SIGGRAPH 2003

SAN DIEGO

ATI R3*x*0 Pixel Shaders

Jason L. Mitchell
3D Application Research Group Lead
ATI Research

Outline

- Architectural Overview
  - Vertex & Pixel
- Focus on pixel shader
  - Precision
  - Co-issue
  - Case Study: Real-time Überlight
    - Choosing correct frequency of evaluation
    - Vectorizing

R3*x*0 Shaders

- Vertex Shader
  - Longer programs than previous generation
  - Static flow control
- Pixel Shader
  - Floating point
  - Longer programs than previous generation

R3*x*0 Pixel Shaders

- ARB_fragment_program & OpenGL Shading Language
- DirectX® 9 ps_2_0 pixel shader model
- 64 alu ops
  - On R3*x*0 hardware, these can be a vec4 operation or a vec3 coissued with a scalar op
  - ps_2_0 model does not expose co-issue
  - For this and other reasons, hardware cycle counts are less than or equal to ps_2_0 cycle counts
- 32 texture ops
  - 4 levels of dependency
- One and only one precision in shader
  - 24-bit floating point (s16e7)
- Secret sauce
  - Many cycle counts are less than you would think

Why retain co-issue?

- Engineering answer
  - Scalar and vec3 operations are common
  - Allows us to do some vectorization of scalar code
- Marketing answer
  - In the marketplace, a new chip must not only be the best at new features but speed up old ones
  - Co-issue is out there
    - Used often by shipping games and must not run slower on new hardware than on old
    - Microsoft High Level Shading Language (HLSL) compiler does a good job of generating co-issue when compiling for legacy shader models, hence co-issue will continue to be used for those models

Precision

- Single 24-bit floating point data format for the pixel pipeline
- Classic speed and die-area tradeoff
- Interpolated texture coordinates are higher precision but everything else operates at this one specific precision
- Programmers don't have to worry about datatypes with varying precision and performance characteristics
  - Just high performance all the time
- Having a single hardware model used to support all pixel shading models significantly simplifies the driver:
  - Legacy multitexture
  - DirectX 8.*x* pixel shading
  - DirectX 9 pixel shading

Überlight

- Will now illustrate the value of these architectural properties with an example: Überlight
  - Intuitive enough to cover here
  - Complex enough to be interesting
  - Scalar-heavy but vectorizable
  - Requires reasonable precision

What is Überlight?

- Intuitive light described by Ronen Barzel in "Lighting Controls for Computer Cinematography" in the *Journal of Graphics Tools*, vol. 2, no. 1: 1-20
- See also Chapter 14 in *Advanced RenderMan*® by Apodaca and Gritz
- Überlight is procedural and has many intuitive controls:
  - light type, intensity, light color, cuton, cutoff, near edge, far edge, falloff, falloff distance, max intensity, parallel rays, shearx, sheary, width, height, width edge, height edge, roundness and beam distribution
- There's a good RenderMan® sl version in the public domain written by Larry Gritz

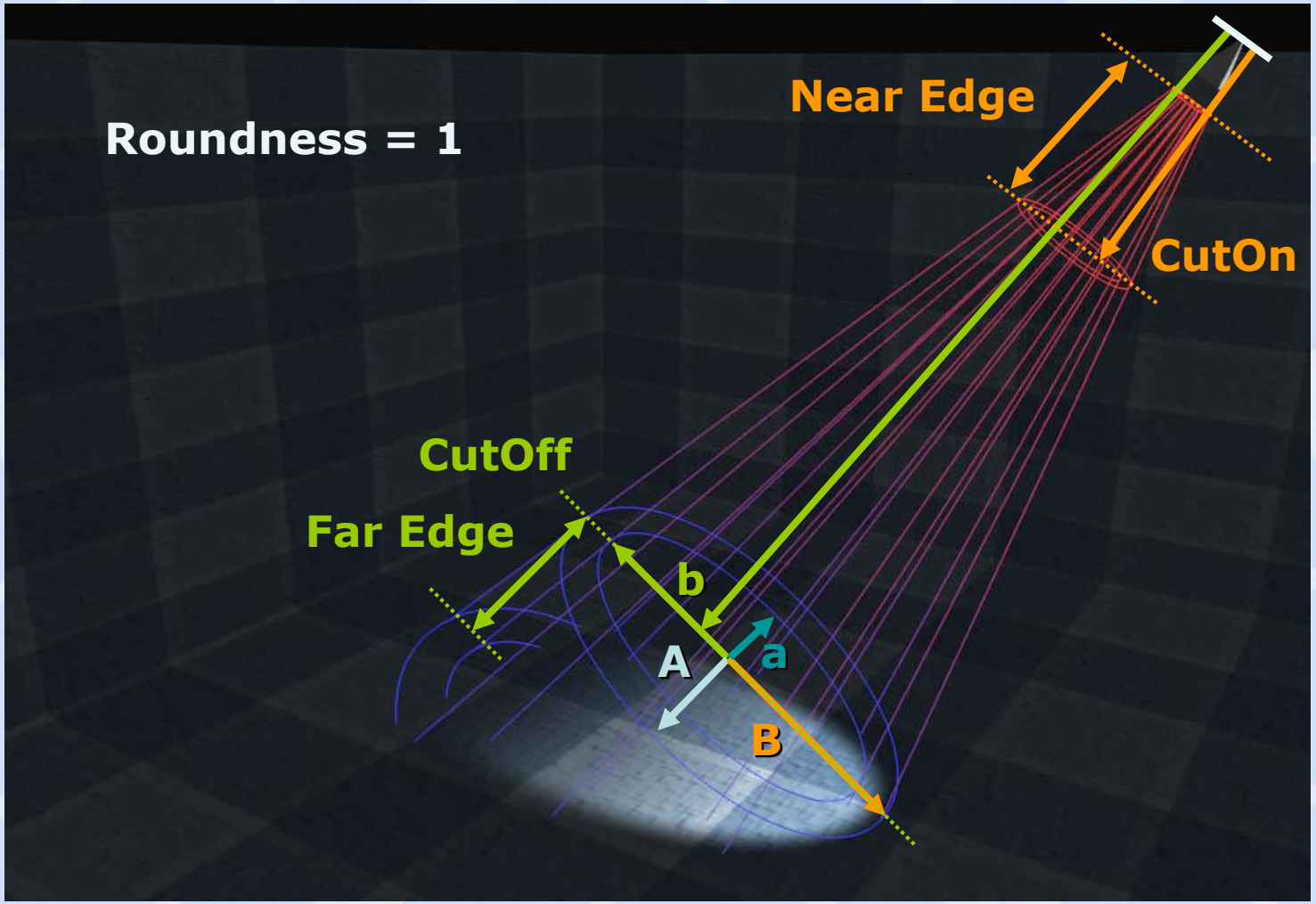
Überlight Overview

- For each light
  - Transform P to light space
  - Smooth clip to procedural volume
    - Near, far and nested superellipses
  - Distance falloff
  - Beam distribution
  - Ray direction
  - Blockers
  - Projective textures
    - Shadow, noise & cookies
- Today, I'll talk about one light and ignore blockers

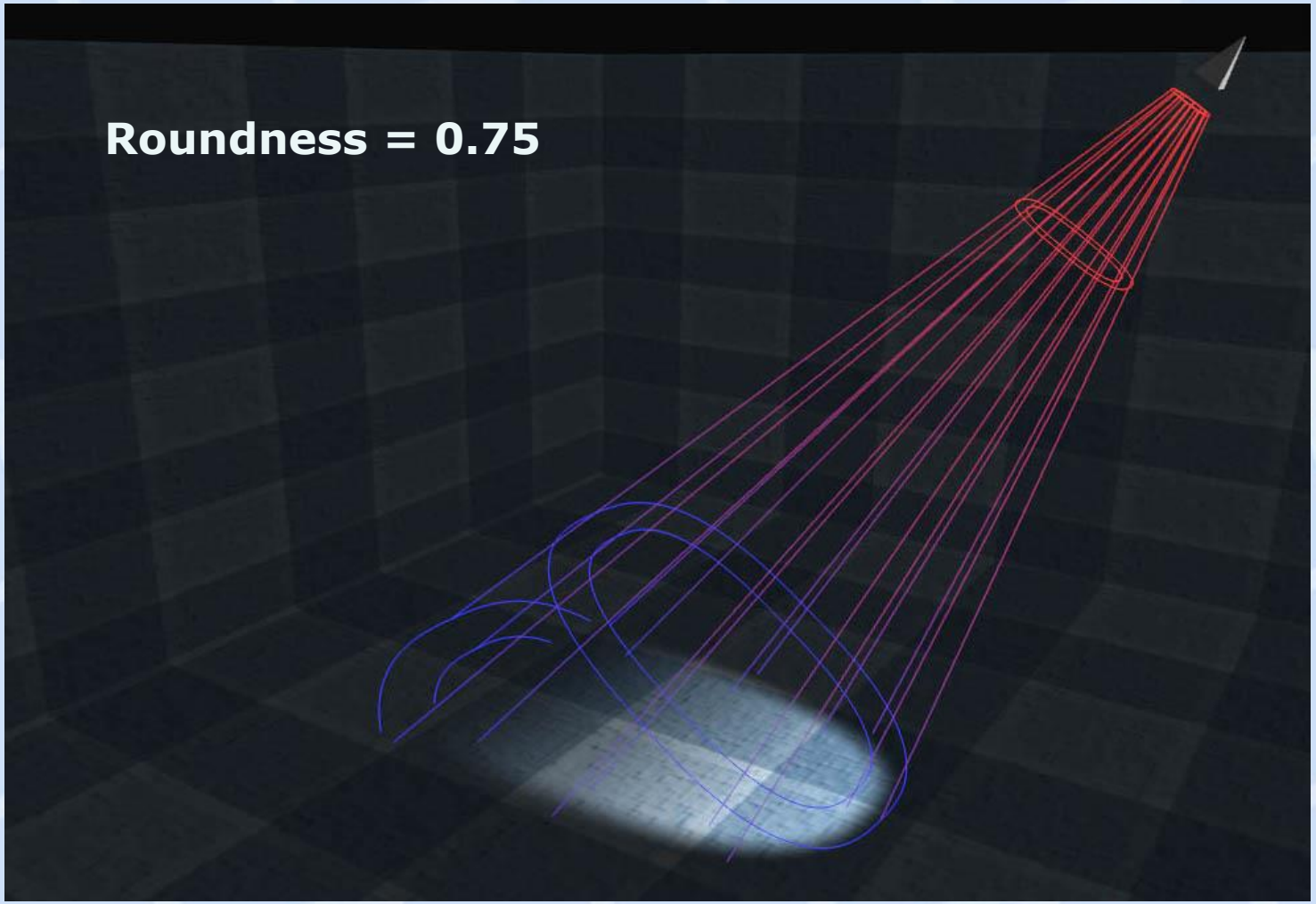
Überlight Volume

- Volume defined in space of light source
- Omnilight or spotlight modes
  - Will discuss spotlight today
- Nested extruded superellipses
  - White inside inner superellipse
  - Black outside outer superellipse
  - Smooth transition in between
- Near and Far planes
  - Smooth cuton and cutoff

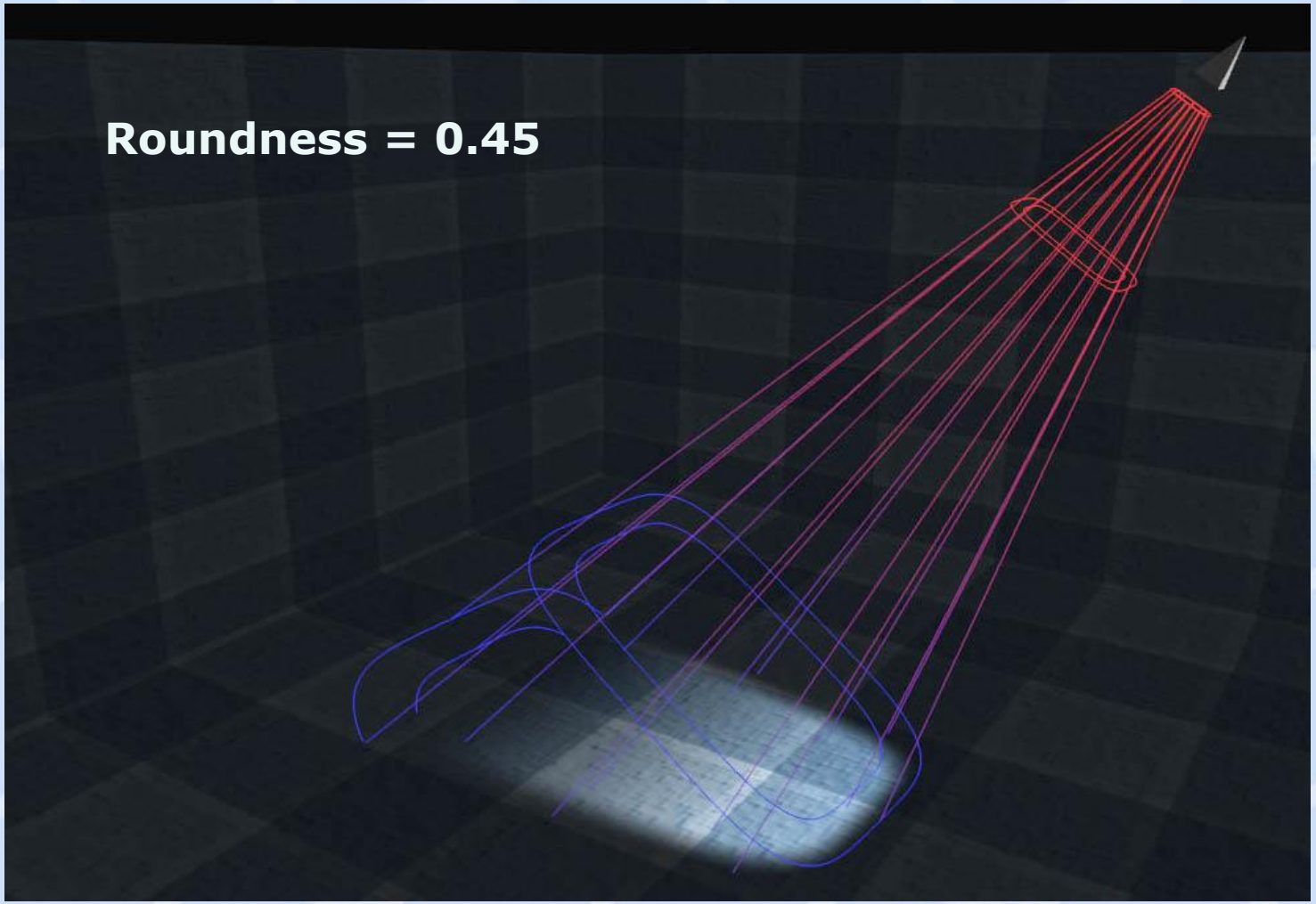
Procedural Light Volume



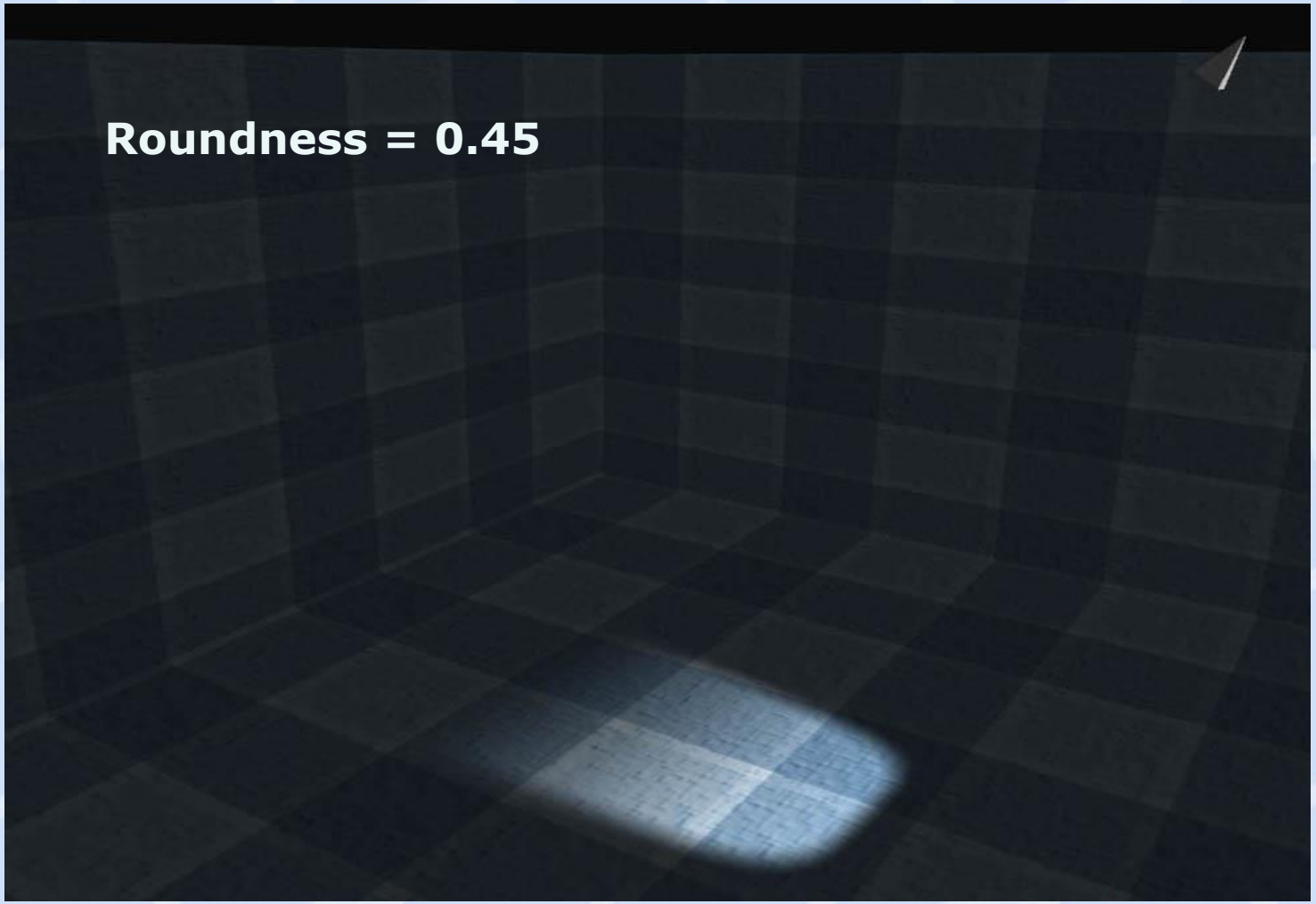
Procedural Light Volume



Procedural Light Volume



Procedural Light Volume



DirectX 9 HLSL Überlight Implementation

- Manually ported from sl to DirectX 9 HLSL using ps_2_0 compile target on R3*x*0
- Perform computation at right frequency
  - Perform some computation in vertex shader
    - Transformation to light space
    - Projective texture coordinate generation for cookies etc
  - Do some precomputation outside of the shader
- Vectorize
  - There are clear opportunities for vectorization

RenderMan® clipSuperellipse()


```
float clipSuperellipse (point Q;     /* Test point on the x-y plane */
                        float a, b;  /* Inner superellipse */
                        float A, B;  /* Outer superellipse */
                        float roundness;  /* Same roundness for both ellipses */
                       )
{
    float result;
    float x = abs(xcomp(Q)), y = abs(ycomp(Q));

    if (roundness < 1.0e-6)
    {
        /* Simpler case of a square */
        result = 1 - (1-smoothstep(a,A,x)) * (1-smoothstep(b,B,y));
    }
    else
    {
        /* Harder, rounded corner case */
        float re = 2/roundness;   /* roundness exponent */
        float q = a * b * pow (pow(b*x, re) + pow(a*y, re), -1/re);
        float r = A * B * pow (pow(B*x, re) + pow(A*y, re), -1/re);
        result = smoothstep (q, r, 1);
    }

    return result;
}
```

Ignore this case today

Straight Port to HLSL

- Non-rectangle case; minor syntactic changes
- Compiles to 42 cycles in ps_2_0, 40 cycles on R3*x*0

```
float clipSuperellipse (
          float3 Q,          // Test point on the x-y plane
          float a,           // Inner superellipse
          float b,
          float A,           // Outer superellipse
          float B,
          float roundness)   // Roundness for both ellipses
{
  float x = abs(Q.x), y = abs(Q.y);

  float re = 2/roundness;


  float q = a * b * pow(pow(b*x, re) + pow(a*y, re), -1/re);
  float r = A * B * pow(pow(B*x, re) + pow(A*y, re), -1/re);

  return smoothstep (q, r, 1);
}
```

Heavy use of scalar uniform parameters results in greedy use of constant

Vectorizable

Can be precomputed

threads

Vectorized Version

- Pack relevant scalars together
  - Reduces constant register usage
  - Allows us to vectorize **`abs()`** and the multiplications
- Precompute functions of *roundness* in app
- Compiles to 33 cycles in ps_2_0 (28 cycles on R3*x*0)

```
float clipSuperellipse (
                float2 Q,     // Test point on the x-y plane
                float4 aABb   // Dimensions of superellipses
                float2 r)     // Two functions of roundness
{
    float2 qr, Qabs = abs(Q);

    float2 bx_Bx = Qabs.x * aABb.wzyx;  // Unpack bB
    float2 ay_Ay = Qabs.y * aABb;

    qr.x = pow(pow(bx_Bx.x, r.x) + pow(ay_Ay.x, r.x), r.y);
    qr.y = pow(pow(bx_Bx.y, r.x) + pow(ay_Ay.y, r.x), r.y);

    qr *= aABb * aABb.wzyx;

    return smoothstep (qr.x, qr.y, 1);
}
```

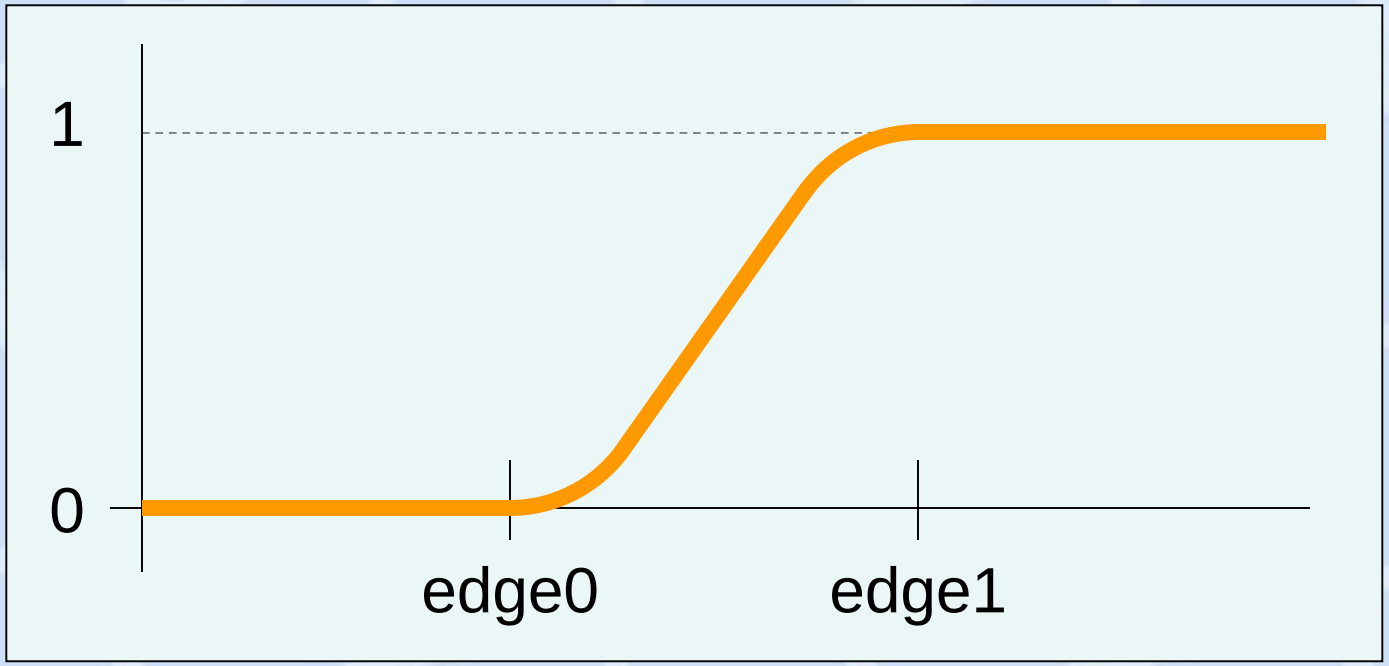
Scalar constants

Precomputed scalars packed into a **`float2`**

Vector operations

smoothstep()

- Standard function in procedural shading
- Intrinsics built into RenderMan® and DirectX HLSL:



C implementation


```
float smoothstep (float edge0, float edge1, float x)
{
   if (x < edge0)
      return 0;

   if (x >= edge1)
      return 1;

   // Scale/bias into [0..1] range
   x = (x - edge0) / (edge1 - edge0);

   return x * x * (3 - 2 * x);
}
```

HLSL implementation

- The free saturate handles *x* outside of [edge0..edge1] range without the conditionals

```
float smoothstep (float edge0, float edge1, float x)
{
   // Scale, bias and saturate x to 0..1 range
   x = saturate((x - edge0) / (edge1 - edge0));

   // Evaluate polynomial
   return x*x*(3-2*x);
}
```

Vectorized HLSL

- Precompute `1/(edge1 - edge0)`
  - Done in the app for edge widths at cuton and cutoff
- Parallel operations performed on **`float3`**s
- Whole spotlight volume computation of überlight compiles to 47 cycles in ps_2_0 (41 cycles on R3*x*0)

```
float3 smoothstep3(float3 edge0, float3 edge1,
                   float3 OneOverWidth, float3 x)
{
    // Scale, bias and saturate x to [0..1] range
    x = saturate((x - edge0) * OneOverWidth);

    // Evaluate polynomial
    return x*x*(3-2*x);
}
```

Precompute

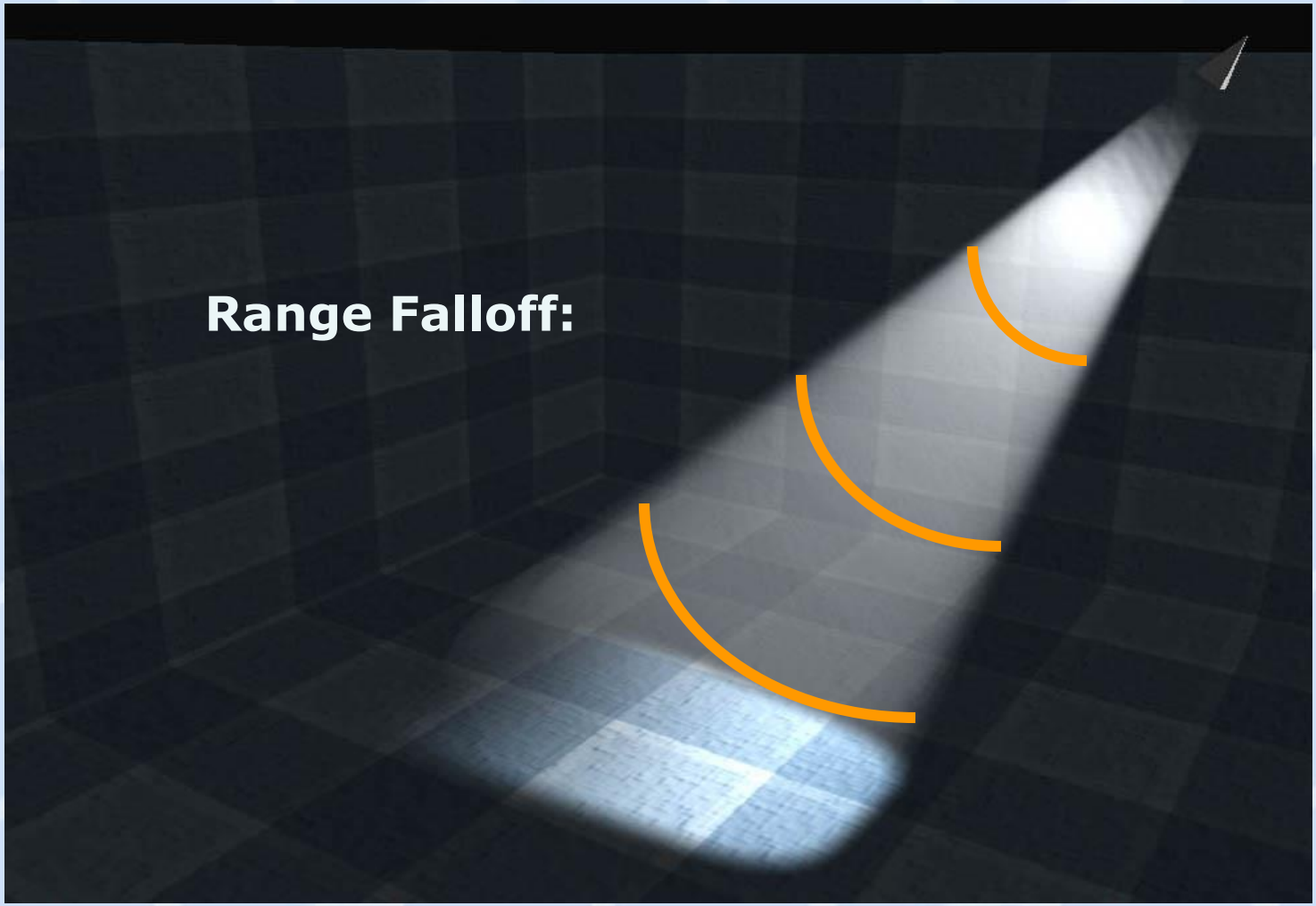
$\frac{1}{\text{OneOverWidth}}$

Vector operations

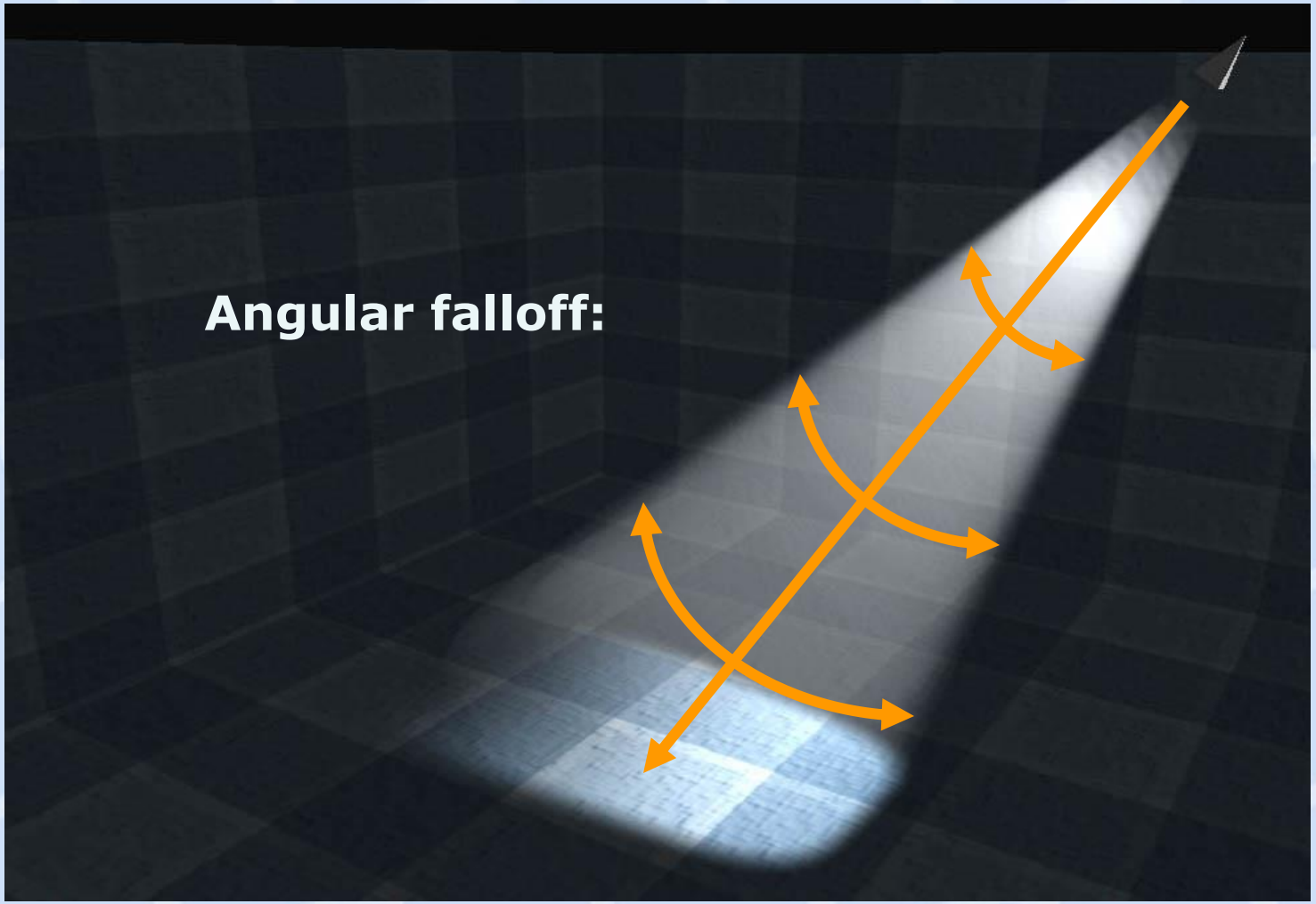
More überlight controls

- Shear
  - Can be useful to match desired light direction with orientation of shaped light source such as a window in a wall
- Distance falloff
- Beam Distribution
  - Angular falloff
- Ray direction
  - Parallel light or radiating from source

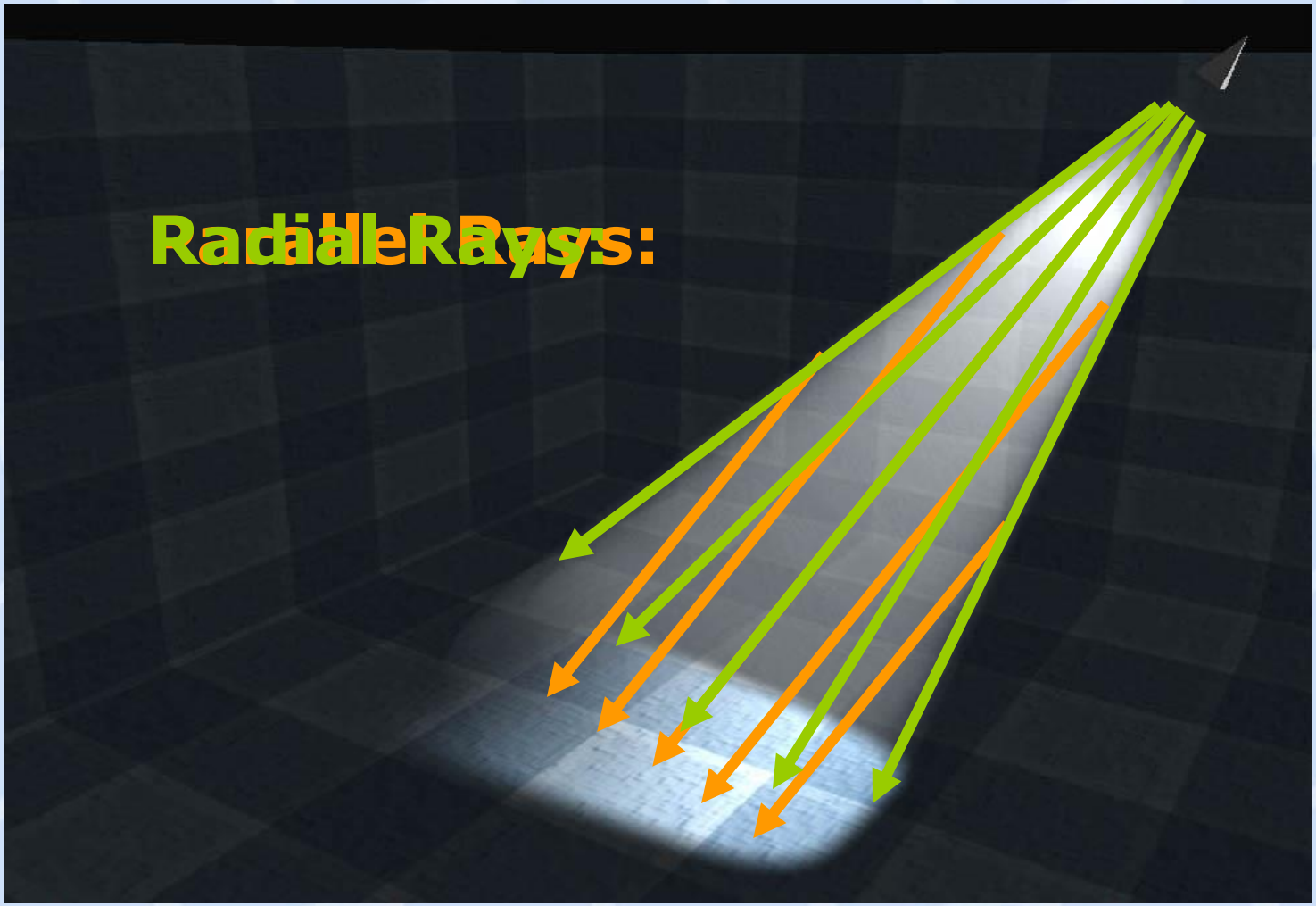
Distance Falloff



Beam Distribution



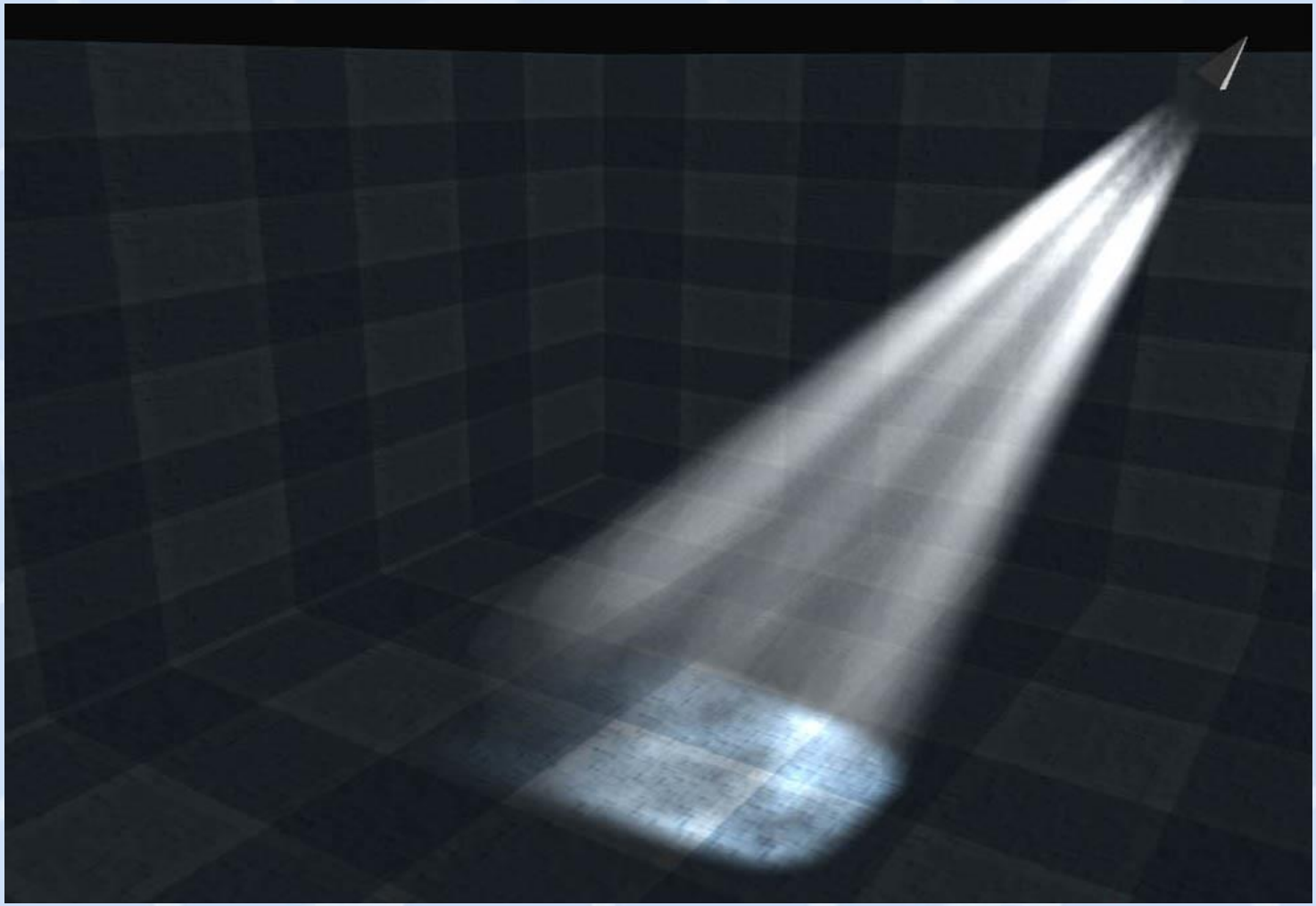
Ray Direction



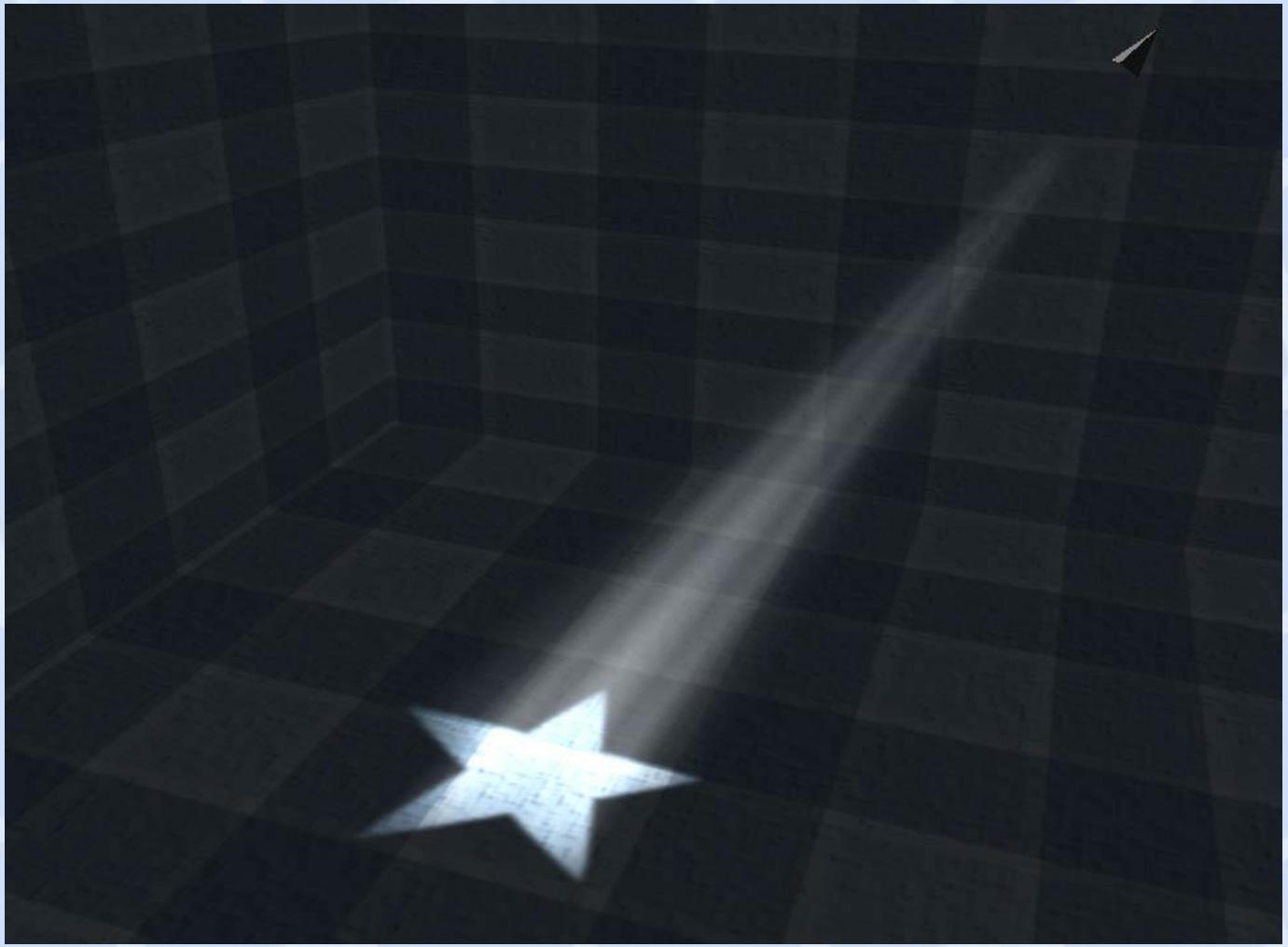
Projective Textures

- Cookie, noise and shadow map
- Generate projective texture coordinates in vertex shader
- Do projective texture loads in pixel shader
- Modulate with überlight intensity

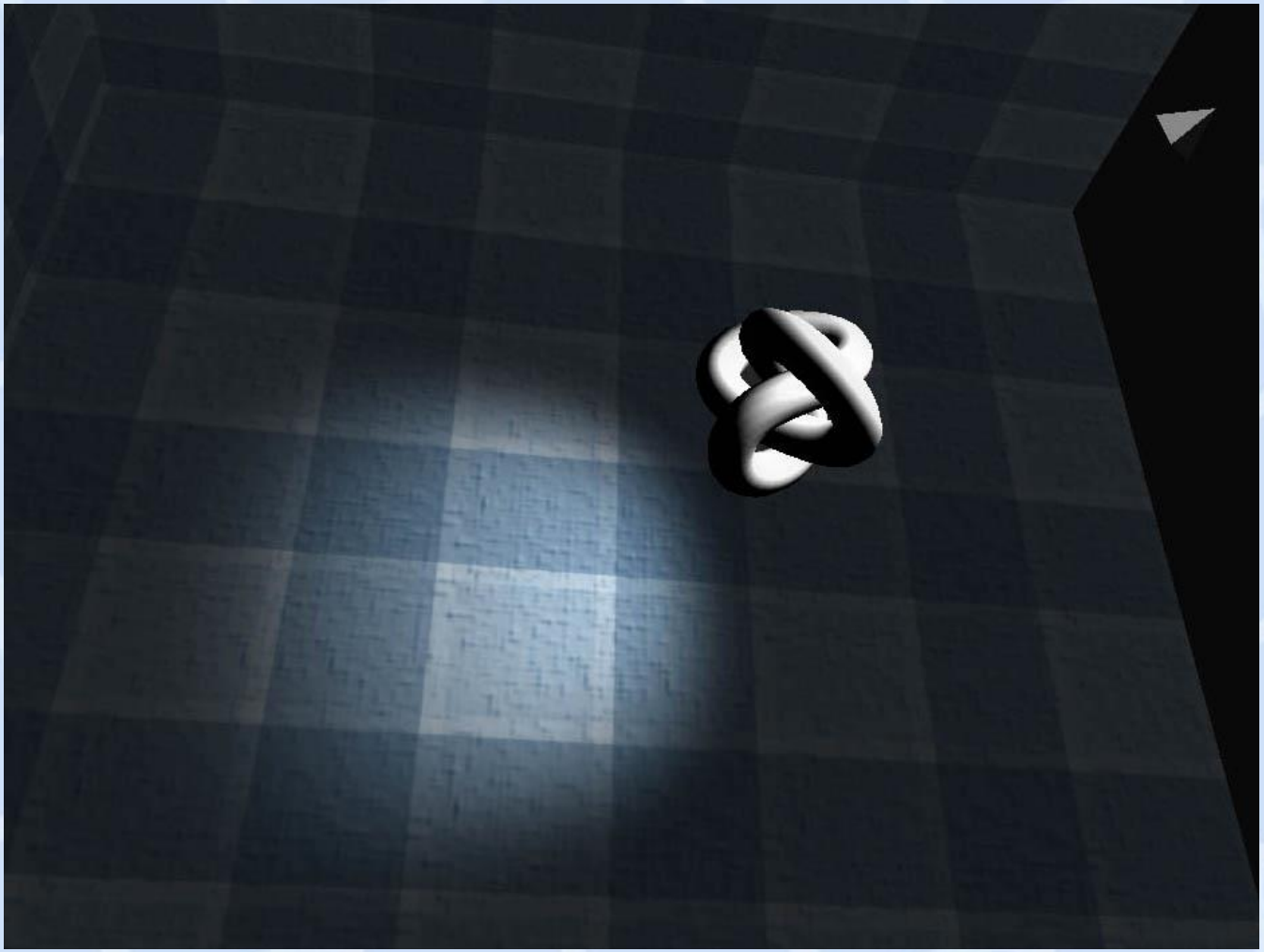
Projected 2D Noise



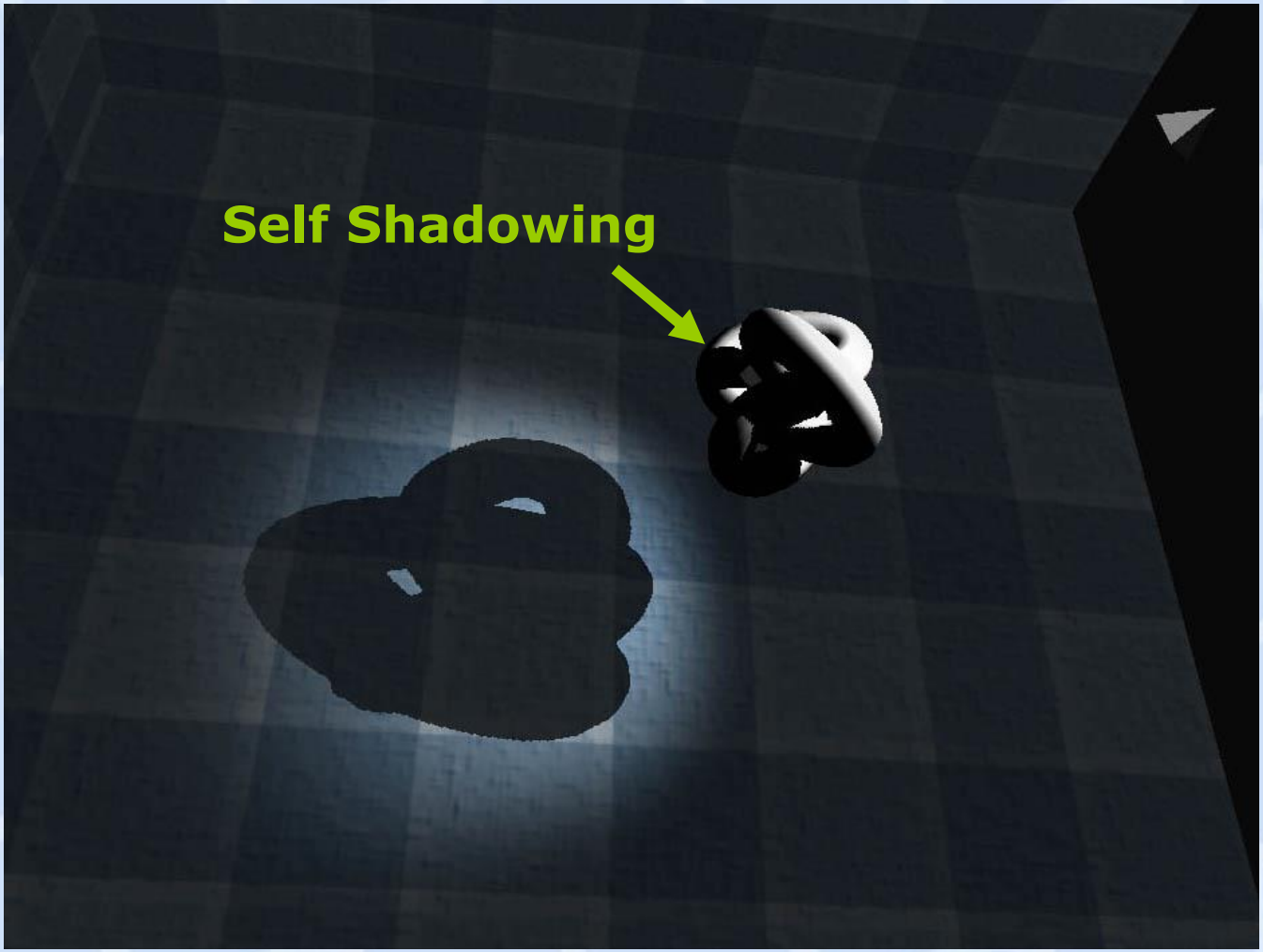
Cookie



Shadows

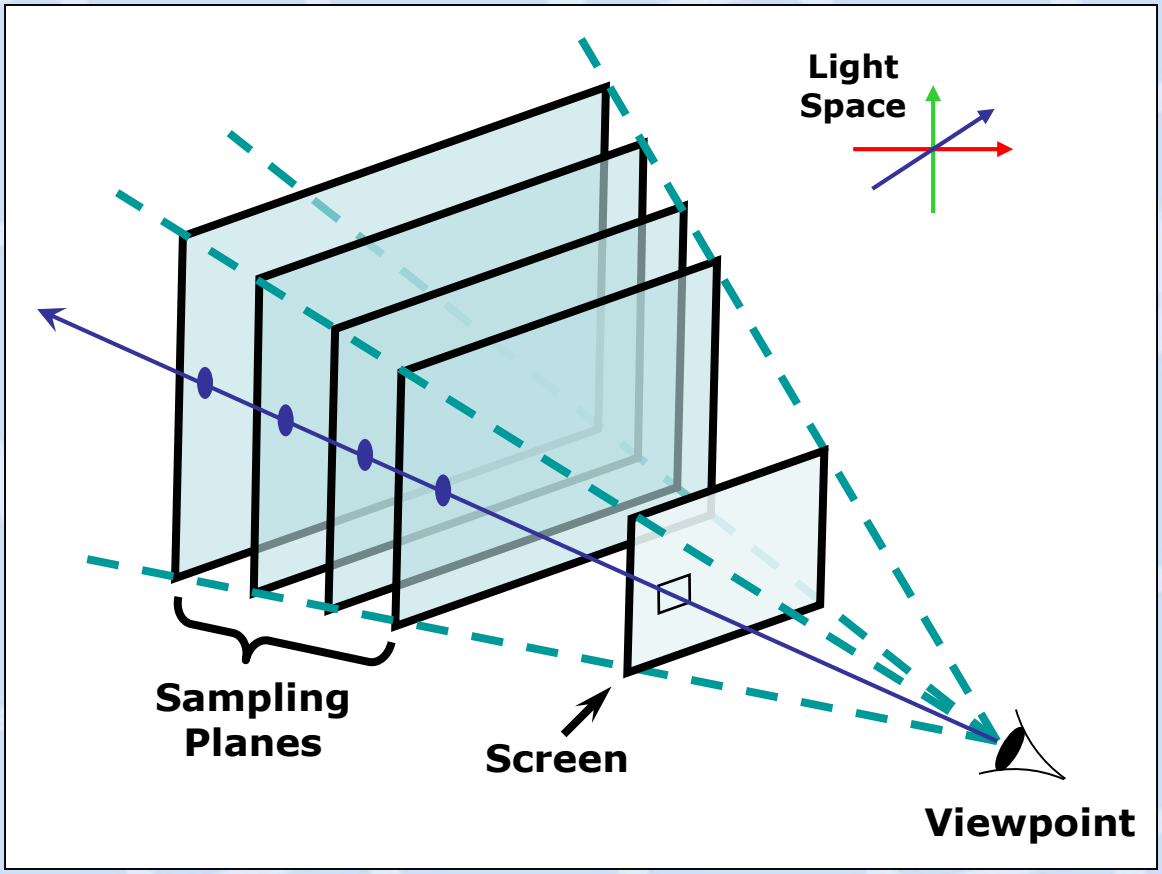


Shadows



Fog Volume Rendering

- Technique developed in several papers by Dobashi and Nishita
- Borrows from medical volume visualization approaches
- Shade sampling planes in light space
- Composite into frame buffer to approximate integral along view rays



Sampling Planes

- Shaded in light space
  - Project cookies as in Dobashi papers
  - Run shader like überlight
- Parallel to view plane
- Vertex shader stretches them to fill view-space bounding box of light frustum
- Clipped to light frustum with user clip planes
  - Absolutely required due to extreme fill demands

Summary

- Focused on R3*x*0 pixel shader
  - Illustrated architectural properties with real-time überlight implementation
  - As a side effect, gave some tips on how to write HLSL that generates efficient code
  - Rendered shafts of light through participating medium in order to illustrate some of the überlight controls
- Will put demo app online at some point

References

- [Barzel97] Ronen Barzel, “Lighting Controls for Computer Cinematography” in the *Journal of Graphics Tools*, vol. 2, no. 1: 1-20

- [Dobashi02] Yoshinori Dobashi, Tsuyoshi Yamamoto and Tomoyuki Nishita, “Interactive Rendering of Atmospheric Scattering Effects Using Graphics Hardware,” *Graphics Hardware* 2002.

SIGGRAPH 2003

SAN DIEGO