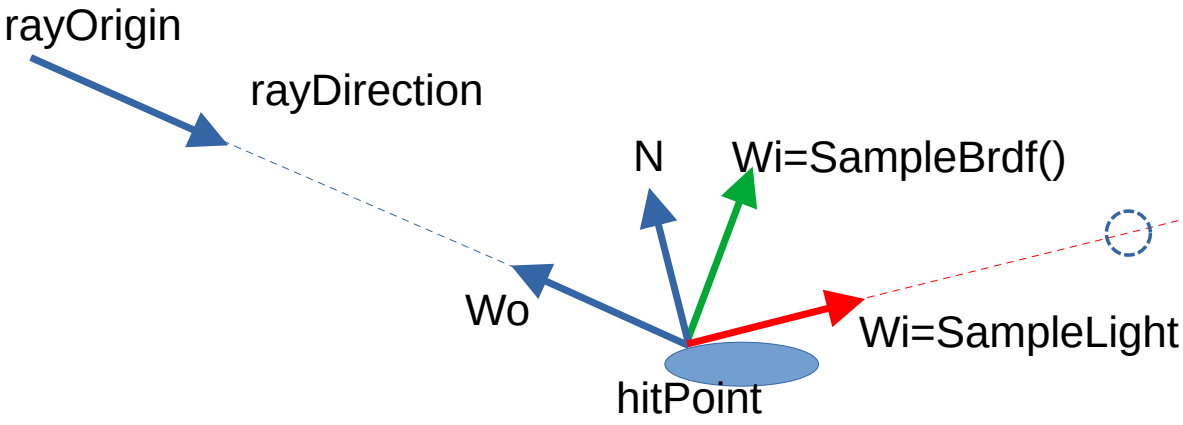
Project 4a: Full path tracing

Synopsis

Path Tracing is a relatively small step beyond Ray Casting. In Ray Casting, the color computed for a pixel we the result of tracing a single ray from the eye to the first object hit. In path Tracing that single ray becomes a PATH of rays traced from the eye out into the scene through multiple bounces until a light is encountered (or a Russian-rouelette test gives up).

The path tracing algorithm is very generic Monte-Carlo accumulation loop. It depends on three functions **EvalBrdf**, **SampleBrdf**, and **PdfBrdf** for all interactions with the surfaces in the scene.

There is almost no Vulkan programming here, only GLSL in the ray-generation shader, and a bit of C++ code to feed some new push-constants values into the shader.



Instructions

Push constants: Create several new fields in the **PushConstantRay** structure (carefully following the alignment rules) and initialize them in **VkApp::raytrace**:

- To ensure each frame is independent of previous frames, choose a frame-specific random number:
  m_pcRay.frameSeed = rand() % 32768;
- Choose a russian-roulette threshold:
  m_pcRay.rr = 0.7;
- Precomputed maximum depth based on the **m_pcRay.rr** value:
  m_pcRay.depth = 1;
  while (float(rand())/RAND_MAX < m_pcRay.rr) m_pcRay.depth++;
  - On rare occasions, execution of the path tracing step can take long enough to cause a skipped refresh, producing a "ragged" feeling refresh rate. If so, cap **m_pcRay.depth** to 4 or less. This breaks the Monte-Carlo property of **guaranteed convergence**, but perhaps that's OK for a realtime setting.
- Set a variable to tell the ray generation shader if the calculated color value should accumulated with previous values, or used to initialize for future accumulations:
  m_pcRay.clear = app→myCamera.modified;
  app->myCamera.modified = false;

Remove temporary light values: The several push constant variables named **temp...** having to do with a global light and the ambient light can now be removed. The path tracing algorithm will seek out lights in the scene's data, so these temporary sources of light are no longer necessary.

Random numbers in GLSL:

Notice the **#include RNG.glsl** iin the ray generation shader. This supplies two functions which compute random numbers in GLSL: **tea** (Tiny Encryption Algorithm) and **rnd()**.

We provide a seed for this random number generator so as to ensure there is no correlation between the pixel and neighboring pixels, nor this pixel in successive frames. This seed is stored in the ray **payload**, so as a path is generated ray-by-ray, the seed follows with it. The **payload** is defined in **shaders/shared_structs.h** and initialized in **raytrace.rgen**:

```
payload.seed = tea(  gl_LaunchIDEXT.y * gl_LaunchSizeEXT.x + gl_LaunchIDEXT.x,
                     pcRay.frameSeed          );
```

BRDF

Implement these three functions which the Monte-Carlo algorithm uses as its interface with the scene. Details are on later pages. We choose here to **importance-sample** the diffuse reflection. A great number of other choices exist.

- **vec3 EvalBrdf(vec3 N, vec3 L, vec3 V, Material mat) :** Evaluates the non-light portions of the BRDF lighting equation), either in full, or just the diffuse portion.

$$(N\cdot L)_+\left[\frac{K_d}{\pi} + \frac{D()\,F()\,G()}{4\;N\cdot L\;N\cdot V}\right]$$

- v**ec3 SampleBrdf(inout uint seed, in vec3 N) :** Produces a random sample direction centered around N and distributed according to the **PdfBrdf** function

  return SampleLobe(N, sqrt(rnd(seed)), 2π*rnd(seed));

- **float PdfBrdf(vec3 N, vec3 Wi)** :
  return $|N\cdot Wi|/\pi$

Keep a running average

The output of the ray casting was a single color written to the output image. The output of path tracing is the running average of many such calculations. Instead of tracking a running sum and a count (for later calculation of average=sum/N), we can track the running average (and count) directly.

We can store the running average and the count in a vec4: average in .xyz and count in .w;

To fold in a new value C, calculate the old sum, add the new value, and divide by the new count. This recipe can be manipulated into a easy and numerically stable formula:

In Pseudo-code:

```
Retrieve (Ave, N) from colCurr
// Ave  =  (Ave*N + C)/(N+1); // What you want, but
Ave  += (C – Ave)/(N+1) // Faster and more numerically stable.
Store (Ave, N+1) to colCur
```

In GLSL: Retrieve the accumulated value with

```
vec4 old = imageLoad(colCurr, ivec2(gl_LaunchIDEXT.xy));
Ave = old.xyz
N = old.w
```

and store the new value with

```
imageStore(colCurr, ivec2(gl_LaunchIDEXT.xy), vec4(Ave,N+1));
```

The Monte-Carlo algorithm

Implemented in the ray generation shader **raytrace.rgen.**
Text in **<...>** brackets is a direct reference to code from the ray casting step.

```
<ALL the layout portions unchanged>
main() {
   payload.sead = ...
   <Calculation of rayOrigin and rayDirection>
   C = (0,0,0) : Pixel color, accumulated color along the path; output at the end
   W = (1,1,1) : Accumulation of f/p weights along path; multiplied by light

   // Loop ray-by-ray along a path:
   loop for pcRay.depth iterations:
      <Call traceRayEXT to fire the ray; the hit and miss shaders will fill in the payload>

      if (!payload.hit)  break;  // Path escaped the scene – break from loop

      <Dereference objDesc, calc vertices ..., mat, nrm, uv;  read texture>

      if material mat is a light: // test with: dot(mat.emission,mat.emission) > 0.0
         C += mat.emission*W;
         break;

      <FUTURE: This is where the EXPLICIT light connection will occur>

      // From the current hit point, setup N,L,V for BRDF calculation
      P = payload.hitPos;  // Current hit point
      N = normalize(nrm);  // Its normal
      // Wi and Wo play the same role as L and V, in most presentations of BRDF
      // … but makes more sense then L and V notation in the middle of a long path
      Wi = SampleBrdf(payload.seed, N); // Importance sample output direction
      Wo = -rayD;

      f = EvalBrdf(N, Wi, Wo, mat);      // Color (vec3) according to BRDF
      p = PdfBrdf(N,Wi)*pcRay.rr;        // Probability (float) of above sample of Wi
      if (p < epsilon)  // epsilon = 10^-6;
         break;
      W *= f/p;          // Monte-Carlo estimator

      // Step forward for next loop iteration
      rayOrigin = payload.hitPos;
      rayDirection = Wi;
      // End of loop

// Accumulate (or not) C into output image
if (pcRay.clear)
   write C,1 to colCurr image; initializing the accumulation
else
    accumulate C into running average found in colCurr (see above)
```

BRDF lighting synopsis:

$$I_o = I_i (N \cdot L)_+ \, BRDF$$

where the BRDF portion is:

$$BRDF = \frac{K_d}{\pi} + \frac{D(H)\ F(L, H)\ G(L, V, H)}{4 (L \cdot N)(V \cdot N)}$$

and a nice **starter set** for D, F, and G is:

$$D(H) = \frac{\alpha+2}{2\pi}(N \cdot H)^{\alpha}$$

$$F(L, H) = K_s + (1 - K_s)(1 - L \cdot H)^5$$

$$\frac{G(L, V, H)}{(L \cdot N)(V \cdot N)} \approx \frac{1}{(L \cdot H)^2}$$

(See later pages for more detailed versions of these functions.)

Auxiliary function SampleLobe:

To choose a direction vector cosine-distributed around a given vector A

Here, c specifies the cosine of the angle between the returned vector and A, while ϕ gives an angle around A.

SampleLobe(A, c , ϕ **)**

$s = \sqrt{(1 - c^2)}$

// Create vector K centered around Z-axis and rotate to A-axis

$K = (s \cos\phi,\ s \sin\phi,\ c)$ **// Vector centered around Z-axis**

if $|A_z - 1| < 10^{-3}$: return K **// A=Z so no rotation**

if $|A_z + 1| < 10^{-3}$: return $(K_x, -K_y, -K_z)$ **// A=-Z so rotate 180 around X axis**

~~A = normalize(A)~~ **// Not needed if you know A is unit length**

B = normalize($(-A_y, A_x, 0)$) **// Z x A**

C = $A \times B$

return $K_x B + K_y C + K_z A$

Microfacet Models for Refraction through Rough Surfaces

https://faculty.digipen.edu/~gherron/references/References/BRDF/EGSR07-btdf.pdf

All microfacet BRDFs have this general form:

$$\frac{K_d}{\pi} + \frac{D(m)\,G(\omega_i,\omega_o,m)\,F(\omega_i\cdot m)}{4\left|\omega_i\cdot N\right|\left|\omega_o\cdot N\right|}$$

Where

- $m = (\omega_o+\omega_i)/\|\omega_o+\omega_i\|$ is the half-vector.
- K_d is the diffuse reflection (albedo) of the surface
- K_s is the specular reflection in the $\omega_i=\omega_o=N$ direction,
- α_p, α_b, α_g are the surface roughness values for **Phong**, **Beckman** and **GGX** respectively.

The paper lists three common versions for **D** and **G: Phong**, **Beckman** and **GGX**.
The easiest is **Phong**, but recently, graphics has been trending toward **GGX**.

Characteristic factor:

The so called characteristic function in the **D** and **G** factors below is defined as

$$\chi^+(d) = \begin{cases} 1 & \text{if } d>0 \\ 0 & \text{if } d\le 0 \end{cases}$$

and implemented with a simple "if" statement.

F factor

F is the Fresnel (reflection) is usually approximated by Schlick as

$$F(d) = K_s + (1-K_s)(1-|d|)^5$$

where K_s is the specular reflection color at $L=V=N=H$.

The exact formulation (if you are interested) is

$$F(L,H) = \frac{1}{2}\,\frac{(g-c)^2}{(g+c)^2}\left(1+\frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right)$$

where

$$g = \sqrt{\frac{\eta_t^2}{\eta_i^2} - 1 + c^2}$$

and

$$c = |L\cdot H|$$

and η_i and η_t are indices of refraction of the two materials.

D factor

D is the micro-facet distribution.

In the following, $\tan\theta_m = \sqrt{(1.0 - (m \cdot N)^2)} \,/\, (m \cdot N)$ and $\tan\theta_v = \sqrt{(1.0 - (v \cdot N)^2)} \,/\, (v \cdot N)$.

Phong: $$D_p(m) = \chi^+(m \cdot N) \; \frac{\alpha_p + 2}{2\pi} (m \cdot N)^{\alpha_p}$$

(α_p : 1..∞ ; increasing means smoother surface)

Beckman: $$D_b(m) = \chi^+(m \cdot N) \; \frac{1}{\pi \alpha_b^2 (N \cdot m)^4} \; e^{\frac{-\tan^2\theta_m}{\alpha_b^2}}$$

(α_b : 0..1 ; increasing means rougher surface)

similar to Phong for smooth surfaces using $\alpha_p = 2\alpha_b^{-2} - 2$

GGX: $$D_g(m) = \chi^+(m \cdot N) \; \frac{\alpha_g^2}{\pi (N \cdot m)^4 (\alpha_g^2 + \tan^2\theta_m)^2}$$

G factor:

The G term should be calculated via the smith method:

$$G(\omega_i, \omega_o, m) = G_1(\omega_i, m) \; G_1(\omega_o, m)$$

where $G_1(...)$:

Beckman:

$$G_1(v, m) = \chi^+\left(\frac{v \cdot m}{v \cdot N}\right) \begin{cases} \dfrac{3.535a + 2.181a^2}{1.0 + 2.276a + 2.577a^2} & \text{if } a < 1.6 \\ 1 & \text{otherwise} \end{cases}$$

where $a = 1/(\alpha_b \tan\theta_v)$, and $\tan\theta_v$ is defined above.

Phong:

Same G_1 as **Beckman**, but with $a = (\sqrt{\alpha_p/2 + 1}) \,/\, \tan\theta_v$.

GGX:

$$G_1(v, m) = \chi^+\left(\frac{v \cdot m}{v \cdot N}\right) \frac{2}{1 + \sqrt{1 + \alpha_g^2 \tan^2\theta_v}}$$

Beware round off errors in calculating the G_1 function:

- The value of $(v \cdot N)$ may round up to greater than 1.0 (mathematically it shouldn't, but computationally it sometimes does). If so, return $G_1(...) = 1.0$.
- The calculation of $\tan\theta_v$ may be zero. If so, don't divide by it, instead return $G_1(...) = 1.0$.