VU Design & Implementation of a Rendering Engine

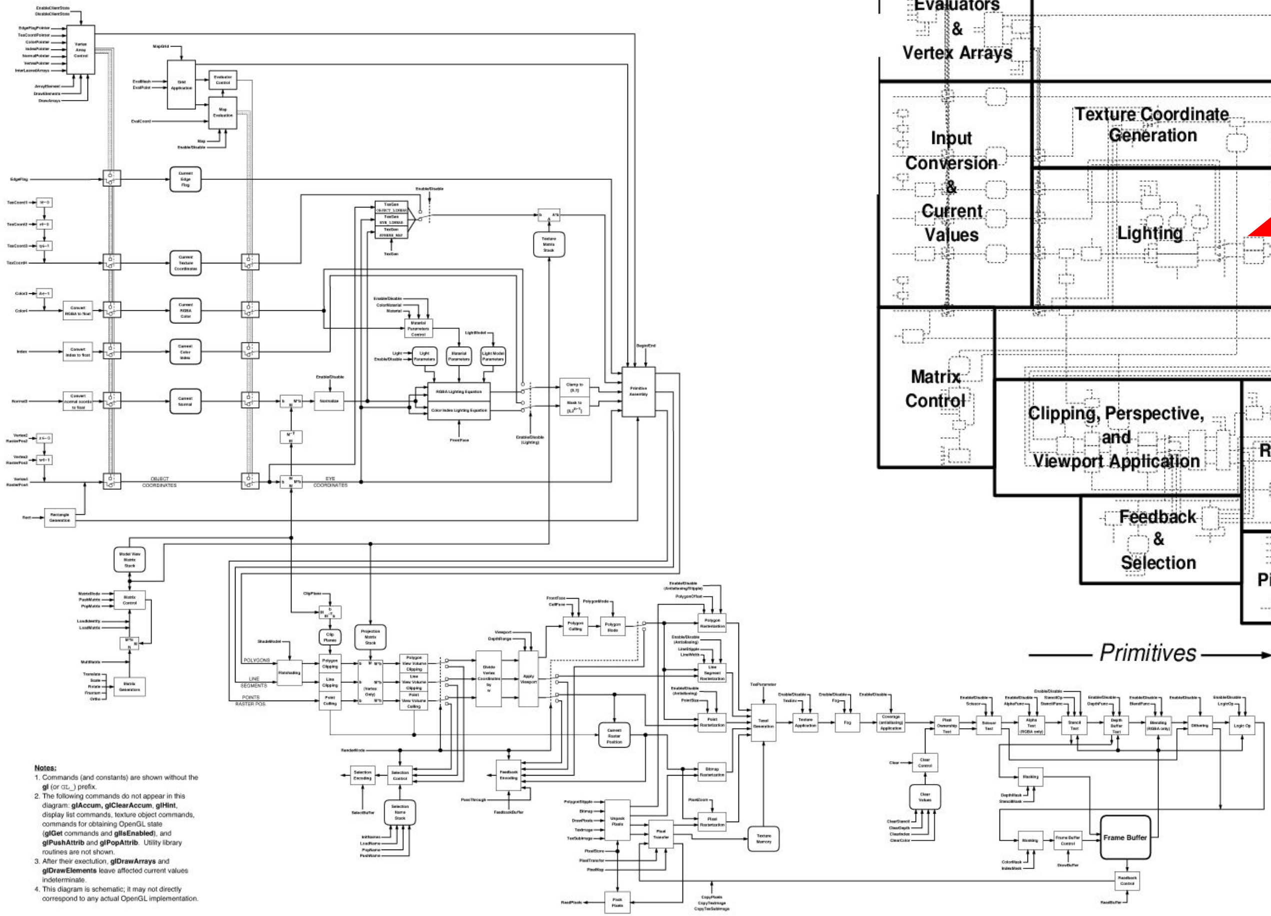
Graphics Hardware & API Insights

Outline

- Graphics APIs (OpenGL, GLES, WebGL, WebGPU, Direct3D, Vulkan) provide abstractions
- As always, abstractions have different overheads
- **Overheads** occur due to
  - **Communication** efforts
  - **Validation** efforts
  - Abstractions sometimes expose hardware features in a suboptimal manner
- Graphics APIs often failed to catch up with graphics hardware features
- The general theme is
  - Graphics hardware became **more powerful**
  - Engine features more and more **moved into the driver**
    - Example: Instancing
- Graphics API have an interesting history: looking at past development helps in understanding design trade offs….

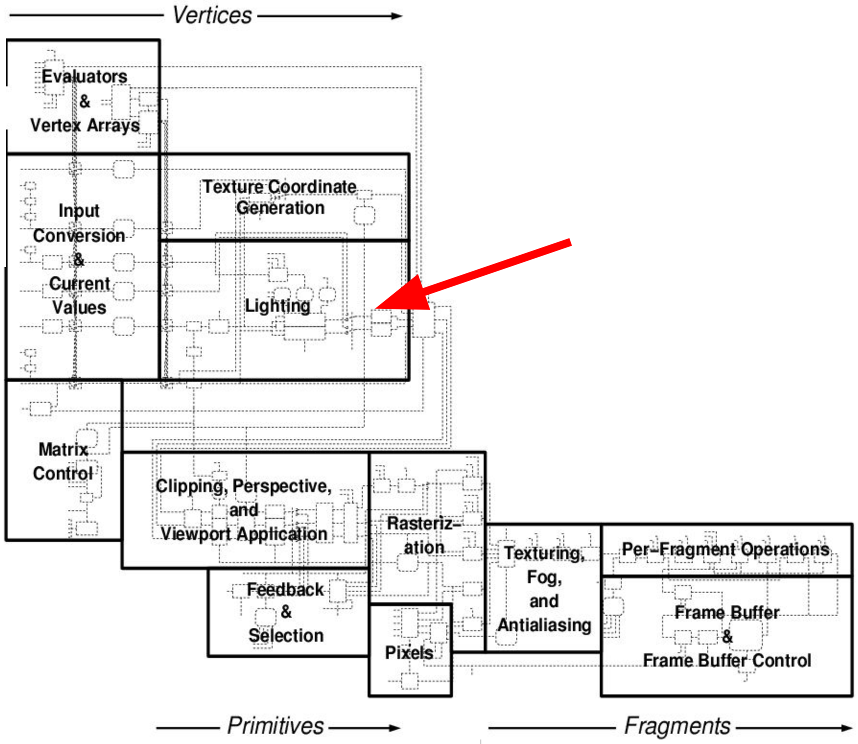
What is the right abstraction level

- Graphics API provides mechanisms for drawing primitives
- or ... Graphics API understands meshes, **lights** etc
- or … Graphics API understands complete scene graphs (scene database)
- It depends…
  - In early 2000s, graphics hardware was very restricted, light could be implemented **in the driver efficiently**, thus light specification needed to be in graphics api
  - OpenGL 3.1 “Longs Peak Reloaded” **drops fixed function pipeline**

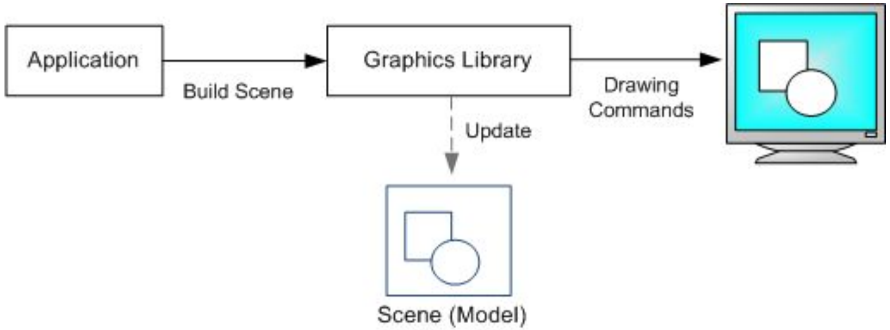


<u>**Notes:**</u>

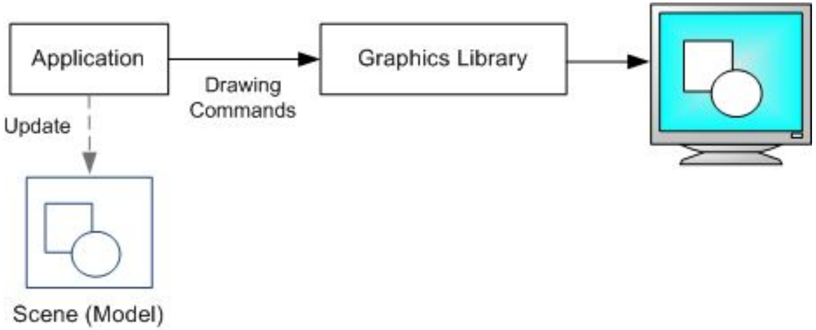
1. Commands (and constants) are shown without the **gl** (or GL_) prefix.
2. The following commands do not appear in this diagram: **glAccum, glClearAccum**, **glHint**, display list commands, texture object commands, commands for obtaining OpenGL state (**glGet** commands and **glIsEnabled**), and **glPushAttrib** and **glPopAttrib**. Utility library routines are not shown.
3. After their exectution, **glDrawArrays** and **glDrawElements** leave affected current values indeterminate.
4. This diagram is schematic; it may not directly correspond to any actual OpenGL implementation.



Direct3D 2: Retained mode vs Immediate mode



Retained mode



Immediate mode

https://msdn.microsoft.com/en-us/library/windows/desktop/ff684178(v=vs.85).aspx

Direct3D 2.0 concept

- Direct3D immediate mode
  - Commands (e.g. draw) issued to execute buffer, parameters in structs

```
c->operation = DRAW_TRIANGLE;
c->vertexes[0] = 0;
c->vertexes[1] = 1;
c->vertexes[2] = 2;
IssueExecuteBuffer (buffer);
```

- Direct3D retained mode (completely dropped later)
  - High level description of scene

```
#define IDirect3DRM_CreateMesh(p,a)
#define IDirect3DRM_CreateMeshBuilder(p,a)
#define IDirect3DRM_CreateFace(p,a)
#define IDirect3DRM_CreateAnimation(p,a)
#define IDirect3DRM_CreateAnimationSet(p,a)
#define IDirect3DRM_CreateTexture(p,a,b)
#define IDirect3DRM_CreateLightRGB(p,a,b,c,d,e)
```

First approaches of Graphics APIs

- OpenGL 1.0 1994 (according to red book)
  - Standard defined by OpenGL ARB (Architecture Review Board)
- Mid 90s, proprietary, then open source, 3dfx creates glide which runs in hardware on voodoo
  - Designed for games, small API

```
for (n=0; n<1000; n++) {
      p.x = (float) (rand() % 1024);
      p.y = (float) (rand() % 1024);
      grDrawPoint(p);
}
```

- DirectX 2.0, 1996
- DirectX 5.0, 1997
  - Added `DrawPrimitive`, no need to construct explicit command to issue draw

OpenGL vs D3D8

- Direct3D 8 (Immediate mode)

```
(psuedo code, and incomplete)
v = &buffer.vertexes[0];
v->x = 0; v->y = 0; v->z = 0; v++;
v->x = 1; v->y = 1; v->z = 0;v++;
v->x = 2; v->y = 0; v->z = 0;
c = &buffer.commands;
c->operation = DRAW_TRIANGLE;
c->vertexes[0] = 0;
c->vertexes[1] = 1;
c->vertexes[2] = 2;
IssueExecuteBuffer (buffer);
```

…"With **D3D**, you have to do everything the **painful way** from the beginning. Like writing a complete program in assembly language, taking many times longer, missing chances for algorithmic improvements, etc. And then finding out it doesn't even go faster." [Carmack 1996]

- OpenGL 1.0 (Immediate mode)

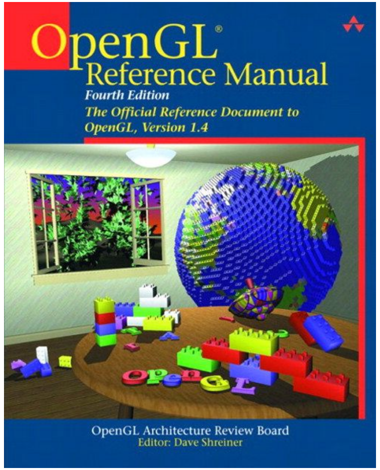
```
glBegin (GL_TRIANGLES);
glVertex (0,0,0);
glVertex (1,1,0);
glVertex (2,0,0);
glEnd ();
```

"...A month ago, I ported quake to OpenGL. It was an **extremely pleasant experience.** "
[Carmack 1996]

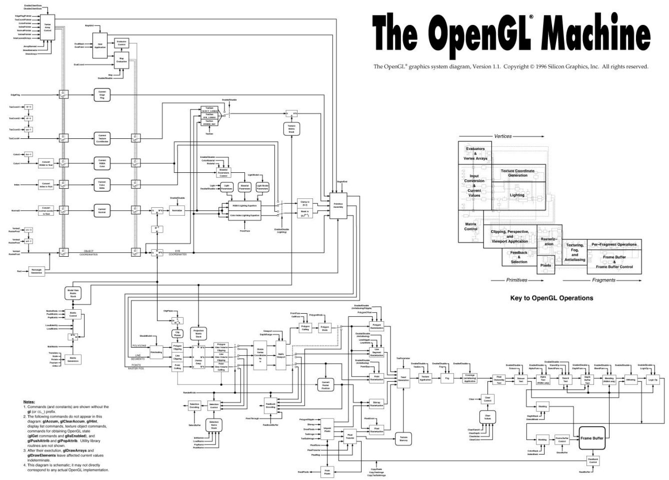
Graphics infrastructure: a distributed system

- Graphics hardware and main computing unit form distributed system
- Essentially, we are programming a co-processor
- OpenGL took this view seriously: OpenGL = Client-Server

Client in main memory



API



Server with separate memory and execution engine

https://www.lri.fr/~mbl/ENS/IG2/docs/opengl-stm.pdf

OpenGL 1.0 concept

- State machine
- Commands modify state which is expected in subsequent commands…
- Strict client/server architecture:
  - really?
- Primitives can be sent, piece by Piece to the server….

```
InitializeAWindowPlease();

  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f (1.0, 1.0, 1.0);
  glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0)
  glBegin(GL_POLYGON);
     glVertex3f (0.25, 0.25, 0.0);
     glVertex3f (0.75, 0.25, 0.0);
     glVertex3f (0.75, 0.75, 0.0);
     glVertex3f (0.25, 0.75, 0.0);
  glEnd();
  glFlush();

  UpdateTheWindowAndCheckForEvents();
```

(Client-side) Vertex Arrays (GL 1.1)

- Obviously, telling the server each vertex, piece-by-piece might be inefficient….
- First extension opening up client memory for the server

```
GLfloat vertices[] = {...}; // 36 of vertex coords
...
// activate and specify pointer to vertex array
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);

// draw a cube
glDrawArrays(GL_TRIANGLES, 0, 36);

// deactivate vertex arrays after drawing
glDisableClientState(GL_VERTEX_ARRAY);
```

OpenGL: Display Lists

- Record commands for repeated execution on server
- Great idea, but inflexible
- But: data cannot be modified

```
// create one display list
GLuint index = glGenLists(1);

// compile the display list, store a triangle in it
glNewList(index, GL COMPILE);
    glBegin(GL TRIANGLES);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glEnd();
glEndList();
...

// draw the display list
glCallList(index);
...
```

OpenGL **V**ertex **B**uffers **O**bject (VBO, ARB 2003)

- Similar to vertex arrays, but data needs to be **explicitly uploaded into GPU** memory

```
//Create a new VBO and use the variable id to store the VBO id
glGenBuffers(1, &triangleVBO);
//Make the new VBO active
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);
//Upload vertex data to the video device
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
//Make the new VBO active. Repeat here incase changed since initialisation
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);
//Draw Triangle from VBO - do each time window, view point or data changes
//Establish its 3 coordinates per vertex with zero stride in this array; necessary here
glVertexPointer(3, GL_FLOAT, 0, NULL);
//Establish array contains vertices (not normals, colours, texture coords etc)
glEnableClientState(GL_VERTEX_ARRAY);
//Actually draw the triangle, giving the number of vertices provided
glDrawArrays(GL_TRIANGLES, 0, sizeof(data) / sizeof(float) / 3);
```

OpenGL **V**ertex **A**rray **O**bjects (VAO)

- Bindings for a draw call can be packaged together in vertex array object
- Replaces repeated

```
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(TVertex), 0);  // 3
floats für Position
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(TVertex), 12); // 3
floats für den Normalenvektor
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(TVertex), 24); // 2
floats als Textur-Koordinaten
draw….
```

- With `glBindVertexArray`(.), which restores this complete state at once
- ARB Extension in 2008

Uniform buffer objects (UBO, ARB 2009)

- VAOs are for binding vertex inputs
- In GL2, glUniform is used for binding uniform values
- Calling glUniform for each attribute is SLOW
- Group parameters by frequency of change
  - Bind glBindBufferBase(GL_UNIFORM_BUFFER, 0, uboMatrix)
  - Use glBufferSubData to update the buffer (whenever necessary)
  - Actually this is an important hot optimization point: for more information see further reading.

```
// matrices
uniform mat4 matrix_world;
uniform mat4 matrix_worldIT;
// material uniform
vec4 material_diffuse;
uniform vec4 material_emissive;
```

```
layout(std140,binding=0) uniform matrixBuffer {
        mat4 matrix_world;
        mat4 matrix_worldIT;
};
layout(std140,binding=1) uniform materialBuffer {
        vec4 material_diffuse;
        vec4 material_emissive; ...
};
```

Evolution: Immediate mode considered harmful

- Clearly the trend in OpenGL's early days was:

Remove the number of API interactions...

- Vertex arrays improved performance by reducing the number of calls
- VBO improved performance because data is stored 'near' the GPU
- VAO again, reduced number of calls

Evolution: State considered harmful

- Resize a buffer:

```
var tmpBuffer = GL.GenBuffer();
GL.BindBuffer(BufferTarget.CopyWriteBuffer, tmpBuffer)
GL.BufferData(BufferTarget.CopyWriteBuffer, copyBytes, 0n, BufferUsageHint.StaticDraw)
GL.CopyBufferSubData(BufferTarget.CopyReadBuffer, BufferTarget.CopyWriteBuffer, 0n, 0n, copyBytes)
GL.BufferData(BufferTarget.CopyReadBuffer, newCapacity, 0n, BufferUsageHint.StaticDraw)
GL.CopyBufferSubData(BufferTarget.CopyWriteBuffer, BufferTarget.CopyReadBuffer, 0n, 0n, copyBytes)
GL.BindBuffer(BufferTarget.CopyWriteBuffer, 0)
GL.DeleteBuffer(tmpBuffer)
```

- Resize a buffer with **EXT_direct_state_access (2013)**

```
let tmpBuffer = GL.GenBuffer()
GL.NamedBufferData(tmpBuffer, copyBytes, 0n, BufferUsageHint.StaticDraw)
GL.NamedCopyBufferSubData(x.Handle, tmpBuffer, 0n, 0n, copyBytes)
GL.NamedBufferData(x.Handle, newCapacity, 0n, BufferUsageHint.StaticDraw)
GL.NamedCopyBufferSubData(tmpBuffer, x.Handle, 0n, 0n, copyBytes)
GL.DeleteBuffer(tmpBuffer)
```

On reducing number of API calls…

Extending the draw call API

- Instanced rendering (ARB_draw_instanced, 2008)
  - void **glDrawArraysInstanced**(GLenum *mode*, GLint *first*, GLsizei *count*, GLsizei *primcount*);

```
for (i = 0; i < primcount; i++) {
    gl_InstanceIDARB = i;
    glDrawArrays(mode, first, count);
}
gl_InstanceIDARB = 0;
```

} equivalent, but runs in the driver

- Base instanced rendering (ARB_base_instance, 2011)
  - Offset for instance attributes
  - void **glDrawArraysInstancedBaseInstance**(GLenum *mode*, GLint *first*, GLsizei *count*, GLsizei *primcount*, GLuint *baseinstance*);
  - void **glDrawElementsInstancedBaseVertexBaseInstance**(enum mode,sizei count, num t,const void *indices, sizei primcount, int basevertex,uint baseinstance);

You see where this is going ;)

Extending the draw call API (on steroids)

- **ARB_multi_draw_indirect** (2012)
- Lift (draw) command arguments into struct
- **NV_command_list** (2015) lifts more commands into buffers...
- `glMultiDrawElementsIndirect`

Hint: In your exercise you could generate an indirect buffer on the gpu using compute shaders.

Features such as (view-frustum) culling would be a great fit!

Frostbite is doing this heavily…..

```
void glMultiDrawElementsIndirect(....) {
    for (n = 0; n < drawcount; n++) {
        const DrawElementsIndirectCommand *cmd;
        if (stride != 0) {
            cmd = (const DrawElementsIndirectCommand  *)((uintptr)indirect + n * stride);
        } else {
            cmd = (const DrawElementsIndirectCommand  *)indirect + n;
        }

        glDrawElementsInstancedBaseVertexBaseInstance(mode,cmd->count,type, cmd->firstIndex *
                               size-of-type, cmd->instanceCount, cmd->baseVertex, cmd->baseInstance);
    }
}
```

The evolution of OpenGL/Direct3D

- More **specialization** in the driver
  - Gradually features moved into the driver
  - Draw calls became more expressive
  - Binding points were reduced
- More **flexibility** of the graphics pipeline
  - Fixed function pipeline deprecated since OpenGL 3.3
- Similar story for Direct3D
  - D3D always preferred structs

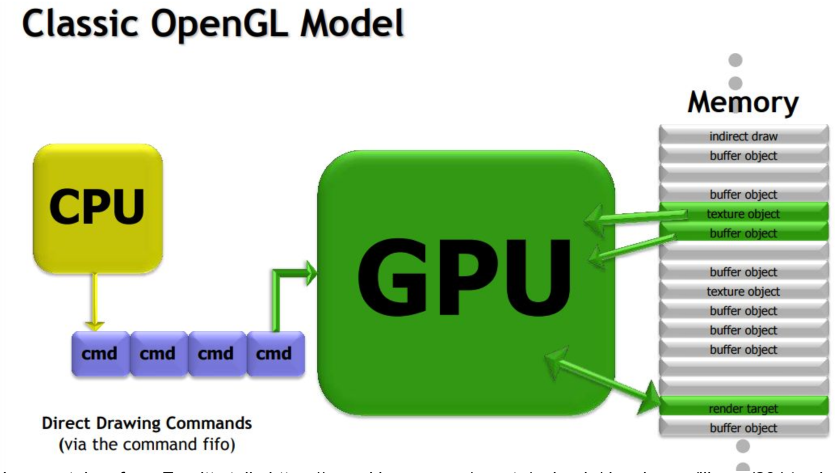
Multiple contexts/Multithreading

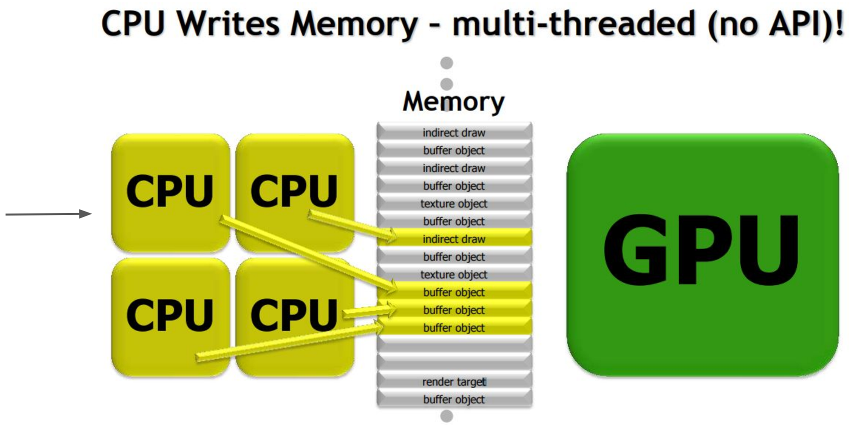
- Each thread can create its own opengl context (separate state)
- Many resources can be shared among threads
- Some resources (VAO) cannot be shared!

- OpenGL implementations use **reference counting** for resources
  - Resources which are created, but in use (e.g. another thread) are not immediately destroyed..
  - Need to **unbind all buffers** before worker thread releases context.
- OpenGL (mostly) does not specify threading behavior
- Especially, because recent extensions with explicit memory usage, this becomes a problem
  - How to synchronize data (yes there are fences…)

Approaching the zero driver overhead (1)

- For complex scenes, driver becomes bottleneck (further reading: [OpenGL Efficiency: AZADO](https://www.khronos.org/assets/uploads/developers/library/2014-gdc/Khronos-OpenGL-Efficiency-GDC-Mar14.pdf), Everitt 2014)
- If so, reduce as much API calls as possible
- **Persistently map memory** (ARB_buffer_storage 2013) and update data in multi-threaded manner.

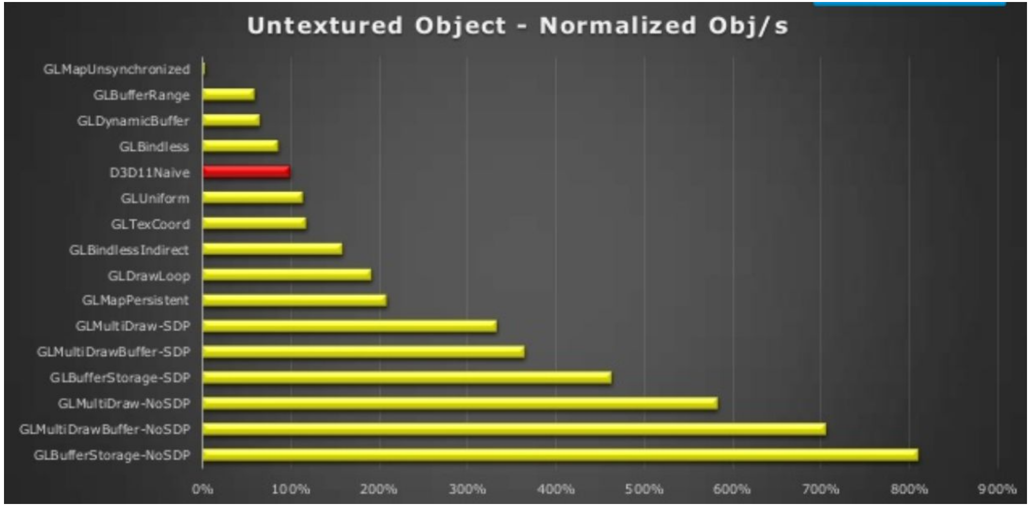




Images taken from Everitts talk, https://www.khronos.org/assets/uploads/developers/library/2014-gdc/Khronos-OpenGL-Efficiency-GDC-Mar14.pdf

Approaching the zero driver overhead (2)

- Shortly before Vulkan came up, there was quite some work on reducing OpenGL driver overhead further.
- Popular talk: [Approaching the zero driver overhead](#) (Everitt et al. 2014, see further reading)
  - All big GPU vendors present mechanisms for efficient OpenGL code.
  - Sometimes techniques perform differently on hardwares or drivers (!)
  - If you really want best GL Performance….



Rendering 64^3 unique object: huge difference for implementation techniques!

https://github.com/nvpro-samples/gl_cadscene_rendertechniques

Lessons learned from OpenGL

- OpenGL *was* user friendly
  - until features did not fit the original design anymore
- The API should be **stateless**
- Graphics API's need clean semantics for **threading**
- We need efficient mechanisms to download/upload/**manipulate GPU memory**

- Ideally,
  - We do not pay for validation
  - We have the best validation possible

Direct3D 8 vs Vulkan

“I think D3D has managed to make the worst possible interface choice at every oportunity. COM. Expandable structs passed to functions. Execute buffers…” [Carmack 1999]

- User friendly OpenGL idea did not work out anymore on modern graphics hardware
- APIs kind of moved from
  - Imperative code which performs the operations
  - Towards big structs which describe what to do

```
(psuedo code, and incomplete)
v = &buffer.vertexes[0];
v->x = 0; v->y = 0; v->z = 0; v++;
v->x = 1; v->y = 1; v->z = 0;v++;
v->x = 2; v->y = 0; v->z = 0;
c = &buffer.commands;
c->operation = DRAW_TRIANGLE;
c->vertexes[0] = 0;
c->vertexes[1] = 1;
c->vertexes[2] = 2;
IssueExecuteBuffer (buffer);
```

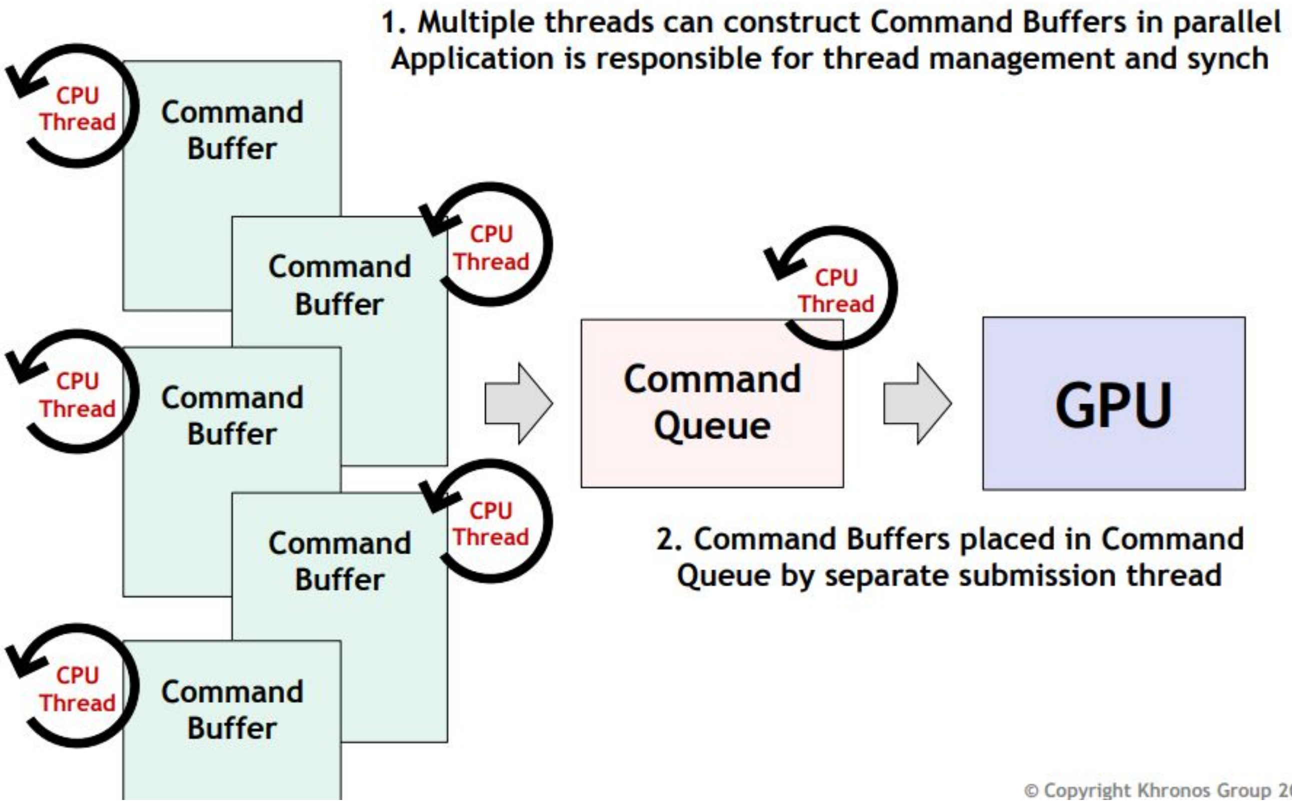
```
Setting up the pipeline…
…
…
… and create the pipeline ….
…

vkCmdBindPipeline(
     commandBuffers[i],
     VK_PIPELINE_BIND_POINT_GRAPHICS,
     graphicsPipeline);
vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);
```

Vulkan

- General theme:
  - Application responsible for
    - Memory allocation (limited allocation count on some devices), synchronization, Command buffers, queue submission
  - More direct GPU control
- Our xp from switching projects to Vk:
  - Direct port is problematic
  -
- https://github.com/cg-tuwien/VulkanLaunchpad
- https://www.vulkan.org/events/vulkanised-2023

https://twitter.com/sosowski/status/699703187289284608

Vulkan Multi-threading Efficiency



https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf

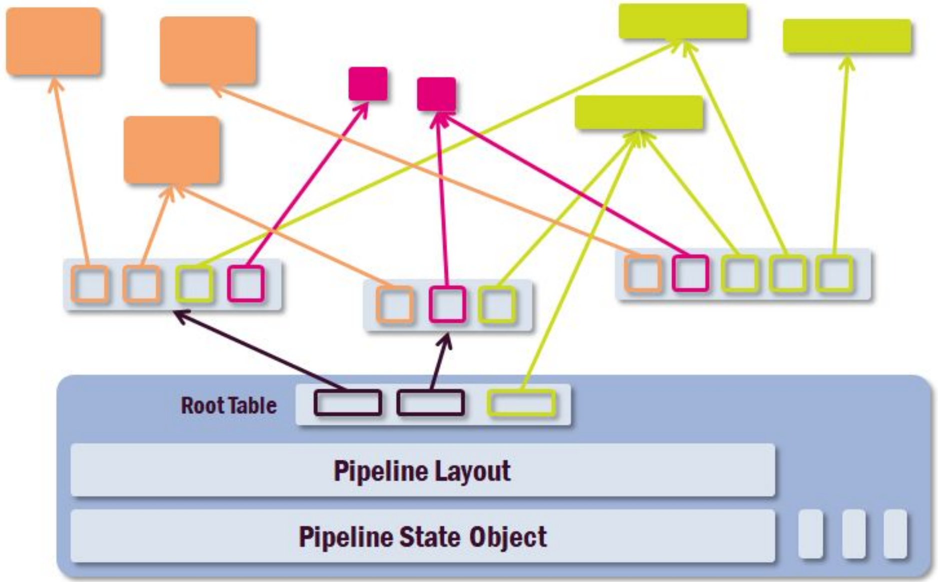
Pipeline state object (PSO in DX)

- Encapsulates most of the GPU state vector
  - Application switches between full PSO
  - Quite heavyweight - compile and create them in advance
- Content:
  - Shaders for each active stage
  - Much fixed function state
    - blend , rasterizer, depth/stencil
  - Format information
    - Vertex attributes
    - Color, depth targets
  - Compare with OpenGL glEnable approach
- Some things not part of PSO
  - State such as viewport may live outside (via VkDynamicState)
  - Line width, Scissors state

Pipeline layout and descriptor sets

Additional resources: Pipeline layout and descriptor sets

See [SIGGRAPH 15] course for excellent introduction material (further reading)



Taken from Foleys talk on Next Gen APIs: http://nextgenapis.realtimerendering.com/presentations/1_Foley_Overview.pptx

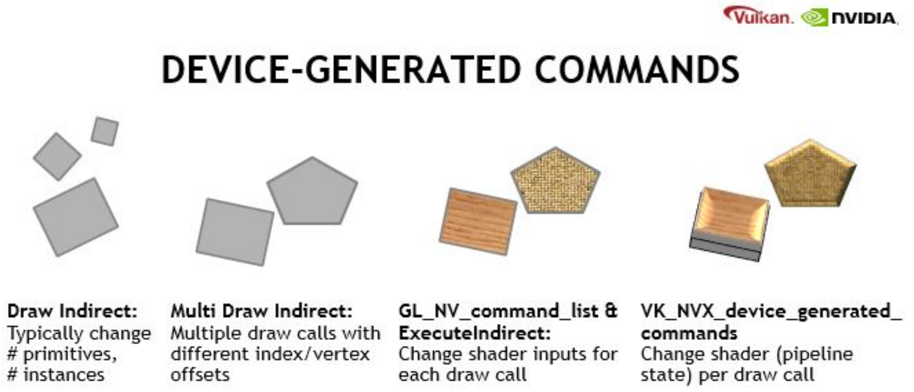
Faster switching

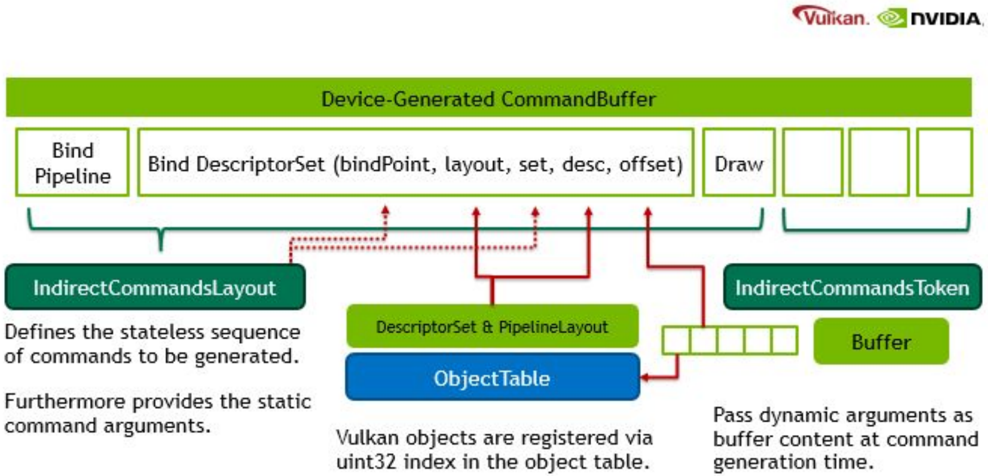
- Dynamic State
- Faster switching
- Derived pipelines

https://registry.khronos.org/vulkan/specs/1.3-khr-extensions/html/chap10.html#pipelines-pipeline-derivatives

```
// Provided by VK_VERSION_1_0
typedef enum VkDynamicState {
    VK_DYNAMIC_STATE_VIEWPORT = 0,
    VK_DYNAMIC_STATE_SCISSOR = 1,
    VK_DYNAMIC_STATE_LINE_WIDTH = 2,
    VK_DYNAMIC_STATE_DEPTH_BIAS = 3,
    VK_DYNAMIC_STATE_BLEND_CONSTANTS = 4,
    VK_DYNAMIC_STATE_DEPTH_BOUNDS = 5,
    VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK = 6,
    VK_DYNAMIC_STATE_STENCIL_WRITE_MASK = 7,
    VK_DYNAMIC_STATE_STENCIL_REFERENCE = 8,
  // Provided by VK_VERSION_1_3
    VK_DYNAMIC_STATE_CULL_MODE = 1000267000,
  // Provided by VK_VERSION_1_3
    VK_DYNAMIC_STATE_FRONT_FACE = 1000267001,
  // Provided by VK_VERSION_1_3
```

- **VK_EXT_device_generated_commands**
- Big state objects -> less binding
- But… Tasks such as Occlusion culling, Object sorting, Level-of-detail structurally change the calls to be performed
- This introduces synchronization and **latency**
  - Decide about draw calls (introduces latency, even if fast)
  - Submission of computed draw calls
- Idea: simply do **everything on GPU**
- First class support for commands
- See [further reading]





Images taken from:
https://developer.nvidia.com/device-generated-commands-vulkan

Compute shaders for efficient rendering

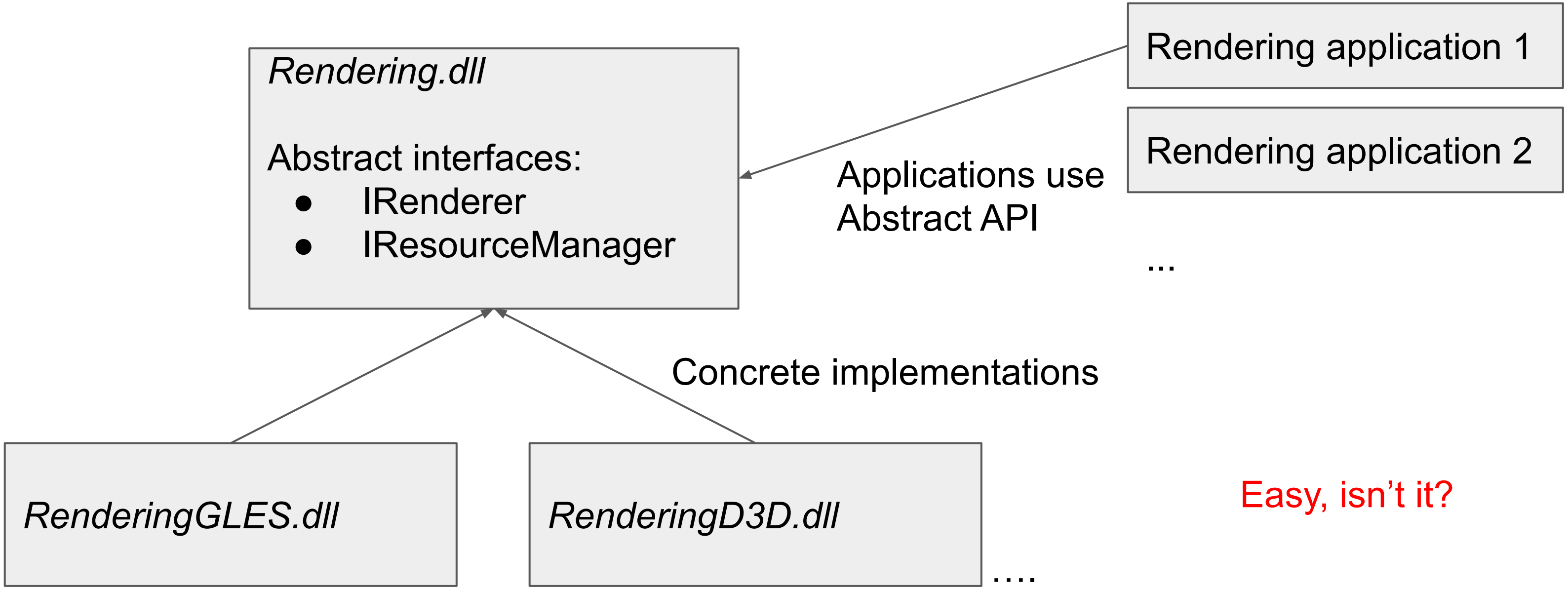
- Also in OpenGL, we can generate indirect buffers or command lists on GPU using compute shaders
- Frostbite engine heavily uses this technique for all sorts of culling
  - Optimizing the Graphics Pipeline with Compute, Wihilidal, Frostbite, GDC 2016, (see further reading)

Lessons learned

- First high level graphics APIs died (retained mode)
- **Immediate mode** based graphics APIs could not take up graphics hardware developments
- OpenGL, D3D12, Vulkan went towards higher level graphics instructions
    - Big state objects
    - MultiDrawIndirect
    - But not too far - no mesh understanding (as in retained mode)
- So higher level abstractions need to be built in rendering engine
    - Find proper level of abstraction
    - Find design goals and assumptions suitable for range of applications which should work
- How to abstract over Graphics APIs properly?

Our overall motivation...

- Use common rendering lib for many rendering applications



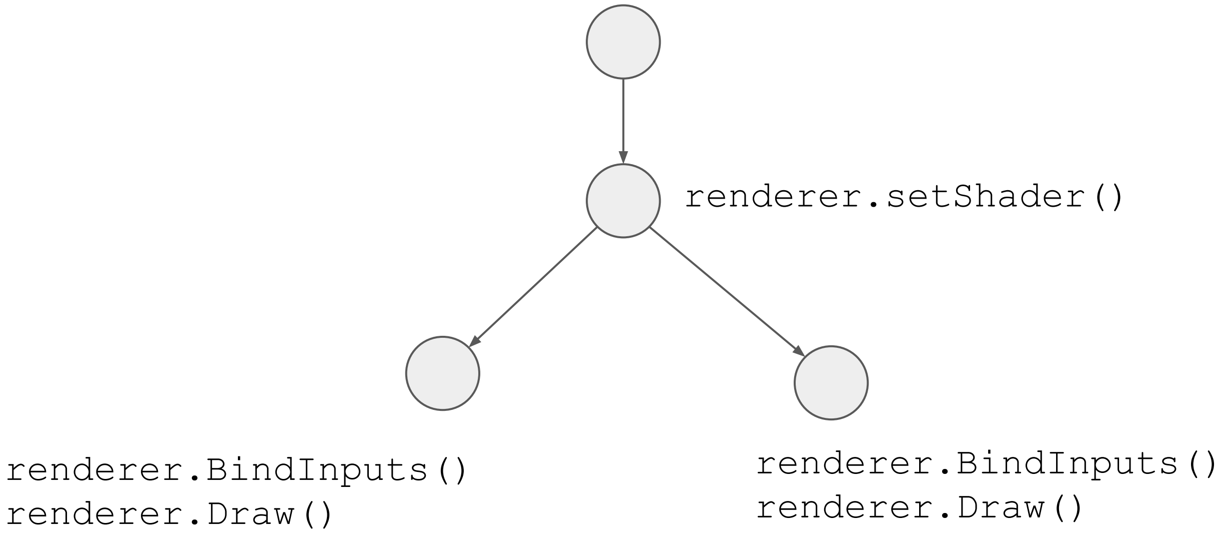
Easy, isn't it?

Object oriented abstraction

- Renderer interface (`IRenderer`) on top of concrete renderer
- Leaf buffer types abstract in the interface, e.g. IBuffer
- Concrete implementations could be `D3D9Buffer` or `D3DD11Buffer`
- At application startup create renderer of your choice and all is fine...

```
// The Aardvark.Rendering library knows nothing about concrete
// rendering APIs like DirectX or OpenGL. In order to create a
// binding to a concrete API we have to create a renderer, e.g.
var renderer = new SlimDx9Renderer();

// The fastest way to get up and running is to use the
// SimpleRenderApplication class which hides all the low-level
// plumbing and is simple to use.
var app = new SimpleRenderApplication(renderer, true);
```

Attempt: towards Graphics API agnostic modules...



```
public interface IRenderer : IDisposable
	{
	/// <summary>
	/// Creates a buffer declaration using the currently bound
	/// semantics and buffers.
	/// </summary>
	object CreateBufferDeclaration();

	/// <summary>
	/// Disposes a buffer declaration.
	/// </summary>
	void DisposeBufferDeclaration(object bufferDeclaration, bool disposeBuffers);

	/// <summary>
	/// Draw currently bound buffers using given GeometryMode.
	/// </summary>
	void DrawPrimitives(GeometryMode mode, object bufferDeclaration, bool isIndexed);

	/// <summary>
	/// Draw currently bound buffers using given GeometryMode and a range of indices.
	/// </summary>
	void DrawPrimitives(GeometryMode mode, object bufferDeclaration, int elementOffset, int
primitivesCount, bool isIndexed);

	void BeginScene();

	void EndScene();
….
}
```

Vertex Declaration (Direct3D 9)

A vertex declaration defines the vertex buffer layout and programs the tessellation engine. D3DVERTEXELEMENT9 structures (instead of using flexible vertex format (FVF) codes). Eac codes into the new format is covered in the following topics:

Instancing? MultiDrawIndirect?

Why???

And the abstraction leak goes on….

We had no choice :(

```
16 references | Christian Luksch, 2662 days ago | 2 authors, 2 changes
public interface IRendererNew: IRenderer
{
    2 references | Robert F. Tobler, 2746 days ago | 1 author, 1 change
    IRenderingResourceManager CreateRenderingResourceManager();

    7 references | Robert F. Tobler, 2746 days ago | 1 author, 1 change
    void BindBufferResource(IResourceNew resource, ShaderSemantic semantic);
                                                                fferDeclaration);
```

```
public TResult Pop<TInterface, TTraversal, TResult>(
    GenericTraversal<TInterface, TTraversal, TResult> traversal,
    LazyResourceUpload store, TResult result)
{
    var renderer = traversal.Renderer as IManagedRenderer;
    if (renderer != null) renderer.LazyResourceUploadEnabled = st
    return result;
}
```

Still kepping the illusion. IManagedRenderer has 1 Implementation: Dx10Renderer...

Suddenly, in D3D10 we can do proper Multithreading and asynchronous uploads...

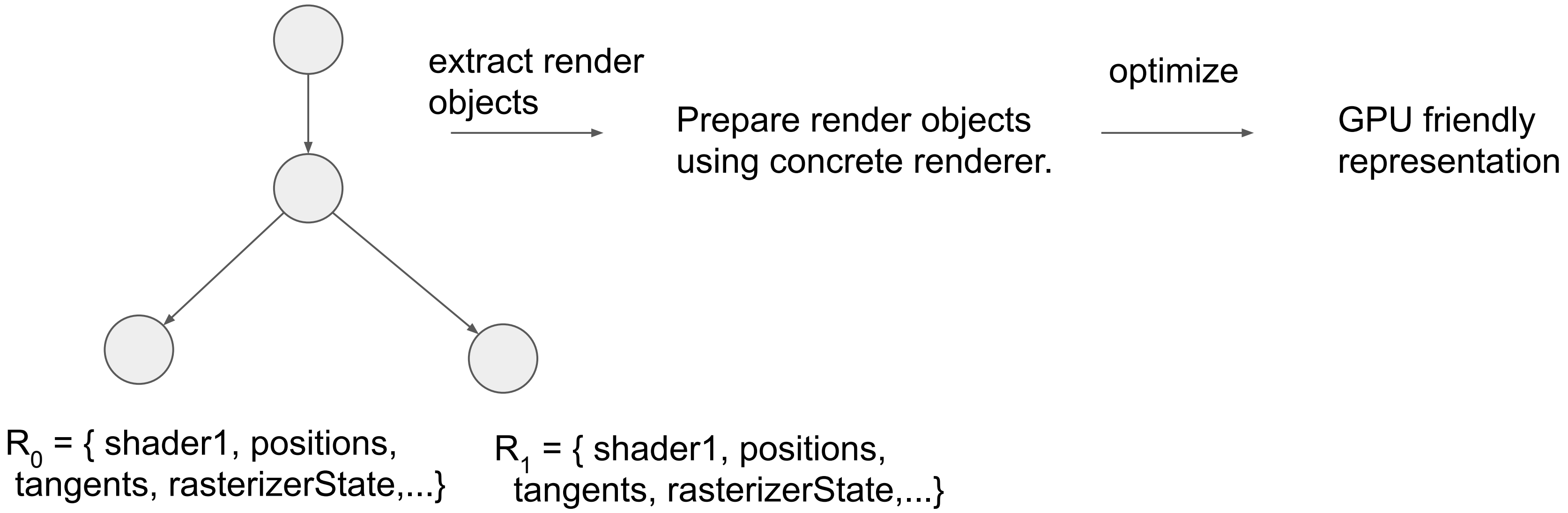
OOP does not help here

- Often there is no (uniquely defined) **1:1** mapping for abstract render commands and graphics api instructions
- (non-core) rendering backend features **leak** into the abstract api
  - e.g. D3D9 vertex declaration
- Some features cannot be expressed
- API usage often is not best-practice for specific API
- However, especially in face of mobile devices, multi API support is often required

Suggestions?

Intermediate representations to the rescue...

- By using a common intermediate representation we don't need to give up reusability….



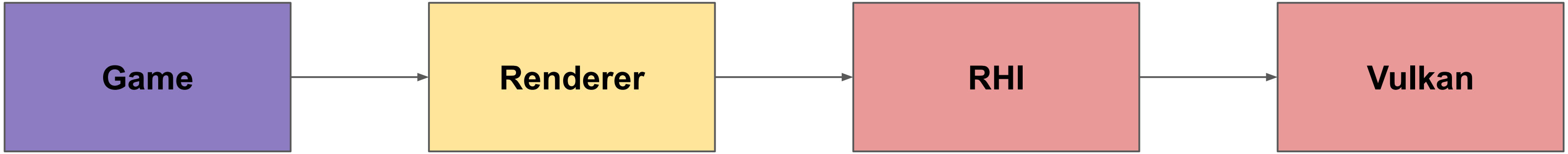
Unreal Engine architecture

- Original architecture
  - Game Thread enqueues rendering commands
  - Rendering Thread generates Vulkan Cmd Buffers

https://www.khronos.org/assets/uploads/developers/library/2016-siggraph/3D-BOF-SIGGRAPH_Jul16.pdf

Unreal Engine architecture (improved)

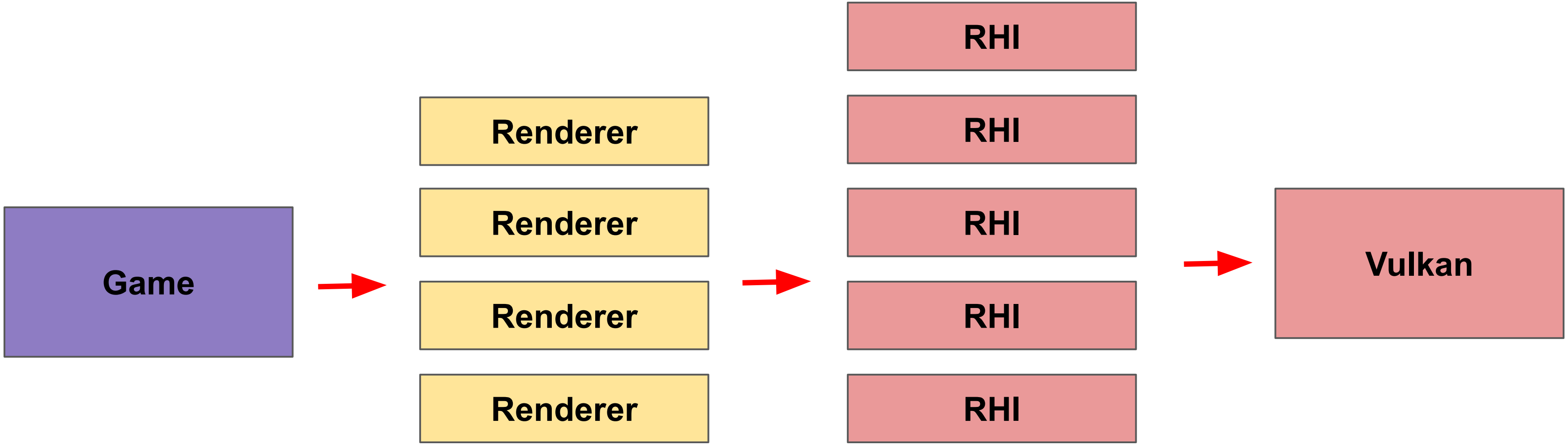
- Render Hardware Interface (RHI)
  - Cross platform way to talk to each Gfx API
- Improved architecture
  - Game Thread enqueues rendering commands
  - Rendering Thread generates RHI command list
  - RHI Thread translates into Vulkan Cmd Buffers



https://www.khronos.org/assets/uploads/developers/library/2016-siggraph/3D-BOF-SIGGRAPH_Jul16.pdf

Unreal Engine architecture (multithreaded)

- UE4 RHI Architecture is multithreaded
  - N Render threads with M RHI Threads



https://www.khronos.org/assets/uploads/developers/library/2016-siggraph/3D-BOF-SIGGRAPH_Jul16.pdf

Takeaways

- Finding the **right abstraction** mechanism for rendering engines **is difficult**
- **Same** occurs for **graphics APIs** itself (e.g. DX2 retained mode is not a graphics API as we know it but already a rendering engine)
- OpenGL’s abstraction was well-suited for many years
  - Because of performance issues, many extensions have been introduced
  - The evolution and shifted focus towards **gray zones** in opengl (e.g. multithreading, explicit synchronization, explicit GPU memory access)

Takeaways (2)

- Next-gen APIs (DirectX 12, Vulkan, Metal) provide lower level abstractions
- More work to do for the rendering engine developer (e.g. writing memory managers)
- Next-gen APIs are too different to just port code and being more efficient
  - All game engines (we know of) had problems at first.
  - There is great material which describes game engine evolution towards next-gen
- Standard object oriented abstraction via renderer interface in our experience was an illusion

Takeaways (3)

- Abstracting renderable objects is much more suitable than low level commands
- Given renderable objects, we can prepare all gpu resources - it remains to
  - Add them to an scene optimization data-structure
  - And update the scene optimization data-structure given changes in the input scene representation

Further reading (⅓)

- Carmack on OpenGL, Carmack 1996, http://rmitz.org/carmack.on.opengl.html
- Direct3D Immediate mode 2000, https://www.gamedev.net/articles/programming/graphics/direct3d-immediate-mode-r911/
- Direct3D Retained mode headers: https://github.com/lifthrasiir/w32api-directx-standalone/blob/master/include/d3drm.h
- Migrating from OpenGL to Vulkan, Kilgard 2016: https://www.slideshare.net/Mark_Kilgard/migrating-from-opengl-to-vulkan
- **OpenGL Scene Rendering Techniques:** http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4117-OpenGL-Scene-Rendering-Techniques.pdf
- **Siggraph BOF, interesting porting work of game engine developers for vulkan:** https://www.khronos.org/assets/uploads/developers/library/2016-siggraph/3D-BOF-SIGGRAPH_Jul16.pdf

Further reading (⅔)

- Shader-driven compilation of rendering assets, Lalonde and Schenk, EA, Siggraph 2002, https://dl.acm.org/citation.cfm?id=566641
- Shader components: modular and high performance shader development, He et al. 2017, https://dl.acm.org/citation.cfm?id=3073648
- Vulkan Device Generated Commands, Kubisch 2016, https://developer.nvidia.com/device-generated-commands-vulkan
- VK_NVX_device_generated_commands, NVIDIA 2016, https://www.khronos.org/registry/vulkan/specs/1.0-extensions/html/vkspec.html#VK_NVX_device_generated_commands
- **Approaching the Zero Driver Overhead (AZDO talk)**, Everitt, Sellers, McDonald, Foley, Siggraph, GDC 2014, https://de.slideshare.net/CassEveritt/approaching-zero-driver-overhead
- OpenGL Efficiency: AZDO overview talk, https://www.khronos.org/assets/uploads/developers/library/2014-gdc/Khronos-OpenGL-Efficiency-GDC-Mar14.pdf
- **Optimizing the Graphics Pipeline with Compute**, Wihlidal (Frostbite) 2016, https://frostbite-wp-prd.s3.amazonaws.com/wp-content/uploads/2016/03/29204330/GDC_2016_Compute.pdf

Further reading (3/3)

- Vulkan overview, Khronos group, 2015, https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf
- Siggraph 2015 had a course on Next-Gen Graphics APIs:
    - http://nextgenapis.realtimerendering.com/