How to use Voxel Cone Tracing with two bounces for everything

instead of just Global Illumination

Who are we?

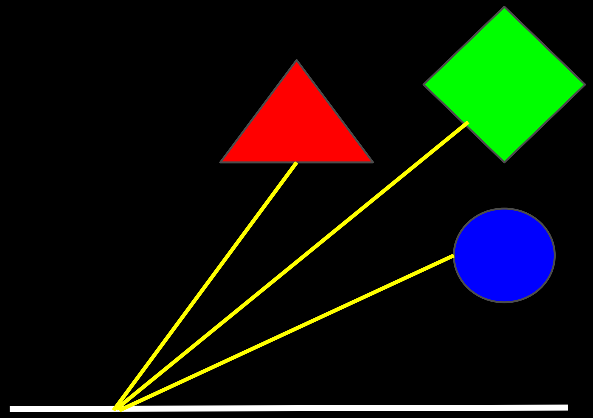
BeRo

urs

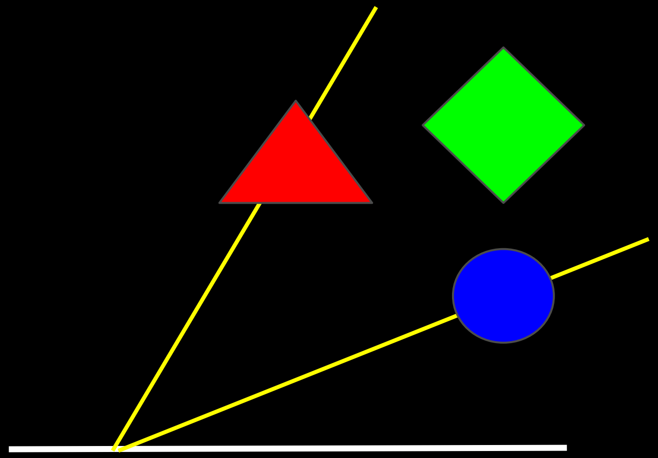
What is Voxel Cone Tracing?

- For diffuse Global Illumination:
  - Multiple directions depending on normal with a aperture angle size depending of the cone count of your choice
- For specular lighting, reflections, refractions, hard and soft shadows, and so on:
  - Single direction where the cone aperture angle size is or can be depending on the roughness material parameter and so on

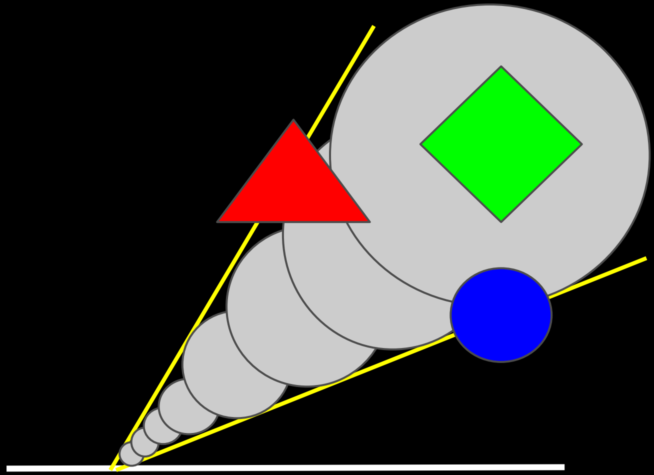
Rays



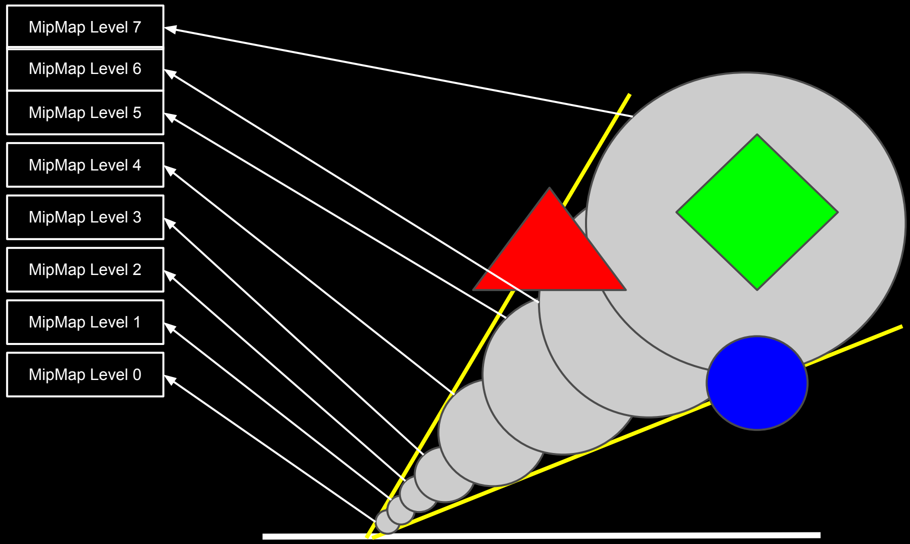
Cones



Cone tracing



Voxel cone tracing



MipMap Level 7

MipMap Level 6

MipMap Level 5

MipMap Level 4

MipMap Level 3

MipMap Level 2

MipMap Level 1

MipMap Level 0

What is Voxel Cone Tracing?

- It’s a kind of Volume ray casting based on the cone tracing idea
  - It starts with some start bias
  - For further steps content-adaptive step size
  - At each lookup radiance and occlusion and accumulate light with occlusion
  - And we stop when occluded or we’re far enough
  - Cone tracing
    - Mip level from local cone aperture angle size with progressively increasing step size

What is Voxel Cone Tracing?

- So it’s a bit like “Ray-Tracing into a simplified scene”
  - but it’s not as precise as ray-tracing
    - but:
      - Fractional geometry intersection support
      - No noise issues
      - Level of detail control
- And it’s only an approximation after all and not an exact precise solution
  - Light leaking can occur in some cases as an example

Voxel Cone Tracing

- Algorithm, how it’s implemented in the 64k intro from yesterday
    - It operates in world space
    - Reset buffer counters
    - Record geometry triangles into a buffer (in an SSBO in case of OpenGL)
        - with help of geometry shader for the triangle-into-a-buffer-recording
    - Voxelize the recorded triangles for the first bounce to a 3D Voxel texture
        - with conservative rasterization (either per geometry shader or per hardware support)
        - Propagate and mipmapping (both at the same step)
    - Reset some (but not all) buffer counters
    - Voxelize the recorded triangles for the second bounce to a 3D Voxel texture
        - with conservative rasterization (either per geometry shader or per hardware support)
        - Propagate and mipmapping (both at the same step)
    - Render the recorded triangles
        - together with gathering the radiance by cone tracing from the second bounce mipmapped 3D Voxel texture

Data resources

- Textures
  - Volume first bounce (3D)
    - 128x128x128
    - 7x RGBA16F targets
      - because of multi-directional anisotropic voxel
    - mipmapped
  - Volume second bounce (3D)
    - 128x128x128
    - 7x RGBA16F targets
      - because of multi-directional anisotropic voxel
    - mipmapped
  - Volume front-view (2D)
    - 128x128
    - not mipmapped
    - The pixel format does not matter here, as it is only used as an empty projection screen for the triangle recording into a buffer and so on

- SSBO buffers
  - Volume globals
  - Volume triangles
  - Volume triangle list items
  - Volume voxels
  - Volume voxel cells

Reset buffer counters

- Count of already recorded triangles inside the buffer
- Triangle linked list index counter
- Count of processed voxel cells

Record geometry triangles into a buffer

- Normal vertex shader
  - As you would otherwise write it normally also, without any ifs and buts.
- Geometry shader
  - which records every incoming 3 vertices as a triangle in a SSBO buffer
    - together with metadata like material ID, vertex position coordinates, vertex normals, vertex texture coordinates and so on
- Dummy no-operation pixel shader
  - which does really nothing
    - together with disabled color and depth writes at OpenGL state machine level

Voxelize the recorded triangles for the first bounce to a 3D Voxel texture (1/2)

- First pass for each buffer triangle
    - Vertex shader
        - fetch the data from the triangle buffer
    - Geometry shader
        - for cross-GPU conservative rasterization
            - in a non-perfect variant, so the pixel shader needs further triangle-Voxel-bounding-box intersection checking
    - Pixel shader
        - Check if the conservative dilated triangle from the geometry shader intersects really the current voxel for to eliminate false-positive voxel-fills
        - Add the result to the voxel-wise triangle linked list
- Second pass for each voxel to the first mipmap-level in the first bounce 3D Voxel texture
    - Vertex shader
        - “#extension GL_AMD_vertex_shader_layer : enable” and gl_Layer magic (for often better load-balancing than with a compute shader)
    - Pixel shader
        - Sum and average the color values by evaluate shading for the corresponding triangle from the triangles from the voxel-wise triangle linked list for the corresponding voxel, **but without any informations of a previous bounce**

Voxelize the recorded triangles for the first bounce to a 3D Voxel texture (2/2)

- Third pass for each voxel to the other mipmap-levels in the first bounce 3D Voxel texture
  - Vertex shader
    - “#extension GL_AMD_vertex_shader_layer : enable” and gl_Layer magic (for often better load-balancing than with a compute shader)
  - Pixel shader
    - Multi-directional anisotropic voxel preintegration
    - Otherwise normal doing-it-yourself-by-hand mipmapping

Isotropic vs anisotropic

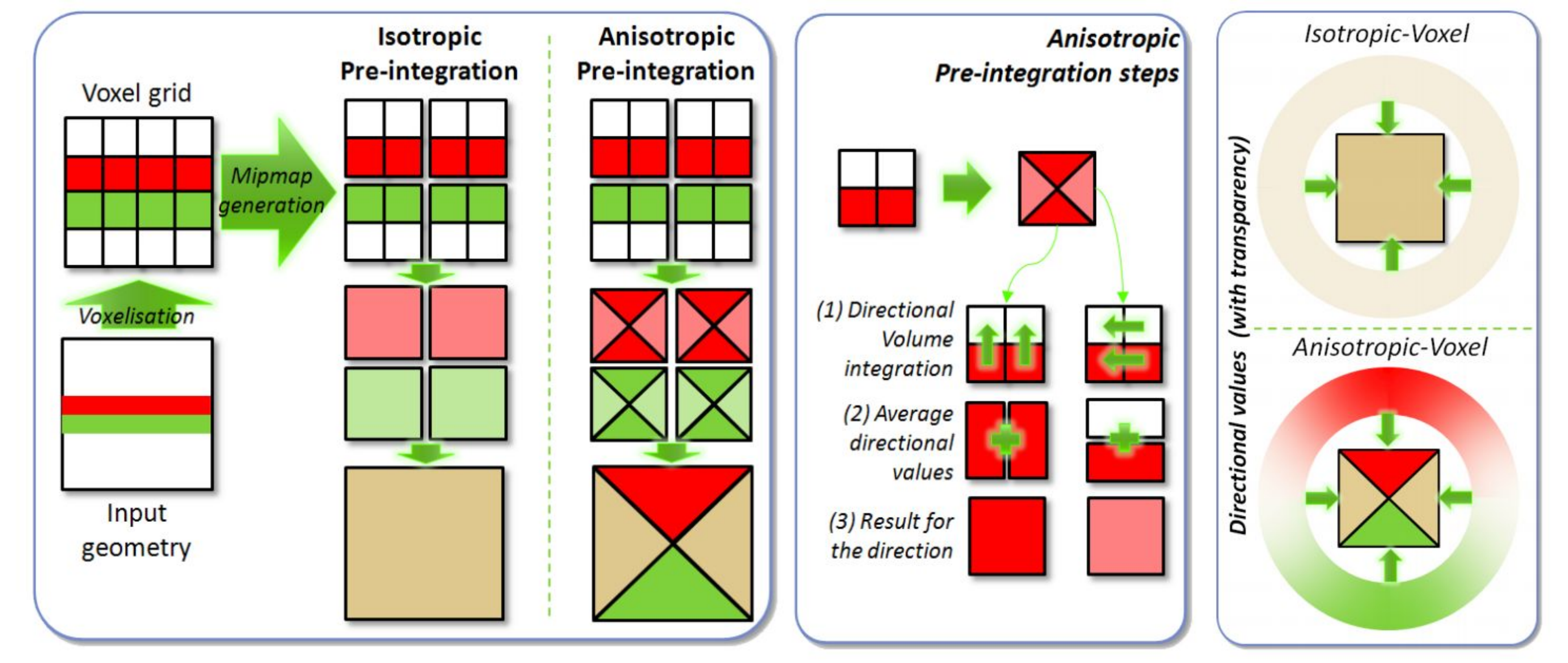


Image source: http://maverick.inria.fr/Publications/2011/CNSGE11b/GIVoxels-pg2011-authors.pdf

Reset some buffer counters

- Triangle linked list index counter
- Count of processed voxel cells

Voxelize the recorded triangles for the second bounce to a 3D Voxel texture (1/2)

- First pass for each buffer triangle
    - Vertex shader
        - fetch the data from the triangle buffer
    - Geometry shader
        - for cross-GPU conservative rasterization
            - in a non-perfect variant, so the pixel shader needs further triangle-Voxel-bounding-box intersection checking
    - Pixel shader
        - Check if the conservative dilated triangle from the geometry shader intersects really the current voxel for to eliminate false-positive voxel-fills
        - Add the result to the voxel-wise triangle linked list
- Second pass for each voxel to the first mipmap-level in the second bounce 3D Voxel texture
    - Vertex shader
        - “#extension GL_AMD_vertex_shader_layer : enable” and gl_Layer magic (for often better load-balancing than with a compute shader)
    - Pixel shader
        - Sum and average the color values by evaluate shading for the corresponding triangle from the triangles from the voxel-wise triangle linked list for the corresponding voxel, **but now with the informations of the previous first bounce**

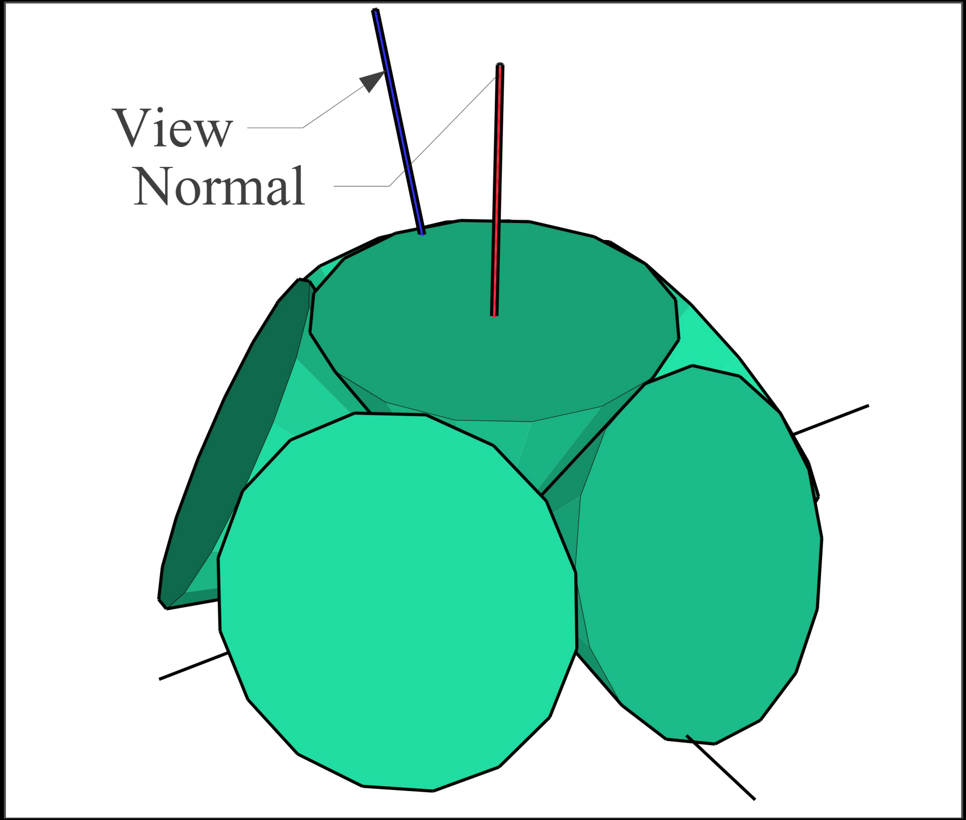
Voxelize the recorded triangles for the second bounce to a 3D Voxel texture (2/2)

- Third pass for each voxel to the other mipmap-levels in the second bounce 3D Voxel texture
    - Vertex shader
        - “#extension GL_AMD_vertex_shader_layer : enable” and gl_Layer magic (for often better load-balancing than with a compute shader)
    - Pixel shader
        - Multi-directional anisotropic voxel preintegration
        - Otherwise normal doing-it-yourself-by-hand mipmapping

Render the recorded triangles

- Almost normal vertex shader
  - As you would otherwise write it normally also, but by fetching the data from the recorded triangle buffer
- Almost normal pixel shader
  - Evaluate shading for the corresponding triangle with the informations of the second bounce 3D texture with cone tracing

Diffuse cones



Multiple directions depending on normal with a aperture angle size depending of the cone count of your choice

Image source: https://github.com/jose-villegas/VCTRenderer

BRDF

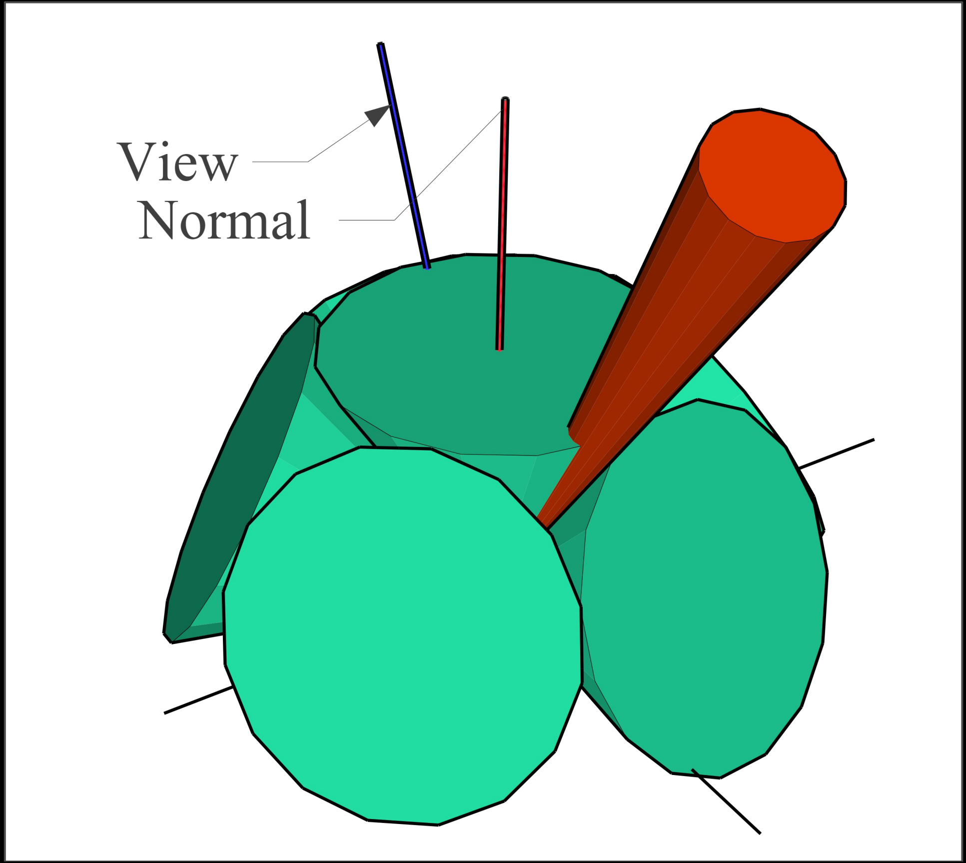
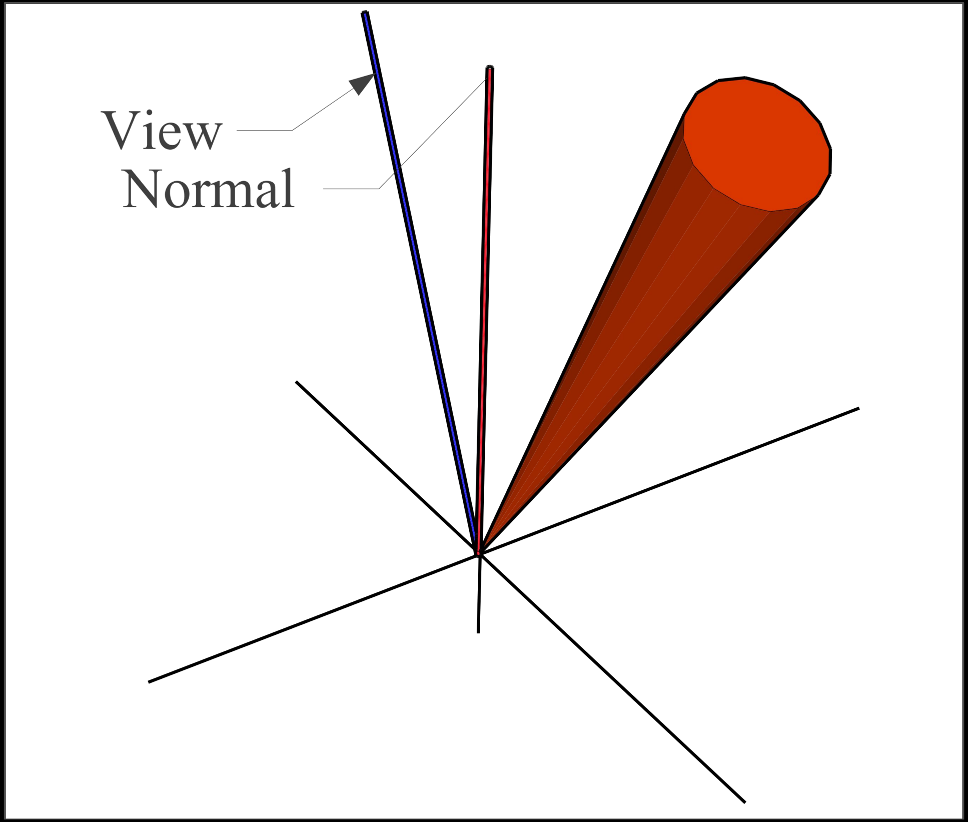


Image source: https://github.com/jose-villegas/VCTRenderer

Specular cone (reflections, refractions, etc.)



Single direction where the cone aperture angle size is or can be depending on the roughness material parameter and so on

Image source: https://github.com/jose-villegas/VCTRenderer

UE4 BRDF PBR model roughness to cone aperture mapping

Various found options:

```
float roughnessToVoxelConeTracingApertureAngle(float roughness){
        roughness = clamp(roughness, 0.0, 1.0);
#if 1
        return tan(0.0003474660443456835 + (roughness * (1.3331290497744692 - (roughness * 0.5040552688878546)))); // <= used in the 64k
#elif 1
        return tan(acos(pow(0.244, 1.0 / (clamp(2.0 / max(1e-4, (roughness * roughness)) - 2.0, 4.0, 1024.0 * 16.0) + 1.0))));
#else
        return clamp(tan((PI * (0.5 * 0.75)) * max(0.0, roughness)), 0.00174533102, 3.14159265359);
#endif
}
```

```
float apertureAngle = roughnessToVoxelConeTracingApertureAngle(materialRoughness);
```

Soft shadows

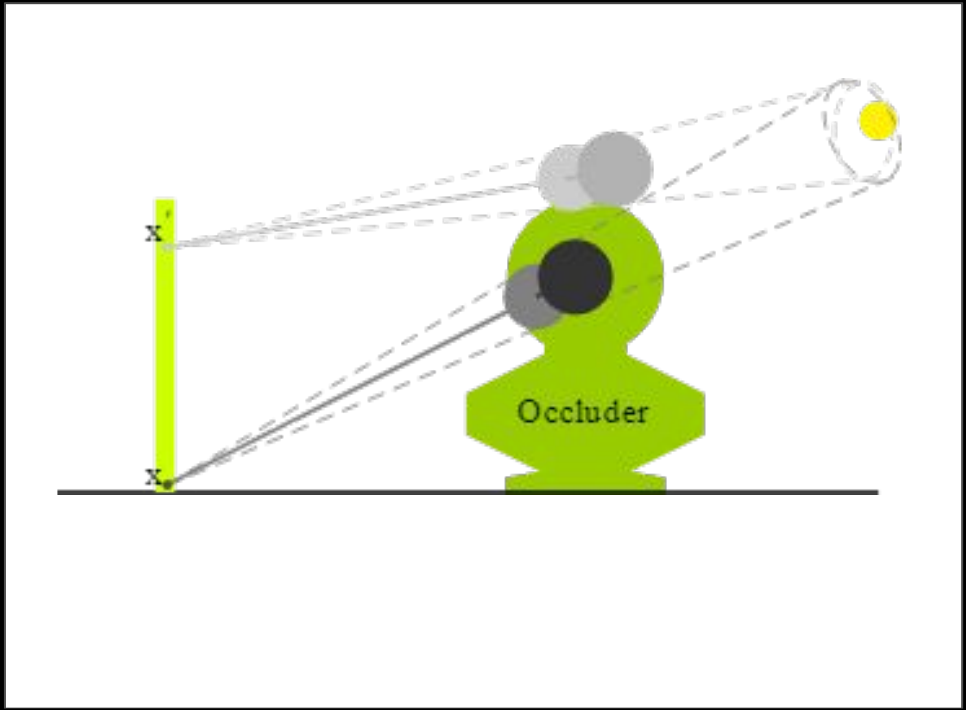


Image source: https://github.com/jose-villegas/VCTRenderer

We do want to see CODE

and you’ll get some CC0-licensed code parts

(but please give credits)

```
const int voxelVolumeBits = 7;
const int voxelVolumeSize = 1 << voxelVolumeBits;

const float voxelVolumeInverseSize = 1.0 / float(voxelVolumeSize); // Size of a voxel

const mat4 voxelToWorldMatrix = mat4(vec4(worldSize.x / float(voxelVolumeSize), 0.0, 0.0, 0.0),
                                     vec4(0.0, worldSize.y / float(voxelVolumeSize), 0.0, 0.0),
                                     vec4(0.0, 0.0, worldSize.z / float(voxelVolumeSize), 0.0),
                                     vec4(worldMin, 1.0)),

           worldToVoxelMatrix = mat4(vec4(float(voxelVolumeSize) / worldSize.x, 0.0, 0.0, 0.0),
                                     vec4(0.0, float(voxelVolumeSize) / worldSize.y, 0.0, 0.0),
                                     vec4(0.0, 0.0, float(voxelVolumeSize) / worldSize.z, 0.0),
                                     vec4(-(worldMin * (float(voxelVolumeSize) / worldSize)), 1.0)),

           normalizedVoxelToWorldMatrix = mat4(vec4(worldSize.x, 0.0, 0.0, 0.0),
                                               vec4(0.0, worldSize.y, 0.0, 0.0),
                                               vec4(0.0, 0.0, worldSize.z, 0.0),
                                               vec4(worldMin, 1.0)),

           worldToNormalizedVoxelMatrix = mat4(vec4(1.0 / worldSize.x, 0.0, 0.0, 0.0),
                                               vec4(0.0, 1.0 / worldSize.y, 0.0, 0.0),
                                               vec4(0.0, 0.0, 1.0 / worldSize.z, 0.0),
                                               vec4(-(worldMin / worldSize), 1.0));
```

```
vec3 safeNormalize(vec3 n){
  float l = max(length(n), 1e-6);
  n /= l;
  return n;
}

vec4 voxelJitterNoise(vec4 p4){
  p4 = fract(p4 * vec4(443.897, 441.423, 437.195, 444.129));
  p4 += dot(p4, p4.wzxy + vec4(19.19));
  return fract((p4.xxyz + p4.yzzw) * p4.zywx);
}

mat3 vctRotationMatrix(vec3 axis, float angle){
   axis = normalize(axis);
   float s = sin(angle), c = cos(angle), oc = 1.0 - c;
   vec3 as = axis * s;
   return (mat3(axis.x * axis, axis.y * axis, axis.z * axis) * oc) + mat3(c, -as.z, as.y, as.z, c, -as.x, -as.y, as.x, c);
}
```

```
vec3 getDirectionWeights(vec3 direction){
#if 0
  vec3 d = abs(normalize(direction));
  return d / dot(d, vec3(1.0));
#elif 0
  return abs(direction);
#else
  return direction * direction;
#endif
}

vec4 traceVoxelCone(vec3 from,
                    vec3 direction,
                    float aperture,
                    float offset,
                    float maxDistance){
  direction = normalize(direction);
  bvec3 negativeDirection = lessThan(direction, vec3(0.0));
  float doubledAperture = max(voxelVolumeInverseSize, 2.0 * aperture),
        dist = offset;
  vec3 directionWeights = getDirectionWeights(direction),
       position = from + (dist * direction);
  vec4 accumulator = vec4(0.0);
  maxDistance = min(maxDistance, 1.41421356237);
  while((dist < maxDistance) &&
    /*  all(greaterThanEqual(position, vec3(0.0))) &&
        all(lessThanEqual(position, vec3(1.0))) && */
        (accumulator.a < 1.0)){
    float diameter = max(voxelVolumeInverseSize * 0.5, doubledAperture * dist),
          mipMapLevel = max(0.0, log2((diameter * float(voxelVolumeSize)) + 0.0));
    vec4 voxel = (textureLod(negativeDirection.x ? VoxelVolumeTexture3 : VoxelVolumeTexture0, position, mipMapLevel) * directionWeights.x) +
                 (textureLod(negativeDirection.y ? VoxelVolumeTexture4 : VoxelVolumeTexture1, position, mipMapLevel) * directionWeights.y) +
                 (textureLod(negativeDirection.z ? VoxelVolumeTexture5 : VoxelVolumeTexture2, position, mipMapLevel) * directionWeights.z);
    accumulator += (1.0 - accumulator.w) * voxel;
    dist += max(diameter, voxelVolumeInverseSize) * 1.0;
    position = from + (dist * direction);
  }
  return max(accumulator, vec4(0.0));
}
```

```
float traceShadowCone(vec3 normal,
                      vec3 from,
                      vec3 to){
  const float aperture = tan(radians(5.0)),
              doubledAperture = max(voxelVolumeInverseSize, 2.0 * aperture),
              s = 0.33333;
  from += normal * voxelVolumeInverseSize * 2.0;
  vec3 direction = to - from;
  float maxDistance = length(direction),
        dist = 2.5 * voxelVolumeInverseSize,
        accumulator = 0.0;
  direction /= maxDistance;
  maxDistance = min(maxDistance, 1.41421356237);
  dist += voxelJitterNoise(vec4(from.xyz + to.xyz + normal.xyz, tc.x)).x * s * voxelVolumeInverseSize;
  vec3 position = from + (direction * dist);
  while((accumulator < 1.0) && (dist < maxDistance) && isInsideCube(position, 0.0)){
    float diameter = max(voxelVolumeInverseSize * 0.5, doubledAperture * dist),
          mipMapLevel = max(0.0, log2((diameter * float(voxelVolumeSize)) + 1.0));
    accumulator += (1.0 - accumulator) * clamp(textureLod(VoxelVolumeTexture6, position, mipMapLevel).w * 1.0, 0.0, 1.0);
    dist += max(diameter, voxelVolumeInverseSize) * s;
    position = from + (direction * dist);
  }
  return clamp(1.0 - accumulator, 0.0, 1.0);
}
```

```
vec3 indirectDiffuseLight(vec3 from,
                          vec3 normal){
  #define NUM_CONES 5
  const vec3 coneDirections[5] = vec3[5](
                                   vec3(0.0, 0.0, 1.0),
                                   vec3(0.0, 0.707106781, 0.707106781),
                                   vec3(0.0, -0.707106781, 0.707106781),
                                   vec3(0.707106781, 0.0, 0.707106781),
                                   vec3(-0.707106781, 0.0, 0.707106781)
                                 );
  const float coneWeights[5] = float[5]( 0.28, 0.18, 0.18, 0.18, 0.18 );
  const float coneApertures[5] = float[5]( /* tan(45) */ 1.0, 1.0, 1.0, 1.0, 1.0 );
  const float coneOffset = -0.01, offset = 4.0 * voxelVolumeInverseSize, maxDistance = 2.0;
  normal = normalize(normal);
  vec3 normalOffset = normal * (1.0 + (4.0 * 0.70710678118)) * voxelVolumeInverseSize, coneOrigin = from + normalOffset,
       t0 = cross(vec3(0.0, 1.0, 0.0), normal), t1 = cross(vec3(0.0, 0.0, 1.0), normal),
       tangent = normalize((length(t0) < length(t1)) ? t1 : t0), bitangent = normalize(cross(tangent, normal));
  tangent = safeNormalize(cross(bitangent, normal));
  mat3 tangentSpace =
#ifdef indirectDiffuseLightJitter
                      vctRotationMatrix(normal, voxelJitterNoise(vec4(from + normal + (vec3(gl_FragCoord.xyz) * 1.0), fract(tc.x * 0.01) *
100.0)).x) *
#endif
                      mat3(tangent, bitangent, normal);
  vec4 color = vec4(0.0);
  for(int i = 0; i < NUM_CONES; i++){
    vec3 direction = tangentSpace * coneDirections[i].xyz;
/*  if(dot(direction, tangentSpace[2]) >= 0.0)*/{
      color += vec4(traceVoxelCone(coneOrigin + (coneOffset * direction),
                                   direction,
                                   coneApertures[i],
                                   offset,
                                   maxDistance).xyz,
                     1.0) * coneWeights[i];
    }
  }
  return color.xyz / max(color.w, 1e-6);
}
```

```
vec3 indirectSpecularLight(vec3 from,
                           vec3 normal,
                           vec3 viewDirection,
                           float aperture,
                           float maxDistance){
  normal = normalize(normal);
  viewDirection = normalize(viewDirection);
  return traceVoxelCone(from + (normal * 2.0 * voxelVolumeInverseSize),
                        normalize(reflect(viewDirection, normal)),
                        aperture,
                        2.0 * voxelVolumeInverseSize,
                        maxDistance).xyz;
}

vec3 indirectRefractiveLight(vec3 from,
                             vec3 normal,
                             vec3 viewDirection,
                             float aperture,
                             float indexOfRefraction,
                             float maxDistance){
  normal = normalize(normal);
  viewDirection = normalize(viewDirection);
  return traceVoxelCone(from + (normal * 1.0 * voxelVolumeInverseSize),
                        normalize(refract(viewDirection, normal, 1.0 / indexOfRefraction)),
                        aperture,
                        1.0 * voxelVolumeInverseSize,
                        maxDistance).xyz;
}
```

```
#define voxelIndex(x, y, z) ((((z * 2) + y) * 2) + x)

void fetchVoxels(out vec4 voxels[8], const in sampler3D t, ivec3 pos, int level){
  voxels = vec4[8](
              texelFetch(t, pos + ivec3(0, 0, 0), level), // 0
              texelFetch(t, pos + ivec3(1, 0, 0), level), // 1
              texelFetch(t, pos + ivec3(0, 1, 0), level), // 2
              texelFetch(t, pos + ivec3(1, 1, 0), level), // 3
              texelFetch(t, pos + ivec3(0, 0, 1), level), // 4
              texelFetch(t, pos + ivec3(1, 0, 1), level), // 5
              texelFetch(t, pos + ivec3(0, 1, 1), level), // 6
              texelFetch(t, pos + ivec3(1, 1, 1), level)  // 7
            );
}

vec4 mipMapVoxelColor(const in sampler3D t, ivec3 uvw, int level){
  return (texelFetch(t, uvw + ivec3(0, 0, 0), level) +
          texelFetch(t, uvw + ivec3(1, 0, 0), level) +
          texelFetch(t, uvw + ivec3(1, 1, 0), level) +
          texelFetch(t, uvw + ivec3(0, 1, 0), level) +
          texelFetch(t, uvw + ivec3(0, 0, 1), level) +
          texelFetch(t, uvw + ivec3(1, 0, 1), level) +
          texelFetch(t, uvw + ivec3(1, 1, 1), level) +
          texelFetch(t, uvw + ivec3(0, 1, 1), level)) * 0.125;
}


  ivec3 uvw = ivec3(ivec2(gl_FragCoord.xy), In.layer) << 1;
  int level = mipMapLevel - 1;

  vec4 voxels[8];
```

```
{
  // +x
  fetchVoxels(voxels, VoxelVolumeTexture0, uvw, level);
  oOutput0 = ((voxels[voxelIndex(0, 0, 0)] + (voxels[voxelIndex(1, 0, 0)] * (1.0 - voxels[voxelIndex(0, 0, 0)].w))) +
              (voxels[voxelIndex(0, 1, 0)] + (voxels[voxelIndex(1, 1, 0)] * (1.0 - voxels[voxelIndex(0, 1, 0)].w))) +
              (voxels[voxelIndex(0, 0, 1)] + (voxels[voxelIndex(1, 0, 1)] * (1.0 - voxels[voxelIndex(0, 0, 1)].w))) +
              (voxels[voxelIndex(0, 1, 1)] + (voxels[voxelIndex(1, 1, 1)] * (1.0 - voxels[voxelIndex(0, 1, 1)].w)))) * 0.25;
}
{
  // +y
  fetchVoxels(voxels, VoxelVolumeTexture1, uvw, level);
  oOutput1 = ((voxels[voxelIndex(0, 0, 0)] + (voxels[voxelIndex(0, 1, 0)] * (1.0 - voxels[voxelIndex(0, 0, 0)].w))) +
              (voxels[voxelIndex(1, 0, 0)] + (voxels[voxelIndex(1, 1, 0)] * (1.0 - voxels[voxelIndex(1, 0, 0)].w))) +
              (voxels[voxelIndex(0, 0, 1)] + (voxels[voxelIndex(0, 1, 1)] * (1.0 - voxels[voxelIndex(0, 0, 1)].w))) +
              (voxels[voxelIndex(1, 0, 1)] + (voxels[voxelIndex(1, 1, 1)] * (1.0 - voxels[voxelIndex(1, 0, 1)].w)))) * 0.25;
}
{
  // +z
  fetchVoxels(voxels, VoxelVolumeTexture2, uvw, level);
  oOutput2 = ((voxels[voxelIndex(0, 0, 0)] + (voxels[voxelIndex(0, 0, 1)] * (1.0 - voxels[voxelIndex(0, 0, 0)].w))) +
              (voxels[voxelIndex(1, 0, 0)] + (voxels[voxelIndex(1, 0, 1)] * (1.0 - voxels[voxelIndex(1, 0, 0)].w))) +
              (voxels[voxelIndex(0, 1, 0)] + (voxels[voxelIndex(0, 1, 1)] * (1.0 - voxels[voxelIndex(0, 1, 0)].w))) +
              (voxels[voxelIndex(1, 1, 0)] + (voxels[voxelIndex(1, 1, 1)] * (1.0 - voxels[voxelIndex(1, 1, 0)].w)))) * 0.25;
}
{
  // -x
  fetchVoxels(voxels, VoxelVolumeTexture0, uvw, level);
  oOutput3 = ((voxels[voxelIndex(1, 0, 0)] + (voxels[voxelIndex(0, 0, 0)] * (1.0 - voxels[voxelIndex(1, 0, 0)].w))) +
              (voxels[voxelIndex(1, 1, 0)] + (voxels[voxelIndex(0, 1, 0)] * (1.0 - voxels[voxelIndex(1, 1, 0)].w))) +
              (voxels[voxelIndex(1, 0, 1)] + (voxels[voxelIndex(0, 0, 1)] * (1.0 - voxels[voxelIndex(1, 0, 1)].w))) +
              (voxels[voxelIndex(1, 1, 1)] + (voxels[voxelIndex(0, 1, 1)] * (1.0 - voxels[voxelIndex(1, 1, 1)].w)))) * 0.25;
}
{
  // -y
  fetchVoxels(voxels, VoxelVolumeTexture1, uvw, level);
  oOutput4 = ((voxels[voxelIndex(0, 1, 0)] + (voxels[voxelIndex(0, 0, 0)] * (1.0 - voxels[voxelIndex(0, 1, 0)].w))) +
              (voxels[voxelIndex(1, 1, 0)] + (voxels[voxelIndex(1, 0, 0)] * (1.0 - voxels[voxelIndex(1, 1, 0)].w))) +
              (voxels[voxelIndex(0, 1, 1)] + (voxels[voxelIndex(0, 0, 1)] * (1.0 - voxels[voxelIndex(0, 1, 1)].w))) +
              (voxels[voxelIndex(1, 0, 1)] + (voxels[voxelIndex(1, 0, 1)] * (1.0 - voxels[voxelIndex(1, 1, 1)].w)))) * 0.25;
}
{
  // -z
  fetchVoxels(voxels, VoxelVolumeTexture2, uvw, level);
  oOutput5 = ((voxels[voxelIndex(0, 0, 1)] + (voxels[voxelIndex(0, 0, 0)] * (1.0 - voxels[voxelIndex(0, 0, 1)].w))) +
              (voxels[voxelIndex(1, 0, 1)] + (voxels[voxelIndex(1, 0, 0)] * (1.0 - voxels[voxelIndex(1, 0, 1)].w))) +
              (voxels[voxelIndex(0, 1, 1)] + (voxels[voxelIndex(0, 1, 0)] * (1.0 - voxels[voxelIndex(0, 1, 1)].w))) +
              (voxels[voxelIndex(1, 1, 1)] + (voxels[voxelIndex(1, 1, 0)] * (1.0 - voxels[voxelIndex(1, 1, 1)].w)))) * 0.25;
}
oOutput6 = mipMapVoxelColor(VoxelVolumeTexture6, uvw, level);
```

More different possible diffuse cone configurations:

```
#define NUM_CONES 1
const vec3 coneDirections[1] = vec3[1](
                                 vec3(0.0, 0.0, 1.0)
                               );
const float coneWeights[1] = float[1](
                               1.0
                             );
const float coneApertures[1] = float[1]( // tan(63.4349488)
                                 2.0
                               );
```

More different possible diffuse cone configurations:

```
#define NUM_CONES 6
const vec3 coneDirections[6] = vec3[6](
                                 normalize(vec3(0.0, 0.0, 1.0)),
                                 normalize(vec3(-0.794654, 0.607062, 0.000000)),
                                 normalize(vec3(0.642889, 0.607062, 0.467086)),
                                 normalize(vec3(0.642889, 0.607062, -0.467086)),
                                 normalize(vec3(-0.245562, 0.607062, 0.755761)),
                                 normalize(vec3(-0.245562, 0.607062, -0.755761))
                               );
const float coneWeights[6] = float[6](
                               1.0,
                               0.607,
                               0.607,
                               0.607,
                               0.607,
                               0.607
                             );
const float coneApertures[6] = float[6](
                                 0.5,
                                 0.549092,
                                 0.549092,
                                 0.549092,
                                 0.549092,
                                 0.549092
                               );
```

More different possible diffuse cone configurations:

```
  #define NUM_CONES 6
  const vec3 coneDirections[6] = vec3[6](
                                   vec3(0.0, 0.0, 1.0),
                                   vec3(0.0, 0.866025, 0.5),
                                   vec3(0.823639, 0.267617, 0.5),
                                   vec3(0.509037, -0.700629, 0.5),
                                   vec3(-0.509037, -0.700629, 0.5),
                                   vec3(-0.823639, 0.267617, 0.5)
                                 );
  const float coneWeights[6] = float[6](
#if 0
                                 3.14159 * 0.25,
                                 (3.14159 * 3.0) / 20.0,
                                 (3.14159 * 3.0) / 20.0,
                                 (3.14159 * 3.0) / 20.0,
                                 (3.14159 * 3.0) / 20.0,
                                 (3.14159 * 3.0) / 20.0
#else
                                 0.25,
                                 0.15,
                                 0.15,
                                 0.15,
                                 0.15,
                                 0.15
#endif
                               );
  const float coneApertures[6] = float[6]( // tan(30)
                                   0.57735026919,
                                   0.57735026919,
                                   0.57735026919,
                                   0.57735026919,
                                   0.57735026919,
                                   0.57735026919
                                 );
```

More different possible diffuse cone configurations:

```
#define NUM_CONES 8
const vec3 coneDirections[8] = vec3[8](
                                  vec3(0.57735, 0.57735, 0.57735),
                                  vec3(-0.57735, -0.57735, 0.57735),
                                  vec3(-0.903007, 0.182696, 0.388844),
                                  vec3(0.903007, -0.182696, 0.388844),
                                  vec3(0.388844, -0.903007, 0.182696),
                                  vec3(-0.388844, 0.903007, 0.182696),
                                  vec3(-0.182696, 0.388844, 0.903007),
                                  vec3(0.182696, -0.388844, 0.903007)
                                );
const float coneWeights[8] = float[8](
                               1.0 / 8.0,
                               1.0 / 8.0,
                               1.0 / 8.0,
                               1.0 / 8.0,
                               1.0 / 8.0,
                               1.0 / 8.0,
                               1.0 / 8.0,
                               1.0 / 8.0
                             );
const float coneApertures[8] = float[8](
                                 0.4363325,
                                 0.4363325,
                                 0.4363325,
                                 0.4363325,
                                 0.4363325,
                                 0.4363325,
                                 0.4363325,
                                 0.4363325
                               );
```

More different possible diffuse cone configurations:

```
#define NUM_CONES 16
const vec3 coneDirections[16] = vec3[16](
                                  vec3(0.898904, 0.435512, 0.0479745),
                                  vec3(0.898904, -0.0479745, 0.435512),
                                  vec3(-0.898904, -0.435512, 0.0479745),
                                  vec3(-0.898904, 0.0479745, 0.435512),
                                  vec3(0.0479745, 0.898904, 0.435512),
                                  vec3(-0.435512, 0.898904, 0.0479745),
                                  vec3(-0.0479745, -0.898904, 0.435512),
                                  vec3(0.435512, -0.898904, 0.0479745),
                                  vec3(0.435512, 0.0479745, 0.898904),
                                  vec3(-0.435512, -0.0479745, 0.898904),
                                  vec3(0.0479745, -0.435512, 0.898904),
                                  vec3(-0.0479745, 0.435512, 0.898904),
                                  vec3(0.57735, 0.57735, 0.57735),
                                  vec3(0.57735, -0.57735, 0.57735),
                                  vec3(-0.57735, 0.57735, 0.57735),
                                  vec3(-0.57735, -0.57735, 0.57735)
                                );
const float coneWeights[16] = float[16](
                                1.0 / 16.0,
                                .
                                .
                                .
                                1.0 / 16.0
                              );
const float coneApertures[16] = float[16](
                                  0.3141595,
                                  .
                                  .
                                  .
                                  0.3141595
                               );
```

Demo time!

Questions?

Thank you!