Generalized Light Portals
Supplemental Material

Shinji Ogaki

2020/06/22

Barycentric coordinates of N subtriangles

```
// Arity N=4 (SSE)
static const float<N> M[3][3] =
{{
    float<N>(1.0f, 0.5f, 0.0f, 0.5f),
    float<N>(0.0f, 0.5f, 0.5f, 0.0f),
    float<N>(0.0f, 0.0f, 0.5f, 0.5f)
},{
    float<N>(0.5f, 0.0f, 0.5f, 0.0f),
    float<N>(0.5f, 1.0f, 0.0f, 0.5f),
    float<N>(0.0f, 0.0f, 0.5f, 0.5f)
},{
    float<N>(0.5f, 0.0f, 0.5f, 0.0f),
    float<N>(0.0f, 0.5f, 0.5f, 0.0f),
    float<N>(0.5f, 0.5f, 0.0f, 1.0f)
}};
```

```
// Arity N=16 (AVX-512)
static const float<N> M[3][3] =
{{
    float<N>(1.00f, 0.75f, 0.50f, 0.75f, 0.50f, 0.25f, 0.50f, 0.25f,
             0.50f, 0.25f, 0.00f, 0.25f, 0.00f, 0.25f, 0.00f, 0.25f),
    float<N>(0.00f, 0.25f, 0.25f, 0.00f, 0.50f, 0.50f, 0.25f, 0.25f,
             0.00f, 0.75f, 0.75f, 0.50f, 0.50f, 0.25f, 0.25f, 0.00f),
    float<N>(0.00f, 0.00f, 0.25f, 0.25f, 0.00f, 0.25f, 0.25f, 0.50f,
             0.50f, 0.00f, 0.25f, 0.25f, 0.50f, 0.50f, 0.75f, 0.75f)
},{
    float<N>(0.75f, 0.50f, 0.75f, 0.50f, 0.25f, 0.50f, 0.25f, 0.50f,
             0.25f, 0.00f, 0.25f, 0.00f, 0.25f, 0.00f, 0.25f, 0.00f),
    float<N>(0.25f, 0.50f, 0.00f, 0.25f, 0.75f, 0.25f, 0.50f, 0.00f,
             0.25f, 1.00f, 0.50f, 0.75f, 0.25f, 0.50f, 0.00f, 0.25f),
    float<N>(0.00f, 0.00f, 0.25f, 0.25f, 0.00f, 0.25f, 0.25f, 0.50f,
             0.50f, 0.00f, 0.25f, 0.25f, 0.50f, 0.50f, 0.75f, 0.75f)
},{
    float<N>(0.75f, 0.50f, 0.75f, 0.50f, 0.25f, 0.50f, 0.25f, 0.50f,
             0.25f, 0.00f, 0.25f, 0.00f, 0.25f, 0.00f, 0.25f, 0.00f),
    float<N>(0.00f, 0.25f, 0.25f, 0.00f, 0.50f, 0.50f, 0.25f, 0.25f,
             0.00f, 0.75f, 0.75f, 0.50f, 0.50f, 0.25f, 0.25f, 0.00f),
    float<N>(0.25f, 0.25f, 0.00f, 0.50f, 0.25f, 0.00f, 0.50f, 0.25f,
             0.75f, 0.25f, 0.00f, 0.50f, 0.25f, 0.75f, 0.50f, 1.00f)
}};
```

```
vec3<N> mul(const float<N>& f, const vec3<N>& v) { return vec3<N>(f * v.x, f * v.y, f * v.z); }
vec3<N> mul(const vec3<N>& v, const float<N>& f) { return vec3<N>(f * v.x, f * v.y, f * v.z); }

// Initialize barycentric coordinates
void Initialize(vec3<N> &B0, vec3<N> &B1, vec3<N> &B2)
{
    B0.x = float<N>(1); B0.y = float<N>(0); B0.z = float<N>(0);
    B1.x = float<N>(0); B1.y = float<N>(1); B1.z = float<N>(0);
    B2.x = float<N>(0); B2.y = float<N>(0); B2.z = float<N>(1);
}

// Update barycentric coordinates
void Update(vec3<N> &B0, vec3<N> &B1, vec3<N> &B2)
{
    const auto tmp0 = mul(B0, M[0][0]) + mul(B1, M[0][1]) + mul(B2, M[0][2]);
    const auto tmp1 = mul(B0, M[1][0]) + mul(B1, M[1][1]) + mul(B2, M[1][2]);
    const auto tmp2 = mul(B0, M[2][0]) + mul(B1, M[2][1]) + mul(B2, M[2][2]);
    B0 = tmp0;
    B1 = tmp1;
    B2 = tmp2;
}
```

AVX-512 version of Clarberg's equal-area mapping

```
void Vector2UV(__m512& u, __m512& v, const __m512& dx, const __m512& dy, const __m512& dz)
{
        // 16-wide math constants
        const auto mfZERO = _mm512_set1_ps(0.0f);
        const auto mfHALF = _mm512_set1_ps(0.5f);
        const auto mfONE  = _mm512_set1_ps(1.0f);

        // Coefficients for 6th degree minimax approximation of atan(x)*2/pi, x=[0,1].
        const auto t1 = _mm512_set1_ps(0.406758566246788489601959989e-5f);
        const auto t2 = _mm512_set1_ps(0.636226545274016134946890922156f);
        const auto t3 = _mm512_set1_ps(0.615720178982802134934197203466e-2f);
        const auto t4 = _mm512_set1_ps(-0.247333733281268944196501420480f);
        const auto t5 = _mm512_set1_ps(0.881770664775316294736387951347e-1f);
        const auto t6 = _mm512_set1_ps(0.419038818029165735901852432784e-1f);
        const auto t7 = _mm512_set1_ps(-0.251390972343483509333252996350e-1f);

        const auto x = _mm512_abs_ps(dx);
        const auto y = _mm512_abs_ps(dy);
        const auto z = _mm512_abs_ps(dz);

        __m512 a, b, r, phi;
        __mmask16 m;

        // Compute the radius r = sqrt(1-|z|)
        r = _mm512_sqrt_ps(_mm512_sub_ps(mfONE, z));

        // Compute the argument to atan (detect a=0 to avoid div-by-zero)
        a = _mm512_max_ps(x, y);
        b = _mm512_min_ps(x, y);
        m = _mm512_cmpeq_ps_mask(mfZERO, a);
        b = _mm512_mask_blend_ps(m, _mm512_div_ps(b, a), mfZERO);

        // Polynomial approximation of atan(x)*2/pi, x=b
        phi = _mm512_fmadd_ps( t7, b, t6);
        phi = _mm512_fmadd_ps(phi, b, t5);
        phi = _mm512_fmadd_ps(phi, b, t4);
        phi = _mm512_fmadd_ps(phi, b, t3);
        phi = _mm512_fmadd_ps(phi, b, t2);
        phi = _mm512_fmadd_ps(phi, b, t1);

        // Extend phi if the input is in the range 45-90 degrees (u<v)
        m   = _mm512_cmplt_ps_mask(x, y);
        phi = _mm512_mask_blend_ps(m, phi, _mm512_sub_ps(mfONE, phi));

        // Find (u,v) based on (r,phi)
        v = _mm512_mul_ps(phi, r);
        u = _mm512_sub_ps(r, v);

        // Southern hemisphere -> mirror u,v
        a = u; // store old u
        m = _mm512_cmplt_ps_mask(dz, mfZERO);
        u = _mm512_mask_blend_ps(m, u, _mm512_sub_ps(mfONE, v));
        v = _mm512_mask_blend_ps(m, v, _mm512_sub_ps(mfONE, a));

        // Move (u,v) to the correct quadrant based on the signs of (x,y)
        m = _mm512_cmplt_ps_mask(dx, mfZERO);
        u = _mm512_mask_blend_ps(m, u, _mm512_sub_ps(mfZERO, u));
        m = _mm512_cmplt_ps_mask(dy, mfZERO);
        v = _mm512_mask_blend_ps(m, v, _mm512_sub_ps(mfZERO, v));

        // Transform (u,v) from [-1,1] to [0,1].
        u = _mm512_fmadd_ps(u, mfHALF, mfHALF);
        v = _mm512_fmadd_ps(v, mfHALF, mfHALF);
}
```

Neighboring pixel coordinates in Clarberg's parameterized space

```
std::tuple<int32_t, int32_t>
GetNeighbour(const int32_t width, const int32_t u, const int32_t v)
{
    const auto u_inside = 0 <= u && u < width;
    const auto v_inside = 0 <= v && v < width;
    if (u_inside && v_inside)
    {
        return std::tuple(u, v);
    }
    const auto bits = width - 1;
    if (v_inside)
    {
        if (0     >  u) return std::tuple(-u - 1             , v ^ bits);
        if (width <= u) return std::tuple(-u - 1 + width * 2, v ^ bits);
    }
    if (u_inside)
    {
        if (0     >  v) return std::tuple(u ^ bits, -v - 1);
        if (width <= v) return std::tuple(u ^ bits, -v - 1 + width * 2);
    }
    return std::tuple((u + width) % width, (v + width) % width);
}
```