Using 3D Graphics in Combination with other Models of Computation

Yasemin Demir, Man-Kit Leung

EECS 290N Report
May 15, 2009

University of California at Berkeley
Berkeley, CA, 94720, USA

Abstract

This paper presents a framework for developing a new graphics domain, called GRO, in Ptolemy II [1]. GRO is an OpenGL-based implementation that imports utilities, semantic and functionalities of OpenGL. The specification provides users to fully utilize the potential in the communication between the rendering engine and the computation in the model. This bottom-up design approach allows developers to create a event-based MoC that provides user, a sophisticated and exible specification. Based on OpenGL advantages, GRO is a user-friendly graphic domain that provides a declarative specification for the user to use. At the same time, it can also achieves better computation efficiency by migrating computation from the model (CPU) onto GPU's. Demos are provided showing the implementation of the new graphics domain.

1 Introduction

3D computer graphics is popular in modern technologies. 3D displays and real-time 3D viewing of modern systems, such as 3D human animations, tele-immersion systems, real-time organ viewing, entertainment environments and other applications of 3D computer graphics shows us the attractiveness of 3D technologies in nowadays life. Thus, 3D computer graphics turn out to be a significant tool that allows users and developers more entertaining and visual platforms and this off-course increases the share in the market.

Modern embedded systems also puts high demands on simulations using 3D graphics. This also increases the heterogeneity of embedded systems that is another important issue in modern technologies. However, this heterogeneity needs special environments that supports simulating each different subsystem and enabling the interaction between these subsystems. A real-time walking robot control system with a synchronous 3D visualization of its movements can be a good example for such heterogeneous embedded systems. In this example, a model of motion including calculations of next position and next movement can be a subsystem.

This makes Ptolemy II a good example for such environments that provides a broad range of computational models in an actor based platform. Ptolemy II allows users combine different models in different domains in the specified hierarchy. This advantage of Ptolemy II encourages developers to implement different platforms including such graphical interfaces, allowing the combination of 3D graphics with other models of computation (MoC) such as GR domain [2].

GR domain provides rendering of two-dimensional and three-dimensional graphics in Ptolemy II that is based on Java3D semantics and has a scheduling order based on scene graphs that are used to optimized the rendering of complex scene. It imposes certain constraints on scene objects. The representation is an directed acyclic graph which is a directed graph with no closed cycles. Each leaf node of a scene graph contain an object of within the scene to be drawn. The object can possibly be empty or null, in which case it is not rendered in the resulting image. Each immediate node (s.t. it has one or more children nodes) represents a coordinate space in which its children exist. The immediate node itself is a transformation that transforms the coordinate space. These transformations can composed together s.t. we can connect multiple transformation nodes before connecting to an object (leaf) node. There is one special node called the root, which is the original unchanged coordinate space-scene graphs are acyclic directed graphs where actors are connected in an acyclic directed graph. Based on scene graph semantics actors are fired according to GRScheduler that is done by performing a topological sort on all the actors. In addition to GRScheduler, GR domain employs the SDFScheduler (Synchronous Data Flow Scheduler) to determine firing rules for other MoC that are used within a GR model where each GR actors is fired according to scene graph based GRScheduler and every other non-GR actors obeys the firing rules determined by SDFScheduler. GR is a developing platform that has some restrictions because of Java3D specifications. For this reason, we implement a new graphic domain avoiding the restrictions of Java3D and allowing a more user-friendly platform to developers by implementing the functionalities that OpenGL API specifies.

This paper introduces the implementation of OpenGL concepts within the new domain GRO in the Ptolemy II environment. We describe graphics modeling using actor-oriented models in Section 2. We give a brief introduction about OpenGL API specifications and give a reasoning for employing OpenGL API in our implementations. in Section 3. In Section 4 we introduce the basic idea behind the new graphical domain GRO. In Section 5, we explicate our implementation scheme in detail. In Section 6 we state our conclusions and we point out our potential future work.

2 Graphics Modeling using Actor-Oriented Models

Graphics modeling is the way to specify the appearance of a space which consists of a collection of objects. 3D modeling has three basic phases that first describes the process of forming the shape of an object, and second defines the motion and placement of objects within a scene and renders and produces an image of an object finally. The specification expresses the phenotypic attributes (e.g. shape, size, color, and etc.) of the objects. In addition, it specifies several types of relationship. For instance, one of these relationship is orientation, which can be seen as the relationship between the objects and the space. Shading is another relationship, which is between the objects and the light sources. The primary application of graphics modeling is animation, which often involves relationship across time or frames. A general graphics modeling language should provide a sufficient set of primitives to allow one to express these relationship.

Actor-oriented modeling [3] is a conceptual modeling framework for modeling, understanding, and reasoning about, a wide range of concurrent systems. It provides basic constructs called *actors* that encapsulate internal states. Actors can also have *port(s)* that send or receive data to communicate with other actors. Communication semantics is purposefully delegated to the model for flexibility. Actor-oriented modeling gives us a clean visual syntax and a structured way of composing components. It allows users to, still, customize various constraints on the model semantics. Our work is an extension to Fong's [2], which shows how actor-oriented modeling, using dataflow semantics, can adapt to hierarchical scene graph, which is a popular approach in graphics modeling. We want to continue in this direction of using actor-oriented modeling in the graphics domain.

3 OpenGL API Functionalities

OpenGL (Open Graphics Library) is a standard specification that describes a set of functions and the precise behaviours that they must perform writing applications that produce 2D and 3D computer graphics. It serves two main purposes, one is to hide the complexities of interfacing with different 3D accelerators, by presenting the programmer with a single, uniform API and second is to hide the differing capabilities of hardware platforms, by requiring that all implementations support the full OpenGL feature set.

OpenGL and other various rendering and modeling software packages often contain some support for scene graphs in order to avoid complicated task of keeping track of all the matrices. In OpenGL, we have the matrix stack and various operations allowing to modify the current model view matrix, most of them allowing to multiply it by some simple transformation on the right. Together with the matrix stack, this makes it very easy to convert a scene graph into OpenGL code which is implemented as below;

- Save the current modelview matrix by pushing it onto the matrix stack.
- Multiply the modelview matrix on the right by the transformation associated with the edge.
- Call the drawing procedure recursively, pretending that the endpoint of the edge is the root.
- Restore the original modelviev matrix, by popping it from the matrix stack.

One reason for our effort in building a new graphics domain on the OpenGL API is because OpenGL is industry's most widely adopted graphics standard which brings thousands of applications to a wide variety of workstation platforms. Another important reason is that OpenGL has hardware support which means if hardware 3D acceleration is present, OpenGL can use it. As an API, OpenGL does not depend on any particular language feature, and can be made callable from almost any programming languages with the proper bindings. Such bindings exist for VB, Ada, Pascal, Delphi, Python, Lua, Perl, Haskell, Java, C and C^{++}. OpenGL is capable of high visual quality and performance that allows developers to do broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to generate and display 2D/3D graphics efficiently. In addition, OpenGL serves developer several advantages such as serving user the only truly open, vendor-neutral, multi-platform graphics standard. Since various OpenGL implementations have been available for more than seven years on a wide variety of platforms, it is ensured by backward compatibility requirements that it is stable and legacy programs are not outdated. Briefly, OpenGL provides reliable, portable, evolving, scalable, user-friendly and well-documented platform in which developer is able to render well-structured graphics with an intuitive and logical design.

OpenGL is widely used in implementing a variety of graphics tools that are commonly used animation, game and virtual reality technologies. Blender, Celestia, RenderMan are popular examples for such tools that are implemented on the top of OpenGL.

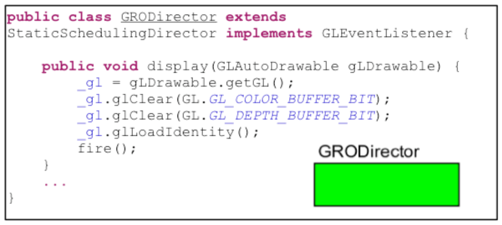
4 GRO Domain

The basic idea behind the GRO domain is to build a precise model that handles geometry and transformation of 3D objects which would rely on OpenGL semantics. Opengl convert scene graph semantics in to OpenGL code. This is implemented by using the matrix stack and various operations allowing to modify the current model view matrix. By mapping scene graph semantics in to these matrix stack operations OpenGL avoids complicated task of keeping track of all the transformation matrices and save more space for other calculations which increases the rendering rate on each different platform.

We mapped these scheduling semantics in to our new domain and implemented a GROScheduler that is basically similar with scene graph semantics but has an opposite ordering rule that is also rely on object semantics of OpenGL. We write some of the OpenGL specific 3D objects ,such as Box3D, Line3D, Point3D, Triangle3D, Ellipse3D, Cylinder3D and transformation actors that are Rotation and Translation. The details of our implementation is described in the following section.

5 Implementation

As the first stage, we want to reproduce the usability of OpenGL in an actor based platform by retaining the OpenGL semantics that avoids complicated task of keeping track of all the transformation matrices and save more space for other calculations We Implement the main classes of actors of a graphics API such as view screen, transformation actors that are Translation and Rotation in 2D/3D, and graphic objects such as Point3D, Line3D, Square3D, Ellipse3D, Sphere3D, Triangle3D and Box3D. The visual syntax of GRO is almost identical to GR. This way, user can construct graphic models without learning a new syntax.



```
public class GRODirector extends
StaticSchedulingDirector implements GLEventListener {

    public void display(GLAutoDrawable gLDrawable) {
        _gl = gLDrawable.getGL();
        _gl.glClear(GL.GL_COLOR_BUFFER_BIT);
        _gl.glClear(GL.GL_DEPTH_BUFFER_BIT);
        _gl.glLoadIdentity();
        fire();
    }
    ...
}
```

Figure 1. GRO Director.

5.1 GRO Scheduler

OpenGL exposes a callback mechanism for rendering; in particular, it allows the client to attach GLEventListener objects to perform rendering. Each GLEventListener implements a display() method that is invoked by OpenGL. Mapping OpenGL to the Ptolemy framework, our implemented this listener object which is the GRODirector. It implements the GLEventListener interface (depicted in Figure.1). GRODirector currently extends the StaticSchedulingDirector with a custom GROScheduler, which we will introduce in a later section. The display() method of the GRODirector first initializes the display buffer and renders the frame by firing the graphics actors in the model, according to the order given by the scheduler. A snapshot of result of this model is shown in Figure.2.

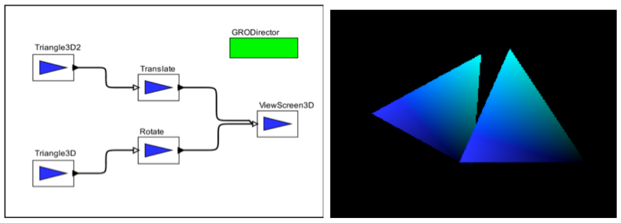
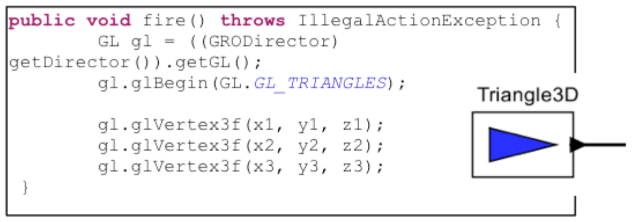


Figure 2. A snapshot of a GRO model.

Each transformation and graphics actor contains a block of OpenGL rendering code. The order of firing them is statically computed from the given structure of the model. Thus, no communication tokens actually need to be passed. The connections, however, do provide information for computing the schedule. The OpenGL rendering code mapped to the fire() method of the Translate and the Triangle3D actors is given in Figure.3 and Figure.4.

5.2 GRO Scheduler

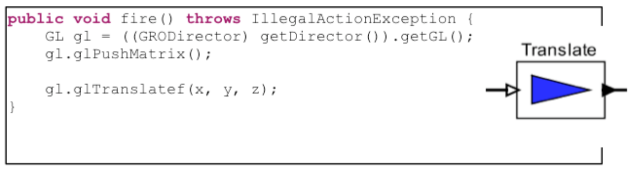
A GRO model is currently executed by iterating a static schedule given by the *GROScheduler*. The design space



```
public void fire() throws IllegalActionException {
        GL gl = ((GRODirector)
getDirector()).getGL();
        gl.glBegin(GL.GL_TRIANGLES);

        gl.glVertex3f(x1, y1, z1);
        gl.glVertex3f(x2, y2, z2);
        gl.glVertex3f(x3, y3, z3);
  }
```

Figure 3. Triangle3D Actor.



```
public void fire() throws IllegalActionException {
    GL gl = ((GRODirector) getDirector()).getGL();
    gl.glPushMatrix();

    gl.glTranslatef(x, y, z);
}
```

Figure 4. Translate Actor.

for the scheduler is pretty flexible since GRO actors do not communicate any data tokens with each other (even though their ports are connected). There is a global OpenGL object, called *GL*, that is shared implicitly by all GRO actors. Firing the GRO actors essentially make method calls to this *GL* object, which does the actual rendering. A static schedule is constructed by a **depth-first pre-order** traversal of the graph. The traversal starts from a GRO root actor which is called *ViewScreen*, and it proceeds to the intermediate and leaf actors. These intermediate actors are Transformation actors like *Rotate*, *Translate*, and *Scale*, while the leaf actors are 3DObject actors like *Line*, *Triangle*, *Box*, and etc.

A benefit of fixing ourselves to such statically computed schedule is that we can also statically determine the pushing and popping of projection matrix and hence lift some of the programming burdens from the user. Operations of pushing and popping projection matrices are central to the OpenGL graphic pipeline in keeping track of the state of the projection for objects. These operations are implicit under the current GRO semantics. The GROSchduler schedules a pushMatrix() before executing each Transformation actor and a popMatrix() after its execution. This ensures that we retain the state of the projection and return to the original state after drawing every object under that projection. We realize that there is an optimization opportunity for the case where the Transformation has only one child. In that case, there is no need to push or pop a matrix.

Here, we described the current execution strategy for a GRO model. However, we acknowledge that there are other execution strategies that are more expressive, possibly involving explicit pushMatrix() and popMatrix(). The challenge is in constructing a corresponding graphical syntax that is understandable to a model designer.

6 Conclusion and Future Work

In this paper, we introduce a framework to implement a graphics domain in Ptolemy II that is on top of OpenGL. We explained how OpenGL semantics can be mapped to the Ptolemy II abstract semantics (e.g. postfire(), prefire() and etc.). We then give details about our implementation of GRODirector, and some actor that we have in our library. By using the advantages of OpenGL we implemented Line3D and Point3D actors which are not supported by Java3D and are really common to use in games, stick figure animations, etc.

As our future goal, we would like to extend the GRO semantics beyond the scene graph. This would let programers to avoid complicated task of keeping track of all the transformation matrices. We want to employ these graphical implementations with other MoC which would be useful in building different concurrent real-time models with graphical interfaces. We think that this would give the user another perspective of understanding real-time models. We want to investigate timing semantics in graphics rendering which now relies on a frame based timing. This would allow the user to use declarative syntax to specify models. In addition to these, we want to implement more sophisticated actors that would support for a variety of geometric primitives, such as polygon meshes, fast subdivision surface modeling, Bezier curves, NURBS surfaces, Meatballs and Digital sculpting. We are pursuing to add key framed animation tools including, inverse kinematics, curve and lattice-based deformations, shape keys (morphing), soft body dynamics including mesh collision detection and particle system with collision detection.

References

[1] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Ptolemy ii: Heterogeneous concurrent modeling and design in java. *Memorandum UCB/ERL M99/44, EECS, University of California, Berkeley*, July 19, 1999.

[2] C. Fong. Discrete-time dataflow models for visual simulation in ptolemy ii. *Master's Report, Memorandum UCB/ERL M01/9, Electronics Research Laboratory, University of California, Berkeley*, January 2001.

[3] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.