
GDC
09
learn
network
inspire
www.GDConf.com
Game Developers Conference®
March 23-27, 2009 | Moscone Center, San Francisco

DirectX 10/11 Visual Effects

Simon Green, NVIDIA

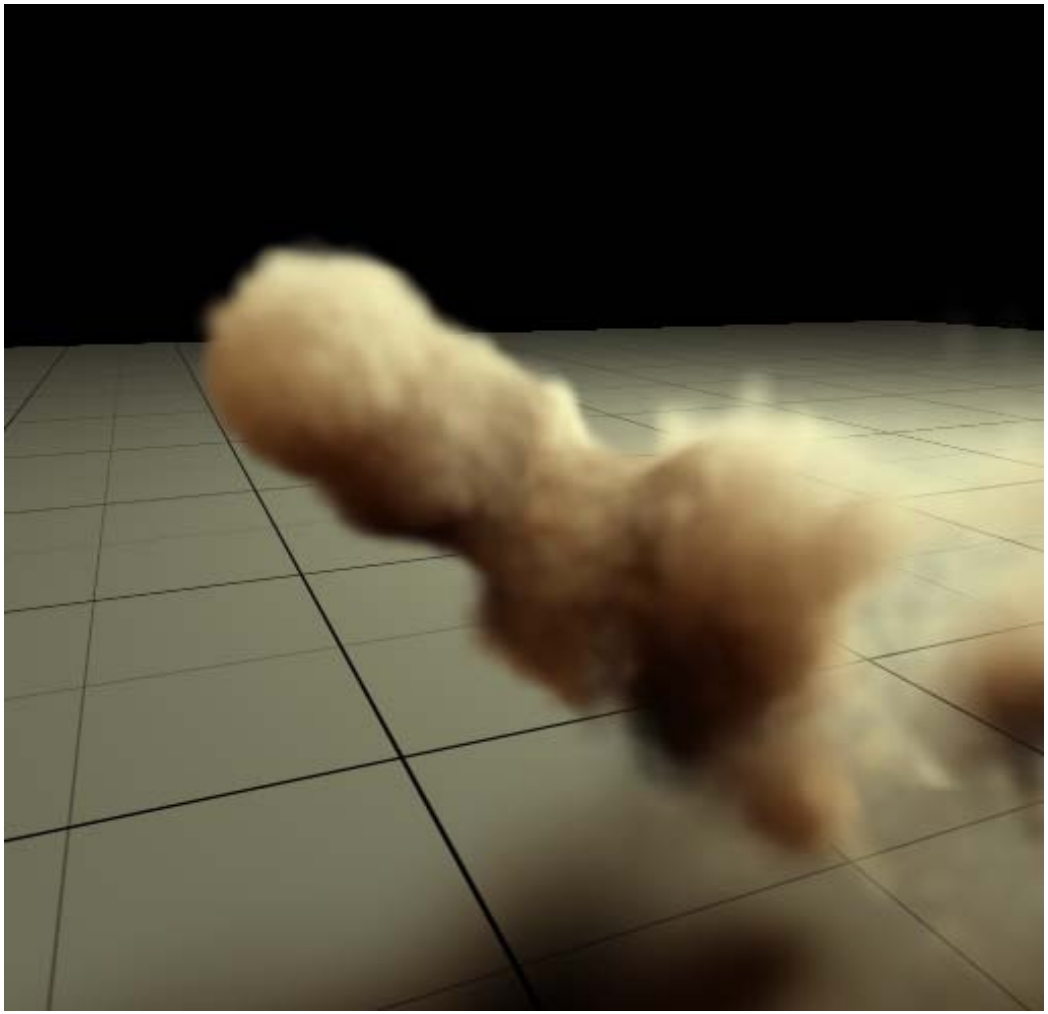
Introduction

- Graphics hardware feature set is starting to stabilize and mature
- But new general-purpose compute functionality (DirectX Compute Shader)
  - enables new graphical effects
  - allows more of game computation to move to the GPU
  - Physics, AI, image processing
- Fast hardware graphics combined with compute is a powerful combination!
- Next generation consoles will likely also follow this path

Overview

- DirectX 10 Effects
  - Volumetric Particle Shadowing
  - Horizon Based Ambient Occlusion (HBAO)
- DirectX Compute Shader
  - Brief introduction
  - Compute Shader on DX10 hardware
  - Demos

Volumetric Particle Shadowing



Particle Systems in Today's Games

- Commonly used for smoke, explosions, spark effects
- Typically use relatively small number of large particles (10,000s)
- Rendered using point sprites with painted or pre-rendered textures
  - Use animation / movies to hide large particles
- Sometimes include some lighting effects
  - normal mapping
- Don't interact much with scene
  - No collisions

Particle Systems in Today's Games

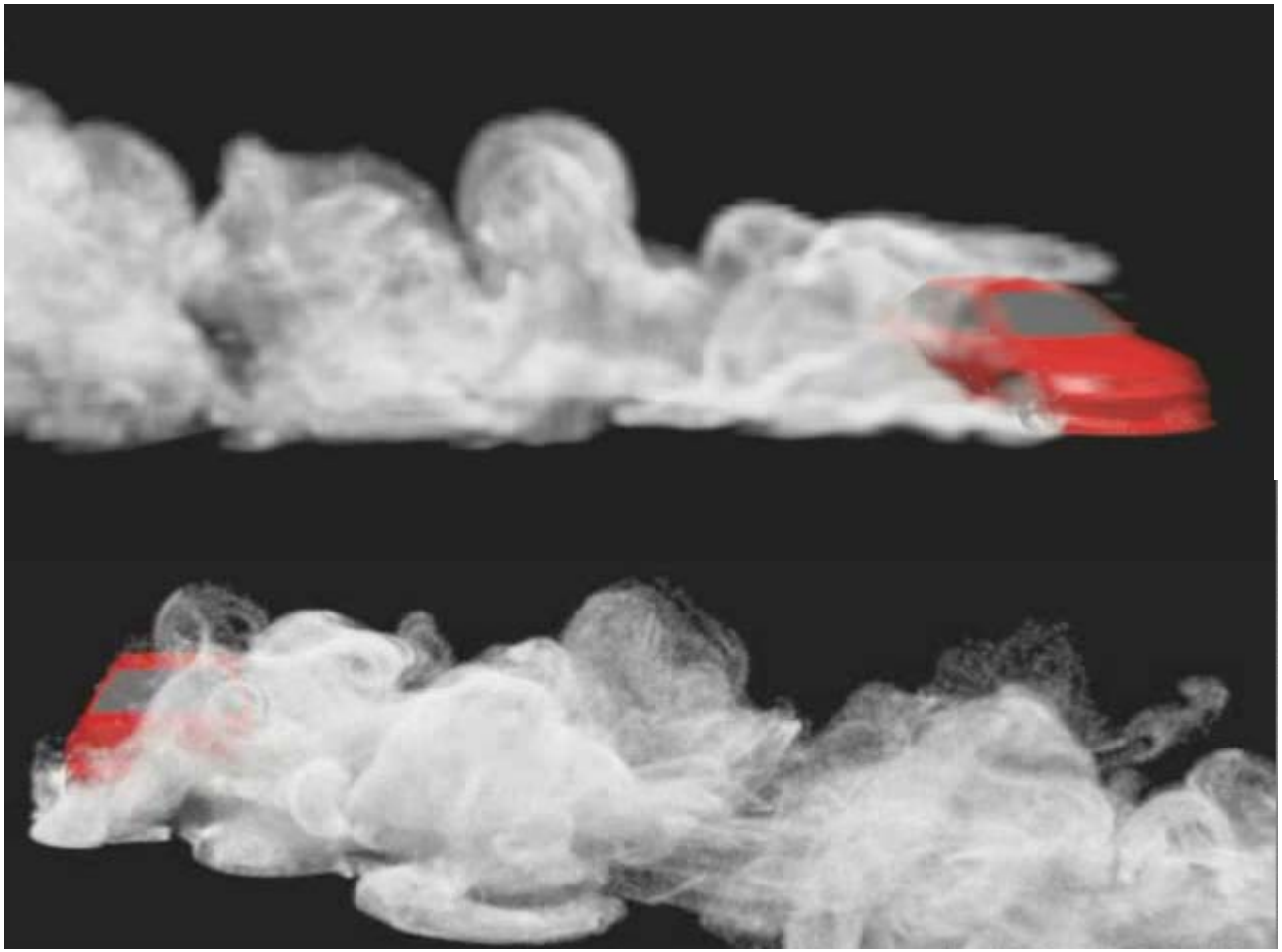
- Can get some great effects with current technology

World in Conflict, Ubisoft / Massive

Tomorrow's Particle Systems

- Will likely be more similar to particle effects used in film
- Millions of particles
- Driven by physical simulations
  - With artist control
- Interaction (collisions) with scene and characters
- Simulation using custom compute shaders or physics middleware
- High quality shading and shadowing

Tomorrow's Particle Systems - Example



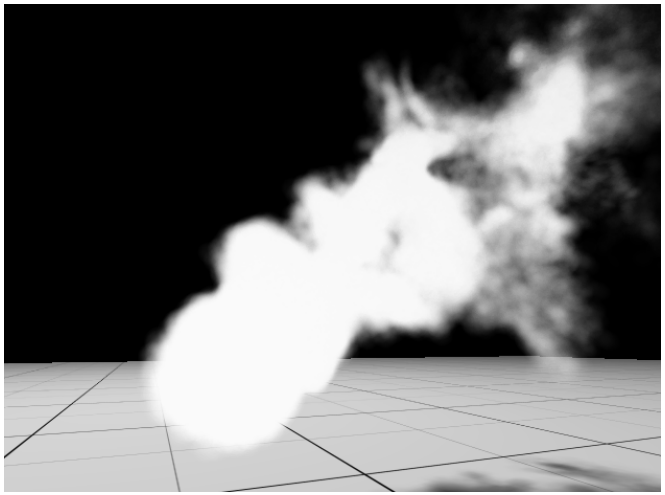
Low Viscosity Flow Simulations for Animation, Molemaker et al., 2008

Volume Shadowing

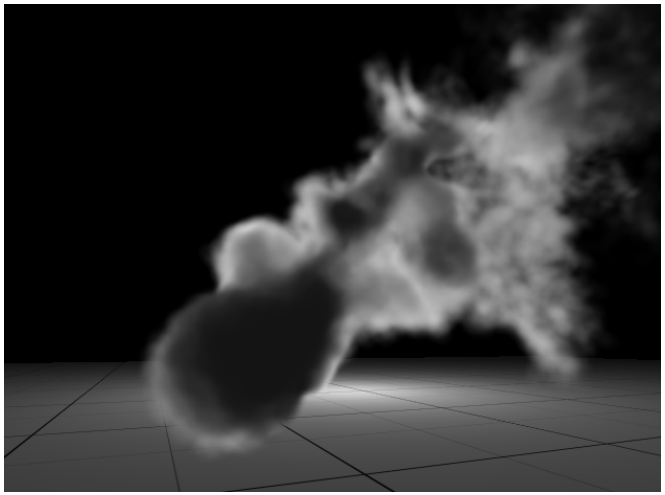
- Shadows are very important for diffuse volumes like smoke
  - show density and shape
- Not much diffuse reflection from a cloud of smoke
  - traditional lighting doesn't help much
- Usually achieved in off-line rendering using deep shadow maps
  - still too expensive for real time

Volume Shadowing

Before



After

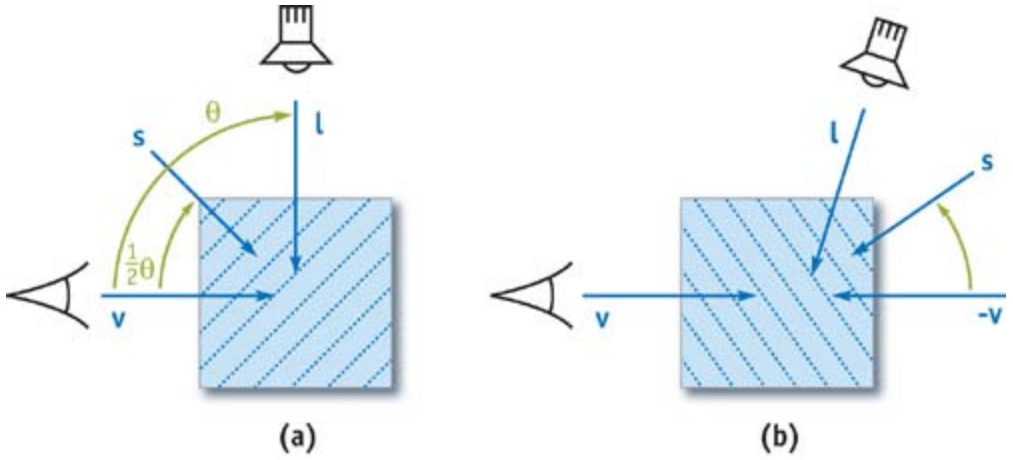


Half-Angle Slice Rendering

- Very simple idea
- Based on a volume rendering technique by Joe Kniss et. Al [1]
- Only requires sorting particles along a given axis
  - you're probably already doing this
- Plus a single 2D shadow texture
  - no 3D textures required
- Works well with simulation and sorting done on GPU (compute)

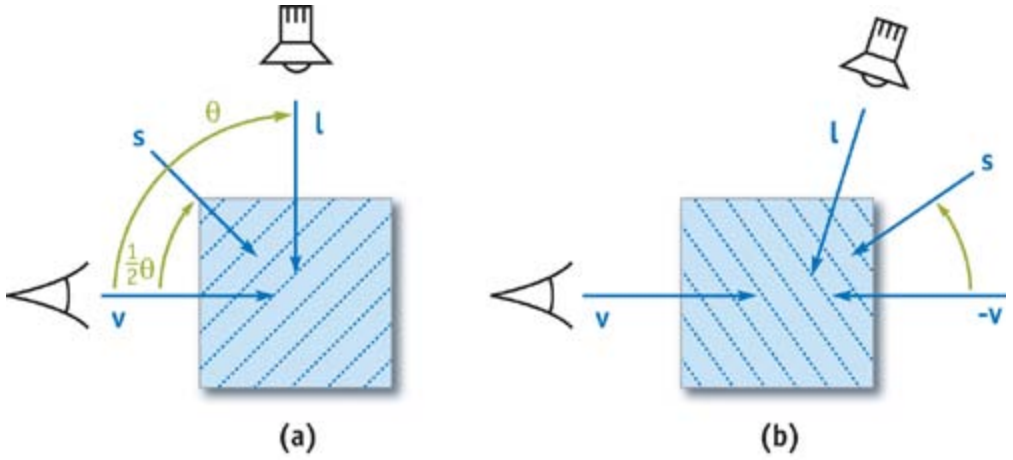
Half-Angle Slice Rendering

- Calculate vector half way between light and view direction
- Render particles in slices perpendicular to this half-angle vector



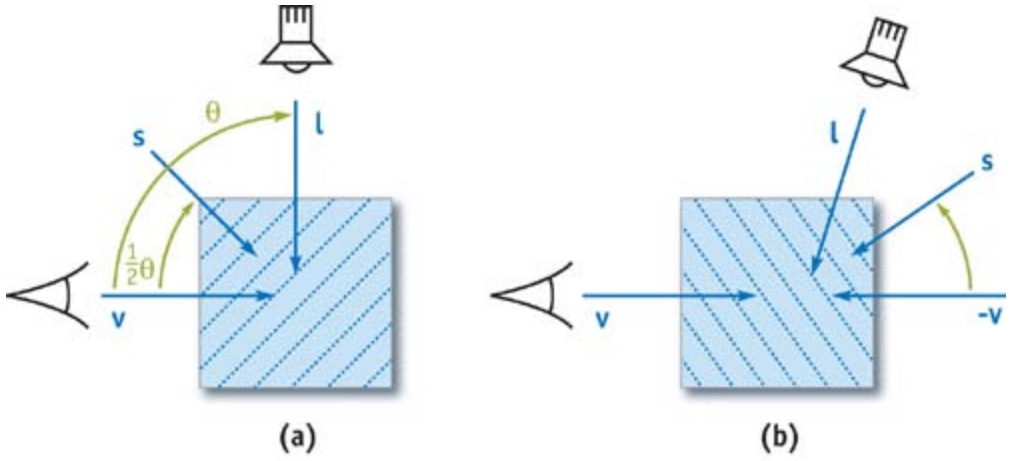
Why Use The Half-Angle?

- Same slices are visible to both camera and light
- Lets us accumulate shadowing to shadow buffer *at the same time* as we are rendering to the screen
- First render slices from light POV to shadow map, and then to the screen



Half-Angle Slice Rendering

- Need to change rendering direction (and blend mode) based on dot(l, v)
- if (dot(l, v) > 0) - render front-to-back (case a)
- if (dot(l,v) < 0) – render back-to-front (case b)
- Always render from front-to-back w.r.t. light



Rendering Slices

- Sort particles along half-angle axis
  - based on dot(p, s)
  - can be done very quickly using compute shader
- Choose a number of slices
  - more slices improves quality
  - but causes more draw calls and render target switches
- batchSize = numParticles / numSlices
- Render slices as batches of particles starting at i*batchSize
- Render particles as billboards using Geometry Shader

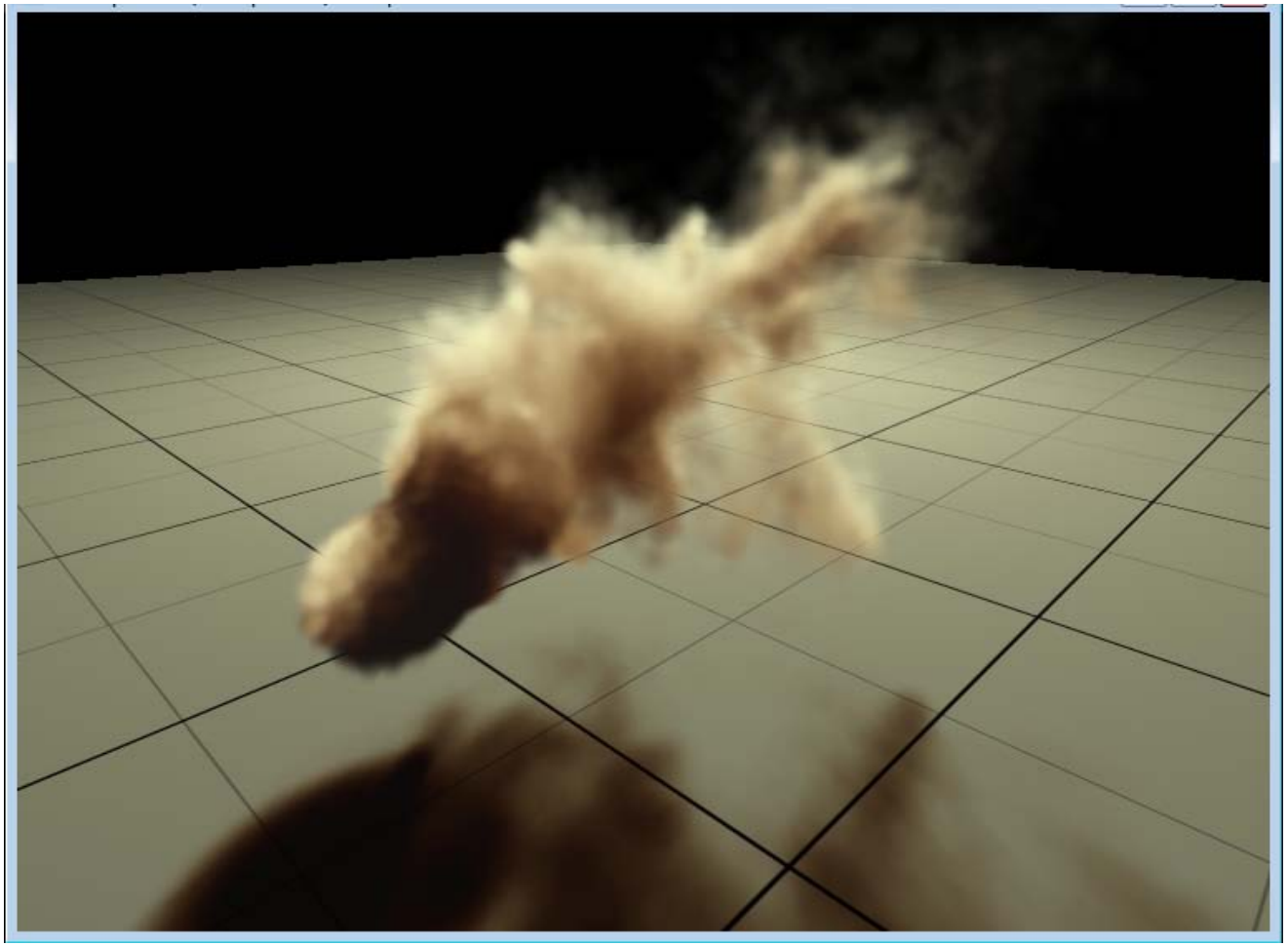
Pseudo-Code

```
If (dot(v, l) > 0) {
   h = normalize(v + l)
   dir = front-to-back
} else {
   h = normalize(-v + l)
   dir = back-to-front
}
sort particles along h
batchSize = numParticles / numSlices
for(i=0; i<numSlices; i++) {
   draw particles to screen
       looking up in shadow buffer
   draw particles to shadow buffer
}
```

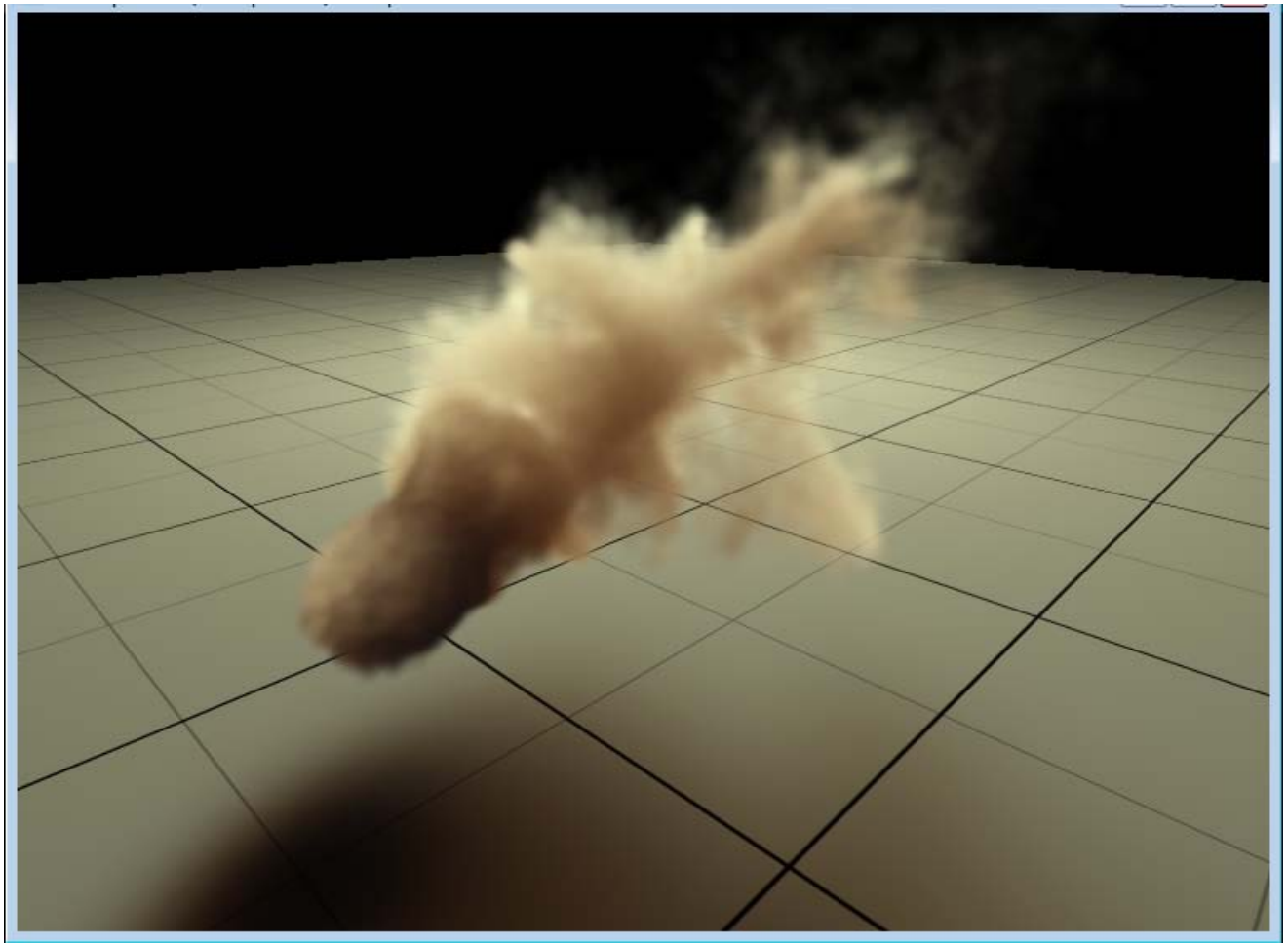
Tricks & Tips

- Shadow buffer can be quite low resolution (e.g. 256 x 256)
- Can also use final shadow buffer to shadow scene
- Screen image can also be rendered at reduced resolution (2 or 4x)
  - Requires destination alpha for front-to-back (under) blending
- Can blur shadow buffer at each iteration to simulate scattering:

Without Scattering



With Scattering



Particle Shadows in Shattered Horizon ™

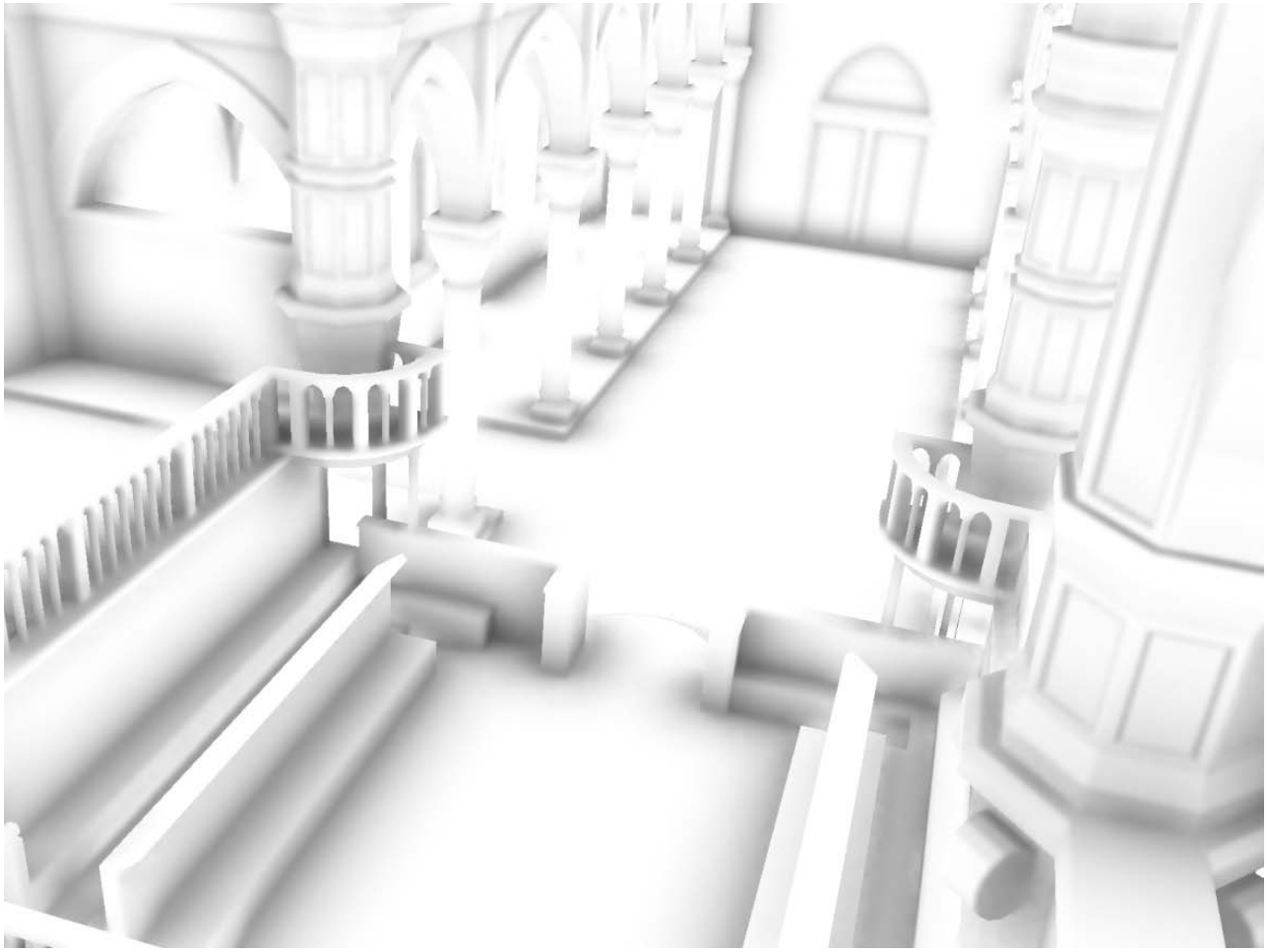
Courtesy Futuremark Games Studio

Demo

Volume Shadowing - Conclusion

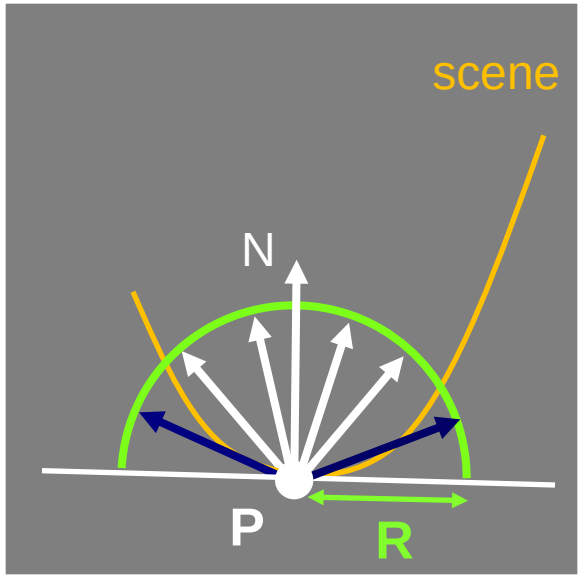
- Very simple to add to existing particle system renderer
- Only requires depth-sorting along a different axis
  - Can be done using CPU radix sort or Compute
- Plus a single 2D shadow map
- Can simulate millions of particles on the GPU in real-time
- DirectX 10 SDK sample coming soon

Horizon Based Ambient Occlusion



Ambient Occlusion

- Simulates lighting from hemi-spherical sky light
- Occlusion amount is % of rays that hit something within a given radius R
- Usually solved offline using ray-tracing



Ambient Occlusion

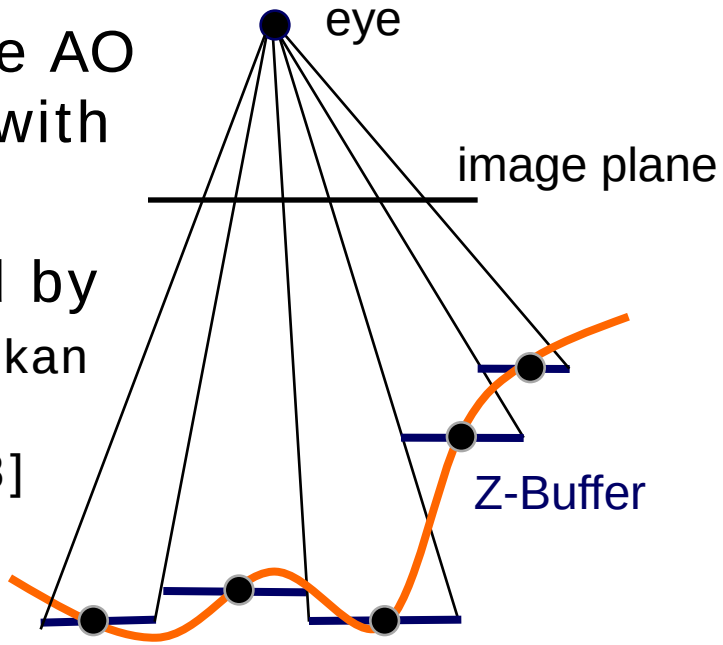
- Gives perceptual clues to depth, curvature and spatial proximity

Without AO

With AO

Screen Space Ambient Occlusion (SSAO)

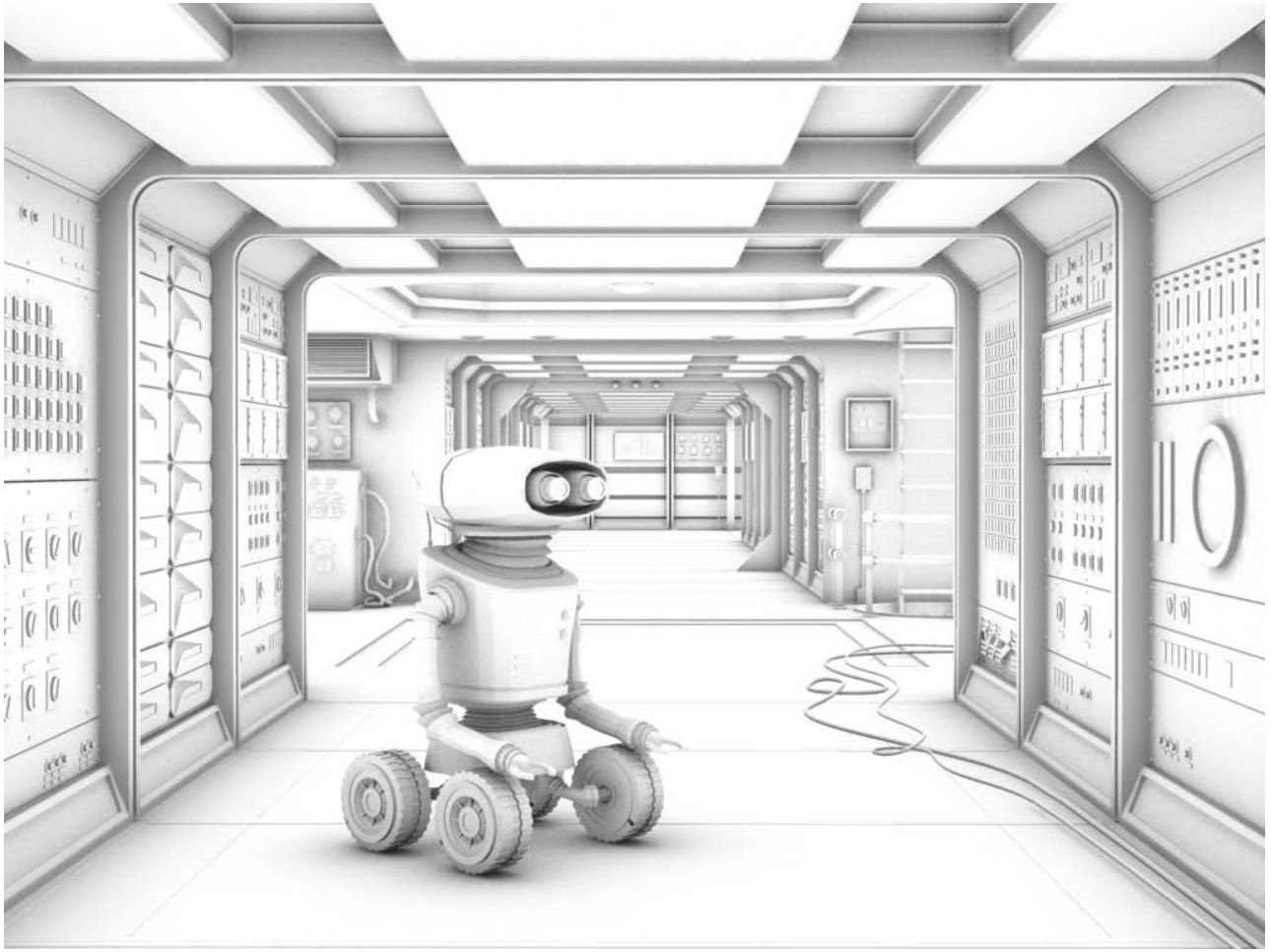
- Has become very popular!
- Renders approximate AO for dynamic scenes with no precomputation
- Approach introduced by
  [Shanmugam and Orikan 07] [Mittring 07] [Fox and Compton 08]
- Input - Z-Buffer
  + normals
- Z-Buffer = Height field
  z = f(x,y)



Horizon Based Ambient Occlusion (HBAO)

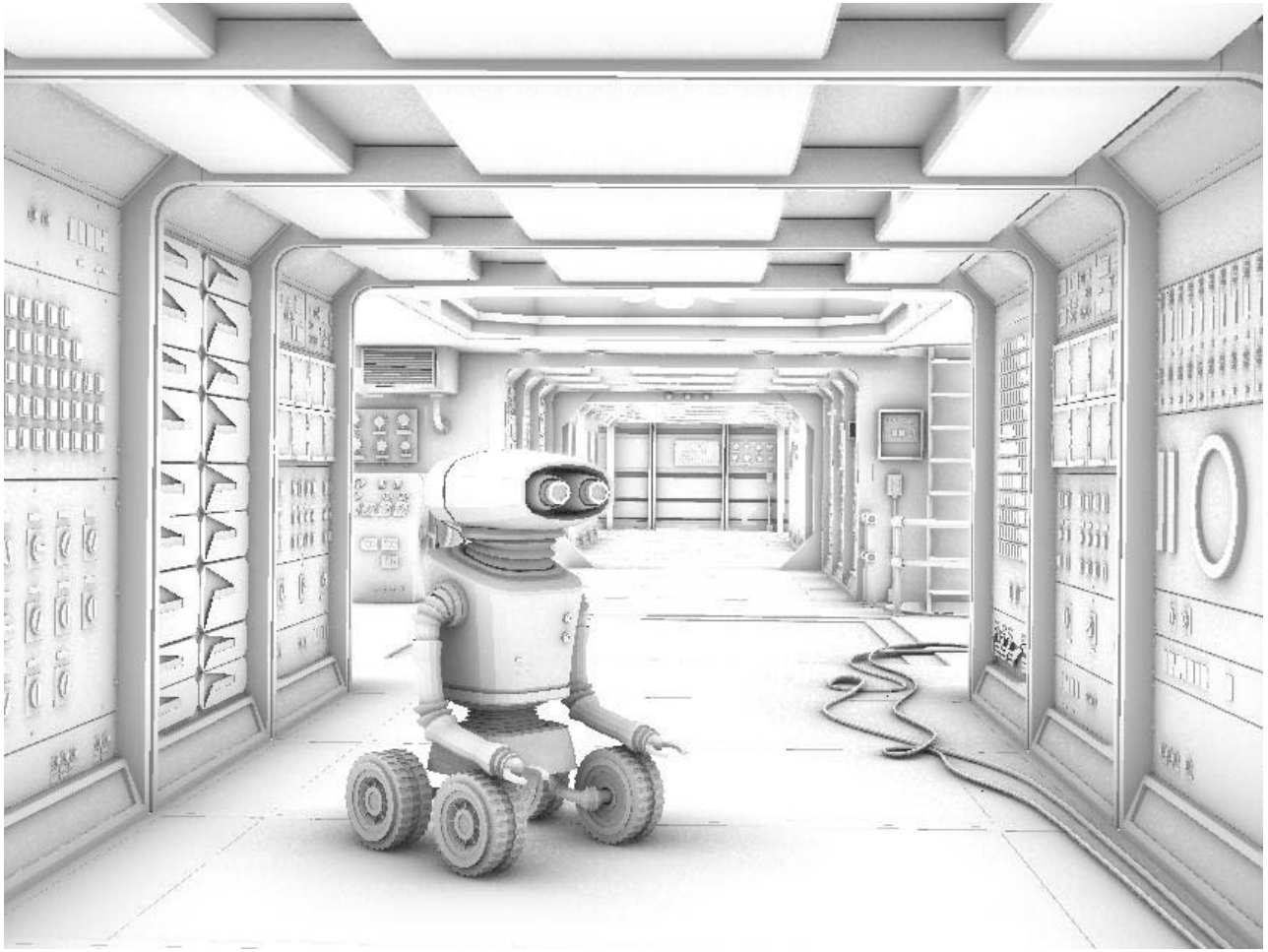
- SSAO technique
- Based on ideas from horizon mapping [Max 1986]
- Goal = approximate the result of ray tracing the depth buffer in 2.5D
- Scalable – performance vs. quality
- Details in ShaderX7 [2]

Ray Traced AO



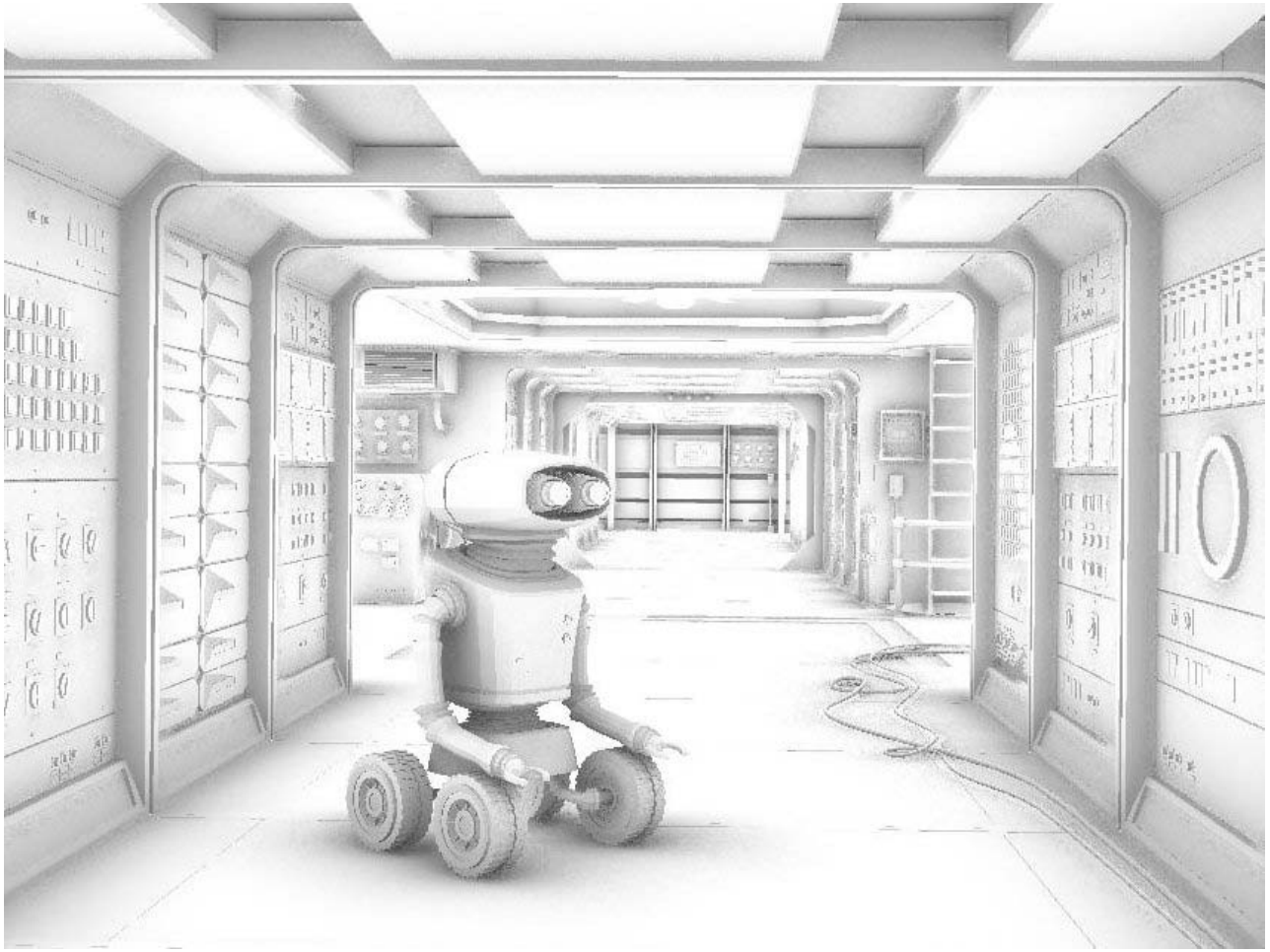
Several minutes with Gelato and 64 rays per pixel

HBAO with large radius



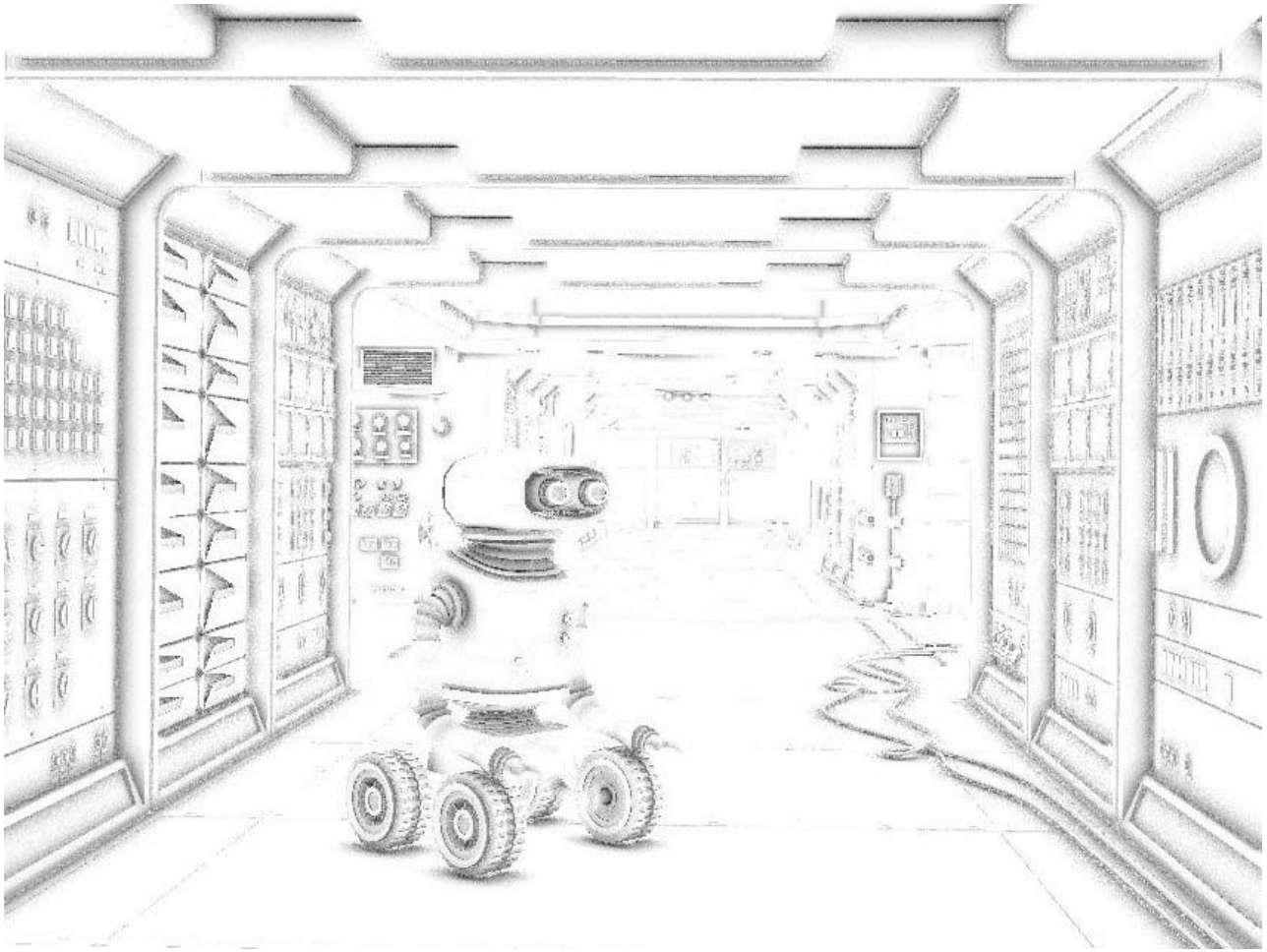
Interactive HBAO with 16x64 depth samples per pixel

HBAO with large radius



Interactive HBAO with 16x16 depth samples per pixel

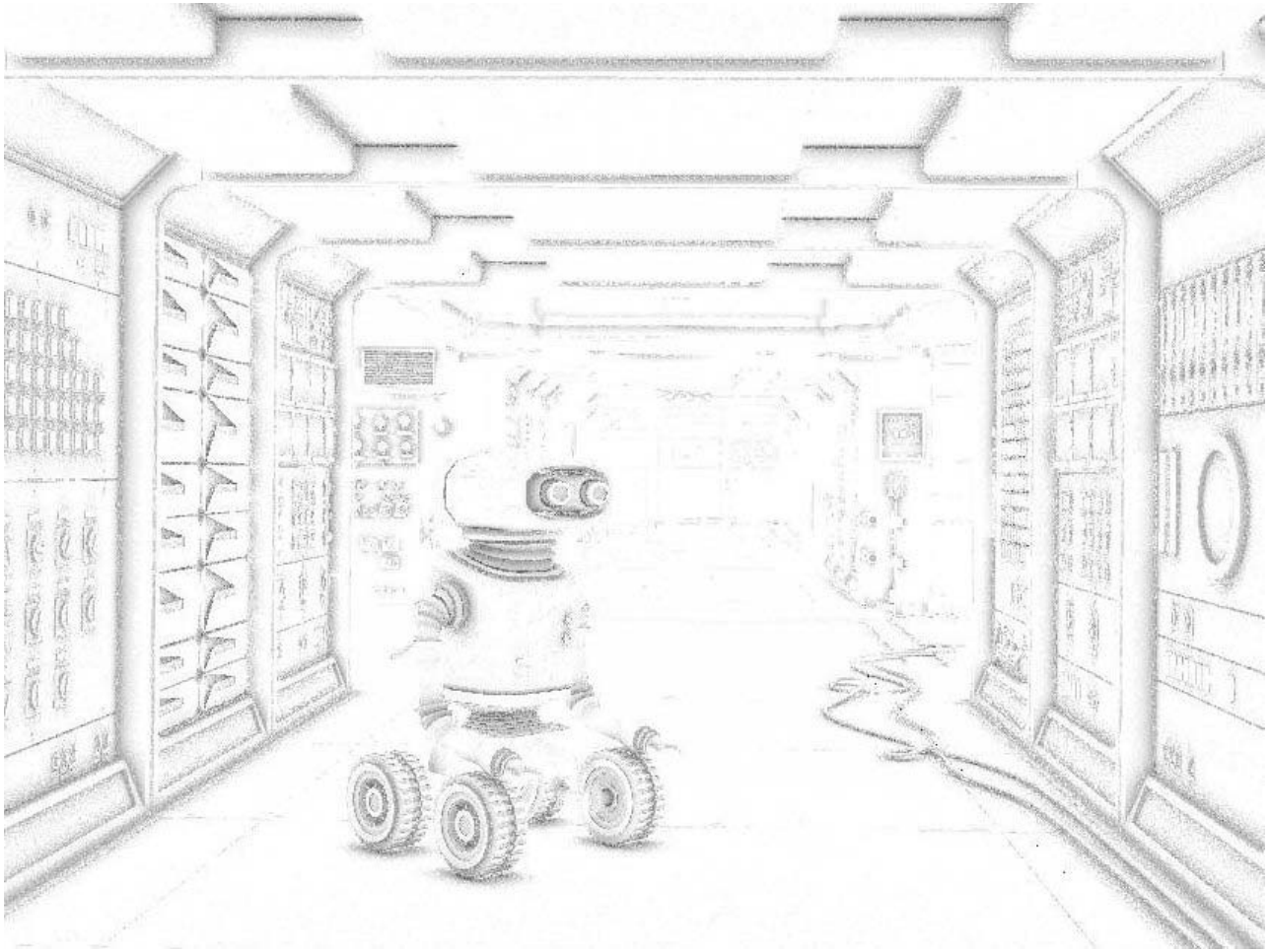
HBAO with small radius



“Crease shading” look

with 6x6 depth samples per pixel

HBAO with small radius



“Crease shading” look

with 4x8 depth samples per pixel

Integration in Games

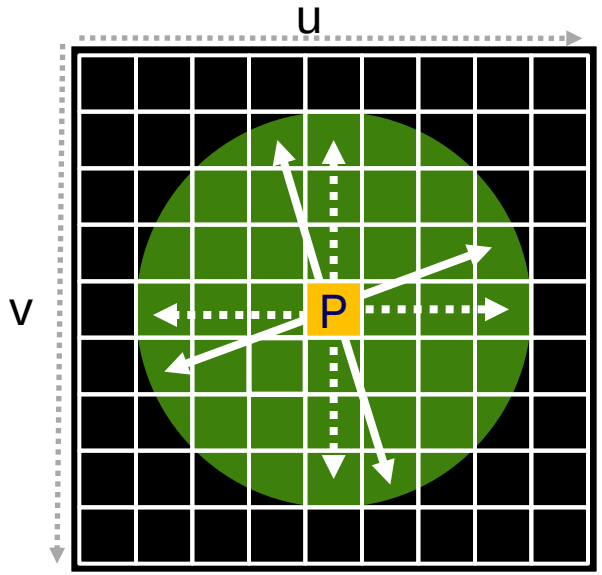
- Implemented in DirectX 9 and DirectX 10
- Has been used successfully in several shipping games

Age Of Conan Without HBAO

Age Of Conan with HBAO

Sampling the Depth Image

- Estimate occlusion by sampling depth image
- Use uniform distribution of directions per pixel
  - Fixed number of samples / dir
- Per-pixel randomization
  - Rotate directions by random per-pixel angle
  - Jitter samples along ray by a random offset



Noise

- Per-pixel randomization generates visible noise

AO with 6 directions x 6 steps/dir

Cross Bilateral Filter

- Blur the ambient occlusion to remove noise
- Depth-dependent Gaussian blur

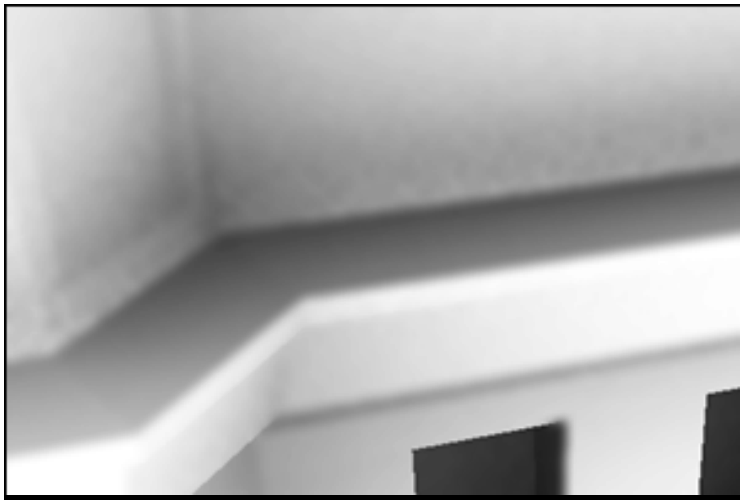
  [Petschnigg et al. 04]
  [Eisemann and Durand 04]

  - Reduces blurring across edges
- Although it is a non-separable filter, we apply it separately in the X and Y directions

  No significant artifacts visible

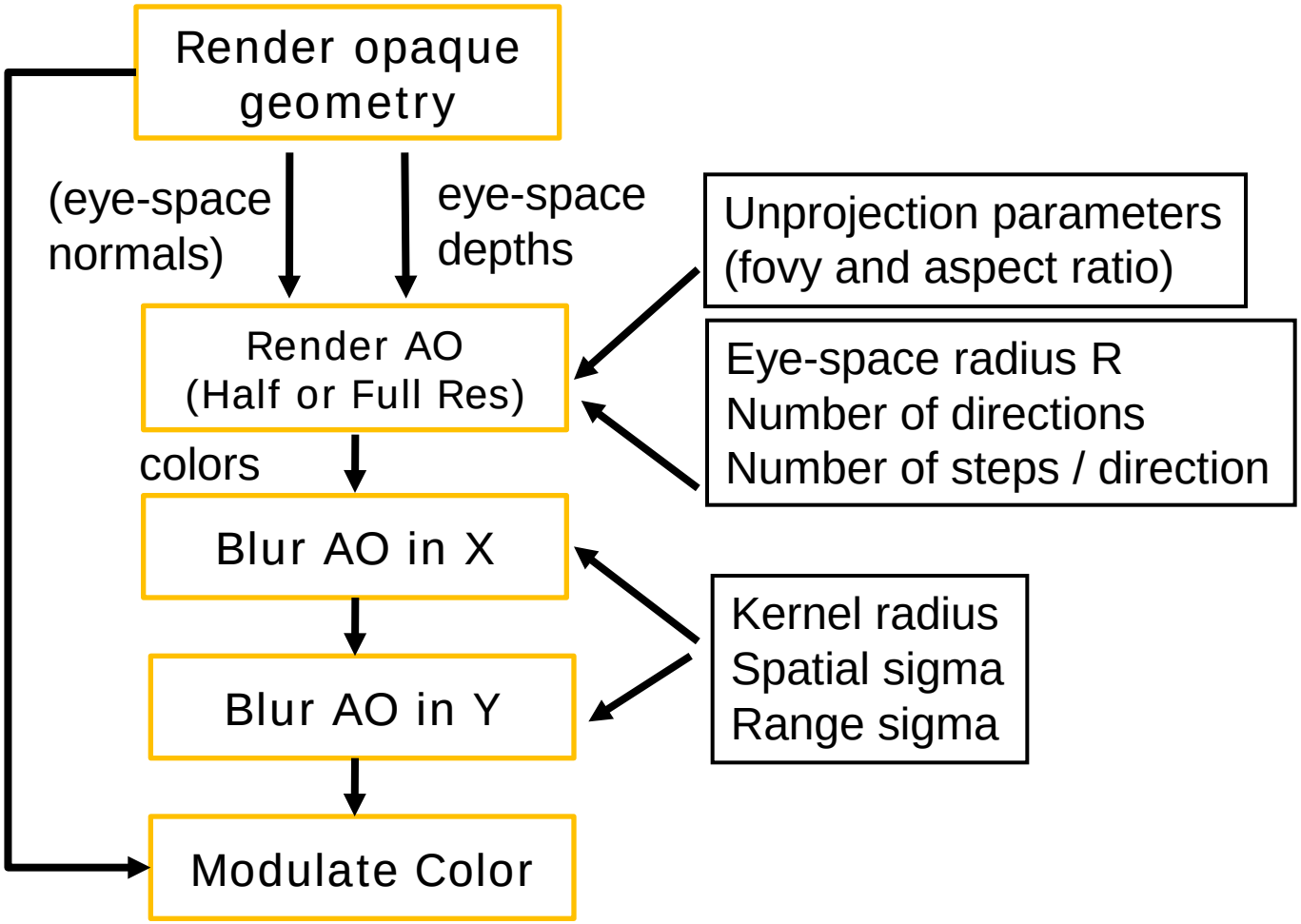
Cross Bilateral Filter

Without Blur

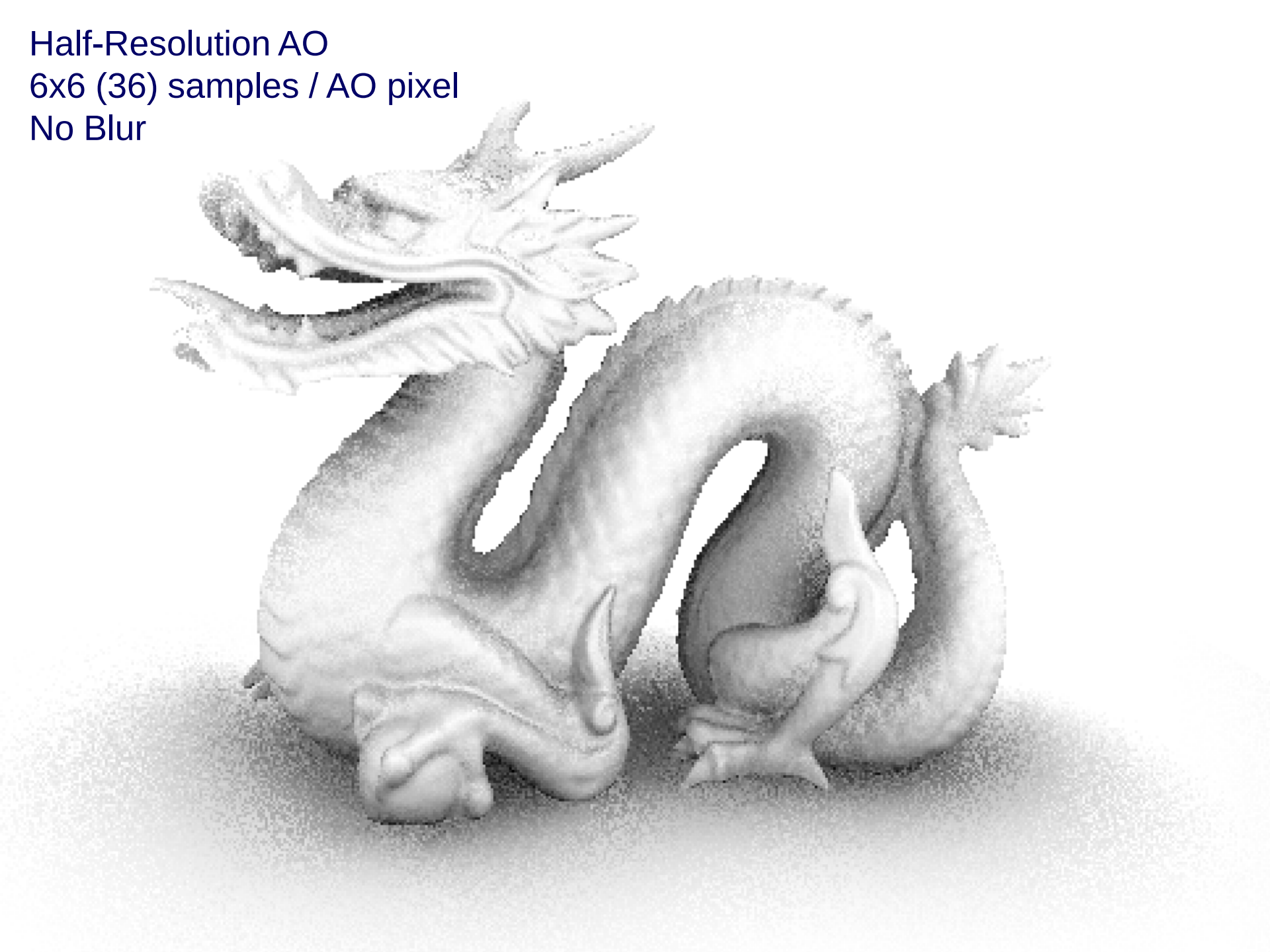


With 15x15 Blur

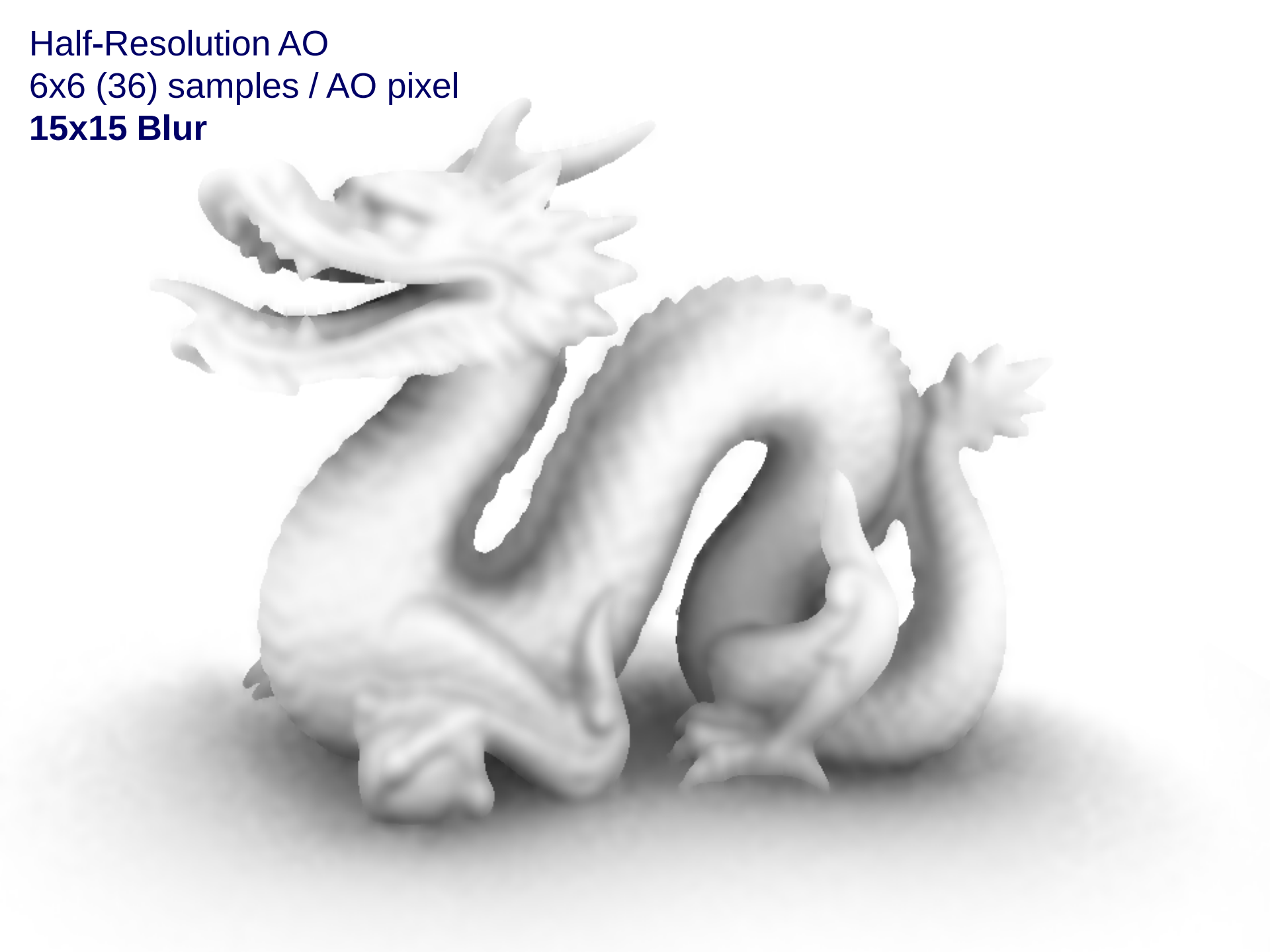
Rendering Pipeline



Half-Resolution AO
6x6 (36) samples / AO pixel
No Blur



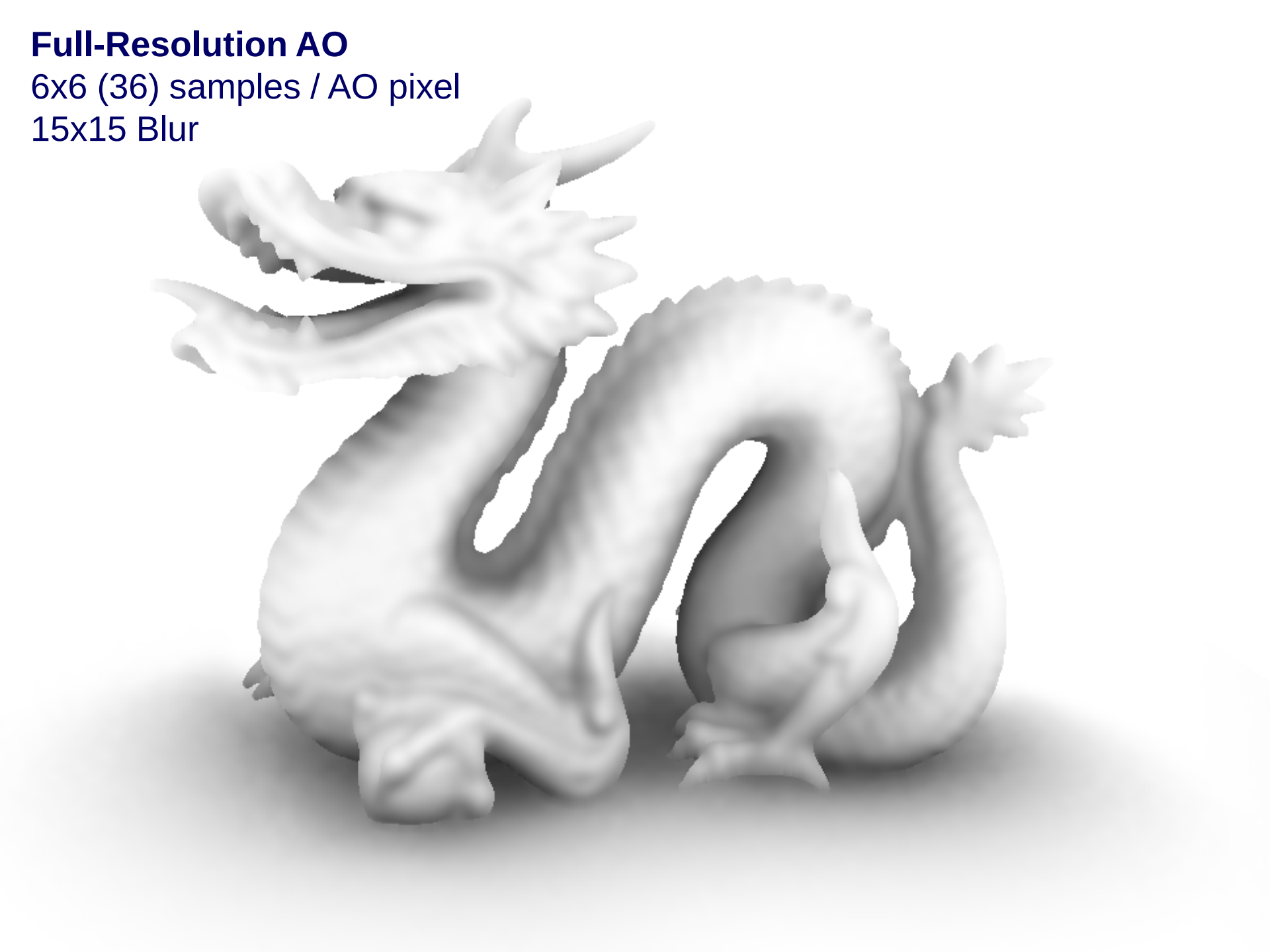
Half-Resolution AO
6x6 (36) samples / AO pixel
15x15 Blur



Full-Resolution AO

6x6 (36) samples / AO pixel

15x15 Blur



Full-Resolution AO

16x16 (256) samples / pixel

No Blur

Full-Resolution AO

16x32 (512) samples / pixel

No Blur

Demo

HBAO - Conclusion

- DirectX10 SDK sample
  - Now available on developer.nvidia.com
  - Including video and whitepaper
- DirectX9 and OpenGL samples to be released soon
- Easy to integrate into a game engine
  - Rendered as a post-processing pass
  - Only requires eye-space depths (normals can be derived from depth)
- More details in ShaderX^7 (to appear)

DirectX 11

DirectX Compute Shader

- New shader type supported in D3D11
  - Designed for general purpose processing
- Doesn't require a separate API - integrated with D3D
  - Shares memory resources with graphics shaders
- Thread invocation is decoupled from input or output domains
  - Single thread can process one or many data elements
- Can share data between threads
- Supports random access memory writes

Compute Shaders on D3D10 Hardware

- Subset of the D3D11 compute shader functionality that runs on current D3D10.x hardware
  - From NVIDIA and AMD
- Drivers available now from NVIDIA and AMD
- **You can start experimenting with compute shaders today**!

Compute Shader 4.0

- New shader models - CS4.0/CS4.1
  - Based on vertex shader VS4.0/VS4.1 instruction set
  - CS4.1 includes D3D10.1 features
    - Texture cube arrays etc.
- Check for support using caps bit:
  - **`ComputeShaders_Plus_RawAndStructuredBuffers_Via_Shader_4_x`**
- Adds:
  - New Compute Shader inputs:
    `vThreadID`*, *`vThreadIDInGroup`*, *`vThreadGroupID`, and ***`vThreadIDInGroupFlattened`***
  - Support for raw and structured buffers

What's Missing in CS4.0 Compared to CS5.0?

- Atomic operations
- Append/consume
- Typed UAV
  (unordered access view)
- Double precision
- DispatchIndirect()

- Still a lot you can do!

Other Differences

- Only a single output UAV allowed
  - Not a huge restriction in practice
- Thread group grid dimensions limited to 65535
  - Z dimension must be 1 (no 3D grids)
- Thread group size is restricted to maximum of 768 threads total
  - 1024 on D3D11 hardware
- Thread group shared memory restricted to 16KB total
  - 32Kb on D3D11 hardware

Thread Group Shared Memory Restrictions

- Each thread can only **write** to its own region of shared memory
- Write-only region has maximum size of 256 bytes, and depends on the number of threads in group
- Writes to shared memory must use a literal offset into the region
- Threads can still **read** from any location in the shared memory

So What Does CS4.x Give Me?

- Scattered writes
  - Via “unordered access views”
  - Write to any address in a buffer
  - Was possible before by rendering points, but not efficient
  - Enables many new algorithms – sorting, parallel data structures
- Thread Group Shared Memory
  - Allows sharing data between threads
  - Much faster than texture or buffer reads, saves bandwidth
  - Fast reductions, prefix sum (scan)
- Efficient interoperability with D3D graphics

Optimizing Compute Shaders

- Context switching
  - try to avoid switching between compute and graphics shaders too often
  - ideally only once per frame
- Use shared memory to save bandwidth where possible
  - think of it as a small user-managed cache

Optimizing CS Memory Access on NVIDIA D3D10 Hardware

- Some restrictions for optimal performance on NVIDIA GeForce 8/9 series:
  - (Less of an issue on GeForce GTS series)
- Reads and writes to structured buffers should be linear and aligned
  - thread *i* should read/write to location *i*
- Allows hardware to “coalesce” memory accesses into a minimum number of transactions
- Use textures if you want random read access

Applications

- Image processing
  - Reductions, Tone mapping
  - Blurs, Image Compression
- Physics
  - Particle systems, Fluids
  - Collision detection
- AI
  - Path finding
- Animation
  - Advanced skinning and deformations

Examples

- N-Body Simulation
- Colliding Particles
- Image Processing – Box Blur
- Ocean

N-Body Simulation

- Simulates motion of bodies under gravity
- Uses brute force n^2 comparisons
- Uses shared memory to re-use body positions among threads
  - Reduces bandwidth massively
- 30,720 bodies

N-Body CS Code

```
// all positions, then all velocities
RWStructuredBuffer<float4> particles;

float3
bodyBodyInteraction(float4 bi, float4 bj)
{
    // r_ij  [3 FLOPS]
    float3 r = bi - bj;

    // distSqr = dot(r_ij, r_ij) + EPS^2  [6 FLOPS]
    float distSqr = dot(r, r);
    distSqr += g_softeningSquared;

    float invDist = 1.0f / sqrt(distSqr);
    float invDistCube =  invDist * invDist * invDist;

    float s = bj.w * invDistCube;

    // a_i =  a_i + s * r_ij [6 FLOPS]
    return r*s;
}
```

N-Body CS Code (2)

```
// body positions in shared memory
groupshared float4 sharedPos[BLOCK_SIZE];

float3 gravitation(float4 myPos, float3 accel)
{
    // unroll the loop
    [unroll]
    for (unsigned int counter = 0; counter < BLOCK_SIZE; counter++)
    {
          accel += bodyBodyInteraction(sharedPos[counter], myPos);
    }

    return accel;
}
```

N-Body CS Code (3)

```
float3 computeBodyAccel(float4 bodyPos, uint threadId, uint blockId)
{
    float3 acceleration = {0.0f, 0.0f, 0.0f};

    uint p = BLOCK_SIZE;
    uint n = g_numParticles;
    uint numTiles = n / p;

    for (uint tile = 0; tile < numTiles; tile++)
    {
       // load to shared memory
        sharedPos[threadId] = particles[g_readOffset + tile * p +
    threadId];
        GroupMemoryBarrierWithGroupSync();
        acceleration = gravitation(bodyPos, acceleration);
        GroupMemoryBarrierWithGroupSync();
    }

    return acceleration;
}
```

N-Body CS Code (4)

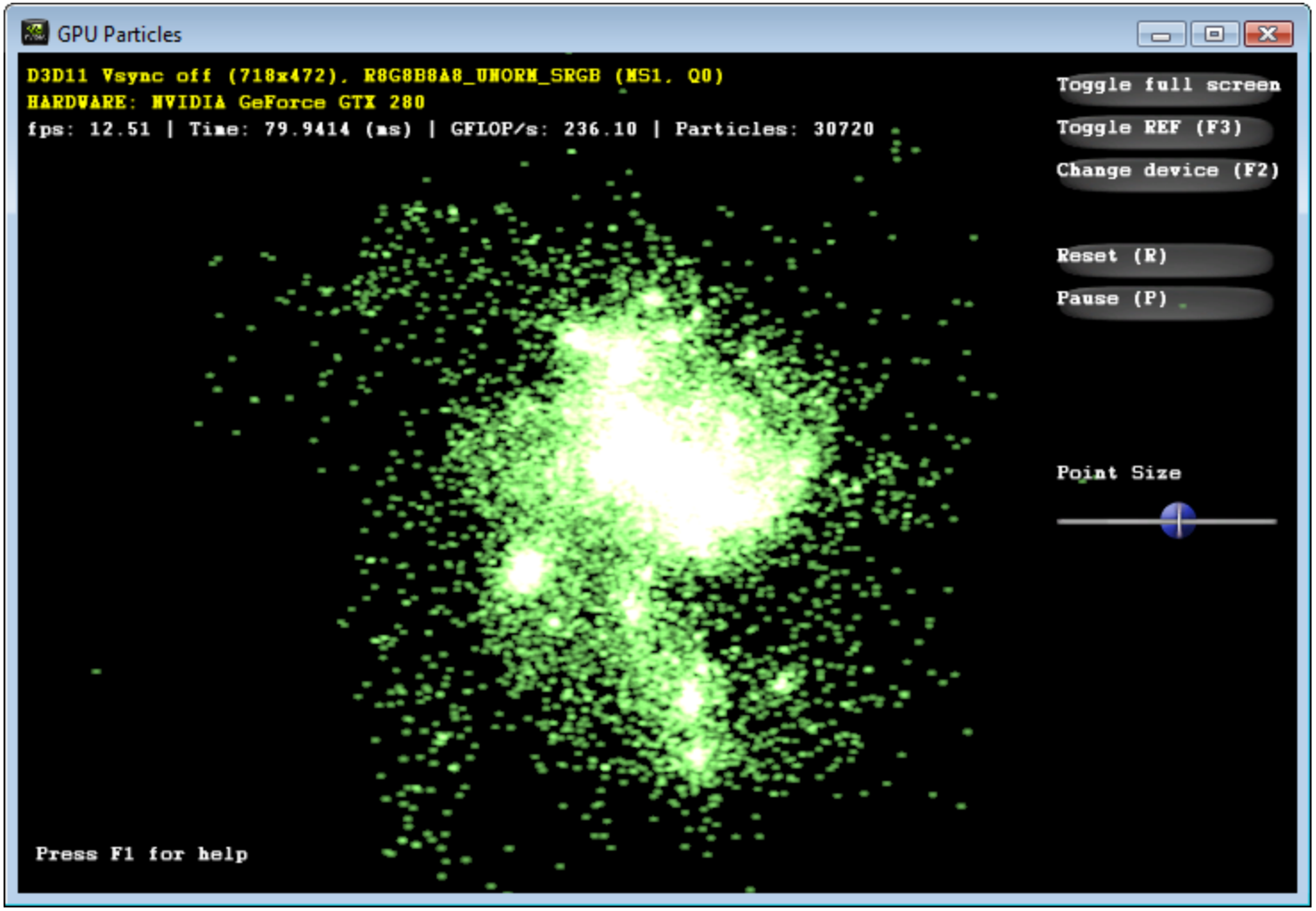
```
[numthreads(BLOCK_SIZE,1,1)]
void UpdateParticles(uint threadId : SV_GroupIndex,
                     uint3 groupId        : SV_GroupID,
                     uint3 globalThreadId : SV_DispatchThreadID)
{
    float4 pos = particles[g_readOffset + globalThreadId.x];
    float4 vel = particles[2 * g_numParticles + globalThreadId.x];

    float3 accel = computeBodyAccel(pos, threadId, groupId);

    vel.xyz += accel * g_timestep;
    pos.xyz += vel   * g_timestep;

    particles[g_writeOffset + globalThreadId.x]      = pos;
    particles[2 * g_numParticles + globalThreadId.x] = vel;
}
```

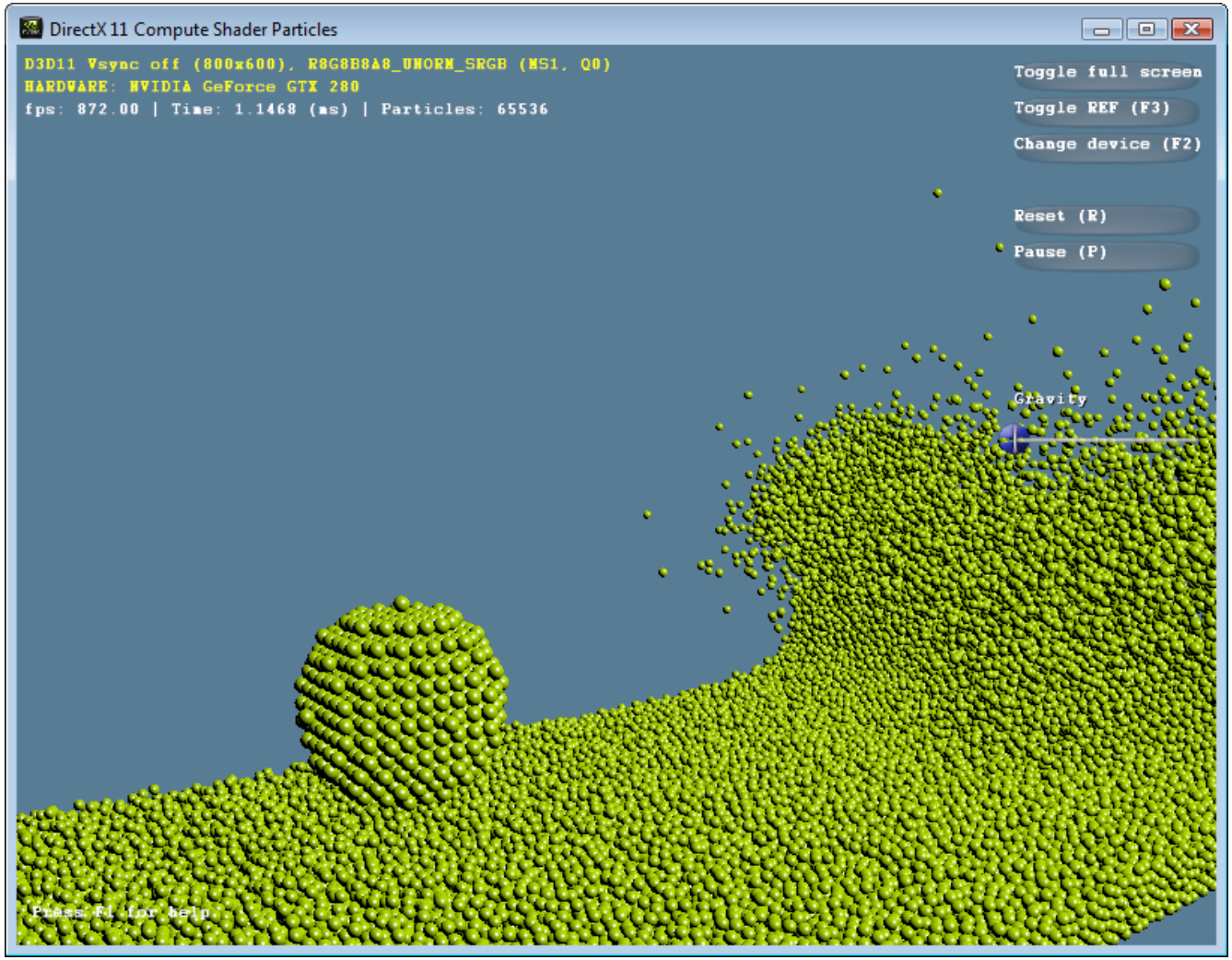
N-Body Simulation



Colliding Particles

- Simulates large number of particles with collisions
- Uses uniform grid to find neighboring particles quickly
- Grid implemented using parallel bitonic sort (uses scattered writes)
  - Calculate which grid cell each particle it is in
  - Sort particles by cell index
  - Find start and end of each cell in sorted list

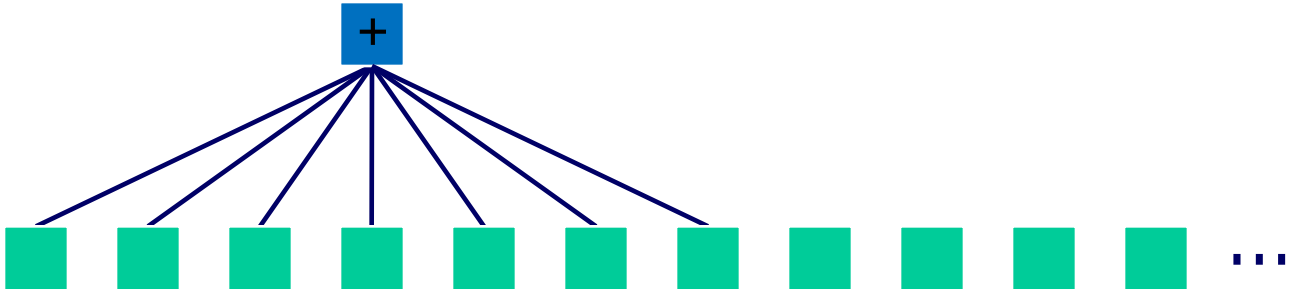
Particles Demo



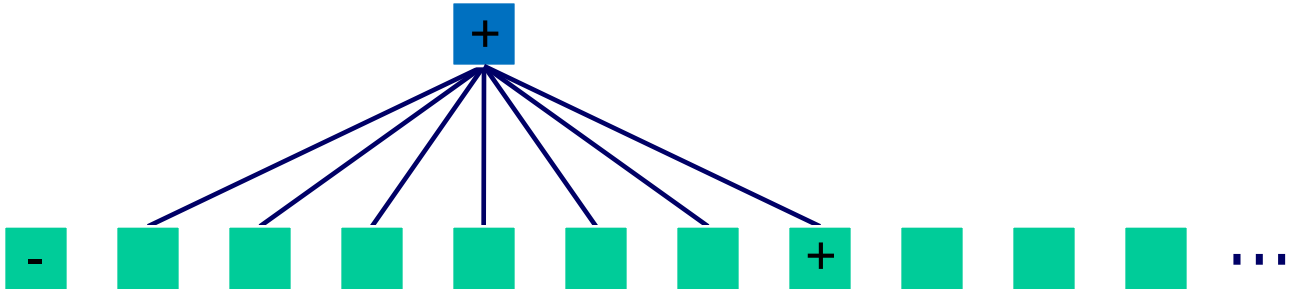
Rolling Box Filter Blur

- Computes box filter *of any radius* for *constant cost*
  - Only 2 adds, one multiply per pixel
- Takes advantage of scattered writes available in CS
- Uses one thread per row/column in the image
  - Parallelism is limited by image size
- At each step, adds incoming new pixel, subtracts pixel leaving window
- Can be iterated to approximate Gaussian blur

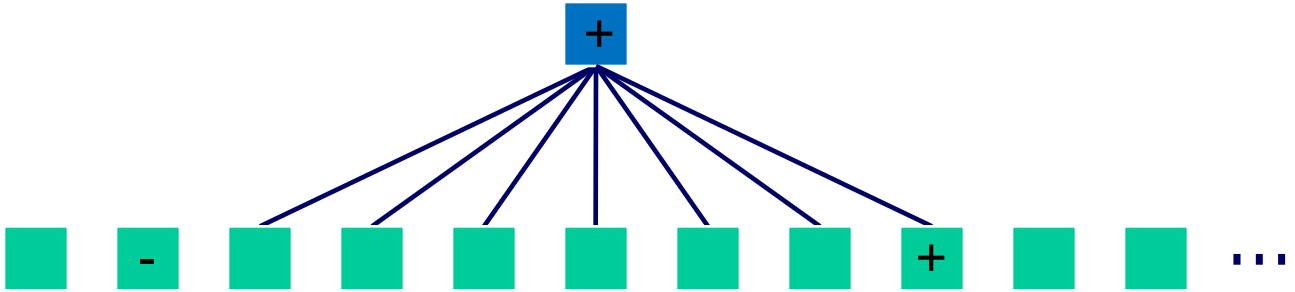
Rolling Box Blur



Rolling Box Blur



Rolling Box Blur



Box Blur CS Code

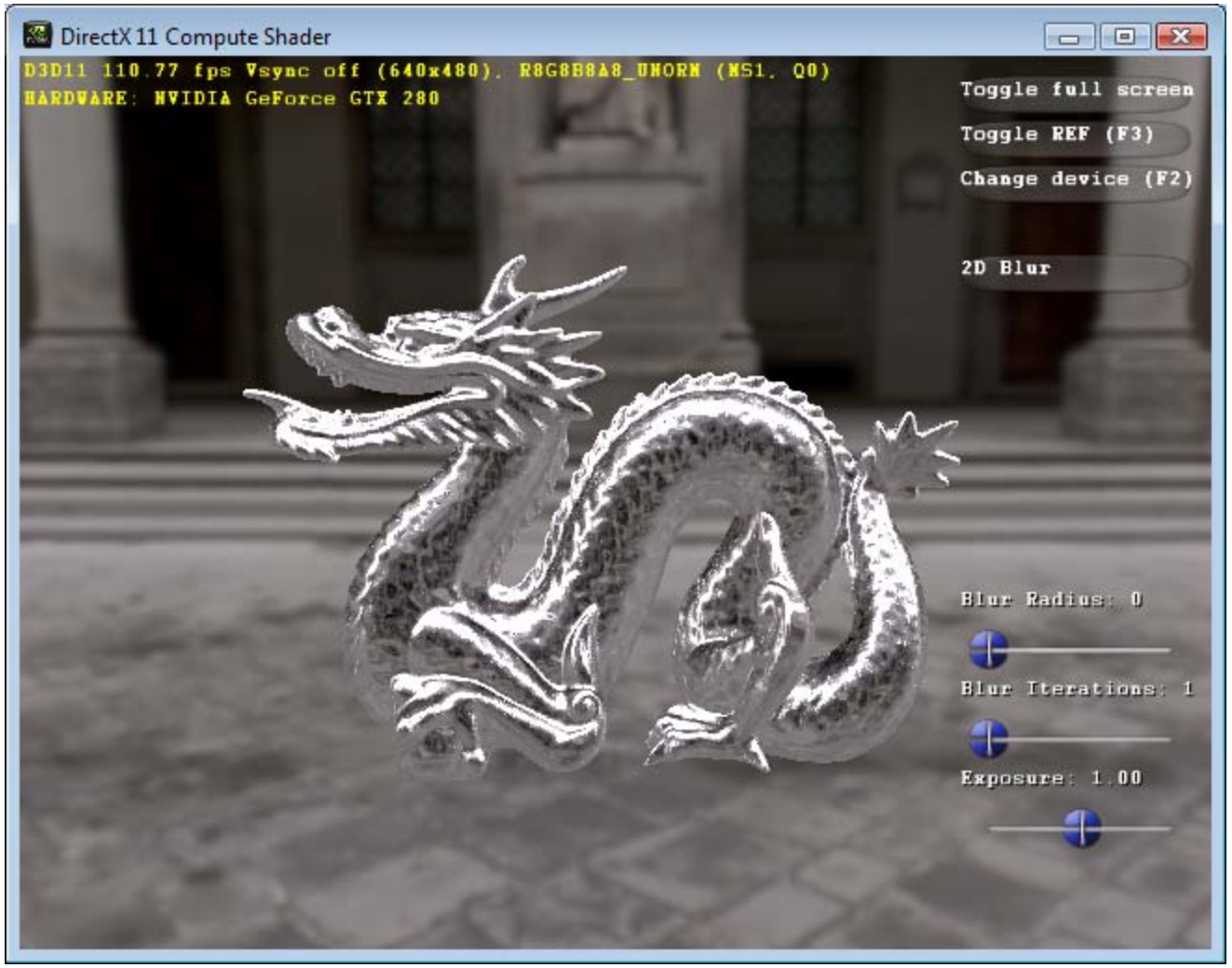
```
Texture2D<float4> Input
RWStructuredBuffer<float4> Output;

[numthreads(256,1,1)]
void boxBlurY(uint3 globalThreadID : SV_DispatchThreadID)
{
    uint x = globalThreadID.x;
    if (x >= imageW) return;
    float scale = 1.0f / (2*blurRadius+1);

    float4 t = 0.0f;
    for(int y=-blurRadius; y<=blurRadius; y++) {
        t += Input.Load(int3(x, y, 0));
    }
    Output[x] = t * scale;

    for(y=1; y<imageH; y++) {
        t += Input.Load(int3(x, y + blurRadius, 0));
        t -= Input.Load(int3(x, y - blurRadius - 1, 0));
        Output[y*imageW+x] = t * scale;
    }
}
```

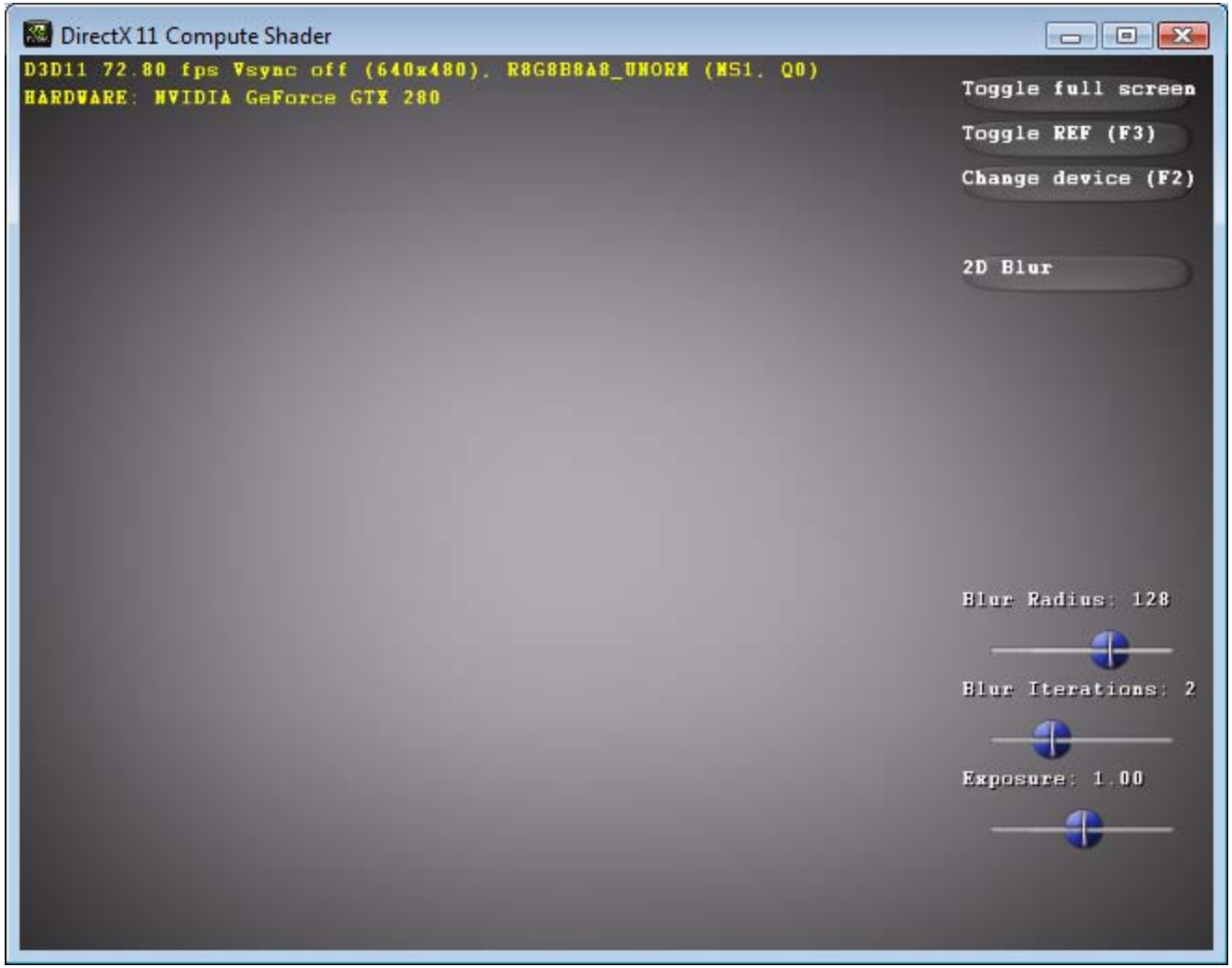
Rolling Box Blur



Rolling Box Blur

Rolling Box Blur

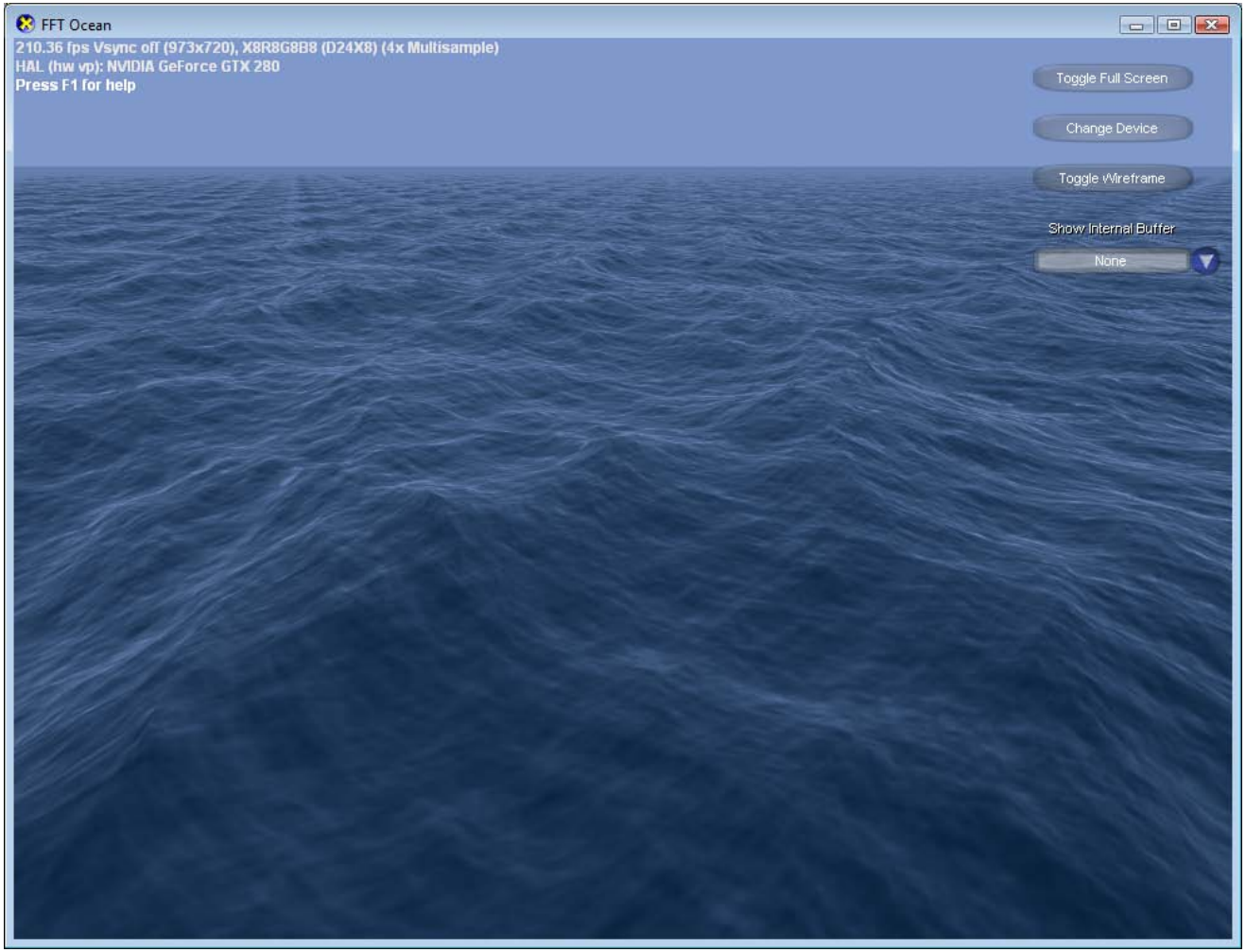
Rolling Box Blur



FFT Ocean

- Generates ocean height field using Compute Shader to compute FFT
- See: Tessendorf, “Simulating Ocean Water”
- Synthesize waves in frequency space based on statistical model
- Use C2R FFT to convert to spatial domain
- Heightfield tiles naturally

FFT Ocean



Summary

- DirectX Compute Shader offers many opportunities for offloading CPU processing to the GPU
- ...and producing unique new effects for your game
- Can achieve much higher performance than vertex or pixel shader based solutions
- Can start developing today on current hardware

Questions?

Acknowledgments

NVIDIA

- Miguel Sainz, Louis Bavoil, Rouslan Dimitrov, Samuel Gateau, Jon Jansen, Mark Harris, Calvin Lin, Victor Podlozhnyuk
- NVIDIA D3D Driver Team

Models

- Dragon - Stanford 3D Scanning Repository
- Science-Fiction scene - Juan Carlos Silva http://www.3drender.com/challenges/index.htm
- Sibenik Cathedral - Marko Dabrovic

References

1. Volume Rendering Techniques, Milan Ikits, Joe Kniss, Aaron Lefohn, Charles Hansen. Chapter 39, section 39.5.1, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (2004).
2. BAVOIL, L., AND SAINZ, M. 2009. Image-space horizon-based ambient occlusion. In ShaderX7 - Advanced Rendering Techniques.