The Design of a Parallel Graphics Interface

Homan Igehy Gordon Stoll Pat Hanrahan

Computer Science Department

Stanford University

Abstract

It has become increasingly difficult to drive a modern high-performance graphics accelerator at full speed with a serial immediate-mode graphics interface. To resolve this problem, retained-mode constructs have been integrated into graphics interfaces. While retained-mode constructs provide a good solution in many cases, at times they provide an undesirable interface model for the application programmer, and in some cases they do not solve the performance problem. In order to resolve some of these cases, we present a parallel graphics interface that may be used in conjunction with the existing API as a new paradigm for high-performance graphics applications.

The parallel API extends existing ideas found in OpenGL and X11 that allow multiple graphics contexts to simultaneously draw into the same image. Through the introduction of synchronization primitives, the parallel API allows parallel traversal of an explicitly ordered scene. We give code examples which demonstrate how the API can be used to expose parallelism while retaining many of the desirable features of serial immediate-mode programming. The viability of the API is demonstrated by the performance of our implementation which achieves scalable performance on a 24 processor system.

CR Categories and Subject Descriptors: C.0 [Computer Systems Organization]: Hardware/Software Interfaces; D.1.3 [Programming Techniques]: Concurrent Programming; I.3.1 [Computer Graphics]: Hardware Architecture.

1 INTRODUCTION

Computer graphics hardware has been rapidly increasing in performance. This has motivated immediate-mode graphics interfaces like OpenGL [20] to adopt constructs such as display lists and packed vertex arrays in order to alleviate system bottlenecks. However, these constructs may impose an undesired paradigm shift for the application programmer, and they may not be useful in resolving the particular performance bottleneck. Furthermore, with the increasing use of multiprocessor systems for graphics applications, a serial interface to the graphics system can be inelegant. A parallel graphics interface seeks to resolve these issues. To provide a common framework, we first review three key issues in graphics interfaces.

{homan,gws,hanrahan}@graphics.stanford.edu

The first issue in designing a graphics interface is state. In a stateless interface, the behavior of every command is independent of every other command; thus, every command must include all the information required to execute that command. Conversely, in an interface with state, a command's behavior can be affected by previous commands. Some of the information required for the execution of commands may reside within the state maintained by the interface, and some commands modify that state. While stateless interfaces simplify many issues (especially with regard to parallelism), they are not well-suited to full-featured rendering systems. The problem is that a large amount of data is needed for each drawing command, and much of it is changed infrequently. Respecifying this data with each primitive is tedious and inefficient, so most graphics interfaces contain state.

The second key issue is whether the graphics interface is immediate-mode or retained-mode. In an immediate-mode API, the application sends commands to the graphics system one at a time, and the graphics system executes them more or less immediately. In a retained-mode API, the application first specifies an entire scene that is built on the graphics system and subsequently requests the scene to be rendered with certain viewing parameters. Though retained-mode interfaces can sometimes provide performance benefits, programmers prefer immediate-mode interfaces due to their flexibility and ease of use. Many well-designed interfaces use the best of both worlds: they are based on immediate-mode semantics, and some retained-mode constructs are included to allow performance benefits.

The third major issue that arises in graphics interface design is ordering. Ordering semantics are the set of rules that constrain the order in which API commands may be executed. In a strictly ordered interface, primitives must be drawn in the order in which they are specified. This behavior is essential for many algorithms such as the placement of ground-plane shadows [3] and transparency through alpha-compositing [21]. Sometimes, however, a programmer may not care whether or not primitives are drawn in the order specified. For example, depth buffering alleviates the need for drawing a scene of opaque 3D primitives in any particular order. In these cases, the programmer would gladly use less constrained ordering semantics if it meant increased performance.

In the rest of the paper, we present the motivations and issues involved in designing a parallel extension to a serial immediate-mode graphics interface with strict ordering and state. By adding synchronization commands (such as barriers and semaphores) into multiple graphics command streams, application threads can issue explicitly ordered primitives in parallel without blocking. We also introduce the notion of a wait context command for synchronizing contexts at the level of application

threads. Given the resulting API, we explore how an application programmer would attain parallel issue of graphics commands. These ideas are demonstrated by an implementation which achieves scalable performance on a 24 processor system. We also discuss the issues surrounding the various ways of implementing the parallel API. Very little research has been done in the area of parallel graphics interfaces. This paper provides a common framework on which a new class of research and commercial systems can be built as well as a common framework on which a new class of parallel algorithms can be designed.

2 MOTIVATION

Graphics interfaces have been around for many years, so why investigate a parallel API now? While it is true that some applications are most naturally expressed through a parallel graphics API, the main motivation is performance: it is becoming more and more difficult to drive a graphics system at full speed using a single CPU. First we look into the reasons behind this, and then we examine possible solutions.

2.1 Performance Limitations

Although graphics systems are on the same technology curve as microprocessors, graphics systems have reached a level of performance at which they can process graphics commands faster than microprocessors can produce them: a single CPU running an immediate-mode interface cannot keep up with modern graphics hardware. This is primarily due to an increasing use of parallelism within graphics hardware. Within a computer, there are three sources of bottlenecks in a graphics application. First, performance may be limited by the speed of the graphics system. In this case, the only solution is to use a faster graphics system. Second, performance may be limited by the rate of data generation. In this case, the programmer can use either a faster data generation algorithm or else, if the algorithm is parallelizable, multiple processors. Third, performance may be limited by the interface between the host system and the graphics system. Possible sources of this limitation are:

1) Overhead for encoding API commands.
2) Data bandwidth from the API host.
3) Data bandwidth into the graphics system.
4) Overhead for decoding API commands.

2.2 Performance Solutions

There are several possible ways to extend a serial immediate-mode API in order to address the interface bottlenecks. First, we describe two techniques that are currently in widespread use, packed primitive arrays and display lists. Then we describe a proposed technique, compression. Finally, we describe our proposal, a parallel graphics interface. It is important to note that many of these techniques can complement each other in resolving performance bottlenecks.

2.2.1 Packed Primitive Arrays

A packed primitive array is an array of primitives that reside in system memory. By using a single API call to issue the entire array of primitives instead of one API call per primitive, the cost of encoding API commands is amortized. Furthermore, because the arrays may be transferred by direct memory access (DMA), bandwidth limitations from the API processor may be bypassed. Nothing is done, however, about the bandwidth limitations into the graphics system. Furthermore, although the decoding may be somewhat simplified, all the primitives in the array still have to be decoded on the graphics system. While packed primitive arrays are useful in a wide variety of applications, they may introduce an awkward programming model.

2.2.2 Display Lists

A display list is a compiled set of graphics commands that resides on the graphics system. In a fashion similar to retained-mode interfaces, the user first specifies the list of commands to be stored in the display list and later invokes the commands within the display list. Because they are essentially command macros, display lists work well semantically with immediate-mode interfaces. In cases where the scene is small enough to fit in the graphics system and the frame-to-frame scene changes are modest, display lists trivially resolve the first three bottlenecks. If the scene is too large and therefore must reside in system memory, display lists are similar to packed primitive arrays and only the first two bottlenecks are resolved. Display lists provide an excellent solution for performance bottlenecks if the same objects are drawn from frame to frame. But on applications that recompute the graphics data on every frame (e.g., [11, 24]), display lists are not useful. Furthermore, the use of display lists burdens the programmer with the task of managing handles to the display lists.

2.2.3 Compression

Whereas the idea of quantizing the data sent through the API has been used for quite some time, the idea of compressing the data has only recently been proposed. One system compresses the geometric data sent through the API [6]; other systems compress the texture data [2, 25]. All compression schemes increase the decoding costs, and systems which compress the data interactively increase the encoding costs. Systems which compress the data off-line, on the other hand, are useful only when the graphics data does not change.

2.2.4 Parallel Interface

The motivation behind a parallel graphics interface is scalability: bottlenecks are overcome with increased parallelism. If the graphics system is too slow, it can be scaled by adding more graphics nodes. If the data generation is too slow, more processors can be used to generate the data in parallel. Similarly, if the serial interface is too slow, then it should be parallelized. In a system with a single graphics port, a parallel API can be used to overcome the first two interface limitations. However, by building a scalable system with multiple graphics ports, all interface limitations can be overcome.

There are many challenges to designing a good parallel graphics interface; in formulating our design, we had several goals in mind. First and foremost were the ability to issue graphics primitives in parallel and the ability to explicitly constrain the ordering of these primitives. Ideally, the API should allow parallel issue of a set of primitives that need to be drawn in an exact order. The parallel API should be a minimal set of extensions to an immediate-mode interface such as OpenGL, and it should be compatible with existing features such as display lists. The design is constrained by the presence of state; this is required for a large feature set. A well designed parallel interface should be intuitive and useful in a wide variety of applications. And finally, the new API should extend the current framework of graphics architectures to provide a rich set of implementation choices.

3 RELATED WORK

In the field of parallel graphics interfaces, Crockett introduced the Parallel Graphics Library (PGL) for use in visualizing 3D graphics data produced by message-passing supercomputers [5]. Due to the characteristics of its target architecture and target applications, PGL was designed as a retained-mode interface. In parallel, each processor adds objects to a scene by passing pointers to graphics data residing in system memory. A separate command is used to render the objects into a framebuffer, and no ordering constraints are imposed by the interface. PixelFlow [8, 18] is another system designed to support multiple simultaneous inputs from a parallel host machine, and PixelFlow OpenGL includes extensions for this purpose. However, due to the underlying image composition architecture, PixelFlow OpenGL also imposes frame semantics and does not support ordering. Because of these constraints, PGL and PixelFlow OpenGL do not meet the requirements of many graphics applications.

The X11 window system provides a parallel 2D graphics interface [9, 23]. A client with the proper permissions may open a connection to an X server and ask for X resources to be allocated. Among these resources are drawables (which are on- or off-screen framebuffers) and X contexts (which hold graphics state). Since resources are globally visible, any client may subsequently use the resource within X commands. Since X drawing calls always include references to a drawable and an X context, client requests are simply inserted into a global queue and processed one at a time by the X server. Though it is not explicitly encouraged, multiple clients may draw into the same drawable or even use the same graphics context.

While a 3D graphics interface was beyond the scope of the original design of X, OpenGL is a 3D interface that has been coupled with X. For explicitness, OpenGL within X will serve as our example API due to its popularity and elegant design [20]. OpenGL is an immediate-mode interface whose state is kept within an X resource called the GLX context. In the interest of efficiency, both display lists and packed primitive arrays are supported. Furthermore, both texture data and display lists may be shared between contexts in order to allow the efficient sharing of hardware resources amongst related contexts [12].

Strict ordering semantics are enforced in X and OpenGL: from the point of view of the API, every command appears to be executed once the API call returns. However, in the interest of efficiency, both interfaces allow implementations to indefinitely buffer commands. This introduces the need for two types of API calls. Upon return from the *flush* call (XFlush, glFlush), the system guarantees that all previous commands will execute in a finite amount of time from the point of view of the drawable. Upon return from a *finish* call (XSync, glFinish), the system guarantees that all previous commands have been executed from the point of view of the drawable.

Since OpenGL and X solve different problems, programs often use both. Because of buffering, however, a program must synchronize the operations of the two streams. Imagine a program that wants to draw a 3D scene with OpenGL and then place text on top of it with X. It is insufficient to simply make the drawing calls in the right order because commands do not execute immediately. Furthermore, a flush is insufficient because it only guarantees eventual execution. A finish, on the other hand, guarantees the right order by forcing the application to wait for the OpenGL commands to execute before issuing X commands. In a sense, however, the finish is too much: the application need not wait for the actual execution of the OpenGL commands; it only needs a guarantee that all prior OpenGL commands execute before any subsequent X commands. The call glXWaitGL provides this guarantee, and glXWaitX provides the complement.

Hardware implementations of OpenGL typically provide support for a single context, and sharing of the hardware is done through a context switch. Though context switches are typically inexpensive enough to allow multiple windows, they are expensive enough to discourage fine-grained sharing of the graphics hardware between application threads. A few architectures actually provide hardware support for multiple simultaneous contexts drawing into the same framebuffer [13, 26], but all commands must go through a single graphics port. Furthermore, these architectures do not have a mechanism for maintaining the parallel issue of graphics commands when an exact ordering of primitives is desired.

4 THE PARALLEL API EXTENSIONS

While OpenGL is not intended for multithreaded use in most implementations, the interface provides mechanisms for having multiple application threads work simultaneously on the same image. In this section, we first demonstrate how such an interface may be used to attain parallel issue of graphics commands. Then, we show how additional extensions can be used to increase the performance of parallel issue. The specification of the API extensions is given in Figure 1, and the reader is encouraged to look back to it as necessary.

The API extensions are most easily motivated through the use of an example. Suppose that we want to draw a 3D scene composed of opaque and transparent objects. Though depth buffering alleviates the need to draw the opaque primitives in any particular order, blending arithmetic requires that the transparent objects be drawn in back-to-front order after all the opaque objects have been drawn. By utilizing the strict ordering semantics of the serial graphics API, a serial program simply issues the primitives in the desired order. With a parallel API, order must be explicitly constrained. We assume the existence of two arrays: one holds opaque primitives, and the other holds transparent primitives in

```
glpNewBarrier(GLuint barrier, GLuint numCtxs)
     barrier->numCtxs = numCtxs;
     barrier->count = numCtxs;

glpBarrier(GLuint barrier)
     barrier->count--;
     if (barrier->count == 0)
       barrier->count = barrier->numCtxs;
       signal(all waiting contexts);
     else
       wait();

glpDeleteBarrier(GLuint barrier)
```

```
glpNewSema(GLuint sema, GLuint count)
     sema->count = count;

glpPSema(GLuint sema)
     if (sema->count == 0)
       wait();
     sema->count--;

glpVSema(GLuint sema)
     sema->count++;
     signal(one waiting context, if any);

glpDeleteSema(GLuint sema)
```

```
glpWaitContext(GLXContext ctx)
  Upon return, all subsequent commands from the issuing
  context are guaranteed to execute after all prior commands
  from ctx have finished execution.
```

Figure 1: The Parallel Graphics Interface Extensions.

back-to-front order. We also assume the existence of the following function:

```
DrawPrimitives(prims[first..last])
   glBegin(GL_TRIANGLE_STRIP)
   for p = first..last
      glColor(prims[p].color)
      glVertex(prims[p].coord)
   glEnd()
```

4.1 Existing Constructs

As a first attempt at parallel issue, imagine two application threads using the same context to draw into the same framebuffer. In such a situation, a "set current color" command intended for a primitive from one application thread could be used for a primitive from the other application thread. In general, the sharing of contexts between application threads provides unusable semantics because of the extensive use of state. By using separate contexts, dependencies between the state-modifying graphics commands of the two streams are trivially resolved. Given two application threads using separate contexts on the same framebuffer, the following code could be used to attain parallel issue of the opaque primitives:

```
Thread1                              Thread2
DrawPrimitives(opaq[1..256])         DrawPrimitives(opaq[257..512])
                                     glFinish()
appBarrier(appBarrierVar)            appBarrier(appBarrierVar)
DrawPrimitives(tran[1..256])
glFinish()
appBarrier(appBarrierVar)            appBarrier(appBarrierVar)
                                     DrawPrimitives(tran[257..512])
```

Both application threads first issue their share of opaque primitives without regard for order. After synchronizing in lock-step at the application barrier, *Thread1* issues its half of the transparent primitives. These transparent primitives are guaranteed to be drawn in back-to-front order after *Thread1*'s share of opaque primitives through the strict ordering semantics of the serial API. They are also guaranteed to be drawn after *Thread2*'s share of opaque primitives through the combination of the barrier and the finish: the finish guarantees the drawing of all previously issued commands from *Thread2*. By using this same synchronization mechanism again, *Thread2*'s share of transparent primitives are then drawn in back-to-front order after *Thread1*'s share of transparent primitives.

4.2 The Wait Construct

One inefficiency in the above code is the use of the finish command; in a sense, it is too much. Synchronization between the threads does not require the actual execution of the graphics commands; it only requires a guarantee on the order of execution between the two graphics streams. In a fashion similar to what is used for synchronizing X and OpenGL, we introduce the *wait context* call in order to make guarantees about the execution of commands between contexts. We refer the reader to Figure 1 for an exact specification. In synchronization situations, the wait call is more efficient than the finish call because it does not require any application thread to wait for the completion of graphics commands. The following code demonstrates how the example scene may be drawn using the wait command:

```
Thread1                              Thread2
DrawPrimitives(opaq[1..256])         DrawPrimitives(opaq[257..512])
appBarrier(appBarrierVar)            appBarrier(appBarrierVar)
glpWaitContext(Thread2Ctx)
DrawPrimitives(tran[1..256])
appBarrier(appBarrierVar)            appBarrier(appBarrierVar)
                                     glpWaitContext(Thread1Ctx)
                                     DrawPrimitives(tran[257..512])
```

Intergraph uses a different mechanism to provide the same effect as the wait call [13]. Due to the underlying implementation of the single graphics port, returning from a flush call guarantees that all of a context's primitives will be drawn before any subsequent primitives from any other context. While similar in spirit to the wait call, this mechanism does not scale very well to systems with multiple ports because of its underlying broadcast requirement. The wait construct uses point-to-point communication.

4.3 Synchronization Constructs

While the wait command provides an improvement over the finish command, a large problem remains: the synchronization of the graphics streams is done by the application threads. Consequently, application threads are forced to wait. But why should an application thread wait when it could be doing something more useful? For example, in the code of Section 4.2, the first thread must issue its entire half of the transparent primitives before the second thread can begin issuing its half; thus, the second thread is forced to wait. Every time an explicit ordering is needed between primitives from different threads, the interface essentially degrades to a serial solution.

The answer to this problem is the key idea of our parallel API: synchronization that is intended to synchronize graphics streams should be done between graphics streams—not between application threads. To this end, we introduce a graphics barrier command into the API. As with other API calls, the application thread merely issues the barrier command, and the command is later executed within the graphics system. Thus, the blocking associated with the barrier is done on graphics contexts, not on the application threads. The code below achieves our primary objective—the parallel issue of explicitly ordered primitives: both application threads may execute this code without ever blocking if the graphics system provides sufficient buffering.

```
Thread1                              Thread2
DrawPrimitives(opaq[1..256])         DrawPrimitives(opaq[257..512])
glpBarrier(glpBarrierVar)            glpBarrier(glpBarrierVar)
DrawPrimitives(tran[1..256])
glpBarrier(glpBarrierVar)            glpBarrier(glpBarrierVar)
                                     DrawPrimitives(tran[257..512])
```

We see the utility of the barrier primitive in the above code example, but what other synchronization primitives provide useful semantics within the realm of a parallel graphics interface? Whereas the barrier is an excellent mechanism for synchronizing a set of streams in lock-step, it is not the best mechanism for doing point-to-point synchronization. Borrowing from the field of concurrent programming, we find that semaphores provide an elegant solution for many problems [7]. Among them is a mechanism for signal-and-wait semantics between multiple streams. The specification of the barrier and semaphore commands can be found in Figure 1. As with texture data and display lists, the data associated with barriers and semaphores should be sharable between contexts.

Barriers and semaphores have been found to be good synchronization primitives in the applications we have considered. If found to be useful, other synchronization primitives can also be added to the API. It is important to note that the requirements for synchronization primitives within a graphics API are somewhat constrained. Because the expression of arbitrary computation through a graphics API is not feasible, a synchronization primitive's utility cannot rely on computation outside of its own set of predefined operations. For example, condition variables are not a suitable choice.

```
Serial                                                    (a)
loop:
  glClear()
  get user input
  compute & draw
  glXSwapBuffers()
  glFinish()
```

```
Master                                                    (b)
loop:
  glClear()
  get user input
  appBarrier(appBarrierVar)

  compute & draw
  glpBarrier(glpBarrierVar)
  glXSwapBuffers()
  glFinish()
```

```
Slave                                                     (c)
loop:


  appBarrier(appBarrierVar)
  glpWaitContext(masterCtx)
  compute & draw
  glpBarrier(glpBarrierVar)
```

Figure 2: A Simple Interactive Loop. Application computation and rendering are parallelized across slave threads, and a master thread coordinates per-frame operations.

5 USING THE PARALLEL GRAPHICS API

In order to illustrate how the parallel graphics interface may be used to provide scalable command issue, we present two examples. The first example is a generic interactive loop, and the second example is the marching cubes algorithm.

5.1 Simple Interactive Loop

Figure 2a shows a simple interactive loop expressed in a strictly ordered serial interface. The goal in this example is to parallelize the compute and draw stage. This can yield improved performance in the host computation, the issue of the graphics commands, and the execution of graphics commands.

For parallel issue, a master thread (Figure 2b) creates a number of slave threads (Figure 2c) to help with the compute and draw stage. The master first issues a clear command and gets the user input. The application barrier ensures that the worker threads use the correct user input data for the rendering of each frame. This synchronizes the application threads, but not the graphics command streams. The slaves issue wait commands to ensure that the clear command issued by the master is executed first. The master is assured that the clear occurs first due to the strict ordering semantics of a single stream. After each thread issues its graphics commands, a graphics barrier is issued to restrict the swap operation to occur only after all the graphics streams have finished drawing their share of the frame. Finally, a finish operation is needed to ensure that the image is completed and displayed before getting user input for the next frame. The finish itself is a context-local operation which only guarantees that all previous commands issued by the master are completed. However, in conjunction with the graphics barrier, the finish guarantees that the commands of the slaves are also completed.

5.2 Marching Cubes

As a more demanding example, we consider the marching cubes algorithm [16]. Marching cubes is used to extract the polygonal approximation of an isosurface of a function which is sampled on a 3D grid. The grid is divided into a set of cells, and each cell is composed of one or more voxels. In Figure 3a, we present a simplification of marching cubes to 2D. The mechanics of surface extraction and rendering are abstracted as *ExtractAndRender*. *ExtractAndRender* operates on a single cell of the grid independently. If any portion of the desired isosurface lies within the cell, polygons approximating it are calculated and issued to the graphics system immediately. Due to the grid structure, it is fairly simple to perform the traversal in back-to-front order based on the current viewpoint, thus eliminating the need for depth buffering and allowing for alpha-based translucency. In our example, this corresponds to traversing the grid in raster order.

```
MarchSerialOrdered (M, N, grid)                           (a)
  for (i=0; i<M; i++)
    for (j=0; j<N; j++)
      ExtractAndRender(grid[i, j])
```

```
MarchParallel (M, N, grid)                                (b)
  for (i=0; i<M; i++)
    for (j=(myProc+i)%numProcs; j<N; j+=numProcs)
      ExtractAndRender(grid[i,j])
```

```
MarchParallelOrdered (M, N, grid, sema)                   (c)
  for (i=0; i<M; i++)
    for (j=(myProc+i)%numProcs; j<N; j+=numProcs)
      if (i>0) glpPSema(sema[i-1,j])
      if (j>0) glpPSema(sema[i,j-1])
      ExtractAndRender(grid[i,j])
      if (i<M-1) glpVSema(sema[i,j])
      if (j<N-1) glpVSema(sema[i,j])
```

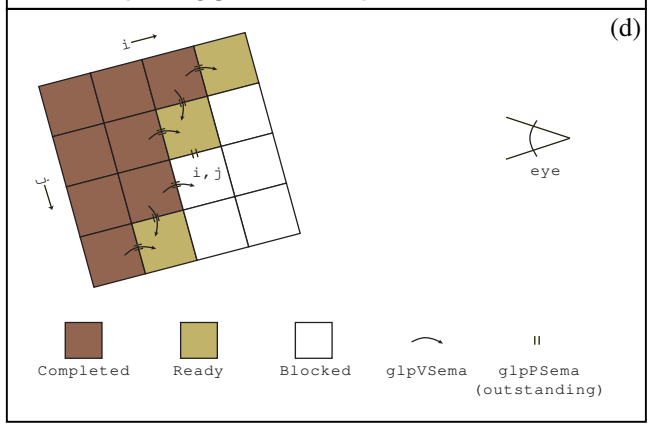


Figure 3: Parallel Marching Cubes. As the rendering of a cell completes, glpVSema operations are performed by the graphics context to release dependent neighboring cells closer to the eye. The rendering commands of the white cells are blocked on glpPSema operations which are waiting for the rendering of adjacent or more distant cells.

Due to the independence of the processing of different cells, marching cubes is easily parallelized. In Figure 3b, traversal is parallelized by interleaving the cells across processing elements. Unfortunately, this simple approach sacrifices back-to-front ordering. Figure 3d illustrates the dependence relationships between cells and their neighbors which must be obeyed in the ordered drawing of primitives. These dependencies can be expressed directly using semaphores injected into the graphics command streams. Such an implementation is shown in Figure 3c. Before processing a cell, a thread issues two *P* operations to constrain the rendering of a cell to occur after rendering of its two rear neighbor cells. After processing the cell, it issues two *V* operations to signal the rendering of its other neighbors. Note that the dependencies and traversal order given here are non-ideal; more efficient (and more complicated) approaches are possible.

6 IMPLEMENTATION

In order to test the viability of the parallel API extensions, we have implemented a software graphics library which is capable of handling multiple simultaneous graphics contexts. The name of this implementation is Argus, and the performance achieved with the parallel API using this system demonstrates the utility and feasibility of the ideas presented in this paper.

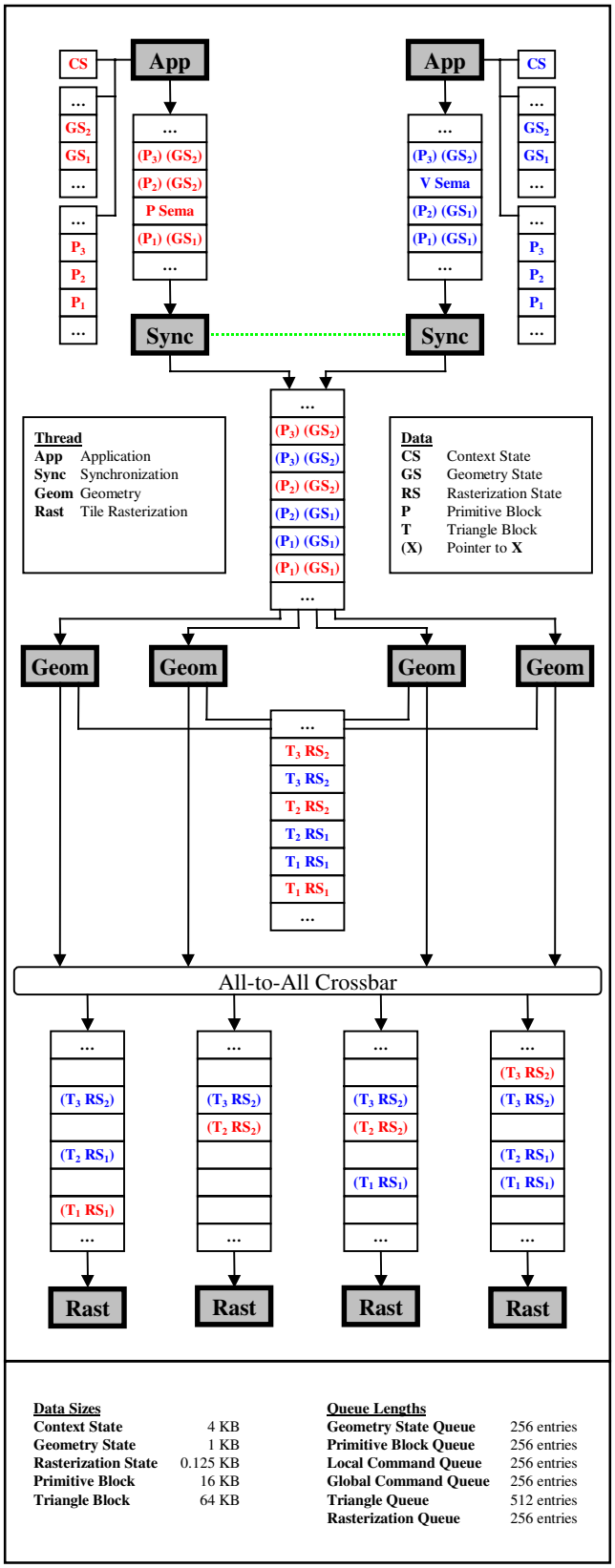
The Argus Pipeline

The diagram on the right shows the flow of data through the Argus pipeline. The pipeline contains several threads, shown as gray boxes, which communicate through a variety of queues. In this example, two application threads are drawing into the same framebuffer through two different contexts. The graphics data from the two contexts is shown in red and blue.

One key design issue that comes up in implementing the parallel API is the handling of the graphics state since most commands affect rendering through state changes. API commands are issued by the 'App' threads shown at the top of the diagram. Commands that modify state which is not necessary for the rendering of the current GL primitive (e.g., the bottom entries of the matrix stack) are tracked in the context state (e.g., CS). Commands that modify state which is necessary for rendering the current GL primitive (e.g., the top entry of the matrix stack) are tracked in the current geometry state (e.g., GS_2), but old versions of the geometry state (e.g., GS_1) are kept until they are no longer needed by the rest of the pipeline. Commands which specify the current primitive (i.e., commands which are allowed within glBegin and glEnd, such as glNormal and glVertex) are grouped into fixed-size primitive blocks (denoted by P_i). A primitive block and its related geometry state contain all the information necessary for the rendering of the primitives, and multiple primitive blocks can share the same geometry state. For example, primitive blocks P_2 and P_3 both use the same geometry state GS_2. Every time a primitive block fills up or the geometry state changes, a pair of pointers (which are represented in the diagram by parentheses) is added to the local command queue (LCQ) by the 'App' thread; synchronization commands (Sema) are inserted into this queue directly.

Another key implementation design issue that comes up in any parallel API implementation is the merging of graphics streams and the resolution of synchronization commands. In Argus, each context has a 'Sync' thread which is responsible for moving data from its LCQ onto a global command queue (GCQ). 'Sync' threads execute the synchronization commands found in the LCQ (as illustrated by the dotted green line). When 'Sync' threads are not blocked due to synchronization, they copy the pointers from their LCQ onto the GCQ. This creates a sequence in the GCQ which is strictly ordered with respect to any one context and consistent with the constraints imposed by the synchronization commands. For example, the sequence found in the GCQ of the diagram keeps the order {P_1, P_2, P_3} and {P_1, P_2, P_3}. The sequence is also consistent with the semaphore pair (which requires an ordering that puts {P_1, P_2} in front of {P_2, P_3}).

Beyond the GCQ, the Argus pipeline is very similar to a graphics pipeline which implements a serial API. The 'Geom' threads drain the GCQ and fill the triangle queue by converting the geometry state (GS_i) and the 3D data from primitive blocks (P_i) into rasterization state (RS_i) and 2D triangle blocks (T_i). Each 'Rast' thread is responsible for drawing into one tile of the framebuffer, and the 'Geom' threads insert pointers into the appropriate rasterization buffers based on the tiles which are overlapped by the triangles in the triangle block. These reorder buffers are used as a mechanism for maintaining the ordering found in the GCQ across the rasterizers.



Data Sizes	
Context State	4 KB
Geometry State	1 KB
Rasterization State	0.125 KB
Primitive Block	16 KB
Triangle Block	64 KB

Queue Lengths	
Geometry State Queue	256 entries
Primitive Block Queue	256 entries
Local Command Queue	256 entries
Global Command Queue	256 entries
Triangle Queue	512 entries
Rasterization Queue	256 entries

6.1 Argus

Argus is a shared memory multiprocessor graphics library which was designed to serve as a test-bed for various studies in graphics architecture. Argus implements a subset of OpenGL as well as the parallel API extensions. At the heart of Argus is a lightweight multiprocessor threads package. We implement a graphics architecture by allocating a thread for each processing node in the architectural design. A custom scheduler is used to schedule these threads onto system processors appropriately. Furthermore, if a system processor running application code is blocked for some reason due to the graphics (e.g., a buffer fills up or a glFinish is pending), the threads package will run graphics threads on the otherwise idle application processor.

There are three basic types of threads in the serial API version of Argus. An application thread runs application code and manages the graphics context. A geometry thread transforms and shades the primitives encoded in the graphics instruction stream. A rasterization thread is responsible for drawing these transformed primitives into the framebuffer. The version of Argus that implements the serial API is a sort-middle tiled parallel graphics system [17]. Graphics commands from a single application thread fill a global command queue which is drained by many geometry threads. The number of geometry threads is scalable since the data in this global command queue can be read in parallel. Of course, the geometry threads must synchronize at a single point of contention in order to distribute the work in the queue amongst themselves, but because the contention is amortized over a large number of primitives, this cost is insignificant in our implementation. After the appropriate computation, the geometry threads distribute the transformed primitives among the appropriate tile rasterizers. Though the details are beyond the scope of this paper, reorder buffers in front of each rasterizer are used to maintain the ordering found in the global command queue across the rasterizers. Since each tile rasterizer is responsible for a contiguous portion of the screen, no one rasterizer needs to see all of the primitives; thus, the rasterization architecture is scalable. Argus supports a variety of schemes for load balancing tile rasterization. For the results presented here, we used distributed task queues with stealing.

The version of Argus that implements the parallel API extends the serial API architecture to allow multiple simultaneous graphics streams. Each application thread is augmented by a local command queue and a synchronization thread. Instead of entering graphics commands onto the global command queue, each application thread fills its local command queue. The synchronization thread is then responsible for transferring commands from this local command queue onto the global command queue. Since the global command queue may be written in parallel, the architecture is scalable. The box on the adjacent page describes the pipeline in greater detail and explains how state management and synchronization commands are implemented within Argus.

6.2 Performance

Because poor performance often hides architectural bottlenecks, Argus was designed with performance as one of its main criteria. Although Argus can run on many architectures, particular care was taken to optimize the library for the Silicon Graphics Origin system [15]. The Origin is composed of 195 MHz R10000 processors interconnected in a scalable NUMA architecture. Depending on the rendering parameters, the single processor version of Argus is able to render up to 200K triangles per second; this rendering rate scales up to 24 processors. In its original incarnation, Argus was designed for a serial interface and many serial applications were not able to keep up with the scalable performance of the graphics system. Remedying this situation led us to the development of the parallel API.

To study the performance of our parallel API implementation, we ran two applications: *Nurbs* and *March*. *Nurbs* is an immediate-mode patch tessellator parallelized by distributing the individual patches of a scene across processors in a round-robin manner. By tessellating patches on every frame, the application may vary the resolution of the patches interactively, and because depth buffering is enabled, no ordering constraints are imposed in the drawing of the patches—synchronization commands are utilized only on frame boundaries. Our second application, *March*, is a parallel implementation of the marching cubes algorithm [16]. By extracting the isosurface on every frame, the application may choose the desired isosurfaces interactively. Rendering is performed in back-to-front order to allow transparency effects by issuing graphics semaphores which enforce the dependencies described in Section 5.2. One noteworthy difference between our implementation and the one outlined in Section 5.2 is that cells are distributed from a centralized task queue rather than in round-robin order because the amount of work in each cell can be highly unbalanced. The input characteristics and parameter settings used with each of these applications are shown below:

Nurbs	**March**
armadillo dataset	skull dataset
102 patches	256K voxels (64x64x64)
196 control points per patch	cell size at 16x16x16
117504 stripped triangles	53346 independent triangles
1200x1000 pixels	1200x1000 pixels

Figure 4a and Figure 4b show the processor speedup curves for *Nurbs* and *March*, respectively. The various lines in the graph represent different numbers of application threads. The serial application bottleneck can be seen in each case by the flattening of the "1 Context" curve: as more processors are utilized, no more performance is gained. Whereas the uniprocessor version of *Nurbs* attains 1.65 Hz, and the serial API version is limited to 8.8 Hz, the parallel API version is able to achieve 32.2 Hz by using four contexts. Similarly, the uniprocessor version of *March* gets 0.90 Hz, and the serial API version of *March* is limited to 6.3 Hz, but the parallel API version is able to attain 17.8 Hz by utilizing three contexts. These speedups show high processor utilization and highlight the implementation's ability to handle extra contexts gracefully.

One extension to Argus that we have been considering is the use of commodity hardware for tile rasterization. Although this introduces many difficulties, it also increases rasterization rate significantly. In order to simulate the effects of faster rasterization on the viability of the parallel API, we stress the system by running Argus in a simulation mode which imitates infinite pixel fill rate. In this mode, the slope calculations for triangle setup do occur, as does the movement of the triangle data between the geometry processors and the tile rasterizers. Only the rasterization itself is skipped. The resulting system increases the throughput of Argus and stresses the parallel API: Figure 4c and Figure 4d show how a greater number of contexts are required to keep up with the faster rendering rate. The parallel API allows Argus to achieve peak frame rates of 50.5 Hz in *Nurbs* and 40.9 Hz in *March*. This corresponds to 5.9 million stripped triangles per second in *Nurbs* and 2.2 million independent triangles per second in *March*. These rates are approximately double the rate at which a single application thread can issue primitives into Argus even

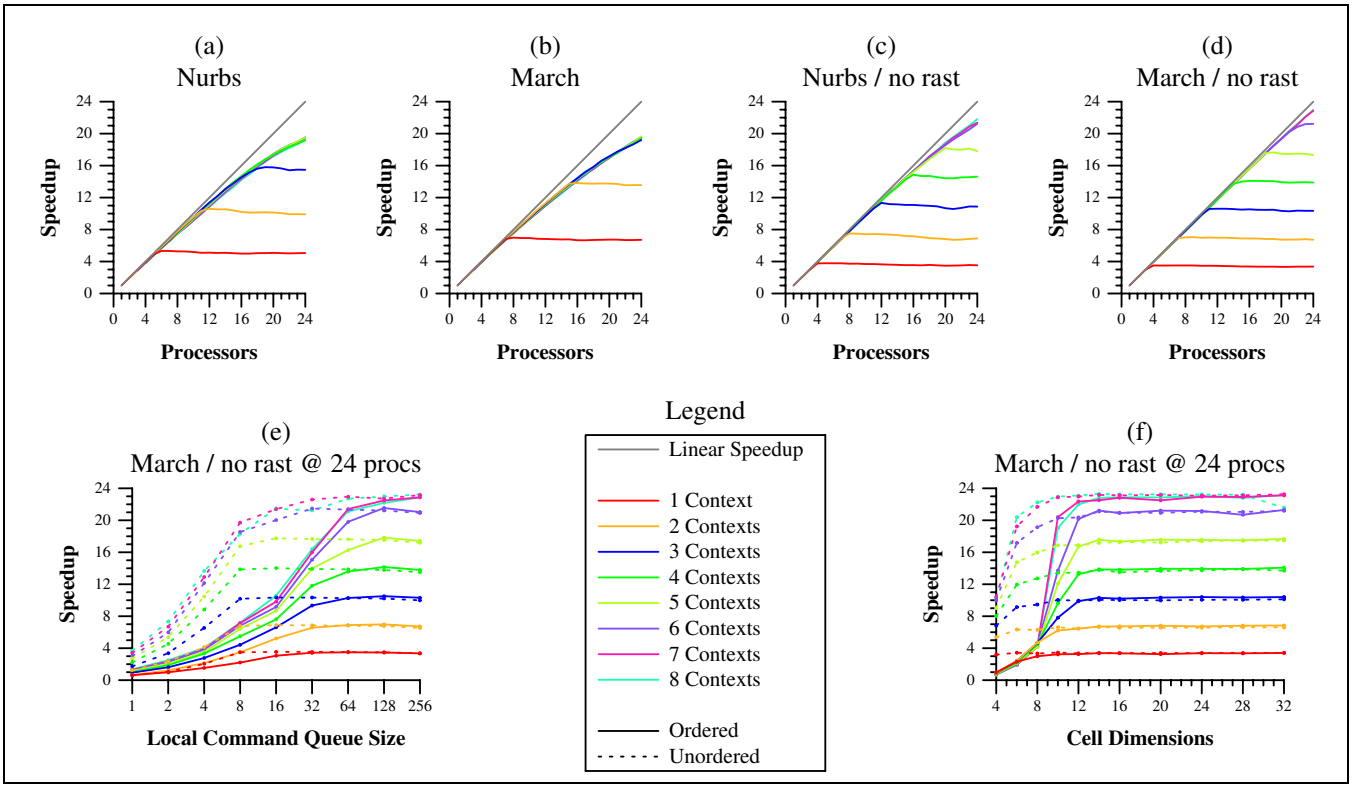


Figure 4: Performance Graphs. The speedup curves for two applications, *Nurbs* and *March*, are drawn in (a) and (b) for a varying numbers of contexts using the Argus graphics library. These same graphs are plotted for a version of Argus which assumes infinite fill rate in (c) and (d). The effects of buffering are illustrated in (e), and the effects of synchronization granularity are demon strated in (f).

when *no* application computation is involved, thus demonstrating the importance of multiple input ports. Again, the graphs illustrate Argus's capability of handling extra contexts without performance penalties.

One important aspect of any implementation of the parallel API is the amount of buffering required to make the API work. Without enough buffering, the parallel API serializes: in Argus, if a local command queue fills up before its synchronization commands are resolved, the application thread is forced to wait. Intuitively, we expect the amount of buffering required to be sensitive to the amount of synchronization between different threads. This is quantified in the speedup curves of Figure 4e for 24 processors. The number of entries in the local command queue (which can each point to a 16 KB block of primitive commands or hold a single synchronization command) was varied from 1 to 256. The runs were performed on the *March* application with the semaphores both enabled (the solid "Ordered" lines) and disabled (the dotted "Unordered" lines). As one would expect, the ordered version requires significantly larger buffers.

Another key aspect of any parallel API implementation is its ability to minimize the cost of synchronization. If the granularity of the application is too fine, synchronization costs can dominate, and the application is forced to use a coarser subdivision of work. If the work is subdivided too coarsely, load imbalance can occur within the application. The effects of granularity on Argus were tested by varying the dimensions of the cells on both the ordered and unordered versions of *March*. The number of processors was held at 24 and timings were taken with varying numbers of contexts, as illustrated in Figure 4f. A granularity which is too fine deteriorates performance in both the application (as demonstrated by the unordered runs) as well as in the graphics system (as demonstrated by the extra performance hit taken by the ordered runs). For the *March* application, there is a wide range of granularities (well over an order of magnitude in the number of voxels) that work well since Argus was designed to keep the cost of synchronization low. When *March* is run without isosurface extraction and rendering (i.e., nothing but the synchronization primitives are issued), several hundred thousand semaphore operations are resolved per second.

7 DISCUSSION

Argus is one implementation of the parallel API which performs well. Obviously, the architecture embodied by Argus is not the only possible choice, and it is instructive to examine the design considerations of alternative implementations due to the special architectural requirements imposed by the extensions.

7.1 Consistency and Synchronization

Until now, we have not said much about how the operations of the parallel API can be interleaved. Supporting multiple contexts that share a framebuffer means that the system must provide a consistency model. We borrow the notion of sequential consistency from the field of computer architecture [14]. Imagine a system consisting of multiple processes simultaneously performing atomic operations. A sequentially consistent system computes a result that is realizable by some serial interleaving of these atomic operations. By making a single API command be the level of apparent atomicity, we define the notion of *command-sequential consistency*, the strongest form of consistency possible

within the parallel API. At the other end of the spectrum is framebuffer-sequential consistency—only framebuffer accesses are atomic. A whole spectrum of consistency models can be enumerated in such a fashion. The OpenGL specification does not require an implementation to support any consistency model. In order to support the parallel API, however, a graphics system should provide at least *fragment-sequential consistency* in order to support features which depend on an atomic read-modify-write operation on the framebuffer (such as depth buffering).

The consistency model which an architecture supports is related to the location in the pipeline where synchronization constraints between graphics streams are resolved. The Argus pipeline described in Section 6.1 synchronizes and merges multiple graphics streams early in the pipeline, thus supporting command-sequential consistency. One problem with such an architecture is that geometry processing cannot occur on primitives which are blocked due to synchronization constraints. Another problem is that ordering dependencies not required by the synchronization commands are introduced early in the pipeline at the global command queue.

An alternate architecture addresses these problems by merging graphics streams at the rasterizers, thus supporting fragment-sequential consistency. We implemented such an alternate version of Argus in which the entire pipeline up to and including the tile rasterization threads is replicated for each context. Every tile thread executes every synchronization command, and threads which share the same tile merge their streams by obtaining exclusive access to the tile. One disadvantage of this approach is the extra buffering requirements due to the fact that the size of the graphics data expands as it gets farther down the pipeline. Another problem with this alternate approach is the high cost of synchronization since synchronization commands must be executed by every tile rasterizer—this proved prohibitively expensive in the framework of Argus. Of course, many architectures other than the two we tried are possible, and an architect should evaluate the effects of a proposed architecture on these same issues.

7.2 Architectural Requirements

While a graphics system which implements the parallel API is in many respects similar to one which implements a serial API, an architecture should take special care in addressing three particular areas. First, the architecture must have a mechanism that efficiently handles multiple simultaneous input streams. Second, the state management capabilities of the architecture must be able to handle multiple simultaneous graphics states. And third, the rasterization system must be able to handle texture data for multiple streams efficiently.

In designing current systems, graphics architects have gone to great lengths to allow the seamless sharing of the graphics hardware between multiple windows by significantly reducing the context switch time. Although this same mechanism can be used for the parallel API, the context switch time must be reduced even further in order to handle multiple input streams at a much finer granularity. Argus does this by making use of a thread library which can switch threads in less than a microsecond as well as allowing multiple input ports. A hardware system could allow multiple input ports by replicating command processors. Ideally, each of the command processors could handle either a single graphics stream at a high rate or multiple graphics streams at lower rates. This would result in peak performance on serial applications and good performance on highly parallel applications.

The parallel API imposes special requirements on the handling of state. In past architectures, state changes have been expensive due to pipeline flushing. Recent graphics architectures, however, have taken measures to allow large numbers of state changes [19]. To a first order, the number of state changes for a given scene as issued by one application thread is the same as the number of state changes for the same scene as issued by multiple application threads since the number of inherent state changes in a scene is constant. However, the parallel API increases the amount of state that has to be accessible to the different portions of the graphics system: the various graphics processors must be able to switch between the states of different graphics streams without dramatically affecting performance. Hardware implementations which allow for multiple simultaneous contexts have already been demonstrated [13, 26]. In Argus, multiple simultaneous contexts are handled efficiently by taking advantage of state coherence in the state management algorithm through the use of shared memory and processor caching.

One type of state which requires special attention is texture. Unlike the rest of the state associated with a context (with the exception of display lists), texture state can be shared amongst multiple contexts, thus exposing the need for efficient download of and access to shared texture data. The semantics of texture download are the same as all other graphics commands: it is susceptible to buffering, and synchronization must occur to guarantee its effects from the point of view of other contexts. Efficient implementations of synchronized texture download can be realized by extending the idea of the "texture download barrier" found in the SGI InfiniteReality [19]. The access of texture memory may also require special care. Since hardware systems have a limited amount of local texture memory, applications issue primitives in an order which exploits texture locality. The parallel API can reduce this locality since the rasterizers can interleave the rendering of several command streams. In architectures which use implicit caching [4, 10], the effectiveness of the cache can possibly be reduced. In architectures which utilize local texture memory as an explicit cache, texture management is complicated. In Argus, shared texture download is facilitated by shared memory, and locality of texture access is provided by the caching hardware.

8 FUTURE WORK

The parallel API provides a new paradigm for writing parallel graphics applications. Many graphics algorithms exist that need an immediate-mode interface but are limited by application computation speed (e.g., [11, 24]), and parallelizing them can help greatly. There are two other uses of the parallel API which are of special interest. Scene graph libraries such as Performer [22] are parallel applications which traverse, cull, and issue scenes on multiple processors. Pipeline parallelism is used to distribute different tasks among different processors, but Performer is limited on most applications by the single processor responsible for the issuing of the graphics commands. The parallel API can be used to write such libraries in a homogeneous, scalable fashion. A second novel use of the parallel API is to write a "compiler" that can automatically parallelize the graphics calls of a serial graphics application. Recent advances in compiler technology allow automatic parallelization of regular serial applications [1], and extending this work to encompass graphics applications would be an interesting research direction.

Another significant step in validating the parallel API is implementing an architecture with hardware acceleration. While

Argus is an excellent software system for studying the issues in the design of the parallel API, its performance is limited by poor rasterization speed, especially when texturing is enabled. One possible architecture consists of implementing the parallel API on a cluster of interconnected PCs with rasterization hardware. Another possibility is to extend the basic sort-middle interleaved architecture [17] of a high-end system such as the SGI InfiniteReality [19]. Though this task is by no means easy, we believe that such a system is feasible with the techniques described in Section 7.2. The parallel API can also be implemented by a variety of other, more exotic architectures.

Image composition architectures such as PixelFlow [8, 18] are one class of rendering architectures that have not been addressed by the parallel API. Because these machines are not designed to do ordered drawing of primitives, the parallel API needs to be extended to allow a relaxed ordering model in which the requirement of drawing in strict order can be enabled and disabled.

9 CONCLUSION

We have designed a parallel immediate-mode graphics interface. By introducing synchronization commands into the API, ordering between multiple graphics streams can be explicitly constrained. Since synchronization is done between graphics streams, an application thread is able to continue issuing graphics commands even when its graphics stream is blocked. The API provides a natural paradigm for parallel graphics applications that can be used in conjunction with the existing retained-mode constructs of an immediate-mode interface. The feasibility of this API has been demonstrated by a sample implementation which provides scalable performance on a 24 processor system.

Acknowledgements

We would like to thank Matthew Eldridge, Kekoa Proudfoot, Milton Chen, John Owens, and the rest of the Stanford Graphics Lab for their insights about this work. We thank Kurt Akeley and the anonymous reviewers for their helpful comments in revising the paper. We thank Dale Kirkland for describing the parallel interface used by Intergraph. For support, we thank Silicon Graphics, Intel, and DARPA contract DABT63-95-C-0085-P00006. For machine time, we thank Chris Johnson and the University of Utah. And finally, we thank our loved ones.

References

[1] S. Amarasinghe, J. Anderson, C. Wilson, S. Liao, B. Murphy, R. French, M. Lam, and M. Hall. Multiprocessors from a Software Perspective. *IEEE Micro*, 16:3, pages 52-61, 1996.

[2] A. Beers, M. Agrawala, and N. Chaddha. Rendering from Compressed Textures. *Computer Graphics* (SIGGRAPH 96 Proceedings), volume 30, pages 373-378, 1996.

[3] J. Blinn. Me and My (Fake) Shadow. *IEEE Computer Graphics and Applications*, 8:1, pages 82-86, 1988.

[4] M. Cox, N. Bhandari, and M. Shantz. Multi-Level Texture Caching for 3D Graphics Hardware. *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[5] T. Crockett. Design Considerations for Parallel Graphics Libraries. *Proceedings of the Intel Supercomputer Users Group 1994*, 1994.

[6] M. Deering. Geometry Compression. *Computer Graphics* (SIGGRAPH 95 Proceedings), volume 29, pages 13-20, 1995.

[7] E. Dijkstra. Cooperating Sequential Processes. *Programming Languages*, pages 43-112, 1968.

[8] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The Realization. *Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57-68, 1997.

[9] J. Gettys and P. Karlton. The X Window System, Version 11. *Software—Practice and Experience*, 20:S2, pages 35-67, 1990.

[10] Z. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[11] H. Hoppe. View-Dependent Refinement of Progressive Meshes. *Computer Graphics* (SIGGRAPH 97 Proceedings), volume 31, pages 189-198, 1997.

[12] M. Kilgard. *OpenGL Programming for the X Window System,* Addison-Wesley, 1996.

[13] D. Kirkland. Personal Communication. Intergraph Corp., 1998.

[14] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28:9, pages 241-248, 1979.

[15] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Proceedings of the 24th Annual Symposium on Computer Architecture*, 1997.

[16] W. Lorensen and H. Cline. Marching Cubes: A High-Resolution 3D Surface Reconstruction Algorithm. *Computer Graphics* (SIGGRAPH 87 Proceedings), volume 21, pages 163-169, 1987.

[17] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14:4, pages 23-32, 1994.

[18] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics* (SIGGRAPH 92 Proceedings), volume 26, pages 231-240, 1992.

[19] J. Montrym, D. Baum, D. Dignam, and C. Migdal. InfiniteReality: A Real-Time Graphics System. *Computer Graphics* (SIGGRAPH 97 Proceedings), volume 31, pages 293-302, 1997.

[20] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[21] T. Porter and T. Duff. Compositing Digital Images. *Computer Graphics* (SIGGRAPH 84 Proceedings), volume 18, pages 253-259, 1984.

[22] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics* (SIGGRAPH 94 Proceedings), volume 28, pages 381-395, 1994.

[23] R. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5:2, pages 79-109, 1986.

[24] T. Sederberg and S. Parry. Free-Form Deformation of Solid Geometric Models. *Computer Graphics* (SIGGRAPH 86 Proceedings), volume 20, pages 151-160, 1986.

[25] J. Torborg and J. Kajiya. Talisman: Commodity Real-Time 3D Graphics for the PC. *Computer Graphics* (SIGGRAPH 96 Proceedings), volume 30, pages 57-68, 1996.

[26] D. Voorhies, D. Kirk, and O. Lathrop. Virtual Graphics. *Computer Graphics* (SIGGRAPH 88 Proceedings), volume 22, pages 247-253, 1988.