Introduction

Mike Houston (AMD)
Aaron Lefohn (Intel)
Logistics

• Website
  – http://cs448s.stanford.edu

• This is a project course
  – Pretty much all the grade is the final project
  – Start thinking about what you may want to do and read a lot

• We will have many guest lecturers
  – Aaron Lefohn (Intel in Seattle) and is co-organized this course, is co-creating the lecture material, and he will be presenting 3 of the lectures and helping with the projects.
  – Tim Foley (Intel/Stanford)
  – John Owens (UC Davis)
  – Kayvon Fatahalian (Stanford)
  – Justin Hensley (AMD)
  – Steve Parker (Nvidia)
Interactive rendering techniques are created using an inseparable mix of data- and task-parallel algorithms and graphics pipelines.
How do users write new interactive 3D rendering algorithms?
Fixed Function Pipelines (DX7)

• Writing new rendering algorithms means
  – Tricks with multitexture, stencil buffer, depth buffer, blending ...

• Examples
  – Stencil shadow volumes
  – Hidden line removal
  – ...

CS448S – Spring 2010 – Beyond Programmable Shading
Programmable Shaders (DX8-10)

• Writing new rendering algorithms means
  – Tricks with stencil buffer, depth buffer, blending ...
  – Plus: Writing shaders

• Examples
  – User-defined materials
  – User-defined lights
  – User-defined data structures (built in texture memory)
Software Graphics: Part I (DX11)

• Writing new rendering algorithms means
  – Tricks with stencil buffer, depth buffer, blending ...
  – Plus: Writing shaders
  – Plus: Writing data- and task-parallel algorithms
    – Analyze results of rendering pipeline
    – Synthesize data structures

• Examples
  – Dynamic summed area table
  – Dynamic quadtree adaptive shadow map
  – Dynamic histogram-analysis shadow map
  – Dynamic ambient occlusion
  – ...

“Fast Summed-Area Table Generation and its Applications,” Hensley et al., Eurographics 2005

“Real-Time Approximate Sorting for Self Shadowing and Transparency in Hair Rendering,” Sintorn et al., I3D 2008


“Dynamic Ambient Occlusion and Indirect Lighting,” Bunnell, GPU Gems II, 2005
Software Graphics: Part II (DX11+)

• Writing new rendering algorithms means
  – Tricks with stencil buffer, depth buffer, blending ...
  – Plus: Writing shaders
  – Plus: Writing data- and task-parallel algorithms
    – Analyze results of rendering pipeline
    – Synthesize data structures
  – Plus: Creating new and extended rendering pipelines

• Examples
  – Micropolygon rendering
  – Ray tracing pipelines
  – ...

“FreePipe: a Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects.”
Liu et al., ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2010

“Hardware-Accelerated Global Illumination by Image Space Photon Mapping.”
McGuire and Luebke, High Performance Graphics 2009

“RenderAnts: Interactive Reyes Rendering on GPUs,”
Zhou et al., ACM SIGGRAPH Asia 2009

OptiX, NVIDIA 2008
Order Independent Transparency
There is no single graphics pipeline

• There is no single workload to optimize

• Moving forward, interactive rendering is an inseparable mix of
  – Task- and data-parallel algorithms
  – Standard, extended and custom graphics pipelines
Braided Parallelism

• Intermixed task-, data-, and pipeline parallelism

"Braid: Integrating Task and Data Parallelism," West and Grimshaw, 1994

Image from Johan Andersson, DICE
But Some Food For Thought...
The Wheel of Reincarnation

Gradually the processor became more complex.... Finally the display processor came to resemble a full-fledged computer with some special graphics features. And then a strange thing happened. We felt compelled to add to the processor a second, subsidiary processor, which, itself, began to grow in complexity. It was then that we discovered a disturbing truth. Designing a display processor can become a never-ending cyclical process. In fact, we found the process so frustrating that we have come to call it the "wheel of reincarnation."

Will There Be Another Turn of The Wheel of Reincarnation?

• Is “the rise of SW graphics” a temporary (5-10) year window as we go around the wheel of reincarnation or has the wheel stopped turning?

• If it has stopped turning, why?

• If it hasn’t stopped turning, what will be the next fixed-function?
  – Great time to be a graphics researcher because the killer-app SW rendering pipelines/capabilities created now may define future fixed-function hardware
“Render to Data Structures”

• DX11 PixelShader 5
  – Atomics to global memory
  – Gather/scatter to memory (“unordered access views”)

• Order independent transparency
  – Capture all rendered fragments
    – Render directly to grid-of-lists data structure instead of framebuffer
    – No framebuffer bound while rendering 😊
    – Increment global counter to get unique address for fragment
    – Scatter \{depth, color, prevFrag\} to UAV shared by all pixels
    – Result is grid of linked lists
  – Sort and blend lists for final image (pixel shader or compute shader)
Introduction to Compute Shader (+ OpenCL/CUDA)
Compute Shader / OpenCL / CUDA

• Execution/Programming Model

• SIMD Processing

• Latency Hiding

• Use Cases
Execution Model

• Fundamental unit is **work item**
  – Single instance of scalar program

• Work items collected in **work groups**
  – Work group scheduled to a single core
  – Supports efficient sharing of data

• Users launch a **grid** of work groups
  – Spawn many independent tasks

```c
void f(...) {
    int x = ...;
    ...;
    ...;
    if(...) {
        ...
    }
}
```
## Terminology Decoder Ring

<table>
<thead>
<tr>
<th>Compute Shader</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>Traditional CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>thread</td>
<td>work item</td>
<td>SIMD lane</td>
</tr>
<tr>
<td>thread group</td>
<td>warp</td>
<td>-</td>
<td>thread</td>
</tr>
<tr>
<td>thread block</td>
<td></td>
<td>work group</td>
<td>-</td>
</tr>
<tr>
<td>streaming</td>
<td>compute unit</td>
<td></td>
<td>core</td>
</tr>
<tr>
<td>multiprocessor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>grid</td>
<td>N-D range</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

• We will use a cross-section of these terms
Execution Model

• Work groups intended to be unit of locality
  – All work items in group share a read/write region of memory
  – Memory fences, atomics, barriers used for communication, synchronization

• Intended usage
  – All work items cooperate to fetch data (e.g. matrix sub-blocks)
  – Barrier/fence
  – Compute, utilizing data loaded by peers

• Implementations expected to exploit locality hint
  – Map shared memory region to on-chip scratchpad or cache
Programming Model

• User writes per-work-item code in kernel language
  – Write scalar, sequential code for one work item
  – DirectX: HLSL-based Compute Shader language
  – OpenCL, CUDA: subset of C language

• Explicit construct to launch a grid
  – OpenCL: `clEnqueueNDRangeKernel( ... )`
  – DirectX: `ID3D11DeviceContext::Dispatch( GridX, GridY, GridZ )`
  – CUDA: `f<<< GridDims, GroupDims >>>( arguments )`

• Work groups always assumed independent
  – Application explicitly declares parallelism
Always-SIMD Parallelism

• User writes program for a single work item
  – Serial, scalar code

• Launches large batch of parallel work items
  – Explicit declaration of independence of work
  – Abstracts over SIMD lanes, threads, and cores

• Programming model abstracts/exploits SIMD hardware
  – Implicitly
  – Consistently: scales across any SIMD width
  – With high utilization: 100% SIMD utilization is default

• This is the key advantage of GPU-derived programming models
Compute Shader Usage Patterns
Use Cases

• 1:1 Mapping

• Simple Fork/Join

• Switching Axes of Parallelism

• Custom Work Queues: Task Parallelism

Increasing Sophistication
### 1:1 Mapping

- One work item per ray / per pixel / per matrix element
- Every work item executes the same kernel
- Often first, most obvious solution to a problem
- Some systems mandate this approach (Brook, RapidMind)

```c
void saxpy( int i,
            float a,
            const float* x,
            const float* y,
            float* result )
{
    result[i] = a * x[i] + y[i];
}
```
Simple Fork/Join

• Some code must run at work-group granularity
  – Example: work items cooperate to compute output structure size
  – Atomic operation to allocate output must execute once

• Idiomatic solution
  – Barrier, then make work item #0 do the group-wide operation

```c
void subdividePolygon(...) {
    shared int numOutputPolygons = 0;

    // in parallel, every work item does
    atomic_add( numOutputPolygons, 1);
    barrier();

    Polygon* output = NULL;
    if( workItemID == 0 ) {
        output = allocateMemory( numOutputPolygons );
    }
    barrier();
    ...
}
```
Multiple Axes of Parallelism

• Deferred rendering with DX11 Compute Shader
  – Example from Johan Andersson (DICE)
  – 1000+ dynamic lights

• Multiple phases of execution
  – Work group responsible for a screen-space tile
  – Each phase exploits work items differently:
    – Phase 1: pixel-parallel computation of tile depth bounds
    – Phase 2: light-parallel test for intersection with tile
    – Phase 3: pixel-parallel accumulation of lighting

• Exploits producer-consumer locality between phases
SW Scheduling: Task Parallelism (aka “persistent threads”)

• Irregular workloads
  – Ray tracing, rasterization, collision detection, ... are poor fit for strict fork/join of homogeneous data-parallel work

• Perform user-space software scheduling
  – Launch just enough work groups/items to fill the machine
  – Each “warp” pulls from a shared request queue
  – Dispatch to appropriate kernel code based on request

• Programming models no longer just data-parallel
  – Fully task-parallel programming model
  – Still gets implicit SIMD utilization from scalar code
Software Graphics Examples

• Two different cases

• Augmenting the standard pipeline
  – Analyzing the output of the pipeline
  – Synthesizing input for the pipeline

• Building new graphics pipelines in software
  – Example 1: FreePipe
  – Example 2: NVIDIA’s OptiX ray tracing API
SW Pipelines: Task-System Based

• Dynamically load balance between stages
• Queues between stages
• Each stage spawning new work by placing on queues
• Each stage runs in SIMD
Interesting Research Directions

• Abstractions to make it easier to write software pipelines

• Programming models to combine task, data, and pipeline parallelism to C++ in a way that is elegant, minimalist, but effective

• Killer-apps and abstractions for real-time SW graphics
  – SW rendering pipeline research needs to demonstrate higher-quality rendering that is also adoptable by game developers (“authorable performance”)
  – The best ideas that come of the of this SW graphics era may define the next generation of fixed-functionality (if the wheel is still turning)
Questions we hope to cover in this course:

• What makes up a graphics pipeline (Thurs lecture)?
• How does a GPU work and how is it different than a CPU?
• How are GPUs programmed?
• How do you design algorithms with data-parallel primitives?
• How do you write SW graphics pipelines?
• How do games use heterogeneous parallel programming?
• How are researchers using these techniques?
• What are the open research problems in this area (throughout the course)?
Conclusions

• Software + hardware graphics is here today (beginning “for real” in DX11)
  – Graphics programming is no longer simply a single pre-defined pipeline
  – Research is ablaze with software rendering research on GPUs and CPUs

• Future real-time rendering programming will consist of
  – A pre-defined (Direct3D/OpenGL) rendering pipeline
  – User-defined software pipelines
  – User-defined data- and task-parallel code tightly coupled to graphics pipelines

• Is the wheel of reincarnation still turning?