

Hierarchical Multi-Layer Screen-Space Ray Tracing

Supplemental Material

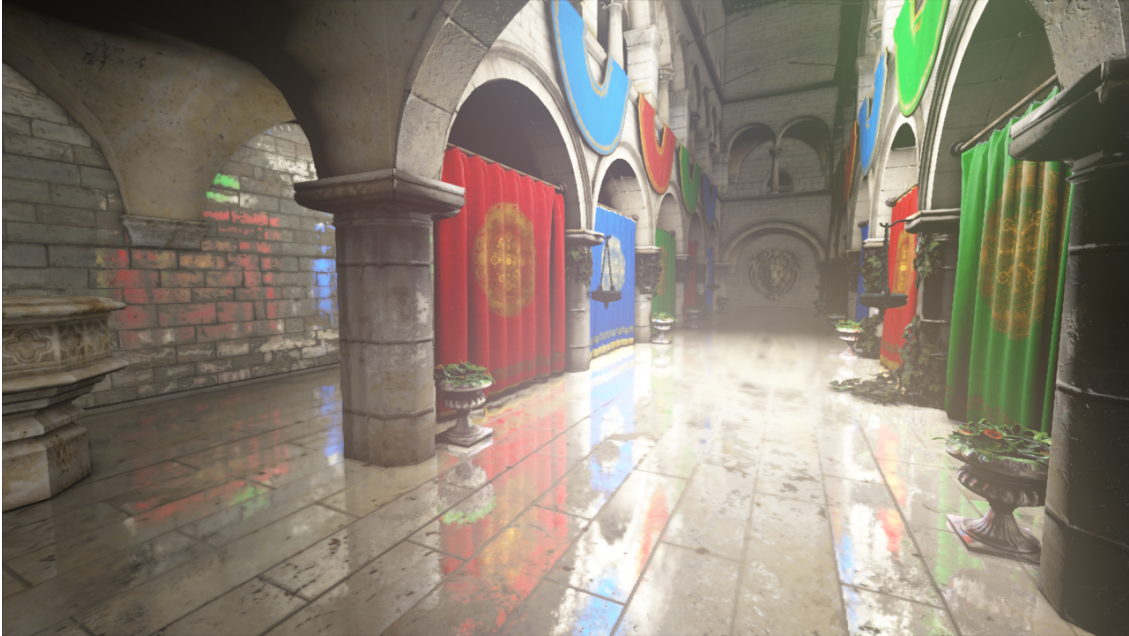
Nikolai Hofmann Phillip Bogendörfer Marc Stamminger Kai Selgrad
Computer Graphics Group, University of Erlangen-Nuremberg

Contents

1	Larger Screenshots	2
2	Construction Times	3
2.1	Sponza	3
2.2	San Miguel	4
3	Full Resolution Trace Times	5
3.1	Sponza	5
3.2	San Miguel	6
3.3	Overall Perf, 1 Bounce Full Res Construction and Full Res Trace	7
3.4	San Miguel	7
4	Half Resolution Trace Times	8
4.1	Sponza	8
4.2	San Miguel	9
4.3	Overall Perf, 1 Bounce Full Res Construction and Half Res Trace	9
4.4	San Miguel	10
5	SSR Code	11
6	HSSR Code	14
6.1	Fragment Collection Pass	14
6.2	Per-Pixel Linked-Lists Sorting Pass	15
6.3	Hierarchy Construction	16
6.4	Hierarchical Screen-Space Ray Tracer	19

1 Larger Screenshots

Times for Sponza refer to this view.



Times for San Miguel refer to this view.

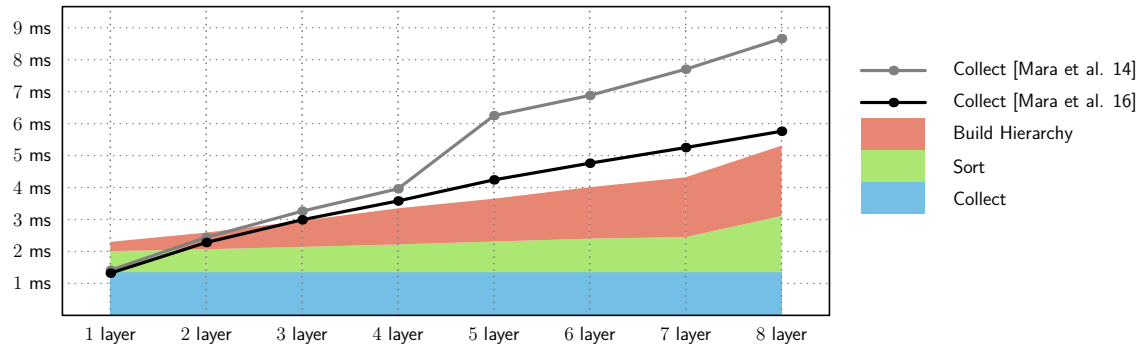




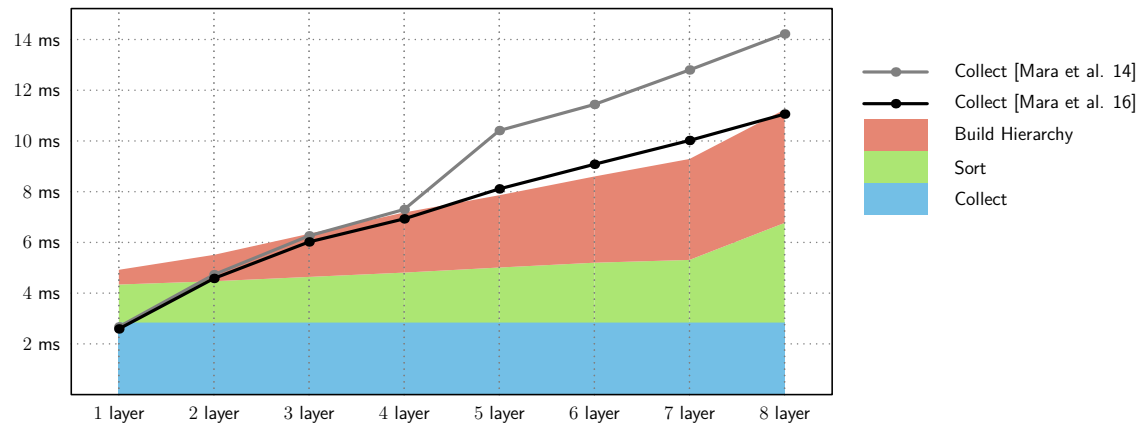
2 Construction Times

2.1 Sponza

720p:

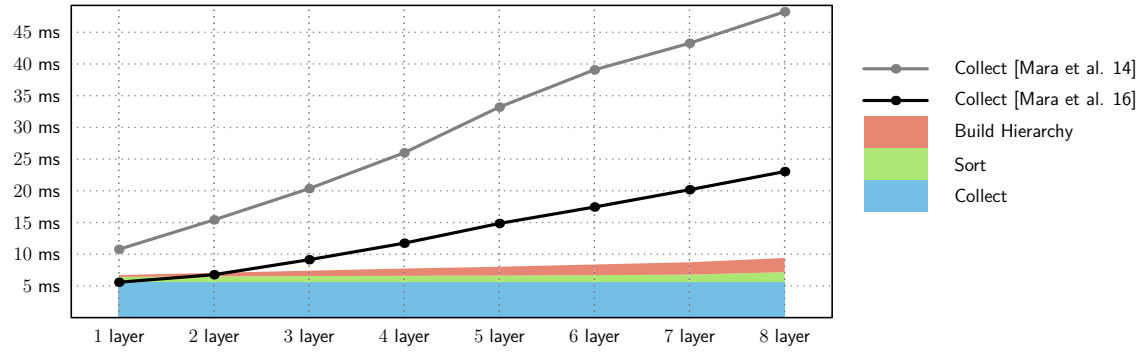


1080p:

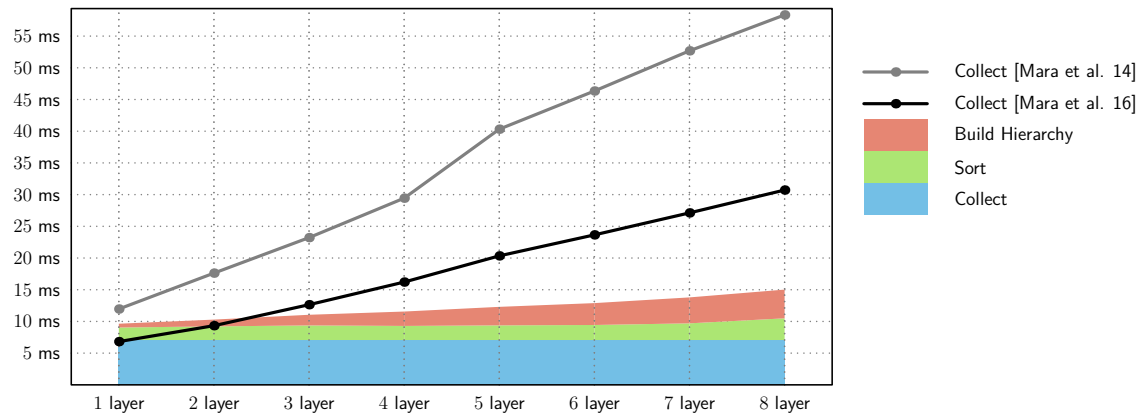


2.2 San Miguel

720p:



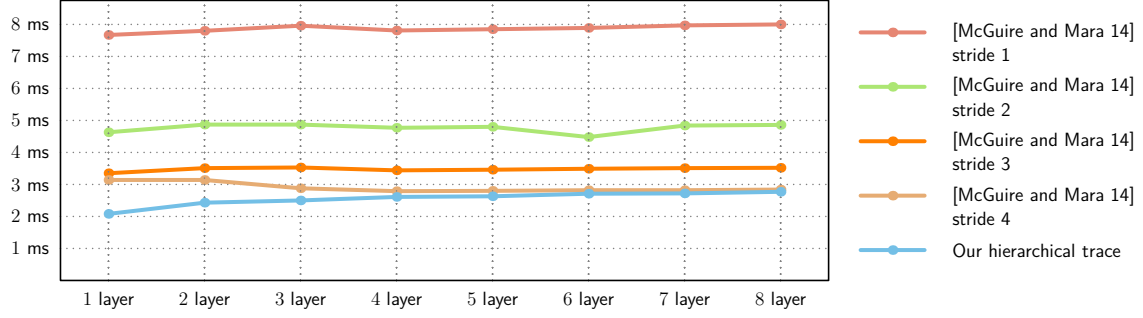
1080p:



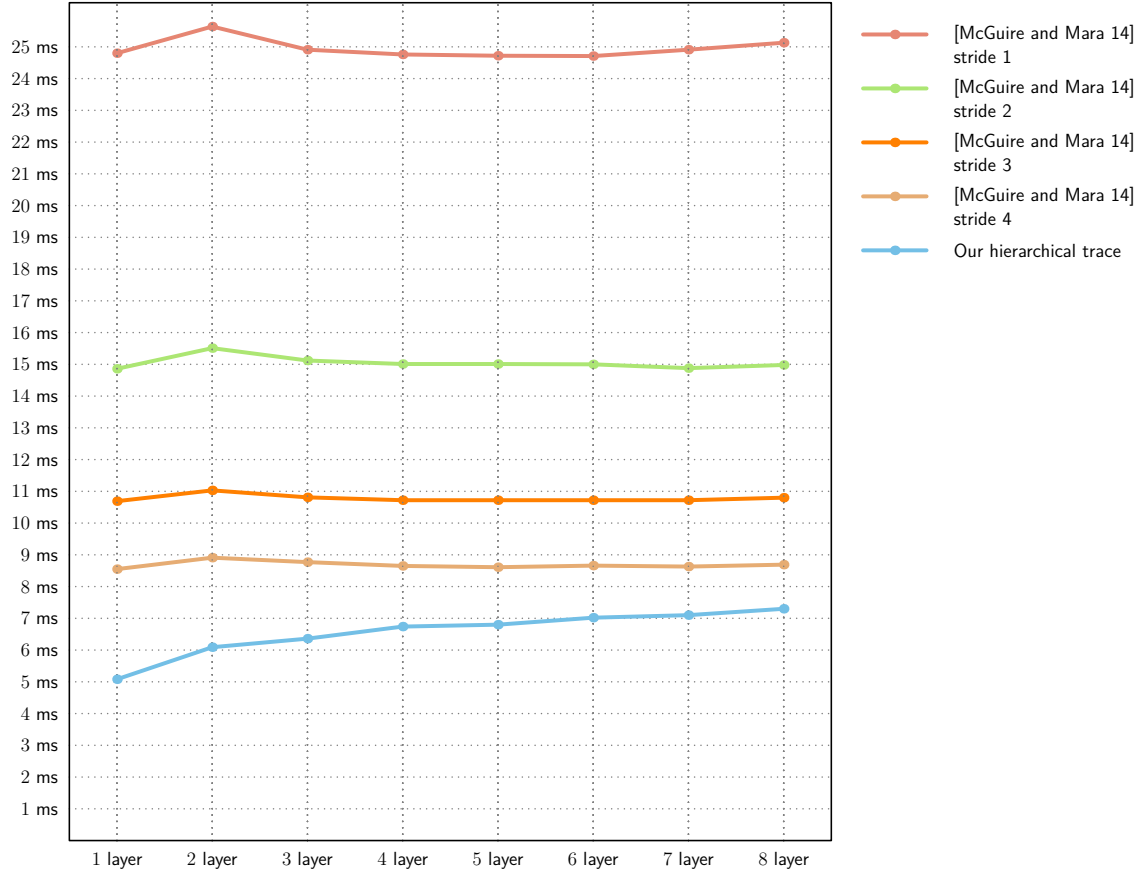
3 Full Resolution Trace Times

3.1 Sponza

720p:

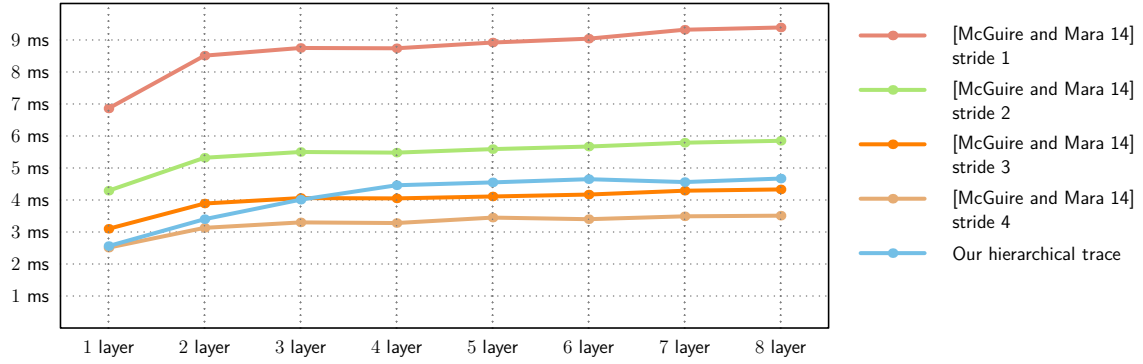


1080p:

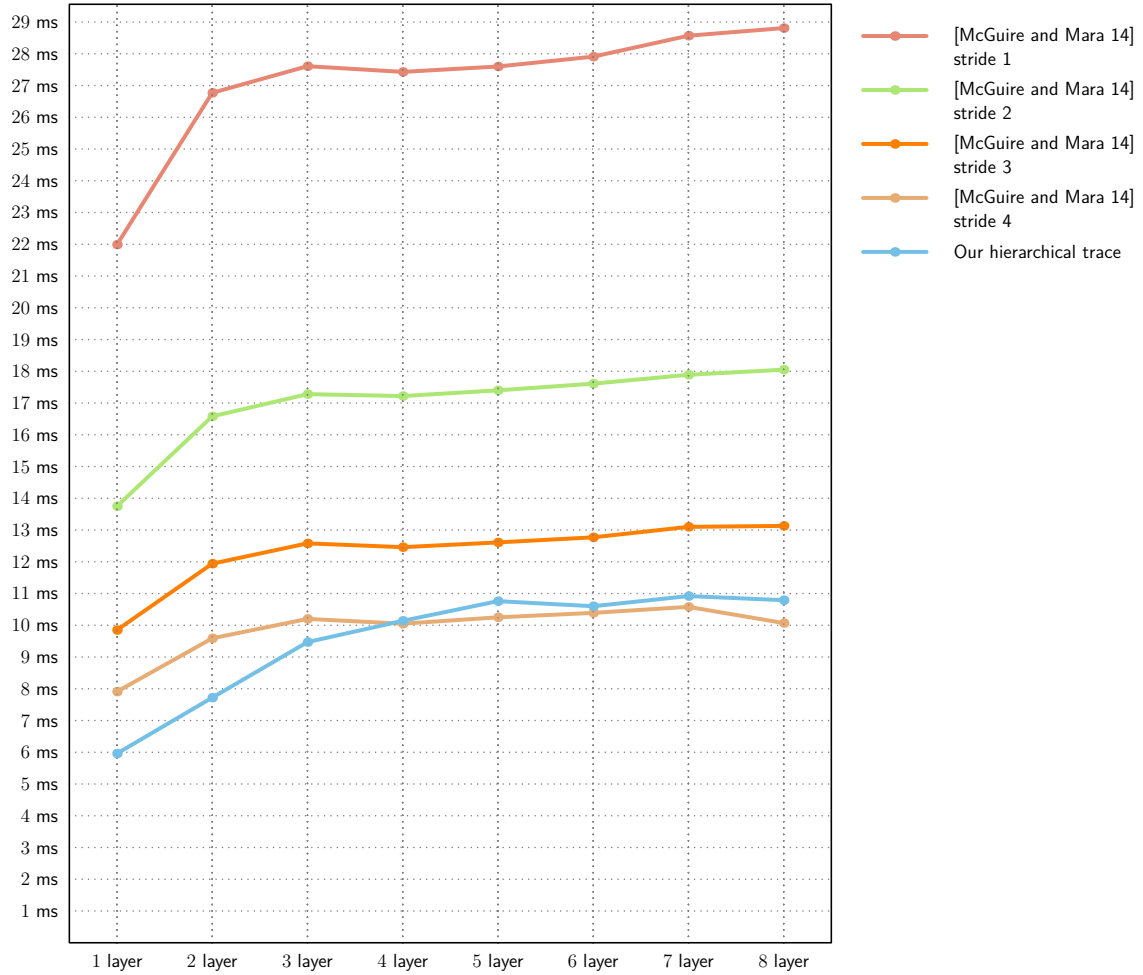


3.2 San Miguel

720p:

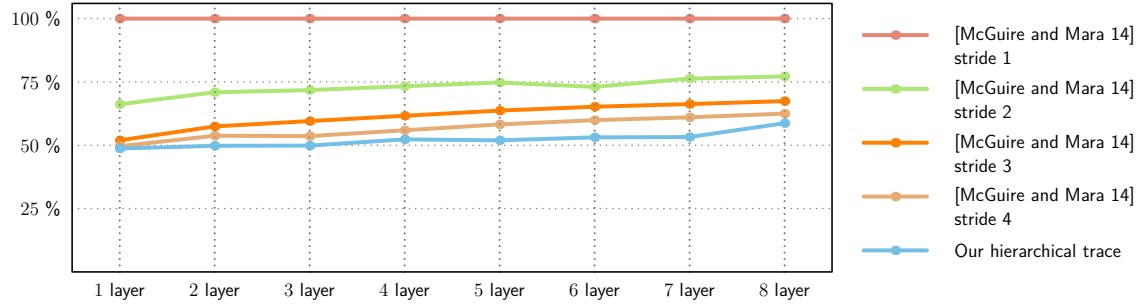


1080p:

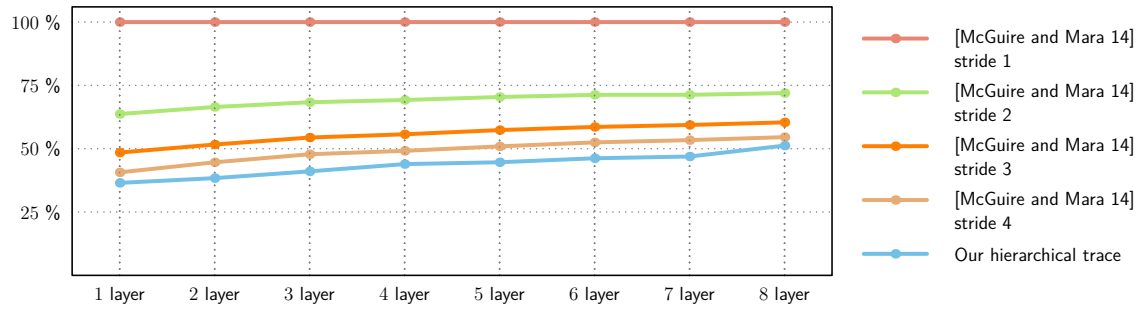


3.3 Overall Perf, 1 Bounce Full Res Construction and Full Res Trace

720p:

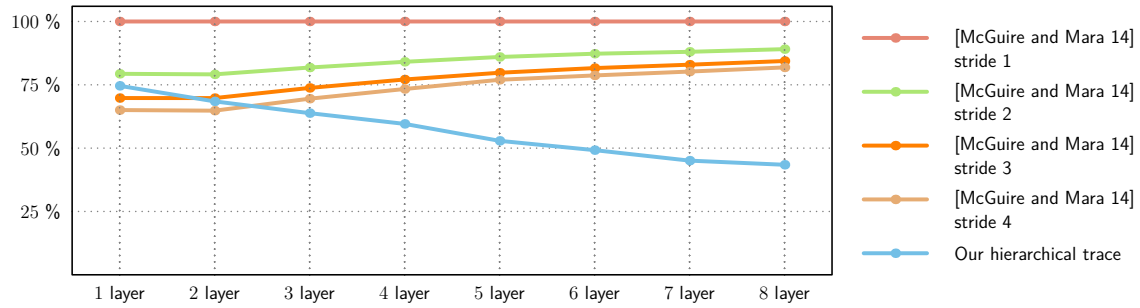


1080p:

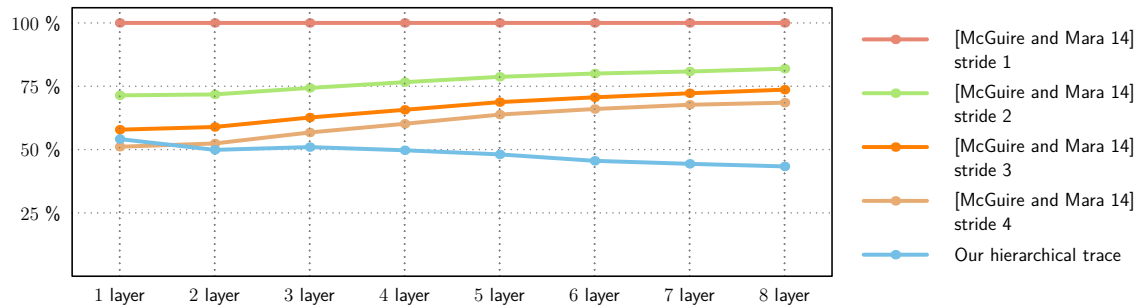


3.4 San Miguel

720p:



1080p:

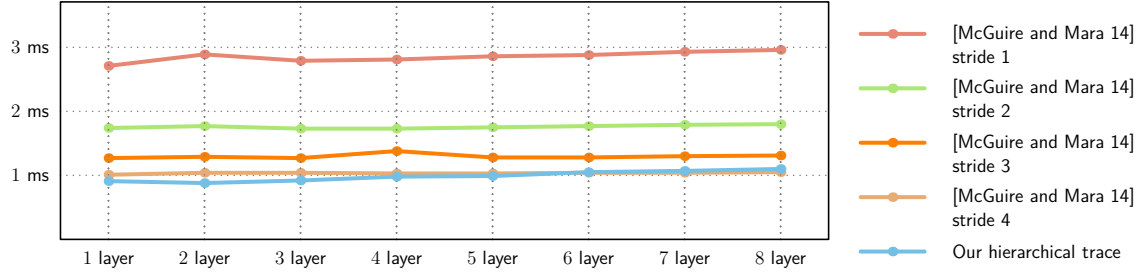


4 Half Resolution Trace Times

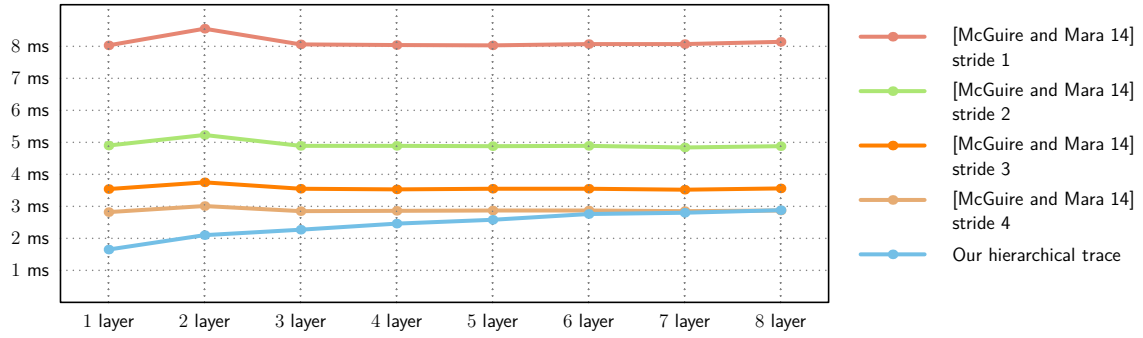
Traced times for 720p: 640×560 , for 1080p: 960×540 . Threads are assigned without ‘holes’.

4.1 Sponza

720p:

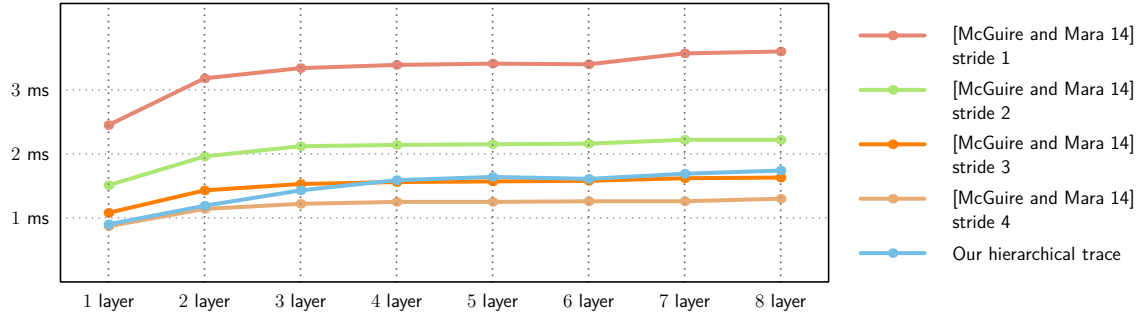


1080p:

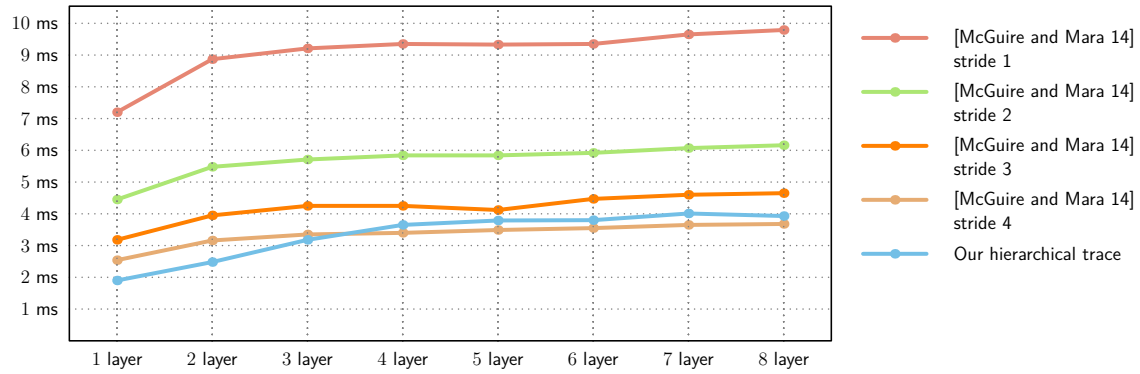


4.2 San Miguel

720p:

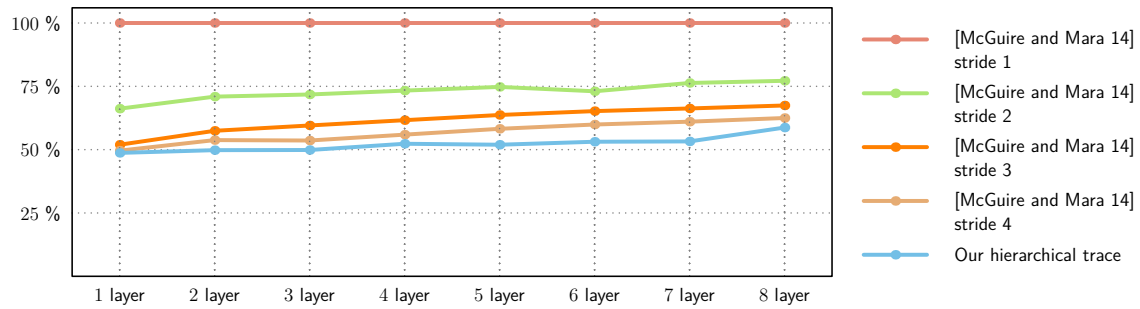


1080p:

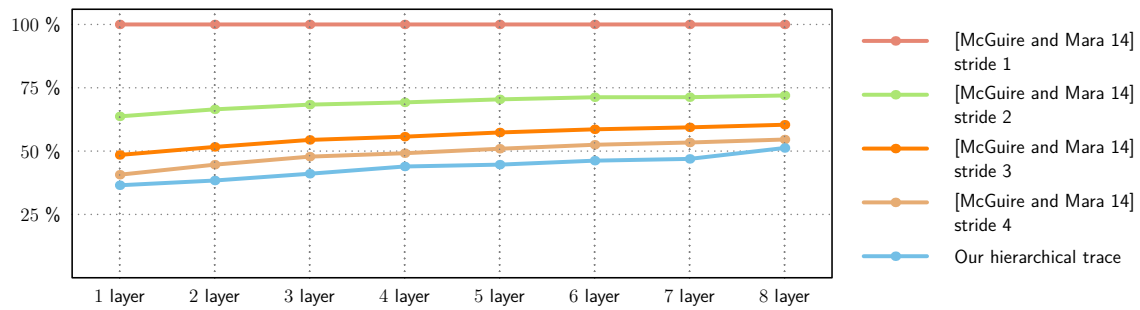


4.3 Overall Perf, 1 Bounce Full Res Construction and Half Res Trace

720p:

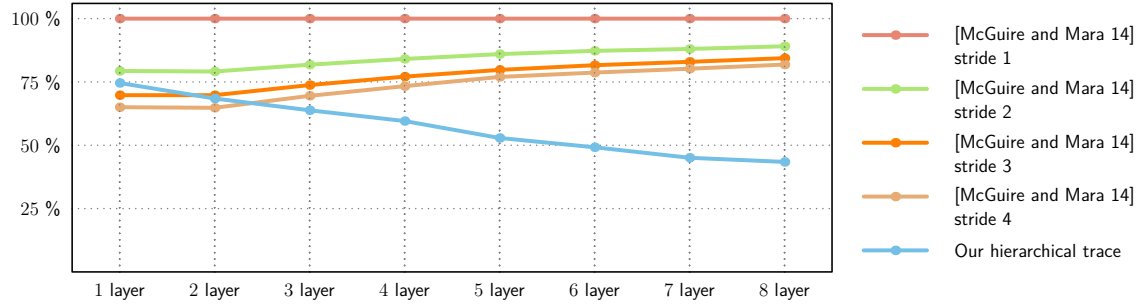


1080p:

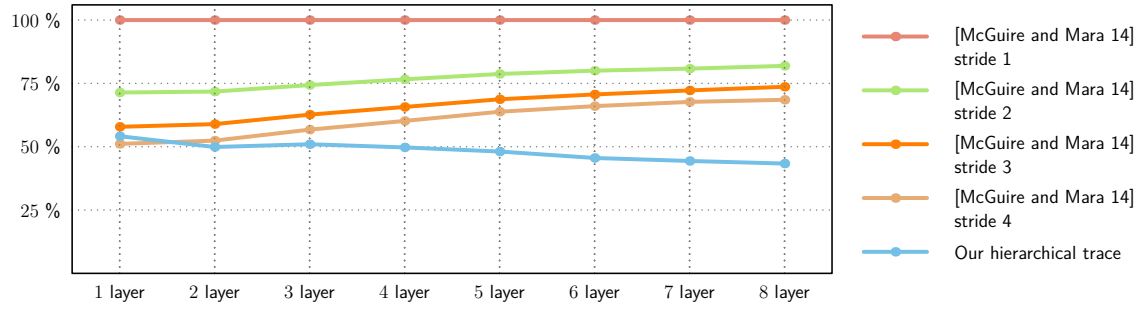


4.4 San Miguel

720p:



1080p:



5 SSR Code

Originally published by McGuire and Mara (2014), adapted to our framework.

```
1  #version 430 core
2  #define LAYERS 8
3  #define THREADS 32
4  layout (local_size_x = THREADS, local_size_y = THREADS) in;
5
6  uniform mat4 view;
7  uniform mat4 proj;
8  uniform ivec2 screendim;
9  uniform vec2 cam_near_far;
10 uniform float aspect;
11 uniform float tan_fovy;
12 uniform int stride;
13 uniform int half_res;
14 const ivec2 dim = ivec2(screendim * (half_res == 1 ? 0.5f : 1.f));
15
16 // Journal of Computer Graphics Techniques Vol. 3, No. 2, 2014:
17 vec2 signNotZero(vec2 v) {
18     return vec2((v.x >= 0.0) ? +1.0 : -1.0, (v.y >= 0.0) ? +1.0 : -1.0);
19 }
20 vec3 oct_to_float32x3(vec2 e) {
21     vec3 v = vec3(e.xy, 1.0 - abs(e.x) - abs(e.y));
22     if (v.z < 0) v.xy = (1.0 - abs(v.yx)) * signNotZero(v.xy);
23     return normalize(v);
24 }
25 vec3 reproject(vec2 at, float depth) {
26     const float fy = 1.f / tan_fovy, fx = fy / aspect, inv_fx = 1.f / fx;
27     const vec2 inv_screendim = 1.f / vec2(screendim);
28     vec2 clip = at * inv_screendim * 2.f - 1.f;
29     float eye_z = -depth * cam_near_far.y;
30     float eye_x = (clip.x * -eye_z) * inv_fx;
31     float eye_y = (clip.y * -eye_z) * tan_fovy;
32     return vec3(eye_x, eye_y, eye_z);
33 }
34
35 // parameters
36 #define OPTIONAL_CLIPPING
37 #define EPSILON 0.0001f
38 #define THICKNESS EPSILON * 5 * cam_near_far.y
39 #define THRESHOLD 0.3f
40 #define MAX_STEPS 1000
41 #define MAX_LEN 3000.f
42 #define BOUNCES 1
43
44 struct Ray {
45     vec3 orig;
46     vec3 dir;
47 };
48 struct RayTracingResult {
49     bool hit[BOUNCES];
50     vec3 hitPos[BOUNCES];
51     float dist[BOUNCES];
52     vec3 hitNorm[BOUNCES];
53 };
54 struct RayTracingState {
55     vec2 P0, dP;
```

```

56     vec3 Q0, dQ;
57     float k0, dk;
58     float stepDir;
59     float end;
60     bool permute;
61 } state;
62
63 float distanceSquared(in vec2 A, in vec2 B) { A -= B; return dot(A, A);
64 }
65 void setupFor(in Ray ray) {
66     // clip near plane
67     float rayLength = (-ray.orig.z - ray.dir.z * MAX_LEN) <
68         cam_near_far.x ? (cam_near_far.x + ray.orig.z) / -ray.dir.z :
69         MAX_LEN;
70     vec3 csEndPoint = ray.orig + ray.dir * rayLength;
71     vec4 H0 = proj * vec4(ray.orig, 1.0), H1 = proj * vec4(csEndPoint,
72         1.0);
73     state.k0 = 1.0 / H0.w; float k1 = 1.0 / H1.w;
74     state.Q0 = ray.orig * state.k0; vec3 Q1 = csEndPoint * k1;
75     state.P0 = H0.xy * state.k0; vec2 P1 = H1.xy * k1;
76     state.P0 = ((state.P0 + vec2(1))/2)*screendim; P1 = ((P1 +
77         vec2(1))/2)*screendim;
78     // [optional clipping]
79 #ifndef OPTIONAL_CLIPPING
80     float xMax = screendim.x - 0.5, xMin = 0.5, yMax = screendim.y - 0.5,
81         yMin = 0.5;
82     float alpha = 0.0;
83     if ((P1.y > yMax) || (P1.y < yMin))
84         alpha = (P1.y - ((P1.y > yMax) ? yMax : yMin)) / (P1.y -
85             state.P0.y);
86     if ((P1.x > xMax) || (P1.x < xMin))
87         alpha = max(alpha, (P1.x - ((P1.x > xMax) ? xMax : xMin)) / (P1.x -
88             state.P0.x));
89     P1 = (1-alpha) * P1 + alpha * state.P0;
90     k1 = (1-alpha) * k1 + alpha * state.k0;
91     Q1 = (1-alpha) * Q1 + alpha * state.Q0;
92 #endif
93     P1 += vec2((distanceSquared(state.P0, P1) < 0.0001) ? 0.01 : 0.0);
94     vec2 delta = P1 - state.P0;
95     state.permute = false;
96     if (abs(delta.x) < abs(delta.y)) {
97         state.permute = true;
98         delta = delta.yx; state.P0 = state.P0.yx; P1 = P1.yx;
99     }
100     state.stepDir = sign(delta.x);
101     float invdx = state.stepDir / delta.x;
102     state.dQ = (Q1 - state.Q0) * invdx;
103     state.dk = (k1 - state.k0) * invdx;
104     state.dP = vec2(state.stepDir, delta.y * invdx);
105     state.dP *= stride; state.dQ *= stride; state.dk *= stride;
106     state.end = P1.x * state.stepDir;
107 }
108
109 RayTracingResult traceRay(in Ray ray) {
110     // setup and init to no hit found
111     RayTracingResult result;
112     int bounces = 0;
113     for (int i = 0; i < BOUNCES; ++i)

```

```

106     result.hit[i] = false;
107     setupFor(ray);
108     // slide P from P0 to P1, same for Q and k
109     vec2 P = state.P0; vec3 Q = state.Q0; float k = state.k0; float
        prevZMax = ray.orig.z;
110     // iterate
111     for (int i = 0; (P.x * state.stepDir) <= state.end && i < MAX_STEPS;
        ++i)
112     {
113         P += state.dP;
114         Q.z -= state.dQ.z;
115         k += state.dk;
116         vec2 hitPixel = state.permute ? P.yx : P;
117         if((hitPixel.x < 0)|| (hitPixel.y < 0)|| (hitPixel.x >=
            screendim.x)|| (hitPixel.y >= screendim.y))
118             return result;
119         float rayZMin = prevZMax;
120         float rayZMax = (state.dQ.z * 0.5f + Q.z) / (state.dk * 0.5f + k);
121         prevZMax = rayZMax;
122         if(rayZMin > rayZMax){ float tmp = rayZMax; rayZMax = rayZMin;
            rayZMin = tmp; }
123         // traverse layers
124         for (int L = 0; L < LAYERS; ++L) {
125             if (i == 0) break; // WTF
126             float sceneZMax = -texelFetch(front_depth, ivec3(hitPixel, L),
                0).r * cam_near_far.y; // eye space z
127             float sceneZMin = sceneZMax - THICKNESS;
128             if (((rayZMax >= sceneZMin) && (rayZMin <= sceneZMax))) {
129                 // hit
130                 Q.xy += state.dQ.xy * i;
131                 vec3 csHitPoint = Q * (1.f / k);
132                 result.hit[bounces] = true;
133                 result.hitPos[bounces] = csHitPoint;
134                 result.hitNorm[bounces] = texelFetch(front_normal,
                    ivec3(hitPixel, L), 0).rgb;
135                 bounces++;
136                 if (bounces >= BOUNCES)
137                     return result;
138                 // setup new ray
139                 vec3 dir reflect(ray.dir, result.hitNorm[bounces-1]);
140                 ray = Ray(result.hitPos[bounces-1], dir, col.a);
141                 setupFor(ray);
142                 P = state.P0; Q = state.Q0; k = state.k0; prevZMax = ray.orig.z;
143             }
144             if(sceneZMin < rayZMin) break;
145         }
146     }
147     return result;
148 }

```

6 HSSR Code

6.1 Fragment Collection Pass

```
1  in vec4 pos_wc;
2  in vec3 norm_wc;
3  in vec2 tc;
4  layout(std430, binding = 0) buffer ListHeadsBuffer {
5      uint list_heads[];
6  };
7  layout(std430, binding = 1) buffer ListDataBuffer {
8      uvec4 list_data[];
9  };
10 layout(binding = 2, offset = 0) uniform atomic_uint counter;
11 // Journal of Computer Graphics Techniques Vol. 3, No. 2, 2014:
12 vec2 signNotZero(vec2 v) {
13     return vec2((v.x >= 0.0) ? +1.0 : -1.0, (v.y >= 0.0) ? +1.0 : -1.0);
14 }
15 vec2 float32x3_to_oct(in vec3 v) {
16     vec2 p = v.xy * (1.0 / (abs(v.x) + abs(v.y) + abs(v.z)));
17     return (v.z <= 0.0) ? ((1.0 - abs(p.yx)) * signNotZero(p)) : p;
18 }
19 uvec4 pack_data(float depth, uint ptr, vec3 norm, vec3 col, float refl)
20 {
21     // x: 32Bit depth, y: 32Bit oct normals
22     // z: 24Bit rgb, 8 bit reflectiviy, w: 32Bit pointer
23     depth = clamp(depth, 0, 1);
24     norm = normalize(norm);
25     uint x = uint(round(depth * 4294967295.f));
26     uint y = packSnorm2x16(float32x3_to_oct(norm));
27     col = clamp(col, vec3(0), vec3(1));
28     uint z = packUnorm4x8(vec4(pow(col, vec3(2.2f)), refl));
29     uint w = ptr;
30     return uvec4(x, y, z, w);
31 }
32 void main() {
33     if (mask() < 0.2) { discard; return; } // sample alphamap
34     vec4 pos_ec = view * pos_wc;
35     // linear depth
36     float depth = (-pos_ec.z) * inv_far;
37     // normalmapping
38     vec3 norm_ec = normal_matrix * norm_wc;
39     norm_ec = normalize(normal(norm_ec, pos_ec.xyz)); // sample normalmap
40     // create linked list
41     int at = int(gl_FragCoord.y) * screendim.x + int(gl_FragCoord.x);
42     uint to = atomicCounterIncrement(counter);
43     uint old = atomicExchange(list_heads[at], to);
44     // pack and store data
45     vec4 spec_col = specular(); // sample specularmap
46     float reflectivity = (spec_col.r+spec_col.g+spec_col.b) / 3.f;
47     list_data[to] = pack_data(depth, old, norm_ec, diffuse().rgb,
48                             reflectivity);
49 }
```


6.2 Per-Pixel Linked-Lists Sorting Pass

```
1 #version 430 core
2 #define THREADS 16
3 #define LAYERS 8
4 // #define MIN_SEPARATION
5 layout (local_size_x = THREADS, local_size_y = THREADS) in;
6 layout(std430, binding = 0) buffer ListHeadsBuffer {
7     uint list_heads[];
8 };
9 layout(std430, binding = 1) buffer ListDataBuffer {
10     uvec4 list_data[];
11 };
12 layout(binding = 2, offset = 0) uniform atomic_uint counter;
13
14 layout(location = 0) uniform ivec2 screendim;
15
16 shared uvec4 insertion[THREADS*THREADS*LAYERS];
17 const uint local_pos = gl_LocalInvocationIndex*LAYERS;
18
19 void main() {
20     const ivec2 gid = ivec2(gl_GlobalInvocationID.xy);
21     if (gid.x >= screendim.x || gid.y >= screendim.y) return;
22     const uint at = gid.y * screendim.x + gid.x;
23     const uint start = list_heads[at];
24     uint pos = start;
25     int n = 0;
26     int i = 0;
27
28     // insertion sort PPLL
29     while (pos != 0xFFFFFFFF) {
30         uvec4 curr = list_data[pos];
31         pos = curr.w;
32         int to = n;
33         for (int i = 0; i < n; ++i)
34             if (curr.x < insertion[local_pos+i].x) {
35                 to = i;
36                 break;
37             }
38         if (to < LAYERS) {
39             for (int move = min(n,LAYERS-1); move > to; --move)
40                 insertion[local_pos+move] = insertion[local_pos+move-1];
41             insertion[local_pos+to] = curr;
42             if (n < LAYERS) ++n;
43         }
44     }
45
46     // write sorted list as array
47     const uint to = atomicCounter(counter) + at * LAYERS;
48 #ifndef MIN_SEPARATION
49     #define SEPARATION uint(0.002f * uint(0xFFFFFFFF))
50     int frags_out = 0;
51     for (int i = 0; i < n; ++i) {
52         if (i > 0 && insertion[local_pos+i].x -
53             insertion[local_pos+frags_out-1].x < SEPARATION)
54             continue;
55         list_data[to+frags_out++] = insertion[local_pos+i];
56     }
```

```

56     n = frags_out;
57 #else
58     for (int i = 0; i < n; ++i)
59         list_data[to+i] = insertion[local_pos+i];
60 #endif
61     list_heads[at] = uint(clamp(n, 0, 15)) << 28 | (to & 0xFFFFFFFF);
62 }

```

6.3 Hierarchy Construction

```

1  #version 430 core
2  #define THREADS 24
3  #define LAYERS 8
4
5  #define COMPRESS_FRAGS
6
7  layout (local_size_x = THREADS, local_size_y = THREADS) in;
8
9  layout(std430, binding = 0) buffer ListHeadsBuffer {
10     uint list_heads[];
11 };
12 layout(std430, binding = 1) buffer ListDataBuffer {
13     uvec2 list_data[]; // uvec2!
14 };
15
16 layout(binding = 2, offset = 0) uniform atomic_uint counter;
17
18 layout(location = 0) uniform ivec2 screendim;
19 layout(location = 1) uniform vec2 cam_near_far;
20 layout(location = 2) uniform vec2 near_plane_size;
21 layout(location = 3) uniform vec2 far_plane_size;
22 layout(location = 4) uniform int epsilon_factor;
23 layout(location = 5) uniform ivec2 source_dim;
24 layout(location = 6) uniform ivec2 target_dim;
25 layout(location = 7) uniform int source_offset;
26 layout(location = 8) uniform int target_offset;
27 layout(location = 9) uniform int level; // [0, mm_levels[
28
29 const float far = cam_near_far.y;
30 const float inv_far = 1.f / far;
31
32 // -----
33 float unpack_depth(uint data) {
34     const float f = 1 / float(uint(0xFFFFFFFF));
35     return float(data) * f;
36 }
37 float unpack_depth(uvec2 data) {
38     const float f = 1 / float(uint(0xFFFFFFFF));
39     return float(data.x) * f;
40 }
41 float unpack_depth_end(uvec2 data) {
42     const float f = 1 / float(uint(0xFFFFFFFF));
43     return float(data.y) * f;
44 }
45 uint to_ptr(uint handle) {
46     return (handle & 0xFFFFFFFF);
47 }

```

```

48 | uint to_len(uint handle) {
49 |     return (handle & 0xF0000000) >> 28;
50 | }
51 | uint to_handle(uint len, uint ptr) {
52 |     return uint(clamp(len, 0, 15)) << 28 | (ptr & 0xFFFFFFFF);
53 | }
54 | uvec2 load(uint ptr, uint offset) {
55 | #ifdef COMPRESS_FRAGS
56 |     int stride = level == 0 ? 2 : 1;
57 |     return list_data[2*ptr + stride*offset];
58 | #else
59 |     return list_data[2*ptr + 2*offset];
60 | #endif
61 | }
62 | uvec2 load(uvec2 filter_handle) {
63 |     return load(to_ptr(filter_handle.x), filter_handle.y);
64 | }
65 | void save(uint ptr, uint offset, uvec2 data) {
66 | #ifdef COMPRESS_FRAGS
67 |     list_data[2*ptr + offset] = data;
68 | #else
69 |     list_data[2*(ptr + offset)] = data;
70 | #endif
71 | }
72 | // -----
73 | float w(float depth, int l = level) {
74 |     const float base_radius = near_plane_size.x / screendim.x;
75 |     const float add_radius = (far_plane_size.x - near_plane_size.x) /
76 |         screendim.x;
77 |     return (base_radius + depth * add_radius) * float(1 << level);
78 | }
79 | float epsilon(float d_min) {
80 |     return epsilon_factor * w(d_min, level+1) * inv_far;
81 | }
82 | shared uvec2 handles[THREADS*THREADS*4];
83 | shared uvec2 frags[THREADS*THREADS*4];
84 | const uint local_pos = gl_LocalInvocationIndex*4;
85 |
86 | bool pop_closest(inout float d_min, inout float d_max) {
87 |     uint closest = min(min(frags[local_pos+0].x, frags[local_pos+1].x),
88 |         min(frags[local_pos+2].x, frags[local_pos+3].x));
89 |     if (closest == -1) return false;
90 |     // found a fragment, update data structures
91 |     int i = closest == frags[local_pos+0].x ? 0 : closest ==
92 |         frags[local_pos+1].x ? 1 : closest == frags[local_pos+2].x ? 2 :
93 |         3;
94 |     d_min = unpack_depth(closest);
95 |     d_max = d_min + epsilon(d_min);
96 |     handles[local_pos+i].y++;
97 |     frags[local_pos+i] = handles[local_pos+i].y <
98 |         to_len(handles[local_pos+i].x) ? load(handles[local_pos+i]) :
99 |         uvec2(-1, -1);
100 |     return true;
101 | }
102 | bool pop_within_epsilon(in float d_min, inout float d_max) {
103 |     uint sec_closest = min(min(frags[local_pos+0].x,
104 |         frags[local_pos+1].x), min(frags[local_pos+2].x,

```

```

    frags[local_pos+3].x));
99  if (sec_closest == -1) return false;
100 // check if within EPSILON range
101 int i = sec_closest == frags[local_pos+0].x ? 0 : sec_closest ==
    frags[local_pos+1].x ? 1 : sec_closest == frags[local_pos+2].x ?
    2 : 3;
102 float d_sec_closest = unpack_depth(sec_closest);
103 float d_sec_closest_end = level == 0 ? d_sec_closest :
    unpack_depth_end(frags[local_pos+i]);
104 if (d_sec_closest - d_min > epsilon(d_min)) return false;
105 // found a fragment, update data structures
106 d_max = max(d_max, d_sec_closest_end);
107 handles[local_pos+i].y++;
108 frags[local_pos+i] = handles[local_pos+i].y <
    to_len(handles[local_pos+i].x) ? load(handles[local_pos+i]) :
    uvec2(-1, -1);
109 return true;
110 }
111
112 void main() {
113     const ivec2 gid = ivec2(gl_GlobalInvocationID.xy);
114     if (gid.x >= target_dim.x || gid.y >= target_dim.y) return;
115     // init data structures
116     const ivec2 base0 = clamp(2*gid + ivec2(0, 0), ivec2(0),
        source_dim-1);
117     const ivec2 base1 = clamp(2*gid + ivec2(1, 0), ivec2(0),
        source_dim-1);
118     const ivec2 base2 = clamp(2*gid + ivec2(0, 1), ivec2(0),
        source_dim-1);
119     const ivec2 base3 = clamp(2*gid + ivec2(1, 1), ivec2(0),
        source_dim-1);
120     handles[local_pos+0].x = list_heads[source_offset + base0.y *
        source_dim.x + base0.x];
121     handles[local_pos+1].x = list_heads[source_offset + base1.y *
        source_dim.x + base1.x];
122     handles[local_pos+2].x = list_heads[source_offset + base2.y *
        source_dim.x + base2.x];
123     handles[local_pos+3].x = list_heads[source_offset + base3.y *
        source_dim.x + base3.x];
124     handles[local_pos+0].y = 0;
125     handles[local_pos+1].y = 0;
126     handles[local_pos+2].y = 0;
127     handles[local_pos+3].y = 0;
128     frags[local_pos+0] = to_len(handles[local_pos+0].x) == 0 ? uvec2(-1,
        -1) : load(handles[local_pos+0]);
129     frags[local_pos+1] = to_len(handles[local_pos+1].x) == 0 ? uvec2(-1,
        -1) : load(handles[local_pos+1]);
130     frags[local_pos+2] = to_len(handles[local_pos+2].x) == 0 ? uvec2(-1,
        -1) : load(handles[local_pos+2]);
131     frags[local_pos+3] = to_len(handles[local_pos+3].x) == 0 ? uvec2(-1,
        -1) : load(handles[local_pos+3]);
132     const uint target_at = target_offset + gid.y * target_dim.x + gid.x;
133     const uint to = int(atomicCounter(counter)) + target_at * LAYERS;
134     int fragments_out = 0;
135     float d_min, d_max = 0;
136
137 #ifdef COMPRESS_FRAGS
138     const int NUM = 2*LAYERS;

```

```

139 #else
140     const int NUM = LAYERS;
141 #endif
142     // iterate lists and filter
143     for (int i = 0; i < NUM; ++i) {
144         if (!pop_closest(d_min, d_max)) break;
145         while (pop_within_epsilon(d_min, d_max)) {}
146         // save first fragment
147         save(to, fragments_out++, uvec2(uint(clamp(d_min, 0, 1) *
            uint(-1)), uint(clamp(d_max, 0, 1) * uint(-1))));
148         d_min = d_max = 0;
149     }
150     // update head handle
151     list_heads[target_at] = to_handle(fragments_out, to);
152 }

```

6.4 Hierarchical Screen-Space Ray Tracer

```

1  #version 430 core
2  #extension GL_ARB_shader_ballot : require
3  #extension GL_ARB_shader_group_vote : require
4  #define THREADS 32
5  // parameters
6  #define COMPRESS_FRAGS
7  #define BALLOT_N 4 // reload when num_active/N rays are done
8  #define BALLOT_KILL_N 1 // kill N long rays in warp
9  #define MAX_STEPS 100
10 #define EPSILON 0.0001f
11 #define THICKNESS EPSILON * 5
12 #define BOUNCES 2
13
14 layout (local_size_x = THREADS, local_size_y = THREADS) in;
15
16 layout(std430, binding = 0) buffer ListHeadsBuffer {
17     uint list_heads[];
18 };
19 layout(std430, binding = 1) buffer ListDataBuffer {
20     uvec2 list_data[];
21 };
22
23 uniform mat4 view;
24 uniform mat4 proj;
25 uniform ivec2 screendim;
26 uniform vec2 cam_near_far;
27 uniform float aspect;
28 uniform float tan_fovy;
29 uniform int mm_levels;
30 uniform int half_res;
31 const ivec2 dim = ivec2(screendim * (half_res == 1 ? 0.5f : 1.f));
32 const float inv_far = 1.f / cam_near_far.y;
33 const mat4 inv_view = inverse(view);
34
35 // Journal of Computer Graphics Techniques Vol. 3, No. 2, 2014:
36 vec2 signNotZero(vec2 v) {
37     return vec2((v.x >= 0.0) ? +1.0 : -1.0, (v.y >= 0.0) ? +1.0 : -1.0);
38 }
39 vec3 oct_to_float32x3(vec2 e) {

```

```

40     vec3 v = vec3(e.xy, 1.0 - abs(e.x) - abs(e.y));
41     if (v.z < 0) v.xy = (1.0 - abs(v.yx)) * signNotZero(v.xy);
42     return normalize(v);
43 }
44 float unpack_depth(uvec2 data) {
45     const float f = 1 / float(uint(0xFFFFFFFF));
46     return float(data.x) * f;
47 }
48 float unpack_depth_end(uvec2 data) {
49     const float f = 1 / float(uint(0xFFFFFFFF));
50     return float(data.y) * f;
51 }
52 vec3 unpack_norm(uvec4 data) {
53     return oct_to_float32x3(unpackSnorm2x16(data.y));
54 }
55 vec4 unpack_col(uvec4 data) {
56     return unpackUnorm4x8(data.z);
57 }
58 uint to_ptr(uint handle) {
59     return handle & 0xFFFFFFFF;
60 }
61 uint to_len(uint handle) {
62     return (handle & 0xF0000000) >> 28;
63 }
64 uvec2 load(uint ptr, uint layer = 0, int level = 0) {
65     #ifdef COMPRESS_FRAGS
66         int stride = level == 0 ? 2 : 1;
67         return list_data[2*ptr + stride*layer];
68     #else
69         return list_data[2*ptr + 2*layer];
70     #endif
71 }
72 uvec4 load_uvec4(uint ptr, uint layer = 0, int level = 0) {
73     #ifdef COMPRESS_FRAGS
74         int stride = level == 0 ? 2 : 1;
75         return uvec4(list_data[2*ptr + stride*layer], list_data[2*ptr +
76             stride*layer + 1]);
77     #else
78         return uvec4(list_data[2*ptr + 2*layer], list_data[2*ptr + 2*layer +
79             1]);
80     #endif
81 }
82 struct Ray {
83     vec3 orig;
84     vec3 dir;
85     float f;
86 };
87 struct RayTracingResult {
88     bool hit[BOUNCES];
89     vec3 hitPos[BOUNCES];
90     i vec3 hitNorm[BOUNCES];
91 };
92 struct RayTracingState {
93     vec2 P0, dP;
94     vec3 Q0, dQ;
95     float k0, dk;
96     float t;

```



```

96     int miplevel;
97     vec2 pixlen;
98     vec2 posdir;
99     int layer;
100    bool done;
101 } state;
102
103 void setupFor(in Ray ray) {
104     // setup ray and perform frustum clipping
105     vec4 H0 = proj * vec4(ray.orig, 1.0);
106     state.k0 = 1.0 / H0.w; state.Q0 = ray.orig * state.k0;
107     state.P0 = ((H0.xy * state.k0 + 1.f)/2.f)*screendim;
108     vec2 P1;
109     // compute distance to screen borders and clip in screen space
110     vec4 Hclip = proj * vec4(ray.orig + ray.dir, 1.f);
111     vec2 Pclip = ((Hclip.xy / Hclip.w + 1.f)/2.f)*screendim;
112     vec2 dir_screen = normalize(Pclip - state.P0);
113     vec2 dest_screen = step(0, dir_screen) * (screendim-ivec2(1));
114     vec2 dist_screen = (dest_screen - state.P0) / dir_screen;
115     P1 = state.P0 + dir_screen * min(dist_screen.x, dist_screen.y);
116     // clip against frustum in camera space
117     vec3 csFar = reproject(P1, 1.f);
118     vec3 csFarDir = normalize(csFar);
119     vec3 n = cross(csFarDir, cross(ray.dir, csFarDir));
120     float len = dot(csFar - ray.orig, n) / dot(ray.dir, n);
121     vec3 ray_end = ray.orig + len * ray.dir;
122     // check for error, clip far plane
123     if (-ray_end.z > cam_near_far.y || len < 0)
124         ray_end = ray.orig + ray.dir * (cam_near_far.y + ray.orig.z) /
            -ray.dir.z;
125     vec4 H1 = proj * vec4(ray_end, 1.0); float k1 = 1.f / H1.w; vec3 Q1 =
        ray_end * k1;
126     P1 = ((H1.xy * k1 + 1.f)/2.f)*screendim;
127     state.dQ = Q1 - state.Q0; state.dP = P1 - state.P0; state.dk = k1 -
        state.k0;
128     // setup variables
129     state.posdir = step(vec2(0), state.dP);
130     state.pixlen = 1.f / abs(state.dP);
131     // interpolation factor and mip level
132     state.miplevel = 0;
133     // one initial step to not collide with the origin fragment
134     vec2 dt = abs(floor(state.P0 + state.posdir) - state.P0) *
        state.pixlen;
135     state.t = min(dt.x, dt.y) + EPSILON;
136     state.done = false;
137 }
138 vec3 getCurrCSPos() {
139     return (state.Q0 + state.dQ*state.t) / (state.k0 + state.dk*state.t);
140 }
141 vec2 getCurrSSPos() {
142     return state.P0 + state.t*state.dP;
143 }
144
145 RayTracingResult traceRay(in Ray ray) {
146     // set dda state
147     setupFor(ray);
148     // additional state
149     int bounces = 0;

```

```

150 float acceleration_delay = 1;
151 // init result to false
152 RayTracingResult result;
153 for (int i = 0; i < BOUNCES; ++i)
154     result.hit[i] = false;
155
156 // iterate
157 for(int i = 0; i < MAX_STEPS && state.t < 1.f; ++i) {
158     // compute offset and divisor for current mip level
159     int off = 0; int div = 1;
160     for (int i = 0; i < state.miplevel; i++) {
161         off += int(ceil(float(screendim.x)/div)) *
162             int(ceil(float(screendim.y)/div)); div *= 2;
163     }
164     // current screen space position
165     vec2 Pcurr = getCurrSSPos();
166     // clip ray vs pixel
167     vec2 dt_entry = abs(ceil(Pcurr/div - state.posdir) * div -
168         state.P0) * state.pixlen;
169     vec2 dt_exit = abs(floor(Pcurr/div + state.posdir) * div -
170         state.P0) * state.pixlen;
171     float t_entry = clamp(max(dt_entry.x, dt_entry.y), 0, state.t);
172     float t_exit = clamp(min(dt_exit.x, dt_exit.y), t_entry, 1);
173     // get sample location
174     uint handle = list_heads[off + int(Pcurr.y/div) *
175         int(ceil(float(screendim.x)/div)) + int(Pcurr.x/div)];
176     uint len = to_len(handle);
177     if (len == 0) { state.t = t_exit + EPSILON; continue; }
178     uint pos = to_ptr(handle);
179     // compute ray-frustum entry, exit depth
180     float rayEntryDepth = -((state.Q0.z + state.dQ.z*t_entry) /
181         (state.k0 + state.dk*t_entry)) * inv_far;
182     float rayExitDepth = -((state.Q0.z + state.dQ.z*t_exit) / (state.k0
183         + state.dk*t_exit)) * inv_far;
184     float rayDepthMin = min(rayEntryDepth, rayExitDepth);
185     float rayDepthMax = max(rayEntryDepth, rayExitDepth);
186     // sample list and check for collisions
187     bool collision = false;
188     for (state.layer = state.done ? state.layer : 0; state.layer < len
189         && !state.done; ++state.layer) {
190         uvec2 data = load(ptr, state.layer, state.miplevel);
191         float sceneDepthMin = unpack_depth(data);
192         float sceneDepthMax = (state.miplevel != 0 ?
193             unpack_depth_end(data) : sceneDepthMin) + THICKNESS;
194         if (sceneDepthMin > rayDepthMax) // early terminate
195             break;
196         if (sceneDepthMax < rayDepthMin) // advance list
197             continue;
198         collision = true;
199         // advance ray to point of collision
200         float f = clamp((sceneDepthMin - rayDepthMin) / (rayDepthMax -
201             rayDepthMin), 0, 1);
202         if (!state.done && ray.dir.z < 0)
203             state.t = max(state.t, t_exit * f + t_entry * (1-f));
204         break;
205     }
206     // check if done
207     if (collision && state.miplevel == 0)

```

```

199     state.done = true;
200 // update
201 if (!state.done) {
202     state.t = collision ? state.t : t_exit + EPSILON;
203     // update current mip map level
204     acceleration_delay = max(0, collision ? 3 : acceleration_delay -
205         1); // conservative delay
206     int new_mip = clamp(state.miplevel + (collision ? -1 : 1), 0,
207         mm_levels);
208     state.miplevel = collision || acceleration_delay == 0 ? new_mip :
209         state.miplevel;
210 }
211 // query warp occupancy
212 uint threadsTracing = bitCount(uint(ballotARB(true)));
213 uint threadsWantToReload = bitCount(uint(ballotARB(state.done)));
214 if (BOUNCES > 1 && threadsWantToReload > threadsTracing/BALLOT_N &&
215     state.done) {
216     // save hit
217     result.hit[bounces] = true;
218     result.hitPos[bounces] = getCurrCSPos();
219     ivec2 pixel = ivec2(Pcurr);
220     handle = list_heads[pixel.y * screendim.x + pixel.x];
221     pos = to_ptr(handle);
222     uvec4 data = load_uvec4(pos, state.layer);
223     result.hitNorm[bounces] = unpack_norm(data);
224     bounces++;
225     if (bounces >= BOUNCES)
226         return result;
227     // setup next bounce
228     vec3 dir = reflect(ray.dir, result.hitNorm[bounces-1]);
229     ray = Ray(result.hitPos[bounces-1]+dir*EPSILON, dir);
230     setupFor(ray);
231     acceleration_delay = 1;
232 }
233 if (threadsTracing <= BALLOT_KILL_N ||
234     allInvocationsARB(state.done))
235     break;
236 }
237 // end of trace loop
238
239 // save result if hasn't been already
240 if (state.done) {
241     result.hit[bounces] = true;
242     result.hitPos[bounces] = getCurrCSPos();
243     ivec2 pixel = ivec2(getCurrSSPos());
244     uint handle = list_heads[pixel.y * screendim.x + pixel.x];
245     uint pos = to_ptr(handle);
246     uvec4 data = load_uvec4(pos, state.layer);
247     result.hitNorm[bounces] = unpack_norm(data);
248 }
249 return result;
250 }

```