1 Hierarchical Russian Roulette

Program 1 shows an example C++ implementation of hierarchical Russian roulette. Although Algorithm 1 in the paper uses the recursion, this implementation uses the iterative form for performance reasons. For simplicity, we abstract the implementation using the classes and math functions shown in Table 1 and Table 2. Russian roulette and vertex connection at a leaf node are performed in the Shade function. The input parameter lobe is $C_q(\omega)$ in the paper.

Optimization of Random Number Generations. Since hierarchical Russian roulette generates two random numbers at each internal node, the cost of random number generation has an impact on the performance. Therefore, we use a fast 64 bit random number generator such as xoroshiro128+ [BV18]. To generate two random numbers rapidly, the upper and lower bits of a 64 bit random number are used as two random numbers. In Program 1, this is performed by `rng.GenerateUintAndFloat()` which returns a 32 bit unsigned integer random number and single precision floating point random number. Unlike the Algorithm 1 in the paper, the unsigned integer random number is used for the random selection of the child node to avoid the precision error of floating point arithmetic.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float3</td>
<td>3D vector</td>
</tr>
<tr>
<td>float3x3</td>
<td>3×3 matrix</td>
</tr>
<tr>
<td>floatvec4</td>
<td>four-wide SIMD vector</td>
</tr>
<tr>
<td>float3vec4</td>
<td>3D vector (four-wide SIMD vector for each element)</td>
</tr>
<tr>
<td>float3x3vec4</td>
<td>3×3 matrix (four-wide SIMD vector for each element)</td>
</tr>
<tr>
<td>RNG</td>
<td>Random number generator</td>
</tr>
<tr>
<td>BVH</td>
<td>Bounding volume hierarchy of light-subpath vertices</td>
</tr>
<tr>
<td>EyeVertex</td>
<td>Eye-subpath vertex</td>
</tr>
<tr>
<td>LightVertex</td>
<td>Light-subpath vertex</td>
</tr>
<tr>
<td>SEL</td>
<td>Squared ellipsoidal lobe</td>
</tr>
</tbody>
</table>

Table 2: Math functions used in our implementation. These functions emulate the same intrinsic functions available in HLSL.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot</td>
<td>Dot product of two 3D vectors</td>
</tr>
<tr>
<td>cross</td>
<td>Cross product of two 3D vectors</td>
</tr>
<tr>
<td>normalize</td>
<td>Normalization of a 3D vector</td>
</tr>
<tr>
<td>max</td>
<td>Maximum value for each dimension</td>
</tr>
<tr>
<td>abs</td>
<td>Absolute value for each dimension</td>
</tr>
<tr>
<td>mul</td>
<td>Matrix multiplication</td>
</tr>
</tbody>
</table>

*yusuke.tokuyoshi@gmail.com
2 Fast Conservative Ellipsoid-Box Intersection Test

2.1 Rough Intersection Test

Since the exact intersection test between an ellipsoid and box is expensive, we use a conservative rough intersection test. Our implementation stretches the test space to transform the ellipsoid into a sphere whose inverse squared radius is the minimum random number, and computes a bounding box for the stretched box (i.e., parallelepiped) in this test space. Thus, the expensive ellipsoid-box intersection test is replaced with a simple sphere-box intersection test [Arv90]. The transformation from the world-space to the test space is represented by the following 3×3 matrix:

\[
A = \begin{bmatrix}
\frac{1}{r_x} & 0 & 0 \\
0 & \frac{1}{r_y} & 0 \\
0 & 0 & \frac{1}{r_z}
\end{bmatrix},
\]

\[
Q = \begin{bmatrix}
\omega_x/r_x \\
\omega_y/r_y \\
\omega_z/r_z
\end{bmatrix},
\]

where \([r_x, r_y, r_z]\) are the semiaxes of the ellipsoid, and \(Q = [\omega_x^T \quad \omega_y^T \quad \omega_z^T]^T\) is the orthogonal matrix to represent the orientation of the ellipsoid. In Program 1, \(A\) is ellipsoidMatrix. The variable \(\text{center}\) in Program 1 is the relative center position of the ellipsoid from the eye vertex.

2.2 Orientation for a Tighter Bounding Box

To make a tighter bounding box for the stretched box, the test space is oriented additionally. For a parallelogram in a 2D space shown in Fig. 1, we can consider two oriented bounding boxes (OBBs) by aligning the axis of the OBB to one edge of the parallelogram. Therefore, for a parallelepiped which consists three different parallelograms in a 3D space, six OBBs can be considered. For efficiency, our implementation uses the same orientation for all nodes. We can select one orientation based on the edge lengths of a stretched unit cube.

Figure 1: Two tight OBBs for a parallelogram.

2.3 More Accurate Intersection Test Using SIMD Optimization

For a more accurate intersection test, we can consider the intersection of the OBBs. For this, intersection test is performed for each oriented test space. To accelerate these multiple intersection tests, we utilize four-wide SIMD instructions on the CPU. Our implementation selects four most dominant orientations, and computes the four intersection tests at once in a SIMD fashion. Program 2 is our intersection test using SIMD optimization, and Program 3 is the selection of orientations for this SIMD optimization. Optimization using eight-wide SIMD instructions is a future work.

References

Program 1: Example code (C++) of hierarchical Russian roulette.

```cpp
float3 HierarchicalRussianRoulette(RNG& rng, const BVH& bvh, const EyeVertex& eyeVertex, const SEL& lobe) {
    // Get the ellipsoid represented by the 3x3 matrix and center position.
    const auto [ellipsoidMatrix, center] = lobe.GetEllipsoid();

    // Transform the space of the intersection test for tightening the AABB of a BVH node.
    // 4 transformation patterns are used for our SIMD-optimized intersection test.
    const float3x3vec4 transformation = AlignTransformationMatrix(ellipsoidMatrix);

    struct StackElement { uint32_t nodeIndex; float randomNumberMin; float infimum; }
    std::array<StackElement, STACK_SIZE> stack;
    uint_fast8_t stackLevel = 0;
    uint32_t nodeIndex = 0;
    float randomNumberMin = rng.GenerateFloat() / bvh.GetLeafCount();
    float infimum = 1.0f / bvh.GetLeafCount();
    float3 result = 0.0f;

    while(true) {
        if(BVH::IsInnerNode(nodeIndex)) {
            const BVH::Node& childL = bvh.GetNode(nodeIndex);
            const BVH::Node& childR = bvh.GetNode(nodeIndex + 1);
            const auto [randomNumberUint, randomNumberFloat] = rng.GenerateUintAndFloat();
            const uint32_t totalLeafCount = childL.leafCount + childR.leafCount;
            const bool transmitToLeft = randomNumberUint % totalLeafCount < childL.leafCount;
            const int32_t leafCount = transmitToLeft ? childR.leafCount : childL.leafCount;
            const float stratumSize = (1.0f - infimum) / leafCount;
            const float supremum = infimum + stratumSize;
            const float newRandomNumberMin = infimum + stratumSize * randomNumberFloat;
            const float randomNumberMinL = transmitToLeft ? randomNumberMin : newRandomNumberMin;
            const float randomNumberMinR = transmitToLeft ? newRandomNumberMin : randomNumberMin;

            // Rough intersection test between the ellipsoid and AABB for each child.
            const float3 centerL = eyeVertex.position + center / sqrt(randomNumberMinL) - childL.bbCenter;
            const float3 centerR = eyeVertex.position + center / sqrt(randomNumberMinR) - childR.bbCenter;
            const bool hitL = Intersect(transformation, centerL, childL.bbHalfSize, randomNumberMinL);
            const bool hitR = Intersect(transformation, centerR, childR.bbHalfSize, randomNumberMinR);

            if(hitL && hitR) {
                nodeIndex = transmitToLeft ? childL.index : childR.index;
                stack[stackLevel].nodeIndex = transmitToLeft ? childR.index : childL.index;
                stack[stackLevel].randomNumberMin = newRandomNumberMin;
                stack[stackLevel].infimum = supremum;
                ++stackLevel;
                continue;
            } else if(hitL) {
                nodeIndex = childL.index;
                if(!transmitToLeft) {
                    randomNumberMin = newRandomNumberMin;
                    infimum = supremum;
                }
                continue;
            } else if(hitR) {
                nodeIndex = childR.index;
                if(!transmitToLeft) {
                    randomNumberMin = newRandomNumberMin;
                    infimum = supremum;
                }
                continue;
            } else {
                const uint32_t leafIndex = BVH::DecodeLeafIndex(nodeIndex);
                const LightVertex& lightVertex = bvh.GetLeaf(leafIndex);
                result += Shade(eyeVertex, lightVertex, randomNumberMin, lobe);
            }
        } else {
            const StackElement& stackElement = stack[--stackLevel];
            nodeIndex = stackElement.nodeIndex;
            randomNumberMin = stackElement.randomNumberMin;
            infimum = stackElement.infimum;
        }
    }

    return result;
}
```
bool Intersect(const float3x3vec4& transformation, const float3& center, const float3& boxHalfSize, const float invRadius2) {
    // Bounding box calculation of the transformed box
    const float3 vec4 x = float3x4(transformation[0].x, transformation[1].x, transformation[2].x);
    const float3 vec4 y = float3x4(transformation[0].y, transformation[1].y, transformation[2].y);
    const float3 vec4 z = float3x4(transformation[0].z, transformation[1].z, transformation[2].z);
    const float3 vec4 bbHalfSize = abs(x) * boxHalfSize.x + abs(y) * boxHalfSize.y + abs(z) * boxHalfSize.z;

    // Sphere-box intersection test [Arv96] in the transformed space
    const float3 vec4 transformedCenter = mul(transformation, float3vec4(center));
    const float3 vec4 squaredDistance = dot(p, p);
    return mul(orientation4, float3x3vec4(ellipsoidMatrix));
}

float3x3vec4 AlignTransformationMatrix(const float3x3& ellipsoidMatrix) {
    const float3 x = (ellipsoidMatrix[0].x, ellipsoidMatrix[1].x, ellipsoidMatrix[2].x);
    const float3 y = (ellipsoidMatrix[0].y, ellipsoidMatrix[1].y, ellipsoidMatrix[2].y);
    const float3 z = (ellipsoidMatrix[0].z, ellipsoidMatrix[1].z, ellipsoidMatrix[2].z);
    const float squaredLengthX = dot(x, x);
    const float squaredLengthY = dot(y, y);
    const float squaredLengthZ = dot(z, z);
    const float3 vec4 tranformedCenter = mul(transformation, float3vec4(center));
    const float3 vec4 bbHalfSize = abs(x) * boxHalfSize.x + abs(y) * boxHalfSize.y + abs(z) * boxHalfSize.z;
    const float3 vec4 p = max(abs(tranformedCenter) - bbHalfSize, float3vec4::zero());
    const float3 vec4 squaredDistance = dot(p, p);
    return mul(orientation4, float3x3vec4(ellipsoidMatrix));
}

Program 2: Conservative ellipsoid-box intersection test using SIMD optimization.

Program 3: Orientations of the test space for the SIMD optimization.