OpenGL
Performances and Flexibility

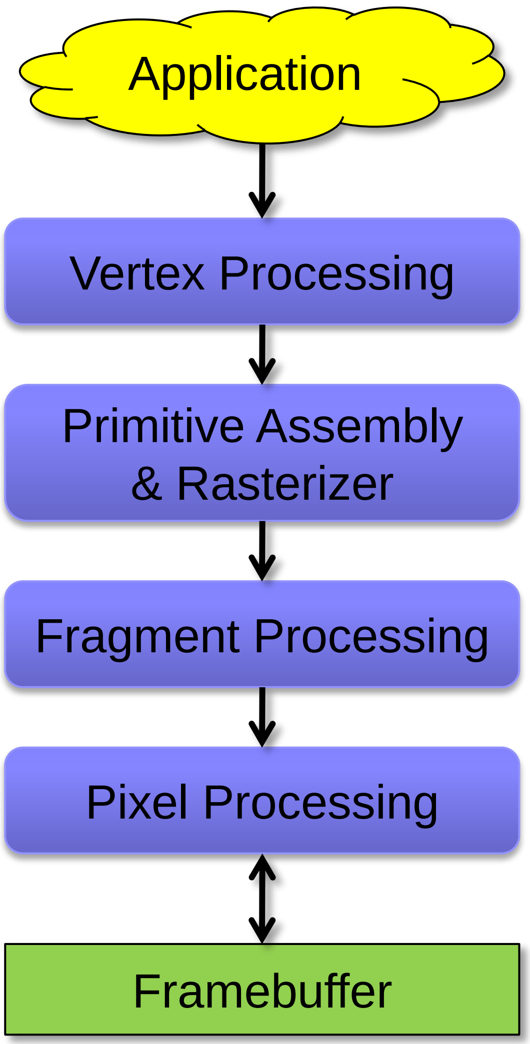
Marco Di Benedetto

Visual Computing Laboratory – ISTI – CNR, Italy

OpenGL Roadmap

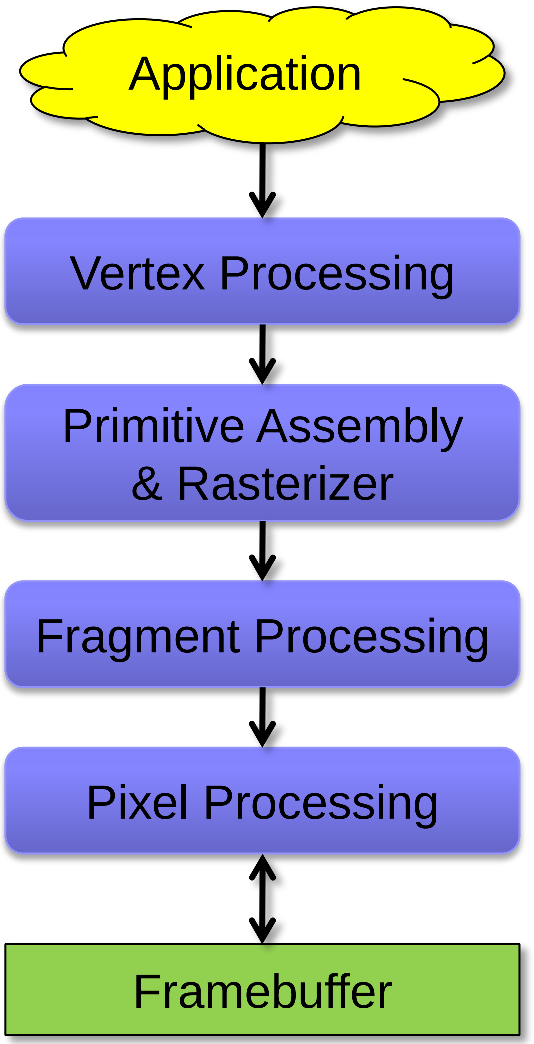
- 1.0 - Jan 1992 - First Version
- 1.1 - Jan 1997 - Vertex Arrays, Texture Objects
- 1.2 - Mar 1998 - 3D Texturing, Separate Specular Color, Vertex Array draw element range
- 1.2.1 - Oct 1998 - Multi-Texturing
- 1.3 - Aug 2001 - Compressed Textures, Cube Maps, Multi-Sampling
- 1.4 – Jul 2002 – Depth Textures, HW Shadowing, Separate Blend, Extended Texture Addressing
- 1.5 – Jul 2003 – Vertex Buffer Objects, Occlusion Queries, Extended Shadow Functions
- 2.0 – Sep 2004 – Vertex and Fragment Shaders, Multiple Render Targets, Separate Stencil
- 2.1 – Jul 2006 – Pixel Buffer Objects, sRGB
- 3.0 – Jul 2008 – Framebuffer Objects, HW Instancing, Vertex Array Objects
- 3.1 – Mar 2009 – Texture and Uniform Buffer Objects, Integer Textures, Fast Buffer Copy (OpenCL)
- 3.2 – Aug 2009 – Geometry Shaders, Multisampled Textures, Synch and Fence Objects
- 3.3 – Mar 2010 – Sampler Objects, Profiles Introduction
- 4.0 – Mar 2010 - Tessellation Shaders, Per-Sample Fragment Shaders, Shader Subroutines, Double Precision

The Abstract Graphics Pipeline



1. The application specifies vertices & connectivity.
2. The VP transforms vertices and compute attributes.
3. Geometric primitives are assembled and rasterized, attributes are interpolated. Culling occurs here.
4. The FP computes final “pixel” color.
5. The PP (output merger) writes pixels onto the FB after stencil/depth test, color blending.

The OpenGL *Fixed Function* Pipeline



1. The application specifies vertices & connectivity.
2. Transform & Lighting
3. Geometric primitives are assembled and rasterized, attributes are interpolated. Culling occurs here.
4. Texture Mapping & Fog
5. Alpha test, Stencil/Depth test, Color Blending

The OpenGL FF Machine

- Most stages are configurable (turn lights on, specify backfacing, ...)
- No stage is programmable (hardwired logic)
- Vertex attributes have explicit semantic (position, normal, color, uv)
- Lighting equation is fixed (Phong illumination model)
- Texture images have fixed color semantic

Rendering Example

```
void render(void)
{
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(fovY, width/height, zNear, zFar);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ);

  glEnable(GL_DEPTH_TEST);

  glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f);  glVertex3f(0.0f, 0.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f);  glVertex3f(1.0f, 0.0f, 0.0f);
    glColor3f(0.0f, 0.0f, 1.0f);  glVertex3f(0.0f, 1.0f, 0.0f);
  glEnd();

  glDisable(GL_DEPTH_TEST);
}
```

Data Storage

- Textures reside in graphics memory
- Vertices:
  - Immediate Mode: client-side attributes queued in a buffer, then sent in batches to the HW
  - Vertex Arrays: HW fetches data from client-side memory addresses (most probably memory mapped I/O)
  - Display Lists: Thou shalt not know my secrets! ☺
  - Vertex Buffer Objects: graphics memory, memory mappable

Buffer Objects (1/2)

- Introduced in OpenGL 1.5 (2003)

- Raw chunk of graphics memory
  - Allocation should be considered slow but usually done only once
  - Fixed size
  - Application can query a pointer to them and read/write
  - The Gfx Pipeline can use them as data sources for internal stages or as data sinks from other stages (data reinjection)

- Given the *handle* to a buffer, we can *bind* it to several binding sites
  - Named vertex attribute (VBO)
  - Primitive indices array (EBO)
  - Pixel read/store (PBO) (GL 2.1 – 2006)

Buffer Objects (2/2)

- Used for static & dynamic data
  - Access through memory pointers, forget glVertex, glNormal etc.
- To modify content:
  - Recreate the whole buffer (optimized if same size)
  - Respecify data subsection (offset & size)
  - Request a read and/or write pointer to whole or range
  - Specifying a NULL pointer tells the GL to invalidate data, possibly speeding up the following update operations
- Whenever a buffer is bound to a site, all relative GL calls which accept a pointer will interpret the pointer value as an offset into the bound buffer
  - Otherwise it points to client-side memory

VBO Example

```
// vertex array
float positions[] = { ... };
glVertexPointer(3, GL_FLOAT, 3*sizeof(float), positions);

// vbo
// creation & fill
Gluint positionVBO = 0;
glGenBuffer(1, &positionVBO);
glBindBuffer(GL_ARRAY_BUFFER, positionVBO);
glBufferData(GL_ARRAY_BUFFER, vertCount*3*sizeof(float),
positions, GL_STATIC_DRAW);
...
// usage
glBindBuffer(GL_ARRAY_BUFFER, positionVBO);
glVertexPointer(3, GL_FLOAT, 3*sizeof(float), 0);
```

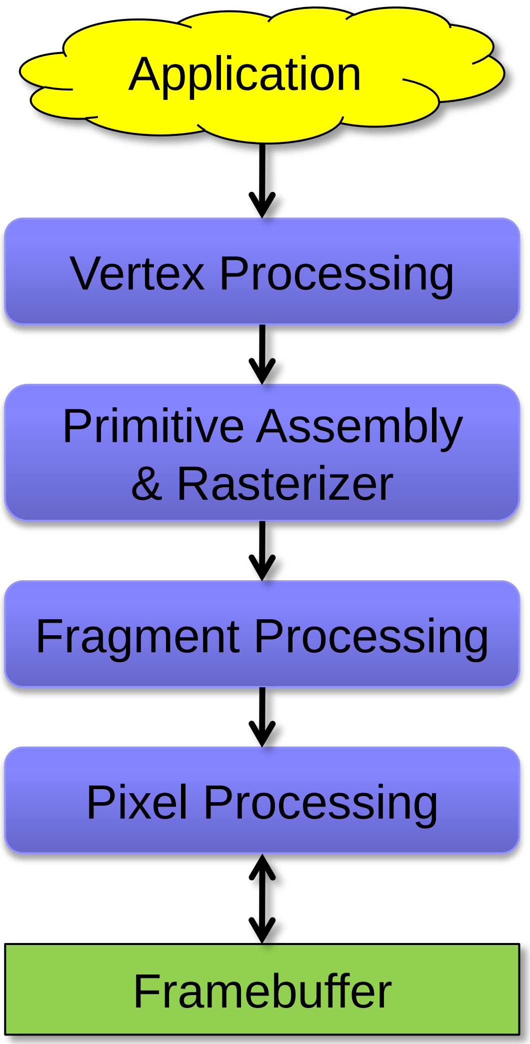
A note on GL objects usage pattern

- Bind to EDIT / Bind to USE
- All subsequent calls refer to the bound object
- Complicates development of layered libraries
  - 1. Query the current bound object
  - 2. Bind the object to edit
  - 3. Edit the object
  - 4. Bind the previously bound one

- What if, when editing, the GL calls simply take the referred object as a parameter? (like C functions acting on structs)

- GL_EXT_direct_state_access comes to help
  - John Carmack (id Software) as a main contributor & promoter; hopefully moved into core specifications the next release

The Abstract Graphics Pipeline



1. The application specifies vertices & connectivity.
2. The VP transforms vertices and compute attributes.
3. Geometric primitives are assembled and rasterized, attributes are interpolated. Culling occurs here.
4. The FP computes final “pixel” color.
5. The PP (output merger) writes pixels onto the FB after stencil/depth test, color blending.

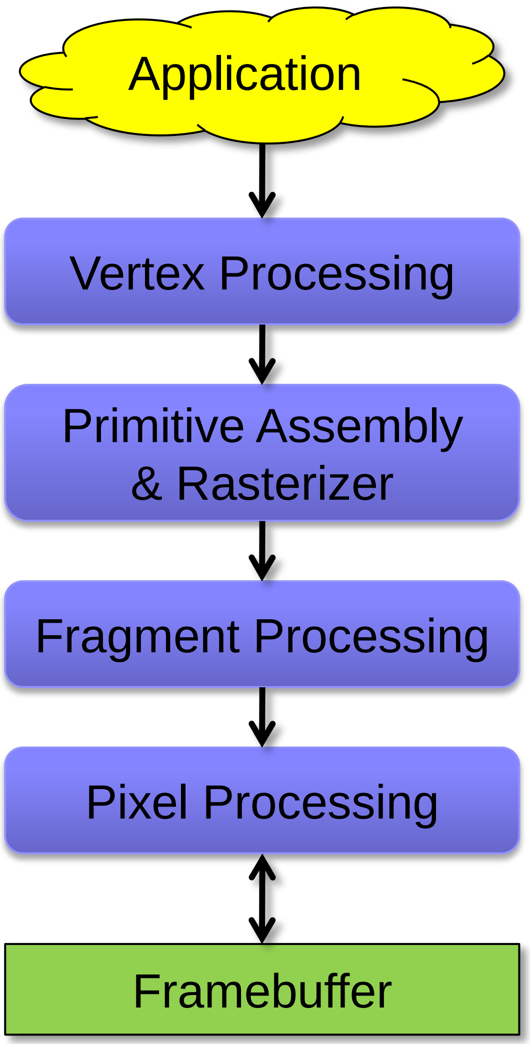
Some Fixed Function Limitations

- glVertex/Normal/Color/TexCoord: we have only 4 vertex attributes, each directing its data stream into a fixed block of computation
  - Semantic is fixed because calculations are fixed (lighting eq., texturing)
  - If we need, say, per-vertex tangent space for bump maps?
    Even if we had it, we cannot use it

- glMaterial, glLight: emission, ambient, diffuse, specular
  - Directly derived from the Phong model, no way to implement fancy/more complex lighting models

- glTexImage: textures are raw containers of color
  - Texel color is added, multiplied, ..., to the frag final color
  - A specular map for simulating inhomogeneously specular surfaces?
  - Anisotropic materials (velvet) ?

The OpenGL *Programmable* Pipeline



1. The application specifies vertices & connectivity.

2. The VP runs a general purpose program for each vertex. The Vertex Shader is mandate to output position.

3. A Geometry Shader can be optionally run to modify or kill the assembled primitive or emit new ones (GL 3).

4. The FP runs a general purpose program for each frag. The Fragment Shader is mandate to output a color.

5. The PP (output merger) writes pixels onto the FB after stencil/depth test, color blending.

5.1. The output merger can be told to operate on one or more textures at the same time, rather than the screen framebuffer.

The Program Model

- Creation
  - Create a vertex, a fragment and optionally a geometry shader object
  - Specify the shader source code as a string, compile it
  - Create a program object
  - Attach shaders to program
  - Specify how vertex attributes are mapped to vertex shader input
  - Specify how fragment shader outputs are mapped to framebuffer
  - Link program

- Usage
  - Bind Program (program 0 → FF)
  - Set program input arguments (they are *global* variables in shaders)
  - Draw as usual

Example: Vertex Shader

```
uniform mat4 u_model_view_projection_matrix;
uniform mat3 u_view_space_normal_matrix;

in  vec4 a_position;
in  vec3 a_normal;
in  vec2 a_texcoord;

out vec3 v_normal;
out vec2 v_texcoord;

void main(void)
{
  v_normal    = u_view_space_normal_matrix * a_normal;
  v_texcoord  = a_texcoord;
  gl_Position = u_model_view_projection_matrix * v_position;
}
```

Example: Fragment Shader

```
uniform vec3      u_view_space_light_position;
uniform vec3      u_color;
uniform sampler2D s_texture;

in  vec3 v_normal;
in  vec2 v_texcoord;

out vec4 o_color;

void main(void)
{
  vec3  normal   = normalize(v_normal);
  float lambert  = dot(normal, u_view_space_light_position);
  vec3  texcolor = texture(s_texture, v_texcoord);
  o_color        = vec4(texcolor * lambert, 1.0);
}
```

What went out of core specs

- Immediate Mode (glBegin/End, glVertex, ...)
  - Everything through data pointers
- Vertex attributes semantic
  - glVertex/Normal/...Pointer → generic glVertexAttribPointer(***index***, ...);
  - Must link ***index*** with VS ***in*** parameter
- Display Lists
- Matrix stacks
  - We have to calculate by hand our matrices, no native push/pop
  - Specify through uniforms
- Lighting stuff
  - No concept of light sources or material
  - Specify through uniforms

General considerations

- Immediate Mode is the most flexible way but it's slow
- Display Lists overcome rendering speed limitations
  - But are very very slow to compile!
- VBOs are the fastest way
  - But updating them is not easy
- Every vertex attributes must be fetched from (client)buffers
  - Forget calculations in place like in IM, we have to compute AND store them
- We have to implement matrices and matrix stacks
  - Easy, several libs available
- Light sources must be accounted inside shaders
  - A variable # of lights means procedurally generate and compile shaders code or use uber shaders

What's more in OpenGL 3

- Most things are meaningful for real-time games
- Transform feedback: catch output generated from geometry shaders and reinject them into the pipe (also for physical sims)
- Extended occlusion queries (from GL 1.5)
- Synchronization objects
- Facilities for interoperability with OpenCL
- Framebuffer objects: render-to-texture
- Sampler objects: decouple texture images from their sampling

And GL 4.0

- New PATCH primitive
- Tessellator

Whath we'll see in the tutorial

- Define and update an indexed triangle mesh
- Use several mesh rendering/update strategies from GL 1.0 to 3.0
- Fixed function configuration and its programmable counterpart

The Framework: Base Classes

- Mesh
  - Holds vertex & connectivity information
  - Knows how to render/update its data with the GL system
- Builder
  - A simple objects which construct the initial state of a Mesh
- Updater
  - A simple object which knows how to update/animate mesh vertices
- Renderer
  - Sets up and control the rendering environment

The Framewark: Specialized Classes

- MeshImmediateMode
- MeshDisplayList
- MeshVertexArray
- MeshVertexBuffer

- BuilderGrid

- RendererFixed
- RendererProgrammable

What we will do

- Under the hood a window with a GL context is created with GLUT
- We have to deal with four event handlers:
  - Initialize(): application startup
  - Finalize(): application termination
  - Update(dt): called before rendering
  - Draw(): the actual rendering process

Tutorial

EOF