Computational Visual Media
https://doi.org/10.1007/s41095-0xx-xxxx-x

Research Article

Supplementary Note III - Temporal Vectorized Visibility for Direct Illumination of Animated Models

Zhenni Wang[1], **Tze Yui Ho**[1], **Yi Xiao**[2](✉) **and Chi Sing Leung**[1]

Abstract abstract

Keywords keywords

1 More implementation details

This supplementary note documents more implementation details of our algorithm *temporal vectorized visibility*, the BVH generation time. These information might be helpful for readers to better understand our algorithm.

We apply our algorithm to a direct illumination rendering application with an animated 3D model. Fig. 1 shows the flowchart of our application. Our application consists of two major components. The first component is written in CUDA for the temporal vectorized visibility generation, and the second component uses Vulkan and RTX to invoke the hardware accelerated ray tracing for rendering the direct illumination.

In the first component, we use our front back edge oriented infinite triangle BVH traversal algorithm to generate temporal vectorized visibility. For each frame, after the model animation, we build a BVH over the updated model. We construct bottom up BVH using the parallel locally-ordered clustering BVH generation algorithm [1] and the BVH library [2]. The BVH is generated in CPU, and then we upload the BVH and the model data to CUDA for the infinite triangle and the chosen ray BVH traversal.

Similar to [3], we sample temporal vectorized visibility per vertex. Given the input model data, we gather all front back edges and orphan edges of each vertex using Eq. (7) in the manuscript and generate infinite triangles. Alg. 1 shows the definition of our infinite triangle. The generated infinite triangles then traverse the BVH in parallel by using the algorithm described in Section 3 in the manuscript.

Algorithm 1 The definition of our infinite triangle and intersection output.

class Edge
1: int ui; `// Starting point index`
2: int vi; `// End point index`
3: int ti; `// Owner triangle index`
4: int $pair$; `// Edge pair index`
class Infinite triangle
5: vec3 ***origin***;
6: int e_i; `// the index of the specified front back edge`
class IntersectOutput
7: float p_r; `// Relative distance of` $P.e$
8: float e_r; `// Relative distance of` ei
9: int e_i; `// Intersecting front back edge index`
10: int δ; `// Delta occlusion count`

During the infinite triangle BVH traversal, we perform the triangle-AABB intersection at the branch nodes. At the leaf nodes, we first identify the front back edges of the scene triangles belonging to the lead nodes. Then, we perform the *infinite triangle vs. front back edge intersection*. The relative distance, the delta occlusion counts, and the indices of the intersected front back edges are recorded as the intersection output (see Alg. 2).

When an infinite triangle BVH traversal finishes, we have all intersections and the corresponding sectors. Then, we perform a ray tracing in a sector to get an exact occlusion count. We randomly choose a sector and randomly choose a ray in it. The chosen ray has a ray origin at the vertex and a ray direction within the chosen sector. The occlusion count of the chosen ray begins with 0.

1 City University of Hong Kong, Hong Kong, China. E-mail: Z. Wang, zhenni126@126.com; T.Y. Ho, ma_hty@hotmail.com; C.S. Leung, eeleungc@cityu.edu.hk.

2 Hunan University, Changsha, China. E-mail: yixiao1984@gmail.com.

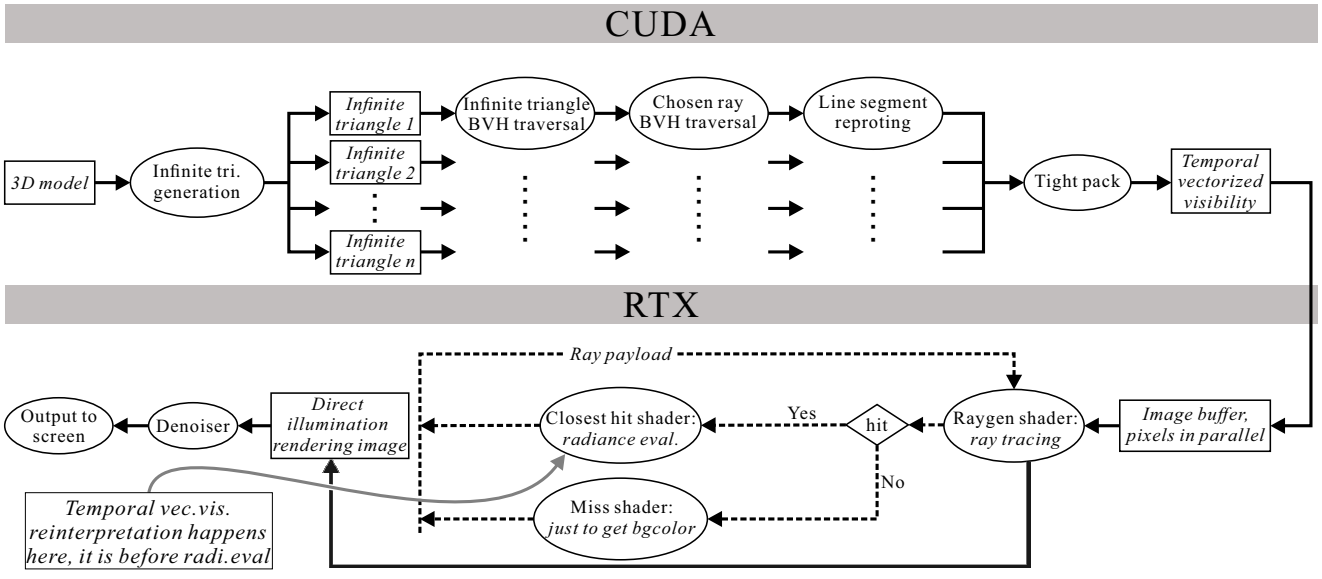

Fig. 1 The flowchart of our application.

We implement the chosen ray BVH traversal in CUDA. During the traversal, we perform the ray-AABB intersection at the branch nodes. At the leaf nodes, we perform a ray-triangle intersection test and increment the occlusion count for a positive result. With the exact occlusion count and the delta occlusion counts, we deduce the occlusion counts of all sectors and report line segments from each visible sector.

For both memory access efficiency and easier manipulation, the reported line segments are tight packed and grouped per vertex using *Thrust Library* [4], a built-in template library from CUDA Toolkit. The prefix sums of the number of the reported line segments per vertex are also prepared for fetching data from the tight packed results. The tight packed result is the temporal vectorized visibility of all vertices.

Note that some precautions must be implemented in the above processes. First, we will ignore the front back edges that are close to being parallel to the infinite triangle during the *infinite triangle vs. front back edge intersection*. Ignoring such edges might cause gaps in temporal vectorized visibility and introduce errors to the rendering results. However, as we just sparsely sample the 3D model surface for temporal vectorized visibility, the chance of encountering this numerically important situation is small and can be ignored.

Second, we will ignore the owner triangles of the specified front back edge during the chosen ray-triangle intersection. The chosen ray touches the specified front back edge of the infinite triangle. During the chosen ray BVH traversal, the chosen ray will more than likely encounter the two owner triangles. In this case, the chosen ray may view these triangles as occluders and increase the occlusion count by one or two randomly. Fortunately, each chosen ray will only have two of such triangles, and we can conditionally ignore them by using the triangle indices.

Our temporal vectorized visibility generation does not require any geometric animation in advance, and the visibility is generated in real time, instead of using pre-computation. As the temporal vectorized visibility can be shared across time, we generate new temporal visibility on demand, e.g. a generation every 8 frames. If a generation does take place, all generated temporal vectorized visibility, i.e. the tight packed line segments, is promptly uploaded to RTX for the radiance evaluation, i.e. the second components.

In the second component, we utilize hardware accelerate ray tracing written in Vulkan to render the direct illumination. The BVH of the animated 3D model is prepared again per frame, and this time we are using the display card driver to generate it. The rays have the ray origin at the camera position and the ray direction from the camera through pixels, i.e. one ray per pixel. We prepare rays in the Vulkan raygen shader and apply the rays to traverse the BVH.

When a ray reaches the closest hit shader, i.e. the ray has a closest hit position on the scene triangle, we evaluate the direct illumination of the hit position using the temporal vectorized visibility radiance evaluation. The conversion of our endpoint representation to position vectors happens right before the radiance evaluation, and this facilitates the temporal vectorized visibility sharing across time. The illumination contribution is stored in the ray payload and added to the total contribution when the ray returns to the raygen shader.

Note that in the closest hit shader, we randomly choose a

Algorithm 2 *Infinite triangle vs. front back edge intersection.*

Local : *buf* - a buffer for storing intersection outputs
Global: *vbuf* - the vertex buffer
Global: *ebuf* - the edge buffer

foreach edge index $e_i \in leaf\_node$ **do**
  Edge $e \leftarrow$ *ebuf*$[e_i]$
  `// Eq. (7) in the manuscript`
  if isFrontbackedge(P.***origin***, e) **then**
    $IntersectOutput\ output$;
    if isEdgeIntersect(P, p_e, e, e_i, $output$) **then**
      buf $\leftarrow output$
end foreach
function *isEdgeIntersect(P, p_e, e, e_i, $output$)*
  vec3 $\boldsymbol{p_u} \leftarrow$ *vbuf*$[p_e.ui]$;
  vec3 $\boldsymbol{p_v} \leftarrow$ *vbuf*$[p_e.vi]$;
  vec3 $\boldsymbol{e_u} \leftarrow$ *vbuf*$[e.ui]$;
  vec3 $\boldsymbol{e_v} \leftarrow$ *vbuf*$[e.vi]$;
  vec3 $\boldsymbol{p_n} = (\boldsymbol{p_v} - P.\boldsymbol{origin}) \times (\boldsymbol{p_u} - P.\boldsymbol{origin})$;
  vec3 $\boldsymbol{e_n} = (\boldsymbol{e_v} - P.\boldsymbol{origin}) \times (\boldsymbol{e_u} - P.\boldsymbol{origin})$;
  `// triangle-line segment intersection`
  if isTriLineIntersect(P, e) **then**
    `// Eq. (10) in the manuscript`
    $output.p_r = RD(P.\boldsymbol{origin}, \boldsymbol{e_n}, \boldsymbol{p_u}, \boldsymbol{p_v})$;
    $output.e_r = RD(P.\boldsymbol{origin}, \boldsymbol{p_n}, \boldsymbol{e_u}, \boldsymbol{e_v})$;
    $output.e_i = e_i$;
    $output.\delta = getDelta(P, e, output.e_r, \boldsymbol{p_n})$;
    return true;
  else
    return false;
  end if
function *getDelta($P, e, e_r, \boldsymbol{p_n}$)*
  `// change relative distance to point`
  `// Eq. (11) in the manuscript`
  vec3 $\boldsymbol{p_t} = evalRD(e, e_r)$;
  `// the other intersection point`
  `between` P `and` $e.ti$
  vec3 $\boldsymbol{p'_t} = tri\_tri\_intersect(P, e.ti)$;
  `// Eq. (9) in the manuscript`
  int $\delta = (\boldsymbol{p'_t} - \boldsymbol{p_t}) \cdot (\boldsymbol{p_t} - P.\boldsymbol{origin}) \times \boldsymbol{p_n} > 0$? 1: -1;
  if isNonOrphanEdge(e) **then**
    δ = δ*2
  return δ

vertex of the hit triangle using the barycentric coordinate, and the temporal vectorized visibility of the chosen vertex is used for the radiance evaluation. This is significantly faster than [3], which needs to evaluate all three vertices for linear interpolation. The downside is it will give us a tiny bit of noise. As the final touch, we apply the AI denoiser to the rendering images.

References

[1] Meister D, Bittner J. Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE transactions on visualization and computer graphics*, 2017, 24(3): 1345–1353.

[2] Gayot AP. A modern C++17 header-only BVH library, 2020.

[3] Ho TY, Xiao Y, Feng RB, Leung CS, Wong TT. All-Frequency Direct Illumination with Vectorized Visibility. *IEEE transactions on visualization and computer graphics*, 2015, 21(8): 945–958.

[4] NVIDIA. Thrust Quick Start Guide, 2011.