Real-Time
Graphics Architecture

Kurt Akeley
Pat Hanrahan

http://www.graphics.stanford.edu/courses/cs448a-01-fall

Display and Framebuffer

Displays
- Key properties
- Bandwidth

Framebuffers
- Definitions and key properties
- Bandwidth
- Architecture

Required reading
- Frame-Buffer Display Architectures, Sproull, Annual Review of Computer Science, '86
## Terminology

**CRT**
- Cathode Ray Tube

**LCD**
- Liquid Crystal Display (flat panel)

**DLP**
- Digital Light Processing
- Texas Instruments technology
  - Clever adaptation of IC/photo lithography

## Raster vs. Calligraphic

**Raster (image order)**
- dominant choice

**Calligraphic (object order)**
- Earliest choice (Sketchpad)
- E&S terminals in the 70s and 80s
- Works with light pens
- Scene complexity affects frame rate
- Monitors are expensive
- Still required for FAA simulation
  - Increases absolute brightness of light points
Display Sequence Issues

Raster video signal takes a full frame to deliver
- Adds almost one frame of latency (worst-case)

Persistence
- Flying dot: CRT, scanning Laser
- Skewed full-frame: LCD panel, DLP?
- Field sequential: consumer DLP, head-mount CRT

Visual artifacts
- Tearing in tiled displays
- Color separation in field sequential displays
- Motion blur of moving objects?

Display Sequence Issues (Cont.)

Interlace (vs. progressive)
- Two interlaced fields per frame
- Makes no sense for MPEG compression
- Included in HDTV spec!

Visual artifacts
- Flicker if image is poorly filtered
- Image doubling if render rate <= frame rate
- Disappearing objects
Display Resolution History

Rate of increase is low (1.1 compound overall)
LCD display has peak foveal pixel density at 3-feet

<table>
<thead>
<tr>
<th>Date</th>
<th>Format and Technology</th>
<th>Bandwidth</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>1024 x 768 x 60Hz, CRT</td>
<td>0.14 GB</td>
<td></td>
</tr>
<tr>
<td>1988</td>
<td>1280 x 1024 x 72Hz, CRT</td>
<td>0.29 GB</td>
<td>1.1</td>
</tr>
<tr>
<td>1996</td>
<td>1920 x 1080 x 72Hz, HD CRT</td>
<td>0.60 GB</td>
<td>1.1</td>
</tr>
<tr>
<td>2001</td>
<td>3840 x 2400 x 56Hz, active LCD</td>
<td>1.55 GB</td>
<td>1.2</td>
</tr>
</tbody>
</table>

All figures are the author’s estimates!

IBM’s Bertha LCD Display

3840 x 2400 resolution, 22” diagonal 16:10 screen
Video Signal Generation

Implemented on GPU

Analog and digital streams
- Analog: complex waveform, critical timing
- Digital: emerging standards and capabilities

Typically supported:
- Gamma correction
- Different resolution displays

Optionally supported:
- Multiple signals / displays
- Genlock synchronization

Display Summary

RGB raster displays are prevalent
- Calligraphics as a pedagogical tool
- Ignore 3D displays

Video bandwidth
- Is a steady load on an operating GPU
- Is increasing slowly
Framebuffer Definitions

What is a framebuffer?
What can we learn by considering different definitions?

Framebuffer Definition #1

*Storage for commands that are executed to refresh the display*

Allows for raster or calligraphic display (e.g. Megatech)
“Framebuffer” for calligraphic display is a “display list”
  ■ OpenGL “render list”?

Key point: framebuffer contents are *interpreted*
  ■ Color mapping
  ■ Image scaling, warping
  ■ Window system (overlay, separate windows, …)
  ■ Address Recalculation Pipeline
Framebuffer Definition #2

*Image memory used to decouple the render frame rate from the display frame rate*

Meets common understanding of framebuffer as image

Leads naturally to *double buffering*
- One render buffer, one display buffer, swap
- *n*-buffering also possible, can control latency

Key idea: decoupling enables general-purpose GPU
- Visual simulation has high render frame rate
- MCAD has low render frame rate
- Window manager has no frame rate

Framebuffer Definition #3

*All pixel-assigned memory used to assemble and display the images being rendered*

Key point: framebuffer is active participant in rendering

Leads to non-color buffers: depth, stencil, window control
- OpenGL treats these buffers as part of framebuffer
- Some reserve "framebuffer" for color images
- Should be *n*-buffered in some cases (sort last)
- RealityEngine framebuffer can be deeper than wide or high

History cycles through this definition
- 2D manipulation
- 3D painters algorithm
- 3D depth, stencil, accumulation, multi-pass
- Programmable shading
**Framebuffer is Optional**

Calligraphic display
- If we don’t treat display list as framebuffer

“Follow-the-beam” rendering
- Minimizes latency
- Saves cost if frames are never “dropped”

Talisman-like image assembly (3D sprites)
- Old idea (visual simulation, window systems)

GigaPixel render tile
- Framebuffer stores color images only
- Depth, stencil, etc. in small tile

**Dominant Architecture is Consistent**

SGI architectures look like
ATI architectures look like
NVIDIA architectures
Details are evolving, but big picture remains the same

Why is this?
- Simplicity of design
- Simplicity of algorithms
- Simplicity of immediate-mode approach
Simplicity of Design

Framebuffer fragment operations
- Blending: merge fragment and pixel color
- Depth Buffering: save nearest fragment
- Stencil Buffering: simple pixel state machine
- Accumulation Buffering: high-resolution color arithmetic
- Antialiasing: (to be covered later)
- ....

Key points:
- All utilize pixel data (not just fragment data)
- All are pixel independent (no neighbor data dependencies)

Why aren’t fragment operations programmable?

Simplicity of Algorithms

Framebuffer employs brute-force simplicity
- Hidden surface elimination: Depth-buffer vs. sort/painter
- Capping: Stencil-based vs. object calculations
- Image-space algorithm is efficient
  - Just samples, never "object" information, locality
  - Just-in-time calculation, steady cost function

Accumulation Buffer (high-resolution color arithmetic)
- *The Accumulation Buffer*, Haeberli and Akeley, Proceedings of SIGGRAPH ’90
- Volume rendering using 3D textures

Multi-pass rendering
**Simplicity of Immediate-mode**

Framebuffer is “context”
Matches 2D/window rendering model

![Diagram](image)

*Little graphics state is in rendering hw*  
*Most graphics state is in framebuffer*

---

**Decreasing Display Bandwidth**

Historically display bandwidth was a limiting factor
- Hence “Sproull’s Rule”: fill rate $\geq$ display rate

Now display bandwidth is almost inconsequential

<table>
<thead>
<tr>
<th>Year</th>
<th>FB Bwth</th>
<th>Disp Bwth</th>
<th>Disp / FB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>0.3GB</td>
<td>0.14GB</td>
<td>1/2</td>
</tr>
<tr>
<td>1988</td>
<td>1.8GB</td>
<td>0.29GB</td>
<td>1/6 *</td>
</tr>
<tr>
<td>1996</td>
<td>12.8GB</td>
<td>0.60GB</td>
<td>1/20</td>
</tr>
<tr>
<td>2001</td>
<td>8.0GB</td>
<td>1.55GB</td>
<td>1/5(1/20)**</td>
</tr>
</tbody>
</table>

* VRAM provided separate video bandwidth  
** Display requires four separate video signals
Maximize Effective Bandwidth

Display bandwidth is inconsequential, but
Framebuffer bandwidth is still critical, so
- Optimize access locality
- Utilize special purpose memory parts
- Maximize real bandwidth
- Embed framebuffer memory
- Minimize bandwidth needs
- Utilize parallelism
- Pool framebuffer memory
Consider these in more detail ....

Optimize Access Locality

DRAMs run faster when "local" accesses are back-to-back
Imagine that you have a "locality budget"
Allocate it carefully to
- Optimize for display refresh cycles, and/or
  - Scan line locality
- Optimize for triangle fill cycles, and/or
  - Square "tile" of locality
- Optimize for overlay display cycles, and/or
  - Pixel component locality
- ....
**Utilize Special Purpose DRAM**

Video DRAM (VRAM) in '80s
- Popular for a short period. E.g. SGI GTX.

Sun 3DRAM in the '90s
- Constrains the architecture
  - Pixel format, fragment operations, etc.
- Expensive

Standard DRAMs have evolved for framebuffer use
- Time-to-fill limits utility of narrow-deep DRAMs
- Wide-shallow parts result (current 32-bit DDRRAM)
- Will DRAMs fall behind? Have they already?

---

**FBRAM**

FBRAM is DRAM with video output buffers (as in VRAM) and a cached ALU to perform fragment operations.

This was not a successful product.

*FBRAM: A New Form of Memory Optimized for 3D Graphics*, Deering, Schlapp, and Lavelle, SIGGRAPH ’94 Proceedings
Maximize Framebuffer Bandwidth

Use the fastest, widest DRAMs possible
Operate them at the highest possible clock rate
  ■ Separate “pixel” clock and “memory” clock
  ■ Bin memory (and GPU) parts
  ■ Provide elasticity (FIFO) and synchronization
Make all wiring point-to-point
  ■ Optimize signal paths
  ■ Separate memory controller for each DRAM

GTX Block Diagram

Each of the 20 Image Engines was conceived as little more than a stand-alone memory controller with attached VRAM.

**Embed Framebuffer Memory**

Examples

- Pixel Planes (earlier versions)
- Play Station 2

May be the ultimate answer

- When framebuffer memory is inconsequential

But

- It’s expensive compared with commodity DRAM
- NVIDIA and ATI have done well without it

**Minimize Bandwidth Requirements**

*Add transistors to make better use of bandwidth*

Be frugal, make each memory cycle count

- Aggregate memory transactions
- Cache to get efficient use of memory bandwidth

Compress framebuffer data

- Utilize area redundancy

Optimize occlusion culling

- Backface, early depth test, hierarchical depth

Minimize need for multi-pass rendering

- Programmable shading
### SGI Historicals - FB Bandwidth

Bandwidth increases at 1.4, pixel fill rate at 2.2

<table>
<thead>
<tr>
<th>Year</th>
<th>Product</th>
<th>Zbuf rate</th>
<th>Yr rate</th>
<th>FB Bwth</th>
<th>Yr rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>Iris 2000</td>
<td>100K</td>
<td>-</td>
<td>0.3GB</td>
<td>-</td>
</tr>
<tr>
<td>1988</td>
<td>GTX</td>
<td>40M</td>
<td>4.5</td>
<td>1.8GB</td>
<td>1.6</td>
</tr>
<tr>
<td>1992</td>
<td>RealityEngine</td>
<td>380M</td>
<td>1.8</td>
<td>6.4GB</td>
<td>1.4</td>
</tr>
<tr>
<td>1996</td>
<td>InfiniteReality</td>
<td>1000M</td>
<td>1.3</td>
<td>12.8GB?</td>
<td>1.2</td>
</tr>
</tbody>
</table>

* DRAM*

** VRAM**

* Physically separate front and back color buffers

** Not counting shift output bandwidth

---

### NVIDIA Historicals - FB Bandwidth

Bandwidth increases at 1.5, pixel fill rate at 2.5

<table>
<thead>
<tr>
<th>Season</th>
<th>Product</th>
<th>Fill rate</th>
<th>Yr rate</th>
<th>FB bwth</th>
<th>Yr rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2H97</td>
<td>Riva 128</td>
<td>20M</td>
<td>-</td>
<td>1.6GB</td>
<td>-</td>
</tr>
<tr>
<td>1H98</td>
<td>Riva ZX</td>
<td>31M</td>
<td>2.4</td>
<td>1.6GB</td>
<td>1.0</td>
</tr>
<tr>
<td>2H98</td>
<td>Riva TNT</td>
<td>50M</td>
<td>2.6</td>
<td>2.0GB</td>
<td>1.6</td>
</tr>
<tr>
<td>1H99</td>
<td>TNT2</td>
<td>75M</td>
<td>2.3</td>
<td>2.9GB</td>
<td>2.1</td>
</tr>
<tr>
<td>2H99</td>
<td>GeForce</td>
<td>120M</td>
<td>2.6</td>
<td>4.0GB</td>
<td>1.9</td>
</tr>
<tr>
<td>1H00</td>
<td>GeForce2</td>
<td>200M</td>
<td>2.6</td>
<td>6.4GB</td>
<td>2.6</td>
</tr>
<tr>
<td>2H00</td>
<td>NV16</td>
<td>250M</td>
<td>1.6</td>
<td>8.0GB</td>
<td>1.3</td>
</tr>
<tr>
<td>1H01</td>
<td>NV20</td>
<td>500M</td>
<td>4.0</td>
<td>8.0GB</td>
<td>1.0</td>
</tr>
</tbody>
</table>

** DRAM**

** SDRAM**

<table>
<thead>
<tr>
<th>Year</th>
<th>Product</th>
<th>Fill rate</th>
<th>Yr rate</th>
<th>FB bwth</th>
<th>Yr rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Rent’s Rule

Rent’s rule:

\[ \text{Bandwidth} = K R^{0.7} \]

NV series exponent is 0.5 (against 0.46 expected)

NV20 does:
- Transaction aggregation
- Clever depth buffer fragment elimination
- Lossless data compression
- ...

Utilize Parallelism

Single-Instruction, Multiple-Data Parallelism (SIMD)
- Usually tiled rendering stamp (e.g. Stellar)
- Efficiency poor due to “pixel depth complexity”

Multiple-Instruction, Multiple-Data Parallelism (MIMD)
- Fragment operations are independent
- Individual memory controllers are more efficient
- SGI approach, merge them into Image Engines
  - Became massively parallel (hundreds of engines)
- NVIDIA approach also?
  - Parallelism limited to 4 or so, more pipelining
InfiniteReality Block Diagram

Fully-configured InfiniteReality system includes 320 Image Engines. Each combines a fragment processor with a memory controller.

Image Engines are packaged in groups of four.


Pool Framebuffer Memory

Single shared memory for all GPU needs
- Framebuffer, texture, "display list"
- Standard GPU solution (including SGI desktop)

Can share CPU memory too
- “System company” solution
- Lots of issues (latency, error correction, locality)
- SGI O²

Automatically balances bandwidth needs
Addresses time-to-fill issue nicely
Requires crossbar for multiple memory controllers
Other Issues

Coordinate system
- Pixel is a region, not a point sample
- Pixels have integer coordinates, but
- Screen/window coordinates are continuous

Error detection/correction
- No SGI framebuffer has this (even O^2)
- Do others?

Why not map framebuffer into CPU address space?
- Lots of reasons
- DrawPixels/ReadPixels is the right interface

Conclusion

_Elegant_ brute-force is working
- Complexity is localized
- Architecture remains unchanged

More transistors buy lower bandwidth needs
- CPU designers add cache memory
- GPU designers have lots of tools
Real-Time Graphics Architecture

Kurt Akeley
Pat Hanrahan

http://www.graphics.stanford.edu/courses/cs448a-01-fall