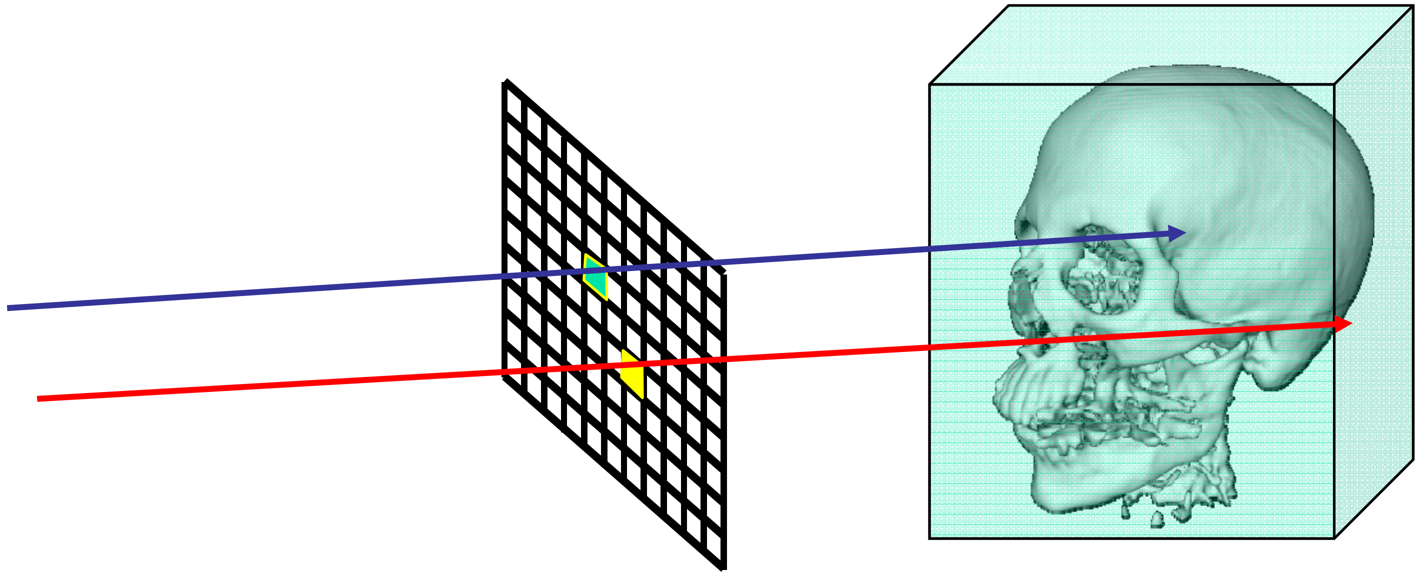
What we will cover

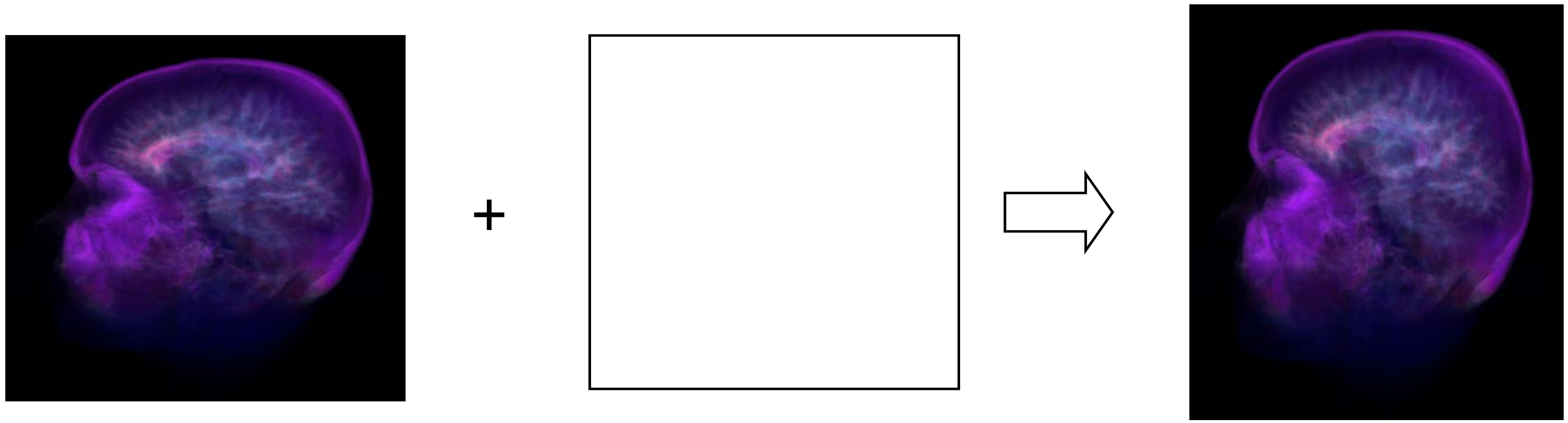
- Contour Tracking
- Surface Rendering
- Direct Volume Rendering
- Isosurface Rendering
- Optimizing DVR
- Pre-Integrated DVR
- Unstructured Volume Rendering
- GPU-based Volume Rendering

Ray Casting Idea



How we do *parallelize* ray casting
and traversal of all view rays!!

Texture Mapping

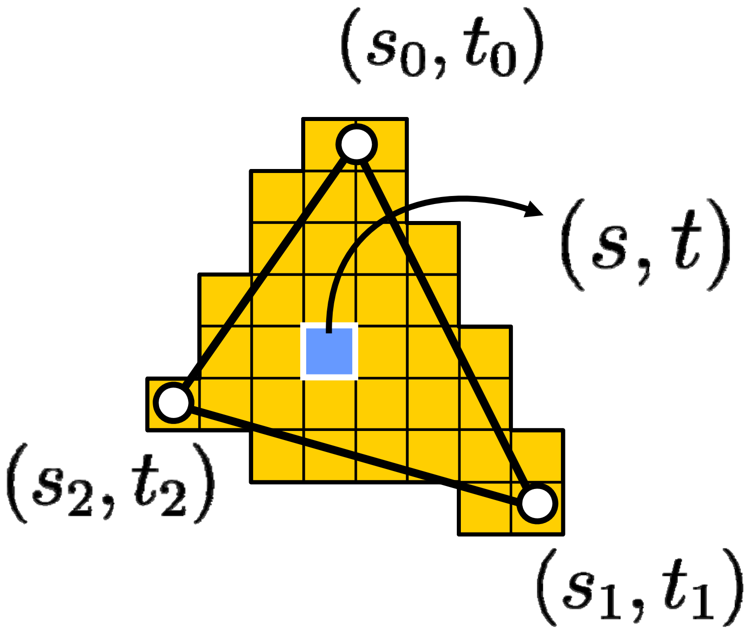


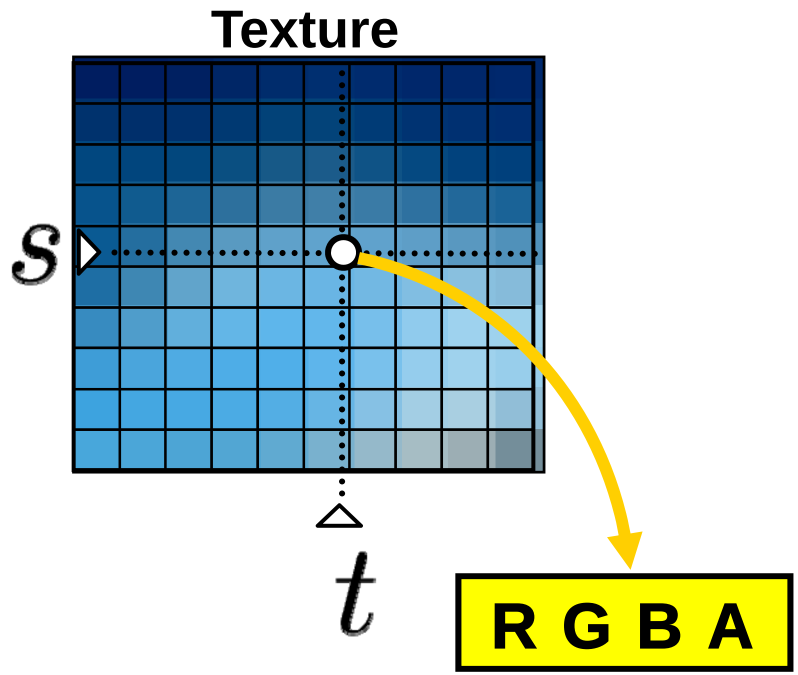
2D image

2D polygon

Textured-mapped polygon

How does a texture work?

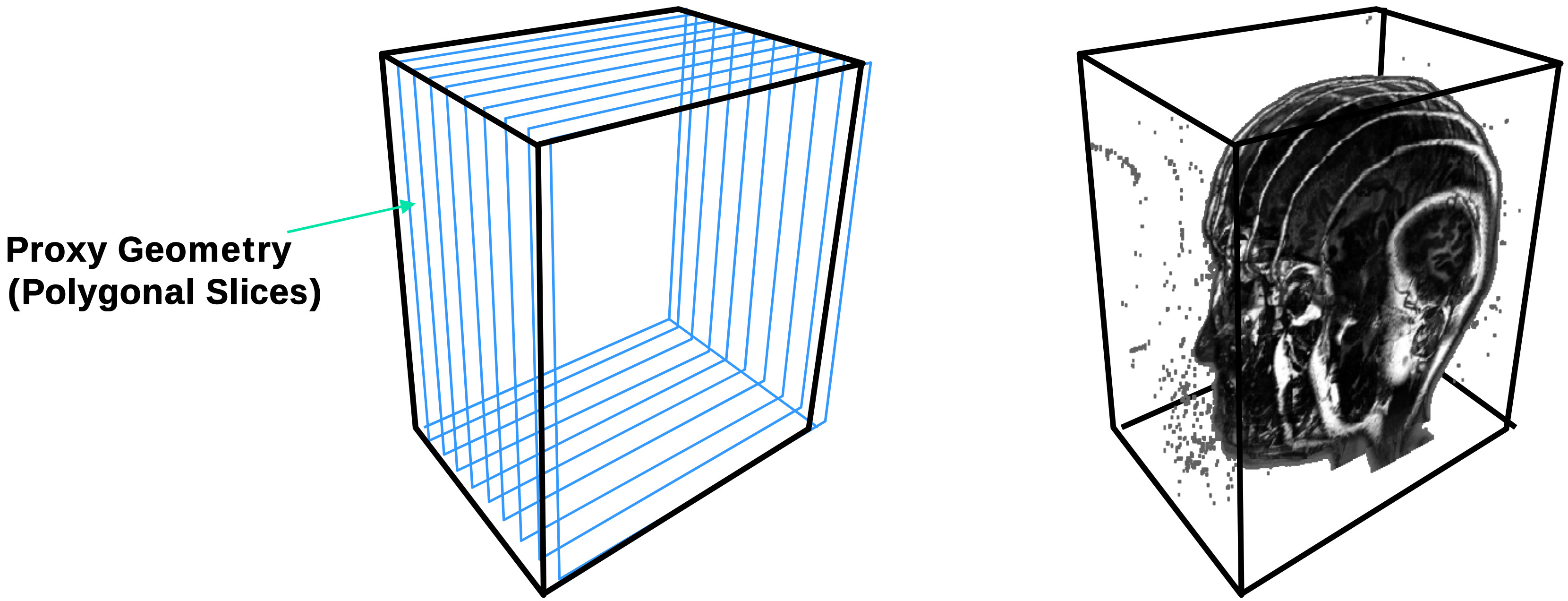




For each fragment:
interpolate the
texture coordinates
(barycentric)

Texture-Lookup:
interpolate the
texture color
(bilinear)

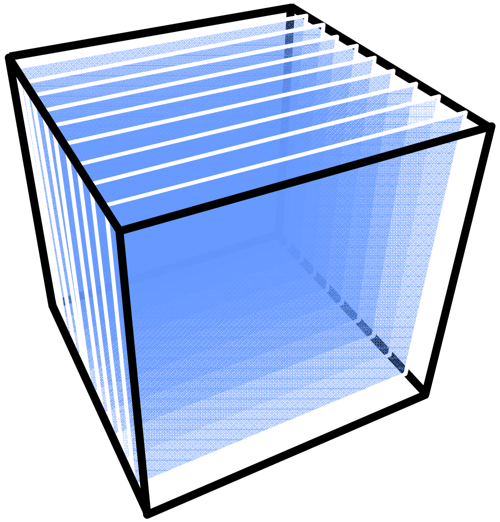
Texture based volume rendering



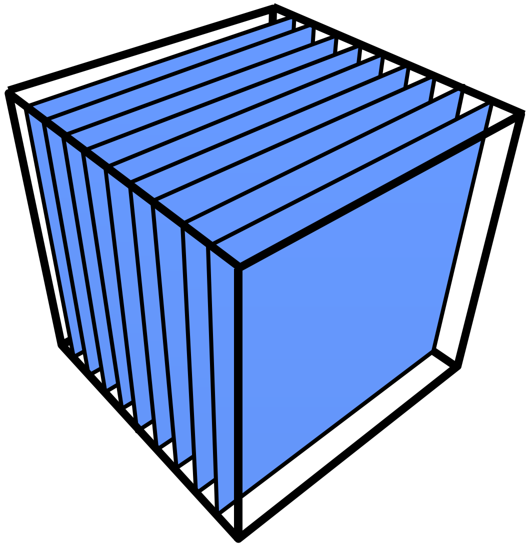
1. Render every slice in the volume as a texture-mapped polygon
2. The proxy polygon will sample the volume data
3. The polygons are blended from back to front

2D Textures

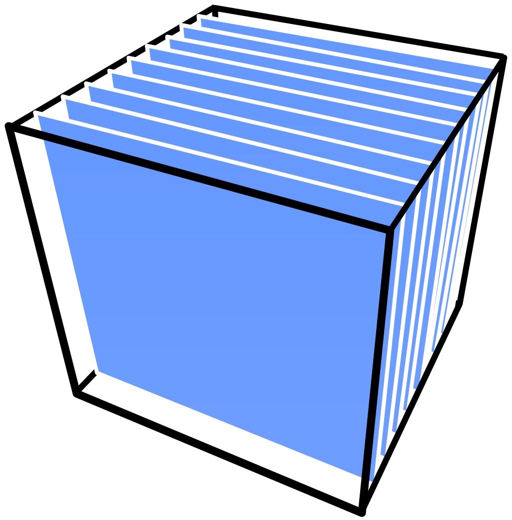
- Axis-aligned slices
- Bilinear Interpolation in Hardware
- 3 copies of the data set in memory
  - Reorganize the textures on the fly is too time consuming. We want to prepare the texture sets beforehand



xz slices



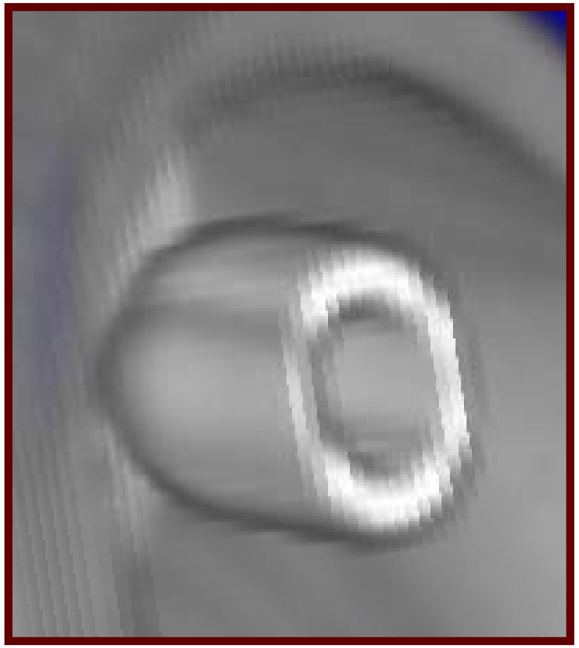
yz slices



xy slices

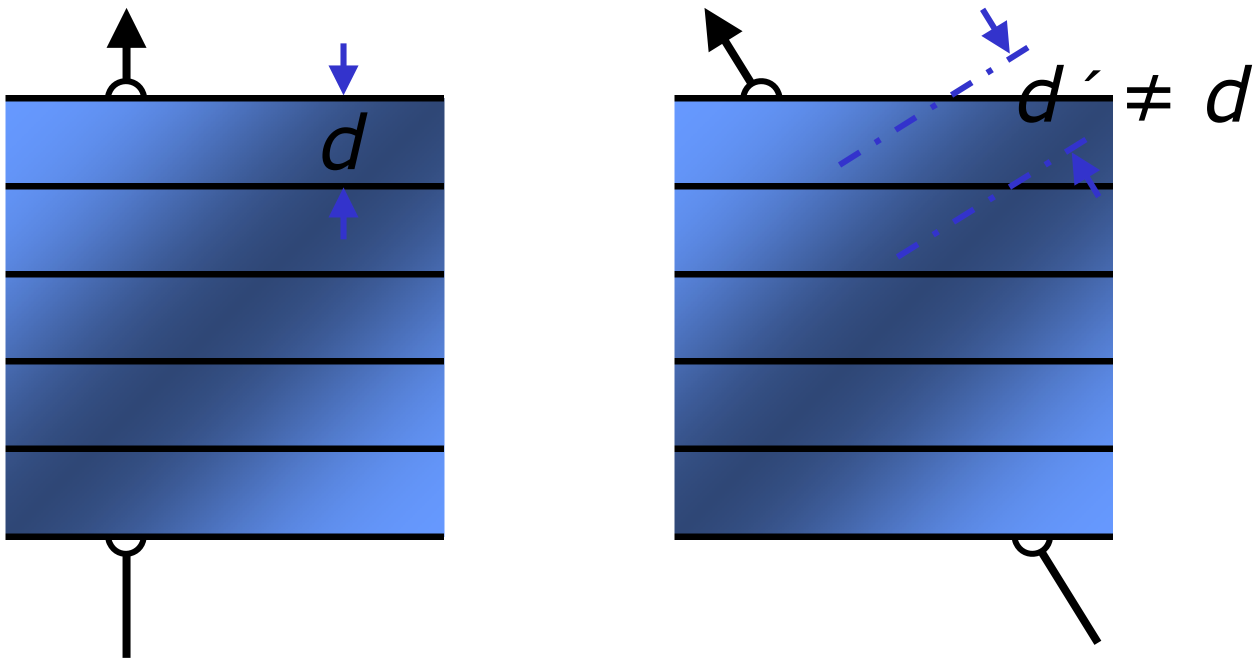
2D Textures: Drawbacks

- Bilinear instead of trilinear interpolation



2D Textures: Drawbacks

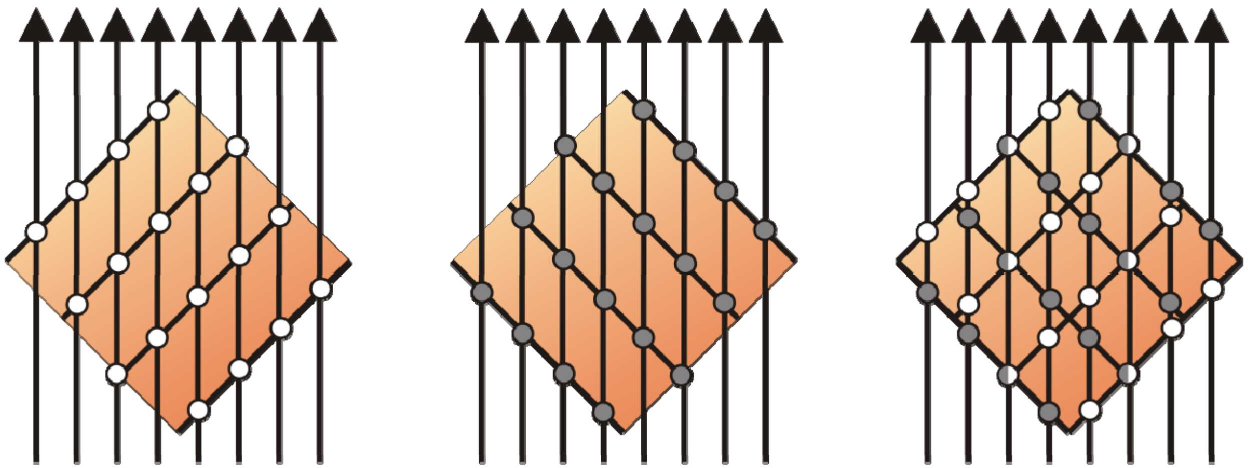
- Inconsistent sampling rate



- Emission/Absorbtion incorrect
- Supersampling not possible!

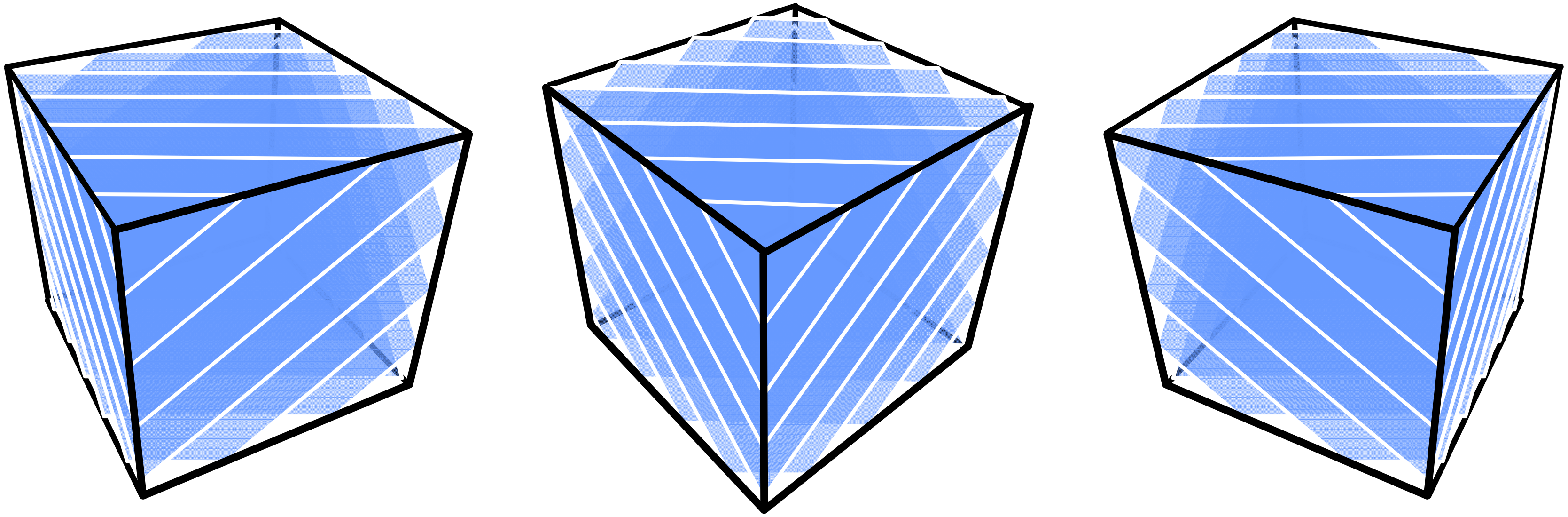
2D Textures: Drawbacks

- Popping effect: There is a sudden change of slicing direction when the view vector transits from one major direction to another
- The change in the image intensity can be quite visible



3D Texture

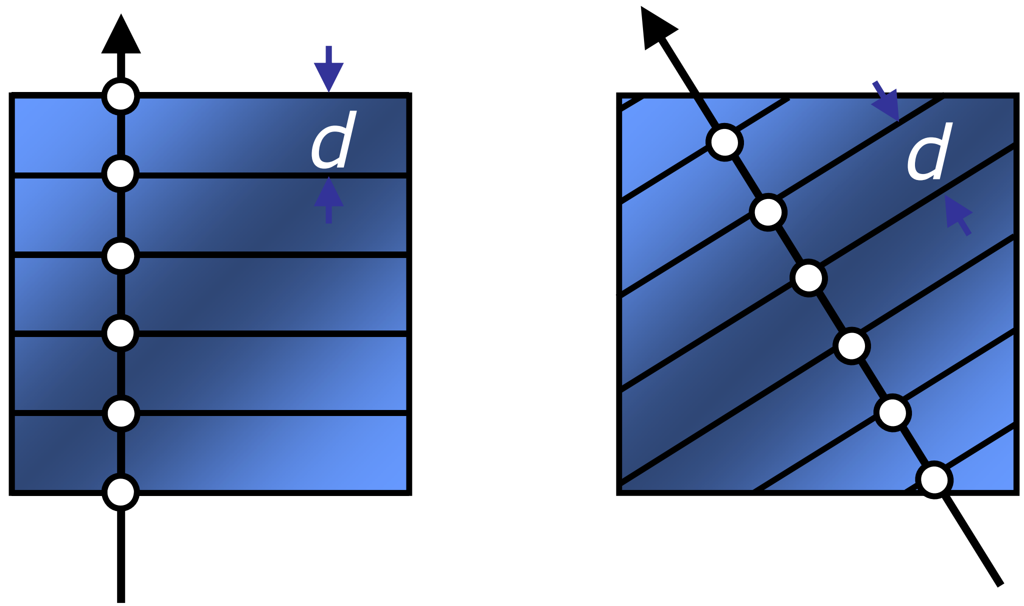
- Trilinear interpolation in hardware
- Slices are parallel to image plane



- Volume is one texture block in memory

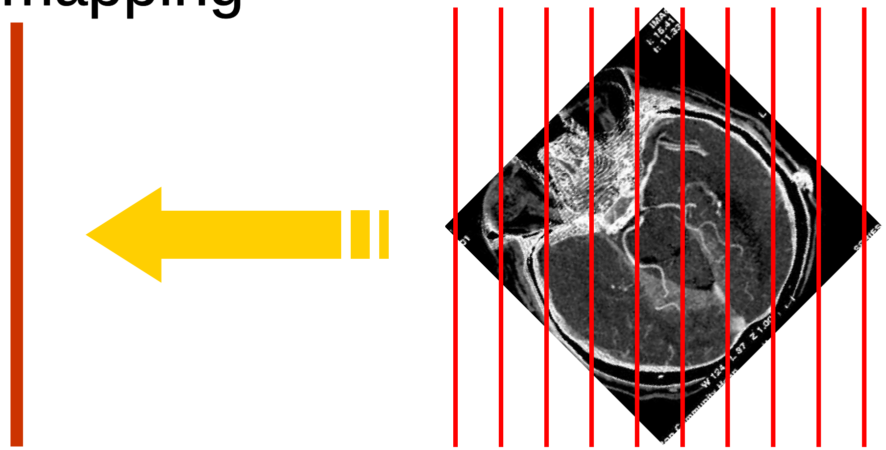
3D Texture: Advantages

- Consistent sampling rate
  (except for perspective projection)
- Supersampling by increasing the number of slices



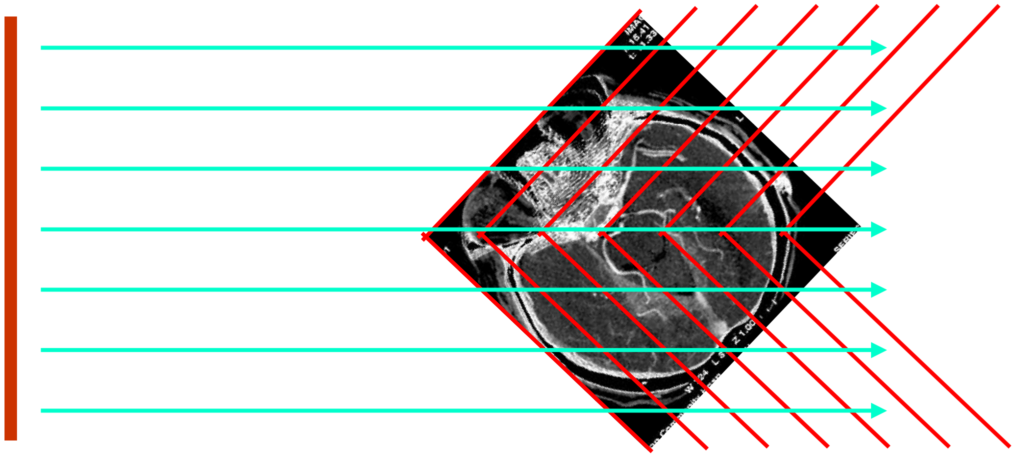
Volume Rendering

- Texture-mapping



- Ray Casting

How we do *parallelize!!*



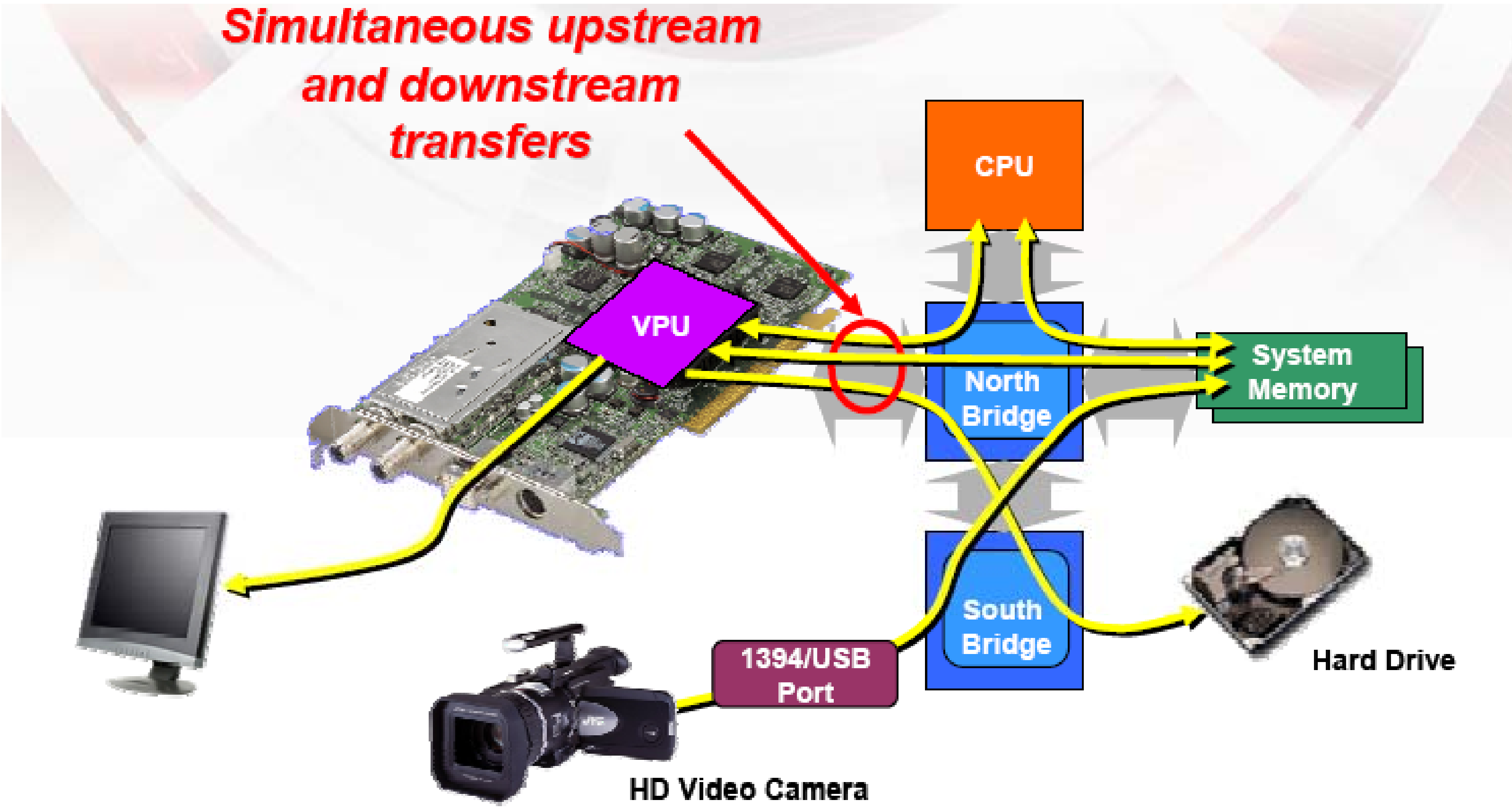
Graphics Hardware

- Graphics hardware is used on most PCs now
- Dedicated hardware 2D and 3D graphics processing unit (GPU)
  - nVIDIA: GeForce series (latest: GeForce 8800/G80)
  - ATI:Radeon series (latest: Radeon HD2900/R600)
- Derived by game & graphics applications
- Input: Triangle list, textures, etc.
- Output: Pixels in the frame buffer
- Programmable pixel, vertex, video engines

Graphics Hardware

- CPUs are optimized for high performance on sequential code
  - Branch prediction, out-of-order execution
- GPUs are optimized for highly data-parallel nature of graphics computation
  - multiply & add vectors in 1 clock
- Highly Parallel processing
  - 64~320 processing units for vertex and/or pixel processing
- High level language
  - Direct3D 10
  - OpenGL 1.5 / 2.0

GPU in modern PCs



AGP/PCI Express Bus

- AGP bus
  - 1x/2x/4x/8x
  - 2.1 GB/s bandwidth with AGP 8x
    - Asymmetric (2GB/s for Download, 0.1GB/s for Upload)
  - Motherboard should support the expected speed
- PCI Express
  - 2x/4x/8x/16x
  - 2 x 4GB/s bandwidth with PCIE 16x

AGP/PCI Express Bus

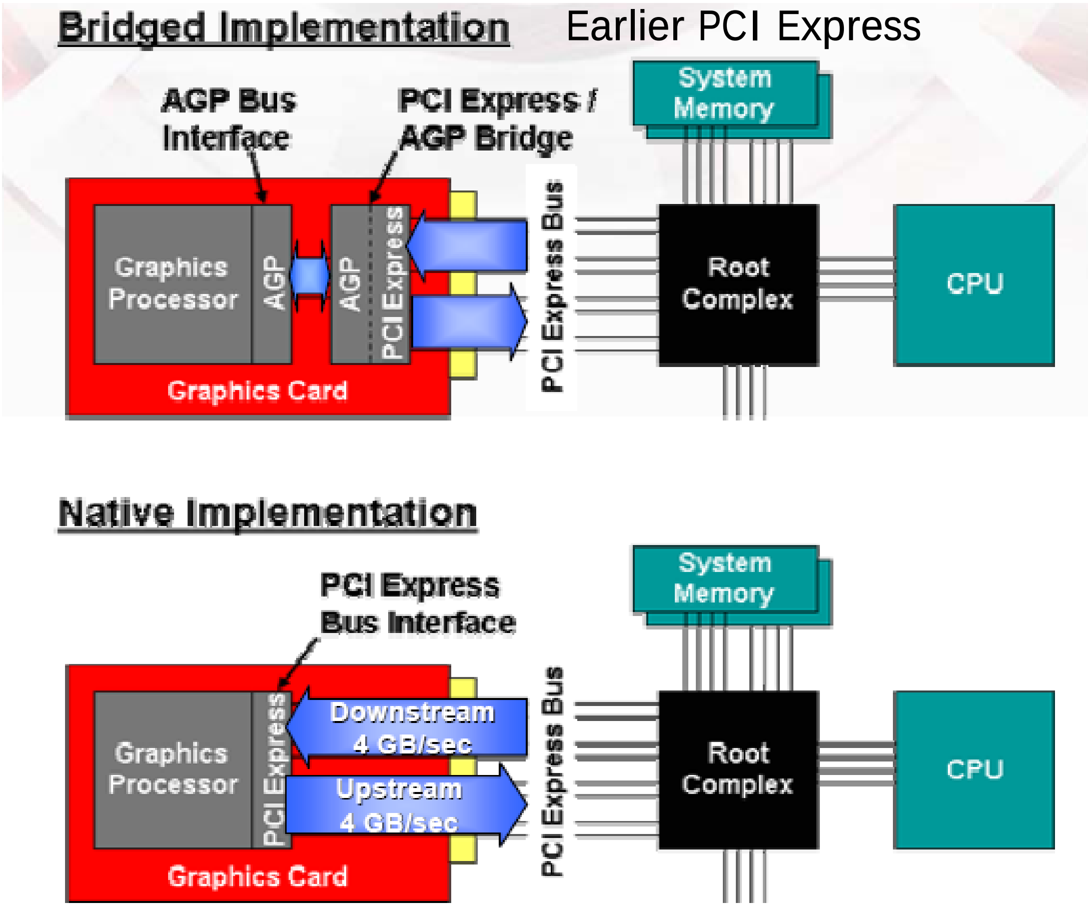
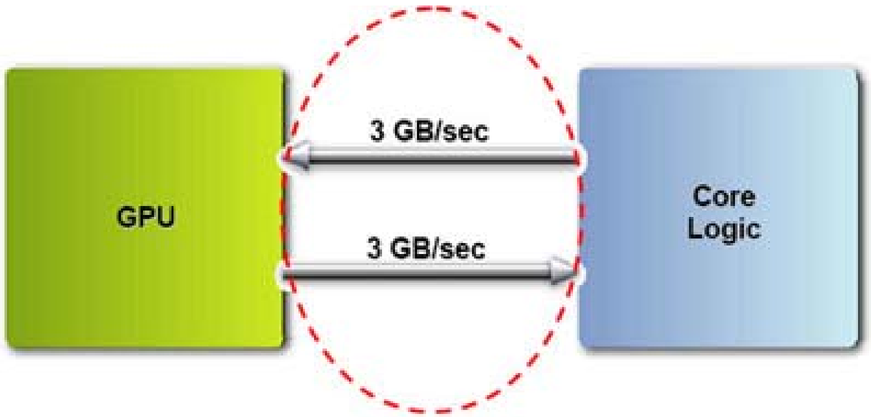
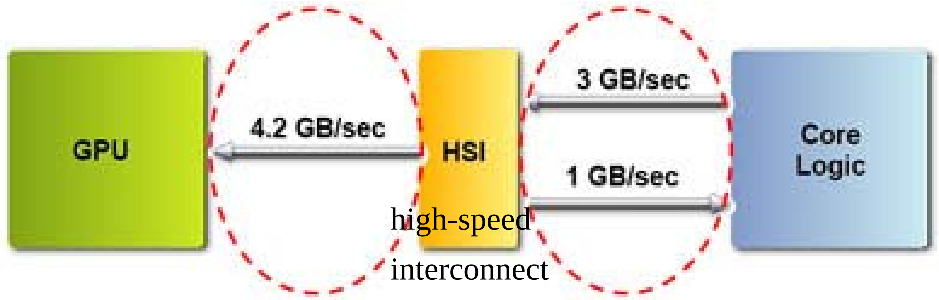


Figure 1: Comparison of Bridged and Native PCI Express Implementations

AGP/PCI Express Bus



- *Effective PCI Express Bandwidth*

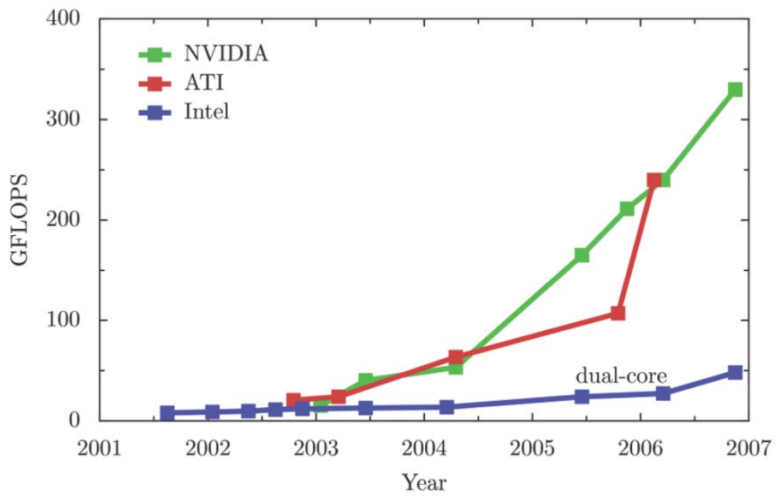


- *Typical PCI Express Usage, Per NVIDIA*

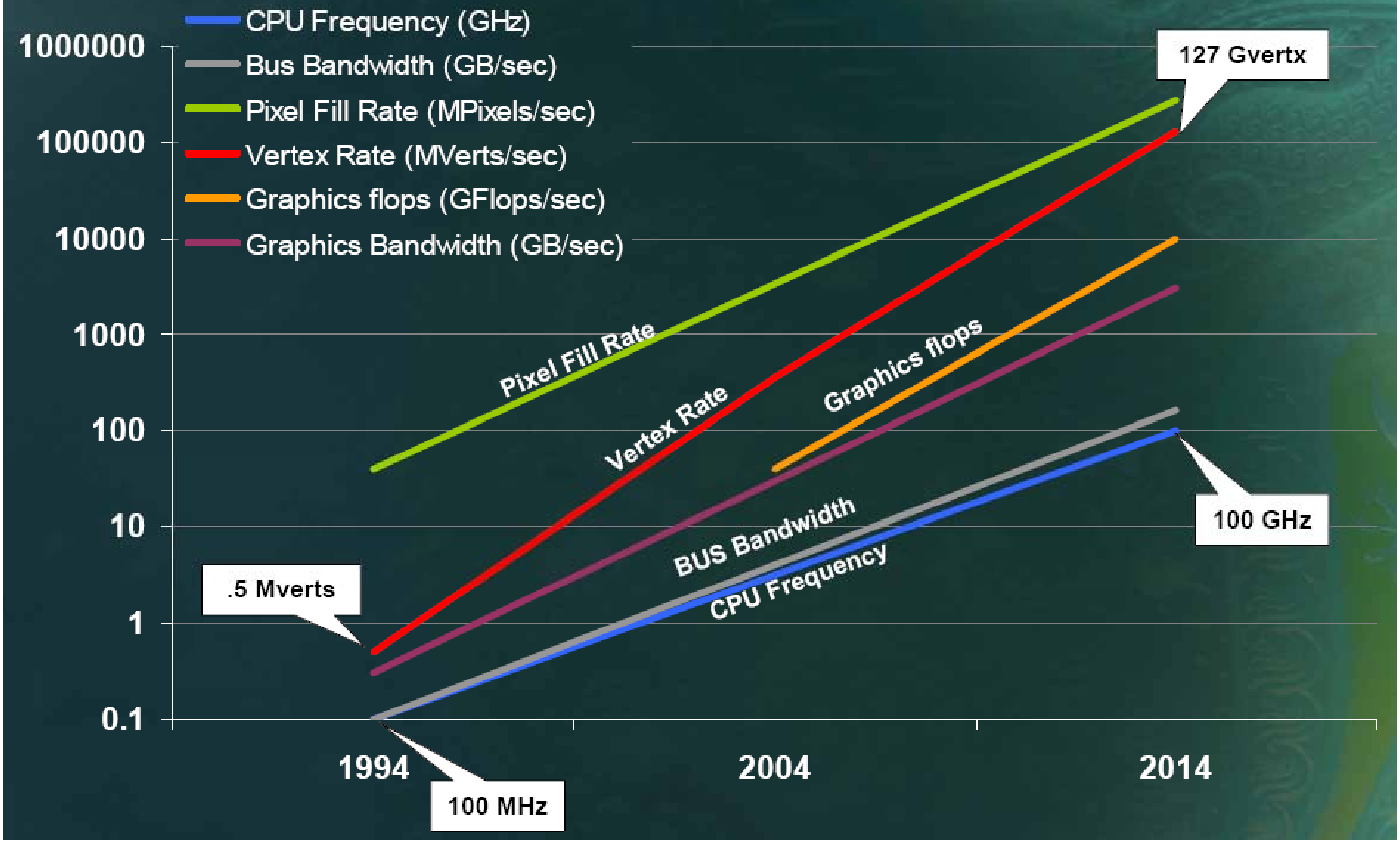
Computational Power

- GPUs are fast…
  - quad-core 3 GHz Intel Core 2 Extreme QX6850 *theoretical* : 38.6 GFlops, 8.5 GB/sec peak memory bandwidth
  - GeForce 8800GTX *observed* : 518 GFlopss, 86.4 GB/s peak memory bandwidth

- GPUs are getting faster
  - CPUs: annual growth ; 1.5× → decade growth : 60×
  - GPUs: annual growth > 2.0× → decade growth > 1000



Looking Ahead: Now + 10 years

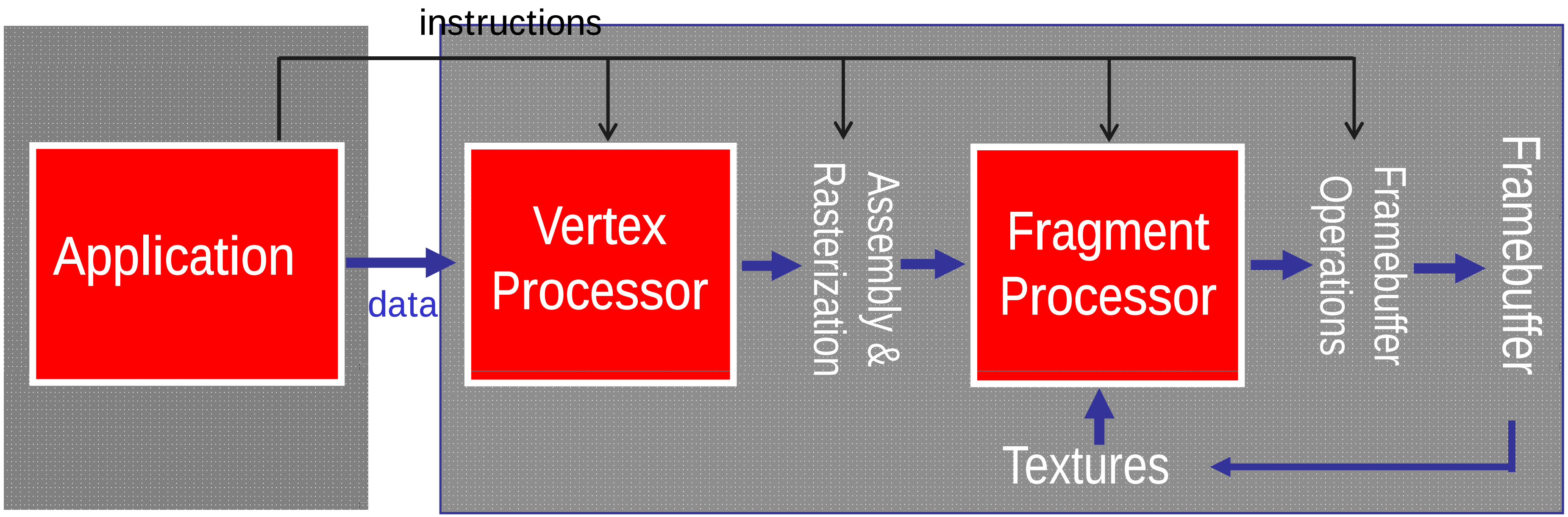


Performance 1994-2014

	1994	2004	2014
CPU Frequency (GHz)	.1	3.2	100
Memory Frequency (GHz)	.03	1.2	44
Bus Bandwidth (GB/sec)	.1	4	160
Hard Disk Size (GB)	.5	200	30 TB
Pixel Fill Rate (MPixels/sec)	.40	3300	270 GP
Vertex Rate (MVerts/sec)	.5	356	127 GV
Graphics Flops (GFlops/sec)	.001	40	10 TF
Graphics Bandwidth (GB/sec)	.3	30	3 TB
Frame Buffer Size (MB)	2	256	32 GB

GPU

1. GPU is a stream processor
   - Multiple programmable processing units
   - Connected by data flows



GPU

2. Greater variation in basic capabilities
   - Recent GPU support branching, but not perfect
     - Performance problem caused by pipeline stall
     - Limited capability
   - Vertex processors don’t support filtered texture mapping
     - Still slow
   - Some processors support additional texture types
     - In ATI, 3Dc which is an exciting new compression technology designed to bring out fine details in games while minimizing memory usage

GPU

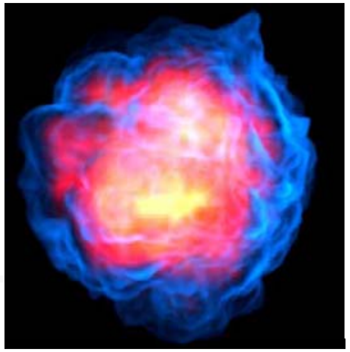
3. Optimized for 4-vector arithmetic
   - Useful for graphics – colors, vectors, texcoords
   - Easy way to get high performance/cost

- **Shading languages have vector data types and operations**
  e.g. Cg has float2, float3, float4
- **Obvious way to get high performance**
- **Other matrix data types**
  e.g. Cg has float3x3, float3x4, float4x4

Why GPU for Volume Rendering

- A massively parallel architecture
- A separation into two distinct units (vertex and fragment shader) that can double performance if the workload can be split
- Incredibly fast memory and memory interface
- Dedicated instructions for graphical tasks
- Vector operations on 4 floats that are as fast as scalar operations (intrinsic parallel processing)
- Trilinear interpolation is automatically (and extremely fast) implemented in the 3D-texture

Ray Casting with GPU



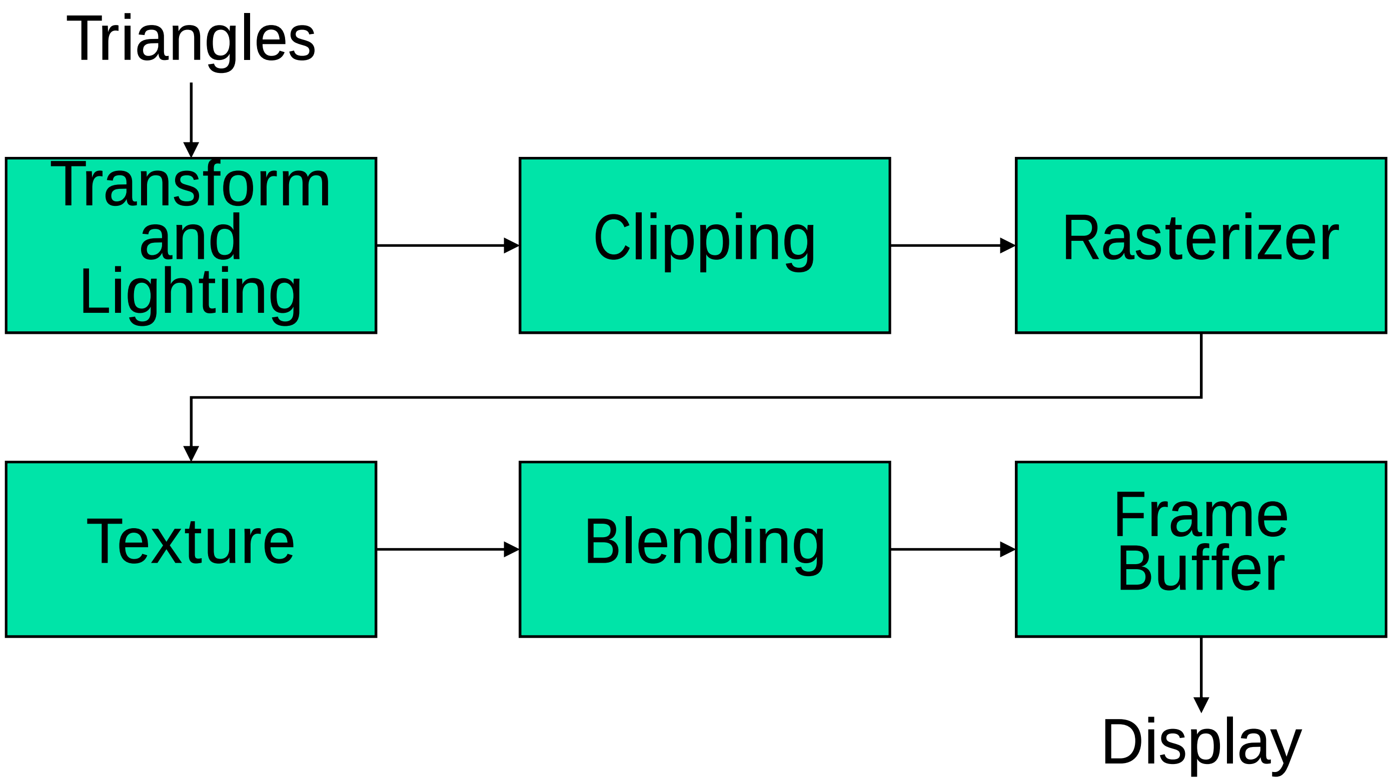
- Automatic calculation of ray positions by letting the hardware interpolate color values
- Built-In fast tri-linear interpolation of 3D Textures
- Full floating point compositing at almost no cost
- Changing from orthogonal to perspective projection without additional effort
- Automatic calculation of intersections in the depth buffer

Limitations and difficulties

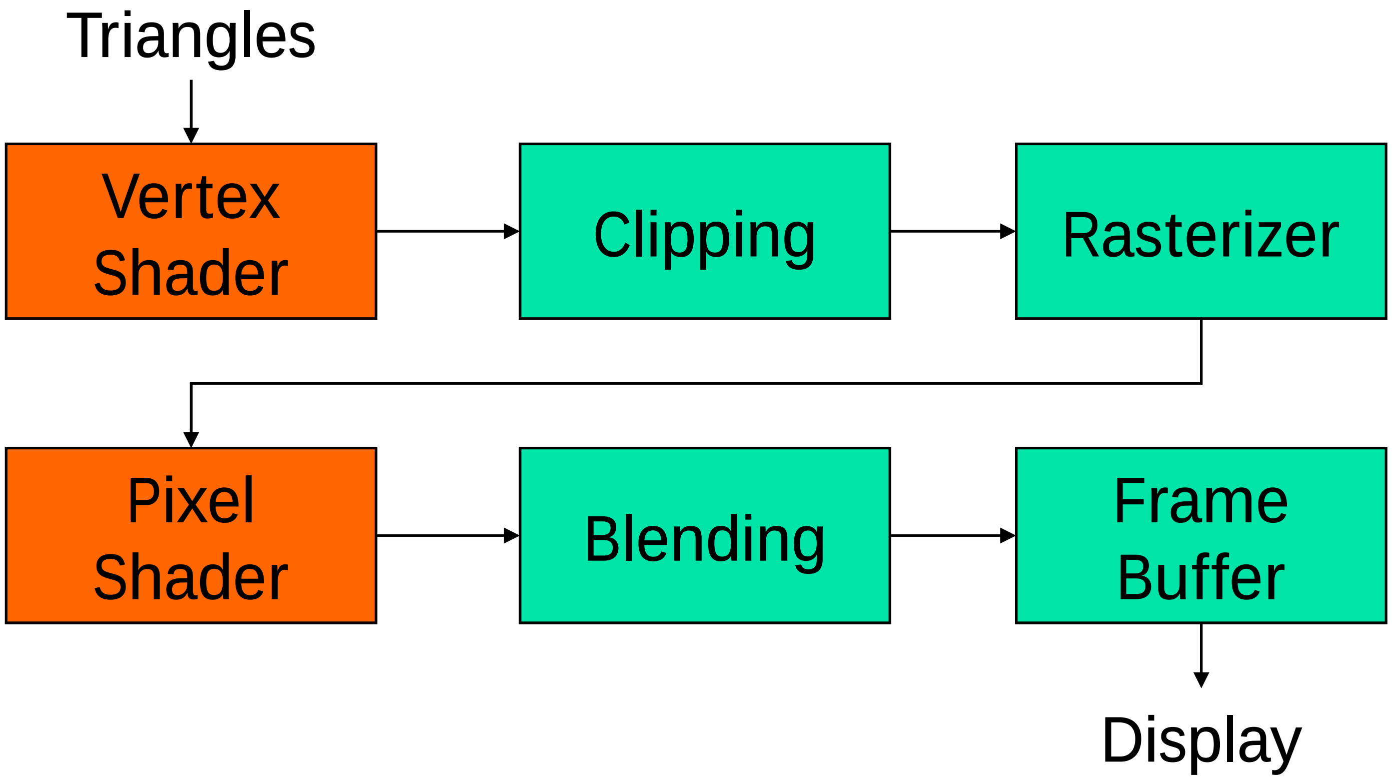
- Restriction of video memory size (upto 1GB)
- No full support of integer operations
- The lack of double precision
- Programmability still restricted in a number of ways, like limited loop count and limited conditional statements
- Readability of a GPU shader is still inferior to standard high-level languages
- Different vendors support different features and extensions, making it difficult to write an algorithm for every platform
- Choice of API may be more crucial than on the CPU (OpenGL or DirectX? Assembler fragment programs or high-level shading language? And if so, which shading language?)
- Unstable drivers, half-implemented features etc…
- Difficult to apply non-graphics tasks

Rendering Pipeline

Rendering Pipeline (fixed)



Rendering Pipeline (programmable)



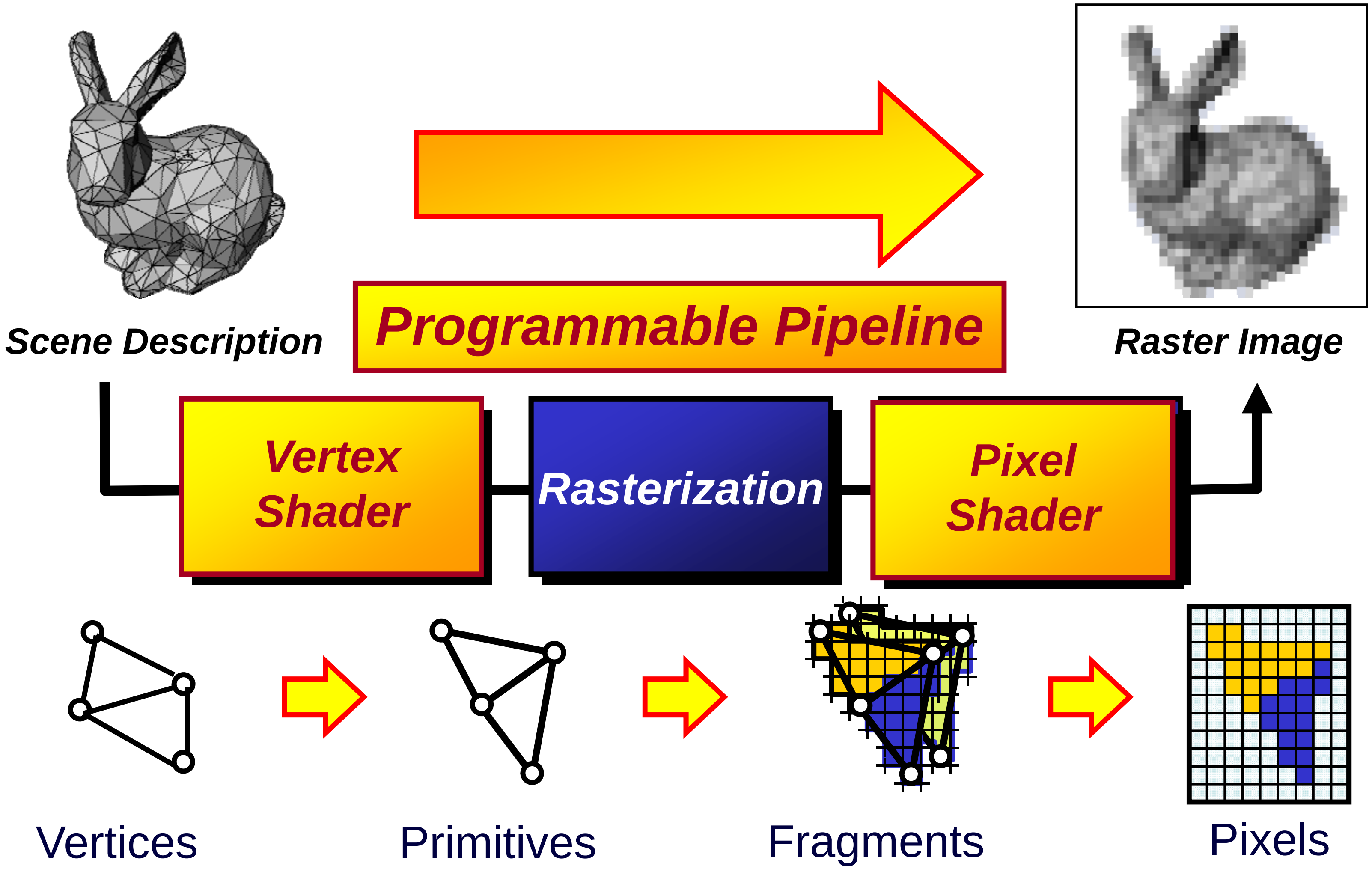
Data Flow in Streaming Architecture

1. Vertex Shader
   - Input: vertex attributes
     - position, color, normal vector, texture coordinates, etc.
   - Output: vertex attributes
     - transformed position, lit color, processed texture coordinates
2. Rasterization
   - Fragments are generated
   - Attributes are interpolated linearly

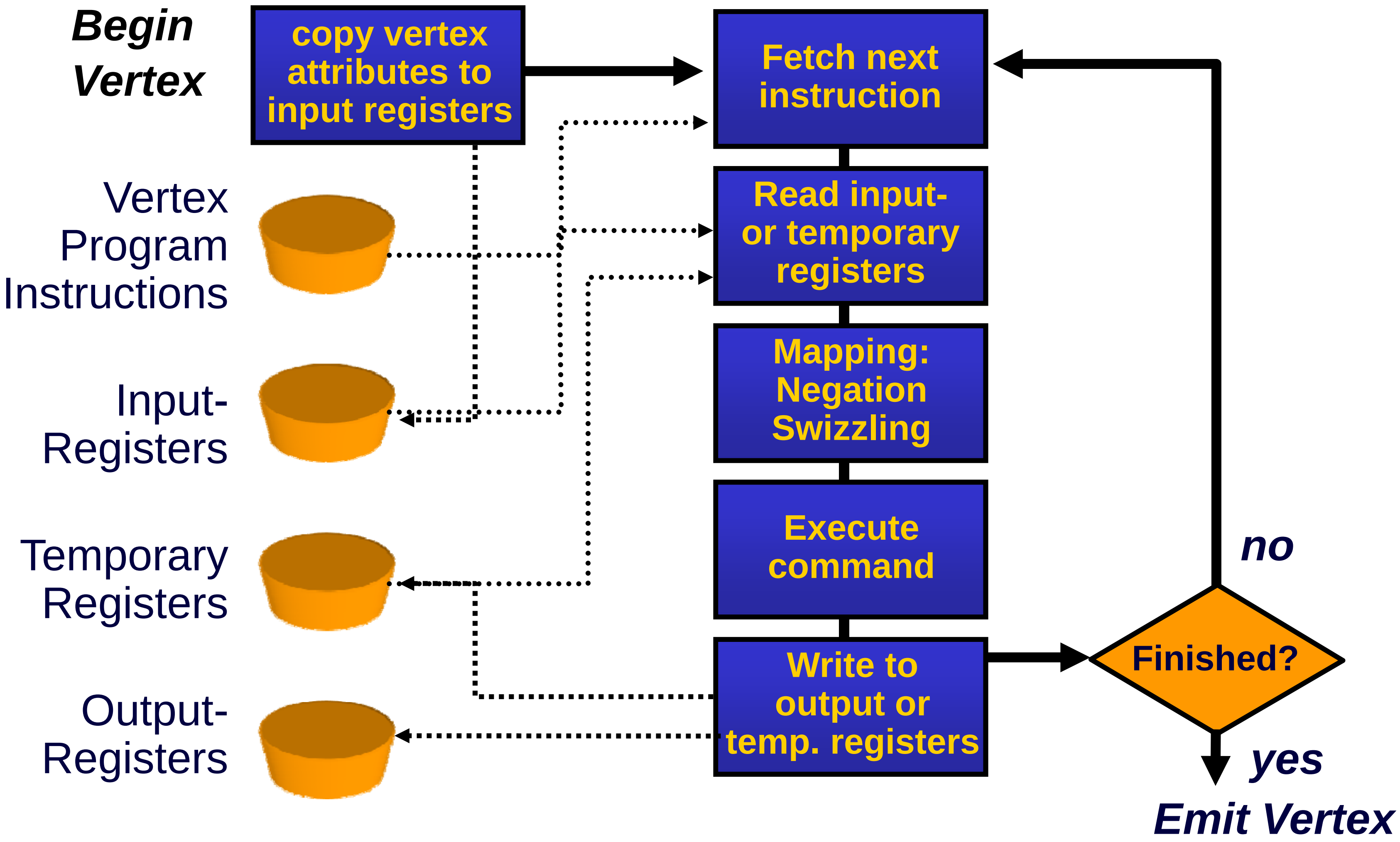
Data Flow in Streaming Architecture (cont.)

3. Pixel Shader
   - Input: fragment attributes
     - lit colors (diffuse&specular), texture coordinates (multiple sets)
   - Output: fragment attributes
     - final color (including alpha channel)
     - Any values can be written to texture memory with multiple target setting
4. Fragments tests and frame-buffer alpha blending

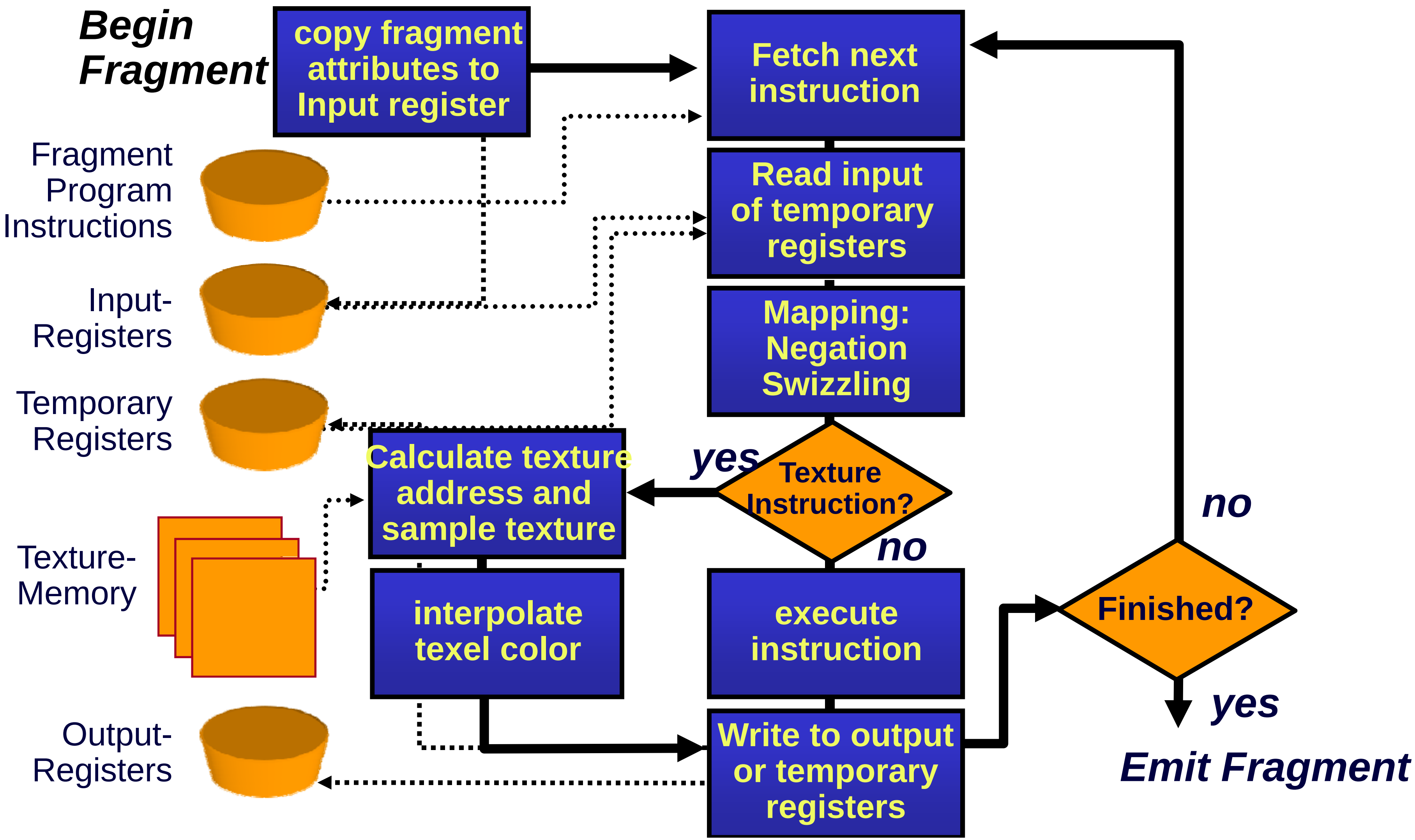
Graphics Hardware



Programmable Vertex Processor



Fragment Processor



Phong Shading

- *Per-Pixel Lighting:* Local illumination in a fragement shader

```
void main(          position   : TEXCOORD0, : per each fragment
                    normal     : TEXCOORD1,

                    oColor      : COLOR,

                    ambientCol,
                    lightCol,
                    lightPos,
                    eyePos,
                    Ka,
                    Kd,
                    Ks,
                    shiny)
{




          P = position.xyz;
          N = normal;
          V = normalize(eyePosition - P);
```

Programmable Shader

- Flexibility in rendering pipeline
- All advanced rendering techniques can be programmed
- Shader program cannot have global memory
  - Global constants can be fed thru constant registers
  - Interpolants can be fed thru texture addresses
  - Global vector data can be fed thru textures

32-bit IEEE floating-point throughout pipeline

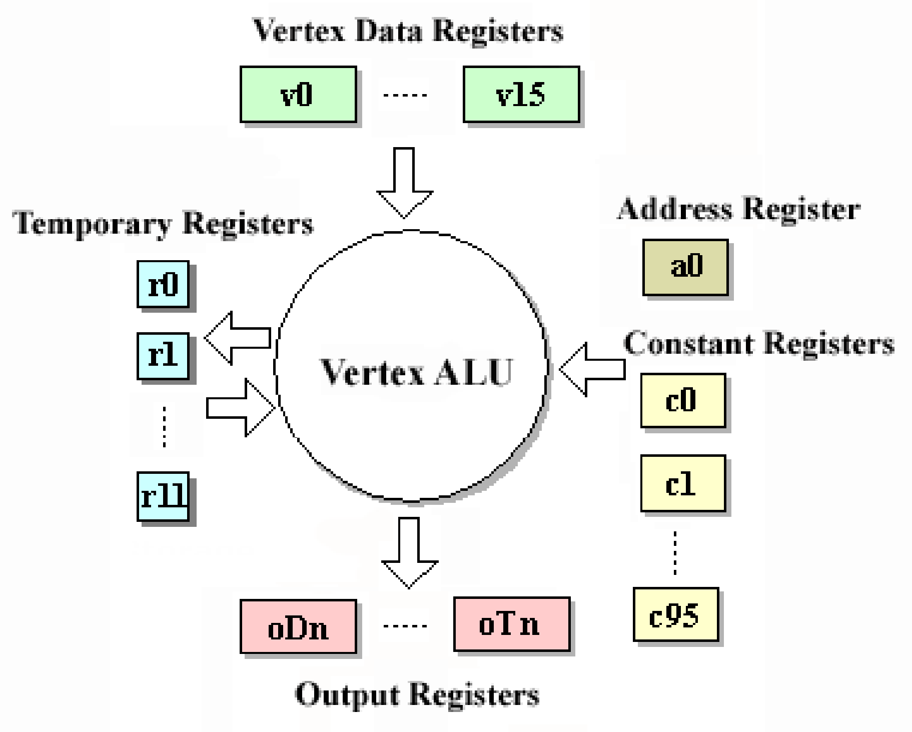
- Framebuffer
- Textures
- Fragment processor
- Vertex processor
- Interpolants

Vertex Shader

- Vertex shader or vertex program
  - Replaces fixed transformation and lighting engine to flexible one
  - Vertex can be animated
  - Current version: Shader Model 4 with Direct3D 10

Block Diagram of Vertex Shader 1.0

- Registers
  - v*: vertex stream data
  - r*: temporary register
  - c*: constant register
  - oD0, oD1, oFog, oPos, oPts, oT1-oT7: output registers



Vertex Shader 2.0

- 256 instructions with loop
- Registers
  - Constant registers: 16 boolean / 256 floating-point / 16 integer
  - 12 temporary floating point registers
  - 16 vertex data registers
  - 2 color output registers
  - 8 texture coordinate registers

Vertex Shader 2.0 (cont.)

- Instructions
  - add, dp3, dp4, dst, expp, lit, logp, mad, max, min, mov, mul, rcp, rsq, sge, slt, sub
- Macros
  - exp, frc, log, m3x2, m3x3, m3x4, m4x3, m4x4
- Modifiers
  - Destination mask: r.{x}{y}{z}{w}
  - Source swizzle: r.[xyzw] [xyzw] [xyzw] [xyzw]
  - Source negation: -r

Vertex Shader 2.0 capabilities

- 4-vector FP32bit operations, as in GeForce3/4
- True data-dependent control flow
  - Conditional branch instruction
  - Subroutine calls, up to 4 deep
  - Jump table (for switch statements)
- Conditional clause
  - No performance gain
- New arithmetic instructions (e.g. COS)
- User clip-plane support

Vertex Shader 3.0

- Branching and looping
  - Up to 24 dynamic flow controls
  - Causes drastic decline of performance
- Texture sampling w/o filtering
- 512 instructions per program (effectively much higher w/branching)
- 32 temporary 4-vector registers
- 256 “uniform” parameter registers
- 4 texture samplers
- 6 clip-distance outputs
- 16 per-vertex attributes (only)

Vertex Shader 4

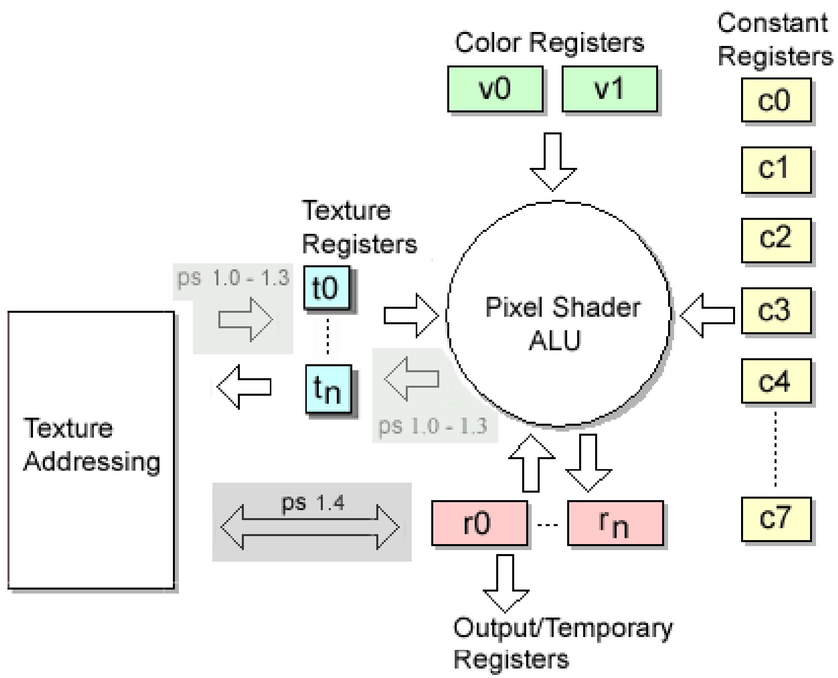
- More flexible branching / loop
- Supports filtered texture sampling
- Supports native integer type and boolean ops
- 4096 temporary 4-vector registers
- 16x4096 constant registers
- 128 texture samplers

Pixel Shader

- Pixel shader or fragment program
  - Replaces Texture engine
  - Complex per-pixel lighting
  - Flexible Operations with multiple textures
  - Flexible Texture coordinate manipulation
  - Current version: Shader Model 4 with Direct3D 10

Block Diagram of Pixel Shader 1.4

- Pixel-based processing
  - Lighting, texturing, etc.
- Registers
  - c*: constant register
  - r*: temporary register
  - t*: texture register
  - v*: color register



Pixel Shader 2.0

- 32 texture instructions *(no limit in 3.0)*
- 64 arithmetic instructions
- Instructions for vector processing and texture fetches
  - similar to vertex shader, but limited set of instructions
- Floating point registers: 32 constant and 12 temp
- Per-pixel shading
- Texture coordinate manipulation
- Operations with multiple textures

Pixel Shader 2.0

- Instructions
  - Arithmetic instructions
  - Texture instructions
- Modifier
  - Source selector: access each channel
  - Data modifier: bias, negate, invert, scalex2, signed scaling
  - Instruction modifier: _x2,_x4,_x8,_d2,_d4,_d8,_sat
  - +: co-issued instructions

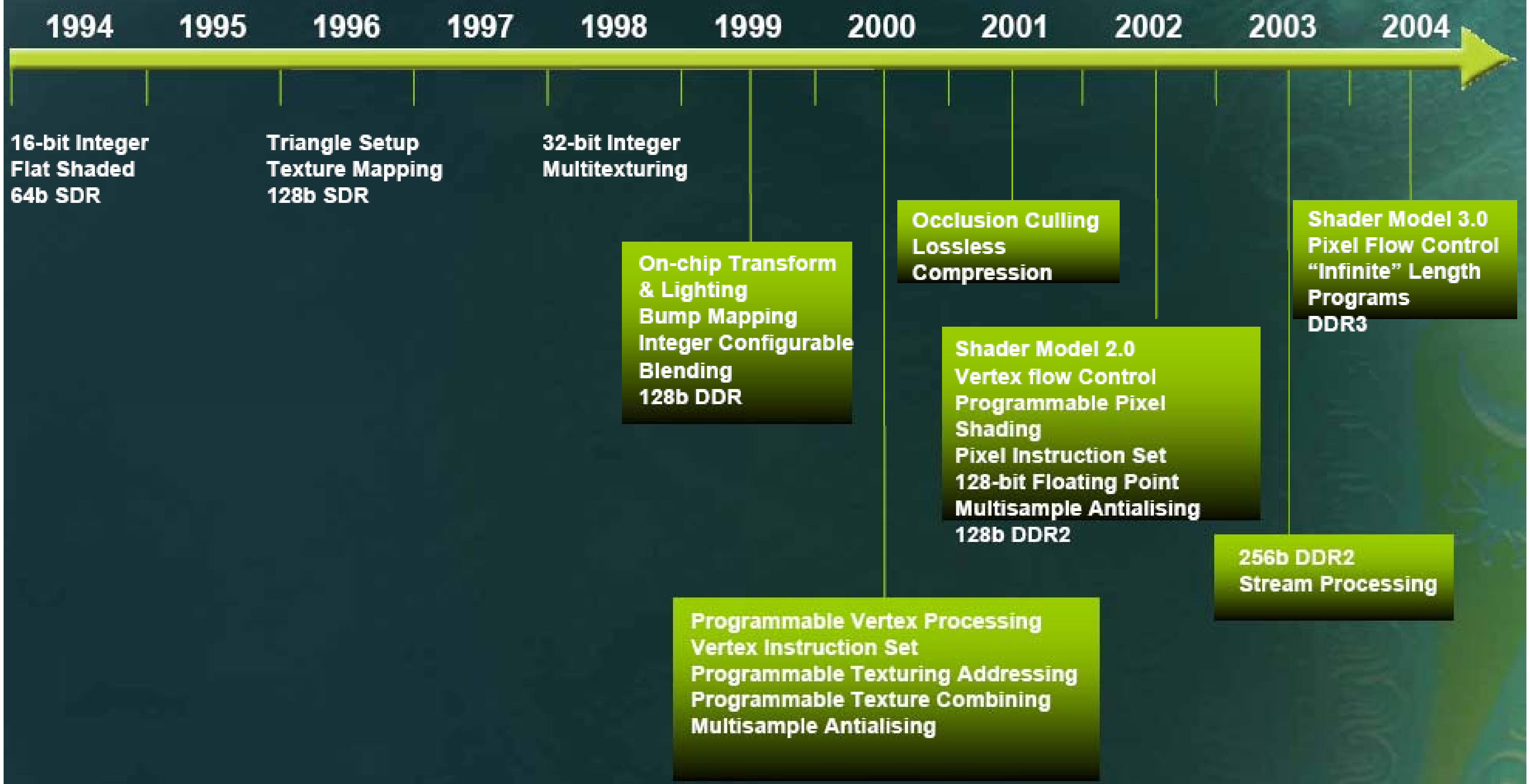
Pixel Shader 3.0

- Branching and looping
  - Up to 24 dynamic flow controls
  - Causes drastic decline of performance
- More than 512 instruction slots
- 32 temporary registers
- 224 constant registers
- 10 interpolated registers
- No indexed reads from registers
  - Use texture reads instead
- No CPU memory writes

Pixel Shader 4

- More than 64k instruction slots
- 4096 temporary registers
- Supports indexed data loading
- Supports native integer type
- 32 interpolated registers
- 16x4096 constant registers
- 8 FP32 x 4 perspective-correct inputs

Technology Shifts in Graphics



Complete Native Shader Model 3.0 Support

	DirectX 9.0	Shader Model 3.0
Vertex Shader Model	**2.0**	**3.0**
Vertex Shader Instructions	256	2^{16} (65,535)
Displacement Mapping	-	✓
Vertex Texture Fetch	-	✓
Geometry Instancing	-	✓
Dynamic Flow Control	-	✓
Pixel Shader Model	**2.0a**	**3.0**
Required Shader Precision	fp24	fp32
Pixel Shader Instructions	512	2^{16} (65,535)
Subroutines	-	✓
Loops & Branches	-	✓
Dynamic Flow Control	-	✓

High Level Shading Languages

High Level Shading Languages

- Assembly language is too difficult to program
- High level languages similar to C language
- Similar to general shading language like RenderMan
  - But this is for real time rendering
- Compiled for various back-ends
  - According to the hardware or rendering library
- Being developed now
  - Cg, HLSL, RenderMonkey, OpenGL 2.0, etc.

Design Goals of High Level Shading Languages

- High level enough to hide hardware specific details
- Simple enough for efficient code generation
- Familiar enough to reduce learning curve
- With enough optimizing back-ends for portability

CG

- C language for graphics
- By nVIDIA
- Similar syntax to C with many restrictions and exceptions
- Integrated with Cg SDK
- Supports various targets
  - GeForce series or DirectX versions
  - OpenGL

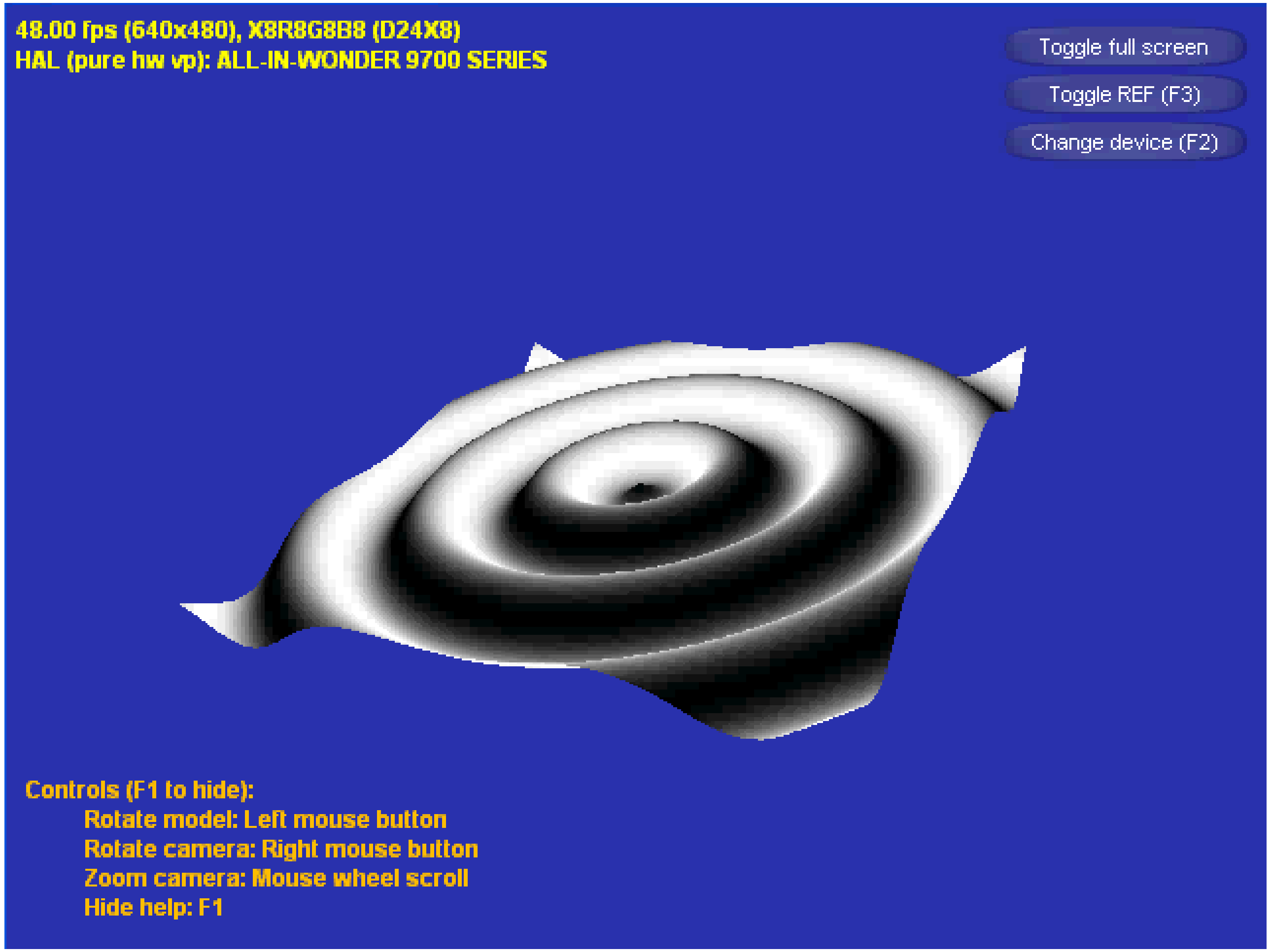
HLSL

- High level shading language
- By Microsoft
- Included in DirectX 9 spec and Visual Studio .NET
- Similar syntax to C with many restrictions and exceptions
- Not support OpenGL
- Compatible with Cg now
  - But in the future(?)

General Purpose Languages

- Microsoft Accelerator
  - Precompile general codes to shader codes
- Nvidia CUDA
- ATI CTM

HLSL Example



```
//-------------------------------------------------------
// This shader computes standard transform and lighting
//-------------------------------------------------------
VS_OUTPUT RenderSceneVS( float4 vPos : POSITION,
                         float3 vNormal : NORMAL,
                         float2 vTexCoord0 : TEXCOORD0,
                         uniform int nNumLights,
                         uniform bool bTexture,
                         uniform bool bAnimate )
{
VS_OUTPUT Output;
float3 vNormalWorldSpace;
float4 vAnimatedPos = vPos;

// Animation the vertex based on time and the vertex's object space position
if( bAnimate )
        vAnimatedPos += float4(vNormal, 0) * (si
(g_fTime+5.5)+0.5)*5;

// Transform the position from object space to homogeneous projection space
Output.Position = mul(vAnimatedPos, g_mWorldViewProjection);
// Transform the normal from object space t
vNormalWorldSpace = normalize(mul(vNorm

// Compute simple directional lighting equat
float3 vTotalLightDiffuse = float3(0,0,0);
for(int i=0; i<nNumLights; i++ )
    vTotalLightDiffuse += g_LightDiffuse[i] *

Output.Diffuse.rgb = g_MaterialDiffuseColor
                     g_MaterialAmbientColor * g_l
Output.Diffuse.a = 1.0f;

// Just copy the texture coordinate through
if( bTexture )
    Output.TextureUV = vTexCoord0;
else
    Output.TextureUV = 0;

return Output;
}
```