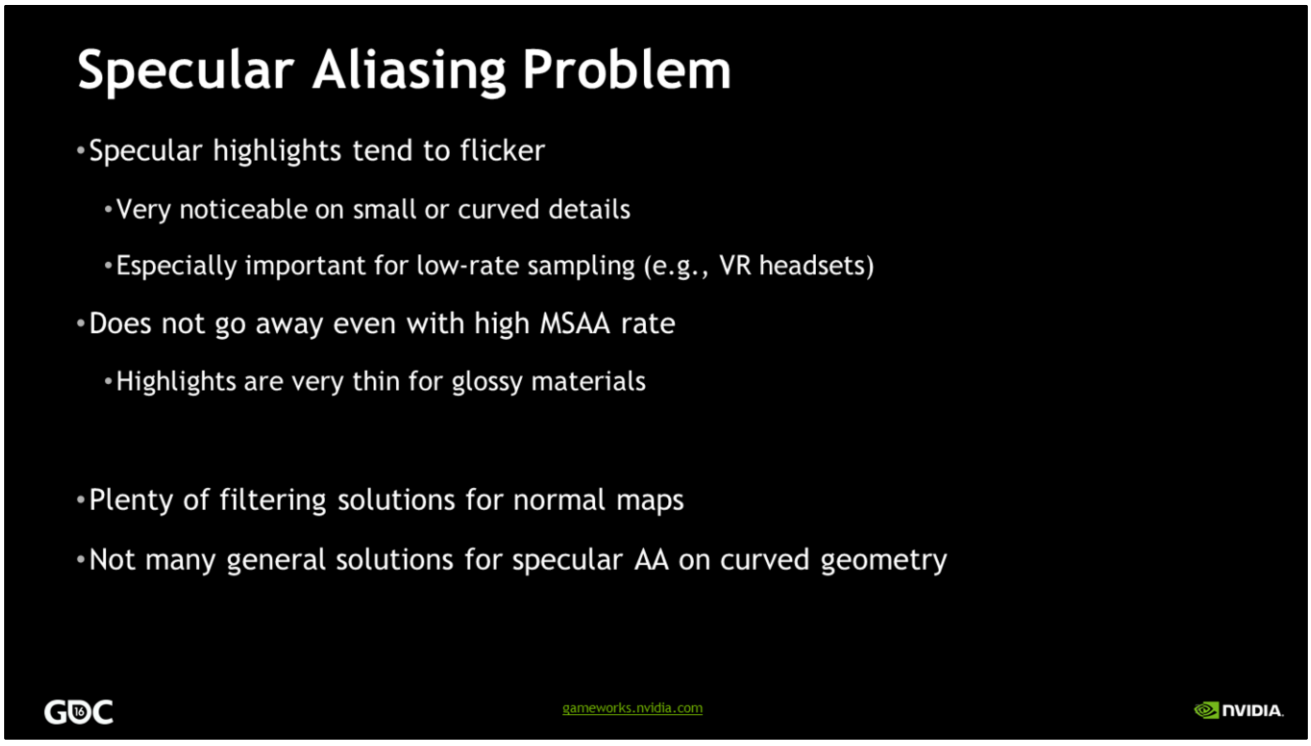

Stable Specular Highlights
Anton S. Kaplanyan, Nvidia Research, 28 March 2016
NVIDIA




Main aliasing in shading is caused by specular highlights and looks like sparkling.

The problem of a small highlight can be more pronounced on small curved geometry.

Specular sparkling is especially noticeable in VR headsets, where pixels cover large solid angles and the camera is in continuous movement due to the head tracking.

Throwing even up to 16x MSAA rate does not help with the problem because specular highlights are usually smaller than geometric details.

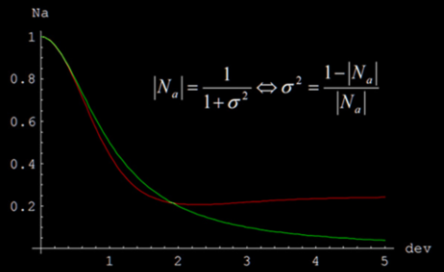
There were many improvements to the stability of specular shading, most of them are related to the problem of normal map filtering (e.g., LEAN/CLEAN, vMF, etc.).

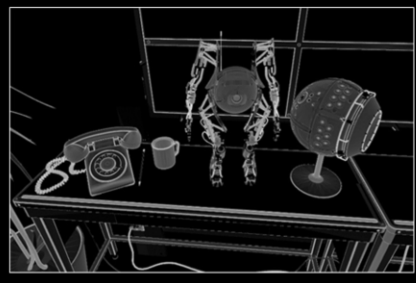
However, not many methods take into account all factors, including geometry curvature.

# Prominent Solutions

- Toksvig Gaussian fit normals *[Toksvig04, Hill12]*
  - Look at the spread of shading normals (ddx/ddy)
  - Compute a standard deviation of the spread
  - Use it to increase the roughness
- Roughness clamping *[Vlachos15]*
  - Compute curvature with ddx/ddy of shading normal
  - Clamp roughness using an empirical formula

$$r = \max\left(r, \sqrt[3]{\max(dndx^2, dndy^2)}\right)$$



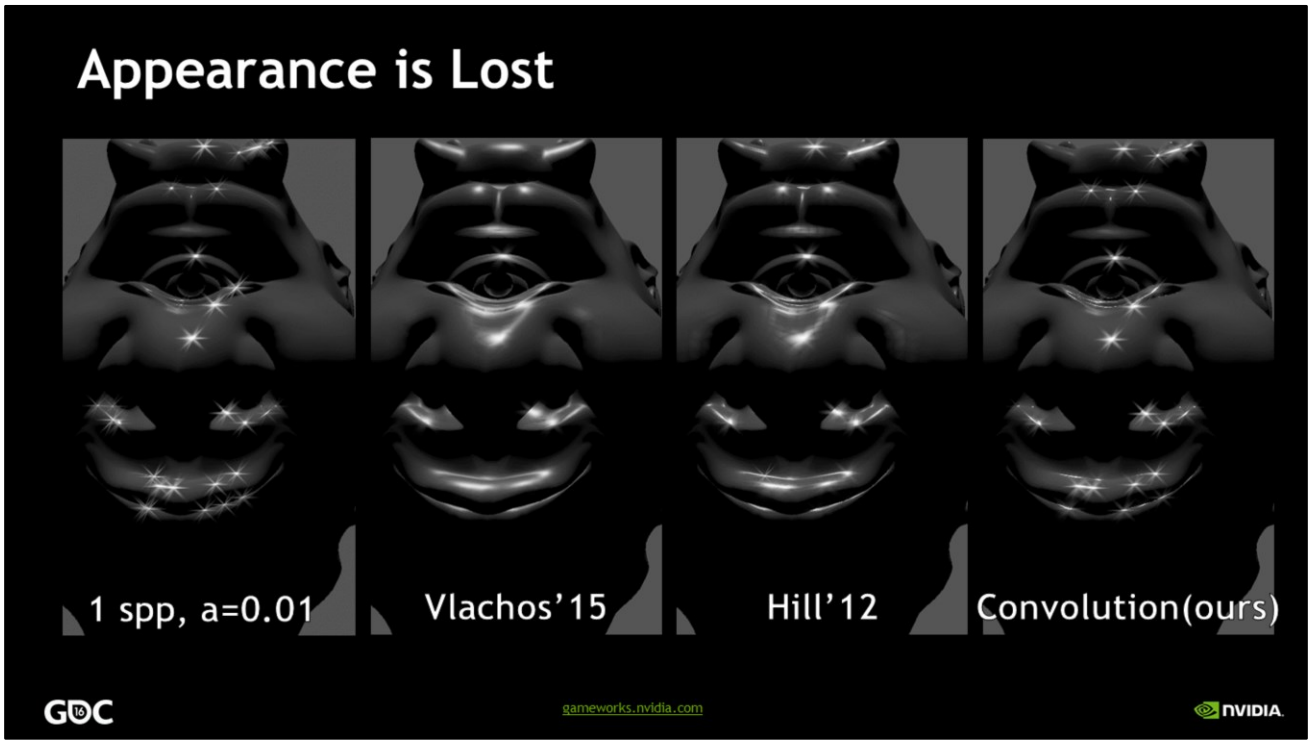


One such solution is to look at the spread of shading normals induced by the curvature of the surface.

Michael Toksvig introduced a Gaussian fitting into a spread of normals, which was then used to adjust the specular power.

This method was used by Stephen Hill (SIGGRAPH 2012) along with the estimation of normal derivatives using ddx/ddy.

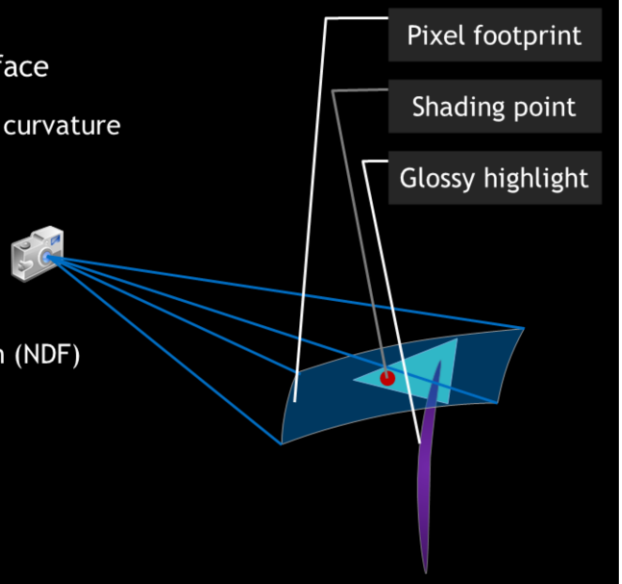
Alex Vlachos (GDC 2015) proposed a similar method with another empirical formula for clamping the roughness from below.




Both of these methods, while reducing specular aliasing, oftentimes lead to the change of the material appearance, especially on the objects with constant curvature.

# Why it Sparkles

- Pixel footprint spans a large area on the surface
  - Strongly anisotropic at grazing angles and with curvature
- Specular highlight can be small
  - As small as 1/10 and even 1/100 of a pixel
  - Gets brighter when gets smaller
  - Caused by a sharp Normal Distribution Function (NDF)
- We shade at a **single** point at sample center
  - Can easily miss the highlight






Looking at the anatomy of a sparkling highlight, the sparkling problem appears because we have a relatively large pixel footprint, while the specular highlight takes a relatively small area of it. The problem manifests itself even more when the footprint projected on the surface is stretched by grazing angles or high curvature.

Specular highlights of highly glossy materials can be very small, taking a tiny fraction of a pixel.

When using physically based materials, the highlight also gets brighter when decreasing the roughness due to the energy conservation. This amplifies the sparking problem, making it near practically impossible to render highly glossy materials.

With a small and bright highlight, it is also hard to find it on the surface. Because with rasterization we usually shade once per pixel per triangle, this shading point rarely hits the bright highlight. This makes the sparking prominent during camera movement.

Now we will look into the NDF part of the shading.

# Filtering a Normal Distribution Function

- During shading $h$ is a halfway vector between incident and outgoing direction

$$h = T\frac{w_i + w_o}{||w_i + w_o||}$$

  - Lives in local shading frame (transformed by matrix $T$)
- NDF $D(h)$: how many microfacets on a unit patch have a normal $h$
  - Vector $h$ sets a normal for perfect reflection
- On a pixel footprint $h$ can vary due to
  - Change of the tangent frame $T$ (surface curvature)
  - Change of the incident and outgoing direction
- Want to integrate NDF over the *whole footprint*



For shading with physically based microfacet materials, we use a vector h, a halfway vector between the incident and the outgoing direction. This vector is defined in the local shading frame on the surface, which is achieved by multiplying by the rotation matrix T.

Normal Distribution Function (NDF) is then used to query how many perfectly reflecting microfacets are aligned with the half vector h on average. This models rough reflection and is used for shading.

If a pixel footprint is projected on a curved surface, it takes a large area and thus the half vector h can significantly change within the pixel footprint. This is the same problem of hitting a small highlight with a single shading sample, viewed from the point of the NDF.

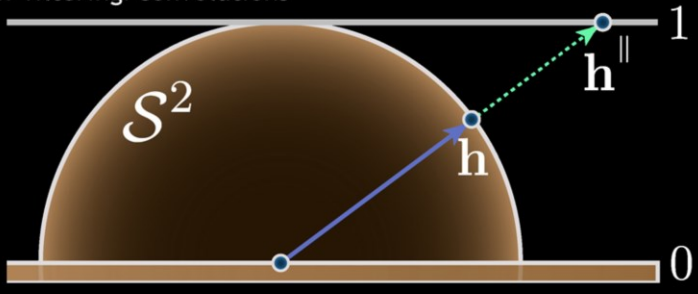
Therefore, in order to eliminate shading aliasing, we want to filter the NDF with the variation of half vectors on the pixel footprint.

# Working with Half-vectors in Parallel Plane

- Half-vector lives in parallel plane domain (aka slope domain)

$$h = T\frac{w_i + w_o}{|\langle (w_i + w_o) \cdot n \rangle|}$$

  - Beckmann is a 2D Gaussian
  - GGX is an NDF of an ellipsoid
  - Convenient for filtering/convolutions



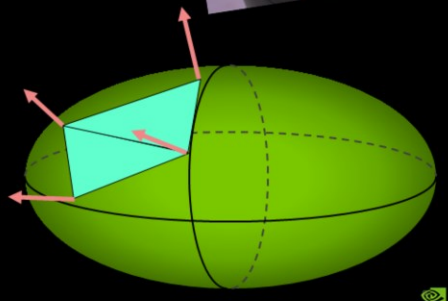



As an important detour, we need to redefine the half vector h to be in the parallel plane domain. This domain, also known as the slope domain for microfacet materials, is more suitable for filtering the NDF.

For example, Beckmann is just a 2D Gaussian in this domain; GGX distribution has a solid geometric interpretation.

# Shading a Triangle

- We shade a triangle with *varying* shading normal
- It's a virtual **quadric** for shading!
  - Can extract curvature
  - Use all three triangle's normals
  - Have their rate of change
- Smooth surface is described by many triangles
  - One quadric can span across multiple triangles

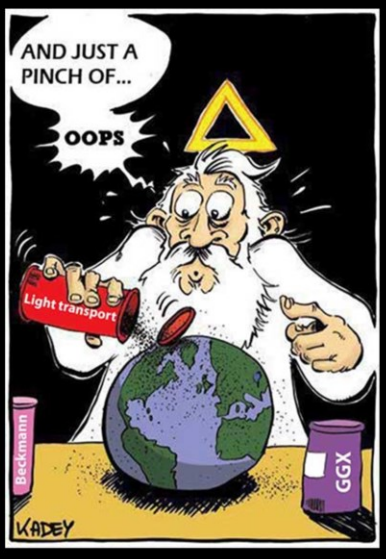





When shading a single triangle, we are given with three normals. This is just enough to define a quadric.

Moreover, this virtual quadric used for shading usually defines a larger virtual surface. This surface is described only by shading normals is usually a more accurate and smooth version of the surface described by geometric triangles. The shading surface can smoothly span across multiple triangles, making it meaningful to rely on its curvature.

# Shading Meets Light Transport





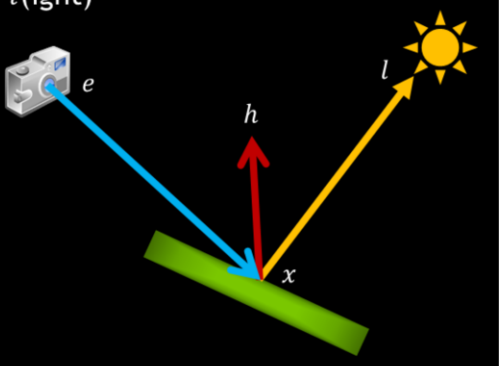

Now that we have overviewed the microfacet NDFs and the shading process of rasterization, let's take a look at a more global picture.

# Shading Meets Light Transport

- Direct lighting is a path with three vertices: $e$(ye), $x$, $l$(ight)
- Use machinery from half-vector space light transport
  - Manifold exploration *[Jakob12]*, glossy paths *[Hanika15]*
- When $x$ changes, how does a half-vector $h$ change?
  - Use first-order 2x2 derivative matrix $M$ of $h$ w.r.t. $x$

$$M = \frac{\mathrm{d}h}{\mathrm{d}x} = \begin{pmatrix} \frac{\mathrm{d}h_s}{\mathrm{d}x_s} & \frac{\mathrm{d}h_s}{\mathrm{d}x_t} \\ \frac{\mathrm{d}h_t}{\mathrm{d}x_s} & \frac{\mathrm{d}h_t}{\mathrm{d}x_t} \end{pmatrix}$$

  - Defined in local tangent frame $T$ of the surface






We're performing a direct lighting. From the light transport stand point, we have a path with three vertices: e, x, l.

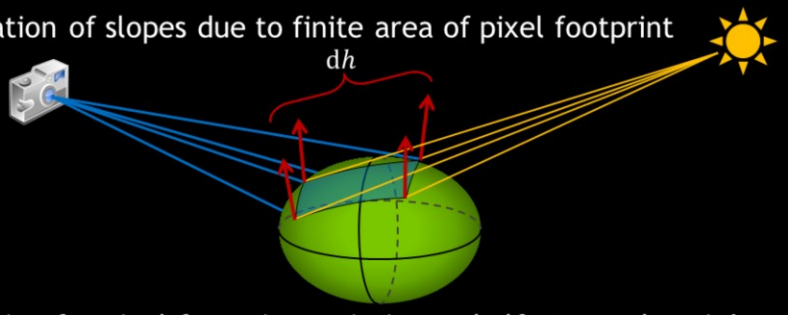
Wenzel Jakob presented derivatives of the half vector h (at vertex x) with respect to all three vertices in his work on manifold exploration.

We can use these derivatives (modified by [Hanika et al. 2015] for parallel plane domain) to convert a change at any of the vertices into the first-order-precise change of the half vector!

We are particularly interested in the derivative matrix M (Jacobian) of dh/dx. This matrix differentiates the expression for half vector h described previously. Note that the half vector is defined in the local shading tangent frame, so we need to compute the derivative of the tangent frame with respect to x as well. This incorporates the curvature of the shading surface into the derivatives matrix M.

# Pixel Footprint in Half-vector Domain

- Obtain variation of slopes due to finite area of pixel footprint




- $M$ is a mapping for pixel footprint variation to half-vector domain!
  - First transform ray differentials into $\Delta x$ on surface
  - Then multiply by matrix $M$ to get $\Delta h$ in slope domain

$$\Delta h = M \Delta x$$



Once we obtained the derivative matrix M, we can project the pixel footprint from the camera onto the tangent plane around our shading point x.

We usually project only two vectors of this projected footprint from the camera to the surface.

This parallelogram formed by such vectors then defines the change $\Delta x$ of the shading point x when we change the sample position within the pixel.

By multiplying with the derivatives matrix M, we can convert the variation of the shading point x that is induced by the pixel area into the variation $\Delta h$ of the half vector h in the parallel plane domain. This is then again a parallelogram that describes the first-order change of the half vector induced by the pixel footprint.

# Computing Derivatives on GPU

- Benefit from quad shading on GPU!
  - Use dFdx, dFdy to obtain the final value with finite differencing:

```
void main()
{
...
    // Compute plane-plane half vector (hpp)
    vec3 hppWS = hWS / dot(hWS, shadingFrame.n);
    vec2 hpp = vec2(dot(hppWS, shadingFrame.s), dot(hppWS, shadingFrame.t));
    vec3 h = normalize(vec3(hpp, 1.f));

    // Use ddx/ddy, thanks to quad shading!
    mat2 dhduv = mat2(dFdx(hpp), dFdy(hpp));
```

- Matrix $U = \mathrm{d}h/\mathrm{d}uv = M\Delta x$ is first-order change of $h$ induced by pixel footprint
  - Implicitly accounts for surface curvature
- Let's filter some NDFs with this *parallelogram!*



Computing the derivatives matrix M is a tedious and computationally expensive procedure, which may be not suitable for real-time budgets even on high-end GPUs.

Luckily, each shading point is guaranteed to be shaded at least within a quad of 2x2 samples within the same triangle on the GPU.

This makes the quad-shading pipeline a perfect candidate for computing the derivative matrix using finite differencing with ddx/ddy.

Moreover, when taking the derivatives, they are taken in screen space and already encode the change corresponding to a one pixel offset.

So, we can easily compute the matrix U, which encodes both the derivatives matrix M and the multiplication by the variation $\Delta x$ of shading position within a pixel.

This matrix is easily computed by just taking the derivatives of the half vector in parallel plane domain (hpp) with respect to the change of the sample position in screen space. This matrix U then defines two vectors of a parallelogram in half vector domain. In order to take into account all possible half vectors within the pixel footprint during shading, we filter the NDF around the region defined by the parallelogram $\Delta h$ (matrix U) in the slope domain. This allows us to approximate shading across the whole pixel footprint with one shading sample!

# Filtering Beckmann

- Convert a parallelogram into a covariance matrix
  - Use quadratic form $C = UU^T$
  - Defines another 2D Gaussian $N$ in half-vector domain!
  - Assumes pixel filter is a Gaussian
- Filtering is a convolution of two Gaussians: $N(footprint) * D(h)$
  - Convolution of two 2D Gaussians is another one with covariance matrix

$$R' = R + 2C, \text{ where } R = \begin{pmatrix} \alpha_s^2 & 0 \\ 0 & \alpha_t^2 \end{pmatrix}$$

  - Then treat as a regular Beckmann with *full roughness matrix* $R'$



Now that we have a parallelogram filter for NDFs, let's take a look at the concrete examples.

First we will filter the Beckmann NDF with this parallelogram before doing shading.

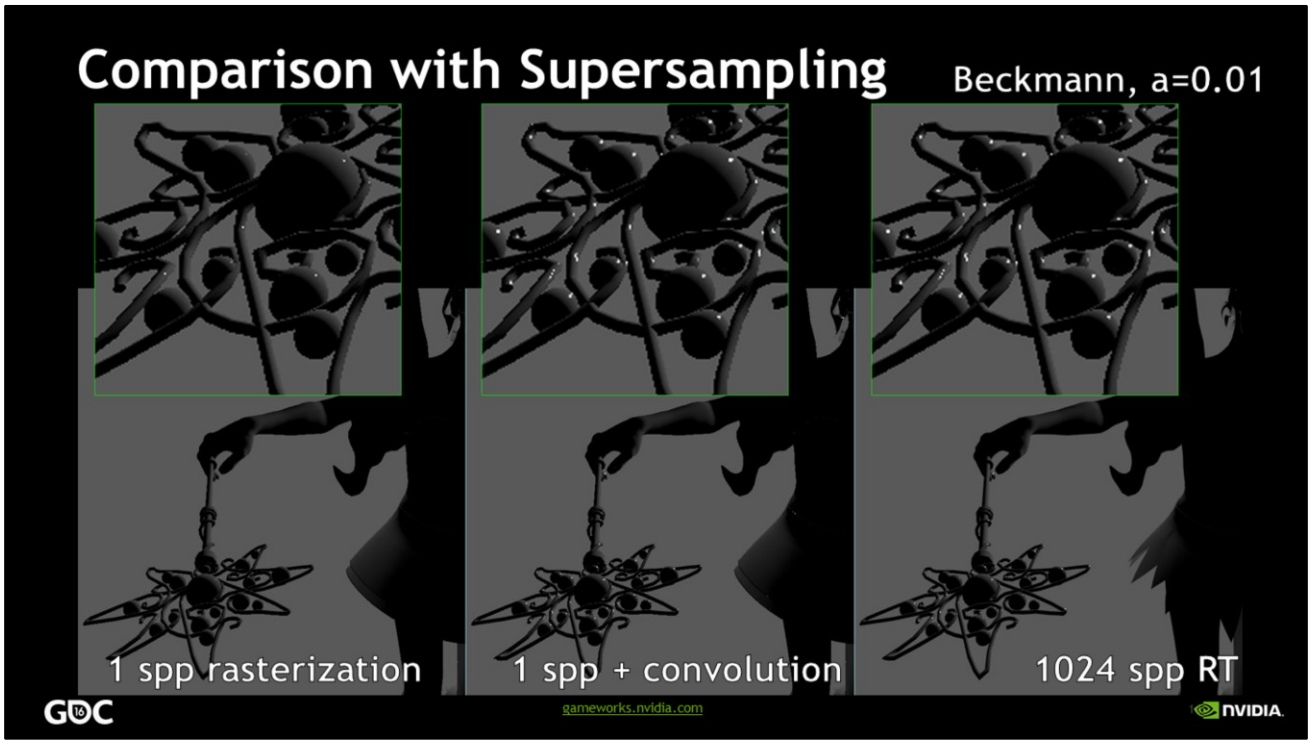
For that we interpret the vectors of a parallelogram as standard deviations of a 2D Gaussian. This means that we assume a Gaussian pixel filter with a standard deviation of 0.5 pixel wide, which is a reasonable reconstruction filter.

After propagating both standard deviations of such Gaussian all the way from screen space to the slope domain, we have a matrix U that contains the propagated vectors of two standard deviations. We use a simple squaring in order to create a covariance matrix C out of matrix U.

Then we have a Beckmann NDF, which is a 2D Gaussian in slope domain, and another 2D Gaussian for filtering that is induced by the pixel. The convolution of two 2D Gaussians is another Gaussian, whose covariance matrix is simply a sum of the two source covariance matrices.

Note the factor '2' in front of the matrix C. It comes from the fact that we work with roughness, which is a scaled standard deviation.

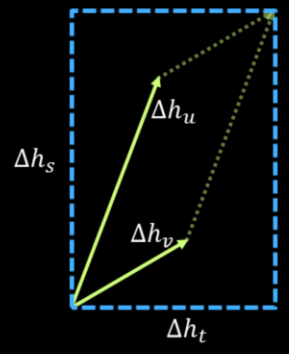
Then we need to evaluate Beckmann NDF with the resulting full roughness matrix R'.




Here are the results. Our method (in the middle) manages to find all highlights present in the reference using just one shading sample per pixel!

# More Temporal Stability

- Use a bounding axis-aligned rectangle around of parallelogram
  - Aligned along $s$ and $t$ axes of the slope domain
- Overfilters the NDF
- Much better temporal stability
- Beckmann distribution filtering with a-a rect:




```
void shade()
{
    ...
    // Compute pixel footprint's axis-aligned rectangle
    vec2 fp = vec2(abs(dFdxFine(hpp)) + abs(dFdyFine(hpp))) * 0.5f;
    fp = min(vec2(fp_max), fp);

    // Convolve with pixel footprint diagonal covariance matrix
    rghns_sq += fp*fp * 2.f;    // x2 because roughness = sqrt(2) * pixel_sigma_hpp

    value = evalBeckmannDistribution(h, rghns_sq);
```

- Clamping value fp_max is usually in a range 0.1…1

GDC gameworks.nvidia.com NVIDIA

For production assets having a tight parallelogram might be not sufficient for temporal stability due to many reasons (many small quadrics, non-manifold meshes, imprecise or averaged shading normals).

With NDF filtering we have a standard noise (aliasing) vs. bias approach.

Similarly to h/w texture filtering, we can prefer to have more bias for better temporal stability.

One option is to filter over an axis-aligned bounding rectangle region around the parallelogram. This way, we don't rely on the tightly estimated region, but rather allow some error and noise for our filter.

Obviously, better stability with this filter comes at a price of having a few false-positive specular highlights.

The code snippet for filtering with a rectangle is using a regular anisotropic Beckmann shading.

Pixel footprint can also get very large in slope domain, especially at grazing angles and on small details with extremely high curvature.

In order to avoid false highlights due to the overestimated filter size, it is important to clamp the filter region from above.

The value of fp_max is in roughness units and we usually use a value of 1.0 as a limiting factor.

# Filtering GGX NDF

- With GGX there is no closed-form convolution
- Use a bounding axis-aligned rectangle $(\Delta h_s, \Delta h_t)$
- Compute a rectangular 2D integral over this area

$$E_{\Delta h}[D(h)] = \frac{1}{|\Delta h|} \int_{h_s-\Delta h_s/2}^{h_s+\Delta h_s/2} \int_{h_t-\Delta h_t/2}^{h_t+\Delta h_t/2} D(s,t)\mathrm{d}s\mathrm{d}t$$

- Assumes box pixel filter
- Slightly overfiltering due to axis-aligned rect



Similarly, we can filter a GGX NDF. Instead of filtering with a Gaussian kernel, this time we just filter with a constant flat kernel.

This leads to a simple axis-aligned 2D integral over the GGX NDF. This filter kernel assumes a box pixel filter, which is the case for rasterization.

Note that the integral is taken in the slope domain. In order to come back to the solid angle domain of NDF, we need to also multiply by the corresponding transformation Jacobian afterwards.

# Convolving with GGX

- Rectangular integral over the pixel footprint in slope domain:

```
float GGXRectIntegral(in vec2 Hpp, in vec3 H, in vec2 roughness, in vec2 footprint)
{
  float a = roughness.x, b = roughness.y;
  float a_2 = a*a, b_2 = b*b;
  float x1 = Hpp.x-footprint.x, x2 = Hpp.x+footprint.x;
  float y1 = Hpp.y-footprint.y, y2 = Hpp.y+footprint.y;
  float x1_2 = x1*x1, x2_2 = x2*x2;
  float y1_2 = y1*y1, y2_2 = y2*y2;

  float val ((x1*(atan((y1*a)/(sqrt(x1_2 + a_2)*b)) -
       atan((y2*a)/(sqrt(x1_2 + a_2)*b))))/sqrt(x1_2 + a_2)
       + (x2*(-atan((y1*a)/(sqrt(x2_2 + a_2)*b)) +
       atan((y2*a)/(sqrt(x2_2 + a_2)*b))))/sqrt(x2_2 + a_2)
        + (y1*sqrt(y2_2 + b_2)*(atan((x1*b)/(a*sqrt(y1_2 + b_2)))
        - atan((x2*b)/(a*sqrt(y1_2 + b_2))))
        + y2*sqrt(y1_2 + b_2)*(-atan((x1*b)/(a*sqrt(y2_2 + b_2))) +
       atan((x2*b)/(a*sqrt(y2_2 + b_2)))))/(sqrt((y1_2 + b_2)*(y2_2 + b_2))))
      /((2*M_PIf)*(x1 - x2)*(y1 - y2));
  return val / max(1e-3f, H.z*H.z*H.z*H.z); // Jacobian hpp->h
}

void shade()
{
...
     vec2 fp = vec2(abs(dFdxFine(hpp)) + abs(dFdyFine(hpp))) * 0.5f;
     vec2 fp_min = min(rghns, vec2(1e-1f));
     fp = clamp(fp, fp_min, vec2(fp_max));
     value = GGXRectIntegral(hpp, h, rghns, fp);
```

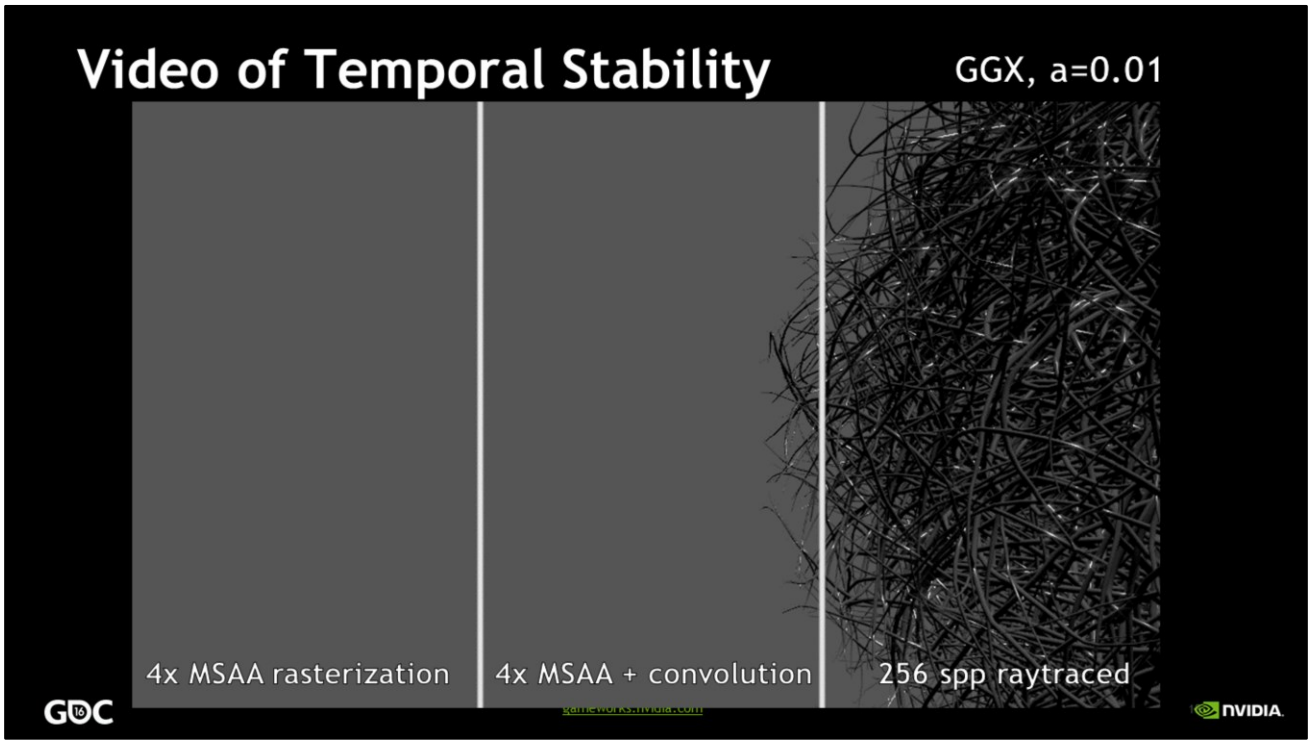
Optimize me!

GDC 16 gameworks.nvidia.com NVIDIA

Here is the closed form of the 2D rectangular integral over the GGX NDF in slope domain. It takes half-sides of a rectangle as two scalars in the ‘footprint’ argument.

The equation is relatively bulky, contains eight square roots and eight arctangents, however, it can be significantly optimized with approximations if necessary.

It is also important to clamp the footprint from below, otherwise the integrated equation can be numerically unstable.




Here are the video results of filtering a GGX NDF with isotropic roughness a=0.01 on the hairball model.

You can see how the subpixel-sized highlights are accurately reconstructed using NDF filtering with one to four samples per pixel (middle).

We deliberately apply star-shaped posteffect to emphasize the temporal distribution of the HDR intensity of the highlight as well.


Here is a more detailed scene, this time it is a Beckmann NDF with roughness a=0.01 and no filtering.

Again, a posteffect is used to convey the HDR flickering of the highlights.


The same sequence with rectangular NDF filtering.

# Conclusion

- Addresses only *shading* aliasing
  - No improvements for geometric aliasing
  - Can still alias with high-frequency bumpy geometry
- Extracts full highlight on anisotropic quadric
- Average highlight energy across the footprint
- Relies on *properly modeled* shading normals



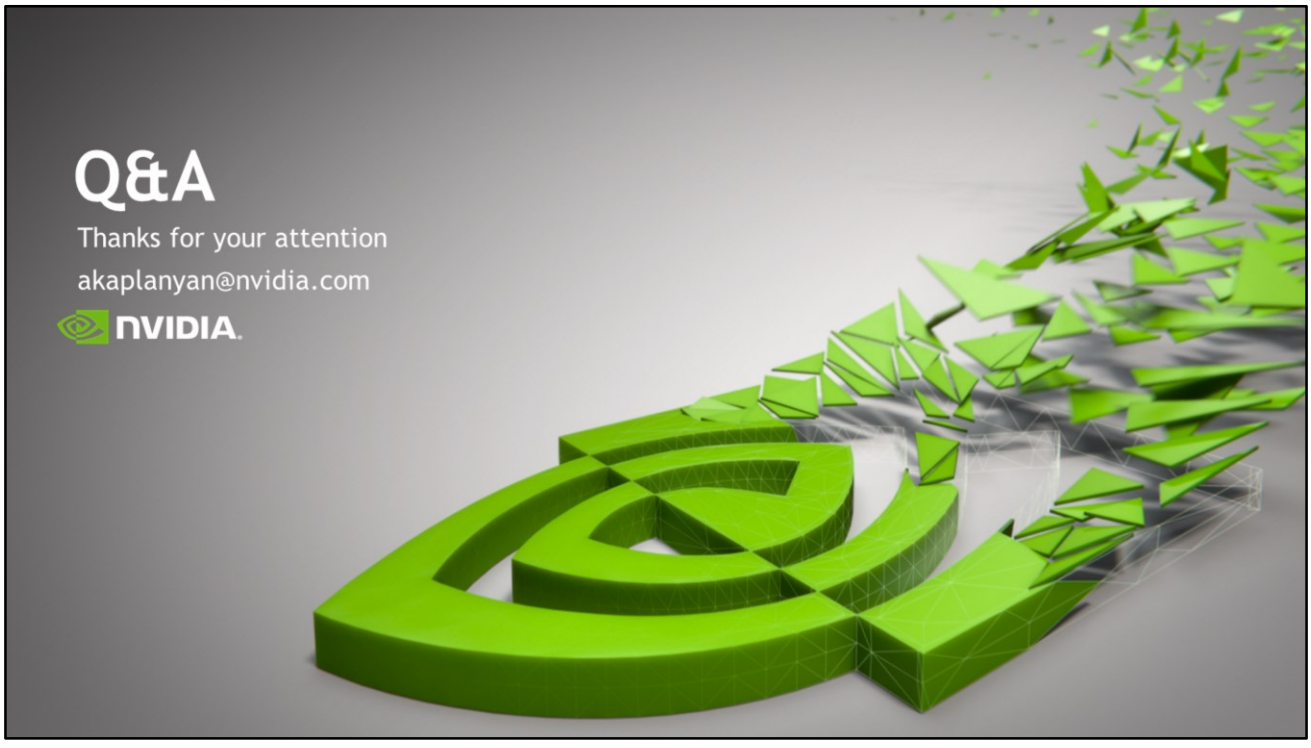
We address only aliasing that comes from specular shading.

Other sources of aliasing, such as visibility, are not covered, therefore, no improvements should be expected on geometric aliasing.

We also take into account only a single quadric when filtering an NDF. Thus, on a geometry with high-frequency details quadrics can change rapidly.

We can integrate over a first-order-precise pixel footprint in slope space, thus the filtering is good at extracting small highlights and, more importantly, their average intensity across the pixel.

Since the filtering method works on shading quadrics, it is crucial to have accurate shading normals on geometry, which define the virtual shading surface used for NDF filtering.


Q&A
Thanks for your attention
akaplanyan@nvidia.com
NVIDIA

# References


- [Toksvig04] M. Toksvig, "*Mipmapping Normal Maps*", tech. report, Nvidia, 2004
- [Hill12] S. Hill and D. Baker, "*Rock-Solid Shading: Image Stability Without Sacrificing Detail*", SIGGRAPH Advances in Real-time Rendering in Games Course, 2012
- [Jakob12] W. Jakob and S. Marschner, "*Manifold exploration: a Markov chain Monte Carlo technique for rendering scenes with difficult specular transport*", SIGGRAPH 2012
- [Vlachos15] A. Vlachos, "*Advanced VR Rendering*", Talk, GDC, 2015
- [Hanika15] J. Hanika, A. Kaplanyan, and C. Dachsbacher, "*Improved half vector space light transport*", EGSR, 2015