Commodity Components 1

Greg Humphreys
October 10, 2002

Life Is But a (Graphics) Stream

“There’s a bug somewhere…”

You Spent How Much Money On What Exactly?

Several Years of Failed Experiments

1975-1980
1980-1982
1982-1986
1986-Present

The Problem

Scalable graphics solutions are rare and expensive

Commodity technology is getting faster

But it tends not to scale

Cluster graphics solutions have been inflexible

Big Models

Scans of Saint Matthew (386 MPolys) and the David (2 GPolys)
Stanford Digital Michelangelo Project
Modern Graphics Architecture

NVIDIA GeForce4 Ti 4600
Amazing technology:
- 4.8 Gpix/sec (antialiased)
- 136 Mtris/sec
- Programmable pipeline stages
Capabilities increasing at roughly 225% per year

But it doesn’t scale:
- Triangle rate is bus-limited - 136 Mtris/sec mostly unachievable
- Display resolution growing very slowly

Result: There is a serious gap between dataset complexity and processing power

Why Clusters?

Commodity parts
- Complete graphics pipeline on a single chip
- Extremely fast product cycle

Flexibility
- Configurable building blocks

Cost
- Driven by consumer demand
- Economies of scale

Availability
- Insufficient demand for "big iron" solutions
- Little or no ongoing innovation in graphics "supercomputers"

Stanford/DOE Visualization Cluster

32 nodes, each with graphics
Compaq SP750
- Dual 800 MHz PIII Xeon
- i840 logic
- 256 MB memory
- 18 GB disk
- 64-bit 66 MHz PCI
- AGP-4x

Graphics
- 16 NVIDIA Quadro2 Pro
- 16 NVIDIA GeForce 3

Network
- Myrinet (LANai 7 - 100 MB/sec)

Virginia Cluster?
Ideas

Technology for driving tiled displays
- Unmodified applications
- Efficient network usage

Scalable rendering rates on clusters
- 161 Mtri/sec at interactive rates to a display wall
- 1.6 Gvox/sec at interactive rates to a single display

Cluster graphics as stream processing
- Virtual graphics interface
- Flexible mechanism for non-invasive transformations

The Name Game

One idea, two systems:
WireGL
- Sort-first parallel rendering for tiled displays
- Released to the public in 2000

Chromium
- General stream processing framework
- Multiple parallel rendering architectures
- Open-source project started in June 2001
- Alpha release September 2001
- Beta release April 2002
- 1.0 release September 2002

General Approach

Replace system’s OpenGL driver
- Industry standard API
- Support existing unmodified applications

Manipulate streams of API commands
- Route commands over a network
- Track state!
- Render commands using graphics hardware

Allow parallel applications to issue OpenGL
- Constrain ordering between multiple streams

Cluster Graphics

- Raw scalability is easy (just add more pipelines)
- One of our goals is to expose that scalability to an application

Cluster Graphics

- Flexible number of clients, servers and displays
- Compute limited = more clients
- Graphics limited = more servers
- Interface/network limited = more of both
Output Scalability

Larger displays with unmodified applications
Other possibilities: broadcast, ring network

Protocol Design

1 byte overhead per function call

```
glColor3f( 1.0, 0.5, 0.5 );
glVertex3f( 1.0, 2.0, 3.0 );
glColor3f( 0.5, 1.0, 0.5 );
glVertex3f( 2.0, 3.0, 1.0 );
```

Efficient Remote Rendering

<table>
<thead>
<tr>
<th>Network</th>
<th>Mvert/sec</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>None</td>
<td>21.50</td>
</tr>
<tr>
<td>GLX</td>
<td>100 Mbit</td>
<td>0.73</td>
</tr>
</tbody>
</table>

Application draws 60,000 vertices/frame
Measurements using 800 MHz PIII + GeForce2
Efficiency assumes 12 bytes per triangle
Efficient Remote Rendering

<table>
<thead>
<tr>
<th>Network</th>
<th>Mvert/sec</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>None</td>
<td>21.50</td>
</tr>
<tr>
<td>GLX</td>
<td>100 Mbit</td>
<td>0.73</td>
</tr>
<tr>
<td>WireGL</td>
<td>100 Mbit</td>
<td>0.90</td>
</tr>
<tr>
<td>WireGL</td>
<td>Myrinet</td>
<td>9.18</td>
</tr>
</tbody>
</table>

Application draws 60,000 vertices/frame
Measurements using 800 MHz PIII + GeForce2
Efficiency assumes 12 bytes per triangle

Efficient Remote Rendering

<table>
<thead>
<tr>
<th>Network</th>
<th>Mvert/sec</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>None</td>
<td>21.50</td>
</tr>
<tr>
<td>GLX</td>
<td>100 Mbit</td>
<td>0.73</td>
</tr>
<tr>
<td>WireGL</td>
<td>100 Mbit</td>
<td>0.90</td>
</tr>
<tr>
<td>WireGL</td>
<td>Myrinet</td>
<td>9.18</td>
</tr>
<tr>
<td>WireGL</td>
<td>None</td>
<td>20.90</td>
</tr>
</tbody>
</table>

Application draws 60,000 vertices/frame
Measurements using 800 MHz PIII + GeForce2
Efficiency assumes 12 bytes per triangle

*None: discard packets, measuring pack rate

Sort-first Stream Specialization

Update bounding box per-vertex
Transform bounds to screen-space
Assign primitives to servers (with overlap)

Graphics State

OpenGL is a big state machine
State encapsulates control for geometric operations
- Lighting/shading parameters
- Texture maps and texture mapping parameters
- Boolean enables/disables
- Rendering modes

Example: `glColor3f( 1.0, 1.0, 1.0 )`
- Sets the current color to white
- Any subsequent primitives will appear white

Lazy State Update

Track entire OpenGL state
Precede a tile’s geometry with state deltas

How Does State Tracking Work?

Tracking state is a no-brainer, it’s the frequent context differences that complicate things

Need to quickly find the elements that are different

Represent state as a hierarchy of dirty bits

18 top-level categories: buffer, transformation, lighting, texture, stencil, etc.

Actually, use dirty bit-vectors. Each bit corresponds to a rendering server

Ian Buck, Greg Humphreys and Pat Hanrahan, Graphics Hardware Workshop 2000
Inside State Tracking

Client State
- Transformation
- Pixel
- Lighting
- Spot cutoff
  - Ambient color
  - Diffuse color
  - Spot cutoff
- Light 0
- Light 1

Server 2’s State
- Transformation
- Pixel
- Lighting
- Spot cutoff
  - Ambient color
  - Diffuse color
  - Spot cutoff
- Light 0
- Light 1

Transformation
Pixel
Lighting
Spot cutoff
Light 0
Light 1

Context Comparison

Client State
- Bit 2 set?
  - No, skip it
- Bit 2 set?
  - Yes, drill down

Server 2’s State
- Bit 2 set?
  - Yes, drill down
- Bit 2 set?
  - No, skip it

Inside State Tracking

Context Comparison

Client State
- Bit 2 set?
  - No, skip it
- Bit 2 set?
  - Yes, drill down

Server 2’s State
- Bit 2 set?
  - Yes, drill down
- Bit 2 set?
  - No, skip it

Output Scalability Results

Marching Cubes
- Point-to-point “broadcast” doesn’t scale at all
- However, it’s still the commercial solution [SGI Cluster]

Output Scalability Results

Quake III: Arena
- Larger polygons overlap more tiles

WireGL on the SIGGRAPH show floor

Input Scalability

App

Server

Display

Parallel geometry extraction
Parallel data submission
Ordering?
Parallel OpenGL API

Introduce new OpenGL commands:
- `glBarrierExec`
- `glSemaphoreP`
- `glSemaphoreV`

Express ordering constraints between multiple independent graphics contexts
Don’t block the application, just encode them like any other graphics command
Ordering is resolved by the rendering server

Homan Igehy, Gordon Stoll and Pat Hanrahan. SIGGRAPH 98

Parallel OpenGL Example

def Init:
    glBarrierInit(barrier, 2)
    glSemaphoreInit(sema, 0)
def Display:
    if my_client_id == 1:
        glClear(...)
        glBarrierExec(barrier)
        DrawOpaqueGeometry(my_client_id)
        glBarrierExec(barrier)
        if my_client_id == 1:
            glSemaphoreP(sema)
            DrawTransparentGeometry1()
        else:
            DrawTransparentGeometry2()
        glSemaphoreV(sema)
        glBarrierExec(barrier)
    if my_client_id == 1:
        SwapBuffers()

Serial OpenGL Example

def Display:
    glClear(...)
    DrawOpaqueGeometry()
    DrawTransparentGeometry()
    SwapBuffers()

Inside a Rendering Server

Optional in Chromium

Client 1

Client 2

Inside a Rendering Server

Client 1

Client 2

Optional in Chromium
Inside a Rendering Server

Client 1
Client 2

Transparent
Geom 1
Barrier
Swap

Inside a Rendering Server

Client 1
Client 2

Swap

Inside a Rendering Server

Client 1
Client 2

Swap

Input Scalability Results

Multiple clients, one server
Compute limited application

A Fully Parallel Configuration

App
Server
Display

App
Server
Display

App
Server
Display
Fully Parallel Results

1-1 rate: 472 KTri/sec
16-16 rate: 6.2 MTri/sec

Peak Sort-First Rendering Performance

Immediate Mode
- Unstructured triangle strips
- 200 triangles/strip
Data change every frame
Peak observed: 161,000,000 tris/second
Total scene size = 16,000,000 triangles
Total display size at 32 nodes: 2048x1024 (256x256 tiles)

WireGL Image Reassembly

App
App

Server
Server

Network
Network
Geometry
Imagery

Composite
Display

WireGL Image Reassembly

App
App

Server
Server

Network
Network
Geometry
Imagery

Composite
Display

Lightning-2 = Digital Video Switch from Intel Research
Route partial scanlines from n inputs to m outputs
Tile reassembly or depth compositing at full refresh rate
Lightning-2 and WireGL demonstrated at SIGGRAPH 2001

Image Reassembly in Hardware

Gordon Stoll et al., SIGGRAPH 2001
**Example: 16-way Tiling of One Monitor**

Framebuffer 1

Framebuffer 2

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

**WireGL Shortcomings**

- **Sort-first**
  - Can be difficult to load-balance
  - Screen-space parallelism limited
  - Heavily dependent on spatial locality

- **Resource utilization**
  - Geometry must move over network every frame
  - Server’s graphics hardware remains underutilized

**We need something more flexible**

---

**Stream Processing**

- **Streams:**
  - Ordered sequences of records
  - Potentially infinite

- **Transformations:**
  - Process only the head element
  - Finite local storage
  - Can be partitioned across processors to expose parallelism

---

**Why Stream Processing?**

- **Elegant mechanism for dealing with huge data**
  - Explicitly expose and exploit parallelism
  - Hide latency

- **State of the art in many fields:**
  - Databases [Terry92, Babu01]
  - Telephony [Cortes00]
  - Online Algorithms [Borodin98, O’Callaghan02]
  - Sensor Fusion [Madden01]
  - Media Processing [Halfhill00, Khailany01]
  - Computer Architecture [Rixner98]
  - Graphics Hardware [Owens00, NVIDIA, ATI]

---

**Cluster Graphics As Stream Processing**

- **Treat OpenGL calls as a stream of commands**

- **Form a DAG of stream transformation nodes**
  - Nodes are computers in a cluster
  - Edges are OpenGL API communication

- **Each node has a serialization stage and a transformation stage**

---

**Stream Serialization**

- **Convert multiple streams into a single stream**
- **Efficiently context-switch between streams**
- **Constrain ordering using Parallel OpenGL API**

- **Two kinds of serializers:**
  - **Network server:**
  - **Application:**
    - Unmodified serial application
    - Custom parallel application
Stream Transformation
Serialized stream is dispatched to “Stream Processing Units” (SPUs)

Each SPU is a shared library
- Exports a (partial) OpenGL interface

Each node loads a chain of SPUs at run time
SPUs are generic and interchangeable

Example: WireGL Revealed

SPU Inheritance
The Readback and Render SPUs are related
- Readback renders everything except SwapBuffers

Readback inherits from the Render SPU
- Override parent’s implementation of SwapBuffers
- All OpenGL calls considered “virtual”

Example: Readback’s SwapBuffers

Virtual Graphics
Separate interface from implementation
Underlying architecture can change without application’s knowledge

Example: Sort-Last
Application runs directly on graphics hardware
Same application can use sort-last or sort-first

Example: SPU Inheritance
The Readback and Render SPUs are related
- Readback renders everything except SwapBuffers

Readback inherits from the Render SPU
- Override parent’s implementation of SwapBuffers
- All OpenGL calls considered “virtual”

Example: Readback’s SwapBuffers

void RB_SwapBuffers(void)
{
    self.ReadPixels( 0, 0, w, h, ... );
    if (self.id == 0)
        child.Clear( GL_COLOR_BUFFER_BIT );
    child.BarrierExec( READBACK_BARRIER );
    child.RasterPos2i( tileX, tileY );
    child.DrawPixels( w, h, ... );
    child.BarrierExec( READBACK_BARRIER );
    if (self.id == 0)
        child.SwapBuffers( );
}

Optional

Example: Virtual Graphics
Separate interface from implementation
Underlying architecture can change without application’s knowledge

Example: Sort-Last
Application runs directly on graphics hardware
Same application can use sort-last or sort-first

Example: WireGL Revealed

Douglas Voorhies, David Kirk and Olin Lathrop, SIGGRAPH 88
Example: Sort-Last Binary Swap

Application
Readback BSwap Send
Application
Readback BSwap Send
Application
Readback BSwap Send
Application
Readback BSwap Send
Server
Render

Binary Swap Volume Rendering Results

Example: User Interface Reintegration

Serial applications can drive the T221 with their original user interface
Parallel applications can have a user interface

Example: User Interface Reintegration

CATIA Driving IBM’s T221 Display

Jet engine nacelle model courtesy Goodrich Aerostructures
X-Windows Integration SPU by Peter Kirchner and Jim Klosowski, IBM T.J. Watson
Chromium is the only practical way to drive the T221 with an existing application
Demonstrated at Supercomputing 2001

A Hidden-line Style SPU

A Hidden-line Style SPU
Is “HiddenLine” Really a SPU?

Technically, no!
Requires potentially unbounded resources
Alternate design:

Alternate design:

Future Directions

End-to-end visualization system for 4D data
• Data management and load balancing
• Volume compression
Remote/Ubiquitous Visualization
• Scalable graphics as a shared resource
• Transparent remote interaction with (parallel) apps
• Shift away from desktop-attached graphics
Taxonomy of non-invasive techniques
• Classify SPUs and algorithms
• Identify tradeoffs in design

Observations and Predictions

Manipulation of graphics streams is a powerful abstraction for cluster graphics
• Achieves both input and output scalability
Providing mechanisms instead of algorithms allows greater flexibility
• Data management algorithms can be built into a parallel application or embedded in a SPU
Flexible remote graphics will lead to a revolution in ubiquitous computing

Current Shortcomings

General display lists on tiled displays
• Display lists that affect the graphics state
Distributed texture management
• Each node must provide its own texture
• Potential \( N^2 \) texture explosion
• Virtualize distributed texture access?
Ease of creating parallel applications
• Input event management

Demo

Is “HiddenLine” Really a SPU?