &tex_id);  
glBindTexture(GL_TEXTURE_2D, tex_id);  
glTexImage2D(GL_TEXTURE_2D, GL_RGBA, width, height, ..., tex_data);
```
SPRITES: Rendering

OpenGL rendering (every frame):

**Bind Buffers**

```c
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
```

**Enable Alpha Blending**

```c
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

// Alpha Channel Interpolation

```c
// RGB_o = RGB_src * ALPHA_src + RGB_dst * (1 - ALPHA_src)
```
SPRITES: Rendering

Bind Texture

```c
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texmesh.texture.texture_id);
```

Draw

```c
glDrawElements(GL_TRIANGLES, 6, ..); // 6 is the number of indices
```
Color and Texture Mapping

• *How to map from a 2D texture to a 3D object that is projected onto a 2D scene?*
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
Self study:
Interpolation with barycentric coordinates

- *linear combination of vertex properties*
  - e.g., color, texture coordinate, surface normal/direction, …

- *weights are proportional to the areas spanned by the sides to query point P*
Texture mapping

\[(s_0, t_0), (s_1, t_1), (s_2, t_2)\]
Texture mapping

(s_0, t_0)

(s_1, t_1)

(s_2, t_2)
Texture mapping

(s₀, t₀) → (s₁, t₁) → (s₂, t₂)
Blending

**Blending:**

- Fragments -> Pixels
- Draw from farthest to nearest
- No blending – replace previous color
- Blending: combine new & old values with some arithmetic operations
  - *Achieve transparency effects*

*Frame Buffer*: video memory on graphics board that holds resulting image & used to display it
Depth Test / Hidden Surface Removal

Remove occluded geometry

• Parts that are hidden behind other geometry
• For 2D (view parallel) shapes – use depth order
  • draw objects back to front
    • sort objects: furthest object first, closest object last
Self study: Alternative to ordering
Depth buffer with transparent sprites

- **Fragment shader writes depth to the depth buffer**
  - discard fragment if depth larger than current depth buffer (occluded)
  - alleviates the ordering of objects

- **Issue, depth buffer written for fragments with alpha = 0**

- **Solution:**
  - explicitly discard fragments with alpha < 0.5

  - note, texture sample interpolation leads to non-binary values even if texture is either 0 or 1.

```glsl
#version 330
in vec2 texCoord;
out vec4 outColor;
uniform sampler2D theTexture;

void main() {
  vec4 texel = texture(theTexture, texCoord);
  if(texel.a < 0.5)
    discard;
  outColor = texel;
}
```