C++ Example Code of Hierarchical Russian Roulette

Yusuke Tokuyoshi*
SQUARE ENIX CO., LTD.

Takahiro Harada
Advanced Micro Devices, Inc.

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 $Cq_{\mathbf{z}}(\boldsymbol{\omega})$ 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 1: Classes used in our implementation.

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

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

Function	Description
dot	Dot product of two 3D vectors
cross	Cross product of two 3D vectors
normalize	Normalization of a 3D vector
max	Maximum value for each dimension
abs	Absolute value for each dimension
mul	Matrix multiplication

*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:

$$\mathbf{A} = \begin{bmatrix} \frac{1}{r_x} & 0 & 0 \\ 0 & \frac{1}{r_y} & 0 \\ 0 & 0 & \frac{1}{r_z} \end{bmatrix} \mathbf{Q} = \begin{bmatrix} \boldsymbol{\omega}_x / r_x \\ \boldsymbol{\omega}_y / r_y \\ \boldsymbol{\omega}_z / r_z \end{bmatrix},$$

where $\left[r_x, r_y, r_z\right]$ are the semiaxes of the ellipsoid, and $\mathbf{Q} = \left[\boldsymbol{\omega}_x^{\mathrm{T}} \quad \boldsymbol{\omega}_y^{\mathrm{T}} \quad \boldsymbol{\omega}_z^{\mathrm{T}}\right]^{\mathrm{T}}$ is the orthogonal matrix to represent the orientation of the ellipsoid. In Program 1, $\mathbf{A}$ is *ellipsoidMatrix*. The variable *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.

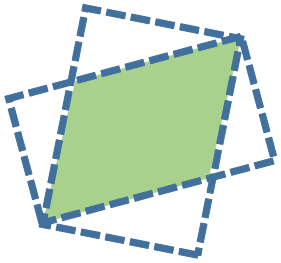

Figure 1: Two tight OBBs for a parallelogram.

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

[Arv90] James Arvo. A simple method for box-sphere intersection testing. In *Graphics Gems*, pages 335–339. 1990.

[BV18] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators, 2018.

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);
    }

    if(stackLevel == 0) { break; }

    const StackElement& stackElement = stack[--stackLevel];
    nodeIndex = stackElement.nodeIndex;
    randomNumberMin = stackElement.randomNumberMin;
    infimum = stackElement.infimum;
  }

  return result;
}
```

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

```
bool Intersect(const float3x3vec4& transformation, const float3& center, const float3& boxHalfSize, const float invRadius2)
{
  // Bounding box calculation of the transformed box
  const float3vec4 x = float3vec4(transformation[0].x, transformation[1].x, transformation[2].x);
  const float3vec4 y = float3vec4(transformation[0].y, transformation[1].y, transformation[2].y);
  const float3vec4 z = float3vec4(transformation[0].z, transformation[1].z, transformation[2].z);
  const float3vec4 bbHalfSize = abs(x) * boxHalfSize.x + abs(y) * boxHalfSize.y + abs(z) * boxHalfSize.z;

  // Sphere-box intersection test [Arv90] in the transformed space
  const float3vec4 tranformedCenter = mul(transformation, float3vec4(center));
  const float3vec4 p = max(abs(tranformedCenter) - bbHalfSize, floatvec4::zero());
  const floatvec4 squaredDistance = dot(p, p);

  // Return true only if all intersection tests are true.
  return all(squaredDistance * invRadius2 < 1.0f);
}
```

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

```
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 axisX = x / sqrt(squaredLengthX);
  const float3 axisY = y / sqrt(squaredLengthY);
  const float3 axisZ = z / sqrt(squaredLengthZ);
  const float3 axisXY = normalize(cross(axisX, y));
  const float3x3 orientationXY = {axisX, cross(axisXY, axisX), axisXY};
  const float3 axisXZ = normalize(cross(axisX, z));
  const float3x3 orientationXZ = {axisX, cross(axisXZ, axisX), axisXZ};
  const float3 axisYZ = normalize(cross(axisY, z));
  const float3x3 orientationYZ = {axisY, cross(axisYZ, axisY), axisYZ};
  const float3 axisYX = normalize(cross(axisY, x));
  const float3x3 orientationYX = {axisY, cross(axisYX, axisY), axisYX};
  const float3 axisZX = normalize(cross(axisZ, x));
  const float3x3 orientationZX = {axisZ, cross(axisZX, axisZ), axisZX};
  const float3 axisZY = normalize(cross(axisZ, y));
  const float3x3 orientationZY = {axisZ, cross(axisZY, axisZ), axisZY};
  float3x3vec4 orientation4;

  if(squaredLengthX <= squaredLengthY && squaredLengthX <= squaredLengthZ) {
    // Align to YZ, YX, ZX, ZY planes.
    orientation4[0].x = floatvec4(orientationYZ[0].x, orientationYX[0].x, orientationZX[0].x, orientationZY[0].x);
    orientation4[0].y = floatvec4(orientationYZ[0].y, orientationYX[0].y, orientationZX[0].y, orientationZY[0].y);
    orientation4[0].z = floatvec4(orientationYZ[0].z, orientationYX[0].z, orientationZX[0].z, orientationZY[0].z);
    orientation4[1].x = floatvec4(orientationYZ[1].x, orientationYX[1].x, orientationZX[1].x, orientationZY[1].x);
    orientation4[1].y = floatvec4(orientationYZ[1].y, orientationYX[1].y, orientationZX[1].y, orientationZY[1].y);
    orientation4[1].z = floatvec4(orientationYZ[1].z, orientationYX[1].z, orientationZX[1].z, orientationZY[1].z);
    orientation4[2].x = floatvec4(orientationYZ[2].x, orientationYX[2].x, orientationZX[2].x, orientationZY[2].x);
    orientation4[2].y = floatvec4(orientationYZ[2].y, orientationYX[2].y, orientationZX[2].y, orientationZY[2].y);
    orientation4[2].z = floatvec4(orientationYZ[2].z, orientationYX[2].z, orientationZX[2].z, orientationZY[2].z);
  } else if(squaredLengthY <= squaredLengthZ && squaredLengthY <= squaredLengthX) {
    // Align to ZX, ZY, XY, XZ planes.
    orientation4[0].x = floatvec4(orientationZX[0].x, orientationZY[0].x, orientationXY[0].x, orientationXZ[0].x);
    orientation4[0].y = floatvec4(orientationZX[0].y, orientationZY[0].y, orientationXY[0].y, orientationXZ[0].y);
    orientation4[0].z = floatvec4(orientationZX[0].z, orientationZY[0].z, orientationXY[0].z, orientationXZ[0].z);
    orientation4[1].x = floatvec4(orientationZX[1].x, orientationZY[1].x, orientationXY[1].x, orientationXZ[1].x);
    orientation4[1].y = floatvec4(orientationZX[1].y, orientationZY[1].y, orientationXY[1].y, orientationXZ[1].y);
    orientation4[1].z = floatvec4(orientationZX[1].z, orientationZY[1].z, orientationXY[1].z, orientationXZ[1].z);
    orientation4[2].x = floatvec4(orientationZX[2].x, orientationZY[2].x, orientationXY[2].x, orientationXZ[2].x);
    orientation4[2].y = floatvec4(orientationZX[2].y, orientationZY[2].y, orientationXY[2].y, orientationXZ[2].y);
    orientation4[2].z = floatvec4(orientationZX[2].z, orientationZY[2].z, orientationXY[2].z, orientationXZ[2].z);
  } else {
    // Align to XY, XZ, YZ, YX planes.
    orientation4[0].x = floatvec4(orientationXY[0].x, orientationXZ[0].x, orientationYZ[0].x, orientationYX[0].x);
    orientation4[0].y = floatvec4(orientationXY[0].y, orientationXZ[0].y, orientationYZ[0].y, orientationYX[0].y);
    orientation4[0].z = floatvec4(orientationXY[0].z, orientationXZ[0].z, orientationYZ[0].z, orientationYX[0].z);
    orientation4[1].x = floatvec4(orientationXY[1].x, orientationXZ[1].x, orientationYZ[1].x, orientationYX[1].x);
    orientation4[1].y = floatvec4(orientationXY[1].y, orientationXZ[1].y, orientationYZ[1].y, orientationYX[1].y);
    orientation4[1].z = floatvec4(orientationXY[1].z, orientationXZ[1].z, orientationYZ[1].z, orientationYX[1].z);
    orientation4[2].x = floatvec4(orientationXY[2].x, orientationXZ[2].x, orientationYZ[2].x, orientationYX[2].x);
    orientation4[2].y = floatvec4(orientationXY[2].y, orientationXZ[2].y, orientationYZ[2].y, orientationYX[2].y);
    orientation4[2].z = floatvec4(orientationXY[2].z, orientationXZ[2].z, orientationYZ[2].z, orientationYX[2].z);
  }

  return mul(orientation4, float3x3vec4(ellipsoidMatrix));
}
```