Technical Report: Deriving a Domain Specific Language for purely functional 3D Graphics

The Aardvark Platform Team

05.05.2017

In this paper, we present a simple yet powerful Domain Specific Language for working with three-dimensional scene data. We start by specifying the domain types (see DDD) of a example problem, and extend the domain model with functionality for rendering and interaction.

Our first example is a little drawing tool which allows users to draw polygons on a vertical plane centered in the 3D scene. Later, we will extend the program with functionality for picking and translating objects by using a Maya-style 3D controller.

Note that this document is written as F# literate script, i.e. this document is both, the paper and the implementation ;)

Domain Driven Design

Let us start with the domain model for polygons:

```
1: open Aardvark.Base // Aardvark.Base provides vector types
2:
3: type Polygon = list<V3d>
```

Thus, a polygon is simple a immutable list of vectors (vertices). Next, we need a way to model polygons which are not yet fully defined. A mouse click extends these, and eventually they are finalized.

```
1: type OpenPolygon = {
2:     finishedPoints  : list<V3d> // points already added to the polygon
3:     cursor          : Option<V3d>
4:     (* the last point of the polygon which can still be modified.
5:     Note that the last point of the polygon might be undefined
6:     (represented as None). *)
7: }
```

The Domain Model can be completed using a list of polygons in the scene, as well as one open polygon the user is currently working on.

```
1: type DrawingModel = {
2:     finished : list<Polygon>
3:     working  : Option<OpenPolygon>
4: }
```

Creating a visual representation for our model

The simplest way of specifying a visual representation for data is by specifying a function which maps from data to graphics. Since we aim for rendering our graphics using graphics hardware (OpenGL), we don't directly compute pixels for the data, but we map our domain model to an explicit description of the scene, which we can render efficiently. Aardvark uses the typical description of this sort, a scene graph with geometric primitives at the leaves.

Let us start with the type describing 3D entities in the scene:

```
type Primitive =
    | Sphere      of center : V3d * radius : float
    | Cone        of center : V3d * dir : V3d * height : float * radius : float
    | Cylinder    of center : V3d * dir : V3d * height : float * radius : float
    | Quad        of Quad3d
```

This type allows us to specify individual entities in a 3D scene.
Next let us extend the API to support many entities being packed together.

```
type Scene1 =
        | Transform of Trafo3d * list<Scene1>
        | Colored   of C4b     * list<Scene1>
        | Render    of Primitive
        | Group     of list<Scene1>
```

This definition allows us to describe scenes.
An example:

```
let scene =
    Transform ( Trafo3d.Translation(10.0,10.0,10.0), [
                  Render ( Sphere(V3d.OOO, 1.0))
                  Render ( Sphere(V3d.OOO, 0.1))
                ])
```

Which represents the following scene:
todo image
Thus, a visual representation for our polygon-sketching scene can be implemented as follows:

```
// curried version of Colored in order to nicely compose
let colored1 c xs = Colored(c,xs)
let transformed1 t x = Transform(t,[x])

let view1 (m : DrawingModel) : Scene1 =

    let groundPlane =
        [ Quad (Quad3d [| V3d(-1,-1,0); V3d(1,-1,0); V3d(1,1,0); V3d(-1,1,0) |])
          |> Render
        ]
        |> colored1 C4b.Gray

    let viewPolygon (p : list<V3d>) : Scene1 =
        [ for edge in Polygon3d(p |> List.toSeq).EdgeLines do
            let v = edge.P1 - edge.P0
            yield Cylinder(edge.P0,v.Normalized,v.Length,0.03) |> Render
        ] |> Group

    let openPolygon =
        match m.working with
            | None   -> [] // no open polygon -> just return empty list)
            | Some v -> [viewPolygon v.finishedPoints]

    let cursor =
        match m.working with
            | Some v when Option.isSome v.cursor ->
                // if we have a last point (cursor)
                [ [ Sphere(V3d.OOO,0.1) |> Render ]
```

```
                    |> colored1 C4b.Red
                    |> transformed1 (Trafo3d.Translation(v.cursor.Value))
                ]
            | _ -> [] // no working polygon or no cursor

    let polygons =
        m.finished |> List.map viewPolygon

    Group [
        yield  groundPlane
        yield! openPolygon
        yield! cursor
        yield! polygons
    ]
```

Altough the function is rather verbose F# (many let bindings can be inlined), it precisely captures the semantics of the transformation. Next, we would like to implement interaction techniques. The user usually wants to issue commands to the program and observe the resulting effects.

In order to model interactions cleanly, let us introduce a union type modeling all possible interactions.

```
type DrawCommand =
    | ClosePolygon
    | AddPoint   of V3d
    | MoveCursor of V3d
```

Remember, our model is immutable. Therefore, adding a point to our model actually means computing a new model, which is the old model containing the new point.

```
let updateDrawing (m : DrawingModel) (cmd : DrawCommand) =
    match cmd with
        | AddPoint p -> // we want to add a point to our drawing model
            match m.working with // do we have a polygon we could add to point to?
                | Some v ->
                    // yes we have, create a new OpenPolygon with our new point
                    // added and update the model
                    { m with working =
                        Some { v with finishedPoints = p :: v.finishedPoints }
                    }
                | None -> { m with working = Some { finishedPoints = [ p ];
                                                    cursor = None;  }}
        | ClosePolygon ->
            match m.working with
                // we have no polygon to close -> identity.
                // the updated model is the old model
                | None -> m
                | Some p ->
                    { m with
                        // close a polygon. there is no more open polygon ...
                        working = None
                         // the formerly opened polygon is now part of the finished polygons
                        finished = p.finishedPoints :: m.finished
                    }
        | MoveCursor p ->
            // the last point of the current working polygon follows the mouse cursor
            match m.working with // check if we have a working polygon
                | None ->
```

```
                    // start a new open polygon with our current cursor set to mouse position
                    { m with working = Some { finishedPoints = []; cursor = Some p }}
                | Some v ->
                    // override current cursor position
                    { m with working = Some { v with cursor = Some p }}
```

Again, the function precisely captures the semantics of our update commands. Who generates these update commands?

ClosePolygon, for example, could be associated with a mouse click. This means that mouse clicks must produce 3D positions. But a mouse click doesn't usually produce anything, so what can we do? This is exactly the point where our clean API seems to get messy.

One common approach here is to define a picking manager which handles mouse events and translates the events to domain specific data (such as 3D positions). However, the scope and responsibilities of such a manager are often difficult to define, and an implementation using mutable/global state becomes chaotic.

In this work, we take a different approach inspired by the Elm Architecture. The key insight is to perform operations exactly at the points where all necessary information is at hand, or can be computed easily. Since we already defined a DSL for specifying 3D scenes, the obvious choice is to resolve our commands there! This leads to a clean seperation of concerns and encapsulation of state.

Let us take the expression which effectively creates the drawable 3D plane in our scene:

```
let groundPlane =
    [ Quad (Quad3d [| V3d(-1,-1,0); V3d(1,-1,0); V3d(1,1,0); V3d(-1,1,0) |]) |> Render ]
    |> colored1 C4b.Gray
```

We have the domain model at hand, as well as specfic information such as 3D coordinates. In order to express pick operations here, we need to enrich our Scene representation with pick operations.

Let us first define an API for specifying picks. For our example, we need to handle clicks and mouse moves. Since we want to work with 3D data, our events should be equipped with 3D coordinates:

```
// Aardvark.Application provides helpers for working with user inputs
open Aardvark.Application

type MouseEvent =
    | Move of V3d
    | Down of MouseButtons * V3d
```

Let us now define a type which describes the reaction of a mouse event in a generic manner. This means that the produced command message is a type argument for our function type.

```
type PickOperation<'msg> = MouseEvent -> Option<'msg>
```

This abstraction here leads the way to specifying message-agnostic 3D scenes. As a consequence, our scene type now carries a type argument representing the type of message which can be produced by the scene itself.

Additionally we equip our render constructor with the capability of producing pick commands.

```
// scene produces messages of type 'msg
type Scene<'msg> =
        | Transform of Trafo3d * seq<Scene<'msg>>
        | Colored   of C4b     * seq<Scene<'msg>>
        | Render    of list<PickOperation<'msg>> * Primitive // new bits in here
        | Group     of seq<Scene<'msg>>
```

Let us define functions which provide curried constructions of our message type. This comes in handy soon:

```
let transform t xs = Transform(t,xs)
let transform' t x = Transform(t,[x])
let translate x y z xs = Transform(Trafo3d.Translation(x,y,z),xs)
let colored c xs = Colored(c,xs)
let render picks primitive = Render(picks,primitive)
let group xs = Group xs
```

We can check our design by implementing a visual representation of our drawing plane and let us associate a pick operation to it.

```
let drawGroundPlane =
    Quad (Quad3d [| V3d(-1,-1,0); V3d(1,-1,0); V3d(1,1,0); V3d(-1,1,0) |])
        |> render [
                // Pick operations for moving the cursor.
                // if the ground plane is hit by a mouse move event,
                // construct the MoveCursor command
                (fun evt ->
                    match evt with
                        | Move p -> Some (MoveCursor p)
                        | _ -> None
                )
                (fun evt ->
                    match evt with
                        | Down(MouseButtons.Left,p) -> Some (AddPoint p)
                        | _ -> None
                )
                (fun evt ->
                    match evt with
                        | Down(MouseButtons.Right,_) -> Some ClosePolygon
                        | _ -> None)
        ]
```

The above code is a pure expression, i.e. no side effects are involved. The value simply stores first class functions. When provided with a world space pick point, and the description of a mouse event, they produce a command. Although rather nice, the code is still repetitive.

Some helpers will make the function look nicer.

```
// some predicates which take in a mouseEvent and check whether
// the mouse event is interesting
module Event =
    // "function" is the combined lambda/match syntax
    let move : MouseEvent -> bool = function Move _ -> true | _ -> false
    let down = function Down _ -> true | _ -> false
    let down' p = function Down(p',_) when p = p' -> true | _ -> false
    let leftDown = down' MouseButtons.Left
    let rightDown = down' MouseButtons.Right
    let position = function Move s -> s | Down(_, s) -> s

// Look back on the repetitive code.
// One can see that all PickOperations perform some filtering/matching on the
// observed input.

// This pattern can be catpured by a higher order function such as:
let on (p : MouseEvent -> bool) (r : V3d -> 'msg) (k : MouseEvent) =
    if p k then Some (r (Event.position k)) else None
```

Applied to our original drawGroundPlane, the code duplication disappears:

```
let drawGroundPlane2 =
    Quad (Quad3d [| V3d(-1,-1,0); V3d(1,-1,0); V3d(1,1,0); V3d(-1,1,0) |])
        |> render [
                on Event.move      (fun p -> MoveCursor p)
                on Event.leftDown  (fun p -> AddPoint p)
```

```
                on Event.rightDown (fun _ -> ClosePolygon)
            ]
```

Eta reduced:

```
let drawGroundPlane3 =
    Quad (Quad3d [| V3d(-1,-1,0); V3d(1,-1,0); V3d(1,1,0); V3d(-1,1,0) |])
        |> render [
                on Event.move      MoveCursor
                on Event.leftDown  AddPoint
                on Event.rightDown (fun _ -> ClosePolygon)
            ]
```

Since moving cursors, adding points and closing polygons are the only operations in our example, our implementation is complete.

Our final DSL

Next we define a type which captures two functions and a domain model:

```
type App<'model,'msg,'view> =
    {
        initial   : 'model
        update    : 'model   -> 'msg -> 'model
        view      : 'model   -> 'view
    }
```

In that context, we always work with views of the type Scene¡'msg¿. Therefore this type alias makes sense:

```
type ThreeDApp<'model,'msg> = App<'model,'msg,Scene<'msg>>
```

We refactor the app instance for our polygon-sketching example, including the view function, a bit:

```
module Pick =
    let ignore = [] // empty list of pick operations shorthand


let viewDrawing (m : DrawingModel) =
    let viewPolygon (p : list<V3d>) =
        [ for edge in Polygon3d(p |> List.toSeq).EdgeLines do
            let v = edge.P1 - edge.P0
            yield Cylinder(edge.P0,v.Normalized,v.Length,0.03) |> render Pick.ignore
        ] |> group

    group [
        yield [ Quad (Quad3d [| V3d(-1,-1,0); V3d(1,-1,0); V3d(1,1,0); V3d(-1,1,0) |])
                    |> render [
                            on Event.move MoveCursor
                            on Event.leftDown  AddPoint
                            // // constF == fun a -> fun _ -> a
                            on Event.rightDown (constF ClosePolygon)
                        ]
                ] |> colored C4b.Gray
        match m.working with
            | Some v when v.cursor.IsSome ->
                yield
                    [ Sphere(V3d.OOO,0.1) |> render Pick.ignore ]
```

```
                        |> colored C4b.Red
                        |> transform' (Trafo3d.Translation(v.cursor.Value))
                yield viewPolygon (v.cursor.Value :: v.finishedPoints)
            | _ -> ()
        for p in m.finished do yield viewPolygon p
    ]

let drawingApp = {
    initial = { finished = []; working = None }
    update  = updateDrawing
    view    = viewDrawing
}
```

It's no accident. Our final formulation matches the formulation of Elm applications!
In other words, we just reinvented the core part of the Elm Architecture!

An implementation for our API using the aardvark rendering engine.

Proof of Concept: Rendering values of type Scene

We want to render the scene using the Aardvark rendering engine.

To achieve this, there are several possibilities: *Manually compute Render Objects for our scene.* Create a Scene Graph which describes our scene in rendering terms.

In this work we chose the latter. Please note that the implementation is far from optimal, but serves as a simple base that we can optimize subsequently.

```
[<AutoOpen>]
module ConvertToSceneGraph =

    open Aardvark.SceneGraph

    type State = { trafo : Trafo3d; color : C4b }

    open IndexedGeometryPrimitives

    let createQuad p =
        IndexedGeometry(
            IndexedGeometryMode.TriangleList, [| 0; 1; 2; 0; 2; 3 |],
            SymDict.ofList [
                DefaultSemantic.Positions, p.Points |> Seq.map V3f |> Seq.toArray :> Array
                DefaultSemantic.Colors   ,  Array.replicate 4 s.color  :> System.Array;
                DefaultSemantic.Normals,
                    Array.replicate 4 (p.Edge03.Cross(p.P2-p.P0)).Normalized :> System.Array
            ], SymDict.empty)

    let toSg (scene : Scene<'msg>) : ISg =
        let rec toSg (s : State) (scene : Scene<'msg>) =
            match scene with
                | Transform(t,children) ->
                    children |> Seq.map ( toSg { s with trafo = s.trafo * t } ) |> Sg.group'
                | Colored(c,children) ->
                    children |> Seq.map ( toSg { s with color = c } ) |> Sg.group'
                | Render (_, Cone(center,dir,height,radius)) ->
                    solidCone center dir height radius 10 s.color
                    |> Sg.ofIndexedGeometry |> Sg.transform s.trafo
```

```
                | Render (_, Cylinder(center,dir,height,radius)) ->
                    solidCylinder center dir height radius radius 10 s.color
                    |> Sg.ofIndexedGeometry |> Sg.transform s.trafo
                | Render (_, Sphere(center,radius)) ->
                    solidSubdivisionSphere (Sphere3d(center,radius)) 5 s.color
                    |> Sg.ofIndexedGeometry |> Sg.transform s.trafo
                | Render(_, Quad(p)) ->
                    createQuad p |> Sg.ofIndexedGeometry |> Sg.transform s.trafo
                | Group xs -> xs |> Seq.map ( toSg s) |> Sg.group'

        toSg { trafo = Trafo3d.Identity; color = C4b.White } scene
```

The function toSg is a mapping from our scene representation to the Aardvark.Rendering Scene Graph.

Now we can render scenes. Next, we implement a simple picking scheme which computes command messages out of mouse clicks:

```
module Picking =
    // implement ray - object intersections for all renderable primitives:
    let hitPrimitive (p : Primitive) (trafo : Trafo3d) (ray : Ray3d) action =
        let mutable ha = RayHit3d.MaxRange
        match p with
            | Sphere(center,radius)->
                let transformed = trafo.Forward.TransformPos(center)
                let mutable ha = RayHit3d.MaxRange
                if ray.HitsSphere(transformed,radius,0.0,Double.PositiveInfinity, &ha) then
                    [ha.T, action]
                else []
            | Cone(center,dir,height,radius) | Cylinder(center,dir,height,radius) ->
                let cylinder = Cylinder3d(trafo.Forward.TransformPos center,
                                          trafo.Forward.TransformPos (center+dir*height),radius)
                let mutable ha = RayHit3d.MaxRange
                if ray.Hits(cylinder,0.0,Double.MaxValue,&ha) then
                    [ha.T, action]
                else []
            | Quad q ->
                let transformed = Quad3d(q.Points |> Seq.map trafo.Forward.TransformPos)
                if ray.HitsPlane(Plane3d.ZPlane,0.0,Double.MaxValue,&ha) then [ha.T, action]
                else []

    // given a ray and a scene -> perform a PickOperation.
    let pick (r : Ray3d) (s : Scene<'msg>)  =
        let rec go (state : State) s =
            match s with
                // collect pick operations for all children
                | Group xs -> xs |> Seq.toList |>  List.collect (go state)
                | Transform(t,xs) ->
                    xs |> Seq.toList |> List.collect (go { state with trafo = state.trafo * t })
                | Colored(_,xs) -> xs |> Seq.toList |> List.collect (go state)
                | Render(action,p) ->
                    // do the actual work
                    hitPrimitive p state.trafo r action
        match s |> go { trafo = Trafo3d.Identity; color = C4b.White } with
            | [] -> []
            | xs ->
                // sort PickOperations by camera distance
```

```
40:                 xs |> List.filter (not << List.isEmpty << snd) |>  List.sortBy fst
```

Up to now, all presented functions are pure! For running the application, however, we need one single (!) side effecting function.

In Aardvark.Rendering Scene Graphs are compiled into an IRenderTask. This is then assigned to an IRenderControl, which displays the result.

The following code runs the evaluation of Elm-style program logic (the "Elm loop"), and attaches the 3D app to a IRenderControl.

```
1:  open Aardvark.Base.Incremental
2:  open Aardvark.SceneGraph
3:
4:  // note: this function contains side effects. we just left the functional programming world!
5:  let createApp (ctrl : IRenderControl) (camera : IMod<Camera>)
6:                (app : App<'model,'msg, Scene<'msg>>) =
7:
8:      let mutable model = app.initial
9:      let view = Mod.init (app.view model)
10:     let sceneGraph = view |> Mod.map ConvertToSceneGraph.toSg |> Sg.dynamic
11:
12:     let updateScene (m : 'model)  =
13:         let newView = app.view m
14:         transact (fun _ ->
15:             view.Value <- newView
16:         )
17:
18:     let handleMouseEvent (createEvent : V3d -> MouseEvent) =
19:         let ray = ctrl.Mouse.Position |> Mod.force  |> Camera.pickRay (camera |> Mod.force)
20:         match Picking.pick ray view.Value with
21:             | [] -> ()
22:             | (d,f)::_ ->
23:                 for msg in f do
24:                     match msg (createEvent (ray.GetPointOnRay d)) with
25:                         | Some r -> model <- app.update model r
26:                         | _ -> ()
27:         updateScene model
28:
29:     ctrl.Mouse.Move.Values.Add(fun _-> handleMouseEvent MouseEvent.Move)
30:     ctrl.Mouse.Down.Values.Add(fun p -> handleMouseEvent ((curry MouseEvent.Down) p))
31:
32:     sceneGraph
```

This concludes our first implementation. Let us try to run our app:

```
1:  open Aardvark.Base.Rendering
2:  open Aardvark.Application.WinForms
3:  let app = new OpenGlApplication()
4:  let win = app.CreateSimpleRenderWindow()
5:
6:  let frustum = win.Sizes |> Mod.map (fun s ->
7:      Frustum.perspective 60.0 0.1 10.0 (float s.X / float s.Y)
8:     )
9:  let cameraView = CameraView.lookAt (V3d.III*3.0) V3d.OOO V3d.OOI
10: let camera =
11:     frustum |> Mod.map (fun f -> Camera.create cameraView f)
12:
```

```
13: let sg = createApp win camera drawingApp
14:
15: win.RenderTask <-
16:     win.Runtime.CompileRender(win.FramebufferSignature,
17:         sg
18:          |> Sg.effect [ DefaultSurfaces.trafo |> toEffect;
19:                         DefaultSurfaces.simpleLighting |> toEffect]
20:     )
21:
22: win.Run()
```

Comparison to Related Work and Analysis of the Approach

In interactive graphics applications, the common approach is to map the domain model into some domain specific representation. Whenever a user input and the associated interactions lead to changes in the model, those changes need to be applied to the application state and the graphics representation (e.g. the scene graph). This approach puts the burden of synchronizing application state and graphics state on the application programmer, as found by Tobler 2011.

Where does the synchronization code go? There are basically two approaches:

- Move all state including domain logic into the scene representation
- Store deep references into the graphics representation and modify them on change

In Tobler's work, a semantic scene graph serves as model, and a rendering scene graph (view), including rendering state, is generated on the fly while traversing the semantic graph. Since the semantic scene graph serves as domain model description language, all domain modelling needs do be done directly in the semantic scene graph.

However versatile, domain logic which is not easily implemented as graph traversal poses difficulties.

In particular, cross cutting concerns (e.g. game objects need to modify other game objects far away in the graph) result in a lot of graph searching for the appropriate nodes. When optimizing away this search effort, the scene representation approach gradually converges toward the reference-keeping approach. Graphics entities, or global state, increasingly keep references on modifiable cells in order to be able to update those immediately.

The main subject of our design pattern is to not use any graphics specific state at all. The graphics representation is entirely recomputed after each domain model modification instead. This completely eliminates the need to track state in the graphics representation, but the trivial implementation is very inefficent.

Can this model be implemented efficiently?

Towards an efficient implementation

Let us summarize how our Elm-style polygon-sketching application performs updates. First, we apply the view function to the initial model, producing a value of type `Scene<_>`. Next, we use a subscription on potential mouse events, which, when fired, traverses the scene and collects all potential `PickOperations`. Each pick operation potentially yields a message, which is fed into the update function, which recomputes the model.

Feeding back this model to our view function ties the knot. We receive our final interaction loop.

Let us investigate the involved functions and their runtime behavior:

- `view` is linear in the size of the model (independent of the real change).
- in the worst case, each `update` message touches every single entity in the model, resulting in linear runtime. In that case, the change really is of that size, i.e. a mutable update has `O(n)` as well. In practise however, operations often affect a single value deeply nested in the immutable structure, resulting in logarithmic runtime and memory cost. Accordingly, for updating immutable domain models we achieve `log(n)` blowup compared to a mutable implementation.

Update cannot be optimized in the purely functional case.

Looking at the view function, however, we identify that the function is repeatedly invoked with an almost identical input.

Incremental Evaluation helps they said

Incremental evaluation, as used in Aardvark, can be used to incrementalize purely functional algorithms. Conforming to the provided API, all inputs need to be wrapped into ModRef¡-¿ cells. Given a set of modifiable input cells, functions can be implemented by utilizing combinator functions, such as Mod.map or the `adaptive` Computation Expression. These functions become agnostic to input value changes:

```
1:  // Adaptively compute scene for input value of type cursor
2:  let cursorGeometry (hasCursor : IMod<Option<V3d>>) : IMod<list<Scene<DrawCommand>>> =
3:      adaptive {
4:              let! hasCursor = hasCursor
5:              match hasCursor with
6:                  | None -> return []
7:                  | Some c ->
8:                     return [[ Sphere(V3d.OOO,0.1) |> render Pick.ignore ]
9:                              |> colored C4b.Red
10:                             |> transform' (Trafo3d.Translation(c))]
11:     }
12:
13: let hasCursor = Mod.init None
14: let adaptiveScene = cursorGeometry hasCursor
15: // compute scene with current input state
16: let sceneWithDisabledCursor = adaptiveScene |> Mod.force
17: // change input state to have a cursor
18: transact (fun _ -> hasCursor.Value <- Some V3d.OO
19: // compute scene with current input state
20: let sceneWithCursor = adaptiveScene |> Mod.force
```

From a high level perspective, our incremental system has the following features:

- Inputs can be defined as modifiable cells
- Purely functional algorithms can be formulated on top of such modifiable cells
- The outpus of the algorithm can be computed by using Mod.force
- The inputs can be changed by using transact. Subsequent calls to Mod.force will return a value consistent with the current input values.

Thus, incremental evaluation does NOT provide us with a solution to our view function efficiency problem. We are working with purely functional input data. The entire domain model is new after a modification, meaning that our inputs alone will never be modified without everything else. This makes incremental evaluation infeasible.

As we discovered in practise, incremental evaluation can greatly speed up rendering performance. However, from an engineering point of view, programs quickly become large and interactions between various value changes complex (comparable to callback hell).

In order to utilize incremental evaluation, we need to map our purely functional model to a modifiable one. This allows us to perform updates as specific, targeted changes, which in turn can be used by the incremental evaluation system.

Mapping immutable values to mutable data sounds plausible. But where should this conversion occur?

While fablish and web frameworks use immutable data until the final rendering phase, we chose to map immutable values to mutable data prior to scene description generation. The rationale behind this is:

- The domain model is relatively small compared to a scene description. Working with a mutable scene description is more efficient.
- Typically, domain models are built on simple datatypes consisting of records and similar things. These might be easier to work with than general purpose scene descriptions.

Unpersisting data structures

Consider the transition of a record from immtuable to modifiable:

```
// Immutable data we would like to work on
type ImmutableData1 = { value : int }
// Modifiable data the rendering system would like to work on
type MutableData1 = { mvalue : IModRef<int> }
```

ImmutableData1 and MutableData1 are structurally equal, with all leaf fields wrapped into modifiable cells. //what? please rephrase.

How can we implement a procedure that applies immutable data changes into the mutable variant as update?

When immutable changes are detected, updates must be performed onto the according mutable fields. For records containing primitive values, the procedure boils down to traversing the data structure and applying new values to their mutable counterparts.

The problem arises with set data structures. In order to associate old entries in the mutable data with new values in the immutable set, some artificial references need to be introduced.

In this setup we propose to use integer values wrapped in a container class:

```
module IDs =
    open System.Threading // for Interlocked.Increment

    [<AllowNullLiteral>]
    type Id() =
        static let mutable current = 0
        let id = Interlocked.Increment(&current)
        static member New = Id()
        override x.ToString() = sprintf "Id %d" id
```

Let us now derive immutable and mutable models for our polygon-sketching app:

```
type OpenPolygon2 = {
    finishedPolys  : list<V3d>
    cursor         : Option<V3d>
}

type DrawingModel2 = {
    id : Id
    finished : pset<Polygon>
    working  : Option<OpenPolygon>
}

type MDrawingModel2 = {
    original : DrawingModel2
    finished : cset<Polygon>
    working  : ModRef<Option<OpenPolygon>>
}

let unpersist (m : DrawingModel2) : MDrawingModel2 =
    {
        original = m
        finished = CSet.ofSeq m.finished
        working = Mod.init m.working
    }

let apply (m : MDrawingModel2) (newModel : DrawingModel2) =
```

```
26:     transact (fun _ ->
27:         let added = newModel.finished |> Seq.filter (not << m.finished.Contains)
28:         let removed = m.finished |> Seq.filter  (not << newModel.finished.Contains)
29:         for add in added do m.finished.Add add |> ignore
30:         for rem in removed do m.finished.Remove rem |> ignore
31:         //...
32:         ()
33:     )
```

Such a transformation seems to be rather mechanical. It can be formalized into the following procedure:

- Equip each immutable domain type with an identifier (to be used for nested changes in sets)
- For each domain type, create an associated mutable version. It contains one extra field which points to the immutable structure, and:
  - for each field which is a domain type, use the mutable variant of the domain type in the mutable version of the parent structure
  - for each primitive field, create a field which wraps the original value in a Mod
  - for each field of type set or list, generate a data structure which tracks immutable references and allows to efficiently merge an immutable representation of the set/list into the mutable version

This translation sheme can be implemented as compiler plugin. For the sake of completeness we present the compilation result for our drawing model (We use the `DomainType` attribute as a hint for the plugin to handle this type as domain type):

```
1:  // original user written model
2:  module SimpleDrawingApp =
3:
4:      type Polygon = list<V3d>
5:
6:      type OpenPolygon = {
7:          cursor         : Option<V3d>
8:          finishedPoints : list<V3d>
9:      }
10:
11:     [<DomainType>]
12:     type Model = {
13:         finished : pset<Polygon>
14:         working  : Option<OpenPolygon>
15:     }
16:
17: // under-the-hoood compiled variant for the domain model
18: module SimpleDrawingAppGenerated =
19:     type Polygon = list<V3d>
20:
21:     type OpenPolygon =
22:         { cursor : Option<V3d>
23:           finishedPoints : list<V3d> }
24:
25:     [<DomainType>]
26:     type Model =
27:         { mutable _id : Id
28:           finished : pset<Polygon>
29:           working : Option<OpenPolygon> }
30:
```

```
31:         member x.ToMod(reuseCache : ReuseCache) =
32:             {   _original = x
33:                 mfinished = ResetSet(x.finished)
34:                 mworking = Mod.init (x.working) }
35:
36:         interface IUnique with
37:
38:             member x.Id
39:                 with get () = x._id
40:                 and set v = x._id <- v
41:
42:     and [<DomainType>] MModel =
43:         { mutable _original : Model
44:           mfinished : ResetSet<Polygon>
45:           mworking : ModRef<Option<OpenPolygon>> }
46:         member x.Apply(arg0 : Model, reuseCache : ReuseCache) =
47:             if not (System.Object.ReferenceEquals(arg0, x._original)) then
48:                 x._original <- arg0
49:                 x.mfinished.Update(arg0.finished)
50:                 x.mworking.Value <- arg0.working
```

The mutable model allows us to implement an incremental variant of the view function.

Our immutable scene representation, as-is, cannot express inner changes efficiently. As an example, consider the constructor for scene transformations: `val Transform : Trafo3d * seq<Scene1> -> Scene<'msg>` In order to efficiently handle trafo changes, the following signature is required: `val Transform : IMod<Trafo3d> * aset<Scene1> -> Scene<'msg>>` An incremental version of the scene data type can be derived by replacing all immutable values with their modifiable counterparts:

```
1:  type Scene2 =
2:          | Transform of IMod<Trafo3d> * aset<Scene2>
3:          | Colored   of IMod<C4b>     * aset<Scene1>
4:          | Render    of Primitive
5:          | Group     of aset<Scene1>
```

In our implementation, we chose a different representation.

Instead of using a discriminated union, we prefer to use the extensible (regarding data types) OOP approach found in Aardvark.Rendering. We can directly integrate with the Aardvark Scene Graph this way:

```
1:  module AdaptiveScene =
2:      open Aardvark.SceneGraph
3:
4:      type ISg<'msg> = inherit ISg // ISg comes from Aardvark.SceneGraph
5:
6:
7:      type Group<'msg>(xs : aset<ISg<'msg>>) =
8:          interface ISg<'msg>
9:          interface IGroup with
10:             member x.Children = xs |> ASet.map (fun a -> a :> ISg)
11:         member x.Children = xs
12:
13:     type Render<'msg>(xs : Primitive) =
14:         interface ISg<'msg>
15:         member x.Primitive = xs
16:
17:     //...
```

The Aardvark Scene Graph implementation is based on Attribute Grammars. We can extend the Aardvark Scene Graph by defining additional Ag-rules for our types dynamically. We augment our type `ISg<'msg>` with semantic functions such as:

```
module AgExtension =
    open Aardvark.Base.Ag
    open Aardvark.SceneGraph.Semantics
    open AdaptiveScene

    [<Semantic>]
    type LeafSemantics() =
        member x.RenderObjects(l : Render<'msg>) =
            match l.Primitive with
                | Sphere(center,radius) ->
                    Sg.sphere 5 (l?InhColor) (Mod.constant radius)
                    |> Sg.transform (Trafo3d.Translation center)
                    |> Semantic.renderObjects
                | _ -> failwith "..."
```

This translates our ISg¡'msg¿ into Aardvark ISg nodes whenever their RenderObjects semantic is requested. The complete implementation of the scene graph nodes can be found here.

An efficient version of the drawing application

```
module FinalDrawingApp =

    open Aardvark.ImmutableSceneGraph
    open Aardvark.Elmish
    open Primitives

    open SimpleDrawingAppGenerated


    type Action =
        | ClosePolygon
        | AddPoint   of V3d
        | MoveCursor of V3d

    let update e (m : Model) (cmd : Action) =
        match cmd with
            | ClosePolygon ->
                match m.working with
                    | None -> m
                    | Some p ->
                        { m with
                            working = None
                            finished = PSet.add p.finishedPoints m.finished
                        }
            | AddPoint p ->
                match m.working with
                    | None -> { m with working =
                        Some { finishedPoints = [ p ]; cursor = None;  }}
                    | Some v ->
                        { m with working =
                            Some { v with finishedPoints = p :: v.finishedPoints }}
```

```
32:             | MoveCursor p ->
33:                 match m.working with
34:                     | None -> { m with working =
35:                         Some { finishedPoints = []; cursor = Some p }}
36:                     | Some v -> { m with working =
37:                         Some { v with cursor = Some p }}
38:
39:
40:     let viewPolygon (p : list<V3d>) =
41:         [ for edge in Polygon3d(p |> List.toSeq).EdgeLines do
42:             let v = edge.P1 - edge.P0
43:             yield Primitives.cylinder edge.P0 v.Normalized v.Length 0.03
44:                     |> Scene.render Pick.ignore
45:         ] |> Scene.group
46:
47:
48:     let view (m : MModel) =
49:         let t =
50:            aset {
51:                 yield [ Quad (Quad3d [| V3d(-1,-1,0); V3d(1,-1,0); V3d(1,1,0); V3d(-1,1,0) |])
52:                             |>  Scene.render [
53:                                  on Mouse.move MoveCursor
54:                                  on (Mouse.down' MouseButtons.Left)  AddPoint
55:                                  on (Mouse.down' MouseButtons.Right) (constF ClosePolygon)
56:                                ]
57:                       ] |>  Scene.colored (Mod.constant C4b.Gray)
58:                 for p in m.mfinished :> aset<_> do yield viewPolygon p
59:                 let! working = m.mworking
60:                 match working with
61:                     | Some v when v.cursor.IsSome ->
62:                         yield
63:                             [ Sphere3d(V3d.OOO,0.1) |> Sphere |>  Scene.render Pick.ignore ]
64:                             |> Scene.colored (Mod.constant C4b.Red)
65:                             |> Scene.transform'
66:                                 (v.cursor.Value |> Trafo3d.Translation |> Mod.constant)
67:                         yield viewPolygon (v.cursor.Value :: v.finishedPoints)
68:                     | _ -> ()
69:             }
70:         Scene.agroup  t
71:
72:     let viewScene (sizes : IMod<V2i>) (m : MModel) =
73:         let cameraView = CameraView.lookAt (V3d.III * 3.0) V3d.OOO V3d.OOI |> Mod.constant
74:         let frustum = sizes |> Mod.map (fun (b : V2i) ->
75:             Frustum.perspective 60.0 0.1 10.0 (float b.X / float b.Y)
76:         )
77:         view m
78:             |> Scene.camera (Mod.map2 Camera.create cameraView frustum)
79:             |> Scene.effect [ toEffect DefaultSurfaces.trafo;
80:                               toEffect DefaultSurfaces.vertexColor;
81:                               toEffect DefaultSurfaces.simpleLighting]
82:
83:
84:     let initial = { finished = PSet.empty; working = None; _id = null }
85:
```

```
86:     let app s =
87:         {
88:             initial = initial
89:             update = update
90:             view = viewScene s
91:             ofPickMsg = fun _ _ -> []
92:             subscriptions = Aardvark.Elmish.Subscriptions.none
93:         }
```

Discussion

We ended up with an API for submitting changes to a domain model and evaluating a view function on that model. For efficiency reasons, we utilized adaptive functional programming in order to write one single view function which is agnostic to changes. Since the incremental system requires changes at inputs in order to perform necessary recomputations, we, at some point, need to translate changes in immutable data to reference cell mutations. This procedure can be automated and implemented as a compiler extension.

In other words: Previous work focused on feeding changes into a scene representation efficiently. Incremental evaluation is greatly beneficial (even indispensable) to those procedures. Although this is very efficient, keeping track of incremental computations and adaptive dependencies becomes difficult as complexity increases. On the other side, working with immutable updates on domain models is easy, but an efficient implementation is not tangible.

In this work we combine the two, and indeed we get the best of both. The immutable update mechanism can be seen as automatic, convenient input changer, while the incremental evaluation backend takes care of mapping updates efficiently to the underlying graphics hardware.

Refining the Model

External events

So far we only worked with mouse inputs and their interactions with 3D objects. In general, however, we often need to emit messages without user interaction, or due to the occurance of an external event. Elm extends its application pattern with an additional function. It allows subscriptions to be registered between the current model and sources of events.

```
1:  module Subscriptions1 =
2:
3:      type Sub<'msg> =
4:          | TimeSub  of TimeSpan * (TimeSpan -> 'msg)
5:          | KeyPress of (Keys -> Option<'msg>)
6:          | Many     of list<Sub<'msg>>
7:
8:      let timeSub ts f = TimeSub(ts,f)
9:
10: // todo subscription example
11:
12: module Animation =
13:
14:     open System
15:     open Aardvark.ImmutableSceneGraph
16:     open Aardvark.Elmish
17:     open Primitives
18:
19:     type Model = { rotation : float; _id : Id }
20:     type MModel = { mrotation : ModRef<float>} // mutable model impl skipped
21:
22:     type Msg = TimeStep of TimeSpan
```

```
23:
24:     let update e (m : Model) (msg : Msg) =
25:         match msg with
26:             | TimeStep t -> { m with rotation = m.rotation + t.TotalMilliseconds * 0.1 }
27:
28:     let view (m : MModel) =
29:          [ Sphere3d(V3d.OOO,0.1) |> Sphere |>  Scene.render Pick.ignore ]
30:            |> Scene.transform (m.mrotation |> Mod.map  Trafo3d.RotationZ)
31:
32:     let subscriptions (m : Model) : Subscriptions1.Sub<Msg> =
33:         Subscriptions1.timeSub (TimeSpan.FromMilliseconds 10.0) TimeStep
```

After each update, we purely functionally recompute the subscription set. Then we compute the difference between the current and the old subscription set. Unnecessary subscriptions can be removed safely, while new subscriptions are added where needed.

Subscriptions — The observer pattern's new clothes? At first, subscriptions look like a rather imperative pattern.

Traditional subscription-based techniques have severe flaws:

- Subscriptions need to be registered, and they need to be unregistered after expiration. Subscribe in Rx has type: 'val subscribe : ('a -¿ unit) -¿ IDisposable' which already hints at the significant effort required for tracking subscriptions in application code.
- Classic subscriptions compose by building up subscription networks, through which input events travel. Observing results from such networks can only be accomplished by registering (mutating) actions. Whenever an input changes, the change propagates through subscription chains. Eventually, actions are triggered to write new values into application state. Since change propagation in such networks is performed eagerly, the program can be assigned with faithful semantics. But understanding such networks statically, or even debugging their runtime behavior, can be extremely tricky.

In our design pattern, the set of subscriptions is recomputed automatically after each update. This means that the problem of "cleaning up" subscriptions completely disappears. Secondly, subscriptions in our model are purely functional. They build up lists of messages representing their effects, which in turn are fed back into the update function. This greatly improves debugging, since the results of external events can be observed at a single point in code instead of various points scattered in the domain logic.

Depth ordering of Pick Events

In our original implementation, we always used the frontmost pick and ignored all further hits along the pick ray. Howewer, it is often necessary to compute all pick occurences along the pick ray. The user might want to move polygon corners around by click-dragging. When hovering over control points of a polygon, the pick must go "through" the control point geometry and hit the ground plane below. This can be accomplished by a simple extension to the pick API:

```
1:  type Transparency = Solid | PickThrough
2:  type PickOperation2<'msg> = (V3d -> Option<'msg>) * Transparency
3:
4:  let rec depthTest (xs : list<float*PickOperation2<'msg>>) =
5:      match xs |> List.sortBy fst with // sort picks by distance
6:          | [] -> [] // no picks
7:          | (d1,(f1,Solid))::(d2,(f2,Solid))::rest when d1 = d2 ->
8:              // two solid picks, equal distance, use both
9:              (d1,(f1,Solid)) ::(d2,(f2,Solid)) :: depthTest rest
10:         | (d, (f,Solid))::_ ->
11:             // solid pick, stop here
```

```
            [d, (f,Solid)]
        | (d, (f,PickThrough))::xs ->
            // transparent object, take and pick test further
            (d, (f,PickThrough)) :: depthTest xs
```

Asynchronous computations

One main design guideline when designing the API for pick operations was to have necessary information available at one point in the code. Considering commands which trigger long-running or computationally intensive tasks, the best place for implementation is in the update function, where the message itself and the immutable domain model are available. However, performing long-running computations in the update function directly stalls the application. Therefore, to express such operations, we introduce the Cmd type and an additional argument to the update function.

```
module EnvExample =

    open System
    open Aardvark.ImmutableSceneGraph
    open Aardvark.Elmish
    open Primitives
    open Fablish.CommonTypes // env is defined here

    type Model = { image : list<PixImage>; }

    type Msg =
        | LoadImages of list<string>
        | ImagesLoaded of list<PixImage> // pixImage is a loaded image
        | Progress of float

    let loadImage (p : list<string>) (e : Env<Msg>) =
        async {
            let cnt = List.length p
            let images =
                p |> List.mapi (fun i p ->
                    async { return Progress (float i / float cnt) } |> Cmd |> e.run
                    PixImage.Create(p)
                )
            return ImagesLoaded images
        }
    let update (env : Env<Msg>) (m : Model) (msg : Msg) =
        match msg with
            | LoadImages paths ->
                loadImage paths env |> Cmd |> env.run
                m
            | ImagesLoaded img -> { m with image = img }
            | Progress _ -> m
```

Further examples

Implementing a transformation controller

```
module TranslateController =

    open Aardvark.ImmutableSceneGraph
    open Aardvark.ImmutableSceneGraph.Scene
    open Primitives
```

```
6:      open Aardvark.Elmish
7:
8:      open Scratch.DomainTypes
9:
10:     [<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
11:     module Axis =
12:         let dir = function | X -> V3d.XAxis | Y -> V3d.YAxis | Z -> V3d.ZAxis
13:         let moveAxis (trafo : Trafo3d) = function
14:             | X -> Plane3d(trafo.Forward.TransformDir V3d.OOI, trafo.Forward.TransformPos V3d.OOO)
15:             | Y -> Plane3d(trafo.Forward.TransformDir V3d.OOI, trafo.Forward.TransformPos V3d.OOO)
16:             | Z -> Plane3d(trafo.Forward.TransformDir V3d.OIO, trafo.Forward.TransformPos V3d.OOO)
17:
18:     type Action =
19:
20:         // hover overs
21:         | Hover           of Axis * V3d
22:         | NoHit
23:         | MoveRay         of Ray3d
24:
25:         // translations
26:         | Translate       of Axis * V3d
27:         | EndTranslation
28:
29:         | ResetTrafo
30:
31:     open TranslateController
32:
33:     let hasEnded a =
34:         match a with
35:             | EndTranslation -> true
36:             | _ -> false
37:
38:     let hover      = curry Hover
39:     let translate_ = curry Translate
40:
41:     let initalModel =
42:         { hovered = None; activeTranslation = None; trafo = Trafo3d.Identity; _id = null }
43:
44:     let initial =  {
45:             scene = initalModel
46:             camera =
47:                 Camera.create ( CameraView.lookAt (V3d.III*3.0) V3d.OOO V3d.OOI )
48:                               ( Frustum.perspective 60.0 0.1 10.0 1.0 )
49:             _id = null
50:         }
51:
52:
53:     let updateModel (m : TModel) (a : Action) =
54:         match a, m.activeTranslation with
55:             | NoHit, _             ->  { m with hovered = None; }
56:             | Hover (v,_), _       ->  { m with hovered = Some v}
57:             | Translate (dir,s), _ ->
58:                 { m with activeTranslation =
59:                     Some (Axis.moveAxis m.trafo dir, m.trafo.Backward.TransformPos s) }
```

```
60:             | EndTranslation, _    -> { m with activeTranslation = None;  }
61:             | MoveRay r, Some (t,start) ->
62:                 let mutable ha = RayHit3d.MaxRange
63:                 if r.HitsPlane(t,0.0,Double.MaxValue,&ha) then
64:                     let v = (ha.Point - start).XOO
65:                     { m with trafo = Trafo3d.Translation (ha.Point - start) }
66:                 else m
67:             | MoveRay r, None -> m
68:             | ResetTrafo, _ -> { m with trafo = Trafo3d.Identity }
69:
70:     let update e (m : Scene) (a : Action) =
71:         let scene = updateModel m.scene a
72:         { m with scene = scene }
73:
74:     let viewModel (m : MTModel) =
75:         let arrow dir = Cone(V3d.OOO,dir,0.3,0.1)
76:
77:         let ifHit (a : Axis) (selection : C4b) (defaultColor : C4b) =
78:             adaptive {
79:                 let! hovered = m.mhovered
80:                 match hovered with
81:                     | Some v when v = a -> return selection
82:                     | _ -> return defaultColor
83:             }
84:
85:         transform m.mtrafo [
86:                 translate 1.0 0.0 0.0 [
87:                     [ arrow V3d.IOO |> render [on Mouse.move (hover X); on Mouse.down (translate_ X)] ]
88:                         |> colored (ifHit X C4b.White C4b.DarkRed)
89:                 ]
90:                 translate 0.0 1.0 0.0 [
91:                     [ arrow V3d.OIO |> render [on Mouse.move (hover Y); on Mouse.down (translate_ Y)] ]
92:                         |> colored (ifHit Y C4b.White C4b.DarkBlue)
93:                 ]
94:                 translate 0.0 0.0 1.0 [
95:                     [ arrow V3d.OOI |> render [on Mouse.move (hover Z); on Mouse.down (translate_ Z)] ]
96:                         |> colored (ifHit Z C4b.White C4b.DarkGreen)
97:                 ]
98:
99:                 [ cylinder V3d.OOO V3d.IOO 1.0 0.05 |> render [
100:                     on Mouse.move (hover X); on Mouse.down (translate_ X)
101:                   ] ] |> colored (ifHit X C4b.White C4b.DarkRed)
102:                 [ cylinder V3d.OOO V3d.OIO 1.0 0.05 |> render [
103:                     on Mouse.move (hover Y); on Mouse.down (translate_ Y)
104:                   ] ] |> colored (ifHit Y C4b.White C4b.DarkBlue)
105:                 [ cylinder V3d.OOO V3d.OOI 1.0 0.05 |> render [
106:                     on Mouse.move (hover Z); on Mouse.down (translate_ Z)
107:                   ] ] |> colored (ifHit Z C4b.White C4b.DarkGreen)
108:
109:                 translate 0.0 0.0 0.0 [
110:                     [ Sphere3d(V3d.OOO,0.1) |> Sphere |> render Pick.ignore ]
111:                      |> colored (Mod.constant C4b.Gray)
112:                 ]
113:
```

```
                Everything |> render [whenever Mouse.move MoveRay]
        ]

    let viewScene (sizes : IMod<V2i>) s =
        let cameraView = CameraView.lookAt (V3d.III * 3.0) V3d.OOO V3d.OOI |> Mod.constant
        let frustum = sizes |> Mod.map (fun (b : V2i) ->
            Frustum.perspective 60.0 0.1 10.0 (float b.X / float b.Y)
        )
        viewModel s.mscene
            |> Scene.camera (Mod.map2 Camera.create cameraView frustum)
            |> effect [toEffect DefaultSurfaces.trafo; toEffect DefaultSurfaces.vertexColor;
                       toEffect DefaultSurfaces.simpleLighting]

    let ofPickMsgModel (model : TModel) (pick : GlobalPick) =
        match pick.mouseEvent with
            | MouseEvent.Click _ | MouseEvent.Down _  -> []
            | MouseEvent.Move when Option.isNone model.activeTranslation -> [NoHit]
            | MouseEvent.Move ->  []
            | MouseEvent.Up _   -> [EndTranslation]
            | MouseEvent.NoEvent -> []

    let ofPickMsg (model : Scene) noPick =
        ofPickMsgModel model.scene noPick

    let app (sizes : IMod<V2i>) = {
        initial = initial
        update = update
        view = viewScene sizes
        ofPickMsg = ofPickMsg
        subscriptions = Subscriptions.none
    }
```

Composing translation controller and polygon drawing

Related Work

Conclusions

In this work we used the ELM architecture in the context of 3D graphics. We developed a DSL for specifying 3D scenes and user interaction with graphical elements in a purely functional manner. When rendering UI elements, the resulting scene description can be efficiently rendered using HTML virtual dom diffing. This approach is not feasable for data intensive 3D applications. In this paper we show how to compute changes of immutable data structures in order to use those changes in the incremental rendering engine provided by the aardvark platform.