An Adaptive Acceleration Structure for Screen-space Ray Tracing Supplemental Material

S. Widmer[1,2] D. Pająk[1,3] A. Schulz[4] K. Pulli[1,3] J. Kautz[1] M. Goesele[2] D. Luebke[1]

[1]NVIDIA [2]Graduate School of Computational Engineering, TU Darmstadt [3]Light [4]TU Darmstadt

1 Pseudo code

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

input : depth ; /* depth buffer data */
output: out ; /* node texture at level 0 */

$D_{0,0\ldots2,2} \leftarrow$ depth ; /* read 3x3 depth neighborhood */
/* discontinuity hint computed via Laplacian thresholding */
$O \leftarrow$ step(λ_d, getLaplacian (D))
/* compute forward and backward differentials */
$df_{xy} \leftarrow$ vec2 $(D_{2,1} - D_{1,1}, D_{1,2} - D_{1,1})$
$db_{xy} \leftarrow$ vec2 $(D_{1,1} - D_{0,1}, D_{1,1} - D_{1,0})$
/* enforce smoothness by picking the smallest derivative */
$d_{xy} \leftarrow$ mix(df_{xy}, db_{xy}, abs(df_{xy}) < abs(db_{xy}))
/* zero large derivatives that connect different surfaces */
$d_{xy} \leftarrow$ step(λ_h, abs(d_{xy})) $\cdot d_{xy}$
/* compute normal */
$\vec{N} \leftarrow$ normalize(cross(vec3 $(P_{size}.x, 0, d_x)$, vec3 $(0, P_{size}.y, d_y)$))
/* compute plane's top-left corner z-coordinate */
$P \leftarrow D_{1,1} - dot(\vec{N}_{xy}/\vec{N}_z, -0.5 \cdot P_{size})$
out $\leftarrow$ outputPlane $(\vec{N}, P, O)$; /* output a plane node */

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× 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

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4.

Algorithm 2: GLSL pseudo-code for construction and compression of a single level of the quad-tree.

input : in ; /* node texture at level i ($i > 0$) */
output: out ; /* node texture at level i-1 */

$Q_{0\ldots3} \leftarrow$ in ; /* read 2x2 neighboring nodes */
if containOnlyPlanes $(Q_{0\ldots3})$ **then**
  /* get plane normal vector, origin and "discontinuity" flag */
  $(\vec{N}, P, O)_{0\ldots3} \leftarrow$ getPlaneData $(Q_{0\ldots3})$
  /* set proxy plane normal to mean of children normals */
  $\vec{N}_{proxy} \leftarrow$ normalize(mean($\vec{N}_{0\ldots3}$))
  /* compute angle differences via dot-product */
  float $d_{0\ldots3} \leftarrow 1-$ dot($\vec{N}_{proxy}$, $\vec{N}_{0\ldots3}$)
  /* proxy plane origin P_{proxy} is least-square fit to child planes with fit errors stored in $p_{0\ldots3}$ */
  $(P_{proxy}, p_{0\ldots3}) \leftarrow$ getPlaneOrigin $(\vec{N}_{proxy}, \vec{N}_{0\ldots3}, P_{0\ldots3})$
  /* output a plane node if the proxy is close enough to children in terms of orientation and position */
  if max$(d_{0\ldots3}) < \gamma_{norm}$ **and** max$(p_{0\ldots3}) < \gamma_{dist}$ **then**
    out $\leftarrow$ outputPlane $(\vec{N}_{proxy}, P_{proxy}$, any$(O_{0\ldots3}))$
    return
  end
end
/* output AABB node that encompasses all children */
vec2 $Z_{0\ldots3} \leftarrow$ getMinMaxZ $(Q_{0\ldots3})$
float $min_z \leftarrow$ min($Z_{0\ldots3}$.x)
float $max_z \leftarrow$ max($Z_{0\ldots3}$.y)
out $\leftarrow$ outputAABB (min_z, max_z)

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.

```
input  : T_{0···n−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 */
int Q_{level} ← n − 1 ;                         /* current quad-tree level, start at the root */
/* current node position in the quad-tree */
ivec2 Q_{xy} ← ⌊R.origin.xy ∗ sizeof(T_0)/2^{Q_{level}}⌋
while insideBounds (pos, T_{Q_{level}}) do
    vec2 Q_{data} ← T_{Q_{level}}(Q_{xy}) ;     /* read the node data */
    if nodeStoresPlane (Q_{data}) then
        plane ← getPlaneData (Q_{data})
        /* Get far and near Z of node */
        FandN ← getFarNearOfNode (R, Q_{xy}, Q_{level})
        \vec{N} ← plane.xyz ;                   /* plane normal */
        P_0 ← vec3 (Q_{xy}/2^{Q_{level}}, plane.w) ;   /* and origin */
        /* compute ray-plane intersection */
        d ← dot(P_0 − R.origin, \vec{N})/dot(R.dir, \vec{N})
        if dot(R.dir, \vec{N}) > 0 then /* plane is front-facing the ray */
            if d < FandN.near then occlusionHit ← true ;
            if d ≥ FandN.near and d < FandN.far then
                rayHit ← true
                plane ← vec4 (\vec{N}, dot(P_0, \vec{N}))
                return
            end
        else /* plane is back-facing the ray */
            if d ≥ FandN.near then occlusionHit ← true ;
        end
    else
        /* compute intersection with both node bounding box and occlusion
           volume */
        (hitAABB, hitOV) ← rayIntersectAABB (R, Q_{data})
        if hitAABB then
            if hitAABB.near = hitOV.far then
                occlusionHit ← true
            end
            /* progress down to the next child */
            ip ← R.dir.xy ∗ hitAABB.near + R.origin.xy
            Q_{xy} ← Q_{xy} ∗ 2 + step(0, ip − (Q_{xy} + 0.5)/2^{Q_{level}})
            Q_{level} ← Q_{level} − 1
            continue
        else
            if hitOV then occlusionHit ← true ;
        end
    end
    /* plane or AABB miss, progress to the next node */
    /* current node successor position at Q_{level} */
    Q^*_{xy} ← getNextNode (R, Q_{xy}, Q_{level})
    /* compute how many levels up we need to go */
    int levelShift ← findMSB ((Q_x ⊕ Q^*_x) | (Q_y ⊕ Q^*_y))
    /* prevent the traversal from going above the quad-tree root */
    Q^*_{level} ← min(Q_{level} + levelShift, n − 1)
    /* update the node location and level to new values */
    Q_{xy} ← ⌊Q^*_{xy}/2^{(Q^*_{level} − Q_{level})}⌋
    Q_{level} ← Q^*_{level}
end
rayHit ← false
```

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

```
Function getNextNode(Ray R, ivec2 Q_{xy}, int Q_{level})
    /* get node exit corner coordinate */
    vec2 B ← (Q_{xy} + step(0, R.dir.xy)) / 2^{Q_{level}}
    /* get distances to node edges that intersect at B_{xy} */
    vec2 D ← (B − R.origin.xy) / R.dir.xy
    /* compute position shift */
    ivec2 S_{xy} ← sign(R.dir.xy) ∗ step(D.xy, D.yx)
    return Q_{xy} + S_{xy};                     /* return new position */
```

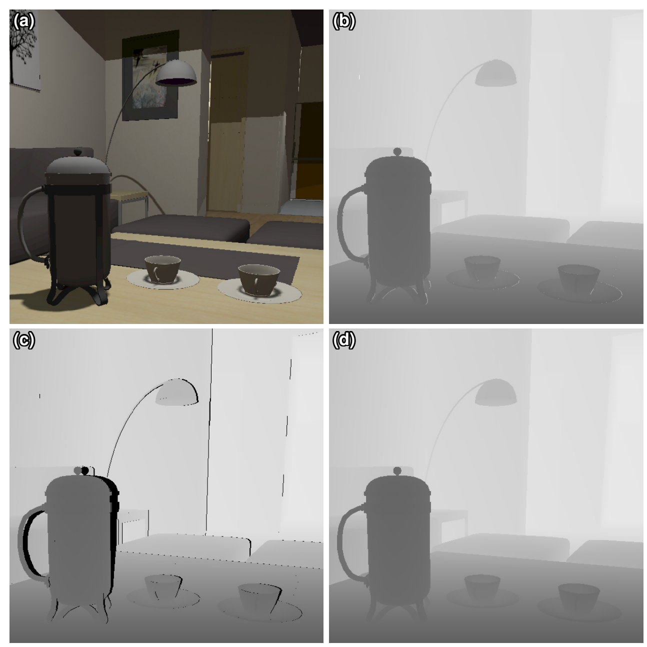

Figure 1: *A comparison of depth reconstruction quality.* ***(a)*** *A new view synthesized with our method (using two depth-layers). The full-image took 6.5ms to render. The synthesized depth for* ***(b)*** *two-layer and a* ***(c)*** *single-layer configuration.* ***(d)*** *The reference was generated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