

Interfaces and Software Systems

Dale Beermann
Oct. 1 2002

Papers Presented

- IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics (SIGGRAPH '94)
- The Design of a Parallel Graphics Interface (SIGGRAPH '98)
- Curved PN Triangles (ACM '01)

Typical Performance Limitations

- Graphics System
 - The graphics system cannot keep up with the commands it's being issued
- Data Generation
 - Commands may not be issued as fast as the graphics system can process them
- Graphics Interface
 - The limited bandwidth between the graphics system and the host could be the limiting factor

Typical Performance Solutions

- Packed Primitive Arrays
 - May not solve the bandwidth problem
 - Primitives still have to be decoded
 - Can introduce an awkward programming model
- Display Lists
 - Easy - basically a compiled macro of commands
 - Still could hold too much information
 - If data is recomputed every frame, display lists are not useful
- Compression
 - Can compress the geometry or the texture data
 - Increases encoding/decoding costs

Newer Performance Solutions

- Parallel Interfaces
 - IRIS Performer
 - Use rigorous Database requirements and State Management to allow for multiprocessing
 - A Parallel Graphics Interface
 - Adds synchronization commands to an immediate-mode interface.
- Geometry Compression
 - Curved PN Triangles
 - Provide an encoding for higher order primitives so less data needs to be transferred

Three Issues of a Parallel API #1

- State
 - A stateless interface
 - Requires that every command must include all information to execute
 - Each command could hold redundant information
 - State changes are expensive
 - An interface with state
 - Don't need to re-specify data
 - A command's behavior is affected by previous commands

Three Issues of a Parallel API #2

- Immediate-mode vs. Retained-mode
 - Immediate-mode
 - Primitives can be sent off immediately
 - Flexible programming style, easy to use
 - Retained-mode
 - A full scene representation is created and then sent to the graphics hardware
 - Can improve performance, but can require lots of bandwidth
 - Hybrid
 - Many interfaces use both for better flexibility and performance benefits

Three Issues of a Parallel API #3

- Ordering Semantics
 - Rules that constrain the order of commands to be executed need to be maintained
 - Important for special behaviors
 - Ground-plane shadows
 - Transparency with alpha-compositing
 - Sometimes it doesn't matter
 - Z-Buffering can solve ordering issues for opaque primitives

IRIS Performer

- Rohlf and Helman, SIGGRAPH '94
- A toolkit for the best possible performance for graphics workstations
 - Uses Graphics Optimizations (e.g. LOD) and Multiprocessing
 - Targeted at workstations using immediate mode graphics and small-scale shared memory

IRIS Performer

The Problem

- Achieving Maximal Performance using Immediate Mode Rendering Libraries
 - Often the Graphics Hardware is idle while the CPU crunches data
 - It is hard to do multiprocessing and still maintain data synchronization
 - It is hard to do multiprocessing without duplication of primitive data

The Solution

IRIS Performer

- Simplification
 - Scene Graph, LOD
 - Allows for frustum culling and model selection
- State Management
 - pfState, pfDispList, pfGeoState
 - Removes redundant state switching
- Shared Memory and Data Synchronization
 - pfBuffer, pfUpdatable
 - Reduces the amount of memory required
- Multiprocessing
 - Combine everything and parallelize

The Libraries

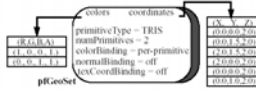
IRIS Performer

- libpr
 - Efficient Rendering
 - Provides high performance foundation
 - Manages geometry and graphics state
 - Also supports intersection and shared memory utilities
- libpf
 - Database Hierarchy and Automated Multiprocessing
 - Uses a hierarchical scene graph to organize geometry
 - Handles graph traversal configuration and control

libpr

pfGeoSet – Efficient Geometry Primitive

- Maintains a set of homogeneous primitives
 - Knowing structure lets Performer avoid if-tests like "should I send a color down the pipe with each vertex"?



- Construction is costly
 - Should have more than just a few primitives
 - pfuBuilder supplies convenient meshing and performance-oriented construction of pfGeoSets

libpr

pfState – Immediate Mode

- Generally used to set global state (e.g., fog)
- Provides state management to avoid costly redundant mode changes
 - Maintains all current and previous state in a state stack
- Partitions Graphics state into Modes and Attributes
 - Modes - state changes like backface culling or flat shading
 - Set by the application
 - Attributes – state changes like texture and material parameters
 - Applied by the application

libpr

pfDispList – Display List Mode

- Captures an entire frame's worth of data
- Made up of references to libpr objects
 - Does not contain individual vertex or primitive commands, these are in the libpr objects themselves

libpr

pfGeoState – Encapsulated Mode

- Used to specify the appearance of geometry (e.g. texture and material)
 - Encapsulates all modes and attributes managed by libpr
 - Globally inherits states when it doesn't need to set them
 - Only pushes what it needs changed (Lazy Push/Pop)
 - Eliminates useless mode changes
 - Guarantees Order Independence
 - pfGeoStates do not inherit from one another

libpr

Multiprocessing Support

- Shared Memory
 - References to memory allocations are counted to support operations like deleting
- pfMultiBuffer – Multibuffered Arrays
 - Queues and multibuffered memory are supported to pass data between processes
 - Manages multiple copies for multi-stage software pipelines
 - Global Index keeps track of which process is working on which buffer

libpr

Database Intersection

- Allows picking and collision detection
- Ability to intersect line segments with polygons of a pfGeoSet
 - Line segments are simple to test
 - Don't have to duplicate data, intersection just works on the current database

libpf

- Adds Database Hierarchy and Automated Multiprocessing to libpf
 - Manages a class hierarchy
 - Allows for scene graph traversal
 - Allows for Data Synchronization
 - Performs automated multiprocessing on defined pipes

libpf

- The Database – not just a scene graph
 - Nodes are broken into three groups: abstract, internal, and leaf
 - pfNode is the abstract class for all nodes
 - Derived from pfUpdatable – maintains multiple copies for multiprocessing
 - Internal node types are pfGroup, pfScene, pfSwitch, pfLOD, pfPartition, ...
 - Provide higher level functions
 - Leaf node types are pfGeode, pfBillboard, pfLightPoint, pfLightSource



libpf

- Node Hierarchy
 - Provides state inheritance (top-down)
 - Again, pfGeoState is not inherited, but pfState is
 - Primary type of inherited state is 3D transformations
 - Defines a hierarchy of bounding volumes
 - Used to accelerate intersection and culling
- Traversals
 - ISECT – process intersection requests
 - CULL – rejects objects outside viewing frustum, LOD
 - DRAW – sends info to graphics subsystem
 - Not exactly a traversal

libpf

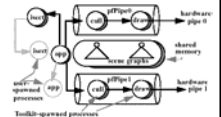
- Traversal Control
 - Each type has a mask
 - Allows traversal to exclude subgraphs
- Traversal Callbacks
 - pre- and post-traversal callbacks for each traversal type
 - Allow application control of custom culling (e.g. arbitrary termination) and custom rendering (e.g. Video textures)
- Efficiency
 - Dependent on depth and balance of the scene graph
 - pfPartition – creates a 2D grid of underlying data, line segment intersection determines pfGeoSets to draw

libpf

- Performance Optimizations
 - pfFlatten – Eliminating Transformations
 - Small primitives consume time for transformations
 - Applies static transform to static geometry
 - pfLOD – Level of Detail
 - Uses various parameters (eye-point distance, FOV, etc.) to select right model
 - pfSequence – Animation
 - Use precomputed animations (e.g. flames)
 - Each child has a period of time to be displayed
 - pfBillboard – Billboarded Geometry
 - Rotates geometry to face the eye

libpf

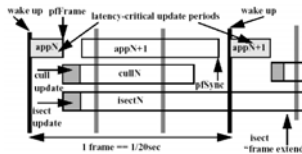
- Multiprocessing
 - pfPipe – provides access to multiple graphics subsystems
 - Consists of CULL and DRAW stages
 - The intersection pipeline is separate, has just ISECT
 - pfMultiprocess – allows access to multiple CPUs
 - Can combine or split rendering and intersection pipelines at stage boundaries



libpf

Process Synchronization

- Pipelined synchronization is enforced by Performer
 - However, user created processes must be synchronized by the application



libpf

Frame Control

- pfFrame() tells the rendering and intersection pipelines to start on a new frame
 - Pipelined process can frame extend
 - If the APP process frame extends, frame rate drops even if pipelines can keep up – APP must keep up
- pfSync() suspends the calling process until the next frame boundary

libpf

Data Synchronization

- Multibuffering
 - All libpf objects are pfUpdatables and contain a full scene graph
 - Partitioned into pBuffer
 - Each pBuffer is associated with one process
 - Modifications made to pfUpdatables are kept in a update list
 - APP is working on future frame
 - Changes are applied downstream when other processes get to the current frame
 - libpr primitives are *not* multibuffered
 - Avoid this large memory penalty

libpf

Achieving Real-Time

- pfVclockSync()
 - Called by pfSync()
 - Tells the APP to wait until it should start on the next frame
- Fixed Frame Rate
 - Need to maintain a fixed frame rate for visual coherence
 - Stress-feedback filtering helps to reduce LOD artificially
 - Always a frame late, can become oscillatory
- phase of the DRAW process
 - Locked says that DRAW can only start on a frame boundary
 - Frame-extending will halve frame rate
 - Floating says DRAW can start as soon as possible

Conclusions

- Database structure lends to many improvements
 - culling, multibuffering, delayed updates
- Maintaining Real-Time is difficult
 - Many things can jeopardize frame rates
 - Simplification makes this easier
- Multiprocessing is only possible once the issues are taken care of

The Design of a Parallel Graphics Interface

- Igehy, Stoll and Hanrahan, SIGGRAPH '98
- Adding synchronization commands to an immediate-mode interface.
 - Application threads can issue explicitly ordered primitives in parallel
 - Can synchronize contexts so every process uses the correct one.

Past Work: OpenGL and X

- Can use glFlush() and glFinish() in an attempt to preserve ordering
 - Flush (Xflush, glFlush) is insufficient
 - Only guarantees eventual execution
 - Finish (XSync, glFinish) is too much
 - Don't necessarily need to wait for everything to finish
- Solution: glXWaitGL and glXWaitX
- But no mechanism for maintaining parallel issue of commands

The Parallel API Extensions

- Provides extensions for multithreaded use of OpenGL
 - glpBarrier – provides synchronization for a set of streams in lock-step
 - glp[P/V]Sema – provides point-to-point synchronization
 - glpWaitContext – guarantees ordering between contexts

Simple Interactive Loop

<pre>Serial loop: glClear() get user input compute & draw glXSwapBuffers() glFinish()</pre>	<pre>Master loop: glClear() get user input appBarrier(appBarrierVar) compute & draw glpBarrier(glpBarrierVar) glXSwapBuffers() glFinish()</pre>	<pre>Slave loop: appBarrier(appBarrierVar) glpWaitContext(masterCtx) compute & draw glpBarrier(glpBarrierVar)</pre>
---	---	---

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

Marching Cubes

<pre>MarchCubesOrdered (M, N, grid) for (i=0; i<M; i++) for (j=0; j<N; j++) EmitOutAndRender (grid[i], j)</pre>	(a)
<pre>MarchParallel (M, N, grid) for (i=0; i<M; i++) for (j=(i*P)+1; j<M*P+1; j+=numProcs) EmitOutAndRender (grid[i], j)</pre>	(b)
<pre>MarchParallelOrdered (M, N, grid, sema) for (i=0; i<M; i++) for (j=(i*P)+1; j<M*P+1; j+=numProcs) if (i==0) glpPsema (sema[i], j) if (j==0) glpVsema (sema[i], j) EmitOutAndRender (grid[i], j) if (i==0) glpVsema (sema[i], j) if (j==0) glpPsema (sema[i], j)</pre>	(c)

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.

Argus

- Shared Memory Multiprocessor graphics library
- Implements a subset of OpenGL and the parallel API extensions
- Sort-Middle tiled parallel graphics system

Argus

- Three types of threads
 - Application Thread
 - Runs application code and manages graphics context
 - Geometry Thread
 - Transforms and shades primitives
 - Sends primitives to correct Rasterizer
 - Rasterization Thread
 - Draws transformed primitives into the framebuffer

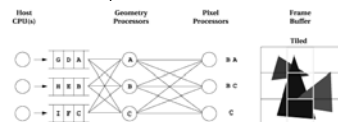


Figure 2. The Argus Pipeline

Performance of Argus

- Two tests, *Nurbs* and *March*
- Nurbs*
 - Patch Tessellator, no ordering needed
 - Synchronization done only on frame boundaries
- March*
 - Parallel implementation of marching cubes
 - Rendering done in back-to-front to allow transparency effects

<u>Nurbs</u>	<u>March</u>
Armadillo dataset	00011 dataset
102 patches	256K voxels (64x64x64)
19% control points per patch	cell size at 16x16x16
117604 stripped triangles	3336 independent triangles
1200x1000 pixels	1200x1000 pixels

Performance of Argus

	<i>Nurbs</i>	<i>March</i>
Uniprocessor	1.65 Hz	0.90 Hz
Serial API	8.8 Hz	6.3 Hz
Parallel API (4 contexts)	32.3 Hz	17.8 Hz
Parallel API (8 contexts)*	50.5 Hz	40.9 Hz

* Double the maximum rate at which a single application can issue primitives into Argus with no application computation

* Assuming infinite fill rate

Conclusions

- By introducing synchronization commands Argus supports command-sequential consistency
- The application can continue issuing graphics commands since synchronization is done between streams, not by the application
- The API is scalable on a 24 processor system

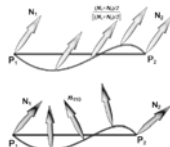
Curved PN Triangles

- Vlachos, Peters, Boyd and Mitchell, ACM Symposium on Interactive 3D Graphics '98
- Replace flat triangles with curved patches and higher order normal variation
 - Require minimal or no change to existing authoring tools and hardware designs
 - Inexpensive means of improving visual quality
 - Effectively sub-triangulates the original triangle into a programmable number of smaller, flat, triangles



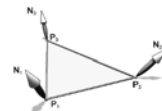
Geometry

- A PN Triangle is defined as one cubic Bezier patch
 - Matches point and normal information at the vertices
 - Why cubic?
 - Quadratics don't have strict inflection
- The normal is independently specified
 - Separate linear or quadratic Bezier interpolant of the normals
 - Why quadratic?
 - Linear interpolation ignores inflection



Geometry Coefficients

- Start with P_1, P_2, P_3 and N_1, N_2, N_3 as the only information we know

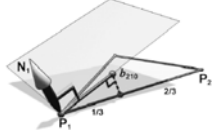


- b_{ijk} will represent the vertex coefficients for the new triangles
- n_{ijk} will represent the normal coefficients for the new triangles
- Calculating coefficients
 - Step 0
 - Spread out b_{ijk} evenly across the triangle
 - b_{jk} become $(iP_1 + jP_2 + kP_3)$

Geometry Coefficients

Calculating coefficients

- Step 1
 - Leave the vertex coefficients in place
 - $b_{200} = P_1$; $b_{100} = P_2$; $b_{000} = P_3$;
- Step 2
 - For each corner, project the closest b_{ijk} to the tangent plane defined by the normal vector
 - $w_i = (P_i - P_j) \cdot N$
 - $b_{210} = (2P_1 + P_2 - w_{i2}N_i)/3$;
 - $b_{120} = (2P_2 + P_1 - w_{i2}N_i)/3$;
 - ...



Geometry Coefficients

Calculating coefficients

- Step 3
 - Move the center from where we originally put it to the average of all six tangent points, and move it in the same direction $1/2$ the distance already traveled
 - $E = (b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201})/6$;
 - $V = (P_1 + P_2 + P_3)/3$;
 - $b_{111} = E + (E - V)/2$;

Normal Coefficients

Linear Normal Variation

- $N(u,v) = N/\text{magnitude}(N)$, $N = (1 - u - v)N_1 + uN_2 + vN_3$;
- Approximates phong shading
 - Shading stage will get triangles that look like $N(u,v)$, $N(u+h,v)$, $N(u,v+h)$ for some step h
- Ignore inflections in geometry (picture from before)

Quadratic Normal Variation

- Mid-edge coefficient is used for inflection
 - Average of the end normals is reflected across the plane perpendicular to the edge

Normal Coefficients

- The normals at the vertices will stay the same
- The mid-edge coefficient is used for intermediate normals

$$n_{200} = N_1,$$

$$n_{020} = N_2,$$

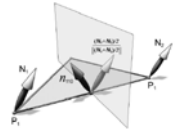
$$n_{002} = N_3,$$

$$v_{ij} = 2 \frac{(P_j - P_i) \cdot (N_i + N_j)}{(P_j - P_i) \cdot (P_j - P_i)} \in \mathbf{R}$$

$$n_{110} = h_{110}/\|h_{110}\|, \quad h_{110} = N_1 + N_2 - v_{12}(P_2 - P_1),$$

$$n_{011} = h_{011}/\|h_{011}\|, \quad h_{011} = N_2 + N_3 - v_{23}(P_3 - P_2),$$

$$n_{101} = h_{101}/\|h_{101}\|, \quad h_{101} = N_3 + N_1 - v_{31}(P_1 - P_3).$$



Curved Sharp Edges

- Connect two triangulations by identifying vertices on their boundary to form the crease
- With entirely local information, cracks can not be avoided
 - Have two normals per vertex on the boundary
- Can add a seam of small triangles to create sharper edges

Hardware Performance

- PN triangles are generated between the vertex-and-primitive-assembly stage and the vertex-shading stage
- Fill rate is not a bottleneck because the screen area in pixels is unchanged
- Vertex shading could be a problem
 - State that performance is usually limited by bandwidth needed to feed vertices to the graphics processor
 - If the graphics processor is sufficiently fast and the bus is busy, curved PN triangles render at the same speed as flat triangles

Hardware Performance

- Form of geometry compression
 - Reduces bus traffic
 - Acts as a level of detail mechanism
 - Higher LODs obtained on the fly
 - Models for LOD 0-4 would require 70 times the amount of memory

Conclusion

- Curved PN triangles provide a smoother silhouette and more organic shape
- Can be viewed as a triangle multiplier
- Sending a coarser triangulation can be seen as a form of geometry compression