GLOD: Level of Detail for the Masses

Jonathan Cohen[*] David Luebke[+] Nathaniel Duca[*] Brenden Schubert[+]

[*]Johns Hopkins University [+]University of Virginia

Abstract

We present GLOD, a geometric level of detail system integrated into the OpenGL rendering library. GLOD provides a low-level, lightweight API for level of detail operations. Unlike heavyweight scene graph systems, GLOD supports incremental adoption and may be easily integrated into existing OpenGL applications. GLOD provides a simple path for developers to add level of detail to their system, while retaining a minimalist close-to-the-hardware approach compatible with high-performance rendering and future extension of the base OpenGL layer.

Keywords: level of detail, OpenGL, application programmer's interface

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. The field of LOD has grown quite mature. For example, many excellent algorithms exist for LOD *generation*, or creating simplifications of polygonal meshes; they range from fast and simple to slow but sophisticated, and the resulting simplified models (themselves called levels of detail or *LODs*) range from crude to excellent. A wide gamut of techniques have also been presented for LOD *management*, or the run-time task of adjusting the level of detail to respond to changes in the scene (such as movement of the viewpoint or objects) while balancing detail with performance. These range from simple distance-based approaches that select one of a set of discrete LODs to elaborate view-dependent LOD methods that perform fine-grained adaptation of the polygonal tessellation on the fly.

Probably almost every high-performance interactive graphics application or toolkit built in the last five years utilizes LOD to trade off visual fidelity for interactive performance. However, no widely accepted programming model has emerged as a standard for incorporating LOD into programs. Existing tools fall into two categories: mesh simplifiers and scene graph toolkits. Mesh simplifiers provide only a partial solution; they take a complex object and produce simpler LODs, but do not attempt to address LOD management at all. Scene graphs 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: hierarchical transformations and instancing, view-frustum culling and visibility, memory management and paging, and so on. A good example of a scene graph is OpenGL Performer [Rohlf and Helman 1994]; though a powerful and well-written library for high-performance rendering, a developer wishing to use Performer for LOD must use the full set of Performer scene graph constructs, and indeed must build his or her entire interactive graphics system on Performer, from the ground up. Often this requirement is too restrictive or inappropriately burdensome, and instead the developer ends up with a different burden: creating yet another custom LOD system.

In this paper we present GLOD, a tool for geometric level of detail that provides the full LOD pipeline in a lightweight and flexible application programmer's interface (API). This API is a full-featured, 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 into OpenGL, an existing and popular low-level rendering API. Before presenting the GLOD API, we first discuss the design principles underlying this choice.

2 DESIGN PRINCIPLES

Four fundamental principles have driven the formation of GLOD and its API:

- **Incremental adoption**: Unlike scene graph approaches, users of the GLOD API should not be forced into the use of the entire pipeline, but should instead be allowed to independently adopt just the portions of the system that they desire. For example, GLOD can be used to generate discrete LODs which the user then manages and renders directly, or GLOD can be used to manage LODs produced by the user.
- **FastPath principle**: For every type of LOD task, the API must support a way to achieve that task in a high-performance fashion. For example, if the advanced user wishes to specify that GLOD cache and render LODs from a specific portion of fast on-card video memory, the API supplies parameters to support this.
- **Ease of use**: The API should be straightforward to use, especially to any developer already familiar with OpenGL. For example, using GLOD must not require developers to add lots of complex code to their system, nor handle their OpenGL code in any new or unusual fashion, nor learn any new calls if an existing OpenGL call will suffice.
- **Extensibility**: We require two forms of system extensibility. First, through development efforts, the GLOD API must be capable of supporting a wide variety of geometric level of detail tasks. For example, researchers and developers must be easily able to extend the capabilities of GLOD. Second, the GLOD API must not lose its usefulness through lack of development, but instead must evolve without code redesign as OpenGL acquires new extensions. For example, GLOD should support any new OpenGL drawing extensions that emerge without requiring extra development effort.

These design goals suggest a low-level and lightweight API supporting a highly flexible input/output model. This minimalist API should leave as much as possible to the user, keeping the interface simple for simple applications while providing parameters where necessary for advanced users to hook into features necessary for high-performance rendering. We should model our API on existing OpenGL semantics, such as features and extensions for vertex arrays, textures, lights, and texture compression. For extensibility and performance, we should keep the API very near the rendering layer, while making the bare minimum of changes to that layer.

3 FOLLOW THE RED BOOK ROAD

Our implementation of the GLOD API is tightly integrated with the industry standard OpenGL API. We consciously place GLOD as close to the OpenGL driver as possible: following design techniques used by Chromium [Humphreys 2002], our prototype implementation actually intercepts OpenGL commands to effectively masquerade as the OpenGL driver, and our design

decisions are made as if GLOD were a fundamental part of OpenGL. Following through on this strategy, we plan to propose GLOD as an official ARB-recognized OpenGL extension.

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

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

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

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 1: The GLOD API.

The gains of such a design meet our basic design principles well: first, by choosing an industry standard, we base our system on a robust developer-backed system that is guaranteed to stay up-to-date as graphics technology changes. Second, the use of the OpenGL brings along with it a programming model with which users are accustomed and comfortable. Most importantly, OpenGL brings along a design philosophy that can be used to guide the complex decision processes of generating a general purpose geometric level of detail API.

4 GLOD API

The GLOD API focuses on providing a lightweight model for the creation, management, and rendering of geometry. Here, we discuss both the API itself, and the design choices made to reach the format presented here. We discuss several topics relating to GLOD: management of dataflow in the system, the interface into the system, and how we manage data formats within the API to achieve maximum versatility. Finally, we have implemented the GLOD API and have tested it against a variety of usage scenarios as a means to verify and establish the viability of the API as a Level of Detail interface.

The fundamental achievements of GLOD all derive from two judicious choices for fundamental data types: first, we must choose the fundamental geometric unit that GLOD operates on, and next, we must chose the fundamental units that data can enter and leave the GLOD system. These units, if chosen properly, make it possible for GLOD to exist peacefully at a driver level without interfering either with OpenGL capabilities or system resources.

4.1 Geometric Primitives

Our fundamental geometric primitive in GLOD is the *patch*. We define a patch as the smallest unit of geometry for which the application developer may change the global rendering parameters. A patch, like a compiled vertex array, is rendered as a block of geometry with a single rendering state; the developer is responsible for setting the rendering state. Choosing an atomic drawing unit for GLOD frees GLOD from the burden of managing the large (and ever-increasing) amount of OpenGL rendering state. In fact, not only does it make it unnecessary to manage GL state, but it expressly forbids the very idea: Drawing a GLOD patch is as simple as setting up the rendering parameters in the usual fashion and then calling `glodDrawPatch()` on the patch in the exact same way that one might call `glDrawArrays()`, the chief difference being that what you get is an LOD of the original arrays.

Although patches are appropriate units for rendering, they are not sufficient for describing the construction of a multiresolution hierarchy. Some models contain multiple sets of geometry, each of which must be rendered with different rendering parameters, but which are topologically connected. If we were to assign each such set to a patch and independently construct a multiresolution hierarchy for each, we could not prevent the formation of *cracks* between these models when we later adapt their resolutions and render them. To address this problem, we introduce the notion of an *object*, which is a collection of patches that are combined during the `glodBuildObject()` process into a single multiresolution hierarchy. This abstraction provides enough information to the system to maintain connectivity between adjacent patches during the both the building and later adaptation processes.

Just as a developer might render an OpenGL vertex array multiple times during a frame to draw multiple instances of the same object, a GLOD object may need to be rendered multiple times at different adaptation levels. Thus GLOD provides an Instance construct to represent different instantiations of a GLOD Object.

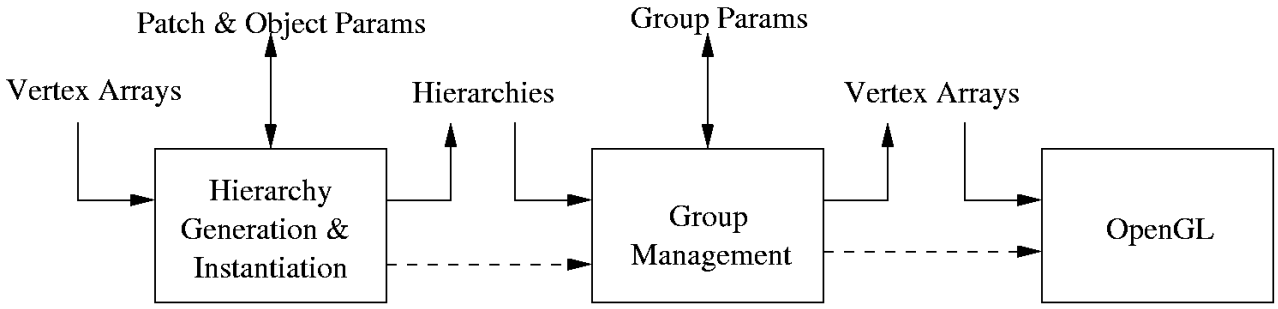

Figure 2: The GLOD object and dataflow model.

Although GLOD's Object construct provides enough information to adapt an individual object to a particular level of detail, significant performance gains can be made by amortizing adaptation costs across multiple objects. The user may also wish to specify an adaptation criterion, such as a triangle budget or a screenspace error threshold, that spans a group of objects. This necessitates a GLOD Group construct. A GLOD group is simply a collection of GLOD instances; just as a patch is the fundament rendering unit in GLOD, a group is the fundamental adaptation unit.

We support all of these approaches with little added complexity: Groups are limited to a flat namespace, as are objects. New objects (`glodNewObject`) can be added only once to one group. After the object has been created, the `glodInstanceObject` call is used to instantiate multiple objects into the same or other groups. Finally, since some rendering state parameters affect adaptation (for example, the `modelview` and `projection` matrices are used to determine the error of a LOD), relevant state is captured directly from OpenGL using a simple `bind` call for each object. This transformation is used only for the adaptation of the objects and not during rendering; the user is free as always to control the OpenGL state throughout the rendering process.

4.2 GLOD Dataflow

So far, we have presented the derivation of the GLOD object model, which strives to be lightweight and interfere minimally with standard OpenGL drawing methods. Next we present a model for bringing data into and out of GLOD in a way that allows GLOD interaction to take place with the same mechanisms that are ordinarily used by OpenGL.

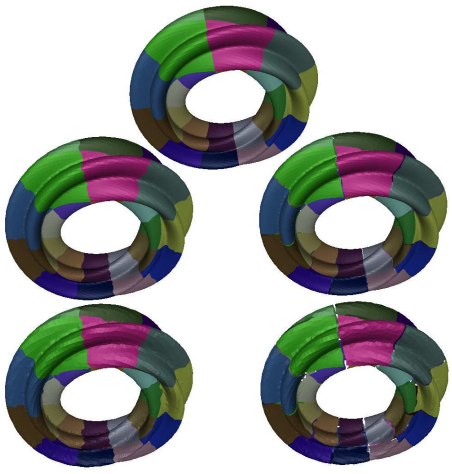

Figure 3: Torus model containing multiple patches. On the left, the patches have been simplified as a unified object; on the right, individually.

Our design choice for a fundamental I/O primitive is the Vertex Array. By following the OpenGL Vertex Array specification, we are able to cleanly capture all of the types of possible geometric output used by an application, starting with coordinates, but also normals, multiple sets of texture coordinates, and even vertex program parameters. In short, the use of Vertex Arrays allows us to maximize the compatibility of the library with the latest in graphics hardware trends with very little programming effort. At a high level, the GLOD system may be seen as three modules that receive and produce data, as we will show here.

The user specifies the input model patch-by-patch to the hierarchy builder using vertex arrays. This input data either goes to a simplifier or directly into a discrete hierarchy if the user selects a special DISCRETE_MANUAL mode. Inspired by the load interface for texture mip-map levels, this special build mode is a convenient way to get discrete levels of detail created by an artist or by another LOD system into GLOD to take advantage of its management and rendering features. After a group is adapted, the user can read back adapted patches using vertex arrays, and of course, such data can be specified in just the same format directly to OpenGL using the same vertex arrays.

In many cases, it may be useful for an application developer to read back an entire hierarchy from GLOD and push it back into GLOD at a later time. This is especially useful for avoiding the time to build the object every time your application starts. This ability is inspired by the OpenGL texture compression extension, which similarly allows direct read back of internal format data for later re-use.

Finally, GLOD also provides a standard OpenGL-style Set/Get interface for a large number of parameters to patches, objects, and groups. We use these to set build parameters (e.g. simplification operator, error metric, etc.), adaptation modes (e.g. error threshold or triangle budget), error thresholds, importance values, transition modes, and so on. Viewed collectively, all of these flow mechanisms allows flexible, and incremental uses of the GLOD system, and lends longevity to its specification through use of the vertex array interface.

5 USING THE GLOD API

We have implemented a basic version of the GLOD API in order to verify its completeness and usability for the breadth of LOD applications wherein it might be used. In the previous sections, we have described the core process of using GLOD – object creation, array insertion, building, and subsequent mixed adaptation and patch drawing. We have found that a number of additional features are essential as part of GLOD to maximize the system's performance and versatility.

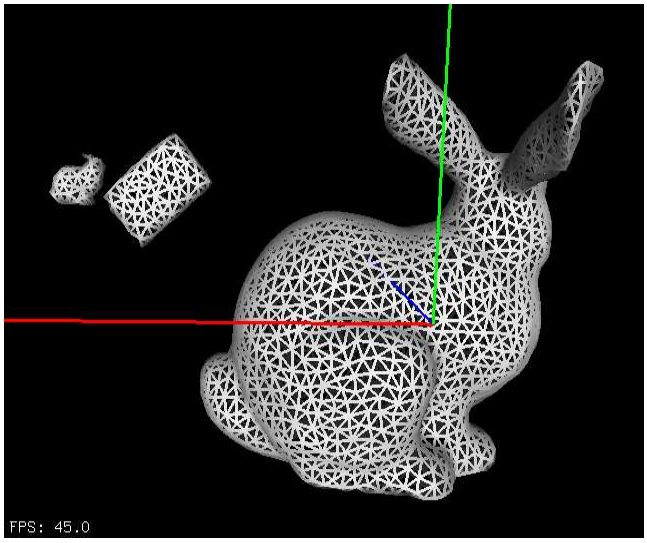

Figure 4: Several objects and instances rendered using GLOD with GLOD_CONTINUOUS format.

Two generic formats for creation of objects, DISCRETE and CONTINUOUS, are exported by GLOD, ,but may be mapped into internal formats in an implementation-specific way. Our particular implementation maps discrete LODs to vertex arrays directly, but defers view dependent rendering to VDS [Luebke and Erickson 1997]. Beyond our implementation, however, any number of internal and external formats and structures might be supported.

The GLOD API uses a generic algorithm to support a wide variety of adaptation modes, which are set on a per group basis using the `glodGroupParameter[if]` interface. We support triangle budgets and error budgets in both screen and object space modes, but consciously avoid frame time guarantees on the basis that OpenGL provides no clear notion of frame times either. For completeness and efficiency's sake, we also allow the behavior of these generic modes to be controlled through additional parameters to maximize rendering memory use and fine tune individual object and patch

Priority hinting for Objects and Patches are provided through the object and patch get/set interfaces: group priorities at an object level enhance or decrease the importance of a given component of the scene beyond the threshold that would be predicted by standard refinement policies. One particularly interesting use of this is in user-directed view dependent LOD.

High performance graphics applications are well known for exploiting long graphics command queues to perform asynchronous processing while vertex arrays draw. There are well understood mechanisms to allow this in OpenGL extensions – chiefly, the NV_FENCE mechanism. In all implementations of `glodDrawPatch()`, such implementations are fully supported without any additional code on our part. However, other components of GLOD allow levels of Asynchrony: thread safety is easily guaranteed by the `glodBuildObject()` call, making parallel simplification trivially possible. However, efficient versions of view dependent refinement make it almost impossible to adapt and draw simultaneously. Our solution to this to avoid the issue entirely using requiring backoff, set by a group parameter, which forces an adaptation algorithm to stop within a hard time limit. This allow the user to soft-schedule the adaptation process if the need arises.

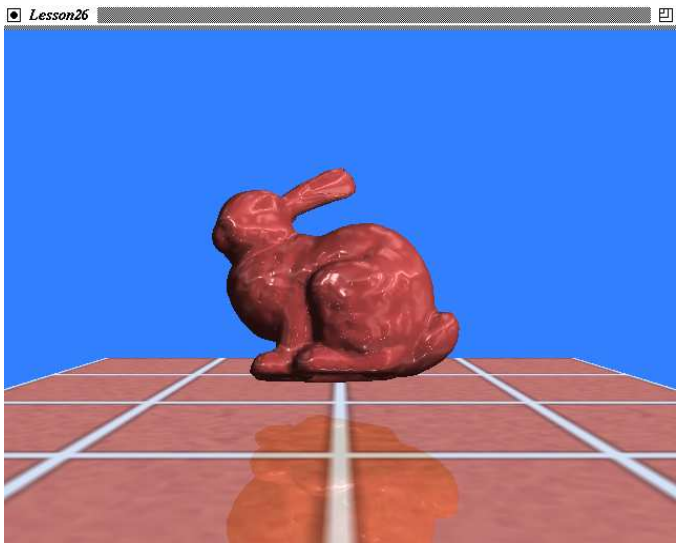

Figure 5: Bunny rendered in GLOD using a multipass rendering algorithm, demonstrating GLOD's policy of non-interference with the underlying graphics system.

6 DISCUSSION

The GLOD API that we have presented here exists not only on paper but also as an experimental system used to verify our design decisions so far, in the domains of discrete and continuous simplification, rendering, and management. This system, used for generating the images in this paper will ultimately become an open source system enabling a path 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

Humpreys, Greg, Mike Houston, Ren Ng, Randall Frank, Sean Ahearn, Peter Kirchner, and James Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. Proceedings of SIGGRAPH 2002.

Luebke, David and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. Proceedings of SIGGRAPH 97. pp. 199-208.

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

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

Woo, Mason, Jackie Neider, Tom Davis, and Shreiner. OpenGL Programming Guide. Addison Wesley 1999.