Optimization techniques for rendering engines
Outline

- Graphics APIs (OpenGL, GLES, 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: MultiDraw features which allow to perform multiple draw calls with a single API call.
- 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**
Direct3D 2: Retained mode vs Immediate mode

Retained mode

Immediate mode

Direct3D 2.0 concept

- **Direct3D immediate mode**
  - Commands (e.g. draw) issued to execute buffer, parameters in structs
    
    ```cpp
    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
    
    ```cpp
    #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
    ```java
    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)

```c
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

Server with separate memory and execution engine
OpenGL 1.0 concept

- State machine
  - Commands modify state which is expected in subsequent commands...
- Strict client/server architecture:
  - Weakened quite early: `glEnableClientState` -> allow the server to read client memory...
- Primitives can be sent, piece by piece to the server....

```c
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

```c
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

```c
// 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 **Vertex Buffers** Object (VBO, ARB 2003)

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

```c
//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 **Vertex Array Objects (VAO)**

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

```c
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.

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

```c
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:
  ```javascript
  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)`
  ```javascript
  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_STATIC_DRAW)
  GLNamedCopyBufferSubData(tmpBuffer, x.Handle, 0n, 0n, copyBytes)
  GL.DeleteBuffer(tmpBuffer)
  ```
Evolution: Binding considered harmful

- Clearly, OpenGL tries to remove as much driver calls as possible......
- For good reasons...each call needs to travel into the driver which needs to validate the call (according to the spec)
- Rumors have it: There are implementations and hacks in the driver to react to common usage patterns of game in order to reduce overhead
- Informally speaking, ARB_bindless_* (2013) allows to the shader to use GL objects without binding them....

With Uniform Buffer Objects, Shader Storage Buffer Objects, and various other means, it is possible to communicate state information to a Shader without having to modify any OpenGL context state. You simply set data into the appropriate buffer objects, then make sure that those buffers are bound when it comes time to render with the shader.

There are lines of communication for which this cannot work. Specifically, the Opaque Types in GLSL: samplers, images, and atomic counters. These all derive their data based on objects bound to locations in the OpenGL context at the time of the rendering call.

This has two performance bearing consequences. The immediate performance cost is that you must bind textures and images to the context before rendering. This process has an intrinsic cost.
OpenGL seems to be very afraid of API calls.

And so should you: if you want to get the best possible performance
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**

```c
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);
    }
}
```

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.....
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)

- If so, reduce as much API calls as possible
- **Persistently map memory** (ARB_buffer_storage 2013) and update data in multi-threaded manner.
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!
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

- 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)

```c
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...

```c
vkCmdBindPipeline(
    commandBuffers[i],
    VK_PIPELINE_BIND_POINT_GRAPHICS,
    graphicsPipeline);
```

```c
vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);
```
Vulkan in a nutshell

- Vulkan: Graphics and Compute Belong Together ;)
- General theme:
  - Application responsible for (everything)
    - Memory allocation, synchronization, Command buffers
  - Direct GPU control
- Our xp from switching projects to Vk:
  - Explicit memory control (though we need to Write many OS things such as memory managers)

https://twitter.com/sosowski/status/699703187289284608
Vulkan Multi-threading Efficiency

1. Multiple threads can construct Command Buffers in parallel. Application is responsible for thread management and sync.

2. Command Buffers placed in Command Queue by separate submission thread.
Pipeline state object (PSO)

- 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

- 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
• 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
  o Decide about draw calls (introduces latency, even if fast)
  o Submission of computed draw calls
• Idea: simply do everything on GPU ;)
• First class support for commands
• See further reading

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?
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...

```csharp
// 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...

```
renderer.setShader()
renderer.BindInputs()
renderer.Draw()
renderer.BindInputs()
renderer.Draw()
```
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();
    ...
}
And the abstraction leak goes on….

We had no choice :(

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

Still keeping the illusion. IManagedRenderer has 1 Implementation: Dx10Renderer...
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?
Leaky abstraction breaks reusability

```cpp
glBindVertexArray(obj1);
glNamedBufferSubData(ubo,...)
Draw();
```
Intermediate representations to the rescue...

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

\[ R_0 = \{ \text{shader1, positions, tangents, rasterizerState,...} \} \]

\[ R_1 = \{ \text{shader1, positions, tangents, rasterizerState,...} \} \]

Prepare render objects using concrete renderer. 

Optimize 

GPU friendly representation
Towards incrementally updated optimization data-structures

Research paper: Lazy Incremental Computation for Efficient Scene Graph Rendering [Wörister et al. 2013]
Instruction types in render caches

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetPrimitiveTopology</td>
<td>a primitive topology</td>
</tr>
<tr>
<td>SetVertexShader</td>
<td>a vertex shader</td>
</tr>
<tr>
<td>SetShaderInputLayout</td>
<td>a shader input layout</td>
</tr>
<tr>
<td>SetFragmentShader</td>
<td>a fragment shader</td>
</tr>
<tr>
<td>SetGeometryShader</td>
<td>a geometry shader</td>
</tr>
<tr>
<td>SetVertexBufferBinding</td>
<td>a vertex buffer binding</td>
</tr>
<tr>
<td>SetIndexedVertexBufferBinding</td>
<td>an indexed vertex buffer binding</td>
</tr>
<tr>
<td>SetConstantBuffers</td>
<td>a set of (slot-index, constant buffer) pairs</td>
</tr>
<tr>
<td>SetShaderResourceBuffers</td>
<td>a set of (slot-index, constant buffer) pairs</td>
</tr>
<tr>
<td>DrawIndexed</td>
<td>start index and element count</td>
</tr>
<tr>
<td>DrawArrays</td>
<td>start vertex and element count</td>
</tr>
</tbody>
</table>

Instruction stream

Primitive Topology | Vertex Shader | Fragment Shader | Shader Input Layout | Shader Resource Buffers | Constant Buffers | Vertex Buffer Binding | Draw Arrays | Constant Buffers | Vertex Buffer Binding | Draw Arrays |
|------------------|---------------|----------------|---------------------|-------------------------|------------------|----------------------|-------------|------------------|----------------------|-------------|

Resources
Implementation details for render caches

```java
interface RenderJobBuilder {
    void SetPrimitiveTopology(PrimitiveTopology pt);
    void SetInputLayout(IShaderInputLayout inputLayout);
    void SetSurface(Surface surface);

    void SetConstantBuffer(int slot, IConstantBuffer buffer, ShaderType shaderType);
    void ClearConstantBuffer(int slot, ShaderType shaderType);
    void SetShaderResourceBuffer(int slot, IShaderResourceBuffer buffer, ShaderType shaderType);
    void ClearShaderResourceBuffer(int slot, ShaderType shaderType);

    void SetIndexBuffer(Buffer indexBuffer);
    void BeginVertexBufferBinding();
    void BindVertexBuffer(String semantic, Buffer buffer);
    void EndVertexBufferBinding();

    void DrawIndexed(int startIndex, int elementCount);
    void DrawIndexed(int elementCount);
    void DrawArrays(int startIndex, int elementCount);
    void DrawArrays(int elementCount);
}
```
Interleaved, multithreaded resources updates
Memoization for efficient trafo updates
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
- Updating optimization data-structures is not straightforward
  - Lazy Incremental Computation for Efficient Scene Graph rendering shows to do incrementally updated GPU optimized representations for scene graphs (more to come)
  - Lazy polling can be used to support culling
Further reading (1/3)

- Carmack on OpenGL, Carmack 1996, [http://rmitz.org/carmack.on.opengl.html](http://rmitz.org/carmack.on.opengl.html)
- Direct3D Immediate mode 2000, [https://www.gamedev.net/articles/programming/graphics/direct3d-immediate-mode-r911/](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](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](https://www.slideshare.net/Mark_Kilgard/migrating-from-opengl-to-vulkan)
Further reading (2/3)

- Shader components: modular and high performance shader development, He et al. 2017, https://dl.acm.org/citation.cfm?id=3073648
Further reading (3/3)

- Siggraph 2015 had a course on Next-Gen Graphics APIs:
  - http://nextgenapis.realtimerendering.com/
Status and next steps

- 9.10 Intro: What is a rendering engine, what are the requirements
- 16.10 Scene representation: how to model scenes
- 23.10 Graphics API insights and optimizations
- 30.10 Applied data-structures
- 6.11 Scene representation for dynamic scenes, Aardvark tutorial and UE Vorbesprechung
- 13.11 Towards a fully incremental rendering engine
- 20.11 Domain Specific Languages
- 27.11 Lights and Materials
- 04.12 Lights and Materials
- 11.12 Rundown and Questions