Eurographics Symposium on Parallel Graphics and Visualization (2026)
S. Rizzi, C. Gueunet, S. Dutta (Editors)

Supplementary Material

A. Kreskowski and G. Rendle and B. Froehlich

Virtual Reality and Visualization Research Group, Bauhaus-Universität Weimar, Germany

A: Notes on In-Kernel Bounding Geometry Computation

In-Kernel Bounding Volume Computation. It is beneficial to allow the rendering component to cull meshlets when they are not contained in the viewing frustum. This can be enabled by calculating a bounding volume for each meshlet, which can be compared with the view frustum. Our implementation includes different approaches to creating bounding volumes, which are compared in Figure A. In the first approach, we use the implicit axis-aligned bounding box (AABB), which can be derived from the index of each meshlet's parent block. This bounding volume is not as tight-fitting as other approaches, but avoids additional computational overhead during the extraction stage. In the second approach, we explicitly compute a tight-fitting AABB for the meshlet using *subgroupMinMax* operations. This does not incur any significant overhead because the vertices required for the calculation are already available. The third approach computes bounding spheres containing the extracted vertices. We utilize the most straightforward method for bounding sphere creation [Lar08], which performs two in-kernel passes through the vertices: one to establish the sphere center and one to find the minimum radius that contains all vertices, given the established center. This approach is therefore more computationally demanding than AABB methods (see gray bars in Figure A). Although bounding spheres may offer a benefit in certain mesh rendering scenarios, we notice that when meshlets are extracted from small, compact blocks, there is little added culling benefit from bounding spheres. Accordingly, we recommend employing one of the AABB-based representations.

In-Kernel Normal Cone Computation. In addition to cluster frustum culling, it is common practice to perform cluster backface culling based on normal cones [SAE93], which describe the spread of normal vectors within a cluster of triangles with a cone direction vector and an opening angle. This data allows clusters to be backface-culled according to the viewing position [Kap25]. To achieve conservative and effective cone culling, tight normal cones need to be derived from the per-face normals of the triangles to be rendered. Approximating the per-face normals by volume gradients does in general not guarantee the creation of conservative normal cones. Deriving exact per-face normal vectors for the triangles that comprise our meshlets is also not directly possible within an unmodified *PMB* kernel, because (by design) no thread has access to all the vertices of the triangles extracted from its occupied cell(s), since extraction overhead is distributed across all threads within the workgroup. To experimentally address this, we build a hybrid between *PMB* and classical MC approaches, allowing each thread of the *PMB* kernel to temporarily extract the (up to) 12 vertices created by its cell in a classic MC manner, which allows the triangles to be built and face normals to be calculated.

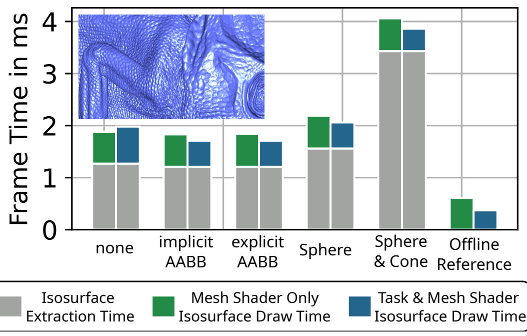


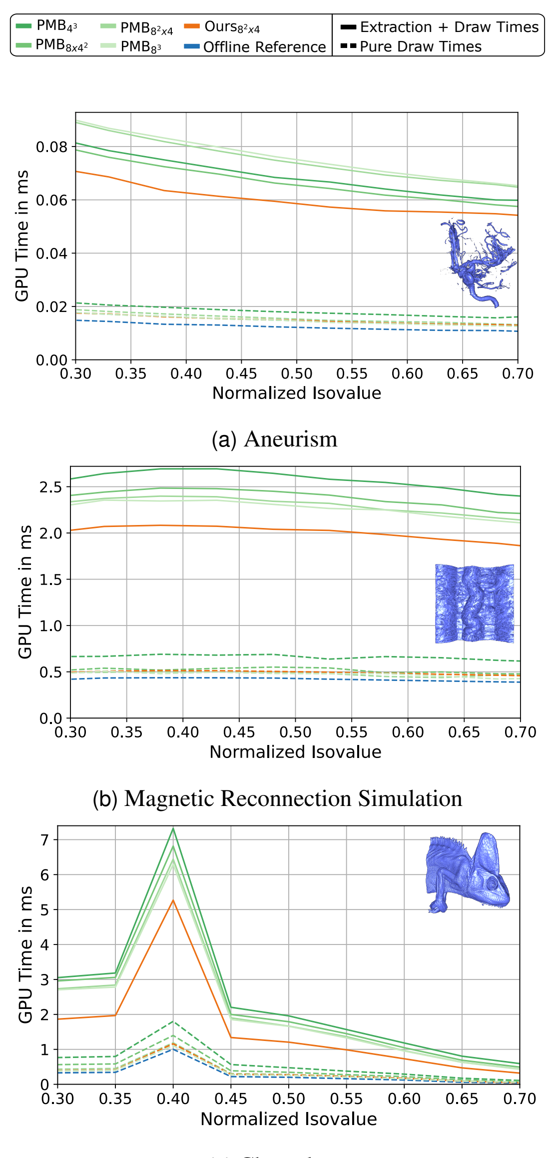
Figure A: Comparison of complete frame times for the shown camera perspective for which the isosurface (consisting of meshlets) is rendered either using only mesh shaders (no cluster culling is performed, in green) or using task and mesh shaders (with task shaders performing cluster culling, in blue). The extraction time strongly varies depending on the desired bounding volume (and normal cone) computation. While the offline reference approach benefits from cone-based culling, our real-time extraction method does not, since the cost of creating normal cones outweighs the culling savings when the extracted isosurface is not drawn often enough.

This use of registers and computation, together with the normal cone computation itself [AMHH*18], outweighs the benefits of cone-based backface culling at draw time (see Figure A). An important factor here is the lack of explicit control that our approach has over the shape of the created meshlets, meaning that the normal spread of a meshlet cannot be limited to a specific value. Accordingly, we do not recommend the use of normal cones for our approach. However, we note that more complex pipelines that employ in-pipeline occlusion culling [KRF22] are likely to be able to discard many back-facing clusters during occlusion tests, meaning that future investigations into efficient normal cone extraction methods may have value.

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

B: Additional Performance Measurement Figures

Complementary Dynamic Isosurface Exploration Graphs The performance graphs in Figure B complement our descriptions and evaluation of the dynamic isosurface exploration scenario in Section 4.3 of the main document.



(a) Aneurism

(b) Magnetic Reconnection Simulation

(c) Chameleon

Figure B: Combined extraction and draw times encountered in a dynamic isosurface exploration scenario for the proposed meshlet extraction and rendering approach (*Ours*), compared to the extraction and draw times of the four most competitive non-meshlet extraction modes of *PMB* and the draw times of the offline-optimized meshlet reference, which serves as a best-case reference where no time is spent on extraction during runtime. In addition to the combined extraction and draw times (solid lines), we also plot the corresponding pure draw times (dashes) of the online extraction approaches.

C: Pseudocode for our Region Analysis Stage using Task Shaders and Extraction Stage using Mesh Shaders

Listing 1 Region Analysis Task Shader.
(cmp. Figure 4 in the main document)

```
////////////////////////////////////////////////////////////
//SUMMARY:                                                //
// Find occupied cells and count vertices & primitives of //
// isosurface geometry created by subblocks with          //
// subblock sizes (SBS) -> (stats_SBSxyz).               //
// Subsequently create subblock regions which create      //
// meshlets honoring desired geometry count limits.       //
////////////////////////////////////////////////////////////
/*Executed by exactly 32 threads, each analyzing distinct 2x2x2
  regions of our 256 cell subblock to encode every occupied
  cell with an 8 bit index. Meshlets created from 2x2x2 regions
  are guaranteed to honor geometry count limits, so we fall
  back to those regions if others produce too much geometry */
for Thread T_i=0..31 in subgroup:

  //helper variable to count number of subblocks created
  subblock_count_per_thread = 0;
  i =get_local_thread_id();

  //////////////////////////////////////////////////////////
  // Each thread calculates statistics for its subregion //
  // corresponding to a subblock with SBS 2x2x2          //
  //////////////////////////////////////////////////////////
  stats_SBS222 =analyze_SBS222(i);

  //////////////////////////////////////////////////////////
  // Top-down hierarchical region analysis.              //
  // Using a subgroup operation, threads sum up          //
  // vertex and index counts for entire 8x8x4 block      //
  //////////////////////////////////////////////////////////
   stats_SBS884 =subgroupClusteredAdd(stats_SBS222, 32);

  if(fulfills_meshlet_constraints(stats_SBS884) ) then:
  submit_subblock_info_for_extraction(stats_SBS884);
    //only count the 8x8x4 block on one thread
    subblock_count_per_thread = 0 == i ? 1 : 0;
  else:
      //if meshlet from 8x8x4 block too large, analyze further
      //calculate geometry and cell counts for 4x4x4 subblocks
      stats_SBS444 =subgroupClusteredAdd(stats_SBS222, 8);
      if(is_region_occupied(stats_SBS444) ) then:
        if(fulfills_meshlet_constraints(stats_SBS444) ) then:
          submit_subblock_info_for_extraction(stats_SBS444);
           // only count the 4x4x4 block on one out of 8 threads
           subblock_count_per_thread = (0 == (i mod 8)) ? 1 : 0;
        else:
          if(is_region_occupied(stats_SBS222) ) then:
           submit_subblock_info_for_extraction(stats_SBS222);
            // count the 2x2x2 block on each thread
            subblock_count_per_thread = 1

  //obtain total number of active subblocks
  num_active_subblocks =subgroupAdd(subblock_count_per_thread);

only Thread T_31:
  spawn_num_extraction_workgroups(num_active_subblocks);
```

Listing 2 Meshlet Extraction Mesh Shader.
(cmp. Figure 4 in the main document)

```
////////////////////////////////////////////////////////////
// SUMMARY:                                               //
// Retrieve region size and active cells per thread.      //
// Then execute Parallel Marching Blocks extraction       //
// and meshlet boundary computation.                      //
////////////////////////////////////////////////////////////
/*For this stage, we do not impose any limit on the extraction
  group size on top of current hardware and extension
  limits. To allow sufficiently large extraction regions,
  smaller thread counts (e.g. 32 in the GL_NV_mesh_shader
  extension) can be compensated for by allowing threads
  to work on several cells.                          */

for Thread T_i=0..n-1 in subgroup:

  /*the region analysis shader provides global write offsets
  for vertices, triangles and meshlet descriptors */
  global_offsets =get_global_offsets(local_workgroup_id);
  i =get_local_thread_id();
  //retrieve occupied cell index
  cell_id =read_occupied_cell_id_from_task_mesh_interface(i);

  //reserve space for up to 3 unique positions and normals
  num_verts = 0; v_pos[3] = {}; v_nor[3] = {};
  num_verts, v_pos, MC_case =extract_PMB_vertices(cell_id);
  v_nor =sample_normals_at_vertices(num_verts, v_pos);

  /*this synchronization refers to the in-kernel stream
    compaction explained in detail in the original paper
    "Parallel Marching Blocks: A Practical Isosurfacing
    Algorithm for Large Data on Many-Core Architectures"
    by Liu, Clapworthy, Dong and Wu (2016)
    For details refer to their supplementary pseudocode
    and our own implementation in the provided framework*/
  local_offset
    =synchronize_vertex_offsets_across_workgroup(num_verts);

  //this computation happens across the entire workgroup and
  //is carried out using subgroup communication as well as
  //additional shared memory for synchronizing boundary
  //information across different workgroups
  bounds
    =compute_meshlet_bounds_coop(num_verts, v_pos, v_nor);

  write_unique_positions_to_storage_buffer(num_verts, v_pos);
  write_unique_normals_to_storage_buffer(num_verts, v_nor);
  resolve_and_write_triangle_indices(num_verts, MC_case);

only Thread T_n-1:
   stats =compute_counts_and_offsets(local_offset, MC_case);
  write_meshlet_descriptor(stats, bounds);
```

References

[AMHH*18] AKENINE-MÖLLER T., HAINES E., HOFFMAN N., PESCE A., IWANICKI M., HILLAIRE S.: *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, 2018. 1

[Kap25] KAPOULKINE A.: Niagara task and mesh shaders, 2025. URL: https://github.com/zeux/niagara. 1

[KRF22] KRESKOWSKI A., RENDLE G., FROEHLICH B.: Efficient direct isosurface rasterization of scalar volumes. *Computer Graphics Forum 41*, 7 (2022), 215–226. 1

[Lar08] LARSSON T.: Fast and tight fitting bounding spheres. In *Proceedings of the Annual SIGRAD Conference* (2008), vol. 34 of *Linköping Electronic Conference Proceedings*, pp. 27–30. 1

[SAE93] SHIRMUN L. A., ABI-EZZI S. S.: The cone of normals technique for fast processing of curved patches. *Computer Graphics Forum 12*, 3 (1993), 261–272. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1230261. 1