A Relational Debugging Engine for the Graphics Pipeline

Nathaniel Duca Krzysztof Niski Jonathan Bilodeau Matthew Bolitho Yuan Chen Jonathan Cohen

Johns Hopkins University

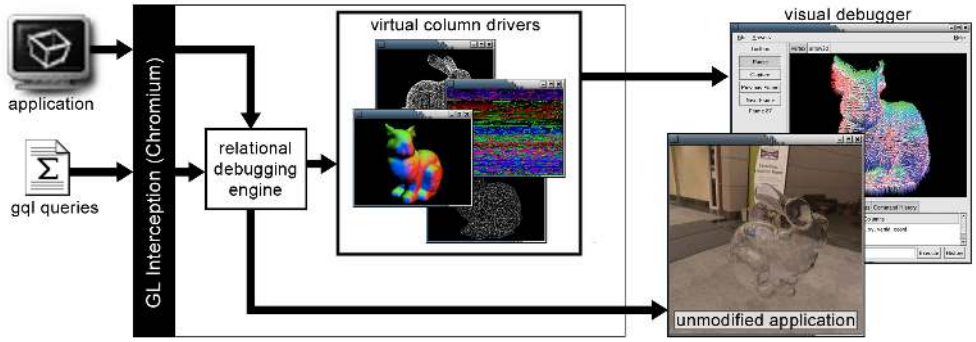

Figure 1: The graphics debugging engine presented in this paper makes it possible to capture, manipulate and visualize a wide range of data from the graphics pipeline without making changes to the underlying application.

Abstract

We present a new, unified approach to debugging graphics software. We propose a representation of all graphics state over the course of program execution as a relational database, and produce a query-based framework for extracting, manipulating, and visualizing data from all stages of the graphics pipeline. Using an SQL-based query language, the programmer can establish functional relationships among all the data, linking OpenGL state to primitives to vertices to fragments to pixels. Based on the Chromium library, our approach requires no modification to or recompilation of the program to be debugged, and forms a superset of many existing techniques for debugging graphics software.

CR Categories: I.3.4 [Computer Graphics]: Graphics Utilities—Support Software; I.3.6 [Computer Graphics]: Methodologies and Techniques—Languages; D.2.5 [Software Engineering]: Testing and Debugging—Debugging Aids; H.2.4 [Database Management]: Systems—Relational Databases; H.2.8 [Database Management]: Database Applications

Keywords: relational algebra, debugging, graphics hardware, graphics pipeline, visualization, SQL, streaming, SIMD

1 Introduction

The graphics pipeline embodied by today's 3D rendering platforms has a complex nature that defies classification. From a high level, it appears as a coarse-grained pipeline. Yet on closer examination, it seems at once both a streaming architecture and a SIMD architecture. Programming the various stages of this pipeline to produce some desired rendering output resembles the orchestration of a symphonic masterpiece, to be conducted at run-time by code on the CPU.

As daunting as writing such modern graphics software is, however, it pales in comparison to the task of *debugging* that software. The tried-and-true, standard debugging tools only give access to variables on the CPU, losing track of data once it passes into the graphics API. In general, tracking the data beyond that point requires the programmer to make extensive modifications to the code on the CPU as well as on the programmable vertex and fragment processors. Such debugging techniques are akin to the classic approach to debugging with print statements, except that in this case, it is much more difficult to extract the variables to be printed, and the data sizes are considerably larger.

Fortunately, there are several tools that can assist in this task. Some give access to the OpenGL state, including state variables, logging calls, reporting errors, etc. Others provide means of stepping through a fragment or vertex program. However, none of these tools provide a means of debugging the entire pipeline — the data are scattered across the pipeline with scant relations between.

In this paper, we present a new, comprehensive approach to debugging software on the graphics pipeline. All the data that exists throughout the graphics pipeline during program execution is represented in a set of virtual tables. We define a query language based on SQL to extract and instantiate the desired portions of these tables, then manipulate them using relational algebra. This provides a powerful mechanism for not only getting at the data, but for understanding the relationships among the data. For example, it is easy in this framework to establish the relationships of primitives to vertices to fragments to pixels, and one can trace the origins of particular pixels or examine the legacy of particular primitives. In some contexts, it is helpful to inspect the output of a query in the form of a raw table of numbers. In others we find that it is especially useful to employ visualization techniques to browse the data in a more intuitive fashion.

A key feature of our approach is that it does not require the programmer to rewrite any application code, but simply to run the

application using our dynamically-linked, OpenGL replacement library. Based on the infrastructure of the Chromium library, the debugger non-invasively intercepts the necessary portions of the OpenGL command stream. The main application context is guaranteed to render exactly as before. Meanwhile, a separate debugging context gathers the requested data, performs relational algebra operations, and ships the results off for visualization. Thus one can run the application at interactive rates, navigate to a location of interest for debugging, and initiate debugging for the graphics frames that follow. During debugging, the application window continues to render as expected, but the debugger's user interface also provides visualization of the queried data. One can debug a single frame of interest, or allow the application's main loop to keep running, updating the debugging visualization as it proceeds. The performance of this process while debugging queries are active is dependent on the complexity of the particular queries and their visualization tasks.

This paper makes a number of contributions to the state of the art in debugging graphics software. We define a mapping of data from all stages of the graphics pipeline to a set of data tables and a relational language for selecting from these data and establishing their interrelationships. This novel approach provides a rich environment in which to pose and solve debugging problems. Other graphics debugging solutions, such as OpenGL state access and generation of RGB images from fragment program data, are a subset of what is possible with our approach. In this setting, programmers ask themselves not *if* they can get at some particular data or relationship, but *how* to do it most effectively. Our language and debugger provide the necessary tools. We focus here primarily on the core debugging facilities, what we term the *debugging engine*. Given this comprehensive debugging engine, it is possible to build a variety of more high-level tools and implement a number of strategies for debugging graphics applications.

2 Related Work

Although trivial programs may be debugged purely by inspection, most debugging tasks benefit from the extraction of data from the program execution environment. In some cases, the programmer may modify the program to extract these data (such as adding `print` statements), but it is generally more convenient to use some more automated debugging environment.

Text-based debuggers, such as dbx [Linton 1990], gdb [Stallman 1989], and CodeView, provide the programmer with interactive access to program data during execution. For example, the programmer can step through code, examine variables, set breakpoints and conditional breakpoints, traverse the scopes on the current stack, etc. Similar debugging methodology has been employed since at least DEC's PDP-1 debugger, DDT [MIT 1961]. Graphical debuggers and integrated development environments (IDEs), such as xdbx, xxgdb, DDD [Zeller and Lütkehaus 1996], Saber-C [Kaufer et al. 1988], Visual Studio, XCode, etc., add graphical user interfaces, including widget-based user interaction, more visual representations of data structures and programs as graphs, as well as providing facilities for code development and compilation. These sorts of debuggers enjoy near-universal adoption, and are excellent tools for debugging a single thread of program execution.

In general, additional tool support is required to extend this model to parallel and distributed environments. Debuggers like pdbx, pgdbg, DDT (by Allinea Software), and TotalView extend this break/step/continue debugging model to MPI and OpenMP execution environments for distributed processes and multi-threaded applications. The programmer is given additional controls to manage these commands over several processes, and can apply the commands to an individual process or groups of processes. Even so, the task of managing and interpreting the data becomes increasingly complex with the addition of parallelism.

To deal with parallelism at massive scales, debugging environments such as IVE [Friedell et al. 1991] and Prism [Sistare et al. 1992] for Connection Machines and MPPE for the MasPar provide visualizations which map processors and their data to colored pixels or other small glyphs, such as arrows.

It is not surprising that such pixel-based debugging techniques are common on systems designed for computer graphics. As the common wisdom goes, "the best way to debug graphics is using graphics." Given a pixel-based shader program, a programmer redirects some scalar or vector quantity to the location reserved for the shader's output color (after scaling and biasing into the valid color range). A skilled observer can sometimes make inferences about the data by inspecting the resulting image, or can inspect individual values in an image viewer. In PixelFlow's pfman language, the compiler could instrument a fragment shader to dump an image for any single lvalue in the program. This was done by setting a uniform parameter to an instruction number, thus avoiding an expensive recompilation process [Olano 2005]. Image-based debugging of fragment shaders are similarly common for SGI's OpenGL Shader and for RenderMan [Stephenson 2000] (which can dump many variables at once because it is a pure software renderer).

There are a number of debuggers for fragment programs on today's PC graphics hardware with a range of capabilities [Purcell 2004]. Imdebug [Baxter 2002] provides printf-style statements that the programmer can add to code to bring up the shader output in an image window. Shadesmith [Purcell and Sen 2003] allows stepping through fragment programs to inspect individual variables as images. Apple's OpenGL Shader Builder allows debugging of individual shaders in a stand-alone fashion, but not in the context of the real application program. Microsoft's Shader Debugger Tool operates using a software rasterizer, providing access to vertex and fragment data for DirectX Programs. The visualization aspects of these tools are generally restricted to the fragment program portion of the graphics pipeline.

The ATI RenderMonkey and NVIDIA FX Composer tools' approach to shader development sidestep the issue of debugging by using a GUI to build a a scene out of a wide variety of basic graphics "building blocks". Because this approach allows shaders to be changed on the fly, the basic challenges of shader debugging are avoided.

The debuggers above can be supplemented with additional tools that instrument the OpenGL API calls. Tools like Microsoft's PiX [Microsoft 2005], gDEBugger [Graphics Remedy 2005], GLIntercept [Trebilco 2004], and GLSurveyor [Gould 2005] track and log all graphics API calls, report error conditions, and let the programmer look at the state associated with contexts while walking through the application.

While these debugging tools perform useful functions and can certainly increase the programmer's ability to investigate a variety of bugs that occur in graphics applications, the data they report are difficult to relate to one another. At the vertex and fragment level, they also tend to follow the myopic approach of standard debuggers, presenting data in a program-counter-oriented fashion. (For a qualitative comparison of tools, see Figure 2.)

There are a several alternative debugging concepts from outside the graphics domain that have some of the characteristics we are seeking. The first is the concept of *omniscient debugging* [Lewis 2003], which includes a debugger that captures all program state over the

Tool	Application Integration	State Debugging	Texture Debugging	V. Shader Debugging	F. Shader Debugging	Profiling	Postprocessing via
DirectX9/Microsoft PiX	Yes	Yes	Yes	Emulation	Emulation	Yes	GUI
NVPerf	Yes	No	No	No	No	Yes	None
PixelFlow	Yes	No	No	No	Yes	No	Image files
Apple GL Tools	Yes	Yes	Yes	No	No	Yes	GUI
gDEBugger(etc)	Yes	Yes	Varies	No	No	Yes	GUI
RenderMonkey/FX Composer	No	No	No	No	No	No	GUI and shader editor
This Paper	Yes	Yes	Yes	Yes	Yes	Yes	Graphics Query Language

Figure 2: There is considerable variety in currently available graphics development tools. Sometimes the tool integrates with unmodified applications. Most tools allow examination of textual state, e.g. viewport coordinates. A limited number of these tools also allow textures to be viewed. Line-by-line debugging of vertex and fragment shaders is becoming more common. Some tools, like RenderMonkey and FX Composer, avoid the debugging paradigm entirely by allowing on-the-fly reloading of shader programs. Performance profiling is the most universally supported feature of graphics tools. However, the method used to study the data obtained by all of these tools is universally limited: some tools to perform analysis using a GUI while others output text or image files.

entire program execution. This idea is primarily used to allow the programmer to move backward as well as forward in time over the program execution during the debugging process. Another useful concept is *query-based debugging* [Lencevicius et al. 1997]. Lencevicius et al. propose a debugging framework in which queries are used to to search out data with specific properties and relationships from amongst the state of a standard CPU-based program.

These concepts are relevant to our debugging engine for the graphics pipeline. We do not explicitly capture all data as in the omniscient debugging technique, instead organizing the data into *virtual* tables that are then built only in response to user input. The user interacts with these tables using relational algebra queries, providing a uniform mechanism to control the capturing, organization and data-manipulation processes that are involved in typical debugging sessions. This approach is tightly coupled with the use of visualization techniques wherever appropriate, for example mapping data elements to pixels to help debugging scalar quantities. The use of relational algebra allows a simple set of visualization modules to be extended to solve a variety of debugging and problems. While visualization has been applied to the debugging of graphics software many times over the years, it has not been applied in such a general manner to data extracted from inside the graphics pipeline itself.

3 Query-based Approach

The standard approach to debugging a program by sequentially following the progress of the CPU's program counter over time has been in use since at least 1961 [MIT 1961]. Although this technique works reasonably well for tracing programs on a single CPU or even multiple CPUs, it is more problematic for debugging the entire graphics pipeline. In particular, consider the flow of data elements from the vertex processing stage, through the rasterization, and to the fragment processing stage. For triangle primitives, there is a three-to-one relationship of data elements going from the vertex processor to the rasterizer, and then there is a one-to-many relationship from the rasterizer to the fragment processor. It is hard to conceive an effective linear time sequence to follow data from the start of the pipeline to the end.

Consider instead a model where individual graphics API commands contribute new rows to an organized relational database. Some tables track the OpenGL state while others track triangles, vertices and fragments created in the pipeline. Using standard relational operators we can not only restrict our query to specific parts of the database but also study the relationships among the data that are often lost in standard debugging approaches. This working model proves to be a powerful tool for examining the graphics pipeline.

Table Name	Primary Key	Foreign Keys	Description
App	stateID		Application stack and variables
GL	stateID		OpenGL state
Prims	primID	vertID [0,1,2,...]	Maps triangles to vertices
Verts	vertID		Vertex shader inputs
SVerts	vertID		Vertex shader outputs
Frags	xy	stateID, primID	Fragment shader inputs
SFrags	xy	stateID, primID	Shaded fragments, pre-culling
FB	xy	stateID, primID	The frame buffer

Figure 3: Virtual data tables and their primary and foreign keys used to model the OpenGL pipeline.

3.1 State Tables

We represent data in the graphics pipeline as a set of virtual *state tables*. There is a separate table for each logical part of the graphics pipeline, starting with the application point and terminating with the frame buffer. Generally speaking, each column of a table represents some state variable, and each row represents a new point in time. So, for example, there is a row for each OpenGL call in the `GL` table, a row for each vertex in the `Verts` table, and a row for each fragment in the `Frags` table. Columns of the `GL` table are pieces of the OpenGL state, whereas the `SVerts` (shaded vertices) and `SFrags` (shaded fragments) tables have columns not only for the output variables of these stages but also for variables set as lvalues at each line of a bound vertex or fragment program (and possibly multiple iterations of these). Clearly, these tables can get quite enormous; our goal will be to populate the rows and columns of these tables only upon demand.

Primary and foreign keys for each table, shown in Figure 3, make it possible to perform relational joins between tables, establishing functional relationships such as vertices to fragments. These keys allow, for example, individual pixels to be associated both to an OpenGL state value that was set when the pixel was created and to the particular primitive that was rasterized to cover this pixel. These relationships can be established via the stateID and primID keys respectively.

As we reach the end of the graphics pipeline, data elements pass through stencil, alpha, and depth tests, with each stage killing some fragments. Queries interested in these culling operations can access all the columns from `SFrags`, but with decreasing numbers of rows. Queries on these tables can be achieved either by implementing separate virtual tables for each culling stage, or more simply by creating a relational algebra expression to emulate the same functionality on the CPU.

The frame buffer requires careful treatment because it is technically OpenGL state. As would be expected, the frame buffer contains individual columns for each buffer attribute — color, depth, stencil and so on. Rows are another matter: every time an OpenGL call modifies the frame buffer, for example a `glDrawPixels` call, a whole new set of pixels becomes available in the frame buffer. Thus, querying of this table needs to be judicious: querying throughout a drawing call would yield a new frame buffer after each triangle, whereas querying at the end of a draw call would give the buffer state at the end of that call.

Another interesting question regarding the frame buffer is its initial state. Sometimes it is more useful for a frame buffer query to provide only the newly-generated pixels while other times a user will want to watch the frame buffer as it evolves from its initial state. Both types of outputs can be obtained in our model.

3.2 Specifying Data

In order to access and manipulate these tables, we create the graphics query language, *GQL*, basing it on the popular relational database language SQL. As in SQL, a single command, `SELECT`, serves to perform several relational algebra operations: projection, which filters columns; selection, which filters rows; and join, which merges rows from one or more tables according to a column relationship. For example, the following statement selects the normal vector from line 37 of a fragment program:

```
SELECT normal:37 FROM SFrags
```

In practice, the notation is a bit more verbose since you usually want to name your output for later processing. The following command is a little more common:

```
CREATE TABLE normalTable FROM
(SELECT xy, normal:37 FROM SFrags)
```

This sort of table can then be handed to an RGB or normal visualizer to show us the normals computed by our fragment program. Of course, since this command selected from `SFrags`, it will obtain *all* fragments, not just the forward facing ones. To get rid of most of these stray pixels, we can add a restriction to the expression:

```
SELECT xy,normal:37 FROM SFrags
WHERE (normal:37).z > 0
```

We have found that support for vectors, vector swizzling (shown a little bit here), and matrices to be critical for GQL's usability. Such notational conveniences help maintain clarity in an already verbose language.

The utility of GQL becomes evident when we start merging data from different queries. Instead of studying just fragments, we can figure out which vertex indices contributed to each forward-facing pixel:

```
SELECT SFrags.normal:37, Prims.vertID0,
       Prims.vertID1, Prims.vertID2
FROM SFrags,Prims
WHERE (normal:37).z > 0) &&
      (SFrags.primID == Prims.primID))
```

From here, a variety of things are possible. For example, we might join this table with a variable captured from the vertex shader. While the resulting statements may seem a bit long, they are actually a fairly compact shorthand for a sweeping set of modifications to an OpenGL application and its shaders. Making such modifications by hand may be prohibitively difficult and time-consuming.

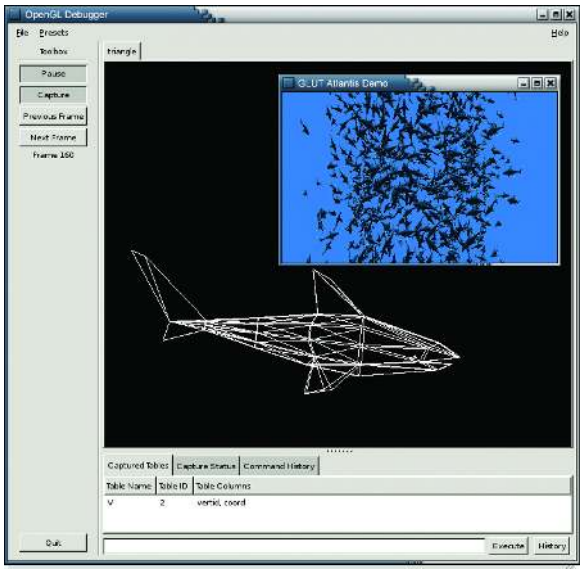

Figure 4: Using a `WHEN` statement to extract a single shark from a scene of 900 sharks in the Atlantis demo program.

3.3 Selecting Objects

To apply the debugger more selectively in the context of a large application with many objects, the user can prepend to a selection query a `WHEN` expression, which subsets the OpenGL command stream and effectively defines which rendered objects should be captured for debugging (see Figure 4). The `WHEN` expression can only access variables from the `App` and `GL` tables, since these can be evaluated without the special handling necessary to extract data from inside the graphics driver. For example, we may say:

```
WHEN App.Stack0=="main.cpp:42"
SELECT GL.ModelViewMatrix*Verts.Coord
FROM Verts, GL
WHERE (coord > (0,0,0)) && (coord < (1,1,1))
```

When the application reaches line 42 in the file main.cpp, which is presumably a `glEnd`, `glDrawArrays`, or `glDrawElements` call, we select all vertices inside a unit cube and transform them by the current modelview matrix. This example demonstrates GQL integrating with an application's namespace. It is possible in GQL to access the application's stack as well as its global variables (via the `App` table). This proves useful when integrating with large applications whose OpenGL output is complex and variable across frames (due, for example, to view-frustum culling).

The `WHEN` statement also handles cases where a simple `SELECT` statement would otherwise lead to results containing thousands of identical rows. This can happen in the `FB` and `GL` state tables. For example, if we want to study the OpenGL modelview matrix over a frame's duration, we use the following statement:

```
WHEN CHANGED(GL.ModelViewMatrix)
SELECT App.Stack0,GL.ModelViewMatrix
FROM GL,App
```

This gives us a list of the different modelview matrices and where they were set during the course of a frame. Without the `CHANGED` operator, we would have captured as many rows as there were OpenGL calls in the frame, a potentially huge number.

Figure 5: Extracting a single missile object from the BZFlag application. BZFlag issues between 300 and 1000 drawing blocks per frame, depending on the viewing parameters, number of missiles, and other game state. The ability for `WHEN` statements to use the application's namespace (in this case, the stack) is essential for this sort of application.

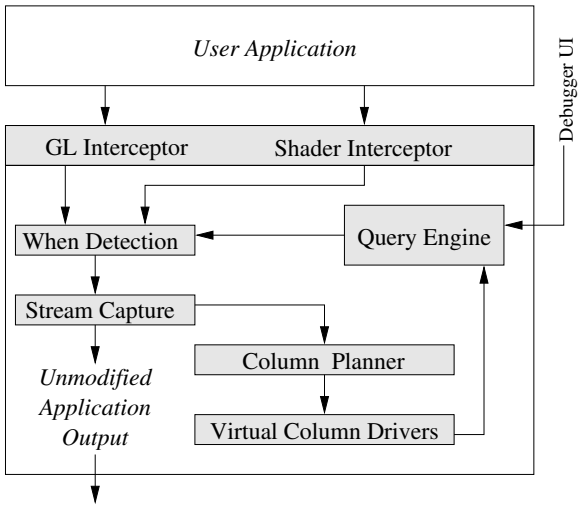

Figure 6: Data flow through the OpenGL debugger library. The debugger library runs inside the user's process using fake OpenGL libraries. An external program then connects to this process to issue queries and visualize the results.

From a design standpoint, the effect of the `WHEN` clause can be achieved by ANDing together the `WHEN` condition and the `SELECT` statement's `WHERE` clause. While such statements have more-elegant query trees, evaluating them efficiently at runtime is extremely difficult. We discuss this problem in Section 4.3.1.

4 Implementation

Building a debugger around GQL involves many different decisions, some algorithmic and some design-oriented. We used the following design principles to guide our choices in these matters:

1. Recompilation and/or special instrumentation of the application should not be required to debug a program.
2. Application output should remain unchanged during the debugging session.
3. The debugging algorithms should not depend on custom hardware or driver modifications. Similarly, the debugging should take place on the actual target hardware, not in software simulation.
4. Debugging statements involving fragment and vertex programs should be issued in the level of language (i.e., high or low) used by the programmer.
5. The ability to visualize data should be an integral part of the debugging environment, but should not inhibit data manipulation.

The system resulting from these goals is a compelling demonstration of the power of a comprehensive debugger for the graphics pipeline.

4.1 System Overview

Our system uses a conventional debugger model: a graphics application is instrumented so that its internal graphics calls can be debugged. To this application process is attached a user interface process that, in our case, is used both to issue queries and also to visualize the results.

We use the model pioneered by WireGL [Humphreys et al. 2001] and later Chromium [Humphreys et al. 2002] to instrument the graphics application's rendering calls. By masquerading as an OpenGL driver we are able to intercept and modify an application's drawing commands.

The actual modifications we make to the graphics stream are derived from the user's GQL expressions. First, the `WHEN` statements in the user's entered queries are used to break the OpenGL command stream into "chunks" of the stream that either are or are not of interest. Next, when a query references a column from one of GQL's virtual tables, we modify the captured OpenGL chunk and re-render it to extract the virtual column's data. The resulting columns are then handed to a relational algebra engine that evaluates the rest of the GQL query and returns it to the user.

This basic algorithm is modified slightly to prevent interruption of the output of an existing OpenGL application. A system capable of evaluating GQL queries without interruption (depicted in Figure 6) operates in the following way:

1. Two interceptor modules are used to capture OpenGL commands as well as the source code for any shaders used by the graphics application.
2. The useful parts of the captured OpenGL stream are located by the `WHEN` detection unit.
3. The sequences of commands marked as useful by the `WHEN` detector are buffered by *Stream Capture*.
4. This buffer is played once to the application to guarantee *unmodified application output* and then handed to the *Column Planner*
5. We build a number of *Virtual Column Drivers* which, when handed a sequence of OpenGL commands, satisfy requests for specific columns from GQL's virtual tables.

6. In the *Column Planner* we decide which virtual column drivers to use to satisfy outstanding column requests as well as the order in which to run them.

7. Satisfied column requests are passed back to the query engine, which evaluates the remaining query tree and passes it back to the debugger front-end.

The remainder of this paper reviews the various algorithms and methods used in these modules. First, we show how to perform multiple rendering passes on an incoming OpenGL stream without (a) a significant performance hit and (b) affecting the existing application's output. Next, we discuss how to map GQL, whose domain is the query tree, onto linear OpenGL rendering passes. Finally, we will discuss the various virtual column drivers that each execute in a single pass to extract various types of information from the input OpenGL stream.

4.2 Low-Level Subsystems

The basic approach of this debugger is to transparently capture OpenGL commands from the application and, controlled by the GQL language, render these captured commands multiple times into a separate but identical OpenGL context. Each pass, controlled by its virtual column driver, will modify the command stream slightly in order to capture a particular attribute from the graphics pipeline.

There are two basic components to our low-level debugging engine that make this whole process possible. First, there are a pair of interceptor libraries that capture the OpenGL and shader command streams. Second, there is a system that buffers the OpenGL commands and sets up these virtual OpenGL contexts.

4.2.1 Interceptors

The basis of our system is a pair of fake stub libraries – one that intercepts OpenGL commands and one that intercepts the high level source code for the application's shaders. The Chromium library [Humphreys et al. 2002] (and thus OpenGL) is a natural choice for our graphics API interception problem because it requires little modification to be adopted to our goals. Not only does Chromium provide a standard stream processing framework for a nearly-complete subset of OpenGL, it provides a number of auxiliary tools that turn out to be very useful for our purpose, notably the state tracker [Igehy et al. 1998] and the display-list manager. We will discuss how we use these Chromium subsystems in detail in the next section.

One of the goals for this system is to debug the fragment program in its high-level (or highest-level) shader language. In other words, if the user coded a shader in an ARB shader assembly language, it should be debugged at the assembly level, and so on. Because GLSL was unstable when development on this system began, we focused on debugging shaders written in Cg. Although this leads to some Cg-specific nuances in our approach, there is no fundamental reason why our approaches cannot be adapted to one of the many other available shader languages.

Because Cg source code never reaches the OpenGL driver, we have built a secondary interception library that intercepts NVIDIA's CgGL library. The actual structure of our CgGL interceptor mirrors that of the WireGL and Chromium libraries: we build a shared object with identical exports as the regular CgGL library, then place the library in such a way that the dynamic linker finds our library rather than the real one.

Making high-level shader debugging work with languages that compile to an intermediate representation, e.g. Cg, requires access to the mapping of high-level symbol names to their intermediate counterparts. For example, when Cg compiles to ARB assembly, the identifiers for shader parameters are converted to an integer-based representation. Because instrumenting a high-level language often changes this mapping, it is critical for it to be exported from the shader compiler. In Cg, this data can be obtained via the `cgGetParameterResourceIndex` interface.

4.2.2 Stream Capture

One inevitable consequence of using hardware for debugging is output (and other resource) restrictions. For example, one render target-worth of data can be filled to capacity by a request for a single matrix. This is compounded by the fact that a single GQL query can result in an arbitrary number of requests for virtual columns. Although this output restriction problem may fade with time, trying to capture vertex shader variables at the same time as fragment shader variables is very difficult. As long as these restrictions persist, being able to perform multiple passes over the OpenGL command stream seems to be the only viable alternative. Such an approach allows the use of a divide-and-conquer strategy when faced with queries that cannot be accomplished in a single rendering pass. The question is, how do we do this in OpenGL?

One approach to this problem is to create one OpenGL context for every virtual column and then issue replicated OpenGL calls to each context in round-robin fashion. However, this scales badly because each OpenGL call ends up triggering an expensive `glMakeCurrent` call for every active context.

We have experimented with two ways to achieve multi-pass rendering in the OpenGL setting. The first approach, which we call the stream capture approach, uses Chromium's *display list manager*, which allows arbitrary OpenGL command streams to be very efficiently stored and replayed from memory. In effect, we use the GQL `WHEN` statement to select chunks from the OpenGL command stream that need multi-pass rendering to be captured. These chunks are stored in a buffer using the display list manager. For every rendering pass needed by GQL, we replay the display list and then roll back any state changes made in the captured chunk using the Chromium *state tracker*, leaving us with a fresh context ready for a new rendering pass.

This approach fares well enough for applications that issue their geometry using immediate-mode calls. Because the calls get packed into a display-list-like object, an individual capture can contain state changes in addition to drawing calls. This allows large scenes to be captured in a single pass. However, proper OpenGL semantics often dictate that we maintain our own copy of drawn vertex arrays rather than storing the application's original pointers. This over-copying problem can lead to poor drawing performance, especially for large models.

This leads to the second approach for performing multipass rendering, in which we fulfill all the necessary column requests each time we encounter a call that actually performs the drawing. This means that we don't have to worry about rolling back state changes other than the frame buffer and also that we do not have to carefully handle large objects like vertex arrays to guarantee replayability.

A number of graphics-debugging peculiarities limit the performance gains of this approach. In some cases, a virtual column driver will need to rewrite a vertex array to contain extra vertex attributes. This causes us to run into the same over-copying problem that is at issue with the stream buffering approach. Furthermore,

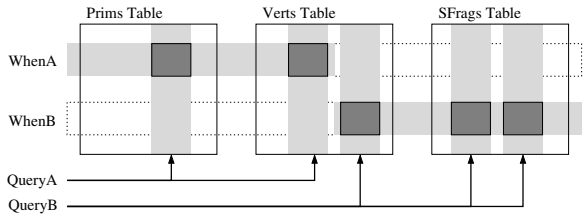

Figure 7: The leaves of GQL expressions form a list of virtual columns that must be extracted from the OpenGL command stream. Similarly, the WHEN statements in the GQL query tell which rows to get for these virtual columns. Efficient evaluation of GQL requires that we use these constraints so that the smallest amount of the OpenGL stream is buffered and the fewest number of rendering passes are then used to extract the data necessary to satisfy the virtual column requests.

when capturing scenes built out of large numbers of draw calls, this approach can lead to over-calling of the virtual column driver and thus large numbers of expensive frame-buffer readbacks. In short, the stream capture approach, while slow for large geometry, is easily coded and generic. In contrast, the per-draw-call approach gives speedups for specific virtual columns, but those costs can be easily lost to boundary conditions and large scenes. Because the gains of the per-draw-call approach vary so much from application to application, we opt to use the stream buffering approach in our debugger implementation.

The final (and key) low-level issue for the graphics debugger is context isolation. The debugger must perform its debug rendering into a secondary slave context that is fully synchronized with the master application context. The first reason for this is aesthetic: the debugger should not interrupt the application's rendered output. The second reason is unavoidable: certain columns are obtained by reading pixel data from the framebuffer. The slave context requires a floating point visual to prevent this data from being clamped and quantized to the framebuffer's depth setting. We accomplish context isolation using the same state-tracker algorithm used in the stream buffering system.

4.3 GQL Evaluation

The basic primitives of GQL evaluation, requests for table columns organized into a tree structure, are difficult to map onto the domain of the graphics debugger. As illustrated in Figure 7, two fundamental problems must be solved. First, we must create a stream processing unit that can can efficiently turn on and off the stream buffering subsystem in response to the set of `WHEN` expressions in the active GQL queries. Second, an order in which to capture the requested columns must be created. Once these two tasks are solved, evaluating GQL is quite similar to the task of evaluating SQL.

The needs of GQL with regard to relational databases are simple in that we do not require persistence or concurrency. However, the design of the buffering system makes it impossible to capture all of a column for an entire frame and then to go back and capture another column for the same frame. Instead, GQL requires that columns be captured all at once with their rows arriving sporadically throughout the frame as defined by the `WHEN` statement. This arrival order limits the applicability of many query optimization and evaluation techniques. Thus, the query processing may always be a significant cost in debuggers of this design. This is certainly the case in our system, which uses a lightweight, custom-built evaluation engine.

4.3.1 When-Detection System

The point of `WHEN` detection is to decide, based on a user-entered expression, when to start and finish capturing an OpenGL command stream. This feature is important for large graphics programs in which the user may wish to debug only a single triangle amongst an entire scene of objects. GQL allows `WHEN` expressions to contain references to OpenGL state as well as global application symbols and the application's stack. Our solution to the `WHEN`-detection problem is simple: evaluate the `WHEN` expression every time an OpenGL command is made.

To keep the overhead of this evaluation to an absolute minimum, we implement a simple, lazy-evaluation system for `WHEN` expressions. When a `WHEN` expression is active, we make note of which OpenGL state values it references. Every state-changing OpenGL function is then coded to consult this table to see if it's state value is being referenced and if so, trigger a reevaluation of that referencing `WHEN` expression. To improve performance of `WHEN`-evaluation for applications making many immediate-mode OpenGL calls, we perform special-case handling of `WHEN` blocks whose expressions are variable inside drawing blocks. This special case handling occurs for the `App.Stack` and `GL.PrimID` columns.

4.3.2 Column Planning

When a chunk of the OpenGL command stream has been captured by the debugger, it is handed to the column planner. A list of virtual column requests is computed by a quick traversal of all the currently active `WHEN` expressions in the system. From here, the column planner's job is to obtain all of these virtual columns from the buffered stream. Ideally, this is to be done using a minimal number of passes.

The first pass of the column planner performs extraction of columns belonging to the `App`, `GL`, `Verts` and `Prims` tables. Column requests for the `App` table is filled during the initial stream capture process since buffering application state in the way that we buffer OpenGL state is redundant. As it turns out, the `GL`, `Verts` and `Prims` tables can be captured during the buffering stage as well. This helps streamline our column planning algorithms, allowing certain queries to execute entirely-without buffering.

The remaining columns are treated in a two-tiered fashion. Certain columns are fundamentally dominant in the formation of rendering passes; these dominant columns are typically those that require significant effort to satisfy, such as a fragment shader variable (e.g., `SFrags.normal:30`). Our scheduling algorithm looks for one of these dominant columns and, if found, tries to capture both the dominant column and any additional columns that can be satisfied in the same pass. For example, we can acquire `SFrags.xy` and `SFrags.z` in the same pass used to capture `SFrags.normal:30`. Once all of the dominant columns have been captured, the scheduler responds to column requests on a pass-by-pass basis.

4.4 Column Acquisition

The final challenge in implementing a GQL environment is to transform (using one or more passes) the input graphics stream to produce the output requested by a virtual column. The following sections discuss the methods that used in this process.

Two approaches are possible in implementing virtual column drivers. In the theoretically elegant approach, individual Chromium stream processing units (SPUs) are chained together during a rendering pass with each SPU capturing a single GQL column. For

example, a single pass might chain together the following three SPUs: `Frags.XY -> Frags.RGB -> Frags.Z`. In practice, however, the stream transformations performed by each column driver are very simple, making this approach organizationally unnecessary. As a result, we opt for the monolithic stream-rewriting approach in which all of the column drivers are implemented inside one stream processing unit.

4.4.1 Basic Shader Debugging

Many of the columns in GQL tables map to variables inside vertex and fragment shaders. To get this data, we follow the same basic approach used by developers and existing debugging systems alike [Stephenson 2000; Purcell and Sen 2003; Olano 2005] — the currently bound shaders are rewritten, outputting the variable of interest to a render target. This approach requires a small number of extra resources to instrument the shader programs. These resources can range from a few extra shader instructions to an extra vertex attribute or texture unit.

There are a number of basic tricks required to make this approach work in practice. First, we render into an off-screen floating point buffer rather than a standard frame buffer to avoid precision problems. Even so, numeric precision must be carefully considered when passing and extracting integer quantities to and from the render target, as is done with the `SFrags.primID` virtual column, for example. Another essential trick to production shader debugging is the stream buffering system, which allows us to ignore the whole problem that the available render targets can store a limited number of output channels. When we need to capture more data than is supported by the hardware, we break the problem up into multiple rendering passes.

When this system was first created, the only mature shading language available that matched our needs was NVIDIA's Cg language. Thus, our debugger focuses on the challenge of debugging shaders written in the Cg language. While this introduces certain nuances that are Cg-specific, we see our overall approach as portable across the gamut of shading languages, from the ARB assembly-style languages to GLSL.

Interestingly, Cg is actually more difficult to debug than some other languages because it compiles to an intermediate representation. Changes that we make to the high level shader can alter variable and attribute assignment in the intermediate shader format. Re-synchronizing the intermediate-level shader API with the new shader requires an extra remapping step. We would not have encountered and addressed this issue had we done our experiments in a language other than Cg.

Once the Cg program string is acquired through the low-level shader interception library mentioned earlier, it is broken down into a tree structure using a simple Bison parser. Conveniently, NVIDIA provides an open-source Cg grammar via their web site [NVIDIA 2001]. We use this to perform a lightweight traversal of the shader's structure, extracting (1) structures and program arguments and (2) line and symbol information for every line in the program. Importantly, this parsing process stores enough information to reconstruct the original source via a tree traversal. This allows us to parse the shader, edit the tree according to column-specific rewriting rules and finally convert the tree back into a string representation for subsequent rendering.

4.4.2 Flow Control Issues and Shader Rewriting

There are a number of basic problems that are fundamental to the task of performing debugging by forcing early exit from any program. The main quandary is flow control: how do we handle cases where the variable to be inspected (henceforth called the target variable) is inside a conditional block?

This is actually a two-part quandary, the first part of which is the issue of return value: How do we distinguish legitimate output from sentinel (i.e., did not execute) output? We solve this problem by designating an arbitrary value to denote the sentinel condition. In practice, we have to use a small range of values to compensate for the limited precision of graphics cards' floating point units. As support for multiple render-targets becomes more ubiquitous it may be possible to avoid this problem entirely by dedicating a channel in the frame buffer exclusively to a valid bit.

The more vexing dilemma with flow control is how to implement immediate returns inside control statements. As noted in Section 2, some shader debuggers have had the ability to exit from a shader regardless of program-counter position. This is not the case on current graphics hardware. Our current solution is to (recursively) rewrite `if` statements so that all possible branches set a return value for shader program.

For-loops require special treatment in our debugging model. To capture a variable from inside a loop, the user specifies a constant iteration number, i, at which to extract a variable. This allows us to break the loop into two parts. The first part performs the zeroth through $i-1$th iterations of the loop, then second part performs the ith iteration, which extracts the target variable. While this approach is somewhat ad hoc, there is no obviously better approach to this general parallel debugging problem.

A number of other small details must also be handled during shader rewriting. First, extracting data from secondary functions requires us to inline the called function and then perform flow-control resolution on the resulting source. Second, large variables that do not fit into the render target (e.g., matrices) must be read back and assembled in several passes instead of just one.

Our approach to this problem is a short term solution to a long term challenge facing graphics programmers: as graphics hardware evolves more complex control flow mechanisms, the system used to force an intermediate variable to the shader output will also need to mature. Hardware support seems to be the clear way to achieve this. In the interim, approaches such as ours and the one used in Shadesmith [Purcell and Sen 2003] will likely continue as the only alternative to software emulation.

4.4.3 The Frags/SFrags Column Driver

To acquire a column for the `Frags` table or the `SFrags` table, all that we have to do is redirect the target variable to the shader output, render the scene and read back the frame buffer into our GQL data structures. The main challenge in the fragment program is how capture all fragments that are generated rather than just those that pass the z-test. To accomplish this we currently use a straightforward depth peeling algorithm [Everitt 2001]. First, we add a depth texture sampler to the shader, as well as a parameter containing the depth value of the current fragment. We then prepend a segment of code to the shader that performs an early fragment kill if the depth of the pixel is greater then the depth in the texture. The depth buffer resulting from each pass is fed back as input to the next pass and we iterate until no new pixels are written to the frame buffer. Fig-

```
float4 main(... ,
            uniform samplerRECT gldbFrontDepthTex,
            in float4 gldbWPOS : WPOS) : COLOR {
    float2 gldbTexCoords = float2(gldbWPOS.x,
                                  gldbWPOS.y);
    float gldbFrontDepth = texRECT(gldbFrontDepthTex,
                                   gldbTexCoords).x;
    if (gldbFrontDepth+0.00001>=gldbWPOS.z)
        discard;
    ... // compute some shading, such as a reflectVec
    float4 debugVar = float4(reflectVec, 0);
    return debugVar; // exit with target variable
}
```

Figure 8: A sample fragment program rewritten by our system. This program is performing depth peeling and fragment variable debugging, both of which have been added by the debugger. Additions made by our system are highlighted in green.

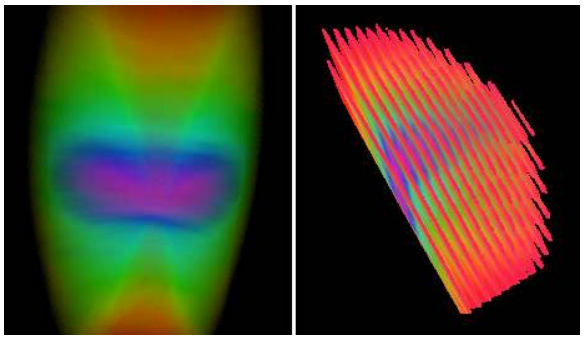

Figure 9: Left: output from a volume rendering program. Right: slices from the volume obtained from the `SFrags` table via automatic depth peeling.

ure 8 shows an example of such code. An example of this process as applied to volume renderer debugging is shown in Figure 9.

Given that our goal here is to acquire data values for *all* fragments, it is worth noting that the depth peeling approach may suffer from robustness problems in some situations. This is due to limited precision of the depth buffer and can be exacerbated by the application's projection matrix. If we have an additional render target available, we can instead record the fragment's primID in this target. Because the primIDs are issued in increasing order, we can test the primID rather than the depth in each successive pass so that every fragment is captured.

Generalized capture of the primID column requires some special handling. This fragment attribute reports which triangle, and thus which vertices, contributed to the fragment. To perform this capture, we *unshare* the vertices being issued to OpenGL so that vertices are unique to a primitive. Then, a custom vertex attribute corresponding to the primitive ID is added to the vertex and the entire model is rendered. We add this parameter as a pass-through in the vertex shader and alter the fragment shader so that this new ID is output to the render target. As with all readbacks, care must be taken when assigning primitive identifiers so that they are stable with regard to floating point precision. The ability to perform multiple rendering passes can be used to solve this problem robustly.

As alluded to before, adding the inputs to a fragment or vertex program can lead to implementation difficulties in languages (like Cg) that compile to an intermediate representation. This compilation step, which removes dead code and more importantly rearranges

```
struct inputs{float gldbVertID : ATTR6;
              float4 Position   : POSITION;
              float4 Normal     : NORMAL;};
struct outputs{float4 gldbReturn : TEXCOORD3;
               float4 hPosition  : POSITION;...};
outputs main(inputs IN,
             uniform float gldbRenderWidth,
             uniform float4x4 ModelViewProj,...){
    outputs OUT;
    OUT.hPosition = mul(ModelViewProj, IN.Position);
    OUT.gldbReturn = OUT.hPosition;
    ...
    float4 gldbVertPos;
    gldbVertPos.x = ((float)((int)IN.gldbVertID.x %
       (int)(gldbRenderWidth))/gldbRenderWidth)*2-1;
    gldbVertPos.y = ((float)((int)IN.gldbVertID.x /
       (int)(gldbRenderWidth))/gldbRenderWidth)*2-1;
    gldbVertPos.zw = float2(0,1);
    OUT.hPosition = gldbVertPos;
    return OUT;
}
```

Figure 10: A sample vertex program rewritten by our system. This program is debugging the transformed vertex coordinates which are returned through the render target. The highlighted sections have been added by our system.

the assignment of parameters to the intermediate registers, means that OpenGL state bound to these intermediate registers must be rebound to the new register assignments. If future shading languages and APIs take this reassignment challenge into account, the problem of shader debugging can be considerably simplified.

4.4.4 The SVerts Column Drivers

The main challenge in debugging vertex programs is creating a deterministic and fast way to map of vertices to pixels in the frame buffer. This precludes simple methods where the scene is simply rendered with points and then scoured for non-sentinel values as a post-process.

Our solution to this problem is to provide an extra vertex attribute that, just before the vertex program ends, is used instead of the newly-computed vertex coordinate. The extra attribute is an vertex-specific identifier that, when combined with a uniform parameter for the frame buffer width, allows us to pack vertices into the frame buffer in row-major order. This is shown in Figure 10.

After this packing step has been added, we debug a variable inside the vertex shader using the now-familiar steps from our fragment shader rewriting. To forward the variable to the frame buffer, we modify the vertex program's output structure to pass the value of the variable being debugged. We then replace the fragment program with a pass-through program that just outputs its one input attribute. Finally, we render the entire scene using single-pixel point primitives, perform a readback and unpacking step, and return the results to the GQL system.

4.4.5 Other Virtual Columns

A number of drivers for GQL tables remain undiscussed, namely the `App`, `GL`, `Prims`, `Verts`, and `FB` tables. As mentioned in the GQL evaluation section, the first four of these tables are implemented inside the stream processor that performs stream buffering for performance reasons. The `FB` table piggy-backs on the `SFrags`

implementation, but disables the depth-peeling algorithm. Thus it is possible and often useful to ask for shader variables from the frame buffer, e.g., `SELECT Normal:30 FROM Prims`. Since application-side camera control is still possible during debugging, frame buffer pixels are often all that is needed.

We use the lazy-evaluation technique used in `WHEN`-detection to efficiently respond to queries over the `GL` and `App` tables. We use the OpenGL state changes in this case to perform run-length encoding of the columns, allowing us to lazily populate the table while also keeping a small memory footprint. This is especially useful in reducing overhead when performing profiling.

5 Practical Usage and Examples

The interface to our debugger provides three things: a way to pause the application, a window for entering GQL commands, and a variety of tools for visualizing the tables created using GQL. One consequence of using GQL so heavily in our system is that common visualization tasks like the application of color mapping functions can be accomplished with GQL queries, allowing us to get by with a relatively simple user interface. In practice, the simplest visualizations are most frequently used in regular debugging sessions. The first is the table view shown in Figure 12. In a surprisingly large number of cases, simply having access to the proper data is sufficient to debug a graphics application.

For many graphics bugs, you can see the problem but don't know which vertex or triangle is to blame. Simple GQL statements and visualizations can be combined to solve these problems quickly. For vertices, you first "`SELECT vertID,coord FROM Verts`". From here, you just visualize the vertices as points, clicking on the problematic points to obtain the vertex indices. To save navigation time, you can match the visualization's camera with the application's camera with "`SELECT modelview,projection FROM GL`". A similar approach is possible with fragments; visualizing and picking the pixels that result from "`SELECT xy,primID,rgb FROM FB`" produces the frame buffer colors plus the triangle ID that is visible at each location. This proves useful in debugging problems that manifest only from certain angles, such as geometry misalignment or texture coordinate errors.

With such large quantities of data, it is not surprising that sometimes a good visualization provides more insight than a tabular view. Visualizations shown in Figure 11 demonstrate this — visualizing points and normals is useful for debugging lighting and similar reflection-driven algorithms. With volume renderers, capturing all fragments rather than just the frame buffer may be useful for debugging things like lighting and color-mapping (see Figure 9). This same approach is useful when debugging multi-pass and compositing algorithms.

Sometimes studying multiple frames of output is important for graphics debugging. The *view* mechanism of GQL is useful here: by creating a view of some GQL expression, e.g. "`CREATE VIEW VertHistory FROM (SELECT coord FROM SVerts)`", multiple frames of geometry will be captured. We can then view each frame independently, or merge multiple frames into one via "`SELECT HISTORY frameID,coord FROM VertHistory`". We expect this to be useful for debugging things like camera movement, vertex skinning, or any other time-varying aspect of a graphics program.

Just as GQL can delve into the finest-grained parts of the pipeline, it may also be used for high-level profiling. For example, draw count profiling can be done in the most general way in GQL using familiar SQL operations like `GROUP BY` and `COUNT`/`SUM`/etc. A simple trace and profile of all the application's OpenGL calls can be obtained using "`SELECT TIME(), command FROM GL`". More sophisticated things are possible: by joining "`WHEN isDrawingCommand(GL.command) SELECT command, blockID FROM GL`" with "`SELECT xy,blockID FROM SFrags`", we can count both the overdraw at each pixel along with which OpenGL drawing block responsible for the fragment.

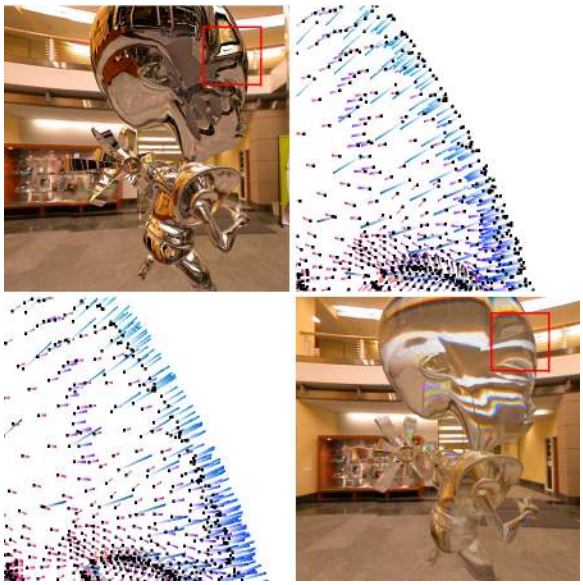

Figure 11: A sequence of screen shots in a typical debugging session. The top-left is an incorrectly rendered scene. The top-right is the result of viewing the main object's vertex coordinates and normals. We notice the normals are inverted and correct the bug in our application program. Viewed again in the debugger, as seen in the bottom-left and right, we see that this has indeed fixed the problem.

Finally, GQL can be used to obtain and study arbitrary OpenGL objects. Things like textures are part of the `GL` table and are thus retrieved with a simple select, e.g., "`SELECT Texture[0] FROM GL`". This approach can be particularly effective when debugging rendered textures, although in such a case a `WHEN` expression is often used to set when the texture is actually captured from the `GL` table. Interestingly, since drawing calls to an OpenGL pbuffer use the same OpenGL stream as the main application context, a GQL debugging session on an offscreen buffer is indistinguishable from a session being applied to a regular OpenGL context.

6 Conclusions and Future Work

The system presented here is an important steps toward the creation of a general purpose graphics debugger. While our methods are by no means the final word on the subject, particularly when considered in terms of speed and ease-of-use, we have shown that a general and transparent graphics debugger is feasible.

Much remains to be done on the OpenGL compatibility front of our implementation. Support for OpenGL's various data types, primitive types and texture formats limits our debugger's current portability. The same goes for multiple contexts, rendered textures, and OpenGL extensions. We do not anticipate any fundamental prob-

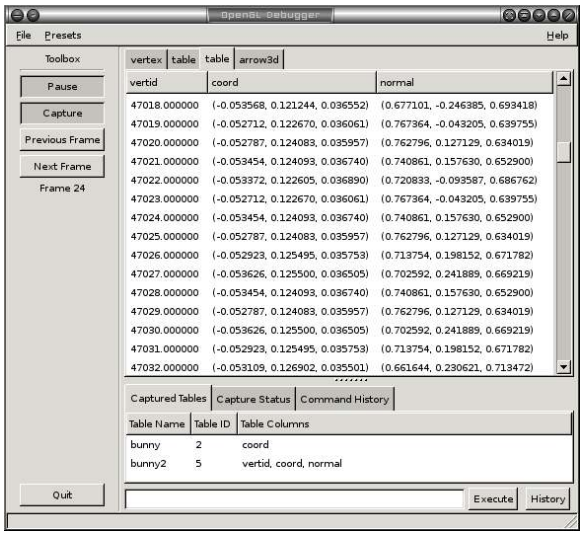

Figure 12: Table view of debugging data.

lems in implementing these features.

The approach of this system is by no means perfect. Integration of the graphics pipeline's namespace with the application's namespace requires further study. Similarly, because we currently respond to queries in a column-by-column fashion, we preclude a number of clever and common query evaluation optimizations. As things stand now, the database is our bottleneck. Coaxing performance from such a system is an interesting problem requiring more study.

Visualization of the data that comes out of the graphics pipeline remains a fascinating problem. For example, how does one visualize the result of a join between fragments, triplets of vertices, and extracted shader variables? We have only begun to scratch the surface of these problems with our present visualization tools. Several interesting directions might be pursued, including re-tasking GQL as a data source for existing tools like OpenDX, VTK or AVS or building an actual debugging environment around the GQL language.

In this paper, we have presented the methods, algorithms and implementation of the Graphics Query Language. This debugging engine provides what we believe to be the first step toward the development of an end-to-end debugging solution for modern graphics hardware.

7 Acknowledgments

We would like to thank the many developers of the Chromium system. Without Chromium our system would not be possible. We would also like to thank the developers of all of our test applications, including the NVIDIA SDK examples, BZFlag, and Mark Kilgard's Atlantis screen saver. The authors also wish to thank Laura Davulis for proofreading this paper. This work has been sponsored in part by NSF Medium ITR IIS-0205586, NSF Small ITR AST-0313031, and a DOE Early Career Award.[1]

[1]The views expressed in this paper are not necessarily the views of our sponsors.

References

BAXTER, B., 2002. Imdebug. [online] http://www.billbaxter.com/projects/imdebug/.

EVERITT, C., 2001. Interactive order-independent transparency. [online] http://developer.nvidia.com/object/Interactive_Order_Transparency.html.

FRIEDELL, M., LAPOLLA, M., KOCHHAR, S., SISTARE, S., AND JUDA, J. 1991. Visualizing the behavior of massively parallel programs. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, ACM Press, 472–480.

GOULD, D., 2005. GLSurveyor. [online] http://www.glsurveyor.com/.

GRAPHICS REMEDY, 2005. gDEBugger. [online] http://www.gremedy.com/.

HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. 2001. WireGL: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 129–140.

HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P., AND KLOSOWSKI, J. 2002. Chromium: A stream processing framework for interactive rendering on clusters. In *SIGGRAPH*, 693–702.

IGEHY, H., STOLL, G., AND HANRAHAN, P. 1998. The design of a parallel graphics interface. In *Proceedings of SIGGRAPH 1998*, 141–150.

KAUFER, S., LOPEZ, R., AND PRATAP, S. 1988. Saber-C: an interpreter-based programming environment for the C language. In *Proceedings of the Summer Usenix Conference*, 161–171.

LENCEVICIUS, R., HÖLZLE, U., AND SINGH, A. K. 1997. Query-based debugging of object-oriented programs. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, 304–317.

LEWIS, B. 2003. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging*, 225–235.

LINTON, M. A. 1990. The evolution of Dbx. 211–220.

MICROSOFT, 2005. Microsoft PIX development tool. [online] http://msdn.microsoft.com/. Available since DirectX9 on XBox and Win32.

MIT. 1961. DDT. Tech. Rep. PDP-4-1 PDP-1 Computer, Department of Electrical Engineering, MIT.

NVIDIA, 2001. Nvparse. [online] http://developer.nvidia.com/nvparse.

OLANO, M., 2005. Shader debugging in PixelFlow's PfMan shader language, SGI's OpenGL Shader, and RenderMan. Personal Communication (e-mail), Jan.

PURCELL, T., AND SEN, P., 2003. Shadesmith. [online] http://graphics.stanford.edu/projects/shadesmith/.

PURCELL, T. 2004. Fragment program debugging tools. In *SIGGRAPH 2004 GPGPU Course Notes*.

SISTARE, S., ALLEN, D., BOWKER, R., JOURDENAIS, K., SIMONS, J., AND TITLE, R. 1992. Data visualization and performance analysis in the prism programming environment. In *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, North-Holland, 37–52.

STALLMAN, R. M. 1989. GDB manual (the GNU source-level debugger). Tech. rep., Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, Jan. Third Edition, GDB version 3.1.

STEPHENSON, I. 2000. Buffy, an sl development environment. *BCS Conference on Digital Content Creation* (April).

TREBILCO, D., 2004. GLIntercept. [online] http://glintercept.nutty.org/.

ZELLER, A., AND LÜTKEHAUS, D. 1996. DDD – a free graphical front-end for unix debuggers. *SIGPLAN Not. 31*, 1, 22–27.