

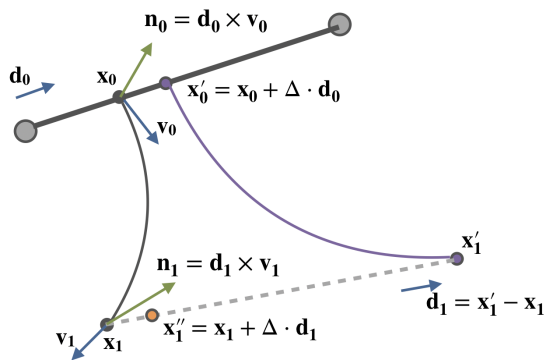


# Progressive Rendering of Transparent Integral Surfaces Additional Material

Xingze Tian  and Tobias Günther 

Department of Computer Science, ETH Zürich, Switzerland



**Figure 1:** To estimate the normal, we seed a ghost particle on the seed curve. If the ghost moves too far away from our particle, it is reset (orange point). The normal is computed as the cross product between flow tangent  $\mathbf{v}_i$  and the direction to the ghost particle  $\mathbf{d}_i$ .

## 1. Implementation Details

**Normal Estimation.** To shade the surfaces, we estimate normals similar to Machado et al. [MSE14], i.e., we approximate the normal of each vertex by advecting a ghost particle, which is reseeded, as illustrated in Fig. 1. For each point  $\mathbf{x}$  on the seeding curve, we select its neighbour  $\mathbf{x}'$  by stepping along the seeding curve in direction  $\mathbf{d}$ , i.e.  $\mathbf{x}' = \mathbf{x} + \Delta \mathbf{d}$ . The normal  $\mathbf{n}$  is then computed as the cross product of  $\mathbf{d}$  and the flow direction  $\mathbf{v}$ . We trace both the sampled particle and its neighbour, and check if their distance to each other remains smaller than  $\Delta$ . If not, we move the neighbour closer to the particle. The reseeded threshold  $\Delta$  should be small enough to ensure an adequate normal estimation. We empirically chose  $\Delta = 10^{-5}$ .

**Per-Pixel Tree Construction.** The unsorted fragment-linked lists of the geometry are then passed to the second step to construct a tree per pixel that maintains the surface layers visible in the pixel. In this step, a full-screen quad is rendered such that each pixel is managed by only one thread. Similar to the fragments, tree nodes also store data (color, depth, uv coordinates and normal). Instead of having one pointer to the next fragment, a tree node has two pointers to its left and right children. The construction algorithm is listed in Alg. 1 and is illustrated in Fig. 2. First, we check if a tree node has been allocated. If not, we use the `CreateNode` function to

**Algorithm 1** Algorithm for per-pixel tree construction.

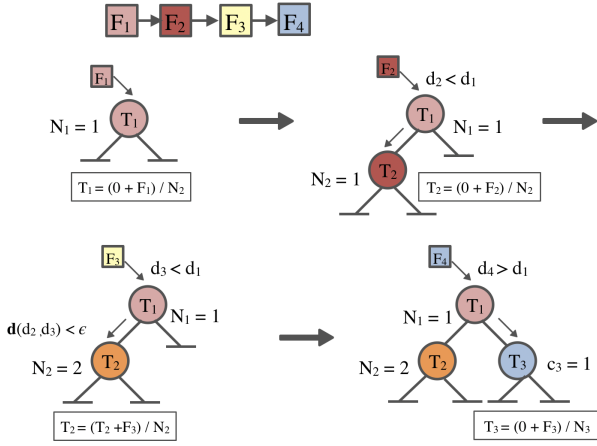
```

function BUILDTREE (fragmentsBuffer, treesBuffer, pixelId)
  [f]  $\leftarrow$  fragmentsBuffer[pixelId]
  [root]  $\leftarrow$  treesBuffer[pixelId]
  while f.next  $\neq$  null do
    if root = null then
      root  $\leftarrow$  CreateNode(f)  $\triangleright$  initialize new node
    end if
    Traverse(f, root)
    f  $\leftarrow$  f.next
  end while
end function

function TRAVERSE (frag, node)
  if node = null then
    node  $\leftarrow$  CreateNode(frag)  $\triangleright$  initialize new node
    return
  end if
  if Diff(frag.Data.depth, node.Data.depth)  $<$   $\epsilon$  then
    node.Data  $\leftarrow$  Average(node.Data, frag.Data)
    return
  end if
  if frag.Data.depth  $<$  node.Data.depth then
    Traverse(frag, node.left)
  else
    Traverse(frag, node.right)
  end if
end function

```

allocate memory and initialize the tree node with the first fragment in the linked list and set its children to NULL. We iterate all fragments in the fragment linked list, and `Traverse` the tree to find their closest nodes. Distance is thereby expressed by the function `Diff`, which measures the view space squared depth difference between the fragment to insert and the respective tree node. If we have found a tree node that is close enough to the fragment, i.e., the fragment belongs to a surface layer that has already been rasterized into the pixel, we merge the fragment into the node by averaging its data (color, depth, uv coordinates and normal) and update the node with the average. The running average is calculated by the function `Average`. If the tree node is not a match, we check if the fragment



**Figure 2:** Tree construction example for four fragments being inserted into the same pixel. The first fragment (light red) allocates the first node. The second fragment (red) is closer to the camera and is therefore inserted as left child. The third fragment (yellow) belongs to the surface layer of the previous red fragment and is therefore averaged to orange. The fourth fragment (blue) is behind the first fragment and therefore allocates a node on the right.

is in front of the tree node, i.e., the depth of the fragment is smaller than the depth of the tree node. If yes, we traverse the node's left child, otherwise we go right. If the tree node does not have a left or right child, we create a new child node.

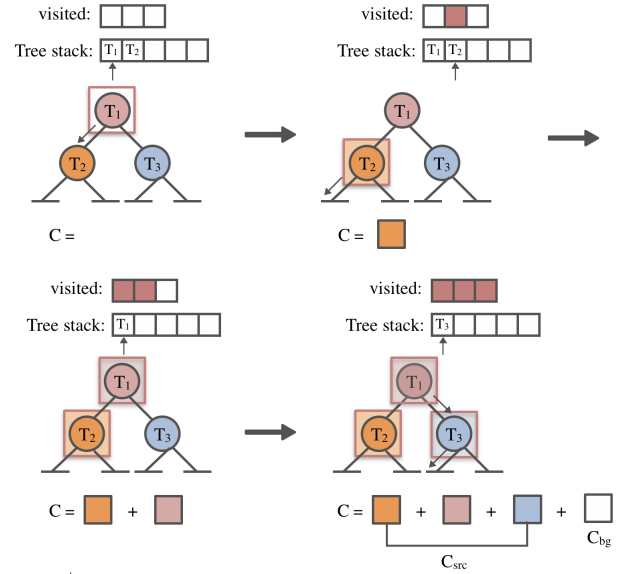
#### Algorithm 2 Algorithm for fragment composition

```

visited[] ← false           ▷ initialize all nodes as unvisited
function BLENDTREE (treesBuffer, pixellId, Cbg)
    [root] ← treesBuffer [pixellId]
    Csrc = DFS (root, 0)     ▷ initialize final color as 0
    return BLEND (Csrc, Cbg)
end function

function DFS (node, C)
    if (!visited [node]) then
        if (node.left ≠ NULL and !visited [node.left]) then
            return DFS (node.left, C)     ▷ go left if possible
        else
            C ← Blend (C, node)           ▷ add color
            visited [node] ← true         ▷ mark as read
            return DFS (node.right, C)    ▷ go right
        end if
    end if
end function
    
```

**Compositing of Transparent Tree Nodes.** The per-pixel tree nodes store the average properties of the fragments that have been merged into surface layers. To compose the final transparent image, we apply the front-to-back blending equation [HLSR09], which requires a sorted traversal of the layers. Since our binary trees are constructed based on the view space depth, a traversal from the left most tree node in the depth-first order obtains a sorted list of layers. The traversal algorithm is listed in Alg. 2 and is illustrated by



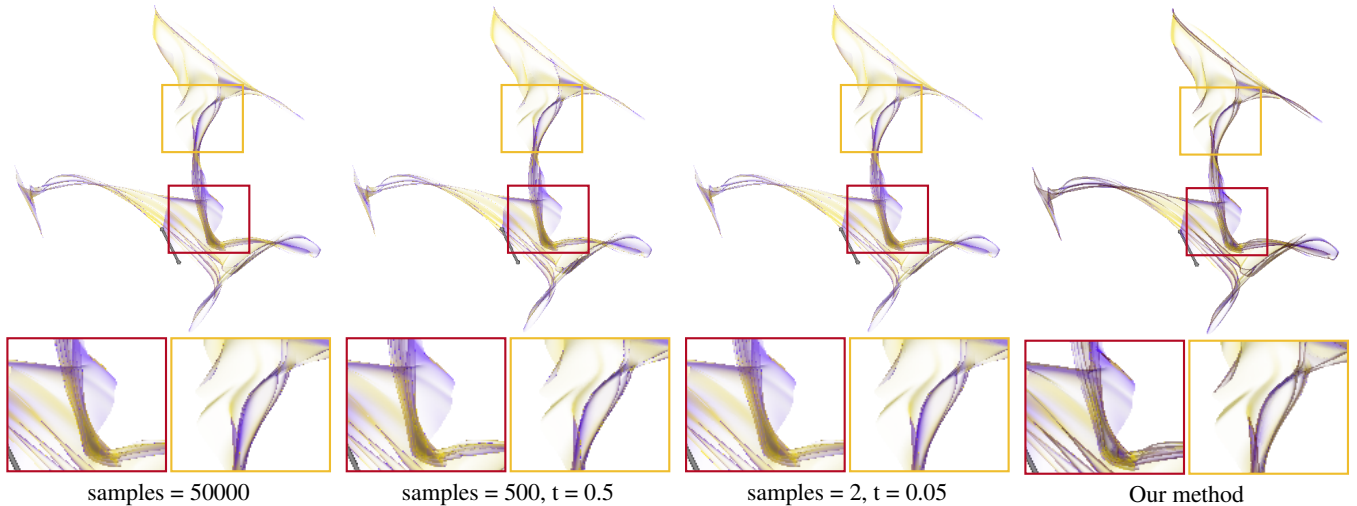
**Figure 3:** Illustration of the blending traversal for a tree with three nodes. To avoid recursion, we implemented the traversal iteratively with a stack data structure. Starting from the root node  $T_1$ , the left child is traversed. Since the left child  $T_2$  has no further children, it is blended in with front-to-back blending. Backtracking leads to the first node  $T_1$ , which then blends its color into the pixel, before ascending to its right node  $T_3$ , which blends its color last.

an example in Fig. 3. Starting from the root node, we continuously check its left child. If a tree node has no unvisited left child, we blend it into the final result  $C$  and mark the node as visited. After blending the tree node itself, we go right and blend its right child.

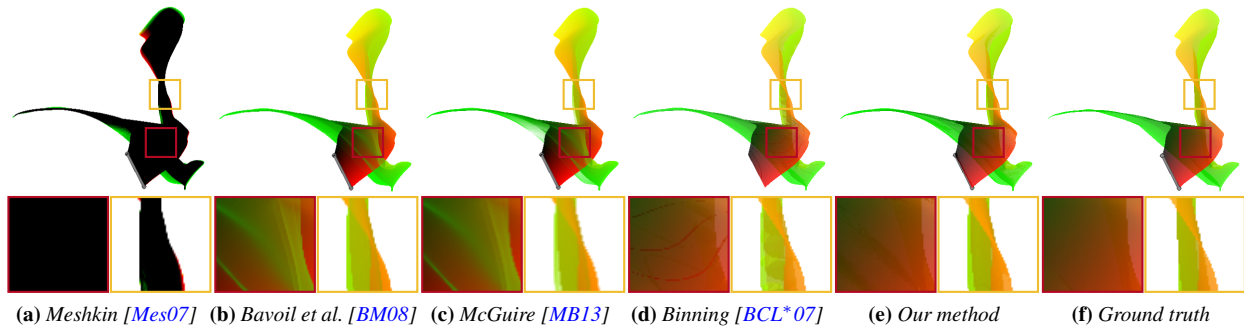
## 2. Further Evaluation

### 2.1. Comparison with Surface Integrators

Unlike traditional surface integration methods, our image-based renderer does not require frontline refinement or a refinement of the full surface. In Fig. 4, we compare our approach with three existing integrators in the ABC flow. First, we test a frontline integrator that does not apply refinement, which will suffer quickly from degeneration of triangles. To obtain a meaningful surface, we needed about 5000 particles on the frontline in order to represent the stretching surface adequately. Still, there are noticeable artifacts in the zoomed in area. Second, we apply a frontline integrator that adaptively inserts or removes particles from the frontline. While this approach maintains a uniform distribution of particles on the frontline, the synchronous advancement of the particle front can lead to needle triangles in shear flows. Here, we initially seeded 500 particles on the front line. Third, we apply the Hultquist [Hul92] algorithm, which advances frontline segments recursively such that the frontline does not suffer from shearing effects. Still, care must be taken when selecting the refinement criterion and thresholds. Here, we seeded 2 particles initially with a refinement threshold of 0.05 to obtain a ground truth quality surface that can serve as benchmark for us. Finally, the result of our method is shown, which matches the baseline methods with large numbers of particles.



**Figure 4:** Comparison with established path surface integrators. Left: advancing front line with fixed topology, middle (left): advancing front line with adaptive refinement, middle (right): Hultquist [Hul92] algorithm, right: our image-based method.



**Figure 5:** Comparison with existing OIT methods. For all the line-based methods we rendered 15000 lines. Approximative techniques (a), (b) and (c) map the fragment count to transparency, which saturates in a progressive renderer to an opaque surface. The binning in (d) shows discontinuities at bin boundaries, whereas our method (e) resembles the ground truth (f) closely.

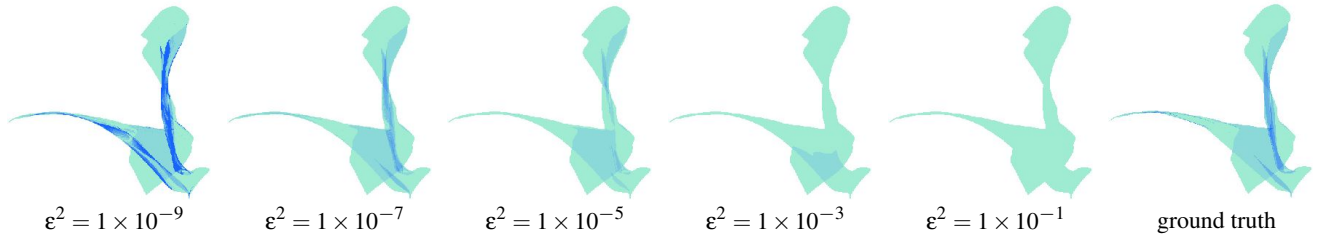
## 2.2. Comparison with Existing OIT Algorithms

A correct depiction of transport surfaces requires a blending of fragments in the correct order. Simple order-independent transparency (OIT) approximations are very fast alternatives in conventional non-progressive rendering, compared to accurate solutions such as fragment linked lists [YHGT10] or depth peeling [Eve01]. However, when rendering the scene progressively by inserting more and more geometry that samples the surfaces, they give visibly wrong results. Fig. 5 presents a comparison between our method and a number of OIT algorithms. Weighted sum [Mes07], weighted average [BM08], and its extension for a correct background blending [MB13] all saturate to fully opaque surfaces, since they relate the transparency to the fragment count, which is continuously growing. Further, we compare our method with a binning approach [BCL\*07] that discretizes the depth range into a fixed number of bins in which the fragments are averaged. The arbitrary classification into bins introduces noticeable discontinuities. Here, we used 200 bins. Increasing the number of bins further reduces the problem, but comes at a very high memory consumption. Our method, on the other hand, matches the ground truth solution

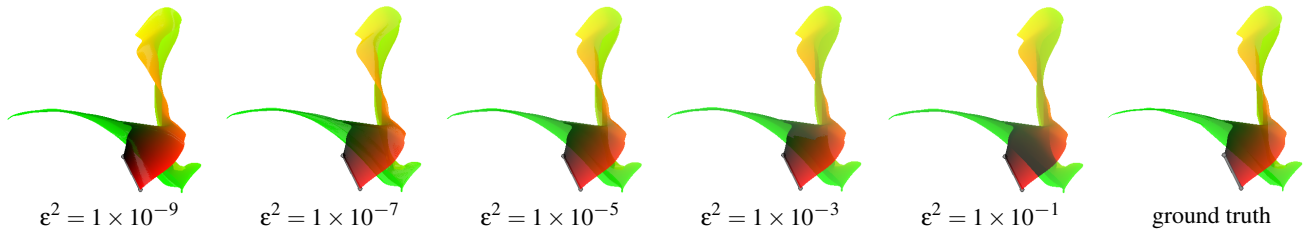
well, which was computed by tracing the surface geometry with the Hultquist algorithm and by rendering with fragment linked lists.

## 2.3. Parameter Study

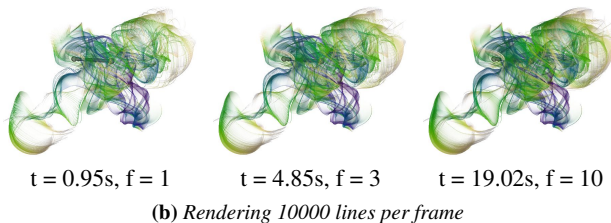
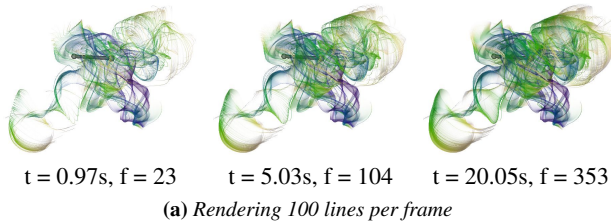
Despite an increasing number of rasterized integral curves, the key to a limited number of tree nodes is the merging of nearby line fragments into nodes that represent the surface layers that are progressively sampled. Whether a fragment is joining a certain tree node is determined by the view space distance between fragment and tree node, for which we introduced a threshold  $\epsilon$ . In Fig. 6, we show the compositing results for the construction with various thresholds. In practice, we found that any threshold between  $1 \times 10^{-7} < \epsilon^2 < 1 \times 10^{-5}$  gave generally good results, which we visualize in Fig. 7 by comparing the depth complexity to the ground truth. Choosing too large threshold will result in unintended merging of surface layers and using a value that is too small, will not merge the fragments into layers, causing an increasing depth complexity and thereby more memory consumption and an incorrect blending of too many transparent layers.



**Figure 6:** When inserting the fragments into the tree, we apply a depth-based heuristic to merge similar fragments into layers. Depending on the threshold, different numbers of layers are recognized at a pixel, as shown here in blue for varying  $\epsilon^2$ .



**Figure 7:** During tree construction, the fragments are added to tree nodes that represent surface layers if their distance to the node is below a user-defined threshold  $\epsilon$ . Here, the results are shown for varying thresholds. Although the parameter is scene dependent, in all of our experiments, any epsilon value that satisfies  $1 \times 10^{-7} < \epsilon^2 < 1 \times 10^{-5}$  gave reasonable results.



**Figure 8:** Intermediate results obtained after a small number of iterations. The overall surface shape becomes apparent quickly.

Compared to the ground truth methods, the clustering simplifies the surface representation depending on the threshold  $\epsilon$ . This could potentially lead to small color biases. We consider such small differences acceptable, given the benefit that our approach neither requires storage nor refinement or the integral surfaces.

## 2.4. Progressive Computation

As with any progressive rendering method, intermediate results can directly serve as preview visualization. Fig. 8 shows our result after an increasing number of iterations and we report the time it took to reach this frame. The overall shape of the surface is recognizable after 5.03 seconds when rendering 100 lines per frame, and can be even lowered to 0.95s if 10000 lines are rendered. On the other side, increasing the number of lines every iteration would decrease the frame rate. Over time, the gaps are closing, resulting in

a smooth and continuous transparent integral surface. Note that the importance sampling of the seeding curve, in order to sample locations with larger separation faster, is an orthogonal problem to the transparent rendering, which we would like to investigate more in the future. In our experiments, we already obtained pleasing results with the current sampling strategy. We refer to the video for animations of the progressive convergence series and user interactions.

## References

- [BCL\*07] BAVOIL L., CALLAHAN S. P., LEFOHN A., COMBA J. L., SILVA C. T.: Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), ACM, pp. 97–104. 3
- [BM08] BAVOIL L., MYERS K.: Order independent transparency with dual depth peeling. *NVIDIA OpenGL SDK* (2008), 1–12. 3
- [Eve01] EVERITT C.: Interactive order-independent transparency. *White paper, nVIDIA 2*, 6 (2001), 7. 3
- [HLSR09] HADWIGER M., LJUNG P., SALAMA C. R., ROPINSKI T.: Advanced illumination techniques for gpu-based volume raycasting. In *ACM SIGGRAPH 2009 Courses* (New York, NY, USA, 2009), SIGGRAPH '09, ACM, pp. 2:1–2:166. 2
- [Hul92] HULTQUIST J. P. M.: Constructing stream surfaces in steady 3D vector fields. In *Proceedings Visualization '92* (Oct 1992), pp. 171–178. 2, 3
- [MB13] MCGUIRE M., BAVOIL L.: Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques* (2013). 3
- [Mes07] MESHKIN H.: Sort-independent alpha blending. *GDC Talk* (2007). 3
- [MSE14] MACHADO G. M., SADLO F., ERTL T.: Image-based stream-surfaces. In *2014 27th SIBGRAP Conference on Graphics, Patterns and Images* (2014), IEEE, pp. 343–350. 1
- [YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum* 29, 4 (2010), 1297–1304. 3