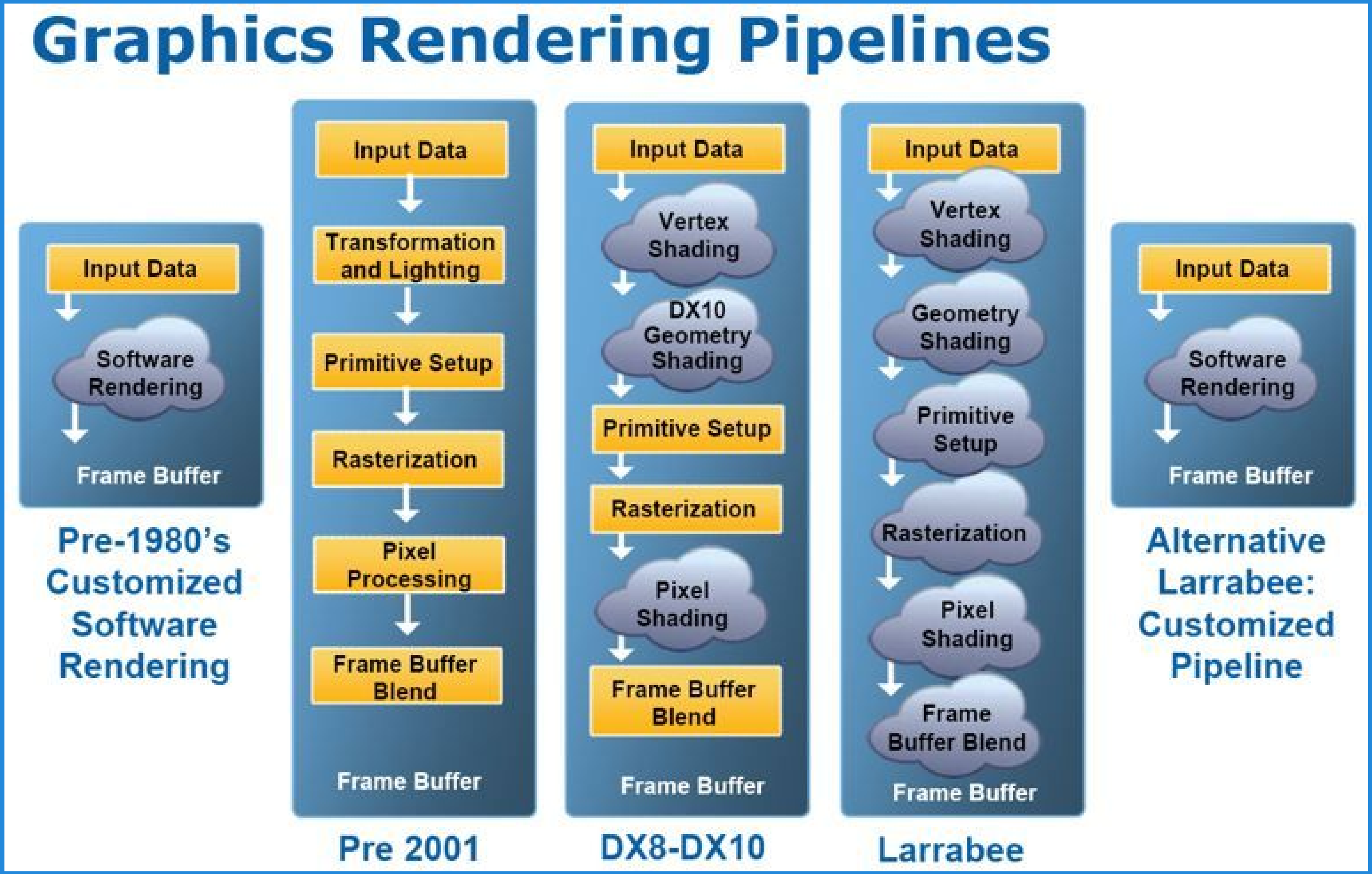
CS195V Week 9

GPU Architecture and Other Shading Languages

GPU Architecture

- We will do a short overview of GPU hardware and architecture
  - Relatively short journey into hardware, for more in depth information, check out...
  - http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15869-f11/www/lectures/07_gpucore.pdf
  - http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf
- We will look in to some old GPU architectures and how they have evolved over the years



Comparison of rendering pipelines through the ages
(ignore the Larrabee stuff)

Older Architectures

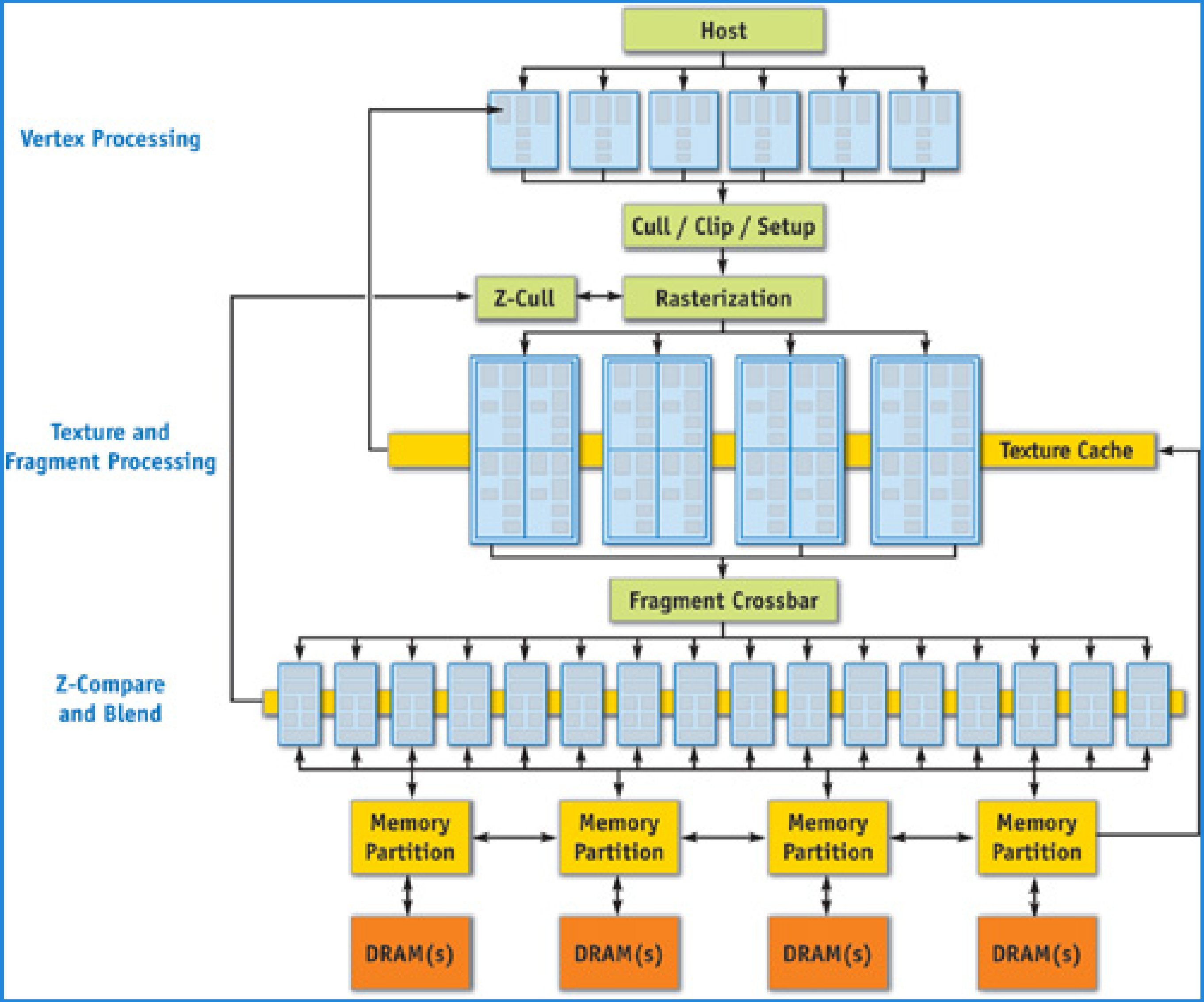
- Fixed function architecture
  - You'll see dedicated units for vertex and fragment processing
  - In even earlier architectures, you would see more rigid blocks in vertex and fragment stages because there were no shaders
- Even at this point, we see notions of parallelism
  - There are multiples of each of these fixed function units
  - Vendors would boast the number of "pixel pipelines" they had

Older Processors

- Fixed function units were implemented either directly in hardware or hardware-level instructions
- At this point, you didn't really have to worry about giving instructions (i.e. programs) to the individual units on how to operate

Programmable Shading

- With the advent of programmable shading, the overall structure of the pipeline remains the same, but programmable units replace the fixed function units
  - Requires some extra hardware for managing instruction fetches
  - These programmable units operate much like a normal processor, with the usual pipeline stages that you might expect in a processor (fetch, decode, ALU, memory, etc.)



Old(er) Architecture - Geforce 6 Series

Adding More Stages

- As OpenGL/Direct3D added more shader stages (geometry, tessellation), the architecture needed to expand to include the necessary hardware
- Add more programmable stages in the corresponding locations in the pipeline
- At this point, vertex/fragment/geometry units are programmable, but only with their respective shader code
  - So you couldn't run vertex shader code on a fragment unit

Unified Architecture

- With the NVIDIA 8 series cards (and some AMD card I don't remember), graphics architecture moved to generic "shader units" rather than programmable units for each stage
  - These shader units can run shaders from any stage
- Since these units are generic, what's to say they can't run arbitrary computations?
  - HMMM?????? MAYBE THAT HAS SOME APPLICATIONS I DON'T KNOW

Gains from Unified Architecture

- Since the units can run any type of shader, you can maximize use of hardware regardless of the program's emphasis on a particular type of operation
  - Previously, if you had an application with disproportionately many fragment operations, you would have some vertex processors sitting around doing nothing
- Allows hardware to balance the workload to improve performance

Streaming Multiprocessors

- These generic processing units are called streaming multiprocessors (SM)
- Each SM has its own hardware for fetching and decoding instructions
  - Scheduling, dispatch, etc.
- It has its own register block
- Various memory and other units
  - The Fermi chips have "special function units" for things like trig functions and "Load/Store" units for memory operations
- Also some shared cache

The Cores

- A single SM owns a number of compute cores (AMD calls them stream processors, NVIDIA uses both stream processor and CUDA core)
    - In the Fermi cards, 32 cores per SM, 16 SM, so 512 cores on the card total
- This means that the SM will give the same program to all of its cores, which will all execute in parallel
    - Parallelism within parallelism!

High level view of GPU core (right)

View of a single streaming multiprocessor (left)

Writing Applications for GPUs

- So given this architecture, what kind of applications run well on it?
- Parallelizable ones, obviously, but what else?
- What kind of programming conventions cross over well? What kinds of operations are more or less costly for this versus a traditional CPU?

Branching

- Say we have one of our streaming multiprocessors from above
  - Instructions and memory are shared between the compute cores
- If our program has branching, some of the cores may take the branch, while some do not
- In this case, some cores may finish execution before others, and will have to wait since the SM as a whole has to finish all of its operations before moving on

More Branching

- In the worst case, one thread lags behind the others, makes them wait
  - Can lead to significant performance losses
- In general, we do not branch as often in our GPU code, though you can certainly do it
  - Especially if the time to complete both branches is roughly equal

Computation versus Memory

- As we know, the GPU has lots of memory and lots of memory bandwidth
  - Has to deal with lots of operations on large textures quite often
- However, the actual memory bandwith is only 6-8 times larger than CPU
  - But there are hundreds of cores which may want to use this bandwidth
  - In contrast, 20+x the raw computational power
- Thus, memory usage is perhaps one of the most important considerations in writing GPU programs

Memory Operations, cont.

- There are many operations both in software and hardware to minimize memory accesses and make them fast
  - This is part of why images are so gimmicky
  - Texture fetches in batches, reordering of memory operations, cache coherency
- Memory bandwidth is a precious commodity in GPU programs, so use it well...
- Also important from a power perspective
  - Moving data across the GPU die uses significantly more power than a single arithmetic operation

Compute-heaviness

- If you look at your average shader, you will probably see many more compute operations than memory operations
- Since we have so much raw compute power available, we favor applications that have a large compute/memory ratio
- However, precomputing some parameters may lead to performance gains as well
- Balancing what to do when is also an important skill

Some other stuff...

- Warps(NV) and Wavefronts(AMD): groups of parallel threads that execute the same instruction
  - These would be assigned to a single streaming multiprocessor
- A single SM can interleave between many of these Warps/Wavefronts, allowing for parallel execution of thousands of threads
  - ex. the Fermi chips interleave 48 warps per SM
  - 16 SM x 48 warps x 32 threads/warp = 24576 threads

Shading Languages

- Cook and Perlin were the first to develop languages for running shader computations
  - Perlin computed noise functions procedurally, introducing control constructs
  - Cook developed shade trees
- These ideas led to the development of Renderman at Pixar (Hanrahan et. al) in 1988
- Most shader languages today are similar - all C like languages
  - This is good - once you know one, you pretty much know them all

Real-time Shading Languages

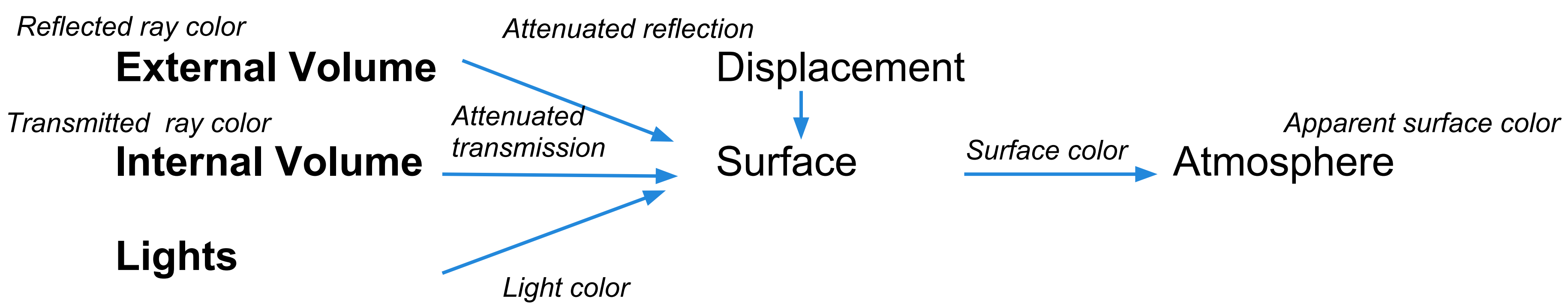
- **ARB Assembly**
  - Standardized in 2002 as a low level instruction set for programming GPUs
  - Higher level shader languages (HLSL/Cg) compile to ARB for loading and execution
- **GLSL**
  - Shading language for OpenGL programs (hopefully you know what this is)
- **HLSL**
  - Probably GLSL's main competitor, High Level Shader Language (HLSL) is essentially DirectX's version of GLSL
- Cg
  - "C for graphics" shader language developed by NVIDIA which can be compiled for both DirectX and OpenGL programs

Offline Shading Languages

- RSL
  - Renderman shading language, probably the most common offline shading language
  - One of the first higher level shading langauges
- Houdini VEX
- Gelato

RenderMan Shading Language

- Six shader types
  - Lights, surfaces, displacements, deformation, volume, imager
- Key idea: separate surface shader from light source shaders



Renderman Shading Language

Some built in variables

- P - surface position
- N - shading normal
- E - eye point
- Cs - surface color
- Os - surface opacity
- L, Cl - light vector and color

Renderman Shading Language (Light Shader)

```
light pointlight(float intensity = 1;
                 color lightcolor = 1;
                 point from = point "shader" (0,0,0);)
{
    illuminate(from)
        Cl = intensity * lightcolor / (L . L);
}
```

- The illuminate statement specifies light cast be local light sources
- There is also the solar statement for distant light sources

Renderman Shading Language

```
surface diffuse(color Kd)
{
    Ci = 0;
    // integrate light over hemisphere
    illuminance (P, Nn, Pi/2)
    {
        Ci += Kd * Cl * (Nn . normalize(L));
    }
}
```

- The surface shader outputs Ci
- Cl is computed by the light shader

GLSL

- By now you know more than you want to about GLSL
- GLSL is cross platform - each hardware vendor includes the compiler in their driver
  - Allows vendor to optimize their compiler for their hardware
  - GLSL compilers compile your program directly down to machine code (not true of HLSL / Cg which first compile to assembly)
  - But causes fragmentation between vendors (and some things may or may not work on different cards / manufacturers)

HLSL

- Developed alongside the NVIDIA Cg shader language and is very similar
- Tightly integrated with the DirectX framework
- Versions are specified via the shader model
  - ex. Shader Model 1 specifies shader profiles vs_1_1, and Shader Model 5 (current iteration) specifies cs_5_0, ds_5_0, etc.
- HLSL has six different shader stages
  - Vertex, Hull, Domain, Geometry, Pixel, Compute
  - Compute shader is the main difference between GLSL stages vs HLSL stages
- HLSL, unlike GLSL can define states in the shader

HLSL

- HLSL shaders are stored in an "effect" file
- Each effect file can contain multiple techniques
  - If more than one technique is specified, it will use other techniques if one technique fails due to inappropriate hardware
- Each technique can be composed of multiple passes
  - Each runs through the shader pipeline once
  - Passes can be blended or accumulated into a framebuffer

HLSL Texture Mapping

```
matrix World, View, Projection;
Texture2D colorMap;
SamplerState linearSampler // texture sampler
{
    Filter = min_mag_mip_linear;
    AddressU = Clamp;
    AddressV = Clamp;
    MaxAnisotropy = 16;
};

RasterizerState rsSolid // rasterizer state
{
      FillMode = Solid;
      CullMode = NONE;
      FrontCounterClockwise = false;
};
```

```
struct VS_INPUT // vs input format
{
      float4 p : POSITION0;
      float2 t : TEXCOORD;
      float3 n : NORMAL;
};

struct PS_INPUT // ps input format
{
      float4 p : SV_POSITION;
      float2 t : TEXCOORD0;
};
```

HLSL Texture Mapping

```
PS_INPUT VS_SIMPLE(VS_INPUT input)
{
	PS_INPUT_PV output;
	//transform position to clip space
	input.p = mul(input.p, World);
	output.p = mul(input.p, View);
	output.p = mul(output.p, Projection);
	output.t = input.t;

	return output;
}

float4 PS_SIMPLE(PS_INPUT input) : SV_Target
{
	return colorMap.Sample(linearSampler,
input.t);
}
```

```
technique10 SIMPLE {
	pass P0
	{
		SetVertexShader(CompileShader
(vs_4_0, VS_SIMPLE()));
		SetGeometryShader(NULL);
		SetPixelShader(CompileShader(ps_4_0,
PS_SIMPLE()));
		SetRasterizerState(rsSolid);
	}
}
```

HLSL

- Note how similar this is to GLSL
- The compute shader (CS), is a new shader stage introduced in DX11
  - I imagine OpenGL / GLSL will come out with something similar soon - for now you have to switch into CUDA or OpenCL to run compute
- Also known as DirectCompute techhnology
- Integrated with Direct3D for efficient interop with the graphics pipeline
- Exposes much more general compute capability

Cg

- Evolved from RTSL from Stanford
- Platform and card neutral shader language
  - In practice Cg tends to work better on NVIDIA cards (I wonder why?)
- Interestingly there is an NVIDIA Cg compiler which can (when configured properly) take HLSL code and output OpenGL compatible shader code
- It looks very similar to HLSL