Appendix: Compact tetrahedralization-based acceleration structures for ray tracing

Aytek Aman · Serkan Demirci · Uğur Güdükbay

A *Tet20* and *Tet16* Tetrahedral Mesh Representations

In *Tet20*, we get rid of vertex indices and store only the xor-sum and the neighboring indices. We use the xor-sum field to get the index of the unshared vertex of the next tetrahedron during traversal. To do this, shared vertices between two tetrahedra must be known. This is guaranteed by *ray connectivity*, meaning that the start and endpoints of rays are always connected in a typical ray-tracing scenario. However, we use a *source tet*, a tetrahedron with complete index information, to initialize the indices at the beginning. We can choose this tetrahedron randomly. Starting from *source tet*, it is possible to reconstruct the indices of the neighboring tetrahedra. It should be noted that we need to sort the neighbor indices in a tetrahedron using their corresponding vertex indices to find the neighbor for a given vertex index. Figure 1 shows the memory representation of the *Tet20* structure.

VX^i	N_0^i	N_1^i	N_2^i	N_3^i

Fig. 1 *Tet20* structure. Each field is an integer and four bytes long. The tetrahedron data occupies 20 bytes of memory.

In *Tet16*, instead of storing four neighbor indices explicitly, we store three values that can be used to reconstruct neighbor indices, given that the previous (neighbor) tetrahedron index is known. We compute these three indices as follows.

$$NX_0^i = N_0^i \oplus N_3^i$$

$$NX_1^i = N_1^i \oplus N_3^i$$

$$NX_2^i = N_2^i \oplus N_3^i$$

Knowing the index of a neighbor tetrahedron and its order, we can reconstruct the rest of the neighbors easily. For example, If we have N_2^i, we retrieve N_1^i as follows.

$$N_2^i = N_2^i \oplus NX_2^i \oplus NX_1^i$$

The resulting *Tet16* structure is given in Figure 2.

A. Aman · S. Demirci · U. Güdükbay
Department of Computer Engineering, Bilkent University, 06800, Ankara, Turkey
E-mail: aytek.aman@cs.bilkent.edu.tr (A. Aman), serkan.demirci@bilkent.edu.tr (S. Demirci), gudukbay@cs.bilkent.edu.tr (U. Güdükbay)
Tel.: +90-312-290-1386
Fax: +90-312-266-4047

VX^i	NX_0^i	NX_1^i	NX_2^i

Fig. 2 *Tet16* structure. Each field is an integer and four bytes long. The tetrahedron data occupies 16 bytes of memory.

B Tetrahedron Traversal for *Tet20* and *Tet16* Representations

In *Tet20*, we use the property that neighbor indices are sorted using their counterpart vertex indices as keys. Thus, to find the next neighbor index, we find the order of $idx_{exit_face_idx}$ among idx_0, idx_1, idx_2, idx_3 (which are actually the vertex indices of the tetrahedron). Because the neighbor indices are sorted using vertex indices, order of the vertex index also happens to be the next neighbor index. We describe this process in Algorithms 1 and 2.

Algorithm 1 Tetrahedron traversal loop for *Tet20*

while $tet_{idx} \geq 0$ **do**
  $idx_{exit_face_idx} \leftarrow idx_3$
  $idx_3 \leftarrow idx_0 \oplus idx_1 \oplus idx_2 \oplus VX^{tet_idx}$
  $v_{new} \leftarrow points_{idx_3} - r_o$
  $p_{exit_face_idx} \leftarrow p_3$
  $p_3 \leftarrow (\vec{u} \cdot v_{new}, \vec{v} \cdot v_{new})$
  $exit_face_idx = \textsc{GetExitFace}(p_{0..3})$
  $order_a \leftarrow$ sorted order of id_3 among id_i
  $next_tet_idx = \textsc{GetNextTet20}(tet_idx, order_a)$
end while

Algorithm 2 Next tetrahedron determination for *Tet20*

procedure $\textsc{GetNextTet20}(tet_idx, order_a)$
  $next_tet_idx \leftarrow N_{order_a}$
  return $next_tet_idx$
end procedure

In *Tet16*, we use the previous tetrahedron index to reconstruct next tetrahedron index using the values NX_j^i. As in $Tet20$, we need to construct the value NX_j^i using sorted vertex indices. To reconstruct the next tetrahedron, sorted order of values are computed for idx_3, which corresponds to a previous tetrahedron and $idx_{exit_face_idx}$, which corresponds to an exit face, must be computed. We describe this process in Algorithms 3 and 4.

Algorithm 3 Tetrahedron traversal loop for *Tet16*

while $tet_idx \geq 0$ **do**
  $idx_3 \leftarrow idx_0 \oplus idx_1 \oplus idx_2 \oplus VX^{tet_idx}$
  $v_{new} \leftarrow points_{idx_3} - r_o$
  $p_3 \leftarrow (\vec{u} \cdot v_{new}, \vec{v} \cdot v_{new})$
  $order_a \leftarrow$ sorted order of id_3 among id_i
  $exit_face_idx = \textsc{GetExitFace}(p_{0..3})$
  $order_b \leftarrow$ sorted order of $id_{exit_face_idx}$ among id_i
  $next_tet_idx =$
   $\textsc{GetNextTet16}(tet_idx, prev_tet_idx, order_a, order_b)$
  $\textsc{Swap}(tet_idx, prev_tet_idx)$
end while

Algorithm 4 Next tetrahedron determination for *Tet16*

procedure GetNextTet16($tet_idx, prev_tet_idx, order_a, order_b$)
  if $order_a \neq 3$ **then**
    $next_tet_idx = prev_tet_idx \oplus NX^{tet_idx}_{order_a}$
  end if
  if $order_b \neq 3$ **then**
    $next_tet_idx = next_tet_idx \oplus NX^{tet_idx}_{order_b}$
  end if
  return $next_tet_idx$
end procedure

C Point Projection Using Specialized Basis

We project newly fetched points to the two-dimensional (2D) coordinate system using two dot product operations, which require six floating-point multiplications and four floating-point additions. We can optimize this step by scaling the basis vectors to make some of the components zero or one. Since the basis vectors are only scaled, the exit face determination still works correctly. To avoid numerical issues, we scale vectors in such a way that only the absolute largest components become one (or minus one). Equation (1) describes the construction of the first basis vector $\vec{u}$, which is orthogonal to $\vec{n}$ (and not necessarily of unit length).

$$\begin{aligned} \vec{u}_{min} &= 0, \\ \vec{u}_{(min+1)\ mod\ 3} &= \frac{\vec{n}_{(min-1)\ mod\ 3}}{\vec{n}_{max}}, \\ \vec{u}_{(min-1)\ mod\ 3} &= -\frac{\vec{n}_{(min+1)\ mod\ 3}}{\vec{n}_{max}}, \end{aligned} \tag{1}$$

where $\vec{v}_0$, $\vec{v}_1$, and $\vec{v}_2$ correspond to $\vec{v}_x$, $\vec{v}_y$, and $\vec{v}_z$, respectively, min and max are the indices of the absolute smallest and largest components of the vector $\vec{n}$.

We construct the second basis vector $\vec{v}$, which is orthogonal to $\vec{n}$ and $\vec{u}$ (and not necessarily of unit length), as in Equation (2).

$$\begin{aligned} \vec{t} &= \vec{n} \times \vec{u}, \\ \vec{v} &= \frac{\vec{t}}{\vec{t}_{(3-max-min)}}. \end{aligned} \tag{2}$$

Now, we can transform three-dimensional (3D) point v to the 2D coordinate system using the basis $b = (\vec{u}, \vec{v})$, as shown in Equation 3. It should be noted that the sign s of the last parameter $\vec{v}_{min}$ can be either positive or negative depending on the sign of $\vec{v}_{min}$.

$$\begin{aligned} other &= 3 - max - min, \\ \vec{p}_x &= \vec{u}_{max}\vec{v}_{max} + \vec{v}_{other}, \\ \vec{p}_y &= \vec{v}_{max}\vec{v}_{max} + \vec{v}_{other}\vec{v}_{other} \pm \vec{v}_{min}. \end{aligned} \tag{3}$$

To perform the above computation, three floating-point multiplications and three floating-point addition/subtractions are sufficient. We implement this fast projection method using a templated function over the variables min, max, and $sign(\vec{v}_{min})$ and call the corresponding function by inspecting the components of the new basis to avoid run-time overhead of keeping additional function arguments.

D Handling Common Ray-tracing Operations

We handle common ray-tracing operations using tetrahedral meshes as follows. Handling mesh lights is straightforward by using the proposed traversal methods. For point lights, we locate the tetrahedron that contains the point light at the start of each frame. Then, we use a slightly modified traversal algorithm where the traversal terminates if the tetrahedron that contains the light source is reached. We cast reflection and refraction rays using the neighboring tetrahedron on the shared face of the tetrahedron in

which the traversal is terminated. In this way, we avoid an intersection with the same face. To handle shadow, reflection, and refraction rays together, we report the two tetrahedra that share the common intersected face in the intersection routine. Figure 3 illustrates different types of rays used in a tetrahedral mesh-based ray tracing. At the start of each frame, we locate the camera and the point light sources and store their tetrahedron indices. For this purpose, we start from a source tetrahedron S that can be arbitrarily chosen and locate the tetrahedra that contain the camera and the point light sources.

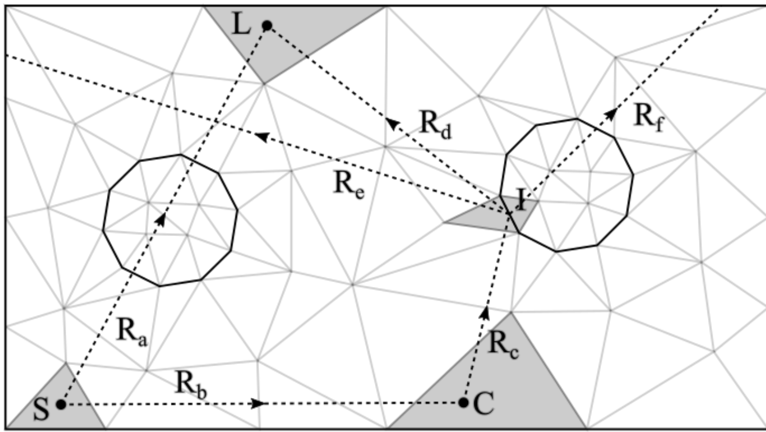
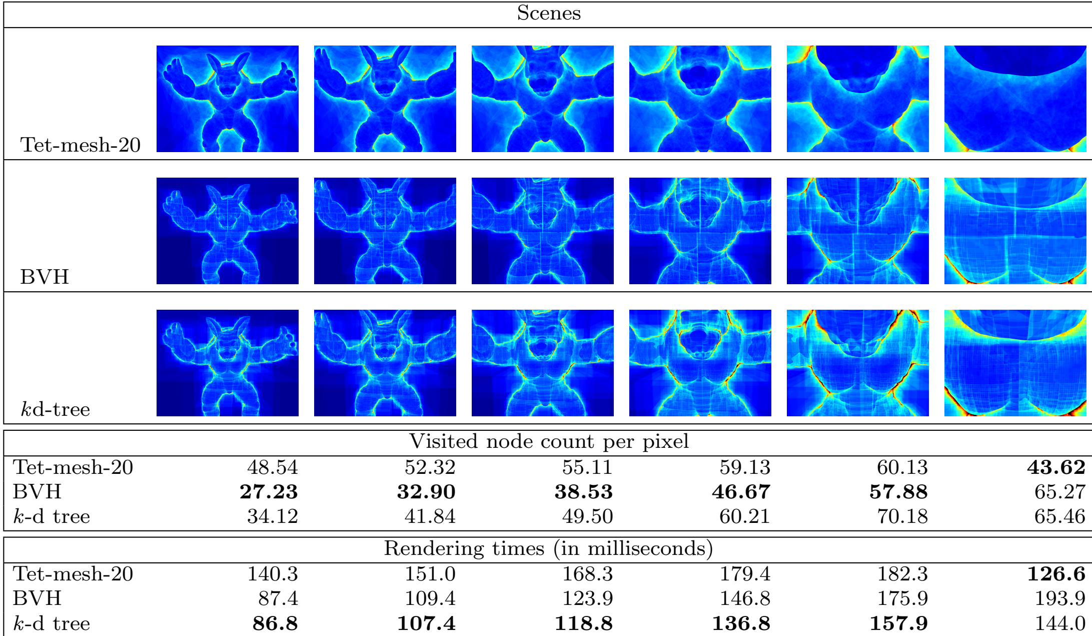


Fig. 3 The types of rays in tetrahedral mesh-based ray tracing. S is the center of source tetrahedron. C is the camera position. I is the intersection point. L is the light source position. Rays R_a and R_b are used to locate the light source and camera position, respectively. R_c is the camera ray. R_d is the shadow ray. R_e and R_f are the reflection and refractions rays, which are cast from two different neighboring tetrahedra.

E Experimental Results Regarding the Camera Distance

Table 1 demonstrates the efficiency of a tetrahedral mesh-based traversal approach when the camera gets closer to a surface. In this experiment, we render the images at varying distances to the Armadillo 3D model and compare the rendering times for different acceleration structures. Both BVH and k-d tree performs much better than the tetrahedral mesh structure when the camera views the object from a fair distance. However, as the camera gets closer to the surface, the traversal cost decreases because the tetrahedral mesh is not hierarchical, unlike the BVH and k-d tree. In the extreme case, when the camera is about to touch the surface, only one tetrahedron is traversed. This is not the case for hierarchical structures because many tree nodes may need to be traversed to find the closest ray-surface intersection.

Table 1 The rendering times and visited node counts for different types of accelerators as the camera gets closer to the mesh surface. We compare our *Tet-mesh-20* structure and traversal method with BVH (Pharr et al., 2016) and k-d tree (Pharr et al., 2016).



Visited node count per pixel						
Tet-mesh-20	48.54	52.32	55.11	59.13	60.13	**43.62**
BVH	**27.23**	**32.90**	**38.53**	**46.67**	**57.88**	65.27
k-d tree	34.12	41.84	49.50	60.21	70.18	65.46

Rendering times (in milliseconds)						
Tet-mesh-20	140.3	151.0	168.3	179.4	182.3	**126.6**
BVH	87.4	109.4	123.9	146.8	175.9	193.9
k-d tree	**86.8**	**107.4**	**118.8**	**136.8**	**157.9**	144.0

References

Pharr M, Jakob W, Humphreys G (2016) Physically Based Rendering: From Theory to Implementation, 3rd edn. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA