Path tracing

Start things off

```
vkCmdBindPipeline(cmdBuf, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, m_rtPipeline);
vkCmdBindDescriptorSets(cmdBuf, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, ...);
vkCmdPushConstants(cmdBuf, m_rtPipelineLayout,, ...);
vkCmdTraceRaysKHR(cmdBuf, ..., m_size.width, m_size.height, 1);
```

General outline:

This is the high level description of the pathtracing algorithm (with so called next-event estimation). The details will be introduced in phases following this.

```
TracePath(Ray ray):
    C = (0,0,0) // Accumulated light

    // Initial ray
    P = Trace ray into the scene to first intersection  // P records: object, distance, normal ...
    if P is no-intersection:  return C  // Which is still (0,0,0)
    if P is a light: return Radiance(P)  // Light objects must provide a radiance method

    while Russian Roulette Test passes:
        // Explicit light connection  (OPTIONAL)
        Choose a random point on a (random) light
        Generate a ray from P to a (random) point on a (random) light  // Called a shadow ray
        L = Trace that ray into scene to first intersection
        if L exists and is the chosen point on the chosen light:
            C += (½) * chosen light's contribution

        // Extend path
        Generate a new ray from P in some random direction (use importance sampling)
        Q = Trace that new ray into the scene to first intersection
        if Q is no-intersection: break

        // Implicit light connection
        if Q is a light:
            C +=  (½) * light Q's contribution;
            break

        // Step forward
        P ← Q

    return C
```

Details: Phase 1 (only black text), Phase 2 (add red text)

```
TracePath(Ray ray):
    C = (0,0,0) // Accumulated light
    W = (1,1,1) // Accumulated weight
    // Initial ray
    P = Trace ray into the scene  // Intersection points must record: object, distance, normal ...
    N = P's normal
    if P indicates no intersection:  return C // which is (still (0,0,0)
    if P is a light: return EvalRadiance(P) // Light objects must provide a radiance method

    while random() <= RussianRoulette: // 0.8 is a good value for RussianRoulette
        // Explicit light connection
        L = SampleLight()      // Randomly choose a light and a point on that light.
        p = PdfLight(L)/GeometryFactor(P, L) // Probability of L, converted to angular measure
        ω_i = direction from P  toward L
        I = Trace ray from P toward L    // This is called a shadow-ray
        if p>0 and I exists and is the chosen point on the chosen light:
            f = EvalScattering(N, ω_i )
            C +=  (½) * W * f/p * EvalRadiance(L)

        // Extend path
        N = <P's> normal
        ω_i = <P's>SampleBrdf(N)  // Choose a sample direction from P
        Q = Trace ray from P in direction  ω_i  into the scene
        if Q is non-existent: break
        f = <P's>EvalScattering(N, ω_i )
        p = <P's>PdfBrdf(N,  ω_i ) * RussianRoulette
        if p < ε : break  // Avoid division by zero or nearly zero:  ε=10^-6
        W *= f/p

        // Implicit light connection
        if Q is a light:
            C +=  (½) * W * EvalRadiance(Q)
            break

        // Step forward
        P ← Q
        N = P's normal
    return C
```

Functions and values used in the algorithm

In the previous statements of the path-tracing algorithm, the following quantities are used:

- P, Q are "intersection records" containing a point, a t value, a normal, and an object.
- An object contains a Brdf (for reflective objects) or a light (for light objects).
- A Brdf contains the usual lighting parameters K_d, K_s, and α and the three methods shown below.
- A light contains a radiance (RGB value) and the three methods shown below.

An emissive object's light methods

This is for only spherical light objects. Other shapes are possible (and easy).

For light objects, we must sample a random point on a random light with a known probability. I choose (uniformly) one light with probability $1/NumberOfLights$, and on that light choose a uniformly distributed point with probability $1/AreaOfLightSphere$. (Smarter choices exist – for instance chose brighter/larger/closer lights with a higher probability.)

***SampleLight*()**

Choose one light (uniformly) randomly.

Choose a uniformly distributed point on the light. (see **SampleSphere** below)

Create and return an "intersection record" with the light, point, and its normal

***PdfLight*(L)**

return $1/(L \rightarrow AreaOfLightSphere() * NumberOfLights)$ // Area of a sphere is $4\pi r^2$

***EvalRadiance*(L)**

return the RGB radiance of the light

A reflective object's BRDF methods

For reflective objects we sample a direction with a distribution that matches the ($N \cdot \omega_i$) term, thereby spending more time probing directions at high angles and less time probing low-angel glancing directions. This is out first example of ***importance sampling***.

***SampleBrdf*(N)**

Choose ξ_1, ξ_2 two uniformly distributed random numbers in $[0,1]$.

return $\omega_i = SampleLobe(N, \sqrt{\xi_1}, 2\pi\xi_2)$

***PdfBrdf*(N, ω_i)**

return $|N \cdot \omega_i|/\pi$

***EvalScattering*(N, ω_i)**

return $|N \cdot \omega_i| \, K_d/\pi$ // Diffuse term. Full BRDF will be implemented in a later project.

Auxiliary functions

Convert between angular measure and area measure

GeometryFactor(A,B) /

// A and B are two intersection records with points A_P, B_P, and normals A_N, B_N

$D = A_P - B_P$;

return $\left|(A_N \cdot D)(B_N \cdot D) \ / \ (D \cdot D)^2\right|$

Choose a direction vector 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 can assume A is unit length**

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

C = $A \times B$

return $K_x B + K_y C + K_z A$

//~~Quaternionf q = Quaternionf::FromTwoVectors(Vector3f::UnitZ(),N);~~ ~~**// q rotates Z to N**~~

//~~return q._transformVector(K);~~ ~~**// K rotated to N's frame**~~

Choose a uniformly distributed point on a sphere with center C and radius R:

SampleSphere(C, R)

ξ_1, ξ_2 = two uniform ramdom numbers in $[0,1]$

$z = 2\xi_1 - 1$

r = $\sqrt{(1 - z^2)}$

a = $2\pi \ \xi_2$

Return an intersection record with

$N = (r \cos(a), r \sin(a), z)$

$P = C + R \ N$

object = this sphere