Vectorized Reservoir Sampling - Supplemental Material

SHINJI OGAKI

1 ADDITIONAL RESULTS OF EXAMPLE 2 - SMAPE PLOTS

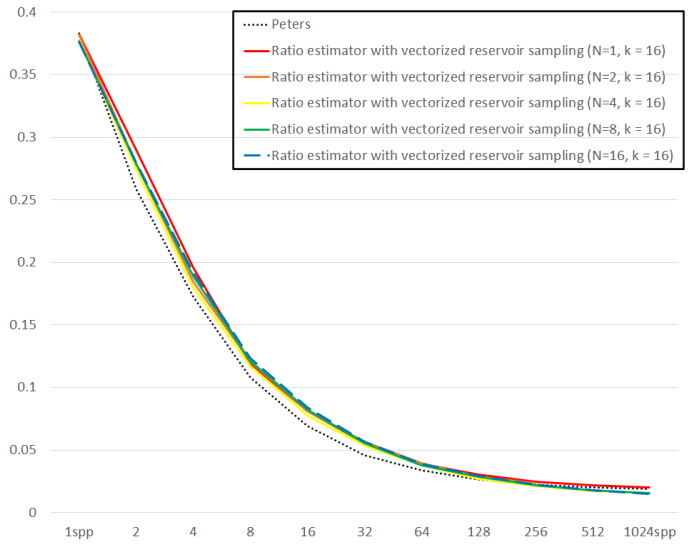

Fig. 1. Comparison of convergence rates for different values of N.

Fig. 1 is a SMAPE plot of the proposed algorithm and method by Peters. Rendering times of ours with $N = 4$ and Peters' method are almost identical. The advantages of our method are:

- Extremely simple implementation.
- Flexibility. Users can trade off between quality and speed by changing N.
- Other terms, including emission profile, can be easily included.

Users can choose a suitable target function from

$$L = \int_A L_e \underbrace{\rho(\boldsymbol{\omega_o}, \boldsymbol{\omega_i}) G(\boldsymbol{x_l}, \boldsymbol{x_s})}_{\text{target function } \hat{p}(\boldsymbol{x_l})} V(\boldsymbol{x_l}, \boldsymbol{x_s}) dA(\boldsymbol{x_l})$$

and

$$L = \int_A \underbrace{L_e \rho(\boldsymbol{\omega_o}, \boldsymbol{\omega_i}) G(\boldsymbol{x_l}, \boldsymbol{x_s})}_{\text{target function } \hat{p}(\boldsymbol{x_l})} V(\boldsymbol{x_l}, \boldsymbol{x_s}) dA(\boldsymbol{x_l})$$

depending on a texture and emission profile.

Author's address: Shinji Ogaki, shinji.ogaki@gmail.com.

Fig. 2 and 3 are comparisons between with and without sample warping, and with and without vectorization, respectively.

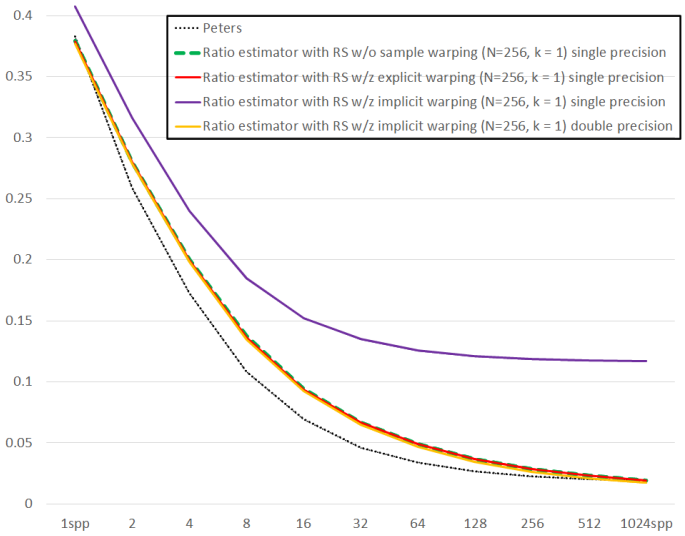

Fig. 2. Comparison of the results with and without sample warping. There is no significant difference between Chao's algorithm with and without explicit warping, indicating that the use of explicit sample warping does not degrade the quality. On the other hand, implicit warping using single precision does not converge to the correct result due to numerical errors.

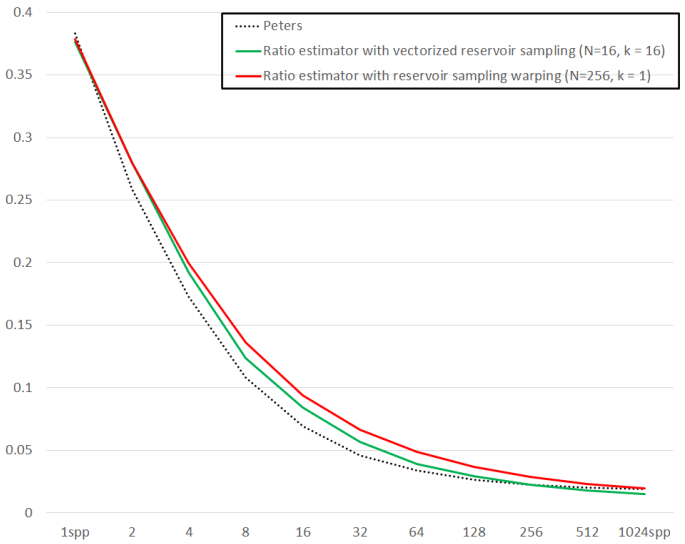

Fig. 3. Comparison of vectorized and non-vectorized reservoir sampling. When touching the same number of samples, the vectorized version gives better results (i.e., lower SMAPE values), which attributes to the improved stratification by splitting a stream into substreams, as described in the paper.

Fig. 4 shows a comparison of three different scenes. Our method is comparable or slightly inferior to Peters' sampling technique.

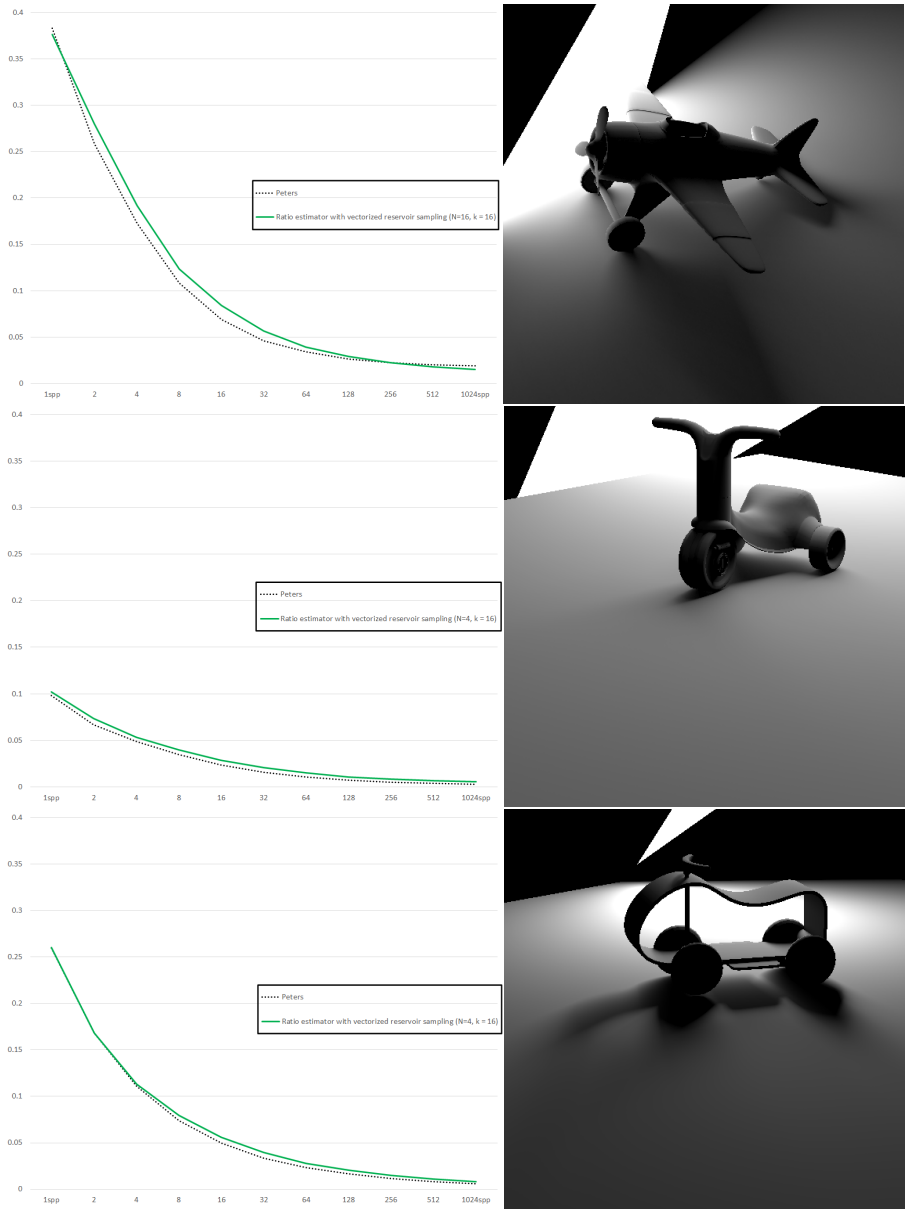

Fig. 4. SMAPE plots of three scenes.

2 DEFINITIONS

```
// see https://enoki.readthedocs.io/en/master/index.html by Wenzel Jakob
#include "enoki/array.h"
const auto k = 16; // SIMD width
using Vector3f  = enoki::Array<float , 3>;
using FloatP    = enoki::Array<float , k>;
using IntegerP  = enoki::Array<int   , k>;
using Vector3fP = enoki::Array<FloatP, 3>;
```

3 VECTORIZED RESERVOIR SAMPLING WITH EXPLICIT SAMPLE WARPING

```
struct Reservoir
{
        IntegerP  indices = IntegerP(-1); // selected sample id
        FloatP    weights;
        Vector3fP samples;
        FloatP    sum = FloatP(0.0f); // weight sum

        void update(const IntegerP& index, const FloatP& weight, FloatP& random)
        {
                sum += weight;
                const auto tmp  = random * sum;
                const auto mask = (tmp < weight);
                indices = enoki::select(mask, index , indices);
                weights = enoki::select(mask, weight, weights);
                random  = enoki::select(mask, tmp / weight, random);
                random  = enoki::select(mask | (0.0f >= weight), random, (tmp - weight) / (sum - weight));
        }

        void update(const Vector3fP& sample, const FloatP& weight, FloatP& random)
        {
                sum += weight;
                const auto tmp = random * sum;
                const auto mask = (tmp < weight);
                samples = enoki::select(mask, sample, samples);
                random  = enoki::select(mask, tmp / weight, random);
                random  = enoki::select(mask | (0.0f >= weight), random, (tmp - weight) / (sum - weight));
        }

        int select_lane(const float xi, float sum_of_lanes)
        {
                FloatP cdf;
                for (auto i = 0; i < k; ++i)
                {
                        sum_of_lanes += sum[i];
                        cdf[i] = sum_of_lanes;
                }
                if (0.0f >= sum_of_lanes)
                {
                        return -1; // invalid
                }

                // when directly using intrinsic functions, remove the loop and use _mm512_cmp_ps_mask and std::countr_zero()
                const auto mask = ((sum_of_lanes * xi) < cdf);
                for (auto i = 0; i < k; ++i)
                {
                        if (mask[i])
                        {
                                return i;
                        }
                }
                return -1;
        }
};
```

4 BOUNDARY INTEGRATION OF CLIPPED TRIANGLE

```
// see "Geometric Derivation of the Irradiance of Polygonal Lights" by Eric Heitz

// solve 0 = <(v0 x (1 - t) + v1 x t), normal>
auto clip(const Vector3f& v0, const Vector3f& v1, const float vn0, const float vn1)
{
        const auto t = vn0 / (vn0 - vn1);
        return normalize(v0 * (1.0f - t) + v1 * t);
}

// boundary integration (triangle)
auto boundary3(const Vector3f& a, const Vector3f& b, const Vector3f& c, const Vector3f& normal)
{
        const auto ab = dot(a, b);
        const auto bc = dot(b, c);
        const auto ca = dot(c, a);
        return
            ((1 <= abs(ab)) ? 0 : dot(cross(b, a), normal) / sqrt(1 - ab * ab) * acos(ab)) +
            ((1 <= abs(bc)) ? 0 : dot(cross(c, b), normal) / sqrt(1 - bc * bc) * acos(bc)) +
            ((1 <= abs(ca)) ? 0 : dot(cross(a, c), normal) / sqrt(1 - ca * ca) * acos(ca));
}

// boundary integration (rectangle)
auto boundary4(const Vector3f& a, const Vector3f& b, const Vector3f& c, const Vector3f& d, const Vector3f& normal)
{
        const auto ab = dot(a, b);
        const auto bc = dot(b, c);
        const auto cd = dot(c, d);
        const auto da = dot(d, a);
        return
            ((1 <= abs(ab)) ? 0 : dot(cross(b, a), normal) / sqrt(1 - ab * ab) * acos(ab)) +
            ((1 <= abs(bc)) ? 0 : dot(cross(c, b), normal) / sqrt(1 - bc * bc) * acos(bc)) +
            ((1 <= abs(cd)) ? 0 : dot(cross(d, c), normal) / sqrt(1 - cd * cd) * acos(cd)) +
            ((1 <= abs(da)) ? 0 : dot(cross(a, d), normal) / sqrt(1 - da * da) * acos(da));
}

// boundary integration
auto boundary_integration(const Vector3f& A, const Vector3f& B, const Vector3f& C, const Vector3f& normal)
{
        const auto An = dot(A, normal);
        const auto Bn = dot(B, normal);
        const auto Cn = dot(C, normal);

        // sign
        const auto sign_a = (0.0f < An);
        const auto sign_b = (0.0f < Bn);
        const auto sign_c = (0.0f < Cn);

        // early termination
        if (!sign_a && !sign_b && !sign_c)
        {
                return 0.0f;
        }

        // normalize
        const auto a = normalize(A);
        const auto b = normalize(B);
        const auto c = normalize(C);

        // 2 vertices below horizon -> triangle
        if ( sign_a && !sign_b && !sign_c) return boundary3(a, clip(A, B, An, Bn), clip(C, A, Cn, An), normal);
        if (!sign_a &&  sign_b && !sign_c) return boundary3(b, clip(B, C, Bn, Cn), clip(A, B, An, Bn), normal);
        if (!sign_a && !sign_b &&  sign_c) return boundary3(c, clip(C, A, Cn, An), clip(B, C, Bn, Cn), normal);

        // 1 vertex below horizon -> rectangle
        if (!sign_a &&  sign_b &&  sign_c) return boundary4(c, clip(C, A, Cn, An), clip(A, B, An, Bn), b, normal);
        if ( sign_a && !sign_b &&  sign_c) return boundary4(a, clip(A, B, An, Bn), clip(B, C, Bn, Cn), c, normal);
        if ( sign_a &&  sign_b && !sign_c) return boundary4(b, clip(B, C, Bn, Cn), clip(C, A, Cn, An), a, normal);

        // no vertices below horizon -> triangle
        return boundary3(a, b, c, normal);
}
```

5 RATIO ESTIMATOR WITH VECTORIZED RESERVOIR SAMPLING

```
float get_sample(
        float  xi[4],
        const  Vector3f p[3], // vertices of triangular light
        const  Vector3f& n_l, // normal of light source
        const  Vector3f& x_s, // shading point
        const  Vector3f& n_s, // shading normal
        Vector3f& sample      // selected sample
)
{
        // constants
        static const auto N = 4; // length of each substream
        static const FloatP step_s(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1);
        static const FloatP step_t(0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7);

        // unshadowed illumination (pi omitted)
        const auto weight = boundary_integration(p[0] - x_s, p[1] - x_s, p[2] - x_s, n_s) / 2;
        if (0 >= weight)
        {
                return 0;
        }

        // enokified reservoir
        Reservoir r;

        // ratio estimator with vectorized reservoir sampling
        FloatP random(xi[2]);
        const auto t = sqrt((step_t + xi[0]) / 8);
        for (auto i = 0; i < N; ++i)
        {
                // see "A Low-Distortion Map Between Triangle and Square" by Eric Heitz
                const auto s    = ((i * 2) + step_s + xi[1]) / 8;
                const auto a    = 1 - t;
                const auto c    = s * t;
                const auto b    = t - c;
                const auto x_l = Vector3fP(p[0]) * FloatP(a) + Vector3fP(p[1]) * FloatP(b) + Vector3fP(p[2]) * FloatP(c);

                // target function: G
                const auto wi     = x_l - x_s;
                const auto sq_dis = dot(wi, wi);
                const auto target = max(0.0f, dot(n_l, -wi)) * max(0.0f, dot(n_s, wi)) / (sq_dis * sq_dis);

                // update
                r.update(x_l, target, random);
        }

        // select SIMD lane
        auto sum = 0.0f;
        const auto lane = r.select_lane(xi[3], sum);
        if (0 > lane)
        {
                return 0;
        }

        // result
        sample.x() = r.samples.x()[lane];
        sample.y() = r.samples.y()[lane];
        sample.z() = r.samples.z()[lane];
        return weight;
}
```