GLOD: A Geometric Level of Detail System at the OpenGL API Level

Jonathan Cohen[*] David Luebke[+] Nathaniel Duca[*] Brenden Schubert[+]
[*]Johns Hopkins University [+]University of Virginia
http://www.cs.jhu.edu/~graphics/GLOD

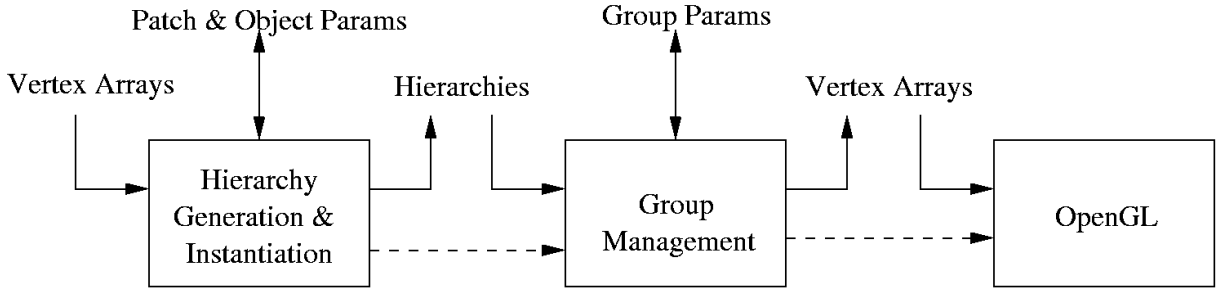

Figure 1: The GLOD object and dataflow model.

1 INTRODUCTION

Level of detail (LOD) techniques are widely used today among interactive 3D graphics applications, such as CAD design, scientific visualization, virtual environments, and gaming, allowing applications to trade off visual fidelity for interactive performance. Many excellent algorithms exist for LOD *generation* as well as for LOD *management* [Luebke 2003]. However, no widely accepted programming model has emerged as a standard for incorporating LOD into programs.

Existing tools generally fall into two categories: mesh simplifiers and scene graph toolkits. Mesh simplifiers address the LOD generation problem, taking a complex object and producing simpler LODs, but they do not attempt to address LOD management at all. Scene graphs such as OpenGL Performer [Rohlf 1994] perform LOD management, but go to the opposite extreme; they provide heavyweight "all or nothing" solutions that lump LOD in with myriad other aspects of an interactive computer graphics system, constraining the form of the overall application.

In this poster we present GLOD, a tool for geometric level of detail that provides a full LOD pipeline in a lightweight and flexible application programmer's interface (API). This API is a powerful, extendible, yet easy-to-use LOD system, supporting discrete, continuous, and view-dependent LOD, multiple simplification algorithms, and multiple adaptation modes. GLOD is *not* a scene graph system; instead, it is an API integrated with OpenGL, an existing and popular low-level rendering API. With this formulation, we start to think of geometric level of detail as a fundamental component of the graphics pipeline, much like mip-mapping is a fundamental component for controlling detail of texture images. The system itself should be an excellent tool for interactive visualization applications written using OpenGL.

2 GLOD API

Our design goals for the GLOD API (see Figure 3) focus on providing a lightweight model for the creation, management, and rendering of geometry. To maximize its appeal to multiple audiences, GLOD should be fast, extensible to different LOD algorithms, and easy to integrate into existing applications. Furthermore, it should allow incremental adoption rather than locking developers into all pieces of the GLOD framework. To accomplish these goals, GLOD API is tightly integrated with the industry standard OpenGL API, so our design decisions are guided as if GLOD were a component of OpenGL.

The data handled by GLOD is organized into three principal units: *patches*, *objects*, and *groups*. A patch is the principal unit of rendering. A patch is specified to GLOD using the OpenGL vertex array interface. Drawing a patch is much like drawing a vertex array, the chief difference being that what you get is an LOD of the original arrays. The application may change rendering state, such as bound textures, on a per-patch basis at the time of rendering; GLOD does not interfere with rendering state.

An object is the principal unit of LOD generation. The application designates one or more patches as an object before initiating the LOD generation process. Thus multiple patches may be simplified together into crack-free levels of detail. GLOD also supports memory-efficient instancing of objects to provide efficient LOD management for applications which render objects in multiple locations.

A group is the principal unit of LOD management. An application places one or more objects into a group. At each frame, GLOD adapts the LOD of all patches of all objects in each group according to the specified adaptation mode and current OpenGL viewing matrices.

The GLOD pipeline is designed to allow flexible motion of data into and out of it as desired by the application, as illustrated in Figure 1. The original geometry is specified as patches using the vertex array mechanism. The application can then set a number of per-patch and per-object LOD generation parameters to determine how the LOD hierarchy is constructed. For example, parameters may be used to select a simplification operator, error metric, hierarchy type (e.g. discrete, continuous, view-dependent), importance values, etc. A special hierarchy type allows the programmer to manually build discrete hierarchies from a set of existing LODs. An entire hierarchy may be read back by the application to save it to disk, allowing it to be re-used in a later execution without regenerating it. Group parameters specify management modes such as the error mode (object-space or screen-space), adaptation mode (error threshold or triangle budget), morphing parameters, etc. After adapting a group, the individual adapted patches may be read back, again through the vertex array mechanism. The application can store these vertex arrays, pass them to OpenGL for rendering, etc. This complete set of data paths allows applications to incrementally adopt GLOD.

3 DISCUSSION

We have currently limited the scope of GLOD to filtering geometric detail without interfering with rendering state. This has several benefits. The application may safely employ complex rendering algorithms, including multi-pass algorithms, as well as custom vertex and fragment programs. For example, applications can use normal mapped LODs without difficulty in GLOD. Many user-

Proceedings of the 14th IEEE Visualization Conference (VIS'03)
0-7695-2030-8/03 $ 17.00 © 2003 IEEE

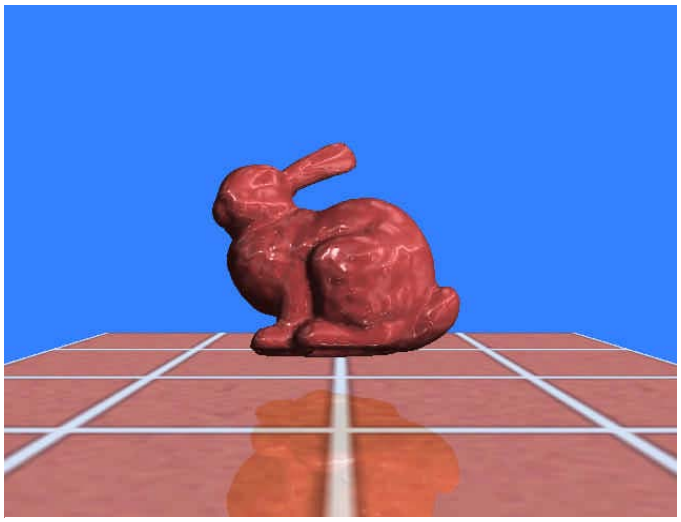

Figure 2: Bunny rendered in GLOD using a multipass rendering algorithm, demonstrating GLOD's policy of non-interference with the underlying graphics system.

defined vertex program parameters can pass through GLOD filtering. However, this is not applicable for all vertex programs. Also, our non-interference policy makes some forms of LODs, such as textured impostors, difficult to support because they require us to change rendering state.

At the time of this writing, a pre-release version of the GLOD system is available from our web site:

http://www.cs.jhu.edu/~graphics/GLOD

The current implementation supports both discrete and view-dependent hierarchy formats, several simplification operators, error threshold and triangle budget adaptation modes, etc. We hope that this open source system will provide a viable and convenient pathway for level of detail research to migrate from the research lab to full deployment. With a wide array of simplification algorithms, hierarchical data representations, and management policies in their hands, all available through the setting of a few parameters, application developers will have tremendous power to select the implementations that meet their needs.

REFERENCES

Luebke, D., M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufman. 2003.

Rohlf, J. and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Proceedings of SIGGRAPH 94*. July 24-29. pp. 381-395.


```
glodNewGroup(grpname);
glodDeleteGroup(grpname);
```

Create a group to contain and manage objects. Deleting a group deletes all its objects.

```
glodNewObject(objname, grpname, format);
```

Create an object for a particular hierarchy format and place in the named group.

```
glodInsertArrays(objname, patchname, mode,
                 first, count, level, error);
glodInsertElements(objname, patchname, mode,
                   count, type, indices,
                   level, error);
```

Put a patch into an object using vertex arrays. Level and error can be used to load an LOD generated elsewhere into a discrete hierarchy, but are typically set to 0.

```
glodBuildObject(objname);
```

Complete an object and convert to hierarchy in the selected output format.

```
glodInstanceObject(objname, instname, grpname);
```

Instantiate an existing object by sharing its geometry hierarchy data, and place into a group.

```
glodDeleteObject(objname);
```

Delete an object (which removes it from its group).

```
glodBindAdaptXform(objname);
```

Capture an object's viewing parameters for adapting (not drawing – GLOD does not change the OpenGL transformation state).

```
glodAdaptGroup(grpname);
```

Adapt LOD for all the objects in a group according to the group's ADAPT_MODE.

```
glodDrawPatch(objname, patchname);
```

Draw one patch of an object.

```
glodFillArrays(objname, patchname, first);
glodFillElements(objname, patchname, type,
                 elements);
```

Read back current adapted object into vertex arrays

```
glodGetObject(objname, data);
glodLoadObject(objname, data);
```

Read back an object's hierarchy so it may be saved and later reloaded to GLOD.

Figure 3: The GLOD API