
SIGGRAPHASIA2008
NEW HORIZONS

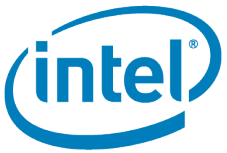
Next-Generation Graphics on Larrabee

Tim Foley
Intel Corp

SIGGRAPHASIA2008
NEW HORIZONS

Motivation

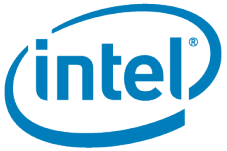
- “The killer app for GPGPU is graphics”
- We’ve seen
  - Abstract models for parallel programming
  - How those models map efficiently to Larrabee
- Now you use those tools to implement the future of graphics



Outline

- A software rendering pipeline
  - Current-generation graphics
  - Optimized for Larrabee
- Extensions to that pipeline
  - A litany of ideas to explore
  - Only a little time spent on each

Software Rendering



Binning/tiling/chunking

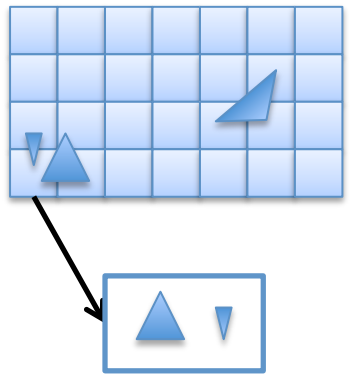
- Parallelize and load-balance pipeline
- Color/depth/stencil buffers stay in L2
  - Eliminates buffering between stages
  - Enables next-generation rendering approaches
- Saves bandwidth to main memory
  - Nice benefit, but the others matter more

Binning?!

- Not as crazy as some other tiled approaches
  - Fully compatible with existing applications
- Each pixel works in submit order
  - No sorting of fragments
  - Each fragment does shading, stencil, Z, blend
- Just works on pixels in a different order
  - Better parallelism, load balancing, locality
  - Graphics APIs don’t define rasterization order

How it works

- Divide render target into tiles of pixels
  - 64x64 or 128x128 are typical sizes
  - Each tile of pixels has a bin of geometry
- Pipeline front-end
  - Hit-test triangles and tiles
  - Add triangles to overlapping bins
- Pipeline back-end
  - Fragment shading, depth/stencil, blending
  - Tiles processed in parallel – about 80% of frame time



Front-end phase

- Give batch of geometry to any core
  - Input assembler
  - Vertex shader
  - Tesselation, geometry shader
  - Culling, clipping
  - Rasterization
- Place shaded tris + rasterized frags into bins

Back-end phase

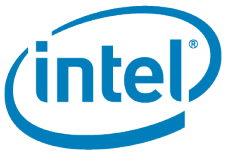
- For each bin
  - Read triangles, shaded vertices, coverage
  - Set up interpolants
  - Early depth/stencil
  - Attribute interpolation
  - Fragment shading
  - Late depth/stencil
  - Render target blending

Front-end / back-end split

- Not set in stone (it's all SW)
- Some parts can move between front and back
  - Vertex shading
  - Tesselation, clipping
  - Rasterization, setup
- Trade off computation and bandwidth
  - Different scenarios benefit from different split
  - Stencil shadows, shadow maps, post processing, particle systems, deferred rendering, …

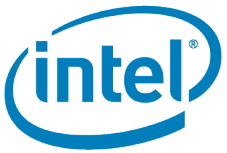
Specialization

- Phase code generated on-the-fly
  - Optimized for state settings
  - Only pay for the features you use
  - Compiler can shuffle order of operations at will
- Other GPUs also rely on “JIT” specialization
  - We just JIT our “fixed function” stages too
  - Compiles are done on Larrabee

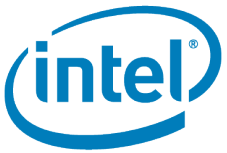


Rasterization

- Hierarchical recursive descent algorithm
  - Similar to Greene [1996]
- Conceptually:
  - Find Ax+By+C=0 edge equations in 2D screen space
  - Start at 64x64 pixel chunks
  - Dice by factor of 4x4 – matches SIMD width
  - Classify each block into full, empty, or partial
  - At leaf (4x4 pixels) test sample locations
- Sounds complex, but optimizes really well
  - In practice it all boils down to integer SIMD addition



Pipeline Extensions



“Standard” pipeline isn’t special

- Renderer is a C/C++ program
  - Intrinsics, assembly for performance-critical loops
- Uses the tools from my previous talk
  - Task system to distribute work to cores
  - Generate data-parallel code for front-/back-end
- Nothing up our sleeves
  - Extended pipelines are on equal footing

Some extensions

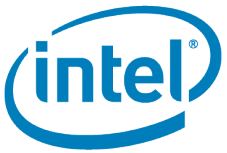
- Today will talk about extending
  - Render targets
  - Rasterization
  - Texturing
- Just scratching the surface
  - Not enough time to cover it all…

Extended render targets

- Back-end phase does
  - Fragment shading
  - Depth, stencil, blending
  - It's all one program
- Render target data is just sitting in L2
  - Core owns the whole tile
  - No need to synchronize
- Maybe we can do something with that…

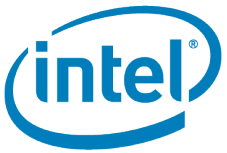
Programmable blending

- Arbitrary custom blend operations
  - Read the target during fragment shading…
  - Non-linear color spaces
  - “Blend” matrices, quaternions, normals
- Arbitrary number and type of targets
  - Just pointers to data in L2
  - Two-sided depth test (near and far)
  - More flexible deferred rendering
  - Custom antialiasing schemes



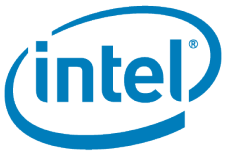
Irregular render targets

- Who says target has to be a uniform grid?
- Variable amount of data per-pixel
  - Per-pixel list of colors, depths, you-name-it
  - True A-buffer implementation
  - Order-independent transparency
  - Deferred rendering with transparent surfaces
  - Multi-layer shadow maps
- Sparse render targets
  - Irregular Z buffer



Extended rasterization

- Custom primitive types
  - Curved surfaces, B-splines, heightmaps, …
- Conservative rasterization
- Logarithmic, hemispherical rasterization
- Stochastic rasterization
  - Motion blur, depth of field
- Don’t have to give up rest of pipeline
  - No “graphics mode” and “compute mode”



Extended texturing

- Texture units use the memory hierarchy
  - Page tables, TLBs, …
  - Generates soft fault on missing page
- Simple answer: bring in tex data on fault
- Better answer: use data you have on hand
  - Re-submit request with lower mip level
  - Page data in the background
  - Keep framerate consistent

While you're at it…

- No need to store whole texture uncompressed
- Demand-decompress from compact format
  - JPEG and variants, PNG, SVG
  - RLE, zip, other lossless encoding
- Demand-generate from procedural description
  - "Texture shader" to generate data
- Demand-render from scene
  - Only render shadow/reflection texels you use

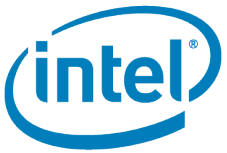
Keep the scene on Larrabee

- Required for demand-rendered textures
  - Also lets us accelerate more of the frame
- Animation
- Scene traversal, culling
  - Integrate with coarse occlusion render
- Render command generation
- Effect physics

Integrating with the pipeline

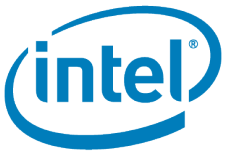
- Resources are just buffers in memory
  - Larrabee application can read and write
  - Bind application object as “constant buffer”
- Graphics pipeline uses a task system
  - C/C++ apps use tasks to share cores, synchronize
- Special “compute” mode not required
  - Process buffers with C/C++ code
  - Or use data-parallel kernel for automatic SIMD

Implementing the Future of Graphics



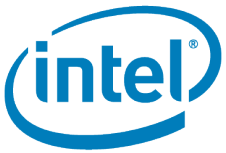
Software is the new Hardware

- Larrabee
  - x86 architecture, full C/C++ support
- Supports established scalable techniques
  - Data-parallel, task-parallel
- Software graphics pipelines
  - Extensible, scalable rendering architecture
  - The future of programmable graphics



Acknowledgements

- Tom Forsyth
- Aaron Lefohn
- Matt Pharr

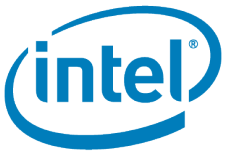


Parallel Computing for Graphics:
Beyond Programmable Shading

SIGGRAPHASIA2008
NEW HORIZONS

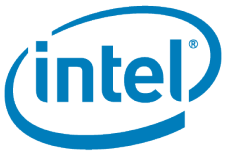
http://sa08.idav.ucdavis.edu/

Beyond Programmable Shading



Questions?

tim.foley@intel.com




SIGGRAPHASIA2008
NEW HORIZONS