Computer Graphics

– OpenGL –

Philipp Slusallek
History

- **Graphics in the ‘80ies**
  - Framebuffer was a designated memory in RAM
  - „HW“: Set individual pixels directly via memory access
    - Peek & poke, getpixel & putpixel, …
    - MDA ('81: text only but 720x350 resolution)
      - Letter code was index into bit pattern for each letter
    - CGA ('81: 160x200:
      16 col w/ tricks;
      320x200: 4 col;
      640x200: 2 col)
    - EGA ('85: 640x350: 16 from 64 col, CGA mode)
    - VGA ('90: 640x480: 16 col @ table with 2^18 col,
      320x200: 256 col)
  - Everything done on CPU
    - Except for driving the display output
History (II)

• **Today (Nvidia Volta Flagship, GV100)**
  – Discrete graphics card via high-speed link
    • e.g. PCIe-3.0 x16: 1-16 GB/s (PCIe 4.0 coming: 2x improvement)
  – Autonomous, high performance GPU (more powerful than CPU)
    • 5376 SIMD processors
    • ~900 GB/s memory bandwidth (HBM2)
    • ~30 TFLOPS 16bit floats
    • ~15 TFLOPS single precision (SP)
    • ~7.5 TFLOPS double precision (DP)
    • 125 TFLOPS via 672 Tensor Cores for DL (4x4 matrix multiply)
    • Up to 16GB of local RAM plus virtual memory
  – Performs all low-level tasks & a lot of high-level tasks
    • Clipping, rasterization, hidden surface removal, …
    • Procedural geometry, shading, texturing, animation, simulation, …
    • Video rendering, de- and encoding, deinterlacing, …
    • Full programmability at several pipeline stages
    • Deep Learning: Training and Inference
History (III)

• Brief history of graphics APIs
  – Initially every company had its own 3D-graphics API
  – Many early standardization efforts
    • CORE, GKS/GKS-3D, PHIGS/PHIGS-PLUS, ...
    • 3D rendering, menus, input, events, text, ... → „Naturally grown“

• OpenGL (1992)
  – By Mark Segal & Kurt Akeley
    • Explicit design of a general vendor independent standard
  – Close to hardware but hardware-independent → Efficient
  – Orthogonal design and extensible
    • Common interface from mobile phone to supercomputer
    • Only real alternative today to Microsoft’s Direct3D
  – OpenGL 3.0/3.2 (2008/2009), 4.0/4.1 (2010), ... , 4.6 (Jul’17)
    • Major redesign & cleanup, deprecated and removed functionality
    • Since Version 3.2: Profiles (core, compatibility, forward compatibility)
    • OpenCL, tesselation shaders, 64 bit variables, multi-viewpoint
    • 4.3: Compute shaders, adv. texture compression, ...
    • 4.5: Direct state access, compatibility to OGL ES3.1, ...
History (IV)

• **Direct3D (Microsoft, Part of DirectX multimedia APIs)**
  – Started as *Reality Labs* by RenderMorphics, bought by MS
    • Strong SW focus
  – First version in 1996, Retained & Immediate Mode API
  – Played catch-up to OpenGL until Direct3D 6.0 (1998)
  – Significantly advanced by close collaboration with HW vendors
  – Largely feature parity since about 2008

• **Race to “Zero Driver Overhead”**
  – Started with initiative by game developers to have better control and avoid driver getting in their way, working with AMD since 2012
  – Goals: Move API closer to HW, give better control, eliminate SW overhead, more direct state handling, better multithreading, …
  – OpenGL showed performance advantages in 4.3 and 4.4 (2012/13)
  – AMD Mantle (2013) showing strong performance advantages
  – Similar approach be Apple with Metal (2014 (iOS) & 2015 (OS X))
  – DirectX 12 (Dec 2015) moved this into mainline gaming
  – Cross-platform with Vulkan API (Khronos, 2016)
    • Much lower level, requires expert programmer, … (not suited for teaching)
  – Vulkan seems to become the way to go (not for teaching, though)
Introduction to OpenGL (II)

• What is OpenGL?
  – Cross-platform, low-level software API for graphics HW (GPUs)
  – Controlled by Khronos (was Architecture Review Board (ARB))
  – Only covers 2D/3D rendering
  – Other APIs: Vulkan, MS Direct3D, Apple Metal
    • Related GUI APIs → X Window, MS Windows GDI, QT, GTK, Apple, ...
  – Was focused on immediate mode operation
    • As opposed to retained mode operation (storage of scene data)
    • Thin hardware abstraction layer – almost direct access to HW
    • Points, lines, triangles as base primitives
  – Today more efficient batch processing (immediate mode is gone)
    • Vertex arrays and buffer objects (controlled by app, but stored on GPU)
    • Vulkan: More of this: prevalidated buffers created by CPU threads
  – Network-transparent protocol
    • GLX-Protocol – X Window extension
      – Only in X11 environment!, now deprecated
Related APIs and Languages

- **glsl** (necessary, released in sync with OpenGL, → later)
  - The OpenGL shading language; defines programmable aspects
- **OpenGL ES (3.2)**
  - Embedded subset (used on most mobile devices)
  - Being better aligned with OpenGL (subset)
- **EGL (GLX, WGL, AGL/CGL)**
  - Glue library to windowing systems, EGL becoming standard now
- **OpenCL (2.2)**
  - Open Computing Language: Many-core computing
  - Cross-platform version of Nvidia’s CUDA
  - SPIR-V as a generic assembler format for GPUs
- **WebGL (2.0)**
  - In the browser, based on OpenGL ES 3.0
- **GUI-Toolkits**
  - QT: QtGLWidget class, Gtk: GtkGLExt widget
  - SDL: Simple DirectMedia Layer (more modern, w/ audio etc)
  - GLUT (OpenGL Utility Toolkit, older but still useful)
Additional Infos

• Just a few selected items (not complete)

• Books
  – Real-Time Rendering, Fourth Edition
    • By Tomas Akenine-Moller, Eric Haines, Naty Hoffman, et al.
    • Advanced Techniques
  – Learning Modern 3D Graphics Programming (Jason L. McKesson)
    • [http://alfonse.bitbucket.org/oldtut](http://alfonse.bitbucket.org/oldtut)
  – OpenGL SuperBible (7th edition, OGL 4.5)

• Tutorials
  – Lighthouse3D: [http://www.lighthouse3d.com](http://www.lighthouse3d.com)

• WebGL
    • Try out WebGL directly in the Web-Browser
Modern OpenGL Pipeline

- (Not looking at pixel input and output)
Complete OpenGL Pipeline (4.5)
OpenGL Rendering

- **OpenGL draws primitives**
  - Primitive types: points, lines, and triangles
  - Drawing subject to selectable *modes* (w/ their state) and *shaders*
  - Commands: Set modes, change parameters, send primitives
    - Data (parameters) is bound when call is made (even for arrays)
  - OpenGL *contexts* encapsulate the state
    - Created, deleted, and changed by windowing system
  - Window system also controls display of frame buffer content
    - E.g. gamma correction tables, bit depth, etc.

- **Frame buffers**
  - Default frame buffer (configured by window system, displayed)
  - Plus arbitrary number of application created frame buffers
Specifying Primitives

- **Geometric primitives**
  - Defined by vertices and their attributes
  - Vertices processed individually, all in the same way and in order
    - Until primitive assembly and rasterization
    - Clipping may change primitives (add/delete)

- **Providing Data Through Vertex Arrays**
  - Each vertex consists of the position and N attribute slots
  - `glEnable/DisableVertexAttribArray(slot)`
    - Enable use of array for specific slot (geometry always in slot 0)
    - Fixed static value can be specified via `glVertexAttrib(slot, ...)`
  - `glVertexAttribPointer(slot, size, type, normalized, stride, data)`
    - *Slot* defines which attribute is specified
    - *Size* specifies number of components (1D, 2D, 3D, 4D, BGRA)
    - *Type* data type in the array
      - Byte, short, int, float, half, double (+ unsigned integers)
    - *Stride* specifies the distance in bytes between two elements
    - *Data* points to the array data
    - *Normalized* defines how integer data is converted to float
Primitive Types

• **Modes for Vertex Arrays**
  - Points
  - Lines: Strips (connected), Loops (closed), Lines (separate)
  - Triangle: Strips (shared common edge), Fans (shared first vertex), Triangles (separate)

  ![Vertex Array Diagrams]

• **Advanced geometry types**
  - With adjacency: Additional vertices around a primitive
    - Lines, Line Strips, Triangles, TriangleStrips

  ![Advanced Geometry Types Diagrams]

  - Patches with a fixed number of vertices per patch
    - Must be used with tessellation shaders
Specifying Primitives

- **Drawing from Vertex Array**
  - `glDrawArrays(mode, first, count)`
    - Sends `count` vertices starting from `first` index
  - `glMultiDrawArrays(mode, first[], count[], elements)`
    - Same but executes `elements` times by iterating through `first` and `count`
  - `glDrawElements (mode, count, type, indices[])`
    - Indexes into vertex arrays via array of `indices` of given `type` (int, short, etc.)
  - `glMultiDrawElements (mode, count[], type, indices[][][], elements)`
    - Similar to MultiDrawArrays() but with indices
  - `glDrawArraysInstanced(mode, first, count, elements)`
    - Calls `glDrawArrays elements` times, incrementing a shader variable `instanceID` for each instance. Shader may have different transform each
  - `glDrawElementsInstanced(mode, count, type, indices[], elements)`
    - As expected …

- Main issue reducing the number of API calls to draw a scene
- Several other & more efficient draw calls available and being designed as extensions
Buffers

• Buffers store data on the server (GPU) side
  – `glGenBuffers(n, out bufferIds[])`, `glDeleteBuffers(…)`
    • Allocates and deletes buffer objects
• Types of BufferBindings

<table>
<thead>
<tr>
<th>Target name</th>
<th>Purpose</th>
<th>Described in section(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY_BUFFER</td>
<td>Vertex attributes</td>
<td>2.9.6</td>
</tr>
<tr>
<td>COPY_READ_BUFFER</td>
<td>Buffer copy source</td>
<td>2.9.5</td>
</tr>
<tr>
<td>COPY_WRITE_BUFFER</td>
<td>Buffer copy destination</td>
<td>2.9.5</td>
</tr>
<tr>
<td>DRAW_INDIRECT_BUFFER</td>
<td>Indirect command arguments</td>
<td>2.9.8</td>
</tr>
<tr>
<td>ELEMENT_ARRAY_BUFFER</td>
<td>Vertex array indices</td>
<td>2.9.7</td>
</tr>
<tr>
<td>PIXEL_PACK_BUFFER</td>
<td>Pixel read target</td>
<td>4.3.1, 6.1</td>
</tr>
<tr>
<td>PIXEL_UNPACK_BUFFER</td>
<td>Texture data source</td>
<td>3.7</td>
</tr>
<tr>
<td>TEXTURE_BUFFER</td>
<td>Texture data buffer</td>
<td>3.8.7</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_BUFFER</td>
<td>Transform feedback buffer</td>
<td>2.17</td>
</tr>
<tr>
<td>UNIFORM_BUFFER</td>
<td>Uniform block storage</td>
<td>2.11.7</td>
</tr>
</tbody>
</table>

Table 2.8: Buffer object binding targets.

• `glBindBuffers(target, bufferId)`
  – Binds a buffer object (with or without data) to a specific target

• `glBufferData(target, size, data, usage)`
  – Assigns data to a buffer object (and allocates memory for it)
  – *Usage* provides hints how the data may be used in future

• `glMapBuffer<Range>(target, <offset, length>, access)`
  – Maps/Copies (a range of) the buffer to address space of the client
  – Must `glUnmapBuffer()` before use of buffer in OpenGL
    • May use copy or mapping of virtual memory
Using Buffers

• All drawing calls and `glVertexAttribPointer` use the currently bound (if any) buffer
  – `ARRAY_BUFFER` for the vertex data
  – `ELEMENT_ARRAY_BUFFER` for the index data
  – All data (pointers) are interpreted as integers that provide offsets into these buffers (so are typically zero)

• A complete set of buffer objects for all slots can be specified with a Vertex Array Object (VAO)
  – `glGenVertexArrays()`, `glDeleteVertexArray()`
  – `BindVertexArray(array)`
    • For setup:
      – Bind all necessary buffers `glBindVertexArray()`
      – Specify the vertex formats `glVertexAttribPointer()`
    • Binding a VAO later sets up all buffers in the VAO simultaneously
    • Draw calls can use all associated buffers immediately
Complete OpenGL Pipeline (4.5)
Shaders

- **Shaders compute what gets rendered**
  - Draw commands just provide input for shaders

- **Shaders Stages communicate via interfaces**

- **Vertex Shaders**
  - Are executed for each vertex passed to OpenGL
    - Receives “uniform” parameters for the shader
    - “Attributes” for each vertex (see above)
    - Writes to a set of “varyings” variables
    - Output is rasterized, interpolated and forms “fragments”
  - The output of a vertex shader can also be recorded (in app)

- **Fragment Shader**
  - Are executed for every pixel covered by a primitive
    - Receive the interpolated (e.g. across triangle) varying variables
    - Outputs color, depth, other data (to eventually go into frame buffers)
  - Writing to buffers is still subject to per-fragment operations
Shaders (II)

• **Geometry Shader**
  – Are executed for every *primitive* that has been assembled
    • Receive an array of vertices (including adjacent vertices, if given)
  – Output primitives of a specific type
    • Generate new primitives by writing to all attribute variables and issuing a `EmitVertex()` call
    • Plus potentially an `EndPrimitive()` to start a new primitive

• **Tessellation Control/Evaluation Shader**
  – Advanced topic
  – Can only be used with Patch primitive
  – Control: Determines the parameters of tessellation
  – Fixed function stage does the tessellation
  – Evaluation Shader: generates and outputs new primitives

• **Programming shaders is discussed separately**
Shaders (III)

- Shaders specify the programmable parts of a pipeline
- Different Types of shaders
  - Must be compiled, combined into a “program”, and linked
- `glGenShader(type)`
  - Create a shader object for a shader of the given type
- `glShaderSource(shader, ...)`
  - Stores shader source code in the object
- `glCompileShader(shader)`
  - Compiles the shader object
- `glShaderBinary(…)`
  - Loads a precompiled shader in some internal format
- `glGenProgram()`
  - Creates a new shader program
- `glAttachShader(program, shader)`
  - Attaches a shader to a program
- `glLinkProgram(program) & glValidateProgram(program)`
  - Sets up the interfaces between the shader stages
- `glUseProgram(program)`
  - Preparea shader and use it for subsequent drawing calls
Shaders (IV)

- **New in OpenGL 4.1: Program Pipeline Object**
  - Encapsulates a preconfigured pipeline of shaders
- **`glGenProgramPipeline()`, `glDeleteProgramPipeline()`**
  - Allocates and deallocates such objects
- **`glBindProgramPipeline(id)`**
  - Activates the pipeline for draw commands and other operation
- **`glUseProgramStages(pipeline, stages, program)`**
  - Binds the program to the indicated shader stages of the pipeline
  - Program must be linked as „separable“ (a la „shared library“, DLL)
  - Special rules apply to handling input/output variables of shaders
- **`glGetProgramBinary(...)`**
  - Obtains back a compiled and linked program as a binary object
- **`glProgramBinary(...)`**
  - Loads a shader binary into an allocated program object
  - Must have been created on same/„compatible“ HW/SW
Shaders (V)

- **Shaders have uniform parameters (instance variables)**
  - May be set to change shader behavior (diffuse color, matrix, …)
    - May be allocated in blocks, stored in a uniform buffer (on the GPU)
  - `glGetUniformLocation(program, name)`
    - Returns the *slot* used for a specific named shader variable
    - Can be used to pass data to the shader through `glVertexAttribPointer`
  - `glUniform*(location, …)`
    - Changes that parameter value

- **Per-vertex data can be send to a program by:**
  - Applications do not necessarily know the shader in advance
  - `glGetActiveAttribute(program, index, …)`
    - Returns information about the attribute at given index
      - Name, type, size of the specified attribute at “index”
  - `glGetAttribLocation(program, name)`
    - Returns the *slot* used for a specific named shader variable
    - Use to send vertex data to the shader through `glVertexAttribPointer()`
  - `glBindAttribLocation(program, index, name)`
    - Assigns the given index to the named attribute
    - Used by next linking process.
  - Binding of names to locations can also be specified in shader code
Shaders (VI): Example

- **Shader Variables**
  
  ```
  uniform float specIntensity;
  uniform vec4 specColor;
  uniform vec4 colors[3];
  ```

- **Access from OpenGL application**
  
  ```
  GLint loc1, loc2, loc3;
  float specIntensity = 0.98;
  float sc[4] = {0.8,0.8,0.8,1.0};
  float colors[12] = {0.4,0.4,0.8,1.0, 0.2,0.2,0.4,1.0, 0.1,0.1,0.1,1.0};

  loc1 = glGetUniformLocation(program,"specIntensity");
  glUniform1f(loc1, specIntensity);
  loc2 = glGetUniformLocation(program,"specColor");
  glUniform4fv(loc2, 1, sc);
  loc3 = glGetUniformLocation(program,"colors");
  glUniform4fv(loc3, 3, colors);
  ```
Complete OpenGL Pipeline (4.5)
Rasterization

- **Rasterization: Generating *fragments* from *primitives***
  - For every covered pixel
    - And potentially many subpixels “samples” within a pixel
  - Computes fragment data by interpolation over triangle
    - All attributes and Z/depth
    - At center (*centroid*) or at true sample position
    - Can be perspectively correct (*smooth*) or *linear in image* space

- **Different rasterization approaches**
  - For points, lines, and triangles (see spec)

- **Backface culling of triangles**
  - Must be first enabled by `glEnable(GL_CULL_FACE)`
  - `glFrontFace(dir)`
    - Defines which triangles are front facing *CLW/CCW* (in screen space)
  - `glCullFace(mode)`
    - Defines which triangles are culled: *FRONT, BACK, both*
Rasterization (II)

- **Strict ordering**
  - Primitives are rasterized as they proceed through the pipeline
    - But pipeline may actually consist of multiple parallel HW pipelines
  - Results must be as if rasterized in order as send by application
    - Requires synchronization between HW pipelines
    - Complicates scalability in HW
Complete OpenGL Pipeline (4.5)
Texturing

• Generating a new texture object
  – `glGenTexture(count, &texture)`

• Each shader can have up to N “textures image units” (128)
  – Selected with `glActiveTexture(GL_TEXTURE0 + i)`

• Binding of texture objects to a unit
  – `glBindTexture(target, texture)`
    • Target: one of
      – `TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_BUFFER, TEXTURE_CUBE_MAP, TEXTURE_2D_MULTISAMPLE, and TEXTURE_2D_MULTISAMPLE_ARRAY`

• Assignment to shader “sampler” variable with
  – `idx = glGetUniformLocation(prog, name)`
  – `glUniform1i(idx, texture)`

• How textures are used is solely the job of the shader
Specifying a Texture

• Definition of Layout in Memory
  – `glPixelStore(param_name, value)`
    • See table below for which parameters define the layout

• Defining texture data
  – `glTexImage3D(target, level, internal_fmt, w, h, d, 0, format, type, data)`
  – `glTexImage2D(target, level, internal_fmt, w, h, 0, format, type, data)`
  – `glTexImage1D(target, level, internal_fmt, w, 0, format, type, data)`
  – *SubImage*: (Re-)define only a part of the texture at given offset
    • level: Mipmaps, array index, or face of a cubemap
    • internal_fmt: One of many formats for storing texture internally
    • w, h, d: width, height, depth; (0 for border width, which must be zero)
    • format, type: see below

• Copying texture data to a GL from buffer
  – `glCopyTex(Sub)Image{1, 2, 3}D(target, level, internal_fmt, …)`
    • Copy from the frame buffer bound to GL_READ_FRAMEBUFFER

• Also
  – Compressed and multisampled formats
  – Rendering directly from texels in a buffer: `glTexBuffer()`
Texture Types, Formats, Layouts

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Valid Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNPACK_SWAP_BYTES</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>UNPACK_LSB_FIRST</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>UNPACK.Row.Length</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>UNPACK.Skip.Rows</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>UNPACK.Skip.Pixels</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>UNPACK.Alignment</td>
<td>integer</td>
<td>4, 1, 2, 4, 8</td>
<td></td>
</tr>
<tr>
<td>UNPACK.Skip.Height</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
</tbody>
</table>

Texture data type in user memory (incomplete)

| Texture data format in user memory (incomplete) |

Image layout in user memory (PixelStore)

<table>
<thead>
<tr>
<th>Format Name</th>
<th>Element Meaning and Order</th>
<th>Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED, GREEN, BLUE</td>
<td>R, G, B</td>
<td>Color</td>
</tr>
<tr>
<td>RGBA, BGRA</td>
<td>R, G, B, A</td>
<td>Color</td>
</tr>
<tr>
<td>RGB, RGBA</td>
<td>R, G, B, A</td>
<td>Color</td>
</tr>
<tr>
<td>RED_INTEGER, GREEN_INTEGER, BLUE_INTEGER, RG_INTEGER, RGB_INTEGER, RGBA_INTEGER, BGRA_INTEGER, BGR_INTEGER, BG_INTEGER, R_INTEGER, I_INTEGER</td>
<td>iR, iG, iB, iA, iB, iG, iR, iB, iA</td>
<td>Color</td>
</tr>
</tbody>
</table>

Texture data type in user memory (incomplete)
### Texture Types, Formats, Layouts

#### Sized internal color formats continued from previous page

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>R bits</th>
<th>G bits</th>
<th>B bits</th>
<th>A bits</th>
<th>Shared bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB_SNORM</td>
<td>RG</td>
<td>s8</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16</td>
<td>RG</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16_SNORM</td>
<td>RG</td>
<td>s16</td>
<td>s16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3_G3_B2</td>
<td>RGB</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB4</td>
<td>RGB</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB5</td>
<td>RGB</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB565</td>
<td>RGB</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8</td>
<td>RGB</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8_SNORM</td>
<td>RGB</td>
<td>s8</td>
<td>s8</td>
<td>s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB10</td>
<td>RGB</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB12</td>
<td>RGB</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16</td>
<td>RGB</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16_SNORM</td>
<td>RGB</td>
<td>s16</td>
<td>s16</td>
<td>s16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA2</td>
<td>RGBA</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RGBA4</td>
<td>RGBA</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RGBA4_A1</td>
<td>RGBA</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RGBA8</td>
<td>RGBA</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RGBA8_SNORM</td>
<td>RGBA</td>
<td>s8</td>
<td>s8</td>
<td>s8</td>
<td>s8</td>
<td></td>
</tr>
<tr>
<td>RGB10_A2</td>
<td>RGBA</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RGB10_A2UI</td>
<td>RGBA</td>
<td>ui10</td>
<td>ui10</td>
<td>ui10</td>
<td>ui2</td>
<td></td>
</tr>
<tr>
<td>RGBA12</td>
<td>RGBA</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>RGBA16</td>
<td>RGBA</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>RGBA16_SNORM</td>
<td>RGBA</td>
<td>s16</td>
<td>s16</td>
<td>s16</td>
<td>s16</td>
<td></td>
</tr>
<tr>
<td>SRGB8</td>
<td>RGB</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>SRGB8_ALPHA8</td>
<td>RGBA</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R16F</td>
<td>RED</td>
<td>f16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16F</td>
<td>RG</td>
<td>f16</td>
<td>f16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16F</td>
<td>RGB</td>
<td>f16</td>
<td>f16</td>
<td>f16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA16F</td>
<td>RGBA</td>
<td>f16</td>
<td>f16</td>
<td>f16</td>
<td>f16</td>
<td></td>
</tr>
<tr>
<td>R32F</td>
<td>RED</td>
<td>f32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG32F</td>
<td>RG</td>
<td>f32</td>
<td>f32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB32F</td>
<td>RGB</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA32F</td>
<td>RGBA</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td></td>
</tr>
<tr>
<td>R11F_G11F_B10F</td>
<td>RGB</td>
<td>f11</td>
<td>f11</td>
<td>f10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Sized internal color formats continued on next page*
Texture Parameters & Objects

• **Changed via**
  – `glTexParameteri(target, param_name, value)`

• **Type of parameters**
  – Wrap-mode in s, t, r: clamp (edge/border), repeat, mirror (alternately)
  – `Min_Filter`: NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, to LINEAR_MIPMAP_LINEAR
  – `Mag_Filter`: NEAREST, LINEAR
  – LOD/Mipmap parameter
  – Compare function for Z comparison (depth texture only)

• **But see Texture Sampler on next slide**
Texture Samplers

- **New in OpenGL 4.X**
  - Two aspects of a texture: The data and how it is to be used
  - Previously a texture object specified both
  - Better reuse if they can be separated

- **Texture Sampler**
  - Specify how the texture data (in a texture Object) should be used
  - Single Sampler can be attached to many units

- **Allocate new/delete texture Sampler**
  - `glGenSampler(...)`, `glDeleteSampler()`

- **Bind a Sampler to a Texture**
  - `glBindSampler(unit, sampler)`
  - Its parameters supersedes those of the texture object

- **Specify Sampler parameters**
  - `glSamplerParameter(...)`
  - Defines: Wrap mode, Filter, LOD, depth comparison
Complete OpenGL Pipeline (4.5)
Per-Fragment Operations

Fragment (or sample) + Associated Data

- Pixel Ownership Test
- Scissor Test
- Multisample Fragment Operations

- Blending
- Occlusion Query
- Depth Buffer Test
- Stencil Test

- sRGB Conversion
- Dithering
- Logic Operations

To Framebuffer

Framebuffer 

Framebuffer 

Framebuffer 

Framebuffer
Per-Fragment Operations

- Consists of multiple steps
- **Pixel ownership test (internal)**
  - Does the pixel belong to this window (might be covered)
- **Scissor test**
  - Is the pixel within a box defined by `glScissor(l, b, r, t)`
- **Multi-sample Fragment Operations**
  - Merge the information of sub-samples in a pixel into a final value
  - Includes an “alpha test” (binary transparency)
    - Ignores (sub-)fragments with an alpha value below some threshold
- **Stencil Operation (see below)**
- **Depth Buffer Test**
  - Tests if the fragment z value passes the depth stored at the pixel
- **Occlusion Query (see below)**
- **Blend operations (see below)**
  - Merge fragments with content of the frame buffer
Stencil and Depth Test

• **Function**
  – Compares value stored in stencil buffer for fragment/pixel
  – If test fails, fragment is discarded
  – Finally, applies operation based on three possible tests
    • $sfail$: Stencil tests failed
    • $dfail$: Stencil test passed, but depth test failed
    • $dpass$: Stencil and depth test passed
  – E.g. used for ShadowVolume algorithms

• **Specification**
  – `glStencilFunc(enum func, int ref, uint mask)`
    • `func`: ALWAYS, NEVER, LESS, LEQUAL, GEQUAL, GREATER, NOTEQUAL
    • `ref`: reference value
    • `mask`: ANDed with both stencil and reference value before comparison
  – `glStencilOp(sfail, dfail, dpass)`
    • Operations: KEEP, ZERO, REPLACE, INC, DEC, INVERT, INCR_WRAP, DEC_WRAP

• **Depth Test**
  – Comparison to the per-pixel value stored in depth buffer
  – `glDepthFunc(func)`
    • Compares fragment z with with content of depth buffer (func: same as stencil)
    • If test passes, overwrites old depth value with fragment depth
Fragment Tests

- **Fragment tests (like stencil and Z)**
  - Require per pixel read operations (high bandwidth)
  - May require per pixel write operations
    - Read-Modify-Write operations – can be expensive (but cached in tiles)
    - Again synchronization issues with multiple pipelines
  - Tests occur late in the pipeline
  - Might have spend significant processing on the data already
    - Should perform tests earlier without violating OpenGL semantics
    - Often can be conservatively pulled forward to right after rasterization
    - E.g. Some form of hierarchical Z-buffer (often called “Early-Z-test”)

- **Occlusion culling (e.g. ViewFrustum Culling)**
  - Must be done at application level (not in HW on GPU)
    - Replicated visibility computation in the application (mostly coarse)
    - Avoids bandwidth to graphics engine completely, but uses CPU
  - Early Z test after rasterization
    - Can cull fragments if known to be occluded (some addition cost)
    - Best if rendering is front-to-back
Occlusion Queries

• Counting the number of passed depth tests
  – Generate Counters: `glGenQueries(int n, int* ids)`
  – Wrap drawing calls in `glBeginQuery(id)/glEndQuery(id)`
  – Can later query the value with `glGetQueryiv()`

• Use for conditional rendering
  – Wrap drawing calls that should be omitted if OC fails in:
    • `glBeginConditionalRender(), glEndConditionalRender()`
    • Will be skipped if OC failed (no fragments passed the depth test)
    • Can specify what happens if OC not ready yet (wait, draw)
  – Can be used to do (limited) frustum culling on the GPU
**Blending**

- **Merging fragment and frame buffer pixel**
  - Weighted combination of *source* (S, fragment) and *destination* (D, frame buffer pixel)
  - E.g. used for semi transparent rendering (ordered in depth!)

- **Specifying the blend equation, function, and constant**
  - `glBlendEquation{,Separate}(mode {,alpha_mode})`
  - `glBlendFunc{,Separate}(src, dst {,alpha_src, alpha_dst})`
  - `glBlendColor(red, green ,blue, alpha)` specifies constant C
  - *Separate* allows to set blending separately for color/alpha

<table>
<thead>
<tr>
<th>Mode</th>
<th>RGB Components</th>
<th>Alpha Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNC_ADD</td>
<td>( R = R_s \cdot S_i + R_d \cdot D_i )</td>
<td>( A = A_s \cdot S_i + A_d \cdot D_i )</td>
</tr>
<tr>
<td>FUNC_SUBTRACT</td>
<td>( R = R_s \cdot S_i - R_d \cdot D_i )</td>
<td>( A = A_s \cdot S_i - A_d \cdot D_i )</td>
</tr>
<tr>
<td>FUNC_REVERSE_SUBTRACT</td>
<td>( R = G_d \cdot D_i - G_s \cdot S_i )</td>
<td>( A = A_d \cdot D_i - A_s \cdot S_i )</td>
</tr>
<tr>
<td>MIN</td>
<td>( R = \min(R_s, R_d) )</td>
<td>( A = \min(A_s, A_d) )</td>
</tr>
<tr>
<td>MAX</td>
<td>( R = \max(R_s, R_d) )</td>
<td>( A = \max(A_s, A_d) )</td>
</tr>
</tbody>
</table>

\( S_i \) and \( D_i \) are the weights from blend functions \( \rightarrow \)

<table>
<thead>
<tr>
<th>Function</th>
<th>RGB Blend Factors ((S_r, S_g, S_b)) or ((D_r, D_g, D_b))</th>
<th>Alpha Blend Factor (S_a) or (D_a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERO</td>
<td>((0, 0, 0))</td>
<td>0</td>
</tr>
<tr>
<td>ONE</td>
<td>((1, 1, 1))</td>
<td>1</td>
</tr>
<tr>
<td>SRC_COLOR</td>
<td>((R_s, G_s, B_s))</td>
<td>(A_s)</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_COLOR</td>
<td>((1, 1, 1) - (R_s, G_s, B_s))</td>
<td>(1 - A_s)</td>
</tr>
<tr>
<td>DST_COLOR</td>
<td>((R_d, G_d, B_d))</td>
<td>(A_d)</td>
</tr>
<tr>
<td>ONE_MINUS_DST_COLOR</td>
<td>((1, 1, 1) - (R_d, G_d, B_d))</td>
<td>(1 - A_d)</td>
</tr>
<tr>
<td>SRC_ALPHA</td>
<td>((A_s, A_s, A_s))</td>
<td>(A_s)</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>((1, 1, 1) - (A_s, A_s, A_s))</td>
<td>(1 - A_s)</td>
</tr>
<tr>
<td>DST_ALPHA</td>
<td>((A_d, A_d, A_d))</td>
<td>(A_d)</td>
</tr>
<tr>
<td>ONE_MINUS_DST_ALPHA</td>
<td>((1, 1, 1) - (A_d, A_d, A_d))</td>
<td>(1 - A_d)</td>
</tr>
<tr>
<td>CONSTANT_COLOR</td>
<td>((R_s, G_s, B_s))</td>
<td>(A_c)</td>
</tr>
<tr>
<td>ONE_MINUS_CONSTANT_COLOR</td>
<td>((1, 1, 1) - (R_s, G_s, B_s))</td>
<td>(1 - A_c)</td>
</tr>
<tr>
<td>CONSTANT_ALPHA</td>
<td>((A_s, A_s, A_s))</td>
<td>(A_s)</td>
</tr>
<tr>
<td>ONE_MINUS_CONSTANT_ALPHA</td>
<td>((1, 1, 1) - (A_s, A_s, A_s))</td>
<td>(1 - A_s)</td>
</tr>
<tr>
<td>SRC_ALPHA_SATURATE</td>
<td>((f, f, f))</td>
<td>1</td>
</tr>
</tbody>
</table>
sRGB, Dithering, Logic Ops

• **sRGB conversion**
  – Performed if the frame buffer is specified to be in sRGB
    • Non-linear mapping with overall gamma ~ 1/2.2 (with linear base)
    • Inverse conversion used for input from textures in sRGB format

• **Dithering**
  – Round each color component
    • Round to either the larger or smaller representable value
    • Decision based only on pixel position (rounding bias)
  – Trades color resolution versus spatial resolution
    • Eye averages over neighboring pixels anyway
  – `glEnable/Disable(GL_DITHER)`

• **Logic Ops**
  – Combine fragment (s) and frame buffer pixel (d) with logic operation
    • `glLogicOp(op)`

```
<table>
<thead>
<tr>
<th>Argument value</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEAR</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>s ∧ d</td>
</tr>
<tr>
<td>AND_REVERSE</td>
<td>s ∧ ¬d</td>
</tr>
<tr>
<td>COPY</td>
<td>s</td>
</tr>
<tr>
<td>AND_INVERTED</td>
<td>¬s ∧ d</td>
</tr>
<tr>
<td>NOOP</td>
<td>d</td>
</tr>
<tr>
<td>XOR</td>
<td>s xor d</td>
</tr>
<tr>
<td>OR</td>
<td>s ∨ d</td>
</tr>
<tr>
<td>NOR</td>
<td>¬(s ∨ d)</td>
</tr>
<tr>
<td>EQUIV</td>
<td>¬(s xor d)</td>
</tr>
<tr>
<td>INVERT</td>
<td>¬d</td>
</tr>
<tr>
<td>OR_REVERSE</td>
<td>s ∨ ¬d</td>
</tr>
<tr>
<td>COPY_INVERTED</td>
<td>¬s</td>
</tr>
<tr>
<td>OR_INVERTED</td>
<td>¬s ∨ d</td>
</tr>
<tr>
<td>NAND</td>
<td>¬(s ∧ d)</td>
</tr>
<tr>
<td>SET</td>
<td>all 1's</td>
</tr>
</tbody>
</table>
```
OpenGL and Frame Buffers

• **OpenGL system frame buffers**
  – Provide memory for storing data for every pixel
    • Color and optionally: depth (Z), stencil, window-id, and others
  – Format must be fixed before windows are opened
    • Window-System specific: `glXGetFBConfigs()`

• **Color buffers**
  – RGBA (RGB+Alpha)
    • Alpha stores transparency/coverage information
    • Today often 8/8/8(/8) bits (10 bit becoming more popular)
    • Recent GPUs also support 16 bit fix and 16/24/32 bit float components
  – Double buffering option (back- and front buffer)
    • Animations: draw into back, display front
    • No flashing or tearing artifacts when swapped between frames
    • Swap buffers during vertical retrace (`glXSwapBuffers()`) or asap.
    • New monitors support “Adaptive Sync” to send FB when ready (w/ limits)
      – No longer limited to fixed frame rate; extensions even allow controlled timing
  – Stereo option (possibly quad buffered)
    • Left and right buffers (also with DB), e.g. for two projectors
    • Requires support from GUI
OpenGL and Frame Buffers

• **Depth/Z buffer**
  – Stores depth/Z coordinate of visible geometry per pixel
  – Used for occlusion test (Z-test)

• **Stencil buffer**
  – Small integer variable per pixel
  – Used for masking fragment operations
  – Write operations based on fragment tests
    • Set/increment/decrement variable

• **Application-defined frame buffers**
  – Application can define any number of additional pixel buffer objects
  – And bind them to frame buffer objects
Draw Buffers

- **Specifying which buffer to render to**
  - `glDrawBuffer(enum buffer)`
  - `glDrawBuffers(int size, enum* buffers)`
    - All drawing operation will be directed to the indicated buffers

- **Enabling specific color planes**
  - `glColorMask(bool r, g, b, a)`
  - `glColorMask(uint r, g, b, a)`
  - `glDepthMask(bool mask)`
  - `glStencilMask{,Separate}(mask)`

- **Clearing the Buffer**
  - `glClear(mask)`
    - With combination of `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, and `STENCIL_BUFFER_BIT`
  - `glClearColor(r, g, b, a), glClearDepth(depth), glClearStencil(int s)`
    - Specifies the color to set the buffer when performing a clear
    - Must be extremely efficient as it touches all pixel but does nothing useful (special HW in the memory path for this)
Frame buffer & Render buffer

- **Definition**
  - Render buffer: Memory for color, stencil, or depth buffer
  - Frame buffer: A combination of the above

- **Generate/delete own RenderBuffer object**
  - `glGenRenderBuffer(int n, int* ids), glDeleteRenderBuffers(n, ids)`

- **Binding**
  - `glBindRenderBuffer(GL_RENDERBUFFER, id)`

- **Allocate memory for a Renderbuffer**
  - `glRenderBufferStorage(GL_RENDERBUFFER, format, w,h)`

- **Generate/delete a new Framebuffer object**
  - `glGenFramebuffers(int n, int* ids) glDeleteFramebuffers(n , ids)`

- **Bind a Framebuffer object for rendering**
  - `glBindFramebuffer(fb_target, fb_id)`
    - `fb_target == GL_DRAW_FRAMEBUFFER/GL_READ_FRAMEBUFFER`
      - Framebuffer will be used for drawing into or reading from it
    - Default framebuffer has id == 0
Framebuffer Attachment

• Attaching a render buffer to a frame buffer
  – `glFramebufferRenderbuffer(fb_target, attach, rb_target, rb_id)`
    • `attach`: GL_{COLOR, DEPTH, STENCIL, DEPTH_STENCIL}_ATTACHMENT
    • `fb_target`: GL_{DRAW, READ}_FRAMEBUFFER
    • `rb_target`: GL_RENDERBUFFER

• Attaching a texture to a frame buffer
  – `glFramebufferTexture(fb_target, attach, texture_id, level)`
    • Level: Mipmaplevel, side of a cube, z-layer in 3D texture
    • Undefined behavior results if
      • A texture is bound for an active frame buffer and to a texture unit
      • A texture is bound for reading and writing in a copy operation
Reading Pixels Back

- **Reading from the framebuffer**
  - `glReadPixels(x, y, w, h, format, type, data)`
  - Reads from the framebuffer bound to `GL_READ_FRAMEBUFFER`

```
RGBA pixel data in

Convert to float

Clamp to [0,1]

Pack

byte, short, int, float, or packed pixel component data stream

Pixel Storage Operations
```
Special Functions

- **glFlush()**
  - Makes sure that all previous commands get sent to the GPU

- **glFinish()**
  - Waits until all previous commands have executed

- **sync = glFenceSync(cond, 0)**
  - Send a sync command in the pipeline
    - cond = SYNC_GPU_COMMANDS_COMPLETE
  - Creates sync object that can later be waited upon with

- **glClientWaitSync(sync, flags, timeout)**
- **glWaitSync(sync, flags, timeout)**
  - Waits in the client or the server
  - Wait in the server is more efficient as commands can already be sent

- **glHint(target, hint)**
  - Allows to tell OpenGL what quality we would like to see

- **glGet*(...)**
  - Querying the state of OpenGL
OpenGL Guaranties

• **Non Guaranties**
  – Many rules as how things must be rendered
  – No exact rule for implementation of graphics operations
    • Such as number of bits, coverage by a primitive, etc.
  – Different implementations can differ on a per-pixel basis

• **Invariants**
  – Invariants within an implementation
    • Same output when given the same input
    • Fragment values are independent of
  – Content of frame buffer
  – Active color buffer, ...
    • Independence of parameter values (e.g. for stencil / blending)
  – No invariance when switching options on and off
    • E.g. depth test, stencil, texturing, ...
    • On-screen versus off-screen buffers