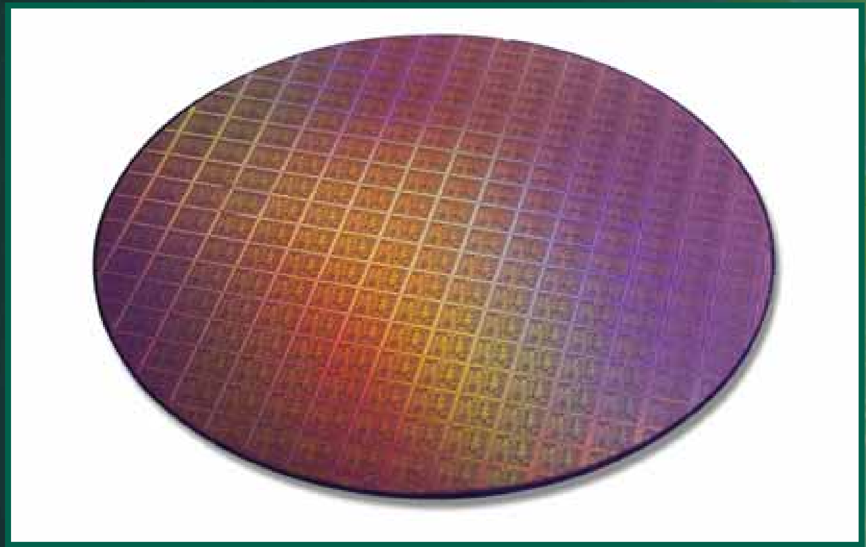

Shader Model 3.0
Ashu Rege
NVIDIA Developer Technology Group

Talk Outline

- Quick Intro - GeForce 6 Series (NV4X family)
- New Vertex Shader Features
  - Vertex Texture Fetch
  - Longer Programs and Dynamic Flow Control
  - Vertex Frequency Stream Divider (Instancing)
- New Pixel Shader Features
  - Longer Programs and Dynamic Flow Control
  - Multiple Render Targets
- Floating-Point Blending and Filtering
- Final Thoughts

GeForce 6 Series

- ***Shader Model 3.0 at all price points***
  - **Full support for shader model 3.0**
  - **Vertex Texture Fetch / Long programs / Pixel Shader flow control**
  - **Full speed fp32 shading**
- OpenEXR High Dynamic Range Rendering
  - **Floating point frame buffer blending**
  - **Floating point texture filtering**
  - **Except 6200**
- 6800 Ultra/GT specs
  - **222M xtors / 0.13um**
  - **6 vertex units / 16 pixel pipelines**
- PCI Express and AGP



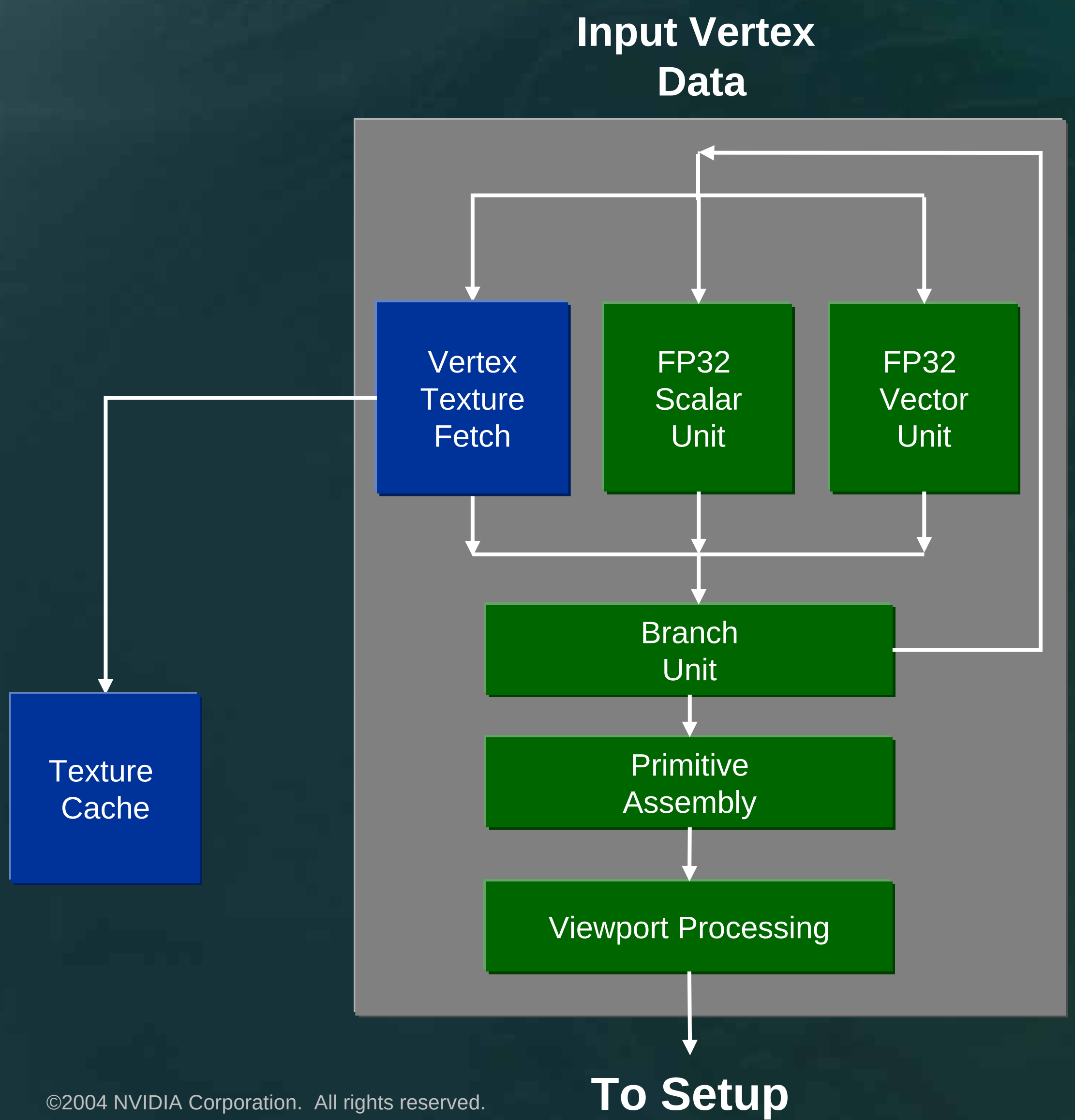
GeForce 6800 – (NV40)

Complete Native Shader Model 3.0 Support

	Shader Model 2.0	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	**2.0**	**3.0**
Required Shader Precision	fp24	fp32
Pixel Shader Instructions	96	2^{16} (65,535)
Subroutines	-	✓
Loops & Branches	-	✓
Dynamic Flow Control	-	✓

Vertex Shader 3.0

Detail of a Single Vertex Shader Pipeline



Vertex Shader Version Summary

	2.0	2.0a	3.0
# of instruction slots	256	256	>= 512
Max # of instructions executed	65535	65535	2^{16} (65,535)
Instruction Predication	-	✓	✓
Temp Registers	12	13	32
# constant registers	>= 256	>= 256	>= 256
Static Flow Control	✓	✓	✓
Dynamic Flow Control	-	✓	✓
Dynamic Flow Control depth	-	24	24
Vertex Texture Fetch	-	-	✓
# of texture samplers	-	-	4
Geometry Instancing Support	-	-	✓

Note: There is no vertex shader 2.0b

Flow Control: Static vs. Dynamic

```
void Shader(
    ...
        // Input per vertex or per pixel
       in float3 normal,

        // Input per batch of triangles
  uniform float3 lightDirection,
  uniform bool computeLight,

    ...
)
{
  ...
  if (computeLight) {
    ...
    if (dot(lightDirection, normal)) {
      ...
    }
    ...
  }
  ...
}
```

Static Flow Control
(condition constant for each batch of triangles)

Dynamic Flow Control
(data dependent, so condition can vary per vertex or pixel)

Static v. Dynamic Flow Control

- Static Flow Control
  - Based on ‘uniform’ variables, a.k.a. constants
  - Same code executed for every vertex in draw call
- Dynamic Flow Control
  - Based on per-vertex attributes
  - Each vertex can take a different code path

Using Flow Control

- Subroutines, loops, and conditionals simplify programming
  - [if, else, endif] [loop, endloop] [rep, endrep]
  - call, callnz, ret
  - Conditionals can be nested
  - Fewer vertex shaders to manage
- Dynamic branches only have ~2 cycle overhead
  - Even if vertices take different branches
  - Use this to avoid unnecessary vertex work (e.g., skinning, N.L<0, ...)
  - If you can branch to skip more than 2 cycles of work, do it!

Geometry Instancing

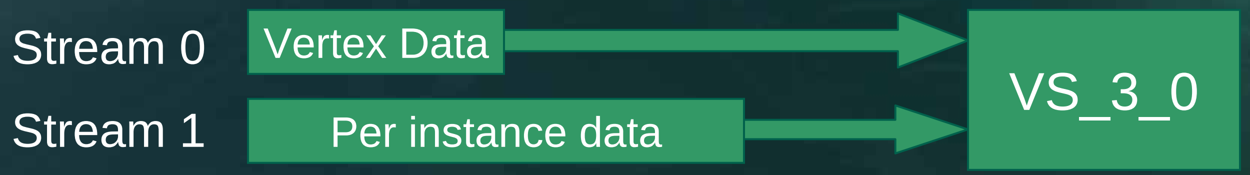
DirectX 9 Instancing

- What is instancing?
  - Allows a single draw call to draw multiple instances of the same model
  - Allows you to minimize draw primitive calls and reduce CPU overhead
- What is required to use it?
  - Microsoft DirectX 9.0c
  - VS 3.0 hardware
  - API is layered on top of **IDirect3DDevice9::SetStreamSourceFreq**

Why Use Instancing?

- Speed
  - Single biggest perf sink is # of draw calls
- We all know draw calls are bad
  - But world matrices and other state changes force us to make multiple draw calls
- Instancing API pushes per instance draws down to hardware/driver
  - Eliminates API and driver overhead

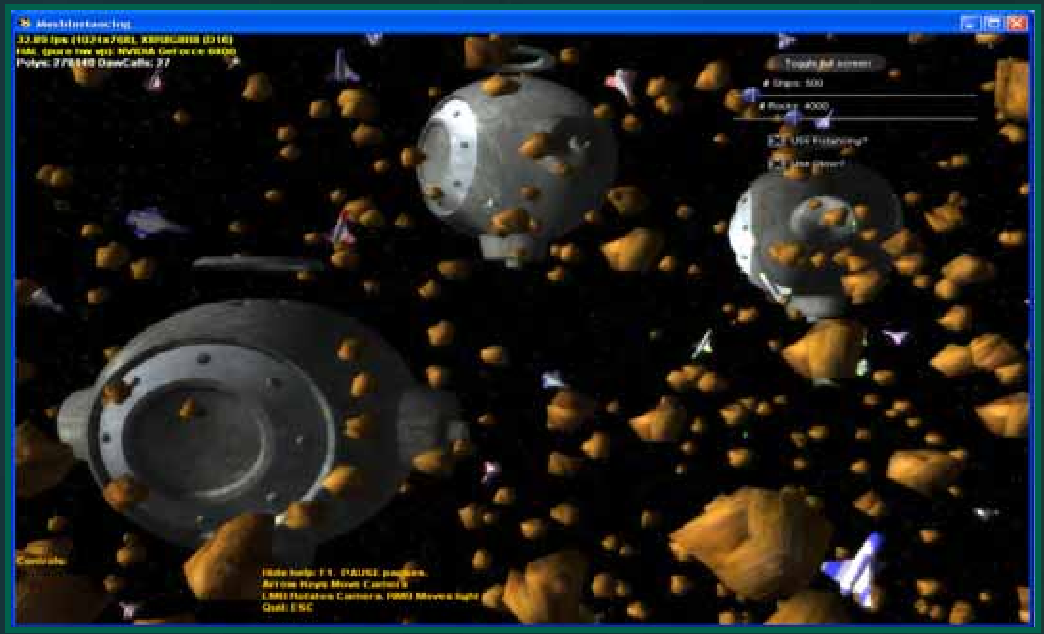
How does it work?



- Primary stream is a single copy of the model geometry
- Secondary stream(s) contain per-instance data
  - Transform matrices, colors, texture indices
  - Vertex shader does matrix transformations based on vertex attributes
  - Pointer is advanced each time an instance of the primary stream is rendered.

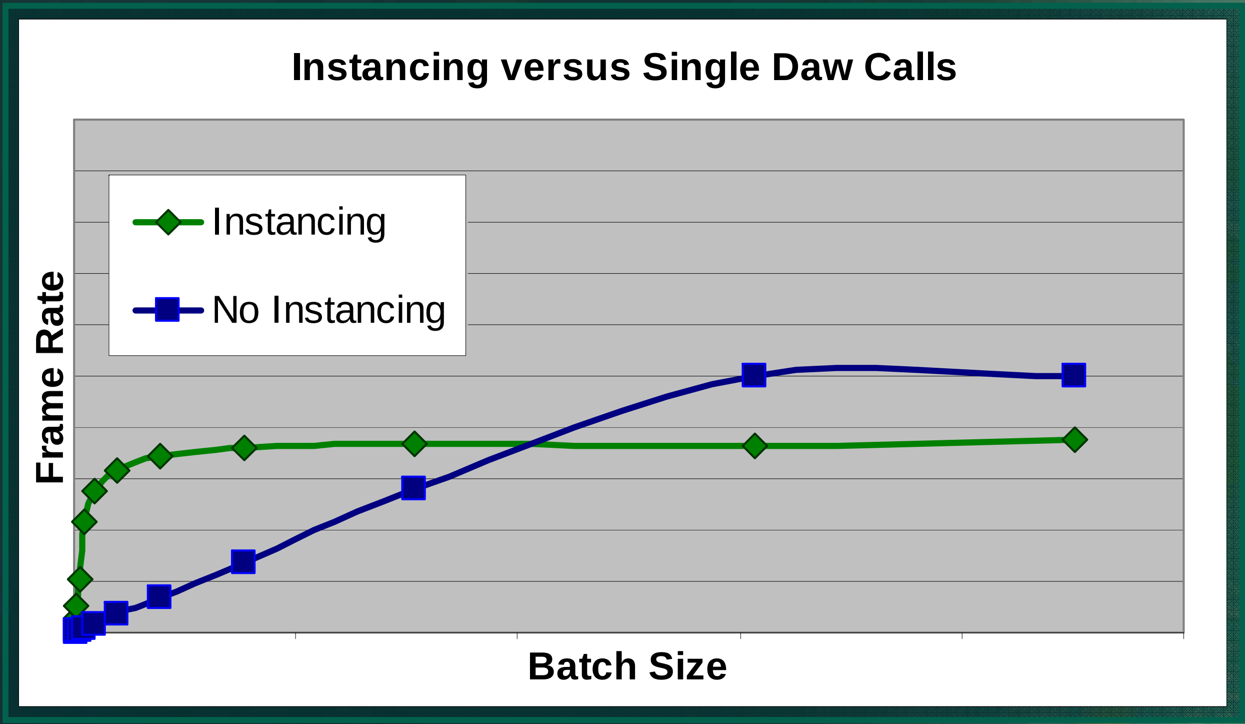
Instancing Demo

- Space scene with 500+ ships, 4000+ rocks
- Complex lighting, post-processing
  - Some simple CPU collision work as well
- Dramatically faster with instancing



Some Test Results

- Test scene draws 1 million diffuse shaded polygons
- Changing the batch size changes # of drawn instances
- For small batch sizes, can provide extreme win due to PER DRAW CALL savings
- There is a fixed overhead from adding the extra data into the vertex stream
- Sweet spot depends on many factors (CPU/GPU speed, engine overhead, etc.)

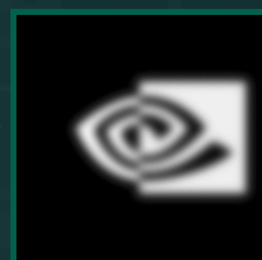


When To Use Instancing

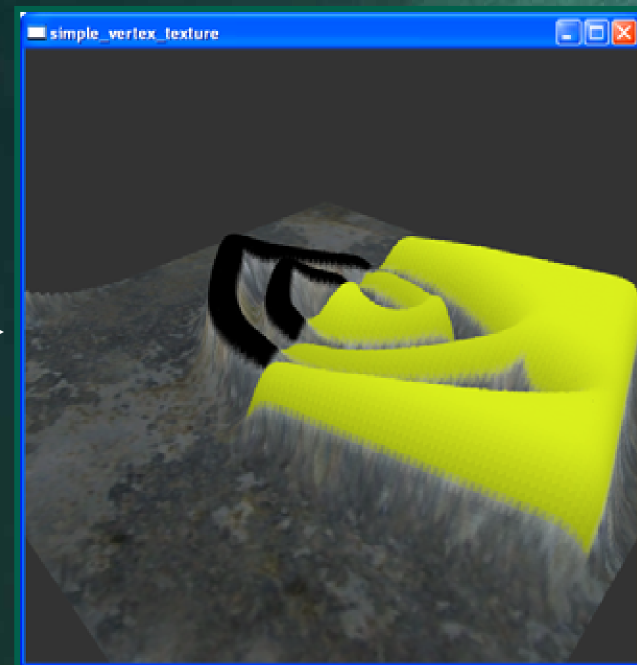
- Many instances of the same model
  - Forest of trees, particle systems, sprites
- Can encode per instance data in aux stream
  - Colors, texture coordinates, per-instance constants
- Not as useful is batching overhead is low
  - Fixed overhead to instancing

Vertex Texture Fetch

An Example of Vertex Texturing: Displacement Mapping



Displacement Texture



Flat Tessellated Mesh

Displaced Mesh

Vertex Texture Examples

Without Vertex Textures

With Vertex Textures

Images used with permission from *Pacific Fighters*. © 2004 Developed by 1C:Maddox Games. All rights reserved. © 2004 Ubi Soft Entertainment.

More Vertex Texture Examples

Without Vertex Textures

With Vertex Textures

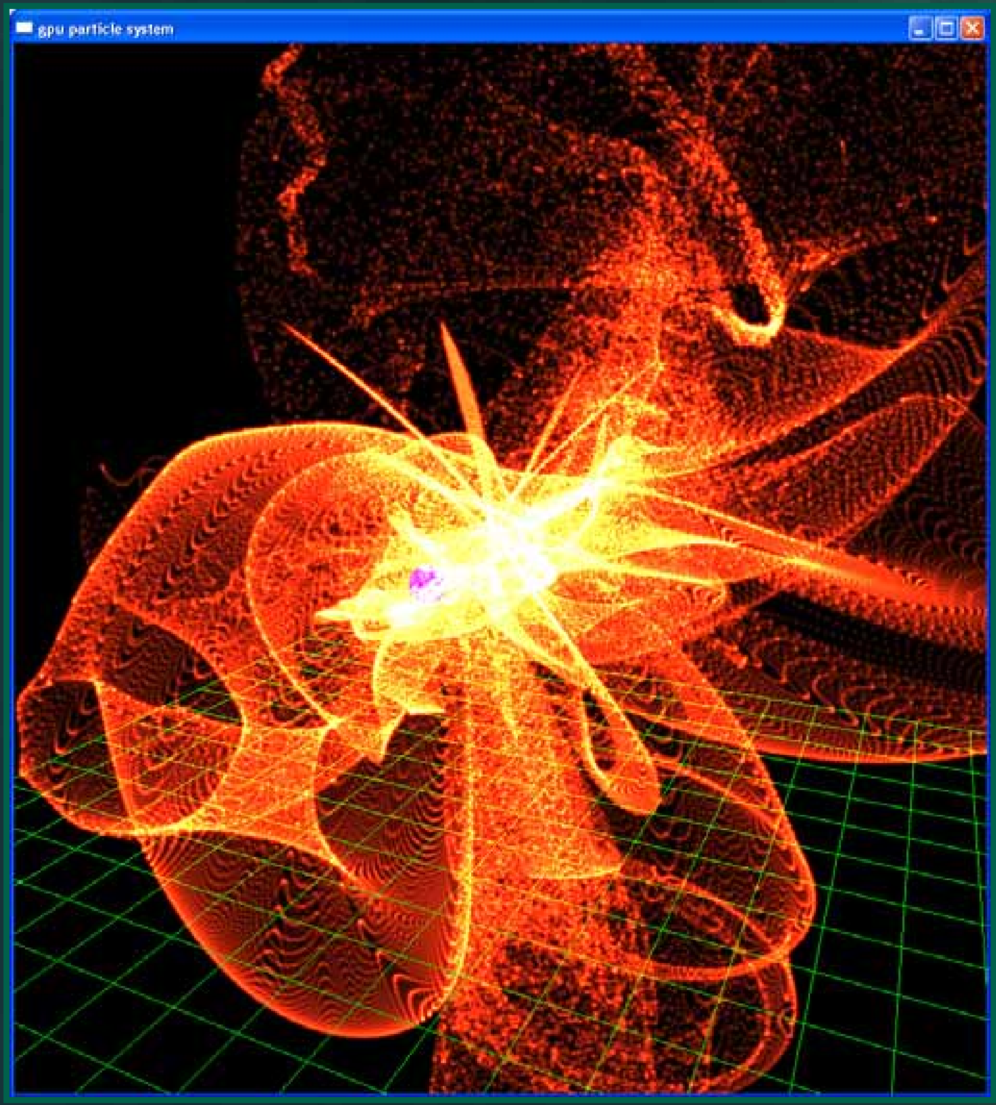
Vertex Texture

- Multiple vertex texture units
  - DX9: 4 samplers (D3DVERTEXTEXTURESAMPLER*n*)
  - **`OGL: glGetIntegerv(MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB)`**
  - 4 units on GeForce 6 Series hardware
- Supports point filtering only (currently)
- Supports mipmapping
  - Need to calculate LOD yourself
- Uses standard 2D texture samplers
- DX9: R32F and R32G32B32A32F formats
- OGL: **`LUMINANCE_FLOAT32_ATI`** or **`RGBA_FLOAT32_ATI`** formats
- Arbitrary number of fetches

Vertex Texture Applications

- Simple displacement mapping
  - Note – not adaptive displacement mapping
    - Hardware doesn't tessellate for you
  - Terrain, ocean surfaces
- Render to vertex texture
  - Provides feedback path from fragment program to vertex program
- Particle systems
  - Calculate particle positions using fragment program, read positions from texture in vertex program, render as points
- Character animation
  - Can do arbitrarily complex character animation using fragment programs, read final result as vertex texture
  - Not limited by vertex attributes – can use lots of bones, lots of blend shapes

GPU Particle System



Pixel Shader 3.0

Pixel Shader Version Summary

	2.0	2.0a	2.0b	3.0
Dependent Texture Limit	4	No limit	4	No limit
Texture Instruction Limit	32	unlimited	unlimited	unlimited
Position Register	-	-	-	✓
Instruction Slots	32 + 64	512	512	>= 512
Executed Instructions	32 + 64	512	512	2^{16} (65,535)
Interpolated Registers	2 + 8	2 + 8	2 + 8	10
Instruction Predication	-	✓	-	✓
Indexed Input Registers	-	-	-	✓
Temp Registers	12	22	32	32
Constant Registers	32	32	32	224
Arbitrary Swizzling	-	✓	-	✓
Gradient Instructions	-	✓	-	✓
Loop Count Register	-	-	-	✓
Face Register (2-sided lighting)	-	-	-	✓
Dynamic Flow Control Depth	-	-	-	24

PS3.0 Branching Performance

- Static branching is fast
  - But still may not be worth it for short branches (less than ~5 instructions)
  - Can use conditional execution instead
- Divergent (data-dependent) branching is more expensive
  - Depends on which pixels take which branches

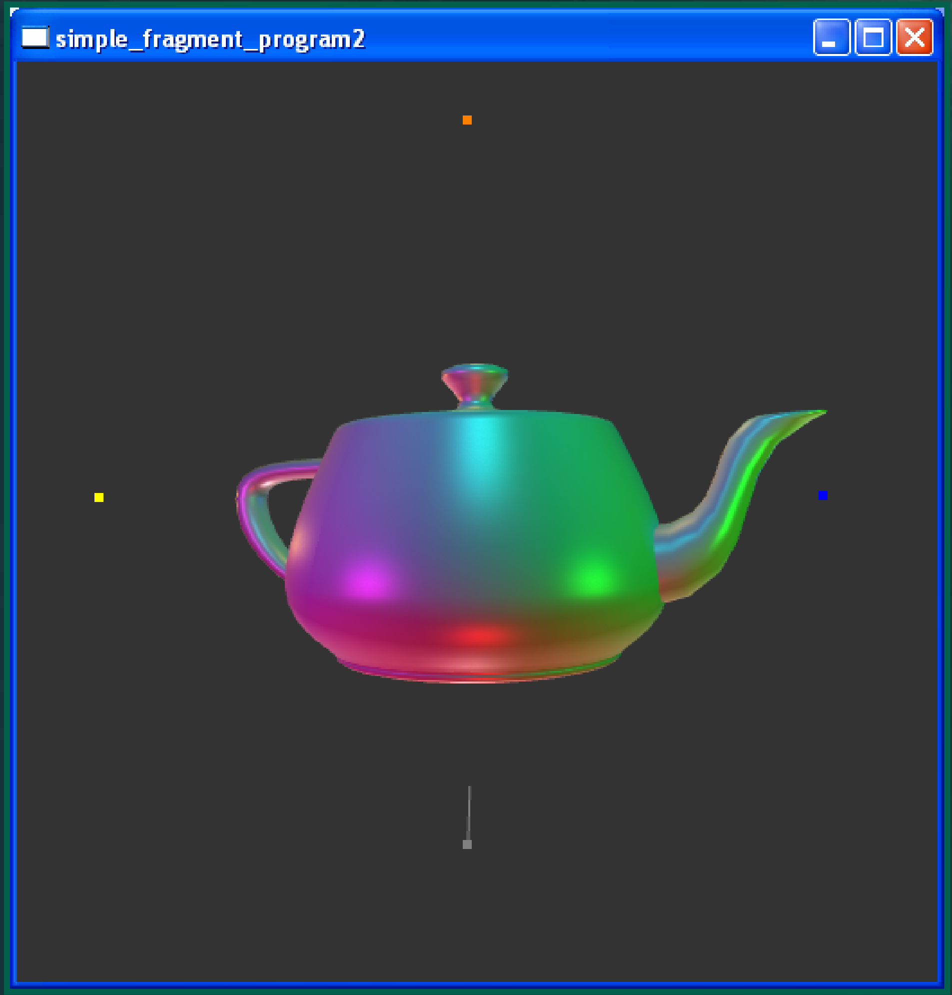
Branch Overhead

- Pixel shader flow control instruction costs:

Instruction	Cost (Cycles)
if / endif	4
if / else / endif	6
call	2
ret	2
loop / endloop	4

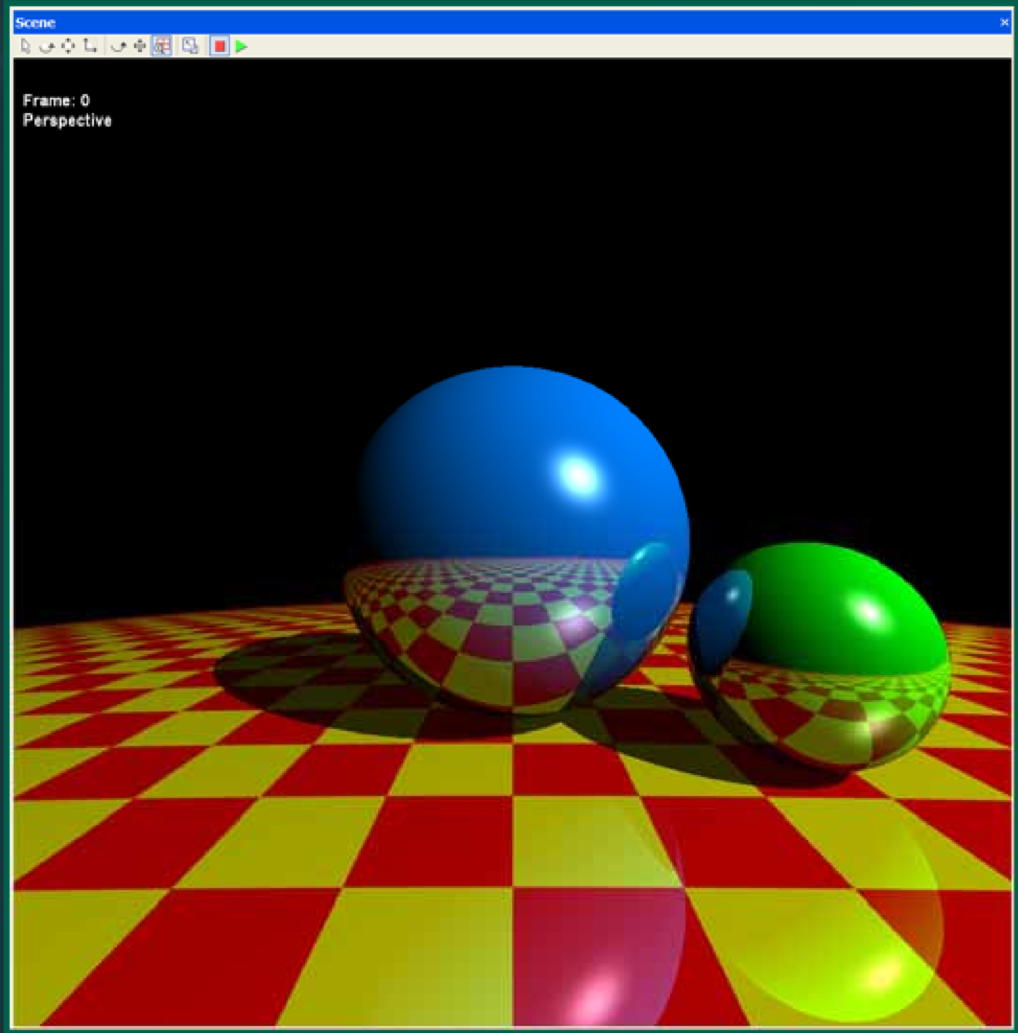
- Not free, but certainly usable and can save a ton of work!

Multiple Lights Demo



Available at http://developer.nvidia.com/object/sdk_samples.html

Pixel Shader Ray Tracer

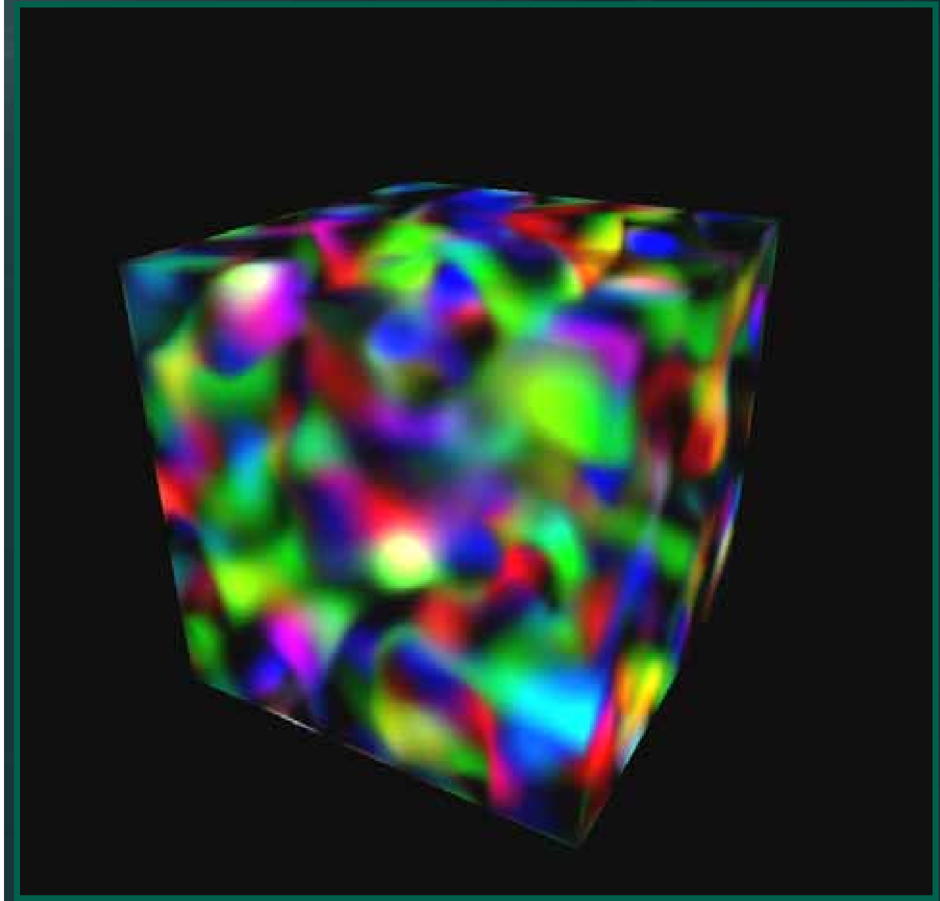


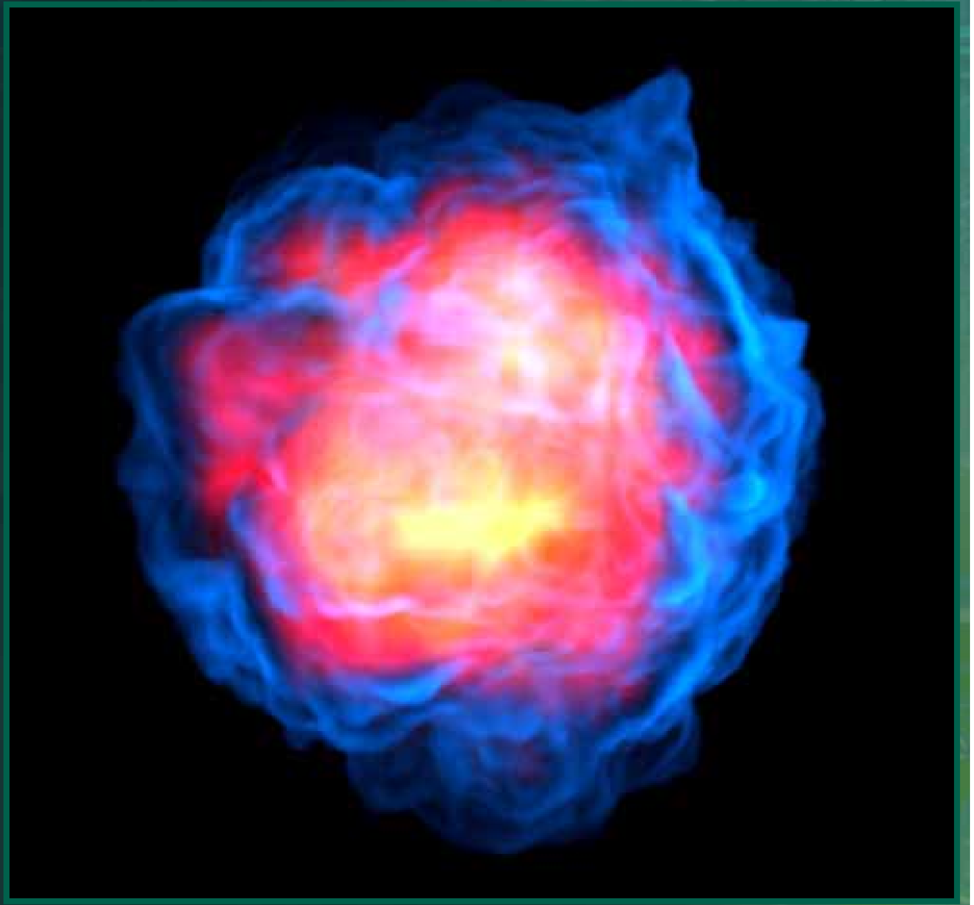
Available at http://developer.nvidia.com/object/sdk_effects.html

Pixel Shader Looping Example - Single Pass Volume Rendering

- Application only renders a single quad
- Pixel shader calculates intersection between view ray and bounding box, discards pixels outside
- Marches along ray between far and near intersection points, accumulating color and opacity
  - Looks up in 3D texture, or evaluates procedural function at each sample
- Compiles to REP/ENDREP loop
  - Allows us to exceed the 512 instruction PS2.0 limit
  - All blending is done at fp32 precision in the shader
  - 100 steps is interactive on 6800 Ultra

1 Pass Volume Rendering Examples





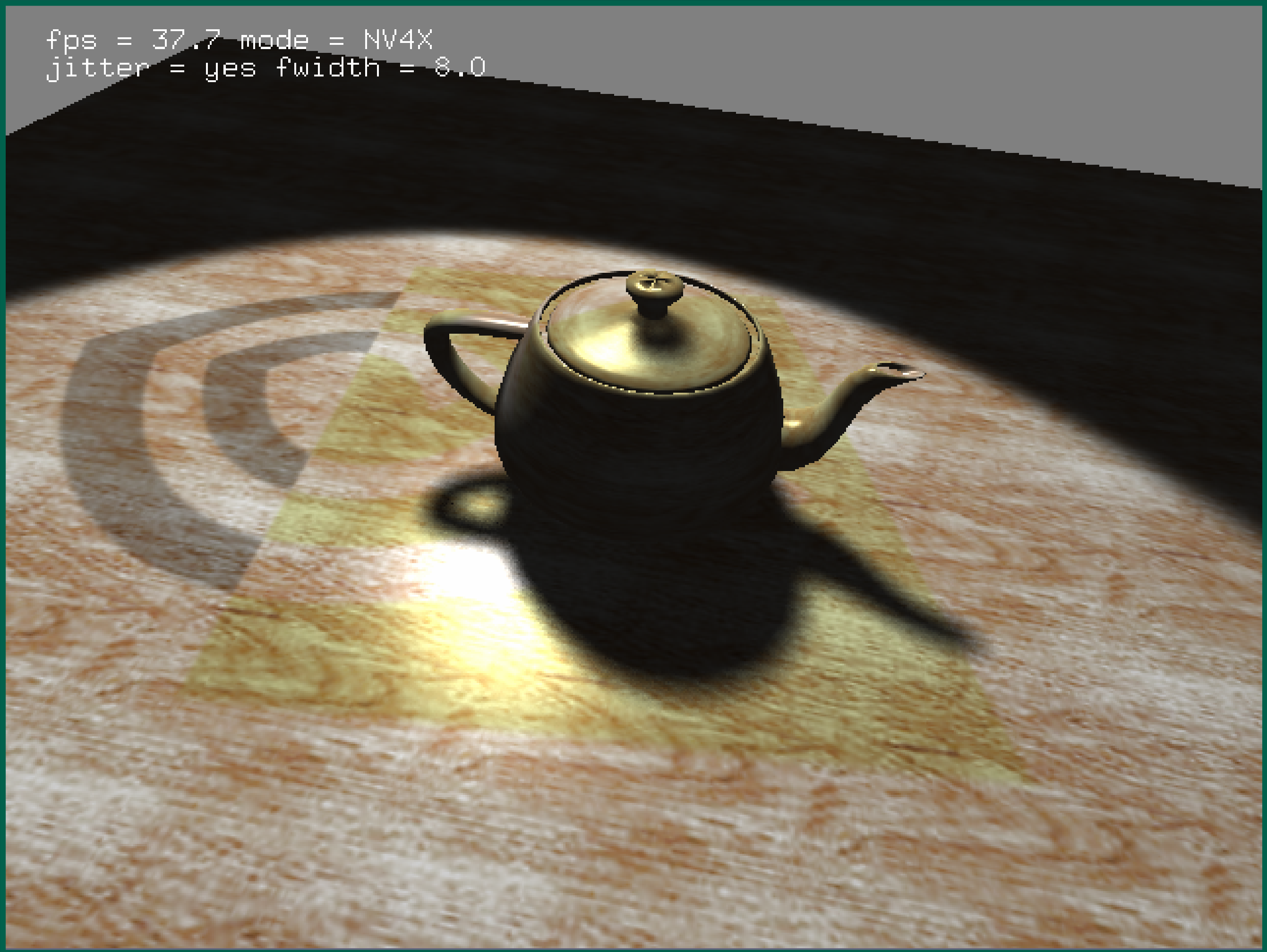
Extra Full Precision Interpolators

- 10 full precision interpolators (texcoords)
  - Compared to 8 in earlier pixel shader versions
- More inputs for lighting parameters, ...
- Multiple lights in one long shader
  - Compared to re-rendering for each light
  - Doesn’t work well with stencil shadows

Early Outs

- “Early out” is a dynamic branch in the shader to bypass computation
- Some obvious examples:
  - If in shadow, don’t do lighting computations
  - If out of range (attenuation zero), don’t light
  - These apply to vs.3.0 as well
- Next – a novel example for soft-edged shadows

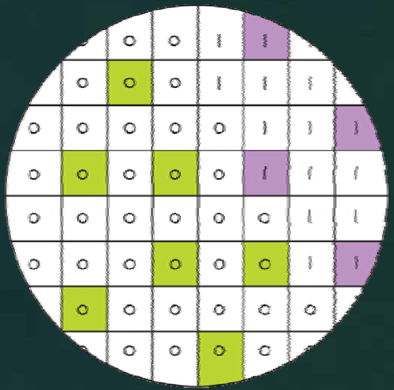
Soft-Edged Shadows with ps 3.0

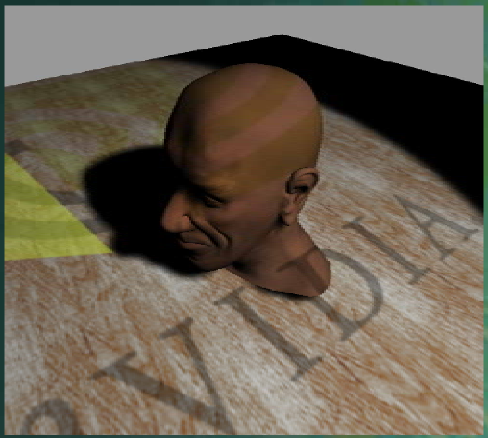


Available at http://developer.nvidia.com/object/sdk_samples.html

Soft-Edged Shadows with ps 3.0

- Works by taking 8 “test” samples from shadow map
  - If all 8 in shadow or all 8 in the light we’re done
  - If we’re on the edge (some are in shadow some are in light), do 56 more samples for additional quality
- 64 samples at much lower cost!
  - Quick-and-dirty adaptive sampling





ps.3.0 – Soft Shadows

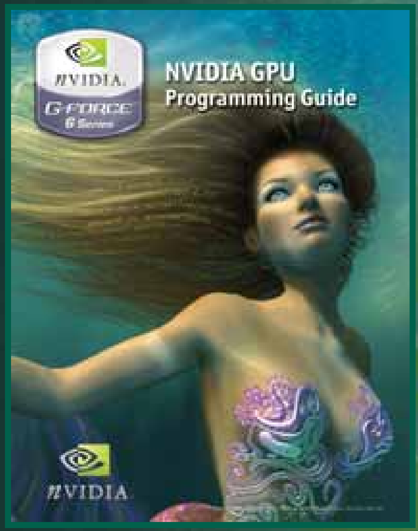
- This demo on GeForce 6 Series GPUs
  - Dynamic sampling > 2x faster vs. 64 samples everywhere
  - Completely orthogonal to other parts of the HW (for example, stencil is still usable)
  - Can do even more complex decision-making if necessary
- Combine with hardware shadow maps
  - High-quality real-time “soft” shadows are a reality

Summary

- Shader Model 3.0 provides a nice collection of useful features
- Looping/branching/conditional constructs allow greater programming flexibility
- Must watch out for performance gotchas
  - Don't make everything a nail for the SM3.0 hammer

References

- Tons of resources at http://developer.nvidia.com
  - NVIDIA SDK
    - http://developer.nvidia.com/object/sdk_home.html
  - Individual Standalone Samples (.zip)
    - http://developer.nvidia.com/object/sdk_samples.html
  - Individual FX Composer Effects (.fx)
    - http://developer.nvidia.com/object/sdk_effects.html



- Documentation
  - NVIDIA GPU Programming Guide
    - http://developer.nvidia.com/object/gpu_programming_guide.html
  - Recent Conference Presentations
    - http://developer.nvidia.com/object/presentations.html

Questions?

- Support e-mail:
  - devrelfeedback@nvidia.com [Technical Questions]
  - sdkfeedback@nvidia.com [Tools Questions]