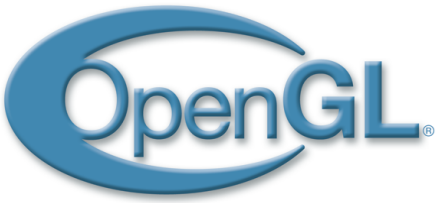
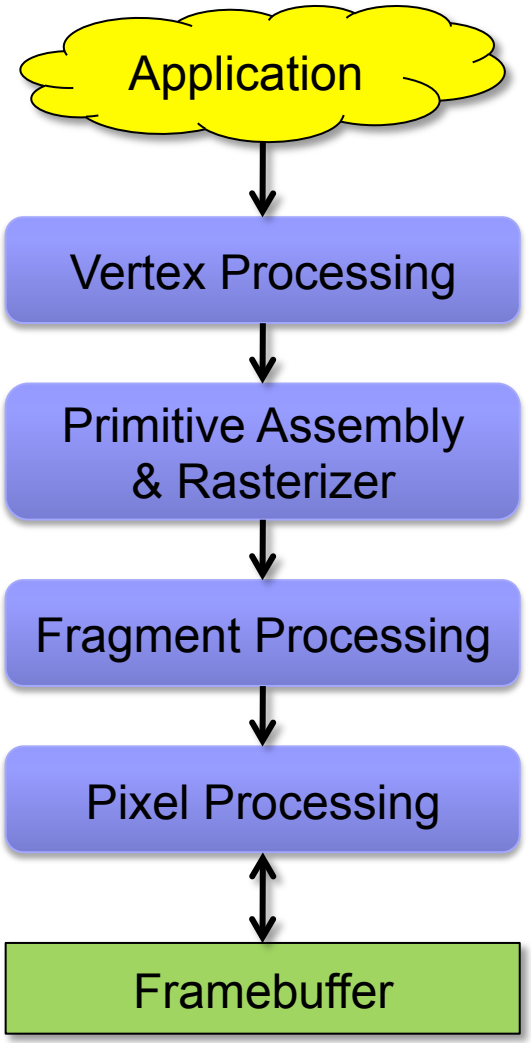


OpenGL
Performances and Flexibility

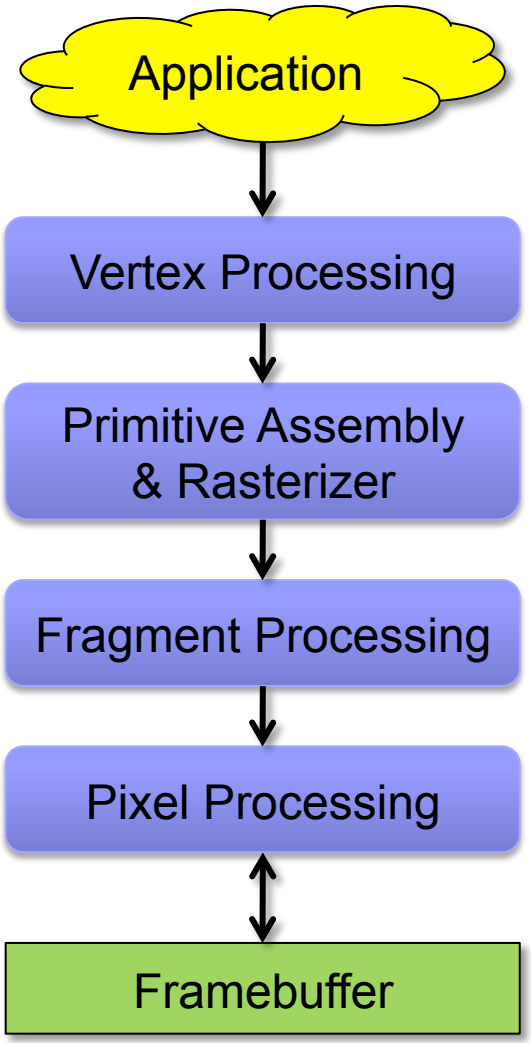
Visual Computing Laboratory – ISTI – CNR, Italy

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)
{
  glPushAttrib(GL_ALL_ATTRIB_BITS);
  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();

 glPopAttrib();
}
```

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);
```

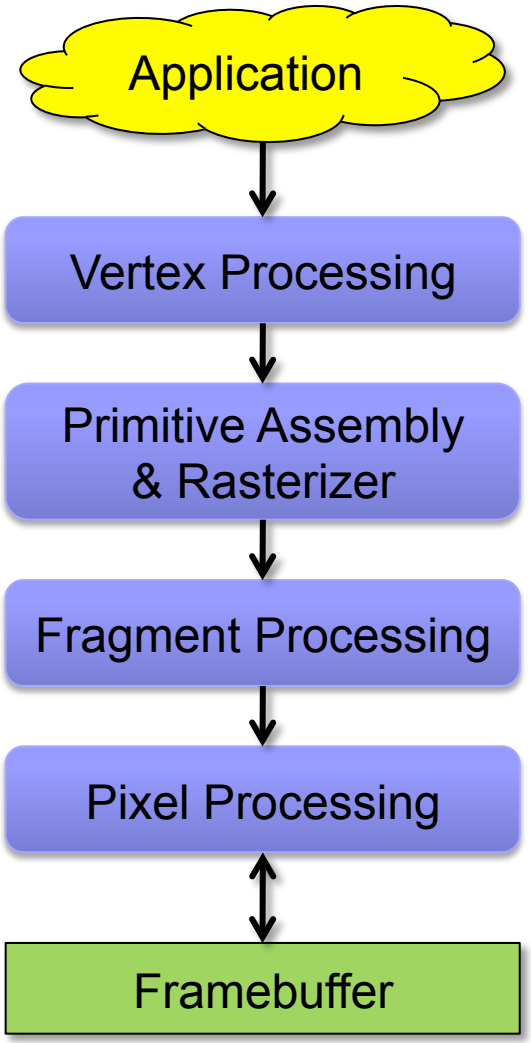
A note on GL objects usage pattern

- 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

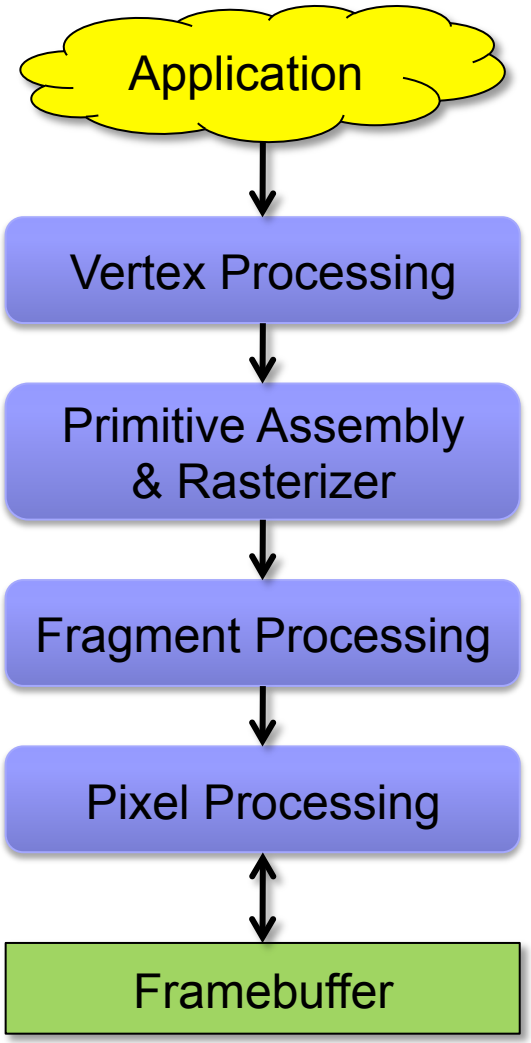


1. The application specifies vertices & connectivity.
2. The VP transforms vertices and compute attributes.
3. Geometric primitives are assembled and rasterized, attributes are interpolated. Culling occurs here.
4. The FP computes final “pixel” color.
5. The PP (output merger) writes pixels onto the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

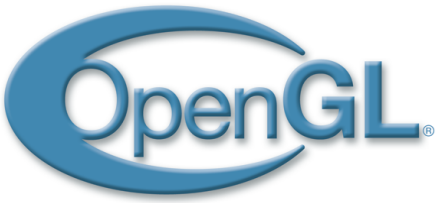
```
uniform vec3      u_view_space_light_position;
uniform sampler2D s_texture;

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



Framebuffer Objects

Introductory Example

- Scenario:
  - We must draw a control room with a live action security camera TV
  - The camera is recording a scene simulated by our system
- → We must be able to draw the TV while it is showing a dynamic, computer-generated scene
- In practice:
  - Render the scene as seen by the security camera
  - Use the result of the rendering as a texture mapped on the TV screen

Use the Color Buffer as a Texture

- We want to be able to use the framebuffer content as a texture
- OpenGL can do this since v1.0
  - glCopyTexImage2D() : copy the content of the color buffer to a texture
- Problems
  - *Pixel Ownership Test*: if the window we are rendering to is partially occluded by something (e.g. other overlapping windows), the occluded pixels will not be written
    → *holes* in the texture
  - We must pay a copy operation

Render-to-Texture (RTT)

- Ideally, we want to draw *directly* on the texture to avoid windows-manager issues and memory copy operations

- Early solution: PBuffers
  - Framebuffers that can be used as textures
  - PBuffers have their own OpenGL context → data must be shared
  - GL context switches → slow synchronization → slow slow slow ...
  - ... but the idea was good!

- Modern solution: Framebuffer Objects
  - Draw *directly* into a texture

The Framebuffer

- A set of *ancillary* buffers: Color - Depth - Stencil
- Double Buffering
  - If we render several objects, one at a time, directly on the memory region used by the screen, we may experience *flickering* in the image
  - To avoid flickering, rendering is done on a ***back*** (frame)buffer (which is not visible), while the screen shows the ***front*** (frame)buffer
  - When rendering is done, buffers are swapped (or flipped)
  - → the screen can present the completely composed image at once
- OpenGL defines:
  - *Main* front/back buffers
  - *Left* and *Right* front/back buffers (for stereo rendering)
  - *Auxiliary* buffers (how many is implementation dependent)

Framebuffer Object (FBO)

- Simply put, the result of a rendering is written in a memory region
- Historically, the front and back buffers are the regions of memory chips written by the graphics hardware accelerator and read by the screen interface
- Texture images are regions of the graphics memory
- With OpenGL FBOs we can tell the hardware:
  - “this is your framebuffer (color, depth, stencil) memory pointer, write there”
  - We can render-to-texture!

FBO

- A FBO is enhough flexible to hold just the ancillary buffers it needs
  - Any combination of color/depth/stencil
  - Some actual implementation have depth-stencil buffers tied
- Render Targets: what a FBO can contain
  - Textures
  - Renderbuffers (for texture formats that are not writable)
- Steps:
  - Create the FBO
  - Attach textures or renderbuffers to attachment points
  - Bind as the target framebuffer

FBO

```
// construction at application init
GLuint fbo = 0;
glGenFramebuffers(1, &fbo);

glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,  GL_TEXTURE_2D, depth_tex, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, color_tex, 0);
const Glenum draw_buffers[] = { GL_COLOR_ATTACHMENT0 };
glDrawBuffers(sizeof(draw_buffers)/ sizeof(draw_buffers[0]), draw_buffers));
glBindFramebuffer(GL_FRAMEBUFFER, 0); // rebind main framebuffer (screen)

// usage
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
draw_texture_content();
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// render scene and use color_tex
render_scene();

// destruction at application exit
glDeleteFramebuffers(1, &fbo);
fbo = 0;
```

Multiple Render Targets (MRT)

- FBOs can contain multiple color buffers
- The actual composition is
  - A set of N color buffers (N is implementation dependent)
  - Zero or one depth buffer
  - Zero or one stencil buffer
- What it means
  - We have one depth/stencil buffer
  - We can output *simultaneously* N different *color* values
- Fragment shader:
  - gl_FragData[***i***] outputs to ***i***th color target
  - gl_FragColor is an alias for gl_FragData[0]

FBO - MRT

```
// construction at application init
GLuint fbo = 0;
glGenFramebuffers(1, &fbo);

glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,  GL_TEXTURE_2D, depth_tex,  0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, color_tex0, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, color_tex1, 0);
const Glenum draw_buffers[] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers(sizeof(draw_buffers)/ sizeof(draw_buffers[0]), draw_buffers));
glBindFramebuffer(GL_FRAMEBUFFER, 0); // rebind main framebuffer (screen)

// usage
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
draw_textures_content();
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// render scene and use color_tex0 and color_tex1
render_scene();

// destruction at application exit
glDeleteFramebuffers(1, &fbo);
fbo = 0;
```

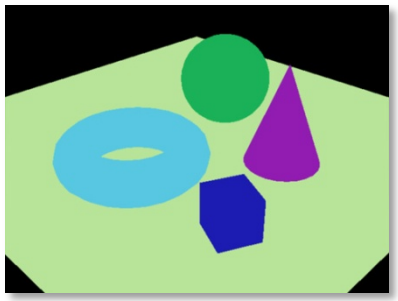
FBO Usage

- Introductory Example
  - The target color texture will be used as the TV texture
- Deferred Shading:
  - Pixels are written several times in a complex scene (depth complexity)
  - The closest one contributes to the final image
  - What if we perform complex calculations (e.g. lighting) just on visible pixels?

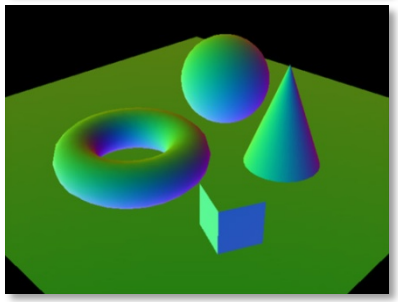
Deferred Shading

- Multipass rendering algorithm
- For each pixel, store in a set of textures all the values we need for our calculations
  - Normals - Vertex position - Base color
  - The set of textures we render to is called G-Buffer
- For each light source and for each pixel (full screen pass) fetch data from the G-Buffer and use them for lighting calculations

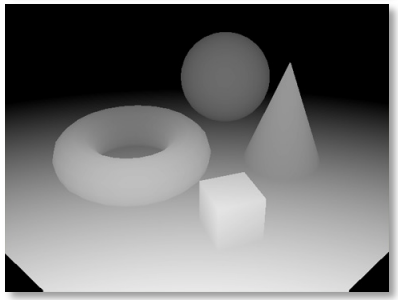
Deferred Shading



Base Color

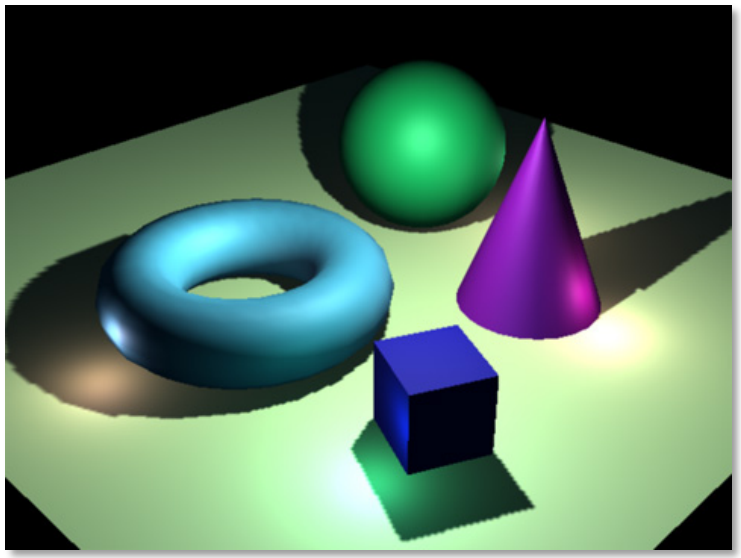


Normal



Depth

G-Buffer



Final Composed Image

Other Usages

- Post processing
  - Draw into a texture, then use image processing techniques
- Apply edge detection to perform selective antialiasing
- Procedural textures
- Screen Space Ambient Occlusion
- …

Render to Texture Arrays

- GL_TEXTURE_2D_ARRAY: a stack of 2D textures
- The active layer can be selected in the Geometry Shader

```
// when creating the FBO
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, color_tex_array, 0);
```

```
// geometry shader
// . . .
gl_Layer = 0;
gl_Position = …;
EmitVertex();
EndPrimitive();

gl_Layer = 1;
gl_Position = …;
EmitVertex();
EndPrimitive();
```