1 Pseudo code

Algorithm 1: GLSL pseudo-code for generating the bottom level of our traversal acceleration structure. \( \lambda_b \) and \( \lambda_d \) are set to \( 10^{-3} \) in our implementation.

```glsl
input : depth;
output: out;

1 D0, 0.2, 2 ← depth; /* node texture at level 0 */
2 O ← step(λ_b, getLaplacian(D)); /* compute forward and backward differentials */
3 Q ← vec2(D1,1 - D0,1, D1,1 - D1,0) /* compute angle differences via dot-product */
4 df_x ← normalize(cross(vec3(D1,1, x, 0, d_y), vec3(0, P.size, y, d_y))) /* compute normal */
5 df_y ← vec2(D1,1 - D0,1, D1,1 - D1,0) /* output plane node if the proxy is close enough to children in terms of */
6 df_z ← 1 - dot(N_proxy, N_0,3) /* output a plane node if the proxy is close enough to children in terms of */
7 df_z ← 1 - dist(O, P) /* output AABB node that encompasses all children */
8 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of orientation and position */
9 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
10 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
11 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
12 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
13 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
14 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
15 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
16 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
17 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
18 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
19 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
20 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
21 P ← −O.3, P_0.3 ← getMinMaxZ(O_0,3) /* output a plane node if the proxy is close enough to children in terms of */
```

We compare in Fig. 1 the depth reconstruction quality of our method against a gold-standard GPU ray tracer—NVIDIA OptiX [Parker et al. 2010]. Even though we only use two depth layers in this example, our approach correctly evaluates the depth at all pixels (except where three layers would be required), while being \( 3\times \) faster than general-purpose ray tracer. Note that our approach allows us to inpaint the remaining holes, so that no artifacts appear. Alternatively, one can simply use more layers.

References

Algorithm 3: GLSL pseudo-code for ray traversal through a single depth layer. For brevity, we assume the quad-tree MIPMAP has power-of-two size.

```cpp
input : Tn−1 : /* texture MIPMAP storing depth quad-tree */
input : R : /* ray structure storing direction and origin */
output: bool rayHit ; /* trace result */
output: bool occlusionHit ; /* did we hit an occlusion volume? */
output: float d ; /* hit point distance along the ray */
output: vec4 plane ; /* hit-plane data */
1 int Qn ← n − 1 ; /* current quad-tree level, start at the root */
2 vec2 Qxy ← \[ R.origin.xy \times \text{sizeof}(Tn)/2^Qn \] /* current node position in the quad-tree */

3 while insideBounds (pos, TQn) do
4 vec2 Qdata ← TQn(Qxy) ; /* read the node data */
5 if nodeStoresPlane (Qdata) then
6 plane ← getPlaneData (Qdata)
7 end
8 /* Get far and near Z of node */
9 FandN ← getFarNearOfNode (R, Qxy, Qn)
10 \[ \vec{N} \leftarrow \text{plane.xyz} ; \] /* plane normal */
11 \[ P₀ \leftarrow \text{vec3}(Qxy/2^Qn, \text{plane.u}) ; \] /* and origin */
12 /* compute ray-plane intersection */
13 d ← \frac{\text{dot}(P₀ − R.origin, \vec{N})}{\text{dot}(R.dir, \vec{N})}
14 if dot (R.dir, \vec{N}) > 0 then /* plane is front-facing the ray */
15 if d < FandN.near then occlusionHit ← true ;
16 if d ≥ FandN.near and d < FandN.far then
17 rayHit ← true
18 /* compute intersection with both node bounding box and occlusion volume */
19 (hitAABB, hitOV) ← rayIntersectAABB (R, Qdata)
20 if hitAABB then
21 if hitAABB.near = hitOV.far then
22 /* prevent the traversal from going above the quad-tree root */
23 Qn ← min (Qn, levelShift, n − 1) /* update the node location and level to new values */
24 Qxy ← \[ [(Qxy/2^Qn)−Qn] \] / 2^Qn /* did we hit an occlusion volume? */
25 Qn ← Qn
26 /* progress down to the next child */
27 Qxy ← getNextNode (R, Qxy, Qn)
28 /* compute how many levels up we need to go */
29 int levelShift ← findMSB ((Qxy \times Qn) \oplus Qn) /* prevent the traversal from going above the quad-tree root */
30 Qlevel ← min (Qlevel + levelShift, n − 1) /* update the node location and level to new values */
31 Qxy ← \[ [(Qxy/2^Qlevel)−Qlevel] \] / 2^Qlevel
32 /* continue */
33 else if hitOV then occlusionHit ← true ;
34 end
35 /* current node successor position at Qlevel */
36 /* progress down to the next child */
37 /* compute intersection with both node bounding box and occlusion volume */
38 /* compute how many levels up we need to go */
39 /* prevent the traversal from going above the quad-tree root */
40 Qlevel ← min (Qlevel + levelShift, n − 1) /* update the node location and level to new values */
41 Qxy ← \[ [(Qxy/2^Qlevel)−Qlevel] \] / 2^Qlevel
42 end
43 rayHit ← false
```

Algorithm 4: GLSL pseudo-code for finding coordinates of the next node that intersects with the ray in screen-space.

```cpp
Function getNextNode(Ray R, vec2 Qxy, int Qlevel)
/* get node exit corner coordinate */
1 vec2 B ← (Qxy + step(0, R.dir.xy)) / 2^Qlevel
2 /* get distances to node edges that intersect at B_xy */
3 vec2 D ← (B − R.origin.xy) / R.dir.xy
4 /* compute position shift */
5 vec2 Sxy ← sign(R.dir.xy) * step(D.xy, D.yz)
6 return Qxy + Sxy ; /* return new position */
```

Figure 1: A comparison of depth reconstruction quality. (a) A new view synthesized with our method (using two depth-layers). The full-image took 6.5ms to render. The synthesized depth for (b) two-layer and (c) single-layer configuration. (d) The reference was generated with NVIDIA OptiX in about 20ms. In both cases we report combined construction and tracing times.