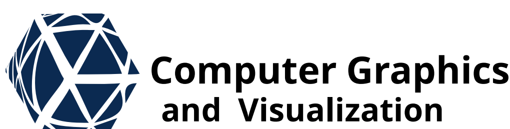
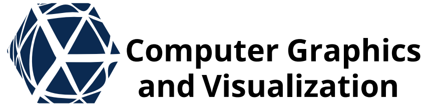


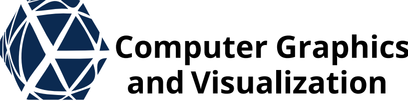
Scientific Visualization

Many Particles & Derived Surfaces

Content



- Large Particle Data
  - Sorting
  - Deferred Shading
  - Occlusion Culling
- Derived Surfaces
  - Metaball Isosurfaces
  - Molecular Surfaces

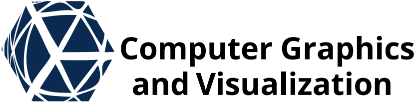


Particles

LARGE PARTICLE DATA

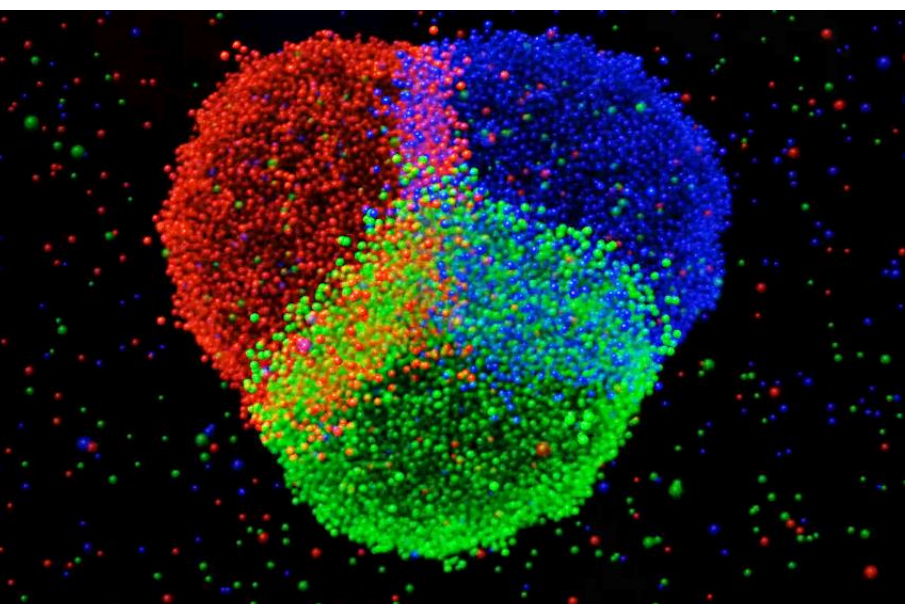
SORTING

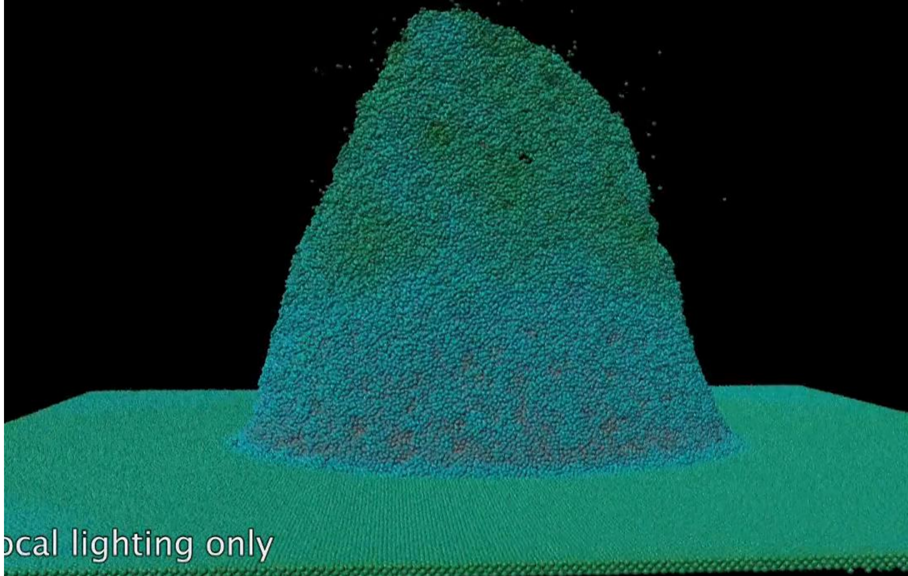
Particle Sorting Applications



Transparent rendering

- per time step transfer particles into particle vbo
- per view compute sort vbo with per particle index and depth (with transform feedback or cuda)
- bind sort vbo as cuda object and sort with radix sort
- use sort vbo as element buffer for indexed rendering



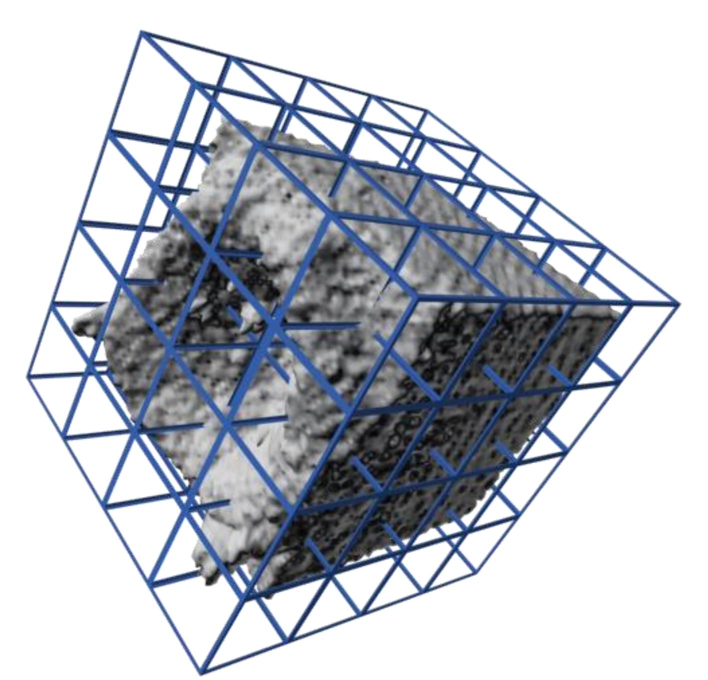


Particle Sorting Applications

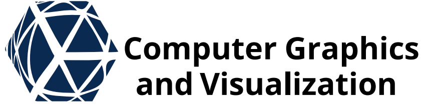
Construction of Particle Grid

Idea: sort particles by grid cell index

- define dimensions of regular grid data structure
- per particle compute index of grid cell in which center of particle lays and store with particle index in sort vbo
- bind sort vbo as cuda object and sort with radix sort
- prefix sums address first particle per cell

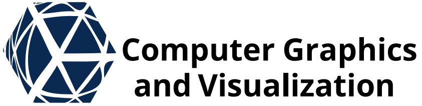


Sorting Algorithms



- Task: rearrange numbers into increasing order
  [170, 45, 75, 90, 2_a, 802, 2_b, 66] => [2_a, 2_b, 45, 66, 75, 90,170, 802]
- A sorting algorithm is called
  - stable if it preserves order of equal numbers from input series (compare 2_a, 2_b)
  - in-place if it does not need additional memory (typically algorithms need an additional array to store the output sequence)
  - non-comparative if it avoids direct comparison of to be sorted elements
- If additional data to be sorted with the numbers is called load (typically particle index)

GPU Sorting



- Architecture: Single Instruction, Multiple Thread (SIMT)
- On SIMT latency is hided by multi-threading scheduler (> 5000 threads should run in parallel)
- If threads execute same code but run into different parts of **`if`** or **`switch`** clause, behavior called divergent
- 32 threads form warp in which code divergence is slow
- Between warps code divergence is no penalty
- Thread local memory is much faster (try copy to local memory, process, copy result back as often as possible)
- If per thread data is consecutive in global memory within each warp, highest performance achievable

Satish, Nadathur, Mark Harris, and Michael Garland. "Designing efficient sorting algorithms for manycore GPUs." 2009 IEEE International Symposium on Parallel & Distributed Processing. IEEE, 2009

Radixsort – with Lists (Bucket Sort)

- Radixsort is stable, non-comparative algorithm
- To be sorted numbers are processed from least significant to most significant digit in one pass per digit
- Per pass split all numbers into per digit value list
- Process/collect numbers in order of lists
- Wikipedia example (no intermediate number collect):

  [170, 45, 75, 90, 2, 802, 2, 66]

  Starting from the rightmost (last) digit, sort the numbers based on that digit:

  [{170, 90}, {2, 802, 2}, {45, 75}, {66}]

  Sorting by the next left digit:

  [{*0*2, 802, *0*2}, {45}, {66}, {170, 75}, {90}]

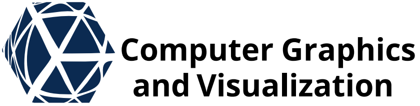
  Notice that an implicit digit *0* is prepended for the two 2s so that 802 maintains its position between them.

  And finally by the leftmost digit:

  [{*00*2, *00*2, *0*45, *0*66, *0*75, *0*90}, {170}, {802}]

- Stable version possible for binary digits based on swap

Radixsort – with Counters



- Per digit value introduce counter that first stores number of digit appearances and then location in output array

1. compute histogram in counters
2. compute cumulative counters (prefix sum)
3. place each number in target location in output array (increment counter for each appearance)

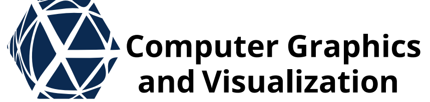
- Example:

1.	**[170, 45, 75, 90, 2, 802, 2, 66]**	[**0**:2,**1**:0,**2**:3,**4**:0,**5**:2,**6**:1,**7**:0,**8**:0,**9**:0]
2.		[**0**:0,**1**:2,**2**:2,**4**:5,**5**:5,**6**:7,**7**:8,**8**:8,**9**:8]
3.	**[170, 90, 2, 802, 2, 45, 75, 66]**	[**0**:2,**1**:2,**2**:5,**4**:5,**5**:7,**6**:8,**7**:8,**8**:8,**9**:8]

repeat for each digit

- In-place version possible

Parallel Radixsort



- Parallel version splits input sequence into one block per thead $j < \mathrm{m}$
- Per digit value $i < n$ and thread j counters c_j^i are computed in parallel and stored in a 2d table
- With global counts $c_i = \sum_j c_j^i$ and global offset $o_i = \sum_{i'<i} c_{i'}$ thread local offsets compute to

$$o_j^i = o_i + \sum_{j'<j} c_{j'}^i$$

- Reorder step is done in parallel again with thread local offsets

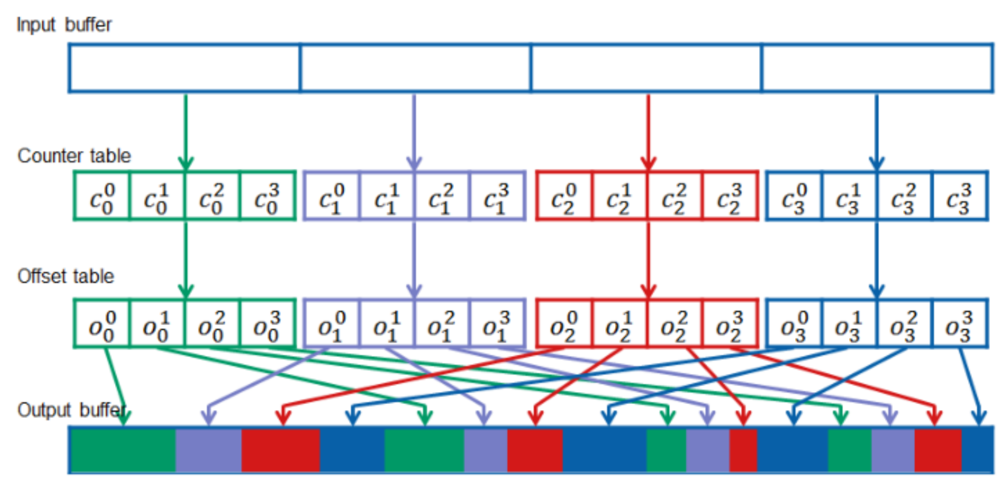
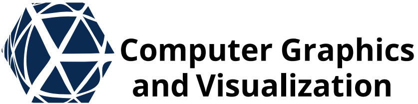


Figure 1: *Parallel radix sort using four threads.*
2-bit digits with 4 digit values

- GPU implementation of parallel radix sort optimizes local memory usage and work group sizes of different steps

Merge Sort



- Merge sort is stable, comparative algorithm

1. Split to be sorted numbers in one element lists which are automatically sorted
2. Merge sort pairs of successive lists in logarithmic number of passes over all lists till one sorted list is left

6 5 3 1 8 7 2 4

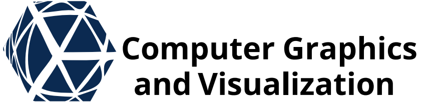
© *Wikipedia, source*

- Stable version with constant additional memory possible

Parallel Merge Sort

- Main question is how to parallelize sequence merging?
- Let $\mathrm{rank}(a, A)$ denote the location of a in A.
- Case 1: sequences are small ($n \leq t = 256$)
  - merge sorted sequences A&B in one warp inside local memory
  - assign per thread small number of elements from A or B
  - For each element $a_i \in A$ compute target location in C from $\mathrm{rank}(a_i, C) = i + \mathrm{rank}(a_i, B)$ using binary search on B. (same for b_i)
- Case 2: sequences are large ($n > t$)
  - Idea: split A, B into small subsequences A_j, B_j and sort along Case 1
  - define split numbers $S_A = (a_{t \cdot k})_k$ and $S_B = (b_{t \cdot k})_k$ on A and B that produce patches individually in A and B with k elements
  - merge sort S_A and S_B into S and use tile ranges $T_j = [S_{2j}, S_{2j+1})$ as split numbers to extract A_j and B_j (resulting patches have $\leq 2k$ elements)
  - Subrange boundaries in A with respect to $s \in S$ computed from $\mathrm{rank}(s, A)$ which is simple if s is from A. For $s = b_i \in B$ compute $t = \mathrm{rank}(s, S_A) = \mathrm{rank}(s, S) - i$ and $\mathrm{rank}(s, A) = t \cdot k$ (same for B)

Parallel Merge Sort



- Example for tile construction for $t = 4$:

A=[1,4,6,10,12,15,20,40,43,55,80,101,111,130,141,150]

B=[2,7,9,13,17,25,31,36,52,67,68,90,95,99,104,120]

S_A =[1,12,43,111]

S_B =[2,17,52,95]

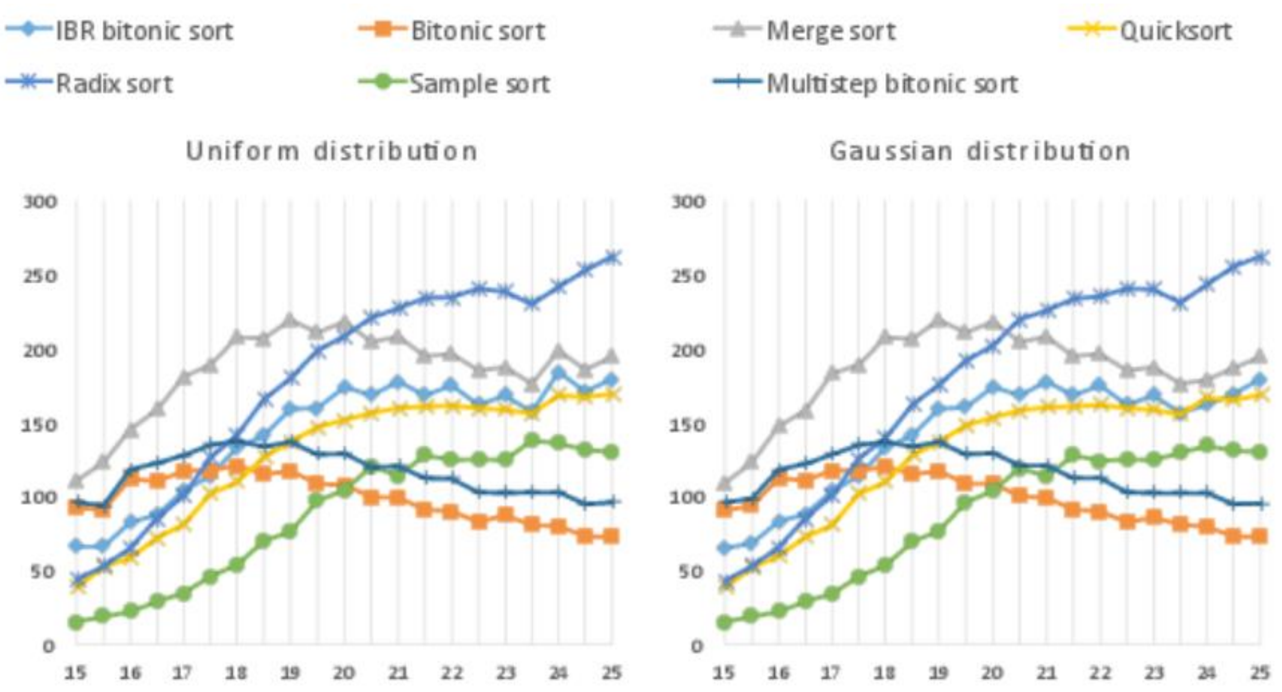
S =[1,2,12,17,43,52,95,111]

T_1 =[1,2) => A_1=[1], B_1=[] => C_1=[1]
T_2 =[2,12) => A_2=[4,6,10], B_2=[2,7,9] => C_2=[2,4,6,7,9,10]
T_3 =[12,17) => A_3=[12,15], B_3=[13] => C_3=[12,13,15]
T_4 =[17,43) => A_4=[20,40], B_4=[17,25,31,36] => C_4=[17,20,25,31,36,40]
T_5 =[43,52) => A_5=[43], B_5=[] => C_5=[43]
T_6 =[52,95) => A_6=[55,80], B_6=[52,67,68,90] => C_6=[52,55,67,68,80,90]
T_7 =[95,111) => A_7=[101], B_7=[95,99,104] => C_7=[95,99,101,104]
T_8 =[111,∞] => A_8=[111,130,141,150], B_8=[120] => C_8=[111,120,130,141,150]

Comparison of Parallel Sorting Algos

- Sorting **32** bit key value pairs

Darko Božidar and Tomaž Dobravec: Comparison of parallel sorting algorithms, University of Ljubljana,Slovenia Technical report November 2015



X-axis: Binary logarithm of the sequence length, Y-axis: sort rate in Million/second

Comparison of Parallel Sorting Algos

- Sorting **64** bit key value pairs

Darko Božidar and Tomaž Dobravec: Comparison of parallel sorting algorithms, University of Ljubljana,Slovenia Technical report November 2015

IBR bitonic sort
Bitonic sort
Merge sort
Quicksort
Radix sort
Sample sort
Multistep bitonic sort

Uniform distribution

180
160
140
120
100
80
60
40
20
0

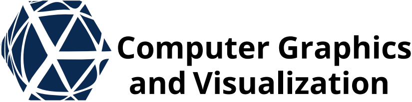
15 16 17 18 19 20 21 22 23 24

Gaussian distribution

180
160
140
120
100
80
60
40
20
0

15 16 17 18 19 20 21 22 23 24

X-axis: Binary logarithm of the sequence length, Y-axis: sort rate in Million/second

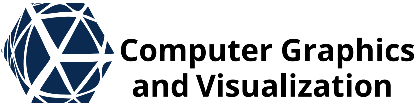


Particles

LARGE PARTICLE DATA

DEFERRED SHADING

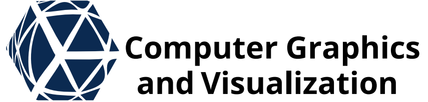
Deferred Shading – Motivation



- Aliasing artefacts arise if particle size is smaller than pixel size ➔ particle shape not visible anymore

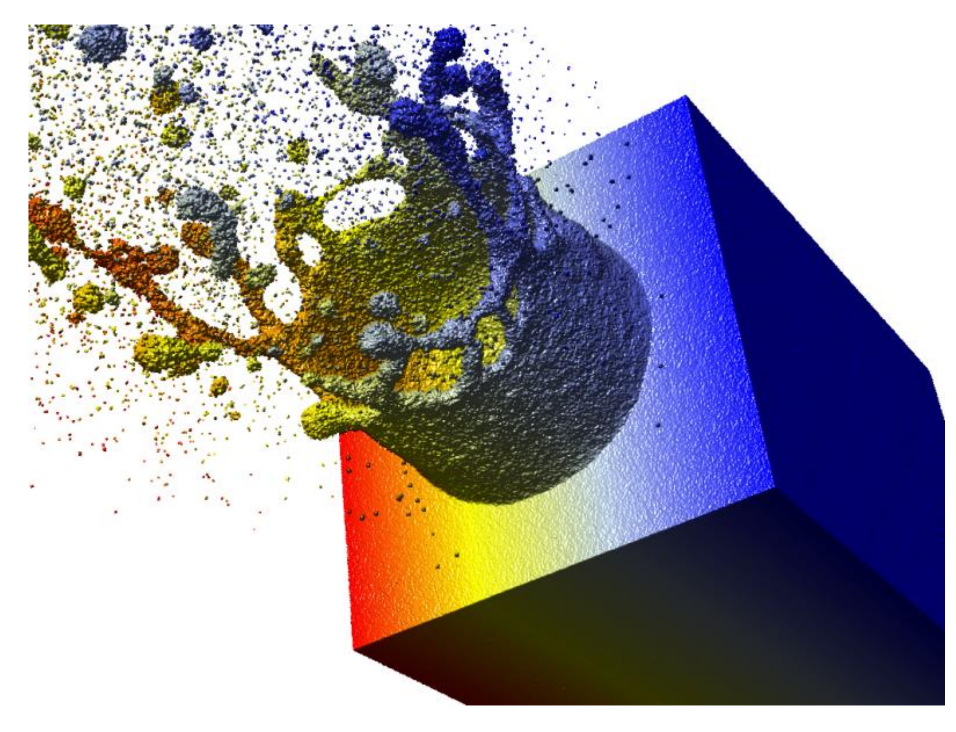
Standard Ray-Casting | 8×8 supersampling | image space normals

Deferred Shading



approach:

- render particles and store depth, normal, projected size and color in fbo with two color textures
- estimate per pixel secondary normal from depth buffer by fitting local plane or higher order surface (careful: ignore neighbors with depth jump relative to current sample)
- Blend primary and secondary normal based on projected particle size to achieve smooth transition between normals

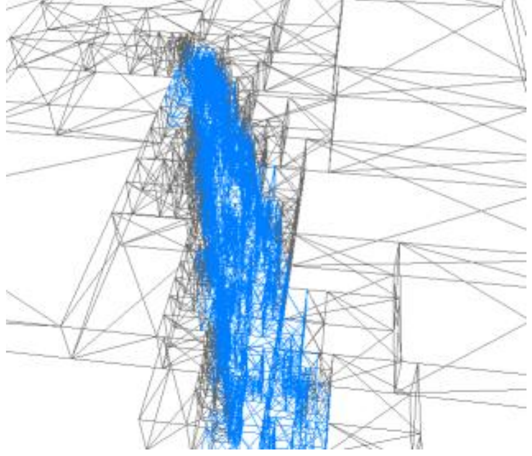


Particles

LARGE PARTICLE DATA

OCCLUSION CULLING

Occlussion Culling – Overview



images from: Bittner, Jiří, et al. "Coherent hierarchical culling: Hardware occlusion queries made useful." Computer Graphics Forum. Vol. 23. No. 3., 2004.

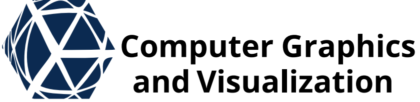
- In CG there are two basic techniques often used in occlusion culling approaches:
- hardware occlusion queries … supports counting of vertices or fragments produced during rendering
- hierarchical depth buffer … extent depth buffer to mipmap and discard primitives or objects in geometry shader

Occlussion Culling – HWOQs

Hardware Occlusion Queries (HWOQ)

- Since OpenGL 2.0 one can generate Query objects with glGenQueries and enclose draw calls in glBeginQuery … glEndQuery brakets
- Queries count the number of fragments passing all fragment filters or the number of vertices generated in the last vertex stage (i.e. geometry shader)
- Write masks are typically turned off to avoid framebuffer changes during queries
- With glGetQueryObject one can check with GL_QUERY_RESULT_AVAILABLE whether query is available and retrieve the counts, which flushes the rendering pipeline
- To avoid pipeline flushes an important strategy is to batch queries or do other things in between

Occlussion Culling – HWOQs



Condition Rendering

- Since OpenGL 3.0 one can make draw calls dependent on the result of HWOQs by enclosing them in glBeginConditionalRender … glEndConditionalRender brackets
- There are different conditional rendering modes
  - GL_QUERY_WAIT … waits for query and therefore flushes pipeline, checks query result and renders only if query succeeded
  - GL_QUERY_NO_WAIT … allows OpenGL driver to decide whether to render now or wait for query result
  - GL_QUERY_BY_REGION[_NO]_WAIT … allows OpenGL driver specific optimization to discard conditional rendering commands that do not affect the fragments (framebuffer region) that caused the query result to be non-zero.

Occlussion Culling – HWOQs

- **Bittner, Jiří, et al. "Coherent Hierarchical Culling: Hardware occlusion queries made useful." Computer Graphics Forum. Vol. 23. No. 3., 2004.**
- **setting:** hierarchical scene description with node bounding volumes. Traverse hierarchy in front to back order and terminate at leaves or invisible nodes.

Main ideas to avoid CPU stalls and GPU starvation:

- Exploit temporal coherence by assuming previously visible nodes to be visible again and simply rendering them
- introduce query queue to interleave rendering of visible nodes with queries

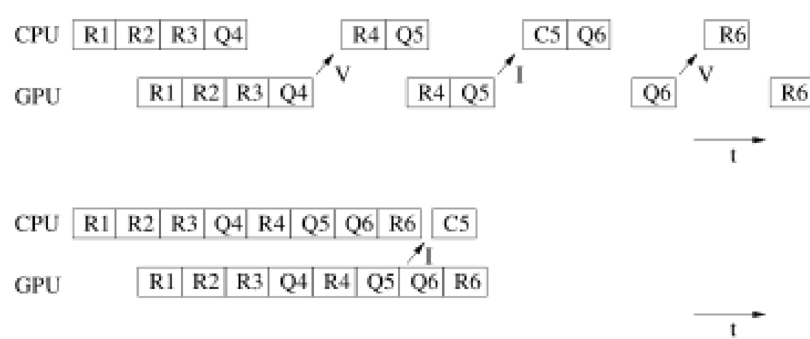


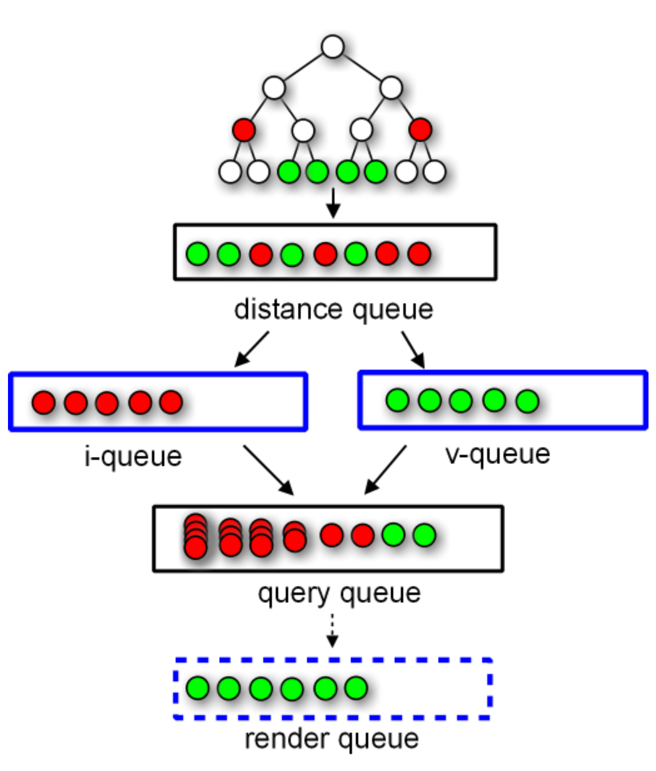
Figure 1: *(top) Illustration of CPU stalls and GPU starvation. Qn, Rn, and Cn denote querying, rendering, and culling of object n, respectively. Note that object 5 is found invisible by Q5 and thus not rendered. (bottom) More efficient query scheduling. The scheduling assumes that objects 4 and 6 will be visible in the current frame and renders them without waiting for the result of the corresponding queries.*

Occlussion Culling – HWOQs

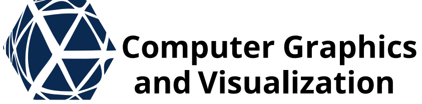
- **Oliver Mattausch, Jiří Bittner, and Michael Wimmer. "CHC++: Coherent Hierarchical Culling revisited." Computer Graphics Forum. Vol. 27. No. 2, 2008.**

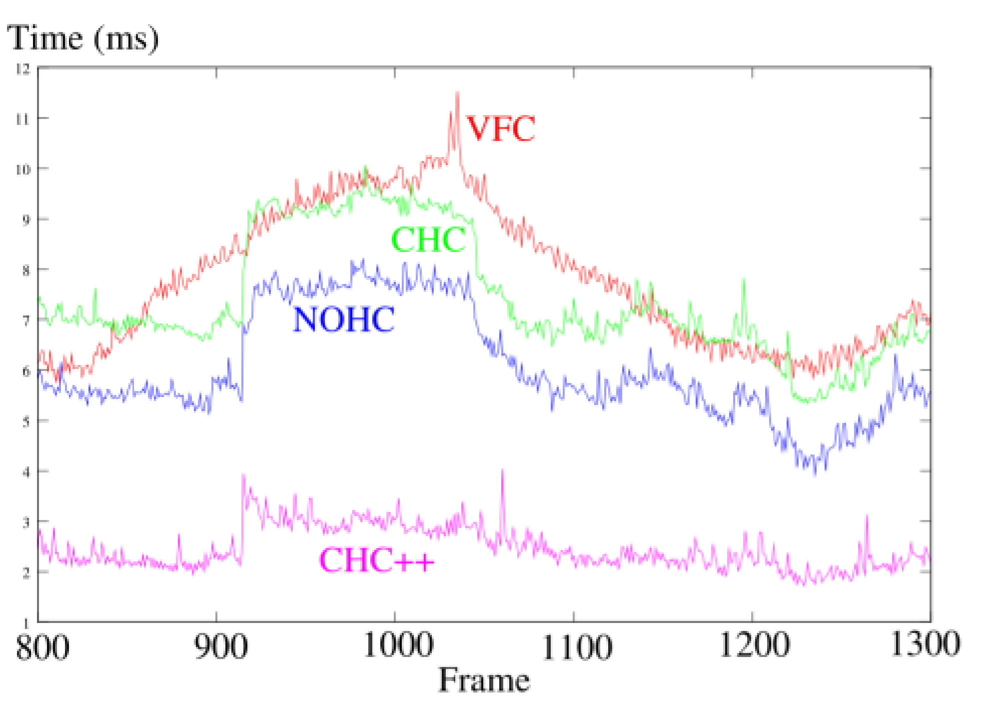
Extension of CHC method to CHC++ with the following new ideas to reduce the number of state changes:

- separate queues for previously visible vs invisible nodes to form query batches
- introduce multi-queries based on grouping with respect to identical OpenGL states
- randomized temporal subsampling of queries for visible nodes
- tighter bounding volumes by rendering child bounding boxes



Occlussion Culling – HWOQs





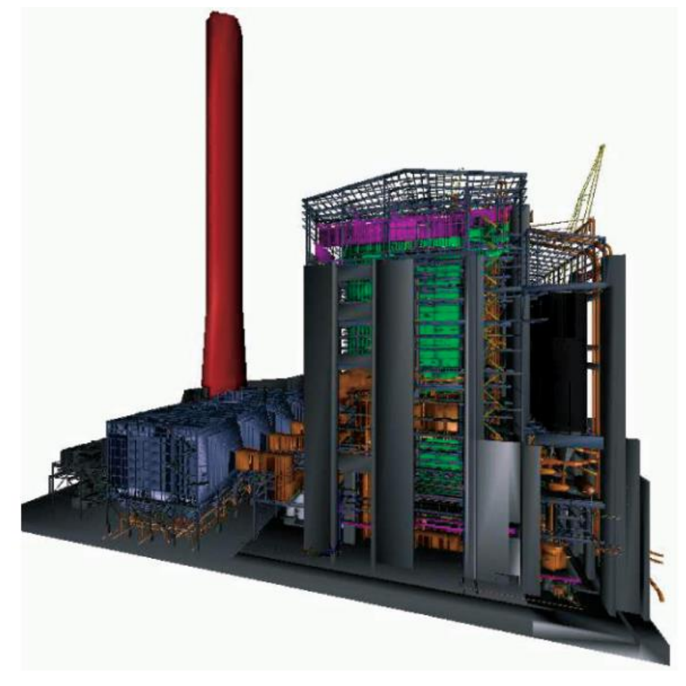


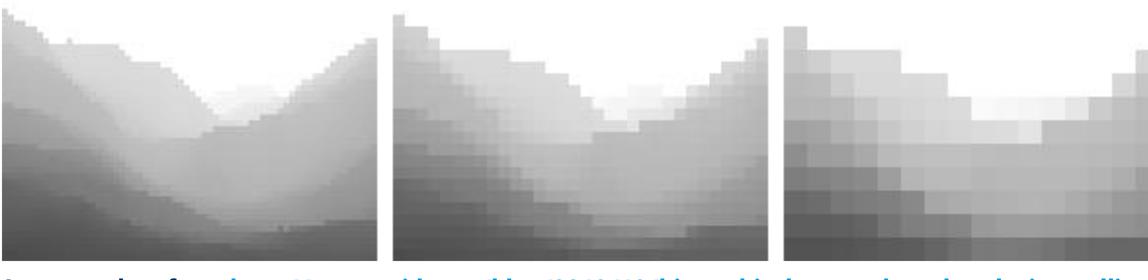
Figure 1: *Frame time comparison for a walkthrough of the Powerplant model for View Frustum Culling (VFC), Coherent Hierarchical Culling (CHC), Near Optimal Hierarchical Culling (NOHC), and our new algorithm (CHC++).*

Occlusion Culling – Hierarchical Z

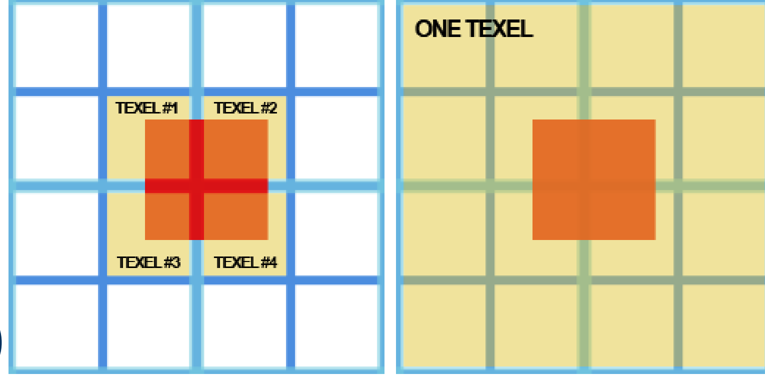
Hierarchical Z/Depth approach

Greene et al. "Hierarchical Z-buffer visibility". SIGGRAPH 1993

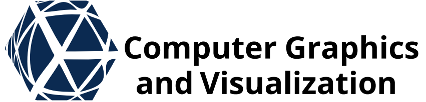
- split scene into occluders and rest
- render occluders to construct depth buffer
- use ping pong rendering to construct depth buffer mipmap pyramid combining texels with max operator
- During rendering of rest
  - transform bounding of primitive to window coordinates
  - determine mipmap level
  - do visibility test by comparing minimal primitive depth with four surrounding texels in mipmap level of hierarchical depth buffer
- >20000 tests per second (see slides)



Images taken from http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling



Occlusion Culling – Particle Data



Grottel et al. "Coherent culling and shading for large molecular dynamics visualization". In CGF 29.3, pp. 953-962. 2010

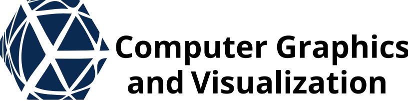
- sort particles in grid
- use particles in previously visible cells as occluders
- conservatively render occluders into depth buffer
- start occlusion queries of previously invisible cells
- build hierarchical z-buffer in parallel to evaluation of queries
- render particles in newly visible cells with hierarchical z-buffer based visibility tests
- use deferred shading for antialiasing and better performance

1. init depth buffer
occluding glyph silhouetts
occluding points/ fragments
depth map
eye
grid cell bounding boxes
2. start occlusion queries (HWOQs)
occlusion queries
max-depth mip map
3. generate max-depth mip map
4.1 collect occlusion queries & render cells
glyph ray casting
point splatting
parameter buffers
4.2 deferred shading w/ normal estimation
final image

Occlusion Culling – Particle Data

	Col01	Col02	Col03	Col04	Col05	Col06	Col07	Col08	Col09	Col10	Col11	Col12	Col13	Col14	Col15
			culling					quantization (with two-level culling)				deferred shading		visible data	
#glyphs	data set	view configuration	none	cell	vertex	both	both	shorts	shorts	GS	inst.	shorts	floats	cells (%)	# glyphs
	caching		no	yes	yes	yes	no	no	yes	no	no	yes	yes		
107k	D1	S-Best	28.21	76.06	172.67	90.23	89.61	80.58	91.62	86.62	84.79	84.39	82.30	8.09	3600
		S-Worst	32.12	57.25	137.37	68.98	69.88	63.65	68.93	69.67	70.03	66.95	67.45	23.38	3136
		P-Best	621.20	99.42	195.34	97.22	104.70	81.02	70.31	75.14	74.62	69.68	104.02	39.38	27713
		P-Worst	593.23	99.27	200.96	98.33	101.75	88.05	72.75	74.20	73.76	72.35	97.39	45.75	46338
4.5M	D2	S-Best	4.60	17.96	7.22	42.60	39.62	37.94	40.52	37.64	39.54	40.53	40.90	6.67	12594
		S-Worst	6.92	18.77	12.64	36.30	33.42	30.97	35.87	28.58	29.47	35.47	36.04	20.53	115003
		P-Best	14.17	69.90	24.44	69.81	42.84	48.04	72.83	45.24	32.53	67.28	65.75	6.67	44804
		P-Worst	15.46	46.78	12.91	73.15	65.27	54.74	68.21	65.56	59.24	66.45	66.70	22.93	261221
44.5M	D3	S-Best	1.29	7.62	1.22	18.27	15.23	14.51	17.89	15.64	13.65	17.99	17.78	6.67	153786
		S-Worst	1.18	3.21	1.06	8.48	6.19	5.81	8.20	6.53	5.62	8.36	8.67	18.70	623188
		P-Best	1.70	16.91	1.62	16.61	9.71	9.37	16.79	11.65	7.31	16.79	16.66	6.70	460425
		P-Worst	1.56	38.02	1.65	38.52	16.53	15.99	37.15	32.54	17.29	35.70	34.69	18.70	989268
48M	D4	S-Best	0.77	1.94	1.23	5.96	4.28	3.80	5.85	4.92	3.77	5.89	5.94	6.67	160066
		S-Worst	0.95	2.33	1.03	7.02	4.44	4.41	6.92	5.72	4.61	7.14	7.23	59.35	654718
		P-Best	1.26	12.41	1.93	12.36	6.56	6.77	12.15	10.03	5.19	12.26	12.29	6.67	214358
		P-Worst	1.19	15.98	1.36	15.43	9.32	9.05	15.77	11.85	5.89	16.61	15.76	58.90	1144420
100M	D5	S-Best	0.88	8.04	0.73	13.65	9.38	9.18	13.45	10.00	7.92	13.43	13.59	6.67	740977
		S-Worst	0.92	2.85	0.55	5.22	3.32	3.42	5.19	3.91	2.80	5.24	5.29	18.70	1313734
		P-Best	1.08	20.71	0.90	20.40	11.70	10.93	20.60	16.92	9.04	19.89	19.63	6.67	799347
		P-Worst	1.10	8.34	0.88	8.26	4.25	3.46	8.27	6.30	3.31	8.19	8.13	18.70	1521383

Table 2: *Performance measurements of our method. All numbers denote frames per second if not otherwise noted. We selected different views such that glyphs are large enough in screen space and rendered as raycast spheres (S-*), or simple points (P-*). We also chose viewing directions and distances such that we obtain a best (*-Best) and worst (*-Worst) case for the culling algorithms (i. e. maximum/minimum number of grid cells occluded). The last two columns show statistics for the case that both culling levels are active: the percentage of the grid cells that are visible under the respective view configuration (column 14), and the number of glyphs that are actually raycast after the vertex culling stage (column 15). Column 3 (culling: none) shows the baseline performance for the unoptimized approach. Columns 6 and 7 demonstrate the impact of caching. The performance impact of deferred shading and normal estimation can be seen comparing columns 7 and 13 as well as 9 and 12, respectively.*

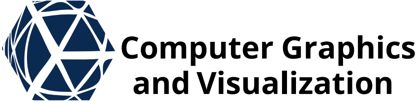


Particles

DERIVED SURFACES

METABALL ISOSURFACES

Metaball Rendering – Motivation



- clusters of general shape can be nicely visualized with metaballs:

Figure 41: Point-based visualisation of an *argon* nucleation simulation data set. The left image highlights molecular clusters using different colours; the middle image uses ellipsoids and the right one uses metaballs. The metaball visualisation shows closed surfaces of molecule clusters while not exaggerating their size.

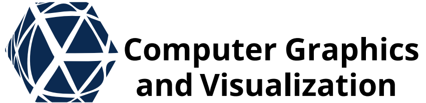
Applications

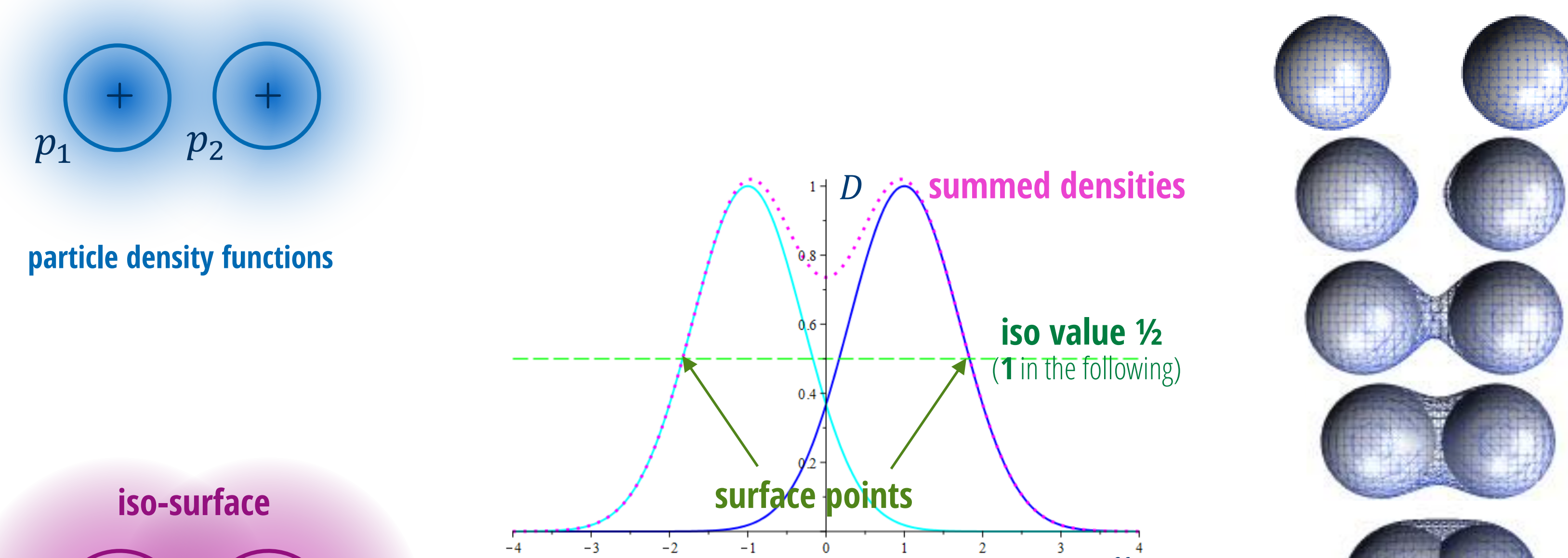
Chromatophore (9.6M atoms)
3.4 fps (Geforce GTX 580)

fluid on paddle wheel

Laser Ablation Dataset

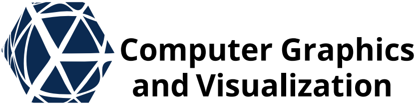
Metaball Surfaces





[Blinn 1982] J. F. Blinn, "A Generalization of Algebraic Surface Drawing," *ACM Transactions on Graphics*, vol. 1, no. 3, pp. 235–256, 1982.

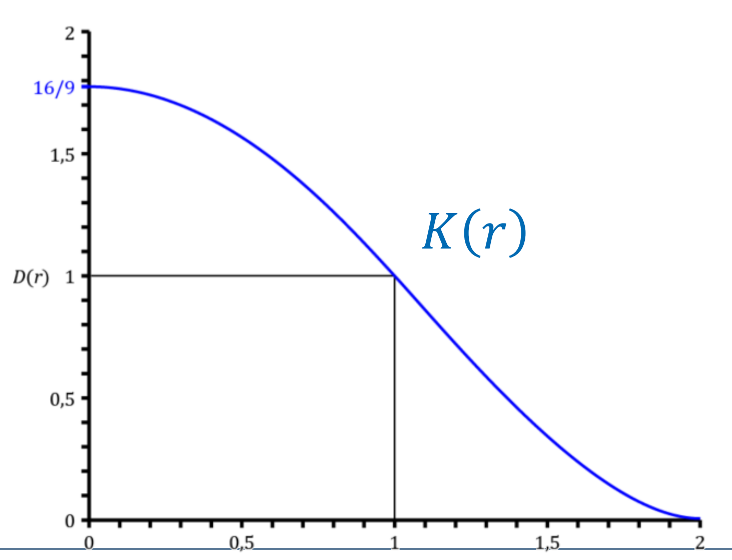
Metaball Surfaces



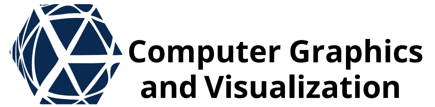
- The metaball surface of a cluster with n particles $\underline{\boldsymbol{p}}_i$ of radius R_i is defined as the iso-surface at value **1** of summed density function:

$$D(\underline{\boldsymbol{x}}) = \sum_{i=1}^{n} K\left(\frac{\left\|\underline{\boldsymbol{x}} - \underline{\boldsymbol{p}}_i\right\|}{R_i}\right) = 1$$

  with a free to choose kernel function $K(r)$.
- Blinn originally proposed a Gaussian kernel function, but in rendering compactly supported kernel functions with a maximum influence sphere of r' times the particle radius are commonly used
- In the following approaches $r' = 2$ is chosen and $K(r) = c\left(1 - \left(\frac{r}{r'}\right)^2\right)^2$
- To ensure that the original particle spheres are recovered for isolated particles, the constant is chosen to be $c = \frac{16}{9}$, such that $K(1) = 1$.

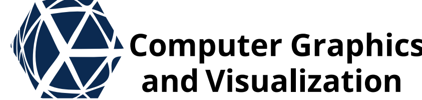


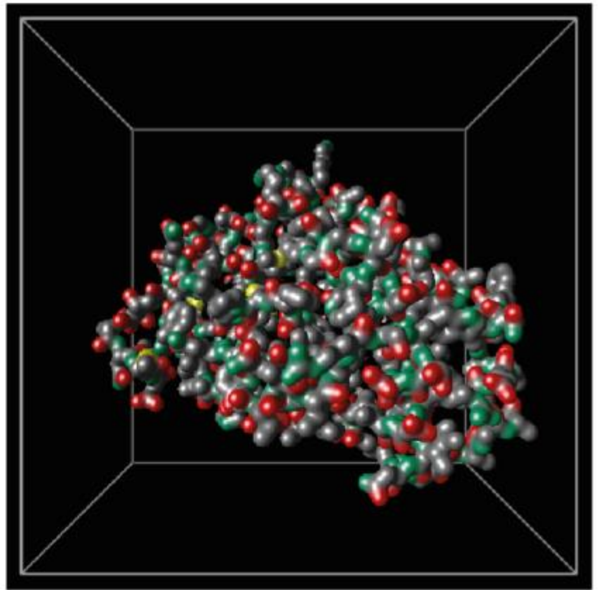
Raycasting Metaball

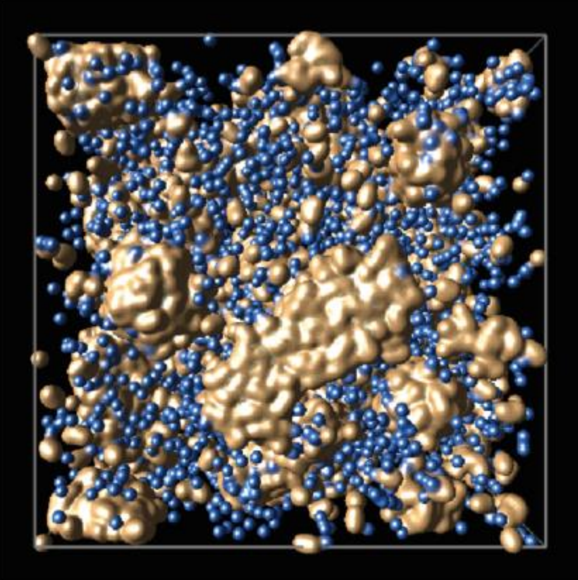


- Two metaball rendering approaches are presented in
  C. Müller, S. Grottel, and T. Ertl, "Image-Space GPU Metaballs for TimeDependent Particle Data Sets," in Proceedings of VMV , 2007, pp. 31–40
- **first approach** is a per particle raycasting of the metaball surface very similar to standard sphere raycasting with support for particle culling
- For access to the density field, a so called vicinity texture stores for each particle all other particles that can influence it
- In fragment shader influencing particles are read from vicinity texture ray-isosurface inter-section is computed iteratively
- Performance is quite bad though

Metaball Rendering – Datasets







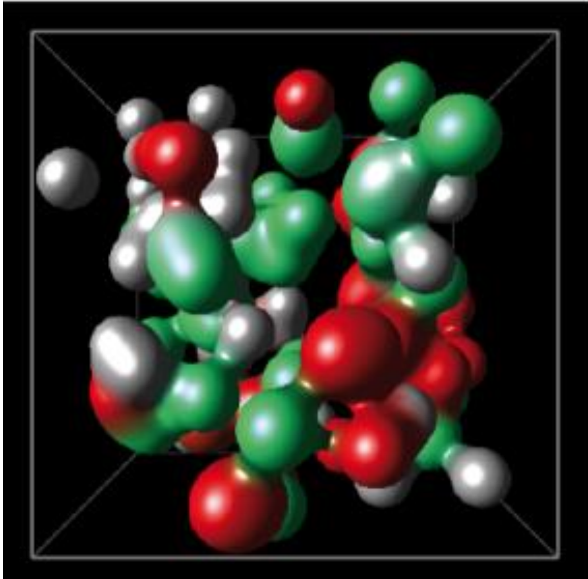
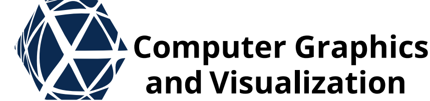


Figure 44: Additional data sets tested with the metaball technique: left: disulfide bond formation protein *1A2J* from the protein data base [BWF+00]; right: a nucleation simulation of supersaturated *ethane*.

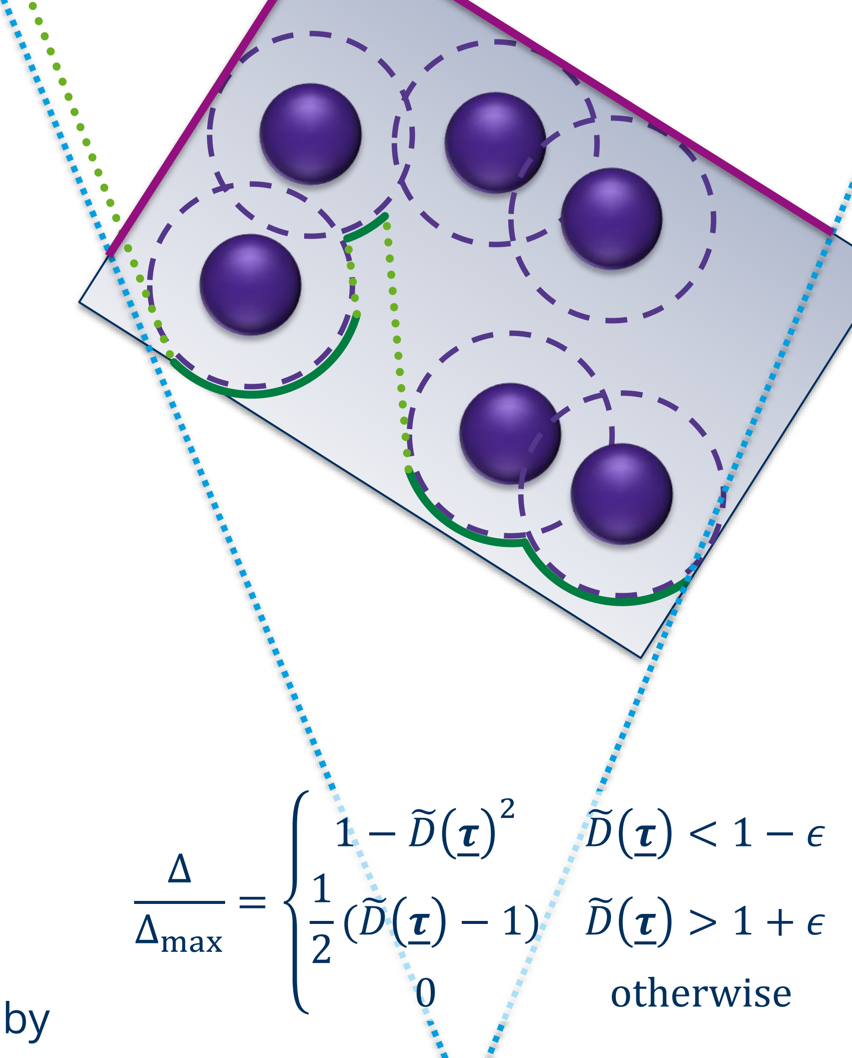
Sim-3 test dataset

Data Set	# Spheres	Processing Time	Non-Empty Groups	Empty Groups	Size Min/ Avr/Max
Argon	5000	113 ms	3684	1316	1/39.3/125
Ethane	25000	2568 ms	24838	162	1/85.2/178
Sim-3	100	<1 ms	99	1	2/8.0/17
1A2J	1454	12 ms	1454	0	2/6.9/13

Parallel Raycasting of Metaballs



- **Second Approach:** find per fragment depth where density becomes 1.
- Initialize depth buffer $\lambda(\underline{\tau})$ by raycasting influence spheres as lower bound
- Use stencil buffer to mask out all fragments not hit in depth initialization
- Define upper bound depth texture from dataset bounding box
- Iterate till all fragments converged
  - Construct density texture $\tilde{D}(\underline{\tau})$
    - Define density texture fbo
    - Turn on stencil test+additive blending
    - Render all influence spheres
    - Per fragment evaluate metaballs at depth in $\lambda(\underline{\tau})$
  - Use depth ping pong buffer to check per fragment convergence and advance depth by Δ if fragment not converged



$$\frac{\Delta}{\Delta_{\max}} = \begin{cases} 1 - \tilde{D}(\underline{\tau})^2 & \tilde{D}(\underline{\tau}) < 1 - \epsilon \\ \frac{1}{2}\left(\tilde{D}(\underline{\tau}) - 1\right) & \tilde{D}(\underline{\tau}) > 1 + \epsilon \\ 0 & \text{otherwise} \end{cases}$$

Parallel Raycasting of Metaballs

Figure 46: Metaball isosurface approximation in *Sim-3* data set after different number of iterations. Lower right image shows final output image of the *Walking Depth Plane* approach. The other images show the content of the λ-buffer after different numbers of iterations (from upper left to lower right): directly after initialisation, after 5, 10, 15, and 31 (final result) iterations.

authors call the intermediate depth buffer λ-buffer

Metaballs Rendering – Performance

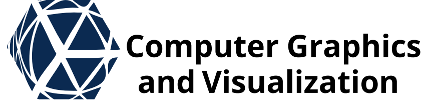


Table 13: Performance values of the *Vicinity-Texture* approach: The processing time specifies the time needed to build up the required data structure texture (view-independent) using a plane-sweep algorithm; Number of potential metaball members in the different data sets: Non-empty groups are the spheres which have at least one neighbour which is close enough to form a metaball with. Spheres which can never form a metaball are counted as empty groups. The size column shows the minimum, average and maximum vicinity group size in the data set.

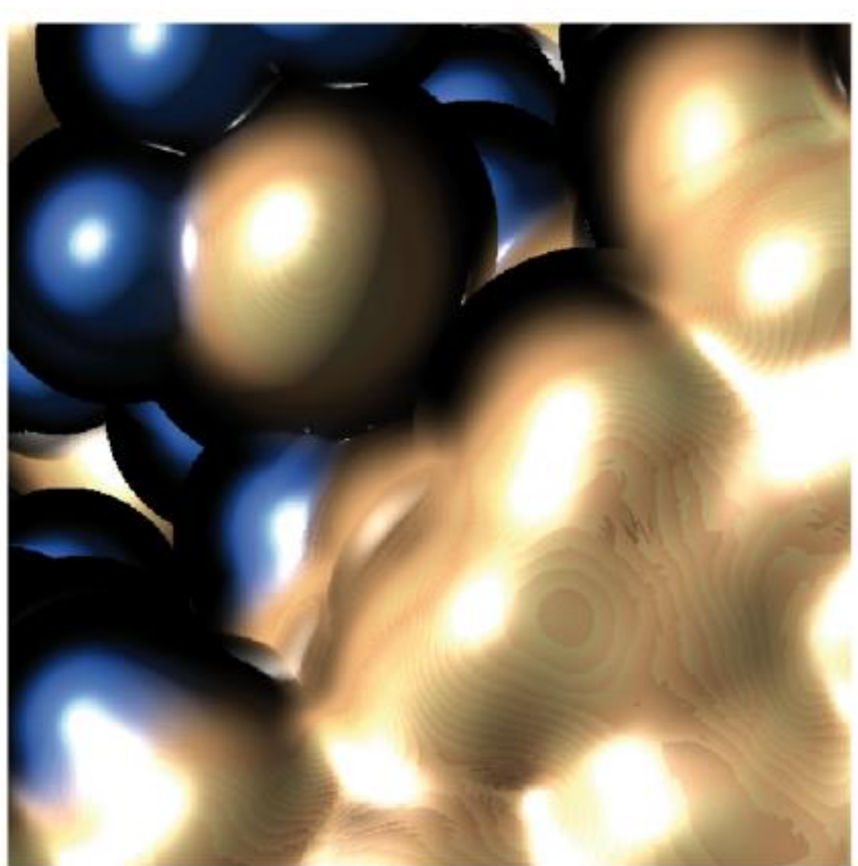
Data Set	# Spheres	Processing Time	Non-Empty Groups	Empty Groups	Size Min/ Avr/Max	fps
Argon	5000	113 ms	3684	1316	1/39.3/125	0.1
Ethane	25000	2568 ms	24838	162	1/85.2/178	0.2
Sim-3	100	<1 ms	99	1	2/8.0/17	4.0
1A2J	1454	12 ms	1454	0	2/6.9/13	0.14

Table 14: Performance values of the *Walking Depth Plane* method. The maximum number of iterations is changed for comparability or for artefact-free visualization.

Data Set	Max. # of Iterations	fps
Argon	100	13
Ethane	100	5
Sim-3	100	15
Sim-3	31	61
1A2J	100	21
1A2J	250	11

Metaballs Rendering – Performance

Computer Graphics and Visualization



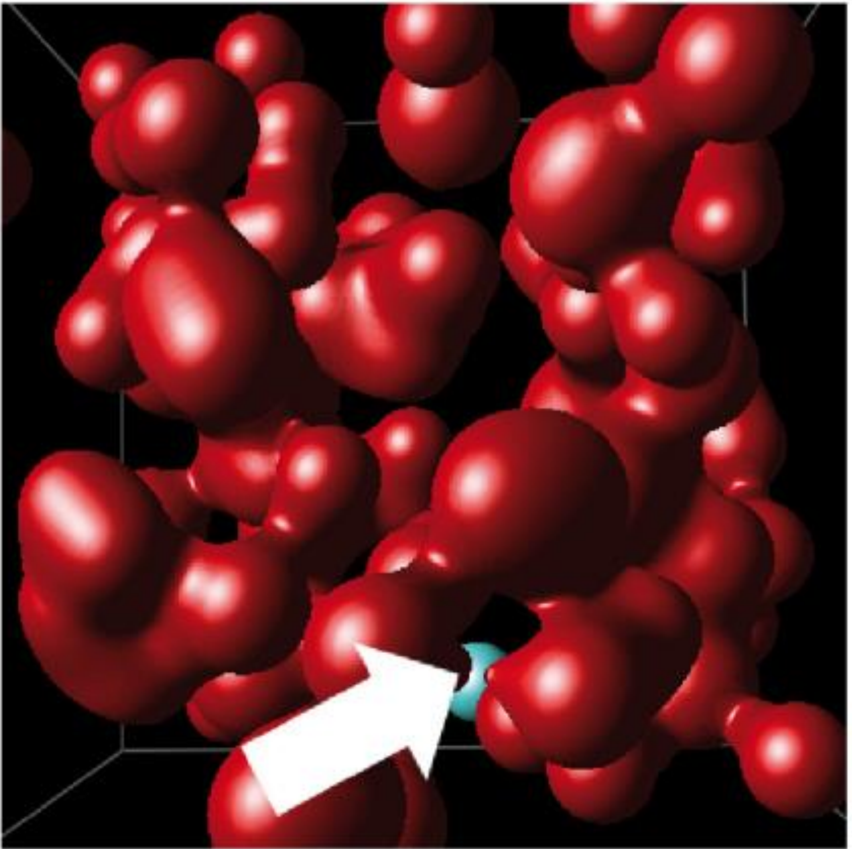


Figure 48: Visual artefacts: left: under-sampling artefacts occur when rendering the ethane data sets with the *Moving Depth Plane* and using too few steps; right: Missed surface parts (arrow) due to under-sampling by the *Vicinity Texture* approach.

High Performance Metaballs

- Third approach supports large datasets:
  M. Falk, S. Grottel, and T. Ertl, "Interactive Image-Space Volume Visualization for Dynamic Particle Simulations," SIGRAD Conference, 2010, pp. 35–43.
- Idea: slice volume around dataset bounding box front to back, reconstruct density function and volume raycasting of iso-surface intersection

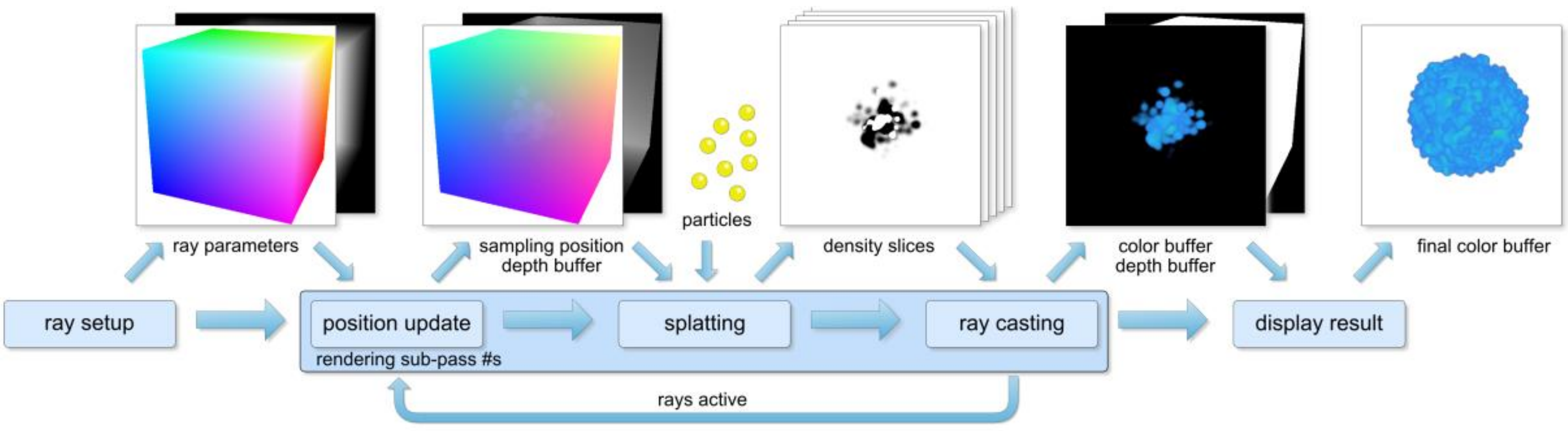
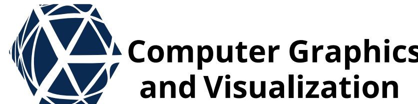


Figure 2: *Flow chart of our sliced ray casting method. In- and output buffers of the various stages are depicted above. After initial ray setup, multiple rendering sub-passes of sampling position update, density reconstruction through splatting and volume ray-casting are performed to generate the final image.*

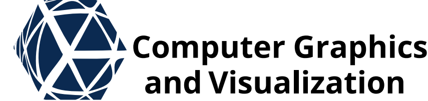
High Performance Metaballs



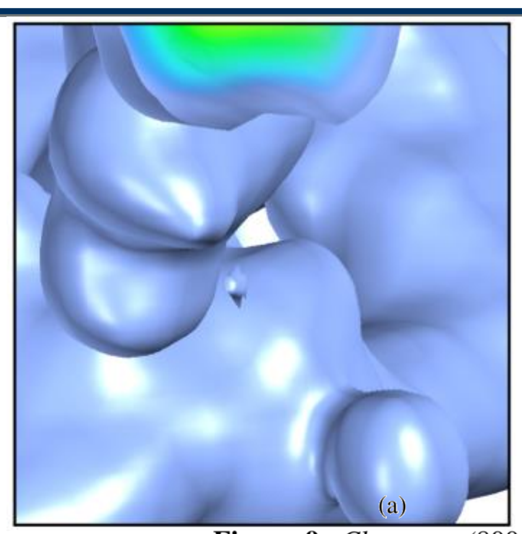
Details

- radix sort particles by their depth
- render dataset bbox to start rays
- compute 4 density slices at a time into the RGBA channels of a density fbo; for this render screen covering quad and use depth test LARGER
- exploit particle sorting to draw only active particles where influence sphere intersects one of the 4 current slices
- raycast density slices and in case of intersection, computer illumination and output color and depth (to terminate ray) to main fbo
- for each iteration count number of remaining rays with HWOQ but use this only to predict iteration count for next frame

High Performance Metaballs



- Through image space evaluation isosurface is reconstructed in pixel resolution (zoom provides arbitrarily high detail)
- Evaluation with NVIDIA GeForce 280 GTX



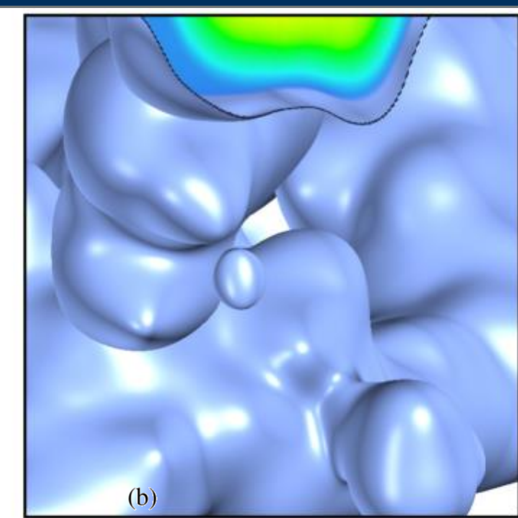
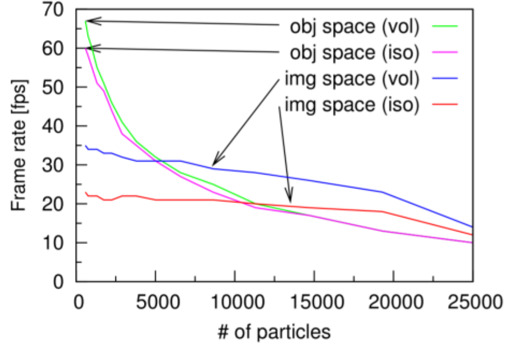


Figure 9: *Close-ups (800 × 800 pixels) of a small region of Fig. 7 are depicted: (a) object space, (b) image space. Ridges, grooves, highlights, and small features are more distinct in image space.*

Data set	Particle number	Object space vol	Object space iso	Image space vol	Image space iso
Synthetic	12,276	26.51	34.00	12.58	8.57
(sorted)		23.40	–	12.74	–
Synthetic	101,664	7.28	7.71	3.23	2.62
(sorted)		9.57	–	3.98	–
Cell	25,000	10.90	10.75	14.62	12.28
	584	67.36	60.08	35.13	23.05
Bulge	509,423	–	1.16	–	0.83
(cutaway)	∼250,000	–	1.59	–	0.30
Splat	742,141	–	0.97	–	0.7

Table 1: *Rendering performance. Timings for direct volume rendering (vol) and isosurface rendering (iso) are measured in frames per second.*



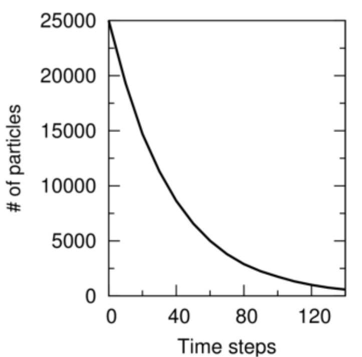


Figure 11: *Rendering performance of object-space and image-space methods for the signal transduction data set (left). Timings for direct volume rendering (vol) and isosurface rendering (iso) were measured. The number of particles is decreasing during the simulation (right).*

Particles

DERIVED SURFACES

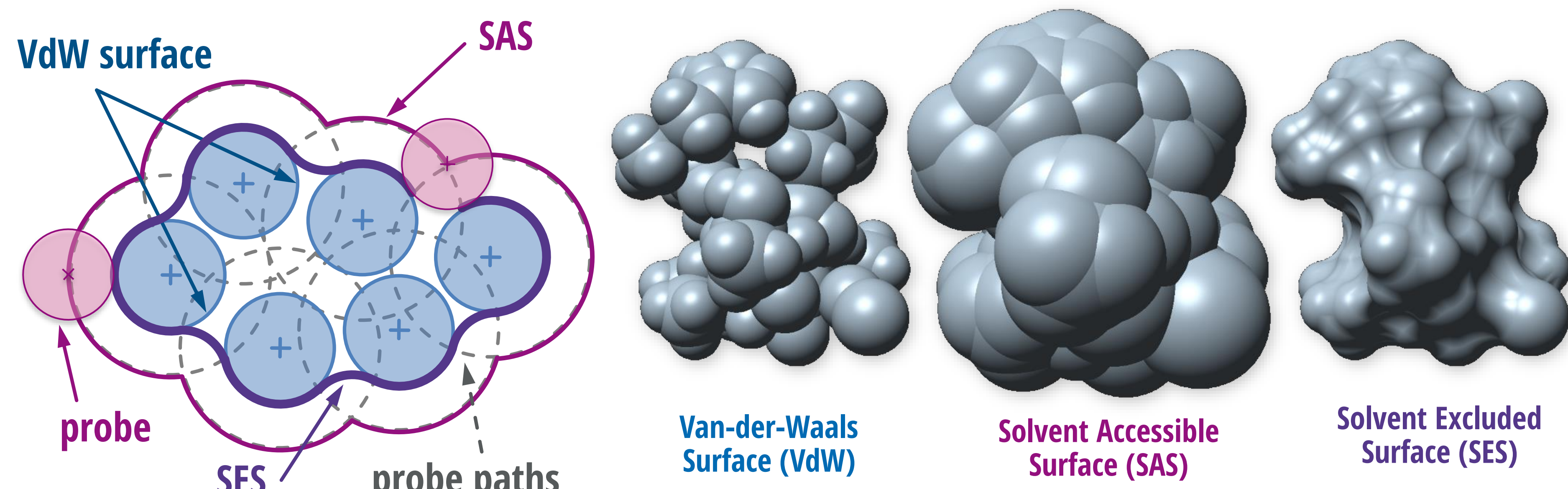
MOLECULAR SURFACES

Molecular Surfaces – Overview

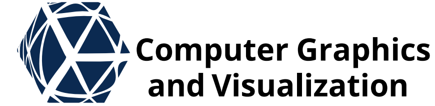
- The Van-der-Waals radius of an atom yields a solid sphere approximation of an atom
- The Van-der-Waals surface / envelop is union of Van-der-Waals spheres
- given probe sphere radius of solvent molecules, role probe over given molecule to define
  - all solvent probe center locations form Solvent Accessible Surface (SAS)
  - inner envelop of probe sphere forms Solvent Excluded Surface (SES)

Van-der-Waals radius in picometer

Period (row)																		
1	H 110[4] or 120																	He 140
2	Li 182	Be 153[5]											B 192[5]	C 170	N 155	O 152	F 147	Ne 154
3	Na 227	Mg 173											Al 184[5]	Si 210	P 180	S 180	Cl 175	Ar 188
4	K 275	Ca 231[5]	Sc	Ti	V	Cr	Mn	Fe	Co	Ni 163	Cu 140	Zn 139	Ga 187	Ge 211[5]	As 185	Se 190	Br 185	Kr 202
5	Rb 303[5]	Sr 249[5]	Y	Zr	Nb	Mo	Tc	Ru	Rh	Pd 163	Ag 172	Cd 158	In 193	Sn 217	Sb 206[5]	Te 206	I 198	Xe 216
6	Cs 343[5]	Ba 268[5]	*	Hf	Ta	W	Re	Os	Ir	Pt 175	Au 166	Hg 155	Tl 196	Pb 202	Bi 207[5]	Po 197[5]	At 202[5]	Rn 220[5]
7	Fr 348[5]	Ra 283[5]	**	Rf	Db	Sg	Bh	Hs	Mt	Ds	Rg	Cn	Nh	Fl	Mc	Lv	Ts	Og
Lanthanides	*	La	Ce	Pr	Nd	Pm	Sm	Eu	Gd	Tb	Dy	Ho	Er	Tm	Yb	Lu		
Actinides	**	Ac	Th	Pa	U 186	Np	Pu	Am	Cm	Bk	Cf	Es	Fm	Md	No	Lr		



Molecular Surfaces – SES



- VdW and SAS surfaces can be rendered with sphere glyphs directly
- The SES surface is composed of three types of surfaces patches: spherical patch, spherical triangle and toroidal patch

Spherical patches

Spherical triangles

Toroidal patches

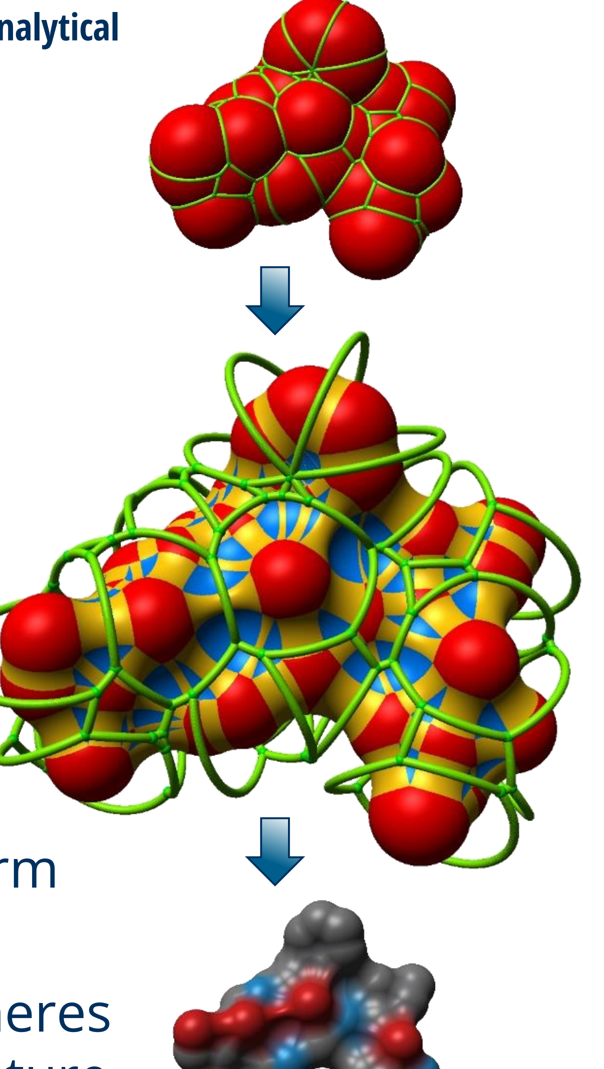
sphere triangle

torus-Patch

Molecular Surfaces – SES

Totrov, M., & Abagyan, R. (1996). The contour-buildup algorithm to calculate the analytical molecular surface. Journal of structural biology, 116(1), 138-143.

- Contour-Buildup Algorithm
  - setup SAS spheres
  - compute sphere-sphere intersection circles
  - compute graph of visible circle segments
- SES surface element extraction based on circle segment graph
  - nodes correspond to circle segment intersections and form sphere patches with negative curvature (mostly triangular)
  - edges correspond to circle segments and form toroidal patches
  - faces correspond to visible parts of VdW spheres and form sphere patches with positive curvature

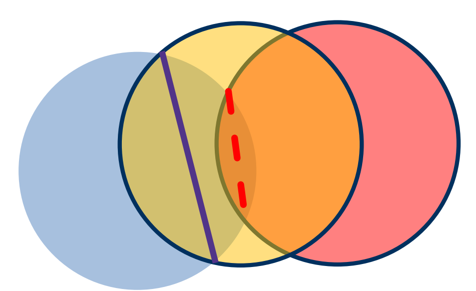


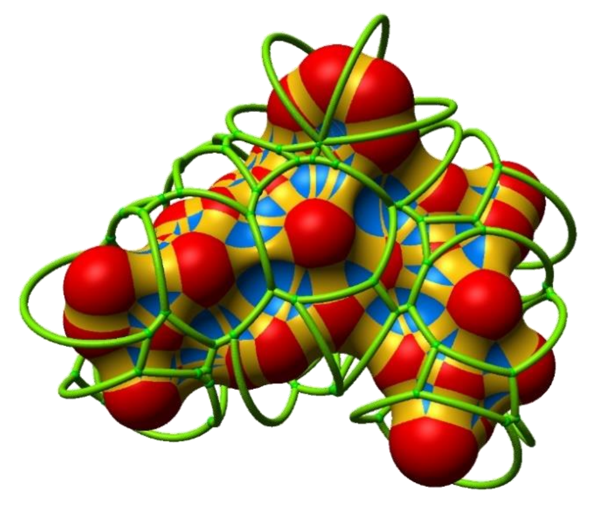
Molecular Surfaces – SES

Krone, M., Dachsbacher, C., & Ertl, T. (2010, August). Parallel computation and interactive visualization of time-varying solvent excluded surfaces. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology* (pp. 402-405). ACM.

- pure GPU implementation in CUDA

1. find neighbor atoms
   - bind atom location vbo with CUDA
   - build grid with radix sort and query neighbors
2. compute and store circles
   - from sphere-sphere intersections
3. filter circles that are too small
4. intersect circles of neighboring atoms
   - store remaining circle segments
   - *Easy to parallelize*
5. per surface type construct one vbo
6. OpenGL based primitive raycasting





Molecular Surfaces – SES

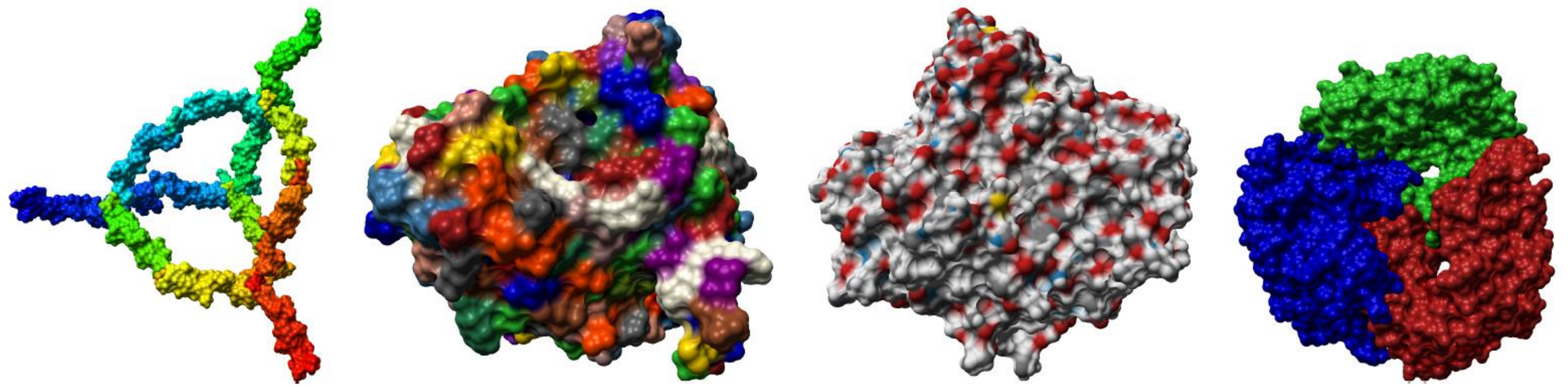


Figure 6: A selection of the data sets used in our performance measurements rendered with our methods using different coloring modes. From left to right: MDSim 2 (rainbow coloring), MDSim 3 (color by amino acid), MDSim 4 (color by element), 1AF6 (color by amino acid chain). See Table 1 for further details.

Table 1: Performance measurements for our CUDA implementation of the Contour-Buildup algorithm. CB denotes the execution time of the Contour-Buildup algorithm (including data transfer), while VBO+SH denotes the time for writing VBOs for rendering and the handling of singularities (in milliseconds). The column labeled FPS states the overall frame rates in frames per second (i. e. data transfer, Contour-Buildup computation and rendering of the SES). (*) marks data sets obtained from the PDB.

Data set	#Atoms	CB	VBO+SH	FPS
1OGZ*	~1,000	9.5 ms	2.1 ms	71 fps
1VIS*	~2,500	14.1 ms	2.6 ms	50 fps
MDSim 1	~4,000	22.7 ms	3.6 ms	33 fps
MDSim 2	~4,800	19.6 ms	5.4 ms	36 fps
MDSim 3	~6,600	31.2 ms	4.5 ms	24 fps
MDSim 4	~8,000	35.7 ms	6.0 ms	21 fps
1AF6*	~10,000	36.1 ms	7.3 ms	20 fps