GLOD: A Minimal Interface for Geometric Level of Detail

(Online ID papers_0633)

Abstract

We present GLOD, a geometric level of detail system integrated with 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 evolution 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, scientific visualization, virtual environments, and gaming. The field of LOD has grown quite mature [Luebke et al. 2002]. 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 *adaptation*, 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.

In this paper we present GLOD, an open source 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 LOD generation algorithms, and multiple LOD management 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 can start to think of geometric level of detail as a fundamental component of the graphics geometry pipeline, much as mip-mapping is a fundamental component for controlling detail of texture images.

2 RELATED WORK

Existing level of detail tools generally fall into three categories: discrete mesh simplifiers, continuous and view-dependent systems, and scene graph toolkits. In this paper and throughout GLOD we use the terminology from Luebke et al. [Luebke et al. 2002]: *discrete LOD* refers to the creation of several static levels of detail which are swapped out directly for each other, *continuous LOD* creates a progressive data structure from which it can extract a continuous spectrum of detail at run-time, and *view-dependent LOD* extends continuous LOD by creating a hierarchical data structure from which it extracts a mesh tailored to the given viewpoint. GLOD supports all three LOD approaches.

Discrete mesh simplifiers, such as QSlim [Garland and Heckbert 1997], Simplification Envelopes [Cohen et al. 1996], and Jade [Ciampalini et al. 1997], address LOD generation but not LOD management. In other words, these tools take a complex object and generate simpler discrete LODs, but do not attempt to address run-time adaptation of those LODs to meet interactive goals such as a triangle budget or error threshold. In fact many simplifiers do not even provide error bounds for use during adaptation, meaning a developer or artist must decide manually at what distance the LOD is appropriate. Furthermore, programmers using a mesh simplifier must convert their model data to and from the tool's format; this can be problematic if the simplifier does not support all the attributes required by the programmer's models. Experimenting with different simplifiers typically requires even more converting back and forth between formats. One strength of GLOD is its use of OpenGL vertex arrays as a unified geometry interface—by design capable of expressing any attributes the user may wish to render—for input and output to many different simplification algorithms.

Continuous and view-dependent LOD systems do address LOD management to some degree, since they are capable of extracting and rendering meshes at run-time. Hoppe's Progressive Meshes (PM) algorithm [Hoppe 1996, Hoppe 1998] deserves special mention because it has been integrated into Microsoft's DirectX API, providing developers with some of the transparency benefits that GLOD offers. However, PM, like other continuous LOD algorithms, limits the user to specifying the number of vertices or triangles desired; it does not attempt to manage the level of detail to minimize error or satisfy a triangle budget, so the task of LOD management still falls to the developer. View-dependent systems allocate detail amongst different portions of an LOD, but still do not address the problem of allocating detail amongst different objects. Furthermore, some view-dependent algorithms, such as FastMesh [Pajarola 2001] and [Lindstrom 2003], do not support rendering to a budget, but only an overall error threshold. Also note that many view-dependent algorithms target only the specialized domain of terrain simplification [Lindstrom and Pascucci 2001][Duchaineau et al. 1997].

Scene graphs toolkits such as Open Inventor [Wernecke 1993], OpenGL Performer [Rohlf and Helman 1994], and OpenSG perform LOD management, but 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. They do not perform LOD generation, but some have external utilities such as OpenGL Optimizer to perform the generation. OpenGL Performer [Rohlf and Helman 1994], for example, is a powerful and well-written library for high-performance rendering. It has a rich set of LOD management and rendering options, such as blended transitions and feedback-guided frame rate control. It even incorporates support for a restricted form of view-dependent simplification. But 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.

Another large and powerful system, the Visualization Toolkit (VTK) [Schroeder et al. 1998], is not a scene graph but does provide support for both generation and management of level of detail. It provides multiple generation algorithms and a target frame rate mode based on measured rendering time for the various LODs. However, although this extensive toolkit allows easier incremental adoption than the scene graph systems, it still wraps all aspects of the underlying graphics API in a higher level abstraction, making it harder for the application to directly control and access the lower level rendering state. VTK is not intended to be a tool for the OpenGL developer, but a higher-level tool for rapid design of visualizations.

3 DESIGN GOALS

GLOD aims to cover the full gamut of LOD generation and management tasks, from discrete mesh simplification to run-time adaptation of discrete, continuous, and view-dependent LOD. However, GLOD follows an entirely different design philosophy from existing systems. The GLOD interface more closely resembles a driver-level extension of the graphics API than a high-level application toolkit. This distinction, which may at first seem subtle, leads to profoundly different usage patterns.

We have designed and implemented GLOD with the following goals in mind:

- **Functionality:** The system should support the many ways that geometric level of detail is used in production graphics applications. It should provide a rich set of options for LOD generation (including manual LOD generation by artists) and LOD management, and be flexible enough to handle a variety of application usage patterns.
- **Ease of adoption:** GLOD should be easy to use and easy to incorporate into existing applications. It should support incremental adoption so that applications retain as much control as they desire.
- **High performance:** GLOD should not sacrifice performance for flexibility. For every type of performance-critical LOD task, the API must support a way to achieve that task in a high-performance fashion.
- **Longevity:** Graphics hardware and associated APIs are continuously evolving. The GLOD API should support current development trends, such as increased programmability in the hardware, and avoid relying on features that seem likely to go away in future hardware.
- **Robustness:** The system should be able to work with arbitrary real-world models in the presence of real-world problems such as degenerate triangles, non-manifold topology, and coincident geometry. Note that it may also provide specialized modes for better performance on models with known properties, such as terrains or closed manifold meshes.
- **Extensibility:** GLOD should be easy for developers to extend by adding new general simplification algorithms, or special-case algorithms intended to handle particular situations (e.g., terrains).

4 DESIGN DECISIONS:

4.1 Use Minimal Structure

Minimizing the structural complexity of the underlying system helps keep the API clean, broadly applicable, and easy to adopt. The conventional design of a library supporting level of detail stores the complete *scene graph*, a hierarchy of objects organized by spatial and functional relationships. Objects are nodes in the hierarchy, each consisting of one of more connected polygonal meshes (often subnodes in the hierarchy). Different portions of each mesh may in turn be split into submeshes, or *patches*, that require different rendering states (for example, different textures). In this framework, LODs might be implemented as a series of sibling nodes at the object level, with some mechanism to select which LOD to traverse during rendering. An LOD system presented with this structure is aware of and has control over the entire rendered scene. Is such a hierarchical scene graph the minimal structure necessary to support level of detail? We argue that full LOD functionality can be achieved with less structural overhead. GLOD is very consciously *not* another scene graph system, and in fact does not even include a full-fledged notion of hierarchy.

What then are the minimum structural requirements for creating, managing, and rendering multiple multiresolution objects? GLOD uses three basic structures: *patches*, *objects*, and *groups*. A patch is a collection of geometry that shares a common rendering state. Patches are the fundamental units of rendering in GLOD: each patch is rendered—potentially simplified from its original form—in its entirety with a single call. An object is a collection of patches, which may be connected portions of the same mesh (i.e., sharing vertices). Objects are the fundamental unit of mesh simplification in GLOD: all patches within an object are simplified together to form a multiresolution hierarchy. Objects may be *instances*, which are duplicated versions of objects that are simplified individually. A group is a collection of objects, some of which may be instances. Groups are the fundamental unit of adaptation in GLOD: the objects in a group can be adapted with a common goal in mind, such as an overall triangle budget.

This structural design of patches, objects, and groups was motivated by consideration of several questions, which we revisit here as they provide insight to our design and design process.

Do we need patches? Clearly the user must be able to vary some aspects of rendering state, such as texture map or shader bindings, across objects. The patch structure is motivated primarily by the pragmatics of high performance rendering. The basic primitive for fast rendering on current hardware is a large indexed array of vertices. Any rendering engine intending to achieve high performance on modern graphics hardware must issue as many vertices as possible without changing graphics state, which incurs a call to the driver and a pipeline flush. In addition, the library should keep those vertices as "close" as possible to the hardware, storing them in fast-access AGP memory or even remotely in on-card video memory to maximize throughput to the GPU. Using patches to "clump" geometry into batches enables GLOD to use the coarse-grained dataflow required for high-performance, while allowing the user to vary rendering state across that geometry.

Do we need objects? Since the rendering state (e.g. active texture map) often varies between adjoining sections of a single mesh, meshes will often comprise multiple patches. Simplifying each patch individually would lead to cracks along their boundaries—a well-known problem in mesh simplification. Aggressive simplification algorithms (e.g. [Rossignac and Borrel 1992], [Garland and Heckbert 1997]) can even merge vertices from separate meshes within an object. We need a structure above the patch level that defines which patches and which vertices are to be simplified jointly.

Do we need groups? Patches and objects alone are not sufficient to implement global goals, such as adapting a set of objects to meet a total triangle budget. Adapting to a budget requires global knowledge, generally treating the cost and visual benefit (or error) associated with all levels of detail as input to an optimization process [Funkhouser and Sequin 1993]. The GLOD library has this knowledge, but the application typically does not. If budget adaptation were implemented at the application level, the

application would have to continuously query the library for each LOD's cost and benefit. This requires a ruinous number of API calls and places a considerable burden on the application programmer. Furthermore, certain optimizations, such as moving LOD geometry into fast on-card video memory, will be expensive or impossible if the application must make many calls to adapt each LOD each frame. So we chose to add another layer of abstraction—groups—to allow the user to specify simplification goals and budgets across multiple objects.

Do we need instances? Often an object must be rendered multiple times per frame with different transformations, for example during geometry instancing or rendering with shadow mapping. Clearly this re-rendering is a common and vital technique in interactive rendering, but must we explicitly address it in the GLOD API? In principle one could simply adapt the object and render its patches multiple times per frame, once per instance of the object. In practice, efficient rendering, adaptation, and memory use dictate explicit support for instancing. For example, adaptation of a continuous LOD often begins from the previous frame's LOD and makes only incremental changes, but repeatedly adapting the LOD to completely different transformations destroys this coherence and becomes much more expensive. Another option is to duplicate the object for each instance, but this can be prohibitively wasteful of memory. Thus it seems an instancing mechanism is a necessary feature of an LOD API. Instancing in GLOD provides a unique copy of the object's simplification state while sharing the same underlying geometry storage. In keeping with our minimalist approach to the API, we were careful to add this capability without making instances their own first-class entity: instances are simply objects created with glodInstanceObject() instead of glodBuildObject().

Do we need additional hierarchy? Objects can collect multiple patches and groups can collect multiple objects. We considered letting groups collect other groups, much as OpenGL display lists can call other display lists. This would give GLOD the expressive power of a scene graph, able to encode very general hierarchies of detail. However, it can also complicate the logic for adaptation, for example if the user-specified triangle budget for a group exceeds that of its parent. Nor is it clear that adding hierarchy provides additional optimization opportunities for efficient rendering. The major efficiency benefits of a scene-graph organization, such as view-frustum culling and sorting by rendering state to minimize state changes, may still be performed if appropriate, even by incorporating GLOD into an existing scene graph. On the other hand, the lack of enforcement of a predetermined GLOD scene graph structure on the application simplifies the adoption of GLOD or its integration into an existing application. Having decided consciously not to make GLOD into yet another scene graph engine, we affirmed that decision by avoiding the temptation to permit additional layers of hierarchy beyond the minimal requirements of patch, object, and group structures.

Can we render groups/objects? Since objects consist of patches and groups consist of objects, it would undoubtedly be convenient at times to render an entire object or an entire group with a single call. However, this fails our principle of minimalism. A hypothetical "**glodDrawObject**" or "**glodDrawGroup**" command could be implemented using the glodDrawPatch command, however, neither is strictly necessary and thus neither belongs in a minimal LOD library. In fact, the decision to expose LOD rendering at the individual patch level is an important distinction from scene graph libraries. It puts control of the entire rendering state in the hands of the application, and frees GLOD from tracking anything except the minimum required to issue geometry.

4.2 Follow the Red Book Road: The OpenGL Way

GLOD builds on, draws inspiration from, and coexists cleanly with OpenGL. This has several advantages: First, by building on 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.

The lightweight procedural interface of OpenGL suits our minimalist approach to LOD management better than a full-featured object-oriented system such as Performer or VTK. These are excellent toolkits, powerful and well-designed, but both are big, full-fledged systems best used "whole hog"; it is difficult to adopt Performer, for example, into an existing rendering engine without starting from scratch. Our system is inspired instead by the minimalist procedural design of OpenGL, and looks to various aspects of that design for motivation. For example, the application can specifying individual level of detail surfaces much as it might load up individual image resolutions in the OpenGL MIP-mapping interface. Similarly, the texture compression interface provides an example of preprocessing data: textures can be uploaded to OpenGL in a known format, processed by the library, and downloaded again in compressed binary form for later use.

GLOD, like GLU and GLUT, sits alongside OpenGL and often directly calls OpenGL. Important examples of GLOD interaction with OpenGL include:

- Reading vertex array pointers and state.
- Reading matrices and viewport for view-dependent LOD (see section 8).
- Caching geometry in GPU-side memory.

We have articulated a handful of design rules to guide our interaction with OpenGL, aiming to maximize the predictability and usability of GLOD given its closeness to the underlying graphics library:

- *Use existing OpenGL calls and state whenever possible*. For example, the application should not have to specify the camera twice, once to OpenGL and once to GLOD. Rather, it can ask GLOD at the appropriate time to grab its transformations from OpenGL. Similarly GLOD takes cues from the current client-state as to which vertex attribute arrays should be captured to generate an object's level of detail hierarchy.
- *Follow OpenGL calling conventions, data types, and standards*. The user should not have to learn new data types when using GLOD.
- *Minimize the number of calls and components in GLOD that will be perceived as "new" to OpenGL*. This rule is motivated by our goal that GLOD should be easy to adopt, a simple and straightforward extension.
- *Consider efficiency*. Minimize interaction with OpenGL; even reading GL state can be costly. Similarly, avoid GL constructs—such as the glPushAttrib/glPopAttrib commands—known to be expensive in practice.
- *Do not interfere with OpenGL state and semantics*. This is a key guiding principal of GLOD.

We expand on these principles and the resulting design decisions below.

4.2.1 Act Like OpenGL

We build on OpenGL programming constructs, such as types and calling conventions, whenever possible to provide a consistent and familiar interface for the programmer. For example, we push the majority of GLOD state control behind an OpenGL-style Get/Set parameter interface. This leads to an interesting design choice worth mentioning: unlike OpenGL, GLOD has no global state. Instead, each instance of the three standard GLOD primitives (patches, objects, and groups) has its own state controlled by get/set functions. (The lack of global state prevents us from putting various global controls on GLOD; for example, we cannot place a global memory budget on GLOD. However, such controls can still be enforced by the user.) We also employ a naming mechanism for objects, patches, and groups inspired by GL texture namespaces.

One especially noteworthy way that we leverage OpenGL is the use of vertex arrays as the geometry interface for GLOD. This is discussed further in Section 7.

Having determined to model GLOD on OpenGL, an interesting design decision arises: Should GLOD *become* OpenGL in some fashion? We considered three options: pretend to be OpenGL, propose an extension to OpenGL, or build a closely-coupled utility library tightly integrated with but separate from OpenGL.

Pretend to be OpenGL: By intercepting OpenGL function calls and masquerading as the graphics driver, it is possible to non-invasively instrument, redirect, or otherwise manipulate OpenGL programs [Humphreys 2002, Mohr and Gleicher 2002, Niederaur et al. 2003]. The earliest stages of GLOD used Chromium to intercept OpenGL commands and present the GLOD interface as if it was built directly into the graphics library (for example, allowing GLOD-specific parameters to be manipulated with the **glGet/Set** interface). We felt that pretending to be the driver enforced the low-level close-to-the-hardware philosophy we desired: OpenGL does know anything about high-level organization of the scene that it is rendering. Operating at the driver level also has its conveniences; for example, it is a simple matter for a driver-level API to track the various matrices and rendering state that GLOD uses. Ultimately, however, we felt that pretending to be part of OpenGL was too unwieldy and would be difficult for developers to adopt. Furthermore, the computationally intense tasks in GLOD (such as mesh simplification) are larger, slower operations than are likely to be included in a real driver; these computations belong in utility libraries and toolkits, not the driver itself.

Propose an extension: GLOD began as a draft proposal for an OpenGL extension. As an extension, GLOD would have all the benefits just listed. However, to convince a vendor that an extension is so valuable to be placed in the driver would require an extremely compelling demonstration of value, with an active user community clamoring for the new functionality. The best way to build such a community, it seemed, was to build a standalone utility library that established the value of a low-level geometric LOD toolkit.

Build a closely-coupled utility library: As the API evolved, we realized that we could encompass the low-level minimalist approach to LOD without needing to be inside the driver. Though lacking direct access to some useful information about rendering state, positioning GLOD as an external library vastly simplifies deployment and adoption by researchers and developers interested in trying it out. Ultimately if the graphics community embraces the concept of low-level geometric LOD, the ideas presented in this paper could move closer to the hardware and the base API in the form of vendor-specific extensions.

4.2.2 Don't Touch the Rendering State

Since GLOD allows and requires the user to set up the rendering state before drawing patches, GLOD should have no side effects on OpenGL state besides the issuing of geometry. During the design stage the temptation to let GLOD change the OpenGL rendering state was sometimes strong. For example, instantiated geometry will typically be transformed before being drawn, with a different transformation for each instance. A conventional LOD management system, built into a scene graph, will store and effect these transformations, saving the programmer the effort of explicitly performing the transformation. However, GLOD treats the GL state as sacred: no GLOD call should change the state as a side effect. Therefore, while GLOD allows binding a transformation to an object, that transformation is used only for adaptation. The actual change to the GL matrices before drawing the patches of that object remains the application's responsibility. Note that this also gives the user the freedom to separate adaptation and rendering transformation, for example to adapt the level of detail in a scene according to one camera position, then render the scene from another position—this is an important if occasional application of LOD that can arise e.g. during shadow mapping and occlusion culling.

In Section 4.1 we discussed how the hypothetical functions **glodDrawObject** and **glodDrawGroup** failed to satisfy the guiding principle of minimalism. These functions would also violate the key principle of noninterference, since the ability to draw a set of patches with a single call implies storing the rendering state of each patch and setting that state before rendering the patch.

4.3 Use an Implicit Camera

Most applications of level of detail need run-time knowledge about the scene. In particular, degrading the detail of objects further from the viewpoint would seem to require global scene knowledge about the camera and objects. Specifically, we require the perspective projection parameters (field of view, aspect ratio, and so on), the position and orientation of the camera, and the position and orientation of each object. How can we capture this knowledge without a scene graph?

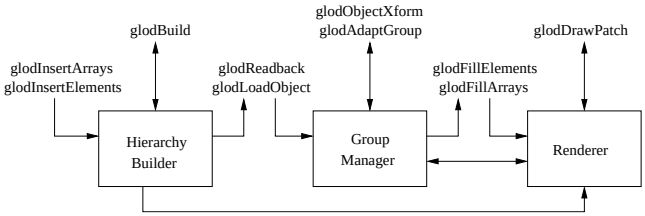
There are several options. First, we could deal with the camera by providing functions for the user to explicitly set camera position, rotation, field of view, etc. as global properties of the scene. This is the traditional approach, storing all camera information in a parametric representation. However, this would place limitations on the camera – for example, if GLOD only provided the parameters used by the `gluPerspective()` function, we would be unable to support an off-axis viewing frustum.

Another option is to track the OpenGL ModelView and Projection matrices and infer the camera parameters and camera-object transforms from these matrices. This implies an API function for the user to tell GLOD when to apply the current matrices to a given object or instance. An earlier version of GLOD did in fact use this approach, analyzing the matrices to calculate viewpoint, view vector, up vector, field of view, etc. However, this approach is still limited by the camera model built into GLOD and our ability to anticipate and reverse-engineer the different matrices a user might employ.

A better option, which we ultimately settled on, is to use a completely implicit model of the camera and camera-object relations. When the user binds the matrices to an object or instance, instead of inferring view parameters we can simply store the matrices with that object. When adapting the LOD groups, we can use these matrices directly, e.g. by transforming vertices from a bounding box into screenspace to estimate the object's size. This method is somewhat more expensive than, for example, estimating the solid angle subtended by a bounding sphere given

known camera parameters, but it is much more general. The screenspace size of objects can be estimated even in non-perspective situations, such as an isometric projection or a planar "fake shadow" matrix.

5 GLOD



The GLOD API is inspired by OpenGL and designed around the pipeline depicted in figure 1. Our system presents three modules: the hierarchy builder, the group manager, and the renderer. The API calls themselves can be separated into module-specific calls, data control calls, and setting calls. The benefit of this design is that it allows individual components of the system to be used autonomously – for example, using GLOD simply as a geometry simplifier with no management or rendering component.

The most important set of calls in GLOD is the parameter interface. These calls take the general form:

```
glod[Object|Group]Parameter[if](name,pname,value);
glodGet[Object|Group]Parameter[if]v(name,pname,
 *dat)
```

These calls for GLOD objects and groups allow you to fully customize the behavior of GLOD. As throughout the library, we consciously mimic the function names and conventions of OpenGL.

Building objects

```
glodNewObject(objname, format, grpname);
glodInsertArrays(objname, patchname, mode,
                 first, count, level, error);
glodInsertElements(objname, patchname,
                   mode, count, type, indices,
                   level, error);
glodBuildObject(objname);
glodInstanceObject(objname, instname, grpname);
```

Building a GLOD object is simple: you call `glodNewObject` to specifying a hierarchy type, a name, and a group. You then insert patches into the object by configuring OpenGL vertex arrays in the standard way and then calling `glodInsertArrays` as if you were calling `glDrawArrays` to render the patches. When all the patches are inserted into the object, you call `glodBuildObject` to turn the geometry into a multiresolution hierarchy. In effect, you are able to convert your fixed-resolution `DrawArrays` call into a multiresolution `DrawArrays` call in a few short steps.

Our present implementation of GLOD can build discrete, continuous, and view-dependent hierarchies. All of our LOD generation is currently done using the XBS library (Section 7). Rendering of discrete LODs is done using an custom module, while runtime adaptation and rendering of continuous and view-dependent LOD is provided using the VDSlib library (Section 8). Using the parameter interface above, one can configure the builder, for example, to use different error metrics for different objects, or to change between full and half-edge collapses.

Managing Groups of Objects:

```
glodBindObjectXform(objname);
glodObjectXform(objname, float[16] m1,
                float[16] m2, float[16] m3);
glodAdaptGroup(grpname);
```

Once an object is built, you must adapt it to a particular level of detail by calling `glodAdaptGroup` to a particular chosen goal. If you have chosen a screen-space-aware adaptation goal, then you must tell GLOD about the object and your camera. This is done either by specifying (up to) three transformation matrices using `glodObjectXform`, or by telling GLOD to automatically obtain the GL matrices using `glodBindObjectXform`. GLOD supports adaptation to different error measures (geometric, normal, even color- or texture-based in principle) in both screen space and object space. It also supports both screen-space and object-space triangle budgets. Finally, various limits can be set on adaptation to prevent GLOD from consuming too much bandwidth and CPU during large changes in the cut.

Rendering

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

Preparing an object to be drawn represents most of the coding necessary to use GLOD. Once a patch has been adapted as part of a group, glodDrawPatch will actually issue its (simplified) geometry to draw it to the screen. The behavior of this call is controlled by the OpenGL vertex array interface: to prevent GLOD from issuing certain attributes (normals, texture coordinates, etc.0 from issuing, disable the corresponding OpenGL array states.

Data Flow

```
glodReadbackObject(objname, *data);
glodLoadObject(objname, *data);
glodFillArrays(objname, patchname, first);
glodFillElements(objname,     patchname,     type,
 *elements);
```

Simplification can be a time consuming process best performed offline, rather than for example during game startup. To do this, you first determine the size of the object's readback buffer using the `glodGetObjectParameter` interface. You then allocate a buffer and pass that to `glodReadbackObject`. Loading of an object reverses this process. Readback is supported on all hierarchy types.

Similarly, sometimes we may not want to render a patch, but rather read its adapted geometry back into our application. Consider, for example, the case of a command-line simplifier. You do this by building and adapting an object to some desired level and then determining its triangle count using the `glodGetObjectParameter` interface. After allocating and binding the OpenGL vertex arrays to appropriate buffers, you call `glodFillArrays` or `glodFillElements`, which copies the current geometry into the currently bound vertex array.

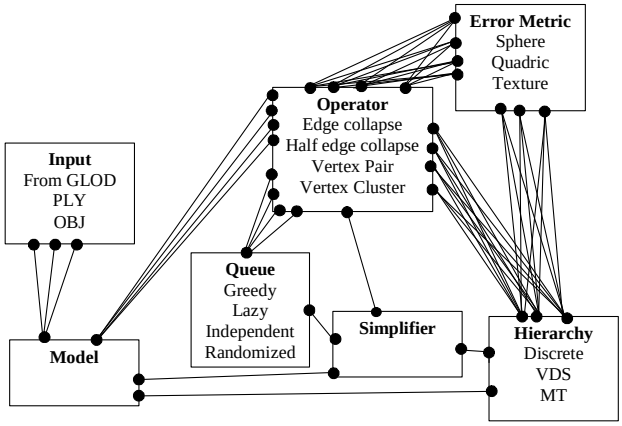
Putting it All Together

Below we show a simple usage of GLOD: render a 10,000-triangle version of some model using view-dependent rendering. First, you set up your object:

```
glodNewObject(0, 1, GLOD_CONTINUOUS);
for(each patch in our object) {
  ... bind the vertex array for this patch ...
  glodInsertArrays(0, patch, GL_TRIANGLES, 0,
                   num_verts, 0, 0);
}
glodBuildObject();
glodGroupParameteri(1, GLOD_ADAPT_MODE,
                    GLOD_TRIANGLE_BUDGET);
glodGroupParameteri(1, GLOD_ERROR_MODE,
                    GLOD_SCREEN_SPACE_ERROR);
glodGroupParameteri(1,       GLOD_MAX_TRIANGLES,
                    10000);
```

Now, in your render loop:

```
// get your camera & modelview matrix just
// right
```




```
glodBindObjectXform(0,   GL_PROJECTION_MATRIX   |
                         GL_MODELVIEW_MATRIX);
glodAdaptGroup(1);

for(each patch in the object) {
  // bind texture & whatever else
  glodDrawPatch(0, patch);
}
```

Of course, many variations on this theme are possible. As discussed in Section 4.2, GLOD operates seamlessly with various advanced GL features ranging from fragment programs and custom vertex attributes to normal maps. This is largely because of our policy of not interfering with OpenGL state. Of course, full support of OpenGL sometimes has a caveat: geometric LOD is not meaningful for certain vertex programs, for instance.

6 IMPLEMENTATION ISSUES

From an implementation standpoint, the building, managing, and rendering modules are abstract concepts within the procedural GLOD API rather than being actual pieces of code. Internally, we map raw API calls on a per-object basis to routines linked in from other libraries. In effect, our GLOD implementation is a (large) body of routines that wraps pre-existing LOD libraries behind a single coherent interface. Because we have tried to build the GLOD API to meet the requirements of a wide variety of LOD algorithms, we must be able to wrap a large number of different LOD libraries behind this single interface, thus unifying the current chaos of LOD APIs.

Despite the apparent simplicity of this architecture, wrapping libraries behind a GLOD interface is a non-trivial effort. Each LOD library has a different programming model, which must be reconciled with the GLOD model. An additional complication is that most LOD systems are application specific and support only a subset of GLOD's capabilities. Similarly, few LOD implementations respect rendering state, and must be modified to do so.

The initial version of GLOD, which has been available in prereleases for six months already, tries to provide a broad but fast feature set that is reasonably performance-competitive. In the following sections, we discuss some important implementation issues we faced in order to make this system operable and efficient.

6.1 Managing Heterogeneous Groups

A major challenge we faced in making GLOD function coherently was providing a flexible mechanism for group adaptation. This process is difficult because we do not want to restrict GLOD to a particular type of error metric. Consider the design space in which we must work: vertices can have any number of different attributes, each of which might have a particular error associated with it. The relative priority of an object compared to other objects in the group might be a weighted function of any of these parameters. Furthermore, because multiple types of hierarchies can exist in a single group, we must make sure that the different hierarchies have a consistent computation for each type of error.

The GLOD get/set interface is sufficient, we believe, to support all of these possible settings. Error metric and error weighting can be set on an per-attribute, per-object basis, or globally on a per-group basis.

Another challenge is providing group management over a heterogeneous set of hierarchy types. For example, a group may contain discrete, continuous, and view-dependent hierarchies, all of which may require adaptation to meet a triangle budget. Toward this end, we have extended the traditional dual-queue optimization algorithm with an additional layer of abstraction[Duchaineau et al. 1997][Funkhouser and Sequin 1993]. The queues actually contain a set of object hierarchies, each of which may be asked to coarsen or refine itself, according to its place on the queues. These requests contain limits as well, informing the hierarchy how much adaptation it can perform before its turn should be relinquished to another hierarchy on the queue.

6.2 Memory Management

Efficient rendering of multiresolution data requires caching the current cut in AGP or video memory. While there are a number of interfaces that exist for this purpose, they are tuned for use by the application user. We should be reluctant to use these extensions, due to our stated design goal of not interfering with OpenGL state. However we felt that the conflicting goal of efficient rendering demanded an exception in this case. In order to minimize interference with OpenGL, we must provide a mechanism to control how much video memory is used by GLOD, how that memory is shared between objects within GLOD, and how much data is transferred over AGP during adaptation.

We experimented with one way to provide these controls: let the user provide to GLOD a pre-allocated "fast memory" buffer, for example from the NVIDIA VAR extension. From that point, GLOD would internally allocate the buffer amongst the rendered objects. Unfortunately, this level of memory allocation and management is complex to implement, and other fast-memory schemes (namely the ARB VBO extension) provide locking and unlocking mechanisms that allow the mapped region to relocate, making direct memory management infeasible. Even if the user provided a vertex buffer object to GLOD which we then mapped ourselves, we would still be left with the ongoing issue of managing the buffer for ourselves –a messy problem with many complicated constraints.

The emergence of the VBO extension, while incompatible with our initial memory-management approach, itself provides a cleaner solution by allowing GLOD to use vertex buffer objects internally. Because VBO serves, in part, as a paging system for the working set of geometry being used by the video card, we can defer both the overall buffer management and the per-object-buffer-management problems to VBO and the graphics driver. The issue of AGP flow to the graphics card during adaptation is controlled with a simple per-group limit on the adapt call.

7 CROSSBAR SIMPLIFIER

We now turn to the crossbar simplifier (XBS), which serves as the LOD generation component of GLOD. Our design goals of functionality and extensibility demand a new and extremely flexible approach to geometric simplification. XBS provides a rich set of options for simplification and can be used as a stand-alone simplifier as well as the unifying component of the GLOD system. This simplifier is responsible for converting raw geometry received from the vertex array-based API calls into one of a number of hierarchy types, using one of several types of operators

and error metrics, driven by one of several types of simplification queues. By choosing among the modes with simple parameter switches, the application developer can affect the speed and quality of the simplification process, as well as the quality and management properties of the output. XBS provides a uniquely flexible architecture for mesh simplification; in this section we discuss the design and implementation to achieve this level of flexibility.

7.1 Algorithm and Component Overview

In the context of GLOD, the input to XBS comes from the GLOD API in the form of several indexed (but possibly unshared) vertex arrays, one for each patch of the object to be built. In stand-alone mode, XBS can read these data from a standard 3D file format such as PLY or OBJ.

Because the input data may not have its vertices fully shared, the Model performs an initial vertex sharing to merge vertices with geometric coordinates that are the same to within some specified tolerance. When vertices have similar geometric coordinates but different attribute values, they are linked together as coincident vertices but not explicitly merged. This is similar to Hoppe's construction of *wedges* [Hoppe 1998], but here we eliminate the need for auxiliary structures by maintaining them as fully-represented, independent vertices. In addition to keeping our representation simple, this choice is also consistent with the expected representation within the hierarchies we produce, which will generally store vertices in vertex arrays. If the Operators support the maintenance of these *multi-attribute vertices*, then the simplification process will maintain the connectivity of these vertices while allowing triangles with different attributes at the same vertex. After sharing, the vertices are *indexed* by storing with them the ids of their adjacent triangles.

After the input data has been processed in the Model, the Simplification Queue initialized according to the type of Operator and the Error Metric. The Queue may employ one of several different policies to issue operations to be applied by the Simplifier, including the Greedy, Lazy, and Independent policies [Luebke et al. 2003]. For example, the goal of the Lazy policy is to reduce the number of times the Error is needlessly recomputed for each simplification operation as a result of applying neighboring operations. In the Greedy scheme, the Error metric is recomputed immediately for all added or modified Operators, whereas the Lazy scheme simply marks them as *dirty*. These dirty Operators only have their costs recomputed when they return to the top of the priority queue, resulting in fewer extraneous cost computations. For more computationally expensive error metrics, we have also developed the Randomized queuing policy. The goal of the Randomized policy is to allow the user to specify exactly how many error evaluations will be performed per operation that is actually applied. The user gives a desired ratio, *R*, of cost computations performed to operations applied. The algorithm randomly chooses a maximal set of independent (non-overlapping) operations and computes their costs. If there are K operations in this maximal set, we apply K/R operations. This approach trades simplification quality for simplification speed.

The Error Metrics may be take many forms. For example the sphere metric is a simple accumulation of bounding spheres enclosing all the vertices that have merged to form the current vertex. This is extremely fast and fairly memory efficient, but is often extremely conservative. The quadric error metric [Garland and Heckbert] can produce higher-quality results, but can be slightly slower and does not produce a guaranteed error bound (although if all of our objects use this same metric, their relative errors may be compared). The texture error is appropriate for textured meshes, but is somewhat slower and may not work well for the most general triangle inputs.

As each operation is applied by the Simplifier, the Model and the Hierarchy are updated by the Operator. The Operator then informs the Queue of which of its neighboring Operators to add, delete, or modify. The Simplifier continues to receive Operators from the Queue and apply them until the Queue is empty. It then tells the Hierarchy to finalize its output and is done.

7.2 Component Interactions

The above discussion presents what amounts to the standard bottom-up simplification algorithm as seen from the point of view of component objects. The procedure relationships among the components are depicted in Figure 2. We can see three fan relationships (Model-Input, Model-Operator, and Queue-Operator). These indicate, for example, that the Model component needs to have one piece of code for each type of Input component and for each type of Operator component. Similarly, each type of Operator component must have specific code that deals with the base type of Queue (but not with each specific type of Queue).

Whereas the inter-relationships of the Input, Model, Simplifier, and Queue components are well-behaved, we can clearly see three crossbar relationships among the Operator, Error Metric, and Hierarchy components. For example, to add a new Operator to our system, we must implement a routine to interact with Model, a routine to interact with Simplifier, a routine to interact with Queue, and routines to interact with each specific type of Error Metric and Hierarchy.

These crossbar relationships are problematic for scaling to large numbers of specific types, but several observations can help us reduce this burden. First, several of the Error Metrics may agree on a type of output value, such as geometric object-space deviation. This can effectively reduce the crossbar relationship between Error Metric and Hierarchy to a fan relationship. Second, we can observe that many Operator types are of the same class. Two of those listed are vertex merges which produce new output vertices, and two are vertex merges that retain one of the input vertices. This can provide a good deal of code sharing within each of the remaining crossbars. Finally, we can always add components without supporting the full Cartesian product of component combinations. Such cases can report an error and revert to some default component.

8 EFFICIENT VIEW-DEPENDENT LOD

Inherent in the design of GLOD is the need for efficiency in both simplification and rendering. This goal is especially challenging for view-dependent LOD, particularly with regard to rendering. Indeed, view-dependent simplification (VDS) has become synonymous with slow rendering in the minds of most graphics practitioners: clever but not very practical. Conventional wisdom holds that VDS is appropriate only for very large continuous surfaces, such as terrains or scanned datasets. The reason behind this perception is a mismatch with current graphics hardware: the typical approach to view-dependent simplification maintains the active representation of a mesh as a linked list of primitives to facilitate incremental adaptation [Luebke and Erikson 1997][Hoppe 1997][Pajarola 2001], or continuously regenerates the simplification from the underlying vertex hierarchy [Duchaineau et al. 1997][Lindstrom and Pascucci 2001]. In both cases the geometric data is stored in main memory and issued to the graphics hardware in immediate mode, limited by CPU load, driver function call overhead, and the efficiency of the bus carrying the data to the GPU.

Nonetheless, we believe view-dependent simplification represents a core element of LOD; VDS offers superior fidelity to discrete LOD for a given triangle count and is mandatory for certain datasets, such as terrains and very large unorganized isosurfaces. As a primary contribution of this paper, we present a

new method for view-dependent mesh simplification that supports drastically more efficient rendering. Our technique is encapsulated in the standalone *VDSlib* library, which has been co-developed and co-designed with GLOD over the past year.

8.1 More Efficient Rendering

The key concept behind VDSlib's efficient rendering is to cache simplification results in coherent vertex arrays. Indexed vertex arrays are the highest-performance rendering path in modern graphics cards. They provide excellent issue efficiency, exploit the post-transform vertex cache, and allow data to change without requiring additional setup costs. Caching the adapted mesh in a coherent array enables the use of vertex array rendering. The arrays are effectively write-only buffers, which means we can even store them in AGP or video memory for still faster rendering. We use the Vertex Array Range (NVIDIA), Vertex Array Object (ATI), or, most recently, the Vertex Buffer Object ARB OpenGL extensions for this purpose. We map this buffer directly into the client address space and write updates directly to it as the mesh is adapted. We have found that this usage pattern is not significantly slower than writing updates to a buffer in system memory. We also organize this buffer as a set of interleaved arrays to further increase write performance.

Another long-standing technique for increasing rendering efficiency is to create triangle strips from the mesh to be rendered. Even taking into account the per-frame cost of building the strips as the underlying mesh changes, Hoppe and more recently [Shafae and Pajarola 2003] have shown that triangle stripping can improve rendering performance of view-dependent LOD. However, since we use indexed vertex arrays, the gains from triangle stripping—a reduction in the number of indices sent to the GPU, rather than in the number of vertices—are relatively small. Thus in our current implementation we do not currently employ this optimization, but it seems to be an obvious next step.

8.2 The VDSlib Hierarchy

VDSlib uses a generalized vertex hierarchy which supports many different simplification methods, such as half edge-collapses, full edge-collapses, and n-way merges of an arbitrary number of vertices. This hierarchy has the structure of a tree, with the vertices of an original model making up the leaf nodes at the bottom of the tree. Nodes above the leaf nodes are created by simplification operations collapsing two or more nodes into a single new "parent" node. An active representation of the simplified model comprises a path, or *cut*, across this vertex hierarchy, where the nodes on the path make up the vertices of the representation. At any point on this cut, a node may be split apart, or *unfolded*, into its child nodes, increasing detail in that node's region of the model; alternatively, if all of the children of a node are on the cut, they can be collapsed together, or *folded*, into their parent node, reducing detail in that node's region of the model. Thus, by continually folding and unfolding nodes in the appropriate regions of the cut, we can adjust the representation to the current viewing parameters, accomplishing view-dependent level of detail.

VDSlib's hierarchy of nodes is called a *forest*. This structure represents an object that is using continuous level of detail; a cut in VDSlib represents an instance of an object at a particular simplification state. A GLOD object instance is divided into patches, which are rendered independently of each other; likewise, a VDSlib cut's triangles are divided into patches, which are rendered separately. Because more than one instance of an object can be created, data specific to a cut must not be in the forest hierarchy. The hierarchy and its store of geometric data are not modified at runtime - all data that change as simplification takes place are associated with each cut. More than one cut can thus be associated with each forest, efficiently implementing compatibility with GLOD's capacity for instancing.

8.3 Simplification Procedure

The typical approach to view-dependent simplification is threshold based: each frame, a process traverses the front of active primitives, for each one calculating a view-dependent error[1] and locally refining the primitive if this error is above some threshold value. To support simplification to a triangle budget, however, we use a dual-queue system similar to that of ROAM [Duchaineau et al. 1997]. We maintain the *active nodes*--all nodes that are eligible to be unfolded--in a priority queue, sorted on their error; we call this queue the *unfold queue*. We also maintain a priority queue, sorted on node error, of all nodes that are directly above the active cut; these are all nodes that are eligible to be folded, and this queue is called the *fold queue*. To unfold a node, we choose the node on the top of the unfold queue; of all nodes eligible for unfolding, this node is contributing the most error to the model representation. Similarly, of all the nodes eligible for folding, the top of the fold queue will introduce the least error into the representation when folded. Simplification to a triangle budget is as simple as folding or unfolding until the target triangle count is reached. Threshold simplification is supported by folding or unfolding until the target error threshold is reached. If we wish to simplify a group of objects to a triangle budget or an error threshold, we maintain only a single unfold queue and a single fold queue for the entire group; nodes from each of the objects reside in each queue. In this way, we can simplify to the target while balancing the error of each object in the group against the errors of the other objects. We can even support simplification with both a triangle budget and an error threshold target, where simplification terminates when either of the targets is reached. This mode is particularly useful when GLOD is simplifying a group containing both discrete and continuous LODs to a budget (described further in section 6.1).

Whenever a node becomes active, we cache the geometric information for the vertex it represents in a vertex array and keep track of its index or *ref* in the vertex array. When that node is deactivated, we record its ref in a free list, denoting that that slot in the vertex array is not being used. We do not need to modify the vertex array at this time, because no triangles are going to reference the inactive vertex. When activating vertices, we first check the free list to see if any unused slots are available in the used section of the vertex array, and if not we simply add the vertex onto the end.

8.4 Support for Patches

Because objects can comprise multiple patches that use different rendering state and have replicated vertices at their boundaries, VDSlib must support simplification of these patches without introducing cracks or overlap. When the forest hierarchy is being built, the nodes corresponding to shared vertices are tagged as *coincident* nodes--nodes which share a location, but differ in one or more other characteristics (such as texture coordinates or normals). When boundary nodes of one patch are merged together, their coincident nodes in the adjacent patch are also merged together, and the two parents are made coincident with each other. During simplification, a fold or unfold of a node *N*

[1] This error can also take into account whether the primitive is in the view frustum; in GLOD, primitives found to be outside the view frustum are assigned an error of zero, meaning that they can be drastically simplified without affecting the error of the rendered scene.

immediately triggers the same operation on any nodes with which *N* is coincident. This simple mechanism synchronizes the simplification of adjacent patches. The exception to the rule of coincident child nodes having coincident parents is when all remaining nodes of a patch are being merged together; in this case, their parent node is made to belong to an adjacent patch, and is not coincident with any other node from that patch. This allows an entire patch to be simplified away and later reintroduced, a property of great benefit when a complex multi-patch model is far from the viewpoint or otherwise requires a very coarse LOD.

8.5 Interruptible Simplification

Because each simplification operation on a node in the fold or unfold queues (or set of simplification operations on a set of coincident nodes) is independent of any others in that queue, we can interrupt the simplification process before it has reached its target budget or threshold and still be guaranteed a valid representation to render. This is useful in cases where an abrupt camera movement or sudden change in adaptation target results in a large volume of simplification needing to take place. If this entire volume of simplification is performed in a single adapt call, it may cause a noticeable pause in the application. Instead, we put a limit on the maximum number of simplification operations that can be performed during an adapt call. This will spread the volume of simplification out over a number of frames, allowing the application to maintain a reasonable framerate over this period. The simplification may t>ake longer to complete than if done all in one frame, but assures interactivity throughout.

9 PLANNED EXTENSIONS TO GLOD

Not all LoD problems can be solved in GLOD. Sometimes, this is a limitation of our implementation --- for example, terrain rendering or out-of-core LoD. Other problems, it seems, cannot be easily solved within GLOD at all: these problems are usually those that require more sophisticated access to OpenGL state. Here, we discuss the issues in extending GLOD's functionality to these domains.

Terrains (and other regular geometry) are fully supported in the present implementation of GLOD. However, the regular nature of the input geometry allows many optimizations that make its rendering far more efficient than unorganized meshes. As a result, it makes sense for GLOD to include a rendering module for regular geometry. As can be seen in the implementation section, this presents no challenge for us as GLOD programmers: GLOD is designed for the addition of new external modules, as was shown by our support of the VDS module. However, the real challenge is wedging regular geometry into the GLOD framework: vertex arrays are not necessarily the most efficient way to represent terrain --- heightfields, for example, are more natural ways to store this geometry. Rather than changing the interface to GLOD, however, we believe that the way to address this is by setting state on the object after it is created but before the patches are inserted: for example, the user might be required to set GLOD_GRID_WIDTH/HEIGHT before calling glodInsertArrays. Beyond this sort of technical discussion, there should be no real technical limitation to adding terrains to GLOD.

Out-of-core rendering, while a giant problem in itself, should conceptually be possible within the GLOD framework. The main reason here is that the out-of-core process still consists (largely) of input data, a pre-processing step which we would bind to adaptation, and a real-time rendering phase. The real challenge in implementing this in GLOD is that this problem requires file-I/O which does not map well to the GLOD interface. The logical way resolve this problem is either using memory mapping or file pointers. The first option, of course, still limits the problem to the computer's addressable memory space. A more scalable solution is to avoid the vertex array interface entirely, using GLOD the get/set interface to pass file-pointers in and out of GLOD. Another challenge with implementing out-of-core problems in GLOD is that vertex programs, which are being used to accelerate transfer of geometry to the graphics card, cannot be used by the renderer because GLOD guarantees that it will not touch OpenGL state.

Many specialized LoD problems these days want to affect OpenGL state. For example, texture impostors and image-based LoD need to load textures into OpenGL for their rendering phase. GLOD guarantees that its main routines will not affect OpenGL state. We can, in order to support more advanced LoD features, loosen this requirement for special hierarchy types under the following restriction: the user should explicitly enable some GLOD option (on a per-object basis using the get/set interface) that might, for example, cause well-understood state changes to take place as a result. The controlling idea here is that basic GLOD will still "not affect" OpenGL state unless directly told to do so by the user.

A number of non-geometric uses of GLOD might eventually become possible with further implementation. For example, the builder interface of GLOD might ultimately be fitted with an optimizer that would allow dynamic optimization of vertex arrays. This same module might be used internally by GLOD as well, or externally using the readback mechanism by applications. This sort of application, too, fits well into the GLOD interface.

10 FUTURE WORK

GLOD is designed—both as a system and as an API—to support hierarchy types beyond the basic ones that we have implemented thus far. Bringing some of these hierarchies and algorithms into GLOD is a matter of programming rather than technical limitation. However, some advanced approaches to LOD present real challenges because they come in conflict with our basic API design and rules. Here, we discuss how GLOD can be made to work in these domains without changing the API.

Terrains (and other regular geometry) are supported in the present implementation of GLOD using the catch-all XBS simplifier. The main challenge in efficiently inserting terrains into GLOD is that vertex arrays clearly provide more data than necessary to specify a terrain. We could add a new call for inserting terrains, but better might be to address this with a combination of parameter settings. For example, if the user inserts regular-grid geometry, they could set the GRID_WIDTH/HEIGHT object parameters before calling glodInsertArrays(name, patch, GLOD_GRID,...). This sort of scheme could be used to insert all sorts of regular geometry formats into GLOD.

Out-of-core rendering, while a difficult problem in itself, should not present major conceptual hurdles within the GLOD framework. The out-of-core process still matches the insert, build, adapt, draw process that we have outlined so far. The challenge in implementing such a system in GLOD is that the input and intermediary steps are file-specific stages in the system, and OpenGL does not deal with files. The logical way resolve this problem is either using memory mapping or file pointers. Neither completely solves the problem they limit us to the computer's address space. Furthermore, depending on the memory mapping policy of the operating system, GLOD might consume the entire available memory before pages begin getting kicked out of RAM again. A more scalable solution is to avoid the vertex array interface entirely, using GLOD the get/set interface to pass file-pointers in and out of GLOD.

Many specialized LoD problems these days want to read or affect OpenGL state. For example, texture impostors and image-based LoD need to load textures into OpenGL for their rendering

phase. We can, in order to support more advanced LoD features that exhibit this behavior, relax our state alteration guarantee: *in order for GLOD to alter OpenGL state, some parameter should be manually set by the user*. This guarantees that the unsuspecting user will never see state changes, but the advanced developer has the power necessary to cause them to happen.

Finally, we can extend the GLOD API into several fundamental new directions. If we ignore the rendering components of the GLOD API, it exports a very clean interface for geometry pre-processing and manipulation. This interface is easily adapted to a wide variety of algorithms ranging from vertex order optimization to parameterization.

11 CONCLUSION

The GLOD system that we have presented here is unique in several ways. GLOD goes to great lengths to present a low-level interface to LOD. This is good for GLOD users: adoption of LOD is easy, yet powerful and above all else, hardware oriented. Users do not have to reconcile the LoD system against their graphics driver and their program model. From the perspective of LOD researchers, our approach has many gains: GLOD is able to abstract the LOD pipeline without breaking hardware compatibility, and therefore provides an ideal model for new LOD libraries as well. The GLOD system can be extended to support new LOD algorithms, making it a good way to deploy LOD code. Finally, GLOD is an experiment in aligning LOD algorithms with the cutting edge in graphics hardware: can we balance abstraction against speed? GLOD, its API and its implementation, shows us how it can be done.

12 BIBLIOGRAPHY

OpenGL Optimizer Programmer's Guide: An Open API for Large Model Visualization. pp. 250 pages.

Ciampalini, A., P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error *The Visual Computer* vol. 13 (5). 1997 pp. 228-246

Cohen, Jonathan, Amitabh Varshney, Dinesh Manocha, Gregory Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification Envelopes. *Proceedings of SIGGRAPH 96.* pp. 119-128.

Duchaineau, M, M Wolinsky, D E Sigeti, M C Miller, C Aldrich, and M B Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. *Proceedings of Visualization '97.* pp. 81-88.

Funkhouser, T. A. and C. H. Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Proceedings of SIGGRAPH 93.* pp. 247-254.

Garland, Michael and Paul Heckbert. Surface Simplification Using Quadric Error Metrics. *Proceedings of SIGGRAPH 97.* pp. 209-216.

Hoppe, Hugues. Efficient Implementation of Progressive Meshes. *Computers & Graphics.* vol. 22(1). 1998. pp. 27-36.

Hoppe, Hugues. Progressive Meshes. *Proceedings of SIGGRAPH 96.* pp. 99-108.

Hoppe, Hugues. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH 97.* pp. 189-198.

Humphreys, et al. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters. *Proceedings of SIGGRAPH 2002.*

Lindstrom, P and V Pascucci. Visualization of Large Terrains Made Easy. *Proceedings of Visualization 2001.* pp. 363-370 and 574.

Lindstrom, Peter. Out-of-Core Construction and Visualization of Multiresolution Surfaces. *Proceedings of 2003 Symposium on Interactive 3D Graphics.* pp. 93-102.

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

Luebke, David, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics.* Morgan Kaufmann Publishers, San Francisco. 2002. 390 pages.

Luebke, David, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics.* Morgan Kaufmann Publishers, San Francisco. 2003. 390 pages.

Morhr, Alex and Michael Gleicher. Non-Invasive, Interactive, Stylized Rendering. *Proceedings of the 2003 Symposium on Interactive 3D Graphics. March 2001.*

Niederauer, Christopher, Mike Houston, Maneesh Agrawala, and Greg Humphreys. Non-Invasive Interactive Visualization of Dynamic Architectural Environments. *Proceedings of the 2003 Symposium on Interactive 3-D Graphics. pp. 55-58.*

Pajarola, Renato. FastMesh: Efficient View-Dependent Meshing. *Proceedings of Pacific Graphics 2001.* pp. 22-30.

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.

Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering Complex Scenes. Technical Report RC 17687-77951. IBM Research Division, T. J. Watson Research Center. Yorktown Heights, NY 10958. 1992.

Schroeder, Will, Ken Martin, and Bill Lorensen. *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics.* Prentice Hall 1998. 645 pages.

Shafae, Michael et al. DStrips: Dynamic Triangle Strips for Real-Time Mesh Simplification and Rendering. *Proceedings Pacific Graphics Conference, 2003.*

Wernecke, J. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor(TM), Release 2.* Addison-Wesley, Reading, MA. 1993.