Computer Graphics

– OpenGL –

Philipp Slusallek
History: Graphics Hardware

- **Graphics in the ‘80ies**
  - Framebuffer was a designated memory area in RAM
  - “HW“: Set individual pixels directly via memory access
    - Peek & poke, getpixel & putpixel, …
    - MDA ('81: text only but 720x350 resolution, monochrome, 4 kB of RAM!)
      - Character code was index into bit pattern in ROM for each character
  - CGA ('81: 160x200:
    - 16 colors 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), with BIOS extension
    - Everything done on the CPU
      - Except for driving the display output
History: Graphics Hardware (II)

• **Today (Nvidia Ampere Flagship GA 102, RTX 3090)**
  – Discrete graphics card via high-speed link
    • e.g. PCIe-4.0 x16: up to 64 GB/s transfer rate
  – Autonomous, high-performance GPU (more powerful than CPU)
    • 10,496 SIMD processors
    • Up to 24GB of local GDDR6X RAM (A6000: 48 GB, x2 via NVLink)
    • 936 GB/s memory bandwidth
    • 35.6 TFLOPS 16bit floats
    • 35.6 TFLOPS single precision (SP) + ? TFLOPS doubles (DP)
    • 35.6/142/284/568 TFLOPS in FP32/16/Int8/4 via 328 Tensor Cores (RTX)
    • ~20 GigaRays/s, 84 RT Cores
    • Dedicated ray tracing HW unit (BVH traversal & tri. interpol & intersect)
    • Total of 28.3 Billion transistors at 350 Watt
  – Performs all low-level tasks & a lot of high-level tasks
    • Clipping, rasterization, hidden surface removal, … + Ray Tracing
    • Procedural geometry, shading, texturing, animation, simulation, …
    • Video rendering, de- and encoding, deinterlacing, …
    • Full programmability at several pipeline stages
    • Deep Learning & Matrix-Multiply (sparse x2): Training and Inference
Nvidia GA102 GPU
History: Graphics APIs

• 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 → highly efficient
  – Orthogonal design and extensible
    • Common interface from mobile phone to supercomputer
    • Major redesign & cleanup, deprecated and removed functionality
    • Since Version 3.2: Profiles (core, compatibility, forward compatibility)
    • OpenCL for compute, tessellation shaders, 64 bit variables, multi-viewpoint
    • 4.3: Compute shaders, adv. texture compression, ...
    • 4.5: Direct state access, compatibility to OGL ES3.1, ...
History: Graphics APIs (II)

• **Direct3D (Microsoft, Part of DirectX multimedia APIs)**
  – Started as *Reality Labs* by RenderMorphics, bought by MS (SW focus)
  – First version in 1996, Retained & Immediate Mode API
  – Played catch-up to OpenGL until Direct3D 6.0 (1998)
  – Advanced fast & significantly 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 idea into mainline gaming

• **Cross-platform API with Vulkan (Khronos, since 2016)**
  – Much lower level, requires expert programmer, …
  – Vulkan is the way to go for most applications and products
Meta Discussion

• **Why teach an API, like OpenGL?**
  – We are not typically doing this at a university (focus on principles)?
    • Yes, but Rasterization on GPUs is a key application topic in graphics, and we should not ignore it
    • Talking about the principles would mostly address the same topics, but only in a more abstract form
    • So, chose a concrete API to show the same principles behind it
  – But APIs change quite often?
    • Yes, but the principles behind it change only slowly, focus on those
  – Multiple APIs available, so which one should we teach?
    • OpenGL is cross-platform and most widely available
  – Why not use OpenGL’s successor, Vulkan?
    • It is much lower level and not well-suited for teaching
    • Requires good understanding of low-level features, e.g. on HW-level
Introduction to OpenGL

• **What is OpenGL?**
  – Cross-platform, low-level software API for graphics HW (GPUs)
  – Controlled by Khronos ([https://www.khronos.org/](https://www.khronos.org/))
  – Only covers 2D/3D rendering (points, lines, triangles, …)
  – Related APIs: Vulkan, MS Direct3D, Apple Metal
    • Related GUI APIs: X11, 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 – X11extension, allowed 3D rendering on remote display
      – Only in X11 environment!, now deprecated
Related APIs and Languages

• **glsI** (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 the 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 (now also WebGPU)

• **GUI-Toolkits**
  – QT: QtGLWidget class, Gtk: GtkGLExt widget
  – GLUT (OpenGL Utility Toolkit, older but still useful)
  – SDL: Simple DirectMedia Layer (more modern than GLUT, w/ audio)
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

• Tutorials
  – Learn OpenGL: https://learnopengl.com/
  – Lighthouse3D (older): http://www.lighthouse3d.com

• WebGL
  – WebGL PlayGround: http://webglplayground.net/
    • 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 an 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 before rasterization (add/delete)

- **Providing Data Through Vertex Arrays**
  - Each vertex consists of the *position data* plus 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, …)`
  - `glVertexAttribAttribPointer(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 beginning of array with the 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)

• **Advanced geometry types** (seldomly used, obsolete with Mesh Shaders)
  - With adjacency: Additional vertices around a primitive
    • Lines, Line Strips, Triangles, TriangleStrips
  - 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
– (Complex calls may now be replaced by mesh shaders)
Buffers

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

- **Types of BufferBindings**

  - `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

<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.
Using Buffers

• **glVertexAttribPointer** and all drawing calls use the currently bound buffer (if any)
  – 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)

• Advanced: 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 glVertexAttribPointer()
    – 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
    - Receive “uniform” parameters for the shader
    - “Attributes” from 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 (after rasterization)**
  - 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 framebuffers 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

- **Newest Addition: Mesh (+ task) shaders**
  - Replace initial pipeline until rasterization, use compute model

- **We discuss programming shaders later**
Shaders (III)

- Shaders specify the programmable parts of a pipeline
- Different Types of shaders (vertex, fragment, geometry, etc.)
  - 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)`
  - Prepare a shader and use it for subsequent drawing calls
Shaders (IV, advanced)

- New since OpenGL4.1: 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, variable-name)`
    - Returns the uniform *slot* used for a specific named shader variable
  - `glUniform*(location, …)`
    - Changes that parameter value

- **Per-vertex attributes can be sent to a program**
  - 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
    - For sending vertex data to the shader through `glVertexAttribPointer()`
    - Binding of names to locations can be specified in shader code
  - `glBindAttribLocation(program, index, name)`
    - Alternative: Assigns the given index to the named attribute
    - Used by next linking process.
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 subpixel “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**
  - Mode must first be 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 be implemented by multiple parallel HW engines
  - Results must be as if rasterized in order as send by application
    - Requires synchronization between HW pipelines
    - Complicates scalability in HW
    - Requirement can be ignored by OpenGL if order does not matter
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 "sampler" variable in shader with
  - `idx = glGetUniformLocation(prog, name)`
  - `glUniform1i(idx, texture)`

- How textures are used is solely the job of the shader
Specifying Content for 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 on GPU
    • 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

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

### Texture Data Types in User Memory

<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</td>
<td>1,2,4,8</td>
</tr>
<tr>
<td>UNPACK_IMAGE_HEIGHT</td>
<td>integer</td>
<td>0</td>
<td>[0, ∞)</td>
</tr>
<tr>
<td>UNPACK_SKIP_IMAGES</td>
<td>integer</td>
<td>0</td>
<td>[0, ∞)</td>
</tr>
</tbody>
</table>

### Texture Data Formats in User Memory

<table>
<thead>
<tr>
<th>Type Parameter Token Name</th>
<th>Corresponding GL Data Type</th>
<th>Special Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE</td>
<td>ubyte</td>
<td>No</td>
</tr>
<tr>
<td>BYTE</td>
<td>byte</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td>ushort</td>
<td>No</td>
</tr>
<tr>
<td>SHORT</td>
<td>short</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td>uint</td>
<td>No</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>HALF_FLOAT</td>
<td>half</td>
<td>No</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_3_3_2</td>
<td>ubyte</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_2_3_3_REV</td>
<td>ubyte</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5_REV</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4_REV</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_5_5_1</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1_5_5_5_REV</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8_REV</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_10_10_10_2</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_2_10_10_10_REV</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_24_8</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_10F_11F_11F_REV</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_5_9_9_9_REV</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>FLOAT_32_UNSIGNED_INT_24_8_REV</td>
<td>float</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 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>STENCIL_INDEX</td>
<td>Stencil Index</td>
<td>Stencil</td>
</tr>
<tr>
<td>DEPTH_COMPONENT</td>
<td>Depth</td>
<td>Depth</td>
</tr>
<tr>
<td>DEPTH_STENCIL</td>
<td>Depth and Stencil Index</td>
<td>Depth and Stencil</td>
</tr>
<tr>
<td>RED</td>
<td>R</td>
<td>Color</td>
</tr>
<tr>
<td>GREEN</td>
<td>G</td>
<td>Color</td>
</tr>
<tr>
<td>BLUE</td>
<td>B</td>
<td>Color</td>
</tr>
<tr>
<td>RG</td>
<td>R, G</td>
<td>Color</td>
</tr>
<tr>
<td>RGB</td>
<td>R, G, B</td>
<td>Color</td>
</tr>
<tr>
<td>RGBA</td>
<td>R, G, B, A</td>
<td>Color</td>
</tr>
<tr>
<td>BGR</td>
<td>B, G, R</td>
<td>Color</td>
</tr>
<tr>
<td>BGRA</td>
<td>B, G, R, A</td>
<td>Color</td>
</tr>
<tr>
<td>RED_INTEGER</td>
<td>iR</td>
<td>Color</td>
</tr>
<tr>
<td>GREEN_INTEGER</td>
<td>iG</td>
<td>Color</td>
</tr>
<tr>
<td>BLUE_INTEGER</td>
<td>iB</td>
<td>Color</td>
</tr>
<tr>
<td>RG_INTEGER</td>
<td>iR, iG</td>
<td>Color</td>
</tr>
<tr>
<td>RGB_INTEGER</td>
<td>iR, iG, iB</td>
<td>Color</td>
</tr>
<tr>
<td>RGBA_INTEGER</td>
<td>iR, iG, iB, iA</td>
<td>Color</td>
</tr>
<tr>
<td>BGR_INTEGER</td>
<td>iB, iG, iR</td>
<td>Color</td>
</tr>
<tr>
<td>BGRA_INTEGER</td>
<td>iB, iG, iR, iA</td>
<td>Color</td>
</tr>
</tbody>
</table>

Texture data **format** in user memory (incomplete)

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

<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>RGB16_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>RGBA</td>
<td>RGBA</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>RGBA8_SNORM</td>
<td>RGB</td>
<td>s8</td>
<td>s8</td>
<td>s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA10</td>
<td>RGBA</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA12</td>
<td>RGBA</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA16</td>
<td>RGBA</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA16_SNORM</td>
<td>RGBA</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>RGBA5_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>RGBA10_A2</td>
<td>RGBA</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RGBA10_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></td>
<td></td>
</tr>
<tr>
<td>SRGB8_ALPHAS8</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>RG32F</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>

### Special cases for compressed textures
- Not covered here
Texture Parameters & Objects

• Changed via
  – `glTexParam*(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 itself & 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 – or vice versa

• **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
Per-Fragment Operations

- Consists of multiple steps
- Pixel ownership test (internal)
  - Does the pixel belong to this window (might be covered by others)
- 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 and depth buffer for each fragment/pixel
  - If test fails, fragment is discarded
  - Finally, applies stencil 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 (e.g., counting f/b-facing fragments)

- **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 the 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, parallel 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
  - Early Z test, right after rasterization
    - E.g., some form of hierarchical Z-buffer (often called “Early-Z-test”)
    - Can cull fragments if known to be occluded (some addition cost)
    - Best if rendering is front-to-back (and Z is not modified in shader!)

- **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
    - *Can now be implemented on GPU with new mesh shaders!*
Occlusion Queries

- **Counting the number of passed depth tests**
  - Generate Counters: `glGenQueries(int n, int* ids)`
  - Wrap drawing calls in `glBeginQuery(id)/glEndQuery(id)`
    - E.g., for drawing of bounding box of a complex part of the scene
  - Can later query the value with `glGetQueryiv()`

- **Use for conditional rendering**
  - Wrap drawing calls that should be omitted if OQ fails in:
    - `glBeginConditionalRender(), glEndConditionalRender()`
      - E.g., drawing of complex part of the scene
    - Will be skipped if OQ failed (no fragments passed the depth test)
    - Can specify what happens if OQ not ready yet (wait, draw)
    - OQ must happen early enough that results are avail. in time
  - 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

\[ S_i \text{ and } D_i \text{ are the weights from blend functions} \]
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 threshold based on pixel position (rounding bias)
  - Trades color resolution versus spatial resolution
    - Eye averages over neighboring pixels anyway
  - Enable Mode: `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</td>
</tr>
<tr>
<td>COPY_INVERTED</td>
<td>¬s ∨ d</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, e.g.: `glXGetFBConfigs()`, `eglGetConfigs()`

- **Color buffers**
  - RGBA (RGB+Alpha)
    - Alpha stores transparency/coverage information
    - Today often 8/8/8(/8) bits (10 or even 12 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 buffer
    - No flashing or tearing artifacts when swapped in 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 operations 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 would have to touch all pixels 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 Attachement

• 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

Clam to [0,1]

Pack

Pixel Storage Operations

byte, short, int, float, or packed pixel component data stream
```
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 (GPU)
  - 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, but …
  – 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
    • Within clearly specified limits/rules

• Guaranty of 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 (glDisable/Enable())
    • E.g., depth test, stencil, texturing, …
    • On-screen versus off-screen buffers