Supraleiter

and its rendering technologies

Who are we?

- BeRo
  - Benjamin Rosseaux
  - Farbrausch
- urs
  - Urs Ganse
  - Mercury

What is Supraleiter?

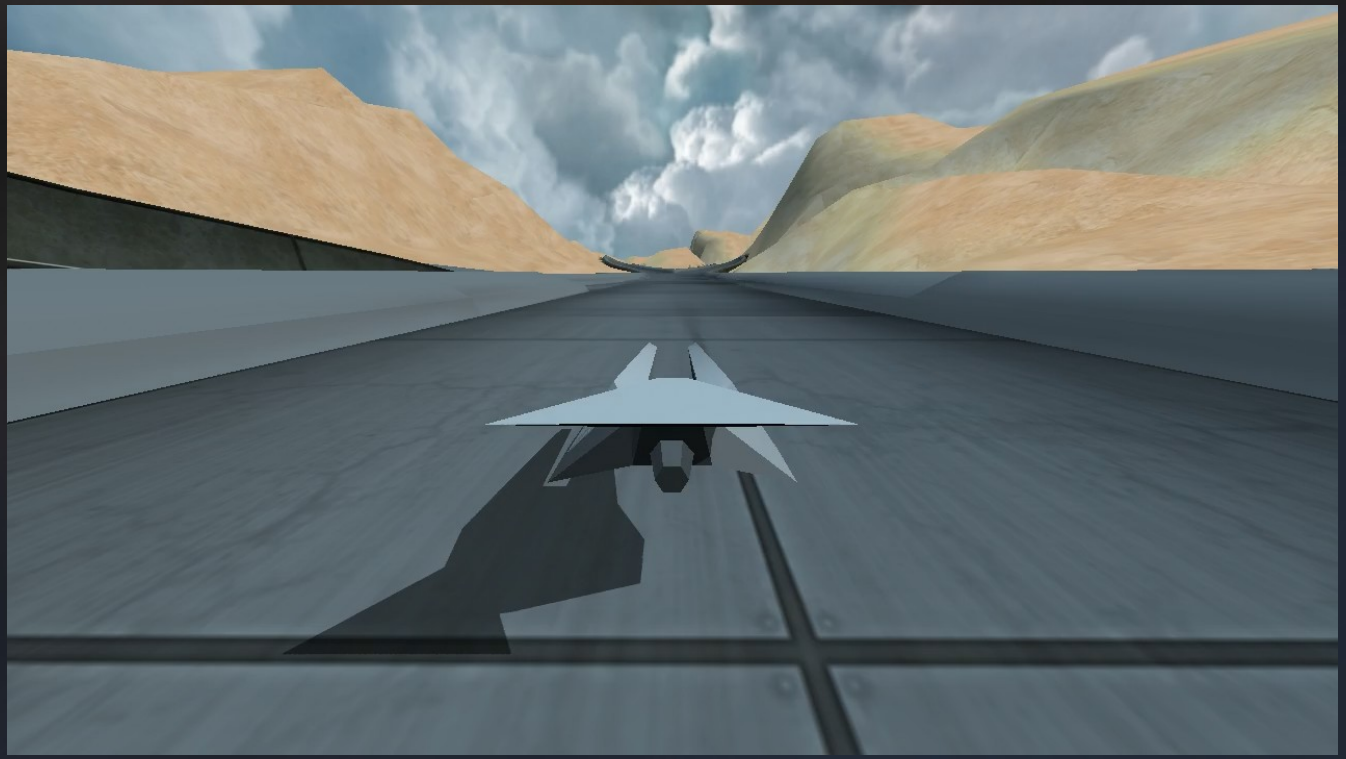
- a futuristic anti-gravity game with its own game engine with
  - 2D graphics engine
    - 2D sprite batch engine, on-the-fly optimized texture atlas system (=> no need for a external texture atlas packer tool), own truetype font engine (with font hinting bytecode support), ...
  - 3D graphics engine (a 3D clustered forward renderer, the 3D extended form of tiled forward rendering a ka Forward+)
    - HDR, motion blur, depth of field, bloom, dynamic lights, realtime global illumination, physically based rendering, CPU-and-GPU-multithreaded with two OpenGL 4.3 contexts with shared resources (except for FBOs, VAOs and other non-shareable “state” objects), on-the-GPU raytracing support, Sample Distribution Shadow Maps, optimized gaussian filtered PCF, optional realtime raytraced hard shadows, volumetric scattering, procedural sky, many lights clustered forward rendering (3D cluster variant of 2D tiled forward rendering a ka Forward+), ...
  - 3D audio engine
    - HRTF-based 3D positional audio stereo sound, CPU-multithreaded, doppler effect, ...
  - 3D physics engine
    - CPU-multithreaded per islands, collision detection with the Minkowski Portal Refinement algorithm for convex shapes, BVH optimized triangle-mesh-based collision detection for non-convex shapes, simple but almost robust sequential impulse solver, self balanced dynamic AABB tree support, all-axes-sweep-and-prune support, joints, constraints, …

And what is the topic of this talk?

The rendering technlogies inside supraleiter
and its development story
(and some of the 64k from yesterday)

The beginning...

The beginning...



The beginning...


Oh! It looks so 90ish!

Well, but how can we improve it?

The first problem zone is the old Blinn-Phong surface shading itself. So replace it with physically based shading.

$$H = \frac{L+V}{|L+V|}$$

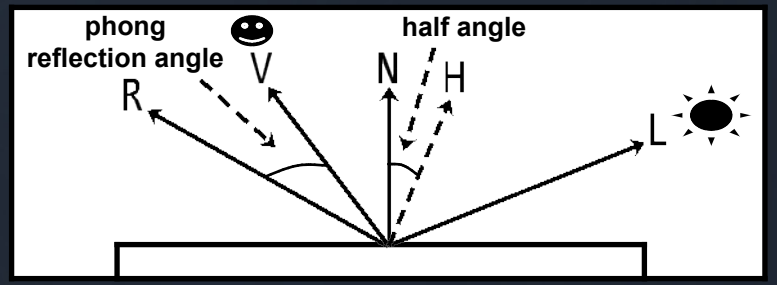
$$\cos\theta' = N \cdot H$$

$$I_{specular} = I_{in} \cdot k_{specular} \cdot \cos^{n}\theta'$$

$$I_{specular} = I_{in} \cdot k_{specular} \cdot \left(\frac{(L+V) \cdot N}{|(L+V)| \cdot |N|} \right)^{n}$$

$k_{specular}$ = empirically determined reflection coefficient for specular component of reflection
I_{in} = intensitiy of the incident light beam from the point light source
$I_{specular}$ = specular illumination of each surface point
n (as exponent) = constant exponent describing the surface characteristics

N = normal
L = light direction
V = view direction
R = reflection vector
H = half vector
(half vector is normalize(viewDirection + lightDirection))



Physically based shading

Supraleiter uses the same PBR model as the guys from Epic, based on their „Real Shading in Unreal Engine 4“ slides and course notes*.

* http://blog.selfshadow.com/publications/s2013-shading-course/

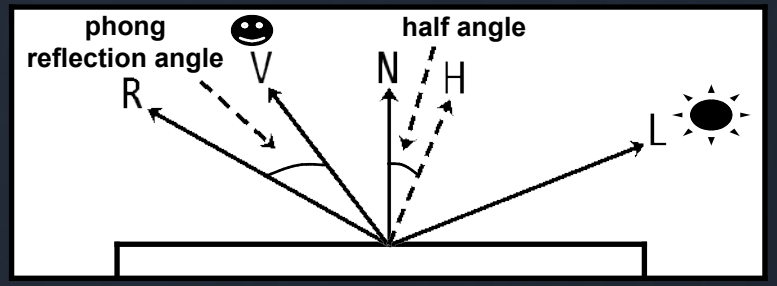
Material parameters

- It’s primarly texture based, so the artist can do texel-exact quasi-material-layering, but each input texture can be also animated
  - Albedo color map (NOT DIFFUSE)
    - Base RGB color with alpha channel
  - Emissive color map
    - Emissive RGB color
  - Specular color map
    - Specular RGB color
  - Normal map
    - RGB = XYZ normal
    - Alpha = Height (ex. needed for parallax occlusion mapping etc.)
  - Material property map
    - Metallic (red color channel)
      - How much metallic
    - Roughness (green color channel)
      - How much roughness
    - Cavity (blue color channel)
      - Used for small scale shadowing
    - Reflectivity (alpha channel)
      - How much image-based-lighting reflectivity

Specular BRDF

$$f(l,v) = \frac{D(h)F(l,h)G(l,v,h)}{4(n \cdot l)(n \cdot v)}$$

n = normal
l = light direction
v = view direction
r = reflection vector
h = half vector
(half vector is normalize(viewDirection + lightDirection))



Specular distribution

Trowbridge-Reitz (GGX)

$$f(l,v) = \frac{D(h)F(l,h)G(l,v,h)}{4(n \cdot l)(n \cdot v)}$$

GGX/Trowbridge-Reitz

[Walter et al. 2007, "Microfacet models for refraction through rough surfaces"]

Specular distribution

Blinn-Phong

GGX

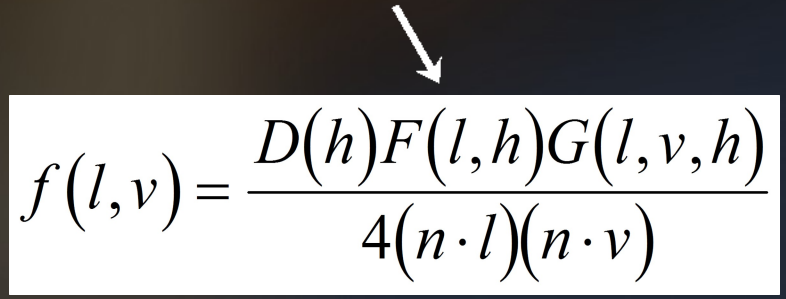
GGX looks more natural with his longer tail

Specular distribution

```
// Specular D normal distribution function (NDF) - GGX/Trowbridge-Reitz
// [Walter et al. 2007, "Microfacet models for refraction through rough surfaces"]
float specularD(const in float roughness, const in float nDotH){
  float a = roughness * roughness;
  float a2 = a * a;
  // optimized from: float d = ((nDotH * nDotH) * (a2 - 1.0)) + 1.0;
  float d = (((nDotH * a2) - nDotH) * nDotH) + 1.0;
  return a2 / (3.14159265358979323846 * (d * d));
}
```

Fresnel

Schlick



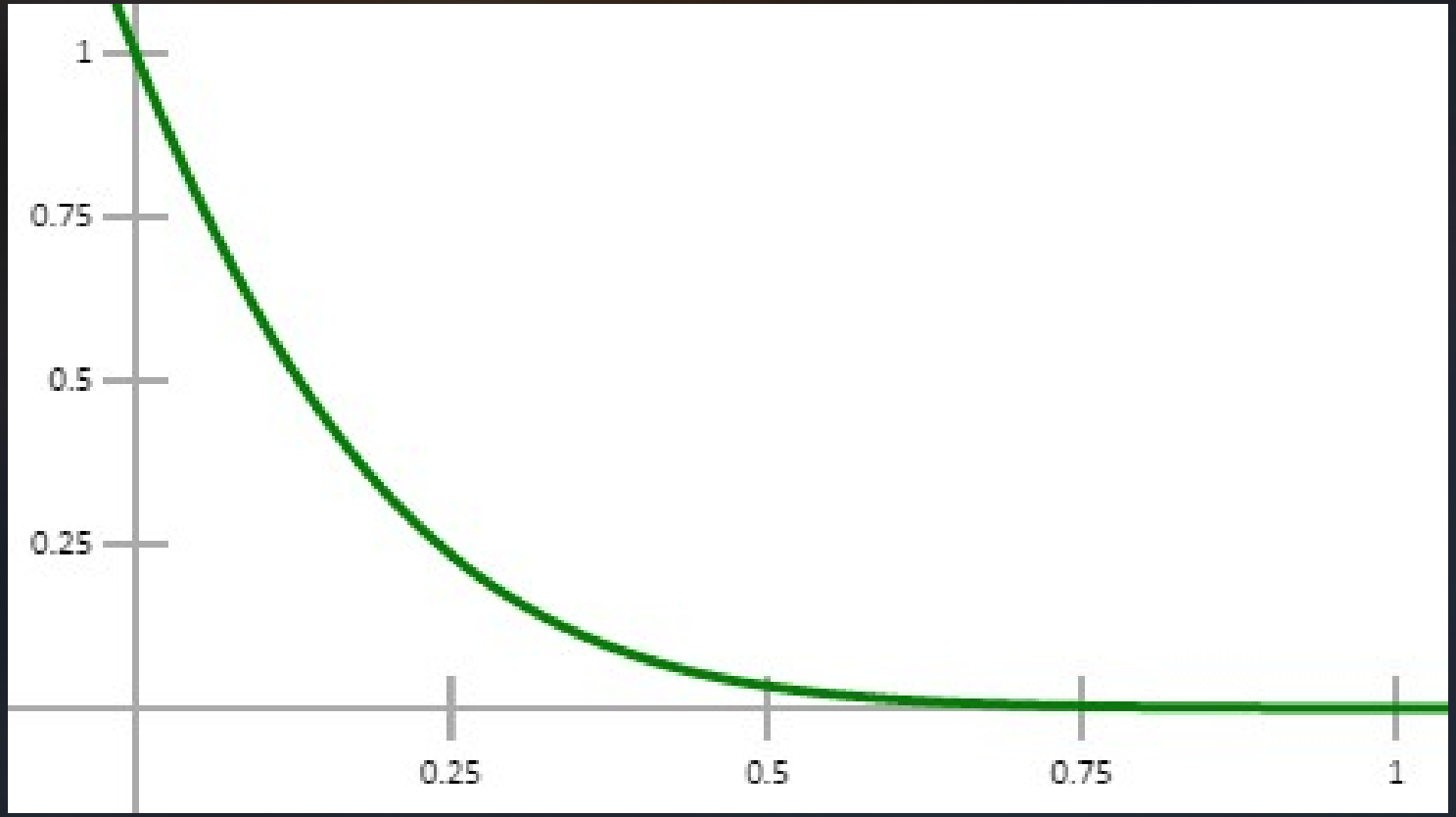
$$f(l,v) = \frac{D(h)F(l,h)G(l,v,h)}{4(n \cdot l)(n \cdot v)}$$

Schlick approximation

[Schlick 1994, "An Inexpensive BRDF Model for Physically-Based Rendering"]

[Lagarde 2012, "Spherical Gaussian approximation for Blinn-Phong, Phong and Fresnel"]

Fresnel



Fresnel

```
// Specular F fresnel - Schlick approximation
// [Schlick 1994, "An Inexpensive BRDF Model for Physically-Based Rendering"]
// [Lagarde 2012, "Spherical Gaussian approximation for Blinn-Phong, Phong and Fresnel"]
vec3 specularF(const in vec3 color, const in float vDotH){
  float fc = exp2((((-5.55473) * vDotH) - 6.98316) * vDotH);
  return vec3(fc) + ((1.0 - fc) * color);
}
```

Geometric shadowing

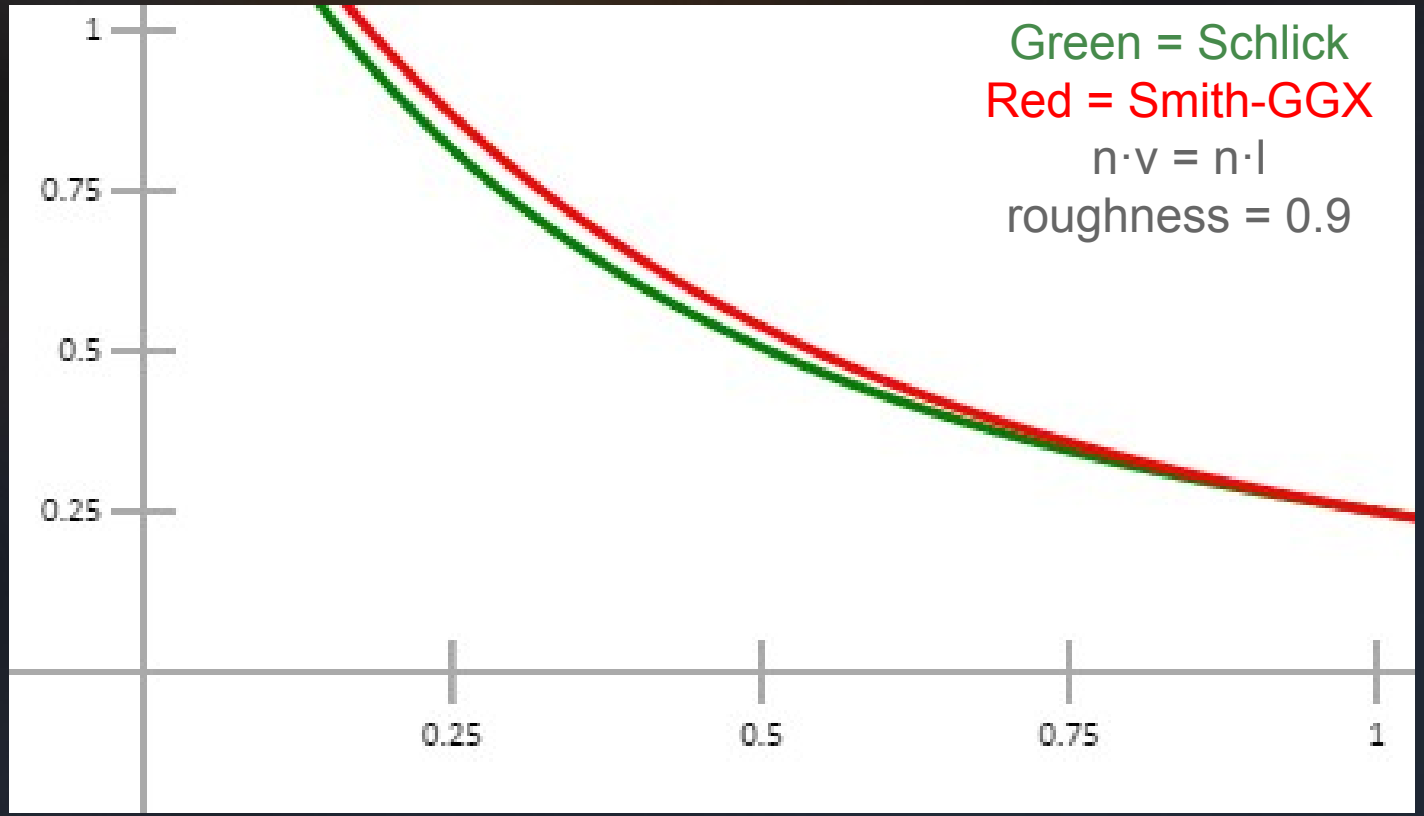
Schlick

$$f(l,v) = \frac{D(h)F(l,h)G(l,v,h)}{4(n \cdot l)(n \cdot v)}$$

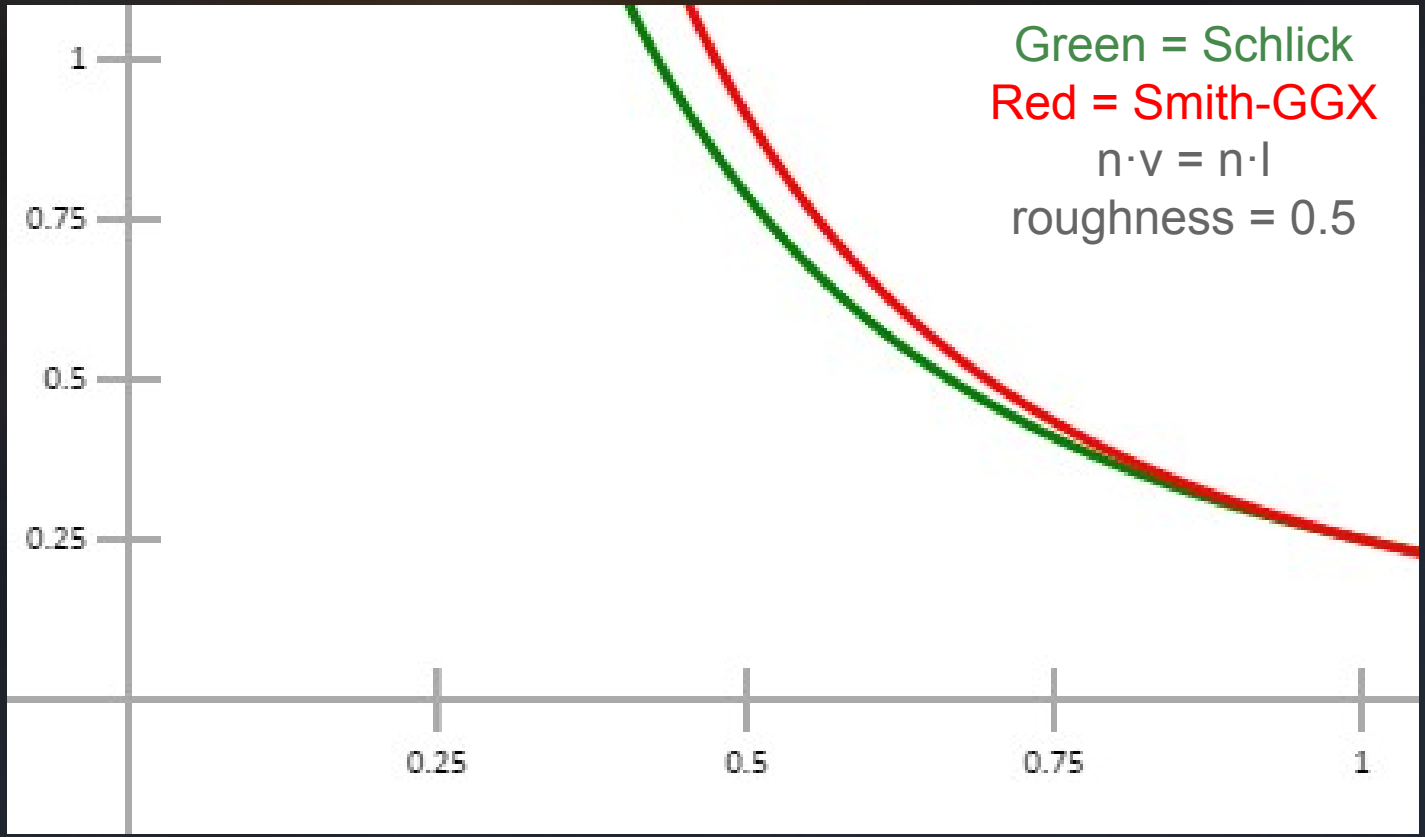
it is cheaper than Smith-GGX, but the difference is minor.

[Schlick 1994, "An Inexpensive BRDF Model for Physically-Based Rendering"]
[Smith 1967, "Geometrical shadowing of a random rough surface"]

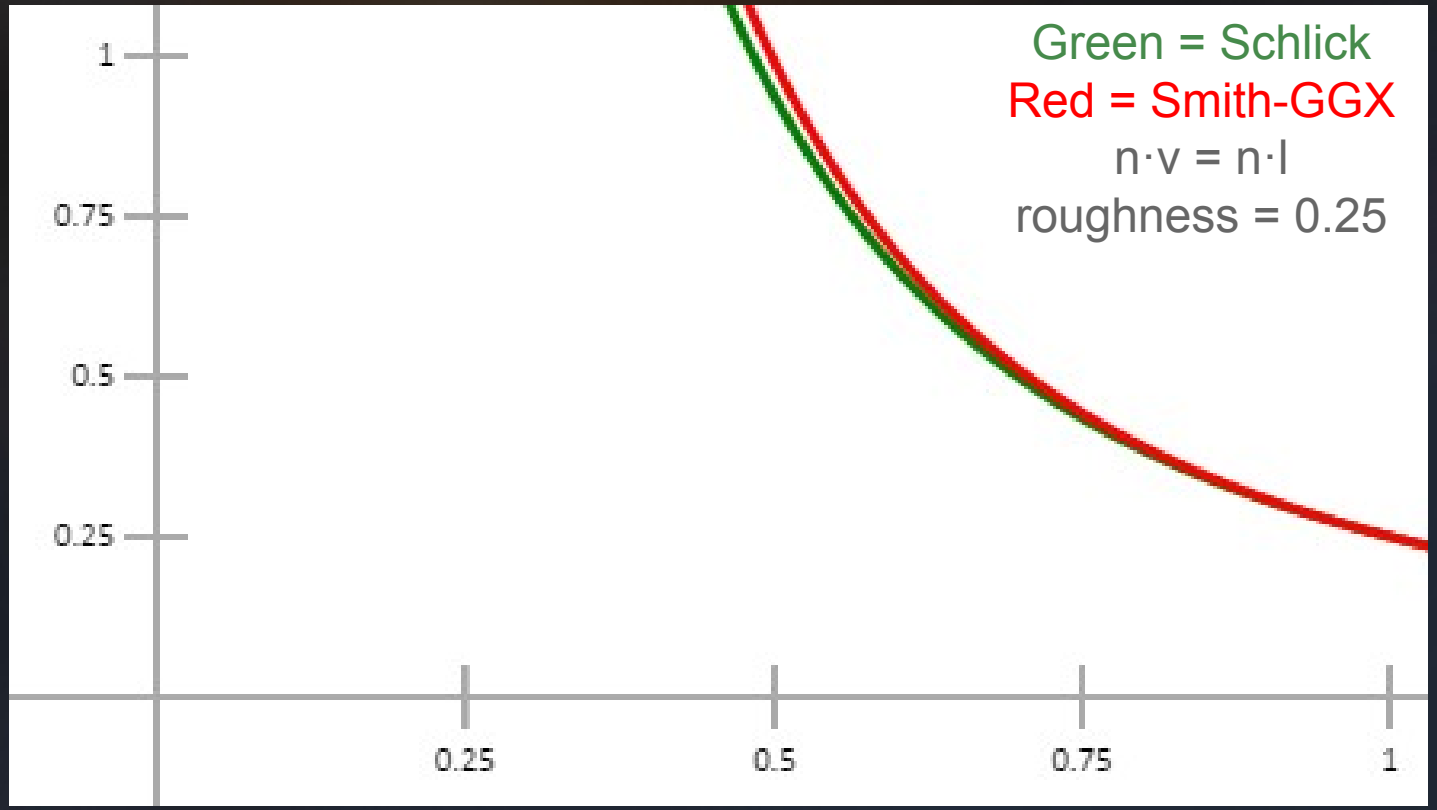
Geometric shadowing



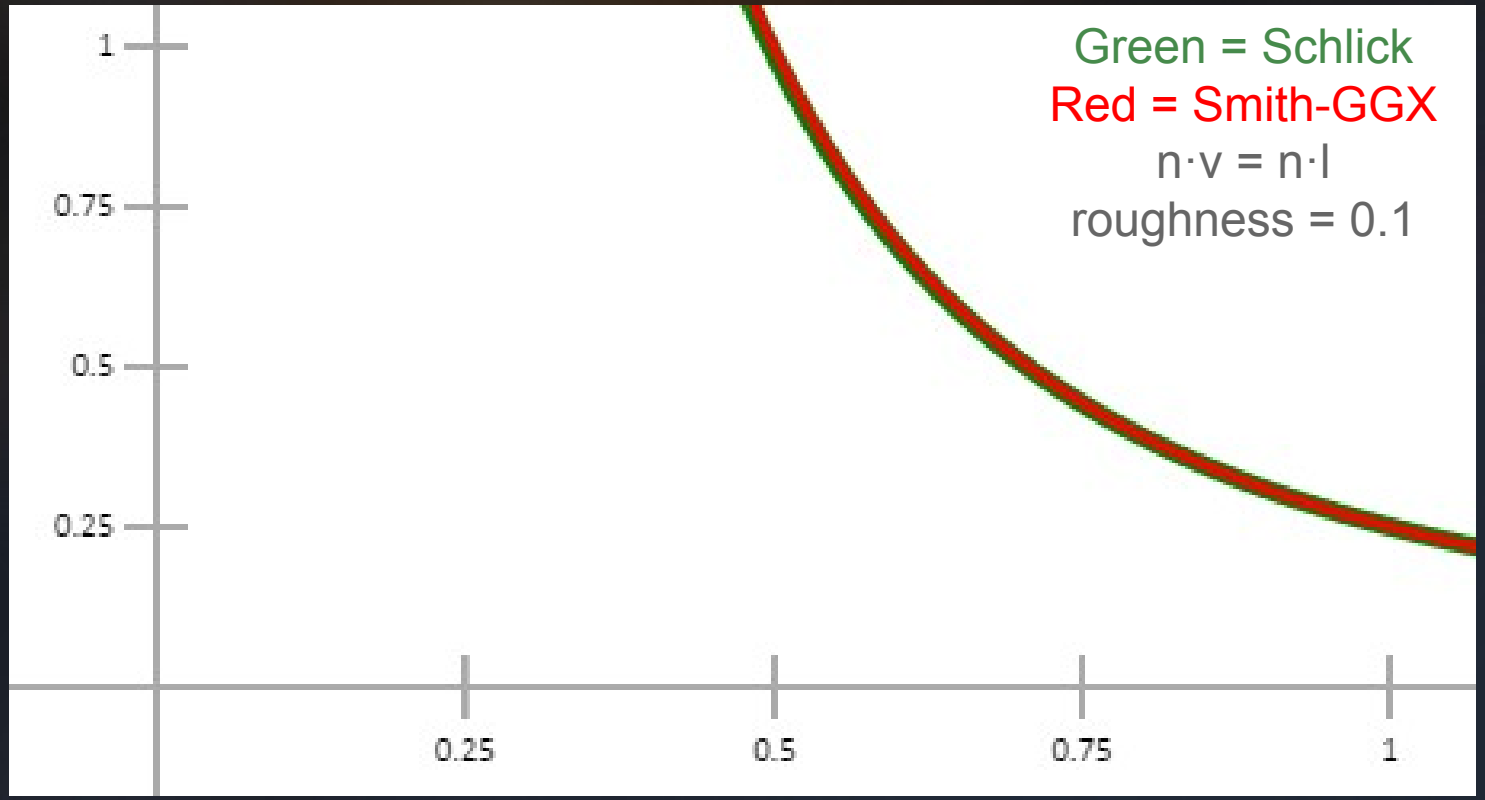
Geometric shadowing



Geometric shadowing



Geometric shadowing



Geometric shadowing

```
// Specular G Smith-GGX
// [Smith 1967, "Geometrical shadowing of a random rough surface"]
float specularG(const in float roughness, const in float nDotV, const in float nDotL){
  float a = roughness * roughness;
  float a2 = a * a;
  vec2 nDotVL = vec2(nDotV, nDotL);
  vec2 GVL = nDotVL + sqrt(((nDotVL - (nDotVL * a2)) * nDotVL) + a2);
  return 1.0 / (GVL.x * GVL.y);
}
```

```
// Specular G - Schlick
// [Schlick 1994, "An Inexpensive BRDF Model for Physically-Based Rendering"]
float specularG(const in float roughness, const in float nDotV, const in float nDotL){
  float k = (roughness * roughness) * 0.5;
  vec2 GVL = (vec2(nDotV, nDotL) * (1.0 - k)) + vec2(k);
  return 0.25 / (GVL.x * GVL.y);
}
```

Example usage

```
vec3 diffuseLambert(vec3 color){
  return color / 3.14159265358979323846;
}

vec3 doSingleLight(const in vec3 lightColor, const in vec3 lightLit,
                   const in vec3 lightDirection, const in vec3 normal,
                   const in vec3 diffuseColor, const in vec3 specularColor){
  // http://graphicrants.blogspot.de/2013/08/specular-brdf-reference.html
  // http://blog.selfshadow.com/publications/s2013-shadingcourse/karis/s2013_pbs_epic_notes_v2.pdf
  vec3 halfVector = normalize(gViewDirection + lightDirection);
  float nDotL = clamp(dot(normal, lightDirection), 1e-5, 1.0);
  float nDotV = clamp(dot(normal, gViewDirection), 1e-5, 1.0);
  float nDotH = clamp(dot(normal, halfVector), 0.0, 1.0);
  float vDotH = clamp(dot(gViewDirection, halfVector), 0.0, 1.0);
  vec3 diffuse = diffuseLambert(diffuseColor);
  vec3 specular = specularF(specularColor, vDotH) * specularD(gMaterialRoughness, nDotH) *
                  specularG(gMaterialRoughness, nDotV, nDotL);
  return (diffuse + specular) * ((gMaterialCavity * nDotL * lightColor) * lightLit);
}

vec3 diffuseColor = baseColor * (1.0 - gMaterialMetallic);
vec3 specularColor = mix(0.08 * specularMapColor, baseColor, gMaterialMetallic);
```

Example usage

```
vec3 diffuseLambert(vec3 color){
  return color / 3.14159265358979323846;
}

vec3 doSingleLightWithExtraDiffuseSpecularDirections(const in vec3 lightColor,
                                                     const in vec3 lightLit,
                                                     const in vec3 lightDiffuseDirection,
                                                     const in vec3 lightSpecularDirection,
                                                     const in vec3 normal,
                                                     const in vec3 diffuseColor,
                                                     const in vec3 specularColor){
  // http://graphicrants.blogspot.de/2013/08/specular-brdf-reference.html
  // http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf
  vec3 halfVector = normalize(gViewDirection + lightSpecularDirection);
  float nDotL = clamp(dot(normal, lightSpecularDirection), 1e-5, 1.0);
  float nDotV = clamp(dot(normal, gViewDirection), 1e-5, 1.0);
  float nDotH = clamp(dot(normal, halfVector), 0.0, 1.0);
  float vDotH = clamp(dot(gViewDirection, halfVector), 0.0, 1.0);
  vec3 diffuse = diffuseLambert(diffuseColor);
  vec3 specular = specularF(specularColor, vDotH) * specularD(gMaterialRoughness, nDotH) *
                  specularG(gMaterialRoughness, nDotV, nDotL);
  return (diffuse + specular) * ((gMaterialCavity *
         clamp(dot(normal, lightDiffuseDirection), 0.0, 1.0) * lightColor) * lightLit);
}

vec3 diffuseColor = baseColor * (1.0 - gMaterialMetallic);
vec3 specularColor = mix(0.08 * specularMapColor, baseColor, gMaterialMetallic);
```

Image based lighting

how the Supraleiter engine does it

Image based lighting

Supraleiter uses the same IBL model as the guys from Epic, based on their „Real Shading in Unreal Engine 4“ slides and course notes*, but just with a small
difference:

Supraleiter is using cube maps but also equirectanglar environment maps.

And the 64k from yesterday is using only equirectanglar environment maps, since these are easier to handle in my .marchfabrik 64k intro engine.

* http://blog.selfshadow.com/publications/s2013-shading-course/

Image based lighting

The first thing you must to do is to generate a BRDF integration 2D lookup texture, where n = l is assumed as a fixed distribution for the simplicity, like this:

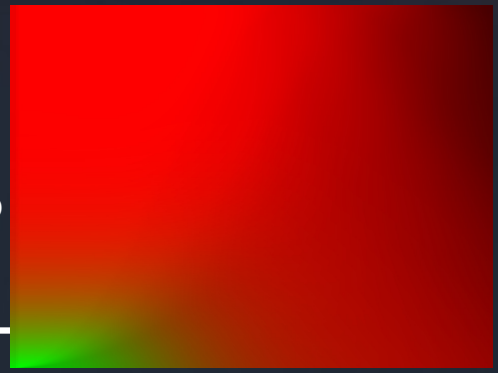
$$\frac{1}{N}\sum_{k=1}^{N}\frac{L_i(l_k)f(l_k,v)\cos\theta_{l_k}}{p(l_k,v)} = \left(\frac{1}{N}\sum_{k=1}^{N}L_i(l_k)\right)\left(\frac{1}{N}\sum_{k=1}^{N}\frac{f(l_k,v)\cos\theta_{l_k}}{p(l_k,v)}\right)$$

The second splitted part

$$\frac{1}{N}\sum_{k=1}^{N}\frac{f(l_k,v)\cos\theta_{l_k}}{p(l_k,v)} = LUT.r * F_0 + LUT.g$$

$$\int_H f(l,v)\cos\theta dl = \int_H \frac{f(l,v)}{F(v,h)}\left(1-(1-v\cdot h)^5\right)\cos\theta_l dl + \int_H \frac{f(l,v)}{F(v,h)}(1-v\cdot h)^5\cos\theta_l dl$$

LUT (RG16UI)



Roughness

$\cos\theta_v$

For details see:
http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_slides.pdf
http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf

BRDF LUT texture generation (1/3)

```
const int numSamples = 1024; // Play with this value

vec2 Hammersley(const in int index, const in int numSamples){
#ifdef RealOpenGL
#ifdef GL4_AND_UP
  uint reversedIndex = bitfieldReverse(uint(index));
#else
  uint reversedIndex = uint(index);
  reversedIndex = (reversedIndex << 16) | (reversedIndex >> 16);
  reversedIndex = ((reversedIndex & 0x00ff00ffu) << 8) | ((reversedIndex & 0xff00ff00u) >> 8);
  reversedIndex = ((reversedIndex & 0x0f0f0f0fu) << 4) | ((reversedIndex & 0xf0f0f0f0u) >> 4);
  reversedIndex = ((reversedIndex & 0x33333333u) << 2) | ((reversedIndex & 0xccccccccu) >> 2);
  reversedIndex = ((reversedIndex & 0x55555555u) << 1) | ((reversedIndex & 0xaaaaaaaau) >> 1);
#endif
  return vec2(fract(float(index) / numSamples), float(reversedIndex) * 2.3283064365386963e-10);
#else
  // Special fake Hammersley variant for WebGL and ESSL, since WebGL and ESSL do not support
  // uint and bit operations
  vec2 r = fract(vec2(float(index) * 5.3983, float(int(int(2147483647.0) - index)) * 5.4427));
  r += dot(r.yx, r.xy + vec2(21.5351, 14.3137));
  return fract(vec2(float(index) / float(numSamples), (r.x * r.y) * 95.4337));
#endif
}
```

BRDF LUT texture generation (2/3)

```
vec3 ImportanceSampleGGX(const in vec2 e, const in float roughness){
  float m = roughness * roughness;
  float m2 = m * m;
  float phi = (2.0 * 3.14159265358979323846264338327795) * e.x;
  float cosTheta = sqrt((1.0 - e.y) / (1.0 + ((m2 - 1.0) * e.y)));
  float sinTheta = sqrt(1.0 - (cosTheta * cosTheta));
  return vec3(sinTheta * cos(phi), sinTheta * sin(phi), cosTheta);
}

// Specular G - Smith [Smith 1967, "Geometrical shadowing of a random rough surface"]
float specularG(const in float roughness, const in float nDotV, const in float nDotL){
  float a = roughness * roughness;
  float a2 = a * a;
  vec2 GVL = vec2(nDotV, nDotL);
  GVL = GVL + sqrt((GVL * (GVL - (GVL * a2))) + vec2(a2));
  return 1.0 / (GVL.x * GVL.y);
}
```

BRDF LUT texture generation (3/3)

```
void main(){
  float roughness = fragCoord.x / iResolution.x;
  float nDotV = fragCoord.y / iResolution.y;
  vec3 V = vec3(sqrt(1.0 - (nDotV * nDotV)), 0.0, nDotV);
  vec2 r = vec2(0.0);
  for(int i = 0; i < numSamples; i++){
    vec3 H = ImportanceSampleGGX(Hammersley(i, numSamples), roughness);
    vec3 L = -reflect(V, H); //((2.0 * dot(V, H )) * H) - V;
    float nDotL = clamp(L.z, 0.0, 1.0);
    if(nDotL > 0.0){
      float vDotH = clamp(dot(V, H), 0.0, 1.0);
      r += (vec2(1.0, 0.0) + (vec2(-1.0, 1.0) * pow(1.0 - vDotH, 5.0))) *
           (nDotL * specularG2(roughness, nDotV, nDotL) *
           ((4.0 * vDotH) / clamp(H.z, 0.0, 1.0)));
    }
  }
  gl_FragColor = vec4(r / float(numSamples), 0.0, 1.0);
}
```

Image based lighting

The second thing you must to do is to rendering your probe cube maps or your one camera-based cube map and prefilter these cube maps to roughness mip-mapped cube maps or better due to the seamless filtering to equirectangular environment maps, which is at least better for WebGL and OpenGL ES 2.0, since these don’t support seamless cube map texture lookups a la glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS) .

For details see:
http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_slides.pdf
http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf

Image based lighting

Cube map prefiltering

$$\frac{1}{N}\sum_{k=1}^{N}\frac{L_i(l_k)f(l_k,v)\cos\theta_{l_k}}{p(l_k,v)} = \left(\frac{1}{N}\sum_{k=1}^{N}L_i(l_k)\right)\left(\frac{1}{N}\sum_{k=1}^{N}\frac{f(l_k,v)\cos\theta_{l_k}}{p(l_k,v)}\right)$$

$$\downarrow$$

$$\frac{1}{N}\sum_{k=1}^{N}L_i(l_k)$$

For details see:
http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_slides.pdf
http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf

Cube map prefiltering (1/3)

```
uniform float uRoughness;
uniform vec2 uInverseResolution;

const int numSamples = 1024; // Play with this value

vec2 Hammersley(const in int index, const in int numSamples){
#ifdef RealOpenGL
#ifdef GL4_AND_UP
  uint reversedIndex = bitfieldReverse(uint(index));
#else
  uint reversedIndex = uint(index);
  reversedIndex = (reversedIndex << 16) | (reversedIndex >> 16);
  reversedIndex = ((reversedIndex & 0x00ff00ffu) << 8) | ((reversedIndex & 0xff00ff00u) >> 8);
  reversedIndex = ((reversedIndex & 0x0f0f0f0fu) << 4) | ((reversedIndex & 0xf0f0f0f0u) >> 4);
  reversedIndex = ((reversedIndex & 0x33333333u) << 2) | ((reversedIndex & 0xccccccccu) >> 2);
  reversedIndex = ((reversedIndex & 0x55555555u) << 1) | ((reversedIndex & 0xaaaaaaaau) >> 1);
#endif
  return vec2(fract(float(index) / numSamples), float(reversedIndex) * 2.3283064365386963e-10);
#else
  // Special fake Hammersley variant for WebGL and ESSL, since WebGL and ESSL do not support
  // uint and bit operations
  vec2 r = fract(vec2(float(index) * 5.3983, float(int(int(2147483647.0) - index)) * 5.4427));
  r += dot(r.yx, r.xy + vec2(21.5351, 14.3137));
  return fract(vec2(float(index) / float(numSamples), (r.x * r.y) * 95.4337));
#endif
}
```

Cube map prefiltering (2/3)

```
vec3 ImportanceSampleGGX(const in vec2 e, const in float roughness, const in vec3 normal){
  float m = roughness * roughness, m2 = m * m;
  float phi = 2.0 * 3.14159265358979323846264338327795 * e.x;
  float cosTheta = sqrt((1.0 - e.y) / (1.0 + ((m2 - 1.0) * e.y)));
  float sinTheta = sqrt(1.0 - (cosTheta * cosTheta));
  vec3 h = vec3(sinTheta * cos(phi), sinTheta * sin(phi), cosTheta);
  vec3 tangentZ = normalize(normal);
  vec3 upVector = (abs(tangentZ.z) < 0.999) ? vec3(0.0, 0.0, 1.0) : vec3(1.0, 0.0, 0.0);
  vec3 tangentX = normalize(cross(upVector, tangentZ));
  vec3 tangentY = cross(tangentZ, tangentX);
  return (tangentX * h.x) + (tangentY * h.y) + (tangentZ * h.z);
}
```

Cube map prefiltering (3/3)

```
void main(){
  // Equirectangular environment map output
  vec2 thetaphi = (((vec2(gl_FragCoord.xy) * uInverseResolution) * 2.0) - vec2(1.0)) *
                  vec2(3.14159265358979323846264338327950,
                       -1.57079632679489661923132169163980);
  vec3 R = normalize(vec3(cos(thetaphi.y) * cos(thetaphi.x),
                          sin(thetaphi.y),
                          cos(thetaphi.y) * sin(thetaphi.x)));
  vec3 N = R;
  vec3 V = R;
  vec4 r = vec4(0.0);
  float w = 0.0;
  for(int i = 0; i < numSamples; i++){
    vec3 H = ImportanceSampleGGX(Hammersley(i, numSamples), uRoughness, N);
    vec3 L = -reflect(V, H); // ((2.0 * dot(V, H )) * H) - V;
    float nDotL = clamp(dot(N, L), 0.0, 1.0);
    if(nDotL > 0.0){
      r += textureCube(iChannel0, L) * nDotL;
      w += nDotL;
    }
  }
  gl_FragColor = r / max(w, 1e-4);
}
```

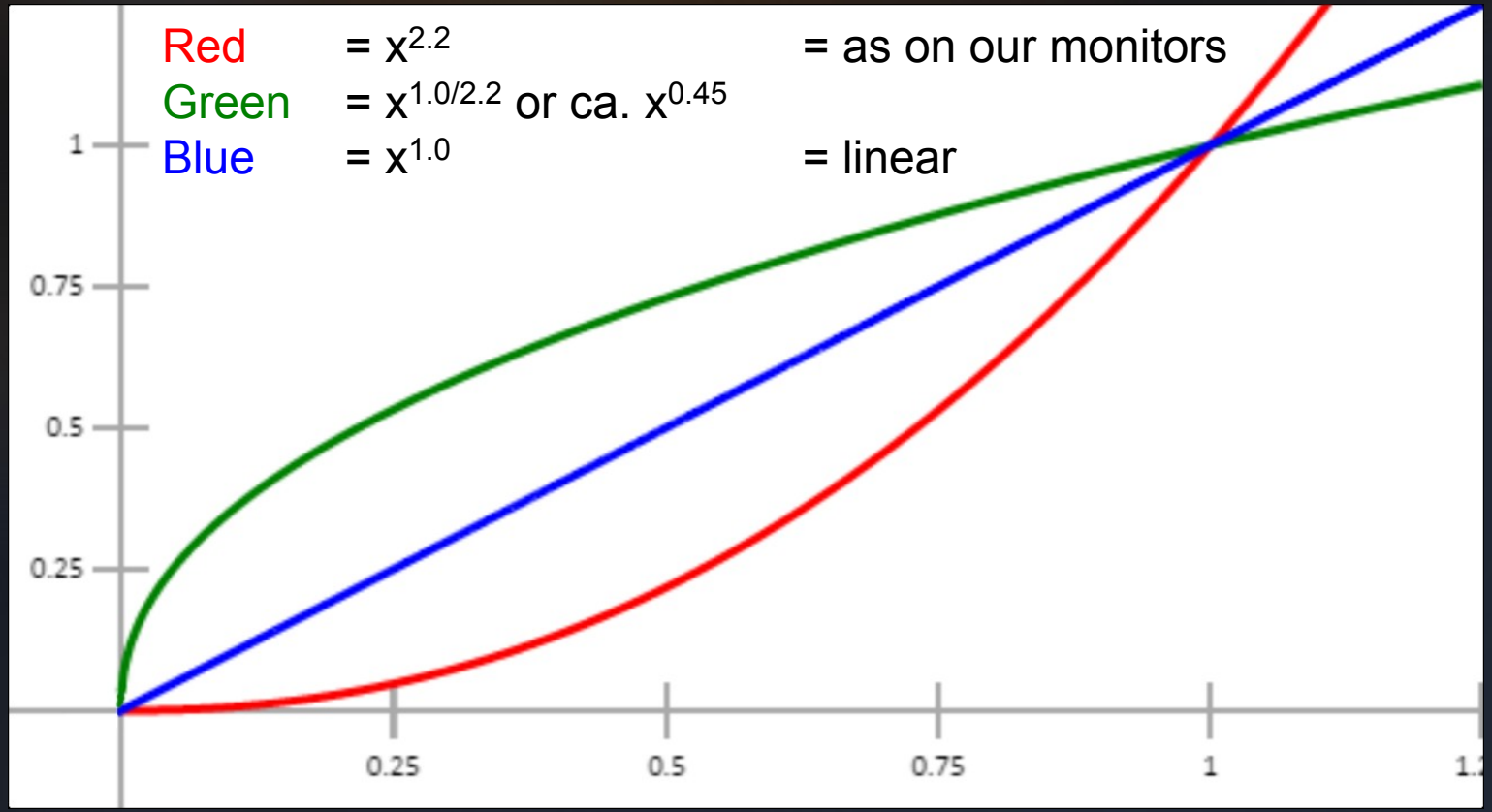
Example usage

```
vec3 specularColor = mix(0.08 * specularMapColor, baseColor, gMaterialMetallic);

float IBLMipMapRemappedMaterialRoughness = [you need to find yourself, as my remapping formula is not perfect];

vec3 IBLEnvMapColor = textureLod(uTexIBLEnvMap,
                                 vec2((atan(IBLDirection.z,
                                            IBLDirection.x) /
                                       6.28318530717958647692528676655 9) + 0.5,
                                      acos(IBLDirection.y) /
                                      3.14159265358979323846264338327 95),
                                      clamp(IBLMipMapRemappedMaterialRoughness *
                                            IBLMaxMipMapLevel, 0.0,
                                            IBLMaxMipMapLevel)).xyz;
vec2 IBLBRDF = texture(uTexEnvIBLBRDF,
                       vec2(gMaterialRoughness, dot(gNormal, gViewDirection))).xy;

vec3 IBLColor = IBLEnvMapColor * ((specularColor * IBLBRDF.x) + IBLBRDF.yyy);
```

Gamma vs. linear-space

you must believe in linear-space lighting

What is Gamma

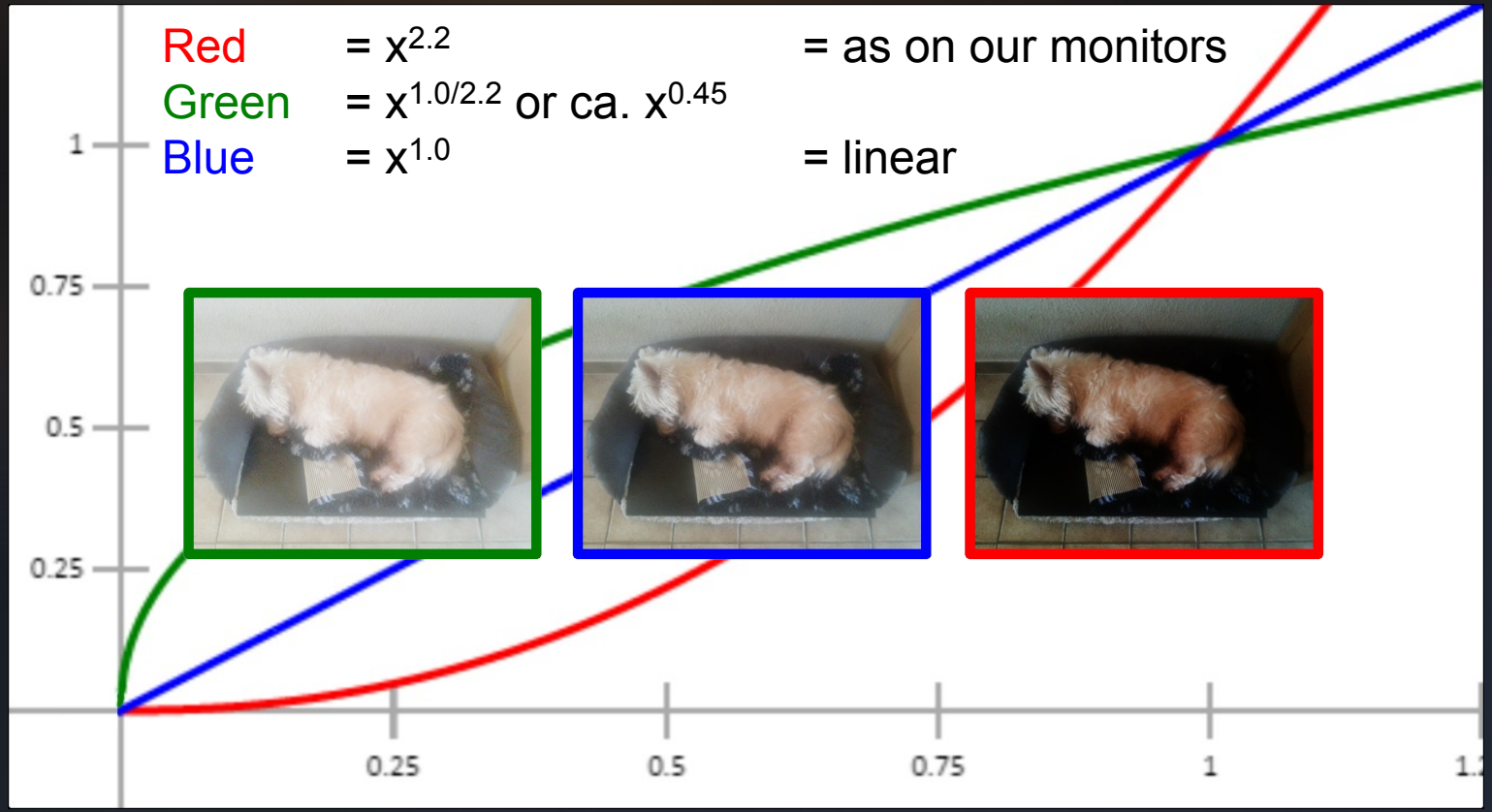


F(x) = x

$$F(x) = x^{1.0/2.2}$$

$$F(x) = x^{2.2}$$

What is Gamma

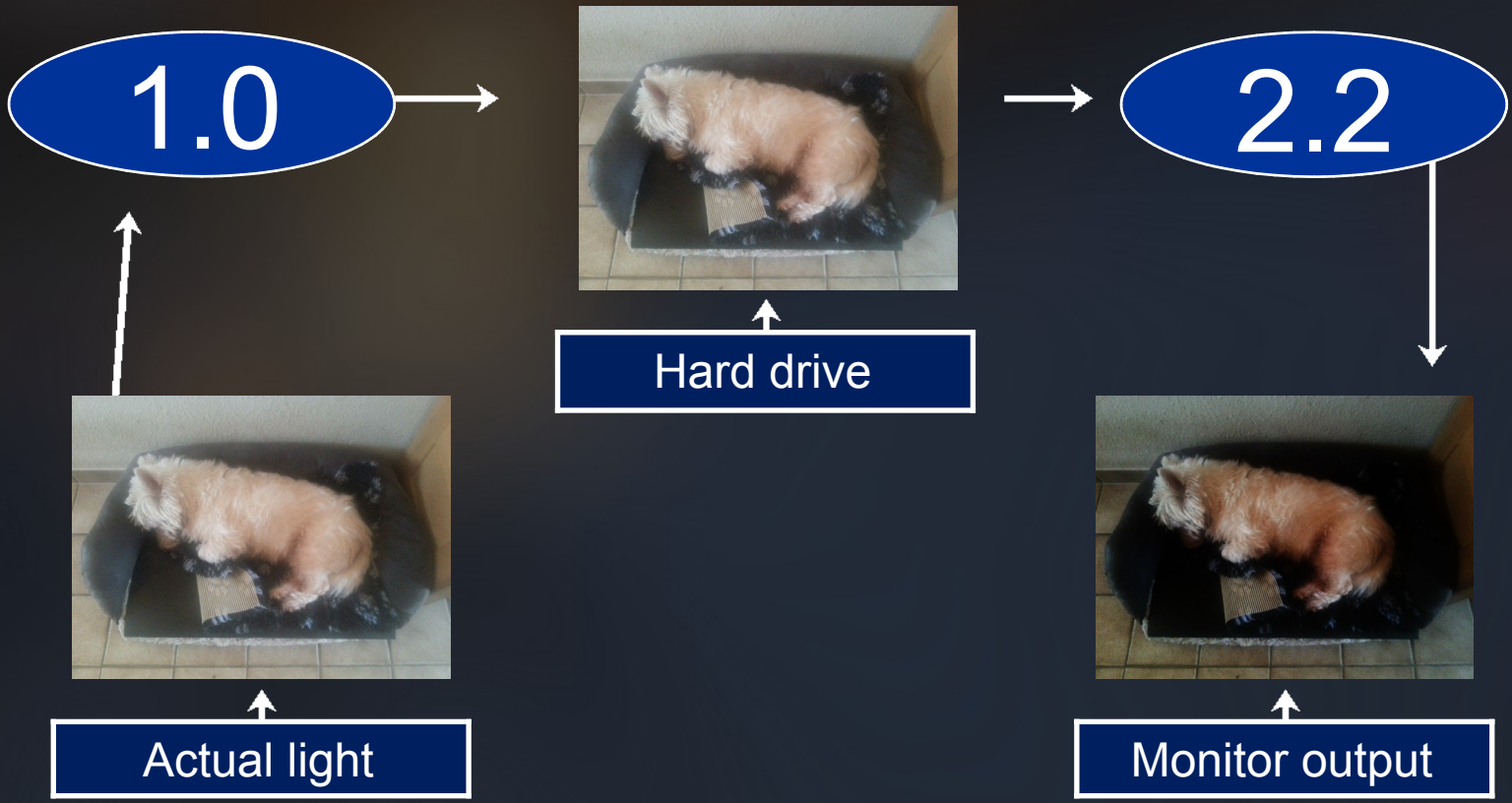


Converting between gamma curves

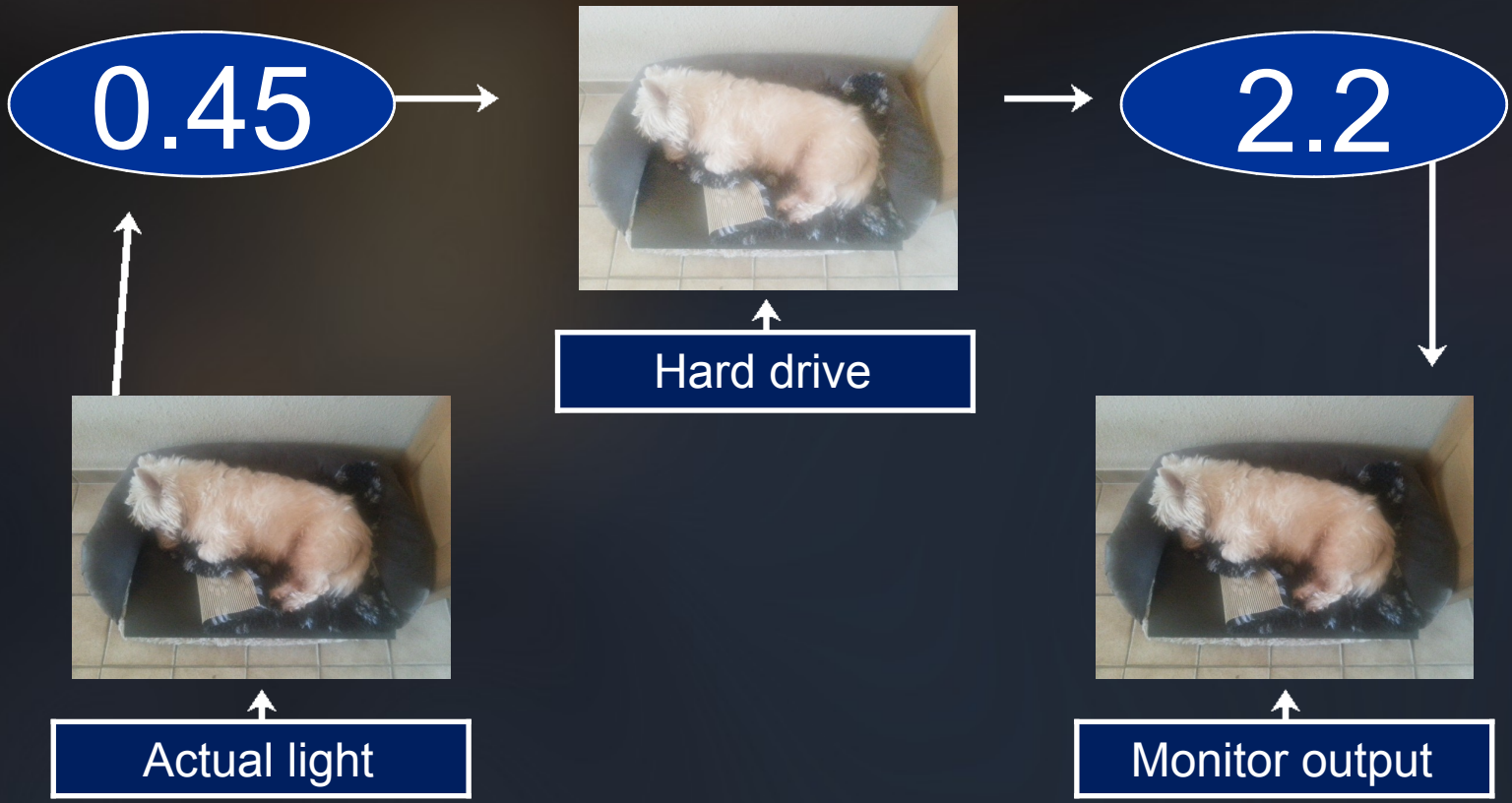
Monitor gamma

A common monitor has a gamma of about 2.2. If the left image will be outputed to the framebuffer directly without any tone mapping operator, then it will actually look like this:

A linear camera...



A consumer camera...



Monitor gamma

So when a common monitor has a gamma of about 2.2, then this left image (how you never see it) is stored on your hard drive:

Gamma vs. linear-space

- Textures, which aren't already in linear-space, must be converted into linear-space, either by at loading time (incidentally also converted to R11G11B10F, RGB16F, RGBA16F RGB32F or RGBA32F etc.), by **pow(texture(Sampler, uv), 2,2)** in the fragment shader or by using **GL_TEXTURE_SRGB**

- The framebuffer output must be handled correctly too, either by **pow(color.rgb, vec3(1.0 / 2.2))** respectively **pow(color.rgb, vec3(0.45))** or by using **glEnable(GL_FRAMEBUFFER_SRGB)**.

  – But attention, some tone mapping operators like the Jim Hejl filmic tone mapping operator includes already the **pow(color.rgb, vec3(1.0 / 2.2))** operation.

Gamma vs. linear-space

- Your HDR frame buffer objects and HDR textures should be at least R11G11B10F, RGB16F, RGBA16F, RGB32F or RGBA32F.

Tone mapping

Tone mapping and why?

- In nature, a dynamic range (ratio between the largest and smallest luminance) of about 10^9:1 exists, when you compare sunlight with starlight.
- At any given time, the typically observed dynamic range is on the order of 1:10^3.
- The human visual perception solves the tone mapping problem because it is able to adapt to the ambient brightness conditions.
- On different absolute brightness conditions (photopic, mesopic, scotopic) the human eye is not linear.
- Tone-mapping operators differ in speed, strength and presence of artifacts, maintaining image detail and the ability to compress HDR images with very high dynamic range.
  - But this presentation will cover only the Jim Heji filmic tone mapping operator.

Tone mapping

Supraleiter and the 64k from yesterday are using the Jim Heji filmic tone mapping operator together with compute-shader-based temporal automatic exposure adaption.

Jim Heji filmic tone mapping operator

```
vec4 hejl(const in vec4 color) {
  vec4 x = max(vec4(0.0), color - vec4(0.004));
  return (x * ((6.2 * x) + vec4(0.5))) /
         max(x * ((6.2 * x) + vec4(1.7)) + vec4(0.06), vec4(1e-8));
}

// From automatic exposure adaption
float adaptedLum = texture(uTexAdaptedLuminance, vec2(0.5)).r;
float exposure = uExposure * (autokey(adaptedLum) / max(adaptedLum, 1e-8));

// Apply it
gl_FragColor = hejl(color * exposure);
```

Note that you don’t need the **pow(color.rgb, vec3(1.0 / 2.2))** for this one.

Automatic exposure adaption

- Mipmap the frame buffer object texture up to 1x1 down, or get the average luminance in another way of your choice.

- Then
  - previousLuminance = mix(previousLuminance, newLuminance, uAdaptionRate);
    - uAdaptionRate is at Supraleiter:
      - 1.0 - exp(-(GetRenderingDeltaTime() * 2.5))

- In the 64k from yesterday, the automatic exposure adaption is completely faked with a automation envelope.

Bloom

Bloom

- Bloom is the effect when the light from a really bright object appears to bleed over the rest of the image.
- The physical basis of bloom is that camera lenses in the real world can never perfectly focus light from a point onto another point.
- Even a perfect camera lens convolves the incoming image with an Airy disk:

The light from a single point lands on the sensor as a bright central spot surrounded by much fainter concentric rings, which are brightened in this picture right so you can see them easier:

Bloom

For common (not really very tiny) light sources, the individual rings won’t be visible because they are all overlapped and blurred together.

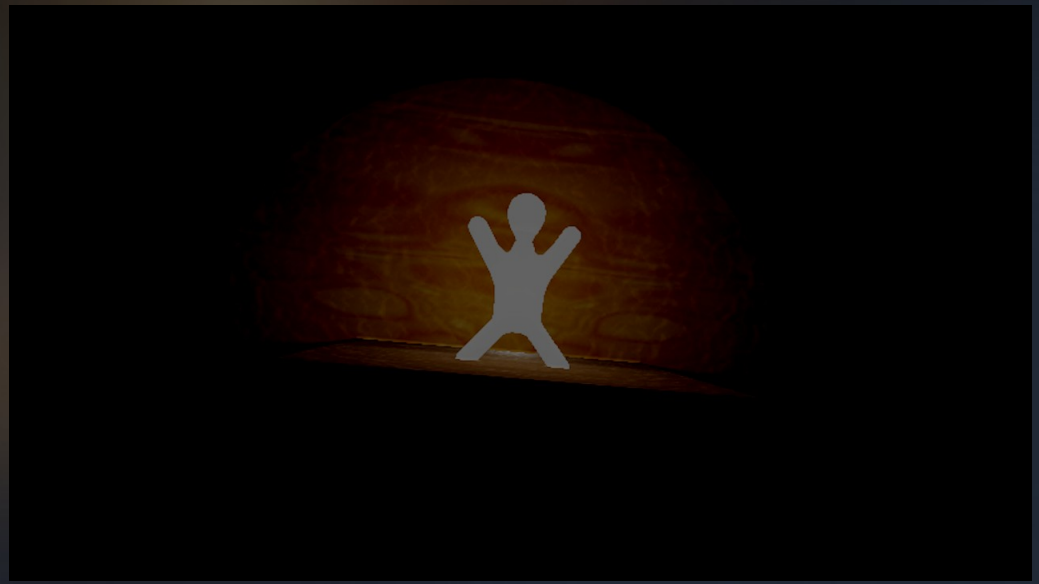
Bloom

Given this input image:

All operations for this bloom example were processed manually with paint.net in LDR range for better visibility of the single processing steps of the bloom effect.

Bloom

Subtract a threshold from it. Result:



For HDR, the threshold value can be 1.0, so that the 0.0 .. 1.0 range will be unbloomed and only the range over 1.0 will be bloomed then.

As optional additional step, you can scale the thresholded image down with a factor.

Bloom

Blur it in two seperate passes in the x und y directions. Result:

Bloom

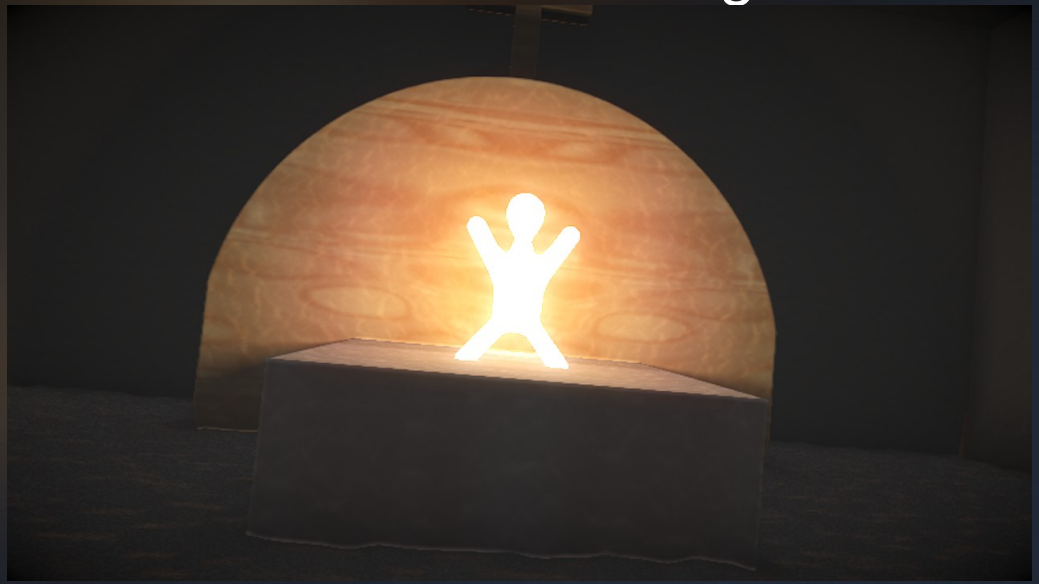
Add the blurred image to the original. Result:

Bloom

The original input image again:

Bloom

The bloomed result again:




Lensflares

Lensflares

- Lensflares are another artifact of camera lenses, caused by various interactions between a or more lens and the light passing through it. A small proportion of it is reflected back.
- Most cameras have multiple camera lenses in series instead of just a single camera lens.
- Therefore, the light can bounce back and forth between the camera lens, so it can land on the camera sensor end up somewhere.
- We can approximating it with some simple tricks.

Physically-Based Real-Time Lens Flare Rendering

Originally I wanted to use

http://resources.mpi-inf.mpg.de/lensflareRendering/

(a mostly physically based lensflare rendering technology), but it is patented.

Hey mostly European paper authors, software patents are here in Germany and generally in Europe in the most not-hardware-bounded cases invalid.

Shame on you!

Note: This is actually a problem only for commercial purposes, but not for artistic purposes. But for me it's a matter of principle.

So, we do need a alternative

the following described technology is not physically based, but it looks good.

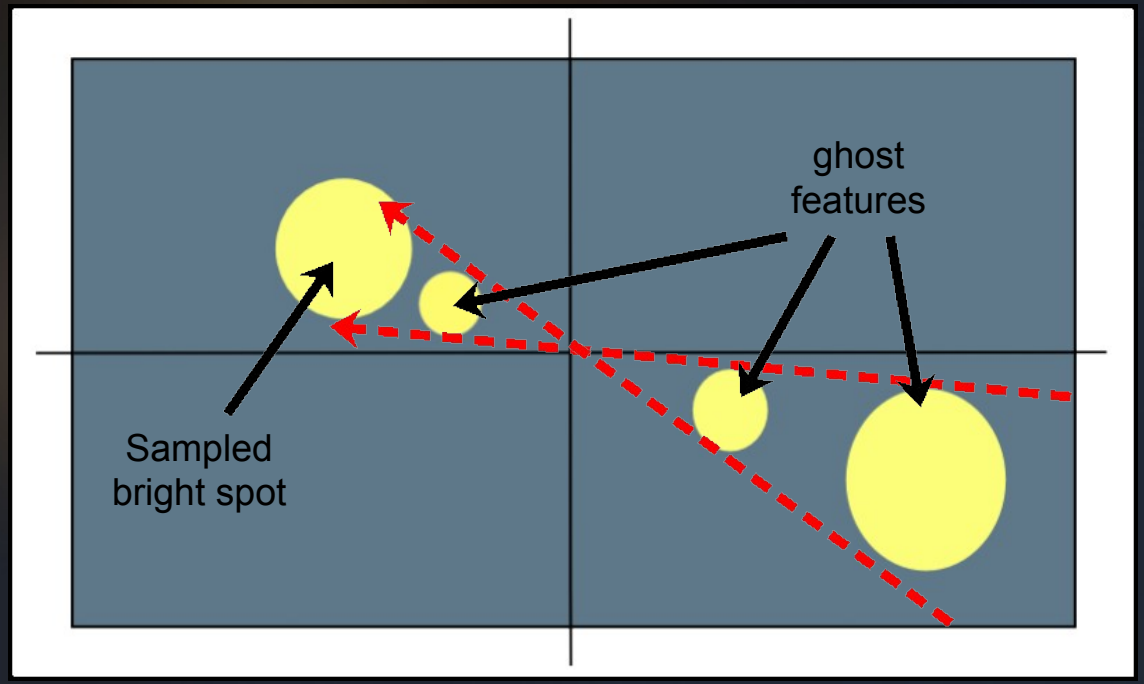
Here is the alternative algorithm

The alternative algorithm consists of 4 stages:

1. Downsample and threshold
2. Generate lens flare features
3. Blurring
4. Upsample and blend over the original image

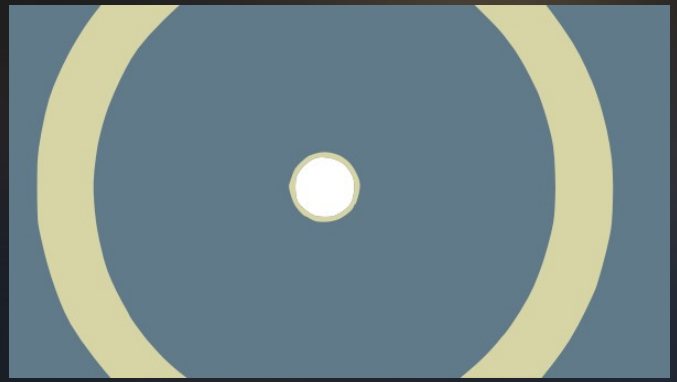
Ghosts

The light from bright spots of the input image is mirroring

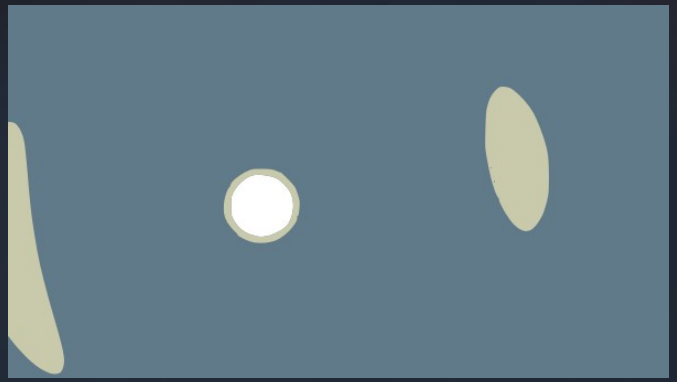


Halos

Halo is the effect you get when pointing directly into a bright light.



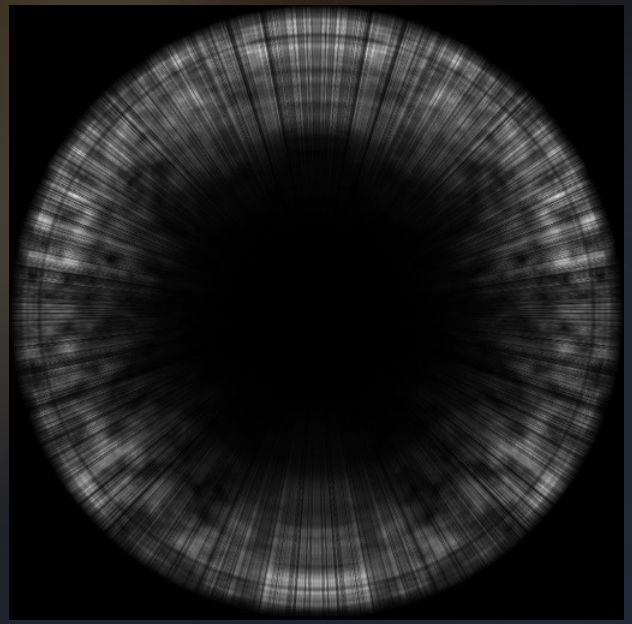
Full halo from a central light source



Partial halo from a light source, which is not in the image center

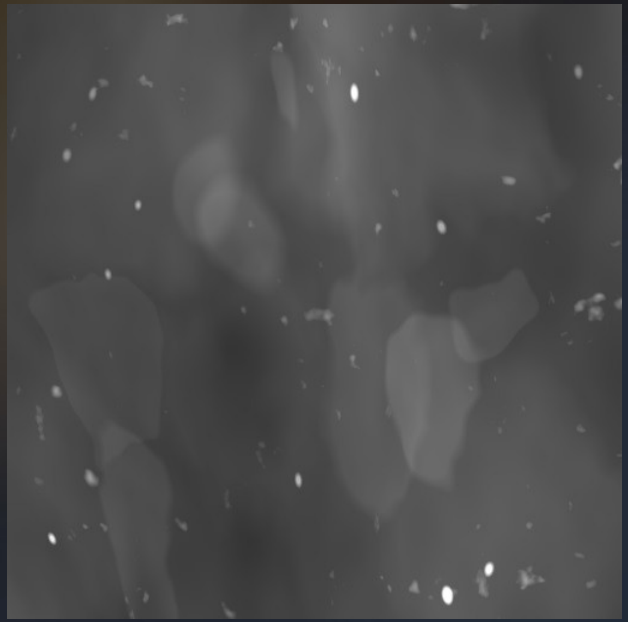
Starburst modulation texture

This is the procedural Starburst modulation texture from the 64k from yesterday



Lens dirt modulation texture

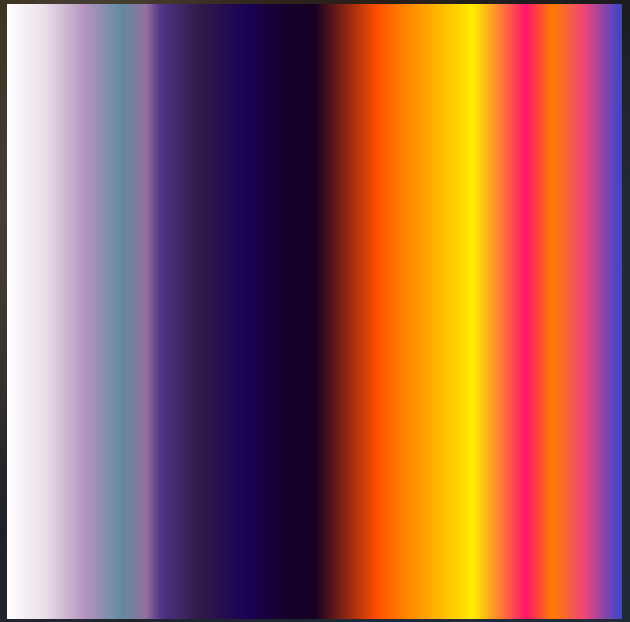
This is the procedural lens dirt modulation texture from the 64k from yesterday



Okay, okay, not exactly like in the 64k, here the texture has somewhat less ambient fbm noise than in the 64k. :-)

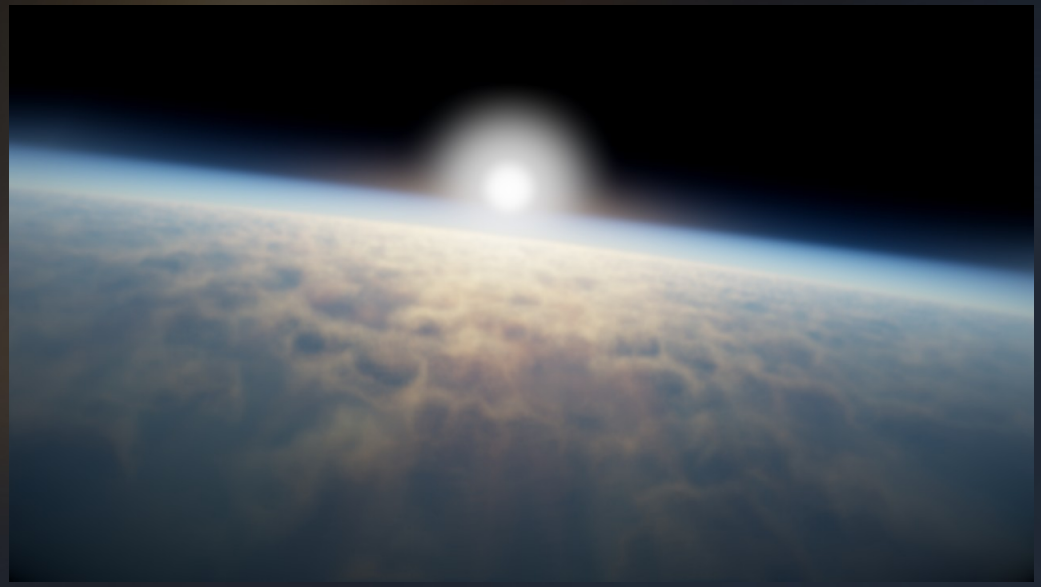
Lens color modulation texture

This is the procedural lens color modulation texture from the 64k from yesterday



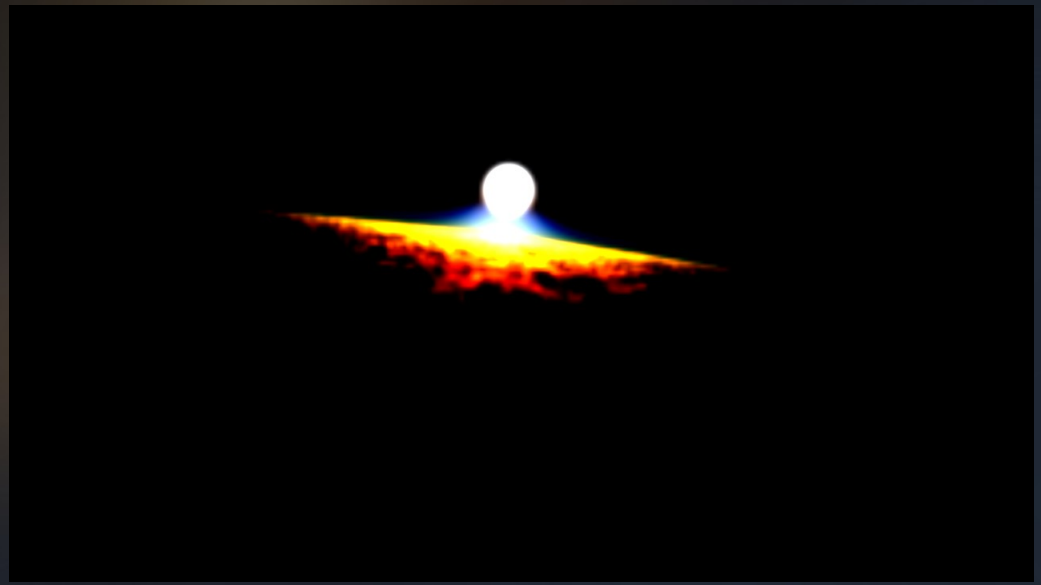
Lensflares

Given this input image



Lensflares

Downsampling and thresholding it. Result:

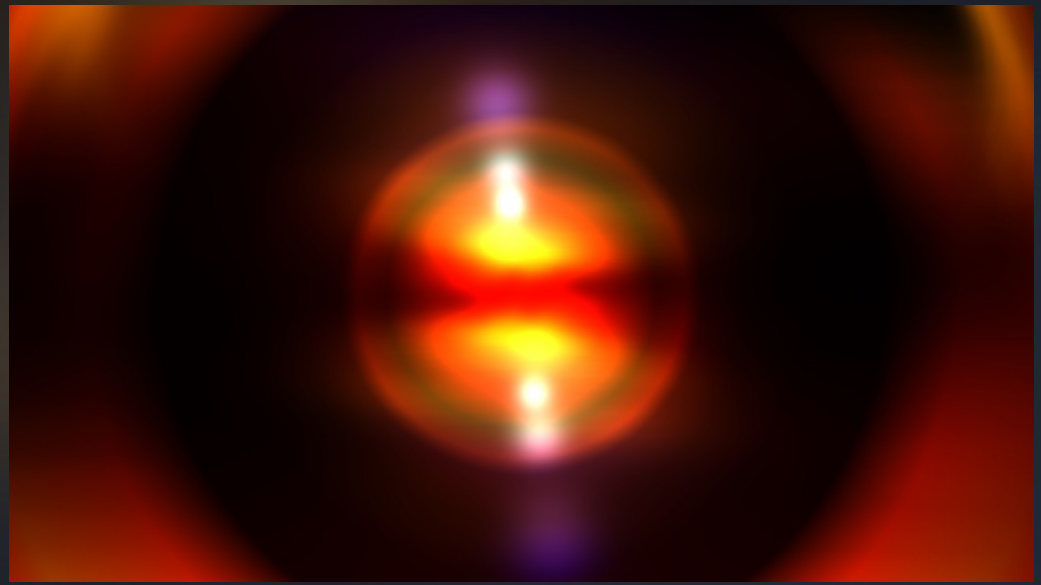


Lensflares

Blurring it in the x and y directions. Result:

Lensflares

Get the lensflares features from it. Result:

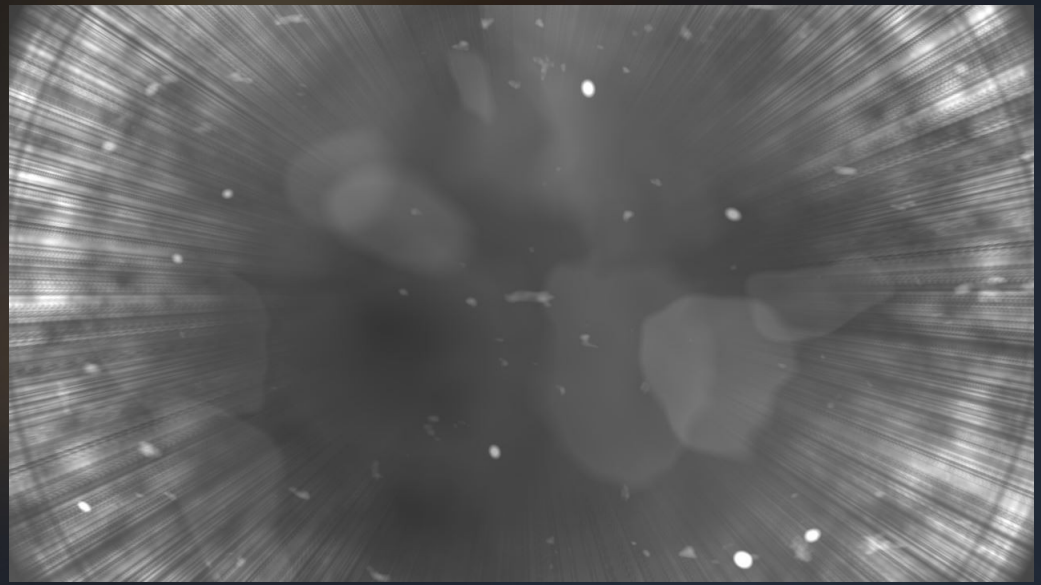


by modulating the lens color lookup texture with:

(c * textureLod(uTexLensColor, vec2(length(vec2(0.5) - aspectTexCoord) / length(vec2(0.5))), 0))

Lensflares

Mix the starbust with lens dirt texture. Result:



Lensflares

Combine these textures all. Result:

without tone mapping

Lensflares

Combine these textures all. Result:

with tone mapping

Downsample and threshold code snippet

```
#version 150
uniform sampler2D fboInput;
void main(){
  ivec2 coord = ivec2(gl_FragCoord.xy) * 4;
  vec4 c = vec4(0.0);
  for(int y = 0; y < 4; y++){
    for(int x = 0; x < 4; x++){
      c += texelFetch(fboInput, coord + ivec2(x, y), 0);
    }
  }
  gl_FragColor = max(vec4(0.0), vec4((c * (1.0 / 16.0))  - vec4(1.0))) * 2.0;
}
```

Seperate linear interpolator optimized 17x17 gaussian blur code snippet

```
#version 150
uniform sampler2D fboInput;
uniform vec2 blurDirection;
varying vec2 vTexCoord;

void main(){
  vec2 lPixelOffset = blurDirection / textureSize(fboInput, 0).xy;
  gl_FragColor =
         ((textureLod(fboInput, vTexCoord + (lPixelOffset * 0.6494988171273325), 0.0) +
           textureLod(fboInput, vTexCoord - (lPixelOffset * 0.6494988171273325), 0.0)) *
           0.22251469736395335)+
         ((textureLod(fboInput, vTexCoord + (lPixelOffset * 2.405752812667325), 0.0) +
           textureLod(fboInput, vTexCoord - (lPixelOffset * 2.405752812667325), 0.0)) *
           0.1934401294345523)+
         ((textureLod(fboInput, vTexCoord + (lPixelOffset * 4.334747038826614), 0.0) +
           textureLod(fboInput, vTexCoord - (lPixelOffset * 4.334747038826614), 0.0)) *
           0.06915657278613227)+
         ((textureLod(fboInput, vTexCoord + (lPixelOffset * 6.27051066767145), 0.0) +
           textureLod(fboInput, vTexCoord - (lPixelOffset * 6.27051066767145), 0.0)) *
          0.013708153234290525);
}
```

Get features + combination + tone mapping code snippet (1/3)

```
varying vec2 vTexCoord;
uniform sampler2D fboLensColor, fboLensStarBurst, fboLensDirt, fboCamera, fboOriginalInput,
                  fboBlurred;
uniform vec3 tc;
uniform float exposureAutomationEnvelope;
const float uAspectRatio = 16.0 / 9.0;
const float uInverseAspectRatio = 9.0 / 16.0;
const float uDispersal = 0.3;
const float uHaloWidth = 0.6;
const float uDistortion = 1.5;
vec4 textureDistorted(const in sampler2D tex, const in vec2 texCoord, const in vec2 direction,
                      const in vec3 distortion) {
  return vec4(textureLod(tex, (texCoord + (direction * distortion.r)), 0.0).r,
              textureLod(tex, (texCoord + (direction * distortion.g)), 0.0).g,
              textureLod(tex, (texCoord + (direction * distortion.b)), 0.0).b,
              1.0);
}
```

Get features + combination + tone mapping code snippet (2/3)

```
vec4 getLensflare(){
  vec2 aspectTexCoord = vec2(1.0) - (((vTexCoord - vec2(0.5)) * vec2(1.0, uInverseAspectRatio))
                        + vec2(0.5));
  vec2 texCoord = vec2(1.0) - vTexCoord;
  vec2 ghostVec = (vec2(0.5) - texCoord) * uDispersal;
  vec2 ghostVecAspectNormalized = normalize(ghostVec * vec2(1.0, uInverseAspectRatio)) *
                                  vec2(1.0, uAspectRatio);
  vec2 haloVec = normalize(ghostVec) * uHaloWidth;
  vec2 haloVecAspectNormalized = ghostVecAspectNormalized * uHaloWidth;
  vec2 texelSize = vec2(1.0) / vec2(textureSize(fboBlurred, 0));
  vec3 distortion = vec3(-(texelSize.x * uDistortion), 0.0, texelSize.x * uDistortion);
  vec4 c = vec4(0.0);
  for (int i = 0; i < 8; i++) {
    vec2 offset = texCoord + (ghostVec * float(i));
    c += textureDistorted(fboBlurred, offset, ghostVecAspectNormalized, distortion) *
         pow(max(0.0, 1.0 - (length(vec2(0.5) - offset) / length(vec2(0.5)))), 10.0);
  }
  vec2 haloOffset = texCoord + haloVecAspectNormalized;
  return (c * textureLod(fboLensColor, vec2(length(vec2(0.5) - aspectTexCoord) /
         length(vec2(0.5))), 0)) +
         (textureDistorted(fboBlurred, haloOffset, ghostVecAspectNormalized, distortion) *
         pow(max(0.0, 1.0 - (length(vec2(0.5) - haloOffset) / length(vec2(0.5)))), 10.0));
}
```

Get features + combination + tone mapping code snippet (3/3)

```
vec4 hejl(const in vec4 color) {
  vec4 x = max(vec4(0.0), color - vec4(0.004));
  return (x * ((6.2 * x) + vec4(0.5))) / max(x * ((6.2 * x) + vec4(1.7)) + vec4(0.06),
         vec4(1e-8));
}
void main(){
  vec2 texCoord = ((vTexCoord - vec2(0.5)) * vec2(16.0 / 9.0, 1.0) * 0.5) + vec2(0.5);
  vec4 lensMod = texture(fboLensDirt, vTexCoord);
  vec3 forwardVector = normalize(texelFetch(fboCamera, ivec2(1, 0), 0).xyz -
                                 texelFetch(fboCamera, ivec2(0, 0), 0).xyz);
  vec3 rightVector = normalize(cross(forwardVector, texelFetch(fboCamera, ivec2(2, 0),
                                                               0).xyz));
  vec3 upVector = normalize(cross(rightVector, forwardVector));
  mat3 lensStarRotationMatrix = mat3(rightVector, upVector, forwardVector);
  float lensStarRotationAngle = ((lensStarRotationMatrix[2].x + lensStarRotationMatrix[1].z) *
                                 3.14159) * 0.5;
  vec2 lensStarTexCoord = (mat2(cos(lensStarRotationAngle), -sin(lensStarRotationAngle),
                                sin(lensStarRotationAngle), cos(lensStarRotationAngle)) *
                                (texCoord - vec2(0.5))) + vec2(0.5);
  lensMod += texture(fboLensStarBurst, lensStarTexCoord);
  vec4 color = textureLod(fboOriginalInput, vTexCoord, 0.0) + (getLensflare() * lensMod);
  float exposure = mix(1.25, 4.0, exposureAutomationEnvelope);
  float vignette = pow(max(0.0, 1.0 - (length(vec2(0.5) - vTexCoord) / length(vec2(0.5)))),
                       0.4);
  gl_FragColor = hejl(color * exposure * vignette);
}
```

Motion blur

A Reconstruction Filter for Plausible Motion Blur

http://graphics.cs.williams.edu/papers/MotionBlurI3D12/

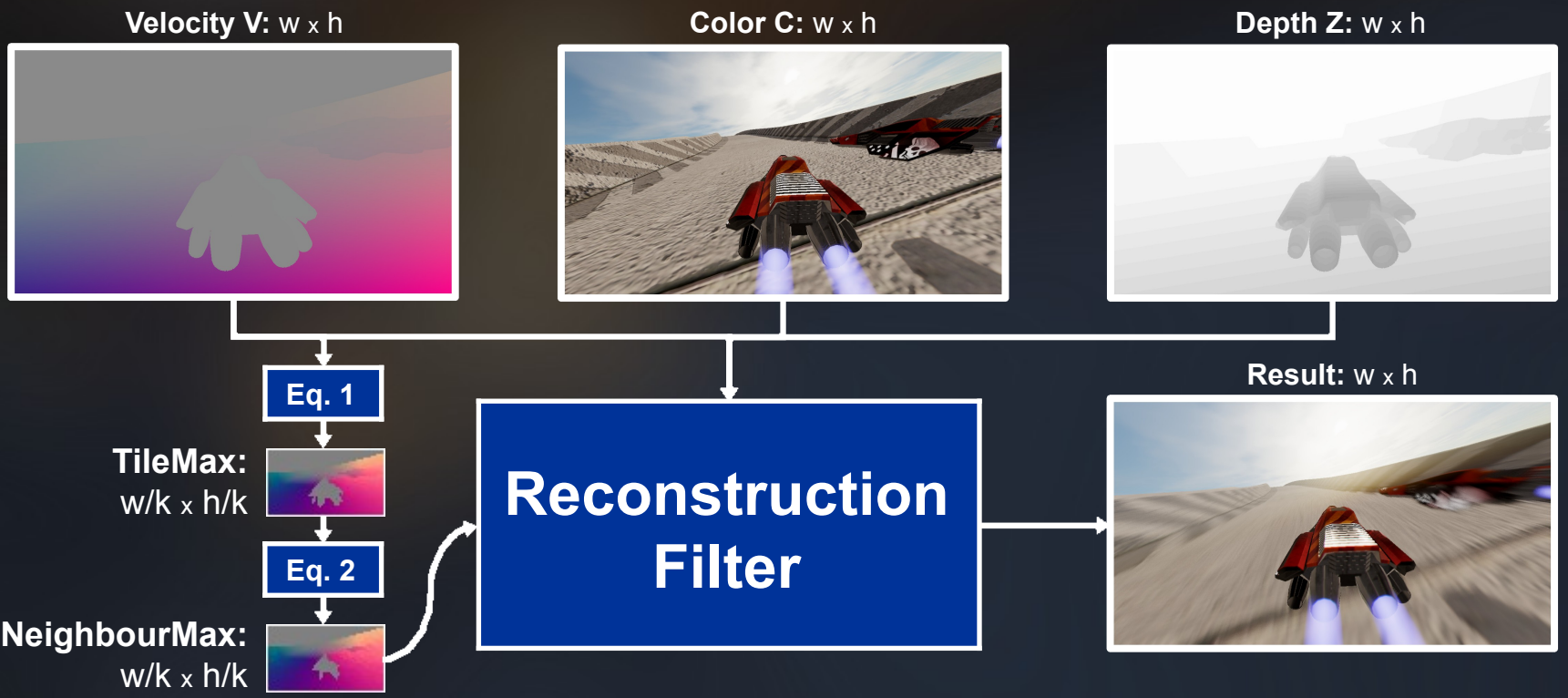
Motion blur

A blur in images of moving objects is called motion blur, that do not result from faulty focusing, aperture setting or other incorrect operation and settings of the photographer or filmmaker.

It is obtained only by a noticeable during the exposure time movement of the subject and increases in proportion to the exposure time and the angular velocity of the object relative to the camera.

In computer games, demos and animation (computer generated movies) motion blur is used to make movements more realistic.

A Reconstruction Filter for Plausible Motion Blur



Assume each tile has a predominant velocity vector

A Reconstruction Filter for Plausible Motion Blur

$$\vec{qx} = \frac{1}{2}(X - X') \cdot (\text{expourse time}) \cdot (\text{frame rate})$$

$$V[X] = \frac{\vec{qx} \max\left(0.5px, \min\left(\left|\vec{qx}\right|, k\right)\right)}{\left|\vec{qx}\right| + \varepsilon}$$

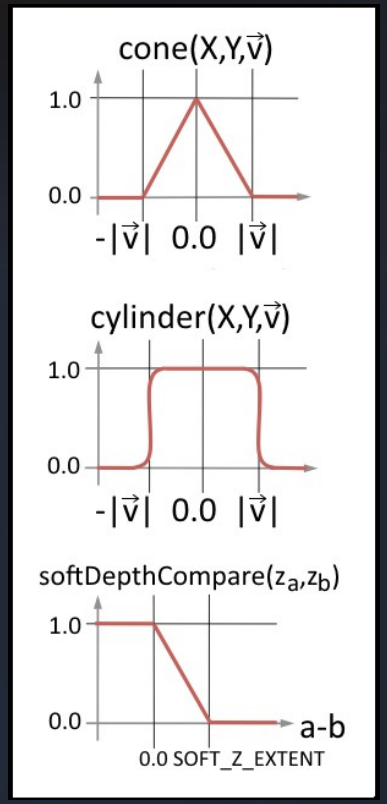
Equation 1: $\text{TileMax}[x,y] = \underset{u\in[0,k]}{\text{vmax}}\ \underset{v\in[0,k]}{\text{vmax}}\left(V[kx+u, ky+v]\right)$

Equation 2: $\text{NeighbourMax}[x,y] = \underset{u\in[-1,1]}{\text{vmax}}\ \underset{v\in[-1,1]}{\text{vmax}}\left(TileMax[x+u, y+v]\right)$

$$cone\left(x, y, \vec{v}\right) = clamp\left(1 - \frac{|X-Y|}{\left|\vec{v}\right|}, 0, 1\right)$$

$$cylinder\left(x, y, \vec{v}\right) = 1.0 - smoothstep\left(0.95\left|\vec{v}\right|, 1.05\left|\vec{v}\right|, |X-Y|\right)$$

$$softDepthCompare(z_a, z_b) = clamp\left(1 - \frac{(z_a - z_b)}{SOFT\_Z\_EXTENT}, 0, 1\right)$$



Velocity buffer filling

in vertex shader:

```
    vCurrentPosition = uModelViewProjectionMatrix * vec4(aPosition, 1.0);
    vPreviousPosition = uPreviousModelViewProjectionMatrix * vec4(aPosition, 1.0);
    gl_Position = vCurrentPosition;
```

in fragment shader:

```
  oVelocity = vec2((vCurrentPosition.xy / vCurrentPosition.w) - (vPreviousPosition.xy / vPreviousPosition.w));
```

Post-process for the optimized reconstruction shader:

```
#version 400
uniform sampler2D uTexVelocityBuffer, uTexDepthBuffer;
uniform vec3 uClipPlaneDepthConstants, uClipPlaneScaleConstants;
uniform int uMotionBlurTileSize;
uniform float uMotionBlurScale;
layout(location=0) out vec4 oVelocityDepth;
void main(){
  ivec2 lTexCoord = ivec2(gl_FragCoord.xy);
  vec2 lQX = (texelFetch(uTexVelocityBuffer, lTexCoord, 0).xy * float(uMotionBlurTileSize)) * (0.5 *
             uMotionBlurScale), lQXLength = length(lQX);
  float lDepth = uClipPlaneDepthConstants.x / (uClipPlaneDepthConstants.y - (texelFetch(uTexDepthBuffer, lTexCoord, 0).x *
                 uClipPlaneDepthConstants.z));
  lDepth = clamp((lDepth - uClipPlaneScaleConstants.x) * uClipPlaneScaleConstants.z, 0.0, 1.0) * 2048.0;
  oVelocityDepth = vec4(vec2((lQX * max(0.5, min(lQXLength, uMotionBlurTileSize))) / max(lQXLength + 1e-4, 1e-4)),
                        vec2(floor(vec2(lDepth, fract(lDepth) * 2048.0))));
}
```

```
State.SetUniform3f(uClipPlaneDepthConstants, 2.0 * (ZNear * ZFar), ZFar + ZNear, ZFar - ZNear);
State.SetUniform3f(uClipPlaneScaleConstants, ZNear, ZFar - ZNear, 1.0 / (ZFar - ZNear));
```

A Reconstruction Filter for Plausible Motion Blur

TileMax

```
#version 400
uniform sampler2D uTexVelocityBuffer;
uniform int uTileSize;
layout(location=0) out vec2 oOutput;
void main(){
  ivec2 inputShift = ivec2(0);
  vec2 m = vec2(0.0);
  float largestSquaredMagnitude = 0.0;
  ivec2 texSize = ivec2(textureSize(uTexMotionBlurTempTileMax, 0).xy);
  ivec2 tileOffset = (ivec2(gl_FragCoord.xy) * ivec2(uTileSize, uTileSize)) + inputShift;
  ivec2 maxCoord = texSize - (ivec2(1, 1) + inputShift);
  ivec2 offset;
  for(offset.y = 0; offset.y < uTileSize; offset.y++){
    for(offset.x = 0; offset.x < uTileSize; offset.x++){
      vec2 v = texelFetch(uTexVelocityBuffer, clamp(tileOffset + offset, inputShift, maxCoord), 0).xy;
      float thisSquaredMagnitude = dot(v, v);
      if(thisSquaredMagnitude > largestSquaredMagnitude){
        m = v;
        largestSquaredMagnitude = thisSquaredMagnitude;
      }
    }
  }
  oOutput = vec2(m);
}
```

Supraleiter is using a seperate two-pass variant of it.

A Reconstruction Filter for Plausible Motion Blur

NeighbourMax

```
#version 400
uniform sampler2D uTexMotionBlurTileMax;
uniform int uTileSize;
layout(location=0) out vec2 oOutput;
void main(){
  vec2 m = vec2(0.0); // Vector with the largest magnitude
  float largestMagnitude2 = -1.0; // Squared magnitude of m
  ivec2 maxCoord = textureSize(uTexMotionBlurTileMax, 0) - ivec2(1);
  ivec2 currentTile = ivec2(gl_FragCoord.xy);
  for(int y = -1; y <= 1; y++){
    for(int x = -1; x <= 1; x++){
      ivec2 offset = ivec2(x, y);
      vec2 vmax_neighbor = texelFetch(uTexMotionBlurTileMax, clamp(currentTile + offset, ivec2(0), maxCoord),
                                      0).xy;
      float magnitude2_neighbor = dot(vmax_neighbor, vmax_neighbor); // Magnitude squared
      if (magnitude2_neighbor > largestMagnitude2) {
        vec2 directionOfVelocity = vmax_neighbor;
        float displacement = abs(offset.x) + abs(offset.y);
        vec2 point = sign(offset * directionOfVelocity);
        float distance = (point.x + point.y);
        if(abs(distance) == displacement){
          m = vmax_neighbor;
          largestMagnitude2 = magnitude2_neighbor;
        }
      }
    }
  }
  oOutput = m;
}
```

Unoptimized reconstruction (1/2)

```
#version 400
uniform sampler2D uTexSource;
uniform sampler2D uTexVelocityBuffer;
uniform sampler2D uTexDepthBuffer;
uniform sampler2D uTexMotionBlurNeighbourMax;
uniform float uMotionBlurScale;
uniform int uTileSize;
uniform float uGameTime;
uniform vec3 uClipPlaneDepthConstants;
in vec2 vTexCoord;
layout(location=0) out vec4 oOutput;
const float SOFT_Z_EXTENT = 0.10; // 10 mm
const int S = 6; // 6 samples, but in two passes, so for 6x6 = 36 samples effective
const int Sm = (S - 1) / 2;
float getDepth(const in float d){
  return uClipPlaneDepthConstants.x / (uClipPlaneDepthConstants.y - (d * uClipPlaneDepthConstants.z));
}
float softDepthCompare(const in float za, const in float zb){
  return clamp(1.0 - ((za - zb) / SOFT_Z_EXTENT), 0.0, 1.0);
}
float cone(const in float lengthXY, const in float lengthV){
  return clamp(1.0 - (lengthXY / (lengthV + ((1.0 - step(1e-7, abs(lengthV))) * 1e-7))), 0.0, 1.0);
}
float cylinder(const in float lengthXY, const in float lengthV){
  return clamp(1.0 - smoothstep(0.95 * lengthV, 1.05 * lengthV,  lengthXY), 0.0, 1.0);
}
float randomjitter(vec2 pos){
  return clamp(fract(sin(dot(vec4(pos, vec2(uGameTime, uGameTime * 0.3)), vec4(12.9898, 78.233, 45.164, 94.673))) * 43758.5453) - 0.5, -0.5,
0.5);
}
```

State.SetUniform3f(uClipPlaneDepthConstants, 2.0 * (ZNear * ZFar), ZFar + ZNear, ZFar - ZNear);

Unoptimized reconstruction (2/2)

```
void main(){
  ivec2 X = ivec2(gl_FragCoord.xy);
  ivec2 Xk = X / ivec2(uTileSize);
  vec2 vn = texelFetch(uTexMotionBlurNeighbourMax, Xk, 0).xy;
  vec4 cX = texelFetch(uTexSource, X, 0);
  if(length(vn) < 0.5125){
    oOutput = cX;
    return;
  }
  vec2 vX = texelFetch(uTexVelocityBuffer, X, 0).xy;
  float lvX = length(vX);
  float zX = -getDepth(texelFetch(uTexDepthBuffer, X, 0).x);
  float weight = 1.0 / max(length(vn), 1e-8);
  vec4 sum = cX * weight;
  float j = randomjitter(vec2(X));
  for(int i = 0; i <= S; i++){
    if(i == Sm){
      continue;
    }
    float t = mix(-1.0, 1.0, ((float(i) + j) + 1.0) / float(S + 1));
    ivec2 Y = ivec2(vec2(vec2(X) + (vn * t)) + vec2(0.5));
    vec2 vY = texelFetch(uTexVelocityBuffer, Y, 0).xy;
    float lvY = length(vY), lXY = length(vec2(X - Y)), zY = -getDepth(texelFetch(uTexDepthBuffer, Y, 0).x);
    float f = softDepthCompare(zX, zY), b = softDepthCompare(zY, zX);
    float alphaY = ((f * cone(lXY, lvY)) + (b * cone(lXY, lvX))) + ((cylinder(lXY, lvY) * cylinder(lXY, lvX)) * 2.0);
    weight += alphaY;
    sum += texelFetch(uTexSource, Y, 0) * alphaY;
  }
  oOutput = sum / max(weight, 1e-8);
}
```

Optimized reconstruction

```
#version 400
uniform sampler2D uTexSource;
uniform sampler2D uTexVelocityDepthBuffer; // Combining of RG16F velocity and 32-bit depth buffer into one RGBA16F texture brings at my old NVIDIA Geforce GTX660 ~30% more performance
uniform sampler2D uTexMotionBlurNeighbourMax;
uniform vec3 uClipPlaneScaleConstants;
uniform int uTileSize;
uniform float uGameTime;
in vec2 vTexCoord;
layout(location=0) out vec4 oOutput;
const int S = 6; // 6 samples, but in two passes, so for 6x6 = 36 samples effective
const int Sm = (S - 1) / 2;
const float softZ = 10.0;
void main(){
  ivec2 X = ivec2(gl_FragCoord.xy);
  vec2 vn = texelFetch(uTexMotionBlurNeighbourMax, X / ivec2(uTileSize), 0).xy;
  vec4 cX = texelFetch(uTexSource, X, 0);
  if(length(vn) < 0.505){
    oOutput = cX;
    return;
  }
  vec4 tsX = texelFetch(uTexVelocityDepthBuffer, X, 0);
  vec2 depthDecodeConstants = vec2(uClipPlaneScaleConstants.y) / vec2(2048.0, 4194304.0), vX = vec2(length(tsX.xy), dot(tsX.zw, depthDecodeConstants));
  float w = 1.0 / max(vX.x, 1e-8);
  vec4 s = cX * w;
  float j = clamp(fract(sin(dot(vec4(vec2(X), vec2(uGameTime, w)), vec4(12.9898, 78.233, 45.164, 94.673))) * 43758.5453) - 0.5, -0.5, 0.5);
  for(int i = 0; i <= S; i++){
    if(i == Sm){ continue; }
    ivec2 Y = ivec2(vec2(vec2(X) + (vn * mix(-1.0, 1.0, ((float(i) + j) + 1.0) / float(S + 1)))) + vec2(0.5));
    vec4 tsY = texelFetch(uTexVelocityDepthBuffer, Y, 0);
    vec2 vY = vec2(length(tsY.xy), dot(tsY.zw, depthDecodeConstants));
    vec4 cV = clamp((1.0 - (vec4(length(vec2(X - Y))) / vec2(vY.x, vX.x).xyxy)) + vec4(0.0, 0.0, 0.95, 0.95), vec4(0.0), vec4(1.0)); // Velocity compare
    float aY = dot(clamp((vec2(1.0) + (vec2(vX.y, vY.y) * softZ)) - (vec2(vY.y, vX.y) * softZ), vec2(0.0), vec2(1.0)), cV.xy) +      // Depth compare
               ((cV.z * cV.w) * 2.0);
    w += aY;
    s += texelFetch(uTexSource, Y, 0) * aY;
  }
  oOutput = s / max(w, 1e-8);
}
```

State.SetUniform3f(uClipPlaneScaleConstants, ZNear, ZFar - ZNear, 1.0 / (ZFar - ZNear));


Depth of field

Depth of field

Depth of field is the area in front of and behind the selected distance at which a photo showing a sharp image. Therefore, Depth of field means that the main motive of your picture is sharp to see and the foreground and background are blurred. This not only looks good, but you can also order exactly the attention of the viewer control and decide what is important and what is unimportant. This effect is used just as in the movies, computer games and demos.

How strong this effect occurs on an image depends on several factors. The two most important are the focal length of the lens (the zoom factor) and the f-number. The closer you zoom to your motive and the larger the aperture (the lower the f-number, for example f3.5) is more blurred your foreground and background.

If you are using auto focus you need to take special care that the camera focuses on the right motive, which is usually the subject in the center.

Circle of Confusion Calculation

```
#version 430
layout(local_size_x=1, local_size_y=1, local_size_z=1) in;
uniform vec4 uClipPlaneDepthOfFieldConstants;
uniform float uDepthOfFieldObjectiveDiameter;
uniform float uDepthOfFieldFOVFactor;
uniform float uDepthOfFieldSpeedFactor;
layout(std430) buffer depthOfFieldData {
  float depthOfFieldDataCoCScale;
  float depthOfFieldDataCoCBias;
  float depthOfFieldDataCurrentFocalPlane;
  float depthOfFieldDataSum;    // Filled by average-z-find-compute-shader
  float depthOfFieldDataWeight; // Filled by average-z-find-compute-shader
};
void main(){
  float lSum = depthOfFieldDataSum;
  float lWeight = depthOfFieldDataWeight;
  float lFocalPlane = mix(depthOfFieldDataCurrentFocalPlane, (lWeight > 1e-5) ? (lSum / lWeight) : 4096.0, uDepthOfFieldSpeedFactor);
  depthOfFieldDataCurrentFocalPlane = lFocalPlane;
  float lFocalLength = 0.035 * uDepthOfFieldFOVFactor;
  float lAperture = lFocalLength / uDepthOfFieldObjectiveDiameter;
  float lApertureFocalLength = lAperture * lFocalLength;
  float lFocalPlaneMinusFocalLength = lFocalPlane - lFocalLength;
  float lCoCScale = ((lApertureFocalLength * lFocalPlane) * uClipPlaneDepthOfFieldConstants.x) / (lFocalPlaneMinusFocalLength * uClipPlaneDepthOfFieldConstants.y);
  float lCoCBias = (lApertureFocalLength * (uClipPlaneDepthOfFieldConstants.z - lFocalPlane)) / (lFocalPlaneMinusFocalLength * uClipPlaneDepthOfFieldConstants.z);
  float lInverseFilmSize = 1.0 / 0.0186; // 35mm widescreen film (21.95mm x 18.6mm)
  depthOfFieldDataCoCScale = lCoCScale * lInverseFilmSize;
  depthOfFieldDataCoCBias = lCoCBias * lInverseFilmSize;
}
```

State.SetUniform4f(uClipPlaneDepthOfFieldConstants, ZFar - ZNear, ZNear * ZFar, ZNear, ZFar);

Depth of field

First the Supraleiter variant

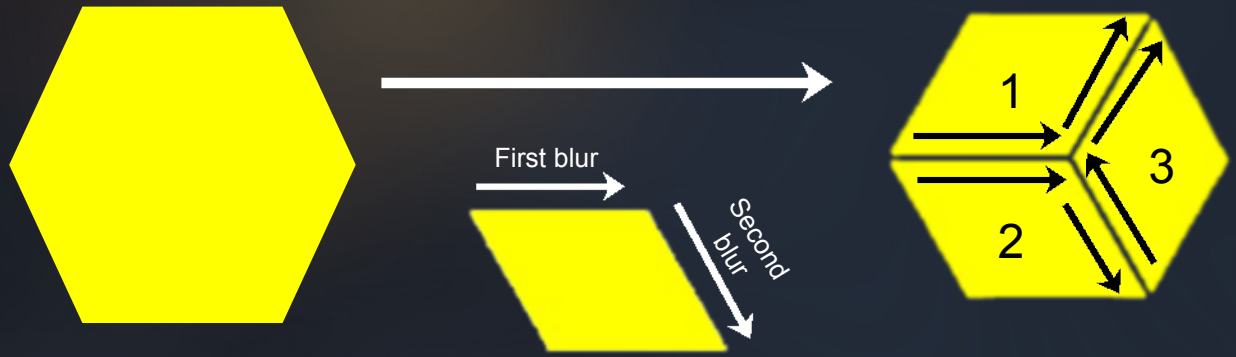
References:

http://de.slideshare.net/DICEStudio/five-rendering-ideas-from-battlefield-3-need-for-speed-the-run

http://www.polygonpi.com/?p=867

Hexagonal blur

- A hexagon can be decomposed into 3 rhombi (skewed 2D rects).
- Each rhombi can be separable blurred.
- 7 total passes, 3 shapes with each 2 blurs and 1 combine
- But the passes can be recuded further, see next slides.



Hexagonal blurs

- The trick: Use multiple render targets.
  1. The first pass is bluring upwards but also down and left at 120° at the same time intto two output frame buffer object MRT targets.
  2. The second pass combines the two MRT textures from the last into the final hexagonal blur, where the first texture (vertical blur) will be blurred down and left at 120° to make a rhombus (upper left third of the hexagon), and the the second MRT texture (vertical plus diagonal blur) down and right at 120° to make the other two thirds of the hexagon.
  3. The third pass combines both of the last two passes together and divide by three, since each individual blur preserves the total brightness, but the final pass adds them three times.
- From 7 passes to just 3 passes!

Hexagonal blur

```
float doGatherAndApply(const in sampler2D pTexSource, const in vec2 pUV, const in float pBaseCoC,
                       const in float pStepDistance, const in vec2 pInverseTextureSize, inout vec4 pColor){
  vec4 lColor = textureLod(pTexSource, pUV, 0.0);
  bool lBlurNear = lColor.w < 0.0;
  float lAbsoluteCoC = abs(lColor.w), lSampleFraction = 0.0;
  if((lAbsoluteCoC > pStepDistance) && (lBlurNear || ((pBaseCoC > 0.0) && (lAbsoluteCoC < (pBaseCoC * 2.0))))){
    if(lBlurNear){
      if(pColor.w < 0.0){
        pColor.w = min(pColor.w, lColor.w);
      }else if((-lColor.w) > pColor.w){
        pColor.w = lColor.w;
      }
    }
    lSampleFraction = clamp((lAbsoluteCoC - pStepDistance) * pInverseTextureSize.y, 0.0, 1.0);
    pColor.xyz += lColor.xyz * lSampleFraction;
  }
  return lSampleFraction;
}
```

Hexagonal blur
blur pass 1

```
#version 400
uniform sampler2D uTexSource;
in vec2 vTexCoord;
layout(location=0) out vec4 oOutputA;
layout(location=1) out vec4 oOutputB;
// doGatherAndApply
void main(){
  vec2 lTextureSize = vec2(textureSize(uTexSource, 0)), lInverseTextureSize = vec2(1.0) / lTextureSize;
  vec4 lBaseColor = textureLod(uTexSource, vTexCoord, 0.0);
  vec4 lOutputA = vec4(vec3(0.0), lBaseColor.w), lOutputB = lOutputA;
  float lSampleCountA = 0.0, lSampleCountB = 0.0, lStepX = 0.866 * (lInverseTextureSize.x / lInverseTextureSize.y);
  for(int i = 0; i < 8; i++){
    float lStepDistance = (float(i) + 0.5) * lInverseTextureSize.y;
    lSampleCountA += doGatherAndApply(uTexSource, vTexCoord + (vec2(0.0, 1.0) * lStepDistance), lBaseColor.w, lStepDistance,
                                      lInverseTextureSize, lOutputA);     // Vertical blur.
    lSampleCountB += doGatherAndApply(uTexSource, vTexCoord + (vec2(lStepX, -0.5) * lStepDistance), lBaseColor.w, lStepDistance,
                                      lInverseTextureSize, lOutputB); // Diagonal blur.
  }
  lOutputA.xyz = (lSampleCountA > 0.0) ? (lOutputA.xyz / lSampleCountA) : lBaseColor.xyz;
  lOutputB.xyz = (lSampleCountB > 0.0) ? (lOutputB.xyz / lSampleCountB) : lBaseColor.xyz;
  lOutputB.xyz += lOutputA.xyz;
  if(abs(lOutputA.w) > abs(lOutputB.w)){
    lOutputB.w = lOutputA.w;
  }
  oOutputA = lOutputA;
  oOutputB = lOutputB;
}
```

Hexagonal blur
blur pass 2

```
#version 400
uniform sampler2D uTexSourceA, uTexSourceB;
in vec2 vTexCoord;
layout(location=0) out vec4 oOutput;
// doGatherAndApply
void main(){
  vec2 lTextureSize = vec2(textureSize(uTexSourceA, 0)), lInverseTextureSize = vec2(1.0) / lTextureSize;
  vec4 lBaseColor = textureLod(uTexSourceB, vTexCoord, 0.0), lColorA = vec4(vec3(0.0), lBaseColor.w), lColorB = lColorA;
  float lSampleCountA = 0.0, lSampleCountB = 0.0, lStepX = 0.866 * (lInverseTextureSize.x / lInverseTextureSize.y);
  vec2 lStepA = vec2(lStepX, -0.5), lStepB = vec2(-lStepX, -0.5);
  for (int i = 0; i < 8; i++){
    float lStepDistance = (float(i) + 0.5) * lInverseTextureSize.y;
    lSampleCountA += doGatherAndApply(uTexSourceA, vTexCoord + (lStepA * lStepDistance), lBaseColor.w, lStepDistance, lInverseTextureSize,
                                      lColorA);
    lSampleCountB += doGatherAndApply(uTexSourceB, vTexCoord + (lStepB * lStepDistance), lBaseColor.w, lStepDistance, lInverseTextureSize,
                                      lColorB);
  }
  lColorA.xyz = (lSampleCountA > 0.0) ? (lColorA.xyz / lSampleCountA) : (lBaseColor.xyz * 0.5);
  lColorB.xyz = (lSampleCountB > 0.0) ? (lColorB.xyz / lSampleCountB) : lBaseColor.xyz;
  oOutput = vec4(vec3((lColorA.xyz + lColorB.xyz) / 3.0), max(abs(lColorA.w), abs(lColorB.w)));
}
```

Hexagonal blur
blur pass 3

```
#version 400
uniform sampler2D uTexSourceA, uTexSourceB;
in vec2 vTexCoord;
layout(location=0) out vec4 oOutput;
void main(){
  float lFringe = 0.7; // bokeh chromatic aberration/fringing
  vec2 lTextureSize = vec2(textureSize(uTexSourceB, 0)), lInverseTextureSize = vec2(1.0) / lTextureSize;
  vec4 lColorA = textureLod(uTexSourceA, vTexCoord, 0.0), lColorB = textureLod(uTexSourceB, vTexCoord, 0.0);
  float lFactor = clamp(1000.0 * (abs(lColorB.w) - 0.001), 0.0, 1.0);
  vec2 lChromaticAberrationFringeOffset = lInverseTextureSize * lFringe * lFactor;
  oOutput = (lFactor < 1e-6) ?
            lColorA :
            mix(lColorA,
                vec4(textureLod(uTexSourceB, vTexCoord + (vec2(0.0, 1.0) * lChromaticAberrationFringeOffset), 0.0).x,
                     textureLod(uTexSourceB, vTexCoord + (vec2(-0.866, -0.5) * lChromaticAberrationFringeOffset), 0.0).y,
                     textureLod(uTexSourceB, vTexCoord + (vec2(0.866, -0.5) * lChromaticAberrationFringeOffset), 0.0).z,
                     lColorA.w),
                lFactor);
}
```

Depth of field

Second the variant from the 64k from yesterday

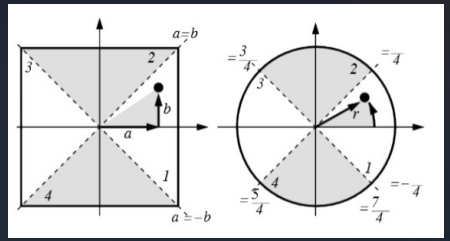
References:

http://www.crytek.com/download/Sousa_Graphics_Gems_CryENGINE3.pdf on slide 36

Depth of field

- Concentric Mapping is used for uniform sample distribution
  - It maps a unit square to a unit circle
  - Square is mapped to (a,b) $[-1,1]^2$ and divided into 4 regions by lines a=b, a=-b
  - First region is:

$$r = \alpha$$
$$\theta = \frac{\pi \cdot b}{4 \cdot \alpha}$$



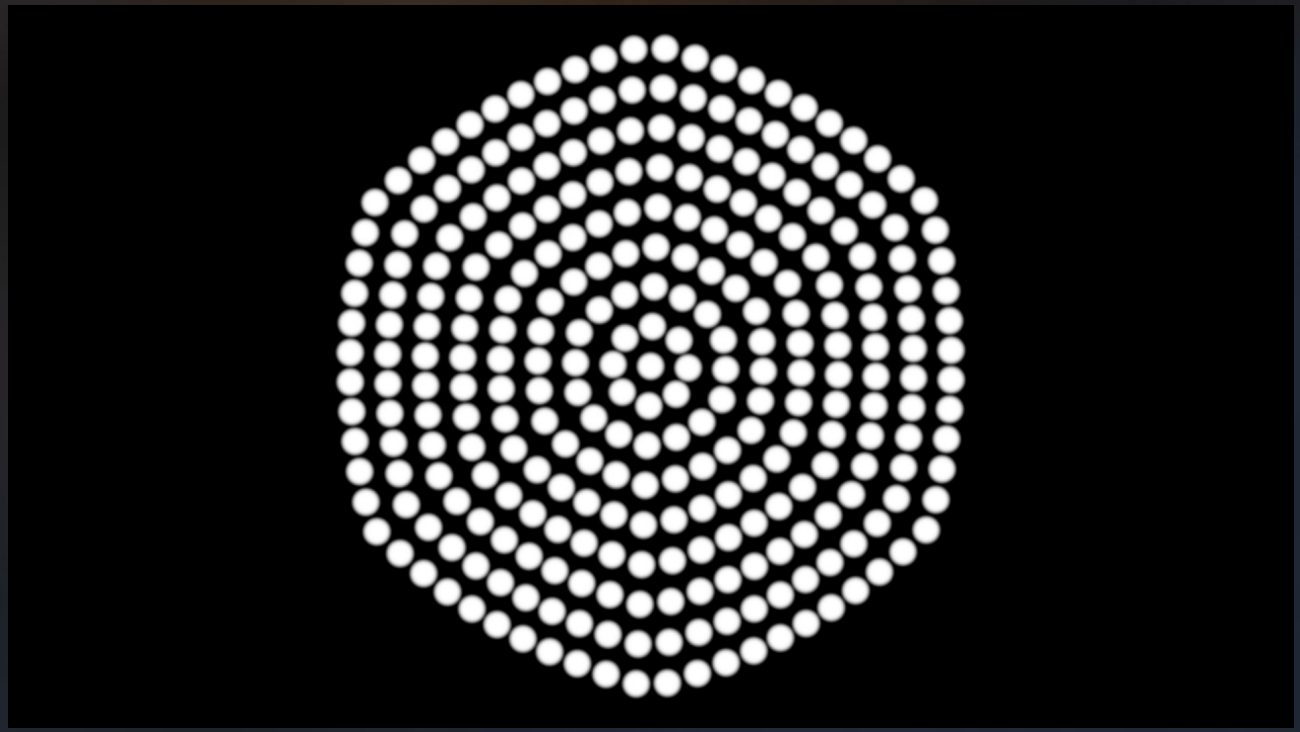
- Diaphragm simulation by morphing samples to n-gons

$$f = \frac{f_{stops} - f_{stops\_min}}{f_{stops\_max} - f_{stops\_min}}$$
$$\theta = \theta + f \cdot \theta_{shutter\_max}$$
$$r_{gnon} = r \cdot \left(\frac{\cos\left(\frac{\pi}{n}\right)}{\cos\left(\theta - \left(\left(\frac{2 \cdot \pi}{n}\right) \cdot floor\left(\frac{(n \cdot \theta) + \pi}{2 * \pi}\right)\right)\right)} \right)^{f}$$

Depth of field

```
const float PI = 3.14159265359;

// o = tap sample xy, f = f-factor, n = diaphragm shape ngon, phiShutterMax = max. lens shutter rotation
vec2 getBokehTapSampleCoord(const in vec2 o, const in float f, const float n, const in float phiShutterMax){
  vec2 ab = (o * 2.0) - vec2(1.0);
  vec2 phir = ((ab.x * ab.x) > (ab.y * ab.y)) ?
                vec2((abs(ab.x) > 1e-8) ? ((PI * 0.25) * (ab.y / ab.x)) : 0.0, ab.x) :
                vec2((abs(ab.y) > 1e-8) ? ((PI * 0.5) - ((PI * 0.25) * (ab.x / ab.y))) : 0.0, ab.y);
  phir.x += f * phiShutterMax;
  phir.y *= (f > 0.0) ?
              pow((cos(PI / n) / cos(phir.x - ((2.0 * (PI / n)) * floor(((n * phir.x) + PI) / (2.0 * PI))))), f) :
              1.0;
  return vec2(cos(phir.x), sin(phir.x)) * phir.y;
}
```

Depth of field



Depth of field
Code (1/2)

```
#version 150
varying vec2 vTexCoord;
uniform float uFStop, uFFactor, uShapeNGon, uChromaticAberration, uMaxTaps;
uniform sampler2D fboDepthBuffer, fboColotBuffer;
float fstop = max(1.0, uFStop * 64.0); // f-stop value
float CoC = 0.03; // circle of confusion size in mm (35mm film = 0.03mm)
float threshold = 0.5; //highlight threshold;
float gain = 2.0; //highlight gain;
float bias = 0.5; //bokeh edge bias
float fringe = 0.7; //bokeh chromatic aberration/fringing
vec4 sampleIt(vec2 coords){
  return max(vec4(0.0), texture(fboColotBuffer, coords));
}
const float PI = 3.14159265359;
// o = tap sample xy, f = f-factor, n = diaphragm shape ngon, phiShutterMax = max. lens shutter rotation
vec2 getBokehTapSampleCoord(const in vec2 o, const in float f, const float n, const in float phiShutterMax){
    vec2 ab = (o * 2.0) - vec2(1.0);
    vec2 phir = ((ab.x * ab.x) > (ab.y * ab.y)) ? vec2((abs(ab.x) > 1e-8) ?
                  ((PI * 0.25) * (ab.y / ab.x)) : 0.0, ab.x) : vec2((abs(ab.y) > 1e-8) ?
                  ((PI * 0.5) - ((PI * 0.25) * (ab.x / ab.y))) : 0.0, ab.y);
    phir.x += f * phiShutterMax;
    phir.y *= (f > 0.0) ?
                pow((cos(PI / n) / cos(phir.x - ((2.0 * (PI / n)) * floor(((n * phir.x) + PI) / (2.0 * PI))))), f) :
                1.0;
    return vec2(cos(phir.x), sin(phir.x)) * phir.y;
}
```

Depth of field
Code (2/2)

```
void main(){
  float f = 35.0; // focal length in mmtexture
  float d = texture(fboDepthBuffer, vec2(0.5)).w * 1000.0; // focal plane in mm (here: depth from middle => auto-focus)
  float o = texture(fboDepthBuffer, vTexCoord).w * 1000.0; // depth in mm
//float a = (o * f) / (o - f), b = (d * f) / (d - f), c = (d - f) / (d * fstop * CoC), blur = abs(a - b) * c;
  // Simplifed formula expression from the stuff by above:
  float blur = (abs(((f * f) * (d - o)) / ((d - f) * (o - f))) * (d - f)) / (d * fstop * CoC);
  vec4 c = sampleIt(vTexCoord);
  vec2 uCanvasInvSize = vec2(1.0) / vec2(textureSize(fboColotBuffer, 0).xy);
  vec2 wh = uCanvasInvSize * blur; // +noise.xy;
  if (length(wh) > 0.0005) {
    vec3 s = vec3(1.0);
    vec2 si = vec2(ceil(clamp(blur * 3.0, 2.0, clamp(uMaxTaps * 255.0, 2.0, 16.0)))), sio = si + vec2(1.0), sii = vec2(1.0) / si;
    for(float sy = 0.0; sy < sio.y; sy += 1.0){
      for(float sx = 0.0; sx < sio.x; sx += 1.0){
        vec2 pwh = getBokehTapSampleCoord(vec2(sx, sy) * sii, uFFactor * 1.0, max(3.0, uShapeNGon * 255.0), PI * 0.5);
        vec2 coords = vTexCoord + (pwh * wh);
        vec2 t = uCanvasInvSize * fringe * blur;
        vec3 col = sampleIt(coords).xyz;
        if(uChromaticAberration > 1e-6){
          col = mix(col, vec3(sampleIt(coords + vec2(0.0, 1.0) * t).x, sampleIt(coords + vec2(-0.866, -0.5) * t).y, sampleIt(coords +
                        vec2(0.866, -0.5) * t).z), uChromaticAberration);
        }
        vec3 b = (vec3(1.0) + mix(vec3(0.0), col, max((dot(col, vec3(0.299,0.587,0.114)) - threshold) * gain, 0.0) * blur));
        c.xyz += col * b;
        s += b;
      }
    }
    c.xyz /= s;
  }
  gl_FragColor = c;
}
```

Screen space ambient occlusion

We are skipping it here, since you can find SSAO-related stuff in the internet on mass

Order independent transparency

how it can done in a effective way

Order independent transparency

Supraleiter uses a hybrid solution based on 32-bit Atomic Loop Order Independent Transparency and Weight Blended Order Independent Transparency

32-bit Atomic Loop Order Independent Transparency

- It is a two pass record algorithm
  - First pass: Find K closest depth values

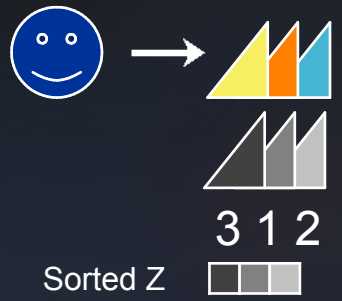

```
uniform int uOrderIndependentTransparencyWidth, uOrderIndependentTransparencyLayers;
layout(std430) coherent buffer orderIndependentTransparencyDepthBufferData {
  uint orderIndependentTransparencyDepthBuffer[];
};

uint lFragDepth = floatBitsToUint(gl_FragCoord.z);
int lBufferIndex = ((int(gl_FragCoord.y) * uOrderIndependentTransparencyWidth) +
                    int(gl_FragCoord.x)) * uOrderIndependentTransparencyLayers;

// for speed-up test against last/middle element of list
// if too far, skip below or start at middle
for(int lLayerIndex = 0; lLayerIndex < uOrderIndependentTransparencyLayers; lLayerIndex++){
  uint lOldDepth = atomicMin(orderIndependentTransparencyDepthBuffer[lBufferIndex++], lFragDepth);
  if((lOldDepth == 0xffffffffu) || (lOldDepth == lFragDepth)){
    break;
  }
  lFragDepth = max(lOldDepth, lFragDepth);
}
```

32-bit Atomic Loop Order Independent Transparency

- Second pass
  - Insert color based on depth with binary seatch
  - Tail blend is stable (primitive-order obeyed)
- Resolve
  - The list is simply already sorted

32-bit Atomic Loop Order Independent Transparency

```
uniform int uOrderIndependentTransparencyWidth, uOrderIndependentTransparencyLayers;
layout(std430) coherent buffer orderIndependentTransparencyDepthBufferData {
  uint orderIndependentTransparencyDepthBuffer[];
};
layout(std430) coherent buffer orderIndependentTransparencyColorBufferData {
  vec4 orderIndependentTransparencyColorBuffer[];
};
uint lFragDepth = floatBitsToUint(gl_FragCoord.z);
int lBufferIndex = ((int(gl_FragCoord.y) * uOrderIndependentTransparencyWidth) +
                    int(gl_FragCoord.x)) * uOrderIndependentTransparencyLayers;
// Binary search (the followed code could be more better optimized, when it is needed)
int lLow = 0, lHigh = uOrderIndependentTransparencyLayers - 1;
while(lLow < lHigh){
  int lMiddle = lLow + ((lHigh - lLow) >> 1);
  uint lTestDepth = orderIndependentTransparencyDepthBuffer[lBufferIndex + lMiddle];
  if(lFragDepth > lTestDepth){
    lLow = lMiddle + 1;
  }else if(lFragDepth < lTestDepth){
    lHigh = lMiddle - 1;
  }else{
    lLow = -1;
    orderIndependentTransparencyColorBuffer[lBufferIndex + lMiddle] = pAlphaBlendingColor;
    break;
  }
}
if(lLow >= 0){
  doTailBlend();
}
```



And now let’s combine 32-bit Atomic Loop OIT with Weight Blended OIT

The followwed code is also available at:
https://gist.github.com/BeRo1985/80fbcdf16a9809c84c7b

And what WBOIT is, see:
http://jcgt.org/published/0002/02/09/
casual-effects.blogspot.de/2014/03/weighted-blended-order-independent.html
http://casual-effects.blogspot.de/2015/03/implemented-weighted-blended-order.html
http://casual-effects.blogspot.de/2015/03/colored-blended-order-independent.html

```
// Hybrid atomic loop weighted blended order independent transparency implementation (work in progress)
// Copyright (C) 2014 by Benjamin 'BeRo' Rosseaux
// Licensed under the CC0 license, since in the German legislation exists no public
// domain.

// This implementation needs at least OpenGL 4.3 as minimum OpenGL version, due to my usage of shader storage buffer objects here

// Supports also additive blending for emission color portions

uniform sampler2D uTexTailWeightedBlendedOrderIndependentTransparencyColor;
uniform sampler2D uTexTailWeightedBlendedOrderIndependentTransparencyAlpha;

uniform int uOrderIndependentTransparencyWidth;
uniform int uOrderIndependentTransparencyLayers;

layout(std430) coherent buffer orderIndependentTransparencyDepthBufferData {
  uint orderIndependentTransparencyDepthBuffer[];
};

layout(std430) buffer orderIndependentTransparencyColorBufferData {
  uvec4 orderIndependentTransparencyColorBuffer[]; // Two packed RGBA16F values: xy = alpha blended color and zw = emission color
};

int oitGetCoordBaseBufferIndex(){
  return ((int(gl_FragCoord.y) * uOrderIndependentTransparencyWidth) + int(gl_FragCoord.x)) * uOrderIndependentTransparencyLayers;
}

// For one-way clearing at after depth buffer creation, when it shouldn't happen for whatever reason on the CPU itself
void oitClear(){
  int lBufferIndex = oitGetCoordBaseBufferIndex();
  for(int lLayerIndex = 0; lLayerIndex < uOrderIndependentTransparencyLayers; lLayerIndex++){
    orderIndependentTransparencyDepthBuffer[lBufferIndex++] = 0xffffffffu;
  }
}
```

```
// Optional: Pass "zero" with alpha test against alpha value 1.0 for better weighted blended order independent transparency portion results

// First pass for finding K closest depth values
void oitPassOne(){
  uint lFragDepth = floatBitsToUint(gl_FragCoord.z);
  int lBufferIndex = oitGetCoordBaseBufferIndex();
  for(int lLayerIndex = 0; lLayerIndex < uOrderIndependentTransparencyLayers; lLayerIndex++){
    uint lOldDepth = atomicMin(orderIndependentTransparencyDepthBuffer[lBufferIndex++], lFragDepth);
    if((lOldDepth == 0xffffffffu) || (lOldDepth == lFragDepth)){
      break;
    }
    lFragDepth = max(lOldDepth, lFragDepth);
  }
}

// Second pass with fragment color recording and tail weighted blended order independent transparency blending
void oitPassTwo(const in vec4 pAlphaBlendingColor, const in vec4 pEmissionColor, const in float pViewSpaceZ,
                out vec4 pTailWeightedBlendedOrderIndependentTransparencyColor,
                out float pTailWeightedBlendedOrderIndependentTransparencyAlpha){
  uint lFragDepth = floatBitsToUint(gl_FragCoord.z);
  int lBufferIndex = oitGetCoordBaseBufferIndex();

  // Binary search (the followed code could be more better optimized, when it is needed)
  int lLow = 0;
  int lHigh = uOrderIndependentTransparencyLayers - 1;
  while(lLow < lHigh){
    int lMiddle = lLow + ((lHigh - lLow) >> 1);
    uint lTestDepth = orderIndependentTransparencyDepthBuffer[lBufferIndex + lMiddle];
    if(lFragDepth > lTestDepth){
      lLow = lMiddle + 1;
    }else if(lFragDepth < lTestDepth){
      lHigh = lMiddle - 1;
    }else{
      lLow = -1;
      orderIndependentTransparencyColorBuffer[lBufferIndex + lMiddle] = uvec4(uvec2(packHalf2x16(pAlphaBlendingColor.rg),
                                                                                    packHalf2x16(pAlphaBlendingColor.ba)),
```

```
                                                                    uvec2(packHalf2x16(pEmissionColor.rg),
                                                                          packHalf2x16(pEmissionColor.ba)));
    break;
  }
}

// Tail weighted blended order independent transparency blending
if(lLow < 0){
  // Fragment was recorded, so we must do no tail weighted blended order independent transparency blending here then
  pTailWeightedBlendedOrderIndependentTransparencyColor = vec4(0.0);
  pTailWeightedBlendedOrderIndependentTransparencyAlpha = 0.0;
}else{
  // Fragment was not recorded, because it is behind the recorded last layer depth, so we must do tail weighted blended order independent
  // transparency blending here then
  if(dot(pAlphaBlendingColor, pAlphaBlendingColor) > 0.0){
    // Alpha blending if alpha value is greater or equal zero (positive)
    float pTailWeightedBlendedOrderIndependentTransparencyWeight = max(min(1.0, max(max(pAlphaBlendingColor.r, pAlphaBlendingColor.g),
                                                                                     pAlphaBlendingColor.b) * pAlphaBlendingColor.a),
                                                                       pAlphaBlendingColor.a) * clamp(0.03 / (1e-5 + pow(pViewSpaceZ /
                                                                                                                         4096.0, 4.0)), 1e-2, 3e3);
    pTailWeightedBlendedOrderIndependentTransparencyColor = vec4(pAlphaBlendingColor.rgb * pAlphaBlendingColor.a, pAlphaBlendingColor.a) *
                                                            pTailWeightedBlendedOrderIndependentTransparencyWeight;
    pTailWeightedBlendedOrderIndependentTransparencyAlpha = pAlphaBlendingColor.a;
  }
  if(dot(pEmissionColor, pEmissionColor) > 0.0){
    // Fake incorrect but in the most cases still acceptable additive-like blending with weighted blended order independent transparency, if
    // alpha value is less than zero (negative)
    // This value must not be zero, because the additive blending like effect is disappearing then,
    // but also not too high, since it is used also as the alpha value for the blending itself.
    // The base resolve step will do the following then:
    // (firstTexel.rgb / firstTexel.a) * secondTexel.r where firstTexel.a and secondTexel.r are this one rescale weight value
    const float pTailWeightedBlendedOrderIndependentTransparencyAdditiveBlendingRescaleWeight = 1e-3;
    pTailWeightedBlendedOrderIndependentTransparencyColor = vec4(pEmissionColor.rgb,
                                                            pTailWeightedBlendedOrderIndependentTransparencyAdditiveBlendingRescaleWeight);
    pTailWeightedBlendedOrderIndependentTransparencyAlpha = pTailWeightedBlendedOrderIndependentTransparencyAdditiveBlendingRescaleWeight;
  }
}
```

```
}

// Resolve the recorded and collected stuff
// Needs active blending with GL_ONE, GL_ONE_MINUS_SRC_ALPHA => dstcolor = srccolor + (dstcolor * (1.0 - srcalpha))
// since the front-to-back layer has rendered over black, so it is pre-multiplied by its alpha value
vec4 oitResolve(){

  ivec2 lFragCoord = ivec2(gl_FragCoord.xy);

  // 1. Clear to translucent black (0.0, 0.0, 0.0, 0.0)
  // The black color meaning nothing to add color-wise, and the transparent alpha value of 0.0 meaning that the background won't be occluded
  vec4 lOutputColor = vec4(0.0);

  // 2. Resolve recorded fragments per front-to-back blending with the UNDER blending operator
  int lBufferIndex = oitGetCoordBaseBufferIndex();
  for(int lLayerIndex = 0; lLayerIndex < uOrderIndependentTransparencyLayers; lLayerIndex++){
    // Get array cell value and clear array cell at the same time
    uint lDepth = atomicExchange(orderIndependentTransparencyDepthBuffer[lBufferIndex], 0xffffffffu);
    if(lDepth == 0xffffffffu){
      break;
    }
    uvec4 lBufferPackedColor = orderIndependentTransparencyColorBuffer[lBufferIndex++];
    vec4 lBufferAlphaBlendingColor = vec4(unpackHalf2x16(lBufferPackedColor.x), unpackHalf2x16(lBufferPackedColor.y));
    vec4 lBufferEmissionColor = vec4(unpackHalf2x16(lBufferPackedColor.z), unpackHalf2x16(lBufferPackedColor.w));
    // Emulating GL_ONE_MINUS_DST_ALPHA, GL_ONE => dstcolor += srccolor * (1.0 - dstalpha)
    lOutputColor += (vec4(lBufferAlphaBlendingColor.rgb * lBufferAlphaBlendingColor.a, lBufferAlphaBlendingColor.a) + lBufferEmissionColor) *
                    (1.0 - lOutputColor.a);
  }

  // 3. Blend the weighted blended order independent transparency portion top on the opaque color layer
  {
    vec4 lTailWeightedBlendedOrderIndependentTransparencyColor = texelFetch(uTexTailWeightedBlendedOrderIndependentTransparencyColor,
                                                                            lFragCoord, 0);
    float lTailWeightedBlendedOrderIndependentTransparencyAlpha = texelFetch(uTexTailWeightedBlendedOrderIndependentTransparencyAlpha,
                                                                             lFragCoord, 0).x;
```

```
    lTailWeightedBlendedOrderIndependentTransparencyColor = vec4((lTailWeightedBlendedOrderIndependentTransparencyColor.rgb /
                                                                  max(lTailWeightedBlendedOrderIndependentTransparencyColor.a, 1e-5)),
                                                                      lTailWeightedBlendedOrderIndependentTransparencyAlpha);

    // Compositing by emulating GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA => dstcolor = (srccolor * (1.0 - srcalpha)) + (dstcolor * srcalpha)
    // inverted for front-to-back compositing GL_ONE_MINUS_SRC_ALPHA * GL_ONE_MINUS_DST_ALPHA, GL_ONE => dstcolor += (srccolor * (1.0 -
    // srcalpha)) * (1.0 - dstalpha)
    // TO-DO: Check if it is correct in this way
    lOutputColor += (lTailWeightedBlendedOrderIndependentTransparencyColor * (1.0 - lTailWeightedBlendedOrderIndependentTransparencyColor.a))
                    * (1.0 - lOutputColor.a);
  }

  // Return the final color
  return lOutputColor;
}
```

You can also replace there the Weight Blended Order Independent Transparency stuff with the newer Color Blended Order Independent Transparency stuff from

http://casual-effects.blogspot.de/2015/03/colored-blended-order-independent.html

Antialiasing

- Supraleiter has support for:
  - No antialiasing
  - FXAA
  - SMAA
    - SMAA1x
    - SMAAT2x (default antialiasing mode at supraleiter)
      - which is SMAA1x with alternative frame jitter and temporal alternative frame blending
- For more details use the search engine of your choice :-)

And now the really interesting stuff

Sample Distribution Shadow Maps

http://visual-computing.intel-research.net/art/publications/sdsm/

Sample Distribution Shadow Maps

SDSM is built on Cascaded Shadow Maps where the Z-Partitioning of the shadow map cascades is automated by scanning the camera view depth buffer for example with a compute shader.

Known previous Z-Partioning schemes

- Logarithmic is best [Lloyd et al. 2006]
  - But only if the entire Z range is covered!
  - Needs tight near/far planes
- Parallel-Split Shadow Maps [Zhang et al. 2006]
  - Mix of logarithmic and uniform
  - Requires user to tune a parameter
    - Optimal value related to tight near plane…
- In practice, artists tunes CSM/PSSM parameters for specific views
  - Annoying, irritating, etc.
  - Not robust to scene and camera view changes
  - Often very suboptimal

Brandon Lloyd, David Tuft, Sung-eui Yoon, and Dinesh Manocha. Warping and Partitioning for Low Error Shadow Maps. In *Proc. Eurographics Symposium on Rendering 2006*.

Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-Split Shadow Maps for Large-Scale Virtual Environments. In *Virtual Reality Continuum And Its Applications 2006*.

Sample Distribution Shadow Maps

(the Supraleiter variant)

- Analyze the shadow sample distribution
  - Find tight minimum Z and maximum Z values
  - Partition logarithmically based on tight Z bounds
  - Adapts to view and geometry with no need for tuning
- Compute tight light-space bounds
  - Tight axis-aligned bound box per cascade partition
  - Greatly increases useful shadow resolution
- Use cascades blending at shadow map sampling
- Use Guassian-filtered PCF at shadow map sampling
  - Idea based on http://the-witness.net/news/2013/09/shadow-mapping-summary-part-1/

Realtime hard shadow raytracing

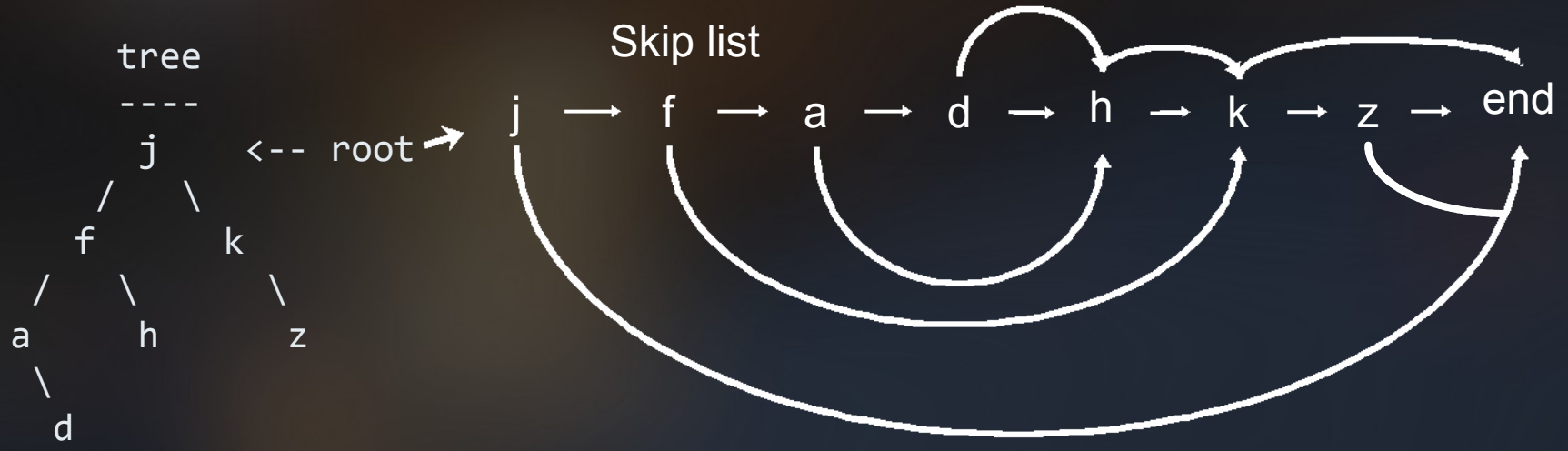
Realtime hard shadow raytracing

- The supraleiter engine has optional support for realtime hard shadow raytracing.
- All static objects have completely static BVH trees (and on GPU-side skip lists).
  - BVH is the fastest solution for my GPU hard shadow raytracing purpose, that I could find. I've tried sparse octrees, normal kd-trees, kd-trees with ropes, brick maps, etc.
- The static object BVH trees/skip-lists are in turn in a global dynamic self-balanced BVH tree/skip-list, so that the rigid body objects can move through the game world.
- The dynamic self-balanced BVH is just plain object AABB based, with best fit AABB volume split cost heuristics.
- All static and dynamic BVH data are stored in OpenGL 4.3 shader storage buffer objects on the GPU-side.
- The BVH trees are converted to skip-lists with a depth-first tree preorder traversal, because skip-lists can be traversaled stack-free but with the more or less same efficiently like trees.

Realtime hard shadow raytracing

- The BVH is built with
  - Sweep triangle-AABB-wise surface area heuristics on low tree depths
  - Bruteforce sweep triangle-vertex-wise surface area heuristics on high tree depths
- Each triangle is only in a single BVH node, even if the current data structure has indirect triangle indices for future triangle deduplication usages.
- The BVH is converted then when constructed to a skip list, for details see next slide.

Realtime hard shadow raytracing



A preorder tree traversal would visit the elements in the order: j, f, a, d, h, k, z. This would be also the skip list item order.

```
struct staticBVHSkipListTriangleVertex {
  vec4 position;
  vec4 normal;
  vec4 tangent;
  vec4 texCoord;
};

struct staticBVHSkipListTriangle {
  staticBVHSkipListTriangleVertex vertices[3];
  uint material;
  uint flags;
  uint tag;
  uint avoidSelfShadowingTag;
};

struct staticBVHSkipListNode {
  vec4 AABBs[2];
  uint flags;
  uint skipToNode;
  uint firstTriangleIndex;
  uint countTriangleIndices;
};

struct staticBVHSkipListObject {
  uint nodeOffset;
  uint countNodes;
  uint triangleOffset;
  uint countTriangles;
};

struct dynamicBVHSkipListNode {
  vec4 AABBs[2];
  uint skipToNode;
  uint proxyIndex;
  uint dummy0;
  uint dummy1;
};
```

```
struct dynamicBVHSkipListProxy {
  mat4 transformMatrix;
  mat4 inverseTransformMatrix;
  uint objectIndex;
  uint dummy0;
  uint dummy1;
  uint dummy2;
};

struct bvhIntersection {
  vec4 barycentric;
  vec4 position;
  vec4 normal;
  vec4 tangent;
  vec4 bitangent;
  vec4 texCoord;
  uint material;
  uint flags;
  uint objectIndex;
  uint tag;
};

uniform uint countDynamicBVHSkipListNodes;

layout(std430) buffer staticBVHTriangles {
  staticBVHSkipListTriangle staticBVHSkipListTriangles[];
};

layout(std430) buffer staticBVHTriangleIndices {
  uint staticBVHSkipListTriangleIndices[];
};

layout(std430) buffer staticBVHNodes {
  staticBVHSkipListNode staticBVHSkipListNodes[];
};

layout(std430) buffer staticBVHObjects {
  staticBVHSkipListObject staticBVHSkipListObjects[];
};
```

```
layout(std430) buffer dynamicBVHNodes {
    dynamicBVHSkipListNode dynamicBVHSkipListNodes[];
};

layout(std430) buffer dynamicBVHProxies {
    dynamicBVHSkipListProxy dynamicBVHSkipListProxies[];
};

bool bvhAABBIntersect(const in vec3 aabbMin,
                      const in vec3 aabbMax,
                      const in vec3 origin,
                      const in vec3 direction,
                      const in float maxDistance){
  /*if(all(lessThanEqual(origin, aabbMax)) &&
       all(greaterThanEqual(origin, aabbMin))){
    return true;
  }*/
  vec3 invDirection = 1.0 / direction;
  vec3 omin = (aabbMin - origin) * invDirection;
  vec3 omax = (aabbMax - origin) * invDirection;
  vec3 vmin = min(omin, omax);
  vec3 vmax = max(omin, omax);
  float tmin = max(vmin.x, max(vmin.y, vmin.z));
  float tmax = min(vmax.x, min(vmax.y, vmax.z));
  return ((tmin < maxDistance) &&
          (tmax >= tmin)) &&
         (tmax >= 0.0);
}
```

```
bool staticBVHIntersectTriangleForVisibilityCheck(const in staticBVHSkipListTriangle tri,
                                                  const in vec3 origin,
                                                  const in vec3 direction,
                                                  const in float minDistance,
                                                  const in float maxDistance) {
  const float EPSILON2 = 0.000001;

  // Get triangle edge vectors and plane normal
  vec3 u = tri.vertices[1].position.xyz - tri.vertices[0].position.xyz;
  vec3 v = tri.vertices[2].position.xyz - tri.vertices[0].position.xyz;
  vec3 n = cross(u, v);

  vec3 w0 = origin - tri.vertices[0].position.xyz;
  float a = -dot(n, w0);
  float b = dot(n, direction);

  // Check if ray is parallel to triangle plane.
  if (abs(b) < EPSILON2){
    return false;
  }

  // Get intersect point of ray with triangle plane
  float r = a / b;
  if(r < 0.0){
    // ray goes away from triangle
    return false;
  }

  // Intersect point of ray and plane
  vec3 intersectionPoint = origin + (r * direction);

  // Is I inside T?
  float uu = dot(u, u);
  float uv = dot(u, v);
  float vv = dot(v, v);
  vec3 w = intersectionPoint - tri.vertices[0].position.xyz;
  float wu = dot(u, w);
  float wv = dot(w, v);
  float D = (uv * uv) - (uu * vv);

  // Get and test parametric coords
  float s = ((uv * wv) - (vv * wu)) / D;
  if((s < -0.00001) || (s > 1.00001)){
      // I is outside T
      return false;
   }

  float t = ((uv * wu) - (uu * wv)) / D;
  if((t < -0.00001) || ((s + t) > 1.00001)){
     // I is outside T
     return false;
  }

  float d = length(intersectionPoint - origin);
  return (d >= minDistance) && (d <= maxDistance);
}
```

```
bool bvhRayShootForVisibilityCheck(const in vec3 origin,
                                   const in vec3 direction,
                                   const in float minDistance,
                                   const in float maxDistance,
                                   const in uint flags,
                                   const in bool doGlobal,
                                   const in bool doStaticOnly) {
  uint dynamicBVHNodeIndex = 0;
  uint dynamicBVHNodeStopIndex = countDynamicBVHSkipListNodes;
  while(dynamicBVHNodeIndex < dynamicBVHNodeStopIndex){
    if(bvhAABBIntersect(dynamicBVHSkipListNodes[dynamicBVHNodeIndex].AABBs[0].xyz, dynamicBVHSkipListNodes[dynamicBVHNodeIndex].AABBs[1].xyz, origin, direction, maxDistance)){
      uint proxyIndex = dynamicBVHSkipListNodes[dynamicBVHNodeIndex].proxyIndex;
      if(proxyIndex != 0xffffffffu){
        mat4 inverseTransformMatrix = dynamicBVHSkipListProxies[proxyIndex].inverseTransformMatrix;
        uint objectIndex = dynamicBVHSkipListProxies[proxyIndex].objectIndex;
        if(((objectIndex == 0u) && doGlobal) || ((objectIndex != 0u) && !doStaticOnly)){
          vec3 transformedOrigin = (inverseTransformMatrix * vec4(origin, 1.0)).xyz;
          vec3 transformedDirection = normalize(((inverseTransformMatrix * vec4(vec3(origin +direction), 1.0)).xyz) - transformedOrigin);
          uint staticBVHNodeIndex = staticBVHSkipListObjects[objectIndex].nodeOffset;
          uint staticBVHNodeStopIndex = staticBVHSkipListNodes[staticBVHNodeIndex].skipToNode;
          while(staticBVHNodeIndex < staticBVHNodeStopIndex){
           if(((staticBVHSkipListNodes[staticBVHNodeIndex].flags & flags) != 0u) &&
                bvhAABBIntersect(staticBVHSkipListNodes[staticBVHNodeIndex].AABBs[0].xyz, staticBVHSkipListNodes[staticBVHNodeIndex].AABBs[1].xyz, transformedOrigin, transformedDirection, maxDistance)){
            uint triangleIndex = staticBVHSkipListNodes[staticBVHNodeIndex].firstTriangleIndex;
            uint triangleStopIndex = triangleIndex + staticBVHSkipListNodes[staticBVHNodeIndex].countTriangleIndices;
            while(triangleIndex < triangleStopIndex){
              uint realTriangleIndex = staticBVHSkipListTriangleIndices[triangleIndex];
              if(((staticBVHSkipListTriangles[realTriangleIndex].flags & flags) != 0u) &&
                   staticBVHIntersectTriangleForVisibilityCheck(staticBVHSkipListTriangles[realTriangleIndex], transformedOrigin, transformedDirection, minDistance, maxDistance)){
                return true;
              }
              triangleIndex++;
            }
            staticBVHNodeIndex++;
           }else{
            // Skip this static BVH node and all its children
            staticBVHNodeIndex = max(staticBVHSkipListNodes[staticBVHNodeIndex].skipToNode, staticBVHNodeIndex + 1u);
           }
          }
        }
      }
      dynamicBVHNodeIndex++;
    }else{
      // Skip this dynamic BVH node and all its children
      dynamicBVHNodeIndex = max(dynamicBVHSkipListNodes[dynamicBVHNodeIndex].skipToNode, dynamicBVHNodeIndex + 1u);
    }
  }
  return false;
}
```

```
bool bvhIntersectsForVisibilityCheck(const in vec3 origin, const in vec3 target, const in float minDistance, const in uint flags, const in bool doGlobal, const in bool doStaticOnly){
    return bvhRayShootForVisibilityCheck(origin, normalize(target - origin), minDistance, length(target - origin), flags, doGlobal, doStaticOnly);
}

 sunLit = staticBVHRayShootForVisibilityCheck(vWorldSpacePosition + ((gNormal - uSunDirection) * 0.01), uSunDirection, 0.01, 16777216.0, 0xffffffffu, true, false) ? 0.0 : 1.0;
```

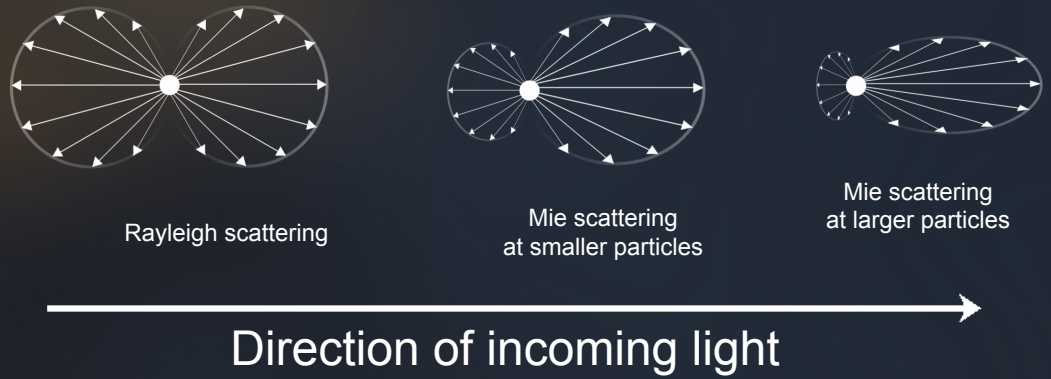
Supraleiter has also function variants of these, which returns the position, normal, texture coordinates, etc. of the ray hits, for future experiments with path tracing and so on, since the BVH data structures have already this data for it.

Procedural sky
Volumetric light scattering
Cloud rendering

Yeah, all this three things are actually the same thing.

Atmospheric scattering

- Atmospheric scattering can be seen as an extension of volume rendering.
- The sky is blue due to particles in the atmosphere scattering the light from the Sun, without these particles the sky would be pitch black even in the middle of the day and the Sun would even not appear larger than the Moon.
- Two common models used for describing the atmospheric scattering are called Rayleigh and Mie, which are named after their originators.



Rayleigh scattering

- If the particles are smaller than the wavelength of light, the Rayleigh scattering predominates.
- The intensity of the scattering due of these particles heavily dependent on the wavelength, proportional to $1/\lambda^4$, where λ is the wavelength.
- Rayleigh scattering which scatters the incoming light in all directions.
- Rayleigh scattering due to its high dependence on the wavelength is the main reason that the color of the sky is blue.

Releigh equation for to render the scattering coefficients:

$$\beta_R^8(h,\lambda) = \frac{8\pi^3\left(n^2-1\right)^2}{3N\lambda^4} e^{-\frac{h}{H_R}}$$

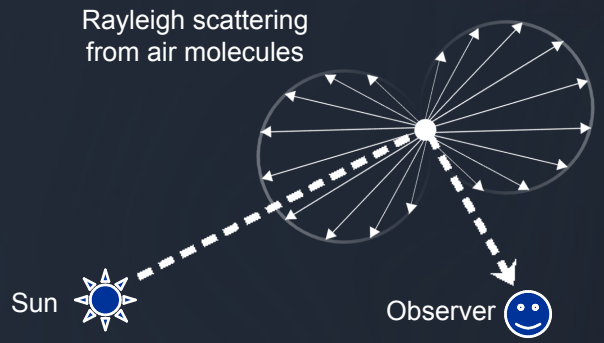
Extinction coefficient for volume rendering:

$$\beta_R^e = \beta_R^s$$

Releigh phase function equation:

$$P_r(\mu) = \frac{3}{16\pi}\left(1+\mu^2\right)$$

β = scattering coefficient
s = scattering
R = Rayleigh
λ = wavelength
N = molecular density
n = index of refraction of air
H_R = scale height
μ = cosine of the angle between the light and the view directions



Mie scattering

- If the particles are larger than the wavelength of light, Mie scattering dominates, for example at the clouds.
- In contrast to the Rayleigh scattering which scatters the incoming light in all directions, Mie scattering has a strong forward scattering in a narrow range.
- Mie scattering is also the main reason for the perceived size of the sun, because without this Mie scattering the sun would appear no bigger than the moon.

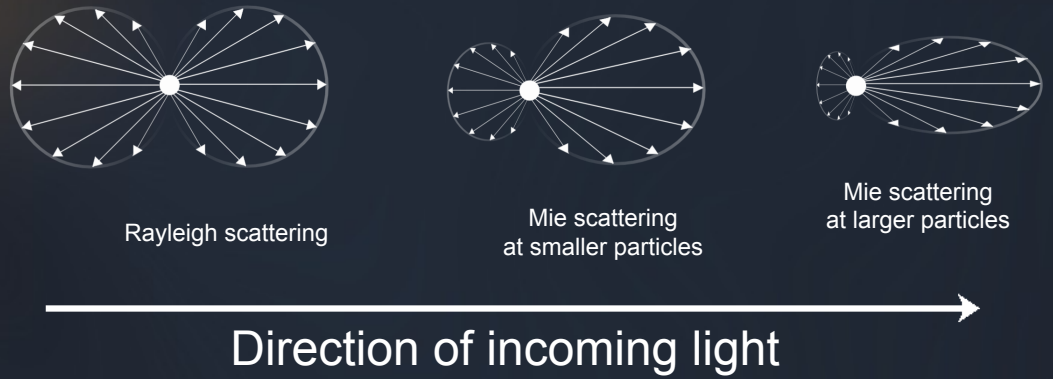
Mie equation for to render the scattering coefficients:

$$\beta_M^s(h,\lambda) = \beta_M^s(0,\lambda)e^{-\frac{h}{H_M}}$$

Mie phase function equation:

$$P_m(\mu) = \frac{3}{8\pi}\frac{(1-g^2)(1+\mu^2)}{(2+g^2)(1+g^2+2g\mu)^{\frac{3}{2}}}$$

β = scattering coefficient
s = scattering
M = Mie
λ = wavelength
H_M = scale height
μ = cosine of the angle between the light and the view directions
g = anisotropy of the medium



Atmospheric density

For the atmospheric density can we make the assumption that it decreases exponentially with height. The equation for it looks like:

$$density(h) = density(0)e^{-\frac{h}{H}}$$

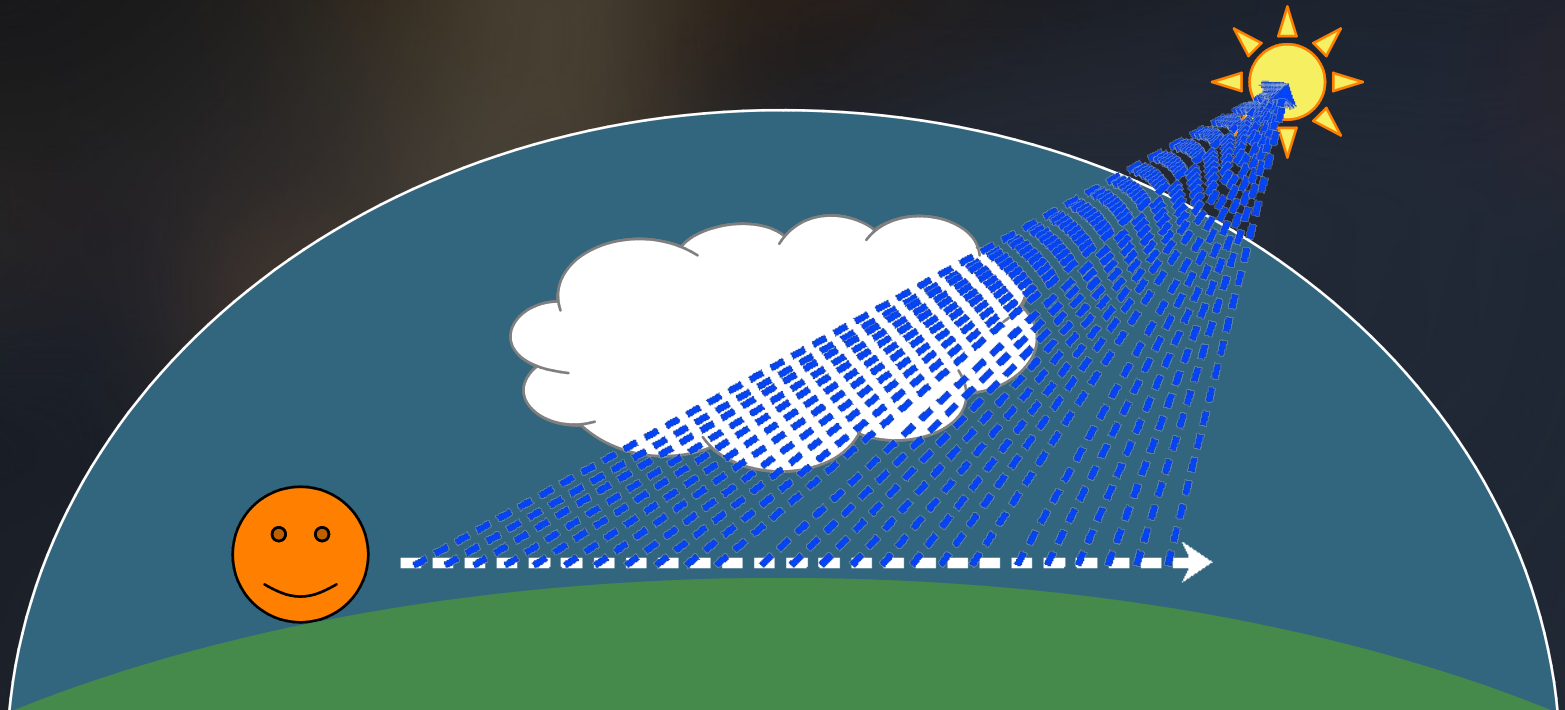
where
density(0) = air density at sea level
h = current height (altitude) where we measure the atmosphere density
H = thickness of the atmosphere if its density were uniform (also called scale height in scientific papers)

Typical constants for our earth

β_M	= Mie scattering coefficients at sea level	= r: 21^{-6} g: 21^{-6} b: 21^{-6}
β_R	= Rayleigh scattering coefficients at sea level	= r: 5.8^{-6} g: 13.5^{-6} b: 33.1^{-6}
R_E	= Earth radius	= 6360^3 meter
R_A	= Atmosphere radius from earth center	= 6380^3 meters
g	= Mean cosine	= 0.76
g2	= Mean cosine squared	= g^2
H_R	= Rayleigh scale height	= 8000
H^M	= Mie scale height	= 1800
I_S	= Sun intensity (just a rendering example value)	= 10
cloudBottom	= Minimum cloud height (also a example value)	= 5000 meters
cloudTop	= Maximum cloud height (also a example value)	= 15000 meters

Okay... But how can I do raymarching with it?

First, a picture that is worth a thousand words.

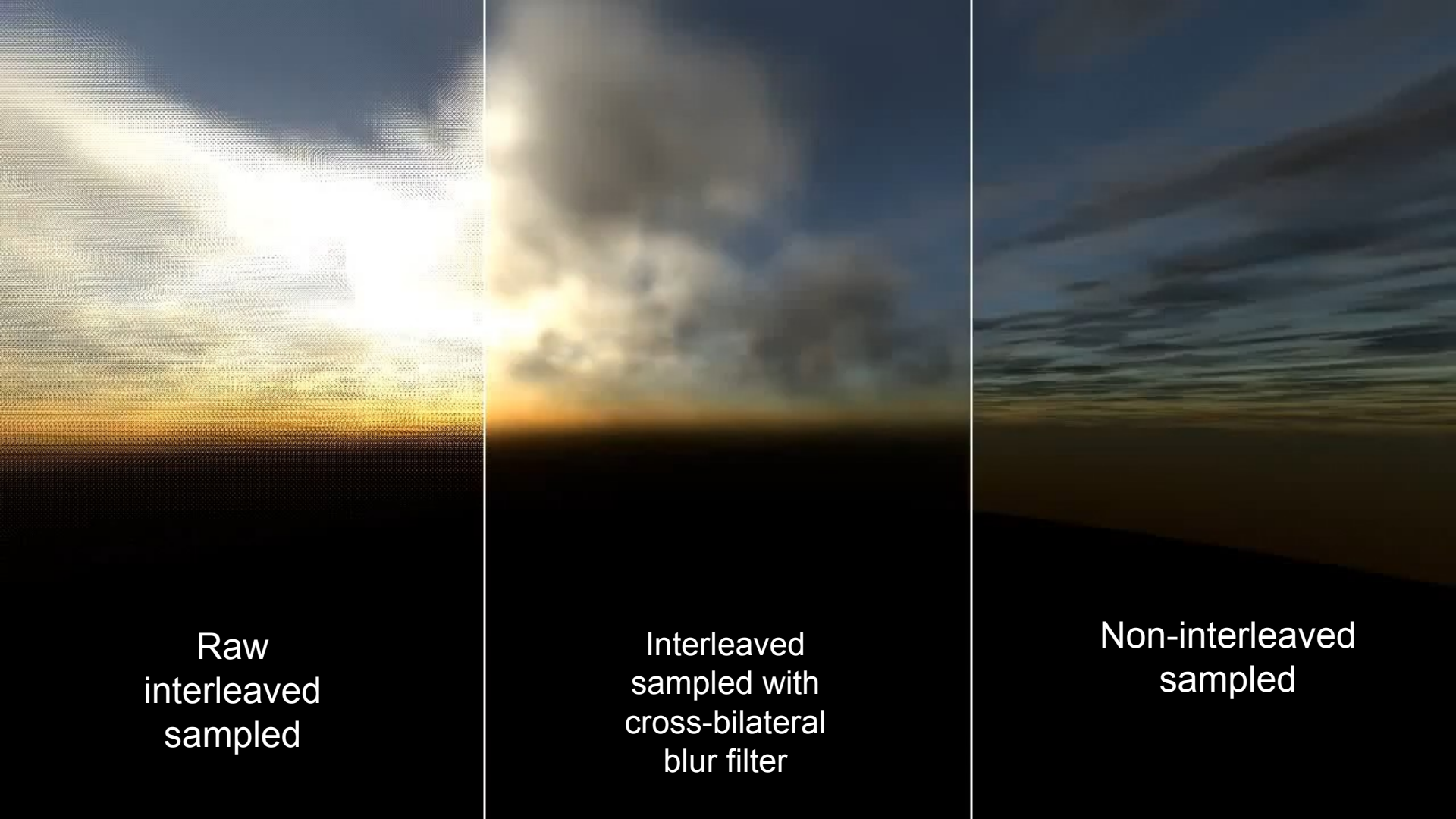


And in words?

- First find the outgoing sphere intersection hit point of your virtual earth sphere from your current camera view origin point into your current ray direction, then calculate the maximal ray length from it. And as volumetric light scattering effect variant the length of your ray stops simply on the respective value from your depth buffer, and if transparent objects are existent, then split your ray depth-peeling-style into multiple single main ray pieces.
- Then raymarching along this ray line with your maximal calculated ray length.
  - But for each main ray raymarching step, you starting a new raymarching subprocess with a ray from the current main ray step position into the direction to your current main light source (for example the sun and/or the moon)
    - Apply the scattering equations while this
      - Clouds are more or less simply density modulations at the Mie scattering equation
- Then apply the phase functions to the accumulated equation results for the current main raymaching ray, for to get the final color for the one current pixel
- You do want to see code, not only these words? Okay, see next slides :-)

But a tip before it

Use Bayer matrix interleaved sample ray offset jittering with a cross-bilateral blur filter, for visually effective more samples with less samples to be calculated.


Raw interleaved sampled
Interleaved sampled with cross-bilateral blur filter
Non-interleaved sampled

```
// Extremly stripped orbit scene code from the 64k from yesterday, which the code parts for the additional
// moon light source, handling of camera handling outer earth atmosphere (in space), object blending,
// water shading, procedural stars, etc. are removed here for code simplicity.

const float Re = 6360e3;    // Earth radius
const float Ra = 6380e3;    // Atmosphere radius
const float g = 0.76;       // Mean cosine
const float g2 = g * g;     // Mean cosine squared
const float Hr = 8000.0;    // Rayleigh scale height
const float Hm = 1800.0;    // Mie scale height
const float Is = 10.0;      // Sun intensity
const float cloudTop = 10000.0;    // In meters
const float cloudBottom = 5000.0;  // In meters


const vec3 bM = vec3(21e-6);                        // Mie scattering coefficients at sea level
const vec3 bR = vec3(5.8e-6, 13.5e-6, 33.1e-6);     // Rayleigh scattering coefficients at sea level
const vec3 Ds = normalize(vec3(0.25, -0.05, -1.0)); // Sun direction

const int steps = 64;
const int subSteps = 16;

float noise3D(const in vec3 p){
  return fract(sin(dot(p, vec3(12.9898, 78.233, 151.7182))) * 43758.5453123);
}


float perlinNoise3D(const in vec3 p){
  vec3 f = fract(p);
  f = (f * f) * (3.0 - (2.0 * f));
  float n = dot(floor(p), vec3(1.0, 57.0, 113.0));
  vec4 a = fract(sin(vec4(n + 0.0, n + 1.0, n + 57.0, n + 58.0)) * 43758.5453123);
  vec4 b = fract(sin(vec4(n + 113.0, n + 114.0, n + 170.0, n + 171.0)) * 43758.5453123);
  return mix(mix(mix(a.x, a.y, f.x), mix(a.z, a.w, f.x), f.y), mix(mix(b.x, b.y, f.x), mix(b.z, b.w, f.x), f.y), f.z);
}
```

```
float cloudFBM(in vec3 p){
  float f = 0.0, m = 0.5, mm = 0.0, s = 0.0;
  for(int i=0; i < 6; i++){
    f += perlinNoise3D(p + vec3(uTime * 0.1)) * m;
    s += m;
    p *= mat3(0.00, 0.80, 0.60, -0.80, 0.36, -0.48, -0.60, -0.48, 0.64) * (2.0 + mm);
    m *= 0.5;
    mm += 0.0059;
  }
  return f / s;
}

float cloudMap(const in vec3 p){
  return pow(max(0.0, smoothstep(0.4 + 0.04, 0.6 + 0.04, cloudFBM(p * 2e-4))), 2.0) * uCloudFactor;
}

void getRayleighMieDensities(const in vec3 pos, const in vec3 dir, out float rayleigh, out float mie){
  float h = max(0.0, length(pos) - Re), d = (((cloudBottom < h) && (h < cloudTop)) ?
                                              (cloudMap(pos + vec3(23175.7, 0.0, -(t * 3e3 * 1.0))) *
                                               sin((3.1415 * (h - cloudBottom)) / (cloudTop - cloudBottom))) : 0.0);
  rayleigh = exp(-(h / Hr));
  mie = exp(-(h / Hm)) + d;
}

float getRayLength(const in vec3 p, const in vec3 d, const in float R) {
  float b = dot(p, d), c = dot(p, p) - (R * R), det = (b * b) - c;
  if(det < 0.0){
    return -1.0;
  }else{
    det = sqrt(det);
    float ta = (-b) - det, tb = (-b) + det;
    float t0 = max(ta, tb), t1 = min(ta, tb);
    return (t0 >= 0.0) ? t0 : t1;
  }
}
```

```
vec3 scatter(in vec3 o, const in vec3 d, const float pOffset) {
  o.y += Re; // Camera height offset
  vec3 sR = vec3(0.0), sM = vec3(0.0), mR = vec3(0.0), mM = mR, p = mR, A = mR;
  float L = getRayLength(o, d, Ra), r = length(o) - Ra, dL = L / float(steps), lf = 1.0, ls = 0.0, depthR = 0.0, depthM = 0.0, dR, dM, dc, l, t, dRs,
        dMs, Ls, dLs, depthRs, depthMs;
  for (int i = 0; i < steps; i++) {
    getRayleighMieDensities(p = o + d * ((float(i) - pOffset) * dl), d, dR, dM);
    dR *= dL;
    dM *= dL;
    depthR += dR;
    depthM += dM;
    if((Ls = getRayLength(p, Ds, Ra)) > 0.0){
      dLs = Ls / float(subSteps);
      depthRs = 0.0;
      depthMs = 0.0;
      for(int j = 0; j < subSteps; j++){
        getRayleighMieDensities(p + Ds * (float(j) * dLs), d, dRs, dMs);
        depthRs += dRs * dLs;
        depthMs += dMs * dLs;
      }
      sR += (A = exp(-(bR * (depthRs + depthR) + bM * (depthMs + depthM)))) * dR;
      sM += A * dM;
    }
  }
  float Su = dot(d, Ds);
  float S1uSqr = 1.0 + Su*Su;
  return Is * ((sR * bR* (0.0596831 * S1uSqr)) +
               (sM * bM * ((0.1193662 * (1.0 - g2) * S1uSqr) / ((2.0 + g2) * pow((1.0 + g2) - (2.0 * g * Su), 1.5)))));
}
```

```
// And here the removed uncommented code for handling of outerspace camera positions:

  vec3 oo = o;
  float L = getRayLength(o, d, Ra), lr = getRayLength(oo, d, Re), r = length(o) - Ra;
  bool overEarth = false;
  if(r < 0.0){
    r = getRayLength(o, d, Re * 0.5);
    if(r > 0.0){
      L = min(L, r);
    }
    L = min(L, mix(100e3, 1000e3, smoothstep(0.5, 1.0, (length(o) - Re) / (Ra - Re))));
    if(lr > 0.0){
     L = min(L, lr);
    }
    overEarth = length(o) > cloudTop;
  }else{
    if(L < 0.0){
      return vec3(0.0);
    }else{
      o += d * r;
      L = getRayLength(o, d, Ra);
    }
    r = getRayLength(o, d, Re);
    if(r > 0.0){
      L = min(L, r);
    }
    L = min(L, 1000e3);
    overEarth = true;
  }
```

Realtime global illumination

How you can throw your offline baked light maps etc. away!

But first: What are Spherical Harmonics?

- As Fourier series are a series of functions that are used to represent functions on a circle, spherical harmonics are a series of functions that are used to represent to defined functions on the surface of a sphere.
- Therefore, Spherical harmonics are a series of functions defined on the surface of a sphere, which are used to dissolve some kinds of differential equations.
- The spherical harmonics are a complete and orthonormal set of eigenfunctions of the angular part of the Laplace operator.

Equation 1. $s = (x, y, z) = (\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta)$

Equation 2. $Y_l^m(\theta,\varphi) = K_l^m e^{im\varphi} P_l^{|m|}(\cos\theta), l \in N, -l \le m \le l$

Equation 3. $K_l^m = \sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}$

Equation 4. $y_l^m = \begin{cases} \sqrt{2}\,\mathrm{Re}(Y_l^m) & m > 0 \\ \sqrt{2}\,\mathrm{Im}(Y_l^m) & m < 0 \\ Y_l^0 & m = 0 \end{cases} = \begin{cases} \sqrt{2}K_l^m \cos m\varphi\ P_l^m(\cos\theta) & m > 0 \\ \sqrt{2}K_l^m \sin|m|\varphi\ P_l^{|m|}(\cos\theta) & m < 0 \\ K_l^0\ P_l^0(\cos\theta) & m = 0 \end{cases}$

Equation 1.
Spherical Harmonics define an orthonormal over the sphere, S. Where s are simply locations on the unit sphere.

Equation 2.
The basis functions are, where P_l^m are the associated Legendre polynomials and K_l^m are the normalization constants.

Equation 3.
The complex form, that is used most commonly used in the non-graphics literature

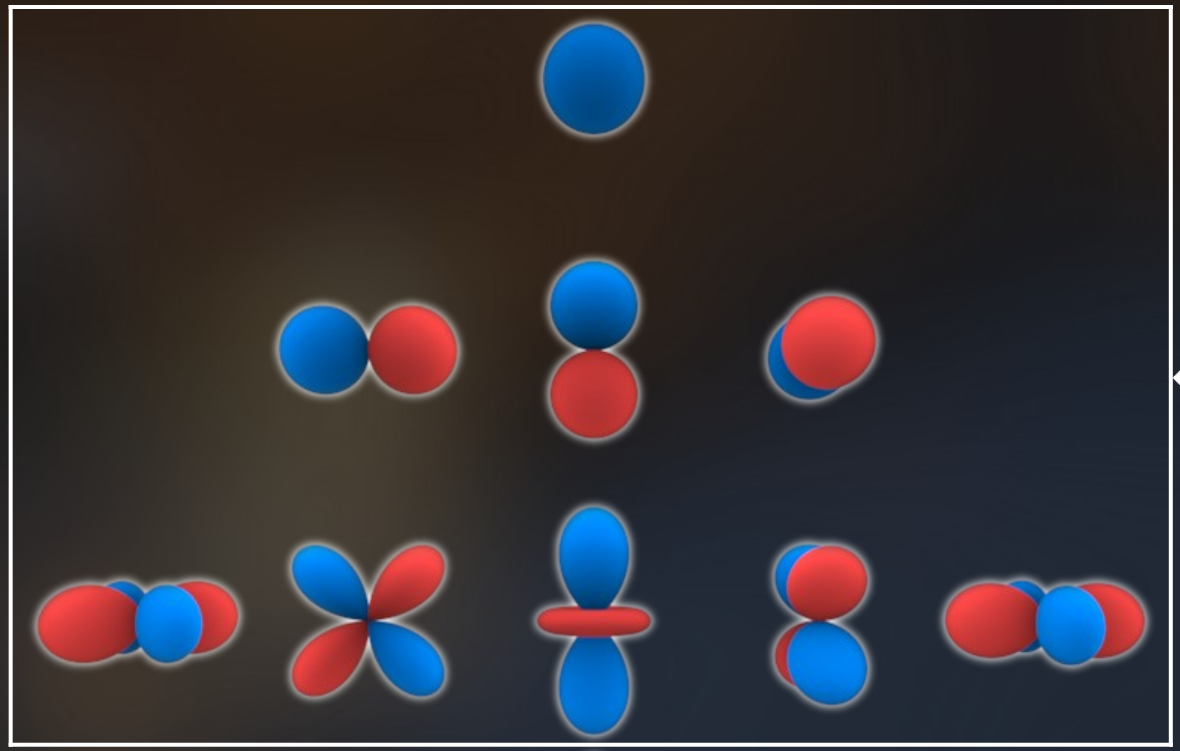
Equation 4.
The real valued basis transformation, where the index l represents the band. Each band is equivalent to polynomials of that degree (so zero is just a constant function, one is linear, etc.) and there are 2l+1 functions in a given band.

1. band

2. band

3. band

4. band



Used by the global illumination solution, that is used in Supraleiter.

Realtime global illumination

- Techniques, that I've tried:
  - Static light maps, static light probes, ..
    - Only for static geometry, not for dynamic geometry
    - Workarounding of light map artifacts very frustratingly
  - Voxel-based (SVOGI, etc.)
    - Too much data for large landscape scenes with 4096 meters visibility range at a visible frame
      - The currently smallest supraleiter map is approximately 3368 x 1160 x 6230 meters big.
      - The currently bigest supraleiter map is approximately 8262 x 748 x 4905 meters big.
    - Too slow for to be really smooth playable
      - primarly bottleneck ⇨ the realtime scene voxelization of these large scenes
  - Light Propagation Volumes (Cascaded with four 32x32x32 volumes)
    - For my taste also too slow with 26 propagation directions and 5 propagation iteration passses (minimum ~12ms per frame on my GTX970), and also still with 6 propagation directions and 10 propagation iteration passses (minimum ~8ms per frame on my GTX970).
      - Together with the another post-process effects too much frame time for smooth 60 frames pro second.
        - » Maybe my compute-based LPV implementation was only bad optimized, but nevertheless...

Is there a better and even faster realtime global illumination solution?

Yes

Blended Cascaded Cached Radiance Hints

Blended Cascaded Cached Radiance Hints

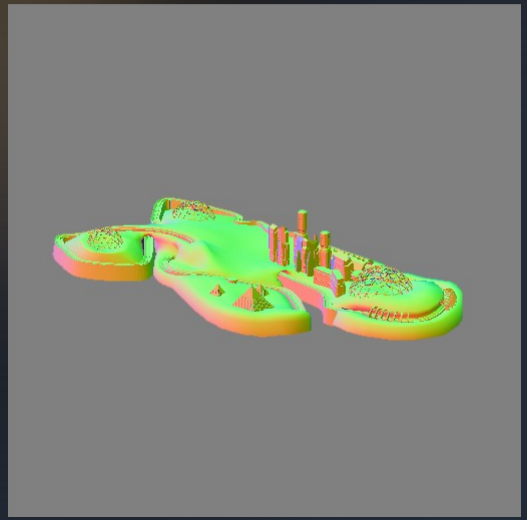
- Based on
  - Real-Time Diffuse Global Illumination Using Radiance Hints
    - http://graphics.cs.aueb.gr/graphics/docs/papers/RadianceHintsPreprint.pdf
  - Real-time Radiance Caching using Chrominance Compression
    - http://jcgt.org/published/0003/04/06/
    - http://graphics.cs.aueb.gr/graphics/research_illumination.html
- Extended by:
  - Four nested 32x32x32 volume cascades
  - Injection of sky light with sky light occlusion
    - Sky light occlusion probe interpolation based on
      - Light probe interpolation using tetrahedral tessellations
        - » http://twvideo01.ubm-us.net/o1/vault/gdc2012/slides/Programming%20Track/Cupisz_Robert_Light_Probe_Interpolation.pdf
  - Blending between cascades
    - even at the bouncing passes
  - Approximately specular lookup, not only diffuse lookup
  - Temporal volume caching

Reflective shadow maps

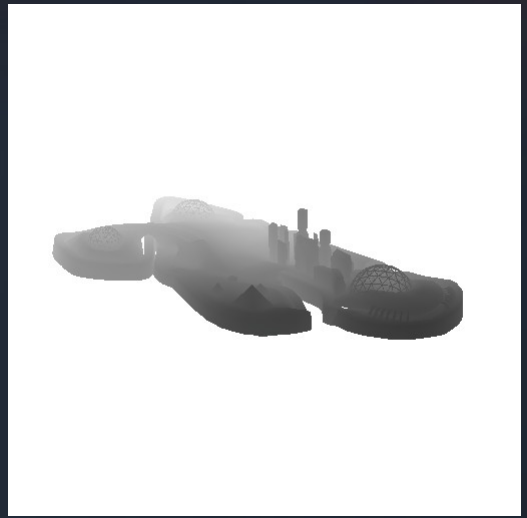
Reflective shadow maps are shadow maps with additional color and normal informations.

Color

Normal



Depth



These RSMs should be world-space-grid sphere or AABB snapped for to avoid flickering etc.

Radiance Hints

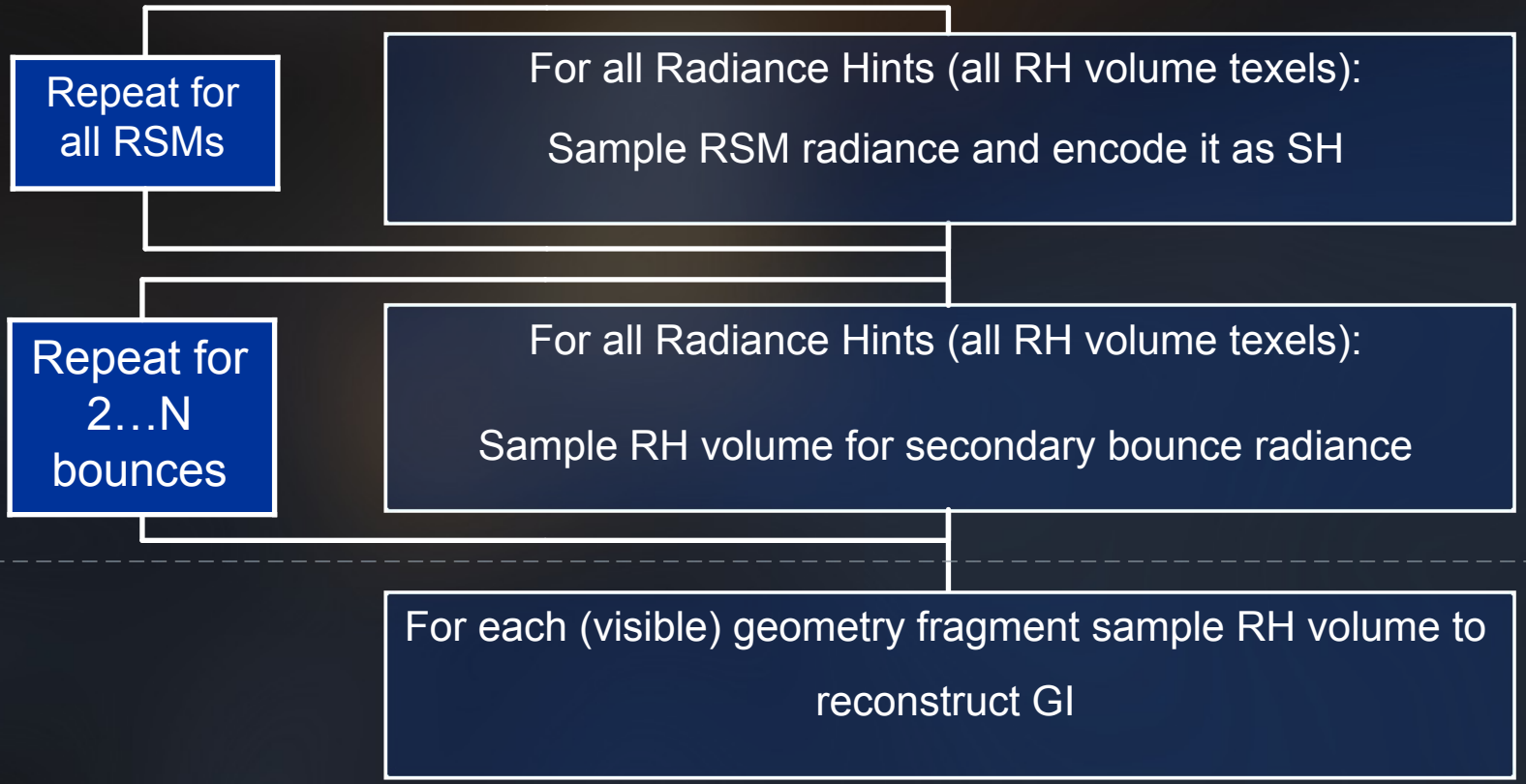
http://graphics.cs.aueb.gr/graphics/docs/papers/RadianceHintsPreprint.pdf

http://graphics.cs.aueb.gr/graphics/research_illumination.html

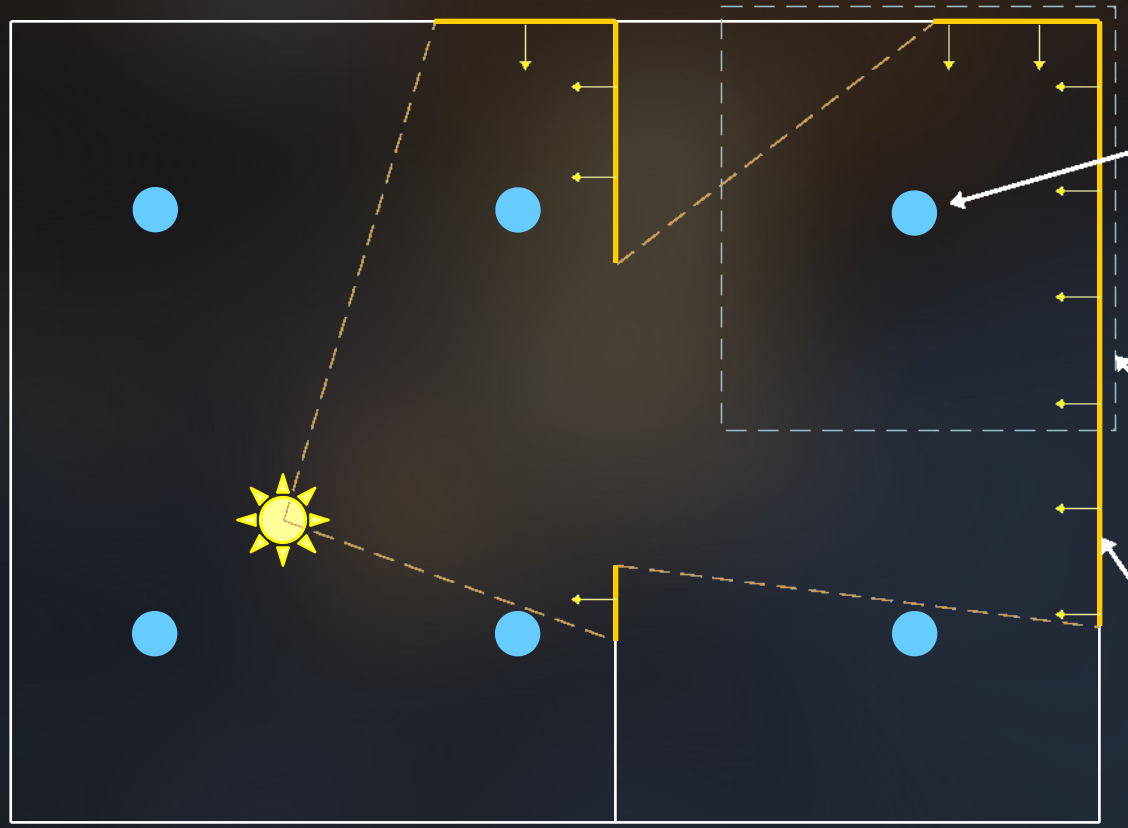
The main idea of Radiance Hints

- The algorithm outline:
  - Radiance Hints samples reflective shadow maps to radiance volumes by stochastically sampling the RSMs
  - Secondary bounces by radiance exchange among RH points
  - Render the scene by sampling the radiance volumes

The main idea of Radiance Hints



Radiance Hint configuration



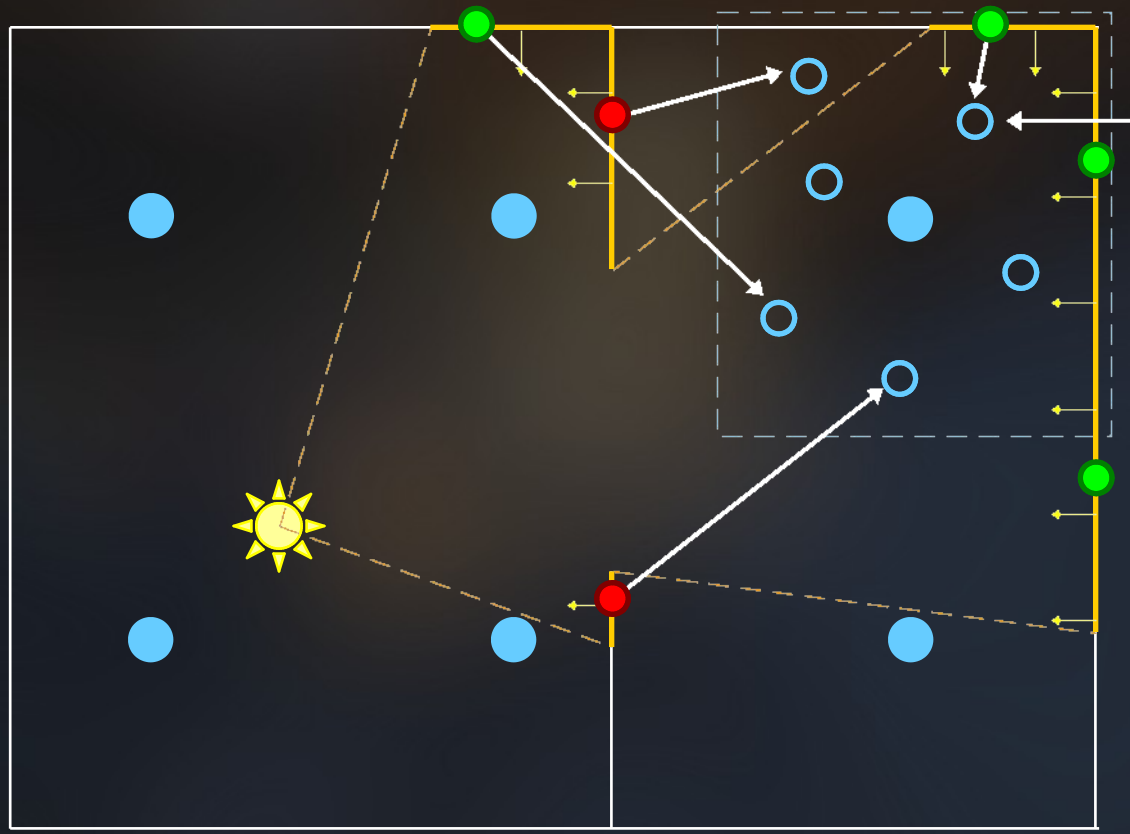
RH points are the nodes of a 3D grid

The grid covers any part of the scene

Extents of each RH grid cell

RSM data (position, flux, normals)

Radiance Hint sampling

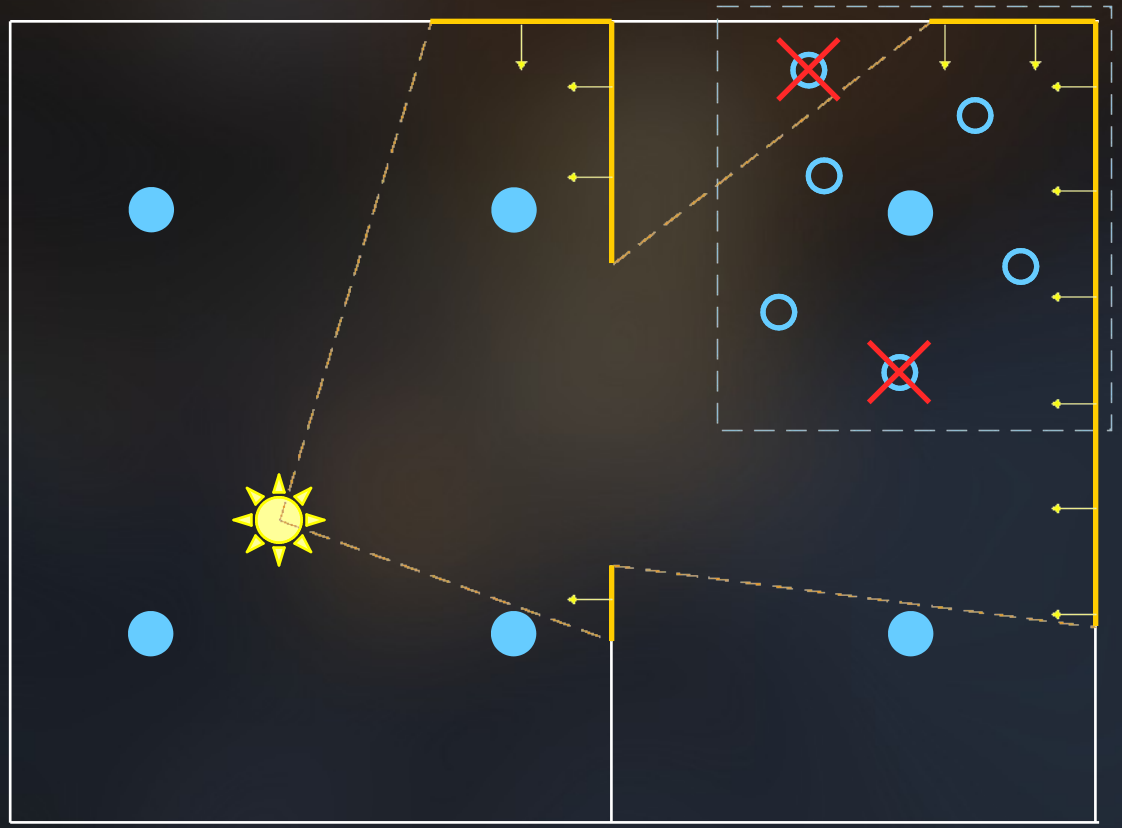


Sample from random points inside a cell

Accepted samples

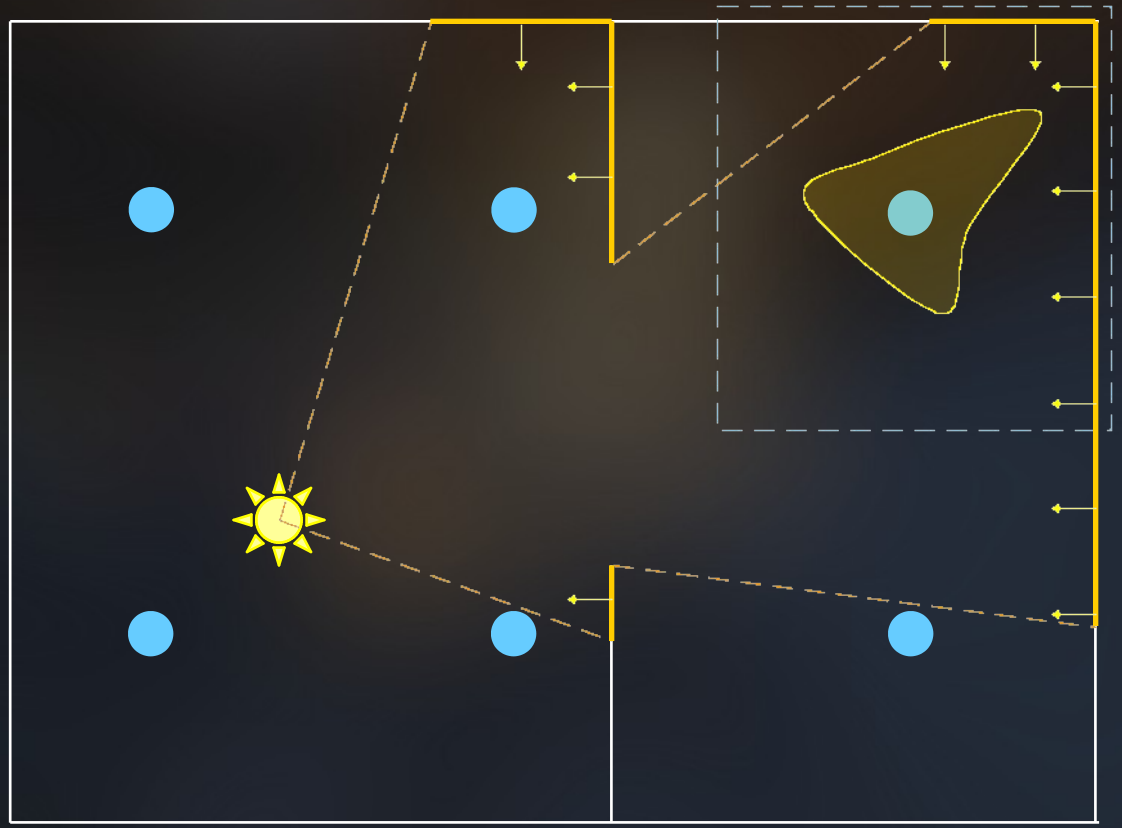
Rejected samples

Radiance Encoding



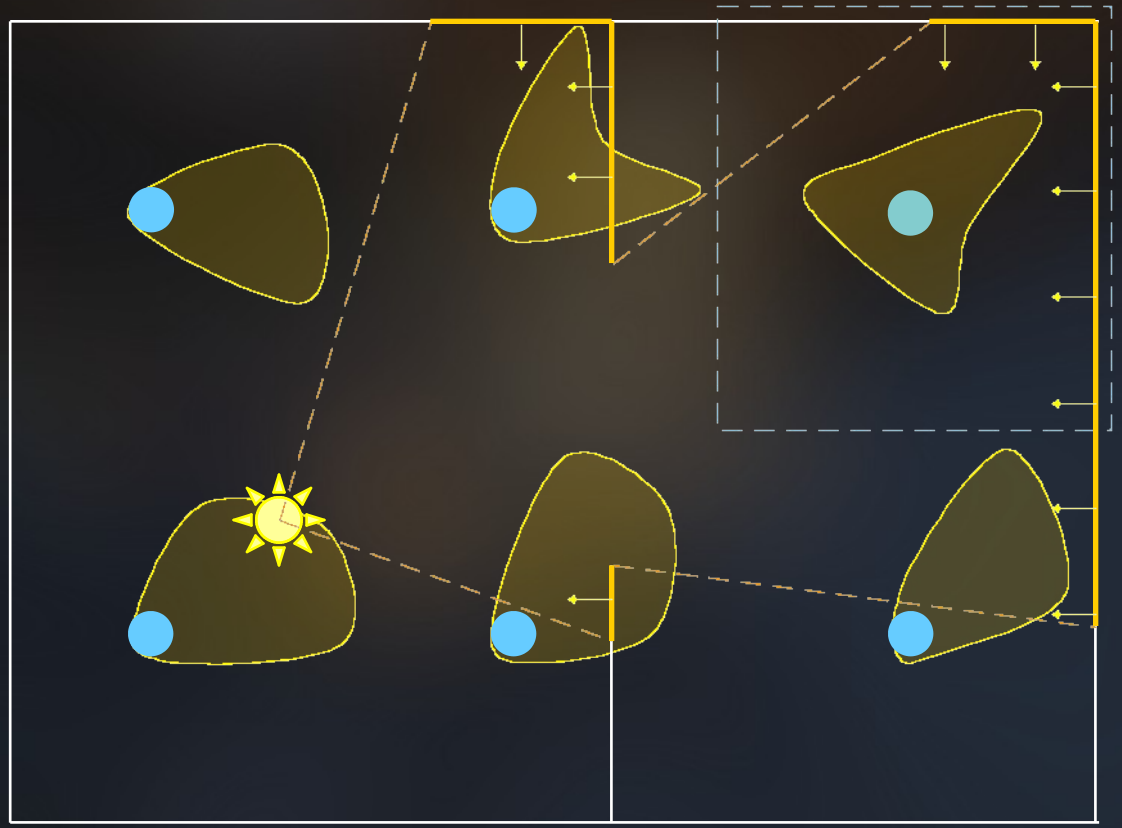
Radiances are encoded with a uniform distribution of sample points in a Radiance Hint cell, since there is nothing special about a Radiance Hint location, therefore it should be representative of entire cell volume.

Radiance Encoding



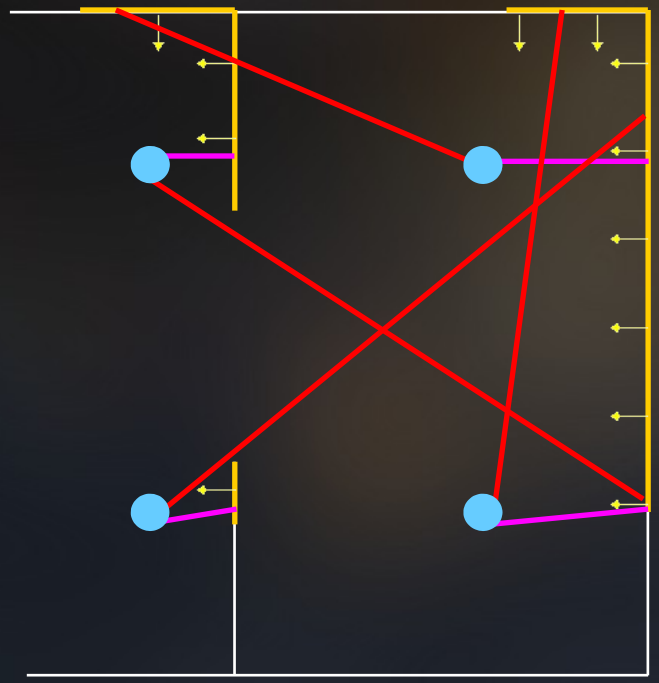
Radiance is encoded as spherical harmonics

Radiance Encoding



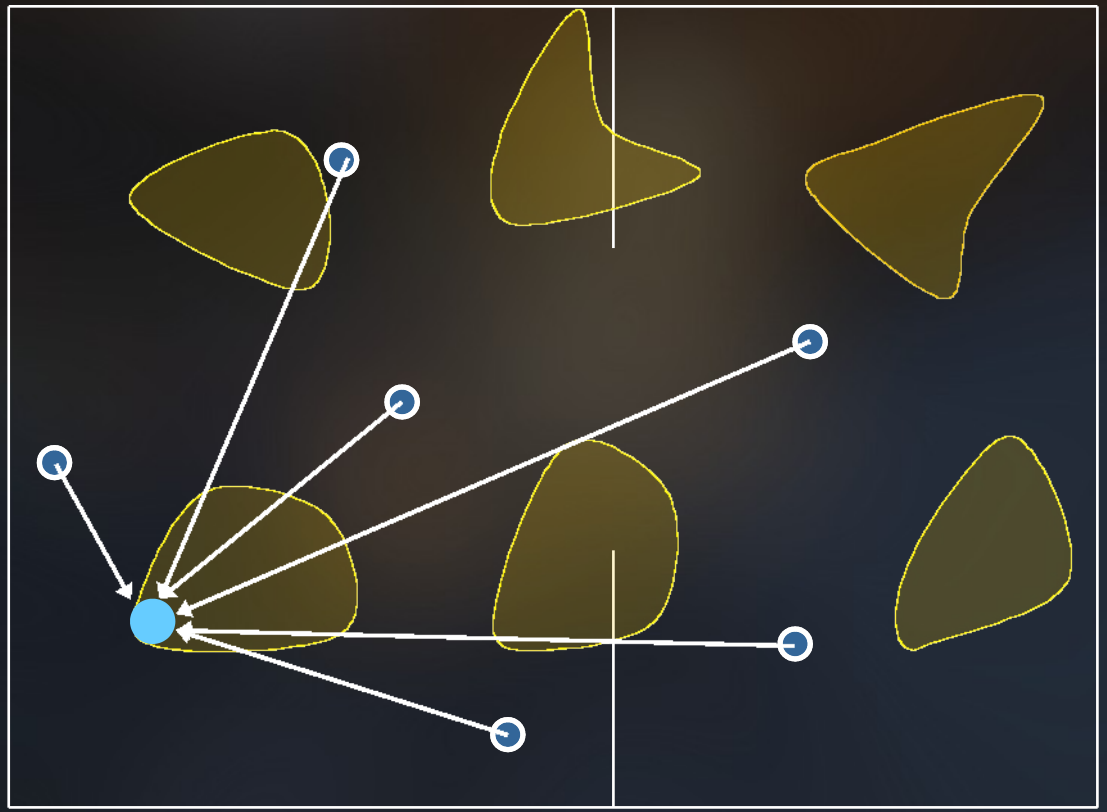
The same for the other cells.

Minimum and maximum RSM sample distances



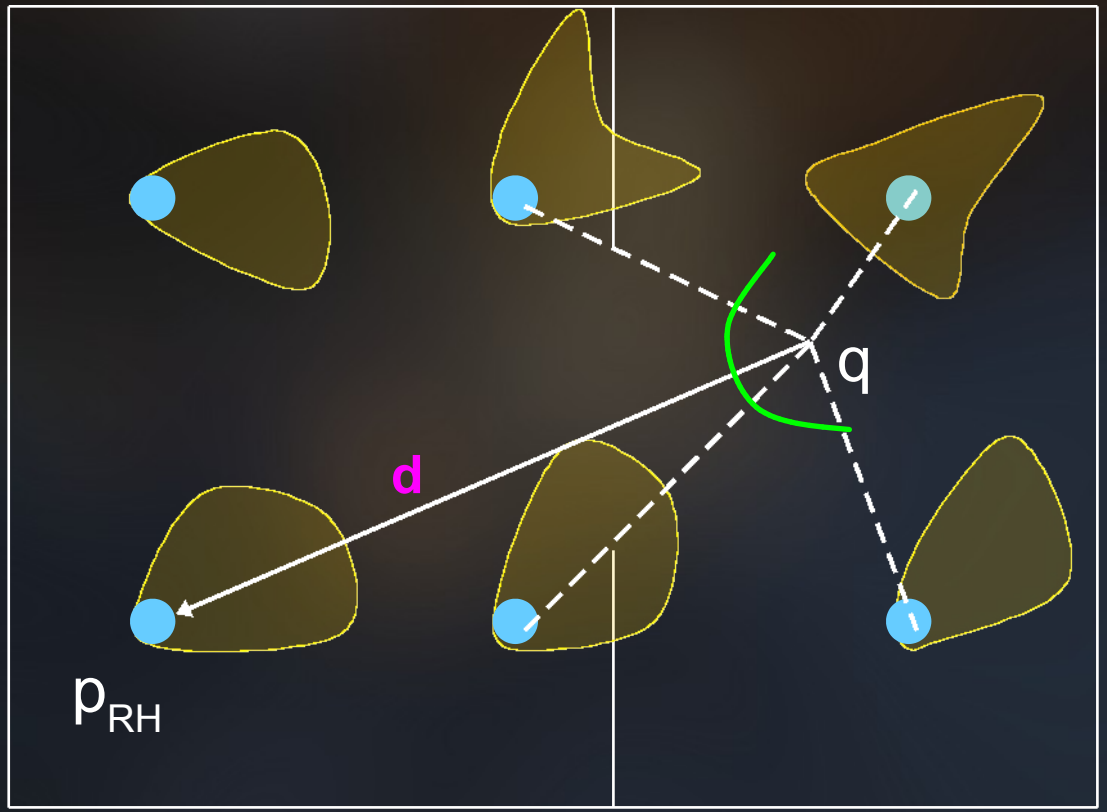
For the approximated occulsion at the secondary bounces later, we do need also sampling the minimum and maximum RSM sample distances

Secondary bounces



For secondary bounces, we sample simply randomly the radiance field!

Secondary bounces

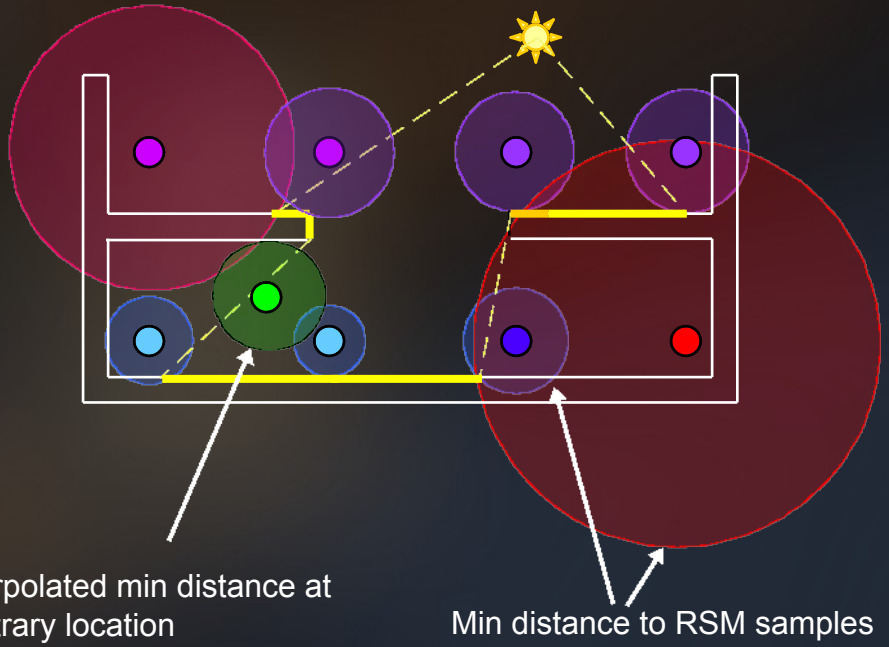


Interpolate SH from nearest RHs (trilinear interpolation)

Integrate incident radiance on the hemisphere aligned with d:

→ “reflected” radiance on an imaginary surface aligned with d

Secondary bounces



But all random RH samples are not equally reliable, because RH near surfaces are reliable, so that these must be favored, and since we know nothing about the geometry, we must use for each RH the minimum distance r_{min} from RSM sample positions with a bias RH contribution using r_{min}

Secondary bounce occlusion

We can use a probabilistic (heuristic) attenuation metric using the stored minimum/maximum distances for this.

```
float lSampleFactor = 1.0 - clamp((lDistance - lMinimumDistance) / (lMinimumDistance - lMinimumDistance), 0.0, 1.0);
float lSampleWeight = clamp(1.0 - lMinimumDistance, 0.0, 1.0);
```

First bounce occlusion

We can use different solutions here:

Camera depth buffer

and/or Voxel tracing

and/or Triangle BVH raytracing

and/or simply just the RSM informations itself, like many light propagation volumes implementations does it in this way too.

etc.

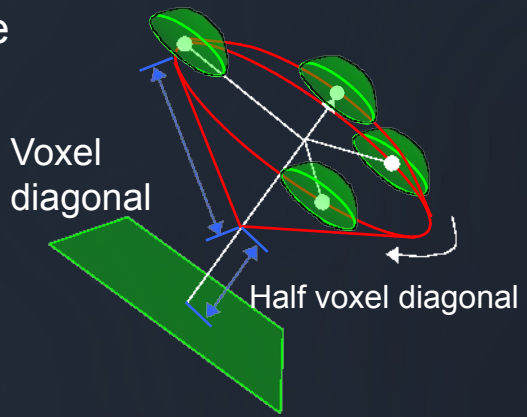
Supraleiter does the here first and additionally the here last solutions.

Global illumination reconstruction

The original Radiance Hint technique uses a four-tap approach at that stage, but my implementation uses only one like all light propagation volumes implementations does it too, with the reason, that I didn't see a visual difference between 1 tap and 4 taps on my tests, and that the approximately specular reconstruction is simple doable with it.

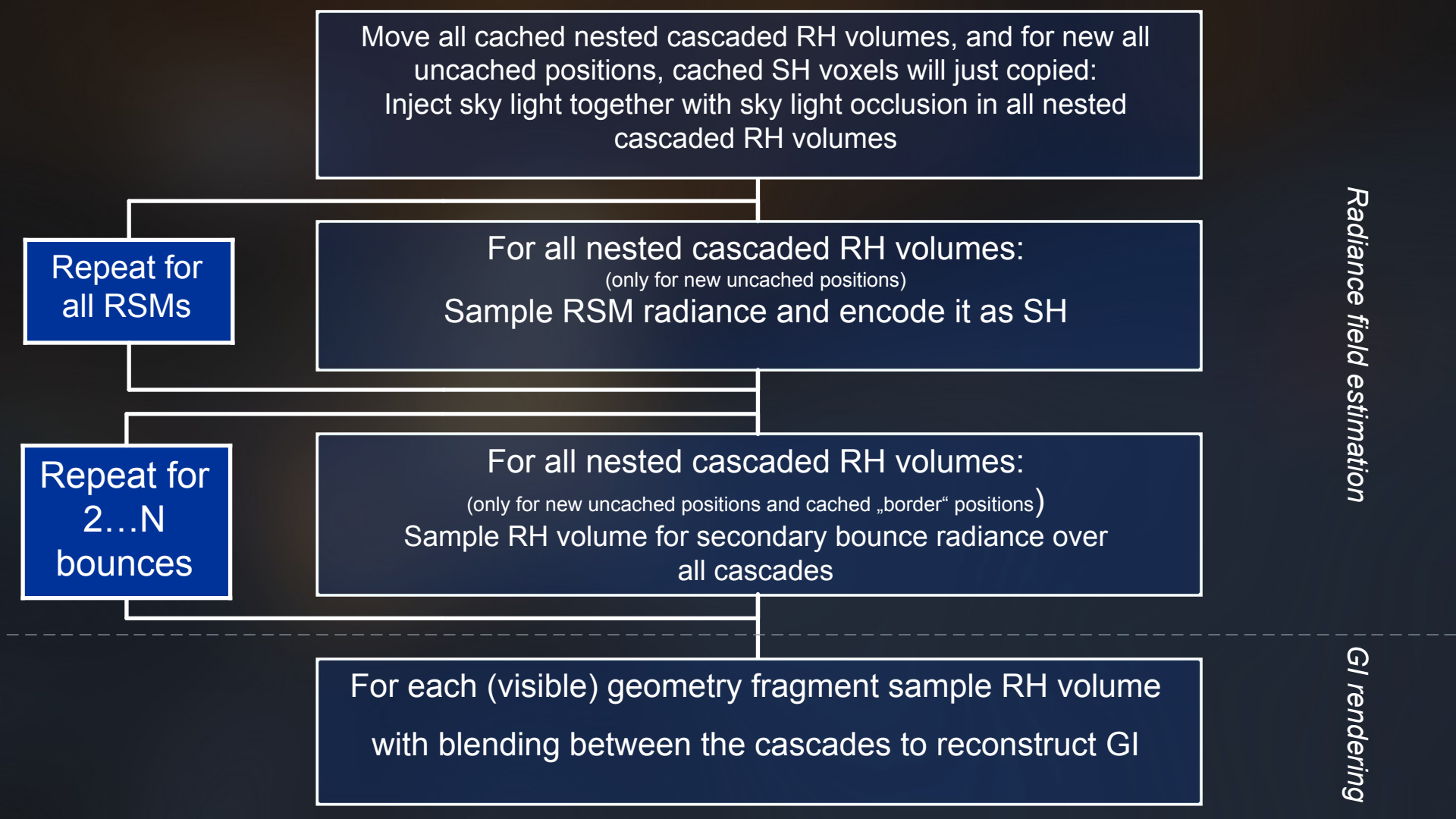
Anyway, here how the original Radiance Hint technique did it:

1. Directly sample the RH volume textures at 4 locations in a rotating kernel above the surface
2. Perform integration in SH domain over the surface-aligned hemisphere

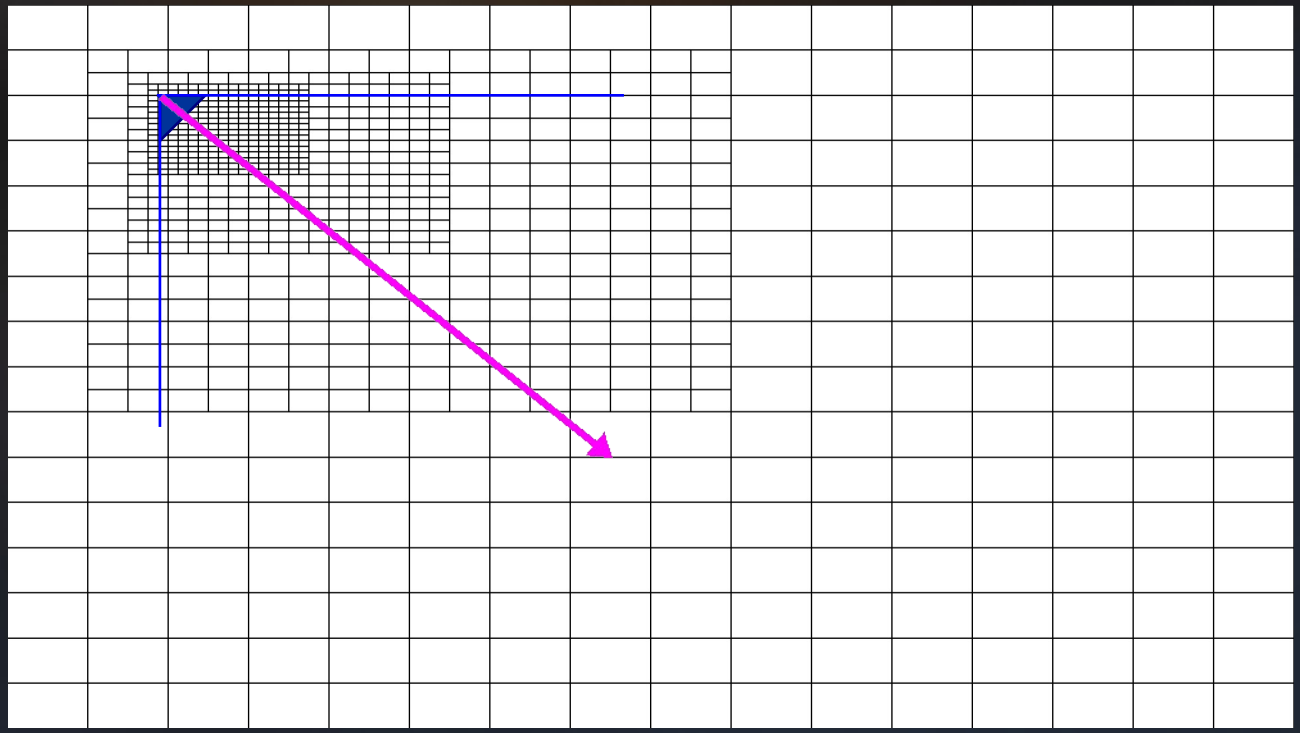


Blended Cascaded Cached Radiance Hints

- Radiance Hints extended by:
  - Four nested 32x32x32 volume cascades
  - Injection of sky light with sky light occlusion
    - Sky light occlusion probe interpolation based on
      - Light probe interpolation using tetrahedral tessellations
        - » http://twvideo01.ubm-us.net/o1/vault/gdc2012/slides/Programming%20Track/Cupisz_Robert_Light_Probe_Interpolation.pdf
  - Blending between cascades
    - even at the bouncing passes
  - Approximately specular lookup, not only diffuse lookup
  - Temporal volume caching


Move all cached nested cascaded RH volumes, and for new all uncached positions, cached SH voxels will just copied:
Inject sky light together with sky light occlusion in all nested cascaded RH volumes
Repeat for all RSMs
For all nested cascaded RH volumes:
(only for new uncached positions)
Sample RSM radiance and encode it as SH
Repeat for 2…N bounces
For all nested cascaded RH volumes:
(only for new uncached positions and cached „border“ positions)
Sample RH volume for secondary bounce radiance over all cascades
For each (visible) geometry fragment sample RH volume
with blending between the cascades to reconstruct GI
Radiance field estimation
GI rendering

Nested cascades



Here nested four 16x16x16 cascades as example

These cascades are world-space AABB seperate axis snapped, for to avoid flickering etc.

Approximately specular lookup

There are two solutions:

1.
Lookup the RH volume with a offset vector a few centimeters or meters forward and then just abuse the IBL split sum lookup texture, where the SH part is just simply the first IBL split sum part.

2.
Extract the dominant light direction and color from the SH and apply your PBR BRDF function specular-only without diffuse.

Supraleiter uses the first solution

Sky light occlusion probe interpolation using tetrahedral tessellations

- Sky light occlusion probe positions will be offline tessellated to a tetrahedron grid and then prebaked.
  - I’m using the Delaunay-Watson algorithm for this.
- A tetrahedron can have 4 points
  - Each point can have 3rd spherical harmonics coefficients for the sky light occlusion.
    - These four SHs can be interpolated with help of the barycentric coordinates as interpolation weights to a SH.
- All tetrahedrons are inserted for the GPU-side into a 3D brick map index volume texture and for CPU-side the tetrahedron grid is itself the lookup structure by traversing the tetrahedron neighbours with the barycentric coordinate signs and a AABB BVH tree for a fast yet uncached first lookup.

Sky light occlusion probe interpolation using tetrahedral tessellations

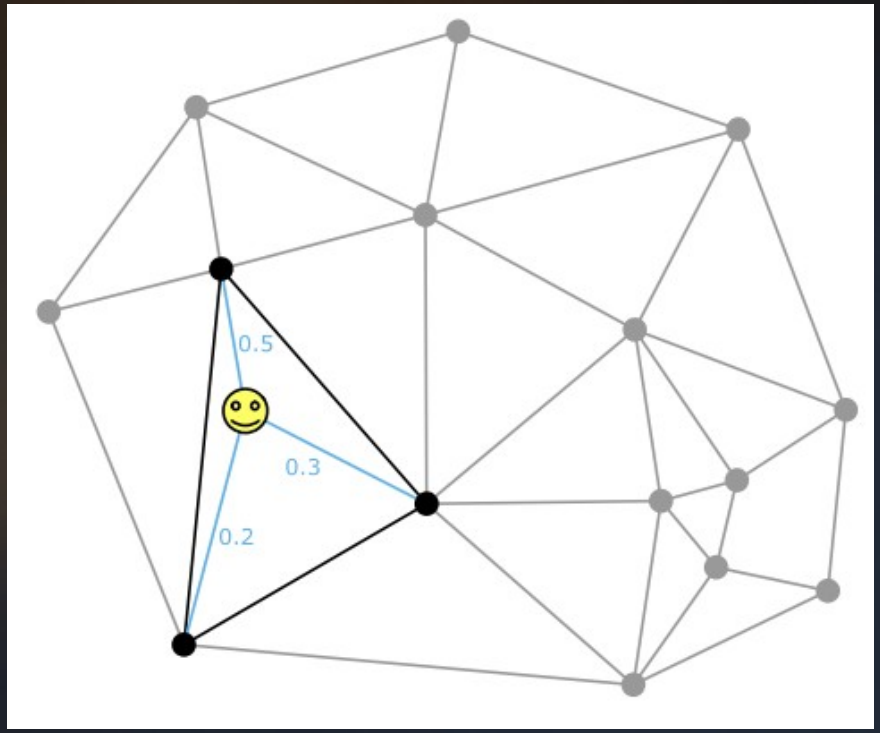


Image source:
http://twvideo01.ubm-us.net/o1/vault/gdc2012/slides/Programming%20Track/Cupisz_Robert_Light_Probe_Interpolation.pdf

Current research area:
Emissive surface light injection

I’ve still on the search for a good RH-compatible solution, where I can inject emissive surface lights.

So stay tuned!

But when somebody here has a idea for it,
then talk to me after the seminar

Blended Cascaded Cached Radiance Hints

Pro:

- Versatile and easy to configure
- Has minimal requirements
- Can nandle multiple bounces
- Can handle large scenes
- The (core) method is view-independent
- Faster than Light Propagation Volumes
- Produces often better visual results than Light Propagation Volumes
- Can handle cases which Light Propagation Volumes fails (at least according to the Radiance Hint original paper)

Contra:

- It’s an approximate approch
- Heuristcally handles occlusion, but this is easy to extensible for example with voxel-trace-based occlusion

Radiance Hints can be a Drop-In-Replacement to Light Propagation Volumes, since it can have the exact same inputs (RSMs etc.) and outputs (SH volumes etc.) like Light Propagation Volumes.

The future

My next research areas in the future

- Emissive surface light injection
- More realtime global illumination optimizations
- Rendering of realistic explosions
- Realtime (bidirectional) GPU path tracing

Questions?

Thanks!