A compilation technique to increase X3D performance and safety

ABSTRACT

As virtual worlds grow more and more complex, virtual reality browsers and engines face growing challenges. These challenges are centered on performance on one hand (an interactive framerate is always required) and complexity on the other hand (the larger and more articulated a virtual world, the more immersive the experience).

The usual implementation of an engine or browser for running virtual worlds features an object-oriented architecture of classes. This architecture is a source of often underestimated overhead in terms of dynamic dispatching, and dynamic lookups by scripts when they try to access portions of the scene are both costly and possible sources of mistakes.

In this paper we discuss how we have tackled the problem of increasing performance in X3D browsers while also making scripts safe. We have used a compilation technique that removes some overhead and which allows us to introduce safety for scripts that access the state

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, performance, safety, compilation

1. INTRODUCTION

As virtual worlds grow more and more complex, virtual reality browsers and engines face growing challenges. These challenges are centered on performance on one hand and complexity on the other hand.

Performance is needed because the framerate at which the virtual world is rendered and animated must be high enough to give the user a feeling of smoothness. The scene must be rendered at least at 30 frames per seconds, but higher framerates (e.g. 60 frames per second) are perceived by the user as more pleasant.

Visual and logical complexity of a virtual world is very important. Visual richness gives the user the impression of a more realistic and detailed world, with many beautifully rendered objects, while logical complexity permits articulated responses that give the user the feeling of being part of a realistic world with its own set of rules and internal laws.

One of the most important tasks for developers of interactive worlds is to find the right trade-off between these opposite requirements. Increasing performance requires a mixture of compromise (reducing the size of the world or "dumbing down" its responses) and time-consuming low-level optimization. We believe that automated optimization of interactive applications is a fundamental frontier if we wish to enable developers to build richer worlds without exponentially increasing costs.

Modern 3D browsers and engines are based on a data-driven architecture as shown in Figure 1, taken from [13].

In a data-driven engine the engine contains only general knowledge about virtual worlds, but nothing specific about the peculiar features of a specific virtual world. The specific virtual world will be loaded from the game content in the form of configuration files and scripts. A data-driven engine loads from files two main datasets:

- a **scene**, the set of entities that populate the virtual world
- **scripts**, the set of (possibly complex) behaviors that animate the scene entities

The scene is composed by a heterogeneous set of entities, each of a different kind. Entities may be virtual characters,

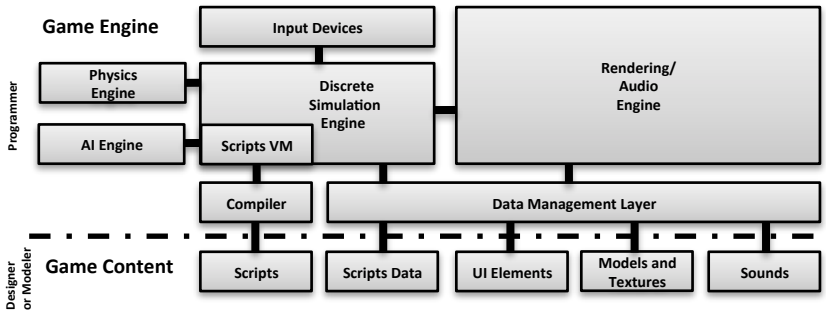

Figure 1: Data-driven engine architecture

trees, 2d or 3d models; entities may also be purely logical and invisible entities such as timers, triggers and proximity sensors.

Scripts give depth to a scene by implementing complex interrelationships between entities. The simplest form of scripting is known as *routing*, and it is a simple transmission of values from one entity to another. More complex scripts can perform data conversions when moving information between entities. The most advanced scripts can even create, remove or modify entities of a scene. Scripts can be seen as behaviors that animate and give life to our entities by making them "act" to simulate interesting or realistic behaviors in our scene.

The usual implementation of an engine (see [3]) features an object-oriented architecture of classes. At the root of this architecture is a class that represents the most generic entity, and from which all other entities are derived. The engine maintains a list of these generic entities, which are all updated and handled through a set of virtual functions. This architecture is a source of often underestimated overhead. Dynamic dispatching is not too costly for a few calls, but when we have many entities, the cost of invoking various virtual functions many times for each frame can become very high. Sometimes the cost of the dynamic dispatching architecture may become higher than the cost of the actual operations being dispatched.

Scripts usually access the scene dynamically. This means that a script must look for the right entities with a mixture of lookups by name and unsafe casts. For example, consider how a Java script may access the `time` field of a `myClock` node of type `timer`:

```
X3DNode myClock =
 mainScene.getNamedNode("myClock");
SFTime time =
  (SFTime) myClock.getField("time");
```

This style is unsafe, since `myClock` may not exist or it may have the wrong type, and it also incurs in significant overhead.

In this paper we will focus exclusively on the X3D language, since it is a recognized standard and it offers a good benchmark to test virtual worlds where we can specify our scene and its various scripts. In the paper we show how we have tackled the problem of increasing performance in X3D browsers while also making scripts safe. We have used a simple compilation technique that removes many unnecessary dynamically dispatched invocations; this technique also allows us to introduce safety for scripts that access the state, so that they do not need to perform unsafe dynamic lookups when searching for specific nodes.

In Section 2 we discuss the general architecture of our system. In Section 3 we show how our technique generates the code and thetype definitions that represent a scene. In Section 4 we discuss how we represent scripts that externally access the scene. In Section 5 we show an example of a compiled scene and its routes. In Section 6 we report some benchmarks that show speed increasing when rendering a sample scene by applying our technique.

1.1 Related Work

To the best of our knowledge, this is the first approach that experiments with compiling X3D scripts and scenes in order to achieve greater performance and safe scripts. None of previous approaches we are aware of focus on compilation of X3D as a means to achieve both higher performance (by reducing overhead) and safety (by introducing compile-time checks).

2. SOLUTION WORKFLOW

In Figure 2 we can see a diagram depicting the steps used by our system when processing an X3D scene (plus its accompanying scripts). In the figure red blocks represent data while the blue blocks represent computations. 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 external scripts into our F# scripts
- a transformation from the original entities and routes of the X3D file into the final program

Our system starts by translating the X3D scene and its routes into F# source code. This source code contains a

type definition that describes the entire scene and the routes, plus an update function that performs a step of the virtual world simulation (by activating routes and scripts). This first processing gives our system support for regular X3D nodes that describe shapes and the various scene entities, and also routing nodes that describe basic scripts.

External scripts are then validated against our type definition, to ensure that they correctly access the scene. If their validation succeeeds, the final program is produced that integrates both the scene and all the scripts. This second processing gives our system support for any other scripts which are not easily expressed with routes.

3. COMPILING THE SCENE

In this section we show an outline of our compilation technique.

The first step our compiler performs is deserializing the xml definition of our X3D scene. The scene is then processed and turned into a record, a type definition that describeds the static structure of our scene. The record contains:

- a field for each static node of the scene; each field has the name of the node if the node has a `DEF` attribute
- a field for a list of dynamic nodes
- a field for a list of active scripts

A sample state for a scene with a timer and a box could be:

```
type Scene  =
  {
    myClock       : Timer
    box           : Box
    dynamic_nodes : List<Node>
    script        : Script
  }
```

Where `Timer` and `Box` are the concrete classes for a timer and a box respectively, and they both inherit from the `Node` class. A list of nodes is needed to represent the dynamic portions of the scene, and a list of scripts maintains the sequence of currently running scripts.

This state definition is quite important, since it represents the interface between our scene and our scripts, and since it allows us fast lookups of specific nodes. Finding a node now just requires reading from a field in the state, an operation which is both fast and certain not to fail. For example, looking for the `time` field of the `"myClock"` node would simply require writing:

```
scene.myClock.time
```

We then proceed to the initialization of the state. This amounts to creating instances of each node, and then assigning these instances to the fields of the `scene` variable.

An `update` function is then constructed that performs the update of all the statically known fields of the state, and which also executes the various routes of the scene. Also, the `update` function invokes the (dynamically dispatched) `update` function of each dynamic node; this is necessary because it would be unrealistic to hope that a complex virtual world can exclusively rely on statically known nodes, and a balance must be struck between optimizing static nodes and supporting dynamic ones.

The `update` function also performs a tick for all currently running scripts.

The update function that updates the state seen above would simply become:

```
let update (dt:float32) =
  scene.myClock.update dt
  scene.box.update dt
  for node in scene.dynamic_nodes do
    node.update dt
  scene.script.update dt
```

4. EXTERNAL SCRIPTS

To represent external scripts, rather than using arbitrary objects that can access the state we have chosen to use coroutines, a widely used mechanism for representing computations in interactive applications [8, 2]. Coroutines are subroutines that can be suspended and resumed at certain locations. With coroutines the code for a SM is written "linearly" one statement after another, but each action may suspend itself (an operation often called "yield") many times before completing. A coroutine stores a temporary, internal state transparently inside its continuation.

We build a monadic framework [9, 10, 11, 6] for coroutines that allows us greater customization flexibility. This way we can define our own system for combining scripts running them in parallel, concurrently, etc. For a detailed discussion of this monadic framework for scripts and coroutines see [5].

A script in our system is defined as a normal F# program surrounded by `{ }` brackets. A script runs another script with the statements `let!` and `do!`, and scripts can be combined with a small set of operators.

The main operators to combine scripts are:

- `parallel` ($s_1 \wedge s_2$) executes two scripts in parallel and returns both results
- `concurrent` ($s_1 \vee s_2$) executes two scripts concurrently and returns the result of the first to terminate
- `guard` ($s_1 \Rightarrow s_2$) executes and returns the result of a script only when another script evaluates to `true`
- `repeat` ($\uparrow s$) keeps executing a script over and over

A sample script that moves the box `myBox` when the user enters a certain region `myRegion` could be the following:

```
let my_script (scene:Scene) =
  let rec animate =
```

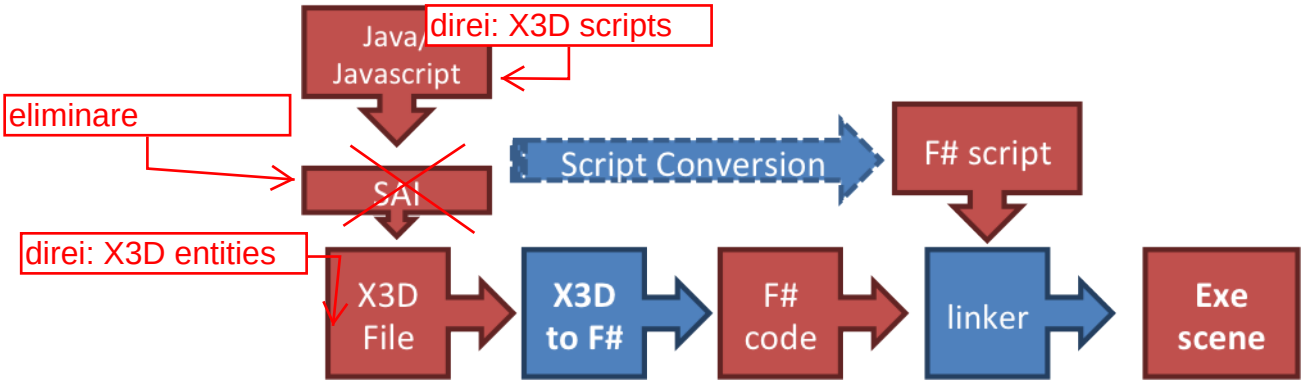

Figure 2: Solution workflow

```
    script {
      if scene.myBox.Position.Y < 100.0f
          then
        scene.myBox.Position.Y <- scene.
            myBox.Position.Y + 0.1f
        do! animate
    }
  script {
    do! guard
          script {
            return inside(scene.Camera.
                Position, myRegion)
          }
          animate
  }
```

Notice that our script has a parameter of type `Scene`. If this parameter is used incorrectly (for example the scene this script is applied to does not have a `Box` node with name `myBox`) we will get a compile-time error. This makes it easier to build larger, reusable script modules since a mistake in using a pre-made module is easier to spot and requires less testing. Using scripts which have been made for different scenes would require extensive testing to ensure at least that all node accesses are correct.

Our scripting system is expressive enough to represent many scripts running together, even if at a first glance it may appear that our system supports only a single script. By using the `parallel` operator we can combine together a large number of scripts. For example, let us say we have many scripts $s_1, ..., s_n$ that must all run together with our scene. Each scripts has a different duration, that is the not all scripts will end at the same time (indeed, a script may even run indefinitely). The main script would chain each of the various actual scripts in the following manner:

```
let my_script (scene:Scene) =
  parallel s1
    (parallel s2
      ...
        sn) ... )
```

5. A MORE DETAILED EXAMPLE

We will now present a more detailed example to see our compiler in action by showing how it handles all the features of an X3D scene: entities and routes. We will consider an X3D scene that contains a looping timer which updates a color that in turn updates the material used when drawing a box:

```
<Scene>
  <ColorInterpolator DEF='myColor'
    keyValue='1␣0␣0,␣0␣1␣0,␣0␣0␣1,␣1␣0␣0'
    key='0.0␣0.333␣0.666␣1.0'/>
  <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>
```

Our compiler produces the following state definition from the above scene:

```
type Scene =
  {
    myColor       : ColorInterpolator
    myClock       : TimeSensor
    myMaterial    : Material
    dynamic_nodes : List<Node>
    script        : Script
  }
```

where pointers to all statically known nodes are maintained.

The initialization function for our state initializes a set of local variables, one for each named node, and then builds the actual scene state. Notice that at this point routes are ignored, since they will be used only for the update function:

```
let scene =
  let myColor =
       ColorInterpolator(
         keyValue = [ ... ],
         key = [ ... ])
  let myClock =
       TimeSensor(
         cycleInterval = 10.0,
         loop = true)
  let myMaterial = Material()
  let dynamic_nodes =
        [
          Shape(
            Value =
              Box(Appearance(Value =
                  myMaterial)))
        ]
  {
    myColor         = myColor
    myClock         = myClock
    myMaterial      = myMaterial
    dynamic_nodes   = dynamic_nodes
    script          = null
  }
```

After initializing the scene without a script, we can load the script from an external parameter that will be assigned in the linking phase. Loading a script requires passing to it the scene, so that the script may access the scene to manipulate it:

```
scene.script := load_script scene
```

The update function invokes the internal update function of all nodes, starting from the statically known and ending with the dynamic ones. Routes are executed in the update function:

```
let update dt =
  scene.myClock.update dt
  scene.myColor.update dt
  scene.myMaterial.update dt
  for node in scene.dynamic_nodes do
    node.update dt
  scene.script.update dt

  myColor.fraction <- myClock.fraction
  myMaterial.diffuseColor <- myColor.value
```

It is important to notice that routes in the update function are represented by the actual chains of field updates that need to be performed; there is no overhead 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 `FieldModified` event with no routes listening.

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 `color interpolators`, 2 `timers` and 6 `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 tested X3D files are an advanced version of the example seen in Section 5: there is a (rather large) set of shapes, colors and timers and the colors of the shapes are changed according to the timers through heavy use of routes. This benchmark shows how heavy the traditional dynamic model is when handling many routes and large scenes.

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

Browser	FPS	FPS (with routes)	Diff %
XNA (300 shapes)	580	510	-12
XNA (680 shapes)	265	224	-15
Octaga (300 shapes)	670	340	-49
Octaga (680 shapes)	372	150	-60
BS C. (300 shapes)	370	300	-19
BS C. (680 shapes)	185	145	-22

Table 1: Intel E6300, 3 GB RAM, nVidia GT 240

Browser	FPS	FPS (with routes)	Diff %
XNA (300 shapes)	670	590	-12
XNA (680 shapes)	310	265	-15
BS C. (300 shapes)	530	368	-31
BS C. (680 shapes)	285	146	-49

Table 2: Intel Core i5, 8 GB RAM, nVidia 310M

Browser	FPS	FPS (with routes)	Diff %
XNA (300 shapes)	640	600	-6
XNA (680 shapes)	310	280	-10
Octaga (300 shapes)	720	403	-44
Octaga (680 shapes)	345	181	-48
BS C. (300 shapes)	500	360	-28
BS C. (680 shapes)	215	135	-37

Table 3: Intel E8500, 2 GB RAM, ATI HD 4800

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 œ3, 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

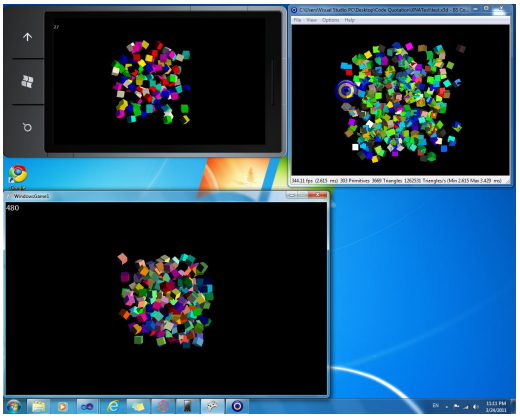

Figure 3: WP7 Emulator, BS Contact and XNA Windows Application

with 150 and 300 shapes respectively plus the usual routes for each shape are summarized in the table below:

Scene	FPS
150 shapes with routes	30
300 shapes with routes	24

Table 4: WP7 (LG Optimus 7)

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.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel approach to optimizing 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.

To make it possible to build safe and powerful scripts we have embedded monadic coroutines into the compiled code. Thanks to type safety 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 and of errors ending up in the final release.

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). We would like to experiment with automated compilation of Javascript scripts, to increase our support of X3D in this direction as well. Also, we wish to study if there are other possible classes of optimizations that can be performed during our code generation phase. Finally, we wish to study what kind of optimizations are needed to ensure fast and high-quality execution of interactive virtual worlds on mobile devices.

References

[1] Raimund Dachselt and Enrico Rukzio. Behavior3d: an xml-based framework for 3d graphics behavior. In *Proceedings of the eighth international conference on 3D Web technology*, Web3D '03, pages 101–ff, New York, NY, USA, 2003. ACM.

[2] L. H. de Figueiredo, W. Celes, and R. Ierusalimschy. Programming advanced control mechanisms with lua coroutines. In *Game Programming Gems 6*, pages 357–369, 2006.

[3] Julian Gold. *Object-Oriented Game Development*. Pearson Addison Wesley, 2004.

[4] Qi Liu and Alexei Sourin. Function-based shape modeling and visualization in x3d. In *Proceedings of the eleventh international conference on 3D web technology*, Web3D '06, pages 131–141, New York, NY, USA, 2006. ACM.

[5] G. Maggiore, F. Pittarello, and M. Bugliesi. Compiling x3d for increased performance and safety. Technical Report 2011-6, Ca' Foscari - DAIS, 2011.

[6] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[7] Daniele Nadalutti, Luca Chittaro, and Fabio Buttussi. Rendering of x3d content on mobile devices with opengl es. In *Proceedings of the eleventh international conference on 3D web technology*, Web3D '06, pages 19–26, New York, NY, USA, 2006. ACM.

[8] Guido Van Rossum and Phillip Eby J. Pep 342 - coroutines via enhanced generators. http://www.python.org/dev/peps/pep-0342/, 2010.

[9] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[10] Philip Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, 1997.

[11] Philip Wadler and Peter Thiemann. The marriage of effects and monads, 1998.

[12] Lei Wei, Alexei Sourin, and Herbert Stocker. Function-based haptic collaboration in x3d. In *Proceedings of the 14th International Conference on 3D Web Technology*, Web3D '09, pages 15–23, New York, NY, USA, 2009. ACM.

[13] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportions. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 31–42, New York, NY, USA, 2007. ACM.