Compiling X3D for increased performance and safety
Compiling X3D for increased performance and safety

Giuseppe Maggiore  Fabio Pittarello  Michele Bugliesi
Mohamed Abbadi
Università Ca’ Foscari Venezia
Dipartimento di Scienze Ambientali, Informatica e Statistica
{maggiore,pitt,michele,mabbadi}@dais.unive.it

ABSTRACT
X3D is a powerful and expressive format for defining interactive 3D scenes. Interactive 3D applications, though, require more than just the scene definition; they require extremely complex interactions which are best modeled through scripts and they require a high frame-rate in order to give the user the illusion of fluid interactivity. In this paper we study two techniques that cover these two aspects in detail: rather than the slower, dynamic approach that X3D browsers use, we propose a faster, compiled model. By compiling X3D scenes and their associated scripts our system becomes both faster and safer to use; speed comes from the fact that the compiled scene is very optimized and has no overhead coming from all the dynamic interpretation of the scene and its script, while safety is increased because the compilation pass validates routes, DEFs and even the source of the script to ensure that it only accesses (correctly) valid nodes of the scene.

Finally, we show how thanks to the use of XNA as a target library we can run our compiled X3D scenes at high speed in Windows Phone 7 mobile phones and Xbox 360 consoles in addition to Windows PCs.

Categories and Subject Descriptors
D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.2.2 [Software Engineering]: Software Libraries—Design Tools and Techniques; D.2.13 [Software Engineering]: Reusable Software—Domain engineering, Reusable libraries, Reuse models; D.3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Programming Languages]: Processors—Optimization, Runtime environments; H.5.1 [Information Systems]: Information Interfaces and Presentation—Multimedia Information Systems

General Terms
Performance,Reliability,Languages

Keywords
x3d, monads, compilation

1. INTRODUCTION
X3D is a powerful and expressive format for defining interactive 3D scenes. When trying to build a complex and articulated application such as a game with X3D, though, some shortcomings in the technology surrounding the standard start to appear. Games in particular can be used as a challenging benchmark for a standard such as X3D because they require:

- very high performance
- safety and modularity
- high programmability

We believe that adapting X3D to games can be an important stress test, and such exercise could greatly help all the other domains where X3D is used. Let us consider the above checklist. Performance is dramatically required in a game to maintain the user’s feeling of immersion. Safety and modularity are required in any large application, where dynamic errors such as wrong DEFs in routes are bound to happen and can become a pointless waste of development time. High programmability is absolutely necessary to encode complex algorithms such as physics, AI, game logic, etc. All these aspects are clearly useful outside of game development too, since more immersive, safe, modular and programmable applications would be useful in most interactive domains.

In this paper we will tackle these three challenges by compiling X3D scenes rather than interpret them. Most X3D browser treat an X3D file as data which is dynamically turned into a tree of instances of node classes; these node classes know each other through virtual methods and dynamic dispatching. This causes a lot of unnecessary overhead, especially for all the structure of the scene which is static (that is once it is defined and loaded it will not be removed at runtime, only its parameters will change) for which all the dynamic lookups are redundant. The most problematic aspect of this kind of architecture is that scripting requires lots of lookups by name and upcasts, like in:

```x3d
X3DNode myClock =`
the above code is difficult to read and heavy to execute: readability is hampered by all the lookups, while we just want to perform a very simple lookup, which statically would be expressed simply as `mainScene.myClock.time`; slow execution comes from the need for the runtime to maintain (and dynamically search) tables of names and nodes or fields. Furthermore, this code is also very error prone; in this lookup operation there are as many as three possible sources of errors:

- "myClock" could not exist in `mainScene`
- "time" could not be a field of `myClock`
- `time` may not be cast to `SFTime`

We will solve these problems by creating a compiler that takes as input an X3D scene and returns as output some code that when run will execute the original scene statically and with no runtime overhead. We will also show how scripts can be integrated statically by validating the static access to nodes done in the script against the actual nodes defined in the scene; scripts are then statically linked so that rather than lookup nodes by name as seen in the example above they directly access the variables that represent the various nodes.

To translate an X3D scene we should (ideally) perform the following three tasks:

- convert the static nodes and routes
- convert the internal and external scripts
- convert the SAI mechanism that allows scripts access to the scene nodes and routes

Our system will not perform the second step since building a translator from Javascript or Java into F# would fall beyond the scope of our work; rather, we assume that the same operations we perform on F# scripts could be adapted to Javascript and Java scripts. Our system will process the static nodes and routes in order to optimize them aggressively and taking advantage of their staticity; we will also offer a direct, compile-time version of the SAI that allows scripts access to the static nodes and routes. It is important to notice that our scripting system is general enough that it could host an entire X3D browser; while this may seem like an overkill at first, the purpose is clear: if we have a large scene with lots of static and also lots of dynamic aspects, then the job of representing the dynamic aspects would be left to the scripting system while our compiler takes care of the static aspects. As for translating other languages into F#, we will not discuss the details of such an operation since it would add very little to the current knowledge of "regular" X3D browsers such as Octaga or BSContact.

In Sec 2 we will discuss related approaches to X3D. In Sec 3 we will present a schematic workflow that describes our solution from a broader perspective. In Sec 4 we will show some snippets of code of the actual compiler of our scene. In Sec 5 we will briefly discuss a sample case study in order to better understand how our compiler generates code from X3D files. At this point we present Sec 6 where we show some benchmarks that document how our approach yields better performance in scenes with many underlying routes and a "heavy logic". In Sec 7 we will discuss how we represent and execute scripts. In Sec 8 we will see how we integrate scripts inside the compiled scene. Finally, in Sec 9 we will present the kind of extensions we expect to build on our system and our conclusions.

1.1 Implementation

In this paper we will base our implementation on a functional language, F#, and the XNA gaming library. This is mostly because functional languages are very indicated to manipulate meta-programs such as compilers and because we need a functional language capable of accessing a powerful library to make games. Also, functional languages are very good at expressive reactive programming such as that found in the routing facilities supported by X3D. The combination of F# and XNA strikes a good balance between these two needs. It is very important to notice that thanks to the use of XNA it is possible to run X3D scenes on additional platforms such as the Xbox 360 and Windows Phone 7 mobile phones. The use of compilation rather than building a browser is also extremely indicated for running X3D scenes on mobile devices, given the limited computational power available on them.

2. RELATED WORK

X3D is a powerful format, but to maintain its importance and relevance it is clear that it must strengthen its support for always more advanced applications, both visually and logically.

X3D and Gaming Libraries. In [17] we see how with the use of OpenGL ES the authors have implemented an X3D browser that is capable of running in a mobile device. While we do not have the same degree of cross-platform compatibility afforded by OpenGL (XNA is limited to Microsoft platforms), it is clear that there is some benefit in opening X3D browsers from exclusively Web-centric technologies to game-development tools. In [7] we see how vertex and pixel shader support (with the HLSL, Cg or GLSL shading languages) is added to X3D scenes with the X3DShaderNode node; this approach helps achieve scenes that are both better-looking and faster to run. Furthermore, being able to leverage the large libraries of shaders currently available gives great opportunities for designers looking to build beautiful worlds.

Scripting in X3D. Many authors have built support for external languages in a manner very similar to ours. Usually this is done to compensate for some shortcomings of the original definition of X3D or to make complex aspects of a scene (like humanoid behaviors) easier to express. Most authors define new nodes where an external script is inserted.
in the form of a text field, thereby adding support for a new language. While we do not describe such a mechanism for our scripting system, where we believe that scripts would be handled separately from the scene (given that usually scripts are written by programmers while X3D scenes are authored by designers), it would not be difficult to define new nodes to host our own flavor of scripts.

In [5] the authors have added support for behavior definitions through Behavior3DNode nodes which are used to guide the generation of Java code for external interfacing; similarly, a logic-based scripting language is introduced in [10] for defining the behaviors of embodied agents such as virtual instructors or assistants; [12] describe the introduction of BML (Behavior Markup Language) for describing humanoid behaviors. Finally, in [13] and [23] we can find examples of a scripting language for defining shapes and volumes procedurally.

3. SOLUTION WORKFLOW

In Fig. 3 we can see a diagram depicting the steps used by our system when processing an X3D scene (plus its accompanying scripts). Notice that in Fig. 3 the red steps represent data while the blue steps represent operations. We start with an X3D file which describes our scene. This file may contain some scripts in its script nodes or the scripts may be stored into an external file. There are two layers of transformations described by our system, but only the second has been actually implemented:

- a transformation from the original scripts into our monadic F# scripts;
- a transformation from the original X3D file into an executable that runs that scene at very high performance.

We assume that we already have the F# scripts, that is the translation of scripts has been done by hand or the scripts have been written directly in F# itself. The equivalent of SAI, that is the original interface that allows scripts to access the nodes of the scene, is not translated; this happens because our scripts are inlined into the original scene (provided that they are correct and compatible, that is they do not access nodes that do not exist or that have a different type) and so they access the nodes directly, and not through an interface.

The first step our system performs is the translation of the initial X3D file into the F# code required for running the scene. This compilation step is quite complex and will be described in detail in Sec 4. At this point we have built a small linker which takes as input the source code of an F# script which refers the nodes and entities of the original scene; our linker embeds into the original source code the additional source code that implements the script. The result is then compiled by the F# compiler, thereby generating an executable (or a library for use in a larger project) which when launched will draw and animate the original scene plus its script.

The compiler that turns the initial X3D scene into F# code has been built in order to ensure that the most static aspects of the scene, that is those nodes that are not deleted or added dynamically, receive the fastest possible processing at runtime. For example, with a scene with a lot of trees or buildings we can safely assume that those entities will never be removed and only their properties will change. Routes that use a tree or a building do not need any kind of dynamic checking at runtime. Our compiler performs a series of steps by traversing the scene many times. Each step build some of the final F# source:

- build a set of nodes and their corresponding DEFs (their names)
- turn the nodes into local let-bindings, so that access by the update or draw code is extremely fast (just a variable lookup)
- create the load and draw methods that take care of loading nodes that use textures or meshes that are stored on disk and to draw all the nodes that require drawing
- create the update method with the code for handling input and timers
- create a map with the transitive closure of the routes from each node; if we have a route between nodes A and B, and another between nodes B and C, our map will contain [A → B → C][B → C]
- for each variable that is written in the update method, if the variable is in the routes map then add the corresponding routes code after writing the variable

This way all aspects of the scene are inserted very tightly into the same source, and whenever two nodes need to communicate with a route then the appropriate code is executed without intermediaries or dynamic lookups; rather, the code simply accesses the appropriate variables and directly executes the route code.

4. COMPILING THE SCENE

We will now discuss how we compile the scene into equivalent F# code. It is important to realize that the following section contains a very simplified version of the original F# code, to the point that it is safer to consider the following a very detailed pseudo-code exposition rather than the actual implementation; we have purposefully removed some low-level details that surround quotation processing which we believe would have seriously hindered readability, only leaving the most important aspects of the inserted snippets of code. We will perform a traversal of the X3D tree and for each node we will define a variable for it and we will add the needed code to correctly load data and draw said node. Code is represented by quotations; quotations are data structures that represent a syntax tree. Quotations can either be created by encapsulating code with special brackets, as in:

```fsharp
<@@ fun x -> x + 1 @@>
```

or by manually instancing the appropriate constructors from a large discriminated union that contains a constructor for
each possible syntactic construct of the F# language; the
same quotation seen above can be written as:

```fsharp
let x = Var("x",typeof<int>)
Expr.Lambda(x,
    Expr.Call(sum, [Expr.Var(x);
        Expr.Value(1)]))
```

where `sum` is the appropriate method info of the integer sum.
We will use the first style whenever possible, but note that
some articulated constructs are hard to express in this style
and so for such constructs we will fallback to the second style
which is less readable but more expressive. Quotations have
type `Expr`. It is crucial to realize that quotations are valid-
dated so that invalid programs cannot be built with them.
This is fundamental to our approach because invalid routes
or scripts will not be successfully compiled.

We start by loading the X3D from some input path:

```fsharp
let objectTree path : X3D =
    use stream =
        new StreamReader(path)
    let xml =
        new XmlTextReader(stream)
in XamlServices.Load(xml) :?> X3D
```

### Initialization and drawing.
At this point we can explore
the input tree; we take as input a list of objects (the nodes
of the X3D scene dynamically typed) and we return a quo-
tation representing the constructed code. The first traversal
of the input tree will build:

- the initialization of the scene (the actual output of the
  function)
- the declarations for all the variable that correspond to
  our inner (non-top-level) nodes
- the set of load operations for all those nodes that re-
  quire loading data from the disk (models and textures, for example)
- the set of draw operations for all those nodes that re-
  quire drawing
- a map of DEFs and their associated types
- a map of routes that associates to each node the cor-
  responding operations that must be performed when
  that node is updated

The above results are stored in the variables listed below (in
the same order):

```fsharp
let mutable load_content_expr =
    Expr.Value(())
let mutable draw_expr =
    Expr.Value(())
let mutable defs : Map<string,Type> =
    Map.empty
let mutable route_map : Map<string,Expr> =
    Map.empty
```

The recursive traversal will try and cast each node to the
type of a concrete node implementation; if the cast succeeds,
then we will process that node and we will recur on its child
nodes:

```fsharp
let rec build_scene_expr (xml: obj list) =
    if xml = [] then Quotations.Expr.list [()]
    else
        match (List.head xml) with
        | :? X3D_CSharp.Class.X3D as x ->
            (* process root node and its
            children *)
        | :? X3D_CSharp.Class.Scene as x ->
            (* process scene node and its
            children *)
        | :? X3D_CSharp.Class.Shape as x ->
            (* process shape node and its
            children *)
        | :? X3D_CSharp.Class.TimeSensor as x ->
            (* process time sensor node and its
            children *)
        | ... (* process remaining nodes *)
Let us consider a few detailed examples of how we would process certain nodes; let us start with a top-level time sensor. To process a time sensor first we add its definition to the `defs` dictionary, then we create a `Var` that represents the time sensor in our quotation, then we parse its parameters and finally we return the declaration and initialization of our time sensor. Notice that the remaining code after the declaration (the second parameter of the `Let` constructor) is constructed by recursively invoking the `build_scene_expr` function:

```csharp
| :? X3D_CSharp_Class.TimeSensor as x ->
    defs <- defs.Add(x.DEF, typeof(TimeSensor))
    let _myTimeSensor = Var(x.DEF, typeof(TimeSensor))
    let rest = List.tail xml
    let loop = bool.Parse(x.loop)
    let cycle_int = Parse(x.cycleInterval)
    Let(_myTimeSensor,
        <@@ new TimeSensor(loop,
            cycle_int) @@>,
        build_scene_expr rest)
```

Let us now consider a more complicated example. We want to build the code corresponding to a nested box node. The complexity comes from the fact that we use a box model which must be loaded from disk (so we have to add some loading code); also, the box must be rendered (so we have to add some drawing code). Finally, the box is nested inside some other nodes and so we need to add its declaration to the top-level declarations:

```csharp
| :? X3D_CSharp_Class.Box as x ->
    let def = x.DEF
    let box_var = Var(def, typeof<Box>)
    let box_expr = <@new Box(def, Vector3.One)@>
    let_expr <- Let(box_var, box_expr, let_expr)
    let model_name_expr = Value("Box")
    let game_var = Var("game", typeof<Game>)
    let game_tp = typeof<Game>
    let content = PropertyGet(game_var, game_tp.GetProperty("Content"), [])
    let content_expr = <@@ (%%content:>(ContentManager)).Load<Model>("Box") @@>
    let loaded_model = Call(box_var, box_tp.GetMethod("Draw"), [dt_expr])
    load_content_expr <- Sequential(loaded_model, load_content_expr)
    let dt_var = Var("dt", typeof<float32>)
    let draw = Value()```

The most important aspects of the above code are that we are building the list of top-level declarations (because this particular box declaration is nested):

```csharp
let_expr <- Let(box_var, box_expr, let_expr)
```

and that we are adding loading and drawing code:

```csharp
load_content_expr <- Sequential(loaded_model, load_content_expr)
...```

Building routes. After the first traversal of the X3D tree, we need to perform another traversal. In this traversal, whenever we encounter a route, we map to the source node the appropriate code that performs the update of the dest node and the appropriate field whenever the source node is modified; the map looks like the following:

```csharp
myClock ⇒ myColor.fraction = −myClock.fraction
myColor ⇒ myMaterial.diffuseColor = −myColor.value
...```

the processing of routes is:

```csharp
| :? X3D_CSharp_Class.ROUTE as x ->
    let from_tp = dictionary.Item x.fromNode
    let from_pi = from_tp.GetProperty(x.fromField)
    let from_var = Var(x.fromNode, from_tp, false)
    let from_get = PropertyGet(from_var, from_pi)
    let to_tp = defs.Item x.toNode
    let to_pi = to_tp.GetProperty(x.toField)
    let to_var = Var(x.toNode, to_tp, true)
    let to_set = PropertySet(to_var, to_pi, from_get)
    if route_map.ContainsKey x.fromNode then
        let tmp = route_map.Item x.fromNode
        route_map <- route_map.Add(x.fromNode, Sequential(tmp, to_set))
    else
        route_map <- route_map.Add(x.fromNode, to_set)
    let ambient = List.tail xml
```
note that whenever a node already has some routes, then those routes are preserved and the new route is added to the map with a Sequential construct which will represent the execution of the new route right after the previous routes.

**Updating.** Similarly to the previous traversals we further traverse the tree to identify all the update operations needed by each node. After this traversal, we perform one last traversal that adds the various routes in the appropriate places of the update function. This is expressed very tersely by simply adding the route code after each PropertySet statement performed on a node that is the source of route; this is intuitive since every time we modify a node then we must add the appropriate routing operations right after the modification operation. Since the added code may contain set operations that spawn further routes, then we recursively process the added code. We temporarily remove the source node from the routes when recursive to avoid looping infinitely in the case of cyclic routes:

```ocaml
let rec add_routes_to_scene expr (rs:Map<string,Quotations.Expr>) =
  match expr with
  | PropertySet (Some (e),_,_,_,_) as str ->
    if rs.ContainsKey str then
      let e' = add_routes_to_scene (rs.Item str) (rs.Remove str)
      Sequential(expr,e')
    else
      expr
  | ... (* recursively traverse the tree *) in
```

As the last step, we put together all the generated code into a function that takes as input a Game object (the handler of the main loop) and returns a record containing the various nodes as its closure.

5. **A CASE STUDY**

We will now present our simple case study. We will consider an X3D scene that contains a looping timer which updates a color which in turn updates the material used when drawing a box:

```xml
<Scene>
  <ColorInterpolator DEF='myColor' value='1,0,0,0,1,0,0,1,0,0,1,0,0'>
    keyValue='1.00f,0.00f,0.00f,0.00f,0.00f,0.00f,0.00f,0.00f'
    key='0.00f,0.33f,0.66f,1.00f'/>
  </ColorInterpolator>
  <TimeSensor DEF='myClock' cycleInterval='10.0' loop='true'/>
  <Shape>
    <Box/>
    <Appearance>
      <Material DEF='myMaterial'/>
    </Appearance>
  </Shape>
  <ROUTE fromNode='myClock' fromField='fraction_changed'
         toNode='myColor' toField='set_fraction'/>
  <ROUTE fromNode='myColor' fromField='value_changed'
         toNode='myMaterial' toField='diffuseColor'/>
</Scene>
```

We do not want to interpret this scene dynamically into a browser. Rather, we want to compile this scene into a specialized browser which has the above scene hardcoded inside its source.

The source code that implements the above scene is the following:

```ocaml
let rec myScene () =
  let rec Appearance1 =
    new Appearance(myMaterial)
  and myMaterial =
    new Material("myMaterial",
      new Vector3(0.80f,0.80f,0.80f))
  and Box1 =
    new Box("Box1",myMaterial,vector3.0)
  and rec Shape1 =
    new Shape(Box1,Appearance1)
  and myColor =
    new ColorInterpolator("myColor",
      [ new Vector3(1.00f,0.00f,0.00f);
        new Vector3(0.00f,1.00f,0.00f);
        new Vector3(0.00f,0.00f,1.00f);
        new Vector3(1.00f,0.00f,0.00f)];
      [0.00f;0.33f;0.67f;1.00f])
  and myClock =
    new TimeSensor(true,10.00f)
  and load_content =
    Box1.set_Model( game.Content.Load("Box"))
  and update dt =
    myClock.time <- (myClock.time + dt)
    myColor.fraction <- myClock.fraction
    myMaterial.diffuseColor <- myColor.value
    if (myClock.time > 10.00f) then
      myClock.time <- 0.00f
      myColor.fraction <- myClock.fraction
      myMaterial.diffuseColor <- myColor.value
  and draw dt =
    Box1.Draw(dt)
  in {
    update = update;
    draw = draw;
    load_content = load_content; }
```

Notice that we need to create all the nodes, unfolding them so that even if there are nested DEFs (such as the one for myMaterial) they appear as top level identifiers accessible
to the rest of the program. This way we have completed our mapping from named nodes to local \texttt{let}-bindings, so that accessing the nodes of our scene will correspond to the common, simple and fast variable lookups.

We also need to define three functions: an initialization function, which loads any required data from disk; an update function, which performs the game logic and executes the routes; a draw function, which draws the scene to the screen.

These three functions are executed in a loop by the XNA framework as depicted in Fig 1.

Notice that routes in the update function are represented by the actual chains of field updates that we need to happen; there is no overhead at all when dynamically propagating the update events. Also, if a field does not start a route then there are no “hidden” costs as we would have when firing a \texttt{FieldModified} event with no listeners.

6. BENCHMARKS

Our system is mainly concerned with optimizing away the overhead that dynamically building and maintaining an X3D scene produces. To show that we have achieved our objective, we have tested the same scene on multiple browsers and profiled the resulting framerates. The browsers we have used are BS Contact and Octaga.

We have tested for scenes with a relatively low number of shapes (300 and 680). We are not really interested in testing the rendering performance, since such a test would mainly compare the efficiency of the underlying rendering APIs and would not be relevant in this context. Both scenes are compared against two other scenes with the same shapes but with 3 \texttt{color} interpolators, 2 \texttt{timers} and 6 \texttt{routes} for each shape. The resulting routing and logic are quite heavy and constitutes a good test the underlying execution model for routes and logical nodes.

The tables below show a comparison in performance for each browser with various hardware configurations:

It is clear that thanks to our approach the scene logic weighs far less than it does in the other browsers.

Moreover, as we can see in Fig 3, the code that is generated by our system can be run, without modification also in Windows Phone 7 devices; in the figure we can see the emulator in action. The results of running two compiled scenes with 150 and 300 shapes respectively plus the usual routes for each shape are summarized in the table below:

At this point we have completed supporting the static aspects of an X3D scene, those that are involved in nodes that are not added or removed dynamically. This approach clearly yields an increase in performance for scenes with a complex logic in terms of timers, routes, interpolators, etc. We now move to the second part of our work, which focuses on integrating our compiled scenes with scripts that can implement nodes that are added or removed dynamically and any other aspect of the scene that would be hard to express in plain X3D.

7. REPRESENTING SCRIPTS

In this section we will define a monad that will act as the runtime of our scripting system. Having a safe, yet powerful language for defining scripts gives us the capability of adding complex logics (AI, physics, etc.) to our scenes, thereby making those scenes far more immersive and entertaining.

The main problem with a scripting system is that we want it to run asynchronously with respect to the main loop implementing the engine. Let us consider an operation in a game which we might want to update with a busy loop:

\begin{verbatim}
wait_destroyed_asteroids_greater 10
\end{verbatim}

If this operation was defined as a recursive function which loops until a certain condition is met (namely that 10 asteroids are destroyed) then running this function would require its own thread separate from the main loop. If this oper-
The monad we define performs a very simple trick. At every bind, it will suspend itself and return its continuation as a lambda. The monad type is:

```plaintext
type Script<'a,'s> = 's -> Step<'a,'s>
and Step<'a,'s> = Done of 'a
| Next of Script<'a,'s>
```

Notice that the signature that is very similar to that of the regular state monad, but rather than returning a result of type 'a, it returns either Done of 'a or the continuation Next of Script<'a,'s>. The current state of the computation, comprised of all the intermediate values that are in scope at the given point, is stored as the closure of the lambda expression of an script value.

Returning a result in this monad is simple: we just wrap it in the Done constructor since obtaining this value requires no actual computation steps. Binding together two statements is a bit trickier. We try executing the first statement; if the result is Done x for some x, then we return Next(k x), that is we perform the binding and we will continue with the rest of the program with the result of the first statement plugged in. If the result is Next p', then we cannot invoke k because we have no value to pass it since we need p to complete to have a value of type 'a. This means that we have to bind p' to k, so that at the next execution step we will continue the execution of p from where it stopped.

We also define a conversion operator that takes a statement of the regular state monad and turns it into a statement of the script monad. This scenario is useful because it allows us to use the script monad in a game where the state is managed with the state monad or one of its variations:

```plaintext
let (!) (p:St<'a,'s>) : Script<'a,'s> =
    fun s -> Done(p s)
```

We define a game script as an instance of the Script datatype where the state (the 's type variable) is instantiated to the script state ScriptState. The main loop will now access the current state of the scripting system to step the current script forward of the game script:

```plaintext
let script_state = ...

let rec update_script (script_step:Script<Unit,GameState>) (script_state:GameState) =
    let script_step' =
        match script_step script_state with
        | Done() -> fun _ -> Done()
        | Next k -> k
    in update script_step'
```

The update function executes a step of the script. If the script has finished, then we create an identity script that will be called indefinitely. When an iteration of the update loop is completed, then we call update with the next state of the script as its parameter.

At this point we can see a sample script that every 10 asteroids destroyed makes an asteroid appear at the center of the screen with a flash (a “warp” effect). This example makes use of predefined stateful operations that add or remove an asteroid to the game state (AddAsteroid, RemoveAsteroid) and which add or remove a warp flash (AddWarp, RemoveWarp) to the game state. The warp flashes created with the mk_warp function are animated, so that they do not suddenly appear but rather they appear extremely small before quickly growing in size; the created object (in our case the asteroid) appears behind a fully grown flash which then quickly shrinks until finally it disappears. The animation of a warp flash is performed by the recursive warp_animation function:
let warp_animation src dst max_dt (a: Entity) =
let rec warp t0 =
script{
  let! t = time
  let dt = t - t0
  if dt < max_dt then
    let size = interpolate src dst (dt / max_dt)
    do! !(SetSize w size)
    return! warp t0
}
in script{
  let! t0 = time
  do! warp t0
}

let rec warp_script n =
script{
  do! wait_destroyed_asteroids n
  let a = mk_random_asteroid
  let p = a.position
  let w = mk_warp p
  do! !(AddWarp w)
  do! warp_animation 0.0f WARP_SIZE WARP_IN_DURATION a
  do! !(RemoveWarp w)
  do! wait 5.0f
  let w = mk_warp a.position
  do! !(AddWarp w)
  do! warp_animation 0.0f WARP_SIZE WARP_IN_DURATION a
  do! !(RemoveWarp w)
  do! wait 10.0f
  do! set_render_text "text1"
      (Vector2(50.0f,50.0f))
      Color.White Vector2.One
  do! wait 2.0f
  do Box1.position.Y <- rand.Next()
  do! remove_render_text
  do! parallel_
    (unfold{
      do! warp_script (n+10)
    }) |
  do! wait 6.0f
  do! set_render_text "text2"
      (Vector2(100.0f,100.0f))
      Color.White Vector2.One
  do! wait 3.0f
  do Box1.position.Y <- rand.Next
  do! remove_render_text
  return! warp_script n
}

8. REPRESENTING THE SAI

An example script in our system (with reference to the scene seen in Sec. 5) could turn on and off some testing text to be printed as an overlay; this text is part of the of the script state (managed by the update_script function). In addition to turning on and off this text we also randomly move around the box Box1. Notice that we are taking advantage of the parallel_combinator which runs two sub-scripts in parallel:

let script (Box1:Box)=
<@
unfold{
  let rand = new Random()
  do! parallel_
    (unfold{
      do! wait 10.0f
      do! set_render_text "text1"
          (Vector2(50.0f,50.0f))
          Color.White Vector2.One
      do! wait 2.0f
      do Box1.position.Y <- rand.Next()
      do! remove_render_text
    })
    (unfold{
      do! wait 6.0f
      do! set_render_text "text2"
          (Vector2(100.0f,100.0f))
          Color.White Vector2.One
      do! wait 3.0f
      do Box1.position.Y <- rand.Next
      do! remove_render_text
    }) |
  ignore_
}@>

A script explicitly declares as its parameters the nodes it will use from the host scene. When the script is compiled into the scene its input parameters are validated against the actual nodes defined in the scene; if the validation succeeds then the parameters are removed from the scene and the script is inlined into the scene code; the result is:

let rec build_scene () =
...
(* declarations *)
let mutable script =
<@
...
(* script body which uses the declarations *)
@>
and load_content =
...
and update dt =
  script <- update_script script dt
and draw dt =
Box1.Draw(dt)
draw_script script
{
    update = update;
draw = draw;
load_content = load_content;
}

where the two functions update_script and draw_script respectively run our script one step for each update calls and draw its state to the screen.

9. CONCLUSIONS AND FUTURE WORK
In this paper we have presented a novel approach to running X3D scenes. Upon recognition that X3D requires the highest possible degree of performance and safety we have experimented with a move from the slower, dynamic interpretation that current X3D browsers do to a faster, static compiled model of execution which creates a “specialized browser” for every X3D scene.

Creating complex applications with X3D alone is not possible, and the included scripting solutions do not scale (in the experience of the authors) to domains such as video games without a lot of effort. For this reason we have studied a way to better integrate and validate scripts into X3D scenes; our solution of embedding monadic F# scripts into the compiled code allows a developer to add complex logic to a script in a seamless manner, since the syntax tree that contains our compiled scene can be further manipulated to include any scripts we want without the overhead of casting and dynamic dispatching. Thanks to quotations we are sure that the compiled result is valid (routes are correct, etc.) and scripts correctly access the scene nodes; any error will be detected at compile time, thus reducing the amount of testing needed by the developers.

Thanks to our system it has been possible to run our compiled X3D scenes with different platforms that support XNA. In particular we have tested our benchmark scenes on the Xbox 360 and Windows Phone 7. While the Xbox is very similar to a PC in terms of hardware, the ability of running X3D scenes on powerful mobile devices is extremely interesting since it unlocks new interaction opportunities; moreover, optimizations such as ours become crucial to make good use of the limited computing power of such devices.

Our work is by no means complete. We still need to implement some of the primitives of our target X3D profile (the Interactive profile). Also, we are planning to extensively test our system for games and interactive applications with a very complex logic. Also, we wish to experiment a further expansion of our compiler to support customized computations layers such as custom physics, custom AI, custom renderers that perform visibility culling or advanced shading such as ray-tracing; our aim is to make X3D more suitable for game development and richer, applications. Finally, we wish to carefully implement all these aspects of interactive applications so that fast and high-quality execution on mobile devices remains possible.

10. REFERENCES

[15] Erik Meijer. There is no impedance mismatch: (language integrated query in visual basic 9). In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, OOPSLA ’06, pages 710–711, New York, NY, USA, 2006. ACM.


