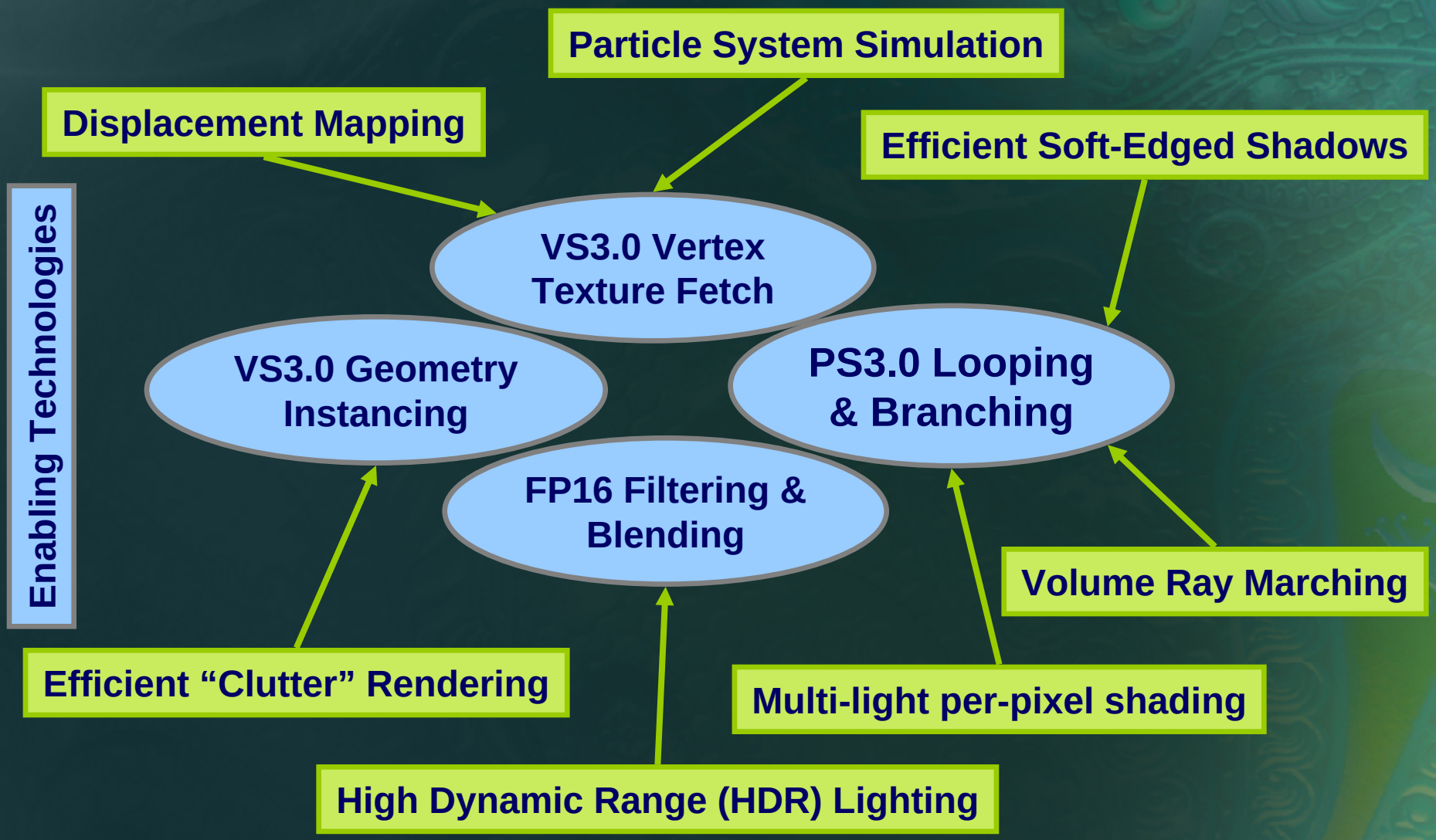
Next-Generation Rendering

Mark Harris

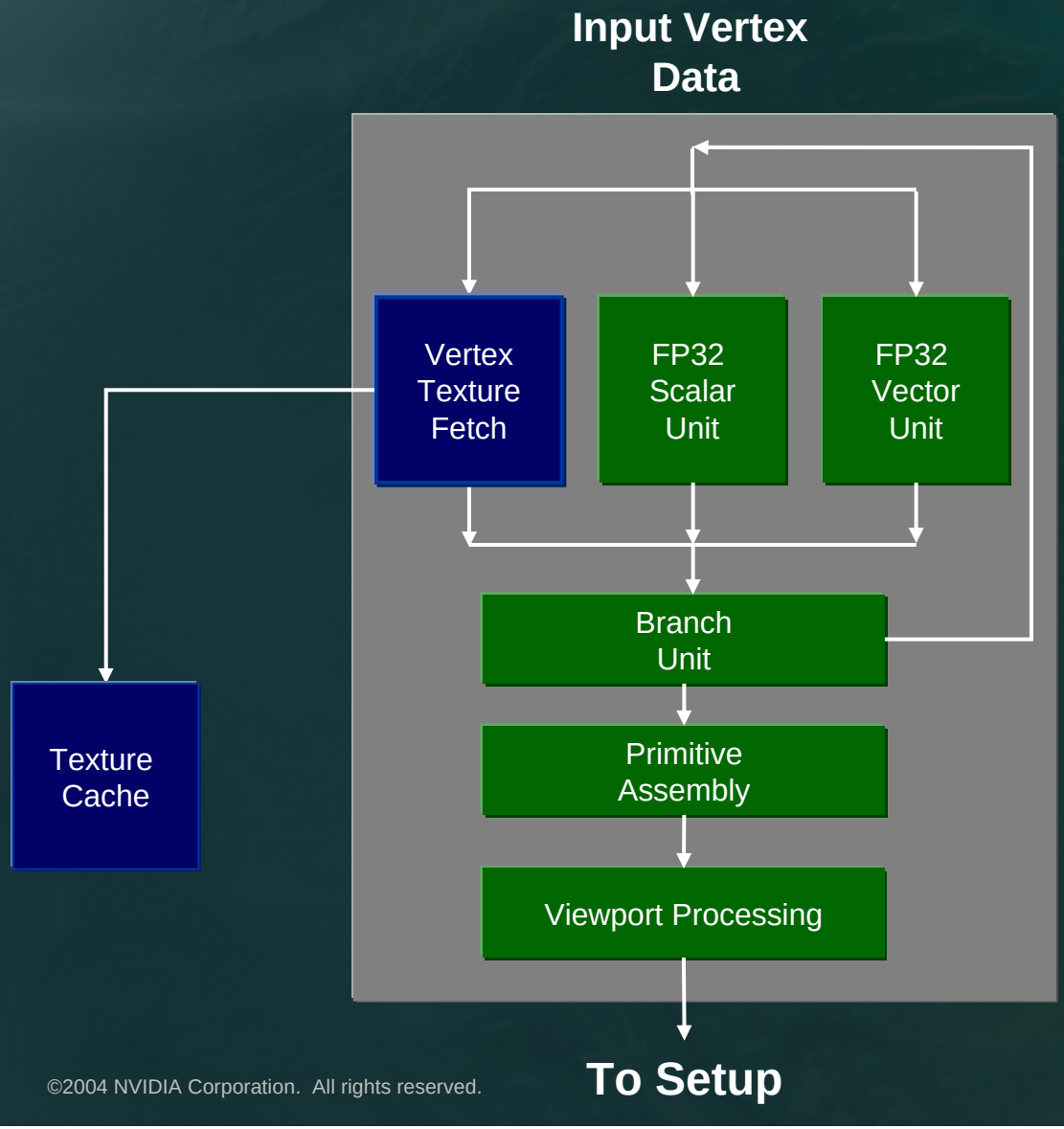
NVIDIA Developer Technology Group

Next-Generation Techniques




Displacement Mapping
... and other applications of
vertex texture fetch
VS3.0 Vertex
Texture Fetch

Detail of NV40 Vertex Shader Pipeline



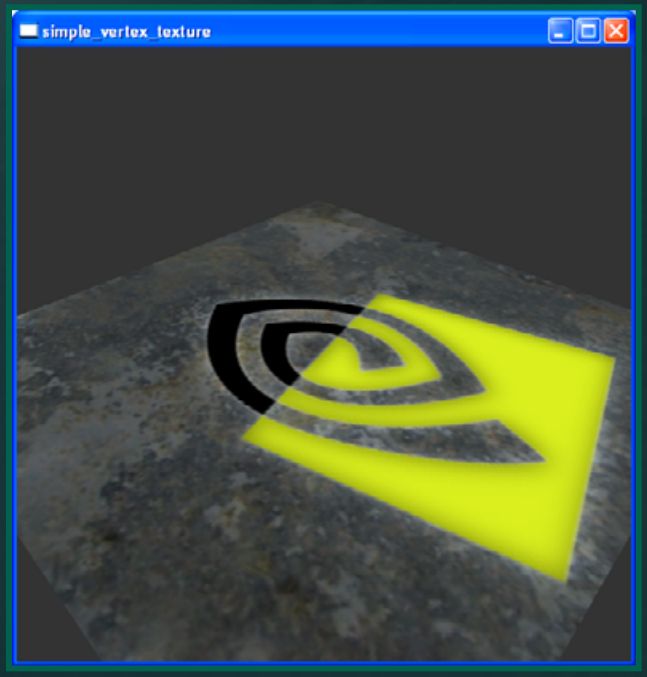
VPE

- MIMD Architecture
- Dual Issue
- Low-penalty branching
- Shader Model 3.0

VTF

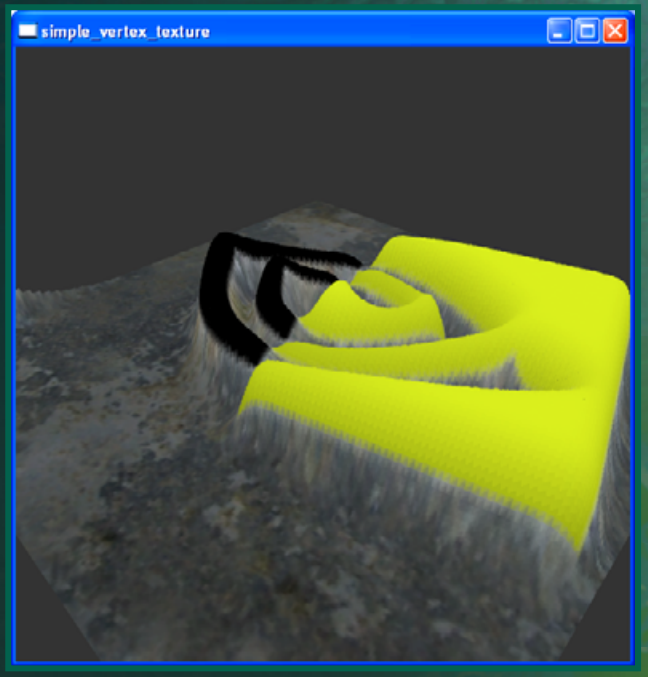
- VPE threads hide latency
- Non-stalling
- Up to 4 textures
- Mip-maps, no filtering

Displacement Mapping



Flat Tessellated Mesh

Displacement Texture



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

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

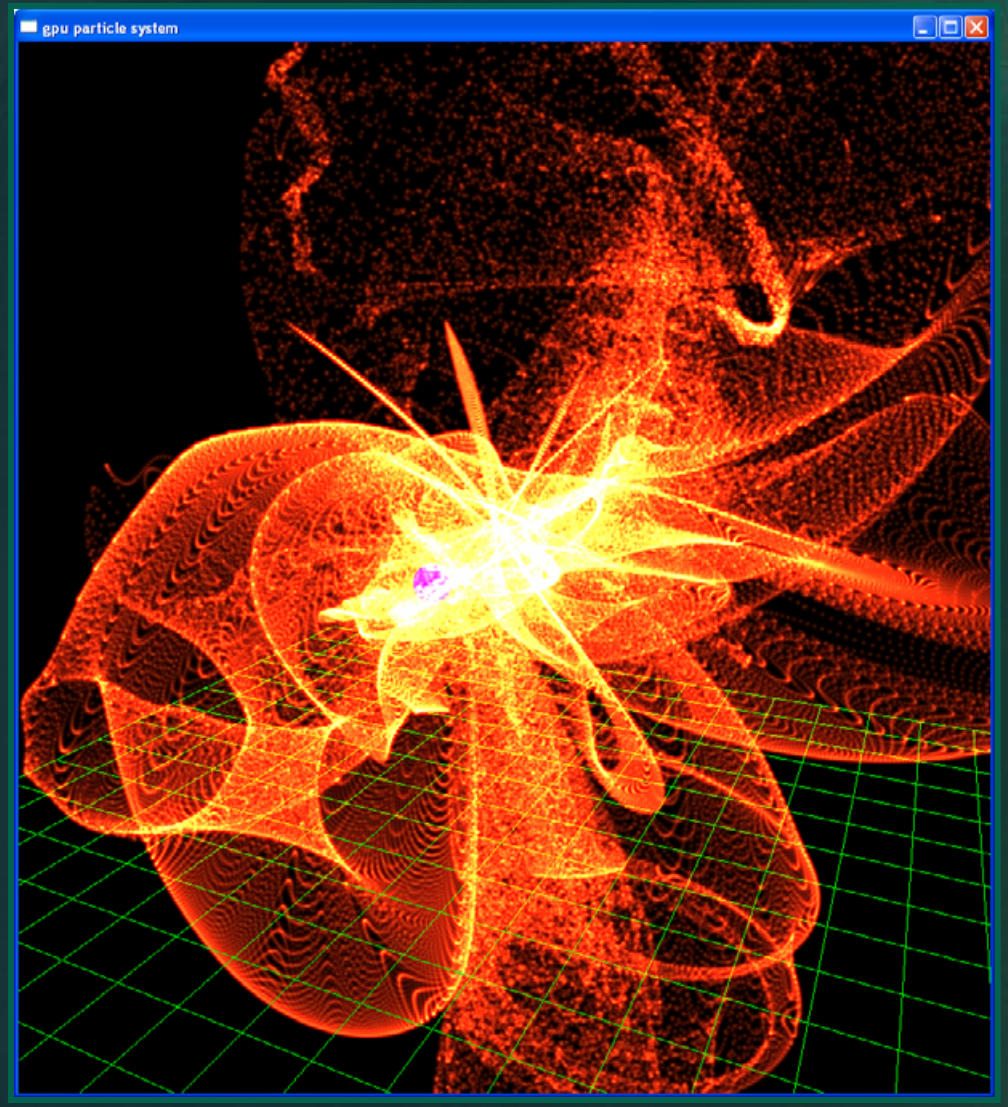
Vertex Texture

- Multiple vertex texture units
  - 4 units on GeForce 6 Series hardware
- Support for POINT filtering only (currently)
- Support for mipmapping
  - Need to calculate LOD yourself (texLDL)
- Supported Texture Formats (current)
  - A32G32B32R32F
  - R32F

Vertex Texture Applications

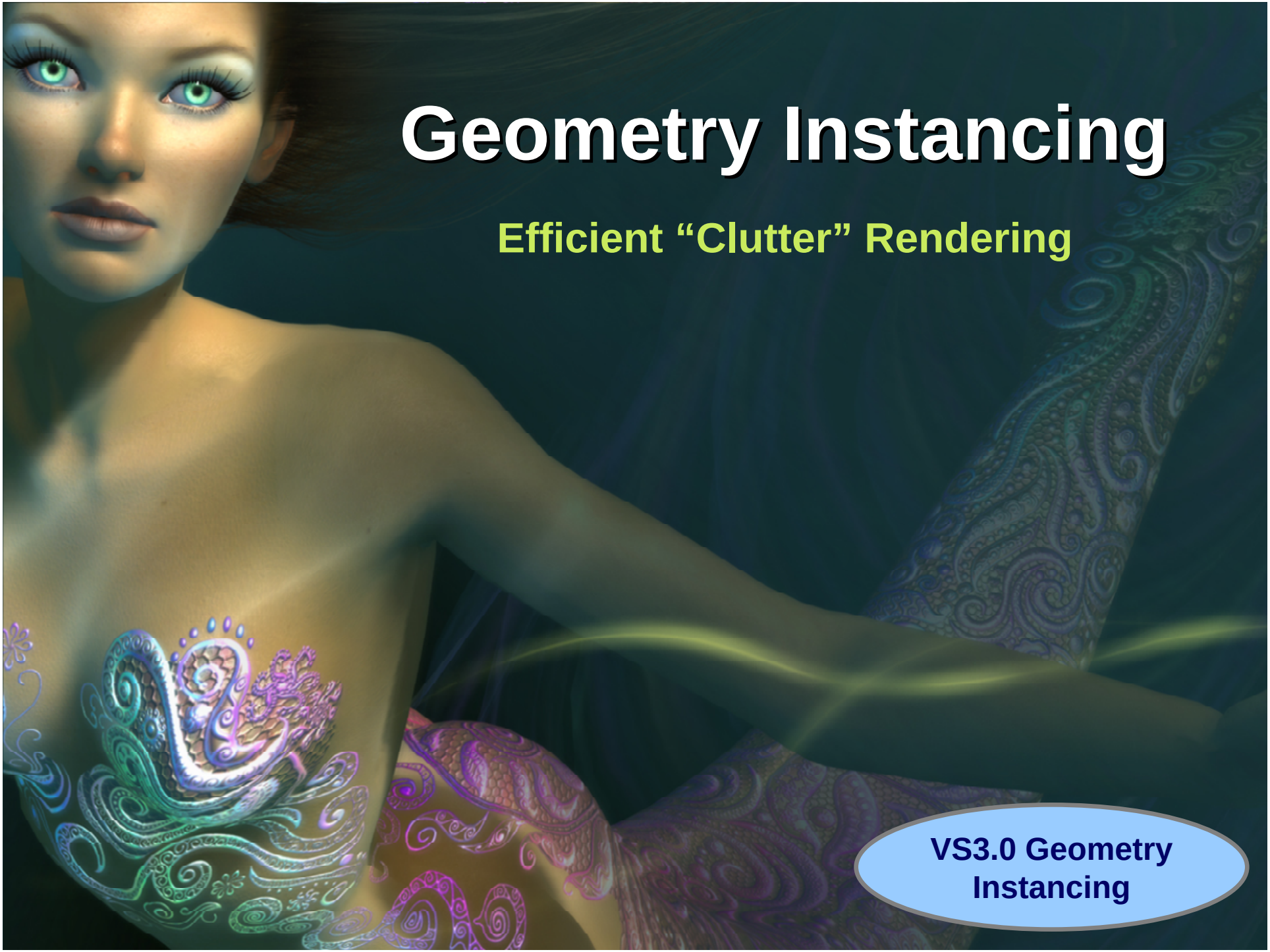
- Simple displacement mapping
  - Note – not adaptive displacement mapping
    - Hardware doesn't tessellate for you
  - Terrain, ocean surfaces
- Render to vertex texture
  - Provides feedback path from fragment program to vertex program
- Particle systems
  - Calculate particle positions using fragment 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



Vertex Texture Fetch Performance

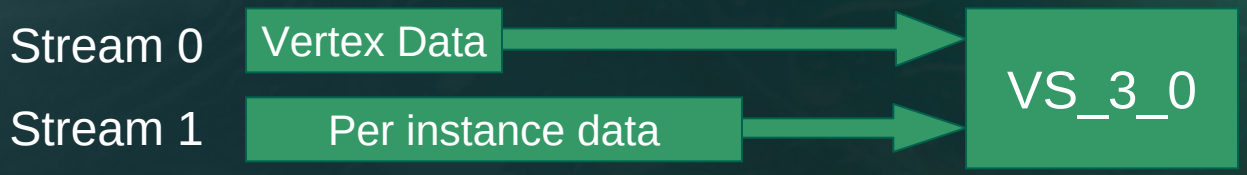
- Look-ups are not free
  - Higher latency than pixel shader tex instructions
  - Try to cover latency with non-dependent instructions
- NOT practical for use as extra constant memory
- Measured performance:
  - 33 million displaced vertices per second for basic displacement mapping
  - That’s more than a million displaced vertices per frame!
- Quite useable now, and this will only improve


Geometry Instancing
Efficient “Clutter” Rendering
VS3.0 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**
- OpenGL extension coming soon

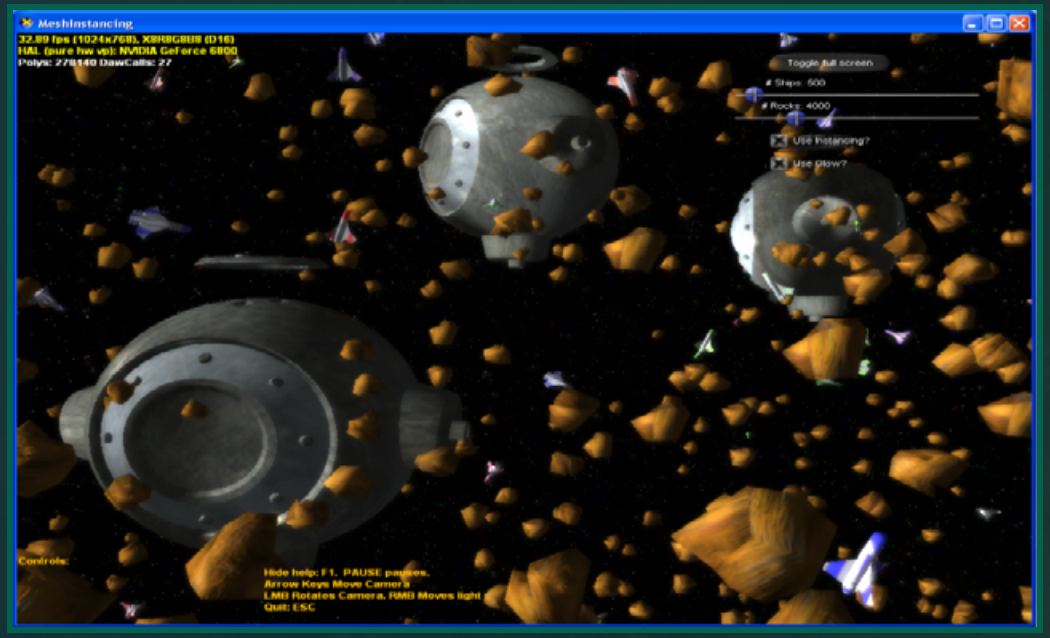
How does it work?



- 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.

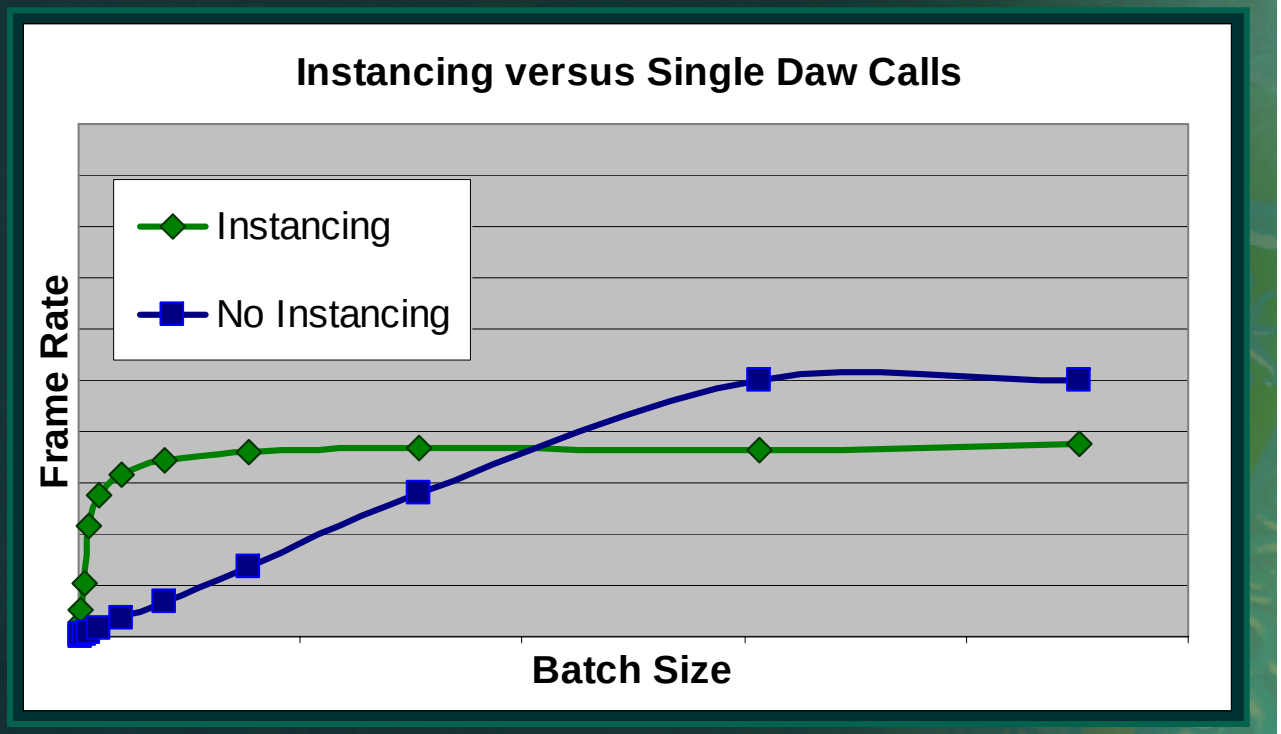
Instancing Demo

- 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.)



Render To Transform Matrix Array?

- Something to try...
- Render to texture + vertex texture fetch gives us a flexible feedback path over the entire GPU
- Combine with Geometry Instancing to provide dynamic animation of massive instanced systems
- You could render to a secondary stream containing transform matrices!
- Effectively allows “render to transform matrix array”
- Would allow directly displaying the results of massive GPU rigid body simulations without read-back

Credit for this idea goes to Simon Green


Applications of
Flow Control
Fast soft-edged shadows,
multi-light per-pixel shading,
volume ray marching, etc.
PS3.0 Looping
& Branching

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 vertices or pixels)

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

Using VS3.0 Flow Control

- Subroutines, loops, and conditionals simplify programming
- On GeForce 6 Series, dynamic branches have only ~2 cycle overhead
  - ***Even if vertices take different branches***
  - Use this to avoid unnecessary vertex work (e.g., skinning)
  - If you can branch to skip more than 2 cycles of work, do it!

PS3.0 Flow Control Overview

- Static and dynamic branching
- Loops
  - Number of iterations based on constants only
- Facing register available in pixel shader
  - Allows two-sided shading
- Mipmap LOD selection support (texLDL, texLDB)
- Indexable input registers (texture coordinates)
- Non-power-of-2 textures with mip-mapping
- 32k instructions

Using PS3.0 Flow Control

- Static branching is fast
  - But still may not be worth it for short branches (less than ~6-10 instructions)
  - Can use conditional execution instead
    - Just like in PS2.0
- Dynamic branching:
  - Lots of incoherent branching can hurt performance
  - Should have coherent regions of > 1000 pixels
    - That is only about 30x30 pixels, so still very useable!

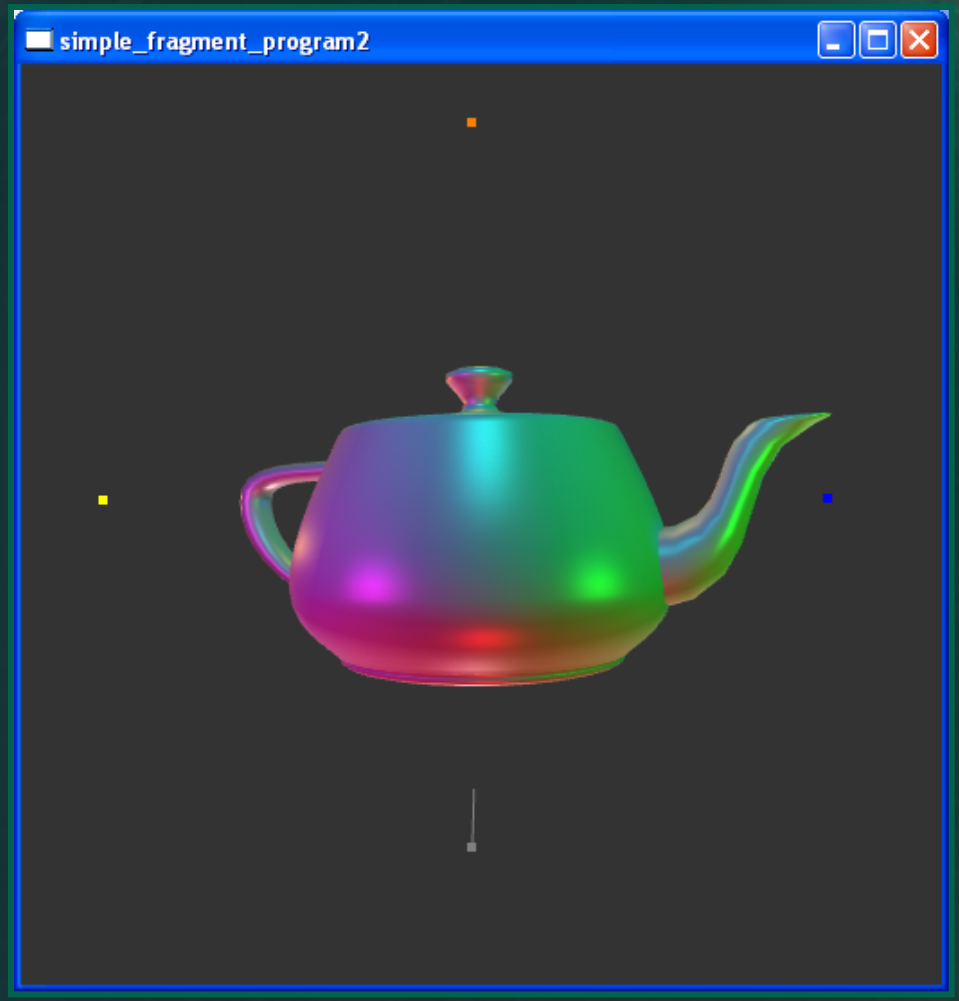
Branch Overhead

- Don’t ignore pixel shader flow control instruction costs:

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

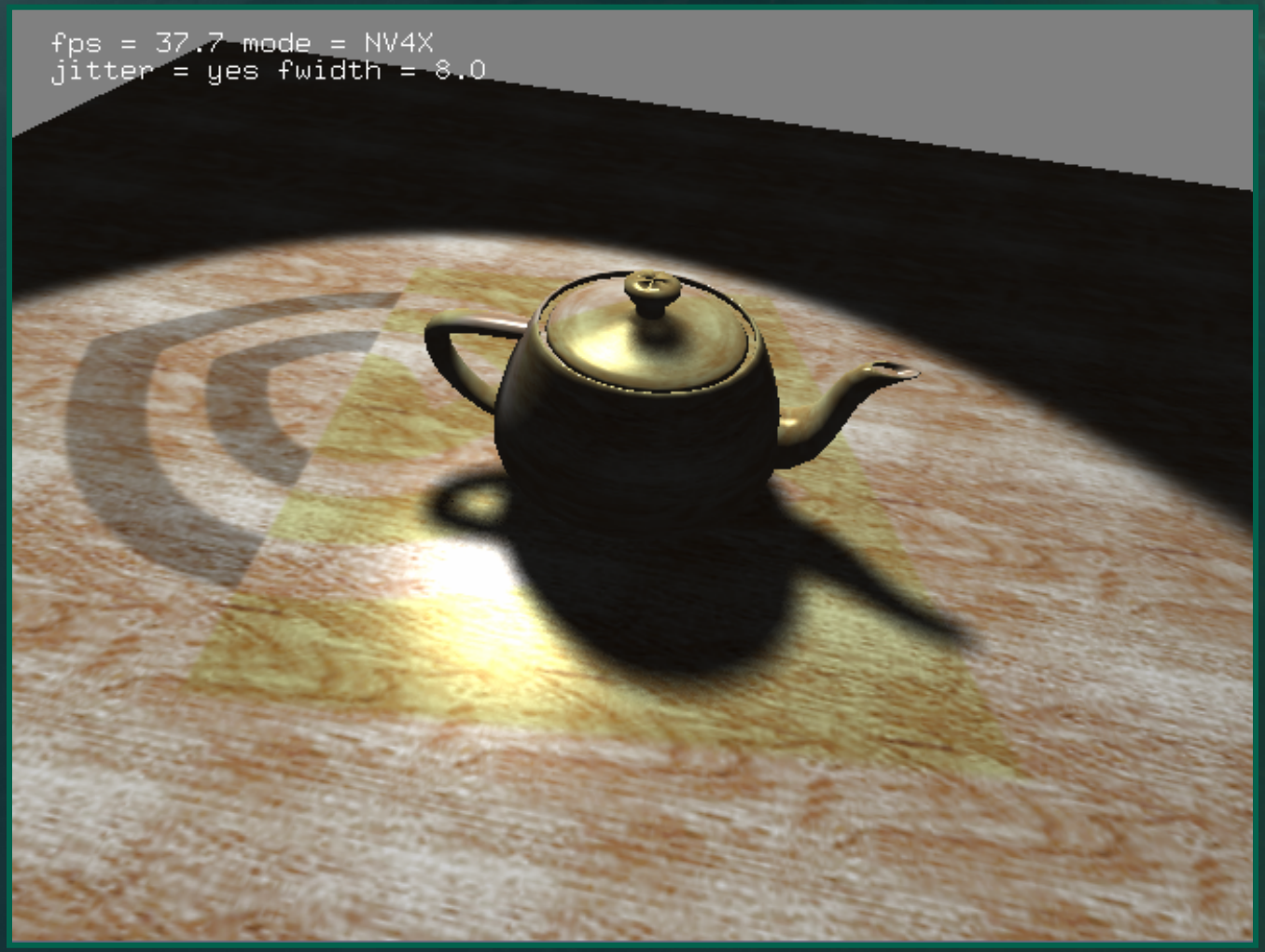
- So: branching over only a few instructions is not worth it,
- But branching over large blocks can be a big win!
  - Also early loop exit

Multiple Lights Demo



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

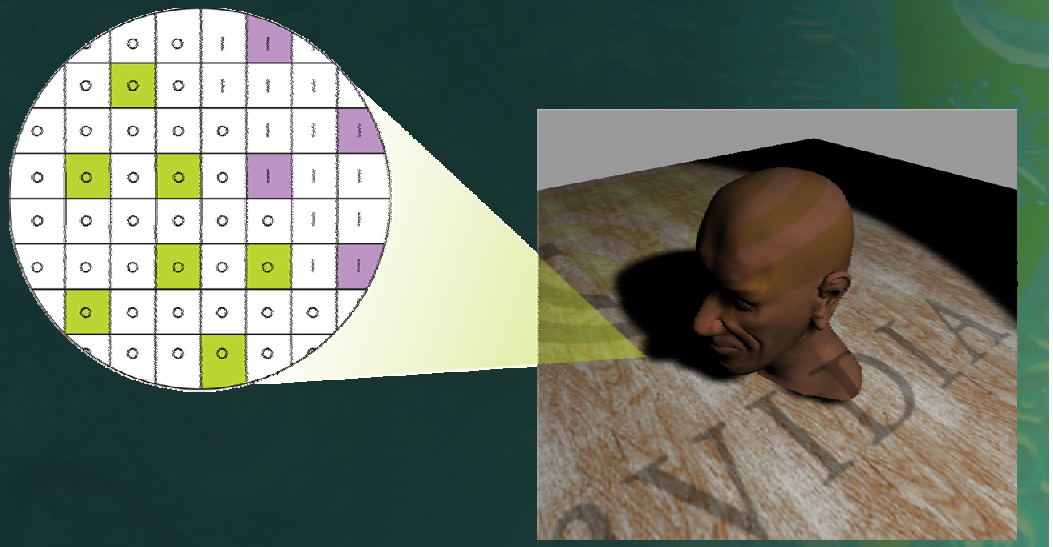
Soft-Edged Shadows with ps 3.0



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

Soft-Edged Shadows with PS3.0

- 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 importance sampling



Soft-Edged Shadows with PS3.0

- 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

Hardware Shadow Maps

- Many developers use R32F or R16F shadow maps
  - Render depth to single-channel float texture in shader
  - Multiple jittered samples for high quality / soft edges
- NVIDIA HW Shadow Maps = drop-in replacement
  - Same setup and pipeline as any shadow map technique
  - Shader code is simpler and faster

Hardware Shadow Maps

- Hardware does shadow map comparison for free
  - No need to compare and filter in the shader
- In D3D, render to a depth format texture
  - D3DFMT_D24X8, D3DFMT_D16
  - Use tex2Dproj to sample
  - Shadow map comparison is automatic

Percentage Closer Filtering

- PCF is “free” on GeForce3 up to GeForce 6800
  - Just enable bilinear filtering on the shadow map
    - Then each tex2Dproj does 4-sample PCF
    - *And* filters them bilinearly for a smooth result!
  - Use a single tap for performance
  - Or filter multiple taps to get higher quality
- Your choice: either 4x performance or 4x quality
  - Compared to rolling your own in a pixel shader

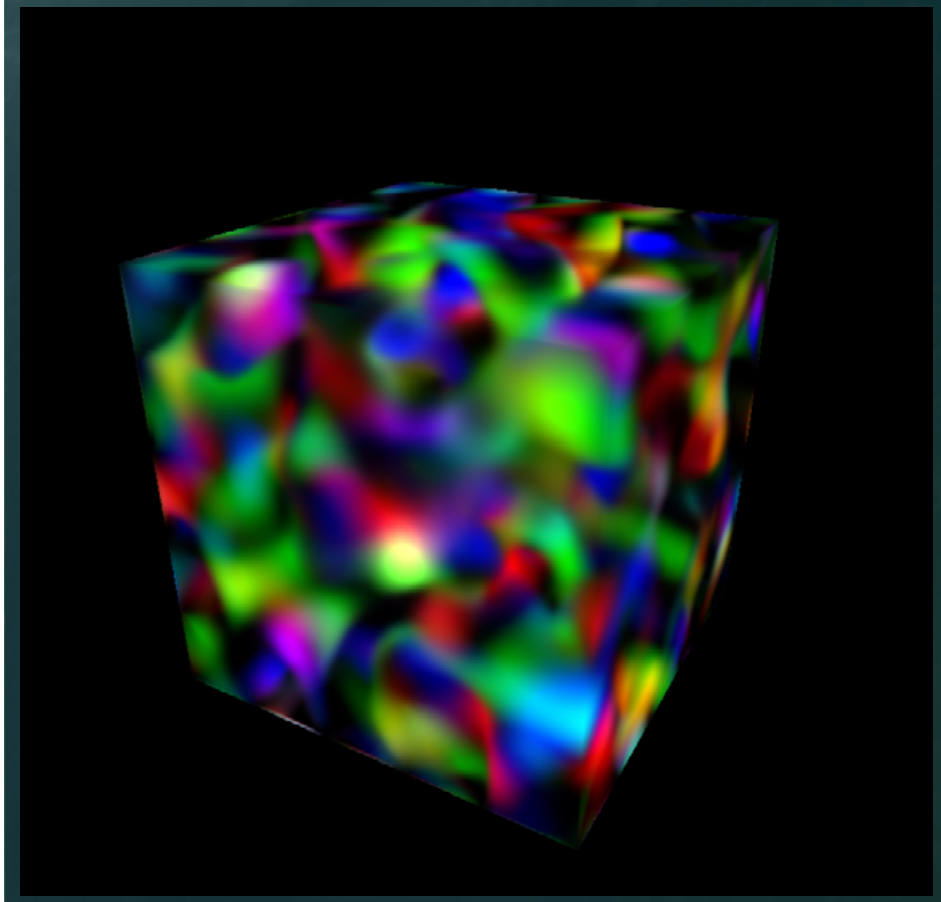
Fast Z-only Rendering

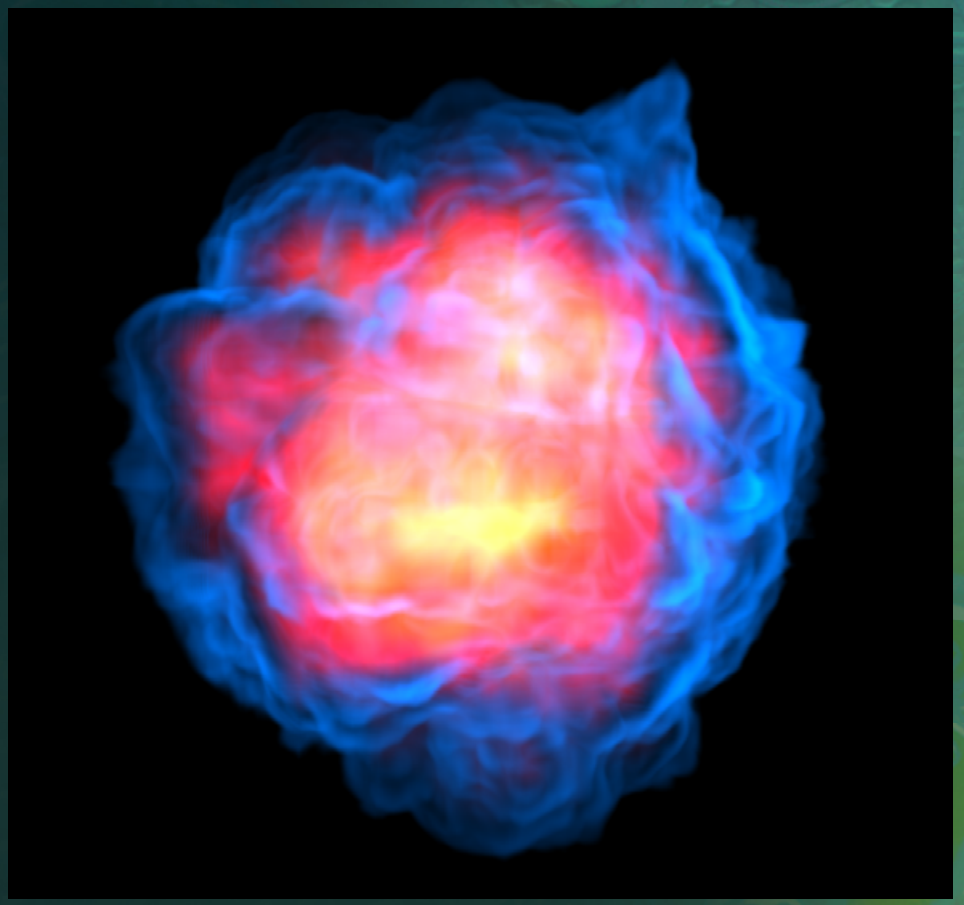
- GeForce FX and 6 Series can render Z-only at double-speed!
  - This is important for dynamic shadow maps!
  - Big speedup for shadow volumes also
  - Can be useful for other applications too.
- Make sure you:
  - Disable color writes
  - Disable alpha test
  - Check the GPU Programming Guide for full list if you're having trouble getting double-speed rendering

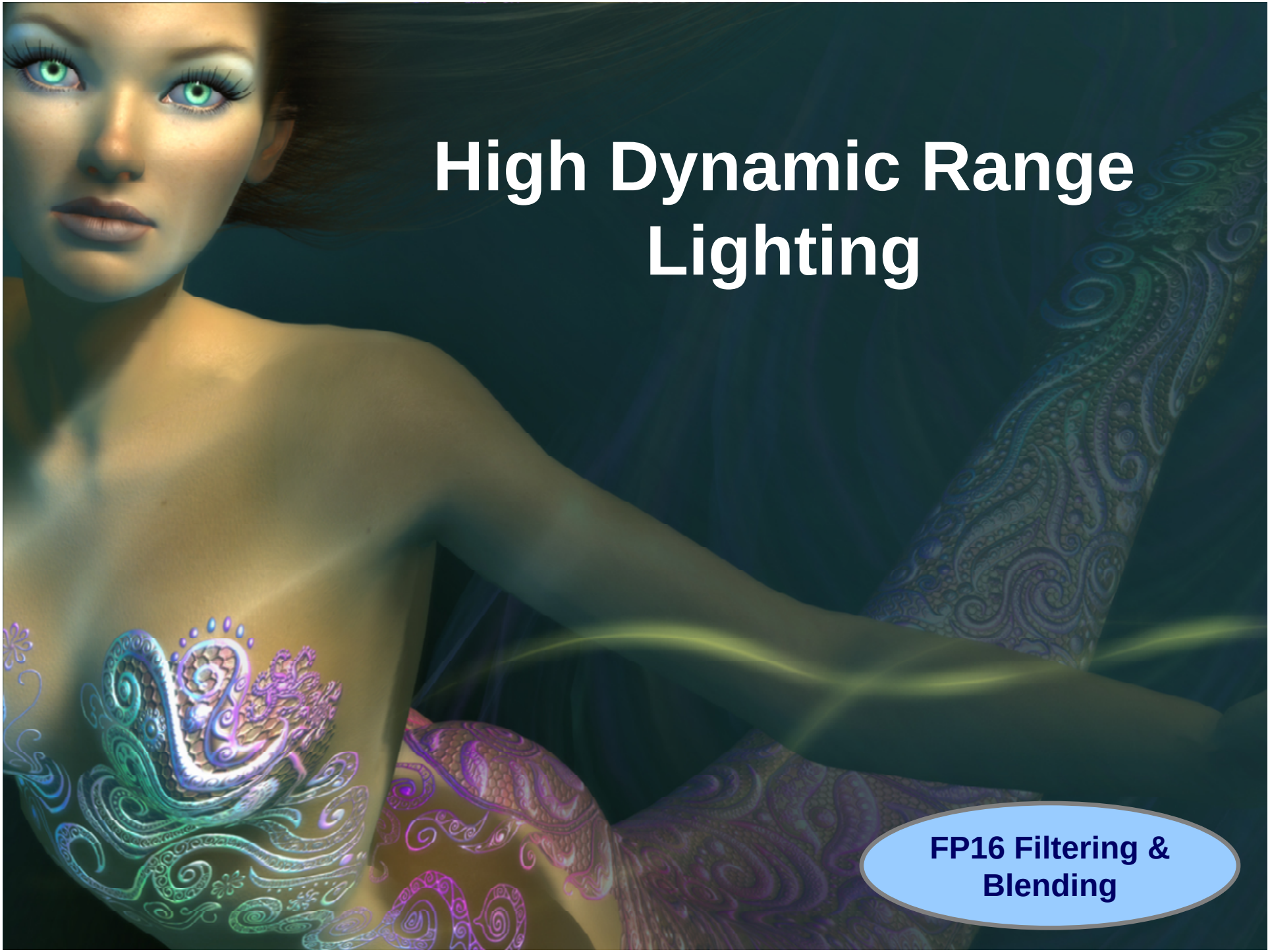
Pixel Shader Looping Example - Single Pass Volume Rendering

- Pixel shader calculates intersection between view ray and bounding box
- 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



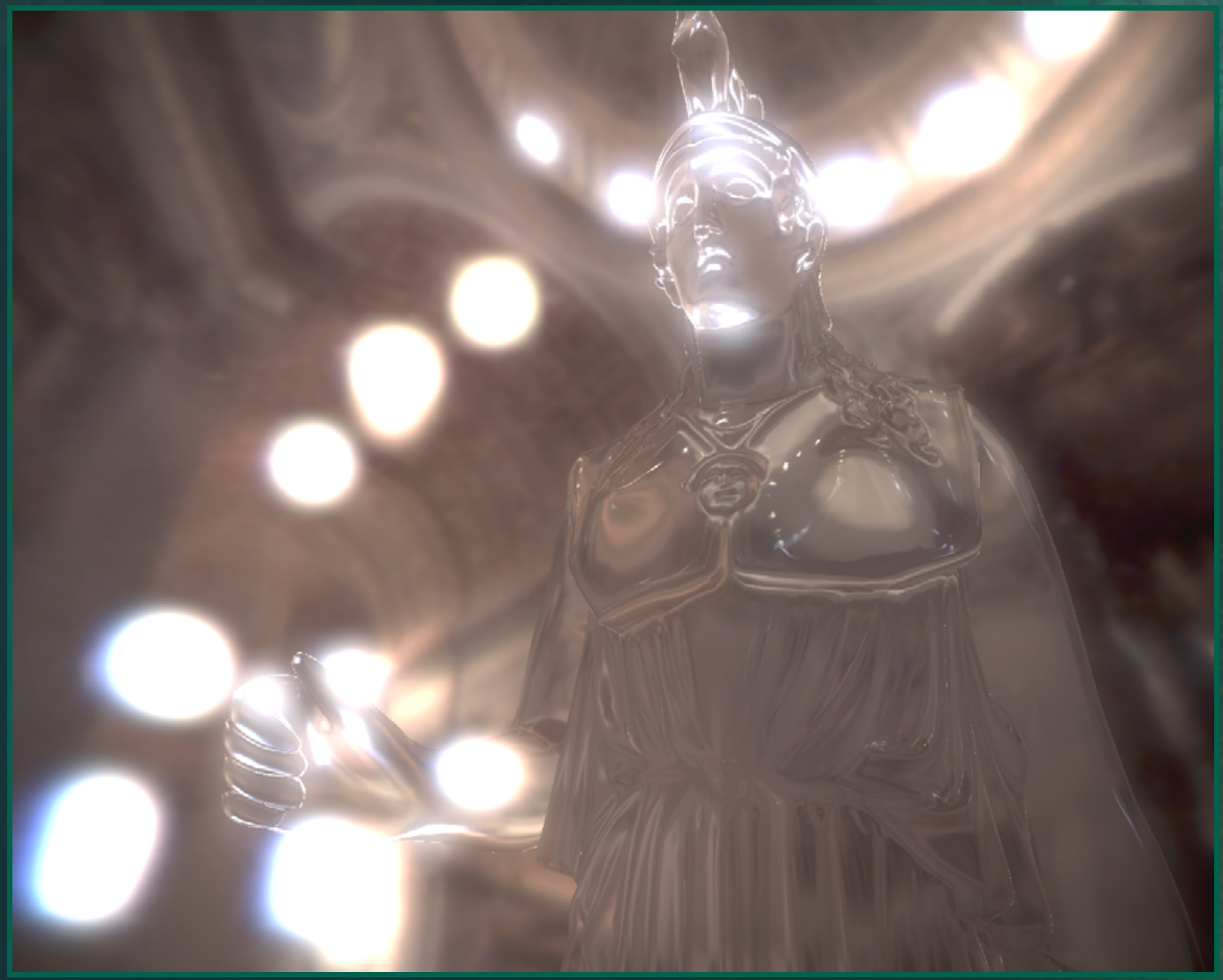



High Dynamic Range Lighting
FP16 Filtering & Blending

What is HDR (High Dynamic Range)?

- Dynamic Range = $\log_{10}(^{max\_intensity}/_{min\_intensity})$
  - Human perception is 10^{14}:1 (14 dB), with log response
  - Standard 32 bpp frame buffer is 255:1 (2.4 dB)
- HDR Rendering Engine:
  - Define light sources in natural units (not [0...1])
  - Compute surface reflectance, save in HDR buffer
    - Contributions from multiple lights are additive (blended)
  - Add image-space special effects to HDR buffer
    - AA, Glow, Depth of Field, Motion Blur
  - Tone-Map HDR buffer to LDR for display
- Good, robust HDR requires floating point

High Dynamic Range Imagery



GeForce 6800 HDR Technology

- Studio-Quality 16-bit floating point
  - 12.0dB dynamic range
  - 11-bit logarithmic precision
  - Sufficient for nearly any application
- Fully Orthogonal Hardware Support
  - Texture Filtering, including trilinear+16x anisotropic
  - All OpenGL 1.5 and DX9.0c alpha blending modes
- Easy to Use
  - No complicated pixel shader encode/decode
  - Already exposed in OpenGL and DirectX 9 APIs

HDR: int16 versus fp16

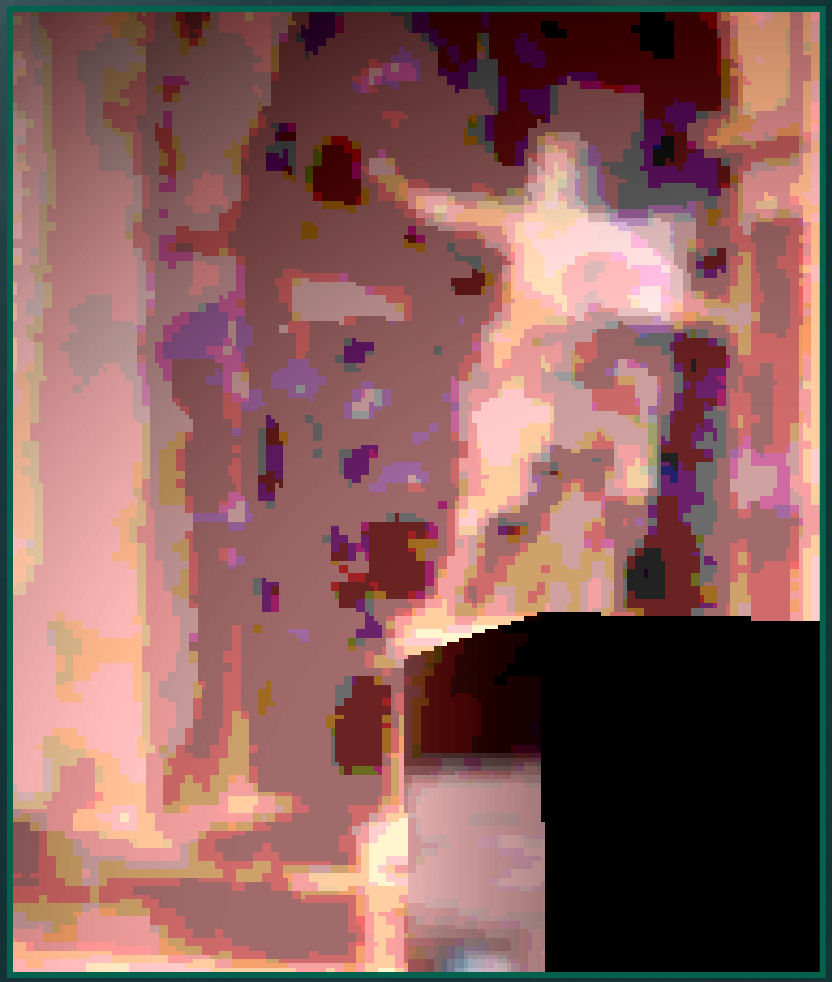
HDR with 16-bit Integer

HDR with 16-bit Floating Point

Dynamic range: 200,000:1

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

HDR: int16 versus fp16



HDR with 16-bit Integer

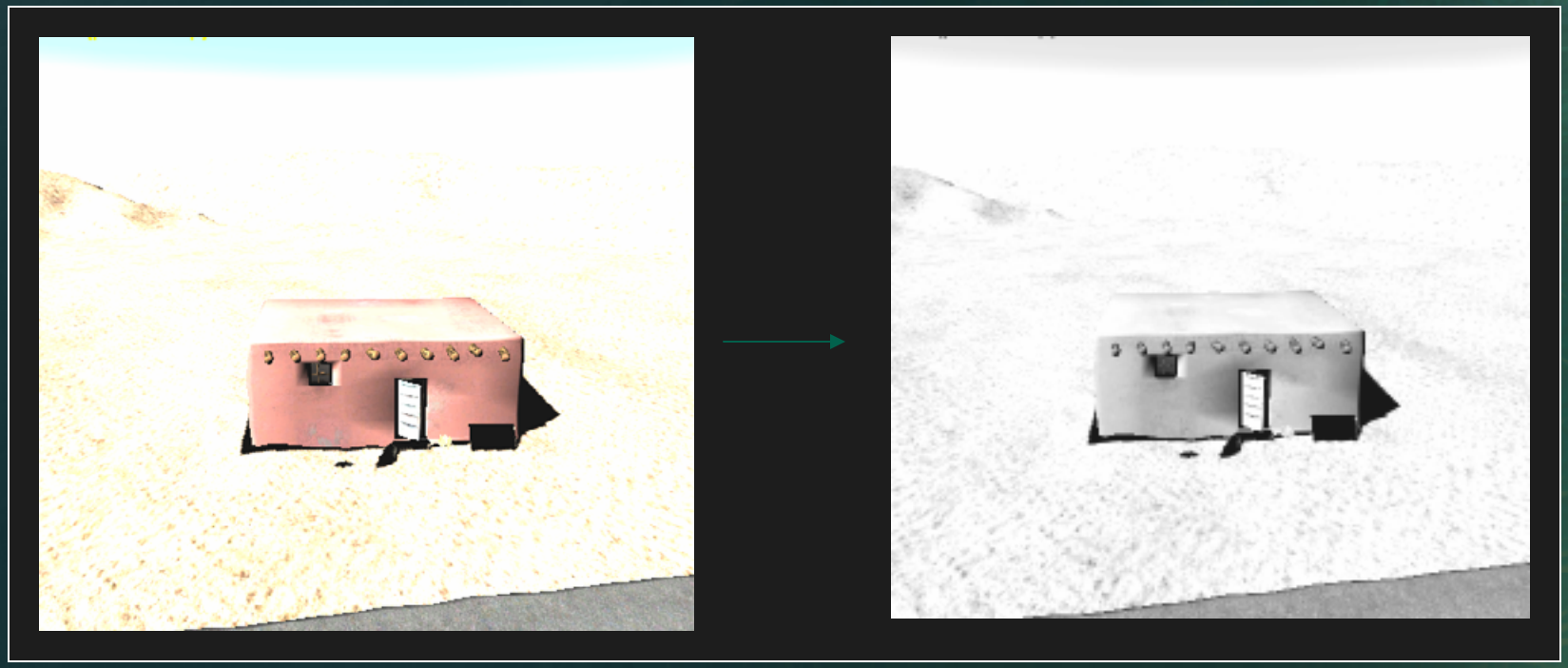
HDR with 16-bit Floating Point

Dynamic range: 200,000:1

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

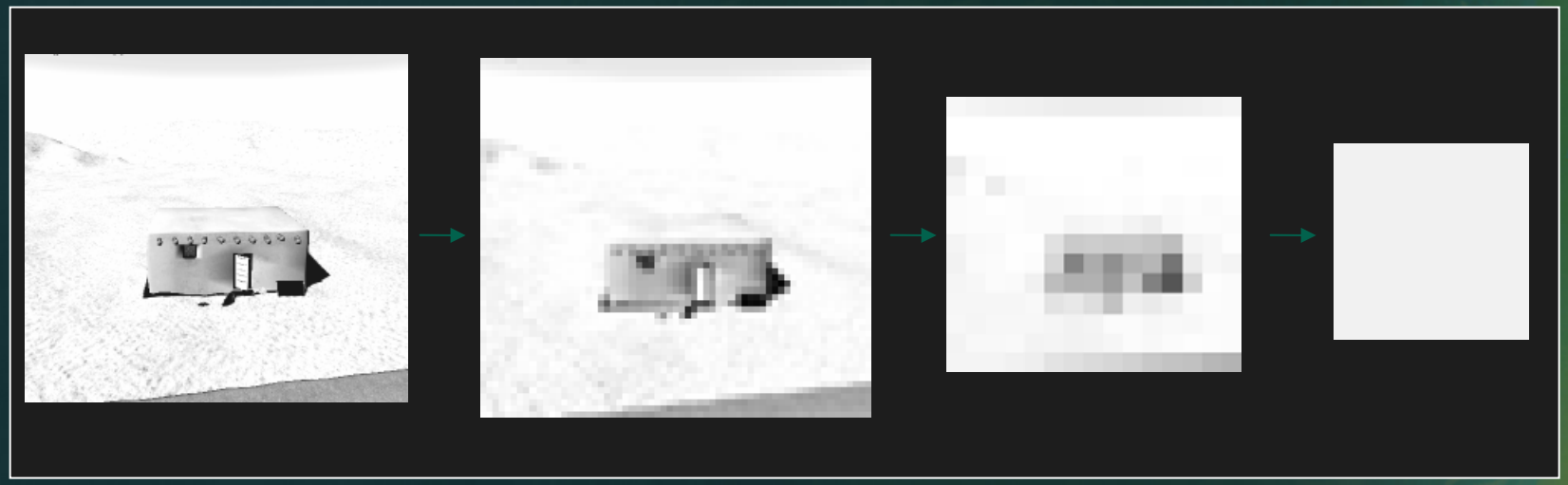
Simple Tone Mapping

- Given an HDR scene, first convert to luminance



Simple - Tone Mapping

- Down-filtered all the way to 1x1
  - This is trivial and fast with 6800's fp filtering
  - Gives you the average luminance for the scene



Simple - Tone Mapping

= 1x1 pixel average luminance

- Use Average Luminance to apply scaling

$$Lum_{scaled}(x,y) = \frac{\alpha}{Lum_{avg}} Lum(x,y)$$

- And compression to LDR [0, 1] range

$$Color(x,y) = \frac{Lum_{scaled}(x,y)}{1 + Lum_{scaled}(x,y)}$$

Floating-Point Blending

- On GeForce FX GPUs, you have to emulate float blending using “ping-pong buffer”
  - Lots of context switches and additional passes
  - Blending lots of particles, e.g., is infeasible
- GeForce 6 Series solves this for fp16
  - Makes many algorithms cheaper
  - Accumulating light contributions, transparency, etc.
  - GPGPU array addition

Resources

- NVIDIA SDK
  - Videos, demos, source code
  - http://developer.nvidia.com/object/sdk_home.html
- Contact
  - Mark Harris (mharris@nvidia.com)