Image Processing and Computer Graphics

OpenGL

Matthias Teschner

Computer Science Department
University of Freiburg
Introduction

- OpenGL is a graphics rendering API
  - display of geometric representations and attributes
  - independent from operating system and window system
- OpenGL realizes the interaction with GPUs
  - hardware-accelerated rendering
**OpenGL 1.0 (1994)**

- fixed-function pipeline
- focus on parallelized implementation
- promoted by quasi-standards of all components of a rasterization-based renderer

Diagram:
- Connectivity Data
- Vertex Data
- Texture Data
- Vertex Processing
- Primitive Setup Rasterization
- Fragment Processing
- Framebuffer
fixed-function pipeline with programmable vertex and fragment processing

- vertex and fragment processing could be replaced by user-defined functionality (shaders)
- shaders are programs that work on each vertex / fragment
**OpenGL 3.0**

- programmable vertex and fragment processing
- no fixed-function pipeline
  - vertex and fragment shaders have to be implemented

![Diagram of OpenGL 3.0 pipeline](image-url)
 OpenGL 3.0

- focus on core functionality
  - removal of OpenGL features
  - deprecation model (full, forward compatible)
- improved handling of large data (buffer objects)
- improved flexibility
  - implementation of non-standard effects not restricted to “misusing” pipeline functionality
- programming
  - not just setting parameters of standard functionality
  - concepts of vertex and fragment processing are not “nice to know”, but required knowledge
    - e.g., transforms, projections
OpenGL 3.0

-Deprecated features, e.g., `glRotate`
  - Generates a transformation matrix
  - Multiplies the matrix with the top element of the current stack
- Typically replaced by OpenGL Mathematics glm
  - `glm::Rotate`
  - Generates a transformation matrix
**OpenGL 3.2**

- geometry shader
  - optional
  - modify geometric primitives
  - generation of geometry no longer restricted to CPU
- flexible use of texture data
OpenGL 4.1

- tessellation shader
  - optional
  - tessellate patches
  - flexible generation of large and detailed geometries
OpenGL 4.3

- compute shader
  - optional
  - perform arbitrary computations
  - are not part of the rendering pipeline
**GPU Data Flow**

- data transfer to GPU
  - vertices with attributes and connectivity
- vertex shader
  - a program that is executed for each vertex
  - input and output is a vertex
- rasterizer
- fragment shader
  - a program that is executed for each fragment
  - input and output is a fragment
- framebuffer update
Data Transfer

- Vertex Buffer Object VBO
  - used to copy memory from CPU to GPU
  - contains arbitrary data, typically vertex attributes

```c
GLuint gVBO = 0;
glGenBuffers(1, &gVBO);
glBindBuffer(GL_ARRAY_BUFFER, gVBO);

GLfloat vertexData[] = {
    // X     Y     Z
    0.0f, 0.8f, 0.0f,
    -0.8f, -0.8f, 0.0f,
    0.8f, -0.8f, 0.0f};

glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);
```

[Tom Dalling]
Data Transfer

- Vertex Array Object VAO
  - link between VBO and shader programs
  - specifies how to interpret VBO data
  - specifies the mapping to input variables of shaders

```c
GLuint gVAO = 0;
glGenVertexArrays(1, &gVAO);
glBindVertexArray(gVAO);

// connect the xyz to the "vert" attribute
// of the vertex shader

glEnableVertexAttribArray(gProgram->attrib("vert"));
glVertexAttribAttribPointer(gProgram->attrib("vert"), 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

[Tom Dalling]
**Shader**

- program
  - written in OpenGL Shading Language GLSL
  - runs on the GPU
  - vertex and fragment shader are mandatory

<table>
<thead>
<tr>
<th></th>
<th>Main Program</th>
<th>Shader Program</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>C++</td>
<td>GLSL</td>
</tr>
<tr>
<td><strong>Main function</strong></td>
<td>int main(int, char**);</td>
<td>void main();</td>
</tr>
<tr>
<td><strong>Runs on</strong></td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td><strong>Gets compiled?</strong></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Gets linked?</strong></td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

[Tom Dalling]
**Vertex Shader**

- works on vertices
  - input and output are vertices
- minimum functionality
  - transformation from local to clip space
  (after clipping, rasterizer only works on vertices in the canonical view volume [-1..1, -1..1, -1..1])
Simple Vertex Shader Example

- `#version 150`

  ```glsl
in vec3 vert;

  void main() {
    // does not alter the vertices at all
    gl_Position = vec4(vert, 1);
  }
  ```

- model, view and projection transform are implicitly set to identity matrices

  $$
  \begin{pmatrix}
  x_{\text{clip}} \\
  y_{\text{clip}} \\
  z_{\text{clip}} \\
  1
  \end{pmatrix} = \mathbf{I} \begin{pmatrix}
  x_{\text{local}} \\
  y_{\text{local}} \\
  z_{\text{local}} \\
  1
  \end{pmatrix}
  $$

  [Tom Dalling]

  gl_Position is a built-in output variable of a vertex shader. It is a 4D vector \((x,y,z,1)\) representing the clip-space position of a vertex.
Simple Vertex Shader Example

- GLfloat vertexData[] = {
  // X   Y   Z
  0.0f, 0.8f, 0.0f,
  -0.8f,-0.8f, 0.0f,
  0.8f,-0.8f, 0.0f};

- results in a visible triangle for the example shader as all input/output vertex positions are within the canonical view volume

[Tom Dalling]
Typical Vertex Shader Example

- `uniform mat4 projection;` set in the main program
- `uniform mat4 camera;` read from VAO
- `uniform mat4 model;`
- `in vec3 vert;`

```cpp
void main() {
    gl_Position = projection * camera * model * vec4(vert, 1);
}
```

- `glm::mat4 projection = glm::perspective(…);` internal camera parameters
- `gProgram->setUniform("projection", projection);`

- `glm::mat4 camera = glm::lookAt(glm::vec3(3,3,3), glm::vec3(0,0,0), glm::vec3(0,1,0));` object placement
- `gProgram->setUniform("camera", camera);`

[Tom Dalling]
**Vertex Shader**

- can compute/set a color at vertices
  - e.g. red, green, blue
- rasterizer can interpolate attributes from vertices to fragments, e.g.

![](image)

result for an empty fragment shader employing the interpolation functionality of the rasterizer
Simple Fragment Shader Example

- #version 150

  out vec4 finalColor;

  void main() {
    // set every drawn pixel to white
    finalColor = vec4(1.0, 1.0, 1.0, 1.0);
  }

- if not set-up otherwise, the output is written to the color buffer

[Tom Dalling]
Typical Fragment Shader Example

```glsl
#version 150

in ... 
out vec4 finalColor;

void main() {
    //calculate the vector from pixel to light source
    vec3 surfaceToLight = light.position - fragPosition;

    //calculate the cosine of the angle of incidence
    float brightness = dot(normal, surfaceToLight) /
    (length(surfaceToLight) * length(normal));
    brightness = clamp(brightness, 0, 1);

    //calculate final color of the pixel, based on:
    finalColor = vec4(brightness * light.intensities *
    surfaceColor.rgb, surfaceColor.a);
}

incomplete shader for Phong illumination

[Tom Dalling]
```
GPU Data Flow

- data transfer to GPU
  - VBOs store data, VAOs interpret the data
  - vertices with attributes and connectivity
- vertex shader
  - input is a vertex in local space
  - output is a vertex in clip space
- rasterizer
  - generates fragments
  - interpolates attributes from vertices to fragments
- fragment shader
  - output is a fragment color
## OpenGL Setup

- Implementations are typically accomplished by additional libraries
  - OpenGL Extension Wrangler GLEW
    - Access to OpenGL x.x API functions
  - GLFW
    - Windowing, mouse and keyboard handling
  - OpenGL Mathematics GLM
    - Processes vectors and matrices

- Implementation
  - Fragment shader
  - Vertex shader