MIT 6.189 IAP 2007
Student Project

Blue-Steel Ray Tracer

Natalia Chernenko
Michael D'Ambrosio
Scott Fisher
Russel Ryan
Brian Sweatt
Leevar Williams

Game Developers Conference
March 7 2007

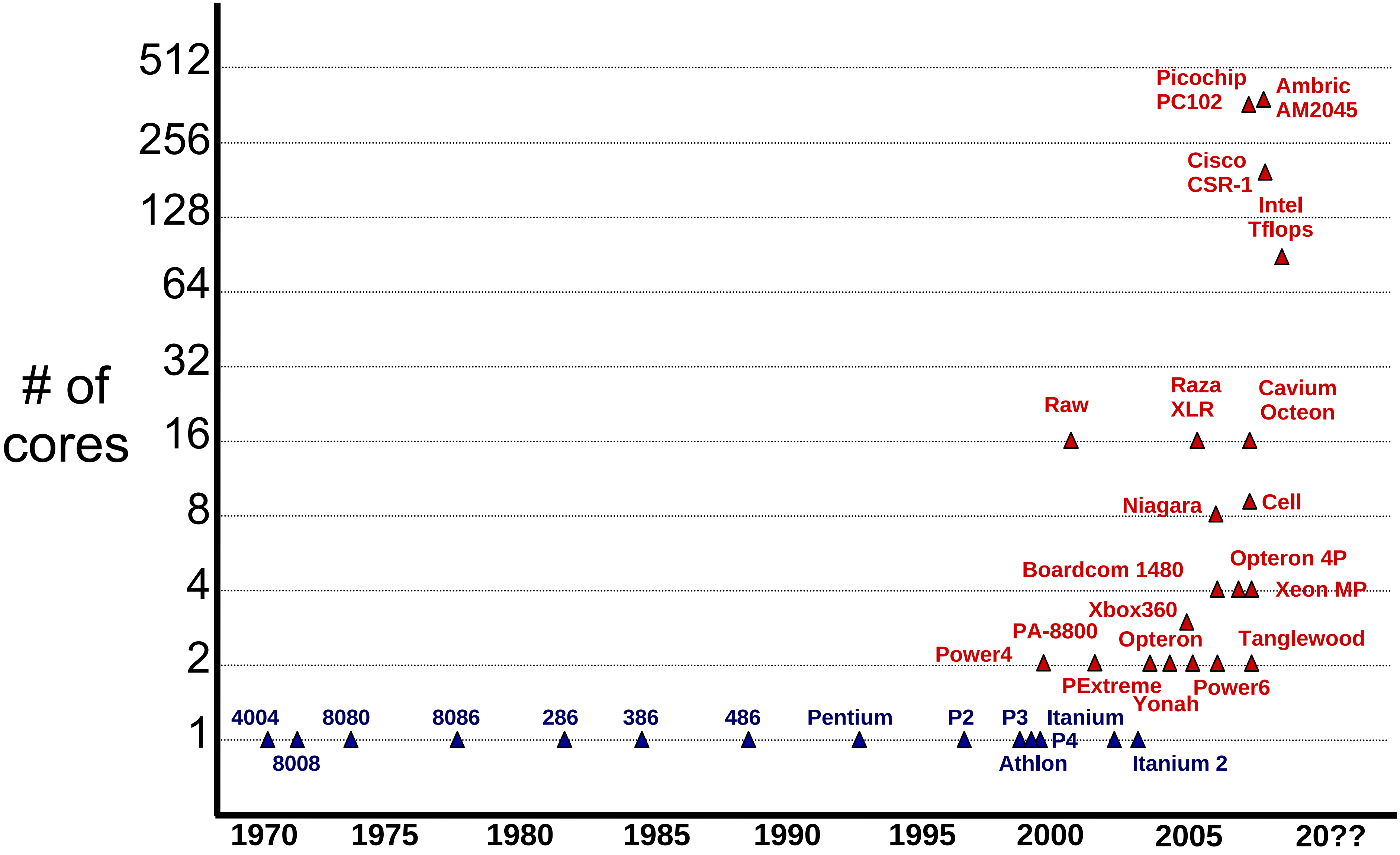
Imperative Need for Parallel Programming Education

The “Software Crisis”

“To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

-- E. Dijkstra, 1972 Turing Award Lecture

Multicores are Here



Teaching Parallel Programming

- Prof. Saman Amarasinghe (MIT) and Dr. Rodric Rabbah (IBM)
  - Month long intensive course
  - http://cag.csail.mit.edu/ps3 for lectures, recitations, and labs
  - Sponsored by Sony, Toshiba and IBM
  - Technical support from Sony, IBM, Terra Soft

- Course outcomes
  - Know fundamental concepts of parallel programming (both hardware and software)
  - Understand issues of parallel performance
  - Able to synthesize a fairly complex parallel program
  - Hands-on experience with the Cell processor
    - Sony PS3 consoles running YDL (Yellow Dog Linux)
    - IBM Cell SDK from developerWorks

Learning From Student Perspective

Fun and challenging context attracted many students

- Using PS3s as the platform for student projects
- Programming the new Cell processor

"*PS3 attracted me but hearing about the future of parallel programming kept me around.*" – student quote

Class Project Competition

- 7 ambitious projects
  - Ray Tracer
  - Global Illumination
  - Linear Algebra Pack
  - Molecular Dynamics Simulator
  - Speech Synthesizer
  - Soft Radio
  - Backgammon Tutor

©2006 Sony Computer Entertainment Inc. All rights reserved.
Design and specifications are subject to change without notice.

- Presentation, including performance results available online
  - http://cag.csail.mit.edu/ps3/competition.shtml
  - Some source code will also be published

Our Project: Ray-Tracer

Blue-Steel

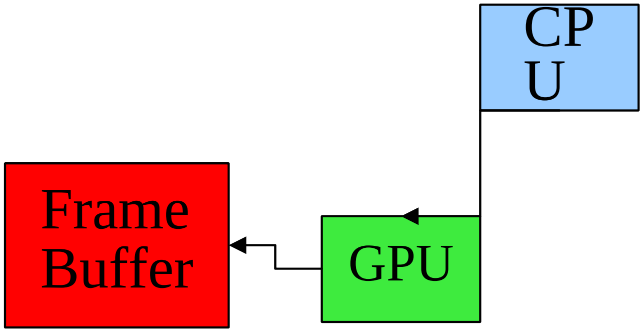
The Idea: Realistic Graphics

A Solution to the rendering equation

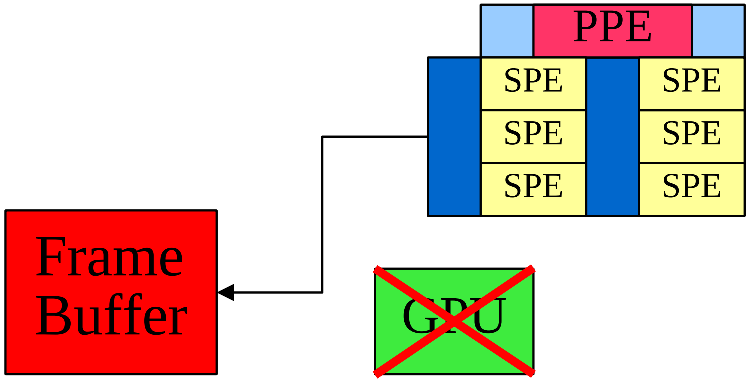
- Triangle Rasterization
  - Fast – possible in real time on a single core
  - Inaccurate or tedious for global effects such as shadows, reflection, refraction, or global illumination
  - “Start with speed, try to get realism”
- Ray Tracing
  - Slow – *unless done on multiple cores*
  - *Accurate and natural shadows, reflection, and refraction*
  - *“Start with realism, try to get speed”*

The Idea: Realistic Graphics

- Real time rasterization is done all the time!
  - Instead, build a fast ray tracer from the ground up to take advantage of multiple cores.
  - PS3 is perfect
    - 6 accessible cores for rendering
    - Fast XDR ram for transferring scene data / frames
    - Practically a GPU on its own – no need for additional hardware



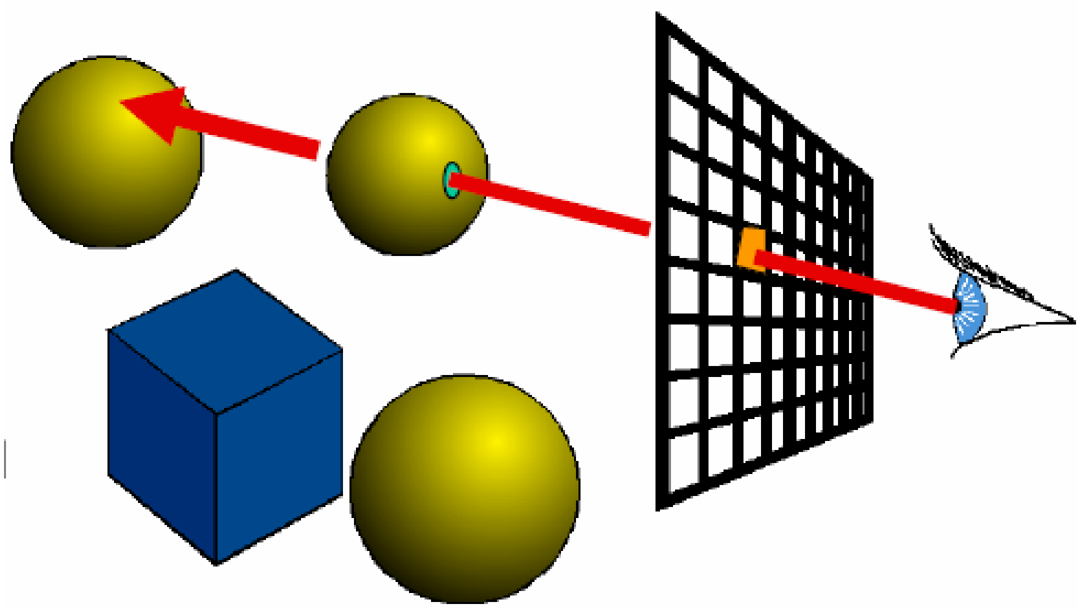
Modern graphics w/ GPU



Without GPU, using Blue-Steel

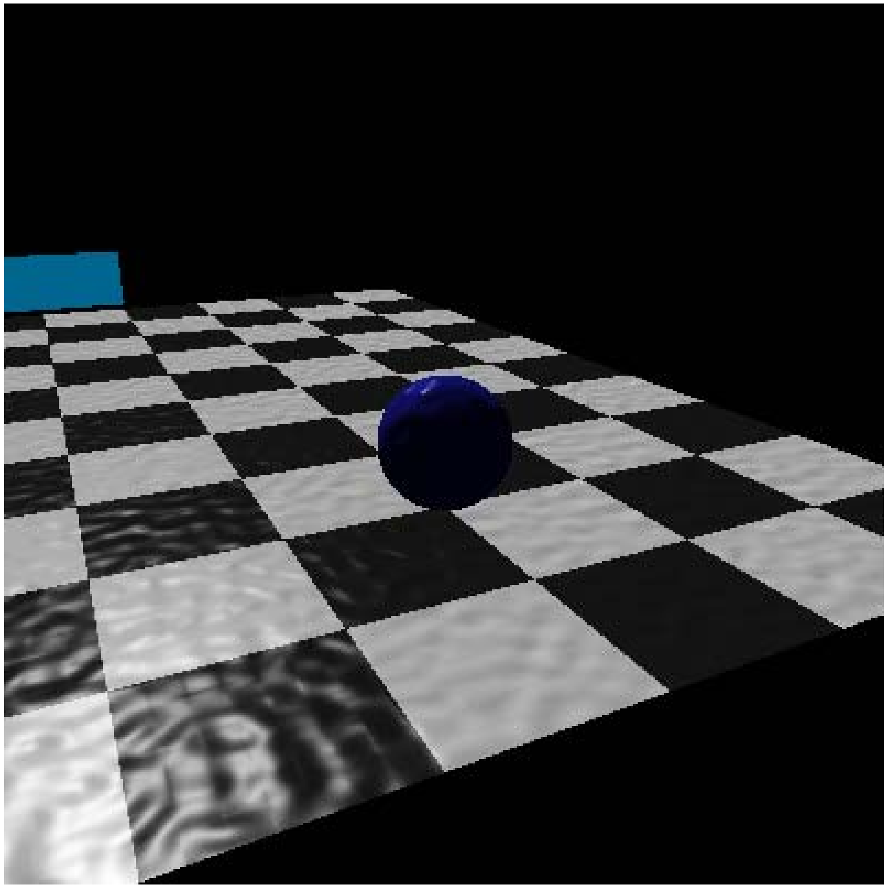
Ray Tracing

- Shoot a ray through each pixel on the screen
- Check for intersections with each object in the scene
- Keep the closest intersection



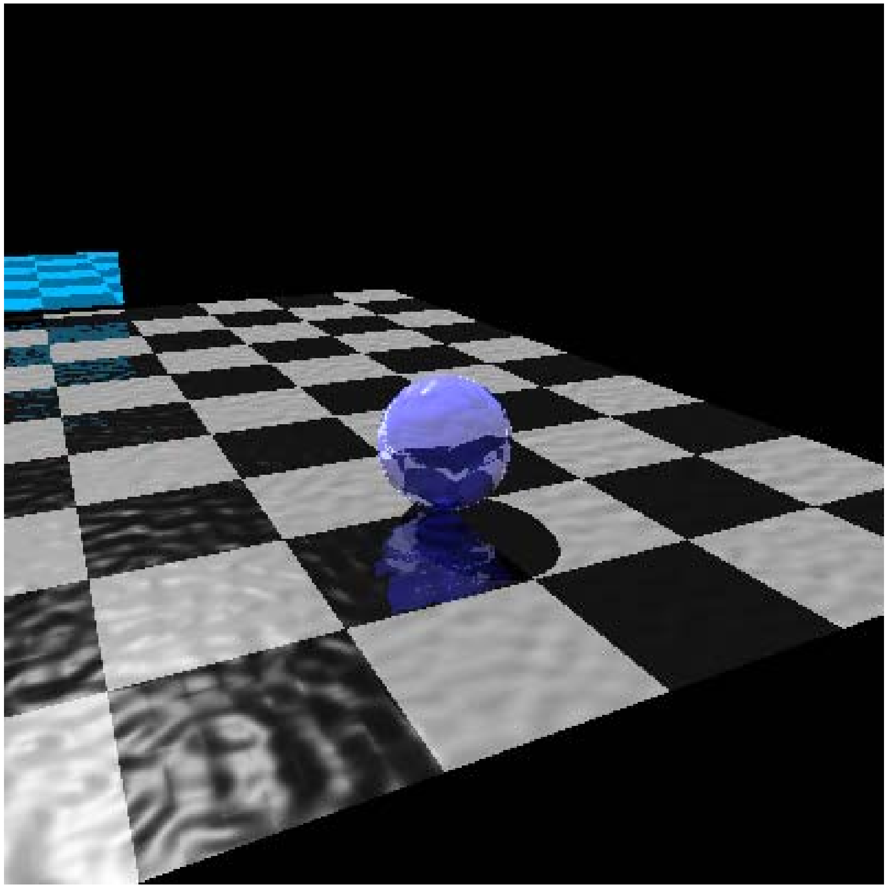
Ray Tracing

- Shade each point according to the material of the object, as well as the lights in the scene
  - Stopping at this level achieves traditional scan-line rasterization quality



Ray Tracing

- Cast rays for shadows, reflection, and refraction
  - Recursive rays are processed identically to primary rays
  - Framework for global effects is built into ray tracing by design



Ray Tracing on the PS3

- Design Challenges
  - Bandwidth & latency of PPE / SPE communication
    - Mailboxes can only hold 128 bits at a time
  - Limited size of local store
    - 256 KB for program, execution stack, scene, and frame data
  - DMA latency
    - Two orders of magnitude slower than local store

Ray Tracing on the PS3

- Design Challenges
  - Inherent SIMD architecture of SPE
    - Scalar code – like most code today – is expensive
  - No Branch Prediction
    - 'if' statements and loops are costly
  - Load-Balancing
    - Splitting up computation so as to minimize communication / computation overhead

Ray Tracing on the PS3

- High level design
  - Clump a set of SPEs together as one rendering engine
    - Each SPE holds a full set of scene data
    - Each SPE renders only part of the scene
    - Run a full ray tracer on every SPE
    - Engine has a set of instructions just like any processor
      - Instructions are sent to this engine using SPE mailboxes
  - SPE-centric framework
    - Each SPE has knowledge of what work it must do, PPE tells it what to render only at the start of the process

Ray Tracing on the PS3

- Tackling the Challenges
  - Bandwidth & latency of PPE / SPE communication
    - SPE-centric framework
      - No need for communication during the rendering process
  - Limited size of local store
    - Pack data efficiently in vectors
    - Split scene into chunks that can be stored one at a time
  - DMA latency
    - Hide latency through double-buffering
    - Work on one type of object while transferring another

Ray Tracing on the PS3

- Tackling the Challenges
  - No branch prediction
    - Only 3 explicit 'if' statements in code
    - Have compiler unroll loops
  - Inherent SIMD architecture of SPE
    - View everything as packets, work on 4 at a time
  - Load Balancing
    - Have each SPE render every sixth line of the screen

Issues During Implementation

- Heterogeneous architecture
  - SPU and PPU have different instruction sets
    - Two versions of many objects needed to be implemented: one optimized for the PPU and one for the SPU
  - Lack of effective debugging tools
    - Many threads running on different cores – no convenient means of viewing everything

Issues During Implementation

- Physics Engine
  - Third-party ODE used
    - Peculiarities in representation of object positions
    - Difficult to kill built-in OpenGL visualization
  - Integration
    - Physics representation vs. rendering representation

Issues During Implementation

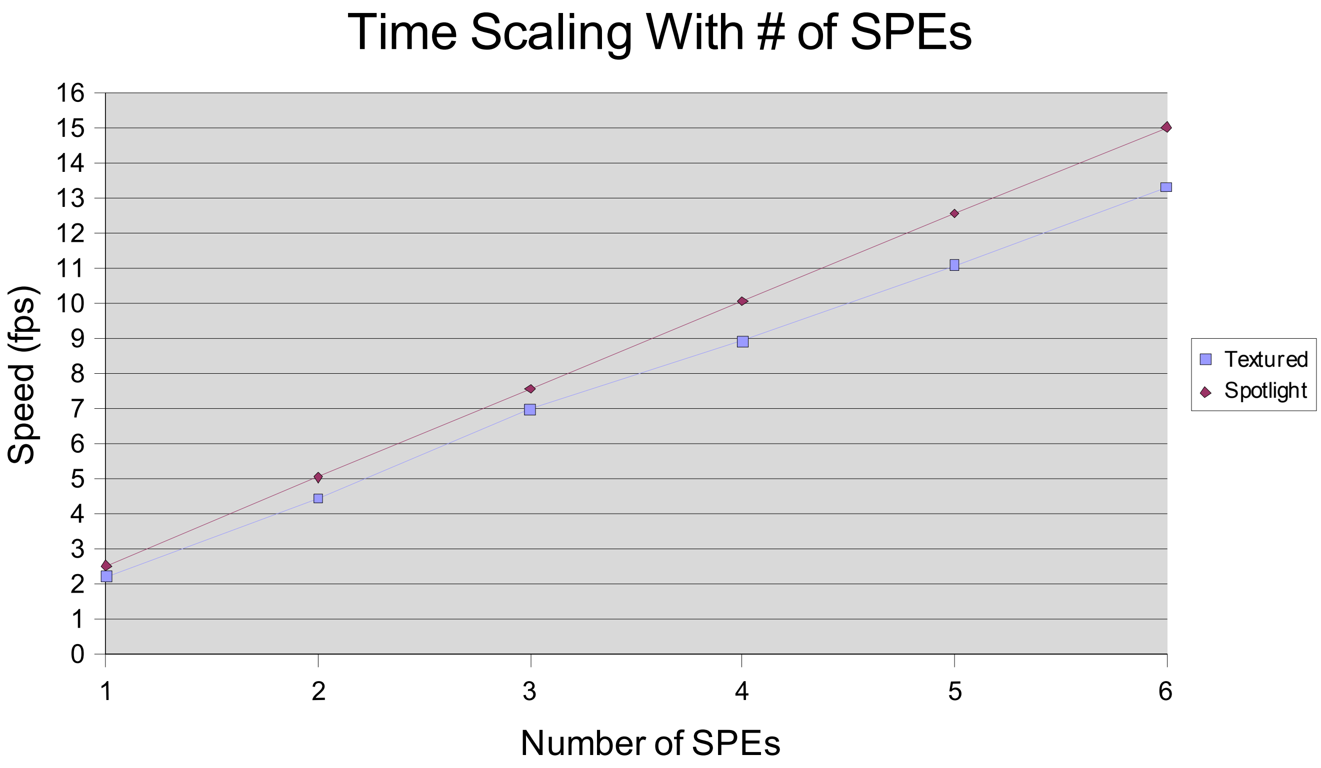
- Time!
  - 4 weeks dedicated to project
    - 1 week for planning
      - Streaming computation or full computation on each SPE?
      - Scene fitting in local store – Software cache, or other means?

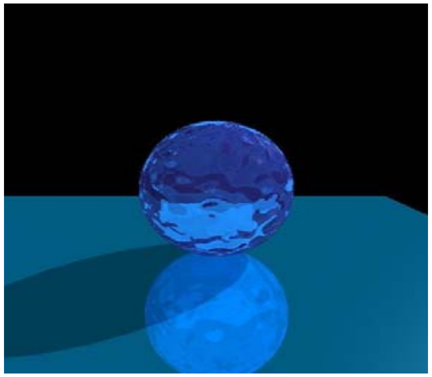
Issues During Implementation

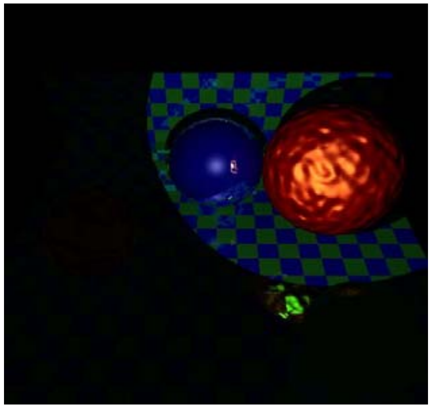
- Time!
  - 4 weeks dedicated to project
    - 3 weeks for coding
      - Many options could not be explored in-depth
      - Simple algorithms chosen over more complex, yet faster ones
      - Dropping parts of initial plan to meet deadline
        - Static, rather than dynamic load balancing
        - Spatial index structure
        - Full scale game with real-time physics done on PPU
        - Other primitives: cylinder, box
        - Larger packets to reduce data dependency stalls

Performance Analysis

- Exact linear speed increase in number of SPEs
  - Test scenes
    - Textured crystal ball: stresses bump mapping / global effects
    - Spotlight: Stresses scene/shading complexity, scene visibility

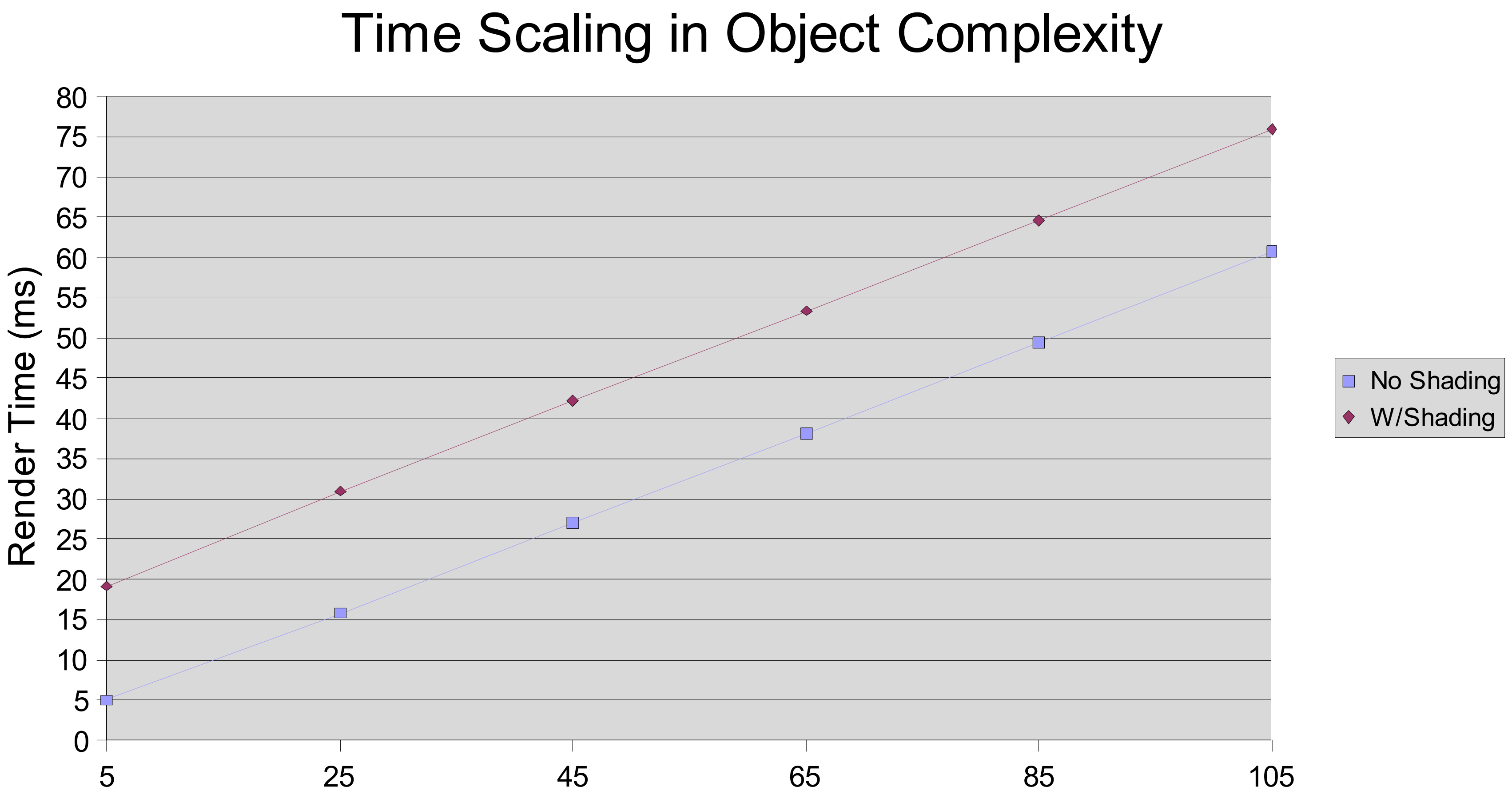






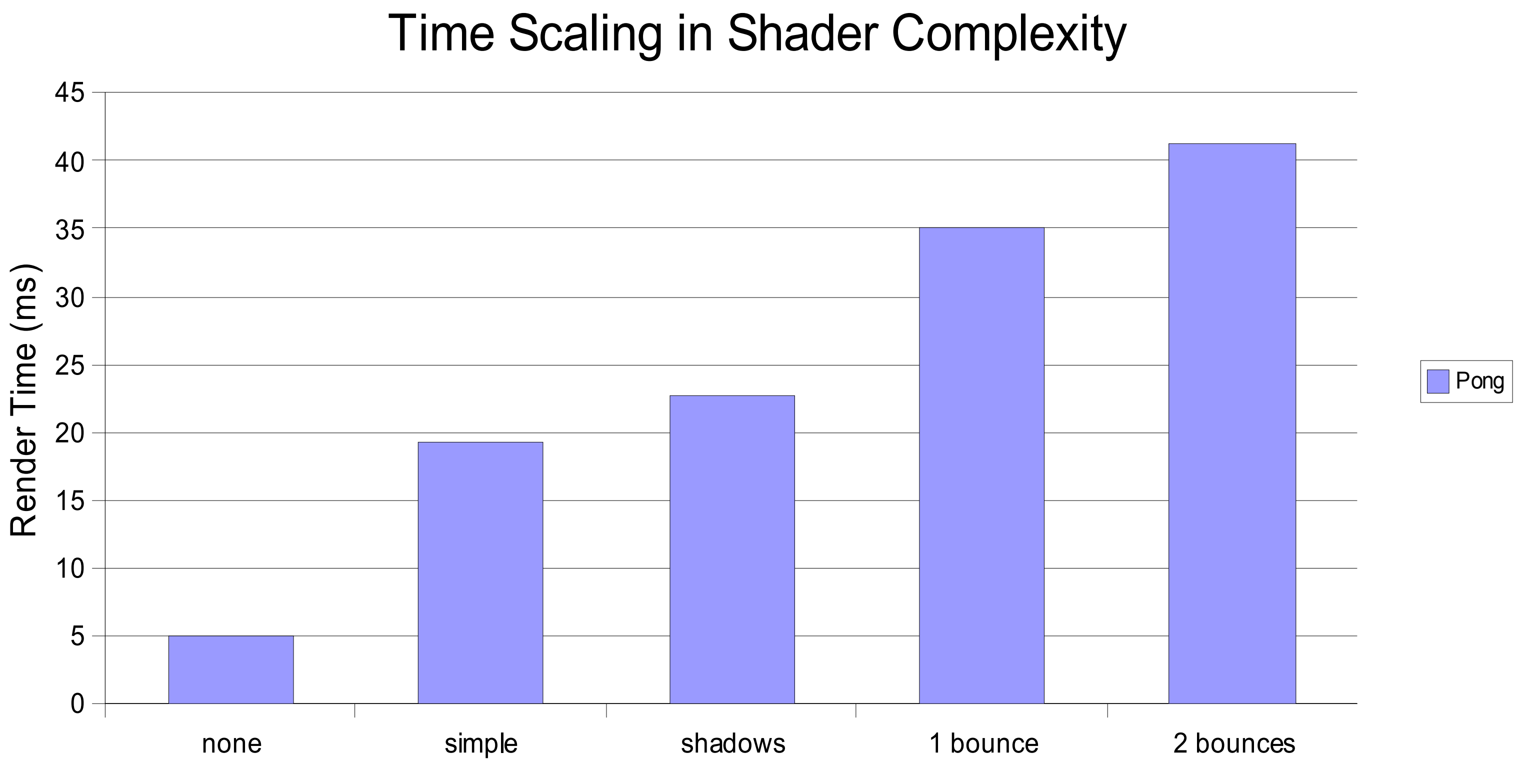
Performance Analysis

- Scalability in object complexity



Performance Analysis

- Scalability in shader complexity
  - Small, constant performance hit for simple shading
  - ~20 ms, constant performance hit for procedural shaders
  - OpenGL-like graphics at ~50 fps



Performance Analysis

- Optimizations
  - Hand-tuning C code to eliminate dependencies
    - Despite compiler optimizations, hand-tuned triangle intersection routine saved ~20ms on complex scenes

```
vector unsigned int valid = spu_and(spu_and(spu_cmpgt(h.t, t),
                                            isgreaterequalf4(one_v, spu_add(u, v)))
                                    spu_and(spu_and(isgreaterequalf4(u, zero_v),
                                                    isgreaterequalf4(v, zero_v))
                                            spu_cmpgt(t, tmin_v)));
```

↓

```
vector unsigned int ugt0 = isgreaterequalf4(u, zero_v);
vector float uPlusv = spu_add(u,v);
vector unsigned int vgt0 = isgreaterequalf4(v, zero_v);
vector unsigned int oldgtnew = spu_cmpgt(h.t, t);
vector unsigned int uPlusvlt1 = isgreaterequalf4(one_v, uPlusv);
vector unsigned int newgttmin = spu_cmpgt(t, tmin_v);
ugt0 = spu_and(ugt0, vgt0);
oldgtnew = spu_and(oldgtnew, uPlusvlt1);
ugt0 = spu_and(ugt0, newgttmin);
vector unsigned int valid = spu_and(oldgtnew, ugt0);
```

Performance Analysis

- Optimizations
  - AOS packing for storage, SOA for computation
    - Goal: Fit as many objects in 16KB (one DMA transfer) as possible

```
vector unsigned char splat0 =
  (vector unsigned char){0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3};
vector unsigned char splat1 =
  (vector unsigned char){4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7};
vector unsigned char splat2 =
  (vector unsigned char){8, 9,10,11, 8, 9,10,11, 8, 9,10,11, 8, 9,10,11};
vector float m_acx = spu_shuffle(m_ac, m_ac, splat0);
vector float m_acy = spu_shuffle(m_ac, m_ac, splat1);
vector float m_acz = spu_shuffle(m_ac, m_ac, splat2);
vector float m_abx = spu_shuffle(m_ab, m_ab, splat0);
vector float m_aby = spu_shuffle(m_ab, m_ab, splat1);
vector float m_abz = spu_shuffle(m_ab, m_ab, splat2);
vector float m_ax = spu_shuffle(m_a, m_a, splat0);
vector float m_ay = spu_shuffle(m_a, m_a, splat1);
vector float m_az = spu_shuffle(m_a, m_a, splat2);
```

Performance Analysis

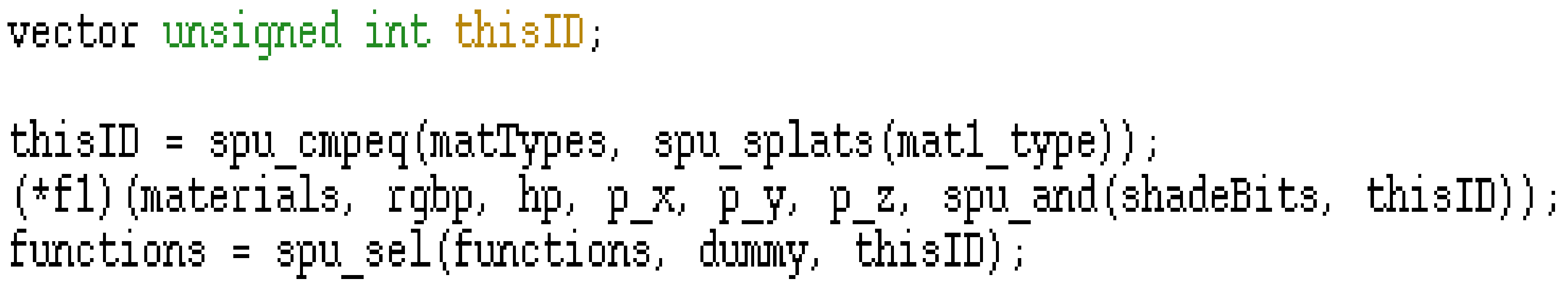
- Optimizations
  - SOA for packets
    - Utilizes full space of four element vector register
    - Perform 3 operations on data, rather than 4

```
struct RayPacket {
  vector float r10;
  vector float r20;
  vector float r30;
  vector float r40;
  vector float r1d;
  vector float r2d;
  vector float r3d;
  vector float r4d;
};
```

```
struct RayPacket {
  vector float x0;
  vector float y0;
  vector float z0;
  vector float dx;
  vector float dy;
  vector float dz;
};
```

Performance Analysis

- Optimizations
  - Approximations
    - No recursion if past threshold depth
    - Assume a shadow if light contribution is less than threshold
  - “Dummy Functions” to assure shaders aren't run twice for the same ray



```
vector unsigned int thisID;

thisID = spu_cmpeq(matTypes, spu_splats(mat1_type));
(*f1)(materials, rgbp, hp, p_x, p_y, p_z, spu_and(shadeBits, thisID));
functions = spu_sel(functions, dummy, thisID);
```

Questions?