Supplemental Material for *SparseLeap:* Efficient Empty Space Skipping for Large-Scale Volume Rendering

Markus Hadwiger, Ali K. Al-Awami, Johanna Beyer, Marco Agus, and Hanspeter Pfister

1 Pseudocode

In this supplementary material, we give detailed pseudocode for the main algorithms in SparseLeap. We list the algorithms in chronological order as they are being called, including the occupancy histogram tree management, ray segment list generation, and ray-casting.

1.1 Occupancy Histogram Tree

The occupancy histogram tree comprises a spatial subdivision of the volume, i.e., each tree node corresponds to a specific region of space. Our occupancy histogram tree is implemented as an octree, i.e., the spatial region corresponding to each node is an axis-aligned box.

1.1.1 Occupancy Histogram Propagation

Occupancy histograms are computed by propagating the occupancy class information of leaf nodes up the tree (see Alg. 1).

```
function PropagateOccupancyHistograms( nodeType node, nodeClass
        parentClass )

   node.classCount[ NON_EMPTY, EMPTY, UNKNOWN ] = 0;
   for all child nodes n
      PropagateOccupancyHistograms( node.children[ n ] );
      node.classCount[ NON_EMPTY, EMPTY, UNKNOWN ] +=
         node.children[ n ].classCount[ NON_EMPTY, EMPTY, UNKNOWN ];

   node.class = majority vote of NON_EMPTY, EMPTY, UNKNOWN;
```

Alg. 1. ***Propagation of occupancy histograms*** is done recursively (depth first, post-order), pulling the node classes from the leaves up the tree.

1.1.2 Occupancy Geometry Generation

For a given occupancy histogram tree, the corresponding view-independent occupancy geometry is computed by traversing the tree, determining for each node whether its bounding box should be emitted. The occupancy histogram tree is traversed in breadth first order, visiting child nodes in a fixed order (not in visibility order). The resulting occupancy geometry therefore always stores larger boxes (coarser tree levels) before smaller boxes (finer tree levels). See Alg. 2.

```
function TraversalOccupancyGeometryGeneration( nodeType root )

   nodeQueue queue;
   queue.push( root, parent=INVALID );

   while ( !queue.isEmpty() )

      nodeType node = queue.popNode();
      if ( node.class != node.parent.class )
         EmitOccupancyGeometryAndOccupancyClass( node,
               node.parent );

      for all child nodes n
         queue.push( node.children[ n ], node );
```

Alg. 2. ***Occupancy geometry generation.*** Breadth-first extraction of bounding box geometries whenever a node's class is different from its parent's class. We emit the geometry, the class, the parent's class.

1.1.3 Occupancy Geometry Visibility-Ordering

The occupancy geometry is view-independent. However, the raster zation of ray segment lists needs to be performed in visibility order with respect to the current view point. Therefore, whenever the view point changes, we compute a visibility-sorted index array that references the already generated occupancy geometry. We traverse the occupancy histogram tree in depth first order, visiting child nodes in front-to-back visibility order. In contrast to a standard tree traversal in visibility order, we do not only emit indexes for leaf nodes, but for all non-leaf nodes as well. See Alg. 3.

```
function TraversalIndexOrder( nodeType node )

   if ( node.isLeaf() )
      if ( node.class != node.parent.class )
         EmitOccupancyGeometryIndex( node, FRONT_FACE );
         EmitOccupancyGeometryIndex( node, BACK_FACE );
   else
      if ( node.class != node.parent.class )
         EmitOccupancyGeometryIndex( node, FRONT_FACE );

      int octant = ComputeViewpointOctant( node );
      for all child nodes n
         int vis_n = visibilityOrderOctantIndexPermutation[ octant ][ n ];
         TraversalIndexOrder( node.children[ vis_n ] );

      if ( node.class != node.parent.class )
         EmitOccupancyGeometryIndex( node, BACK_FACE );
```

Alg. 3. ***Occupancy geometry ordering.*** Depth-first extraction of bounding box indexes in visibility order. Each index is emitted twice: First in *pre-order* (before the child nodes), flagged for front face rasterization. Then again in *post-order* (after the child nodes), for back face rasterization. Leaf nodes also need to emit their index twice (front, and back).

1.2 Ray Segment List Generation

For each new view, the occupancy geometry is rasterized into per-pixel linked lists that store a sequence of successive segments (1D intervals) for each ray, the so-called ray segment list.

Given the occupancy geometry and the computed visibility order, we can now rasterize this geometry to create the ray segment lists. The occupancy geometry array is rasterized in a single rendering pass. For each fragment that is generated during rasterization, the fragment shader that is illustrated in Alg. 4 is invoked. The more detailed pseudo code for merging/deleting events on the fly is given by Alg. 5.

1.3 Ray-casting

Actual volume rendering is performed via ray-casting. Instead of directly iterating over samples, an additional outer loop iterates linearly

- *Markus Hadwiger, Ali K. Al-Awami, and Marco Agus are with King Abdullah University of Science and Technology (KAUST), Thuwal, 23955-6900, Saudi Arabia. E-mail: {markus.hadwiger, ali.awami, marco.agus}@kaust.edu.sa.*
- *Johanna Beyer and Hanspeter Pfister are with the John A. Paulson School of Engineering and Applied Sciences at Harvard University, Cambridge, MA, USA. E-mail: {jbeyer, pfister}@seas.harvard.edu.*

```
function RaySegmentListGenerationFragShader( fragmentType frag )

   raySegList segList = GetRaySegmentList( frag.x, frag.y );

   boundingBoxType boundingBox = frag.boundingBox;
   flagType flagFrontBack = boundingBox.flagFrontBack;

   if ( frag.isFrontFacing() )                          // is fragment from front or back face?
      if ( flagFrontBack == FRONT_FACE )                       // do we want the front face?
         AddRayEvent( frag.depth, EVT_ENTRY, boundingBox.class );
   else
      if ( flagFrontBack == BACK_FACE )                        // do we want the back face?
         AddRayEvent( frag.depth, EVT_EXIT, boundingBox.parent.class );
```

Alg. 4. ***Ray segment list generation*** is performed by rasterizing the occupancy geometry into a linked list of *ray events* for each pixel. The type of event being added in the fragment shader depends on whether the fragment comes from a front face or from a back face, and whether the bounding box is flagged for front face or for back face rasterization.

```
raySegList segList;
function MergeRayEvents( float depth, eventType type, nodeClass class )

   rayEventType eventPrev = segList.getPrevEvent();
   rayEventType eventPrevPrev = segList.getPrevPrevEvent();

   if ( isEqual( eventPrev.depth, depth ) )

      if ( eventPrev.eventType == type )                         // 2x ENTRY or 2x EXIT
         segList.overwritePrevEvent( depth, class, type );             // keep last only
      else if ( type == EVT_EXIT )                                     // ENTRY, EXIT
         segList.deletePrevEvent();                          // remove the coinciding pair
      else if ( eventPrevPrev.class == class )                         // EXIT, ENTRY
         segList.deletePrevEvent();          // merge the two same-occupancy segments

   if ( isClose( eventPrev.depth, depth ) && eventPrev.class == EMPTY )

      if ( class == NON_EMPTY )
         if ( eventPrev.eventType == type )               // start non-empty earlier
            segList.overwritePrevEvent( eventPrev.depth, type, class );
         else
            segList.deletePrevEvent();                        // remove small empty gap

      else if ( class == UNKNOWN )                        // start previous segment later
         segList.overwritePrevEvent( depth, type, class );

function AddRayEvent( float depth, eventType type, nodeClass class )

    MergeRayEvents( depth, type, class );

   if ( no merging or deletion done )
      segList.appendEvent( depth, type, class );                            // default
```

Alg. 5. ***Ray event creation and merging/deletion.*** Before a ray event is added, it is checked for possible on-the-fly merging and deletion of events to reduce the overall depth complexity.

```
function RayTraversalForImagePixel( int x, int y )

   raySegList segList = GetRaySegmentList( x, y );

   rayEventType eventSegBegin = GetNextRayEvent( segList );
   while ( !segList.empty() )

      rayEventType eventSegEnd = GetNextRayEvent( segList );
      if ( eventSegBegin.class != EMPTY )

         SampleRaySegment( eventSegBegin, eventSegEnd );

         if ( rayEventSegBegin.class == UNKNOWN )
            ReportCullCacheMiss( eventSegBegin );

      eventSegBegin = eventSegEnd;
```

Alg. 6. ***Ray-casting.*** The outer loop iterates from ray segment to ray segment (each segment is bounded by two ray events). Segments that are *non-empty* or *unknown* are densely sampled. The latter additionally generate *occupancy misses*. *Empty* segments are simply skipped.

from ray segment to ray segment. Empty segments are skipped over, and non-empty, as well as unknown, segments are sampled as in standard ray-casting. Unknown segments will additionally generate occupancy misses in order to enable output-sensitive culling. See Alg. 6.