A compilation technique to increase X3D 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

...

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

As virtual worlds grow more and more complex, virtual reality browsers and engines face bigger 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).

Modern browsers and engines are based on a data-driven architecture; see Figure 1 where we report Figure 1 from [6].

In a data-driven engine the engine contains only general knowledge about virtual worlds, but nothing specific about the virtual world it will animate and render. 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, trees, 2d or 3d models; entities may also be purely logical and invisible such as timers, triggers and proximity sensors.

Scripts are behaviors that animate and give life to these entities by making them "act" in an interesting way, either autonomously or reacting to the user's input. Scripts are often divided in two categories: **reactive** scripts that define simple interactions between pairs of entities and **behaviors** that define long lasting behaviors such as an AI or the logic according to which a scene keeps generating new entities.

The usual implementation of an engine (see [2]) 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");
```

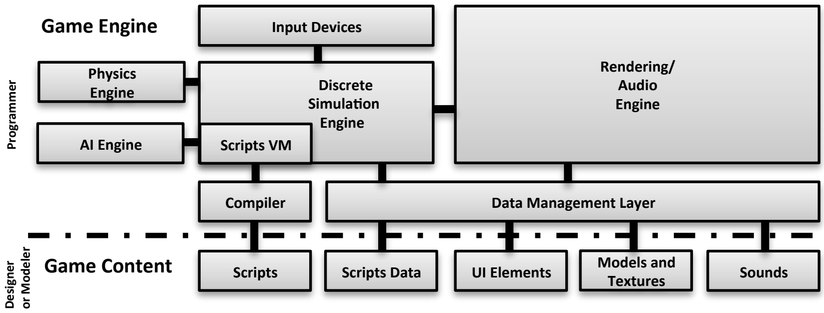


Figure 1: Data-driven engine architecture

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 discuss 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 shape 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. In Section 6 we report some benchmarks that show the speedup of using our technique on a sample scene.

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.

Similar, previous approaches towards extending the X3D standard in order to integrate new nodes that support additional features, such as shaders [4], humanoid behaviors [1] or procedural definitions of shapes and volumes [3, 5].

None of these approaches though, focus on compilation as a means to achieve 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 scripts into our F# scripts
- a transformation from the original X3D file into the final program

Our system starts by translating the X3D scene into F# source code. This source code contains a type definition that describes the entire scene, plus an update function that performs a step of the virtual world simulation (by activating routes and scripts).

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 scene and scripts.

3. COMPILING THE SCENE

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 SceneState =
  {
    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

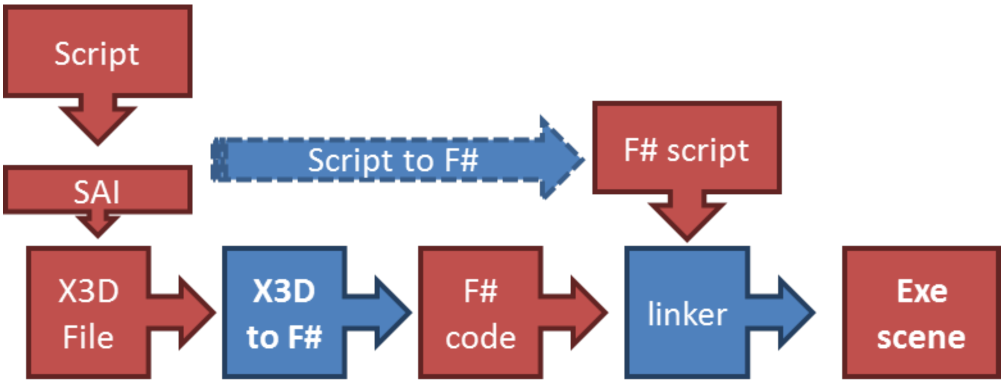


Figure 2: Solution workflow

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:

```
state.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 `state` 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) =
  state.myClock.update dt
  state.box.update dt
  for node in state.dynamic_nodes do
    node.update dt
  script.update dt
```

4. REPRESENTING THE SAI

SCRIPTS as COROUTINES

COMBINING COROUTINES into LARGER and PARALLEL SCRIPTS

SCRIPTS USE A PORTION OF THE STATE DEFINITION

A SIMPLE ELEVATOR SCRIPT

```
let my_script (Box1:Box)=
  script {
    let rand = new Random()
    do! parallel
      script {
        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
      }
      script {
        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
      }
    }
```

5. A CASE STUDY

We will now present a simple case study to see our compiler in action. 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>
```

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:

```
let rec myScene () =

  let rec Appearence1 =
    new Appearance(myMaterial)
  and myMaterial =
    new Material("myMaterial",
      new Vector3(0.80f,0.80f,0.80f))
  and Box1 =
    new Box("Box1",myMaterial,Vector3.One)
  let rec Shape1 =
    new Shape(Box1,Appearence1)
  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 `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.

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 `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 `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 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

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 emu-

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

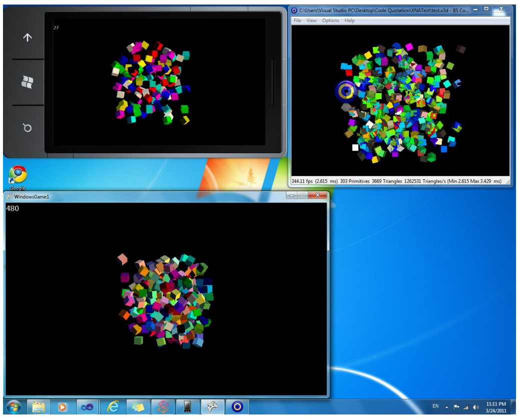
Table 3: Intel E8500, 2 GB RAM, ATI HD 4800

lator 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. 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.


27
C:\Users\Visual Studio PC\Desktop\Code Quotation\XNATest\test.x3d - BS Co...
File View Options Help
Google
WindowsGame1
480

344.11 fps (2.615 ms) 303 Primitives 3669 Triangles 1262531 Triangles/s (Min 2.615 Max 3.429 ms)

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

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

Table 4: WP7 (LG Optimus 7)

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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.