Execution Engines for RenderObjects
class RenderObject {
    Shader[] Shaders;
    BlendMode BlendMode;
    DrawCall Call;
    // ... 
    bool TryGetUniform(string name, out object value);
    bool TryGetAttribute(string name, out Array value);
}
A simple ADT for rendering (Interpreter Approach)

```java
interface Renderer {
    void Render(
        Collection<RenderObject> objects,
        Framebuffer target
    )
    void Clear(Framebuffer target)
}
```

- ✔ Very flexible
- ✗ Resource lifetime unknown
- ✗ Optimizations virtually impossible
Compiler Approach

interface Renderer {
    RenderProgram NewProgram();
    void Clear(Framebuffer target);
}

interface RenderProgram {
    void Run(Framebuffer target);
    void Add(RenderObject obj);
    void Remove(RenderObject obj);
    void Clear();
}

✔ Deterministic resource lifetime
✔ Optimization per RenderProgram
✖ Dynamic resources (uniforms, buffers, etc.)
interface IMod<T> {  
    T GetValue();  
    event OnChange;
}

class RenderObject {  
    Shader[] Shaders;  
    IMod<BlendMode> BlendMode;  
    IMod<DrawCall> Call;  
    // ...  
    bool TryGetUniform(string name, out IMod<object> value);  
    bool TryGetAttribute(string name, out IMod<Array> value);
}
Compiler Approach

interface Renderer {
    RenderProgram NewProgram();
    void Clear(Framebuffer target);
}

interface RenderProgram {
    void Run(Framebuffer target);
    void Add(RenderObject obj);
    void Remove(RenderObject obj);
    void Clear();
}

✔ Deterministic resource lifetime
✔ Optimization per RenderProgram
✔ Dynamic resources (uniforms, buffers, etc.)
Virtual Machine Approach

interface Renderer {
    RenderProgram NewProgram();
}
interface RenderProgram {
    void Run(Framebuffer target);
    void Insert(int index, Command i);
    void Update(int index, Command i);
    void Remove(int index);
    void Clear();
}

union Command {
    Render(RenderObject),
    Clear,
    //...
}

✔ Deterministic resource lifetime
✔ Optimization per RenderProgram
✔ Dynamic resources (uniforms, buffers, etc.)
✔ Extensibility via Commands
Implementing the Virtual Machine
Prepared Commands

interface PreparedCommand {
    Collection<Resource> Resources
    List<APICall> Code
}

Preparing commands is a twofold process

- Creating resources (potentially dynamic)
- Emitting API-calls using the resources

API-calls expect resources to be consistent upon execution
Resource Management

Abstract Resource Definition

```csharp
interface ResourceLocation<THandle> : IMod<THandle> {
    void AddReference();
    void RemoveReference();
}
```

PreparedCommands reference resources

Program references PreparedCommands
Resource Management

Immutable resources (Shaders, etc.)
Mutable resources (Buffers, Textures)

Resources are potentially shared (e.g. Shaders, Textures, etc.)

➔ Reference counting / garbage collection
➔ Dynamic updates via IMod<T>

Sharing per Mod / per Value (TexturePath, etc.)

Batch- vs. inline updates

Aardvark’s implementation is the ~10th rewrite (hard to get it right)
Execution Engine

From now on resources are just values living in mod cells! Therefore (OpenGL):

```c
union APICall {
    UseProgram    (IMod<int> p),
    BindVAO       (IMod<int> v),
    BindUniforms  (int slot, IMod<int> ubo),
    BindTexture   (int slot, IMod<int> tex),
    Draw          (IMod<DrawCall> c),
    //...
}
```

About 60 different instructions in Aardvark’s OpenGL backend
Two Problems

1. How to execute APICalls?

2. How to translate Commands to APICalls?
Two Problems

1. How to execute API Calls?

2. How to translate Commands to API Calls?
foreach (var instr in code) {
    switch (instr) {
    case UseProgram(h):
        glUseProgram(h.GetValue());
        break;
    case BindVAO(h):
        glBindVAO(h.GetValue());
        break;
    // ...
    }
}
Execution: Improved Interpreter

```csharp
var prog = -1;

foreach (var instr in code) {
    switch (instr) {
    case UseProgram(h):
        var p = h.GetValue();
        if (prog != p) {
            prog = p;
            glUseProgram(p);
        }
        break;
        // ...
    }
}
```

✔ Easy implementation
✔ No Redundancies
✖ Case distinction
✖ Redundancy checks everywhere
Execution: Indirect Threaded Code

Action Compile(APICall call) {
    switch (call) {
        case UseProgram(p):
            return () => {
                glUseProgram(p.GetValue());
            };
        // ...
    }
    // ...

    foreach (var action in compiled) {
        action();
    }
}

✔ Case distinction no longer at runtime
✖ Redundancies
✖ Indirect Calls
✖ Many Action allocations

aka. OOP Approach (IIInstruction)
Execution: Direct Threaded Code

Assemble all calls to executable machine code:

```
ldarg p
call GetValue
call glUseProgram

ldarg vao
call GetValue
call glBindVertexArray
```

✔ Best possible runtime performance
✘ Insert/Remove trigger whole program recompile
✘ Redundancy

(pseudo ASM)
Introducing Fragments

- Linked list of executable ASM code
- Use JMP instructions for linking them
- Insertion, deletion efficient

Compile each Command to a Fragment

Insert it at the desired position in the linked list

Fragmentation might be an issue (non-local JMPs) (can be solved using async defragmentation thread)
Execution: Fragments

LDARG P
CALL GetValue
CALL glUseProgram
LDARG DC
CALL GetValue
CALL glDraw
JMP +100

LDARG DC
CALL GetValue
CALL glDraw
RET

✔ Good runtime
✖ Insert/Remove cause fragment compilation
✖ Redundancy
✖ Fragmentation issue
Execution: Synthetic Benchmark

![Graph showing the time per instructions versus instruction count for different execution modes: Indirect Threaded Code, Interpreter, Direct Threaded, Fragments (local jumps), and Fragments (random order).]
Two Problems

1. How to execute APICalls?

2. How to translate Commands to APICalls?
Compiler: Program Representation

Programs need to support

- Insertion/Deletion at arbitrary positions
- Execution of the entire code
- Disposal

(Double) linked list representation provides $O(1)$ for insert/delete

Fragmentation might slow down execution.
Compiler: Static Redundancy Removal

Mod to the rescue!

Compile commands using their predecessor

Some redundancies might be missed when \( p_0.\text{GetValue}() == p_1.\text{GetValue}() \)

Inserting/Deleting/Reordering Commands causes recompilation overhead

\begin{verbatim}
glUseProgram(p0.GetValue());
glDraw(c0.GetValue());
\end{verbatim}

\begin{verbatim}
glUseProgram(p0.GetValue());
glDraw(c1.GetValue());
\end{verbatim}
Compiler: Static Redundancy Removal

UseProgram(p0)
BindVAO(v0)
BindIndex(i0)
Draw(c0)

UseProgram(p3)
BindVAO(v1)
BindIndex(i4)
Draw(c2)

UseProgram(p0)
BindVAO(v1)
BindIndex(i0)
Draw(c1)
Compiler: Static Redundancy Removal

UseProgram(p0)
BindVAO(v0)
BindIndex(i0)
Draw(c0)

UseProgram(p3)
BindVAO(v1)
BindIndex(i4)
Draw(c2)
Compiler: Ordering

Command Order is

- ... important for redundancy removal
- ... given by the user (partially)
- ... potentially dependent on camera, etc. (e.g. back-to-front rendering)

Reorder commands to minimize cost for API calls where possible

Simple approach: Trie grouping Command by Shaders, etc.
Overall System

- Execution techniques (Interpreter, Fragments)
- Redundancy strategies (static, dynamic, both)

<table>
<thead>
<tr>
<th>Redundancy Strategy</th>
<th>Execution</th>
<th>Reorder</th>
<th>Compile</th>
<th>Execution</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Fragments</td>
<td>terribly slow</td>
<td>slow</td>
<td>awesome</td>
<td>okay</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Fragments</td>
<td>good</td>
<td>slow</td>
<td>okay</td>
<td>terribly huge</td>
</tr>
<tr>
<td>Static</td>
<td>Interpreter</td>
<td>slow</td>
<td>okay</td>
<td>good</td>
<td>small</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Interpreter</td>
<td>fast</td>
<td>fast</td>
<td>rather slow</td>
<td>large</td>
</tr>
</tbody>
</table>
## Overall System

- Execution techniques (Interpreter, Fragments)
- Redundancy strategies (static, dynamic, both)

<table>
<thead>
<tr>
<th>Redundancy Strategy</th>
<th>Execution</th>
<th>Reorder</th>
<th>Compile</th>
<th>Execution</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Fragments</td>
<td>terribly slow</td>
<td>slow</td>
<td>awesome</td>
<td>okay</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Fragments</td>
<td>good</td>
<td>slow</td>
<td>okay</td>
<td>terribly huge</td>
</tr>
<tr>
<td>Static</td>
<td>Interpreter</td>
<td>slow</td>
<td>okay</td>
<td>good</td>
<td>small</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Interpreter</td>
<td>fast</td>
<td>fast</td>
<td>rather slow</td>
<td>large</td>
</tr>
</tbody>
</table>
Real World Benchmark

HugeCity (6580 objects)
Real World Benchmark

Sponza24 (9408 objects)
Real World Benchmark

Architecture (7022 objects)
Real World Benchmark

Relative Performance

- Interpreter/None
- Interpreter/Dynamic
- Interpreter/Static
- Fragments/Static

- Radeon HD 7970, AMD FX-9590
- GTX 680, Intel i7-920
Vulkan and DX12

Only a few commands for rendering (due to huge state objects)
=> Redundancy removal not that crucial

Our Programs just record CMD buffers
=> Record only when necessary (e.g. UBO changes in-place)

RenderProgram submits cached CMD buffers
=> minimal CPU overhead
**Special Techniques**

Rendering many objects efficiently
Hardware supports several special cases
Can be introduced as new Commands in VM

union Command {
    Render(RenderObject),
    Clear,
    MultiDraw(...),
    RenderInstanced(...),
    //...
}

render 1 million cubes

<table>
<thead>
<tr>
<th>Pipelines</th>
<th>Textures</th>
<th>Geometries</th>
<th>Technique</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Instancing</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>*</td>
<td>MultiDrawIndirect</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>TextureAtlas, TextureArray, etc.</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>VM, GL_NV_command_lists, VK_NV_device_generated_commands</td>
<td>7 (GL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 (VK)</td>
</tr>
</tbody>
</table>
Takeaways

Extensible API for rendering (RenderCommands)

VM techniques for RenderObject execution

Tradeoffs (update, execute, memory)

Low level optimizations (Instancing, DrawIndirect)
Further Reading

  (interesting note: they used bytecode interpreter…)
- Shader components: modular and high performance shader development, He et al. 2017, [https://dl.acm.org/citation.cfm?id=3073648](https://dl.acm.org/citation.cfm?id=3073648)
- Threaded Code, Ertl, Complang, [https://www.complang.tuwien.ac.at/forth/threaded-code.html](https://www.complang.tuwien.ac.at/forth/threaded-code.html)