Eurographics Symposium on Rendering 2024
E Garces and E. Haines
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 43 (2024), Number 4

Patch Decomposition for Efficient Mesh Contours Extraction Supplemental Materials

P. Tsiapkolis[1,2] and P. Bénard[2]

[1]Ubisoft La Forge, France [2]Inria, Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, France

This supplemental material presents additional data, graphs, and tests using more methods than in the main paper.

1. Mean Sphere statistics

We report in Table 1 the CPU computation time of the patch decomposition as well as the resulting number of patches and their mean bounding sphere radii $\bar{r}$ and mean cone opening angle $\bar{\alpha}$ computed with the Mean Sphere algorithm and a fixed viewing distance $d_{\text{cam}} = 10$.

2. Sensitivity to calibration parameters

To evaluate to which extend runtime performance are sensitive to the calibration, we conducted additional experiments setting $t_{\mathcal{P}} = \beta t_e$ with β from 1 to 4 with a step of 0.2, and comparing the performance to those obtained with the correct calibration parameters for the static models. The results reported in Figure 1 show that, while the best performance are indeed obtained with $t_{\mathcal{P}}/t_e$ ratios close to the calibration parameters, using an approximate ratio has a limited impact on the execution time (about 6% more in the worst case).

Table 1: *Patch decomposition timings (min:seconds), number of patches, mean bounding sphere radii ($\bar{r}$) and normal cone opening angles ($\bar{\alpha}$) computed with the Mean Sphere algorithm.*

	time	# patches	$\bar{r}$	$\bar{\alpha}$
Bunny	3:32	2 998	0.212	9.48
Armadillo	15:14	14 936	0.188	9.63
Dragon	36:23	30 492	0.095	8.17
Thai Statue	9:31:17	339 274	0.162	9.53
Roman Bath	1:20	2 896	0.598	18.21
Spaceship	2:19	3 884	0.793	24.01
Space Station	2:20	3 664	0.638	24.75
Pigman	0:14	1 174	0.156	20.13
Tuba Gunner	1:32	6 771	0.312	18.64
Gawain	2:22	6 086	0.796	11.26

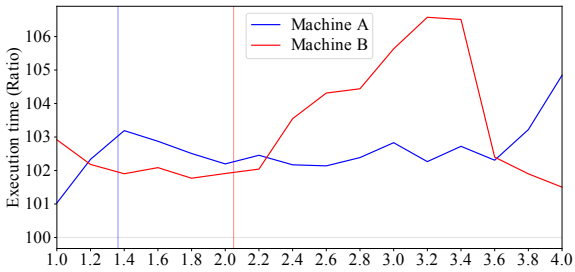

Figure 1: *Execution time measured for different $t_{\mathcal{P}}/t_e$ ratios relative to the timings obtained with the correct calibration parameters (vertical lines), on Machine A (in blue) and Machine B (in red).*

3. Patch decomposition visualization

We show in Figures 2 and 3 the patch decomposition produced by our method and, in the first figure, the patches produced by *meshoptimizer*. We also show the remaining edges after patch culling for the current camera viewpoint.

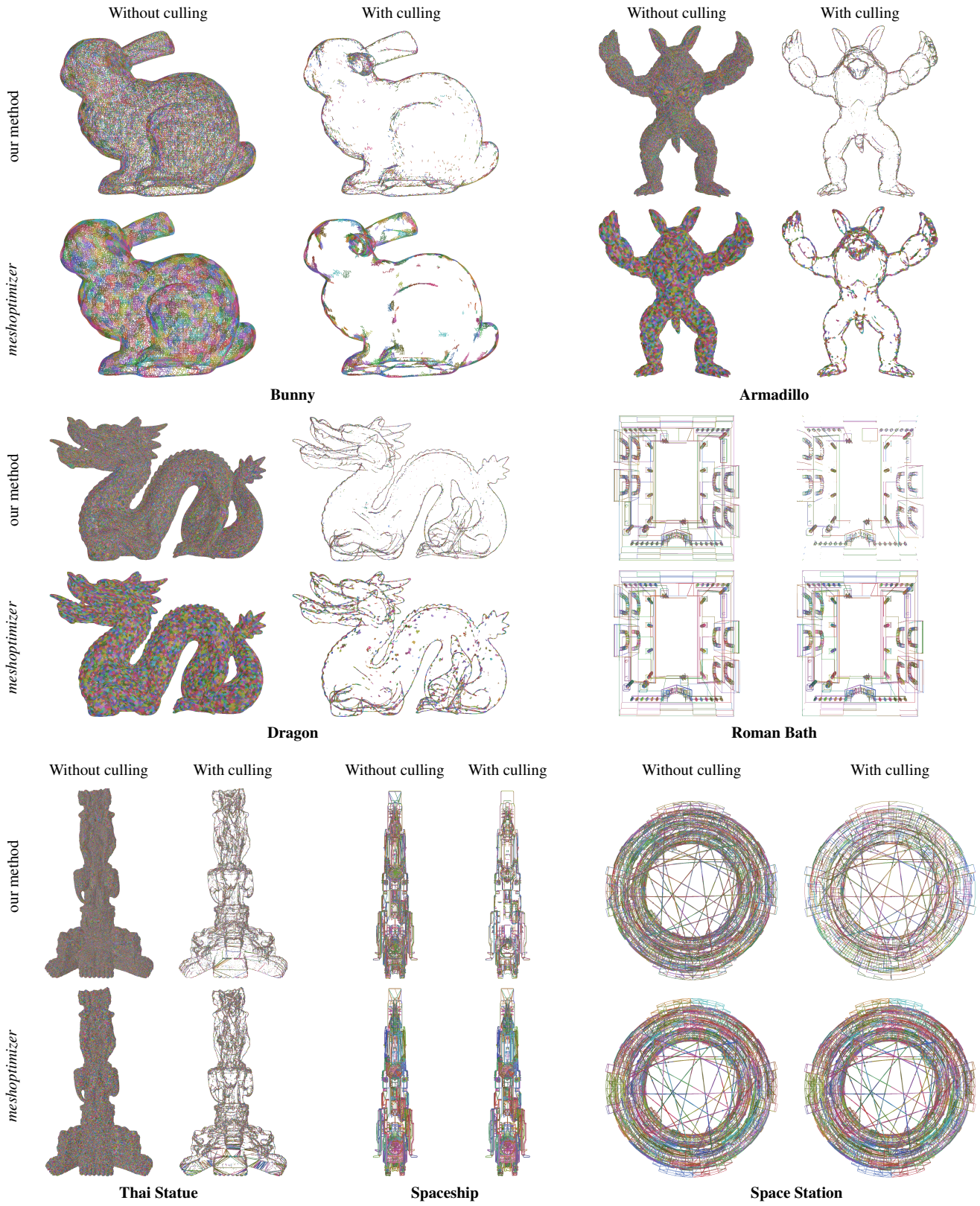

Figure 2: *Visualization of the patch decomposition produced by our method and meshoptimizer without and with patch culling for the Stanford and environment 3D models. The patches are colored by a palette of 64 colors.*

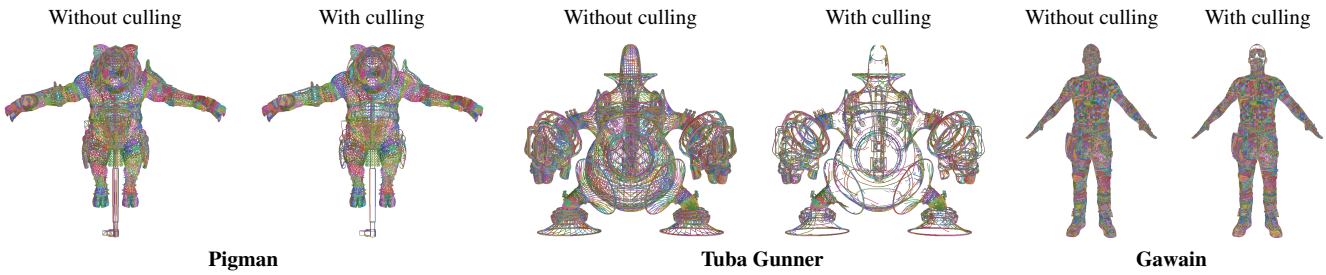

Figure 3: *Visualization of the patch decomposition produced by our method without and with patch culling for the characters 3D models. The patches are colored by a palette of 64 colors.*

4. Additional CPU Tests

This section shows additional methods on the CPU as well as the associated tests using the same testing procedure as in the main document. The methods are as follows:

*Bruteforce Unfiltered (**BFU**)* Brute-force contour edges extratcion on the CPU, computing the normals of the faces adjacent to each edge in a single pass. This is done on unfiltered meshes (i.e., with concave edges).

*Two-Step Bruteforce Unfiltered (**BF2**)* Brute-force on the CPU, computing the normals of every face in a first pass, then using this result when finding the contour edges. This is done on unfiltered meshes.

*Bruteforce Filtered (**Filtered**)* Same as *Bruteforce Unfiltered*, but on filtered models.

*Spatialized Normal Cone Hierarchy (**SNCH**)* Checks the cone hierarchy on the CPU, and then extracts contour edges using *Bruteforce Unfiltered*. It only applies to static objects.

Patch Extraction Culls patches on the CPU, and then extracts contour edges using *Bruteforce Unfiltered*. Four patch decomposition algorithms are compared:

- *Naive Method - 6 Directions (**Naive6**)* Each rigid part's edges are grouped into one of six patches, depending on which direction the sum of that edge's face normals are pointing to. Concavities are filtered, the 6 directions are aligned to the cardinal directions.
- *Naive Method - 42 Directions (**Naive42**)* Same as previous, but with 42 directions based on a twice-subdivided icosphere.
- *Our Method - Mean Sphere (**MS**)* Using the algorithm described in the main document. This version uses the Mean Sphere algorithm and CPU calibration.
- *Our Method - Reduced Sphere (**RS**)* Using the algorithm described in the main document. This version uses the Reduced Sphere algorithm and CPU calibration.

The results are reported in Table 2. Even though we can notice that, on the CPU, the brute-force algorithm in two passes is significantly faster than in a single pass, we chose the single pass brute-force algorithm for our comparisons as it maps better to the GPU method and only requires a list of edges. We can also notice that while **SNCH** can sometimes test less edges than our method, the overhead of the hierarchy is a lot higher than ours, leading to vastly different timings. The Mean Sphere and Reduced Sphere algorithms lead to similar results for a comparable pre-computation time, with a slight advantage for Reduced Sphere.

5. Additional GPU Tests

This section presents additional methods on the GPU as well as the associated tests using the same testing procedure as the main document. The methods are as follows:

*Bruteforce Unfiltered (**BFU**)* Brute-force mesh contours extraction on the GPU, computing the normals of the faces adjacent to each edge and the contour test in a single step. The edges are stored as winged edges represented by four vertex IDs. This is done on unfiltered meshes.

*Two-Step Bruteforce Unfiltered (**BF2**)* Brute-force on the GPU, first computing the normals of every face in a first shader, then using this result when finding the contour edges. This is done on unfiltered meshes.

*Optimized Bruteforce Unfiltered (**Optimized**)* Brute-force on the GPU, working similarly to *Bruteforce Unfiltered*. We use padded buffers to enable vectorization, with an empirically determined Compute shader local size of 16. This is done on unfiltered meshes.

*Optimized Bruteforce (**Filtered**)* Same as the previous method, but on filtered meshes.

Patch Extraction Patch culling on the GPU as a first step, with padded buffers to enable vectorization. This step outputs a list of edge IDs, which are stored in groups of 16 all belonging to the same patch. These are then used by a Compute shader working similarly to *Optimized Bruteforce*. The patch decomposition algorithms described in Section 4 are considered.

The results are reported in Table 3. We can notice that, unlike on the CPU, the brute-force algorithm in two passes is noticeably slower than in one pass. Optimizing the brute-force algorithm to enable vectorization and reducing overhead is also very worthwhile. Our method can once again improve those results significantly.

Table 2: *Mean CPU execution time (in ms) of the various methods tested for each model, and mean percentage of edges tested relative to the number of filtered edges.*

	CPU Reference (ms)			**CPU Patch Methods (ms)**				
	BFU	BFU2	Filtered	SNCH	Naive6	Naive42	MS	RS
Bunny	444.11 *(168.44%)*	99.77 *(168.44%)*	260.41 *(100%)*	162.9 *(11.05%)*	275.45 *(100%)*	211.86 *(76.51%)*	67.65 *(16.29%)*	65.96 *(16.04%)*
Armadillo	2159.16 *(170.85%)*	488.08 *(170.85%)*	1294.27 *(100%)*	805.23 *(11.44%)*	1351.21 *(98.7%)*	1095.7 *(78.62%)*	318.54 *(16.43%)*	320.22 *(16.15%)*
Dragon	5200.37 *(174.35%)*	1162.53 *(174.35%)*	2994.65 *(100%)*	1428.36 *(8.01%)*	3346.96 *(100.0%)*	2763.26 *(81.90%)*	613.61 *(12.65%)*	609.83 *(12.56%)*
Thai Statue	60267.78 *(197.82%)*	13385.85 *(197.82%)*	30168.19 *(100%)*	21600.13 *(11.12%)*	30254.75 *(100.0%)*	29794.83 *(97.56%)*	7397.4 *(16.16%)*	7347.4 *(15.96%)*
Roman Bath	236.96 *(219.11%)*	61.55 *(219.11%)*	139.21 *(100%)*	222.01 *(35.81%)*	153.61 *(100%)*	147.95 *(95.73%)*	70.93 *(33.80%)*	69.76 *(33.26%)*
Spaceship	389.27 *(195.70%)*	95.34 *(195.70%)*	228.36 *(100%)*	383.39 *(67.34%)*	249.75 *(100%)*	253.56 *(99.09%)*	148.73 *(50.52%)*	146.52 *(49.21%)*
Space Station	408.74 *(191.86%)*	96.54 *(191.86%)*	217.29 *(100%)*	338.73 *(82.61%)*	240.65 *(100%)*	238.07 *(97.49%)*	139.77 *(4992%)*	137.33 *(49.05%)*
Pigman	220.18 *(123.83%)*	49.83 *(123.83%)*	183.28 *(100%)*	-	198.37 *(99.68%)*	193.91 *(94.34%)*	154.7 *(79.79%)*	155.39 *(79.67%)*
Tuba Gunner	433.53 *(153.76%)*	99.59 *(153.76%)*	299.46 *(100%)*	-	320.83 *(97.19%)*	249.85 *(69.05%)*	157.63 *(35.39%)*	156.27 *(35.03%)*
Gawain	1559.15 *(138.08%)*	382.86 *(138.08%)*	1253.85 *(100%)*	-	1335.54 *(97.50%)*	1234.65 *(90.38%)*	1066.29 *(81.45%)*	1064.49 *(81.26%)*

Table 3: *Mean GPU execution time (in µs) of the various methods tested for each model.*

	GPU Reference (µs)			**GPU Patch Methods (µs)**			
	BFU	BFU2	Optimized	Filtered	meshoptimizer	Naive42	RS
Bunny	146.27	201.82	25.26	21.01	9.38	13.04	3.95
Armadillo	739.99	1015.81	134.71	88.5	48.77	88.19	28.55
Dragon	1837.48	2564.18	270.69	155.91	96.67	200.29	56.55
Thai Statue	-	-	-	1579.02	1217.16	-	604.38
Roman Bath	87.42	139.33	15.67	8.25	10.4	10.02	4.66
Spaceship	148.12	206.45	23.59	17.86	18.68	16.74	9.82
Space Station	140.99	196.09	25.05	18.68	17.02	15.65	8.87
Pigman	70.16	110.28	13.87	20.81	-	13.83	10.78
Tuba Gunner	158.18	219.3	18.35	27.61	-	19.72	11.54
Gawain	573.69	814.01	89.75	91.4	-	77.91	66.27