7. Supplementary Material

```
NodeState
AccessNode(uint64 index, Payload* payload)
{
  NodeState state;

  Node* node = toPointer(ROOT + index);
  state.node = node;

  /* leaf node, setting values */
  if (isLeaf(node)) {
    LeafNode* leaf = (LeafNode*) node;
    state.isLeaf = true;
    state.data.position = leaf->position;
    state.data.radius   = leaf->radius;
    state.data.weight   = leaf->weight;
  }

  /* node is an internal node */
  else {
    InnerNode* inner = (InnerNode*) node;
    state.isLeaf = computeLoD(node, payload);
    /* whether children are all available */
    if (not state.isLeaf) {
      uint64 d0 = (uint64)inner->children[0];
      uint64 d1 = (uint64)inner->children[1];
      d0 = lookupBlockIndex(
        d0 / BLOCK_SIZE_IN_BYTES
      );
      d1 = lookupBlockIndex(
        d1 / BLOCK_SIZE_IN_BYTES
      );
      if (d0 == INVALID_INDEX ||
          d1 == INVALID_INDEX)
        state.isLeaf = true;
    }
    /* create a fake leaf node */
    if (state.isLeaf) {
      box3f bound = boxExtend(
          inner->bounds[0],
          inner->bounds[1]
      );
      state.data.position = center(bound);
      state.data.radius = 0.5 * diagonal(bound);
      state.data.weight =
        midpoint(node->valueRange);
    }
    /* continue traverse children */
    else {
      state.child0 = d0;
      state.child1 = d1;
    }
  }

  return state;
}
```

Listing 3: *In Section 4.2, we mentioned that for each brick format, an additional wrapper function is required. Here we provide the pseudo-code demonstrating the* `accessNode` *function for RBF volume data.*

```
NodeState
accessNode(
  TraversalStack* stackPtr,
  Payload* payload)
{
  NodeState state;
  Node* node = addressNode(stackPtr->index);
  if (isLeaf(node)) {
    accessLeaf(node, state);
  }
  else {
    state.isLeaf = computeLoD(stackPtr, payload);
    /* node absolute index */
    uint64 nid = childrenOffset(node);
    /* block index */
    uint64 bid = nid / BLOCK_SIZE;
    /* address translation */
    uint64 new_bid = lookupBlockIndex(bid);
    uint64 new_nid =
      new_bid * BLOCK_SIZE + nid % BLOCK_SIZE;

    if (not state.isLeaf) {
      /* block is available in RAM */
      if (new_block_index != INVALID_INDEX) {
        updateCache(children_block_index);
        Brick* brick =
          addressNode(new_index)->currentBrick();
        state.range  = brick->range;
        state.childrenOffset = new_index;
        state.childrenMask = childrenMask(node);
      }
      /* block not loaded, send an I/O request */
      else {
        state.isLeaf = true;
        if (lockCacheLine(children_block_index))
          streamBlockAsync(children_block_index);
      }
    }
    /* access inner brick as a leaf voxel */
    else {
      state.data.value = midpoint(node->range);
    }
  }
  return state;
}
```

Listing 4: *In Section 4.1, we mentioned that for each brick format, an additional wrapper function is required. Here we provide the pseudo-code demonstrating the* `accessNode` *function for TAMR particle volume data.*

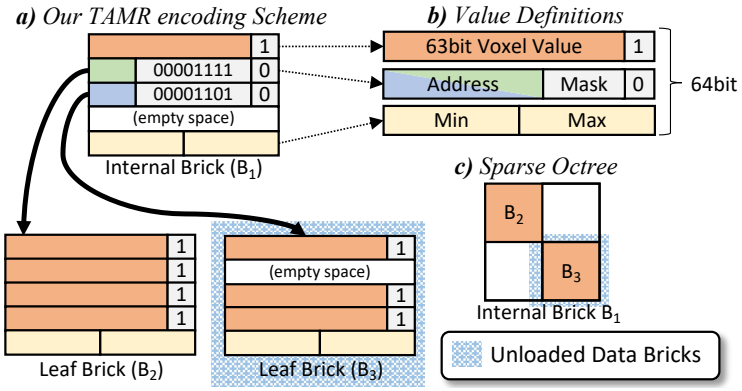


Figure 11: *Same as Figure 6.*

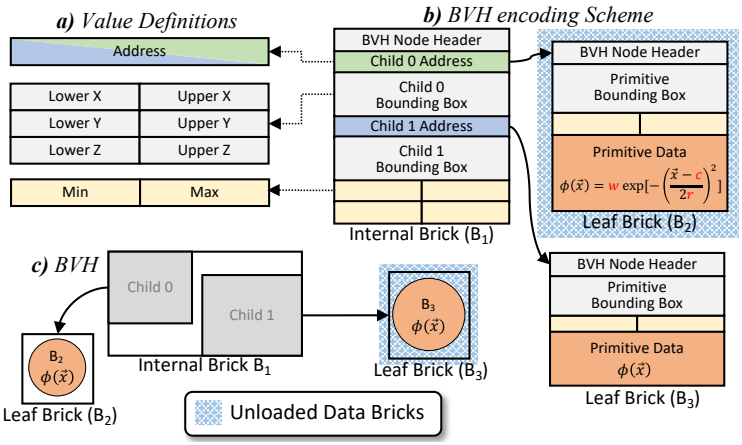


Figure 12: *Same as Figure 7.*

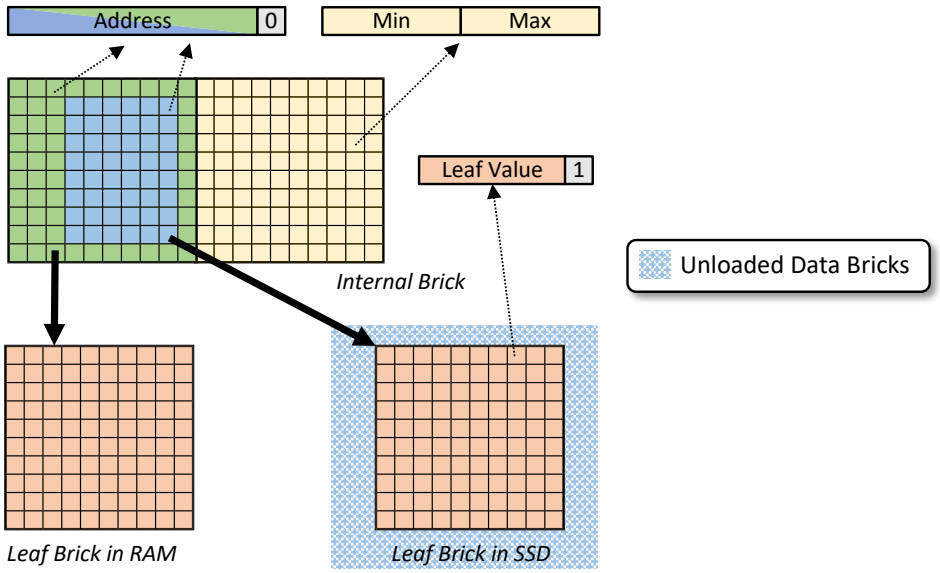


Figure 13: *An example brick format design for the regular grid volume based on descrptions in Section 4.3.*

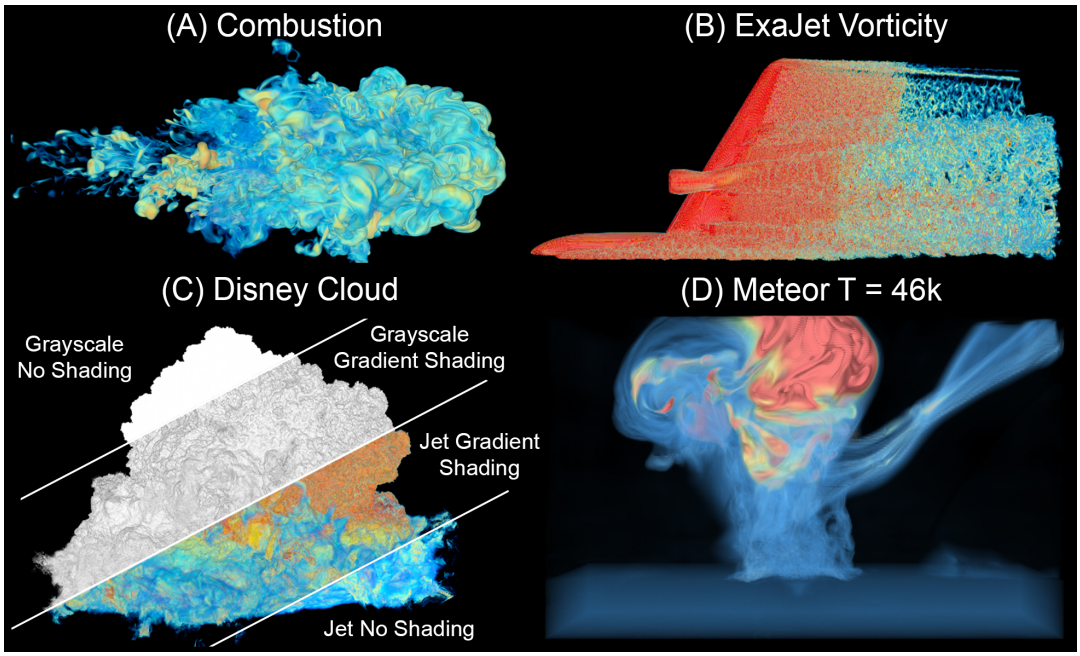


Figure 14: *Datasets used in our benchmark. A) A simulation of multi-injection mixing and combustion at compression ignition engines [TBB*17]. B) A simulation of air flow (the vorticity field) around a jet plane [CH14]. C) A cloud dataset rendered with and without shading effects [Stu]. D) A simulation of an asteroid impact in deep ocean water [PG17]. We did not enable gradient shading in our benchmarks because we used an unmodified OpenVKL example renderer as the immediate renderer, which does not implement gradient shading.*

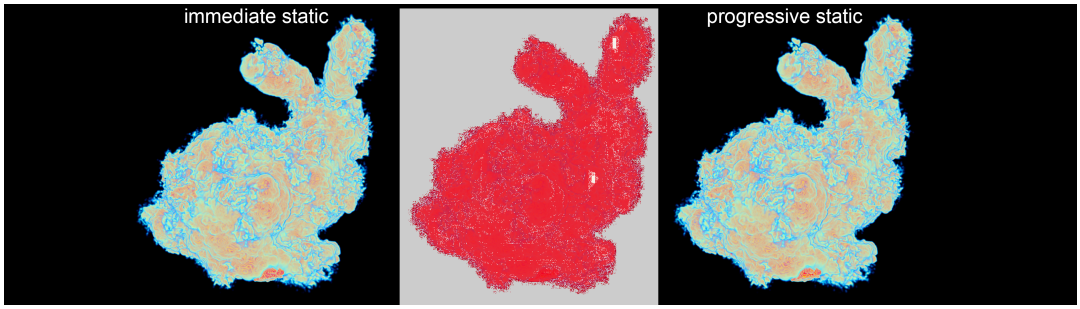


Figure 15: *Image difference between our progressive rendering method and a normal ray marching method, rendering an RBF particle volume. Left: image rendered using the progressive static mode. Right: image rendered using the immediate static mode. Middle: the image difference. The data being rendered is created using the "bunny cloud" data provided by the OpenVDB project. In particular, we mapped every voxel to a particle and using the voxel value as the particle weight.*

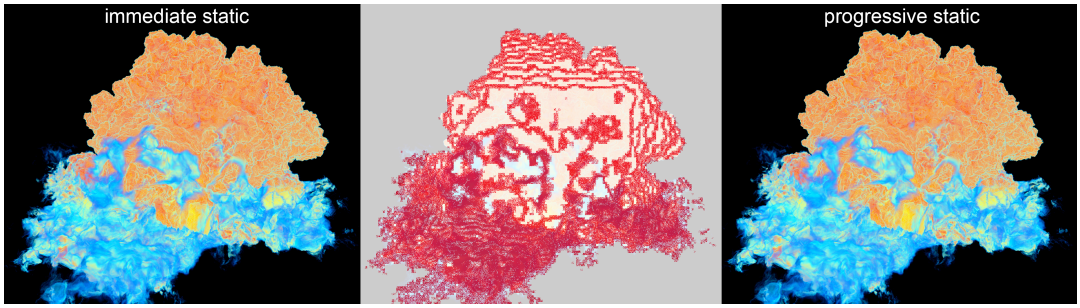


Figure 16: *Progressive rendering can produce a slightly different image. However, this difference is expected because progressive rendering will change sample positions along the ray, and this difference is not visible visually. Left: image rendered using the progressive static mode. Right: image rendered using the immediate static mode. Middle: the image difference.*

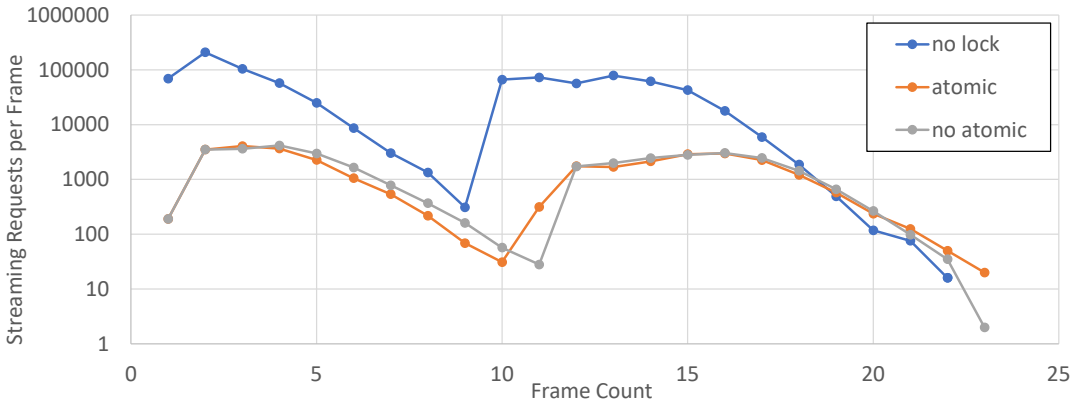


Figure 17: *As mentioned in the third paragraph of Section 3.2.1, we implemented two ways to concurrently update hash flags. In our preliminary experiments, we compared the effectiveness of different methods using the RBF particle volume implementation. Our non-atomic version can significantly reduce the number of streaming requests made per frame. The "bunny cloud" particle data is used for this experiment. We used two camera positions for each run, which caused the dip around frame 8.*

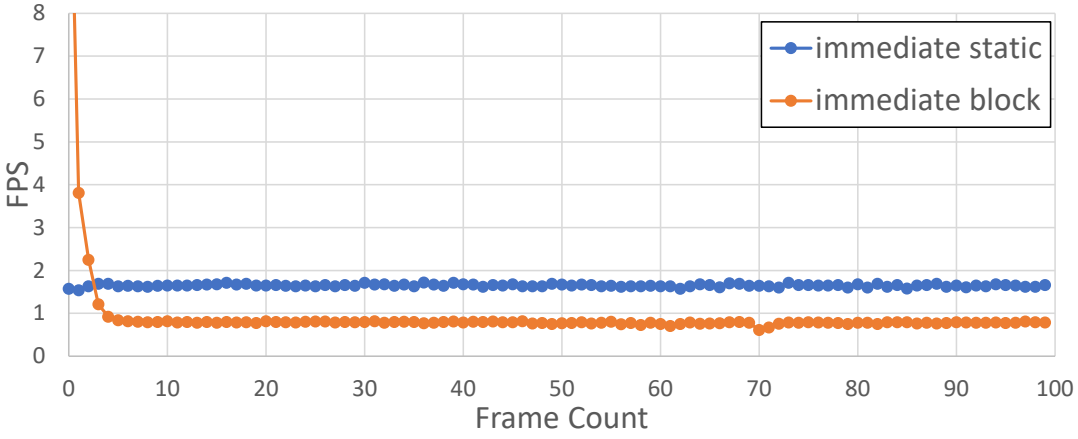
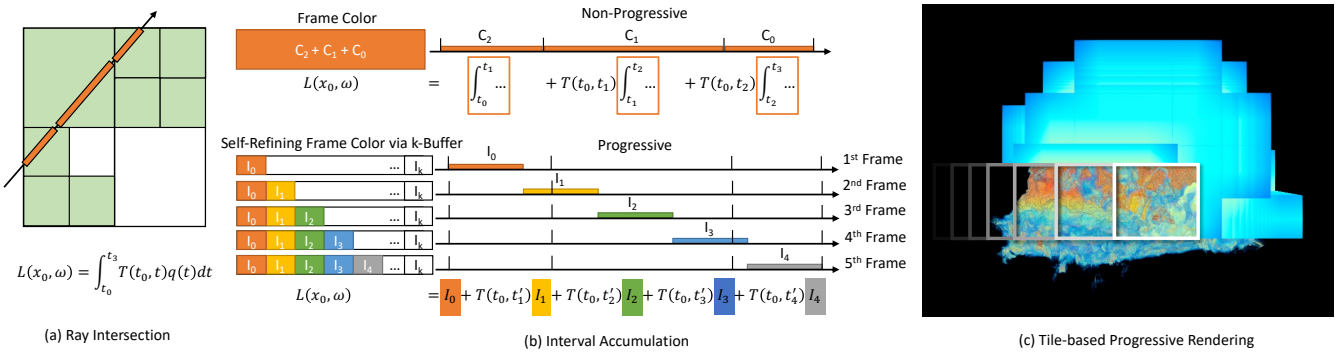


Figure 18: *RBF particle volume rendering performance. Our streaming implementation (i.e., immediate block) slower than the baseline version (i.e., immediate static), but our implementation is still interactive. We uses the no-atomic method for page table and LRU cache updates. The "bunny cloud" particle data is used for this experiment.*



(a) Ray Intersection (b) Interval Accumulation (c) Tile-based Progressive Rendering

Figure 19: *Our progressive rendering system divides a volume integration (a) into K intervals. (b) At each frame, we calculate only one of the intervals. The calculated value is stored in a K-buffer. Then we calculate the final frame color by compositing colors within the K-buffer for each pixel. (c) Our progressive rendering system also supports a tile-based rendering mode. This mode further reduces memory consumption.*

Table 4: *Benchmark Results. We tested each data using 8 modes: progressive static (PS), progressive treelet (PT), progressive block (PB), immediate static (IS), immediate treelet (IT), immediate block (IB), tile treelet (TT) and tile block (TB). Because PT, PB, TT and TB provide better scalability, we tested them using upsampled data. (1) Peak I/O for static modes measures the the sequential read performance. (2) For all progressive and tile modes, the Max FPS essentially measures the color composition performance. (3) For the IS mode, the frame time for the 1st frame is used in the comparison because the converged frame is given by the 1st frame. (4) For all static modes, the first number measures the data pre-loading time, and the second one measures the rendering time; Both should be included when comparing with other streaming methods, therefore all are listed.*

	Mode	File Size (GB)	Memory Footprint (GB)	Peak I/O (MB/s)	FPS Max$^{(2)}$	FPS Min	FPS Interactive	$T_{converge}$ (s)
Disney Cloud	PS	17.7	20.1	862.7$^{(1)}$	67.8	0.67	4.25	29.6$^{(4)}$ + 15.9
	PT	22.7	5.62	516.9	59.4	0.55	3.40	22.0
	PB	17.7	5.49	136.4	59.8	0.29	1.60	42.9
	IS	17.7	20.1	864.6$^{(1)}$	0.09	0.09	0.09	31.7$^{(4)}$ + 11.1$^{(3)}$
	IT	22.7	6.66	790.1	0.07	0.07	0.07	69.9
	IB	17.7	6.01	243.8	0.04	0.04	0.04	207.9
	PT (×8)	178	6.69	592.9	60.6	0.18	2.52	66.3
	PB (×8)	139	6.32	282.1	59.9	0.09	1.29	137.8
	TT (×8)	178	5.61	683.8	58.5	0.77	N/A	99.6
	TB (×8)	139	5.48	341.9	59.5	0.47	N/A	182.3
S3D-AMR	PS	14.7	16.9	751.6$^{(1)}$	30.9	0.14	1.31	29.1$^{(4)}$ + 61.9
	PT	18.4	6.27	468.6	24.9	0.11	1.01	95.7
	PB	14.7	6.00	270.1	25.6	0.05	0.48	240.2
	IS	14.7	16.9	819.8$^{(1)}$	0.02	0.02	0.02	29.4$^{(4)}$ + 50.0$^{(3)}$
	IT	18.4	failed to converge					
	IB	14.7	failed to converge					
	PT (×8)	148	failed to converge					
	PB (×8)	118	failed to converge					
	TT (×8)	148	6.18	905.9	25.7	0.32	N/A	314.9
	TB (×8)	118	5.79	749.0	28.7	0.17	N/A	728.8
Meteor-20k	PS	1.85	2.99	728.6$^{(1)}$	53.8	3.49	5.15	2.9$^{(4)}$ + 3.9
	PT	2.43	5.31	69.8	53.1	2.29	3.58	6.4
	PB	1.85	5.31	75.1	52.8	1.45	1.74	10.6
	IS	1.85	2.99	778.1$^{(1)}$	0.35	0.35	0.35	2.7$^{(4)}$ + 2.9$^{(3)}$
	IT	2.43	5.51	256.2	0.24	0.24	0.24	34.6
	IB	1.85	5.41	102.7	0.15	0.15	0.15	66.4
	PT (×64)	147	5.97	504.0	52.7	0.36	2.05	36.9
	PB (×64)	118	5.81	521.7	52.3	0.23	1.08	69.4
	TT (×64)	147	5.44	546.8	55.2	1.51	N/A	72.1
	TB (×64)	118	5.39	464.5	52.4	0.89	N/A	112.7
Meteor-46k	PS	3.31	5.65	825.9$^{(1)}$	48.4	2.59	3.36	4.8$^{(4)}$ + 4.6
	PT	4.25	5.31	108.6	51.8	2.10	2.48	7.5
	PB	3.31	5.31	68.6	56.2	1.45	1.20	12.4
	IS	3.31	4.56	831.2$^{(1)}$	0.25	0.25	0.25	4.9$^{(4)}$ + 4.0$^{(3)}$
	IT	4.25	5.76	336.1	0.19	0.19	0.19	16.2
	IB	3.31	5.49	142.8	0.12	0.12	0.12	45.0
	PT (×64)	267	6.19	758.1	50.5	0.46	1.63	40.7
	PB (×64)	212	6.19	525.1	50.8	0.32	0.83	72.1
	TT (×64)	267	5.84	783.8	48.0	1.80	N/A	79.7
	TB (×64)	212	5.58	704.7	49.2	1.17	N/A	120.1
Exajet	PS	7.70	9.30	847.1$^{(1)}$	66.7	0.92	6.16	11.2$^{(4)}$ + 7.8
	PT	9.06	5.99	553.0	60.6	0.62	4.75	12.2
	PB	7.70	5.73	474.9	59.7	0.41	2.47	21.7
	IS	7.70	9.29	864.3$^{(1)}$	0.20	0.20	0.20	11.1$^{(4)}$ + 5.0$^{(3)}$
	IT	9.06	6.64	570.8	0.16	0.16	0.16	43.3
	IB	7.70	6.32	428.2	0.10	0.10	0.10	100.6
	PT (×8)	61.5	6.69	814.9	60.2	0.44	3.53	111.8
	PB (×8)	72.0	6.67	770.4	65.0	0.33	1.82	173.1
	TT (×8)	61.5	6.09	904.0	59.5	0.80	N/A	61.7
	TB (×8)	72.0	5.91	784.9	57.9	0.53	N/A	94.3