SEMINAR

ANALYSE, PLANUNG UND KONSTRUKTION

COMPUTERGRAFISCHER SYSTEME

OpenGL-Performance and Bottlenecks

Matthias Trapp

HASSO-PLATTNER-INSTITUT
UNIVERSITÄT POTSDAM

FACHGEBIET COMPUTERGRAFISCHE SYSTEME
PROF. DR. J. DÖLLNER

WINTERSEMESTER 2003/2004

OpenGL-Performance and Bottlenecks

Matthias Trapp

trapp.matthias@freenet.de
Hasso-Plattner-Institut at the University of Potsdam

Abstract This paper provides an introduction and overview on performance and optimization of computer graphical applications. The rapid development of graphic hardware increases the complexity of such systems, accompanying APIs and applications. It could be expected that this topic will gain more attention in the future. This situation requires a structured access to basic performance measurement principles and their implementation as well as the process of optimization. Both and especially basic solutions for potential graphic-bottlenecks will be presented here with OpenGLs API and rendering pipeline as an instrumentation.

CR Categories and Subject Descriptors C.4 [Performance of Systems]: Measurement techniques-Performance attributes; D.2.8 [Software Engineering]: Metrics— performance measures; I.3.1 [Hardware Architecture]:Graphics processors; I.3.6 [Computer Graphics]:Methodologies and Techniques

Keywords OpenGL Performance, Bottleneck, Pipeline Optimization

1 Introduction

The performance of applications is one of the main topics in computer graphic. Application such as time critical or detail critical real-time rendering applications like real-time 3D computer games or medical applications with high grade of interaction. The increasing market potential of the computer game industry just to mention by the way. During the last ten years the development of more complex hardware acceleration allows more realistic effects with increasing data and increasing numbers basic operations. Focusing on graphic performance leads toward the classical trade-off between quality and quantity. Nowadays two graphic APIs gain acceptance: OpenGL and DirectX. Let's focusing on the first one, which has its roots in CAD and scientific visualization. The hardware development along the OpenGL pipeline architecture shows up the question of acceleration versus optimization. OpenGL extensions[1] were released to compensate the problems of rapid hardware development in this sector [2,4]. The optimization of programming time is mostly done by library design. This papers proposal is an introduction and overview upon a very interesting and complex subject.

What are the limiting factors of graphic acceleration? How could they be resolved and what does it cost? Bottleneck is an often-used term in computer science. It describes the cause of a decreasing performance in general. Bottlenecks depend on the hardware architecture that is limited in different parameters. Hereby trying to give a definite meaning in this special context. It is essential to describe the effect of these parameters with respect to possible performance goals. We will distinguish between a logical and an architectural-specific view upon bottlenecks to work out a prototyping optimization process. The optimization process is unfortunately linear and shows up with questions of integration and proceeding. Section 2 gives an introduction to performance parameters, measurement principles, and refers tools. Section 3 denotes bottlenecks as the cause of performance lacks and how they can be classified and detect. In section 4 this paper provides a structured access for optimizing a computer graphical application.

2 Performance and Optimization

In this section we try to determine what kind of performance is meant and how it is related to computer graphics in common. McDaniel [10] gives a common definition of what performance is about. In the case of computer science performance seems to have two meanings:

- The speed at which a computer operates, either theoretically (for example, using a formula for calculating Mtops - millions of theoretical instructions per second) or by counting operations or instructions performed (for example, (MIPS) - millions of instructions per second) during a benchmark test. The benchmark test usually involves some combination of work that attempts to imitate the kinds of

[1] http:// glew.sourceforge.net

work the computer does during actual use. Sometimes performance is expressed for each of several different benchmarks.

- The total effectiveness of a computer system, including throughput, individual response time, and availability.

If an application doesn't perform as expected we need to identify which part of the application is the reason and why. In a pipeline architecture, getting good performance is about finding and eliminating bottlenecks. Note that some portion of the application will always be the limiting factor to performance. This finally leads toward an optimization problem. Generally speaking a computational problem in which the object is to find the best of all possible solutions. More formally, to find a solution in the feasible region which has the maximum (or minimum) value of the objective function. Feasible regions signify the set of all possible solutions of an optimization problem [5]. Or short, in an optimization problem we seek optimal (best possible) values of the variables that lead to an optimal value of the function that is to be optimized. The objective function which is associated with an optimization problem determines how good a solution is, in our case could it be the total time per rendering-pass by a given pipeline setup and amount of data to process. In particular, an optimal pipeline setup for each possible type and amount of data, which are processed under an upper boundary of time (frames per second). In short, how many computational workload or/and how many data is necessary for a chosen speed and quality? We know that some of these variables are determined by the hardware (e.g. critical path) of a system and also depend on the requested quality.

Parameter	Equation
Bytes per vertex	*bpv* (det. by programmer)
Primitive per second	*tps* (det. by hardware)
Polygons per second	*pps* (det. by hardware)
avg. vertex per triangle	*comp* (by primitive type)
Frames per second	$fps = pps / tps$
Triangles per frame	$tpf = tps / fps$
Vertices per frame	$vpf = comp \cdot tpf$
Bytes per frame	$bpf = vpf \cdot bpv$
Bytes per second	$bps = bpf \cdot fps$
Pixel per frame	$ppf = pps / fps$

Table 1. Parameter, Metrics and Connections

The equations in table 1 are idealized and should give an idea on causal relationships and determination of influencing variables. They should help to identify realistic performance goals.

2.1 Basic Measurement Principles

How can we measure performance on a single system for comparison? Attempting to tune an interactive OpenGL application requires an established metric for estimating the applications performance. There are three different methods, each of them with a different implementation complexity. They are:

- Frames per second
- Primitives per second
- Pixel per second

The latter can be done by approximation about the count of pixel of a current primitive. The first metric is surely the classic and is supported by many scene graph-libraries. The usual procedure to get an average value is to measure a definite start value (with respect to time), then rendering, counting frames, and measuring end value. The comparative value (fps) results from division of the number of frames and the passed time in seconds. The more frames can be processed the more exact is the result. To get a high meaning it is recommended to run results for the average frame rate, minimum frame rate, maximum frame rate, and number of measurements. Note that benchmarks are performed in systems of various speeds and configurations, and cannot be considered an "Apples-to-Apples" comparison. However, general trends should be obvious, since many machine configurations are comparably similar. It makes sense to compare only graphic cards or accelerators of the same generation. Some useful tips performing benchmarks in the following. There are several other benchmark tips provided in [19]:

- Take measurements on a quiet system
- Take measurements over period of time that is at least 100 times the clock resolution
- Render a large number of primitives
- Call `glFinish` before reading the clock double buffered programs may stall after calling `glSwapBuffers`

2.2 Benchmark Tools and Libraries

Benchmark-tools could be used to test performance on different system-architectures. Benchmarks can be conditions under which the performance of a system or application could be measured. Also benchmarks can denote performance criterions which can be achieved or

further a special software to measure components, systems or applications. Latter based on standards provided by organizations like SPEC[2] (see also OpenGL Performance Characterization Group (OPC)). The results of such benchmarks can be used for advertisement purposes. The following list contains established benchmarks:

- SPECapc Benchmark to measure performance of 3D applications
- GLperf Benchmark to measure performance of OpenGL graphic operations.
- SPECperf Benchmark for graphic hardware (e.g. workstations)

These tools are designed to measure graphics performance on professional software applications such as 3DStudioMax, Pro/Engineer, and Unigraphics. The usage a performance benchmark, like GLperf, or a custom-written benchmark[3] is practical before start coding. At application runtime it is necessary to allow scalability across many platforms. This can be implemented by building in a mini-benchmark to test performance and select rendering paths which depending on tested performance afterwards [14]. In this case, it is recommended to use no timers that are provided by GUI event management. They are inaccurate and depending on application event handling. This means they occur only when the application is idle. Other possibilities are the `timeGetTime` and the `Query-PerformanceCounter` function. Latter is very accurate and could be found in mostly all high-level languages. It uses the PC's clock counter, but it reads the current value of the countdown register in the timer chip to gain more accuracy[4]. It takes 5 to 10µs to call and results must be convert from counter rate into time [19]. Finally given a list of programming libraries that could support the programmer:

- isfast/pdb is an OpenGL performance database composed of libisfast and libpdb (performance database, routines for measuring execution rates and maintaining a simple database)[5]
- JCCL (for VR Juggler) is a collection of configuration and performance monitoring tools (allows application-side runtime reconfiguration)[6]
- OpenGL Optimizer is an open API for large-model visualization[7]

These libraries aren't a substitute for comprehensive benchmarking and performance analysis, but they can handle simple as well as complex tasks.

3 Bottlenecks

Traditional software tuning focuses on finding and tuning *"hot spots"*, the 10% of code in which a program spends 90% of its time. Today's graphics hardware accelerators are arranged in pipeline stages. Because these stages operate in parallel, it is appropriate to use a different approach: look for

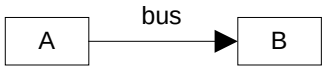


Figure 1. Principle Bottleneck Structure

bottlenecks - overloaded stages that are holding up other processes [11]. The term bottleneck and overloaded denotes the different causes of performance lacks. Modern GPUs are likely programmable pipelines, as opposed to simply configurable, which means more potential bottlenecks and more complex tuning. This increased flexibility of rendering pipeline, mostly to reduce number of passes. The goal is to keep each stage (including the CPU) busy creating portions of the scene to get a pipeline that working to full capacity. Figure 1 shows a principle sketch to describe two types of bottlenecks.

The function components A and B can process a fixed number of operations *n* in a fixed time interval *t*. These components can be classified as time-bounded. A component becomes a bottleneck if it have to process more then *n* operations so that *t* increase. This is denoted as *latency-bottleneck.* The data bus between A and B represent an other type of bottleneck. The performance of a bus is determined by its throughput *m* (in bit/sec). If the amount of data, which should pass through the channel, is greater than *m*, then the bus becomes a bottleneck. This is denoted as *bandwidth-bottleneck*. An overview and more details can be found in [9] based on [8][8].

[2] Standard Performance Evaluation Corporation, http://www.spec.org, http://www.specbench.org

[3] http://www.active-hardware.com provides many of them

[4] down to 1.193MHz (about 800ns)

[5] http://www.berkelium.com/OpenGL/isfast.html

[6] http://www.vrjuggler.org/jccl/index.php

[7] http://www.sgi.com/software/optimizer/

[8] and http://www.intel.com/technology/agp/tutorial/

3.1 Bottleneck Classification

After we have defined the term bottleneck, we discuss different approaches for determining which potential bottlenecks can appear in a computer graphical application and how they can be classified. It is useful to make a distinction between logical and architectural bottleneck classification for detection and optimization.

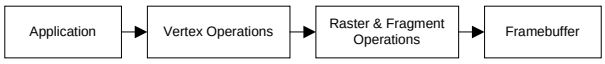


Figure 2. OpenGL Rendering Pipeline

3.1.1 Logical Classification

The OpenGL graphics pipeline in figure 2 consists mainly of three conceptual stages. All three parts may be implemented in software or performed by hardware graphics accelerator. This model reflects one of the first approaches to OpenGL performance tuning [11,18]. This conceptual model is useful in either case: it helps to know where an OpenGL application spends its time; this is hardware independent [8]. We can distinguish between:

- **Application bottleneck.** (or CPU/Bus bound). The application may not pass data fast enough to the OpenGL pipeline.
- **Transform bottleneck.** (or Vertex bound). OpenGL may not be able to process vertex transformations fast enough
- **Fill bottleneck.** (or Pixel bound). OpenGL may not be able to rasterizing primitives fast enough

This macroscopic view is applicable to every OpenGL system or application in general. It is hardware independent and mirrors the OpenGL pipeline. Typical OpenGL systems are hardware accelerated, thus OpenGL software driver will emulate only features that cannot be processed by hardware. It becomes apparent that this may have an impact on performance. To detect and solve bottlenecks more accurate we have to map the macroscopic view to a concrete architecture classification. For instance we can differ between a single processor pipeline and multi processor pipeline.

3.1.2 Architectural Classification

An architectural view is more specific and suitable for an optimization process. It depends on the graphic acceleration hardware (Notice instance in figure 3). Table 2. provides a more detailed categorization of bottlenecks for single processor pipeline given in [20].

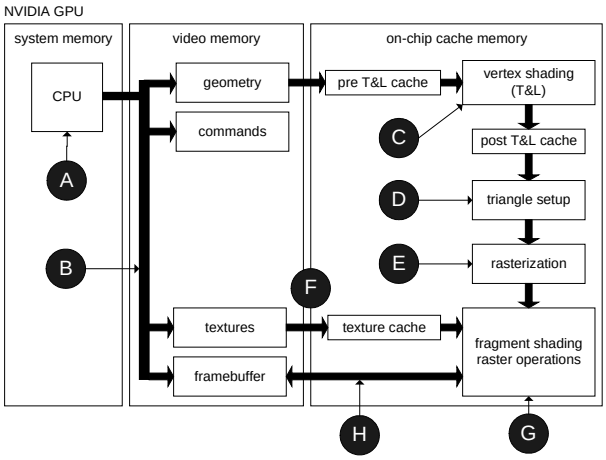


Figure 3. Example for Single Processor Pipeline

Logical Classification	Architectural Classification	Bottleneck Type
Application	CPU-limited (A)	Latency
Application	Transfer-limited (B)	Bandwidth
Transform	Transform-limited (C)	Latency
Transform	Setup-limited (D)	Latency
Fill	Raster-limited (E)	Latency
Fill	Texture-limited (F)	Both
Fill	Fragment-limited (G)	Latency
Fill	Framebuffer-limited (H)	Bandwidth

Table 2. Architectural Classification

It contents show several potential bottlenecks in a graphical system including application and rendering hardware. Therefore we require techniques to determine bottlenecks in OpenGL application.

3.2 Detecting Bottlenecks

In fact there will always be a bottleneck in an application. In general, the bottleneck varies over the course of a frame. Expect a graphics-intensive application (like a real-time computer game) to spend a good amount of time in `glBegin`/ `glEnd`, `glFinish` is a valid as rule of thumb. The basic strategy for isolating bottlenecks is to measure the time it takes to execute a part or a whole program and then change the source code in ways that add or subtract work at a single point in the graphics pipeline. If changing the amount of work at a given stage does not alter performance appreciably, that stage is not the bottleneck. If there is a noticeable difference in performance, a bottleneck exists [11]. So there arise two likely similar test strategies to resolving pipeline bottleneck location:

- (A) As first technique reduce workload of a certain stage. Usually the easiest to test should be tested first. If the performance is getting better then that stage is the bottleneck. We can denote this as direct.
- (B) The second strategy works similar by making the other stages work less or (better) not at all. If performance is the same, then the stage does not include the bottleneck. We can denote this as indirect.

Test the application stage by using a profiler[9]. This stage is the bottleneck if the application uses nearly 100% of CPU time [22]. You can also test application-limited bottlenecks by modifying the application so that no rendering is performed, but all data is still transferred to OpenGL. Rendering in OpenGL is triggered when vertices sent to the pipe. Sending all data to pipe, not necessarily in its original form. This can be done by replacing all `glVertex*` calls with `glNormal*` calls. This only sets the current vertex normal values (see macro in appendix A.1). Application transfers the same amount of data to the pipe, but doesn't have to wait for the rendering to complete.
Changing the workload of the transform stage often changes the workload of an application and a rasterizer as well. This fact makes it difficult to test. However, disabling or enabling all light sources for test strategy B. If performance doesn't vary, then this stage is not the bottleneck. At last, the rasterizer stage is easy and fast to test by decreasing the size of the view port. This does not change the workload for the application or geometry stage but the rasterizer produce fewer fragments. Turning off texturing, fog, blending, depth buffering etc. also reduces the rasterizer work.
A summary for bottlenecks of the logical classification can be examined in table 3. For a more sophisticated test-procedure of architectural classified bottlenecks see appendix A.2 is recommended [7, 20]. Finally, detecting a bottleneck-stage requires an amount time and cannot be done without changing the program source code. During changing, re-integration and building of the corresponding code fragments no other development can be performed in this project. So it is useful to consider about testing while designing an application to save time, hence money. After settling the questions of classification and detecting bottlenecks we shall go on to the next topic, which provides solutions for performance problems.

[9] i.e. top on Unix, TaskManager on MS-Windows systems

Stage	Test Complexity	Test Strategy
Application stage	Middle	A
Transform stage	Middle	A/B
Rasterizer stage	Low	A

Table 3. Test Logical Classification Bottlenecks

4 Optimization Process

For optimizing we have to make some requirements: 1) an established metric for estimating the applications performance exists (e.g. consistent frames/second, number of pixels/primitives per frame), and, of course 2) the applications source code is accessible. Before starting to optimize an application it is important to decide which optimization targets are realistic, convenient and optimal. It is recommended only to optimize where an increase of performance can be expected. Such as the complexity of used algorithms play an important part.

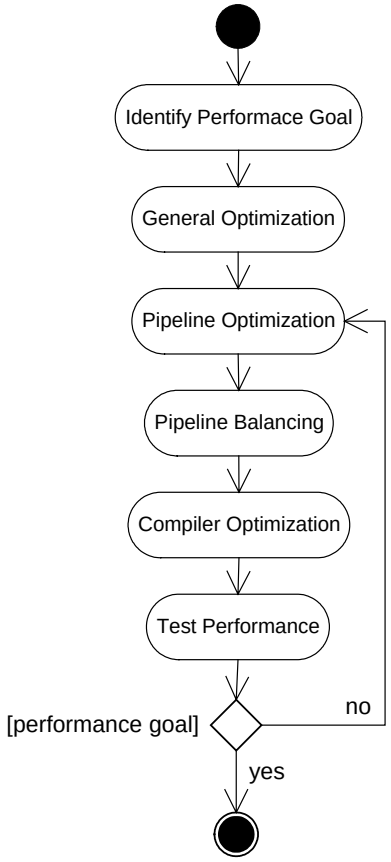


Figure 4. Activity Diagram of Optimization Process

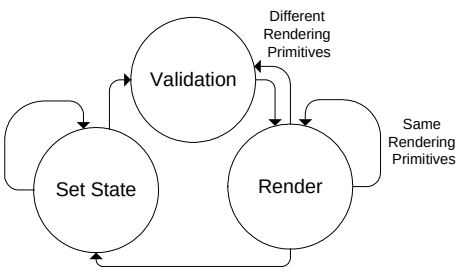


Figure 5. State Diagram OpenGL Validation

The processes of optimization of software consist of different optimization phases and iterations. Figure 4 shows a possible approach for an optimization proceeding. Most of the optimization phases require changes of the source code. Note that the cost of optimization, which means compile and integration time should not be underestimated. The necessary effort could increase along the application scale. To make this paper more applicable, we follow the pipeline and describe OpenGL optimizations in each phase.

4.1 General Optimization

The term *general* means basic optimization principles of application structures. Therefore the following list contains some main principles [6] that were applied in the next sections:

- "Less is more"
- Pre-processing data
- Test invariants of algorithms
- Indexing, batching, sorting
- Object queries

4.1.1 OpenGL Error Handling

In [8] Shreiner and Grantham give an overview about OpenGL error handling. Basically OpenGL provides asynchronous error reporting. So it is favorable to check for errors early and often in application development. Note that only first error is retained [3]. Additional errors are discarded until the error flag will be cleared by `glGetError` call. The C++ macro in appendix B provides a solution for this problem [8].

4.1.2 OpenGL State Changes

Now we focus on OpenGLs synchronization process. The goal is to avoid making redundant mode changes because changing the state do more than just set values in the OpenGL context. It may also require reconfiguration the OpenGL pipeline by selecting a different rasterization routine or a recomputation of vertex or viewpoint independent data in the internal caches [1, 12]. OpenGL state setting commands can be classified into to two different categories:

4.1.2.1. Vertex data commands

Vertex data commands are the calls that can occur between a `glBegin`/ `glEnd` pair, such like `glVertex`, `glColor`, `glIndex`, `glNormal`, and `glEdgeFlag`. The processing of these calls is very fast and does not trigger a validation. So restructuring a program to eliminate some vertex data commands will not significantly improve its performance [18].

4.1.2.2. Modal state setting commands

Which are sometimes also referred to as "mode changes". These commands allow to: turn on/off capabilities, change attribute settings for capabilities, define lights, change matrices, etc. These calls cannot occur between a `glBegin`/`glEnd` pair. Examples are: `glEnable`, `glLoadMatrixf`, `glFogf`, `glLightf`. Changes to the modal state are significantly more expensive to process than simple vertex data commands because of the occurring validation (conceptually see figure 5 [8]). Validation can rob performance from an application and occurs in the transition from state setting to rendering. Validation is a two-step process. At first the determination what data needs to be updated, then the selection of appropriate rendering routines based on the enabled features.
Validations can be minimized by grouping modal state changes, and also by minimizing the number of modal state changes. E.g. try to group primitives by type. That means grouping primitives that require the same modal state together to minimize modal state changes. For example, if only part of a scene's primitives are lighted, draw all the lighted primitives, then turn off lighting and draw all the unlighted primitives, rather than enabling/disabling lighting many times. Or grouping your state changes together (that is, several modal state changes at one time), and render primitives, will provide better performance than doing the modal state changes one by one and intermixing them with primitives. This can be achieved by a method called State Sorting, a simple technique with big payoff. Organize rendering based on the expense of the operation and arrange rendering sequence to minimize state changes [12]. Beware of `glPushAttrib`/ `glPopAttrib` because these operations save lots of OpenGL context data when called. All elements of an attribute group are copied for later. It's almost guaranteed that a validation occurs when calling `glPopAttrib`. Beware also of *"under the covers"* state changes like `GL_COLOR MATERIAL`. This force an update to the lighting cache ever calls to `glColor*`. So context switching is expensive,

keep it to a minimum. If multiple windows are necessary, it is better to re-use a single context by binding it to separate windows.

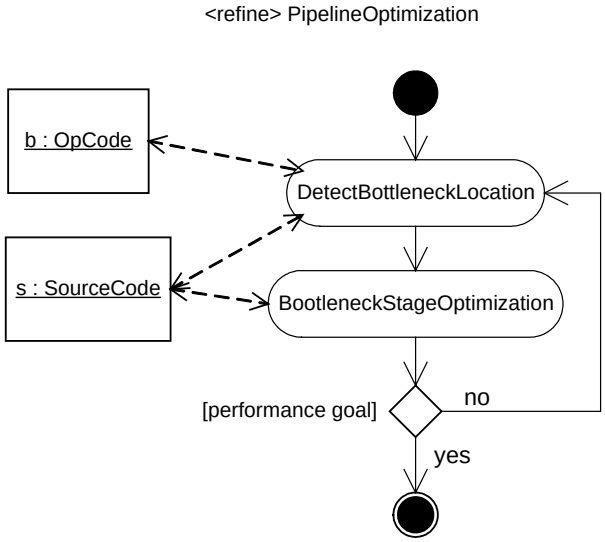


Figure 6. Activity Diagram Pipeline Optimization

4.1.3 Common Techniques

The subject of common techniques can get very extensive. So, it is restricted to a collection of techniques to make suggestions. Some of them could pour in the design phase of the application development process.

- **Indexed and connective primitives.** These are the only way to use the pipeline caches. Use degenerate triangles to join strips together.
- **Batching and sorting.** Use a vertex shader to batch instanced geometry. Sort objects front to back and batches per texture and render states. Eliminate small batches.
- **Occlusion query** should be used to reduce vertex and pixel throughput.
- **Multi-pass rendering.** During the first pass, attach a query to every object to detect if enough pixels have been drawn. Is the amount too small then the subsequent passes could be skipped.
- **Rough visibility determination.** Once by drawing a bounding box with a query to know if a portal or a complex object is visible and if not, skip its rendering. Second example a quad with a query can be used to detect how much of the sun is visible for lens flare.
- **Examine and test** possible resource locking and misplaced GUI event-synchronizations.

By using these methods you have to modify the program source code. Remain the application operation code to mention. Compiler optimization occurs within a compromise of *speed and memory space* vs. *time to compile and link.* Optimize code that uses small amount of the time, or too detailed code optimization is time consuming or rather wasting [6].

4.2 Pipeline Optimization

This section is about locating and eliminating bottlenecks. The activity diagram in figure 6 describes the procedure published in [6]. After detecting the bottleneck, the pipeline stage where it occurs will be optimized by putting enough effort that the bottleneck stage moves where it could be dissolved easily. If the appropriate performance goal is reached the process will be quit. Otherwise it will continue to optimize a new bottleneck (if it has moved). To be able to recognize the potential effects of pipeline optimization we have to measure the total rendering time per frame with double buffering disabled. That means in single-buffer mode because swapping of the buffers occurs only in synchronization with the frequency of the monitor.

Performance Parameter	Logical Pipeline Stage
Amount of data per polygon	All stages
Application overhead	Application
Transform rate and geometry mode setting	Geometry subsystem
Total number of polygons in a frame	Geometry and raster subsystem
Number of pixels filled	Raster subsystem
Fill rate for the current mode settings	Raster subsystem
Duration of screen and/or depth buffer clear	Raster subsystem

Table 4. Factors Influencing Performance

It is important to pay attention to the underlying hardware architecture of the system, which is going to be optimized. The following sections will explain bottlenecks of a single pipeline architecture that can be found on mid-class graphic acceleration cards (see 3.1.2). For further information about multiprocessor or parallel pipeline optimization see i.e. [6]. Table 4 provides an overview of factors that may limit rendering performance and the according pipeline stages [11].

4.2.1 Reducing CPU-limited Bottleneck

No amount of OpenGL transform or rasterization tuning will help if the application is CPU limited, which is often the circumstance. The common factor is computational time often spend in game logic, AI, network, file I/O etc. Graphics should be limited to simple culling and sorting. Extremely rare are driver or API limitations. This could happen by "off the fast path" pathological use of API. The driver should usually spend most of its time idling. The following list contains possible work steps for optimization:

- Revisit application design decisions like use data structures, the implementations of algorithms[10] and the storage formats.
- Use an application profiling tool[11] and performance analyzer (Intel VTune).
- Avoid inefficient context or state management (as describe above)

4.2.2 Reducing Transfer-limited Bottleneck

The cause of the transfer-limited bottleneck is bounded bandwidth of the bus that connects CPU and CPU. Transferring geometric data from the application to GPU couldn't take place just in time. There are two main causes for this problem

- Too much data transferred across the graphic bus. A solution for that problem is to eliminate useless vertex attributes and preprocess data to minimize data sent per vertex. If possible use `GLfloat` instead of `GLdouble` for instance. Use connected primitives such as `GL_LINE_STRIP` or `GL_QUAD_STRIP`. Further (B) too much traffic of dynamic vertices. Decrease number of dynamic vertices by using vertex shaders to animate static vertices, for example. A problem occurs by handling static or immutable data (Display List, fast) and dynamic data (Immediate Mode, not most efficient). The usage of Vertex Array[12] seems to be best of both [16]. In general, use the vector forms of commands to pass pre-computed data, and use the scalar forms of commands to pass values that are computed near call time. (C) Using the right API calls can also prevent poor management of dynamic data. Another cause is (D) overloaded video memory. Therefore make sure frame buffer, textures and static vertex buffers fit into video memory.
- Data transferred in an inadequate format. (A) Vertex size should be multiples of 32 bytes. So adjust vertex size by compressing components and use the vertex shader to decompress or pad to next multiple. (B) Avoid non-sequential use of vertices (pre- and post-T&L cache) through re-order vertices to be sequential in use. This maximizes cache usage.

Another possibility is a misconfigured AGP bus or a too small aperture set [21].

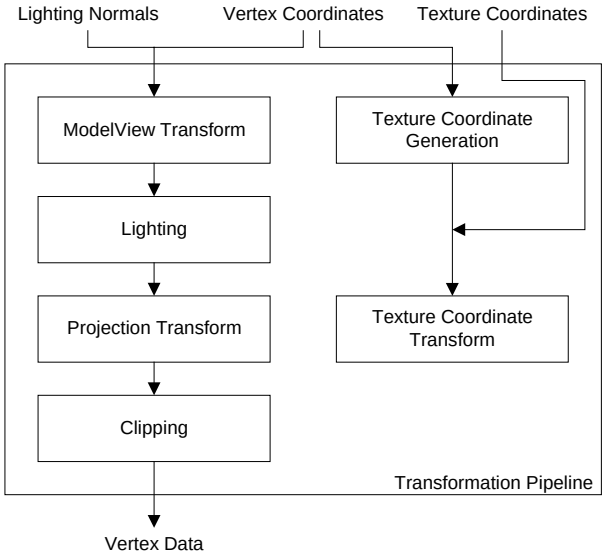


Figure 7. OpenGL Transformation Pipeline

4.2.3 Reducing Transform- limited Bottleneck

The term transform bound or transform-limited often means that the bottleneck is anywhere before the rasterizer (see fig. 7 for a conceptually insight). Mainly three causes are responsible for this problem: too many vertices, too much computation per vertex, or vertex cache inefficiency. The first

[10] traversal methods or scene sort

[11] e.g. pixie and prof, gprof, or other similar tools

[12] see `Compiled_Vertex_Array` and `Vertex_Array_Range` (VAR) Extension

point isn't a problem for two reasons. It can be handled by using level of detail (LOD)[13] and by faking geometric[14]. And further, the GPU has also a lot of vertex processing power (more than $30 \cdot 10^6$ triangles per second). Do less work per vertex as possible. This means:

- Minimize primitives by maximize calls between `glBegin`/`glEnd` to amortize costs over a large number of primitives.
- Use convex and planar polygons; concave and self- intersecting polygons must be tessellated by GLU and non-planar polygons may exhibit shading artifacts.

An important question that arose in the last years is *vertex programs* vs. *fixed function pipeline*. Use fixed function pipeline, for operations that can be performed with it. All other operations should be performed with a vertex program [16]. Vertex programs are not recommended for optimizing fixed function lighting [21]. The following lists of some points, for optimizing fixed function pipeline:

- Lighting considerations and tuning are an opportunity to increase speed and popularity. Different types and modes have different performance characteristics, through different numbers of necessary transformations. This is basically and worth to provide some details here.
  Light modes The use of infinite (directional) lights cause less computation compared to local lights[15]. Infinite lights are fastest with infinite viewer, since half angle vector need not be recomputed for every processed vertex. A frequent change to the `GL_SHININESS` material parameter is typically not for free and two-sided lighting is almost never for free. Performance penalty for two sided lighting vs. one sided is dependent upon the number of lights, `glTexGen*`, and other variables, but should be almost for free for CAD lighting (1 infinite light/ viewer) [16]. If the front and back of polygons shaded the same it is advantageous using two light sources by positioning the light sources on opposite sides. That way, a polygon will always be lit correctly whether it's back or front facing. It is also useful to avoid negative material and light color coefficients.
  Light count Use the amount of hardware accelerated lights sources, because vertex lighting isn't terribly fast when performed on CPU. The performance degradation from one light to two or more lights isn't linear but very fast when performed on GPU. Most today's graphic pipelines support 8 simultaneous lights[16]. Briefs minimize number to maximize performance. So more is in not necessarily better like in case of saturation that often occurs with over four active lights. You can get the effect of lighting by specifying vertex colors instead of vertex normals. Disable normal vector normalization when not needed to avoid the overhead. The GL_NORMALIZE parameter controls whether normal vectors are scaled to unit length before lighting. So, if possible, provide unit length normals and avoid using `glScale` when doing lighting. Note that some game-engines do its own lighting using light maps (blended textures) and does not use the OpenGL lighting path. That this kind of simulated features in texturing (also normal maps) only helps if application not fill-limited.
- Use explicit texture coordinates to avoid the overhead during the generation for complex (dynamic) scenes.

4.2.4 Reducing Setup/Raster-limited Bootlenecks

The setup is never the bottleneck since it is the fastest stage. Rasterization was often the bottleneck in software implementations of OpenGL. The speed of this stage is mainly influenced by the size of the triangles and the number of vertex attributes to be interpolated [21]. It is slowest for extremely small triangles (like degenerate triangles[17]). In the second case disable shading when not needed to avoid computation overhead (e.g. for interpolation). Disable rasterization and per-fragment operations to optimize resources when drawing or copying images. Use back-face culling whenever possible. And maximize depth-culling efficiency [21].

[13] but beware of CPU overhead
[14] e.g. bump maps, to increase visual quality by maintaining geometric complexity
[15] point lights: `GL_LIGHTMODEL_LOCAL_VIEWER`
[16] ATI and Nvidia do so, 3Dlabs varies between 16 or 32
[17] like too small or too long and sharp triangles

4.2.5 Reducing Fragment-limited Bottleneck

The causes of fragment-limited bottlenecks are likely the same as for the vertex bottleneck:

- Too many fragments
- Too much computation per fragment
- Unnecessary fragment operations

This is about avoiding unnecessary per-fragment operations (see figure 8 for amount and order of stages). It is possible to reduce fill requirements by organizing drawing. Using back-face or front-face removal can do this. Specular color summation and fog application is usually for free on many systems like scissor/alpha testing operations. The combination of depth/stencil tests can require a read/modify/write at some cost. The depth buffer should be disabled when drawing large background polygons and enabled to draw more complex depth-buffered objects. Also it *can* be faster to render polygons in front-to-back order [14]. Here for the usage of occlusion culling can reduce the depth complexity of a scene. For the same reason blending modes cut fill rates in half, and should only be used when necessary. The color logical operation `glLogicOp` can make OpenGL system default to software rendering. Anything but default mode (`GL_COPY`) should be avoided for fast passes. Also other fragment operations like polygon stipple became slow in conjunction with texturing [14].

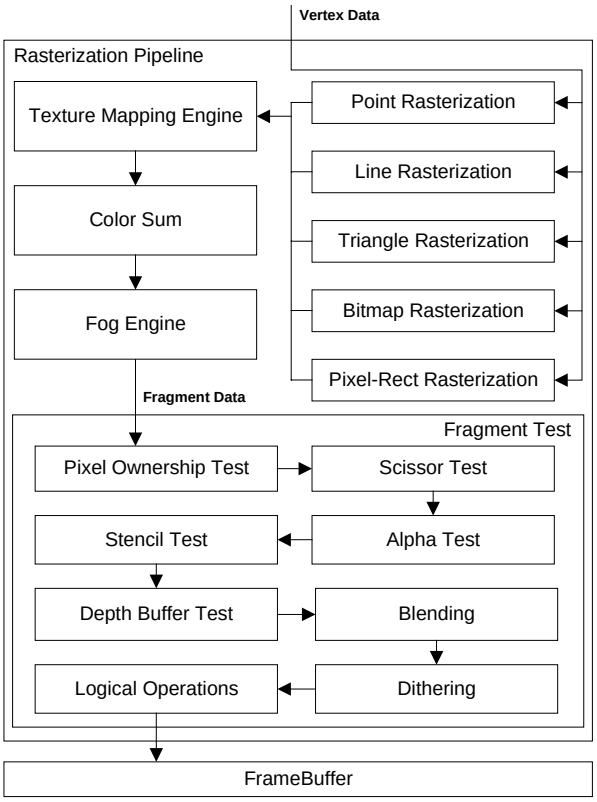


Figure 8. OpenGL Rasterization Pipeline

4.2.6 Reducing Texture-limited Bottleneck

Texturing is hard to optimize [20]. Speed vs. quality requires making it a user settable option. Have a look at figure 9 for separate work stages. The main reasons for this bottleneck [21] are:

- Low texture memory
- Intensive Texture filtering
- Poor texture cache utilization

The following list denotes causes and solutions for these main reasons:

- Guarantee right texture formats. Sometimes textures have too high resolutions or too high measures. This causes an overload of texture cache video memory. So subsequently lots of cache misses or textures are fetched from system memory. There are several solutions for this problem. At first the texture resolutions should be as big as needed and not bigger. Avoid expensive internal formats. Second, compress textures by collapse monochrome channels into alpha and use 16-bit color depth when possible (for environment maps and shadow maps). It is also possible to use palletized texture (for normal maps) or texture compression (S3TC, DTX), if it is available on hardware. See [4] for supported formats.
- Pick the right filtering modes to avoid too many samples per look-up. Trilinear filtering for instance cuts fill rate in half. Anisotropic filtering is even worse and depends on level of anisotropy. So use trilinear or anisotropic filtering only when needed and reduce the maximum ratio of anisotropy in this case. The use of anisotropic filtering for best image fidelity will not be for free, but penalty only occurs when necessary, not on all pixels. Typically, only diffuse maps truly benefit. Light maps posses' too low resolution and environment maps are distorted anyway. Use multi-textures to reduce the number of rendering passes and sort textures to reduce state changes. Still use bilinear mipmapped filtering for best speed when using multi-texturing. The filter modus `LINEAR_MIPMAP_LINEAR` is as fast as bilinear when only a single 2D texture is used [20].
- Use the right texture blend functions. Register combiners for most flexibility could do this. The final combiner is always for free (i.e. full speed) but general combiners can slow down: 1 or 2 can be enabled for free, 3 or 4 can run at one half maximum speed 5 or 6 can run at one third

maximum speed, 7 or 8 can run at one fourth maximum speed [14]. Maximum speed may not be possible due to the choice and number of texture shaders (or other factors) As such, number of enabled general combiners will likely not be the bottleneck in most cases [15, 17].

- Loading and managing the textures efficiently is important for dealing with a lot of textures. Texture up/download performance can be maximized by matching external/internal (native) texture formats.
  Always use `glTexSubImage2D` rather than `glTexImage2D`, if possible. If using copying textures, match texture internal format to that of framebuffer (e.g. 32-bit desktop to `GL_RGBA8`). If using palletized textures, it is possible to share the palette between multiple textures[18]. Lots of cache misses when texture cache is under-utilized. A solution is to localize texture access and beware of dependent texture look-up. Avoid negative LOD bias to sharpen when mipmaps are used. Because texture caches are tuned for standard LOD [20].

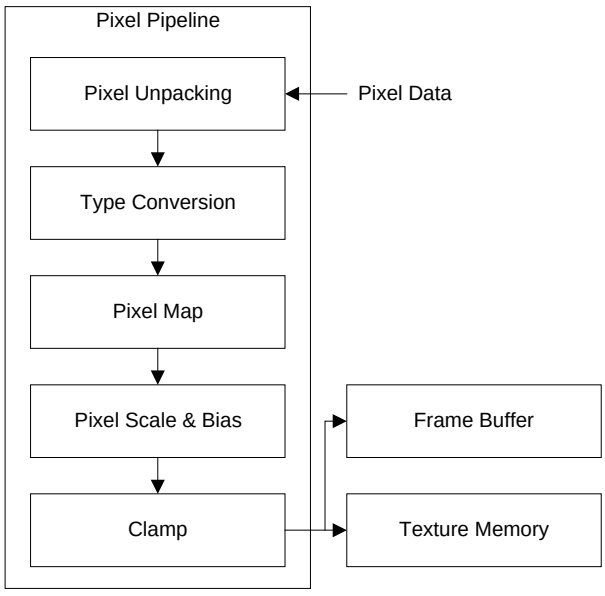


Figure 9. OpenGL Texture Pipeline

4.2.7 Reducing Framebuffer-limited Bottleneck

The main causes for framebuffer-bottlenecks are too much read and write operations on the framebuffer. For example pixel operations like `glDrawPixels`, `glReadPixels`, `glCopyPixels`. So, this is about minimizing framebuffer traffic by matching size and formats[19] closely as possible [21].
There are two easy but not acceptable solutions, which weigh heavenly on the application quality: reducing the frame-rate and decreasing of the view-port size. Better opportunities are skillful usage and configuration of fragment operations. After a screen clear operation, do not fill the graphics pipeline with more graphics command immediately. It is better to do application (CPU)-related work after calling `glClear` . This is not true for accelerators with the technique for fast clear operation implemented. Do not use this call to clear small subregions of the buffer but clear the buffer completely. Concerning about texture shader limitations if you use this technology. Shader language performance tips can be found in [13].

4.3 Pipeline Balancing

After optimizing each pipeline stage it is possible that stages, which are not the bottleneck, possess idle time. This time can be used to increase quality of rendering output by allocating additional work without degradation of performance [6, 20]. Unfortunately it is practically not always so simple to fully use the pipeline. But for the sake of completeness this phase is mentioned here. The relevant stage can be detected by increasing the number of triangles. If the application stage is idle, it is possible to compute more realistic animations, implement more accurate collision detection or use more sophisticated acceleration algorithms and perform other similar tasks. If the geometry stage is idle, use more lights and more expensive light source types. More expensive vertex shader programs can also be used. If the rasterizer stage is idle, use more expensive texture filtering, fog, blending, pixel shaders, etc.

5 Conclusions

We can summarize, that optimization of graphical applications is not a trivial but an important problem. The efforts, which have to make to get optimal performance, can include expense cost of time and experienced personnel by having a minimal performance boost in worst-case. In this case it is better to know the goal before starting optimization. Following list contains a summary of common principles that should be kept in mind.

- Determine requirements of application
- Investigate target platform(s)
- Estimate what hardware is capable of

[18] http://www.3dgpu.com/modules/wfsection/article.php?articleid=37

[19] avoid non-packed 32-bit integer formats, for instance

- Utilize applicable benchmarks
- Have a performance goal
- Organize rendering to minimize workload

To minimize the application workload to runtime it's necessary to pre-process as much data as possible, by guarantee high quality. So no needless conversion time is spending.

An interesting and possible research point in this context is the demands on software architecture. The question of performance delivers the problem that the architecture of the software construction element must be efficient. But this is sometimes a contradiction to a sober and well-defined system. Today's applications have to run on different target platforms[20] and allow scalability by configuration at runtime. This can be done by build in a mini-benchmark to test performance and select rendering paths depending upon the measurements. Not every program could assume a high-end system but should run optimal on desktops. So it is important to structure application code for maximum portability and best performance. In case of OpenGL this is possible because the implementation is available on a multitude of different platforms. But if you consider other graphic libraries like DirectX a complex wrapper is necessary to have an internal data-model at one's disposal, which can be mapped onto OpenGL or DirectX. This result leads toward another problem. Highly object orientated implementations needs to be optimized by the compiler. So the result depends secondarily on the optimization strategies of the used compiler. Note also, that the peculiarity of encapsulation in C++ [8] could be failing to see.

Underneath Donald Knuth words: *"Premature optimization is the root of all evil"* a remark on the influences of the optimization process in software development. Considerations about performance should take place in all phases of the software development process, conceptually from design to implementation phase. Note that today no test-frameworks for this class of problem is available. These thoughts are justified on durable software systems. In future it maybe increasingly important to systematically research different optimization approaches. Thinking about geometry compression, alternative modeling schemes[21] and just beyond geometry. This all can be a challenge for optimization process.

[20] consider about 10 OpenGL language bindings

[21] e.g. parametric surfaces, implicit surfaces, subdivision surfaces, displacement mapping, programmable shaders

References

[1] OpenGL 1.3 for Solaris, Implementation and Performance Guide. Sun Microsystems.

[2] OpenGL on Silicon Graphics Systems. Sun Microsystems.

[3] OpenGL Reference Manual. Addison Wesley, third edition.

[4] OpenGL Specification, Version 1.4.

[5] Algorithms and Theory of Computation Handbook. CRC Press LLC, 2000.

[6] T. Akenine-Moeller and E. Haines. R*eal-Time Rendering*. A.K. Peters Ltd., 2nd edition, 2002.

[7] K. Cok and T. True. Developing Efficient Graphics Software, volume 12. SIGGRAPH Course, 1999.

[8] Dave Shreiner. Performance OpenGL Platform Independent Techniques. 2002.

[9] J. Huang. OpenGL performance techniques & computer architecture implications. 2002.

[10] G. McDaniel. *IBM Dictionary of Computing*. McGraw-Hill Publishing Company, tenth edition, 1993.

[11] T. McReynolds and D. Blythe. Advanced Graphics Programming Techniques Using OpenGL. SIGGRAPH, 1999.

[12] J. Neider, T. Davis, M. Woo, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison Wesley Publishing Company, third edition, 2000.

[13] Nvidia Corporation. Cg Toolkit - User Manual, A Developers Guide to Programmable Graphics. December 2002.

[14] J. Spitzer. Advanced OpenGL Game Development, Maximizing OpenGL Performance for GPUs. March 2000.

[15] J. Spitzer. Texture Compositing With Register Combiners. 2000.

[16] J. Spitzer. OpenGL Performance. 2001.

[17] J. Spitzer. Programmable Texture Blending. 2001.

[18] D. B. Tom McReynolds. Advanced Graphics Programming Techniques Using OpenGL. 1999.

[19] P. Womack. Performance Technique. Silicon Graphics Inc.

[20] C. Zeller. Balancing the graphics pipeline for optimal performance. 2000.

[21] J. Spitzer, E. Hart. OpenGL Performance Tuning. Nvidia Corporation. 2003

[22] Daniel Etiemble. Characterization of graphics activities in PC benchmarks by bottleneck detection. LRI, Université Paris Sud

Appendix

A Detecting Bottlenecks

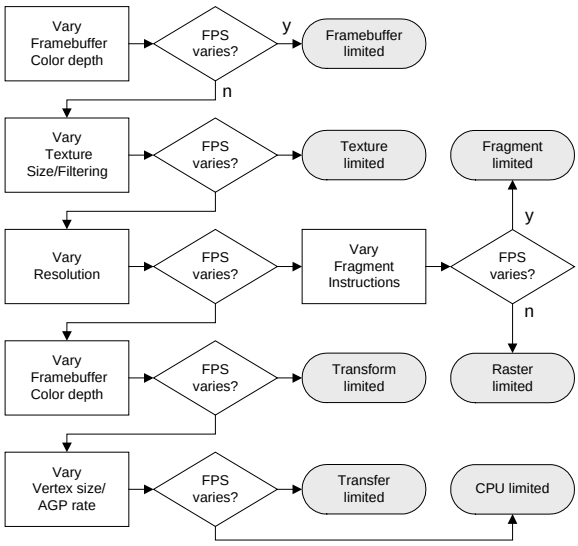
A.1 Transfer-limited Macro

Use this C++ macro for determining applications data-transfer speed.

```
#define glVertex2d(x,y) glNormal3d(x,y,0)
#define glVertex2f(x,y) glNormal3f(x,y,0)
#define glVertex2i(x,y) glNormal3i(x,y,0)
#define glVertex2s(x,y) glNormal3s(x,y,0)
#define glVertex3d(x,y,z) glNormal3d(x,y,z)
#define glVertex3f(x,y,z) glNormal3f(x,y,z)
#define glVertex3i(x,y,z) glNormal3i(x,y,z)
#define glVertex3s(x,y,z) glNormal3s(x,y,z)
#define glVertex4d(x,y,z,w) glNormal3d(x,y,z)
#define glVertex4f(x,y,z,w) glNormal3f(x,y,z)
#define glVertex4i(x,y,z,w) glNormal3i(x,y,z)
#define glVertex4s(x,y,z,w) glNormal3s(x,y,z)
#define glVertex2dv(v) glNormal3d(v[0],v[1])
#define glVertex2fv(v) glNormal3f(v[0],v[1])
#define glVertex2iv(v) glNormal3i(v[0],v[1])
#define glVertex2sv(v) glNormal3s(v[0],v[1])
#define glVertex3dv(v) glNormal3dv(v)
#define glVertex3fv(v) glNormal3fv(v)
#define glVertex3iv(v) glNormal3iv(v)
#define glVertex3sv(v) glNormal3sv(v)
#define glVertex4dv(v) glNormal3dv(v)
#define glVertex4fv(v) glNormal3fv(v)
#define glVertex4iv(v) glNormal3iv(v)
#define glVertex4sv(v) glNormal3sv(v)
```

A.2 Strategy to Detecting Bottlenecks

This exclusion proceeding can be used to detect bottlenecks in OpenGL applications. The test-order results of the test complexity of each pipeline stage.



B OpenGL Error-Handling

Limitation: This macro cannot be use inside of `glBegin`/ `glEnd` pair.

```
#define CHECK_OPENGL_ERROR( cmd ) \
cmd; \
{ GLenum error; \
while ( (error = glGetError) != 
GL_NO_ERROR) { \
printf( "[%s:%d] ’%s’ failed with error
%s\n", \
__FILE__, __LINE__, #cmd, \
gluErrorString(error) ); \}
```

Checking errors more thoroughly through modified `gl.h`. This macro checks almost every situation.

```
#define glBegin( mode ) \
if ( __glDebug_InBegin ) { \
printf( "[%s:%d] glBegin( %s ) called
between" \
"glBegin/glEnd pair\n", \
__FILE__, __LINE__, #mode ); \
} else { \
__glDebug_InBegin = GL_TRUE; \
glBegin( mode ); \
}
```