How we implemented a Rendering Engine

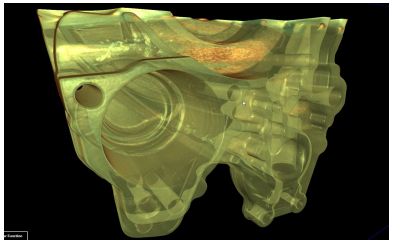
and what we could have done differently

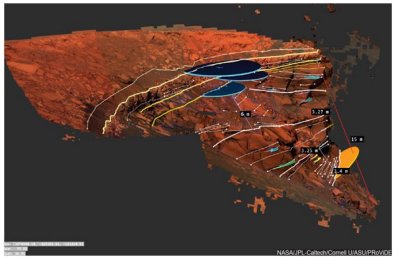
About this talk

- communicate some learnings on working on the aardvark rendering engine for ~15 years
- show some of the most important design decisions & their design space
- reflect on some decision and hint alternatives
- give ideas on future research directions

target application use cases

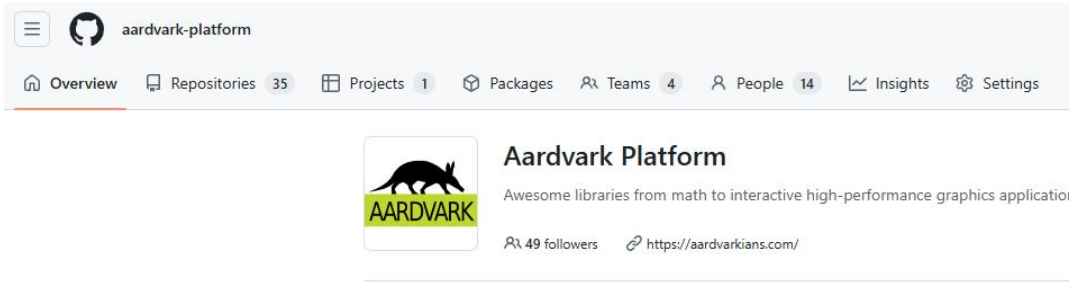
- research platform
- model/cad viewer
- scientific applications
- shadertoy
- AR/VR
- interactive game
- -> Aardvark is a rendering platform for the use cases most important to us





What is Aardvark?

- large set of libraries for geometry/meshes/rendering/UI
- rendering system for scientific visualization
- flexible scene description
- reasonably fast

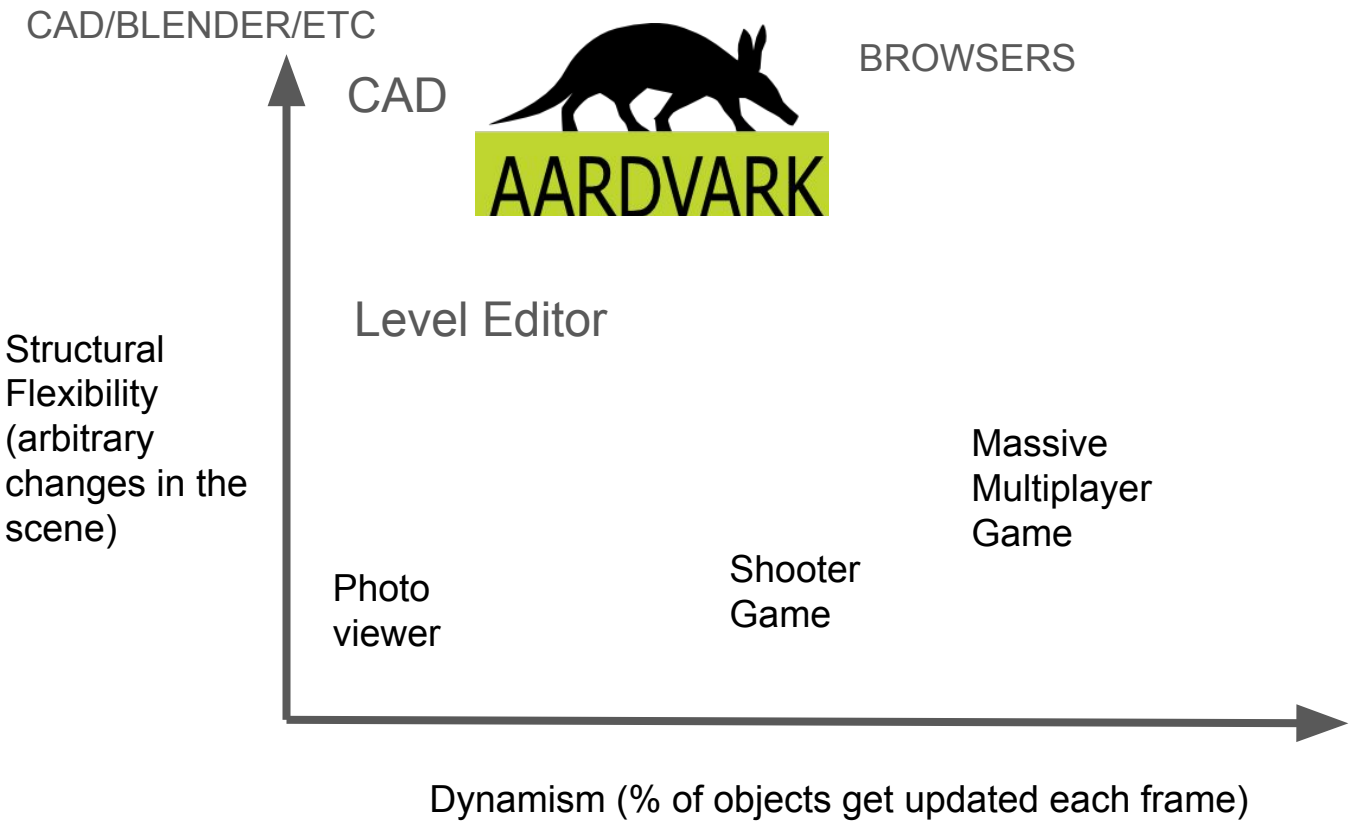


- open source community (https://github.com/aardvark-platform, https://github.com/aardvark-community)
- ~90 repositories on github
- used in many VRVis Research projects
  - https://www.vrvis.at/produkte-loesungen/produkte-lizenzen/aardvark

Necessary Components / Design Space

- scene description?
- how to render scenes?
  - and how do changes enter?
- shader specification?
- resource management? (user/internal, GC/precise)
- rendering backend: graphics api agnostic?
- rendering techniques (deferred shading, post-processing,...)
- user interaction?

Design space for dynamism



Scene Description (as frontend)

- SceneGraph (most flexible scene description)
  - user-friendly (e.g. attributes can be “inherited” down the graph)
  - natural representation (table consists of legs and a tabletop, etc.)
  - attributes can be “inherited” down the graph
- list of renderable things (array of structs)
  - lots of duplicated attributes/states
  - simple implementation, flexible
  - without grouping facilities hard to maintain
- database style representation (struct of arrays)
  - data stored as flat arrays
  - renderable things carry ids for their respective attributes
  - basis for ECS
  - without grouping facilities hard to maintain

SceneGraphs

Come in two main “flavors”

SG describes things to render and their properties (transformations, groups, textures), changes come from outside, ex. HTML DOM

SG contains active logics (object manipulator/controller, physics simulation, animation, camera+LOD, …), ex. OpenSceneGraph, VRML, Aardvark2010

Further Reading [1]

We took this approach [e.g. Tobler 2011]. Now this happens in application logics

Module: Scene Graph

Design Space from user perspective

- Embedded in app language (like API)
- External representation

Implementation techniques:

- In Functional Programming: Union type
- In Object Oriented Programming
  - Interface for nodes (we called it `ISg`)
  - Operation "`getRenderObjects()`" method
  - .. or implementation via *Visitor Pattern*
- Attribute Grammar

We also proposed to use Attribute Grammars for synthesizing render objects adaptively [e.g. Steinlechner et al. 2019] as opposed to plain hand-written traversals. For this presentation, this can be considered as an implementation detail.

Language design

inline css + HTML style

nesting structure

Transformation order semantics

```
sg {
    Sg.Shader { Shader.simpleLighting }
    Sg.BlendMode BlendMode.Add
    sg {
        Sg.Translate(10.0, 0.0, 0.0)
        Primitives.Sphere(radius = 1.0, color = C4b.Red)
    }
    sg {
        Sg.Scale 10.0
        Primitives.Box(size = V3d(2,2,2))
    }
}
```

SceneGraph implementation

Example: Implementation in C# using interfaces/classes.

(see Scene Representation lecture)

```
var scene =
    new Group(
        new List<ISg>()
        {
            new Trafo(Trafo3d.Translation(0.0,1.0,0.0),
                sphereNode),
            new Trafo(Trafo3d.Translation(1.0,0.0,0.0),
                sphereNode)
        });
```

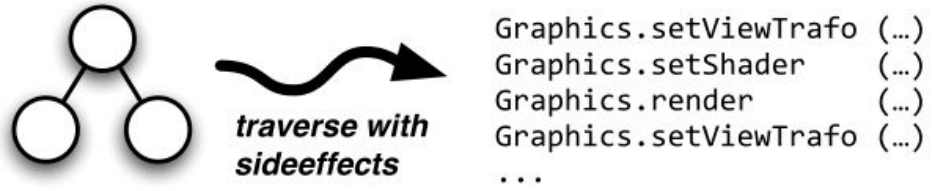
Aardvark implementation

In F# using language constructs help, making it syntactically lightweight.

(conceptually same as approach left)

```
sg {
    Sg.Shader { Shader.simpleLighting }
    Sg.BlendMode BlendMode.Add
    sg {
        Sg.Translate(10.0, 0.0, 0.0)
        Primitives.Sphere(radius = 1.0, color = C4b.Red)
    }
    sg {
        Sg.Scale 10.0
        Primitives.Box(size = V3d(2,2,2))
    }
}
```

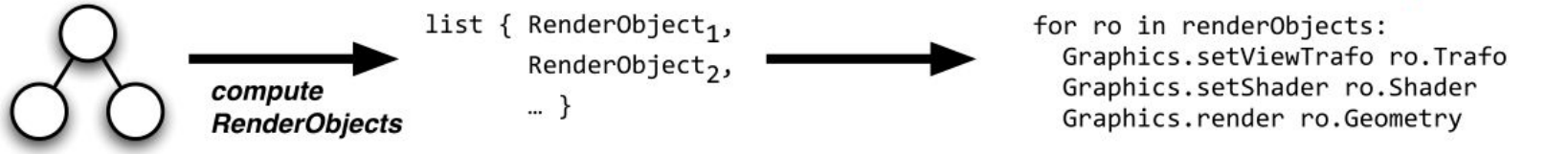
How to Render a SceneGraph classically



old approach

We tried this approach, and presented optimizations in "Lazy Incremental Computation for efficient Scene Graph Rendering" [Wörister 2012], but moved away from in-place graphics commands in favor of a render object intermediate representation.

Command Buffer or Virtual Machine



more modern approach

handling input-changes requires recreation of RenderObject list

diffing/caching algorithms needed for stable resources

The *RenderObject* Abstraction

All inputs to perform draw call
+ some getter functions

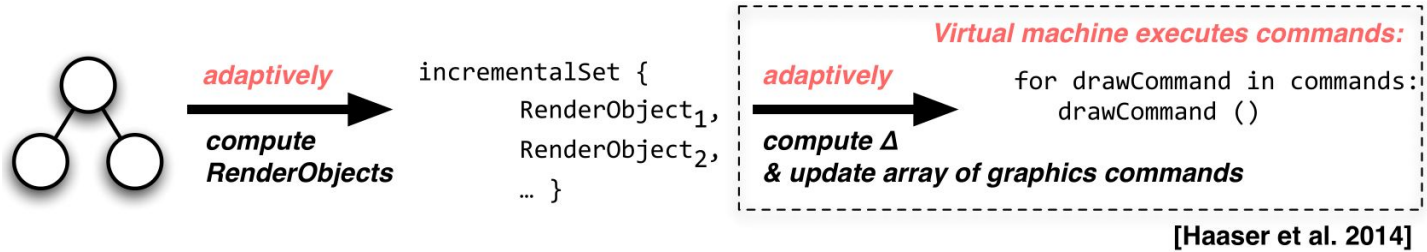
- Flexible ✅
- Lot of boilerplate when hand-written ❌
- Not compact ❌

Possible render object definition.

```
type RenderObject = {
 DrawCallInfos       : list<DrawCallInfo>
 IndirectBuffer      : IIndirectBuffer
 Mode                : IndexedGeometryMode
 Shaders             : ShaderPipeline

 DepthTest           : DepthTestMode
 CullMode            : CullMode
 BlendMode           : BlendMode
 FillMode            : FillMode
 StencilMode         : StencilMode

 Indices             : Option<BufferView>
 InstanceAttributes  : IAttributeProvider
 VertexAttributes    : IAttributeProvider

 Uniforms            : IUniformProvider

 ConservativeRaster  : bool
 Multisample         : bool

 WriteBuffers        : Option<Set<Symbol>>
}
```

```
typedef struct D3D12_GRAPHICS_PIPELINE_STATE_DESC {
  ID3D12RootSignature          *pRootSignature;
  D3D12_PRIMITIVE_TOPOLOGY_TYPE  PrimitiveTopologyType;
  D3D12_SHADER_BYTECODE          VS;
  D3D12_SHADER_BYTECODE          PS;
  D3D12_SHADER_BYTECODE          DS;
  D3D12_SHADER_BYTECODE          HS;
  D3D12_SHADER_BYTECODE          GS;
  D3D12_STREAM_OUTPUT_DESC       StreamOutput;
  D3D12_BLEND_DESC               BlendState;
  UINT                           SampleMask;
  D3D12_RASTERIZER_DESC          RasterizerState;
  D3D12_DEPTH_STENCIL_DESC       DepthStencilState;
  D3D12_INPUT_LAYOUT_DESC        InputLayout;
  D3D12_INDEX_BUFFER_STRIP_CUT_VALUE IBStripCutValue;
  UINT                       NumRenderTargets;
  DXGI_FORMAT                RTVFormats[8];
  DXGI_FORMAT                DSVFormat;
  DXGI_SAMPLE_DESC           SampleDesc;
  UINT                       NodeMask;
  D3D12_CACHED_PIPELINE_STATE  CachedPSO;
  D3D12_PIPELINE_STATE_FLAGS   Flags;
} D3D12_GRAPHICS_PIPELINE_STATE_DESC;
```

Aardvark is a little different



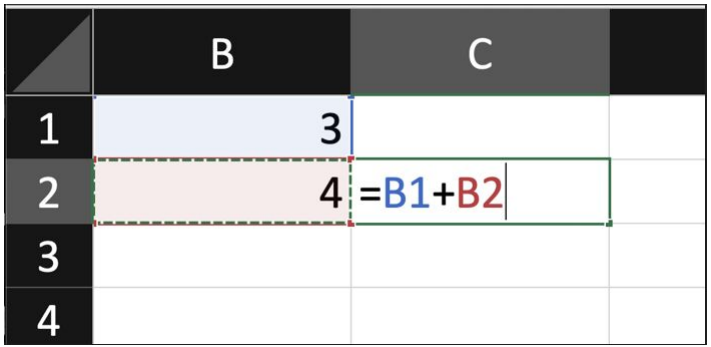
[Haaser et al. 2014]

```
type ISg =
    abstract GetRenderObjects : TraversalState -> aset<IRenderObject>
```

https://github.com/fsprojects/FSharp.Data.Adaptive

What is Incremental Computation?

- Compute only what is needed as inputs change
- Track dependencies in code
- Re-execute affected parts when changes happen




```
main: main.o lib.o
	gcc -o main main.o lib.o

main.o: main.c
	gcc -c main.c

lib.o: lib.c
	gcc -c lib.c
```

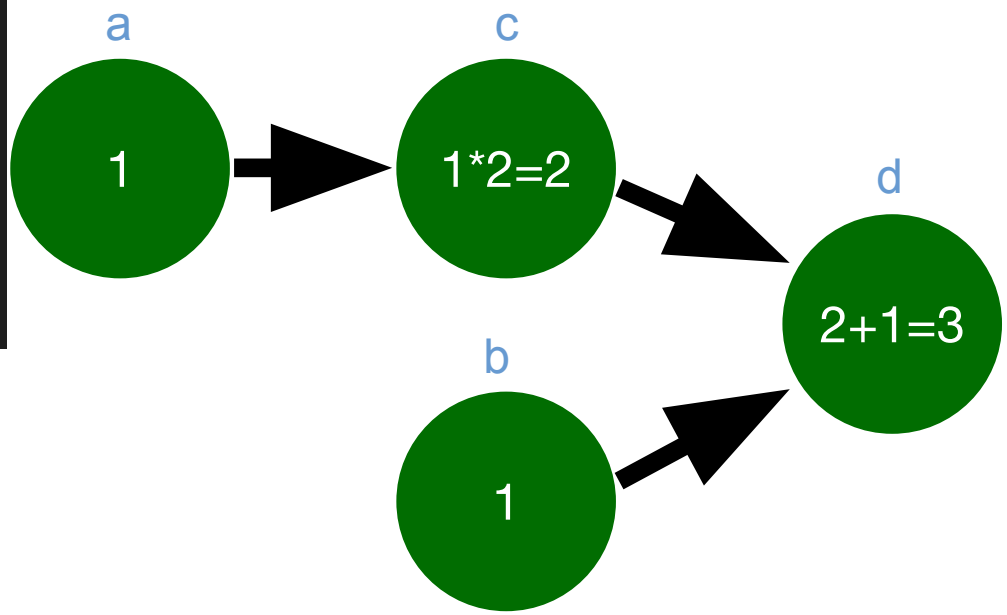
Implementation techniques for dynamism

- Reactive Programming (Publish/Subscribe, Rx, Data-flow systems)
- Incremental evaluation
  - adapt output of an algorithm respecting change in input
  - huge research area, see [Ramalingam, Reps 1993]
- Self-adjusting computation (eager evaluation, change-propagation based)
  - Umut Acar et al.'s work including starting with "Adaptive Functional Programming", his phd thesis "Self-adjusting computation"
- Incremental evaluation based on lazy evaluation
  - Hudson's work on Incremental Attribute algorithm for graphs
  - Particularly well suited for scene graphs (see our work [Wörister et al. 2012])
  - Hammer et al.'s Adapton general purpose language/system but with on-demand evaluati
  - In aardvark we used an implementation FSharp.Data.Adaptive similarly to Hammer's Adapton
- Potential requirements for engines:
  - Minimal updates, Batch Updates, Lazy Updates (for culled parts), Switching (fast switching between variants), low overhead, no global locking needed

```fsharp
type aval<'a> =
    abstract Dirty : IEvent<unit>
    abstract GetValue : unit -> 'a

type cval<'a> =
    interface aval<'a>
    new : 'a -> cval<'a>
    abstract Value : 'a with get, set

module AVal =
    val constant : 'a -> aval<'a>
    val map : ('a -> 'b) -> aval<'a> -> aval<'b>
    val bind : ('a -> aval<'b>) -> aval<'a> -> aval<'b>
```

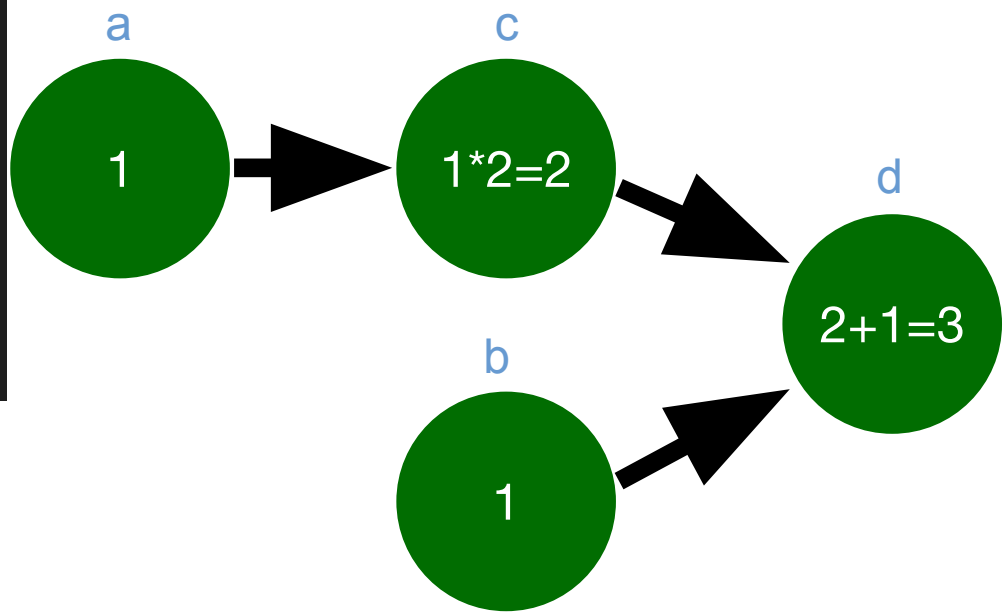
https://github.com/fsprojects/FSharp.Data.Adaptive

```
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d
```



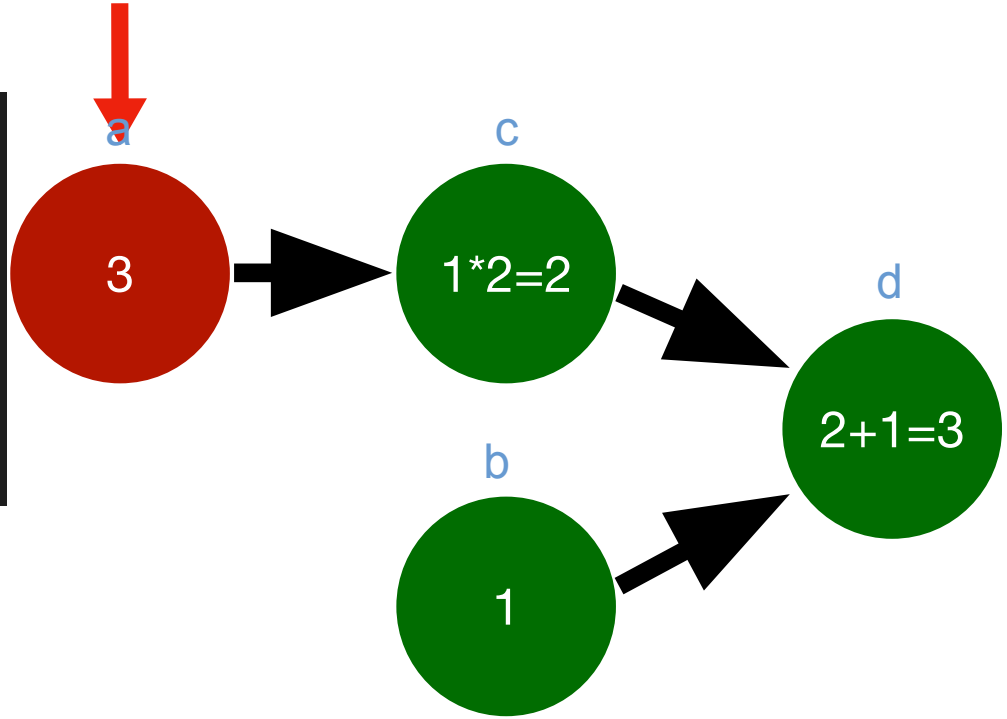
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)
```



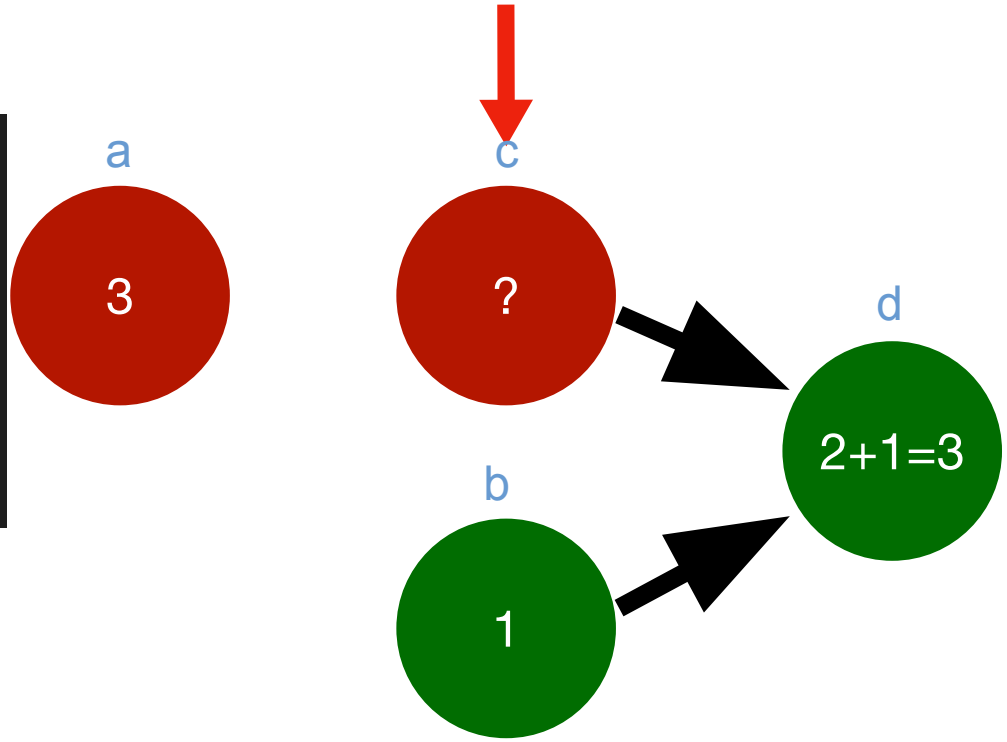
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)
```



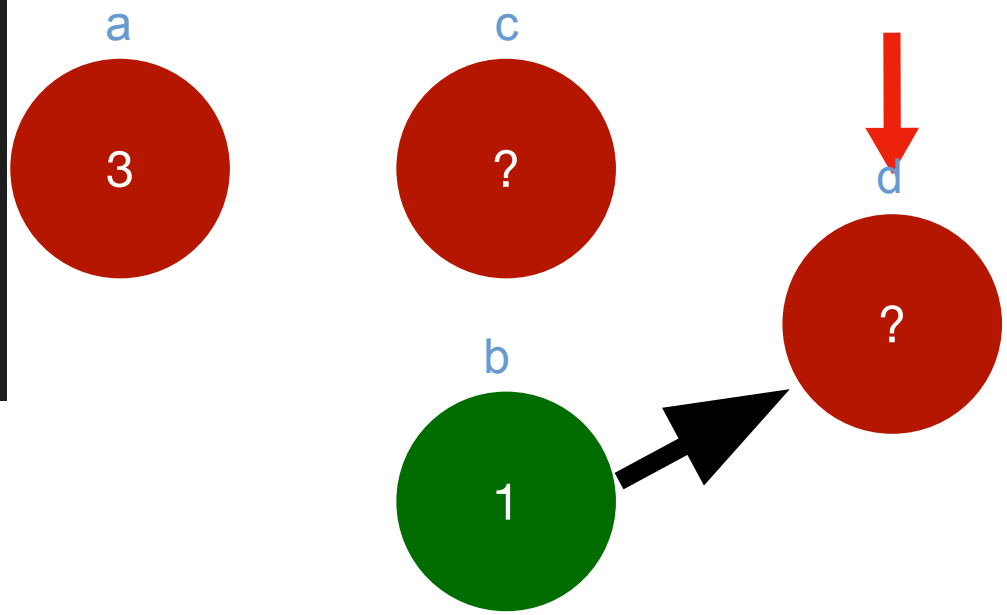
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)
```



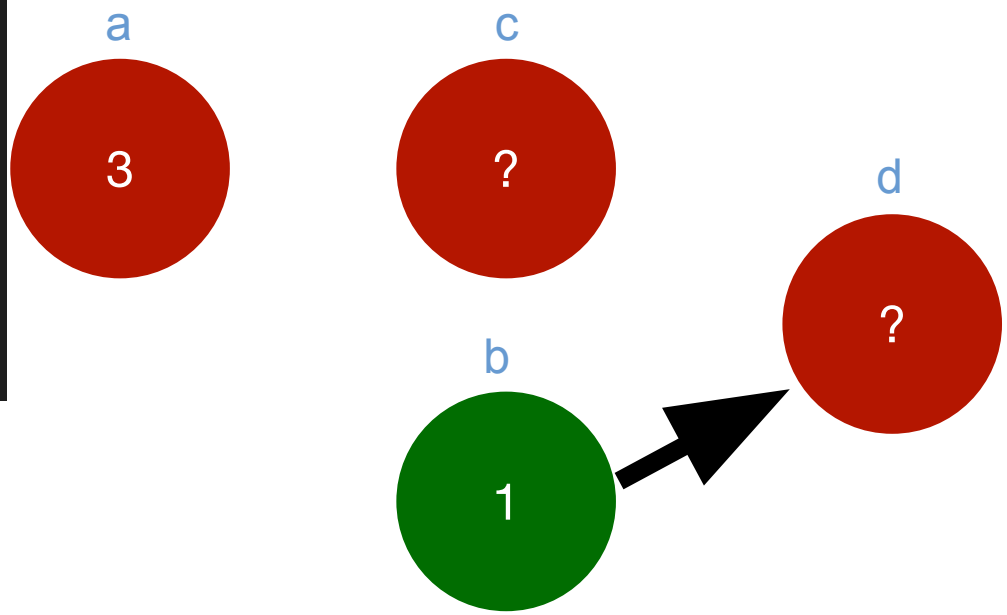
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)
```



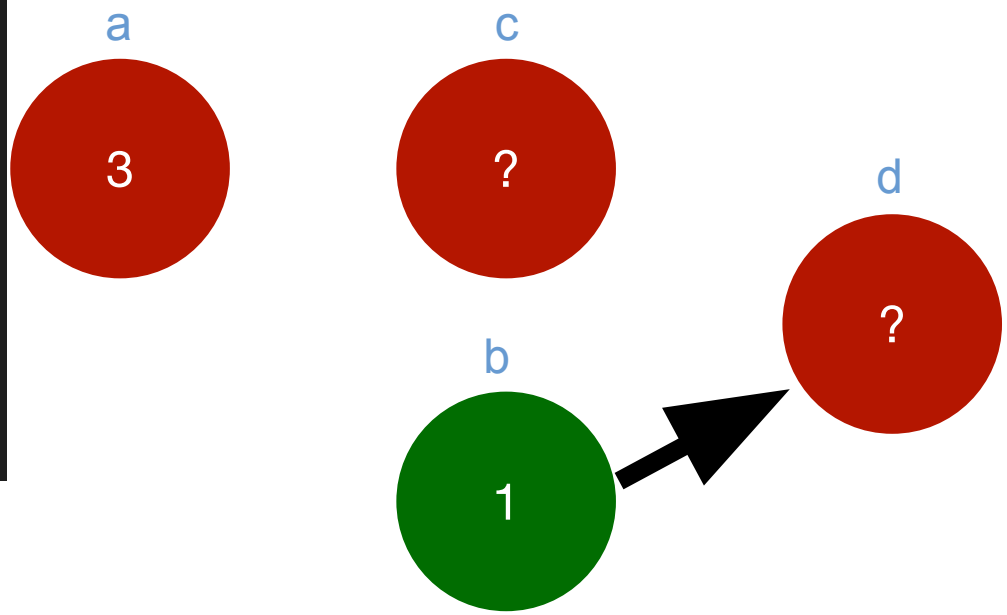
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)
```



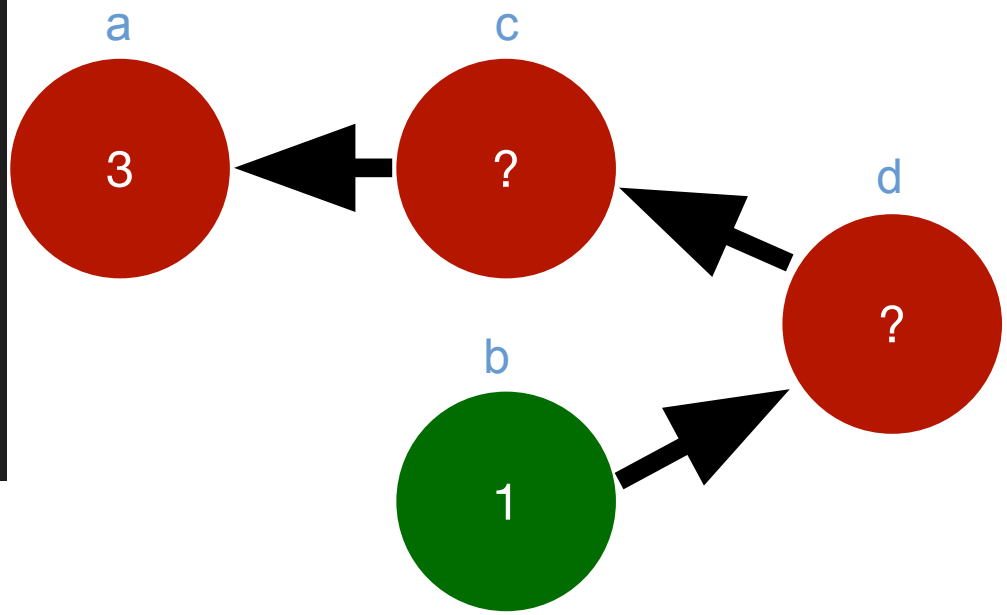
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)

AVal.force d
```



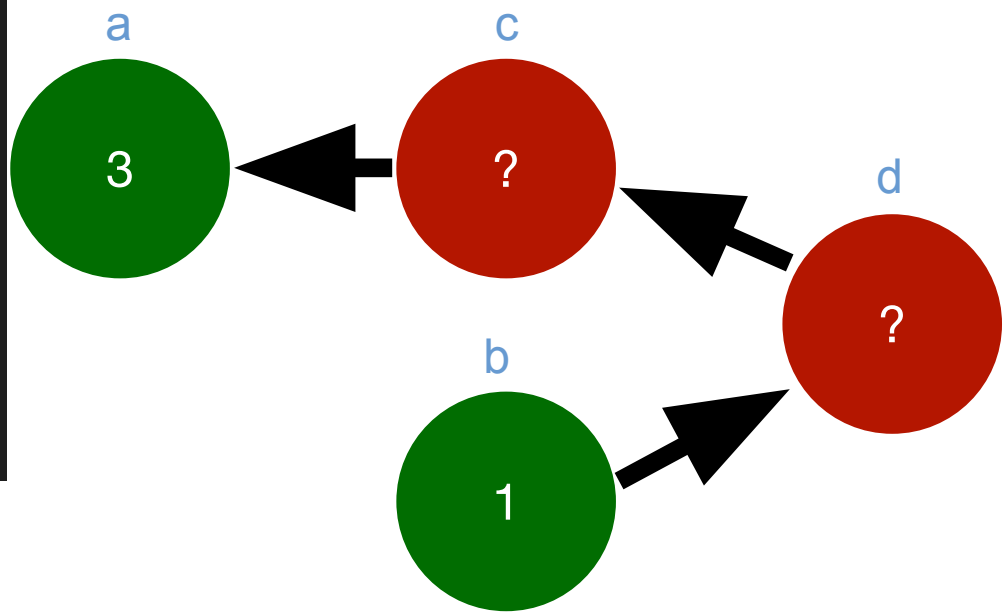
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)

AVal.force d
```



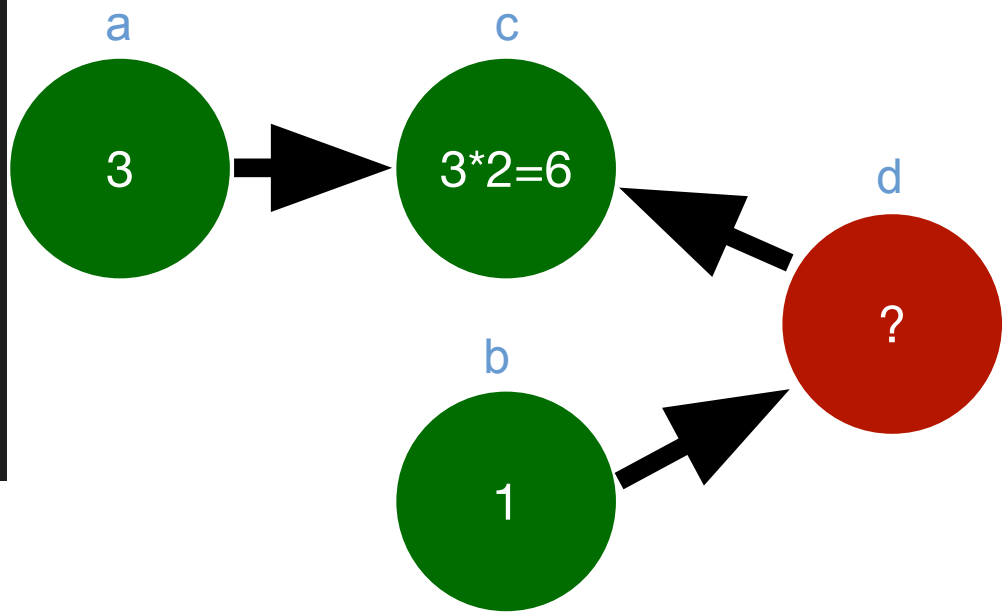
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)

AVal.force d
```



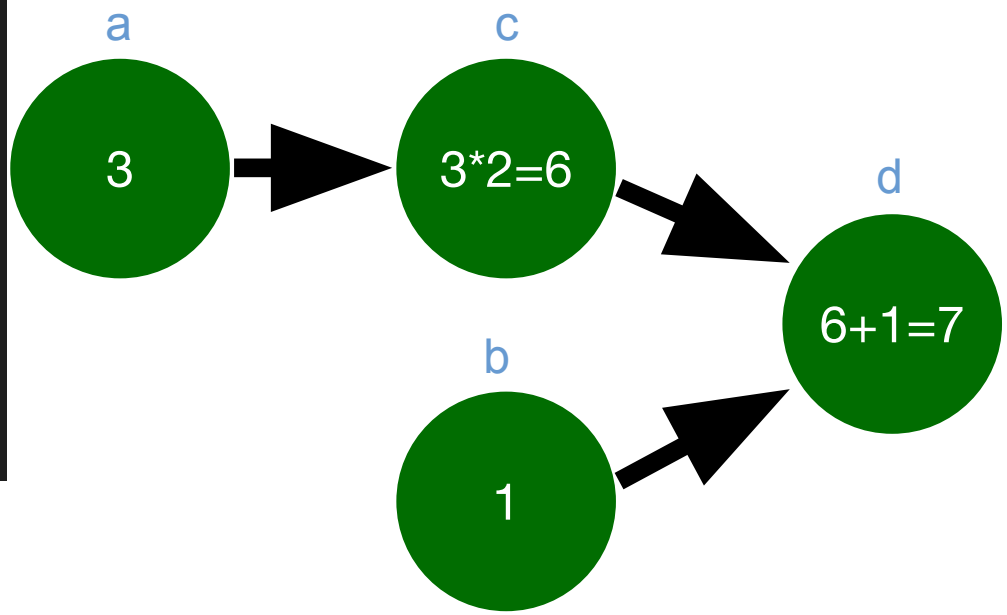
https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)

AVal.force d
```



https://github.com/fsprojects/FSharp.Data.Adaptive

```fsharp
let a = cval 1
let b = cval 1

let c = AVal.map (fun x -> x*2) a
let d = AVal.map2 (fun x y -> x+y) c d

transact (fun () ->
  a.Value <- 3
)

AVal.force d
```



https://github.com/fsprojects/FSharp.Data.Adaptive

Extension to collection types

Easy approach: `aval<array<int>>`

More fine-grained approach: *ChangeableSet, ChangeableList, ChangeableArray* and adaptive versions thereof.

Changes per element.

Related work: [Maier, Odersky 2013]

Backend

Needs to deal with a incremental list of RenderObjects (backend agnostic)

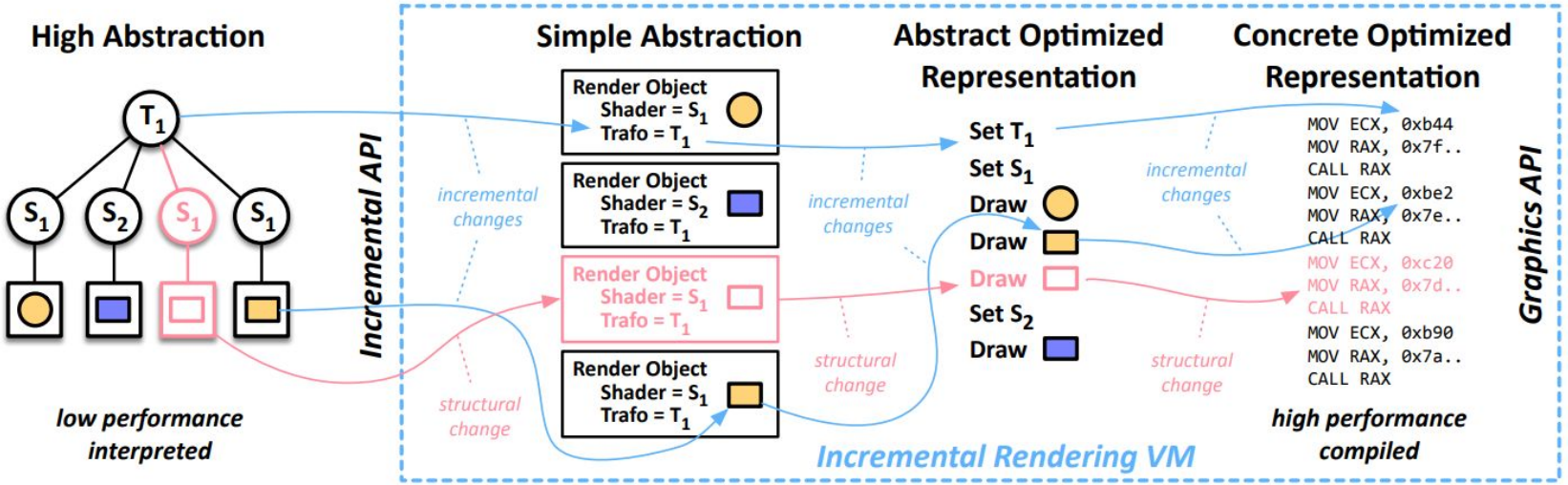
uniforms/transformations/etc. are incremental values too

Important optimizations:

- redundancy removal (OpenGL/DX11)
- state-sorting (e.g. group by shader)

Implementation

- Interpreter (loop over all objects)
- JIT (compile code for each object)



An incremental rendering VM [Haaser et al. 2015]

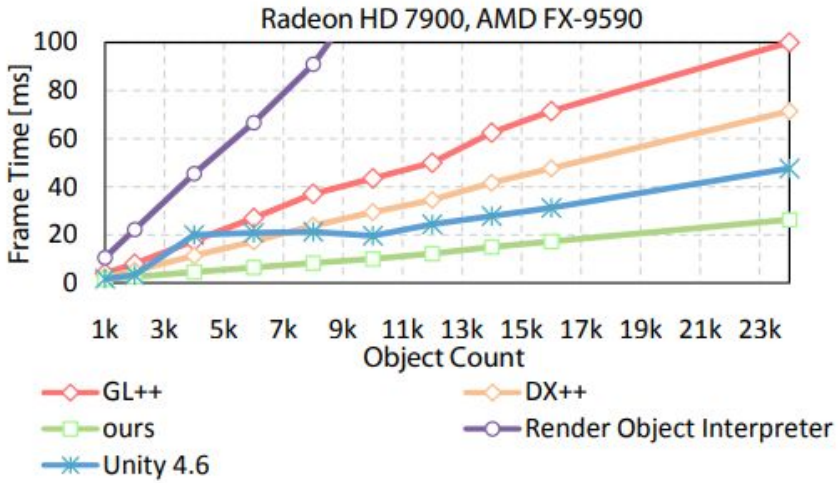
Paper proposes incremental evaluation throughout the backend

1. Incremental list of rener objects
2. Optimization: Sorting (e.g. Front to back), Group by Pipeline, State Sorting (GL)
3. Code generation
   - Generates Rendering API specific code
   - Redundancy Removal,
   - Low overhead execution via JIT

Lecture from 2020 online on youtube

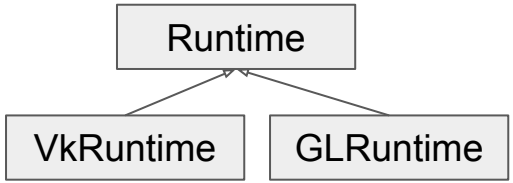
Scalability via *incremental evaluation*

Follows tradeoff: longer startup, faster execution.



Paper evaluates Virtual Machine, Interpreter Implementations & Adaptive Compilation for Rendering Systems

Dependency-aware Abstractions



Runtime interface contains

- Rendering VM implementation
- Resource creation methods

Requirements:

Optimize scene graph once, execute often.

Optimization:
SceneGraph -> runtime.CompileRender() -> `RenderTask`

Execution:
RenderTask.Run(framebuffer)

Resource consistency:

Marking Phase: render task's resources marked dirty

Rendering Phase: render task brings up to date all resources (by pulling the dependencies), patching JIT code/re-execution of command buffers if needed and queue submission

Rendering + Framebuffer output

mainloop based, imperative rendering

```
while( !quit ) {
        while( SDL_PollEvent( &e ) != 0 ) {
            if( e.type == SDL_QUIT ) quit = true;
            SDL_RenderClear( gRenderer );
            SDL_RenderCopy( gRenderer, gTexture, NULL, NULL );
            SDL_RenderPresent( gRenderer );
        }
}
```

Structuring “imperative” rendering pipelines in a graph (e.g. frame graph) allows for parallel execution?

adaptive, on-demand rendering

going via aval<ITexture> and blit to swap-chain uncouples from windowing system.

```
module RenderTask =

    let renderToColor (size : aval<V2i>) (task : IRenderTask) : aval<ITexture> =
        // ...
```

Resource Management

User-explicit (create buffer, load data, dispose buffer, …)

- tedious but flexible
- what happens to backend specific resources? (DescriptorSets, etc.)

Automatically managed (jpeg texture class, …)

- easy to use
- hard to customize
  - add unsupported features
  - e.g. add async loading
  - tempting to break abstraction

We opted for a mixed approach

- backend manages resources by default
- some kinds of user-explicit prepared resources are accepted by the SceneGraph (textures, attribute buffers, …)
- Unity, Unreal also do it like this

Beyond low-level optimizations

- Objects to be drawn
  - thousands: naive approach
  - 10 thousands: Rendering VM etc
  - Millions: Packed, geometries
- How to handle millions of objects
  - engine automatically packs geometries
  - users pack geometries

Two tools

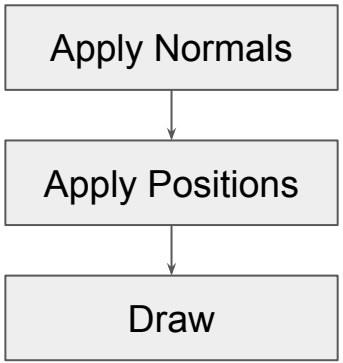
- Flexible scene graph draw call api
- Geometry pool

Draw Calls in the scene graph

Typically scene graph has “geometries”

In aardvark we use attributes inherited in the scene graph for each vertex attribute (“pure” scene graph implementation)

```
// standard scene graph construction for indirect buffer based rendering
Sg.indirectDraw IndexedGeometryMode.TriangleList (AVal.constant indirectBuffer)
|> Sg.vertexBuffer DefaultSemantic.Positions vertices
|> Sg.vertexBuffer DefaultSemantic.Normals   normals
// attach storage buffers to uniform slots
|> Sg.uniform "ObjectColors" objectColors
|> Sg.uniform "MeshTrafo"    perKindAnimation
// not to forget the index buffer
|> Sg.index' packedGeometry.indices
```



https://github.com/aardvark-platform/aardvark.rendering/blob/edf2da5344c23c819c297dde484943db3fa77737/src/Examples%20(netcore)/23%20-%20FlexibleDrawCommands/Program.fs#L145

Shader Systems

Directly using backend-shaders (GLSL/HLSL/SpirV) locks you to the backend

No shader graph system etc., plain language approach (currently)

High Level DSLs that can generate the backend-code

- external DSLs
  - extra compiler
  - live in separate files
- internal DSLs
  - directly embedded in the rendering-engine's language (F#)

There will be a Lecture on Shader Systems

https://github.com/krauthaufen/FShade

Aardvark Recap

- uses a SceneGraph
- synthesizes an incremental set of RenderObjects from the SceneGraph
- uses an internal DSL for shaders (FShade)
- manages resources automatically and has a “backdoor” for prepared resources
- has multiple backends (OpenGL/Vulkan/WebGL) that use JIT-compiler
- uses incremental textures for putting together rendering-pipelines

User Interaction

Picking module integrated

Design space (application programmer perspective)

- All application, rendering engine only for rendering
- Integration in Scene Description


```
<button onclick="myFunction()">Click me</button>
```

Implementation techniques

- Readback based (e.g. object id’s stored to framebuffer attachment)
- Geometric (Ray-Scene Intersections, e.g. using *B*ounding *V*olume Hierarchy)

Integration in Scene Description

```
sg {
    Sg.Shader { Shader.simpleLighting }
    Sg.BlendMode BlendMode.Add

    sg {
        Sg.OnClick (fun e ->
            printfn "clicked sphere: %A" e.WorldPosition
        )
        Sg.Translate(10.0, 0.0, 0.0)
        Primitives.Sphere(radius = 1.0, color = C4b.Red)
    }
    sg {
        Sg.Scale 10.0
        Primitives.Box(size = V3d(2,2,2))
    }
}
```

Examples

Examples

- Simple Primitives with transforms/etc.
- simple shadow-mapping maybe?

Retrospect: Interaction & Application Development in Aardvark

Looking back

- We used the “*Semantic Scene Graph*” approach [Tobler 2011] for “domain” logics of applications

- More complex logic still needs to be done in application
- We moved to a purely functional approach for application programming in ~2017

Functional Programming for 3D Graphics

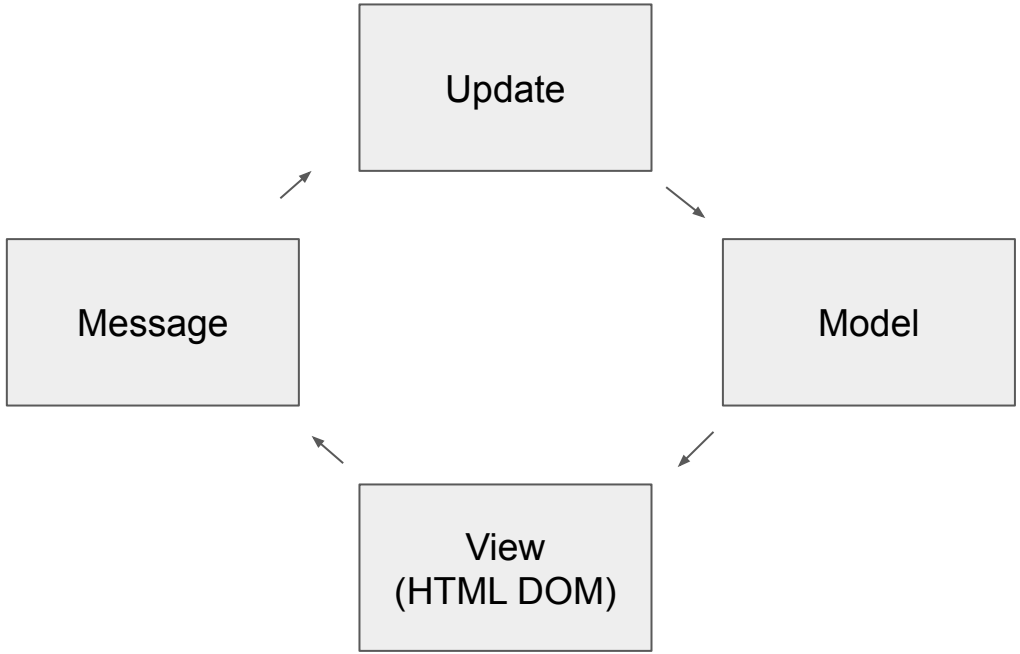
Motivated by ELM

Unidirectional dataflow

Purely functional application model

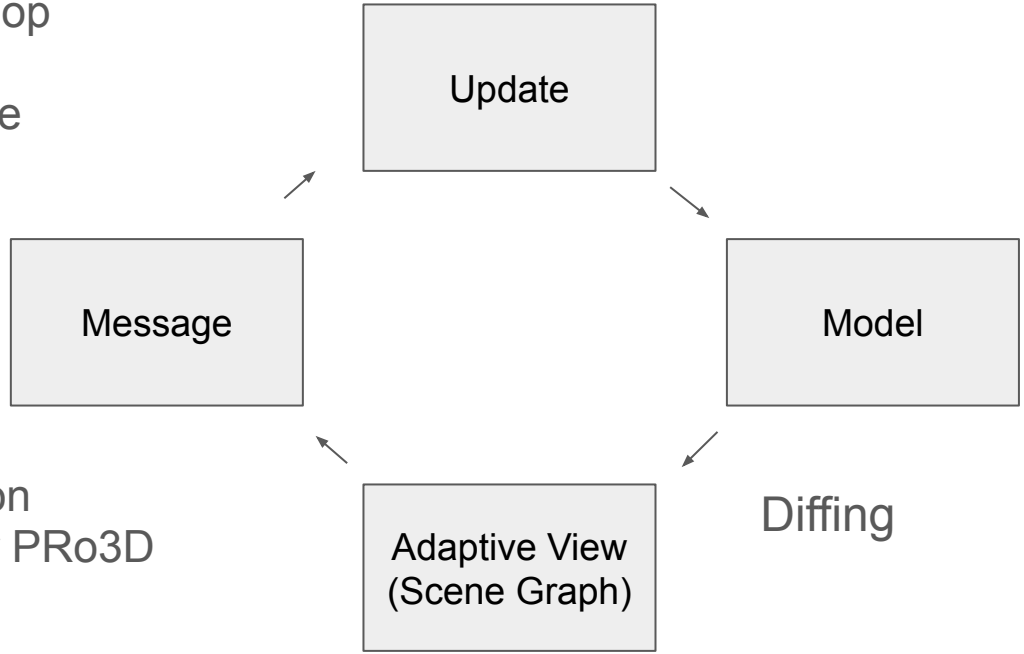
The view function generates HTML DOM

Our idea: The view also creates a scene graph.



Functional Programming for 3D Graphics

- In Aardvark we use an ELM style loop
- We perform diffing on the immutable application state.
- Changes are fed into the adaptive scene graph.
- Some info presented here:
  - GPU Days 2018
  - FableConf 2019
- Used in many applications, transition from OOP to FP presented here for PRo3D
  - GPU Days 2019 (Thomas Ortner)



Reflections on aardvarks

- Cross-cutting concerns/overarching concerns easily not extensible
  - aspects touching all components (scene, render objects, rendering technique, shader), e.g.transparency, culling, shadows
- Principled approach to changes pays off
  - No need to think about “invalid” states (inconsistencies), optimized update routines
  - Effort usually put in optimizing update granularity
  - Easy to maintain!
- Adaptive rendering solves the question
  - “do i need to update a particular framebuffer”
- Many optimizations not possible because
  - shaders/shader inputs are completely up to the user
  - more restrictions to enable optimizations?
- Aardvark goes all in for incremental changes
  - high startup costs
  - not much parallelization done

small changes =>
incremental useful

many changes
data-management =>
parallel execution useful

Future challenges

- Handling the fully dynamic case (without loosing the general scene graph?)
  - workloads where incremental evaluation does not help
  - but parallel execution (& ECS style memory) does (+ parallel command buffer recording)
  - Interesting work by [Burckhardt et al. 2011] combining incremental & parallel execution
- Granularity of changes
  - needs to be well-chosen by application programmer (depending on trade-offs)
  - `aval<array<int>>` vs `aset<int>`
- Diagnosis/Profiling/Finding bottlenecks
- Dealing with changes explicitly has its drawback
  - Adaptive evaluation DSL makes it easier, but has its complexity
  - Changes might become unexpectedly expensive
- Hints when using “slow” patterns in scene graph
  - e.g. many separate text nodes vs rendering many texts using a specialized texts function
  - Code analyzer? AI approach? Execution graph? Profiling?

References

- [Wörister et al. 2012] “Lazy Incremental Computation for Efficient Scene Graph Rendering”, HPG 2013
- [Tobler 2011], “Separating semantics from rendering: a scene graph based architecture for graphics applications”, “The Visual Computer”
- [Acar et al. 2002], Adaptive Functional Programming, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). 2002. Umut A. Acar, Guy E. Blelloch, and Robert Harper.
- [Acar 2005] Self-Adjusting Computation, Phd thesis, 2005.
- [Hudson 1991], Incremental attribute evaluation: a flexibel algorithm for lazy update, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 13, 1991
- [Hammer et al. 2014], “Adapton: composable, demand-driven incremental computation”, ACM SIGPLAN, Volume 49, Issue 6. Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, Jeffrey S. FosterAuthors Info & Claims
- [Ramalingam, Reps 1993], “A categorized bibliography on incremental computation”, G. Ramalingam, Thomas Reps, POPL 1993
- [Haaser et al. 2015] “An Incremental Rendering VM”, Georg Haaser, Harald Steinlechner, Stefan Maierhofer, Robert F. Tobler, HPG’15,
- [Lalonde and Schenk. 2002]. Shader-driven compilation of rendering assets. In Proc. of the 29th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, USA, SIGGRAPH ’02, 713–720
- [Zachmann 17] Lecture on scene graphs https://cgvr.cs.uni-bremen.de/teaching/vr_1718/folien/02%20-%20Scenegraphs,%20VRML,%20game%20engine%20principles.pdf
- [Haaser 2020] Lecture on “An Incremental Rendering VM” https://www.youtube.com/watch?v=71p1MDjLjZI
- [GPU Days 2018] “Functional programming vs. Efficient Computer Graphics” and “Functional Programming in the Wild”
- [FableConf 2019], “Functional Adventures on High Performance Computer Graphics”
- [Ortner, GPU Days 2019], “Functional Programming boosting scientific and industrial research”
- [Burckhardt et al. 2011], “Two for the price of one: a model for parallel and incremental computation”, Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, Thomas Ball, OOPSLA’11
- [Maier, Odersky 2013], “Higher-Order Reactive Programming with Incremental Lists”

Some additional details

Reactive Programming in Rendering Engines

Reactive programming used in many areas (e.g. web apps)

Implementation via Publish/Subscribe uses eager change propagation

Drawbacks

- No batch changes
- No minimal change propagation

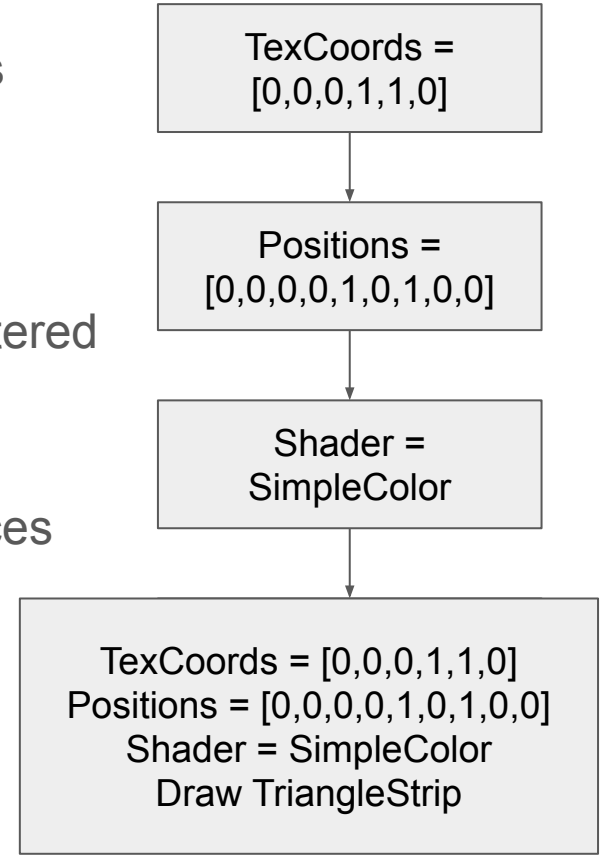
Example: diamond

How to Render a SceneGraph 1

Some nodes cannot decide what to do with their Attributes (may be overwritten further down the graph)

Attributes are typically passed along until a leaf is encountered

For each leaf the system can now create backend resources like buffers/textures/etc. as needed



How to Render a SceneGraph 2

Issue Backend-Calls / create Resources while traversing the Graph

- hard to detect which parts are deleted (no longer visited)
- needs caches in the original graph
- overhead

Synthesize a list of all leaves with all inherited attributes
(like ComputedStyle in HTML debuggers)

- how to deal with dynamism? (list diffing would be an option)