A Functional Scene Graph Interpreter

Corey Nolan
Rensselaer Polytechnic Institute

Abstract

Scene graphs are an increasingly common data structure for scene representation in graphics applications and video games. It organizes objects into a high level representation, usually in C++ or another language in the imperative paradigm. In this paper, I propose a method for creating and editting scene graphs in Haskell, a lazy functional programming language.

Keywords: functional programming, scene graph, haskell

1. Introduction

1.1 Scene Graphs

The graphics world saw the first commercial scene graph in 1988 with PHIGS [1], a 3D graphics programming library and predecessor to OpenGL. Its visualization process was simple: define and link to a model in the graphics database, then create a workstation to display this model. PHIGS suffered many severe limitations such as inefficiency and lack of ray tracing/radiosity solutions.

Scene graphs have evolved to take on a variety of forms, but the general function remains the same: arrange logical and often spatial scene elements into a graph or tree structure. Generally, these structures consist of a series of nodes where each node has some number of children and one or no parents. Scene graphs can also be replaced with or include spatial partitioning and bounding volume hierarchies (BVHs).

Scene graphs are used in many modern-day interactive applications, such as AutoCAD, Adobe Illustrator and OpenSceneGraph. They're also utilized in many modern game engines including Ogre, Panda3D and Gamebryo.

1.2 Haskell

Haskell [2] is a purely functional programming language first introduced in 1990. The goal was to unite existing functional languages at the time to a single highly research-oriented effort. Haskell's main strengths lie in concurrency and parallelism, both promising possibilities in the world of cloud rendering, multicore architectures and supercomputers.

Haskell has several key principles [3]. It's lazy-evaluated and purely functional. It has no formal semantics. It's distinctive in that it's the first programming language to implement type classes (see details in Section 2: Implementation).

1.3 Motivation

A functional approach has two major potential benefits: one, to the

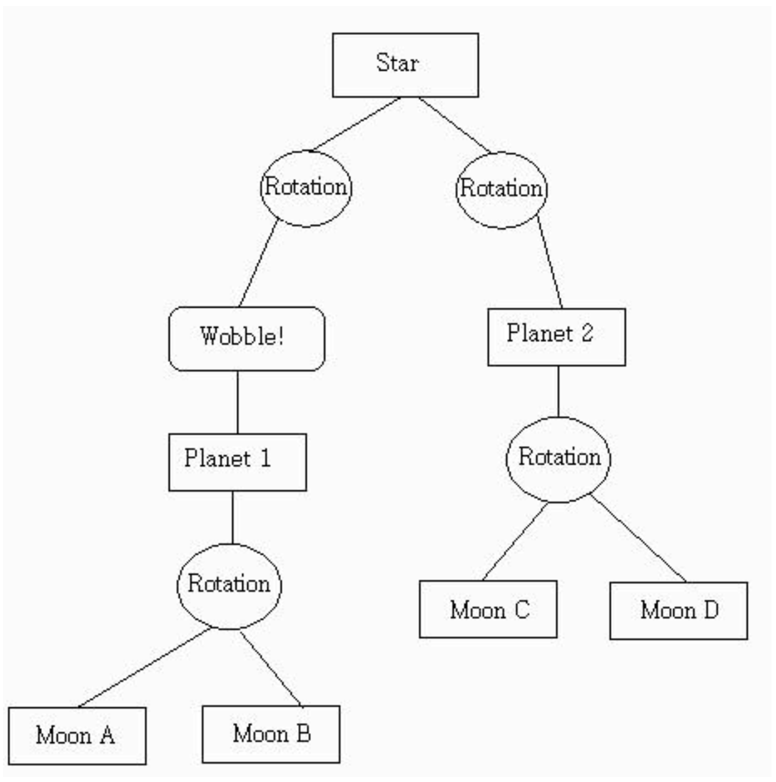

Figure 1: A visualization of a sample scene graph hierarchy. Image taken from article by Garret Foster [8]

community of functional programmers that may have need of it; and two, to users of scene graphs if they find the functional model better suits their needs.

The functional programming style may provide distinct advantages in scene graph implementation. For one, referential transparency and elimination of "side effects" in code allows for quick, efficient debugging. Lazy evaluation could cause drastic improvements in runtime efficiency.

1.4 Related Work

In 2001, Claus Reinke [4] began research into an experiment known as *FunWorlds* (Functional Programming and Virtual Worlds), using VRML (an old standard file format for representing 3D objects) and later HOpenGL. On the project's website he states, "…the current experiments are encouraging, both in terms of expressiveness and in terms of performance." Unfortunately, he stopped working on the project not long after the HOpenGL implementation.

A 2000 paper by Jurgen Dollner and Klaus Hinrichs [5] describes a scene graph model in which graph nodes are used to contains the rendering objects, a set of different categories of objects that appear in the rendered scene. Their work treats light sources and cameras as global objects that are pre-evaluated while render-objects are evaluated in depth-first order.

Gerhard Reitmayr and Dieter Schmalstieg reintroduce a number of declarative properties in their 2005 paper *Flexible Parameterization of Scene Graphs* [6]. Building on Dollner and Hinrich's ideas, they describe the use of a context sensitive scene graph, where the context represents a mapping of indices within the graph to values, and present the exposed graph in a template pattern that decouples content and presentation.

Both papers on generalized scene graphs borrow many ideas from OpenInventor [7], a high level scene graph in OpenGL focused on code convenience and efficiency. Unfortunately it's usually slower in runtime than hand-coding OpenGL, however it remains pretty popular for its ease of use and extensibility.

2. Implementation

The implementation of scene graphs in the functional paradigm has many differences from the object-oriented scene graph, despite the common data structure.

2.1 Data Organization

The current implementation contains datatypes organized in a hierarchical structure: Scenes contain a set of Models and a set of Lights. A Model consists of a Mesh and Material. Outside of the hierarchy are the Camera and RasterImage data used in rendering implementation.

In my implementation, I utilize Haskell's type class feature by representing portions of the scene as Generalised algebraic datatypes (GADTs) [9].

2.2 Utilizing GADTs

GADTs are datatypes that lack a standardly typed constructor. Types can be defined arbitrarily and evaluated according to their definition. GADTs are used here to represent expressions of numerous datatypes, as seen in this code snippit:

```
data Expr x where

  ModelE :: Expr Mesh -> Expr
Material -> Expr Model

  MeshLitE :: Mesh -> Expr
Mesh

  MaterialLitE :: Material ->
Expr Material

  CameraLitE :: Camera ->
Expr Camera

  RasterizeE :: Expr Scene ->
Expr Camera -> Expr
RasterImage
```

Similarly, the code contains an evaluation function for the given expressions:

```
eval :: Expr a -> IO a

eval expr = case expr of

  ModelE meshE matE -> op2
Model meshE matE

  MeshLitE mesh -> return
mesh

  MaterialLitE mat -> return
mat

  CameraLitE cam -> return
cam

  RasterizeE sceneExpr
camExpr -> do

     scene <- eval sceneExpr

     cam <- eval camExpr

     return $ RasterBuffer 0

  _ -> error "Unrecognized
production"
```

GADTs help to simplify types and keep the code organized and succinct, while provided a large amount of flexibility for later additions or modifications to the scene graph structure.

3. Conclusion/Future Work

The implementation as it stands represents a scene graph in only an abstract sense; I'm now working to create a more concrete implementation of the graph along with rendering solutions.

One interesting optimization would take the concurrency and parallelism strengths of Haskell and apply it to ray tracing or radiosity solutions: embarrassingly parallel portions such as form factor calculation could be significantly enhanced by a capable system of parallelization.

Other additions to this research include using an octree to spatially recognize objects in the scene and creating a bounding volume hierarchy. Experiments with the lazy-evaluation characteristic may also produce optimizations.

4. Bibliography

[1] Smith, Terry, "An Introduction to PHIGS", University of Arkansas, 1995.

[2] Haskell Wiki, available at http://haskell.org/.

[3] P. Hudak, J. Hughes, S. L. P. Jones, and P. Wadler, "A history of Haskell: being lazy with class", B. G. Ryder and B. Hailpern, editors, HOPL, pages 1–55. ACM, 2007.

[4] Reinke, Claus, "FunWorlds/HOpenGL – Functional Programming and Virtual Worlds," http://www.cs.kent.ac.uk/people/staff/cr3/FunWorlds/, 2002.

[5] Dollner, Jurgen, Hinrichs, Klaus, "A Generalized Scene Graph", VMV Proceedings, 2000.

[6] G Reitmayr, D Schmalstieg, "Flexible parametrization of scene graphs" proceedings IEEE VR, 2005.

[7] "Open Inventor", Silicon Graphics Inc., http://oss.sgi.com/projects/inventor/, 1993-2009.

[8] Foster, Garret, "Understanding and Implementing Scene Graphs", Gamedev.net: http://www.gamedev.net/reference/articles/article2028.asp, 2003.

[9] "Generalised algebraic datatype", Haskell Wiki, http://haskell.org/haskellwiki/GADT.