OPTIMIZING OpenGL BASED SOFTWARE SYSTEMS THROUGH REVERSE ENGINEERING

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Joseph E. Sullivan, B.S.

San Marcos, Texas
December, 2007

ACKNOWLEDGMENTS

My sincerest thanks and admiration go to Professor Wilbon Davis: his advice, knowledge, guidance, and patience have been invaluable in the preparation of this thesis.

I would like to thank the entire faculty and staff of the computer science department at Texas State University-San Marcos, in particular, the members of my committee.

My thanks also to my family, friends, and colleagues for their support, ideas, and encouragement.

I would also like to mention the legendary blues artists Elmore James, Robert Johnson, and Sonny Boy Williamson.

This manuscript was submitted on October 24, 2007.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

OPTIMIZING OpenGL BASED SOFTWARE SYSTEMS THROUGH REVERSE ENGINEERING

by

Joseph E. Sullivan, B.S.

Texas State University-San Marcos

December, 2007

SUPERVISING PROFESSOR: CAROL HAZLEWOOD

Dynamic execution traces of the OpenGL application programming interface (API) were gathered to identify the functionalities provided by OpenGL. The traces were gathered from the open source implementation of the OpenGL API, Mesa3D. The execution traces were gathered from OpenGL by using a software profiler and three open source *driver* programs to exercise the API. Several tools were developed to preprocess the execution traces. The tools extracted the call graph information from the execution traces and removed any call chains not related to the driver programs or OpenGL. The tools also formatted the call graph so that it would be acceptable input for the tools used to identify the functionalities. The preprocessed execution traces were then used to identify the functionalities of the API. There were 362 unique functionalities discovered with 1212 edges among them. The functionalities were identified. The hierarchical relationships among the functionalities were examined. Several optimizations, tools, and architectural analysis techniques are discussed using the information gathered.

CHAPTER I

INTRODUCTION

1.1 Motivation

Regardless of the application's intent, be it entertainment, medical imaging, or scientific modeling, performance is critical for graphically intensive software systems. Significant performance gains can be achieved utilizing traditional optimization methods, such as profiling a system to isolate bottlenecks and then hand tuning the problem code to mitigate the bottleneck. However, there is a point when the software system's architecture will begin to be the limiting factor of performance, and no amount of code massaging will provide significant performance gains.

Optimizing a software system that utilizes OpenGL, a software interface specification for graphics hardware, presents a significant challenge (Woo, Neider, Davis, & Shreiner, 1997). The transformations of vertices for animation by themselves represent a significant amount of computation and allocation of system resources. Can new optimizations be formulated by examining the usage patterns of the OpenGL functionalities, as defined by Hall and Davis (2004), or of the software that uses the application programming interface (API)? An API is a specification of an software interface. An API does not specify implementation details but rather the behavior of any identifiers specified in the interface. OpenGL is an API.

Graphics are a major component of many software systems, including video games, video and photo editing software, and scientific and medical imaging software. In general most optimizations for graphics are accomplished by hardware acceleration, partially because hardware floating point calculations are much faster than even the best software floating point emulation (Paul, 1997). Hardware is only part of the story where graphics optimizations are concerned. There will be some software component of any implementation of the API, and there is an opportunity to optimize that software (Paul, 1997). This paper discusses the internal structure of the API, and some possible optimizations of interest to developers implementing the API or to developers utilizing the API.

A number of OpenGL based applications were used to gather execution traces of the API. Tools were written to preprocess the execution traces, eliminating call chains from the traces that did have an execution path that crossed into the OpenGL API. The processed execution traces were then used to identify the functionalities of OpenGL, as well as the hierarchical relationships among the functionalities. The functionalities were then examined, and a number of different scenarios using the functionality analysis were identified. Examining the functionalities is useful, not only for identifying possible optimizations, but also for locating defects, and for the refactoring of existing systems. Refactoring is changing the code to be more efficient, reliable, and readable while still performing the same operations (Mancl, 2001).

The applications for this study were chosen to generate complete software execution coverage of the API. Mesa 3D was the OpenGL API implementation examined. Three *libraries* of source code were used to aid in the harvesting of the OpenGL execution traces: (1) A subset of the tutorial style sample programs provided with the *OpenGL Programming Guide* known also as the "Redbook", and two open source video games (2) GLHeretic and (3) Neverball.

A technique for profiling the dynamic execution of OpenGL is developed. The technique developed for profiling a *middle* layer like OpenGL is one of the contributions of this paper. The analysis of OpenGL revealed approximately 360 different functionalities with about 1200 edges among the functionalities. Several optimizations, refactoring strategies, and tools to aid in the development of software and increase performance are suggested from the functional analysis of OpenGL. Functionality analysis provides powerful insight into the software system and can be used to assist in many areas of software maintenance and development.

CHAPTER II

OVERVIEW OF PREVIOUS RESEARCH

Reverse engineering is, in essence, a method for discovering technical details about a particular system by observation of actions and developing a reasonable explanation for those actions (Canfora & Di Penta, 2007). Reverse engineering is useful for determining exactly how a software system works. Knowledge of how a software system works can be used for developing new software systems or for refactoring and optimizing existing software.

2.1 Conceptual Software Functionality

A definition for software functionality is, at best, ephemeral. Developers, users, and system architects all have different ideas regarding what exactly constitutes a "software functionality". Hall and Davis (2004) have developed a conceptual definition of a software functionality:

conceptually, a software functionality is a set of software modules that always execute together (Hall & Davis, 2004, p. 3).

Software systems are created to help a user accomplish a particular task. The purpose of a program can be viewed as the overall, or top level, functionality (Hall & Davis, 2004). An integrated development environment would provide a developer with a single environment in which to develop code for example. There can be many sub-functionalities provided by the system to aid in this task. Editor, debugger, and compiler interfaces may be provided to aid the program purpose. There are layers, or levels, of functionalities with the lower level functionalities acting to help the higher level functionalities fulfill their purposes. In this example the top level functionality would be a full-featured development environment, and the editor, compiler, and debugger interfaces would be sub-functionalities. In this manner a hierarchical view of the functionalities can be developed to show the relationships among functionalities. Each functionality is made up of software modules that can be viewed as procedural functions or member methods of a class (Hall & Davis, 2004). Modules can be members of more than one functionality (Hall & Davis, 2004). There exists a non-empty set of modules that will always be invoked, and these modules constitute the level 0, or top level, functionality (Hall & Davis, 2004).

2.2 Structural Software Functionality

Based on the conceptual definition of a software functionality, a structural definition can be derived allowing the functionalities to be described (Hall & Davis, 2004). The functionalities are derived from equivalence classes. An equivalence class being a subset of a given set with an equivalence relation among the members of the subset (Stoll, 1979). An equivalence relation is a relationship R from which the members of a set can not be differentiated by R (Stoll, 1979). The process for identifying the functionalities is demonstrated using a sample program in section 4.1. A transition is an explicit call between two modules (Hall & Davis, 2004). Using the dynamic execution traces the transitions that make up the software system can be assigned to equivalence classes. A functionality is the set of target modules of the transitions that constitute the equivalence class.

Structurally, a software functionality is defined by the set of transition targets in the equivalence class associated with the functionality (Hall & Davis, 2004, p. 10).

The set of modules that constitute a functionality is unique, however a module may belong to more than one functionality (Hall & Davis, 2004). If two functionalities contain the same set of modules then they are the same functionality (Hall & Davis, 2004).

CHAPTER III

OPTIMIZING GRAPHICAL SOFTWARE SYSTEMS

3.1 Current Optimization Strategies

A developer faced with a software system that is failing to meet performance goals has several options available even with little control over how the OpenGL API is implemented. Most optimization for OpenGL based software systems involves optimizing data to streamline operation. Data reduction is not the only option, but that is the area of the software system over which the developer has the most control. However the reductions in data are almost always accompanied by a corresponding loss in the quality of the images being displayed.

Increases in performance can be realized by reducing the number of bits per color component, reducing the number of pixels when performing polygon fill, and utilizing lower quality texture filters (Kuehne, True, Commike, & Shreiner, 2005). By reducing the color resolution, the number of bits needed per frame will be significantly reduced. For example, lowering the color resolution from 8-8-8 (32-bit) color to 5-6-5 (16-bit) color results in approximately 33% fewer bits to set per pixel (Kuehne et al., 2005). Backface culling is a technique in which polygons not visible are culled or not drawn, or even sent to the drawing buffer. OpenGL has an algorithm to perform this optimization (Woo, et al., 1997). Textures are generally square and, they are manipulated (interpolated) to fit onto a polygon with minimal distortion. Depending on the shape of the polygon OpenGL provides a number of filters, or interpolation functions, to specify how to interpolate the points between the texture and the polygon (Woo et al., 1997). By using an interpolation function with fewer variables, texture coordinates can be calculated faster.

The above optimizations fall into the category of pixel operation reduction. There are also optimizations for reducing the number of vertex operations. Vertex optimization generally concentrates on reduction in lighting, light data, and appropriate use of connected polygons, like triangle strips to reduce the total number of vertices (Kuehne et al., 2005).

State sorting is an additional technique available to developers for optimizing an OpenGL based software system. State sorting consists of organizing an application to reduce the number of state changes in the rendering pipeline. Some changes to the OpenGL state machine result in validation of the state machine. For example, changing the polygon type being rendered from triangle strips to quad strips would cause the state machine to validate. Validation is an internal operation of the API in which the OpenGL state machine is reconfigured to keep its internal state consistent with the rendering pipeline (Kuehne et al., 2005). In order to reduce time lost in validation of the state machine, it makes sense to sort rendering requests to minimize the number of state changes (Kuehne et al., 2005). Some state changes are more expensive in terms of lost cycles, so care must be taken when sorting the render requests (Kuehne et al., 2005).

Data transfer to the graphics hardware can be a bottleneck (Chow, 1997). Geometry compression is one technique used to resolve this issue. Static geometry data are compressed, allowing for a smaller memory footprint. The data are sent to the rendering pipeline in a compressed format and is decompressed using special hardware (Chow, 1997). This technique does result in some data loss due to lossy compression (Chow, 1997).

This listing of optimizations is by no means complete, and is presented only as an example of some of the current techniques. Optimizing software systems that utilize OpenGL is not without challenges. Profiling and other measuring techniques can help to identify hot spots that need to be addressed in order to achieve the desired performance goals. There are relatively few techniques that don't require some kind of trade off between execution speed and quality of display.

CHAPTER IV

STRUCTURE OF OpenGL

4.1 Process for Identifying Functionalities of a Software System

Hall and Davis (2004) describe the process for identifying the functionalities of a software system using a small sample program. Allow M to represent the set of software modules for a system, and assume that M provides a set of functionalities, F (Hall & Davis, 2004). The set P represents the set of all execution profiles possible. An execution profile being a set of transitions that occur during a specific execution of the software system (Hall & Davis, 2004). A call graph is a multigraph with edges representing calls between the modules (Arnold & Grove, 2005). Allow X to represent the set of explicit transitions, a call to a module, in the dynamic call graph (Hall & Davis, 2004). There is at least one special start transition represented by '\$' which represents the flow of control from the operating system to the software system.

Figure 1 contains an example call graph of the sample software system. The digraphs in this paper were created with Graphviz. The sample system has a single entry point. The '\$' symbol in Figure 1 has been replace with 'START' for clarity. The digraph in Figure 1 is used to demonstrated the process of identifying functionalities. The nodes represent the modules and the edges represent the explicit transitions among the modules (Hall & Davis, 2004). The sets M and X can be determined from the call graph of the sample.

M = {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o}

X = {\$a, ab, ac, ad, bf, ce, cg, dh, eg, fi, fj, gl, gk, jm, jn, no}

The number of distinct execution profiles for a system can be large. The total number of execution profiles for the system is the number of unique, connected sub-digraphs with a common root that can be obtained from the system's call graph (Hall & Davis, 2004). The technique for identifying functionalities can be demonstrated with a small number of execution traces (Hall & Davis, 2004). Below is a representation of execution profiles numbered from one to six. Each pair in the list represents a transition in the call graph. The listing of traces may or may not be the complete set of execution traces for the software system. Only a small number of traces are necessary to demonstrate the technique.

1 = {\$a, ab, ac, ad, ce, cg, gk, kn, no}

2 = {\$a, ab, ac, ad, ce, cg, dh}

3 = {\$a, ab, ac, ad, ce, bf, cg, fi, fj, jm}

4 = {\$a, ab, ac, ad, ce, cg, dh, gl}

5 = {\$a, ab, ac, ad, ce, bf, fi, fj, jn, no}

6 = {\$a, ab, ac, ad, ce, eg}

Knowing a functionality is a set of modules which always execute together, the first step is to identify the sets of modules always occuring together (Hall & Davis, 2004). By tracking the execution trace in which a transition occurs, a list of equivalence classes can be built. The labels on the edges in Figure 1 are indices into a table mapping the set of execution profiles in which that edge occurs (Hall & Davis, 2004). For example, the edges in the graph labeled as '6' in the graph indicated that module transitions $b \rightarrow f$, $f \rightarrow j$, and $f \rightarrow i$, occurred in execution traces 3 and 5.

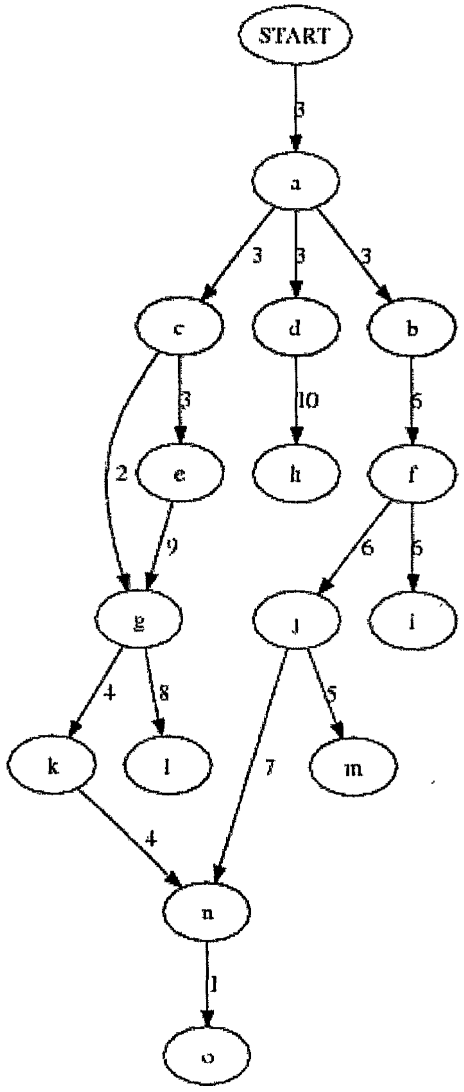


Figure 1: Sample Call Graph

Table 1: Edge Index Mapping

Edge Label	Execution Profile Set
1	{1,5}
2	{1,2,3,4}
3	{1,2,3,4,5,6}
4	{1}
5	{3}
6	{3,5}
7	{5}
8	{4}
9	{6}
10	{2,4}

There will be some set of modules essential to the execution of the system. There will also be an equivalence class representing these essential modules, as well as the flow of control into the software system. The fifth entry of Table 1 represents the set of modules common to every execution of the sample system for this example. Table 1 provides the mapping of edge labels to execution profile sets. This paper does not focus on the top level application layer, or the flow from the OS into the software system, but focuses on flow from software system to API layer. Therefore, there is more than one entry point when analyzing the API. This can be resolved by providing a special transition for each entry point into the system (Hall, n.d.).

Equivalence classes for the system can now be identified by examining recurring patterns of transitions and finding equivalence classes in the call graph (Hall & Davis, 2004). The relation of R on X is made of all pairs (x, y) where x and y have the same label (Hall & Davis, 2004). A set of transitions equivalent to x within R can be created given a transition x (Hall & Davis, 2004). The equivalence classes are sets of transitions that execute together. An unordered set of equivalence classes for this example are listed in Table 2.

Table 2: Example Equivalence Classes

Equivalence Class	Transition Set
1	{no}
2	{cg}
3	{$a,ab,ac,ad,ce}
4	{gk,kn}
5	{jm}
6	{bf,fj,fi}
7	{jn}
8	{gl}
9	{eg}
10	{dh}

A determination of which level a particular equivalence class belongs to can be made by creating a directed graph of the equivalence classes (Hall & Davis, 2004). The nodes in the graph represent the equivalence classes, and the edges indicate that there exist in the parent equivalence class a target module that is the source of a module in the child equivalence class. The root of the digraph will be the equivalence class common to all executions, in this case equivalence class 3 (Hall & Davis, 2004). The other equivalence classes will be represented as nodes in the digraph. The nodes of the digraph are assigned edges between them if, and only if, a transition target in an equivalence class is the source of a transition in another equivalence class (Hall & Davis, 2004). Figure 2 depicts the digraph of equivalence classes for this example. The numbering of the nodes is not consistent and each execution trace may have different numbers for the same functionalities. This is caused by the identification of new functionalities from the software system.

Determining the level of a functionality is accomplished by computing the shortest path from the root to the node in question (Hall & Davis, 2004). Since a module may be shared by multiple functionalities, they cannot be identified by the modules that belong to that functionality alone from an execution trace. However, the equivalence class transitions allow the functionalities to be identified in

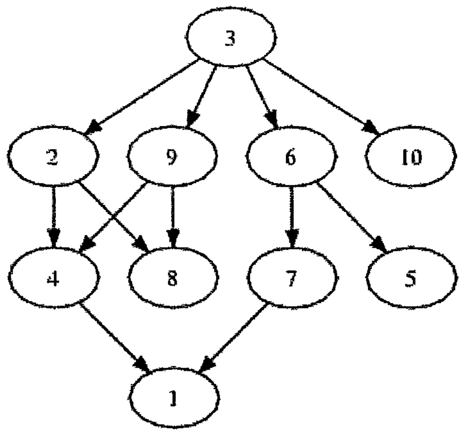


Figure 2: Digraph of Equivalence Classes

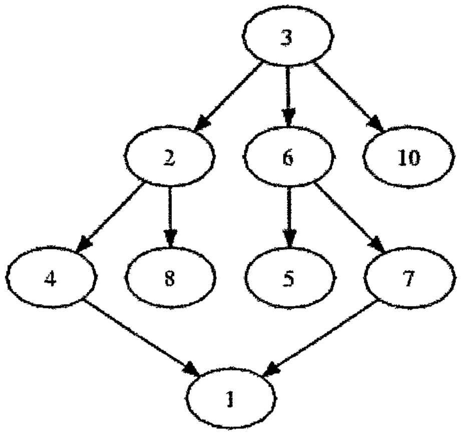


Figure 3: Optimized Digraph of Equivalence Classes

relatively straight forward manner. The target modules of an equivalence class describe a functionality (Hall & Davis, 2004). For example the module 'o' describes the functionality for equivalence class 1.

The nodes 2 and 9 in the digraph depicted in Figure 2 have exactly the same set of target modules. Two functionalities are actually the same functionality when they have the same set of modules that execute together (Hall & Davis, 2004). The graph can be optimized by collapsing these duplicate nodes. Figure 3 depicts the graph with functionalities 2 and 9 merged.

The level of a functionality can now easily be determined from the digraph. From the digraph in Figure 3 it can be determined that the functionality of node 3 is the root functionality. The determination that node 3 is the root functionality is straight forward in this case, simply identify the node with the transition '$a' (Hall & Davis, 2004). Nodes 2, 6, and 10 constitute the level 1 functionalities, and will be conditionally invoked to satisfy the root functionality. There are a total of four level 2 functionalities: nodes 4, 5, 7, and 8, and there is one level: 3 functionality, node 1.

The functionality information is cumulative across the execution traces. As each execution trace is processed, the functionality information from that trace is combined with the information from the previous traces. This cumulative effect means that only the final execution trace from a software system need be examined to observe all the functionalities of that system.

By examining the modules that constitute a particular functionality, some identification of the functionality can be made. This identification, in conjunction with an assignment of functionalities to levels, provides a basis for examining software in terms of performance, architecture, and security, etc.

4.2 Determining the Functionalities of OpenGL

To identify and examine the functionalities provided by the OpenGL API, execution traces were generated using the Valgrind profiling tools suite in particular Callgrind. The goal for profiling is to provide execution coverage so that each functionality of the API is exercised discretely at least one time. The number of unique execution profiles for a system is the number of unique sub-digraphs with a common root that can be obtained from a system's call graphs (Hall & Davis, 2004). When applied to an API with multiple entry points, the number of unique execution profiles becomes quite large indeed. There is a statistical method for determining when generating further profiles does not provide any benefit (Hall, 1997). The stopping criterion will be covered in section 4.2.4. The approach taken for profiling OpenGL was to use a set of small tutorial programs, along with two larger, more complex source bases provided by two open source games. This strategy allows the examination of the usage of OpenGL by complex systems (the games) in terms of the low level functionalities provided by the API. There are approximately 50 small samples. Each one attempts to demonstrate a single feature of the API. These samples are distributed with the *OpenGL Programming Guide* as tutorial programs, and provided as source code on the OpenGL website. The other two applications are both games provided as source, each of which represents a different style of game. GLHeretic is a first person shooter in which a player navigates through an environment engaging the game environment (mobile constructs, doors etc.) using keyboard commands and the mouse. The object of the game is to clear an area of enemies, and then find the exit. The other game is Neverball, an open source puzzle game, in which the player attempts to steer a ball across a playing field by using the mouse, with the purpose being to collect a number of *coins* and then navigate to an *end target* goal.

After the execution traces are processed and functionalities determined, the functionalities have to be identified in terms of what processing is provided. A determination of the nature of a functionality is obtained by examining the modules which constitute the functionality in conjunction with the source code for the modules making up the functionality.

4.2.1 Mesa 3D

Profiling OpenGL itself is not a particularly trivial problem since most distributions of the API are from commercial hardware vendors, making source code access impractical. There is no way to know the names of modules that make up the API without source code or symbolic information from the OpenGL implementation being analyzed. Traces would reveal that one address was called by another address, but there is not a way to determine the name of the module that corresponds to either address. Mesa 3D provides an open source implementation of the OpenGL API on multiple platforms. Mesa 3D has passed the OpenGL conformance tests (Paul, n.d.), making it a suitable for research involving OpenGL. It is understood that hardware implementations provide significant advantages over software implementations in terms of performance, but examining a software implementation provides a greater insight into many areas, and may present ideas for new hardware design.

Another issue is related to the nature of an API. By design, libraries are not stand alone applications. They require a driver program of some kind. This makes the gathering of execution traces more complicated. Instrumenting profilers such as gprof that insert performance tracking code into an executable either by recompiling with flags or by inserting directly into the binary itself, are generally not able to instrument dynamically linked objects. While using an instrumenting profiler is completely satisfactory for profiling the driver program, it will not capture any information about a dynamically linked objects like those of the OpenGL API. Sampling profilers are able to capture data across the dynamic link. Sampling profilers usually accomplish this by recording the stack. A sampling profiler is limited in that it does not record all information rather it records information after some time interval or event. The result is that the gathered profiles are incomplete.

4.2.2 Profiling

The profiler used to gather profiles for this research was Callgrind. Callgrind is part of the Valgrind profiling tools suite, which is an open source application available on PowerPC and X86 Linux platforms. Tools in this suite work by adding instrumentation code to an existing executable and running the instrumented code in a simulated environment. At program termination an execution trace is output to a file as plain text. The execution of the profiled program in the simulator is somewhat slower than native hardware execution, but the execution traces are complete and have information recorded from the dynamically linked objects. Completeness of the execution traces is not a practical expectation for a sampling profiler, and profiling into dynamically linked objects is problematic for many other profiling systems.

The execution traces gathered were typically very large, being anywhere from 20,000 to 80,000 lines long. A small snippet of an execution profile is listed below.

```
fl=(7) ex.c
fn=(220) test1
14 3
+1 1
cfn=(222) test2
calls=1 +7
* 5
* 1
+2 2

fn=(218) main
9 10
+1 1
cfn=(220)
calls=1 +4
* 12
+1 2
```

Each file, function, and binary object, an executable or library, is assigned an unique identifier used throughout that profile to identify that object. Files are abbreviated as "fl", functions are abbreviated as "fn", and child functions are "cfn". Each block of information in the profile lists the calling function followed by each function that it calls. Information about the number of times a child function is called, line number information, and time spent in the calls are listed in the information block as well. After the execution traces were gathered from the various software systems, each profile was preprocessed and formatted for the tools used to identify equivalence classes and functionalities.

The first preprocessor, called sliepnir (see section A.1), is a two pass parser. On the first pass it builds a list of relative binary objects, Mesa3d for example, and a second list of functions (modules) from those binary objects. On the second pass sliepnir.py records any call pairs involving the relative binary objects. The call chains are recorded into an list as a pair of pairs, [[bin:func]:[bin:func]], using the assigned identifier for the object and function. After the file is analyzed the call chains are printed by using the identifiers in the pair of pairs as an index into the lists of objects and functions. The call chains are printed out in the form of bin.func → bin.func. Since the software system to be analyzed was an API, sliepnir.py removed all calls that did not either involve the driver program or cross into the OpenGL API. Calls from the sample program into libc for example, were ignored by sliepnir.py. At this stage the special start transition pair "START" → main(), was added to the list of transitions.

A second tool, fenris.py (see section A.2), further distilled the data by processing the output from sliepnir.py. Fenris.py handled the multiple entries into the API by adding the "START" → "API_ENTRY" transitions and eliminated any sample software call chains that did not cross into the API. Calls internal to the API were simply recorded. Calls with their origins in the driver application that crossed into the API were pruned at the transition into the API. The example call chain below represents a theoretical call chain before it is pruned:
"main()" → "local call A" → "local call B" → "OpenGL call".
The second pass would remove main() and add the special start transition. The resulting call chain would be:
"START" → "local call B" → "OpenGL call".

The removal and pruning of the call chains reduced the amount of data to be processed and restricted the domain to OpenGL as much as possible.

4.2.3 Overview of Applications Used for Generating Profiles

A two pronged approach was used to profile the API. Large complex systems like complete games do not provide a complete breakdown of the functionalities of OpenGL by themselves. The first part of the approach was to use small programs that exercise individual functionalities of the API, giving a base by which more complex systems could be analyzed. For example, the API provides a method for setting the pixel size. At least one test case should exercise the pixel setting functionality, with a minimal number of other functionalities being exercised. The intent is to create a minimal set of the functionalities required for the functionality being executed in a particular run (Eisenbarth, Koschke, & Simon, 2001). There were a total of 51 sample programs profiled. These samples are intended as tutorial programs and, in general, cover only one aspect of OpenGL per sample (Woo et al., 1997). The code for each sample provides the same function names, "display" is the name of the drawing function for each sample for example. This provides uniform entry points into OpenGL. The sample programs were intended to break up the OpenGL functionalities. The programs range in purpose from setting up the view port and drawing functions to rendering complex textured and curved surfaces. There was some consideration given to determining the functionalities that make up that portion of OpenGL dealing with initialization but, due to the limited number of times those functionalities are executed, they are not ideal candidates for optimizations. While some of the initialization routines are called more than one time, in general, most are called only a single time.

GLHeretic is an open source version of Heretic, a commercial first person shooter type game. The player navigates though a rendered environment engaging different animated enemies. In addition to game play, GLHeretic provides options for adjusting various settings such as level of difficulty and effects. GLHeretic also provides operations for saving and loading games. There were a total of 22 profiles generated, and each profile attempts to exercise different functionalities through manipulation of options or interaction with the game environment. For example, one execution trace exercises the dynamic lighting effects.

The second open source game profiled is Neverball. The object of the game is to navigate a ball around a playing field and to gather a predetermined number of coins. The player then has to steer the ball to a goal area on the playing field. Neverball provides user adjustable settings for various graphical effects, as well as saving and replaying games. Unlike GLHeretic, Neverball had fewer options due in part to the different nature of the games. There were 12 total execution profiles derived from Neverball.

4.2.4 Stopping

The number of unique execution paths through a software system can be quite large, and a determination has to be made to stop generating any new execution profiles (Hall & Davis, 2004). Assuming that a software system has a main function that cannot be called recursively or by any other function, a system that has K functions has $K * (K - 1)$ possible explicit calls. If Y is the total number of explicit calls in the system, there are potentially Y^Y execution profiles possible in a system. In practice the number of execution profiles will be less than that. Ideally profiles would no longer be generated after the number of functionalities has stabilized and no new functionalities are being discovered and before the functionalities begin to deteriorate into individual functions. When the functionalities are complete, additional profiles will not yield further discoveries of equivalence classes (Hall, n.d.). Hall (1997) determined a statistical method for halting the gathering of traces. Empirically, stopping can be determined by examining the number of functionalities versus the number of test cases.

It can be determined by examining the graph from Figure 4 when new software systems were introduced into the data set. There is a minor but abrupt rise in the number of functionalities discovered with the introduction of each new

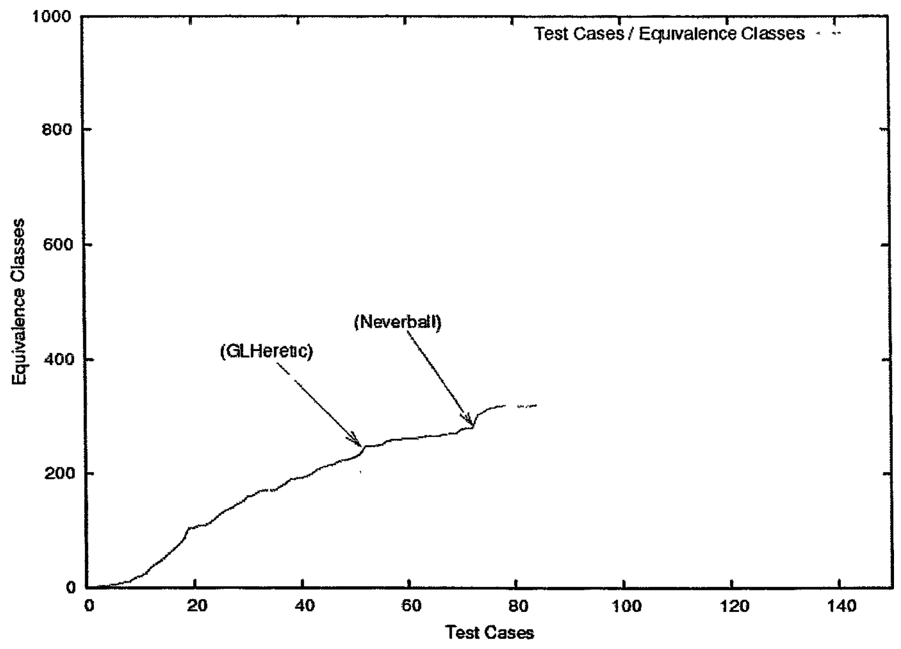


Figure 4: Comparison of New Functionalities versus Test Cases

source base. The introduction of the 22 execution traces from GLHeretic increased the number of functionalities discovered by 15, a 6.4% increase. The introduction of the 12 Neverball traces increased the number of functionalities by 24, an 8.6% increase. A portion of the new functionalities are a result of introduction of new entry point source modules into the analysis. The remaining newly discovered functionalities are OpenGL functionalities revealed from the introduction of the new traces into the analysis.

4.2.5 Functionalities Derived from OpenGL in Relation to Applications

Comparing the number of new functionalities generated per execution profile, the number of functionalities should begin to approach a limit. This curve can be used to help determine when a satisfactory number of profiles has been gathered. The curve in Figure 4 depicts the addition of test cases to the data set, versus the number of equivalence classes discovered. The curve has indeed started to approach a limit indicating the functionality set has stabilized and few

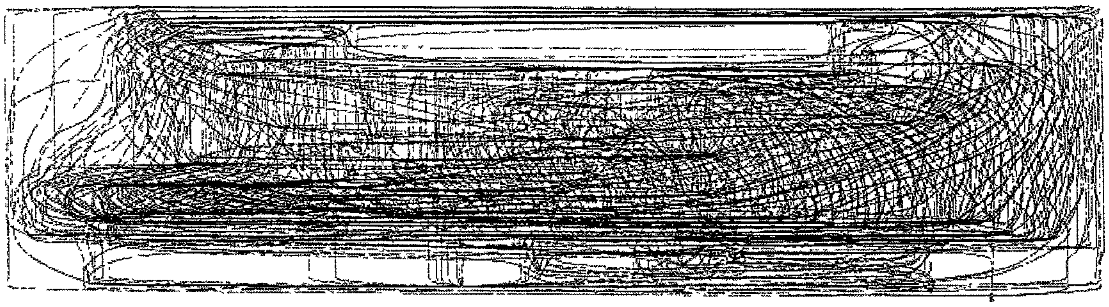


Figure 5: OpenGL Equivalence Class Digraph

functionalities remain undiscovered. There are the two small spikes in the curve that correspond to the addition of a new source base to the analysis. These spikes are expected when a new source base is introduced into the analysis. While the curve has begun to flatten and seems to be approaching a limit, adding execution profiles from a few more applications would be necessary to conclusively say that the functionality set has completely stabilized. The functionalities have stabilized enough however for a proof of concept.

4.2.6 Complexity of OpenGL

OpenGL is a large API. Figure 5 provides a glimpse into the complex nature of OpenGL. The figure depicts the equivalence class digraph derived from OpenGL without a level restriction. There are approximately 360 different nodes, and 1200 different edges between the nodes. A graph this complex provides little useful information about the software system (Hall, & Davis, 2004). Since the overall architecture of the software system is so complex, analyzing the overall system nearly impossible. A different approach will need to be taken. By focusing on levels of functionalities a simplified view of the system can be realized allowing analysis of the system to proceed.

4.3 Sample Programs and OpenGL Functionalities

The Digraph in Figure 6 depicts the level 1 functionalities of the OpenGL API discovered by the sample programs. There are 5 level 1 functionalities

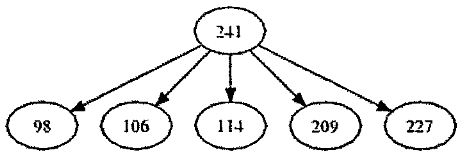


Figure 6: Level 1 Functionalities of OpenGL Based on Samples

discovered through the 51 sample programs. There is not a large fan out of functionalities at level 1. Digraphs generated from the earlier sample program execution traces had a larger number of level 1 functionalities. Figure 7 shows an example of this. Regardless of the order that the sample execution traces are analyzed, the last sample execution trace identified the same functionalities at the same levels. This demonstrates that the order in which the execution traces are analyzed does not affect the functionalities discovered. The early instability demonstrates the need for obeying the stopping rules discussed in section 4.2.4. The high fan out of functionalities from the earlier execution traces indicates that the functionalities had not yet stabilized. There is not much useful information that can be gleaned from Figure 7, however it does help to demonstrate that a number of execution traces are required before the functionalities of the software system will stabilize. The description of the functionalities for Figure 6 can be found in Table 3.

One of the difficulties in analyzing a library layer like OpenGL is that there are multiple legitimate entry points into the system. An application generally has one entry point. In a C language application the entry from the operating system to main() for example. Since there are multiple entries into a library, the root functionality can seem a bit arbitrary. An API does not serve a single program purpose in the traditional sense but instead provides a *service* to multiple programs. Each of which has its own purpose. This explains why the root of the program will tend to drift to some degree as different programs are analyzed.

Table 3: OpenGL Level 1 Funcs.[a] & Descs.[b] Based on Samples

Struct. Func. [c]	Description	Equiv. Class [d]
0	Callback Entry Points & Fog Rendering Calculations	241
1	Software Renderer	98
2	Translate (transform)	106
3	Matrix Manipulation(push and pop)	114
4	Setup and Execute Translation	209
5	Saving Matrix and Attribute State before a change	227

[a]Functionalities
[b]Descriptions
[c]Structural Functionality
[d]Equivalence Class

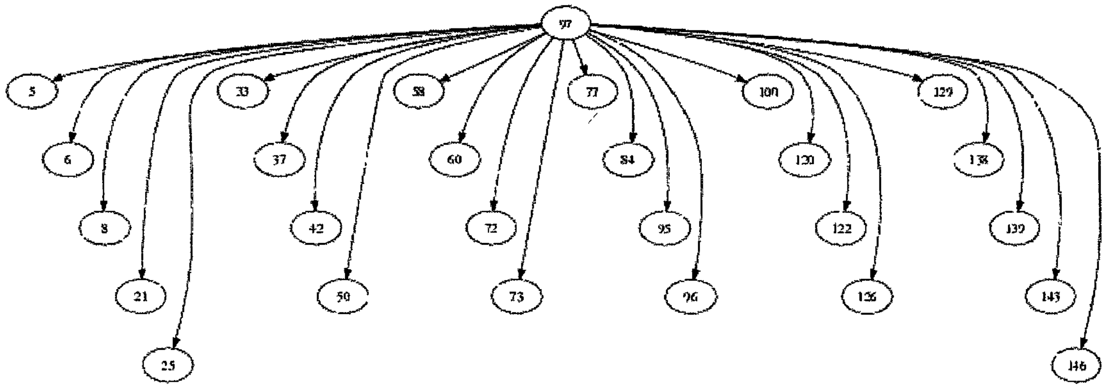


Figure 7: Level 1 Functionalities of OpenGL with Higher Fan Out

The level 1 functionalities are involved with rendering, display list creation, and translation. These are functionalities that a graphics API would be expected to provide as high level functionalities.

The digraph for level 2 has been split into two digraphs. In Figure 8 the functionalities with an edge from node 98 have been mostly removed, only nodes with an edge to 98 and another level 1 functionality are in included in Figure 8. Figure 9 provides a complete listing of the level 2 functionalities for node 98. Figures 8 and 9 depict the functionalities derived from the 51 sample programs at level 2. Several of the level 1 functionalities have a large number of level 2 sub-functionalities. Functionalities 98 and 247 in particular have large numbers of sub-functionalities. In Figure 8 node 241 is still the root functionality. Looking at the graph though, it appears that 98 has become the root. Graphviz attempts to minimize intersecting lines and has moved node 98 higher in the graph. At this level node 98 does have an edge to the root, indicating that the root functionality is one of its sub-functionalities.

The Digraph for Level 2 has 63 nodes and a complete listing of each functionality would not be practical. However, from the digraph for level 2, a few nodes of interest have begun to present themselves. The level 1 functionalities 106 and 209, have the functionality described by node 224 in common. Node 224 involves matrix translation and updating part of the OpenGL state machine. Other nodes of interest are 246, which is the functionality that creates display lists, and node 24, which is the functionality relating to creating and updating the stencil buffer. Another important functionality discovered at level 2 is represented by node 110, the functionality for initialization and startup of OpenGL. Most modules within the initialization functionality are exercised only infrequently, making the functionality a less than ideal candidate for optimizations, however it is still an important functionality. Table 4 provides a listing of some of the level 2 functionalities discovered from the sample programs.

Table 4: OpenGL Level 2 Funcs. & Descs. Based on Samples

Struct. Func.	Description	Equiv. Class
6	Translate and Update	244
7	Stencil Buffer	246
8	Initialization and Start up	110
9	Points Renderer and State Machine Validation	79
10	Rendering and Clipping	3
11	Polygon Render and Clipping	187
12	Setup Software Renderer for Triangles	152
13	Setup Software Renderer for Triangles Strips	211
14	Line Renderer and State Machine Validation	128

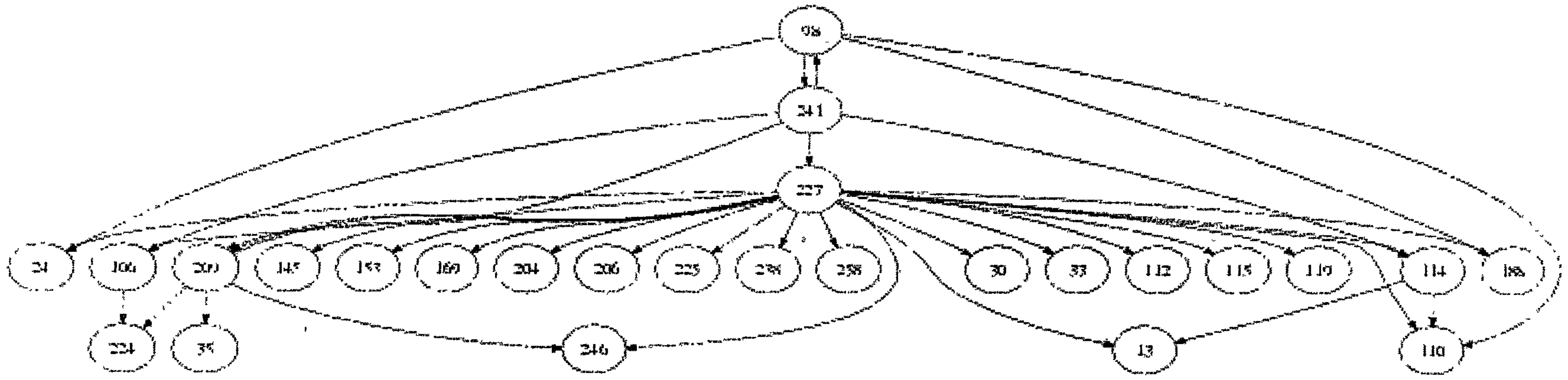


Figure 8: Level 2 Functionalities of OpenGL Based on Samples (part 1)

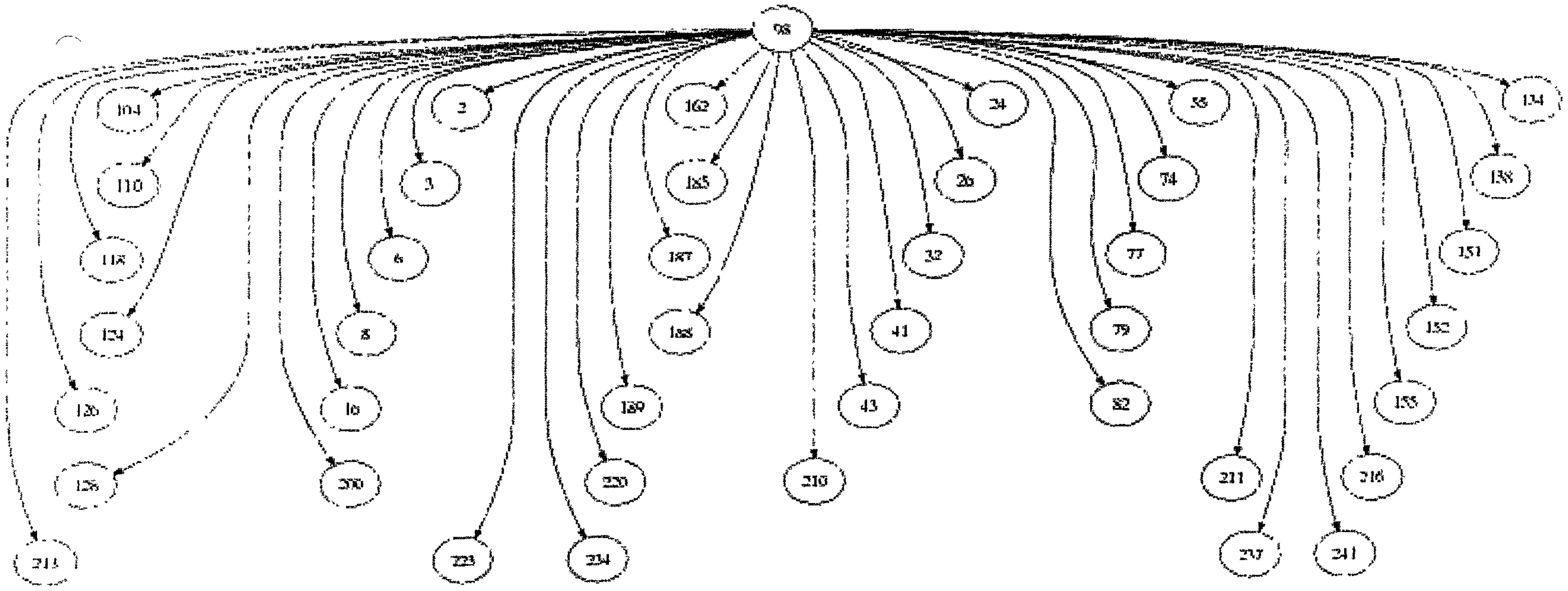


Figure 9: Level 2 Functionalities of OpenGL Based on Samples (part 2)

The Digraph at level 3 becomes large with approximately 235 nodes. Figure 10 presents partial digraphs the level 3. Some nodes have been pruned for readability. Node 110 was one of the major nodes pruned from the graph, the functionality is well understood without visualizing it. Although the digraph is quite large there is some clustering of nodes that warrant further investigation into their nature. There are three level 2 functionalities that have large numbers of level 3 sub-functionalities namely: 79, 110, and 128.

The functionality for node 110 has 36 sub-functionalities. The functionality has already been described as the initialization and startup of OpenGL. The modules that make up this functionality are infrequently called. Gullfaxi.py (see A.3) examines an execution trace and reports the number of times each module is called. Examining the results from this tool confirmed that most routines were called one time. Because the routines are called so infrequently, optimizing the start up is much more challenging, and the optimization investment would likely be better utilized elsewhere.

Functionalities 79 and 128 both have a fairly large number of sub-functionalities and are themselves sub-functionalities to several nodes. These functionalities have high fan out since they are both major components of the software renderer. Functionality 128 builds the polygons, and functionality 79 involves rendering points mostly in terms of polygon fill. Node 128 has 10 sub-functionalities as indicated by the digraph. Node 79 has 19 sub-functionalities. Both functionalities validate the state machine. Nodes 79 and 128 have edges from and to node 98, indicating that they are both integral to the software render of Mesa 3d.

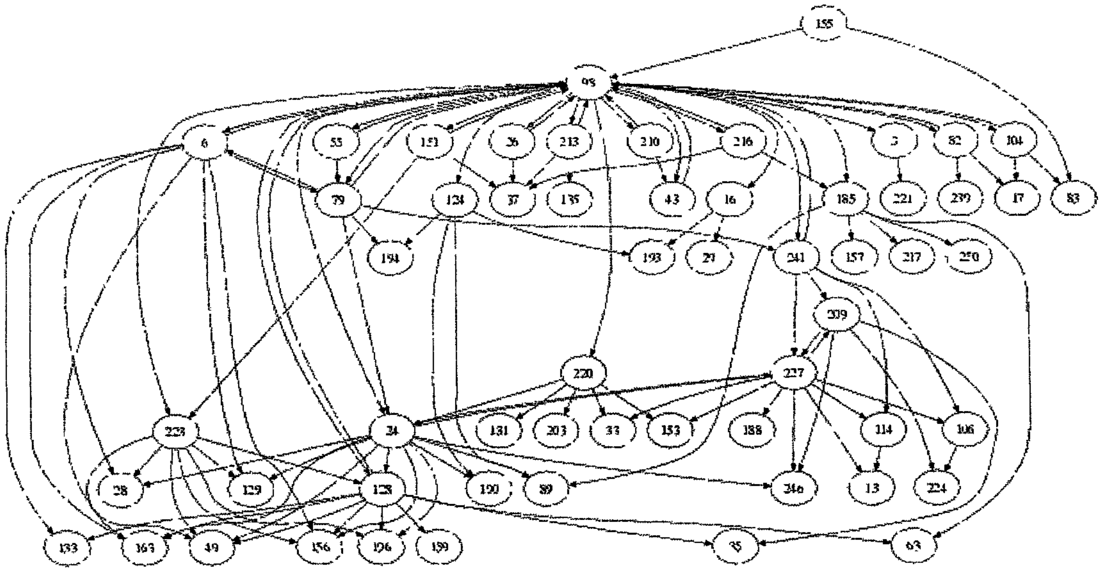
Another interesting cluster of functionalities in the Digraph is the interaction of the level 2 nodes: 3, 211, and 152. Each of which has edges to a subset of the nodes with edges from the level 2 functionality 187, indicating that they all may have very similar purposes. Functionality 42's purpose is to extract and interpolate colors, functionality 122 does setup for interpolation functions, and functionality 197 starts the drawing. Functionalities 42, 122, and 197 are all integral to rendering of polygons and triangles. Examining functionalities 3, 211, 152, and 187, it is apparent that each of these functionalities is involved with rendering of polygons. Table 5 gives a listing of the functionalities. The parent functionality listed in Table 5 can be found in Table 4. Figure 10 provides several partial graphs of the level 3 functionalities.

Table 5: OpenGL Level 3 Funcs. & Descs. Based on Samples

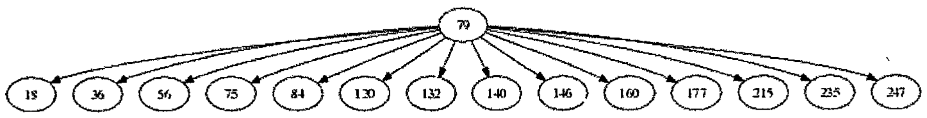
Struct. Func.	Description	Equiv. Class	Parent Func.
15	Flush Vertices	132	79
16	Interpolate Z (depth)	160	79
17	Interpolate fog	235	79
18	Polygon Stippling	18	79
19	Polygon Gesturing	120	79
20	Setup Interpolation Function	122	3 & 187
21	Draw/Render Start	197	3 & 187
22	Extract and Interpolate Color	42	3 & 187
23	Software Triangle Raster	81	152, 211 & 187

4.4 GLHeretic and OpenGL Functionalities

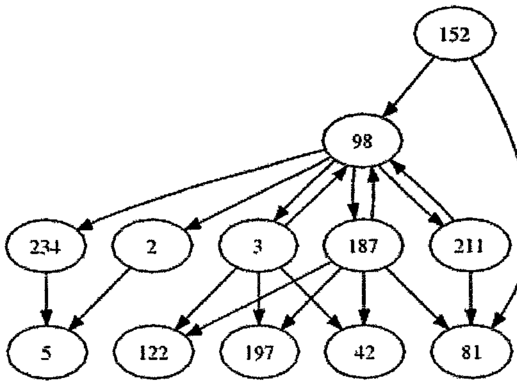
The second software system profiled was GLHeretic. The execution traces from the sample programs were included in the analysis of the execution traces from GLHeretic. Combining the execution traces together may not seem to be correct initially, but this allows the functionalities of OpenGL to be explored in terms of the game. If the GLHeretic data is examined without the execution traces from the samples, there are a total of 24 functionalities identified. When the execution traces from GLHeretic and the sample programs are combined, over 300 functionalities become evident. The low number of functionalities derived from the GLHeretic execution traces alone is explained by the organization of the game and its utilization of OpenGL. GLHeretic provides very few options in terms of the settings



(a) Partial Level 3 Functionalities



(b) Level 3 Functionalities of Node 79



(c) Level 3 Cluster of Rendering Functionalities

Figure 10: Level 3 Functionalities of OpenGL Based on Samples

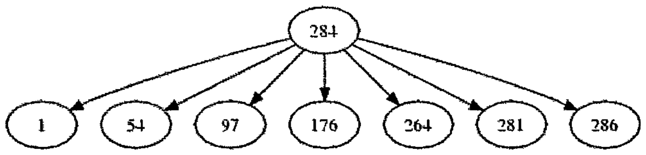


Figure 11: Level 1 Functionalities of OpenGL Based on GLHeretic

for graphics, allowing fewer unique execution paths through the API. With fewer execution paths, there will be fewer functionalities identified.

The Digraph depicted in Figure 11 shows the level 1 functionalities of GLHeretic. As noted in section 4.1, functionality information is cumulative across execution traces. Since the functionality information is cumulative, only the last execution trace from OpenGL was examined.

There are seven level 1 functionalities, and the root functionality has shifted into GLHeretic. The root functionality is related to drawing to the screen and the OpenGL interface callbacks for using the API. In examining Table 6 it appears that Nodes 264 and 281 seem to be very similar, as both deal with Begin/End; however, they each have distinct behavior. Functionality 264 is the entry point to the API, while 281 is used internally by the API.

Table 6: OpenGL Level 1 Funcs. & Descs. Based on GLHeretic

Struct. Func.	Description	Equiv. Class
0	Render Text	284
1	Bind Texture	1
2	Execute Render Pipeline	54
3	Render and State Machine Setup	97
4	Texture Coordinates	176
5	Begin/End for drawling lists	264
6	Setup Begin/End Function Pointers	281
7	Setup Function Pointers for Creating Vertices	286

There is a slightly higher fan out at level 1 for GLHeretic than was observed with the sample programs. GLHeretic has a different usage of the API compared to the sample programs, and this difference has caused some functionalities to move up

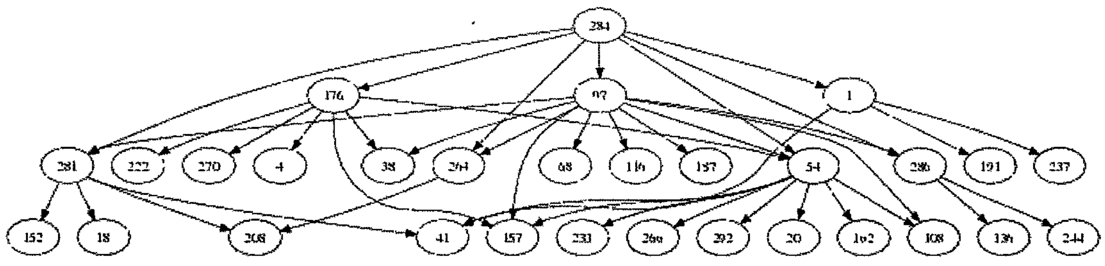


Figure 12: Level 2 Functionalities of OpenGL based on GLHeretic

to level 1 from some lower level. This movement is expected since the functionalities have not completely stabilized.

Nodes 54, 97, and 176 are significant functionalities found at level 1. Functionality 54 is the functionality running the graphics renderer pipeline. Functionality 54 also saves display lists, which allows a programmer to use the same geometry or state changes multiple times while only specifying them once (Woo et al., 1997). Functionality 97 represents the renderer and performs state machine setup. Functionality 176 is the texture coordinates functionality and is responsible for choosing the texture filter that will be used by the API. Functionality 97 initializes the software pipeline and validates the state machine. Functionality 176 is responsible for texturing objects, and then sending the textured objects to the render pipeline. Functionality 176 also is responsible for buffer swapping.

The level 2 digraph of GLHeretic has a total of 30 nodes, with 22 level 2 functionalities. There are fewer level 2 functionalities for GLHeretic than for the sample programs. This is because GLHeretic has less total coverage of the API, and hence, fewer functionalities are encountered.

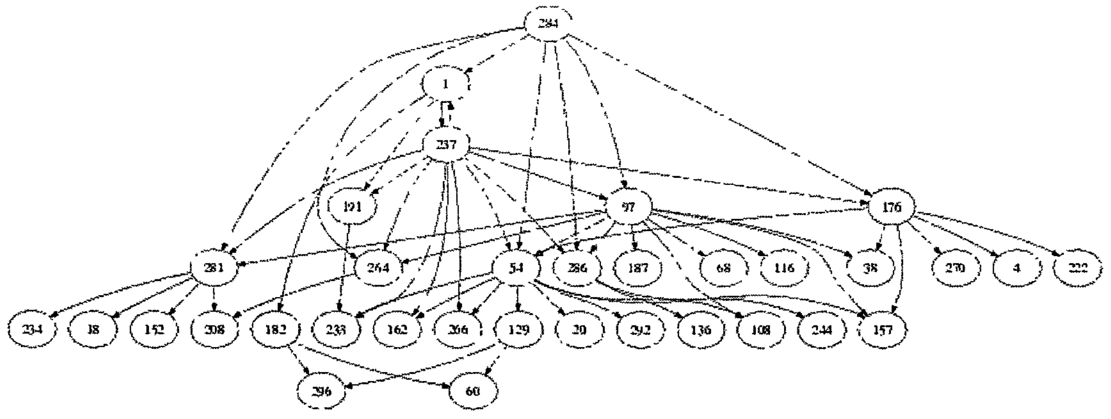
Figure 12 depicts the digraph of GLHeretic and the level 2 Functionalities. Node 54 has eight level 2 sub-functionalities, three of which are shared with several other functionalities. Functionality 97 has 10 level 2 sub-functionalities. Functionality 176 has six level 2 sub functionalities.

The level 2 functionality 157 is shared by level 1 functionalities 54, 97 and 176. Node 157 provides internal support for saving the OpenGL state machine and for allocating memory. Functionality 41, which is shared by level 1 functionalities 281 and 54 sets up the internal function pointers used by the API for flushing vertices. Functionality 108, which is shared by the level 1 functionalities 54 and 97, is the display list compiler responsible for building display lists. A partial listing of the level 2 functionalities can be found in Table 7.

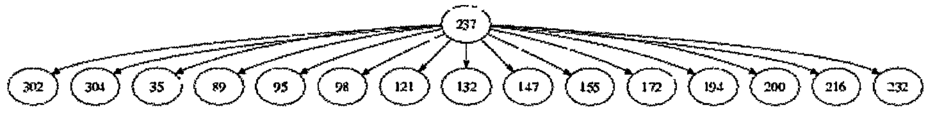
Table 7: OpenGL Level 2 Funcs. & Descs. Based on GLHeretic

Struct. Func.	Description	Equiv. Class	Parent Func.
8	"Compiler" for Building Display Lists	108	54 & 97
9	Execute Display List	292	54
10	setup glBegin through Function Pointer	152	281
11	Update State Machine	162	54
12	Setup glEnd through Function Pointer	208	264 & 281
13	Buffer Swap	222	176
14	Initializations and Start up	233	54
15	Interface to Xwindows & screen drawing	237	1
16	State Machine Update and Memory Allocation	157	54, 97, & 176
17	Setup for Vertex Flush Functions	41	54 & 281

More is understood about the level 1 functionalities By looking at the level 2 functionalities, and some thoughts about the API and its organization begin to become evident. At level 2 the graph for GLHeretic is still small enough to be manageable. It is easy to examine the graph and gain insight into the software system. Even at a relatively high level, examining the functionalities provides some evidence that by introducing threading into the API and placing the functionalities 176 and 54 in separate threads a performance gain may be realized.



(a) Partial Level 3 Functionalities



(b) Level 3 Functionalities of Node 237

Figure 13: Level 3 Functionalities of OpenGL Based on GLHeretic

At level 3 the digraph is still easily understood as only 16 new functionalities are introduced into the graph at that level. The functionality for node 237 has the largest number of sub-functionalities at this level. The unshared portion of sub-functionalities for node 237 have been listed in Figure 13. Functionality 237 has 26 total sub-functionalities, and 15 of the 16 functionalities discovered at level 3. It also has edges to each of the level 1 functionalities. The functionality represented by node 237 is a core functionality of OpenGL. It is the functionality that interfaces with the OS and does the actual drawing on the screen. The digraph for level 3 is depicted in Figure 13. Table 8 provides descriptions for some of the functionalities from level 3.

4.5 Neverball and OpenGL Functionalities

The last software system profiled for this study is Neverball. The execution traces gathered from Neverball were combined with the traces from the samples and GLHeretic. Reasons for combining execution traces from the sample programs and

Table 8: OpenGL Level 3 Funcs. & Descs. Based on GLHeretic

Struct. Func.	Description	Equiv. Class	Parent Func.
18	Texture Validation	232	237
19	Swap Buffer System Call	35	237
20	Viewport	121	237
21	ClearColor	147	237

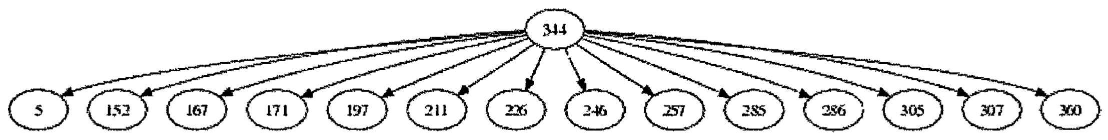


Figure 14: Level 1 Functionalities of OpenGL Based on Neverball

the larger software systems has already been addressed. However, the combining of the two *production* software systems has not. Combining the traces from each of the production systems allows for an examination of the differences in how each system uses the API and, conversely, which functionalities overlap across each system. As in section 4.4, only the last execution trace from Neverball was examined.

The digraph in Figure 14 depicts the level 1 functionalities of OpenGL derived from Neverball. Neverball does provide some of control of the graphic effects, for example shadows can be toggled on and off. The various options provided by Neverball were exercised during profiling.

The Level 0 functionality relates to the OpenGL render mode, and specifically feedbackbuffer mode. The level 0 functionality is represented by node 344. There are a total of 14 level 1 functionalities. Table 7 gives a partial listing of the level 1 functionalities. A complete listing is not necessary as several of the level 1 functionalities are related to OpenGL state validation and function pointer initialization and are very similar to ones already listed.

Examining the level 1 functionalities, there are several functionalities that are also level 1 and level 2 functionalities for GLHeretic. Functionalities 197, 246, and 257 were previously described in Table 6, section 4.4 as part of the level 1

Table 9: OpenGL Level 1 Funcs. & Descs. Based on Neverball

Struct. Func.	Description	Equiv. Class
0	OpenGL RenderMode Setup	344
1	State Machine validation for Vertices	5
2	Setup Orthographic Projection	152
3	Clear Color	167
4	Begin/End for Drawing List	197
5	“Compiler” for Building Display Lists	211
6	Setup Begin/End Function Pointers	246
7	Execute Render Pipeline	257
8	Point Renderer	286
9	Build Normals	305
10	Setup Function Pointers for Normals	260

functionalities discovered from GLHeretic. Functionality 211 has also been seen with GLHeretic as a level 2 functionality. The description for the functionality is in Table 7.

There are three additional level 1 functionalities identified by the Neverball traces. Functionality 152 has to do with setting up the 3D projection matrix for Neverball. Functionality 286 sets up the software point renderer while functionality 305 has to do with building normals. Neverball uses a much more robust lighting model compared to GLHeretic and uses normals for many of the lighting calculations.

A complete digraph of the level 2 functionalities has 104 nodes, with 124 edges between them. The graph has been broken into 3 parts for display, with digraphs for the sub-functionalities of nodes 286, and 211 depicted separately. Figure 15 lists both 286 and 211 for level 1 completeness.

In Figure 15 the nodes 257 and 305 have three sub-functionalities in common. Functionality 10 represents the callbacks the API provides as entries into the system, functionality 20 is memory and list cleanup, and functionality 29 is geometry transform functions.

Node 211 is depicted in Figure 17. In all, Functionality 11 has forty four

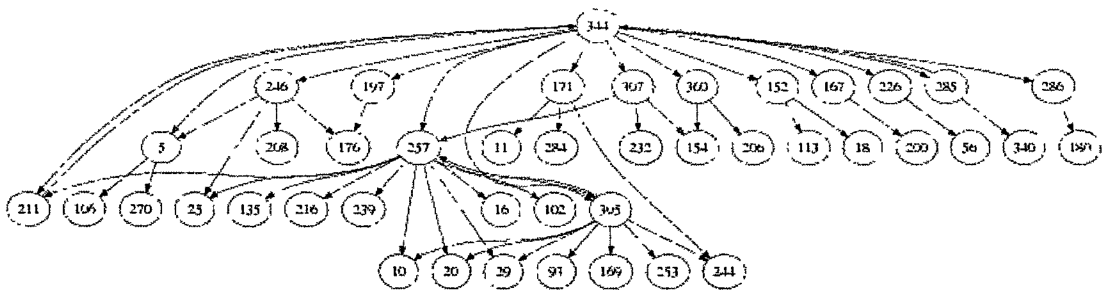


Figure 15: Level 2 Functionalities of OpenGL Based on Neverball (part 1)

sub-functionalities. Most of the sub-functionalities deal with clipping and rendering of a specific primitive, for example functionality 80 represents rendering line strips.

Figure 15 depicts the sub-functionalities of node 286 for which there are 23. Node 286 has 211 as a child node, and is itself a child node of 211, both functionality 211, and 286 are similar, as both deal with rendering. 211 is involved with vertices, and building geometry with primitives like triangle. Functionality 286 is involved with pixel operations polygon stippling, texturing, and color interpolation.

Table 10 lists part of the functionalities from Neverball at level 2. Unlike GLHeretic at this level, there is a significant increase in the number of functionalities generated by Neverball.

There are 122 nodes and 203 edges in the digraph for Neverball functionalities at level 3. The sub-functionalities that make up level 3 provide more detailed information about the level 2 functionalities. For example, present at this level is the function _mesa_pow which is the power function, used primarily for lighting calculations. The module _mesa_pow is the sole member of functionality and is represented by node 131. The parent functionality is 100 which represents fast lighting. Fast lighting is a technique used in graphics to quickly calculate specular highlights and is one of the few calculations in graphics that involves an exponent. Examining the graph, the power functionality is used only by the fast lighting functionality.

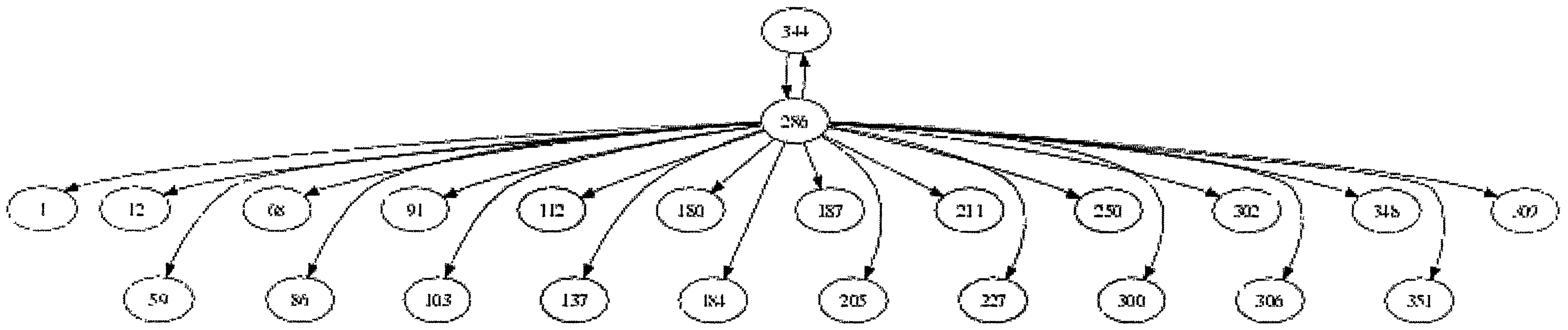


Figure 16: Level 2 Functionalities of OpenGL Based on Neverball (part 2)

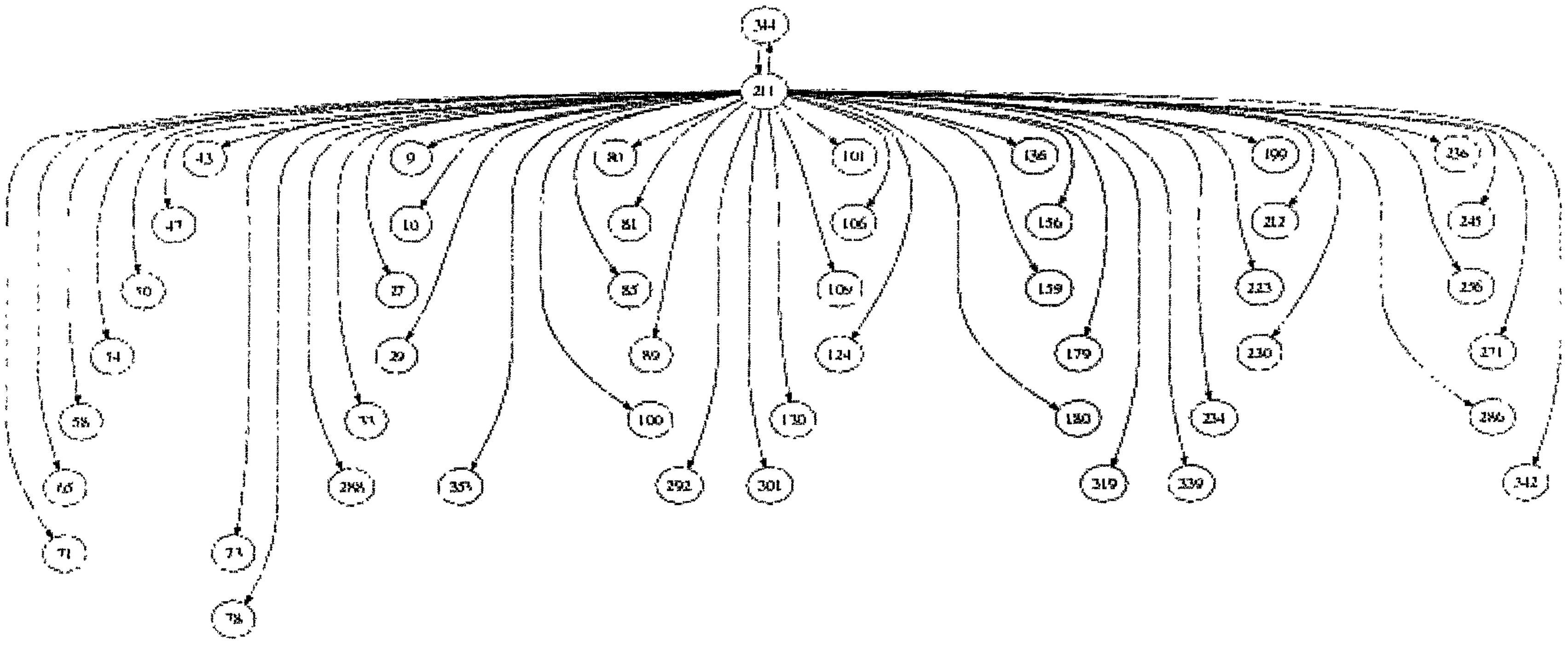


Figure 17: Level 2 Functionalities of OpenGL Based on Neverball (part 3)

Table 10: OpenGL Level 2 Funcs. & Descs. Based on Neverball

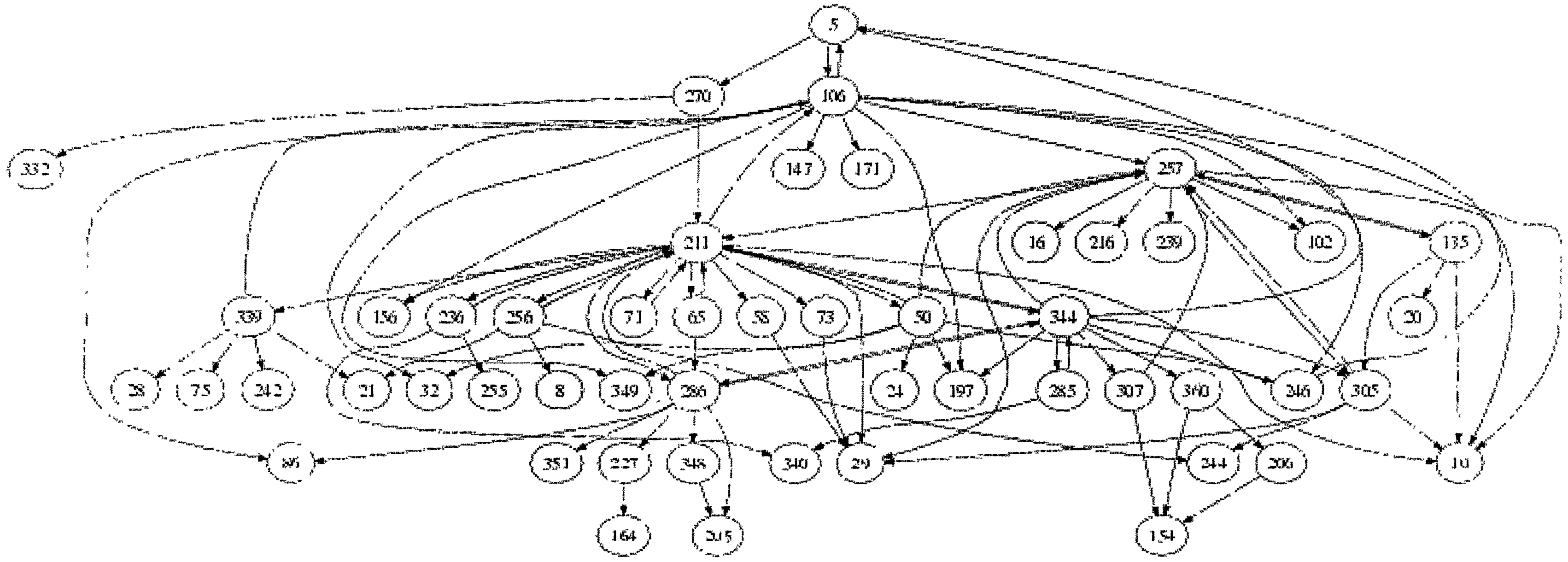
Struct. Func.	Description	Equiv. Class	Parent Func.
11	Software Raster Update	1	286
12	Color Interpolation	12	286
13	OpenGL Start up and Initialization	29	257 & 305
14	Fog Interpolate	59	286
15	Stencil	86	286
16	Update State Machine	102	257
17	Polygon Stipple	103	286
18	Clip	137	286
19	Validate and Run Renderer	211	286
20	Software Renderer for Lines and Triangles	216	257
21	Texture Internals	300	286
22	Depth Test	302	286
23	Clean Up on Shutdown	306	286
24	Clip and Render	43	211
25	Texture Polygons	47	211
26	Raster / Flush	65	211
27	Triangle Strip Clip and Render	9	211
28	Transform Normals	27	211
29	Lighting Pass	85	211
30	Light Table Validation and Update	89	211
31	Fast Lighting	100	211
32	Render Line Strips	80	211
33	Clip and Render Quad Strips	292	211
34	Matrix Scale and Rotate	159	211
35	Matrix Multiply and Invert	212	211
36	Invalidate State	230	211
37	Clip and Render Quad Vertices	236	211
38	Line Stipple	342	211
39	Rotate	339	211

A limited digraph of level 3 is depicted in Figure 18. Functionalities from level 1 and 2 that have no level 3 sub-functionality are removed. Functionality 211 has been separated to some degree as well to help with readability.

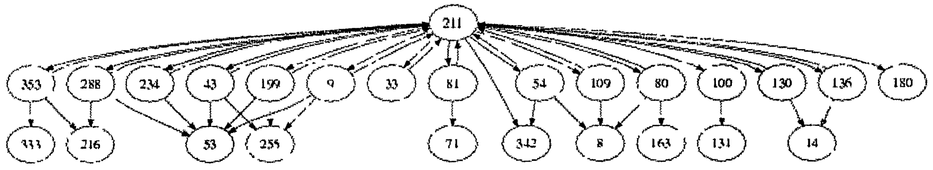
There are several of the nineteen new functionalities discovered at level 3 shared by level 2 functionalities. The level 2 functionalities associated with level 1 functionality 211 have 16 level 3 functionalities. Functionalities 255, 53, and 8, are some of the more interesting functionalities. The functionality represented by node 255 is interpolation and rendering of clipped polygons; 53 is triangle color; and 8 is raster line. Table 11 provides a listing of some of the functionalities of level 3.

Table 11: OpenGL Level 3 Funcs. & Descs. Based on Neverball

Struct. Func.	Description	Equiv. Class	Parent Func.
40	Triangle Color	53	9, 43, 199, 234, &288
41	Interpolate and Render Polygons	255	9, 43, 199, 236
42	Save Attributes and Information	32	50 & 106
43	Translate	75	339
44	Light	242	339
45	Matrix Operations	28	339
46	Raster Triangles	14	130 & 136
47	Quad RGBA	333	353
48	Render Unfilled Triangles	216	257, 288, & 353
49	Power Function	131	100



(a) Partial Level 3 Functionalities



(b) Level 3 Functionalities of Node 211

Figure 18: Level 3 Functionalities of OpenGL Based on Neverball

4.6 Summary of Applications

The OpenGL sample programs provided a very good way of breaking down the functionalities of OpenGL. The sample programs provided a base of 274 functionalities. The main purpose of the sample programs was not to provide enough data for analysis of the API by themselves, but rather to provide supplemental information about the nature of the API so that functionalities could be derived in conjunction with real use cases, in this instance the games. Each application had to be profiled to gather execution traces in order to derive the functionalities.

The tutorial programs had very little user controlled input so capturing traces was a simple matter of running them in the profiler. Each game had several options and game play scenarios to be profiled in order to have reasonable coverage of the source code. Profiling the games was more problematic. In order to profile the games, each option was exercised singularly in the profiler. The save mechanism provided by each game was used to profile game play. The game was saved at a point just before the event of interest was to be observed, and then the game was restarted in the profiler. The saved game would be reloaded at this point, and then the event would be triggered and profiled.

GLHeretic in comparison to Neverball is a much older game. The parent game Heretic was released in 1994. The game does not use as many features of the API as Neverball does. The numbers of level 1 functionalities provides some insight into the feature coverage of the API. GLHeretic has 7, while Neverball has twice that many with 14. GLHeretic is a sprite based game with extensive use of texture mapping of effects and the game environment. Neverball however takes advantage of stipple buffer to provide reflections and shadows, as well as using rendered objects as opposed to textures for the environment, etc. Comparing the various functionalities used by GLHeretic and Neverball, differences and commonalities in usage are apparent. GLHeretic and Neverball have some functionalities in common such as pipeline execution. Differences in usage also appear: Neverball makes use of stippling and stenciling, while GLHeretic tends to use more texturing techniques. Neverball also uses normals indicating that it takes advantage of 3d graphics, as opposed to the 2d used by GLHeretic.

The tutorial programs in combination with the games provided a breakdown of the functionalities of the API. By examining these functionalities and their relationships, a better understanding of the internal operations of the API is attained, allowing for development of new optimizations, tools and development techniques.

CHAPTER V

APPLICATION

5.1 Application

Now that the functionalities of OpenGL are identified, and a hierarchy has been established, can that information be used to improve the software system?

5.2 Threading

As has already been noted in Section 4.4, segregating the major functionalities of the API into separate threads could yield performance improvements. Equivalence class analysis provides a convenient way to start the process of assigning modules to threads and suggests which functionalities require locks (mutexes). Two functionalities begin to emerge from GLHeretic as candidates for threading. Functionalities 54 and 176 are the high level functionalities handling the major portions of the API. Functionality 54 is the level 1 functionality for running the software pipeline, and 176 handles the texturing of polygons, and filling the drawing buffer. These functionalities make good candidates because their behaviors complement each other. One runs the rendering pipeline, and the other swaps the display buffers, and feeds textures into the pipeline. The through put of the pipeline will be increased if both functionalities are allowed to run in parallel. The edges of the digraphs indicate dependency among modules, hence, where locks will most likely be required for safety. They also indicate which modules would be exclusive to a particular thread. Functionality 237 from the level 2 GLHeretic digraph may also prove to be a candidate to add to threading model. More research is required to make any determination however.

The same basic functionalities appear again with the Neverball execution traces. Functionalities 257 and 286 are the candidates for threading. Placing functionality 211, the compiler for display lists, in its own thread may also be a possibility. GLHeretic and Neverball both had the suggested functionalities as major high level functionalities. There was some difference in level for the display list compiler. However, the concept of threading the API is still valid. More software systems would need to be profiled to determine how to best implement the model.

As mentioned in section 1.1, there will always be some portion of the API that will be purely software. Threading will allow for parallel processing of different parts of the API which will provide a significant boost to a purely software implementation of the API, as well as software portions of any other hardware/software implementation. Threading will also show significant boost in performance if the threading model is able to utilize some of the multi-core processor architectures that are increasingly available. A threaded API more closely resembles the multi-core graphics solutions that are in most dedicated graphics hardware currently available.

5.3 Functionality Level Profiling

Recalling that a module may be shared by multiple functionalities, identifying functionalities by module is not possible. However, each functionality will always have at least one unique transition (Hall & Davis, 2004). This unique transition can be used as an identifier for a functionality. By instrumenting the unique call of each functionality with a counter in the software system source code, a simple profiling system at the functionality level can be created.

The ability to identify a unique transition for each equivalence class allows for instrumentation information to be gathered about entire functionalities. It is guaranteed that if one module of a functionality is executed, all other modules will be as well (Hall & Davis, 2004). It is not certain how many times each module in a functionality will be exercised.

Since a functionality is a group of modules always executing together, the functionality profiler has uses in optimizing code, for suggesting a caching strategy, and in debugging (Hall & Davis, 2004).

5.4 Caching

Hall and Davis (2004) suggests that a caching strategy may become apparent by examining the digraph. A strategy for cache miss reduction can be developed by combining data from a functionality profiler and information about call frequency. Caching is a containment problem in that the limited space provided by the cache(s) will not be able to hold the entire program. Every time a datum is loaded into the cache, it forces another datum to be flushed from the cache. There are various strategies designed to help insure that required data are in the cache and that frequently used data are not removed. By examining information on the frequency of calls and cache misses, the cache can be preloaded with the required functionalities helping to reduce cache misses.

The execution traces gathered contain information about the number of times a parent function called a child. A tool was written, gullfaxi.py (see A.3), to extract the call frequency data. By examining this data and the functionality analysis data, determinations can be made on which modules to preload and when to preload them. For example, in the Neverball samples functionality 131 was identified to be the power functionality consisting of the module _mesa_pow, which is used for lighting calculations by functionality 100. When functionality 100 is entered, a cache preloader can preload _mesa_pow. This is a rather obvious example, but it does illustrate the point.

Knowing that a functionality is a group of modules executed together, another approach for reducing cache misses presents itself. Whenever a target module of an identifying transition is loaded, the remaining members of the functionality could be loaded into the cache. Since a module can appear in more than one functionality, it is necessary to insure that the correct functionality is identified so that the correct remaining modules of the functionality are loaded. A way of identifying functionalities by unique transitions has already been discussed in section 5.3. Using the unique transition profiler information and standard profiler information about cache misses, the modules of the functionality that are causing a cache miss can be identified. Once the modules are identified, attempts to rectify the cache misses can be made. Cache misses are unavoidable once a program is larger than the available cache. The goal is not to eliminate the misses all together, but rather to minimize as much as possible the most expensive cache misses. This caching strategy also works to flush functionalities that are no longer needed. Knowing what to flush from the cache is as important as what to load.

This caching strategy has benefits in loading and flushing, both of which are important to good cache performance. This strategy has the potential to provide a significant performance boost.

5.5 Inlining

The inliner is an integral part of the compiler optimization step (Hazelwood & Groove, 2003). Inlining has several nice effects: it allows other optimizations to run on larger blocks of code and it eliminate some branches. Branches are expensive in terms of the time required to load the requested memory, plus they generally invalidate instructions that are already in the pipeline. When inlining, the compiler simply replaces a function call with the actual code of the function being called. This has the nice effect of removing a jump and in some cases, actually preventing an I-cache miss, since the function does not actually have to be loaded. There is a limit to how far inlining can proceed before the returns are negligible, or detrimental to performance. The *inline* keyword is really a hint to the compiler and not guaranteed to have an effect. Examining functionalities can provide insight into how to use the keyword to get maximum efficacy out of the optimization. The _mesa_pow module is a good candidate for inlining as few modules utilize it. Others that would be good candidates are the state machine update functions like save_attrib_2_3 for example. Module save_attrib_2_3 is called approximately 30,000 times, with approximately 95% of the call from one parent, during a single execution of Neverball.

By examining the functionalities and the layout of a program in memory, a determination can be made about the nature of the branches between each module of a functionality, and its sub-functionalities. By selectively inlining modules either from within the functionality or from sub-functionalities the code speed could be improved.

5.6 Functionality Driven Compiles

The functionalities from each graphics system could be used to build a database of OpenGL functionalities. This database could aid in the optimization of new graphics software systems. Figure 19 depicts a build pipeline using a functionality database. Using the functionality database, the code preprocessor adds directives to the compiler and linker into the source code to enable different optimizations. Using the functionality information to direct caching and inlining as has already been discussed could be an important part of the directives to the compiler and linker. Profile directed builds have the potential to greatly increase the execution speed.

5.7 Architecture

The information provided by equivalence class analysis does not have to be used solely for tuning and optimizations. Siff and Reps (1999) use concept analysis to identify a modularization strategy for legacy software systems. This technique provides a similar hierarchical view of the software system. Functionality analysis can readily be applied to examining the architecture of the software system. The digraph can be used to suggest how to proceed when creating a new architecture in cases of a system rewrite. If the API was to be refactored into an object oriented system in the C++ language, the digraph could be used to determine classes and the relationships between classes.

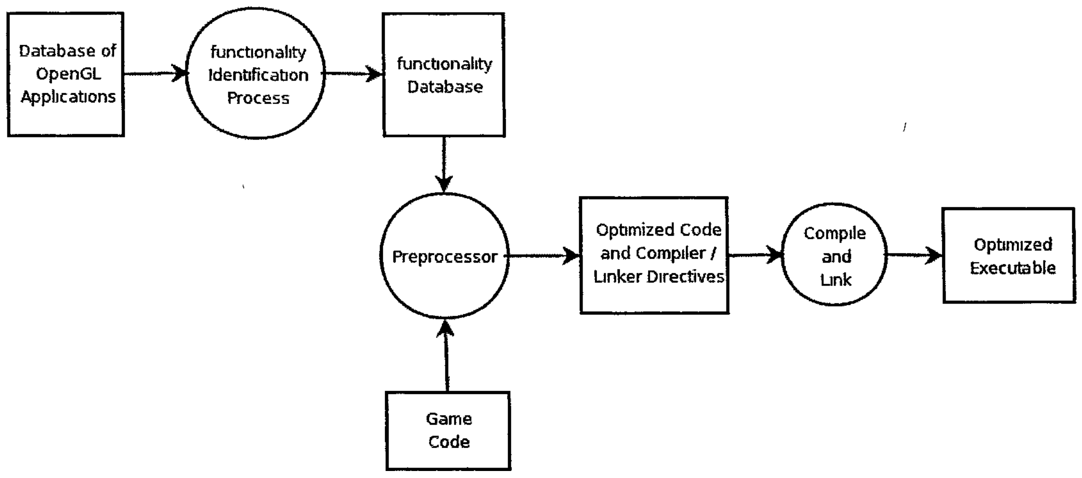

Database of OpenGL Applications
functionality Identification Process
functionality Database
Preprocessor
Optimized Code and Compiler / Linker Directives
Compile and Link
Optimized Executable
Game Code

Figure 19: Functionality Driven Optimized Build Pipeline

This type of analysis could be used to confirm that a software systems architecture meets the design criteria set forth in the design documentation (Hall, n.d.).

Reverse engineering is particularly useful when refactoring a system. Kuhn, Greevy, and Gîrba (2005) suggest a method for examining the dynamic execution traces. Using latent semantic analysis to analyze source bases, similar methods are clustered together (Kuhn, 2005). This information can then be used in a refactoring effort. Functionality analysis is able to perform the same kinds of clustering.

5.8 Tools

A commercial security system has been developed using functionality analysis. By noting changes in how the activities of the Linux kernel deviated from a standard pattern of behavior intrusion attempts into a system were able to be detected and prevented (Elbaum & Munson, 1999).

This type of analysis could also be used to implement a debugging tool for complex systems. Bugs are a reality of software and some can be fiendishly hard to locate and quash. It may be possible to determine which module the bug is located in by using this analysis to monitor a system. It may be possible to locate bug in a system by noting changes in the membership of modules that define a functionality, or by the addition/subtraction of a functionality either as sub-functionality, or as a change in the level of the functionality.

CHAPTER VI

CONCLUSION

A technique for profiling a middle, or library, layer like the OpenGL API has been developed. Execution traces were gathered from OpenGL and processed to identify functionalities provided by the API and the relationship among those functionalities. The functionalities were identified by using the modules that make up the functionality, and examining the source code of the modules. By identifying the functionalities of the API and their interdependent relationships, further ideas of how to optimize the API have been suggested, in particular threading, cache pre-loading strategy, and inlining. New tools have also been theorized, including debuggers and architecture analysis tools.

Most of the current optimization techniques involve some form of trade off between the quality of the graphic displayed and the speed of the application. Finding an acceptable balance between the quality and speed is a challenge. Improvements to the API do not have the same issue, although increasing the architectural complexity of the API introduces its own issues. The changes to the API cannot cause it to become too difficult for developers to understand and use.

Whether a software system that utilizes OpenGL is for entertainment or something more serious, performance is critical. New optimization can be indeed be found using equivalence class analysis. To utilize the technology to its fullest, new tools will have to be created. In the future a dynamic system for performing functionality analysis would be critical.

The optimizations suggested are, for the most part, complex. The inlining strategy is by far the simplest, with simple analysis, minor editing of the source code, and a recompile of the API adequate to implement the optimization. Threading and pre-caching would require a more significant investment. Investments in time, tools, and analysis would be needed to determine the best strategy for proceeding. Pre-caching would require cache analysis to determine the scenarios in which a particular functionality causes cache misses.

The observations on optimizations, tools, and refactoring observations in this paper are by no means the limit of what can be accomplished by examining functionalities of a software system. There are many other uses: and these are just a few of the possibilities.

APPENDIX A

SOURCE CODE LISTINGS

A.1 sleipnir.py

```
#!/usr/bin/python
#**********************************************************
#JOE SULLIVAN
#23/12/2006
#This script will parse an call graph output from callgrind
#and process each function into a set of parent -> child
#calls.
#AREGUMENTS: argv[0] this script
#         argv[1] name of input file,
#         argv[2] name of the concatenated data file to
#                add parsed data to
#
#**********************************************************

#**********************************************************
# TODO #1: BUILD A LIST CORELATING FUNCTIONS NAMES AND
#          NUMBERS (done)

# TODO #2: BUILD A LIST OF CORELATING OBJECT NAMES AND
#          NUMBERS, LIST IS LIMITED TO OBJECTS OF INTEREST,
#          EXAMPLE OpenGL LIBRARY (done)

# TODO #3  MAIN TO DEAL WITH THE SPECIAL CASE FUNCTION:
#          main()

# TODO #4: NEED TO ADD A CHECK SO THAT ALL THE MINIMAL # OF
#          CALLS INVOLVONG SDL ARE RECORDED.  ALGORITHM FOR
#          THIS CHECK IS GOING TO ASSUME THAT OpenGL DOES NOT
#          MAKE ANY CALLS TO SDL.  (done)
#
#          dealing with things like
```

```
#
#
#                        sdl-->sdl
#                       /
#                      /
#          game--->sdl
#                     \
#                      \
#                        sdl-->gl
#
#          poses some problem.  To be sure that situation
#          like the above are handled, the algorithm has
#          to be sure and save any new "PARENT -> CHILD"
#          pairs encountered while processing the list of
#          functions. Any new pairs can be placed on the
#          list to be dealt with later. (done)

# TODO #5: SEVERAL PARENT CHILD CALLS SEEM TO BE
#          REPEASTED IN THE CALLGRIND DATA, NEED TO FILTER
#          THOSE OUT. (done)
#
#***********************************************************

#-----------------------------------------------------------
#INCLUDES
import sys
import os
import re
import copy
import time
#-----------------------------------------------------------

#-----------------------------------------------------------
#GLOBALS
OBJ_LIST = []
OBJ_COUNT = 0

FUNC_LIST = []
FUNC_COUNT = 0

MASTER_LIST = []
MASTER_IDX = 0

SDL_LIST = []
SDL_IDX = 0
```

```
SDL_REG = []
#-----------------------------------------------------------

#-----------------------------------------------------------
#REGULAR EXPESSIONS
glheretic = re.compile('glheretic')
neverball = re.compile('neverball')
binname = re.compile('sample')
libgl = re.compile('libGL.so\.[0-9]\.[0-9]\.[0-9]+')
libsdl = re.compile('libSDL-[0-9]\.[0-9]\.so\.[0-9]\\
                    .[0-9][0-9]\.[0-9]')
#-----------------------------------------------------------

#-----------------------------------------------------------
#REMOVE DUPLICATES FROM THE LIST
#
#INPUT: LIST TO REMOVE DUPS FROM
#CODE IS FROM TUG (Prof. Davis and Hall)
def nodups(list):
    new_list = []
    for i in l:
        if i not in z:
            z.append(i)
    return z

#-----------------------------------------------------------
#FORMATED OUTPUT FUNCTION
#
#INPUT:
#ofile:   name of the output file
#tfile:   name of the file containing the concatenated
#         output of all data from a profile seesion of
#         an application
def output_list(tfile):
     outstr = []

     try:
          catfile = open(tfile, 'a')

     except IOError, e:
          print "could not open file: ", e, "exiting"
          return False

     try:
          catfile.write(str(len(globals()['MASTER_LIST'])))
          catfile.write('\n')
        for x in range(len(globals()['MASTER_LIST'])):
            outstr = globals()['OBJ_LIST'][int(globals()\
                            ['MASTER_LIST'][x][0])][1] + '.' + \
                    globals()['FUNC_LIST'][int(globals()\
                            ['MASTER_LIST'][x][1])][1] + ' : ' + \
                    globals()['OBJ_LIST'][int(globals()\
                            ['MASTER_LIST'][x][2])][1] + '.' + \
                    globals()['FUNC_LIST'][int(globals()\
                            ['MASTER_LIST'][x][3])][1]
            catfile.write(outstr)
            catfile.write("\n")


    except IOError, e:
        print "could not output to file: ", e, exiting
        return False
    except IndexError, e:
        print "index out of range error: ", e
        return False

    catfile.write("\n");
    return True
#-----------------------------------------------------------


#-----------------------------------------------------------
#PRINT RESULTS TO STDOUT
#
#INPUT: VOID
#
def output():
    outstr = []

    try:
        print(str(len(globals()['MASTER_LIST'])))

        for x in range(len(globals()['MASTER_LIST'])):
            outstr = globals()['OBJ_LIST'][int(globals()\
                            ['MASTER_LIST'][x][0])][1] + '.' + \
                    globals()['FUNC_LIST'][int(globals()\
                            ['MASTER_LIST'][x][1])][1] + ' : ' + \
                    globals()['OBJ_LIST'][int(globals()\
                            ['MASTER_LIST'][x][2])][1] + '.' + \
                    globals()['FUNC_LIST'][int(globals()\
                            ['MASTER_LIST'][x][3])][1]
```

```
                print(outstr)


        except IOError, e:
            print "could not output to file: ", e, exiting
            return False
        except IndexError, e:
            print "index out of range error: ", e
            return False

        return True
#---------------------------------------------------------

#---------------------------------------------------------
#RETURN TRUE IF A OBJECT IS AN OBJECT OF INTEREST
#
#INPUT ARGS:  num
#num:  object number assigned to binary by callgrind
def target_obj(num):
    try:
        x = OBJ_LIST[int(num)][0]
    except IndexError, e:
        return False
    return True
#---------------------------------------------------------


#---------------------------------------------------------
#RETURN THE BINARY NUMBER OF AN OBJECT ASSIGNED BY CALLGRIND
#
#INPUT ARGS: bname
#bname:  name of a binary
def lookup_binary_number(bname):
     for x in range(len(OBJ_LIST)):
          try:
               if(OBJ_LIST[x][1] == bname):
                    return(OBJ_LIST[x][0])
          except:
               pass
#---------------------------------------------------------


#---------------------------------------------------------
#RETURN A THE BINARY ABBREVIATION FOR THE GIVEN OBJECT NAME
#
#INPUT ARGS: obj_name
```

```
#obj_name: name of the object
def lookup_brief_name(obj_name):
     name = []
     if(glheretic.match(obj_name)):
          name = 'GLH'
     if(neverball.match(obj_name)):
          name = 'NVB'
     if(libgl.match(obj_name)):
          name = 'oGL'
     if(libsdl.match(obj_name)):
          name = 'SDL'
     if(binname.match(obj_name)):
          name = 'TPROJ'
     return name
#-----------------------------------------------------------


#-----------------------------------------------------------
#RETURN THE OBJECT NUMBER FROM A LINE OF INPUT
#
#INPUT ARGS: line
#line: a line of input from callgrind output
def obj_num(line):
     num = ''
     spl_str = []

     #TOKENIZE THE STRING AND START PROCESSING
     spl_str = line.split(" ")

     #DETERMINE THE CALLGRIND ASSIGNED NUMBER OF THE OBJECT
     num = spl_str[0][spl_str[0].index('(')+1 : -1]

     return num
#-----------------------------------------------------------


#-----------------------------------------------------------
#ADD A NEW "OBJECT" (function, file, or binary) TO THE
#APROPRIATE LIST
#
#INPUT:    o_num, o_name, OB_TYPE
#o_num:    the number of the "OBJECT"  assinged by
#          callgrind
#o_name:   the name of the "OBJECT"
#OB_TYPE:  the "OBJECT" type
def add_name_pair(o_num, o_name, OB_TYPE):
    bin_name = []

    #FUNCTIONS
    if(OB_TYPE == "FUNC"):
        if(o_name[-2:] == "()"):
            o_name = o_name[:-2]
        x_factor = int(o_num) - globals()['FUNC_COUNT']
     while(x_factor >= 0):
            globals()['FUNC_LIST'].append([])
         x_factor = x_factor - 1
         globals()['FUNC_COUNT'] = globals()\
                   ['FUNC_COUNT'] + 1
        globals()['FUNC_LIST'][int(o_num)].append\
                   (copy.deepcopy(o_num))
          globals()['FUNC_LIST'][int(o_num)].append\
                    (copy.deepcopy(o_name))


    #BINARIES
    if(OB_TYPE == "BIN"):
        bin_name = o_name.split('/')
        x_factor = int(o_num) - globals()['OBJ_COUNT']
        while(x_factor >= 0):
            globals()['OBJ_LIST'].append([])
            x_factor = x_factor - 1
            globals()['OBJ_COUNT'] = globals()\
                      ['OBJ_COUNT'] + 1

        if((glheretic.match(bin_name[-1]))
        or (neverball.match(bin_name[-1]))
        or (libsdl.match(bin_name[-1]))
        or (libgl.match(bin_name[-1]))
        or (binname.match(bin_name[-1]))):
            globals()['OBJ_LIST'][int(o_num)].append\
                     (copy.deepcopy(o_num))
            globals()['OBJ_LIST'][int(o_num)].\
                    append(copy.deepcopy\
                          (lookup_brief_name(bin_name[-1])))
#-----------------------------------------------------------


#-----------------------------------------------------------
#PROCESS A LINE OF THE FILE BASED ON THE TYPE
#
#INPUT ARGS: line, OB_TYPE
#line:     a line of input from callgrind output
```

```
#OB_TYPE:  the type of the object
def process_line(line, OB_TYPE):
     num = ''
     name = ''
     spl_str = []

        #TOKENIZE THE STRING AND START PROCESSING
        spl_str = line.split(" ")
        num = spl_str[0][spl_str[0].index('(')+1 : -1]

        #CHECK FOR A NAME FOR THIS FILE
     if(spl_str[0] != spl_str[-1]):
            if(spl_str[-1][-1] == ')'):
                name = spl_str[1].split("(")
                name = name[0]
            else:
                name = spl_str[-1]

            if(len(name) != 0):
                #ADD TO THE APROPRIATE LIST
          add_name_pair(num, name, OB_TYPE)
#-----------------------------------------------------------


#-----------------------------------------------------------
#PARSE THE callgrind OUTPUT AND GENERATE FUNCTION AND
#BINARY LISTS
#
#INPUT ARGS: infile
#infile: the data file generated by a single profiled
#execution
def build_lists(indat):
     os.lseek(indat.fileno(), 0, 0)
     for line in indat.read().split("\n"):
          if(len(line) == 0):
                    continue
          if((line[:2] == "ob") or (line[:3] == "cob")):
                    process_line(line, 'BIN')
          if((line[:2] == "fn") or (line[:3] == "cfn")):
                    process_line(line, 'FUNC')
     return True
#-----------------------------------------------------------


#-----------------------------------------------------------
#CHECK A CALL CHAIN FOR AN SDL CHILD, ADD ANY UNIQUE #
```

```
#CHILDREN TO GAME->SDL LIST
#
#INPUT ARGS clist, PARENT_OBJ, sdl
#clist:       the chain chain pair to add to list
#PARENT_OBJ:  object number of the parent
#sdl:         object number for sdl
def check_sdl_chain(clist, PARENT_OBJ, sdl):
     pobj_num = clist[0]
     cobj_num = clist[2]
     NEW_SDL_CHAIN = True
     ret = False
     #IF CHILD IS IN libSDL, SAVE TO SDL LIST
     if((cobj_num == sdl)
     and (PARENT_OBJ == True)
     and (pobj_num != sdl)):
          ret = True
          if(globals()['SDL_IDX'] == 0):
                    globals()['SDL_LIST'].append\\
                    (copy.deepcopy(clist))
                    globals()['SDL_IDX'] = \\
                    globals()['SDL_IDX'] + 1
                    NEW_SDL_CHAIN = False
          else:
                    for x in range(len(globals()\\
                                       ['SDL_LIST'])):
                        if(clist == globals()\\
                           ['SDL_LIST'][x]):
                            NEW_SDL_CHAIN = False

          if(NEW_SDL_CHAIN == True):
                    globals()['SDL_LIST'].append\\
                    (copy.deepcopy(clist))
                    globals()['SDL_IDX'] = \\
                    globals()['SDL_IDX'] + 1
     return ret
#-----------------------------------------------------------

#-----------------------------------------------------------
#ADD A CALL CHAIN PAIR TO MASTER_LIST AS LONG AS IT IS
#UNIQUE
#
#INPUT ARGS:   clist
#clist:        the call chain pair to add to list
def check_app_chain(clist):
     NEW_APP_CHAIN = True
     if(globals()['MASTER_IDX'] == 0):
          globals()['MASTER_LIST'].append(copy.deepcopy(clist))
          globals()['MASTER_IDX'] = globals()['MASTER_IDX'] + 1
          NEW_APP_CHAIN = False
     else:
          for x in range(len(globals()['MASTER_LIST'])):
               if(clist == globals()['MASTER_LIST'][x]):
                    NEW_APP_CHAIN = False

     if(NEW_APP_CHAIN == True):
          globals()['MASTER_LIST'].append(copy.deepcopy(clist))
          globals()['MASTER_IDX'] = globals()['MASTER_IDX'] + 1
#-----------------------------------------------------------

#-----------------------------------------------------------
#ADD AN EXTENDED SDL CALL CHAIN TO SDL_LIST
#
#INPUT ARGS: chain
#chain:  the new chain to add to the list
def extend_chain(chain):
     globals()['SDL_LIST'].append(copy.deepcopy(chain))
     globals()['SDL_IDX'] = globals()['SDL_IDX'] + 1
#-----------------------------------------------------------

#-----------------------------------------------------------
#APPEND A GAME->SDL...->GL CALL CHAINS TO THE MASTER_LIST
#
#INPUT ARGS: chain
#chain:  call chain to be added to master list
def add_chain(chain):
     x = 0
     temp = []
     while(x < len(chain)-3):
          temp = [ chain[x], chain[x+1], chain[x+2], chain[x+3] ]
          MASTER_LIST.append(copy.deepcopy(temp))
          globals()['MASTER_IDX'] = globals()['MASTER_IDX'] + 1
          x = x + 2
#-----------------------------------------------------------

#-----------------------------------------------------------
def add_reject(chain):
          globals()['SDL_REG'].append(copy.deepcopy(chain))
#-----------------------------------------------------------

#-----------------------------------------------------------
def check_reject(chain):
     try:
          globals()['SDL_REG'].index(chain)
          return True
     except:
          return False
#-----------------------------------------------------------


#-----------------------------------------------------------
#ITERATE THOUGH THE SDL_LIST, CHECKING FOR A OpenGL
#TEMINATING CALL
#
#INPUT ARGS:  index, indat
#index        index into SDL_LIST, lookup for list member
#to process
#indat        file pointer
def close_sdl(index, indat):
     pobj = ''
     conj = ''
     HP = False
     HC = False
     new_chain = []
     temp = []

     temp =[globals()['SDL_LIST'][index][-4],\
     globals()['SDL_LIST'][index][-3],\
     globals()['SDL_LIST'][index][-2],\
        globals()['SDL_LIST'][index][-1]]

     if(check_reject(temp) == True):
          return True

     del temp[:]

     #MOVE CURSOR TO START OF FILE
     os.lseek(indat.fileno(), 0, 0)

     for line in indat.read().split("\n"):
          if(len(line) == 0):
               if(HP == True):
                    return True
               else:
                    continue

          if(line[:2] == "ob"):
               pobj = obj_num(line)
               cobj = pobj
          if(line[:3] == "cob"):
               cobj = obj_num(line)

          if((line[:2] == "fn")
          and (obj_num(line) == globals()\\
               ['SDL_LIST'][index][-1])):
               HP = True
               cobj = pobj

          if((HP == True) and (line[:3] == "cfn")):
               CHECK_FN = False
               new_chain = copy.deepcopy(globals()\\
                                          ['SDL_LIST'][index])
               new_chain.append(cobj)
               new_chain.append(obj_num(line))

               if(cobj == lookup_binary_number("oGL")):
                    add_chain(new_chain)
                    return True
               if(cobj == lookup_binary_number("SDL")):
                    try:
                         globals()['SDL_LIST'].index(new_chain)
                         temp = [new_chain[-4], new_chain[-3],\
                                 new_chain[-2], new_chain[-1]]
                         add_reject(temp)
                         del temp[:]

                    except:
                         extend_chain(new_chain)


               del new_chain[:]
               cobj = pobj
     return True
#-----------------------------------------------------------

#-----------------------------------------------------------
#PARSE THE CALL CHAINS FROM callgrind
#
#INPUT ARGS: infile
#infile: the data file generated by a single profiled
#execution
def parse_chains(indat):
     pobj_num = ''
     cobj_num = ''
```

```
pfunc_num = ''
cfunc_num = ''
PARENT_OBJ = False
CHILD_OBJ = False
HAVE_COB = False


sdl = lookup_binary_number('SDL')
os.lseek(indat.fileno(), 0, 0)

for line in indat.read().split("\n"):
     if(len(line) == 0):
          continue

     #GET THE OBJECT NUMBER
     if(line[:2] == "ob"):
          pobj_num = obj_num(line)
          #NEED TO HANDLE INTERNAL OBJECT CALL
          #HAVE CHILD = PARENT UNTIL CHILD OBJECT IS FOUND
          cobj_num = pobj_num
          PARENT_OBJ = target_obj(pobj_num)
          CHILD_OBJ = PARENT_OBJ

     if(line[:3] == "cob"):
          HAVE_COB = True
          cobj_num = obj_num(line)
          CHILD_OBJ = target_obj(cobj_num)

     if(line[:2] == "fn"):
          pfunc_num = obj_num(line)
          #THE CHILD OBJECT SPECIFIER cob=(xxx) IS ONLY GOOD
          #FOR THE FUNCTION IMMEDIATELY FOLLOWING THE
          #SPECIFIER, HAVE TO RESET COB TO WHATEVER THE
          #PARENT IS
          if(HAVE_COB == True):
               cobj_num = pobj_num
               CHILD_OBJ = PARENT_OBJ
               HAVE_COB = False

     if(line[:3] == "cfn"):
          cfunc_num = obj_num(line)
          #CHECK TO SEE IF IT IS A CHILD AND PARENT
          #OF INTEREST
          if((PARENT_OBJ == True) and (CHILD_OBJ == True)):

               temp = [pobj_num, pfunc_num, cobj_num, cfunc_num]
```

```
                    if(check_sdl_chain(temp, PARENT_OBJ, sdl) == True):
                         continue

                    elif(temp[0] != sdl):
                         check_app_chain(temp)

                    del temp[:]
               #THE CHILD OBJECT SPECIFIER cob=(xxx) IS ONLY
               #GOOD FORTHE FUNCTION IMMEDIATELY FOLLOWING
               #THE SPECIFIER, HAVE TO RESET COB TO WHATEVER
               #THE PARENT IS
               if(HAVE_COB == True):
                    cobj_num = pobj_num
                    CHILD_OBJ = PARENT_OBJ
                    HAVE_COB = False
     return True
#-----------------------------------------------------------

#-----------------------------------------------------------
def main():

     #VARIABLES
     parents = []
     ofile = []
     indat = []
     sdl_func = 0
     s_time = 0.0
     e_time = 0.0
     t_time = 0.0
     global binname

     #REQUIRED ARGUMENT CHECK
     if(len(sys.argv) < 2):
          print "usage: python sleipnir.py <input file> \
          [-f <output file>]"
          sys.exit()
     #else:
     if(True):
          try:
               indat = open(sys.argv[1], "r")
          except IOError, e:
               print "could not open file: ", e, "exiting"
               return False

          #BUILD LIST OF OBJECT AND FUNCTION NUMBER /
          #NAME LISTS
```

```
        s_time = time.clock()
        if(build_lists(indat) == False):
            return False

        print >> sys.stderr, "build list time elapsed = ", \
            (time.clock() - s_time)

        s_time = time.clock()
        if(parse_chains(indat) == False):
            return False

        print >> sys.stderr, "call chain list time elapsed = ", \
            (time.clock() - s_time)

        s_time = time.clock()
        while(sdl_func < globals()['SDL_IDX']):
            if(close_sdl(sdl_func, indat) == False):
                return False
            sdl_func = sdl_func + 1
        print >> sys.stderr, "SDL closure list time elapsed = ", \
            (time.clock() - s_time)

        #WRITE TO OUTPUT FILE IF REQUESTED
        s_time = time.clock()
        if((len(sys.argv) >3 ) and (sys.argv[2] == '-f')):
            if(output_list(sys.argv[3]) == False):
                return False
        else:
            if(output() == False):
                return False
        print >> sys.stderr, "output listing time elapsed = ", \
            (time.clock() - s_time)


    return True
#------------------------------------------------------------


#------------------------------------------------------------
if __name__ == "__main__":
    if(main() == False):
        print "sleipnir parser did not complete\n"
sys.exit()
#------------------------------------------------------------
```

A.2 fenris.py

```
#!/usr/bin/python
#***********************************************************
#JOE SULLIVAN
#20/02/2007
#This script filters out GAME->GL, LIB->GL and GL-GL calls,
#and writes those call chains out to a data file
#AREGUMENTS: argv[0] this script
#            argv[1] name of input file,
#            argv[2] name of the concatenated data file
#                    to add data to
#
#***********************************************************


#-----------------------------------------------------------
import os;
import sys;
import string;
import copy;
#-----------------------------------------------------------


#-----------------------------------------------------------
#GLOBALS
MASTER_LIST = []
SLIST = []
START_OGL = False
#-----------------------------------------------------------


#-----------------------------------------------------------
#PRINT RESULTS TO STDOUT
#
#INPUT: VOID
#
def output():
     #WRITE NUMBER OF CALLS IN OUTPUT FILES
     try:
          print str((len(MASTER_LIST)+len(SLIST)))

          for idx in range(len(globals()['MASTER_LIST'])):
               print(str(globals()['MASTER_LIST'][idx]))

          if(START_OGL == False):
                for cnt in range(len(globals()['SLIST'])):
                     print "$START :", SLIST[cnt]

     except  e:
          print "could not output to file: ", e, exiting.
          return False

     return True
#-----------------------------------------------------------


#-----------------------------------------------------------
#ADD A GL->GL or ENTRY->GL CALL TO THE MASTER LIST
#
#INPUT:
#ostring:    string to be added to list
def mlist_add(ostring):
     try:
          MASTER_LIST.append(copy.deepcopy(ostring))
     except  NameError, e:
          print "Unexpected error adding to Master List:", e
          return False
     return True
#-----------------------------------------------------------


#-----------------------------------------------------------
#ADD A $START->GL ENNTRY TO THE START LIST
#
#INPUT:
#entry:    string to be added to list
def slist_add(entry):
     try:
          if(len(SLIST) == 0):
               SLIST.append(copy.deepcopy(entry))
          else:
               for i in range(len(SLIST)):
                    if(entry not in SLIST):
                         SLIST.append(copy.deepcopy(entry))
     except  NameError, e:
          print "Unexpected error adding to START List:", e
          return False
     return True
#-----------------------------------------------------------
```

```
#-----------------------------------------------------------
#FILTER THE PARSED DATA FROM DATA_PARSER.PY FOR ENTRY->GL
#and GL->GL CALLS
#
#INPUT:
#InFile:     name of the file to be opened for parsing
def filter_gl_calls(InFile):
    indat = []
    outstr = ""
    ret = True


    try:
         indat = open(InFile, 'r')
    except IOError, e:
         print "could not open file: ", e, "exiting"
         return False

    for lines in indat.read().split("\n"):
         #REMOVE ANY BLANK LINES
         if(len(lines) == 0):
              continue

         #SPLIT THE DATA INTO A LIST
            nstr = lines.rsplit(':')

         #REMOVE ANY 1 ELEMENT LIST
         $(number of calls for example)
         if(len(nstr) == 1):
              continue

         #CHECK FOR ENTRY->GL & GL->GL CALLS
         if((nstr[0][:3] == "GLH")
         or (nstr[0][:3] == "NVB")
         or (nstr[0][:3] == "SDL")
         or (nstr[0][:5] == "TPROJ")):
              if(nstr[1][:4] == " oGL"):
                   if(START_OGL == True):
                        entry_str = "$START :" + nstr[1]
                        mlist_add(entry_str)
                   else:
                        slist_add(nstr[0])
                        entry_str = nstr[0] + ':' + nstr[1]
                        mlist_add(entry_str)
         if(nstr[0][:3] == "oGL"):
              outstr = nstr[0][:-1]+ ' : '+ nstr[1][1:]
            ret = mlist_add(outstr)

        if(ret == False):
            return False
        try:
            del outstr
        except:
            pass
    return True
#-----------------------------------------------------------


#-----------------------------------------------------------
#OUTPUT MASTER_LIST TO INDIVIDUAL, AND A "GROUPING" FILE
#
#INPUT:
#OutFile: name of the file to write an individual data
#         file after the list is created
#ccFile:  name of the file that contains data concatenated
#          from all the runs
def output_list(ccFile):
     tdat = []
     try:
          tdat = open(ccFile, 'a')
     except IOError, e:
          print"could not open file: ", e, " exiting"
          return False

     #WRITE NUMBER OF CALLS IN OUTPUT FILES
     try:
          tdat.write(str( len(globals()['MASTER_LIST']) \
            + len(globals()['SLIST'])))
          tdat.write('\n')

          for idx in range(len(globals()['MASTER_LIST'])):
               tdat.write(str(globals()['MASTER_LIST'][idx]))
               tdat.write('\n')
     except IOError, e:
          print "could not output to file: ", e, exiting
          return False
     except e:
          print "index out of range error: ", e
          return False

     return True
#-----------------------------------------------------------
```

```
#-----------------------------------------------------------
def main():

    #LOCAL VARIABLES
    ret = False
    ofile = ""

    #CHECK ARGS
    if(len(sys.argv) < 1):
        print"USAGE: fenris.py FILE <input> \
          [-C:[condense APP->GL calls] -f <output>]"
        return False
    else:
        #FIND THE ENTRY->GL & GL->GL CALLS
     if( (len(sys.argv) >= 3) and (sys.argv[2] == "-C") ):
          globals()['START_OGL'] = True

     if(filter_gl_calls(sys.argv[1]) == False):
          return False

     if((len(sys.argv) >= 3) and (sys.argv[2] == '-f')):
          #DETERMINE OUTPUT FILE NAME
          #BASED ON INPUT FILE NAME
          if(output_list(sys.argv[3]) == False):
               return False
     else:
          if(output() == False):
               return False

    return True

#-----------------------------------------------------------

#-----------------------------------------------------------
#ENTRY:
if __name__ == "__main__":
    if(main() == 0):
        print "fenris.py did not complete"
sys.exit()
#-----------------------------------------------------------
```

A.3 gullfaxi.py

```
#!/usr/bin/python
#*************************************************************
#JOE SULLIVAN
#18/05/2007
#This script count the number of times a child function
#is called by each parent call. Writes to stdout,
#AREGUMENTS: argv[0] this script
#            argv[1] name of input file,
#*************************************************************

#------------------------------------------------------------
#INCLUDES
import sys
import os
import re
import copy
import time
#------------------------------------------------------------


#------------------------------------------------------------
#REGULAR EXPESSIONS
glheretic = re.compile('glheretic')
neverball = re.compile('neverball')
binname = re.compile('sample')
libgl = re.compile('libGL.so\.[0-9]\.[0-9]\.[0-9]+')
libsdl = re.compile('libSDL-[0-9]\.[0-9]\.so\.[0-9]\.\\
                     [0-9][0-9]\.[0-9]')
#------------------------------------------------------------


#------------------------------------------------------------
OBJ_DICT = {}
FUNC_DICT = {}
FUNC_TAB = []
#------------------------------------------------------------


#------------------------------------------------------------
def split_name(line, otype):

    spl_str = []

    spl_str = line.split(" ")
    num = spl_str[0][spl_str[0].index('(')+1 : -1]

    if(spl_str[0] != spl_str[-1]):
        if((otype == "ob") or (otype == "cob")):
            nstr = spl_str[-1].split('/')
            OBJ_DICT[num] =  nstr[-1]
        if((otype == "fn") or (otype == "cfn")):
            if(spl_str[-1][-1] == ')'):
                nstr = spl_str[1].split('(')
                nstr = nstr[0]
            else:
                nstr = spl_str[-1]
            FUNC_DICT[num] = nstr
    return num
#------------------------------------------------------------


#------------------------------------------------------------
def ibin(kval):

    if(kval == ''):
        return False

    obj_name = OBJ_DICT[kval]

    if(glheretic.match(obj_name)):
        return True
    if(neverball.match(obj_name)):
          return True
    if(libgl.match(obj_name)):
          return True
    if(libsdl.match(obj_name)):
          return True
    if(binname.match(obj_name)):
          return True

    return False
#------------------------------------------------------------


#------------------------------------------------------------
def record_call(ncalls, c_func, p_func):
    cf_num = 0
    p_list = []

    y = 0
    p_list.append([])
    p_list[0] = [p_func, ncalls]

    cf_num = int(c_func)
    x_factor = cf_num - len(FUNC_TAB)
    while(x_factor >= 0):
        FUNC_TAB.append([])
        x_factor = x_factor - 1

    if(len(FUNC_TAB[cf_num]) == 0):
        FUNC_TAB[cf_num].append(copy.deepcopy(FUNC_DICT[c_func]))

    FUNC_TAB[cf_num].append(copy.deepcopy(p_list[0]))

    #print len(FUNC_TAB)
    #print FUNC_TAB[cf_num]
#-----------------------------------------------------------


#-----------------------------------------------------------
def parsefile(file):


    WCALLS  = False
    isvalid = False
    P_OBJ = ''
    C_OBJ = ''
    S_OBJ = ''
    pval = ''
    pfunc = ''

    #OPEN THE FILE
    try:
        indat = open(file, "r")

    except IOError, e:
        print "COULD NOT OPEN FILE: ", e, "exiting"
        return False

    for line in indat.read().split('\n'):
        if(len(line) == 0):
            continue

        if(line[:3] == 'cfi'):
            continue
        if(WCALLS == False):
            if(line[:2] == "ob"):
                C_OBJ = split_name(line, "ob")
                S_OBJ = C_OBJ
                continue

            if(line[:3] == "cob"):
                C_OBJ = split_name(line, "cob")
                continue

            if(ibin(C_OBJ) == True):
                if(line[:2] == "fn"):
                    pval = split_name(line, "fn")

                if(line[:3] == "cfn"):
                    WCALLS = True
                    cval = split_name(line, "cfn")
            else:
                C_OBJ = S_OBJ

        else:
            call = line.split('=')
            if(call[0] == "calls"):
                if((pval in FUNC_DICT) == True):
                    pfunc = FUNC_DICT[pval]
                if(pfunc != ''):
                    record_call(call[1].split(' ')[0], cval, pfunc)
                else:
                    record_call(call[1].split(' ')[0],\\
                                cval, "$START")
                WCALLS = False
                C_OBJ = S_OBJ
                pfunc = ''

    return True
#-----------------------------------------------------------


#-----------------------------------------------------------
def main():

    #CHECK ARGS TO SCRIPT
    if(len(sys.argv) < 2):
        print "USAGE: gullfaxi <input file>"
        return False
    #PARSE THE FILE
    if(parsefile(sys.argv[1]) == False):
        return False

    for x in range(len(FUNC_TAB)):
        if(len(FUNC_TAB[x]) > 0):
           print FUNC_TAB[x][0], "CALLED BY:"
           for y in range(len(FUNC_TAB[x])):
               if(y > 0):
                   print '\t',  FUNC_TAB[x][y][1], \
                   '\t:', FUNC_TAB[x][y][0]
           print '\n'
#-----------------------------------------------------------


#-----------------------------------------------------------
if __name__ == "__main__":
    if(main() == 0):
        print "gullfaxi parser did not complete\n"
sys.exit()
#-----------------------------------------------------------
```

A.4 gdrv.sh

Shell script driver for sliepnir.py and fenris.py

```
#!/bin/sh
for f in $1
do
   file=${f##*/}
   base=${file%%.*}
   ./sleipnir.py $f > $base\.pol
   /fenris.py $base\.pol -C > $base\.tst
   n=$(($n+1))
done

mkdir $2
rm \./*\.pol
mv \./*\.tst $2

mv $2 /home/jes/tools/
```

A.5 fdrv.sh

Shell script driver for functionality identification tools

```
#!/bin/sh
n=$3
C=0
for x in $1
do
  path=${x%/*}
  file=${x##*/}
  base=${file%%.*}
  echo $path/$file
  cp $path/$file .
  ./preprocessfiles.1.1 "run*" $2
  python ./tug -d $2
  dot -Tps $2.des.dot -o gl.ps
  ps2pdf gl.ps gl.pdf
  mkdir RUN$n
  mv $2\.* RUN$n
  mv gl\.* RUN$n
  n=$(($n+1))
done

mkdir $4
mv RUN* $4
```

BIBLIOGRAPHY

[1] Anonymous. (n.d.). Sample Programs for the OpenGL Redbook (Version 1.1.) [Computer source code]. Retrieved September 18, 2007 from http://www.opengl.org/resources/code/samples/redbook/.

[2] Armour-Brown, C. (n.d.). Valgrind (Version 3.2.3) [Computer software and manual]. Retrieved October 14, 2007, from http://valgrind.org/.

[3] Canfora, G., & Di Penta, M. (2007). "New Frontiers of Reverse Engineering," *FOSE '07: 2007 Future of Software Engineering,* Washington, DC, USA, pp. 326-341.

[4] Chow, M. (1997). "Optimized Geometry Compression for Real-Time Rendering." *Proceeding of the 8th IEEE Visualization '97 Conference,* Phoenix, Az., United States, pp. 347-354.

[5] Elbaum, S., & Munson, J. (1999). "Intrusion detection through dynamic software measurement," *Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring, USA, 1,* 5-5.

[6] Eisenbarth, T., Koschke R., & Simon, D. (2001). "Derivation of Feature Component Maps by means of Concept Analysis," *5th European Conference on Software Maintenance and Reengineering,* Lisbon, Portugal, pp. 176-179.

[7] Ellson, J. et al. (n.d.). Graphviz (Version 2.2) [Computer software and manual]. Retrieved October 14, 2007, from http://www.graphviz.org/.

[8] Hall, G. (1997). "Usage patterns: Extracting system functionality from observed profiles." Ph.D. dissertation, University of Idaho, United States – Idaho. Retrieved October 24, 2007, from ProQuest Digital Dissertations database. (Publication No. AAT 9731155).

[9] Hall, G. (nd). "Usage Driven Reverse Engineering" Department of Computer Science, *Texas State University-San Marcos,* pp. 1-16.

[10] Hall, G., & Davis, W. (2004). "A Structural Definition of Software Functionality," Department of Computer Science, *Texas State University-San Marcos,* pp. 1-22.

[11] Hazelwood, K., & Grove, D. (2003). Adaptive Online Context-Sensitive Inlining. *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization,* USA, 253-264.

[12] Kooima, R. (n.d.). Neverball (Version 1.4.0) [Source source code]. Retrieved September 18, 2007, from http://icculus.org/neverball/.

[13] Kuehne, B., True, T., Commike, A., & Shreiner, D. (2005). "Performance OpenGL: platform independent techniques," *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses,* New York, NY: ACM Press.

[14] Kuhn, A., Greevy, O., & Gîrba T. (2005). "Applying Semantic Analysis to Feature Execution Traces." *1st International Workshop on Program Comprehension Through Dynamic Analysis,* Pittsburgh, Pennsylvania, pp. 48-58.

[15] Mancl, D. (2001). Refactoring for software migration. *Communications Magazine, IEEE,* 39(10), 88-93.

[16] Paul, B. (n.d.). Mesa (Version 6.4.2) [Computer source code]. Retrieved August 20, 2007 From http://www.mesa3d.org/.

[17] Paul, B. (1997). "OpenGL Performance Optimization." Retrieved October 3, 2007. From Mesa3d Project. Web site: http://www.mesa3d.org/brianp/sig97/perfopt.htm.

[18] Siff, M., & Thomas, R. (1999). "Identifying Modules via Concept Analysis." *IEEE Transactions on Software Engineering,* 25, 749-768.

[19] Stoll, R. (1979). *Set Theory and Logic.* Mineola, New York: Dover Publications, Inc.

[20] Wertman, Andre. (2003). GLHeretic (Version 1.2) [Computer source code]. Retrieved September 18, 2007, from http://heretic.linuxgames.com/np/heretic2.shtml.

[21] Woo, M., Neider, J., Davis, T., & Shreiner, D. (1999). *OpenGL Programming Guide.* Reading, Massachusetts: Addison-Wesley.

Vita

Joseph E. Sullivan was born in Fort Stockton, Texas, in November 1970 to Rosalie M. Sullivan and Joseph F. Sullivan. After graduating from Fort Stockton High School in 1989, he attended Texas Tech University, and Sul Ross University where he received a Bachelor of Science in May 1994. He was employed by the Texas Bowl Weevil Eradication Program in Brazoria County, Texas. In 1997 he entered Texas Tech University for the second time to pursue a degree in computer science. Mr. Sullivan moved to Austin, Texas with the dot-com boom of the late 90s where he was employed by Metrowerks Inc, a computer software company specializing in software development tools. He spent 5 years as a member of the Games Team, doing developer support for the Sony PlayStation 2, and Nintendo GameCube video game consoles. Mr. Sullivan is currently employed as a compiler engineer working for the same group under a new company owner. He entered the Graduate College of Texas State University-San Marcos in January 2003.

Permanent Address:	1113 Glendalough Drive
	Pflugerville, Texas 78660

This thesis was typed by Joseph E. Sullivan.