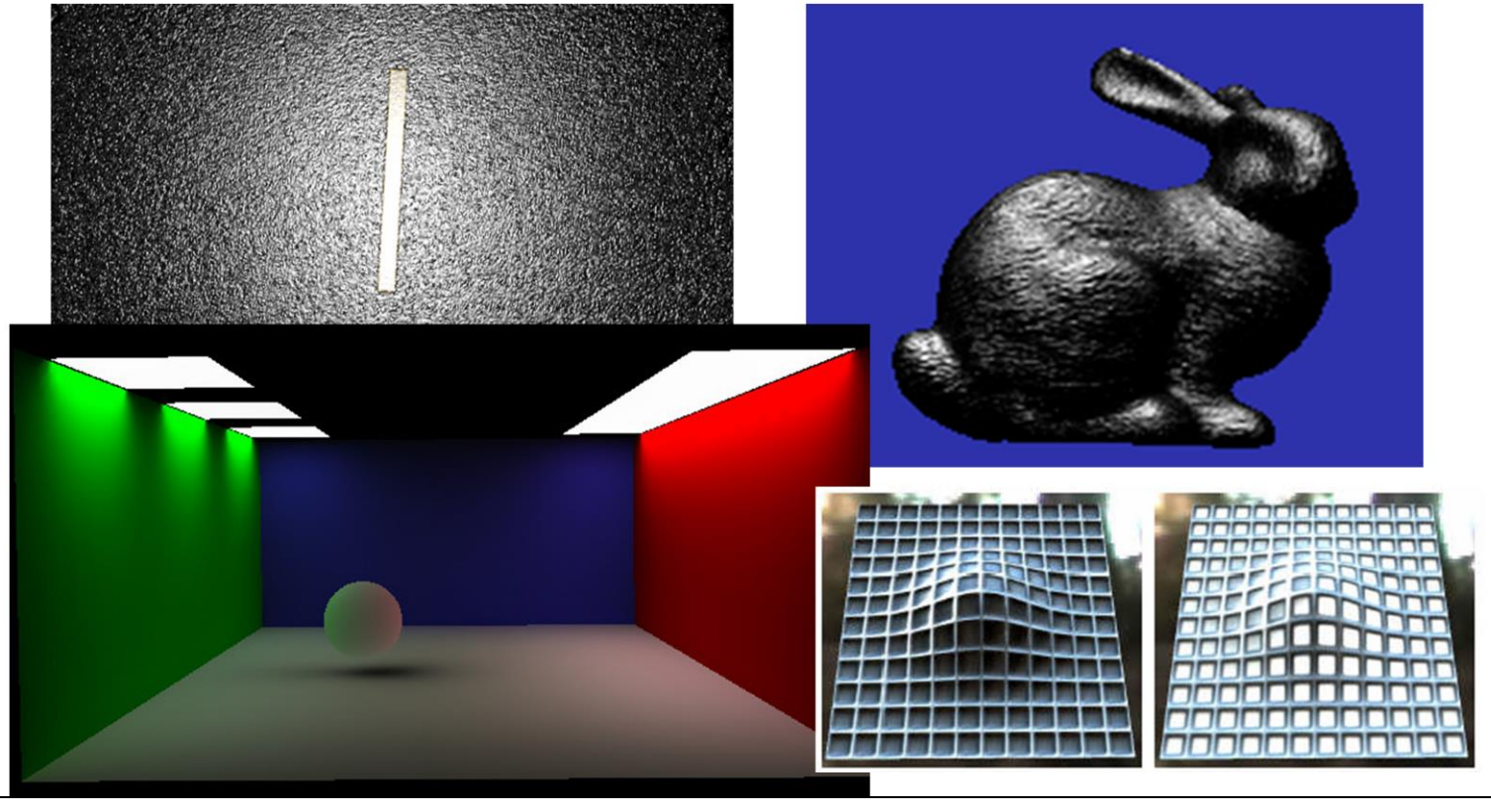
Microsoft DIRECTX®

NAUGHTY DOG

Part II: Techniques

Peter-Pike Sloan and Dan Baker

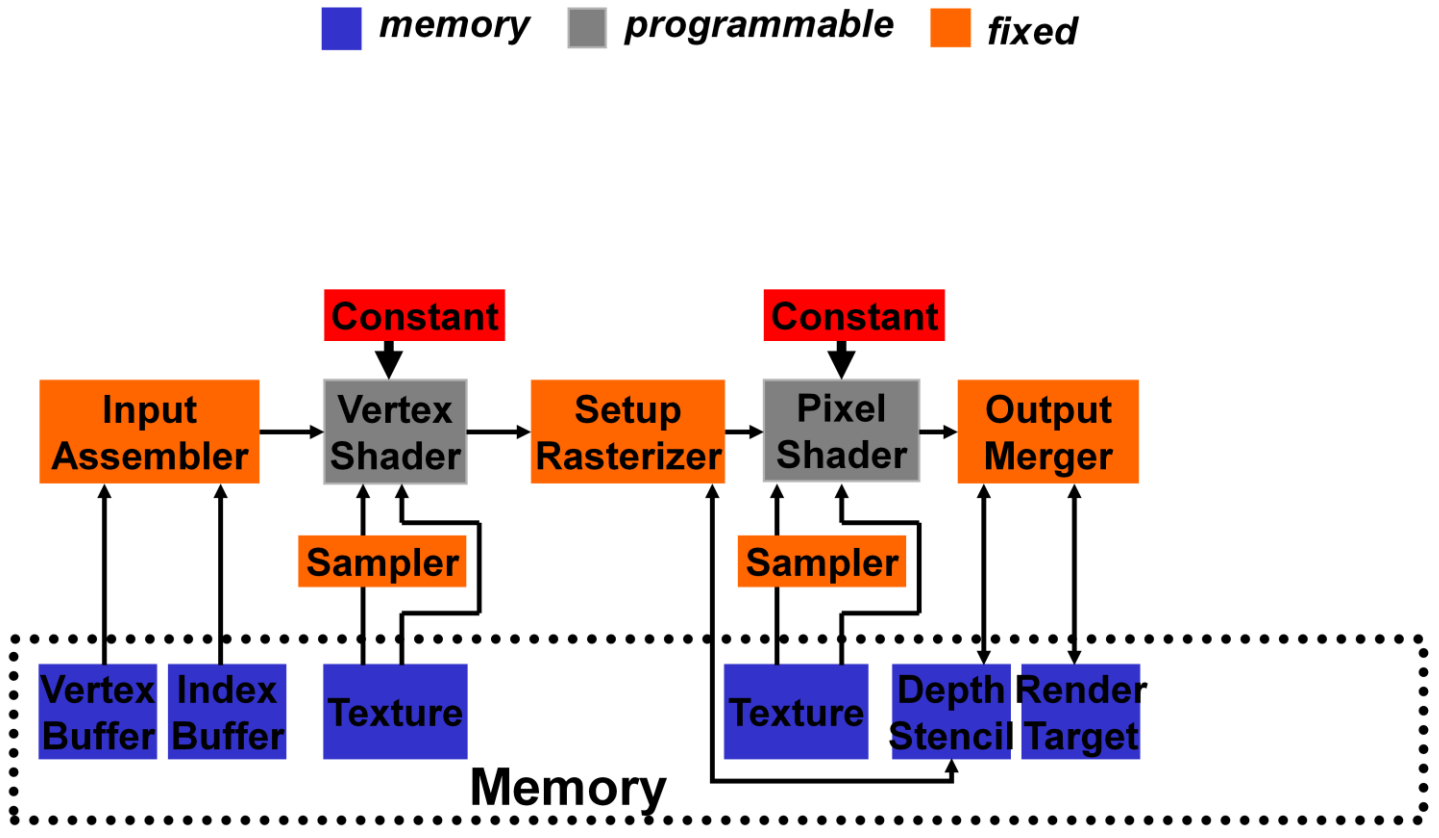


Implementation of Reflectance Models for Games

How Do We Evaluate a Lighting Model?

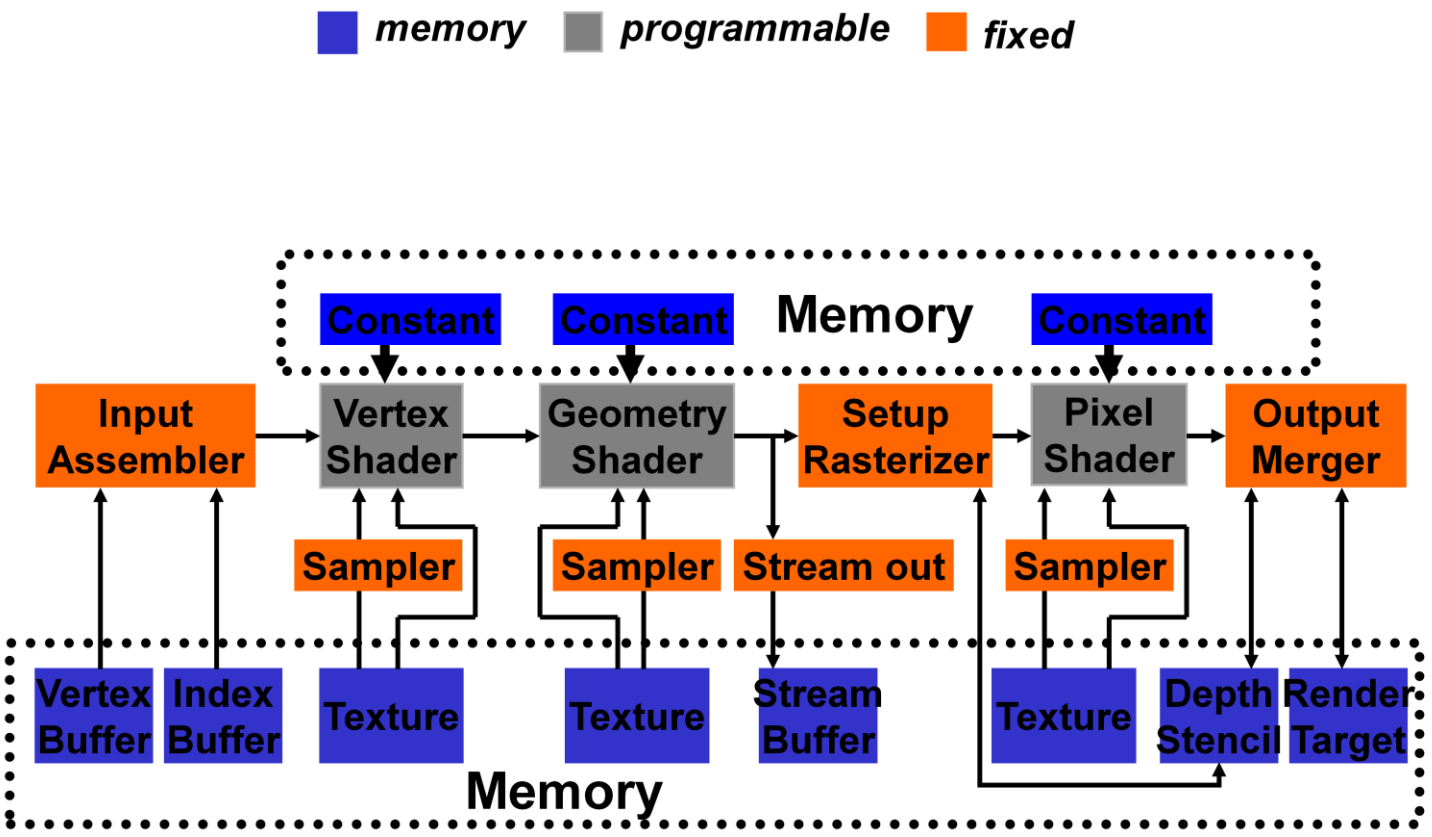
- Real time graphics technology is based on rasterization
- Currently, our pipeline consists of triangles, which get processed by a vertex shader, and then rasterized
- The rasterization step produces pixels (aka fragments), which are then processed and placed into a rendertarget

A Modern Real Time Graphics System



Both shader cores read from memory

Tomorrows Graphic's System



Where We Have Control

- Shader controls each process point
- With some restrictions, can perform any computation we want
- A reflectance model can be evaluated at any of these stages, or any combination
- Historically, computations were done on vertex levels, but have steadily moved into pixels
- Soon, will be able to shade on a triangle

Frequency of Evaluation

- Generally, vertex evaluation is lowest frequency, while pixel is at a higher frequency
- Not the case for low resolution
- Small triangles (sliver) not rendered by realtime hardware
- Geometry aliasing avoided by not drawing sliver triangles

An Example BRDF

- In HLSL, the Blinn-Phong model looks like:

```
float3 BlinnPhong(float3 L, float3 V)
{
   float3 H = normalize(V + L);
   float3 C = Ks*pow(dot(H,
   float3(0,0,1)), Power) +
   Kd*dot(N,L);
}
```

Gamma Space

- BRDFs operate in linear space, not gamma space
- Most albedo textures are authored in gamma space, so must convert into linear space in the shader (and convert them back into gamma space)
- Can use SRGB to convert gamma albedo textures into linear space, gives more precision where needed and acts as a compression scheme

Gamma Correcting

- If we render straight to the screen, and backbuffer isn't linear (usual case), need to go into gamma
- sqrt is a close approximation

```
float3 BlinnPhong(float3 L, float3 V)
{
   float3 H = normalize(V + L);
   float3 C = Ks*pow(dot(H,
   float3(0,0,1)), Power) + Kd*dot(N,L);

   C = sqrt(C);
}
```

Dynamic Range

- BRDFs can have a wide range of color intensities
- 8 bit per channel backbuffer does poor job of capturing range
- Easy for information to be lost – e.g. clamped to max value, or not enough precision

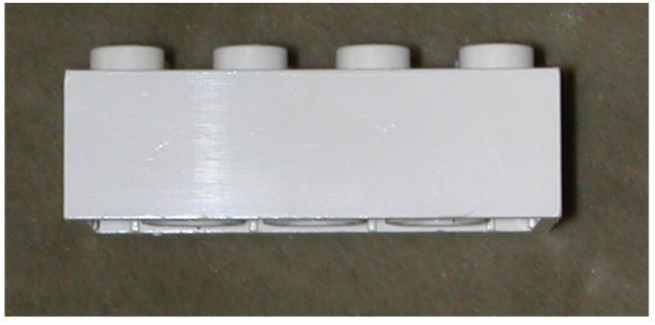
Blooming/Tone Mapping

- Bright pixels bleed to neighbors
- Exposure control
- If these are used – BRDF will store color value stored in rendertarget, and an image space filter applied
- Likely de-facto standard in future

How Do We Evaluate a BRDF?

- Direct Evaluation
  - Make an ALU program in the GPU
- Texture Evaluation
  - Perform a texture lookup
- A combination of the 2.
  - Factor some things into textures
  - Do others with ALU

How Do We Represent Material Variation?



- Microvariation is important
- Want more then just a single BRDF for a surface, this BRDF must change across the surface
- Bumpmapping is a coarse variation

Which Reflectance Model to Use

- Different reflectance models have different costs
- Cost – Data
- Cost – ALU
- Cost – Time to make content
- Cost – Accuracy
- Consider: Strong trend toward more ALU, less data

BRDF Costs, Minimal Surface Variation

Model	Texture costs	ALU costs
Blinn-Phong Direct	0	7
Blinn-Phong factored	1	2
Banks Direct	0	12
Banks Factored	1	5
Ashikhmin/Shirley	0	40
Ashikhmin/Shirley factored	4	10
Lafortune Direct	0	10 + 5 *n
Cook-Torrance	0	35

BRDF Costs With Surface Variation

Model	Texture costs	ALU costs
Blinn-Phong Direct	1	15
Blinn-Phong factored	2	10
Banks Direct	1	25
Banks Factored	2	18
Ashikhmin/Shirley	2	50 (60)*
Ashikhmin/Shirley factored	6	30
Lafortune Direct	2	30 + 5*Lobes
Cook-Torrance	1	40

*data cost is similar, so Ashikhmin/Shirley looks attractive, and also Blinn-Phong.

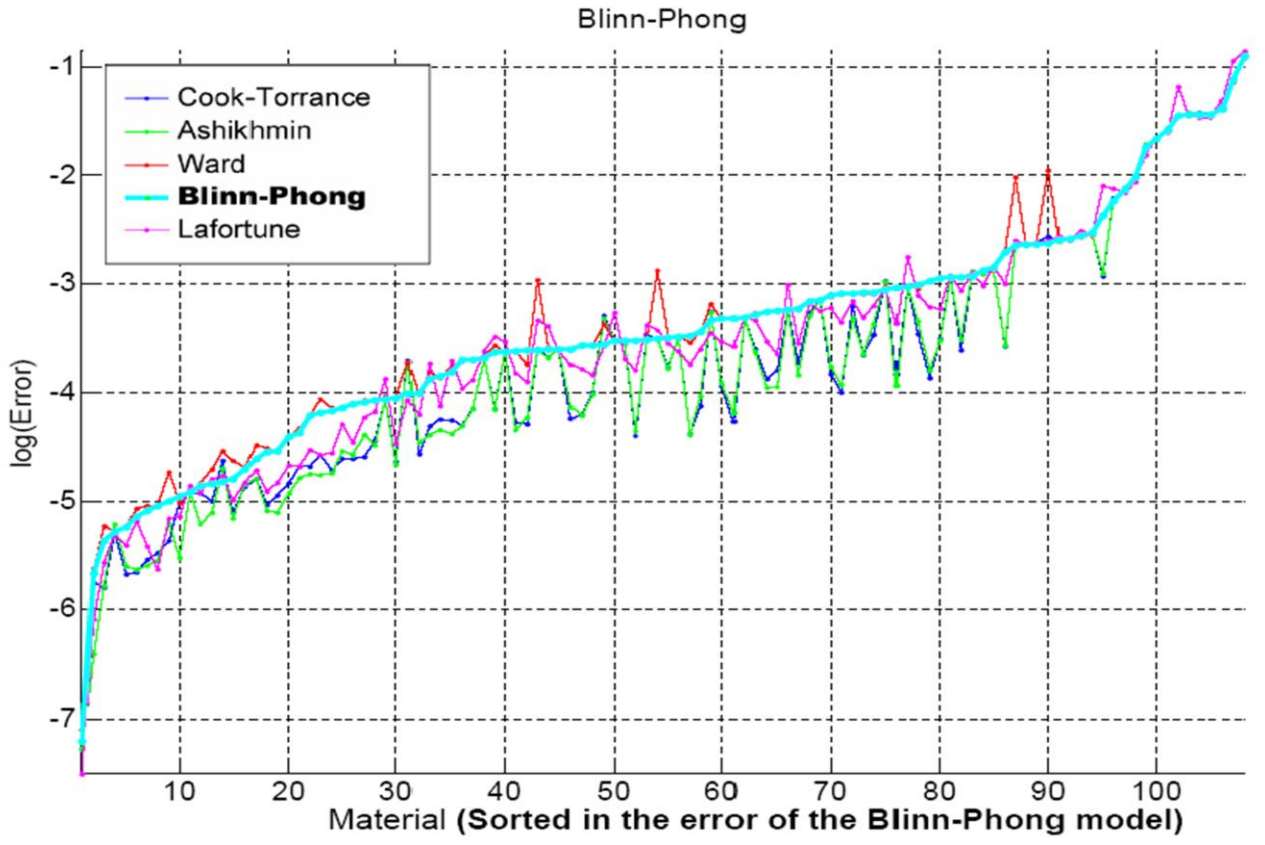


chart courtesy of Addy Ngan, Frédo Durand, and Wojciech Matusik

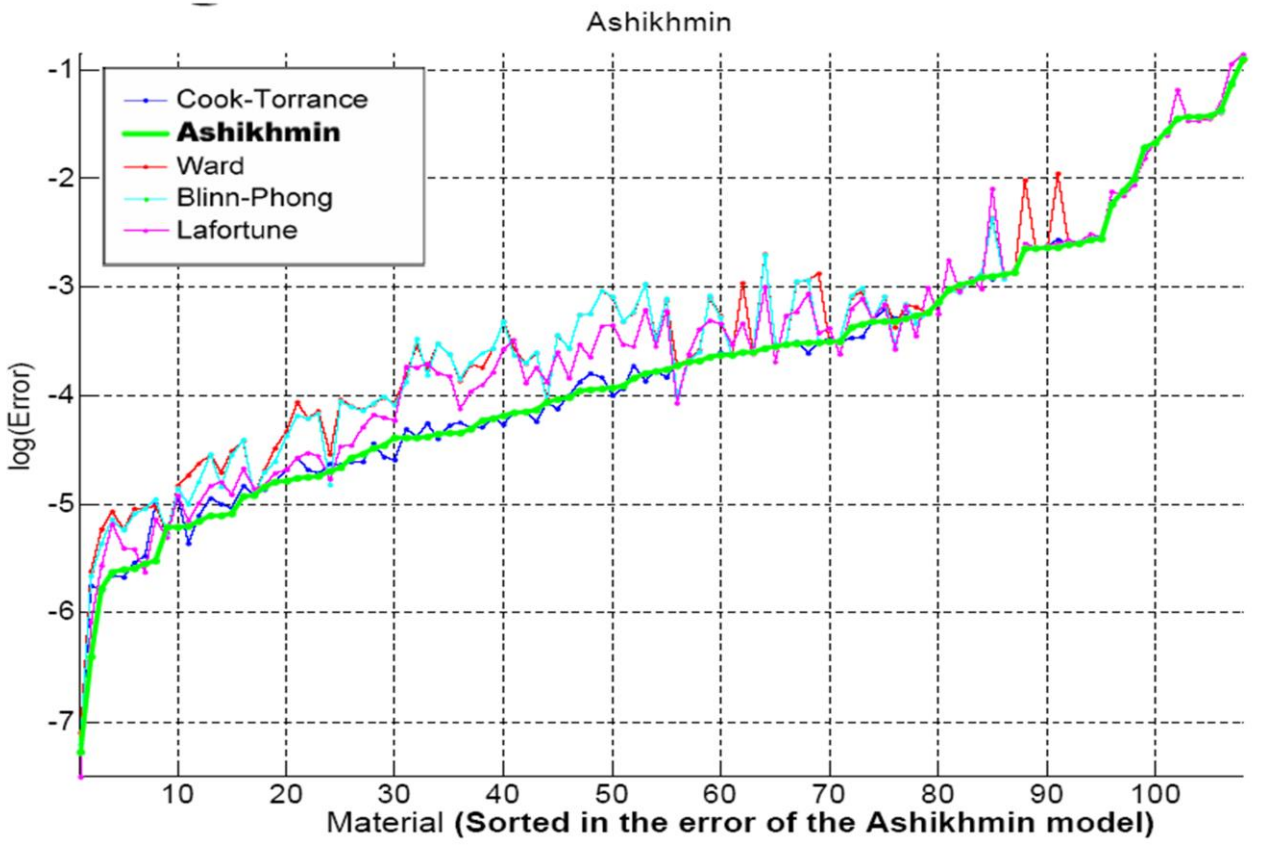


chart courtesy of Addy Ngan, Frédo Durand, and Wojciech Matusik

Getting Variation

- Could sample real materials to generate a BRDF map [McAllister SBRDF].
- But, variation usually done by an artist.
- Sometimes we get variation by using an understanding of mesogeometry
- Sometimes it is done by making changes in the assumption of the microgeometry

Simple Variation

```
float3 BlinnPhong(float3 L, float3 V, float2 texCrd, sampler
      GlossMap, sampler ColorMap)
{
      float4 Gloss = tex2D(GlossMap, texCrd);
      float  Power = Gloss.w;
      float3 Kd = tex2D(ColorMap, texCrd),
      float3 H = normalize(V + L);

      return Gloss*Ks*pow(dot(H,    t3(0,0,1
      + Kd*dot(N,L);
}
```

This adjusts the reflection color of the specular component, and changes its power

Gives our object a diffuse color at each point..

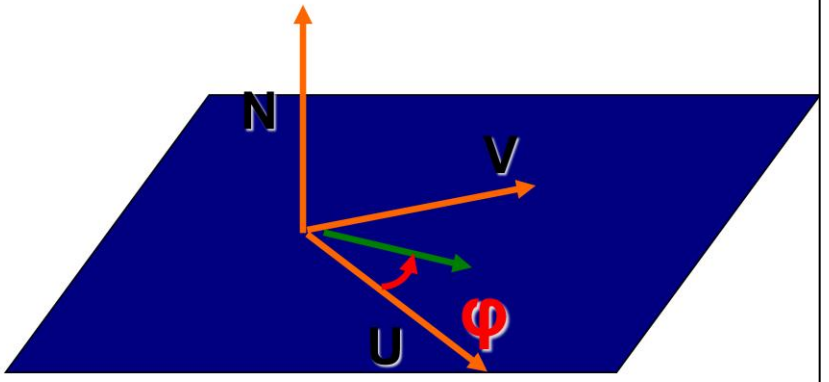
Power is an example of micro variation

Bump Mapping

Bump mapping is a meso level variation

Twist Mapping, Etc.

- Could also store the tangent (twist)
- Can store an Angle
- Sometimes, convenient just to store the first two components of the Tangent, and assume z is 0



Twist mapping can be used for both meso and micro variation

Implementing Tangents and Normals

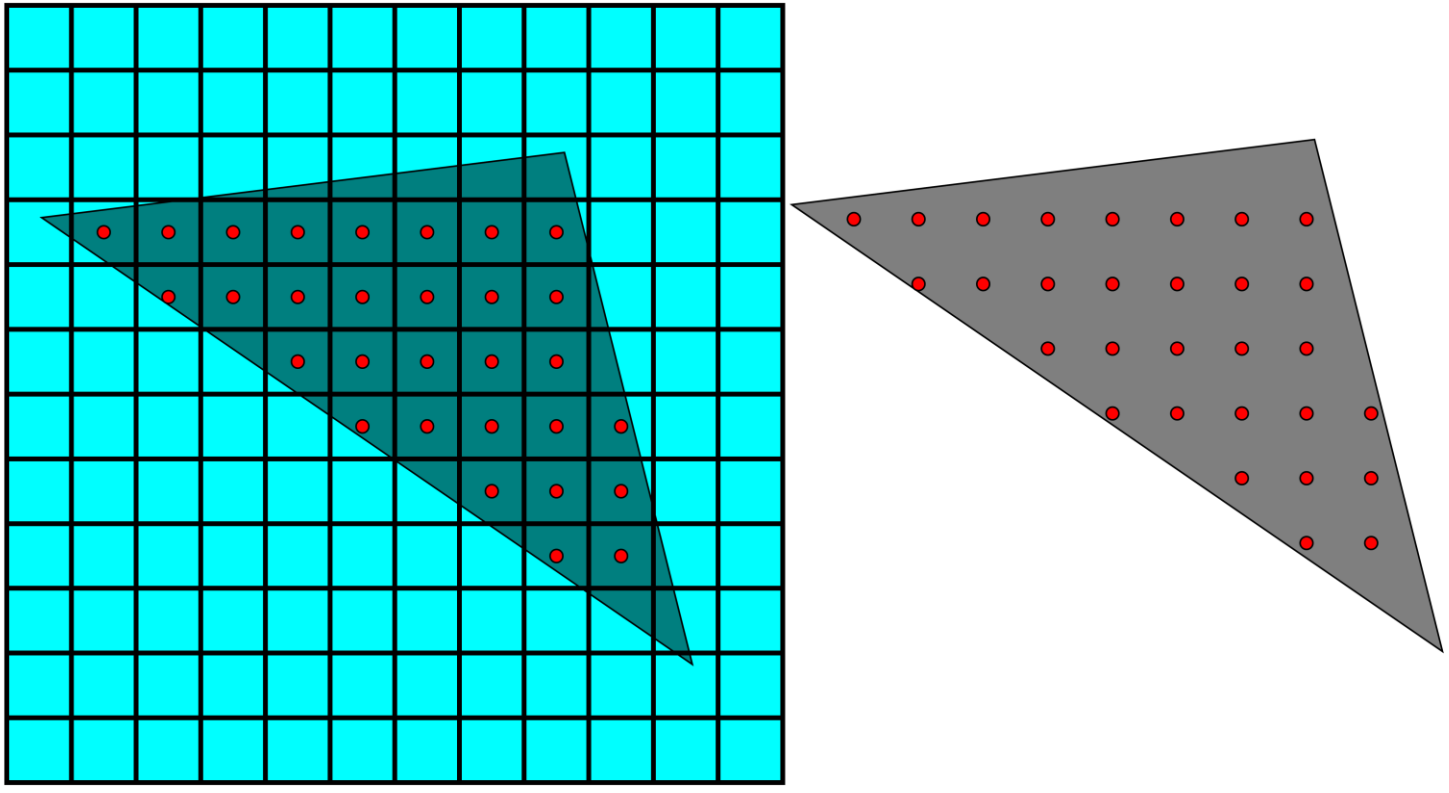
- Must rotate all vector data into per-pixel local fame
- This frame isn't the (per-vertex) tangent space, but rather a per-pixel space above tangent space
- Can implement all BRDFs such that normal can be assumed to be (0,0,1), and the tangents (1,0,0) and (0,1,0) respectively

$$\begin{pmatrix} \text{Tangent} \\ \text{Tangent X Normal} \\ \text{Normal} \end{pmatrix} \qquad \begin{pmatrix} \text{Tangent} \\ \text{Tangent X Normal} \\ \text{Normal} \end{pmatrix}^{T}$$

Adding Tangent Support

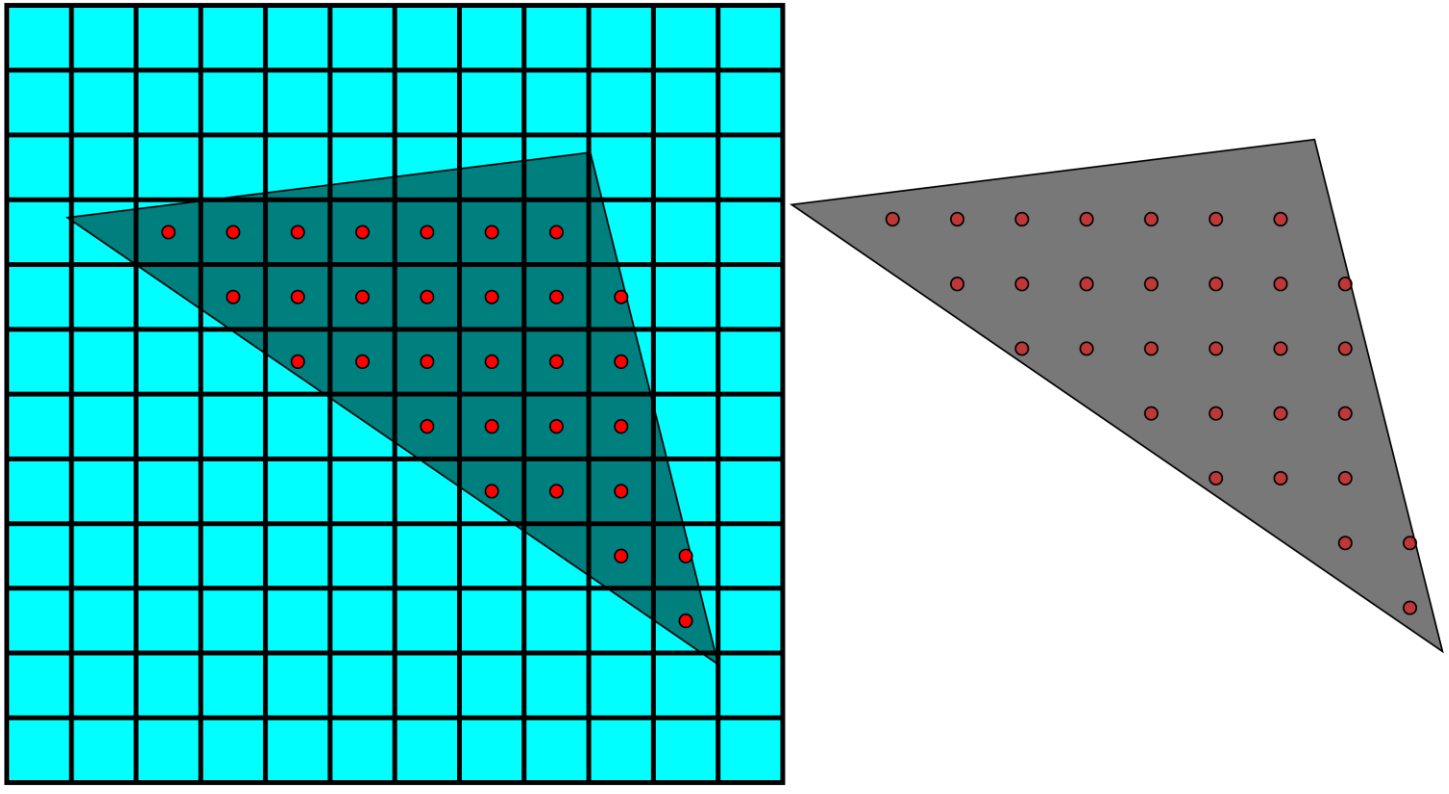
```
float4 Shader(float2 TexCrd: TEXCOORD,float3 LightDir :
    TEXCOORD2,                                          float3
    ViewDir : TEXCOORD3, sampler NormalMap,
              sampler TwistMap ) : COLOR
{
    float4 Out = 0;
    float3 Normal  = tex2D(NormalMap, TexCrd);
    float3 Tangent = tex2D(TwistMap, TexCrd);
    float3x3 reverse = float3x3( Tangent,
                                 cross(Normal,Tangent),
                                 Normal);
    reverse = tranpose(reverse);

        Out.xyz = saturate(dot(Normal, LightDir)) / PI
            * BRDF(mul(ViewDir,reverse), mul(-
    LightDir,reverse);
    return Out;
}
```

Pixel Level Evaluation



Pixel shader will be evaluated at each one of these points

Pixel Level Evaluation, Shift



Shifting the triangle causes the sample points to change

Shifting Samples

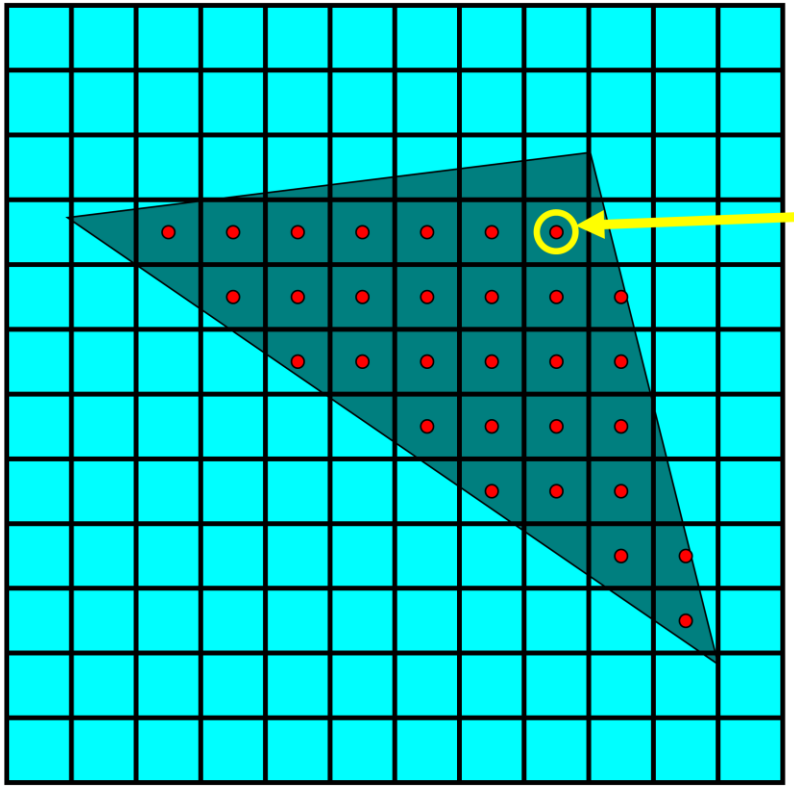
- The sample points change drastically as the triangle moves and changes size
- If using micro variation, e.g. loading data elements from a texture, the evaluation can be significantly different
- Texture filters – solves this problem for linear data elements
- Linear for temporal, MIP mapping for resolution changes

Swimming is priority #1, show demo here

But…

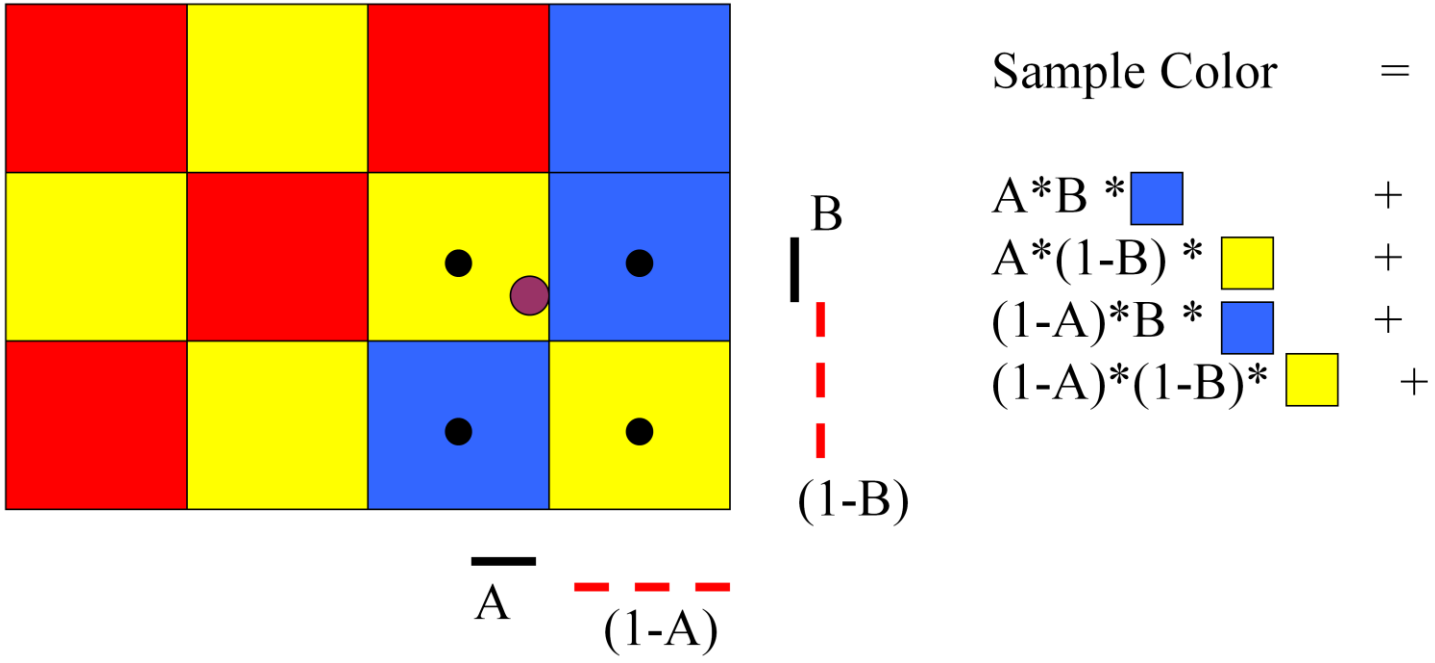
- Texture filters are linear
- BRDFs are usually non linear
- Get some image stability, but not great results
- Must at least prevent radical shifts in image
- Often, variation must be mitigated
- Would be nice to texel shade instead

What Does a Sample Point Mean?



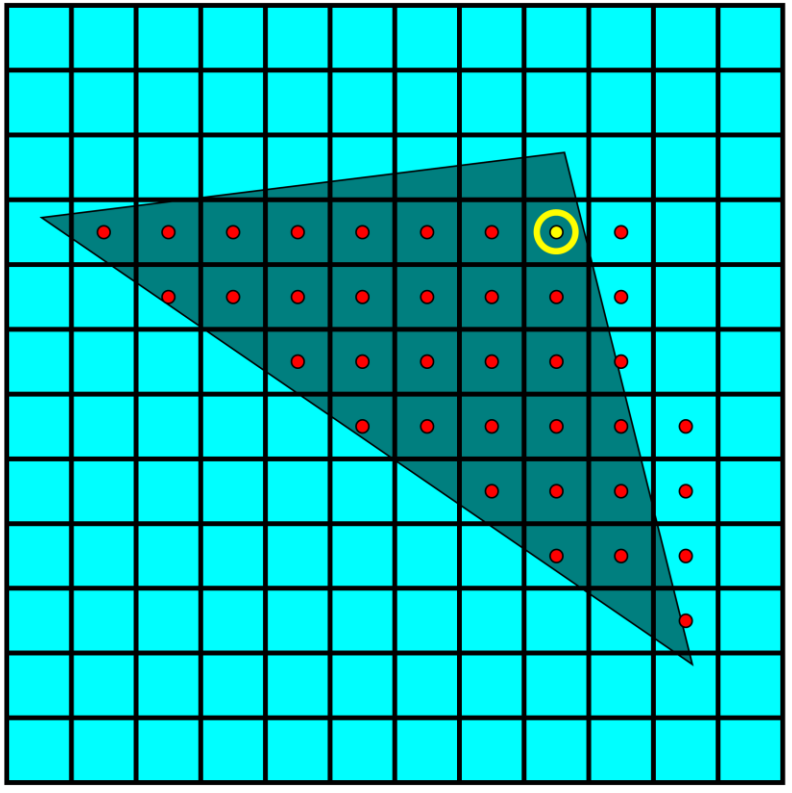
What happens when this pixel is evaluated? Given a set of parameters interpolated from the triangles 3 vertices, how do we go about generating a color?

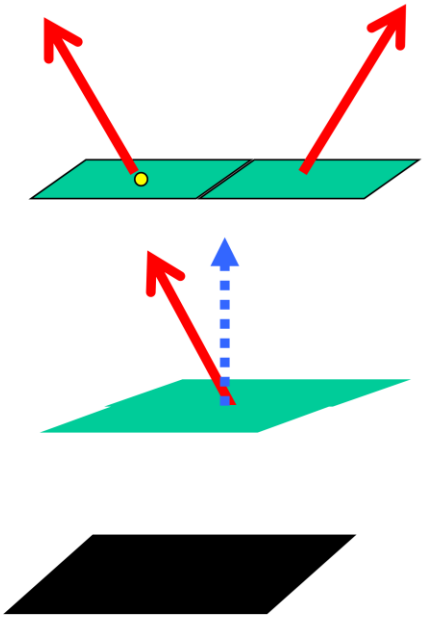
Texture Filtering Review



Don't point sample from our resources, we run through a texture filter to even out the samples

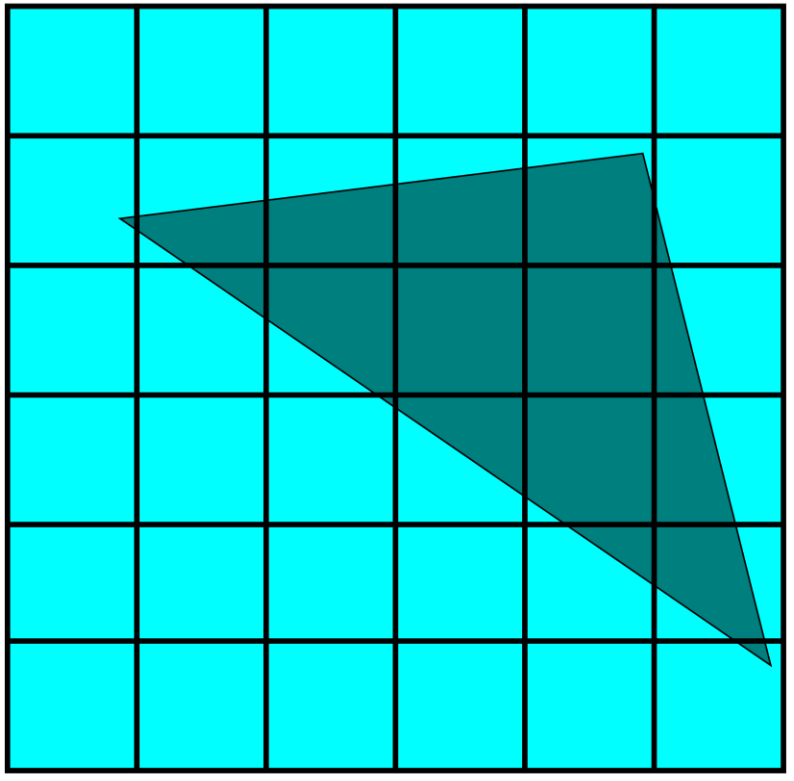
Texture swimming





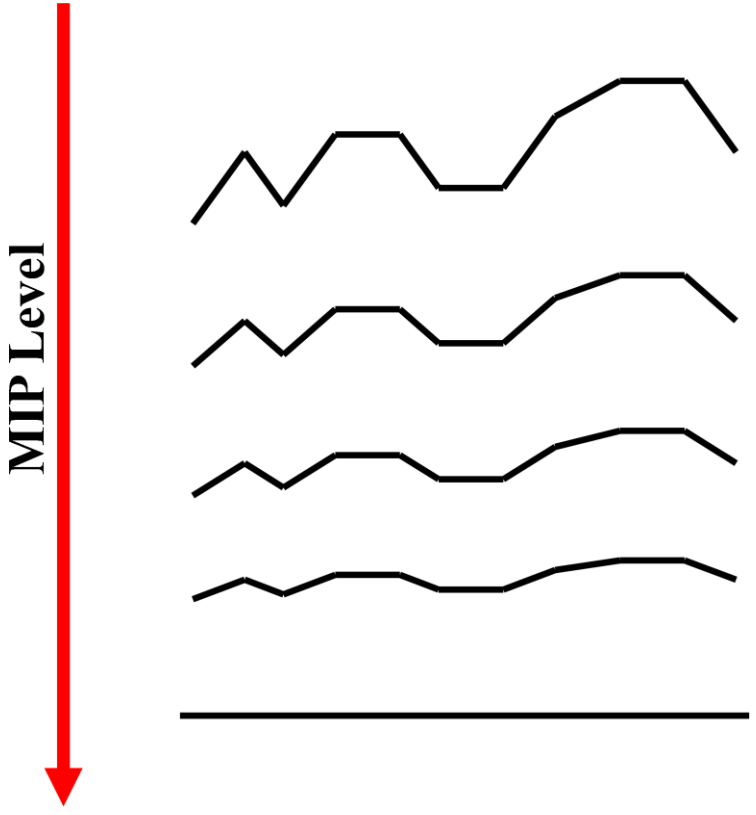
Not only is this pixel swimming, but color doesn't bleed to adjacent pixels in any meaningful way. Average color for a neighborhood shouldn't be changing much.

What about lower resolutions?



Roughly half as many pixels get executed

A common hack: Level of Detail

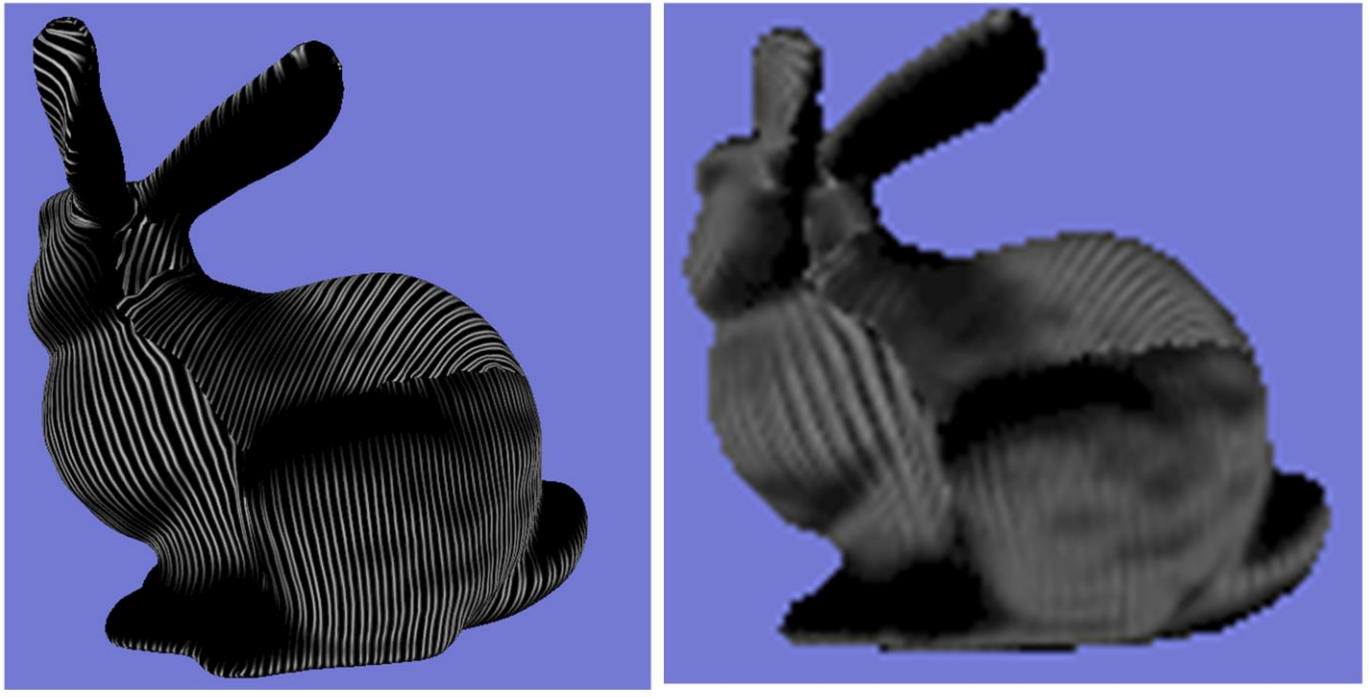


By lowering the amplitude
Of the displacement, effectively decreasing to a less complex model as the model moves further away.
This will prevent aliasing, but isn't accurate

Scale independent lighting

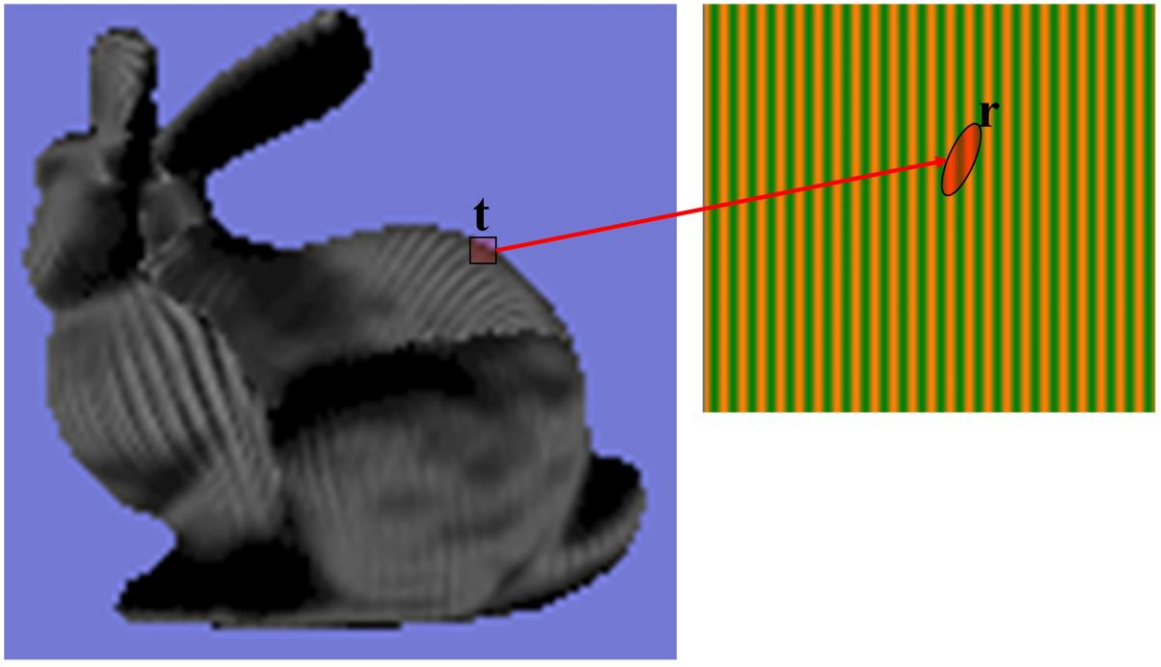
- Ideally, the size in pixels of an object on the screen should not affect its overall appearance
- High frequency detail should disappear
- Global effects should stay the same
- Reasoning behind MIP mapping
- Shouldn't see drastic changes in image as sample points change

Scale independent lighting



A low resolution rendering of an object should look like a scaled down version of the high resolution one

MIP Mapping



A pixel on the screen is the sum of all the texels which contribute to it. The right represents the region of the normal map that the pixels on the rendered image might map to.

Mip mapping for diffuse

- For a simple diffuse case, the lighting calculating can be approximately refactored
- The normal integration can also be substituted by a MIP map aware texture filter

$$\sum_r (L \cdot N_r) A_r W_r \approx (L \cdot \sum_r N_r W_r) * \sum_r A_r W_r$$

$$\sum_r (L \cdot N_r) T_r W_r \approx dot(L, tex2D(normalmap, t)) * tex2D(colormap, t)$$

Non Linear Lighting models

$$\sum_r W_r (N_r \cdot H)^p \neq (H \cdot \sum W_r * N_r)^p$$

$$\sum_r W_r (N_r \cdot H)^p \neq (H \cdot tex2D(normalmap, t))^p$$

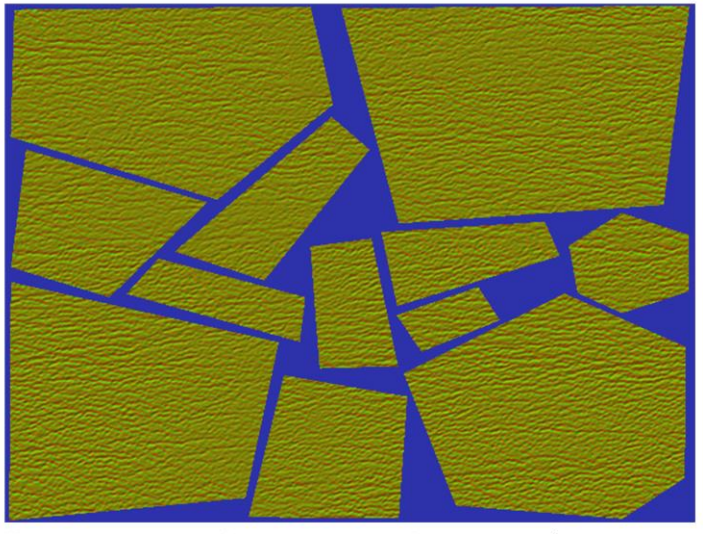
Blinn-Phong isn't Linear

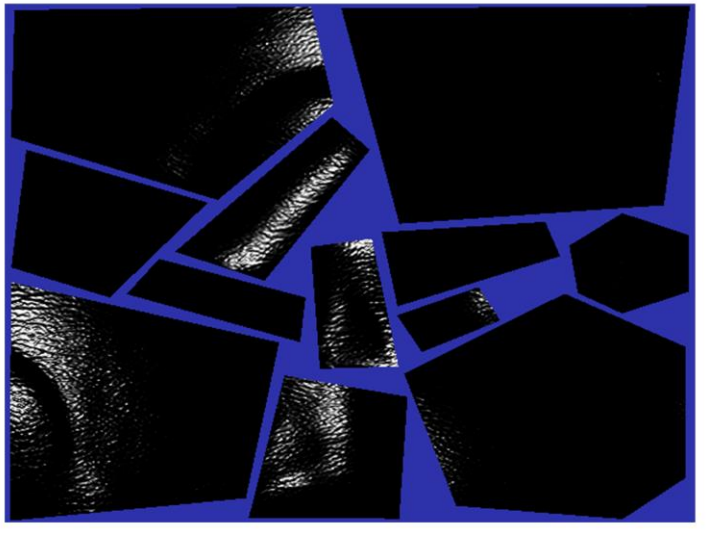
Looking at just the specular component of Blinn-Phong with a normal that varies (the tex2D(r) part), we can see that we cannot use a linear filter. We can think of this as shading on the texels rather then the pixels.

Texture Space Lighting

- Rasterize using texture coordinates as a position
- Equivalent to explicitly evaluating the summation
- An object needs a texture atlas
- The values in the texture space are now colors – and are linear
- Create a MIP chain explicitly or through the auto MIP gen option
- Render the object with this MIP chain

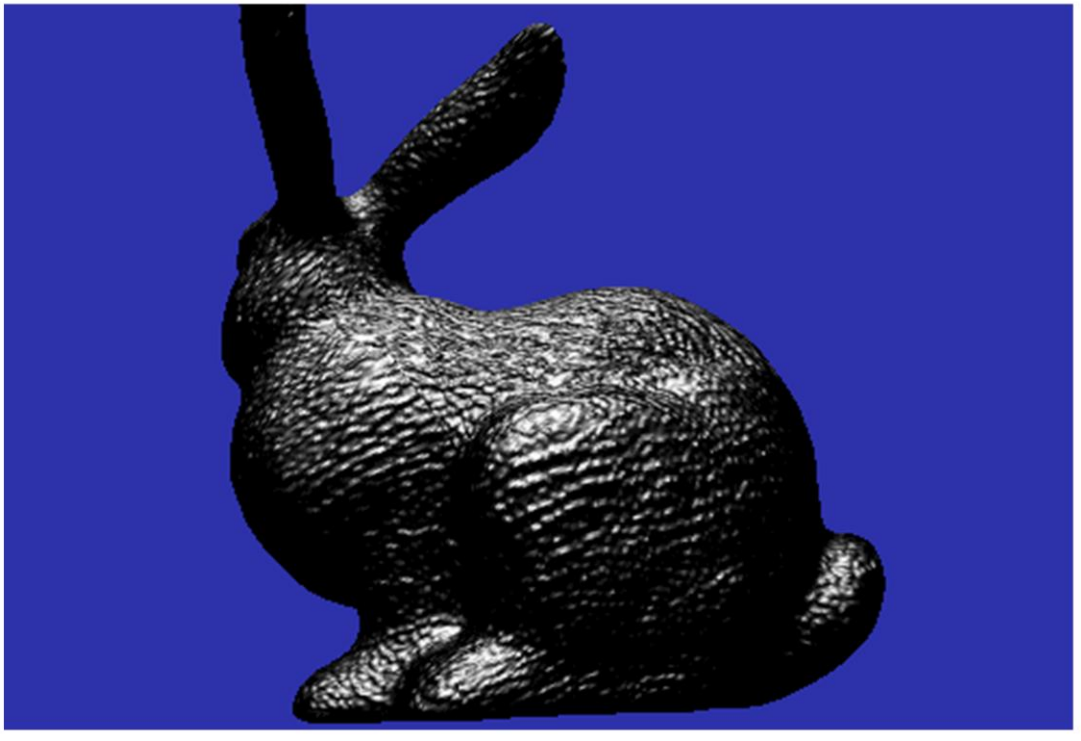
Texture Space Lighting





Rasterize triangles using texture coordinates as positions, the left image is the normal sampled at each point, and the right image is the computed lighting

Texture Space lighting

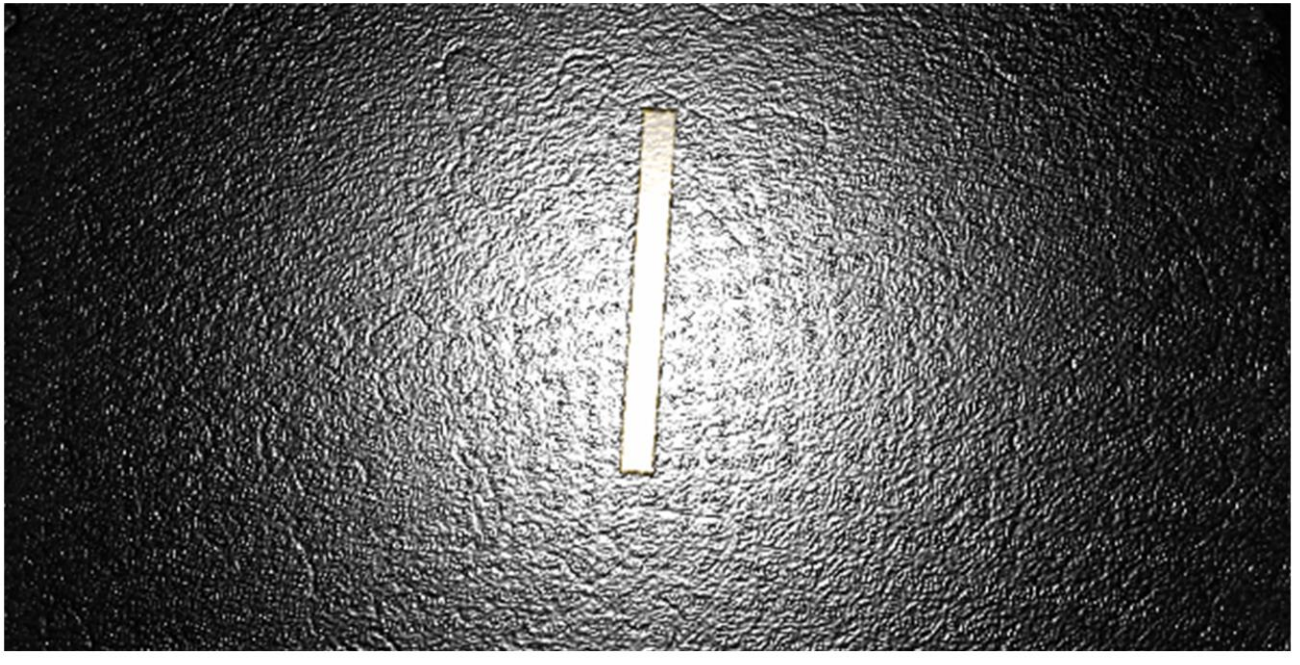


We paste the texture onto the object

TSL: A case study

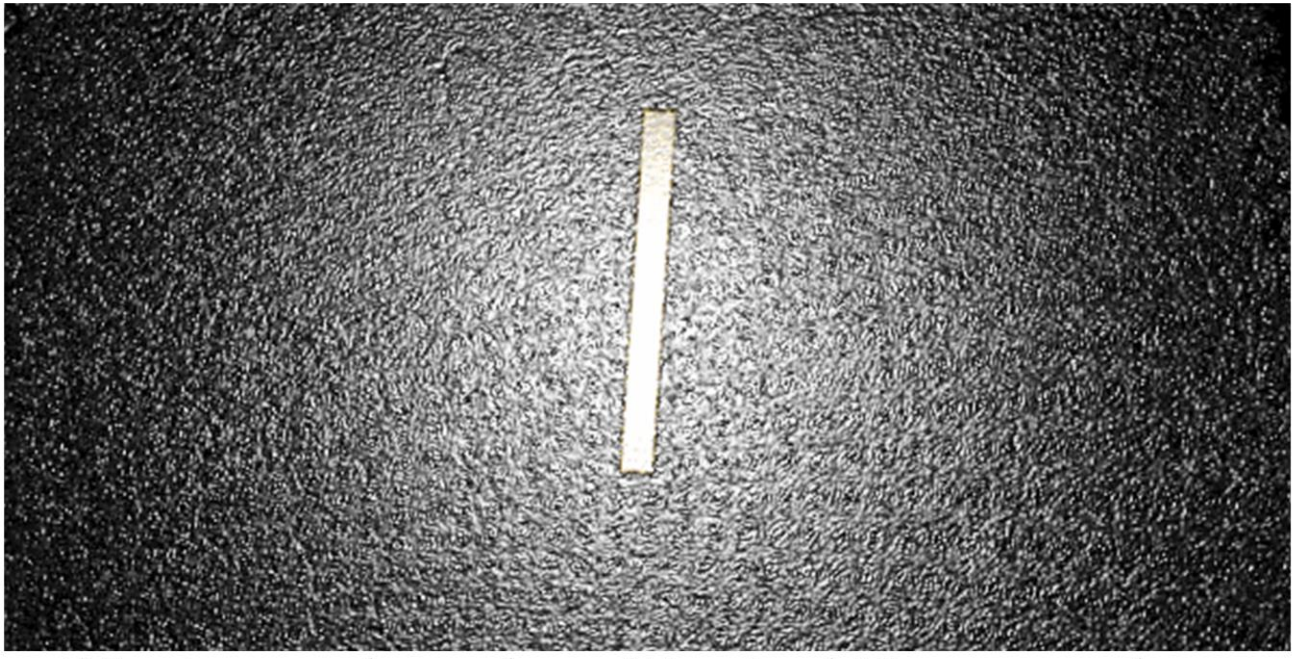
- Texture Space Lighting (TSL) can allows us to have high frequency lighting on fast moving objects
- Will only have aliasing problems associated with standard filtering techniques
- A roadway is a perfect candidate, subtle specular reflections with a high degree of motion

A roadway, snapshots



We can see Moiré patterns on any filtered non linear functions. Additionally, temporal aliasing becomes problematic.

Using TSL



The artifacts are largely mitigated if we render the non linear function and MIP reduce. We can also see more high frequency detail.

How this demo works

- Each road segment is rendered into a 1024x1024 rendertarget in texture space - alpha is set to 0
- AUTOMIPGEN is then used to do a simple mip filter of the highest level rendering
- This model is then light with the prelit, mip filtered texture and rendered with full aniso
- Art content is simple - just an albedo texture, a normal map with a glossy term
- Everything is done in linear space

The Shader

```
void lightPositional(  …)
{
   // norm, light and half are in tangent space
   // power, Kd, Ks and norm come from textures
   …
   //normalized blinn phong
   float fS = pow(saturate(dot(norm, half)), power)) *
      power;
   float fD = saturate(dot(norm, light)) *
      diffuseIntensity;

   vColorOut.xyz = fS * Ks + fD * Kd;
   vColorOut.a = 1;

   //put us in gamma space for the direct rendering only
   if (bTextureView) vColorOut = sqrt(vColorOut);
}
```

Pasting the texture on the Scene

```
void light_from_texture( float2 vTexCoord: TEXCOORD0,
                         out float4 vColorOut: COLOR0 )
{
      // this is done with full anisotropy turned on
      // with SRGBTexture set to true
      vColorOut = tex2D(LightMapSampler, vTexCoord);

      //for atlased objects, we do not want to blend in
      // unrendered pixels
      if( vColorOut.a > 1e-5 )
            vColorOut.xyz /= vColorOut.a;

      //put us in gamma space
      vColorOut = sqrt(vColorOut);
}
```

Drawing polygons on the backside

```
[emittype [triangle MyVType ]]
[maxvertexcount [3]]
void ClipGeometryShader(triangle MyVType TextureTri[3])
{
    float2 coord1 = project(TextureTri[0]);
    float2 coord2 = project(TextureTri[1]);
    float2 coord3 = project(TextureTri[2]);
    float3 Vec1 = float3(coord3 – coord1, 0);
    float3 Vec2 = float3(coord3 – coord2, 0);
    float Sign = cross(Vec1, Vec2).z;
    if (Sign > -EPSILON) {
        emit(TextureTri[0]);
        emit(TextureTri[1]);
        emit(TextureTri[2]);
        cut;
    }
}
```

Ideally, we don't want to draw polygons which are not visible. In the near future, we can cull away polygons which are facing the backside with a handy Geometry Shader.

Problems with TSL

- Main problem is performance
- We can't render each object in the screen at a fixed resolution and scale down
- We need a way to render the object at a reduced resolution which looks close to the higher detail one when zoomed out
- Basically, we need to know how to create non linear MIP chains