Project 4b: Explicit light connection (optional)

Here, we add an explicit connection to a random light into the middle of the path tracing loop. The usual naming convention for this is “Monte-Carlo with Next Event Estimation”.

```
for (int i=0; i<pcRay.depth;  i++) {
        traceRayEXT(...);
        <If nothing is hit break out of loop>
        <If something is hit, find the object data (Important: mat, nrm, hitpoint)>
        <If the hitPoint is a light:>
                C += (½) *  mat.emission * W;  // Notice the new ½ here.
                break;

      // @@ Explicit light connection goes here.
      // <light's info> = SampleLight() // choose a random point on random light L
      //                              returning point and light's area, normal, and emission
      // Wi = normalize(<light's point> - payload.hitPos);
      // dist = length(<light's point> - payload.hitPos);  // Distance to light's point
      // payload.hit = true; // Miss shader may set this to false

      traceRayEXT(topLevelAS,          // acceleration structure
               gl_RayFlagsOpaqueEXT
               | gl_RayFlagsTerminateOnFirstHitEXT
               | gl_RayFlagsSkipClosestHitShaderEXT, // rayFlags
               0xFF,                 // cullMask
               0,                    // sbtRecordOffset
               0,                    // sbtRecordStride
               0,                    // missIndex
               payload.hitPos, // ray origin !!!!
               0.001,                // ray min range
               Wi,                   // ray direction !!!!
               dist-0.001,       // ray max range !!!!
               0                     // payload (location = 0)
               );

      // if !payload.hit  // this means the ray hit the light
      //     N = normalize(nrm);  // HitPoint's normal
      //     Wo = -rayD;
      //     f = EvalBrdf(N, Wi, Wo, mat);
      //     p = PdfLight(<light's info>)/GeometryFactor(payload.hitPos, N,
      //                                       <light's point>, <light's normal>);
      //     C += (½) * W * f/p * EvalLight(<light's info>);
      // @@ End of explicit light connection

    ...
 }
```

Functions for the Monte-Carlo algorithm to interface with light emitters

***SampleLight*()**

Choose one uniformly distributed random light L (from the list of emitting triangles).
Choose one uniformly distributed random point P on L. (see **SampleTriangle**)
Suggested return structure:

```
struct PointOnLight {
    vec3 P;             // Point on light
    vec3 N;             // Normal at P
    float A;            // Area of triangle
    vec3 emission;      // Emission of light
    uint index; };      // Index of light in light-list
```

***PdfLight*(L)**

return $\frac{1}{(AreaOfTriangle()*NumberOfLights)}$

***EvalLight*(L)**

return the emission (radiance) of the light triangle

SampleTriangle(A, B, C)

$b_2 = rnd(seed)$
$b_1 = rnd(seed)$
$b_0 = 1 - b_1 - b_2$
$if\ b_0 < 0$ // Test for outer triangle; If so invert into inner triangle
$b_1 = 1 - b_1$
$b_2 = 1 - b_2$
$b_0 = 1 - b_1 - b_2$
return $b_0 A + b_1 B + b_2 C$

GeometryFactor(Pa, Na, Pb, Nb)

$D = Pa - Pb$
return $\left|\frac{(D\cdot Na)(D\cdot Nb)}{(D\cdot D)^2}\right|$ // Note the absolute value.

Vulkan code fragments

An earlier project had you create a list of lights, i.e., triangles with non-zero emission . Here are instructions to

1. **Create that list with the necessary information**
2. **Wrap that list in a Vulkan Buffer**
3. **Pass that buffer into the ray generation shader via an existing descriptor set**
4. **Access that data during the explicit light calculation**

1. Create the emitter list (you may have already done this):

In **VkApp::myloadModel** build an Emitter list, say

```
std::vector<Emitter> emitterList;
```

with a structure like this defined in **shared_struct.h**

```
struct Emitter
{
    vec3 v0, v1, v2;        // Vertices of light emitting triangle
    vec3 emission;          // Its emission
    vec3 normal;            // its normal
    float area;             // Its triangle area.
    uint index;             // the triangles index in the model's list of triangles
};
```

As a reminder, here is how to build that list:

```
// Loop through all triangles
for (uint i=0;  i<meshdata.matIndx.size();  i++) {
    // Get triangle i's material
    Material& mat = meshdata.materials[meshdata.matIndx[i]];
    // Test if triangle i is an emitter
    if (glm::dot(mat.emission,mat.emission) > 0.0f) {
            // Retrieve the triangle's vertices:
            // v0 = meshdata.vertices[meshdata.indicies[3*i+0]].pos;
            // v1 =                                             ... 1 ...
            // v2 =                                             ... 2 ...
            // emission = mat.emission;  // May need to scale this up by a factor of 4 or 5
            // index = i
            // normal and area calculated from vertices v0, v1, and v2
```

Notes:

$$NormalOfTriangle(A,B,C) = normalize(\ (B-A)\times(C-A)\ )$$

$$AreaOfTriangle(A,B,C) = |(B-A)\times(C-A)|\ /\ 2$$ // Here, absolute value means vector length

2. Wrap as a Vulkan buffer

In Vkapp.h declare:

```
BufferWrap m_lightBuff{};
```

and destroy it along with everything else:

```
m_lightBuff.destroy(m_device);
```

Fill the buffer with the emitter list:

```
m_lightBuff = createBufferWrap(sizeof(emitterList[0])*emitterList.size(),
                               VK_BUFFER_USAGE_STORAGE_BUFFER_BIT
                               | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
                               VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);

VkCommandBuffer commandBuffer = createTempCmdBuffer();
vkCmdUpdateBuffer(commandBuffer, m_lightBuff.buffer, 0,
                  sizeof(emiterList[0])*emitterList.size(), emitterList.data());
submitTempCmdBuffer(commandBuffer);
```

3. Add the buffer to the ray tracing descriptor set:

In **VkApp::createRtDescriptorSet** add a new binding to **rtDesc.setBindings**:

**{2, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1,
VK_SHADER_STAGE_RAYGEN_BIT_KHR|VK_BUFFER_USAGE_STORAGE_BUFFER_BIT}**

and a new write call below that:

m_rtDesc.write(m_device, 2, m_lightBuff.buffer);

4. Access the emitter list in the ray generation shader:

In the shader raytrace.rgen add:

layout(set=0, binding=2, scalar) buffer _emitter { Emitter list[]; } emitter;

Access the length of the list with

emitter.list.length()

and the i'th emitter on the list with

emitter.list[i]