Session Overview

- 3.0 Shader Model Overview
  - ps.3.0 vs. ps.2.0
  - vs.3.0 vs. vs.2.0
- Next-Gen Rendering Examples
  - Dynamic Water
    - Vertex Texture Fetch
    - Floating-point filtering / blending
    - GPU-based physics simulation
  - Volumetric Fog
    - MRT and branching for speed
  - Deferred Rendering
    - MRT and branching for speed
- Geometry Instancing
  - Added visual complexity
  - Performance optimization

Pixel Shader 3.0 Feature Comparison

Pixel Shader Feature	Shader 2.0	Shader 3.0	Description
Shader length	96	65535+	Allows more complex shading, lighting, and procedural materials
Dynamic branching	No	Yes	Saves performance by skipping complex shading on irrelevant pixels
Shader anti-aliasing	Not supported	Built-in derivative instructions	Developers can calculate the screen space derivatives of any function, allowing them to adjust shading frequencies or over-sampling to eliminate artifacts
Minimum Precision	fp24	fp32	Fewer artifacts, more dynamic range
Back-face register	No	Yes	Allows two-sided lighting in a single pass

Pixel Shader 3.0 Feature Comparison

Pixel Shader Feature	Shader 2.0	Shader 3.0	Description
Back-face register	No	Yes	Allows two-sided lighting in a single pass
Interpolated color format	8-bit integer minimum	32-bit floating point minimum	Higher range and precision color allows high-dynamic range lighting at the vertex level
Multiple render targets	Optional	4 required	Allows advanced lighting algorithms to save filtering and vertex work – thus more lights for minimal cost
Fog and specular	8-bit fixed function minimum	Custom fp16-fp32 shader program	Shader Model 3.0 gives developers full and precise control over specular and fog computations, previously fixed-function
Texture coordinate count	8	10	More per-pixel inputs allows more realistic rendering, especially for skin

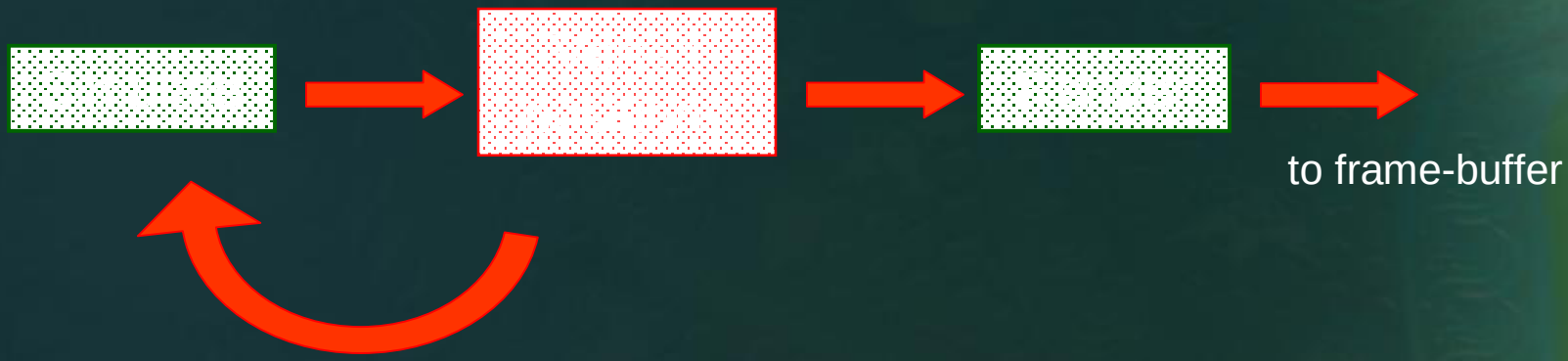
Vertex Shader 3.0 Feature Comparison

Vertex shader feature	Shader 2.0	Shader 3.0	Description
Shader length	256 Instructions	65535 instructions	More instructions allow more detailed character lighting and animation
Dynamic branching	No	Yes	Saves performance by skipping animation and calculations on irrelevant vertices
Vertex texture	No	Any number of lookups from up to 4 textures	Allows displacement mapping, particle effects
Instancing support	No	Required	Allows many varied objects to be drawn with only a single command

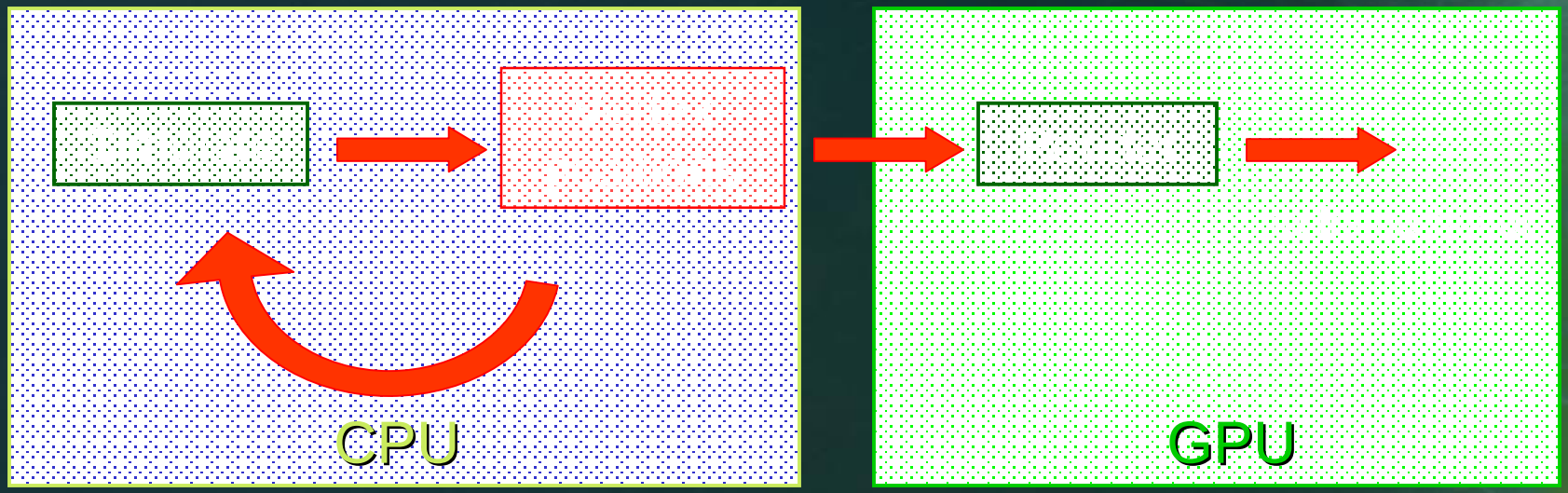
So what can we do with all this?

- Dynamic Water Rendering
  - VS3.0 Vertex Texture Fetch
  - Floating-point filtering / blending
  - GPU-based physics simulation
- Animated Volumetric Fog
  - Use polygon primitives to bound fog
  - MRT and branching for speed
- Geometry Instancing
  - Draw many “instances” of a mesh with one draw call

Typical Workflow

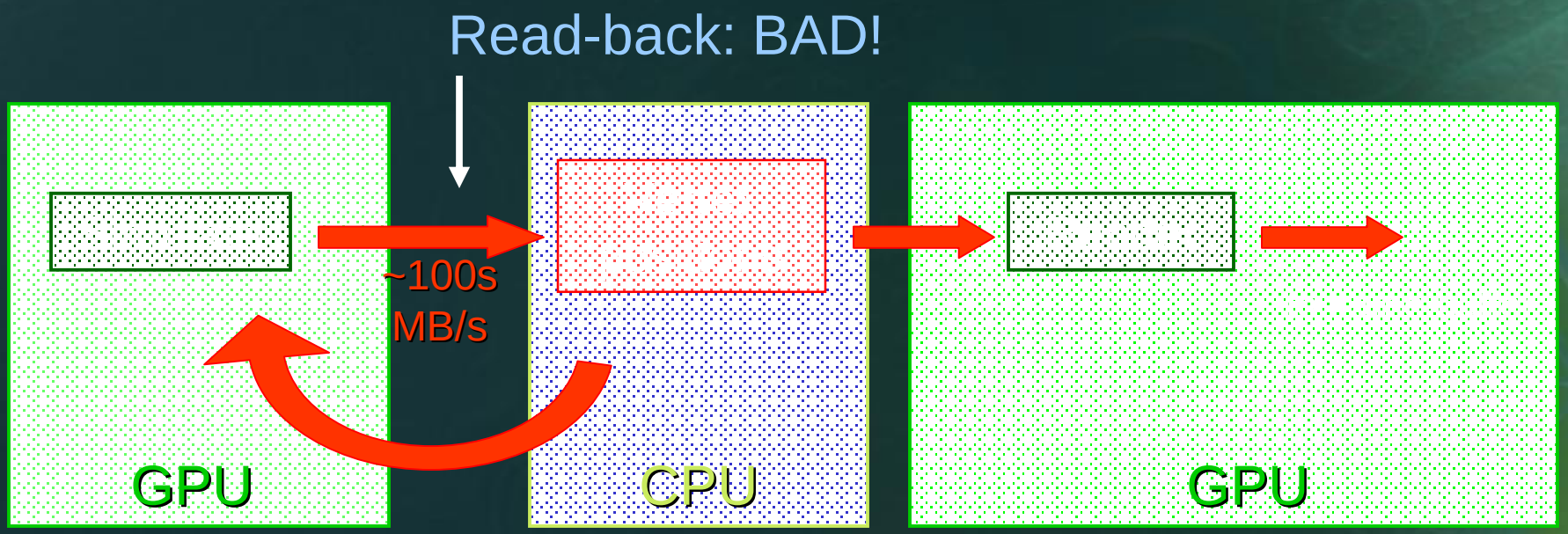


Typical Processing Allocation

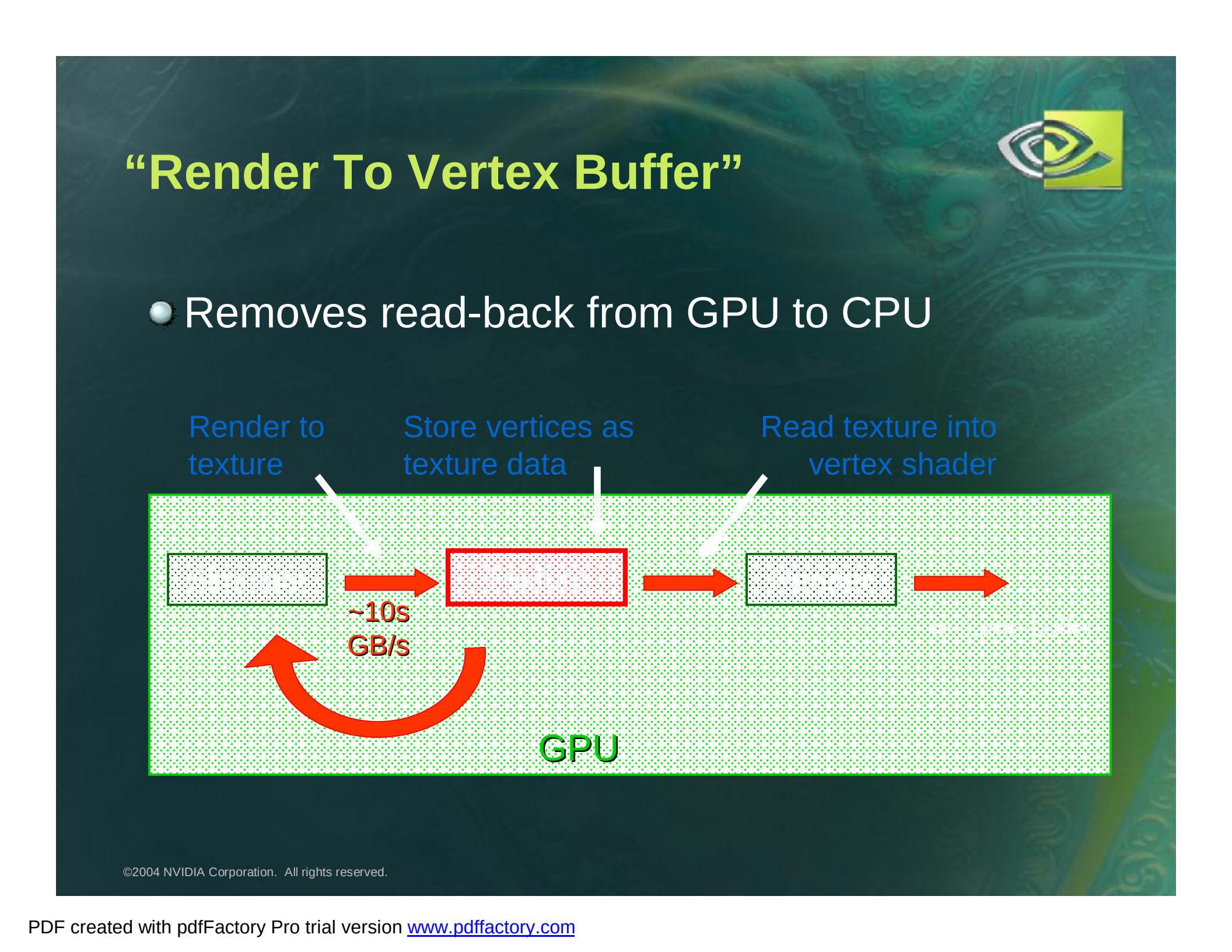


Simulating On the GPU?

- Use them programmable shaders!
  - The read-back can kill you
  - This is for PCI. PCI Express is better.



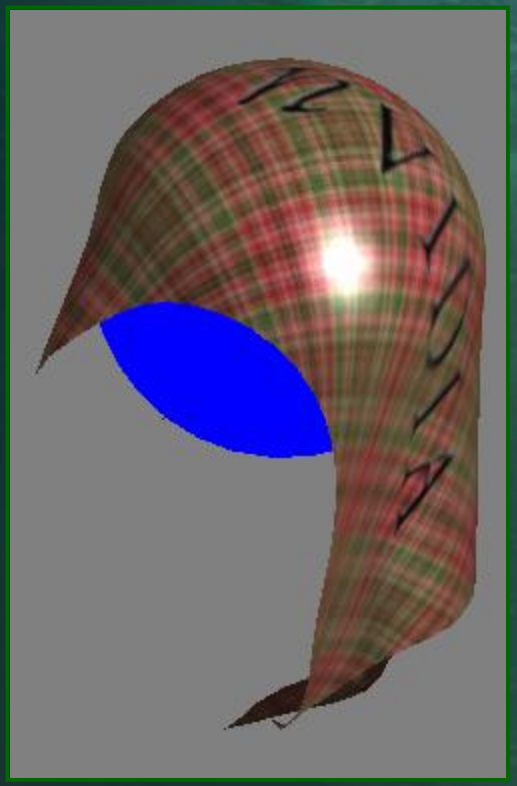
“Render To Vertex Buffer”



- Removes read-back from GPU to CPU

Examples

- Cloth
  - Collide cloth against scene
  - Run cloth physics: damped springs
- Displacement Mapping
  - Displace vertices

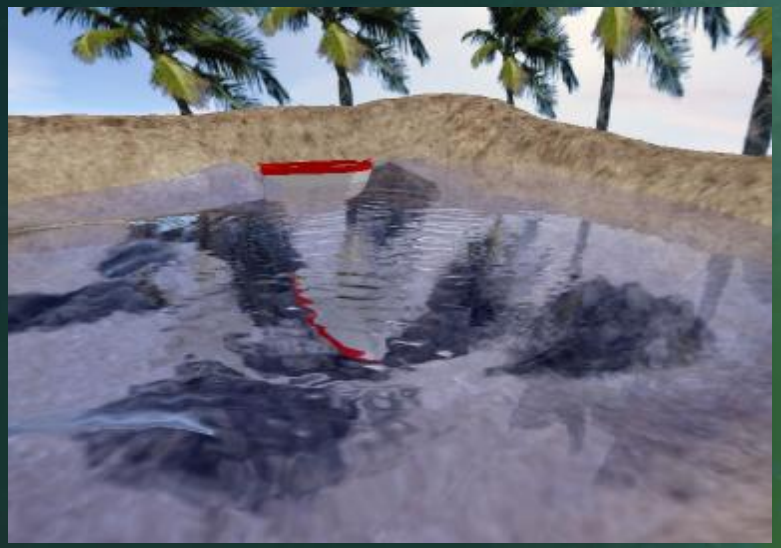


More Examples

- Snow/Sand accumulation
  - Simulate friction/sliding
- Wind (simulation) bending grass
- Particle Systems
- Water waves/wakes

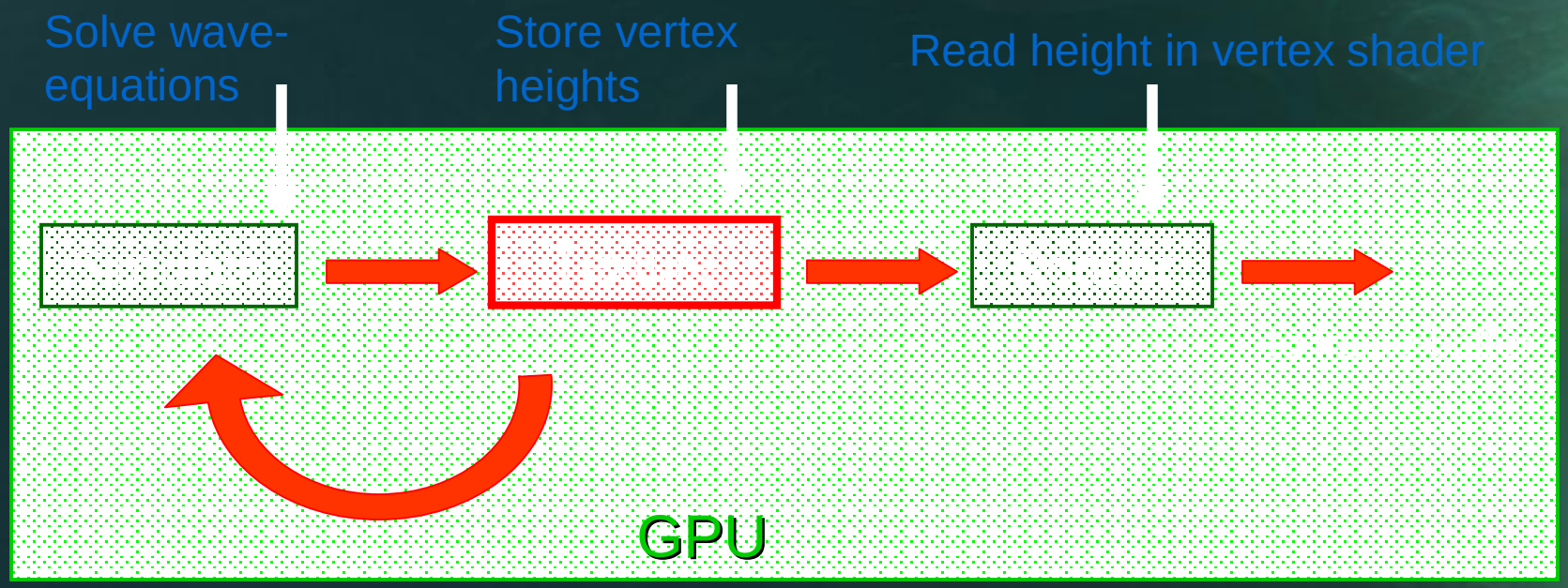
Rendering Water – Algorithm Overview

- Perform water simulation in pixel shader
  - Render to texture (D3DFMT_A16B16G16R16F)
- Render refraction and reflection maps
- Render water surface
  - Use simulation results via VS3.0 vertex texture fetch
  - Compute perturbed texture coordinates
  - Combine refraction and reflection using Fresnel term



Rendering Water

- Tessellated, flat plane for water



How Does It Work?

- Create a vertex-mesh for water surface
  - e.g. 128 by 128 vertices
  - Encode vertex's mesh-position as uv-coordinates

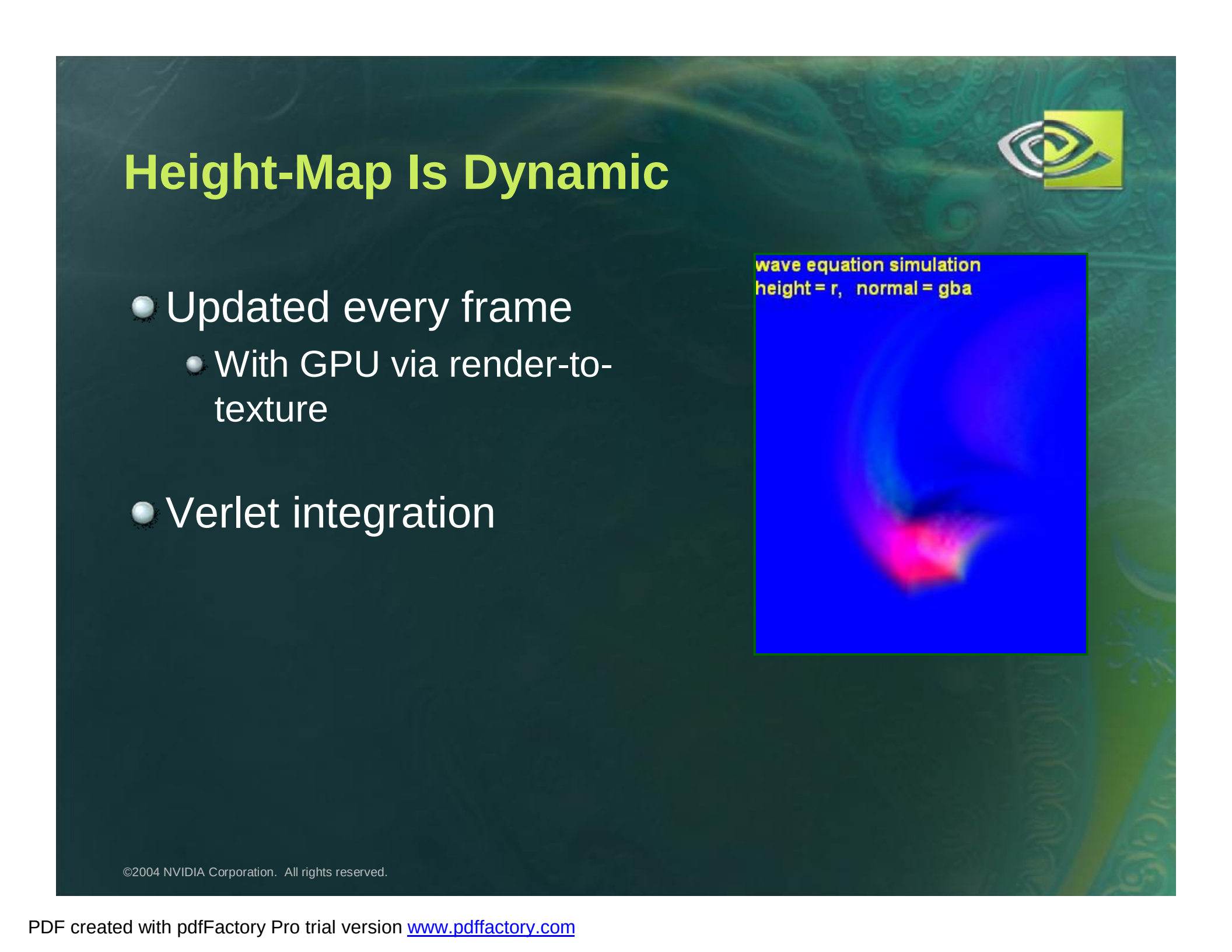
Vertex Shader Work

- Read ‘height-map’
  - Floating-point texture
  - Read texture at vertex’s uv
- Add result to vertex’s y
- Transform/Project vertex

vertex.y += tex(u, v)

Out.pos = WorldViewProj * vertex

Height-Map Is Dynamic



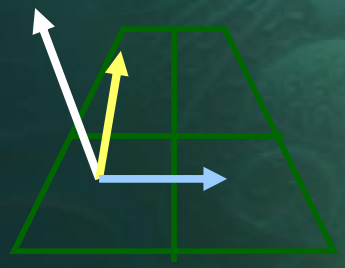
- Updated every frame
  - With GPU via render-to-texture
- Verlet integration

Verlet you say?

$$A = \sum (\text{neighbors}) - 4\,H_n$$

$$H_{n+1} = H_n + (H_n - H_{n-1}) + A$$

$$H_{n+1} = (2*H_n - H_{n-1}) + A$$



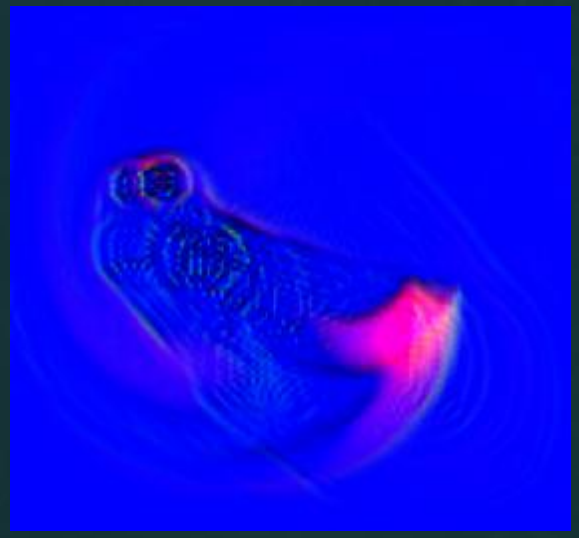
- Operates on positions only
  - No need to store velocity or acceleration
- Compute normal from positions:

  N = Normalize(S x T)

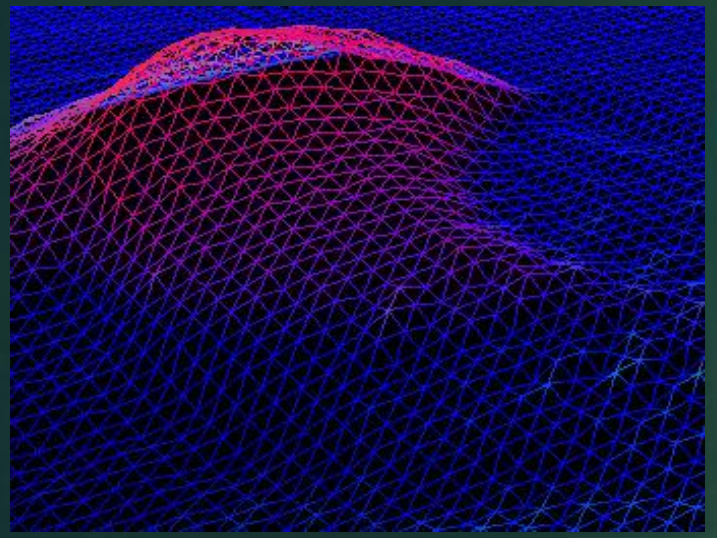
Vertex Texture Fetch (VS3.0)

- Vertex shader reads simulation result with vertex texture fetch

Simulation Texture



Applied Height Map



Add Disturbances to Height Map

- Blend displacements into the water
  - For example: the boat, rocks, shore
- Verlet-integration integrates it next frame
- Yes, floating-point render-target blending

Refraction Map

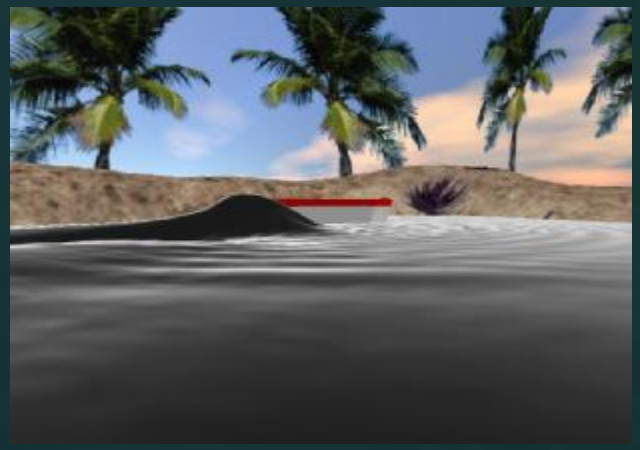
- Render scene from camera viewpoint
- Render what is on other side of the water
  - Use water plane to clip geometry
  - If camera is above water
    - Render under water geometry
  - If camera is under water
    - Render above water geometry

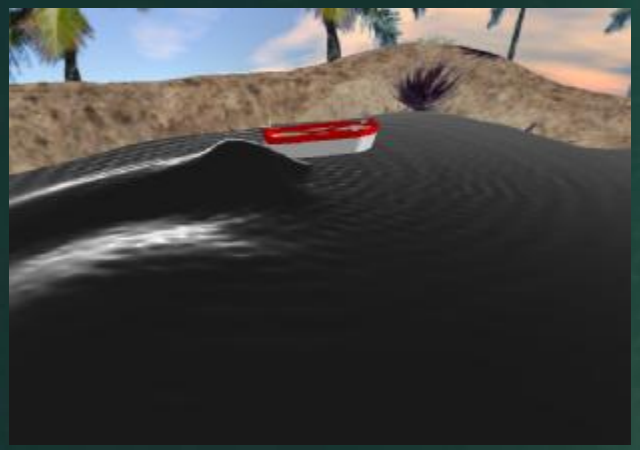
Reflection Map

- Render scene from reflected camera viewpoint
  - Reflect view transform about water plane
- Again clipped using water plane
- Render what is on the same side of the water
  - If camera is underwater
    - Render underwater geometry
  - If camera is above water
    - Render above water geometry

Fresnel Reflection Term

- Determines amount of reflection / refraction
- Roughly pow((1 - dot(eye, normal)), p)
  - Fresnel term = 0 => all refraction
  - Fresnel term = 1 => all reflection

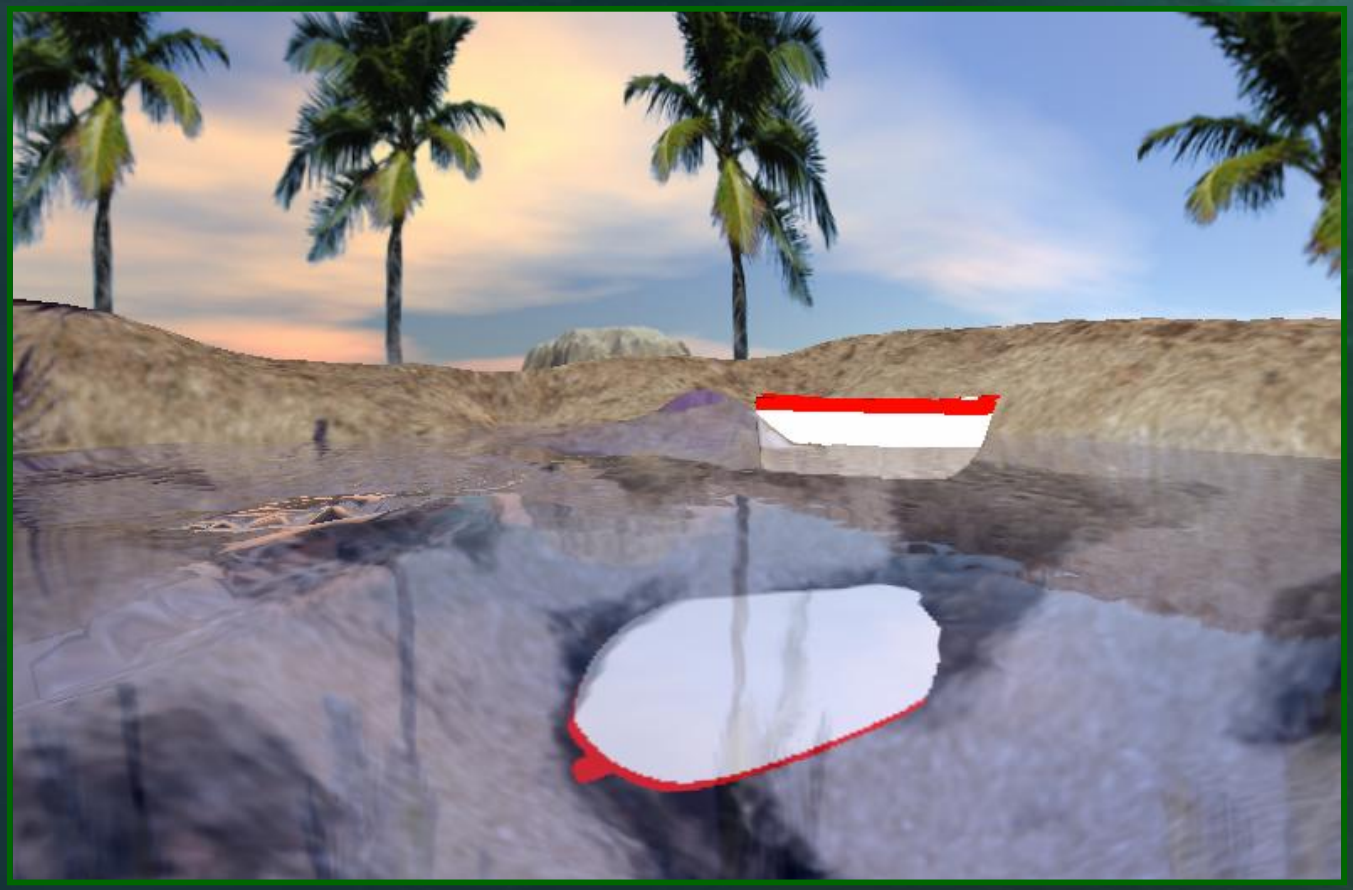




Advantages

- Fast!
  - Simulation happens on 128x128 texture
  - Small by GPU standards
  - Frame-rate unaffected by simulation
- Reasonable geometric complexity
  - 128x128 is 16k vertices

VTF Demo



Translucency and Scattering

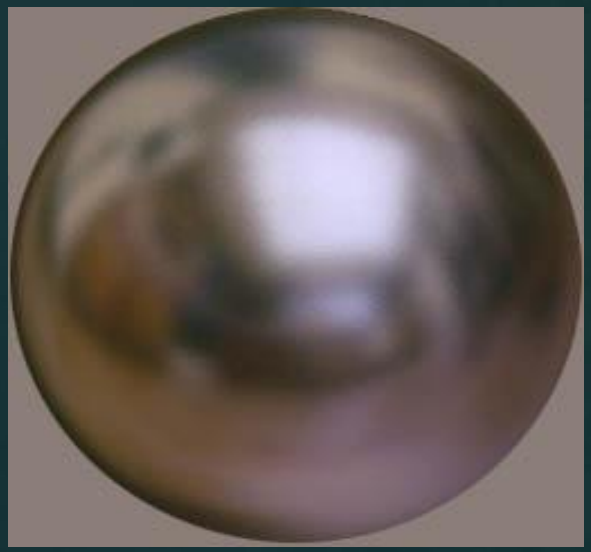
- All materials are translucent
  - Depends on light wavelength
- Light penetrates all surfaces to some degree
  - Different wavelengths have
    - Different penetration depths
    - Different falloff vs. depth
- If not absorbed or reflected, the light might scatter and exit somewhere else

Translucency Basics

- Optical Properties
  - Absorption (probability vs. distance)
  - Scattering (probability vs. distance & angle)
  - Impedance changes (reflection and refraction)
    - Optical impedance determines the index of refraction
- Everything absorbs and scatters
  - fluids, solids, gasses, even pure clean air
- Opacity, transparency, and translucency
  - Vary in the probability of absorption & scattering

Opactiy

- High probability of absorption & scattering
- Light takes short paths
- Light comes from surface, not interior



Transparency

- Low probability of absorption and scattering

Images courtesy of Leigh Van Der Byl

Translucency

- Low probability of absorption
- High probability of scattering

Leigh Van Der Byl

Real-Time Attitude

- Get the look. Forget the math
  - See Hoffman & Preetham for good scattering math
- Various techniques
  - Depth-map rendering for thickness & scattering
  - Texture-space diffusion
- Requirements
  - Artist friendly, content friendly
  - Fast as blazes
  - Fallbacks
  - Animate-able lighting and self-shadowing

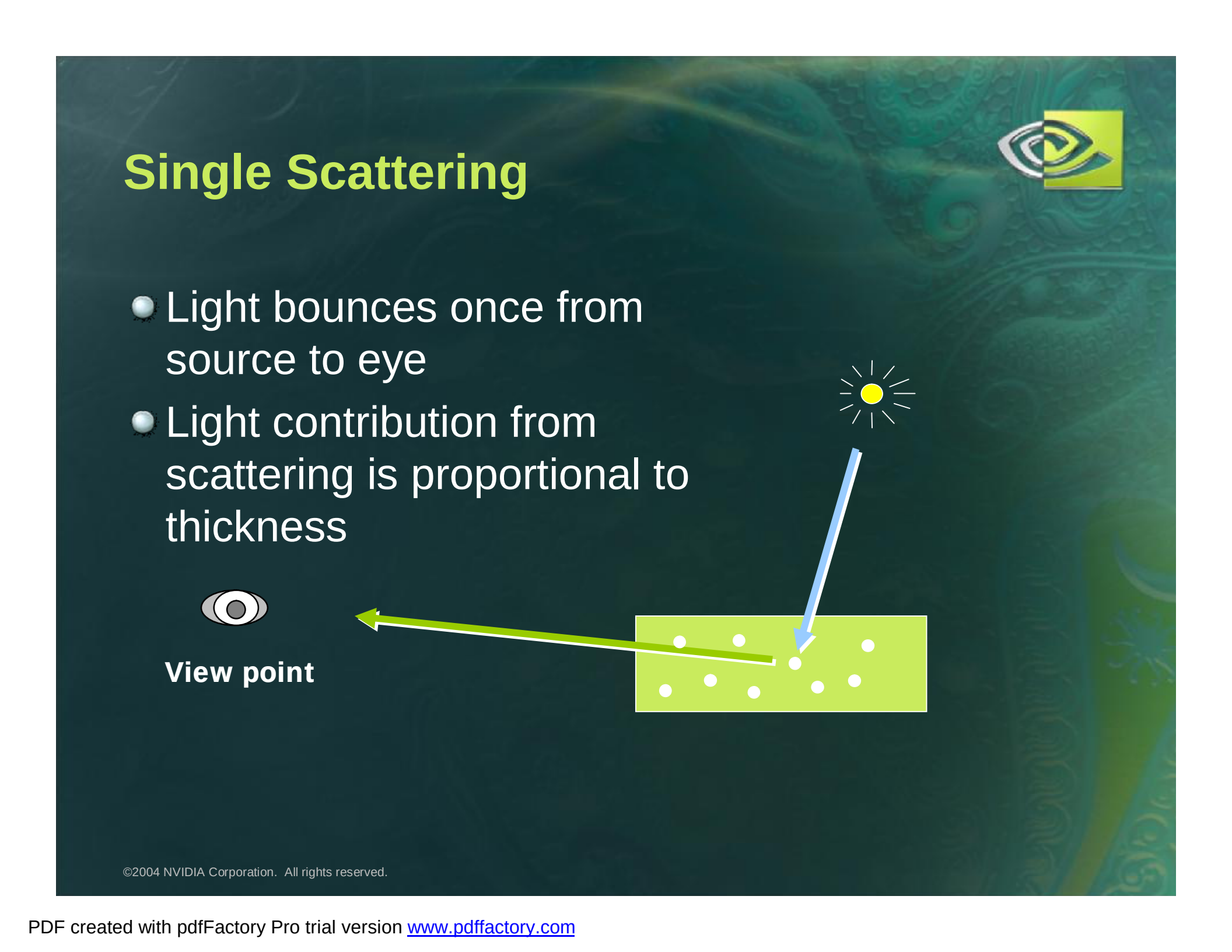
Depth Maps

- Fog is an ordinary polygon model
- Render-to-texture passes used to calculate distance through fog object
- ps.1.3
- ps.2.0 is faster
- ps.3.0 is faster++

Volume Fog Technique

- Inspired by Microsoft's "Volume Fog" DXSDK demo (Dan Baker)
- Inspired by [Mech01]
- Compute thickness through ordinary polygon objects from camera's P.O.V.
  - Render the depths of an object's front and back faces
- Derive color from thickness
- Great method for single scattering

Single Scattering



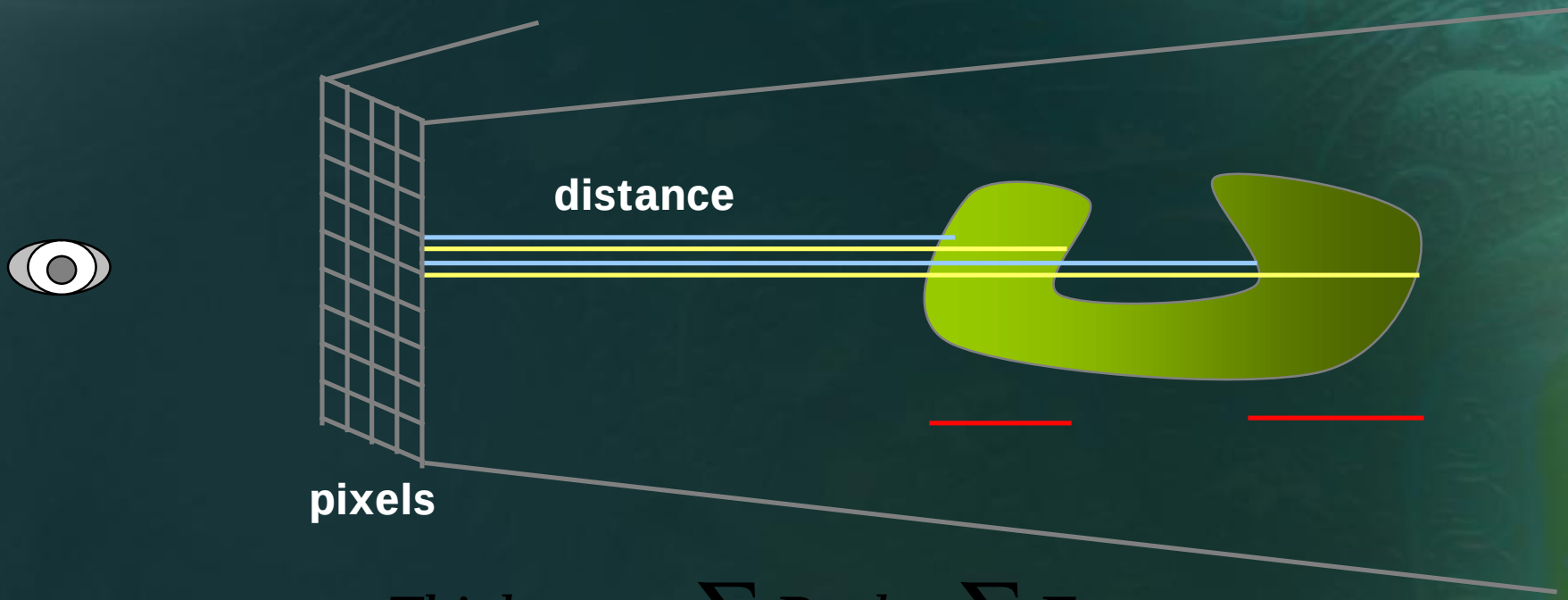
- Light bounces once from source to eye
- Light contribution from scattering is proportional to thickness

Rendering Thickness Per-Pixel

Thickness From Distances

THICKNESS = BACK - FRONT

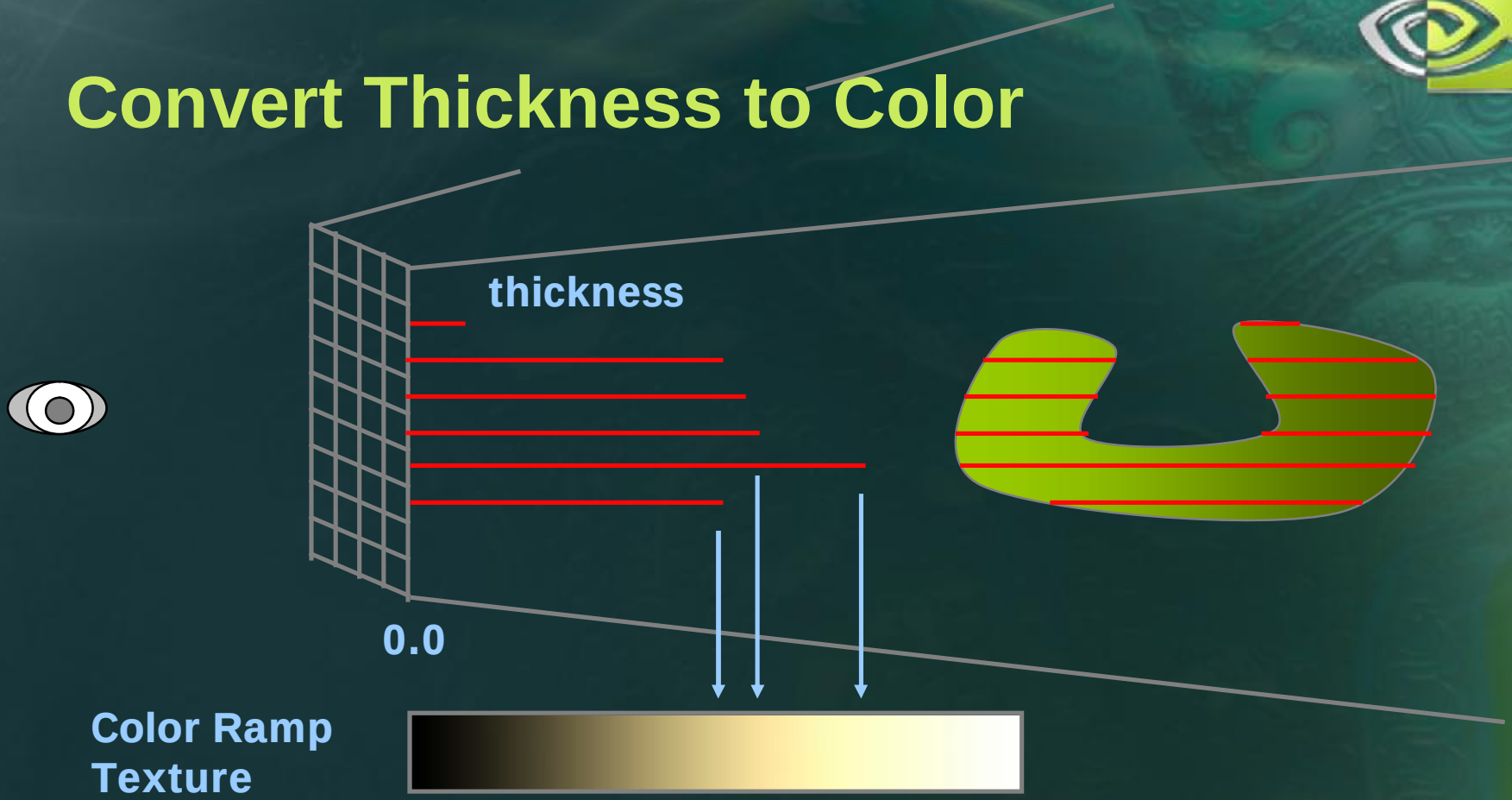
Rendering Thickness Per-Pixel



$$Thickness = \sum Back - \sum Front$$

- Thickness for any uniform density object is easy
- No Z-Buffer. Use additive blending

Convert Thickness to Color



- Thickness * scale **è** TexCoord.x
- Color ramp texture: Artistic or math
- Easy to control the look

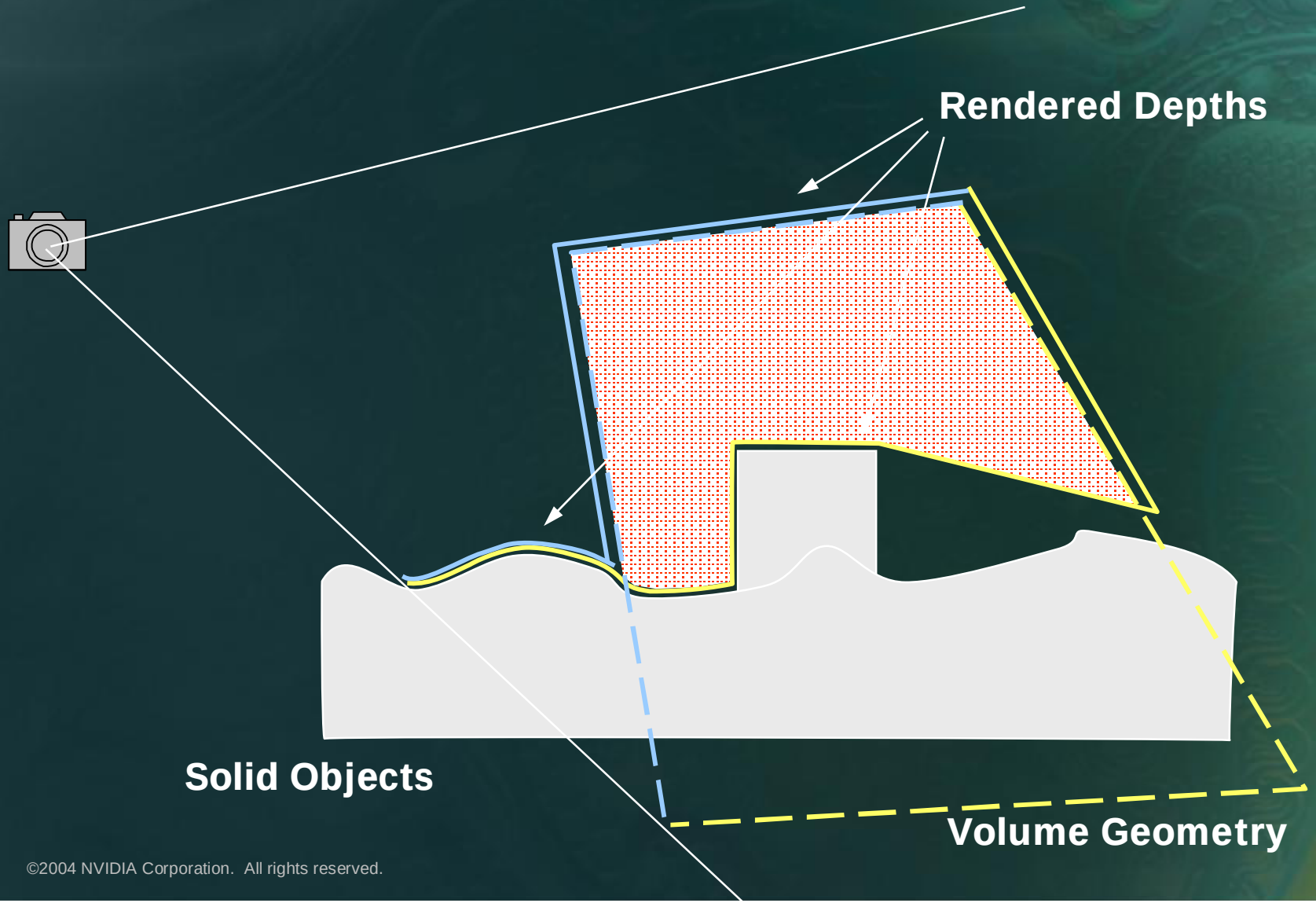
What About Intersection?

- Need depth to solid object
- Not depth to volume object faces

Intersection Solution

- Need depth of nearest solid object
  - Render it to a texture
  - Read the texture in a pixel shader
- As you render each of the volume object's faces
  - Pixel shader outputs lesser of
    - Depth of volume object triangle being drawn
    - Solid object depth (from texture) at pixel being drawn
  - Disable depth testing
  - Additive blend the output depth into the framebuffer

Intersection Solution



Intersection Method Advantages

Advantages

- Does not require stencil
- Does not require multi-pass

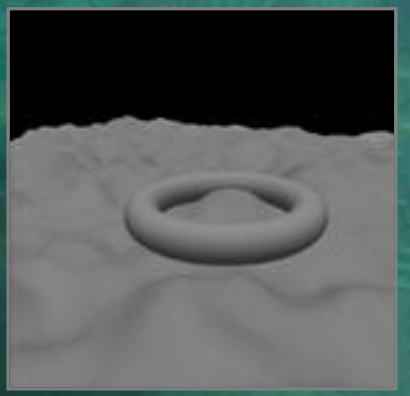
Disadvantages

- Must render depth of
  - Anything intersecting the volumes
  - Anything that can occlude the volumes
    - Can be avoided depending on the scene

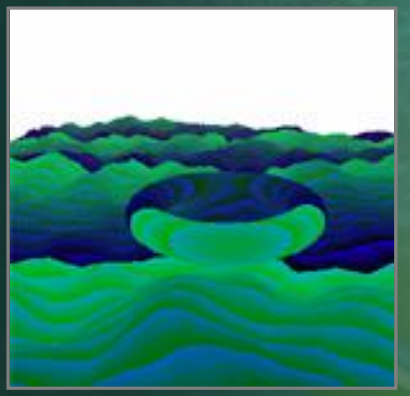
Steps: Pixel Shader 2.0

1. Render solid objects to backbuffer
   - Ordinary rendering
2. Render depth of solid objects that might intersect the fog volumes
   - To ARGB8 texture, “S”
   - RGB-encoded depth. High precision!
3. Render fog volume backfaces
   - To ARGB8 texture, “B”
   - Additive blend to sum depths
   - Sample texture “S” for intersection

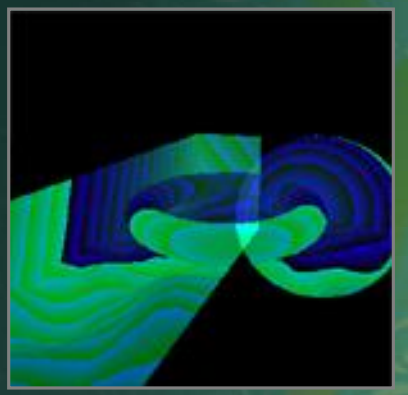
O



S



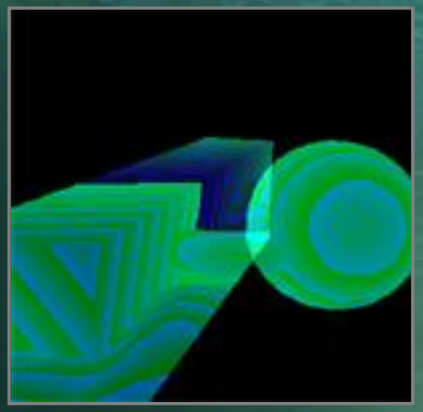
B

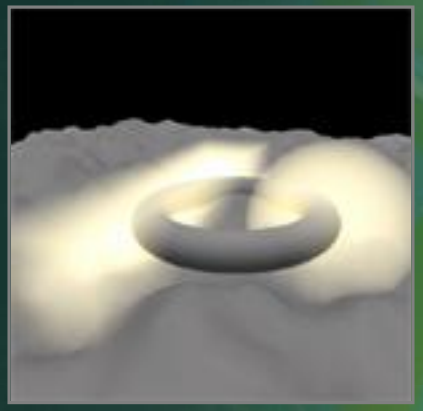


Steps: PS.2.0 contd.

4. Render fog volume front faces
   - To ARGB8 texture, “F”
   - Additive blend to sum depths
   - Sample texture “S” for intersections
5. Render quad over backbuffer
   - Samples “B” and “F”
   - Computes thickness at each pixel
   - Converts thickness to color using fog color ramp texture
   - Blends color to the scene
   - 5 instruction ps.2.0 shader

F





Final

PS.3.0 HW Improvements

- Front / back facing register
- Multiple Render Targets (MRT)
- Floating-point framebuffer blending

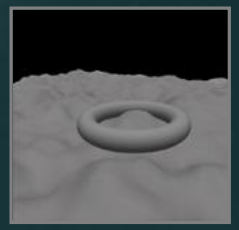
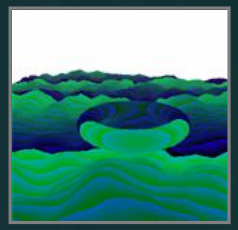
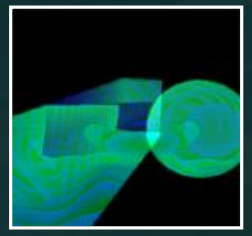
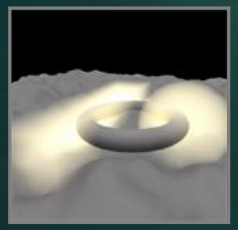
- Fewer passes
- Fewer render-target textures

PS.3.0 vs. PS.2.0

MRT

F/B register

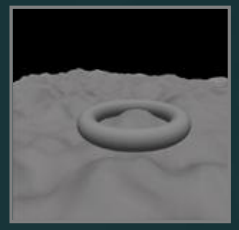
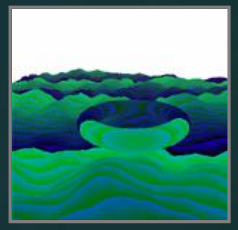
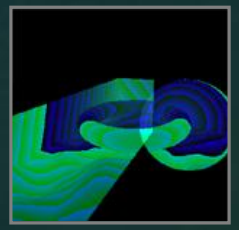
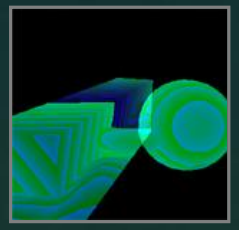
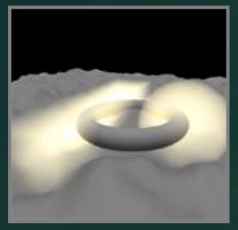
ps.3.0 HW

RT-Tex | RT-Tex 'O' | Floating point RT-tex | Backbuffer

3 Passes

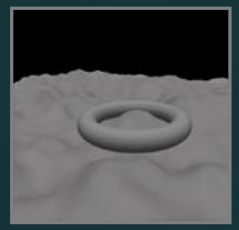
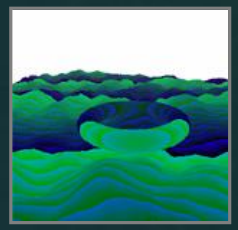
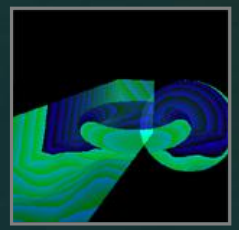
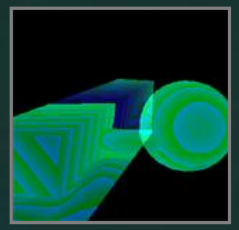
ps.2.0 HW

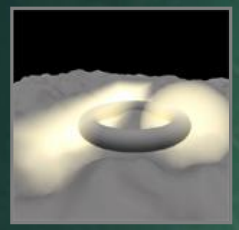






Backbuffer | RT-Tex 'O' | RT-Tex 'B' | RT-Tex 'F' | Backbuffer

5 Passes

ps.1.3 HW



6

Volume Fog Demo

Deferred Rendering

- What is it?
  - Render simple calculated parameters to offscreen buffers
    - Normal, Eye, Light vectors, etc
  - Integrate in a final lighting pass
- Why defer your rendering?
  - Overdraw with complex shaders is a performance nightmare
  - Final pass has a depth complexity of 1
    - PS3.0 branching for increased performance gain
    - MRT to output all lighting data in one pass

DX9 Instancing API

- What is it?
  - Allows a single draw call to draw multiple instances of the same model
  - Allows you to avoid DIP calls and minimize batching overhead
- What is required to use it?
  - DX 9.0c
  - VS/PS 3.0 capable graphics device

Why use it?

- Speed.
  - The single most common performance bottleneck in most games today is draw calls
- Yeah. We all know draw calls are bad.
  - But world matrices often force us to separate draw calls
- The instancing API pushes the per instance draw logic down into the driver/hardware
  - Saves draw call overhead in both D3D and Driver
  - Allows the driver to ensure minimal state changes between instances

Example using Instancing

When to use instancing?

- Scene contains many instances of the same model
  - Forest of Trees, Particles, Sprites, Soldiers
- Instances move or change
  - Thus a static buffer isn't feasible
- Less useful if your batch size is large
  - >1k polys per draw is less draw call bound
  - There is some fixed overhead to using instancing
    - Extra vertex attributes
    - Extra Vertex Shader operations

How does it work?

- DX Instancing API uses the vertex stream frequency divider (VSF) API

- Primary stream is a single copy of the model data
- Secondary stream(s) contain per instance data and stream pointer is advanced each time the primary stream is rendered.
  - Uses **IDirect3DDevice9::SetStreamSourceFreq** entry point

How does it work? (2)

- Modulus on #0, Divider on #1
  - Loops over stream 0
    - Each loop represents one “instance”
  - Divides stream 1
    - This means it only increments after each “instance”

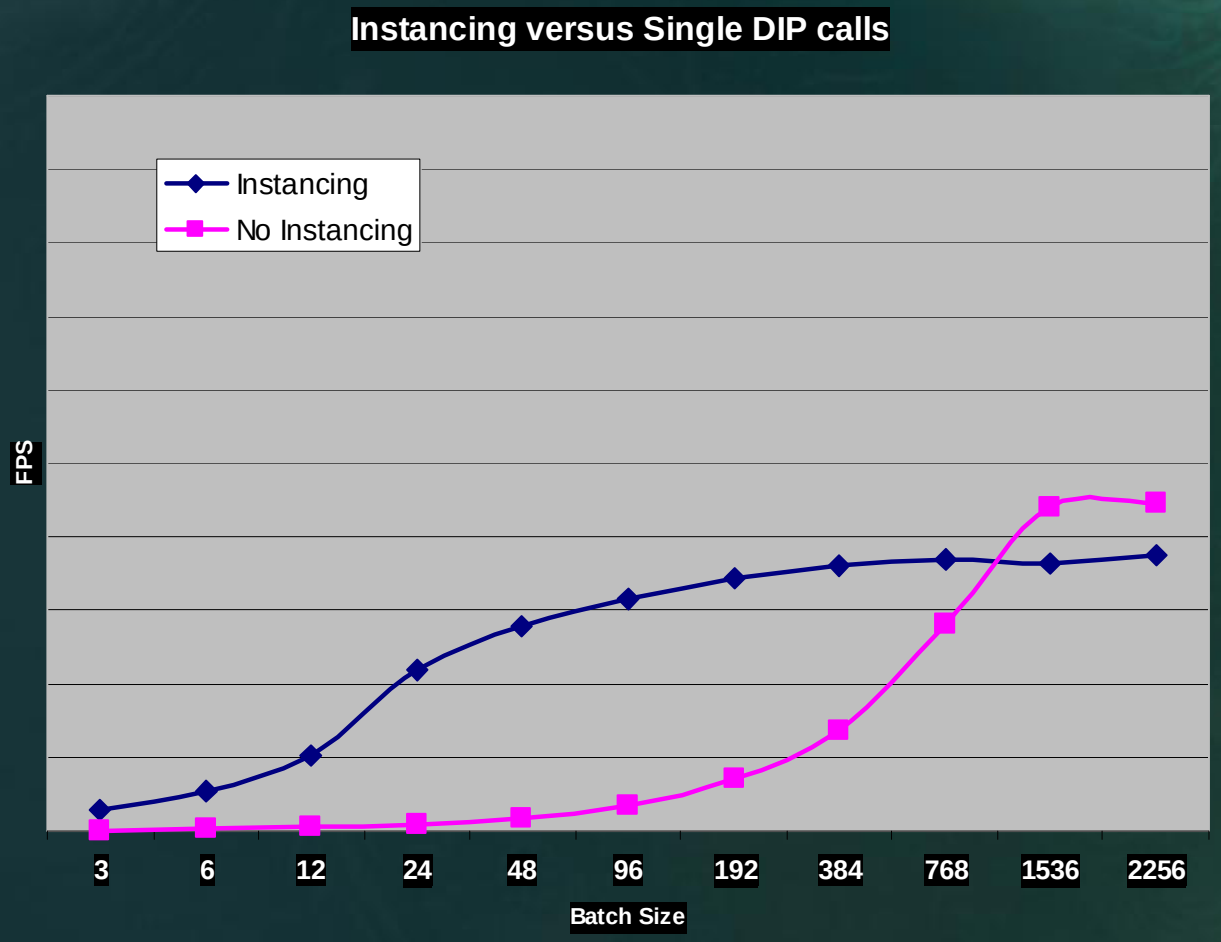
Simple Instancing Example

- 100 poly trees
  - Stream 0 contains just the one tree model
  - Stream 1 contains model WVP transforms
    - Possibly calculated per frame based on the instances in the view
  - Vertex Shader is the same as normal, except you use the matrix from the vertex stream instead of the matrix from VS constants
- If you are drawing 10k trees that's a lot of draw call savings!
  - You could manipulate the VB and pre-transform verts, but it's often tricky, and you are replicating a lot of data

The Million Poly Test

- A real test that draws instances of a mesh.
- The changing axis is the # of polygons per mesh.
- The scene total poly count is fixed at 1 million triangles
  - Thus as the mesh size goes up, the # of instances goes down.
- Very simple shaders
  - More complex shaders will change the behavior of the scene.
    - Big pixel shaders may become bottleneck at a certain mesh size, thus making instancing always at least as fast as single DIP.

Million Poly Test Results



Million Poly Test Results 2

- For small batch sizes, can provide an extreme win as it gives savings PER DRAW CALL.
- There is a fixed overhead from adding the extra data into the vertex stream.
  - Vertex attribute fetch is often a limiting factor
    - More than doubled our vertex stride
  - This overhead may or may not be an issue depending on other bottlenecks in the system
- The sweet spot will change based on many factors (CPU Speed, GPU speed, engine overhead, etc)

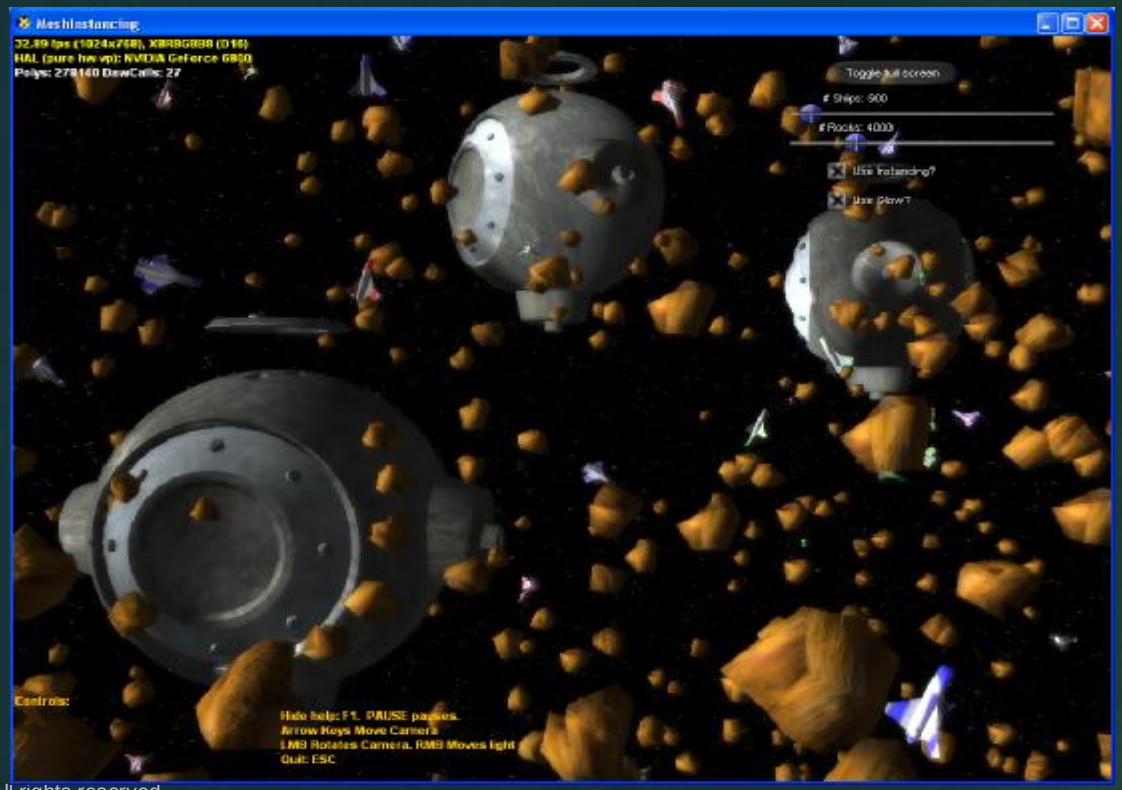
Other Ideas

- Use VS constants
  - Break up VS constant memory into sections of data used by each instance.
  - Encode an “index” in the instance stream to offset into this section of constant memory.
  - This will help alleviate the performance hit of too many vertex shader attributes.
- Differently textured instances
  - Bind multiple textures and use instance data to lerp between them.
  - Use texture pages, and put UV offsets into 2nd stream

Other Ideas(2)

- Colored Instances
  - Push down instance coloring and vary the vertex colors of each instance
- <your idea here>
  - If you can reduce draw calls, you'll be sitting pretty. Remember to be aware of extra overhead like shader muls and vertex attributes!
- Remember that you don't have to reduce your scene to one draw call.
  - If you can batch every 4 instances together, then you've ¼'d your # of draw calls!

Instancing Demo



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

More Instancing Performance

- Instancing Considerations whitepaper
  - A more complete discussion of performance considerations using instancing
  - http://developer.nvidia.com
  - Talks about other methods of reducing draw calls
  - Lots of graphs. :P

Questions?

More info?

- http://developer.nvidia.com
- NVIDIA SDK
- bdudash@nvidia.com
  - Feel free to ask me questions in English or 日本語.

References

[Mech01] Radomir Mech, “Hardware-Accelerated Real-Time Rendering of Gaseous Phenomena.”