Eurographics Conference on Visualization (EuroVis) 2023
D. Archambault, R. Bujack, and T. Schreck
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 3

Memory-Efficient GPU Volume Path Tracing of AMR Data Using the Dual Mesh—Supplementary Document

Stefan Zellmann[1], Qi Wu[2], Kwan-Liu Ma[2], and Ingo Wald[3]
[1]University of Cologne [2]University of California - Davis [3]NVIDIA

1. Additional Color Images

This document provides supplementary material to our paper [ZWMW23]. We present several high-resolution color images that would not fit into the paper due to space limitations. We use presentations similar to Fig. 1 from the paper to convey the spatial structure of the data. Fig. 1 shows the NASA Exajet data set from the front; Fig. 2 shows Exajet from a rear view. The structures near the wing and fuselage are finely tessellated with tiny dual cells. Fig. 3 shows the LANL meteor impact data set; note the finely tessellated ocean surface. We use a clip plane to reveal and further convey the structure in Fig. 4. Fig. 5 shows the NASA Landing Gear, again with a visualization inspired by Fig. 1 from the paper. The data set suffers from the “teapot in a stadium” problem, as can be seen in Fig. 6.

2. Additional Evaluation

In the paper we mention that we use gridlets of size 8^3 cells that store 9^3 voxels. Fig. 7 presents an evaluation of gridlet size vs. memory consumption, both manually computed and measured with `nvidia-smi`. Smaller gridlets have more ghost cells as scalars at gridlet boundaries are replicated; bigger gridlets contain more empty cells. Bigger gridlets also result in a more shallow BVH; we still observed negative returns in both memory as well as rendering performance when scaling beyond gridlet sizes of 8^3.

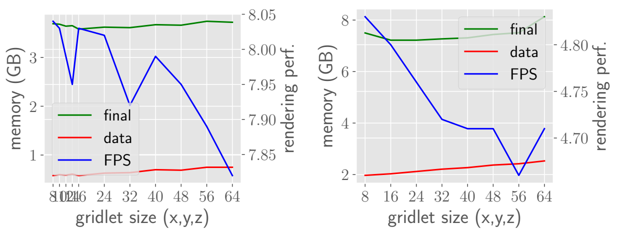

Figure 7: *Gridlet size vs. memory and rendering performance. We use a gridlet size of 8^3 cells (9^3 scalars); gridlet sizes beyond that show negative returns both in memory consumption (“data”, manually computed and “final”, measured with `nvidia-smi`) and in rendering performance (in frames/second, FPS).*

Note that the differences in frames/second are still subtle: always approximately in the range of 7-8 FPS for the TAC Cloud, and 4.5-5 FPS for LANL Impact.

We use a bilinear face test for boundary stitching elements. For that we ported the implementation from OpenVKL [Int] for pyramids, wedges, and hexes. We use ten Newton iterations to refine the result. In Fig. 8 we present a visual comparison, including difference images and contrast-enhanced images to highlight the subtle differences.

3. Code Listings

We provide code snippets in C++ pseudo code that represent gridlet construction (Algorithm 1) Figs. 9 and 10. This implementation assumes that a hashmap is used to represent the macrocells of the virtual grid, i.e., macrocells that are (and stay) empty are never created in memory. Hashmaps are not necessarily the most sensible choice to use when focusing on performance; however, they allow for a very simple implementation of Algorithm 1. In Fig. 11 we provide CUDA pseudo code to sample the gridlet element type in the shader or ray tracing program.

References

[ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020), 15:1–15:23. 5

[Int] INTEL CORPORATION: High Performance Volume Kernels. Available at `https://www.openvkl.org/`, Accessed: 11 November 2022. 1

[ZWMW23] ZELLMANN S., WU Q., MA K.-L., WALD I.: Memory-Efficient GPU Volume Path Tracing of AMR Data Using the Dual Mesh. *Computer Graphics Forum (Proc. EuroVis)* (2023). 1

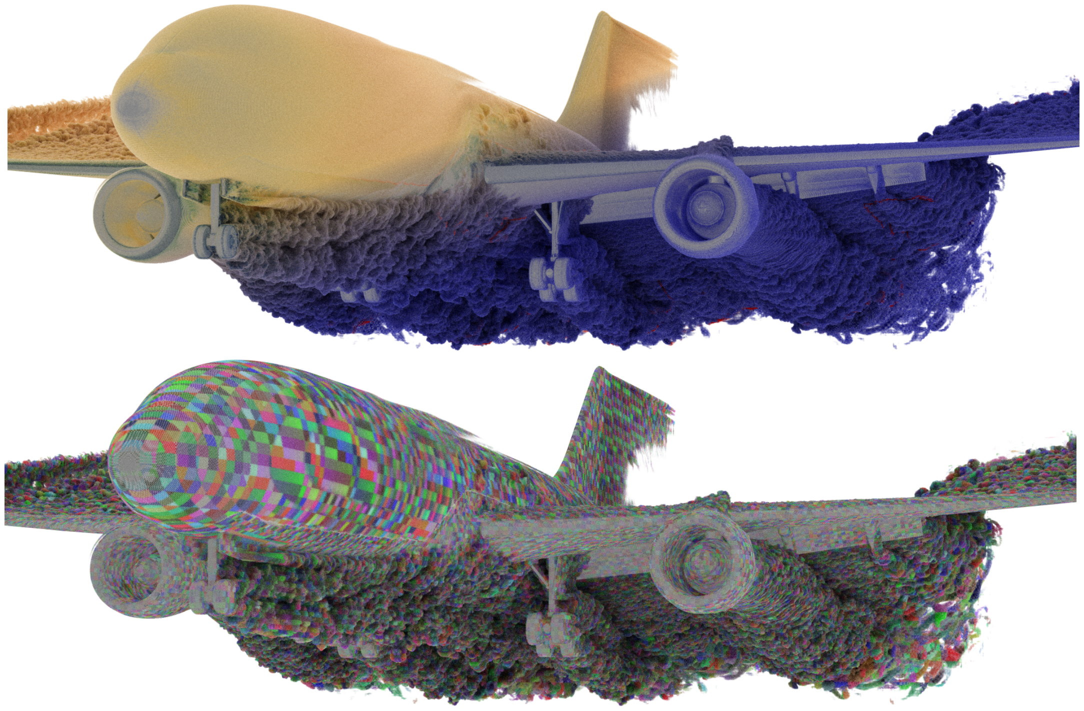

Figure 1: *Exajet front view. Top: shaded (top-right) and voxels+stitching elements (bottom-left). Bottom: gridlets colored by primitive ID.*

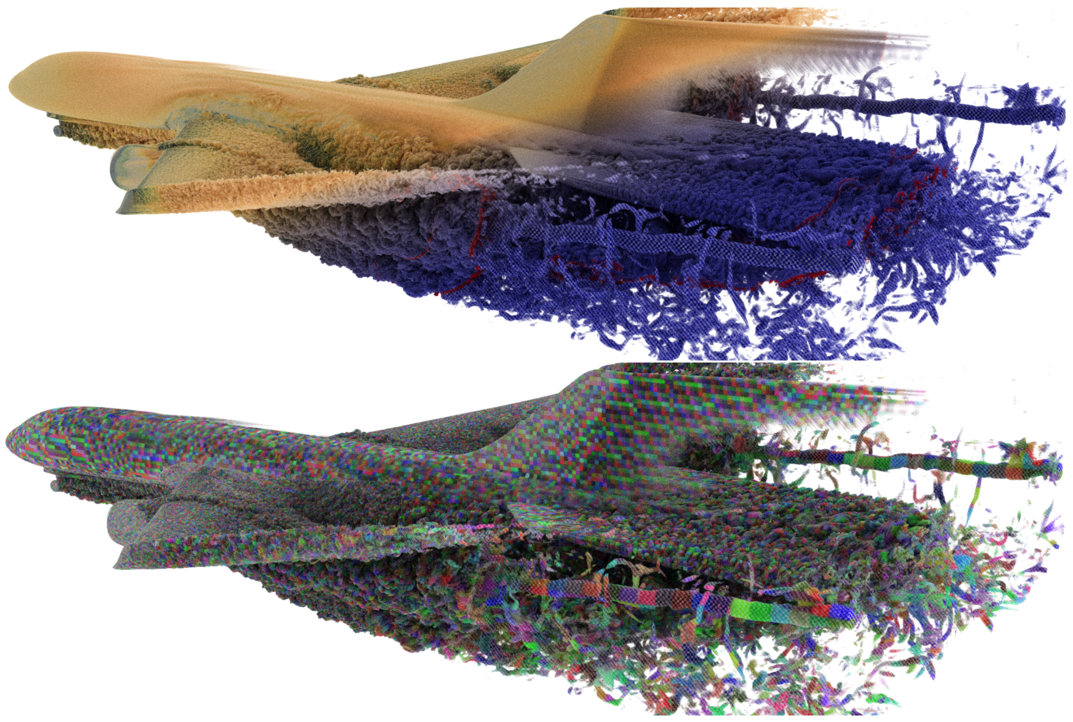

Figure 2: *Exajet back view. Top: shaded (top-right) and voxels+stitching elements (bottom-left). Bottom: gridlets colored by primitive ID.*

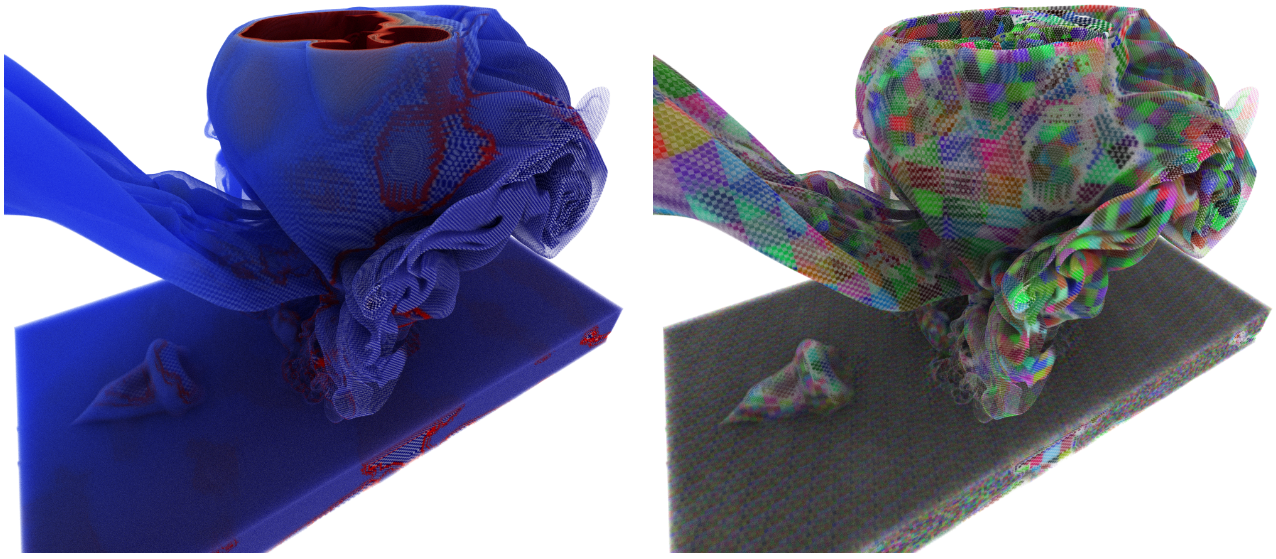

Figure 3: *LANL impact. Left: shaded (top-right) and voxels+stitching elements (bottom-left). Right: gridlets colored by primitive ID.*

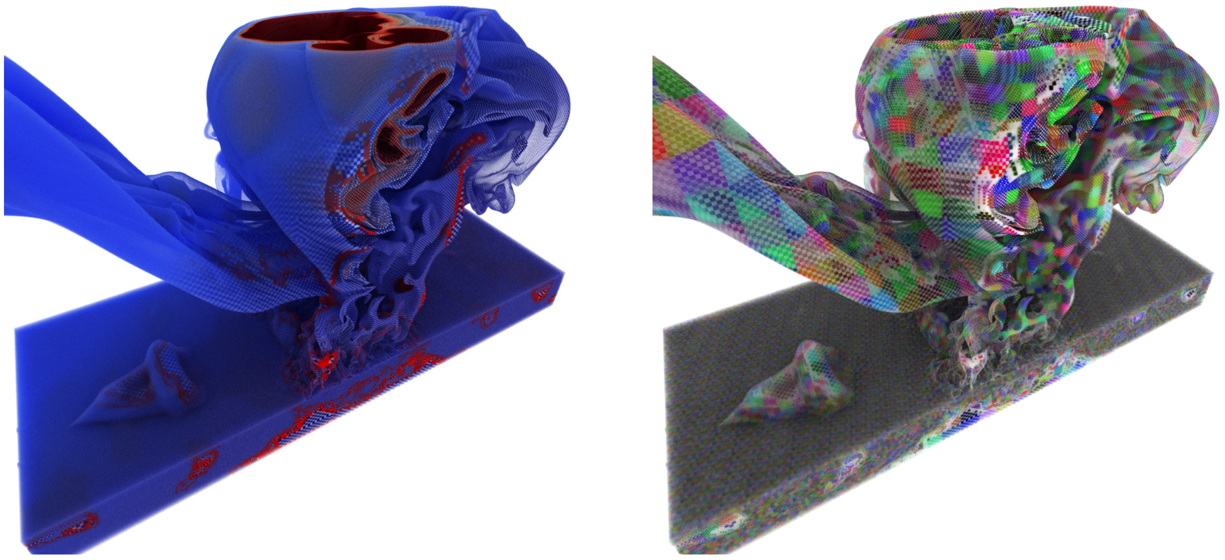

Figure 4: *LANL impact. Left: shaded (top-right) and voxels+stitching elements (bottom-left). Right: gridlets colored by primitive ID. We use a clip plane to reveal the inner structure of the ocean and plume.*

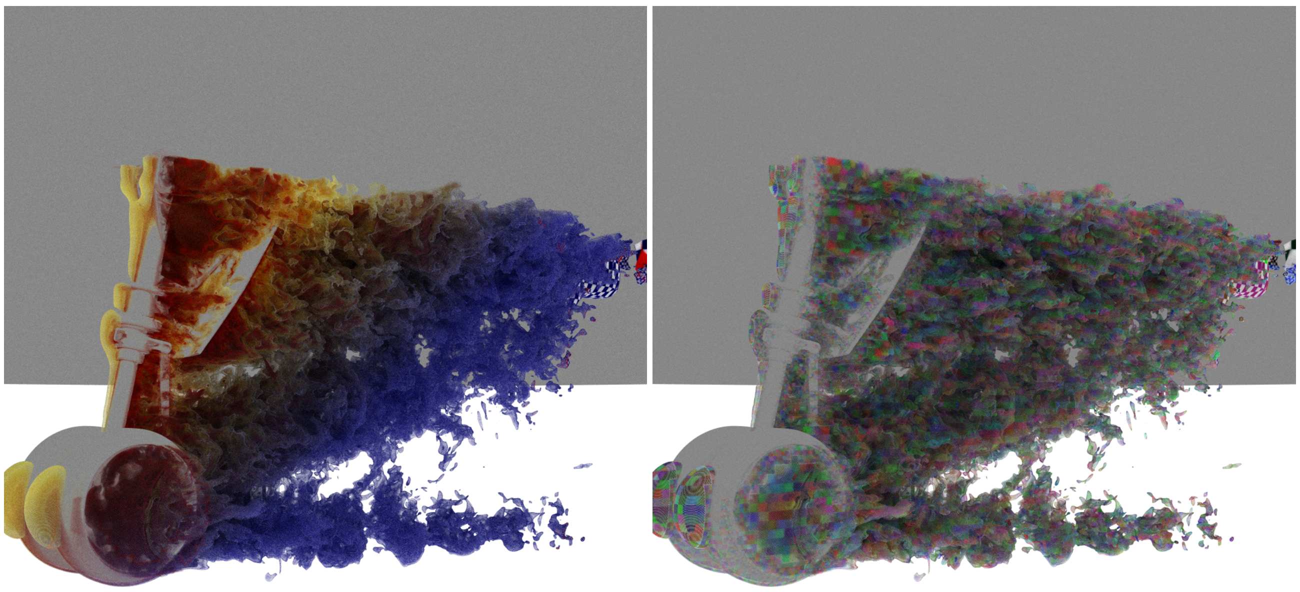

Figure 5: *NASA Landing Gear. Left: shaded (top-right) and voxels+stitching elements (bottom-left). Right: gridlets colored by primitive ID.*

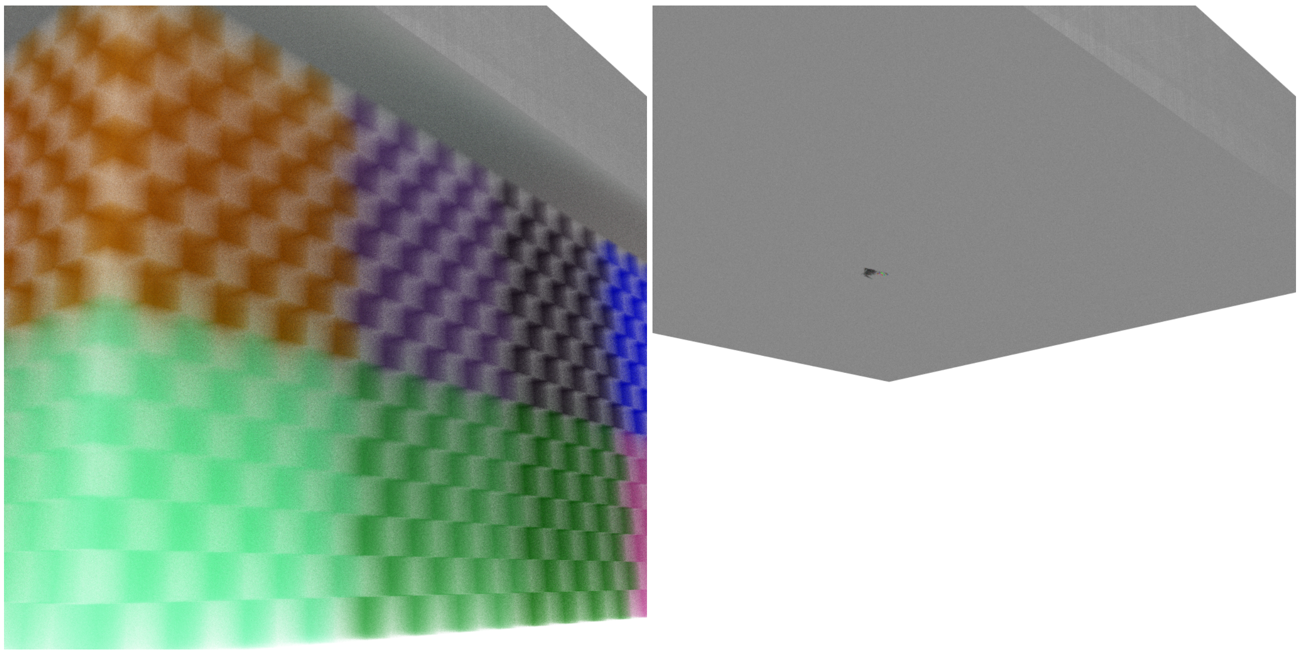

Figure 6: *NASA Landing Gear, zoomed out. Left: showing gridlets colored by primitive ID. Right: same, but with a different transfer function. The gear itself is seen at a distance. This data set suffers from the "teapot in a stadium" problem.*

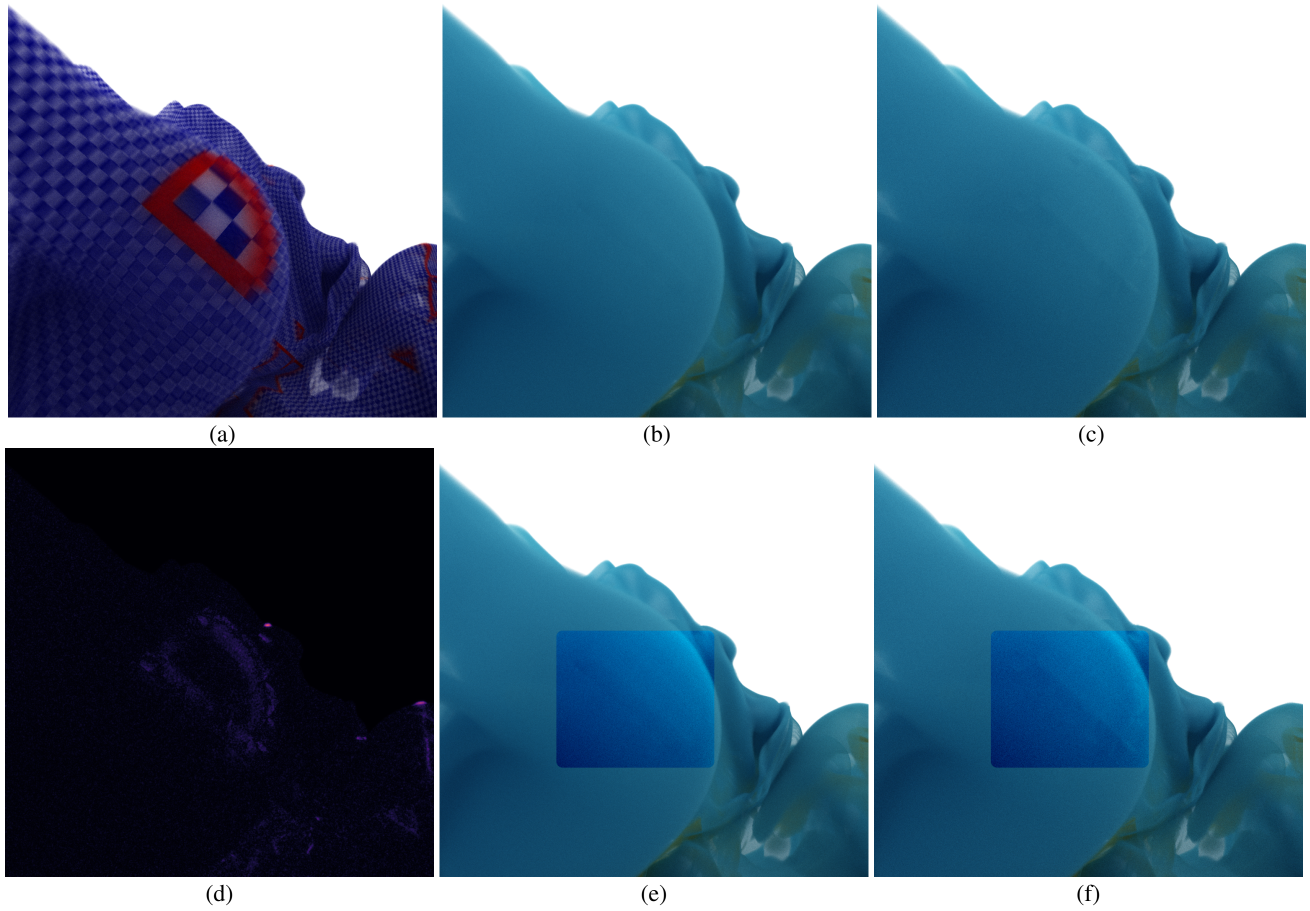

Figure 8: *We use sampling tests that model twisted stitching elements (a) with bilinear faces (b). Element tests that approximate that with planar faces exhibit subtle artifacts (c). In (d) we show a difference image generated with* ꟻLIP *[ANA*20]. (e) shows a contrast-enhanced region of interest of (b). (f) shows a contrast-enhanced region of interest of (c).*

```
// static macrocell size
const static int3 g_mcSize{16,16,16};

// compute macrocell ID from voxel
int3 mcID(Voxel voxel) {
    return voxel.lower/g_mcSize;
}
```

Figure 9: *Mapping voxel coordinates to macrocell IDs.*

```
void makeGrids(int level) {
  // map from voxel corners (int3)
  // to macrocells (3D box, integer coordinates)
  map<int3,box3i> macrocells;

  // macrocells are invalid initially; lazily
  // "activate" only when cubes overlap
  for (auto cube : cubes[level]) {
    Voxel vox(cube,level);
    macrocells[mcID(vox)].extend(cube.bounds());
  }

  // Iterate over all *active* macrocells;
  // these become "gridlets"
  for (auto mc : macrocells) {
    int3 numVoxels = mc.second.size();
    // gridlets store data at the cell corners:
    int numScalars =   (numVoxels.x+1)
                     * (numVoxels.y+1)
                     * (numVoxels.z+1);

    Gridlet gridlet = {
      .lower = mc.second.lower,
      .level = level,
      .numScalars = numScalars,
      .scalarIDs = int[numScalars]
    };

    // Add to gridlet list
    g_gridlets[level].add(gridlet);
  }
}
```

Figure 10: *Algorithm to cluster voxels to gridlets using a virtual uniform grid of macrocells.*

```
bool sample(Gridlet g, float3 pos, float *value){
  box3f bounds = g.bounds();
  if (bounds.contains(pos)) {
    // gridlets store data at the corners
    int3 numScalars = g.dims+1;

    // compute box around position in raster
    // coords on the gridlet's level
    int3 imin = (pos-bounds.lower)/(1<<g.level);
    int3 imax = min(imin+1,numScalars-1);

    // data values at the box corners
    float values[8] = {
      g.scalars[int3(imin.x,imin.y,imin.z)],
      g.scalars[int3(imax.x,imin.y,imin.z)],
      g.scalars[int3(imin.x,imax.y,imin.z)],
      g.scalars[int3(imax.x,imax.y,imin.z)],
      g.scalars[int3(imin.x,imin.y,imax.z)],
      g.scalars[int3(imax.x,imin.y,imax.z)],
      g.scalars[int3(imin.x,imax.y,imax.z)],
      g.scalars[int3(imax.x,imax.y,imax.z)]
    };

    if (all(values) != NAN) {
      // trilinear interpolation
      float3 p = (pos-bounds.lower)/(1<<g.level);
      float3 frac = p-imin;
      *value = trilerp(values,frac);

      return true;
    }
  }

  // either missed bounds, or hit an empty cell
  return false;
}
```

Figure 11: *Gridlet point sampling function.*